CHAPTER 28

A Motor Control Project

Imagine combining the power and convenience of a personal computer with the peripherals of a microcontroller. Motors moving in response to keyboard commands. Plots showing the motor's positional history. Interactive controller tuning. By combining your knowledge of microcontrollers, motors, and controls you can accomplish these goals.

It starts with a menu. This menu will provide a simple user interface: a list of commands that you can choose by pressing a key. We will begin with an empty menu. By the end of this project, it will have nearly 20 options: everything from reading encoders to setting control gains to plotting trajectories. Much work lies ahead, but by breaking the project into subgoals and using discipline in your coding, you will complete it successfully.

28.1 Hardware

The major hardware components for the project are listed below. Data sheets for the chips can be found on the book's website.

- brushed DC motor with encoder, including a mounting bracket and a bar for the motor load (Figure 28.1)
- the NU32 microcontroller board
- a printed circuit board with an encoder counting chip counting in 4x mode (quadrature inputs from the encoder and an SPI interface to the NU32)
- a printed circuit board with the MAX9918 current-sense amplifier and a built-in 15 mΩ current-sense resistor, to provide a voltage proportional to the motor current (this voltage output is attached to an ADC input on the NU32)
- a printed circuit board breaking out the DRV8835 H-bridge (attached to a digital output and a PWM/output compare channel on the NU32)
- a battery pack to provide power to the H-bridge
- resistors and capacitors

We discuss the hardware in greater detail after an overview of the software you will develop.

Embedded Computing and Mechatronics with the PIC32 Microcontroller. http://dx.doi.org/10.1016/B978-0-12-420165-1.00028-7

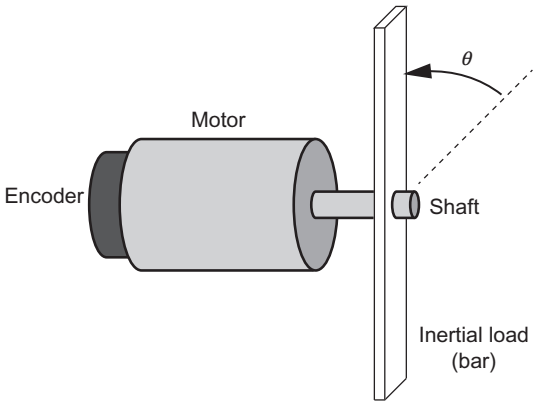

Figure 28.1
A bar attached as the motor's load. Positive rotation θ is counterclockwise when viewing along the axis of the motor shaft toward the motor.

28.2 Software Overview

The motivation for this project is to build an intelligent motor driver, with a subset of the features found in industrial products like the Copley Accelus drive described in Chapter 27. The drive incorporates the amplifier to drive the motor as well as feedback control to track a reference position, velocity, or torque. Many robots and computer-controlled machine tools have drives like this, one for each joint.

Your motor drive system should accept a desired motor trajectory, execute that trajectory, and send the results back to your computer for plotting (Figure 28.2). We break this project into several supporting features, accessible from a menu on your computer. This approach enables you to build and test incrementally.

This project requires you to develop two different pieces of software that communicate with each other: the PIC32 code for the motor driver and the *client* user interface that runs on the host computer. In this chapter we assume that the client is developed for MATLAB, taking advantage of its plotting capabilities. You could easily use another programming language (e.g., python with its plotting capability), provided you can establish communication with the NU32 via the UART (Chapter 11).

For convenience, we will refer to the code on the host computer as the "client" and the code on the PIC32 as the "PIC32."

The PIC32 will implement the control strategy of Chapter 27, specifically Figure 27.7, consisting of a low-frequency position control loop and a nested high-frequency current control loop. This is a common industrial control scheme. In this project, the outer position

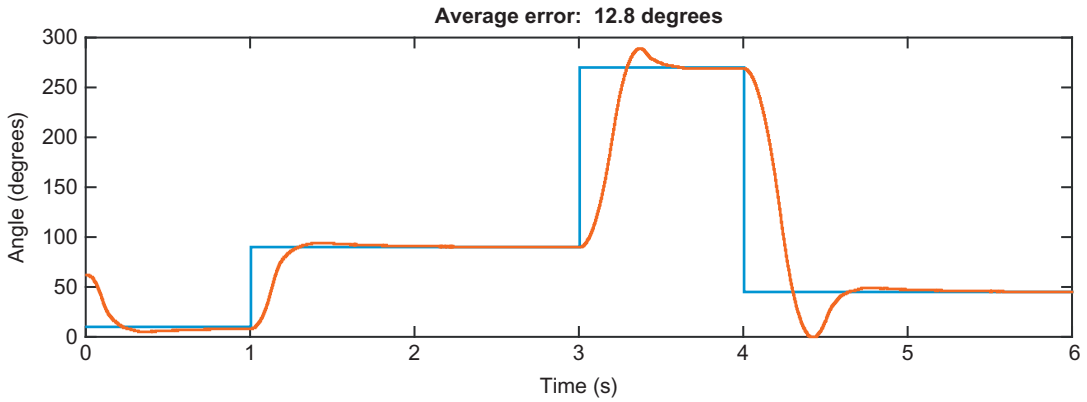

Figure 28.2
An example result of this project. The user specifies a position reference trajectory for the motor and the system attempts to follow it. An average error is calculated as an indication of how well the motor controller achieved its objective.

control loop runs at 200 Hz and the inner current control loop runs at 5 kHz. PWM is at 20 kHz.

Prior to implementing these control loops, we will implement some more basic features. The current control loop must output a PWM signal to the motor and read values from the current sensor, and the position control loop needs encoder feedback. Therefore we begin with menu options that allow the user to directly view sensor readings and specify the PWM duty cycle. Another option allows the user to tune the current control feedback loop independently of the position controller. The ability to interactively specify control gains and see the resulting control performance simplifies this process. All of the features will be accessible from the client menu that you create, depicted in Figure 28.3.

Below we describe the purpose of the individual commands, in order, but let us start with the last one, `r: Get mode`. The PIC32 can operate in one of five *modes*: IDLE, PWM, ITEST, HOLD, and TRACK. When it first powers on, the PIC32 should be in IDLE mode. In IDLE mode, the PIC32 puts the H-bridge in brake mode, with zero voltage across the motor. In PWM mode, the PIC32 implements a fixed PWM duty cycle, as requested by the user. In ITEST mode, the PIC32 tests the current control loop, without caring about the motion of the motor. In HOLD mode, the PIC32 attempts to hold the motor at the last position commanded by the user. In TRACK mode, the PIC32 attempts to track a reference motor trajectory specified by the user (Figure 28.2). Some of the menu commands cause the PIC32 to change mode, as noted below.

`a:` `Read current sensor (ADC counts)`. Print the motor current as measured using the current sensor, in ADC counts (0-1023). For example, the result could look like

```
PIC32 MOTOR DRIVER INTERFACE

     a: Read current sensor (ADC counts)      b: Read current sensor (mA)
     c: Read encoder (counts)                 d: Read encoder (deg)
     e: Reset encoder                         f: Set PWM (-100 to 100)
     g: Set current gains                     h: Get current gains
     i: Set position gains                    j: Get position gains
     k: Test current control                  l: Go to angle (deg)
     m: Load step trajectory                  n: Load cubic trajectory
     o: Execute trajectory                    p: Unpower the motor
     q: Quit client                           r: Get mode

ENTER COMMAND:
```

Figure 28.3
The final menu. The user enters a single character, which may result in the user being prompted for more information. Additional options are possible, but these are a minimum.

```
ENTER COMMAND: a
The motor current is 903 ADC counts.
```

After printing the current sensor reading, the client should display the full menu again, to help the user enter the next command. Alternatively, to save screen space, you could just keep a printout of the commands handy and not print the menu to the screen each time. Although the "a" command is somewhat redundant with the next command, which returns the current in mA, it is sometimes useful for debugging to see the raw ADC data, rather than the scaled version in more familiar units.

`b:` `Read current sensor (mA)`. Print the motor current as measured using the current sensor, in mA. The output could look like

```
The motor current is 763 mA.
```

By a convention we will adopt, the current is positive when trying to drive the motor counterclockwise, in the direction of increasing motor angle (Figure 28.1). If you are getting the opposite sign, you can swap the wires to the current sensor or handle it in software.

`c:` `Read encoder (counts)`. Print the encoder angle, in encoder counts. By convention, the encoder count increases as the motor rotates counterclockwise. An example output:

```
The motor angle is 314 counts.
```

`d:` `Read encoder (deg)`. Print the encoder angle, in degrees. By convention, the encoder angle increases as the motor rotates counterclockwise. If the encoder gives 1000 counts per revolution in 4x decoding mode, then 314 counts would give an output

```
The motor angle is 113.0 degrees.
```

since $360°(314/1000) \approx 113°$. In this project, 0 degrees, 360 degrees, 720 degrees, etc., are all treated differently, allowing us to control multiple turns of the motor.

`e:` `Reset encoder`. The present angle of the motor is defined as the zero position. No output is necessary. Optionally, command `d` could be called automatically after zeroing the encoder, to confirm the result to the user.

`f:` `Set PWM (-100 to 100)`. Prompt the user for a desired PWM duty cycle, specified in the range [−100%, 100%]. The PIC32 switches to PWM mode and implements the constant PWM until overridden. An input −100 means that the motor is full on in the clockwise direction, 100 means full on in the counterclockwise direction, and 0 means that the motor is unpowered. An example prompt, user reply, and result is

```
What PWM value would you like [-100 to 100]?  73
PWM has been set to 73% in the counterclockwise direction.
```

with the motor speeding up to its 73% steady-state speed. The PIC32 should saturate values outside the range [−100, 100] and convert values in the range [−100, 100] to an appropriate PWM duty cycle and direction bit to the DRV8835.

`g:` `Set current gains`. Prompt for the current loop control gains and send them to the PIC32, to be implemented immediately. (This command does not change the mode of the PIC32, however; the gains only affect the motor's behavior when the PIC32 is in the ITEST, HOLD, or TRACK mode.) We suggest two gains, for a PI controller. An example interface:

```
Enter your desired Kp current gain [recommended: 4.76]:  3.02
Enter your desired Ki current gain [recommended: 0.32]:  0
Sending Kp = 3.02 and Ki = 0 to the current controller.
```

It is not necessary to recommend values in the prompting text, but if you find good values, you may wish to put them into your client, so you remember for the future. (The values given here are just for demonstration, not actual recommendations!) You might also wish to use only integers for your gains, depending on the implicit units, to allow for more time-efficient math.

`h:` `Get current gains`. Print the current controller gains. Example output:

```
The current controller is using Kp = 3.02 and Ki = 0.
```

`i:` `Set position gains`. Prompt for the position loop control gains and send them to the PIC32, to be implemented immediately. (This command does not change the mode of the PIC32, however; the gains only affect the motor's behavior if the PIC32 is in the HOLD or TRACK mode.) If you use PID control, the interface could be:

```
Enter your desired Kp position gain [recommended: 4.76] :  7.34
Enter your desired Ki position gain [recommended: 0.32] :  0
Enter your desired Kd position gain [recommended: 10.63]:  5.5
Sending Kp = 3.02, Ki = 0, and Kd = 5.5 to the position controller.
```

Other motion controllers (Chapter 27) require specification of other gains and parameters.

`j:` `Get position gains`. Print the position controller gains. Example output:

```
The position controller is using Kp = 7.34, Ki = 0, and Kd = 5.5.
```

`k:` `Test current control`. Test the current controller, which uses the gains set using the "g" command. This command puts the PIC32 in ITEST mode. In this mode, the 5 kHz current control loop uses a 100 Hz, 50% duty cycle, ±200 mA square wave reference current. Each time through the loop, the reference and actual current are stored in data arrays for later plotting. After 2-4 cycles of the 100 Hz current reference, the PIC32 switches to IDLE mode, and the data collected during the ITEST mode is sent back to the client, where it is plotted. An example of reasonably good tracking is shown in Figure 28.4. The plots help the user to choose good current loop gains.

During the current control test, the motor may move a little bit, but it is important to remember that we are not interested in the motor's motion, only the performance of the current controller. The motor should not move far, since the actual current should be changing quickly with zero average.

The client should calculate an average of the absolute value of the current error over the samples. This score allows you to compare the performance of different current controllers, in addition to the eyeball test based on the plots.

`l:` `Go to angle (deg)`. Prompt for an angle, in degrees, that the motor should move to. The PIC32 switches to HOLD mode, and the motor should move to the specified position immediately and hold the position. Example interface:

```
Enter the desired motor angle in degrees:  90
Motor moving to 90 degrees.
```

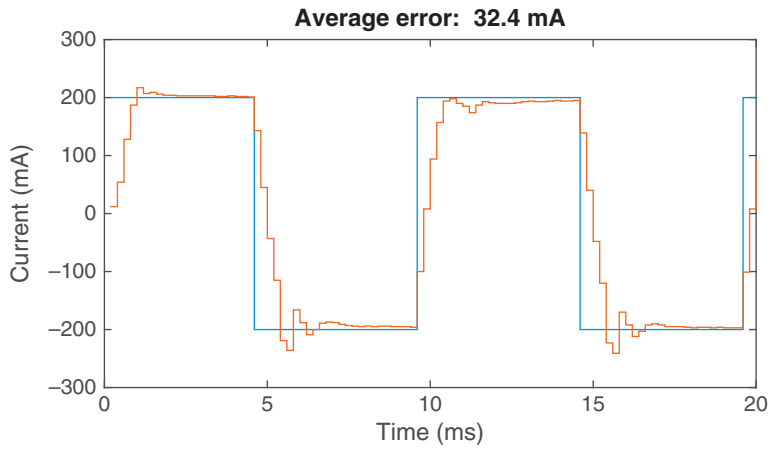

Figure 28.4
A result of a current controller test. The square wave is the reference current and the other plot is the measured current.

`m:` `Load step trajectory`. Prompt for the time and angle parameters describing a series of one or more steps in the motor position. Plot the reference trajectory on the client so the user can see if the trajectory was entered correctly. Convert the reference trajectory to a series of setpoint positions at 200 Hz (each point is separated by 5 ms), store it in an array, and send it to the PIC32 for future execution. In addition, set the number of samples the PIC32 should record in the position control loop according to the duration of the trajectory.

The PIC32 should discard any trajectory data that exceeds the maximum trajectory length. This command does not change the mode of the PIC32.

An example interface is shown below. First the user requests a trajectory that starts at angle 0 degrees at time 0 s; then steps to 90 degrees at 1 s; and holds at 90 degrees until 500 s have passed. This duration is too long: the PIC32 RAM cannot store 500 s × 200 samples/s = 100,000 samples of the reference trajectory. In the second try, the user requests the step trajectory shown in Figure 28.5. Note that the client only plots the reference trajectory that is sent to the PIC32; no data is received from the PIC32, because the trajectory has not been executed yet.

```
Enter step trajectory, in sec and degrees [time1, ang1; time2, ang2; ...]:
  [0, 0; 1, 90; 500, 90]
Error:  Maximum trajectory time is 10 seconds.

Enter step trajectory, in sec and degrees [time1, ang1; time2, ang2; ...]:
  [0, 0; 1, 180; 3, -90; 4, 0; 5, 0]
```

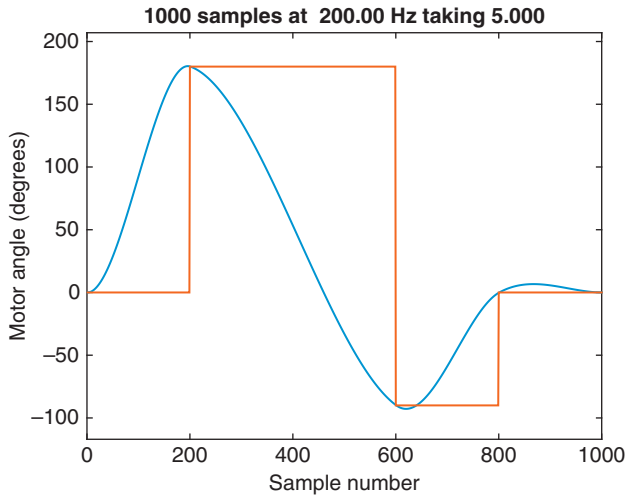

Figure 28.5
Sample reference trajectories for the position controller. The step trajectory is generated in MATLAB using the command `ref = genRef([0,0; 1,180; 3,-90; 4,0; 5,0], 'step')` and the smooth cubic curve is generated by replacing `'step'` with `'cubic'`.

```
Plotting the desired trajectory and sending to the PIC32 ... completed.
```

Now the PIC32 has a reference trajectory waiting to be executed. A MATLAB function `genRef.m` is provided with this chapter to generate the sample points of step trajectories, and is invoked using

```
ref = genRef([0,0; 1,180; 3,-90; 4,0; 5,0], 'step');
```

The resulting trajectory is shown in Figure 28.5, as a plot of the motor angle in degrees vs. the sample number.

`n:` `Load cubic trajectory`. Prompt for the time and angle parameters describing a set of via points for a cubic interpolation trajectory for the motor. This command is similar to the step trajectory command, except a smooth trajectory is generated through the specified points, with zero velocity at the beginning and end. The same MATLAB function `genRef` calculates the cubic trajectory, using the option `'cubic'` instead of `'step'` (Figure 28.5).

```
Enter cubic trajectory, in sec and degrees [time1, ang1; time2, ang2; ...]:
  [0, 0; 1, 180; 3, -90; 4, 0; 5, 0]
Plotting the desired trajectory and sending to the PIC32 ... completed.
```

`o:` `Execute trajectory`. Execute the trajectory stored on the PIC32. This command changes the PIC32 to TRACK mode, and the PIC32 attempts to track the reference trajectory previously stored on the PIC32. After the trajectory has finished, the PIC32 switches to HOLD mode, holding the last position of the trajectory, then sends the reference and actual position data back to the client for plotting, as shown in Figure 28.2 for a step trajectory. The client should calculate an average position error, similar to the current control test. Because step trajectories request unrealistic step changes of the motor angle, the average error for a step trajectory could be large, while the average error for a smooth cubic trajectory can be quite small.

`p:` `Unpower the motor`. The PIC32 switches to IDLE mode.

`q:` `Quit client`. The client should release the communication port so other applications can communicate with the NU32. The PIC32 should be set to IDLE mode.

`r:` `Get mode`. Described above. Example output:

```
The PIC32 controller mode is currently HOLD.
```

Once the circuits have been built, the functions above have been fully implemented and tested (both the client and the PIC32), and control gains have been found that yield good performance of the motor, the project is complete.

To help you finish this project, Section 28.3 gives some recommendations on how to develop maintainable, modular PIC32 code. Section 28.4 breaks the project down into a series of steps that you should complete, in order. Finally, Section 28.5 describes a number of extensions to go further with the project.

28.3 Software Development Tips

Since this is a big project, it is a good idea to be disciplined in the development of your PIC32 code. This discipline will make the code easier to maintain and build upon. Here are some tips. We recommend you read this section to help you better understand the project, but if you are confident in your software skills and your understanding of the project, you need not adhere to the advice.

Debugging

Run-time errors in your PIC32 code are inevitable. The ability to locate bugs quickly is crucial to keeping development time reasonable. Typically debugging consists of having your program provide feedback at breakpoints, to pinpoint where the unexpected behavior occurs.[1] This type of debugging can be challenging for embedded code, since there is no `printf` to a monitor. You should be comfortable using the following tools, however:

- **Terminal emulator.** Using a terminal emulator to connect to the PIC32, instead of your client, allows you to send simple commands and see exactly what information is being sent back by the PIC32, without the complication of potential errors in the client code. Just be aware that there is only one communication port with the NU32, so either the client or the terminal emulator can be used, not both simultaneously.
- **LCD screen.** You can use the LCD screen to display information about the state of the PIC32 without using the UART communication with the host.
- **LEDs and digital outputs.** You can use LEDs or digital outputs connected to an oscilloscope as other ways of getting feedback from the PIC32. To verify that your current control and position control ISRs are operating at the correct frequencies, for example, you could toggle a digital output in each ISR and look at the resulting square wave "heartbeats" on an oscilloscope.

Modularity

Instead of writing one large `.c` file to do everything, consider breaking the project into multiple *modules*. A module consists of one `.c` and one `.h` file. The `.h` file contains the module's *interface* and the `.c` file contains the module's *implementation*. The `.c` file should always `#include` the corresponding `.h` file.

A module's interface (the `.h` file) consists of function prototypes and data types that other modules can use. You can use functions from module `A` in module `B` by having `B.c` include `A.h`.

[1] The PICkit 3 and other Microchip programming tools can be used to set breakpoints and step through code as it executes. These features can be accessed through MPLAB X but cannot be used in conjunction with the NU32's bootloader.

A module's implementation (the `.c` file) contains the code that makes the interface functions work. It also contains any functions and variables that the module uses but wants to hide from other modules. These hidden functions and variables should be declared as `static`, limiting their scope to the module's `.c` file. By hiding implementations in the `.c` file and only exposing certain functions in the `.h` file, you decrease the dependencies between modules, making maintenance easier and helping prevent bugs. For those readers familiar with an object-oriented programming language such as C++ or Java, these concepts roughly mirror the ideas of public and private class members.

In an embedded system, peripherals can be accessed by code in any module; therefore, dividing the code into modules is only the first step in a good design. To maintain proper module separation, you should document which modules *own* which peripherals. If a module owns a peripheral, only it should access that peripheral directly. If a module needs to access a peripheral it does not own, it must call a function in the owning module. These rules, enforced by your vigilance rather than the compiler, will help you more easily isolate bugs and fix them as they occur.

All `.c` and `.h` files should be in the same directory, allowing the project to be built by compiling and linking all `.c` files in the directory.

One module your project will certainly use is `NU32`, for communication with the client. Other modules we suggest are the following. Some example interface functions are suggested for each module, but feel free to use your own functions.

- **main.** This module is the only one with no `.h` file, since no other module needs to call functions in the `main` module. The `main.c` file is the only one with a `main` function. `main.c` calls appropriate initialization functions for the other modules and then enters an infinite loop. The infinite loop waits for commands from the client, then interprets the user's input and responds appropriately.
- **encoder.** This module owns an SPI peripheral, to communicate with the encoder counting chip. The interface `encoder.h` should provide functions to (1) do a one-time setup or initialization of the encoder module, (2) read the encoder in encoder counts, (3) read the encoder in degrees, and (4) reset the encoder position so that the present angle of the encoder is treated as the zero angle.
- **isense.** This module owns the ADC peripheral, used to sense the motor current. The interface `isense.h` should provide functions for a one-time setup of the ADC, to provide the ADC value in ADC counts (0-1023), and to provide the ADC value in terms of milliamps of current through the motor.
- **currentcontrol.** This module implements the 5 kHz current control loop. It owns a timer to implement the fixed-frequency control loop, an output compare and another timer to implement a PWM signal to the H-bridge, and one digital output controlling the motor's

direction (see the description of the DRV8835 in Chapter 27). Depending on the PIC32 operating mode, the current controller either brakes the motor (IDLE mode), implements a constant PWM (PWM mode), uses the current control gains to try to provide a current specified by the position controller (HOLD or TRACK mode), or uses the current control gains to try to track a 100 Hz ±200 mA square wave reference (ITEST mode). The interface `currentcontrol.h` should provide functions to initialize the module, receive a fixed PWM command in the range [−100, 100], receive a desired current (from the `positioncontrol` module), receive current control gains, and provide the current control gains.

- **positioncontrol.** This module owns a timer to implement the 200 Hz position control loop. The interface `positioncontrol.h` should provide functions to initialize the module, load a trajectory from the client, load position control gains from the client, and send position control gains back to the client.
- **utilities.** This module is used for various bookkeeping tasks. It maintains a variable holding the operating mode (IDLE, PWM, ITEST, HOLD, or TRACK) and arrays (buffers) to hold data collected during trajectory tracking (TRACK) or current tracking (ITEST). The interface `utilities.h` provides functions to set the operating mode, return the current operating mode, receive the number *N* of samples to save into the data buffers during the next TRACK or ITEST, write data to the buffers if *N* is not yet reached, and to send the buffer data to the client when *N* samples have been collected (TRACK or ITEST has completed).

Variables

Variables that are shared by ISRs and mainline code should be declared `volatile`. You may consider disabling interrupts before using a shared variable in the mainline code, then re-enabling interrupts afterward, to make sure a read or write of the variable in the mainline code is not interrupted. If you do disable interrupts, make sure not to leave interrupts disabled for more than a few simple lines of code; otherwise you are defeating the purpose of having interrupts in the first place.

Function local variables that need to keep their value from one invocation of the function to the next should be declared `static`. A variable `foo` that is only meant to be used in one `.c` file, but should be available to several functions in that file (i.e., "global" within the file), can be defined at the beginning of the file but outside any function. To prevent this `foo` from colliding with a variable of the same name defined in another `.c` file, `foo` should be declared `static`, limiting its scope to the file it is defined in.

Consider writing your code so that no variable is used outside the `.c` file it is defined in. To share the value of a variable in module `A` with other modules, provide the prototype of a public accessor function in `A.h`, like `int get_value_from_A()`. Do not define variables in header files.

New data types

The five operating modes (IDLE, etc.) can be represented by a variable of type `int` (or `short`), taking values 0 to 4. Instead you could consider using `enum`, allowing you to create a data type `mode_t` with only five possible values, named IDLE, PWM, ITEST, HOLD, and TRACK.

For your data buffers, consider creating a new data type `control_data_t` using a `struct`, where the `struct` has several members (e.g., the current reference, the actual current, the position reference, and the actual position). This way you can make a single data array of type `control_data_t` instead of having several arrays.

Integer math

The PIC32 control laws begin by sensing an integral number of encoder counts and ADC counts, and end by storing an integer to the period register of an output compare module (PWM). Thus it is possible, though not necessary, to calculate all control laws using integer math with integer-valued gains. Using integer math ensures that the controllers run as quickly as possible. If you do use only integers, however, you are responsible for making sure that there are no unacceptable roundoff or overflow errors.

Degrees and milliamps are the most natural units to display to the user on the client interface, however.

28.4 Step by Step

This section provides a step-by-step guide to building the project, testing and debugging as you go. Make sure that each step works properly before moving on to the next step, and be sure to perform all the numbered action items associated with each subsection. Turn in your answers for the items in **bold**.

28.4.1 Decisions, Decisions

Before writing any code, you must decide which peripherals you will use in the context of the controller block diagram in Figure 27.7. The PIC32 will connect to three external circuit boards: the H-bridge, the motor encoder counter, and the motor current sensor. Record your answers to the following questions:

1. The NU32 communicates with the encoder counter by an SPI channel. Which SPI channel will you use? Which NU32 pins does it use?
2. The NU32 reads the MAX9918 current sensor using an ADC input. Which ADC input will you use? Which NU32 pin is it?

3. The NU32 controls the DRV8835 H-bridge using a direction bit (a digital output) and PWM (an output compare and a timer). Which peripherals will you use, and which NU32 pins?
4. Which timers will you use to implement the 200 Hz position control ISR and the 5 kHz current control ISR? What priorities will you use?
5. Based on your answers to these questions, and your understanding of the project, annotate the block diagram of Figure 27.7. Each block should clearly indicate which devices or peripherals perform the operation in the block, and each signal line should clearly indicate how the signal is carried from one block to the other. (After this step, there should be no question about the hardware involved in the project. The details of wiring the H-bridge, current sensor, and encoder are left to later.)
6. Based on which circuit boards need to be connected to which pins of the NU32, and the connections of the circuit boards to the motor and encoder, sketch a proposed layout of the circuit boards relative to the NU32 so that wire crossing is approximately minimized. (Do not make a full circuit diagram at this time.)
7. **Turn in your answers for items 1-6.**

28.4.2 Establishing Communication with a Terminal Emulator

The role of the `main` function is to call any functions initializing peripherals or modules and to enter an infinite loop, dispatching commands that the PIC32 receives from the client.

Code Sample 28.1 `main.c`**. Basic Code for Setup and Communication.**

```
#include "NU32.h"          // config bits, constants, funcs for startup and UART
// include other header files here

#define BUF_SIZE 200

int main()
{
  char buffer[BUF_SIZE];
  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init
  NU32_LED1 = 1;  // turn off the LEDs
  NU32_LED2 = 1;
  __builtin_disable_interrupts();
  // in future, initialize modules or peripherals here
  __builtin_enable_interrupts();

  while(1)
  {
    NU32_ReadUART3(buffer,BUF_SIZE); // we expect the next character to be a menu command
    NU32_LED2 = 1;                   // clear the error LED
    switch (buffer[0]) {
      case 'd':                      // dummy command for demonstration purposes
```

```
        {
          int n = 0;
          NU32_ReadUART3(buffer,BUF_SIZE);
          sscanf(buffer, "%d", &n);
          sprintf(buffer,"%d\r\n", n + 1); // return the number + 1
          NU32_WriteUART3(buffer);
          break;
        }
        case 'q':
        {
          // handle q for quit. Later you may want to return to IDLE mode here.
          break;
        }
        default:
        {
          NU32_LED2 = 0;  // turn on LED2 to indicate an error
          break;
        }
      }
    }
    return 0;
  }
```

The infinite `while` loop reads from the UART, expecting a single character. That character is processed by a `switch` statement to determine what action to perform. If the character matches a known menu entry, we may have to retrieve additional parameters from the client. The specific format for these parameters depends on the particular command, but in this case, after receiving a "d," we expect an integer and store it in `n`. The command then increments the integer and sends the result to the client.

Note that each `case` statement ends with a `break`. The `break` prevents the code from falling through to the next case. Make sure that every `case` has a corresponding `break`.

Unrecognized commands trigger the default case. The default case illuminates LED2 to indicate an error. We could have designed the communication protocol to allow the PIC32 to return error conditions to the client. Although crucial in the real world, proper error handling complicates the communication protocol and obscures the goals of this project. Therefore we omit it.

Before proceeding further, test the PIC32 menu code:

1. Compile and load `main.c`, then connect to the PIC32 with a terminal emulator.
2. Enter the "d" command, then enter a number. Ensure that the command works as expected.
3. Issue an unknown command and verify that LED2 illuminates.

28.4.3 Establishing Communication with the Client

We have also provided you with some basic MATLAB client code. This code expects a single character of keyboard input and sends it to the PIC32. Then, depending on the command sent, the user can be prompted to enter more information and that information can be sent to the PIC32.

Code Sample 28.2 client.m**. MATLAB Client Code.**

```
function client(port)
%   provides a menu for accessing PIC32 motor control functions
%
%   client(port)
%
%   Input Arguments:
%       port - the name of the com port.  This should be the same as what
%               you use in screen or putty in quotes ' '
%
%   Example:
%       client('/dev/ttyUSB0') (Linux/Mac)
%       client('COM3') (PC)
%
%   For convenience, you may want to change this so that the port is hardcoded.

% Opening COM connection
if ~isempty(instrfind)
    fclose(instrfind);
    delete(instrfind);
end

fprintf('Opening port %s....\n',port);

% settings for opening the serial port. baud rate 230400, hardware flow control
% wait up to 120 seconds for data before timing out
mySerial = serial(port, 'BaudRate', 230400, 'FlowControl', 'hardware','Timeout',120);
% opens serial connection
fopen(mySerial);
% closes serial port when function exits
clean = onCleanup(@()fclose(mySerial));

has_quit = false;
% menu loop
while ~has_quit
    fprintf('PIC32 MOTOR DRIVER INTERFACE\n\n');
    % display the menu options; this list will grow
    fprintf('     d: Dummy Command    q: Quit\n');
    % read the user's choice
    selection = input('\nENTER COMMAND: ', 's');

    % send the command to the PIC32
    fprintf(mySerial,'%c\n',selection);

    % take the appropriate action
    switch selection
        case 'd'                        % example operation
            n = input('Enter number: '); % get the number to send
            fprintf(mySerial, '%d\n',n); % send the number
```

```
            n = fscanf(mySerial,'%d');   % get the incremented number back
            fprintf('Read: %d\n',n);     % print it to the screen
        case 'q'
            has_quit = true;             % exit client
        otherwise
            fprintf('Invalid Selection %c\n', selection);
    end
end

end
```

Test the client as follows. Do not move on until you have completed each step.

1. Exit the terminal emulator (if it is still open).
2. Run `client.m` in MATLAB and confirm that you are connected to the PIC32. Issue the "d" command, and verify that it works as you expect. (The client prompts you for a number and sends it to the PIC32. The PIC32 increments it and sends the value back. The client prints out the return value.)
3. Quit the client by using "q."
4. Implement a command "x" on the PIC32 that accepts two integers, adds them, and returns the sum to the client.
5. Test the new command using the terminal emulator. Once you have verified that the PIC32 code works, quit the emulator.
6. Add an entry "x" to the client menu and verify that the client's new menu command works as expected.

After completing the tasks above, you should be familiar with the menu system. Adding menu entries and determining a communication protocol for these commands will become routine as you proceed through this project. If you encounter problems, you can open the terminal emulator and enter the commands manually to narrow down whether the issue is on the client or on the PIC32. Remember, do not attempt to simultaneously open the serial port in the terminal emulator and the client. Also, a mismatch between the data that the PIC32 expects and what the client sends, or vice versa, may cause one or the other to freeze while waiting for data. You can force-quit the client in MATLAB by typing CTRL-C.

When you are comfortable with how the menu system works, you can remove the "d" and "x" commands, as they are no longer needed.

28.4.4 Testing the Encoder

1. Power the encoder with 3.3 V and GND. Be absolutely certain of your wiring before applying power! Some encoders are easy to destroy with the wrong power and ground connections.

2. Attach two oscilloscope probes to the A and B encoder outputs.
3. Twist the motor in both directions and make sure you see out-of-phase square waves from the two encoder channels.

28.4.5 Adding Encoder Menu Options

In this section you will implement the menu items "c" (`Read encoder (counts)`), "d" (`Read encoder (deg)`), and "e" (`Reset encoder`).

Figure 28.6 shows the wiring of the decoder PCB to both the motor encoder and the NU32, based on the assumption that the PIC32 uses SPI channel 4 to communicate with the decoder. The decoder uses 4x decoding of the quadrature encoder inputs and keeps a 16-bit count, 0 to 65,535. The decoder is actually a dedicated PIC microcontroller, programmed only to count encoder pulses and to send the count to the PIC32 when requested. To verify that the decoder chip is programmed, you can look for the 1 kHz "heartbeat" square wave on pin B3.

Once you have connected all the components, it is time to add an encoder reading option to the PIC32 menu code:

```
case 'c':
{
  sprintf(buffer,"%d", encoder_counts());
  NU32_WriteUART3(buffer);  // send encoder count to client
  break;
}
```

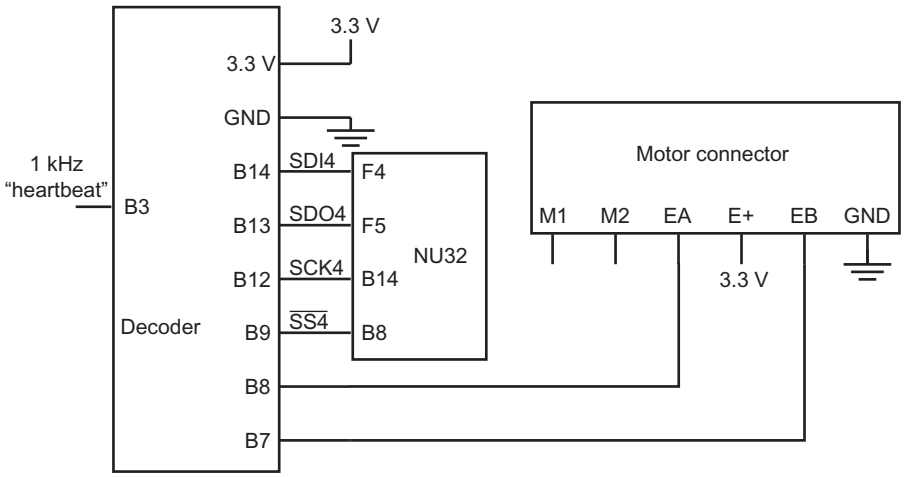

Figure 28.6
Encoder counter circuit.

This code invokes a function called `encoder_counts()`, which apparently returns an integer. Below we give you an implementation of `encoder_counts()`. Here we assume this function lives in a module consisting of `encoder.c` and `encoder.h` (Section 28.3), but it could also be included directly in the `main.c` file with no header `encoder.h`. The function `encoder_counts()` relies on the helper function `encoder_command()`.

Code Sample 28.3 `encoder.c`**. Implementation of Some Encoder Functions.**

```
#include "encoder.h"
#include <xc.h>

static int encoder_command(int read) { // send a command to the encoder chip
                                       // 0 = reset count to 32,768, 1 = return the count
  SPI4BUF = read;                      // send the command
  while (!SPI4STATbits.SPIRBF) { ; }   // wait for the response
  SPI4BUF;                             // garbage was transferred, ignore it
  SPI4BUF = 5;                         // write garbage, but the read will have the data
  while (!SPI4STATbits.SPIRBF) { ; }
  return SPI4BUF;
}

int encoder_counts(void) {
  return encoder_command(1);
}

void encoder_init(void) {
  // SPI initialization for reading from the decoder chip
  SPI4CON = 0;              // stop and reset SPI4
  SPI4BUF;                  // read to clear the rx receive buffer
  SPI4BRG = 0x4;            // bit rate to 8 MHz, SPI4BRG = 80000000/(2*desired)-1
  SPI4STATbits.SPIROV = 0;  // clear the overflow
  SPI4CONbits.MSTEN = 1;    // master mode
  SPI4CONbits.MSSEN = 1;    // slave select enable
  SPI4CONbits.MODE16 = 1;   // 16 bit mode
  SPI4CONbits.MODE32 = 0;
  SPI4CONbits.SMP = 1;      // sample at the end of the clock
  SPI4CONbits.ON = 1;       // turn SPI on
}
```

The function `encoder_init()` initializes SPI4. This function should be called at the beginning of `main`, while interrupts are disabled. The SPI peripheral uses a baud of 8 MHz, 16-bit operation, sampling on the falling edge, and automatic slave detect.

The function `encoder_counts()` uses `encoder_command()` to send a command to the decoder chip and return a response. Valid commands to `encoder_command()` are 0x01, which reads the decoder count, and 0x00, which resets the count to 32,768, halfway through the count range 0 to 65,535. Note that every time you write to SPI you must also read, even if you do not need the data.

If you use a separate `encoder` module (Section 28.3), `encoder_init()` and `encoder_counts()` should have prototypes in `encoder.h` so that they are available to other modules that `#include`

"encoder.h". The function encoder_command() is private to encoder.c, and therefore should be declared static and should have no prototype in encoder.h.

In addition to encoder_counts(), you will implement a function to reset the encoder count (e.g., encoder_reset()) using encoder_command().

1. Implement the PIC32 code to read the encoder counts (the "c" command).
2. Use a terminal emulator to issue commands and display the results from the PIC32. Twist the motor shaft, issue the "c" command, and ensure that the count increases as the shaft rotates counterclockwise and decreases as it rotates clockwise. If you get opposite results, swap the encoder inputs to the decoder PCB.
3. Implement PIC32 code for the "e" command to reset the encoder count to 32,768. Using the terminal emulator and the "c" command, verify that resetting the encoder works as expected.
4. Knowing the 4x resolution of the encoder, implement PIC32 code for the "d" command to read the encoder in degrees. Using the terminal emulator, verify that the angle reads as zero degrees after the reset command "e." Also verify that rotating the shaft 180 or −180 degrees results in appropriate readings with the "d" command.
5. Close the terminal emulator and update the client to handle the "c" (Read encoder (counts)), "d" (Read encoder (deg)), and "e" (Reset encoder) commands. For example,

```
case 'c'
  counts = fscanf(mySerial,'%d');
  fprintf('The motor angle is %d counts.', counts)
```

   Verify that the three commands work as expected on the client.

From now on, when you are asked to implement menu items, you should implement them on both the PIC32 and client. We will not ask you to test with the terminal emulator first. You can always fall back to the terminal emulator for debugging purposes.

28.4.6 PIC32 Operating Mode

In this section you will implement the menu item "r" (Get mode).

The PIC32 can be in one of five operating modes: IDLE, PWM, ITEST, HOLD, and TRACK. You will create functions to both set and query the mode. If you are following the modular design suggested in Section 28.3, these functions, and the variable holding the mode, will likely be in the utilities module.

1. Implement PIC32 functions to set the mode and to query the mode.
2. Use the mode-setting function at the beginning of main to put the PIC32 in IDLE mode.
3. Implement the "r" menu item (Get mode) on the PIC32 and the client. Verify that it works.
4. Update the "q" (quit) menu entry to set the PIC32 to the IDLE state prior to exiting the menu.

Note that there is no client menu command that only sets the mode.

28.4.7 Current Sensor Wiring and Calibration

The current sensor detects the amount of current flowing through the motor. We use a PCB breaking out the MAX9918 current-sense amplifier and an onboard 15 mΩ current-sense resistor, as described in Chapter 21.10.1.

In this section, you will first use the information in Chapter 21.10.1 to set up and calibrate your current sensor, independent of the NU32 and the motor. The questions refer to the circuit in Figure 21.22.

1. Choose the voltage divider resistors R3 to be a few hundred ohms (e.g., 330 Ω).
2. Find the maximum current you expect to sense. If the H-bridge's battery voltage is V and the motor resistance is R_{motor}, then the maximum current you can expect to see is approximately $I_{\max} = 2V/R_{\text{motor}}$. This occurs when the motor is spinning at no-load speed in reverse, with essentially zero current and $-V$ across the motor terminals,

$$-V = k_t \omega_{\text{rev}} \quad \rightarrow \quad \omega_{\text{rev}} = -V/k_t$$

   and then the control voltage switches suddenly to V, yielding (ignoring inductance)

$$V = k_t \omega_{\text{rev}} + I_{\max} R_{\text{motor}} \quad \rightarrow \quad I_{\max} = 2V/R_{\text{motor}}.$$

   Record your calculated $I_{\max}$ for your battery and motor.
3. Calculate the voltage across the 15 mΩ sense resistor if $I_{\max}$ flows through it. Call this $V_{\max}$.
4. Choose resistors R1 and R2 so the current-sense amplifier gain $G = 1 + (\text{R2/R1})$ approximately satisfies
   $1.65\ \text{V} = G \times V_{\max}$.
   This ensures that the maximum positive motor current yields a 3.3 V output from the current sensor and the maximum negative motor current yields a 0 V output from the current sensor, utilizing the full range of the ADC input. Choose R1 and R2 to be in the range of 10^4-10^6 Ω.
5. Choose a resistor R and a capacitor C to make an RC filter on the MAX9918 output with a cutoff frequency $f_c = 1/(2\pi RC)$ in the neighborhood of 200 Hz, to suppress high-frequency components due to the 20 kHz PWM.
6. Build the circuit as shown in Figure 21.22, but do not connect to the motor or the PIC32. You will calibrate the circuit using resistors, an ammeter, and an oscilloscope or voltmeter. Figure 28.7 shows how to use a resistor R0 to provide controlled positive and negative test currents to the current sensor. You will choose different values of R0 to create test currents over the range of likely currents. For example, if you have two 20 Ω resistors, you can use

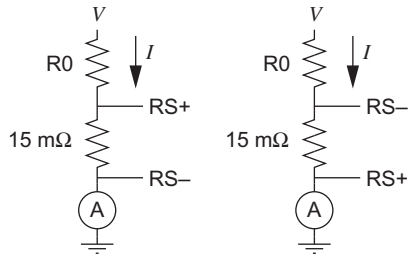

Figure 28.7
Test circuits for calibrating the current sensor. The circuit on the left provides positive currents through the current-sense resistor; switching the RS+ and RS− connections in the right circuit results in negative currents.

them to create an R0 of 20 Ω, 40 Ω (two resistors in series), or 10 Ω (two resistors in parallel). If the battery voltage V is 6 V, this results in expected currents of ±300, ±150, and ±600 mA, respectively.

Important: The calibration resistors must be rated to handle high currents without burning up. For example, a 20 Ω resistor with 300 mA through it dissipates $(300\text{ mA})^2(20\ \Omega) = 1.8$ W, more than a typical 1/4 W resistor can dissipate.

With different resistances R0, use an ammeter to measure the actual current and a voltmeter or oscilloscope to measure the output of the current sensor. Fill out a table similar to the table below, for your particular resistances and battery. If you built your current sensor circuit correctly, zero current should give approximately 1.65 V at the sensor output, and the data points (sensor voltage as a function of the measured current) should agree with the amplifier gain G you designed. If not, time to fix your circuit.

R0 (Ω)	Expected *I* (mA)	Measured *I* (mA)	Sensor (V)	ADC (counts)
10 (to RS+)	600	587	2.82	
20 (to RS+)	300	295	2.24	
40 (to RS+)	150	140	1.93	
Open circuit	0	0	1.63	
40 (to RS−)	−150	−147	1.34	
20 (to RS−)	−300	−322	1.01	
10 (to RS−)	−600	−605	0.45	

Leave the column "ADC (counts)" blank; you will fill in that column in the next section. As a sanity check, you can replace R0 with your motor, stalled, and make sure that the sensor voltage makes sense.

7. **Turn in your answers for items 2-6.**

28.4.8 ADC for the Current Sensor

In this section you will implement the menu items "a" (`Read current sensor (ADC counts)`) and "b" (`Read current sensor (mA)`).

The ADC reads the voltage from the motor current sensor. See Chapter 10 for information on setting up and using the ADC.

1. Create a PIC32 function to initialize the ADC, and call it at the beginning of `main`.
2. Create a PIC32 function that reads the ADC and returns the value, 0-1023. Consider reading the ADC a few times and averaging for a more stable reading.
3. Add the menu item "a" (`Read current sensor (ADC counts)`) to the PIC32 and client and verify that it works. Use simple voltage dividers at the analog input to make sure that the readings make sense.
4. Hook up the current sensor output from the previous section to the analog input. Provide a circuit diagram showing all connections to the current sensor PCB.
5. Using the power-resistor voltage dividers from the previous section, and the "a" menu command, fill in the last column of the current sensor calibration table. (It is a good idea to update your ammeter-measured currents, too, just in case they have changed due to a draining battery.)
6. Plot your data points, measured current in mA as a function of ADC counts. Find the equation of a line that best fits your data (e.g., using least-squares fitting in MATLAB).
7. Use the line equation in a PIC32 function that reads the ADC counts and returns the calibrated measured current, in mA. Add the menu item "b" (`Read current sensor (mA)`) and verify that it works as expected.

28.4.9 PWM and the H-Bridge

In this section you will implement the menu items "f" (`Set PWM (-100 to 100)`) and "p" (`Unpower the motor`).

By now you should have control of both the current sensor and the encoder. The next step is to provide low-level motor control. First you will implement part of the software associated with the current control loop. Next you will connect the H-bridge and the motor. When you finish this section you will be able to control the motor PWM signal from the client.

1. The current controller uses a timer for the 5 kHz ISR, another timer and an output compare to generate a 20 kHz PWM signal, and a digital output to control the motor direction. Write a PIC32 function that initializes these peripherals and call it from `main`.
2. Write the 5 kHz ISR. It should set the PWM duty cycle to 25% and invert the motor direction digital output. Look at the digital output and the PWM output on an oscilloscope

and confirm that you see a 2.5 kHz "heartbeat" square wave for the ISR and a 25% duty cycle 20 kHz PWM signal. Remember to clear the interrupt flag.

3. Now modify the ISR to choose the PWM duty cycle and direction bit depending on the operating mode. You should use a `switch-case` construct, similar to the `switch–case` in `main`, except the value in question here is the operating mode, as returned by the mode-querying function developed in Section 28.4.6. There will eventually be five modes to handle—IDLE, PWM, ITEST, HOLD, and TRACK—but in this section we focus on IDLE and PWM. If the operating mode is IDLE, the PWM duty cycle and direction bit should put the H-bridge in brake mode. If the operating mode is PWM, the duty cycle and direction bit are set according to the value −100 to 100 specified by the user through the client. This leads to the next action item. . .
4. Implement the menu item "f" (`Set PWM (-100 to 100)`). The PIC32 switches to PWM mode, and in this mode the 5 kHz ISR creates a 20 kHz PWM pulse train of the specified duty cycle and a digital output with the correct direction bit.
5. Implement the menu item "p" (`Unpower the motor`). The PIC32 switches to IDLE mode.
6. Test whether the mode is being changed properly in response to the new "f" and "p" commands by using the menu item "r" (`Get mode`).
7. Set the PWM to 80%. Verify the duty cycle with an oscilloscope and record the value of the direction pin. Then set the PWM to −40%. Verify the new duty cycle and that the direction pin has changed.
8. Now that the PWM output appears to be working, it is time to wire up the DRV8835 H-bridge circuit, as discussed in Chapter 27.1.1, to the motor and the PIC32 outputs (Figure 28.8). Notice that the 15 mΩ resistor on the current-sense PCB is in series with the motor. **Turn in a circuit diagram showing all connections of the H-bridge to the NU32, motor, and current sensor PCB.**

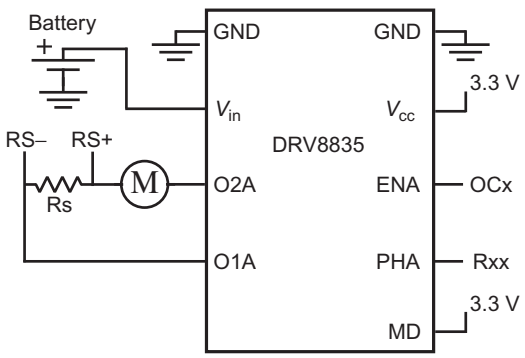

Figure 28.8
H-bridge circuit.

9. Verify the following:
   a. Set the PWM to 100%. Make sure that the motor rotates counterclockwise, that the angle returned by the encoder is increasing, and that the measured current is positive. You may have to swap the motor terminals or the encoder channels if not.
   b. Stall the motor at 100% PWM and see that the current is greater than during free running, and check that the measured current is consistent with your estimate of the resistance of the motor. (Note that the voltage at the H-bridge outputs will be somewhat lower than the voltage of the battery, due to voltage drops at the output MOSFETs.)
   c. Set the PWM to 50% and make sure that the motor spins slower than at 100%.
   d. Repeat the steps above for negative values of PWM.
   e. Make sure the motor stops when you issue the "p" (`Unpower the motor`) command.
   f. Attach the bar to the motor to increase the inertia, if it was not attached already. Get the motor spinning at its max negative speed with PWM set at −100%. Then change the PWM to 100% and quickly query the motor current ("a") several times as the motor slows down and then reverses direction on its way to its max positive speed. You should see the motor current is initially very large due to the negative back-emf, and drops continuously as the back-emf increases toward its maximum positive value (when the motor is at full speed in the forward direction).

You now have full control of the low-level features of the hardware!

28.4.10 PI Current Control and ITEST Mode

In this section you will implement menu items "g" (`Set current gains`), "h" (`Get current gains`), and "k" (`Test current control`).

The PI current controller tries to make the current sensor reading match a reference current by adjusting the PWM signal. For details about PI controllers, see Chapters 23 and 27.2.2.

In this section we focus particularly on the ITEST mode in the 5 kHz current control ISR.

1. Implement the menu items "g" (`Set current gains`) and "h" (`Get current gains`) to set and read the current loop's proportional and integral gains. You can use either floating point or integer gains. Verify the menu items by setting and reading some gains.
2. In the 5 kHz current control ISR, add a case to the `switch-case` to handle the ITEST mode. When in ITEST mode, the following should happen in the ISR:
   - In this mode, the current controller attempts to track a ±200 mA 100 Hz square wave reference current (Figure 28.4). Since a half-cycle of a 100 Hz signal is 5 ms, and 5 ms at 5000 samples/s is 25 samples, this means that the reference current toggles between +200 and −200 mA every 25 times through the ISR. To implement two full cycles of the current reference, you could use a `static int` variable in the ISR that counts

from 0 to 99. At 25, 50, and 75, the reference current changes sign. When the counter reaches 99, the PIC32 mode should be changed to IDLE—the current loop test is over.

- A PI controller reads the current sensor, compares it to the square wave reference, and calculates a new PWM duty cycle and motor direction bit.
- The reference and actual current data are saved in arrays for later plotting (e.g., Figure 28.4).

3. Add the menu item "k" (`Test current gains`). This puts the PIC32 in ITEST mode, triggering the ITEST case in the `switch-case` in the current control ISR. When the PIC32 returns to IDLE mode, indicating that the ITEST has completed, the data saved during the ITEST should be sent back to the client for plotting (e.g., Figure 28.4). This data transfer should not occur in an ISR, as it will be slow. See below for sample MATLAB code for plotting the ITEST data.
4. Experiment by setting different current gains ("g") and testing them with the square wave reference current ("k"). Verify that the measured current is approximately zero if both PI gains are zero. See how good a response you can get with a proportional gain only. Finally, get the best response possible using both the P and I gains.
5. **Turn in your best ITEST plot, and indicate the control gains you used, as well as their units.**

We provide you with a sample MATLAB function to read and plot the ITEST data (Figure 28.4). This code assumes that the PIC32 first sends the number of samples *N* it will be sending, then sends *N* pairs of integers: the reference and the actual current, in mA.

Code Sample 28.4 `read_plot_matrix.m`**. Reads a Matrix of Current Data from the PIC32 and Plots the Results. It also Computes an Average Tracking Error, to Help You Evaluate the Current Controller.**

```
function data = read_plot_matrix(mySerial)
  nsamples = fscanf(mySerial,'%d');       % first get the number of samples being sent
  data = zeros(nsamples,2);               % two values per sample:  ref and actual
  for i=1:nsamples
    data(i,:) = fscanf(mySerial,'%d %d'); % read in data from PIC32; assume ints, in mA
    times(i) = (i-1)*0.2;                 % 0.2 ms between samples
  end
  if nsamples > 1
    stairs(times,data(:,1:2));            % plot the reference and actual
  else
    fprintf('Only 1 sample received\n')
    disp(data);
  end
  % compute the average error
  score = mean(abs(data(:,1)-data(:,2)));
  fprintf('\nAverage error: %5.1f mA\n',score);
  title(sprintf('Average error: %5.1f mA',score));
  ylabel('Current (mA)');
  xlabel('Time (ms)');
end
```

28.4.11 Position Control

In this section you will implement the menu items "i" (`Set position gains`), "j" (`Get position gains`), and "l" (`Go to angle (deg)`).

Of the five PIC32 operating modes, only HOLD and TRACK are relevant to the position controller. If in either of these modes, the 200 Hz position control ISR specifies the reference current for the current controller to track.

Complete the following steps in order.

1. Implement the menu items 'i" (`Set position gains`) and "j" (`Get position gains`). The type of controller is up to you, but a reasonable choice is a PID controller, requiring three numbers from the user. Verify that the "i" and "j" menu items work by setting and reading gains from the client.
2. The position controller uses a timer to implement a 200 Hz ISR. Write a position controller initialization function that sets up the timer and ISR and call it at the beginning of `main`.
3. Write the 200 Hz position control ISR. In addition to clearing the interrupt flag, have it toggle a digital output, and verify the 200 Hz frequency of the ISR with an oscilloscope.
4. Implement the menu item "l" (`Go to angle (deg)`). The user enters the desired angle of the motor, in degrees, and the PIC32 switches to HOLD mode. In the 200 Hz position control ISR, check if the PIC32 is in the HOLD mode. When in the HOLD mode, the ISR should read the encoder, compare the actual angle to the desired angle set by the user, and calculate a reference current using the PID control gains. It is up to you whether to compare the angles in terms of encoder counts or degrees or some other unit (e.g., an integer number of tenths or hundredths of degrees).
   You also need to add the HOLD case to the 5 kHz current control ISR. When in the HOLD mode, the current controller uses the current commanded by the position controller as the reference for the PI current controller.
5. With the bar on the motor, verify that the "l" (`Go to angle (deg)`) command works as expected. Find control gains that hold the bar stably at the desired angle, and move the bar to the new desired angle on the next "l" command. Try to choose gains that give a quick motion with little overshoot. It may be easiest to use zero integral gain.

28.4.12 Trajectory Tracking

All that remains is to implement the menu items "m" (`Load step trajectory`), "n" (`Load cubic trajectory`), and "o" (`Execute trajectory`). These commands allow us to design a reference trajectory for the motor, execute it, and see the position controller tracking results.

We have provided a MATLAB function `genRef.m`, below, to generate and visualize step and cubic trajectories similar to those seen in Figure 28.5. For trajectories that are T seconds long, `genRef.m` generates an array of $200T$ reference angles, one per 200 Hz control loop iteration, and plots it in MATLAB. Before continuing, try generating some sample trajectories using `genRef.m`. See the description of menu items "m" and "n" in Section 28.2 for more information on `genRef.m`.

Code Sample 28.5 `genRef.m`**. MATLAB Code to Generate a Step or Cubic Reference Trajectory.**

```
function ref = genRef(reflist, method)

% This function takes a list of "via point" times and positions and generates a
% trajectory (positions as a function of time, in sample periods) using either
% a step trajectory or cubic interpolation.
%
%   ref = genRef(reflist, method)
%
%   Input Arguments:
%       reflist: points on the trajectory
%       method: either 'step' or 'cubic'
%
%   Output:
%       An array ref, each element representing the reference position, in degrees,
%       spaced at time 1/f, where f is the frequency of the trajectory controller.
%       Also plots ref.
%
%   Example usage:  ref = genRef([0, 0; 1.0, 90; 1.5, -45; 2.5, 0], 'cubic');
%   Example usage:  ref = genRef([0, 0; 1.0, 90; 1.5, -45; 2.5, 0], 'step');
%
%   The via points are 0 degrees at time 0 s; 90 degrees at time 1 s;
%   -45 degrees at 1.5 s; and 0 degrees at 2.5 s.
%
%   Note:  the first time must be 0, and the first and last velocities should be 0.

MOTOR_SERVO_RATE = 200;     % 200 Hz motion control loop
dt = 1/MOTOR_SERVO_RATE;    % time per control cycle

[numpos,numvars] = size(reflist);

if (numpos < 2) || (numvars ~= 2)
  error('Input must be of form [t1,p1; ... tn,pn] for n >= 2.');
end
reflist(1,1) = 0;           % first time must be zero
for i=1:numpos
  if (i>2)
    if (reflist(i,1) <= reflist(i-1,1))
      error('Times must be increasing in subsequent samples.');
    end
  end
end

if strcmp(method,'cubic')  % calculate a cubic interpolation trajectory

  timelist = reflist(:,1);
  poslist = reflist(:,2);
```

```
    vellist(1) = 0; vellist(numpos) = 0;
    if numpos >= 3
      for i=2:numpos-1
        vellist(i) = (poslist(i+1)-poslist(i-1))/(timelist(i+1)-timelist(i-1));
      end
    end

    refCtr = 1;
    for i=1:numpos-1            % go through each segment of trajectory
      timestart = timelist(i); timeend = timelist(i+1);
      deltaT = timeend - timestart;
      posstart = poslist(i); posend = poslist(i+1);
      velstart = vellist(i); velend = vellist(i+1);
      a0 = posstart;            % calculate coeffs of traj pos = a0+a1*t+a2*t^2+a3*t^3
      a1 = velstart;
      a2 = (3*posend - 3*posstart - 2*velstart*deltaT - velend*deltaT)/(deltaT^2);
      a3 = (2*posstart + (velstart+velend)*deltaT - 2*posend)/(deltaT^3);
      while (refCtr-1)*dt < timelist(i+1)
        tseg = (refCtr-1)*dt - timelist(i);
        ref(refCtr) = a0 + a1*tseg + a2*tseg^2 + a3*tseg^3;  % add an element to ref array
        refCtr = refCtr + 1;
      end
    end

  else  % default is step trajectory

  % convert the list of times to a list of sample numbers
    sample_list = reflist(:,1) * MOTOR_SERVO_RATE;
    angle_list = reflist(:,2);
    ref = zeros(1,max(sample_list));
    last_sample = 0;
    samp = 0;
    for i=2:numpos
      if (sample_list(i,1)  <= sample_list(i-1,1))
        error('Times must be in ascending order.');
      end
      for samp = last_sample:(sample_list(i)-1)
        ref(samp+1) = angle_list(i-1);
      end
      last_sample = sample_list(i)-1;
    end
    ref(samp+1) = angle_list(end);

  end

  str = sprintf('%d samples at %7.2f Hz taking %5.3f sec', ...
        length(ref),MOTOR_SERVO_RATE,reflist(end,1));
  plot(ref);
  title(str);
  border = 0.1*(max(ref)-min(ref));
  axis([0, length(ref), min(ref)-border, max(ref)+border]);
  ylabel('Motor angle (degrees)');
  xlabel('Sample number');
  set(gca,'FontSize',18);
```

In the MATLAB client, you can use code of the form

```
A = input('Enter step trajectory: ');
```

to read the user's matrix of times and positions into the variable `A`, to send to `genRef.m`. The user would simply type a string similar to `[0,0; 1,180; 3,-90; 4,0; 5,0]` in response.

1. Implement the menu item "m" (`Load step trajectory`). The client should prompt the user for the trajectory parameters for `genRef.m`. Provided the duration of the trajectory is not too long to store in the PIC32's data array, the client first sends the number of samples *N* to the PIC32, then sends *N* reference positions. It is up to you whether the reference positions are sent as floating point numbers (e.g., degrees) or integers (e.g., tenths of degrees, hundredths of degrees, or encoder counts). But the user should only ever have to deal with degrees, in specifying angles and looking at plots.
2. Implement the menu item "n" (`Load cubic trajectory`). This entry is very similar to the previous item, except `genRef.m` is invoked with the `'cubic'` option.
3. Implement the menu item "o" (`Execute trajectory`). The PIC32 is placed in TRACK mode. In the 200 Hz position control ISR, check if the mode is TRACK. If so, then the ISR increments an index into the reference trajectory array, and the indexed trajectory position is used as the reference to the PID controller, which calculates a commanded current for the current control ISR.
   The position control ISR should also collect motor angle data for later plotting.
   When the array index reaches *N*, the operating mode is switched to HOLD, and the last angle of the reference trajectory array is used as the holding angle. The collected data is sent back to the client for plotting, similar to the ITEST case. The MATLAB client plotting code should be similar to Code Sample 28.4, but using the appropriate data types and scaling of the sample times, so the user sees the results in terms of time, not samples. You also need to add the TRACK case to the 5 kHz current control ISR. To the current controller, the TRACK case is identical to the HOLD case: in both cases, the current controller attempts to match the current commanded by the position controller.
4. Experiment with tracking different trajectories (such as the ones in Figure 28.5) with different position control gains until you get good performance. For example, the performance in Figure 28.2 is reasonable, though a larger derivative gain would help to eliminate the overshoot. In experimentally tuning your gains, it is easiest to start with proportional (P) control alone, then add derivative (D) control. Finally, tune your PD gains simultaneously. You may not need an integral (I) term for good performance.
5. **Turn in your best plots of following the step and cubic trajectories in Figure 28.5 with the load attached. Indicate the control gains you used, as well as their units.**

Congratulations! You now have a full motor control system.

28.5 Extensions

Saving gains in flash memory

Currently you must re-enter gains every time you reset the PIC32; this quickly becomes annoying. You can, however, store the gains in flash memory, which allows them to persist,

even when the PIC32 is powered off. To see how to access flash memory, refer to Chapter 18. Add a menu item that saves gains to flash memory, and modify the startup code in `main` to first read any stored gains.

LCD display

Connect an LCD display to the PIC32. Use it to show information about the current system state, the gains, or whatever else you want.

Model-based control

In Chapter 25, we learned how to characterize a motor. Notice that we have mostly ignored the motor parameters when implementing the control law. Our ability to do this demonstrates the power of feedback to compensate for a lack of a system model. However, incorporating a model of the motor and load into the control loop could improve the motor's performance. See Chapter 27.2.1, for example.

To test your model-based control, try adding a small weight to one end of the bar and track the trajectories in Figure 28.5. Either hardcode the mass, center of mass, and inertia of the total load in your PIC32 code, or allow the user to enter information about the load. See if your model-based control can provide better tracking than your original controller.

As an even more advanced project, you could identify the properties of the motor and the load by spinning the motor, measuring the current (torque) as a function of the position, velocity, and acceleration of the motor, and using the data to fit parameters of a model.

Anytime data collection

Instead of only collecting data during ITEST or TRACK modes, the user could have the option to collect and plot a specified duration of data at any time. This would allow collecting data during a HOLD while the user perturbs the motor, for example.

Real-time data

Currently we employ batch processing to retrieve motor data. This severely limits how long you can collect data before running out of memory. What if you want to see the motor data in real-time? A data structure called a circular (a.k.a. ring) buffer can help. The circular buffer has two position indexes; one for reading and one for writing. Data is added to the array at the write position, which is subsequently advanced. If the end of the array is reached, the write position wraps around to the beginning. Data is read from the read position and the read index advanced, also wrapping around. In one style of circular buffer, if the read position and write position are the same, the buffer is empty. If the write position is one behind the read position, the buffer is full. When using a circular buffer in this project, either the current loop or

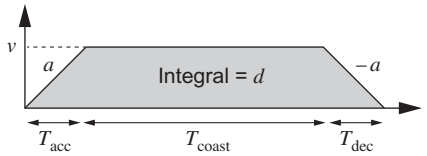

Figure 28.9
A trapezoidal move of length d.

position loop will add data to the buffer. Rather than waiting for the buffer to be full, data can be sent back any time the buffer is not empty. It is up to the client how to handle this continuous stream of data, perhaps by using an oscilloscope-style display. See Chapter 11 for more details about circular buffers.

If the communication cannot keep up with the data, then the stored data can be decimated, sending back only every nth data sample, $n > 1$.

Trapezoidal moves

The reference trajectories in this chapter are rest-to-rest step or cubic motions in position. One type of rest-to-rest trajectory that is common in machine tools is the trapezoidal move. The name comes from the fact that the trajectory, represented in the velocity space, is a trapezoid (Figure 28.9): the motor accelerates with a constant acceleration a for a time T_{acc} until it reaches a maximum velocity v; it coasts with a constant velocity v for a time T_{coast}; and then it comes to a stop by decelerating at $-a$ for a time $T_{\text{dec}} = T_{\text{acc}}$. The user should specify the total move distance d and either (a) the total time $T = T_{\text{acc}} + T_{\text{coast}} + T_{\text{dec}}$ and the maximum velocity v, (b) the acceleration a and the maximum velocity v, or (c) T and a. The other parameters are calculated so that the integral of the velocity trapezoid is equal to d.

Add a menu item that allows the user to generate a trapezoidal reference trajectory based on either (a), (b), or (c).

28.6 Chapter Summary

A typical commercial motor amplifier consists of at least a microcontroller, a high-current H-bridge output, a current sensor, and some type of motion feedback input (e.g., from an encoder or a tachometer). The amplifier is connected to a high-power power supply to power the H-bridge. The amplifier accepts a velocity or current/torque input from an external controller (e.g., a PC), either as an analog signal or the duty cycle of a PWM signal. When the amplifier is in current mode, current sensor feedback is used to alter the PWM input to the H-bridge to achieve close tracking of the commanded current. In velocity mode, the amplifier uses motion feedback to alter the PWM input to achieve close tracking of the desired velocity.

More advanced motor amplifiers, like the Copley Accelus, offer other features, like client graphical user interfaces and trajectory tracking. The project in this chapter emulates some of the capabilities that come with advanced motor amplifiers. This project brings together your knowledge of the PIC32 microcontroller, C programming, brushed DC motors, feedback control, and interfacing a microcontroller with sensors and actuators to achieve capabilities found in every robot and computer-controlled machine tool.

28.7 Exercises

Complete the motor control project. Complete all of the numbered action items in each subsection. Turn in your answers for the **action items in bold** as well as your well-commented code.