CHAPTER 8

Rotary Encoders

Rotary encoders are useful for conveying position input and adjustments. For example, as a tuning knob for software-defined radio (SDR), the rotary control can indicate two things related to frequency tuning.

- When a step has occurred
- The direction of that step

Determining these two things, the rotary encoder can adjust the tuning in the frequency band in discrete steps higher or lower. This chapter will examine the rotary encoder with the help of the economical Keyes KY-040 device and the software behind it.

Keyes KY-040 Rotary Encoder

While the device is labeled an "Arduino Rotary Encoder," it is also quite usable in non-Arduino contexts. A search on eBay for *KY-040* shows that a PCB and switch can be purchased (assembled) from eBay for about $1.28. Figure 8-1 shows both sides of the PCB unit I purchased.

© Warren Gay 2017
W. Gay, *Custom Raspberry Pi Interfaces*, DOI 10.1007/978-1-4842-2406-9_8

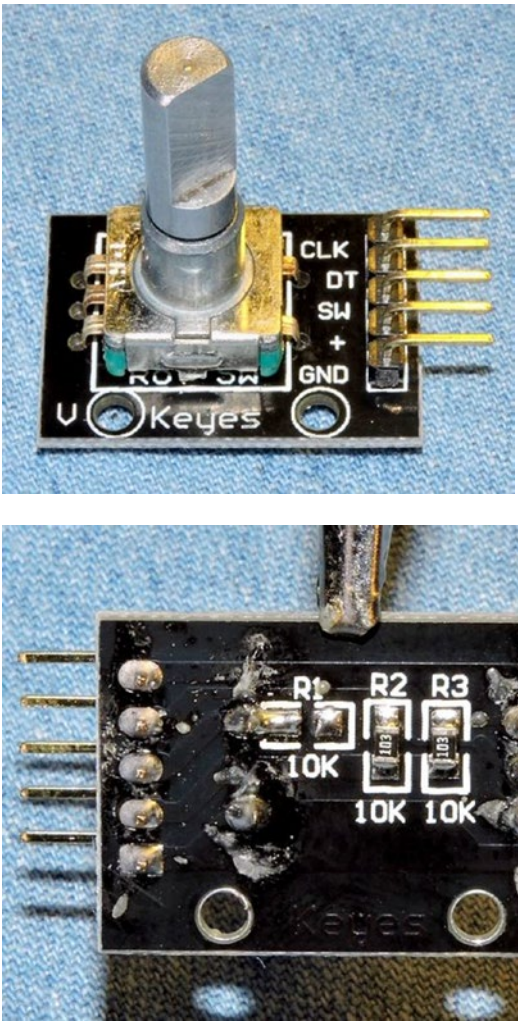

Figure 8-1. *The Keyes YL-040 rotary encoder*

It is also possible to buy the encoder switch by itself on eBay, but I don't recommend it. The PCB adds considerable convenience for pennies more than the cost of the switch. But the main reason to buy the assembled PCB unit is to get a *working* switch. I have given up on buying good rotary switches by themselves from eBay. I suspect that many eBay rotary switch offerings, if not all, are factory rejects and floor sweepings.

The Switch

To help convince you of the need for buying quality, let's take an inside look at how switches are constructed. Figure 8-2 illustrates the contact side of a switch assembly that I did an autopsy on.

Figure 8-2. *Contact side of the rotary encoder*

Figure 8-2 shows that there is a contact pad (lower left) for the wiper arm. The top portion (right side of the figure) shows the other contact points. The smear seen there is some conductive grease. Figure 8-3 illustrates the wiper half of the assembly.

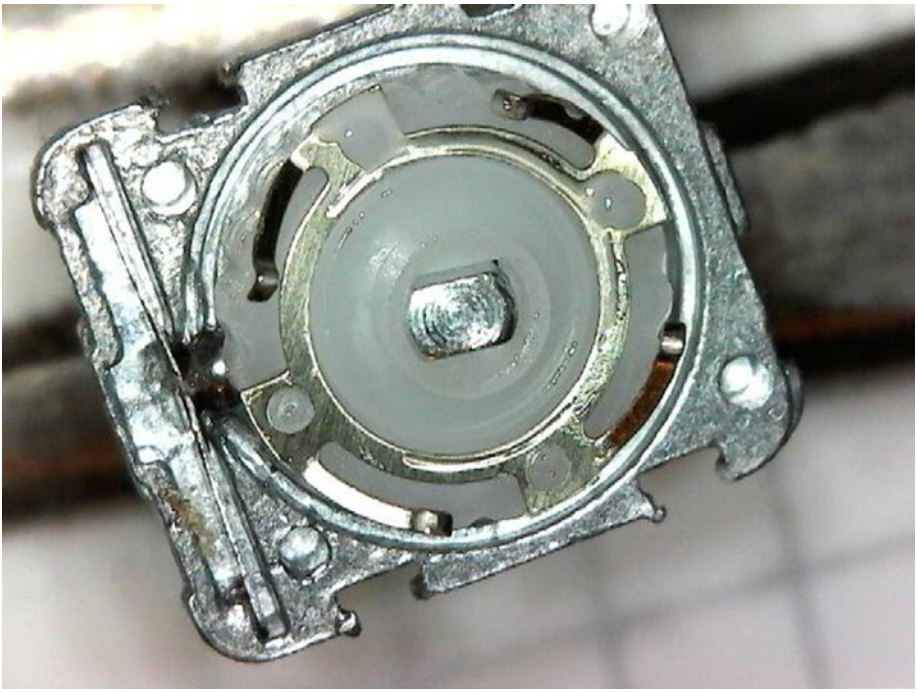

Figure 8-3. *The wiper assembly of the rotary encoder*

The photos illustrate the cheap nature of the rotary encoders' construction. They must go through some sort of quality control at the factory, but it wouldn't take much more than a bent wiper arm to render one faulty.

Figure 8-4 shows the KY-040 schematic without the optional push button switch. The units with the optional push switch have another connection labeled "SW" on the PCB. The return side of that switch connects to ground (GND).

The schematic shown focuses on the rotary switches that connect switches A and B to the common point C. The KY-040 PCB includes two 10kΩ pull-up resistors so that when either switch is open, the host will read a high (1 bit). When the A or B switch is closed, this brings the signal level low (0 bit). Finally, note that the KY-040 PCB labels the connections as CLK and DT. According to reference [1], switch A is the CLK connection, while B is the DT connection. These specifics are unimportant to its operation.

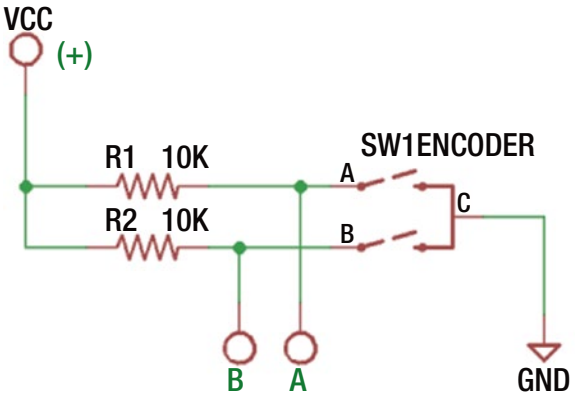

Figure 8-4. *The KY-040 schematic (excluding optional push switch)*

Operation

The rotary encoder opens and closes switches A and B as you rotate the knob. As the shaft is turns, both switches alternate between on and off at the detents. The magic occurs *in between* the detents, however, allowing you to determine the direction of travel. Figure 8-5 illustrates this effect.

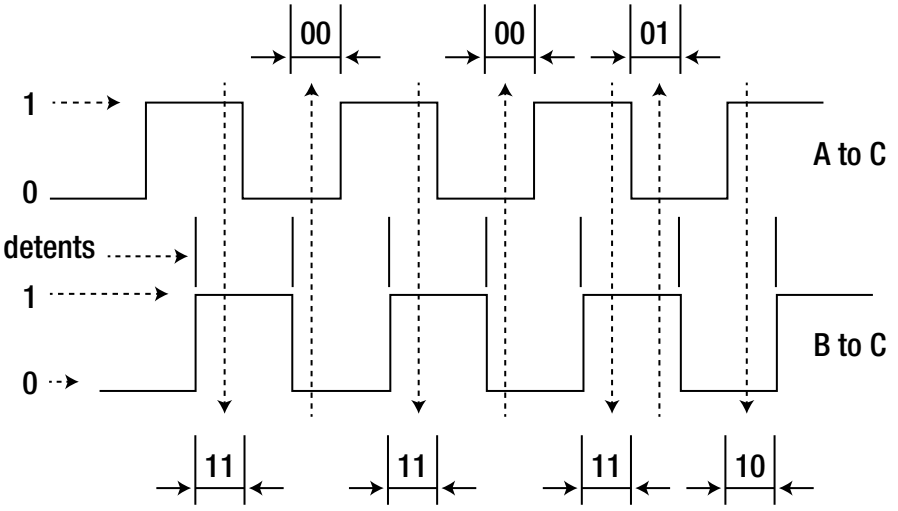

Figure 8-5. *Rotary encoder signals*

Starting from the first detent shown in Figure 8-5, switches A and B are open, reading 11 in bits (both read high). As the shaft is rotated toward the next detent, switches A and B become both closed (reading as 00). *In between* the detents however, you can see that switch A closes first (reading as 01), followed by switch B later (now reading 00). The timing of these changes allows you to determine the direction of travel. Had the shaft rotated in the reverse direction, switch B would close first (reading as 10), followed by A.

Voltage

As soon as you see the word *Arduino*, you should question whether the device is a 5V device because many Arduinos are 5V-based. For the KY-040 PCB, it doesn't matter. Even though the reference [1] indicates that the supply voltage is 5 volts, it can be supplied the Raspberry Pi level of 3.3 volts instead. The rotary encoder is simply a pair of mechanical switches and has no voltage requirement.

Evaluation Circuit

The evaluation circuit can serve as an educational experiment and double as a test of the component. It is simple to wire up on a breadboard, and if the unit is working correctly, you will get a visual confirmation from the LEDs. Figure 8-6 illustrates the full circuit. The added components are shown at the left, consisting of resistors R_3, R_4 and the pair of LEDs.

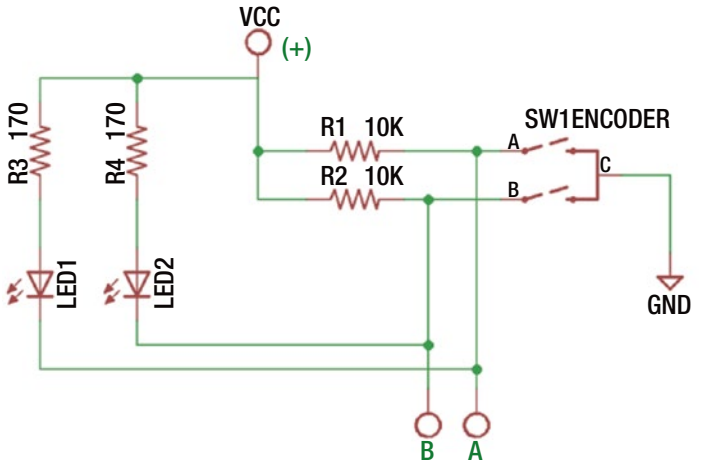

Figure 8-6. *KY-040 evaluation circuit*

The dropping resistors R_3 and R_4 are designed to limit the current flowing through the LEDs. Depending upon the LEDs you use, these should work fine at 3.3 or 5 volts. Assuming a forward voltage of about 1.8 volts for red LEDs and the circuit powered from 3.3 volts, this should result in approximately:

$$\frac{3.3-1.8volts}{170\Omega}=\frac{1.5}{170}=8.8mA$$

If you want to use less current for smaller LEDs, adjust the resistor higher than 170Ω.

Figure 8-7 illustrates my own simple breadboard test setup. You can see the dropping resistors R_3 and R_4 hiding behind the yellow LEDs that I had on hand at the moment. This test was performed using a power supply of 3.3 volts. The photo was taken with the shaft resting at a detent, which had both switches A and B closed, illuminating both LEDs.

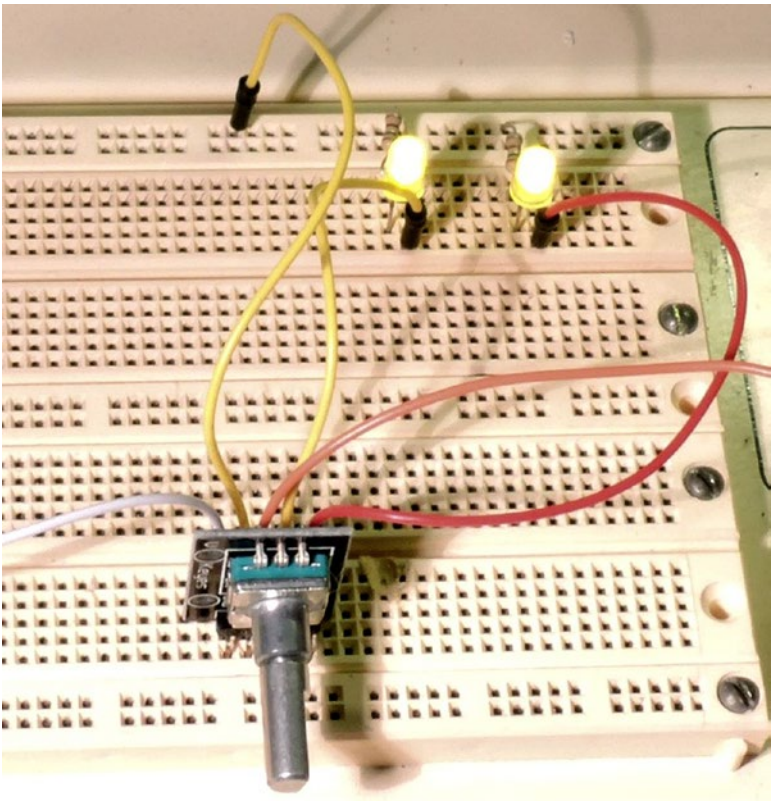

Figure 8-7. *Evaluation test of the rotary encoder*

When rotating the shaft, be sure to hold the shaft (a knob is recommended) securely so that you can see the individual LEDs turn on and off before the shaft snaps to the next detent. You should be able to see individual LED switching as you rotate.

If you see "skips" of LED on/off behavior for LED 1 and/or 2, this may indicate that you have a faulty switch. The only thing you can do for it is to replace it. But do check your wiring and connections carefully before you conclude that.

Interfacing to the Pi

Now it is time to interface this to the Raspberry Pi. Since the KY-040 PCB already provides the pull-up resistors, you only need to connect the + terminal to the Pi's 3.3V supply and wire the A/CLK and B/DT connections to GPIO inputs. Don't forget to connect GND to the Pi's ground.

But before doing so, consider this: what kind of port are the GPIO pins you have selected at boot time? That is, what is their default configuration? If a particular GPIO is an output at boot time and if one or both of the rotary switches are *closed*, then you are short-circuiting the output until the port is reconfigured as an input. Given the amount of boot time, this could result in a damaged GPIO port.

Before hooking up GPIO ports to switches, you should do one of two things:

- Use GPIOs that are known to be inputs at boot time (and stay that way).
- Use isolation resistors to avoid damage when the GPIOs are outputs (*best*).

Using isolation resistors is highly recommended. By doing this, it won't matter if the port is initially (or subsequently) configured as an output port. If your isolation resistor is chosen as 1kΩ (for example), the maximum output current will be limited to 3.3mA, which is too low to cause damage. Figure 8-8 illustrates the Pi hookup, using isolation resistors.

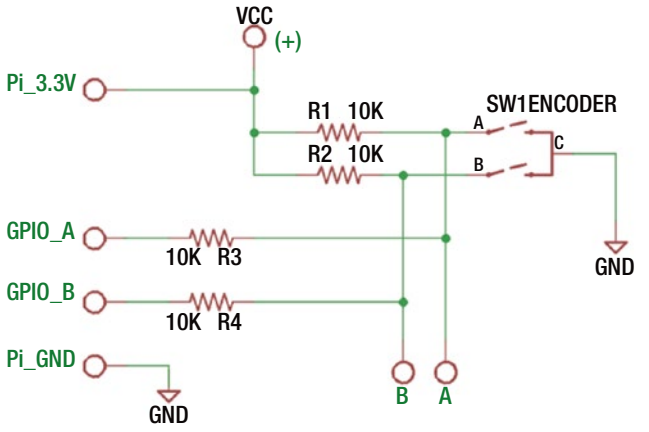

Figure 8-8. *Safe hookup of the rotary switch to the Raspberry Pi*

The schematic uses 10kΩ isolation resistors to hook up the rotary encoder to the Pi. If the selected GPIOs should ever be configured as outputs, the current will be limited to 0.33mA. Once the GPIOs are reconfigured by your application to be *inputs*, the readings will occur as if the resistors were not there. This happens because the CMOS inputs are driven mainly by voltage. The *miniscule* amount of current involved results in almost no voltage drop across the resistors.

Figure 8-9 shows a photo of the breadboard setup. An old knob was put onto the rotary controller to make it easier to rotate. In this experiment, the following GPIOs were used:

- GPIO#20 was wired to CLK (via 10kΩ resistor)
- GPIO#21 was wired to DT (via 10kΩ resistor)

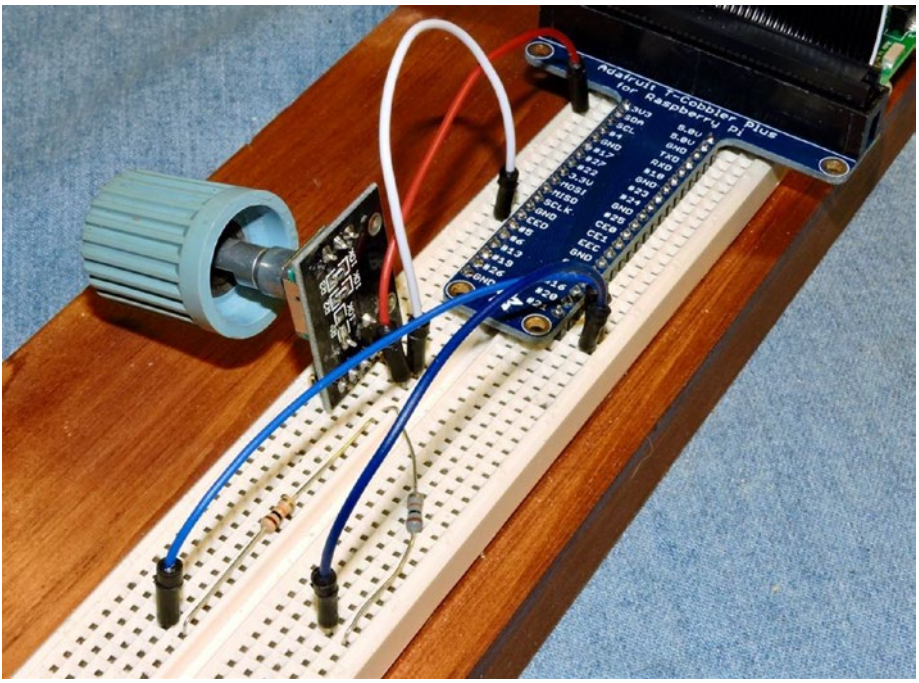

Figure 8-9. *YL-040 rotary encoder hooked up to Raspberry Pi 3*

To verify that things are connected correctly, you can use the gp command to verify operation. Here you are testing GPIO #20 to see that it switches on and off:

```
$ gp -g20 -i -pu -m0
Monitoring..
000000 GPIO 20 = 1
000001 GPIO 20 = 0
000002 GPIO 20 = 1
000003 GPIO 20 = 0
000004 GPIO 20 = 1
000005 GPIO 20 = 0
...
```

Repeat the same test for GPIO #21.

```
$ gp -g21 -i -pu -m0
Monitoring..
000000 GPIO 21 = 0
000001 GPIO 21 = 1
000002 GPIO 21 = 0
000003 GPIO 21 = 1
000004 GPIO 21 = 0
000005 GPIO 21 = 1
000006 GPIO 21 = 0
```

Observe that the switch seems to change cleanly between 1 and 0 without any contact bounce. That may not always happen, however.

Experiment

As a simple programmed experiment, change to the software subdirectory `YL-040`. If you don't see the executable `yl040a`, then type the `make` command to compile it now. With the circuit wired as in Figure 8-8, run the program `yl040a` and watch the session output as you turn the knob clockwise. Use Control-C to exit the program.

```
$ ./yl040a
00
10
11
01
00
^C
```

You should observe the general pattern shown in Table 8-1, perhaps starting at a different point. You should observe this sequence repeating as you rotate the control clockwise. If not, try counterclockwise. If the pattern shown here occurs in the reverse direction, simply exchange the connections for GPIO #20 and GPIO #21.

Table 8-1. *Pattern Observed from Program ./yl040a Run*

GPIO #20	GPIO #21	Remarks
0	0	Both switches closed (detent)
1	0	CLK switch opened, DT switch still closed
1	1	Both switches open (detent)
0	1	CLK switch still closed, DT switch opened
0	0	Both switches closed (detent), pattern repeating

Note that the `00` and `11` patterns occur at the detents. The `10` and `01` patterns occur as you rotate from one detent position to the next. Rotating the knob in the reverse direction should yield a pattern like this:

```
$ ./yl040a
00
01
11
10
00
01
^C
```

Don't worry if you see some upset in the sequence runs. There is likely to be some contact bounce that your next program will have to mitigate. The pattern however, should generally show what Table 8-2 illustrates.

Table 8-2. *Counterclockwise Rotation Run of the ./yl040a Program*

GPIO # 20	GPIO # 21	Remarks
0	0	Both switches closed (detent)
0	1	CLK switch still closed, DT switch opened
1	1	Both switches open (detent)
1	0	CLK switch still open, DT switch closed
0	0	Both switches closed (detent), pattern repeating

From this, you see that rotation in one direction closes or opens a switch ahead of the other, offering information about rotational direction. There is still the problem of contact bounce, which you might have observed from your experiments. The next program will apply debouncing.

Experiment

Chapter 5 showed how software debouncing could be done. You're going to apply that same technique in program `yl040b` for this experiment. Run the program and rotate the knob clockwise.

```
$ ./yl040b
00
10
11
01
00
10
```

```
11
01
00
10
11
01
00
10
11
01
00
10
11
^C
```

In this output, you can see the debounced sequence is faithful until the end. Despite this, it is possible that if you rotated the knob fast enough, there may have been discontinuities in the expected sequence. You'll need to address this in the software.

Sequence Errors

Error handling in device driver software can be tricky. Let's begin by looking at what can go wrong. Then let's decide on an error recovery strategy. Here are events that could go astray:

- Debouncing was not effective enough, returning erratic events.
- The control was rotated faster than can be read and debounced.
- The control became defective or a connection was lost.

You'll look at code at the end of the chapter, but program `yl040b` uses 4 bits of shift register to debounce the rotary input. Switch events from a rotary control can occur more rapidly than a push button, so a short debounce period is used, lasting about 68 $\mu sec \times 4 = 272\ \mu sec$. If you take too long, the control can advance to the next point before you stabilize a reading.

When the control is rotated rapidly, you can lose switch events because the Pi was too busy or because the events occurred too quickly. When this happens, you can have readings appear out of sequence. If you read `00` immediately followed by `11`, you do not know the direction of the travel. Lost or mangled events also occur when the control becomes faulty.

So, what do you do when an event occurs out of sequence? Do you:

- Ignore the error and wait for the control to reach one of the expected two states?
- Reset your state to the currently read one?

The class `RotarySwitch` defined in files `rotary.hpp` and `rotary.cpp` has taken the first approach. If it encounters an invalid state, it simply returns a "no change" event. Eventually when the control is rotated far enough, it will match one of the two expected next states. The second approach would avoid further lost events but may register events in the wrong direction occasionally.

Experiment

Program `yl040c` was written to allow you to test-drive the rotary control with a counter. Run the program with the control ready.

```
$ ./yl040c
Monitoring rotary control:
Step +1, Count 1
Step +1, Count 2
Step +1, Count 3
Step +1, Count 4
Step +1, Count 5
Step -1, Count 4
Step -1, Count 3
Step -1, Count 2
Step -1, Count 1
Step -1, Count 0
Step -1, Count -1
Step -1, Count -2
Step -1, Count -3
Step +1, Count -2
Step +1, Count -1
Step +1, Count 0
^C
```

The program clearly states the direction of the step as clockwise (+1) or counterclockwise (-1) and reports the affected counter. The counter is taken up to `5` by a clockwise motion and taken back to `-3` in a counterclockwise motion and then clockwise again to return to `0`. When your control is working properly, this will be an effortless exercise.

FM Dial 1

While your rotary control seems to work nicely, it may not be convenient enough in its current form. To illustrate this, let's *simulate* the FM stereo tuner knob with the rotary control. Run the program `fm1` with the control ready. Try tuning from end to end of the FM dial (the program starts you in the middle).

```
$ ./fm1
FM Dial with NO Flywheel Effect.
100.0 MHz
```

```
100.1 MHz
100.2 MHz
100.3 MHz
100.4 MHz
100.5 MHz
100.6 MHz
100.7 MHz
100.8 MHz
100.9 MHz
101.0 MHz
101.1 MHz
101.2 MHz
101.3 MHz
101.2 MHz
101.1 MHz
101.0 MHz
100.9 MHz
100.8 MHz
100.7 MHz
100.6 MHz
100.5 MHz
100.4 MHz
100.3 MHz
100.2 MHz
100.1 MHz
100.0 MHz
 99.9 MHz
 99.8 MHz
 99.7 MHz
 99.6 MHz
 99.5 MHz
 99.4 MHz
 99.3 MHz
 99.2 MHz
 99.1 MHz
 99.0 MHz
 99.1 MHz
^C
```

How convenient was it to go from 100.0MHz down to 99.1MHz? Did you have to twist the knob a lot? To save rotational effort, many electronic controls simulate a "flywheel effect."

The program `fm1` uses the `RotarySwitch` class, which responds in a simple linear fashion to the rotary encoder. If the control clicks once to the right, it responds by incrementing the count by 1. If it clicks counterclockwise, the counter is decremented by 1 instead. You'll look at code for this at the end of this chapter.

Spin a bicycle wheel, and it continues to rotate after you stop and watch. Friction causes the rotation to slow and eventually stop. This is flywheel action, and it is convenient to duplicate this in a control. Some high-end stereo and ham radio gear have heavy knobs in place so that they continue to rotate if you spin them fast enough.

FM Dial 2

Let's repeat the last experiment with the program `fm2`. In the following example, the dial was turned clockwise slowly at first. Then a more rapid twist was given, and the dial shot up to about 102.0MHz. One more rapid counterclockwise twist took the frequency down to 98.8MHz.

```
$ ./fm2
FM Dial with Flywheel Effect.
100.0 MHz
100.1 MHz
100.2 MHz
100.3 MHz
100.4 MHz
100.5 MHz
100.6 MHz
100.8 MHz
101.0 MHz
101.2 MHz
101.4 MHz
101.6 MHz
101.7 MHz
101.8 MHz
101.9 MHz
102.0 MHz
101.9 MHz
101.8 MHz
101.7 MHz
101.6 MHz
101.5 MHz
101.3 MHz
101.0 MHz
100.8 MHz
100.6 MHz
100.2 MHz
 99.8 MHz
 99.5 MHz
 99.3 MHz
 99.1 MHz
```

```
 98.9 MHz
 98.8 MHz
 98.7 MHz
 98.6 MHz
 98.7 MHz
 98.8 MHz
 98.9 MHz
 99.0 MHz
```

With a more vigorous twist of the knob, you can quickly go from one end of the dial to the other. Yet, when rotation speeds are reduced, you have full control with fine adjustments. It is important to retain this later quality.

The following session output demonstrates some vigorous tuning changes:

```
$ ./fm2
FM Dial with Flywheel Effect.
100.0 MHz
100.1 MHz
100.2 MHz
100.3 MHz
100.4 MHz
100.5 MHz
100.8 MHz
101.4 MHz
102.3 MHz
103.4 MHz
104.6 MHz
105.8 MHz
107.2 MHz
107.9 MHz
107.9 MHz
107.9 MHz
107.9 MHz
107.9 MHz
107.9 MHz
107.8 MHz
107.7 MHz
107.6 MHz
107.5 MHz
107.4 MHz
107.2 MHz
106.7 MHz
105.8 MHz
104.3 MHz
102.2 MHz
 99.8 MHz
 97.2 MHz
 94.6 MHz
```

```
91.9 MHz
89.2 MHz
88.1 MHz
88.1 MHz
88.1 MHz
88.1 MHz
```

Class Switch

To keep the programs simple, I've built up the programs using C++ classes. The `GPIO` class is from the `librpi2` library, which you should have installed already (see Chapter 1). This gives you convenient direct access to the GPIO ports.

Listing 8-1 illustrates the `Switch` class definition. This functions well as the contract between the end user and the implementation of the class. Aside from instantiating the `Switch` class in the user program, the only method call that is of interest is the `read()` method, which reads from the GPIO port and debounces the signal.

Listing 8-1. The Switch Class for Handling Rotary Encoder Contacts and Debouncing in switch.hpp

```
009 /*
010  * Class for a single switch
011  */
012 class Switch {
013     GPIO&          gpio;   // GPIO Access
014     int            gport;  // GPIO Port #
015     unsigned       mask;   // Debounce mask
016     unsigned       shiftr; // Debouncing shift register
017     unsigned       state;  // Current debounced state
018
019 public: Switch(GPIO& gpio,int port,unsigned mask=0xF);
020     unsigned read();       // Read debounced
021 };
```

Listing 8-2 shows the constructor for this class. The reference (argument `agpio`) to the caller's instance of the GPIO class is saved in the `Switch` class for later use in line 010 (as `gpio`). The GPIO port chosen for this switch is indicated in `aport` and saved in `gport`. The value of `amask` defaults to the value of `0x0F`, which specifies that 4 bits are to be used for debouncing. You can reduce the number of bits to 3 bits by specifying the value of `0x07`.

Listing 8-2. The Switch Class Constructor Code in switch.cpp

```
006 /*
007  * Single switch class, with debounce:
008  */
009 Switch::Switch(GPIO& agpio,int aport,unsigned amask)
010 : gpio(agpio), gport(aport), mask(amask) {
011
```

```
012     shiftr = 0;
013     state = 0;
014
015     // Configure GPIO port as input
016     gpio.configure(gport,GPIO::Input);
017     assert(!gpio.get_error());
018
019     // Configure GPIO port with pull-up
020     gpio.configure(gport,GPIO::Up);
021     assert(!gpio.get_error());
022 }
```

The shift register for debouncing (`shiftr`) is initialized to 0 in line 012, and the variable `state` is initialized to 0 also. Line 016 configures the specified GPIO port as an input and activates its pull-up resistor in line 020.

The `assert()` macros used will cause an abort if the expressions provided ever evaluate as false. These give the programmer the assurance that nothing unexpected has happened. They can be disabled by defining the macro `NDEBUG` at compile time, if you like. The cost of these is low, so I usually leave them compiled in.

Listing 8-3 illustrates the code for reading the switch and debouncing the contacts. The algorithm used is the same procedure as described in Chapter 5.

Listing 8-3. The Switch::read() Method for Debouncing Contacts in switch.cpp

```
024 /*
025  * Read debounced switch state:
026  */
027 unsigned
028 Switch::read() {
029     unsigned b = gpio.read(gport);
030     unsigned s;
031
032     shiftr = (shiftr << 1) | (b & 1);
033     s = shiftr & mask;
034
035     if ( s != mask && s != 0x00 )
036            return state;   // Bouncing: return state
037     if ( s == state )
038             return state;  // No change
039     state = shiftr & 1;    // Set new state
040     return state;
041 }
```

The `Switch` class provides input and debouncing for only *one* contact. To make a rotary control, you need to debounce *two* switches. For that, you define the `RotarySwitch` class shown in Listing 8-4.

Listing 8-4. The RotarySwitch Class for Debouncing and Reading in rotary.hpp

```
008 /*
009  * Class for a (pair) rotary switch:
010  */
011 class RotarySwitch {
012     Switch         clk;    // CLK pin
013     Switch         dt;     // DT pin
014     unsigned       index;  // Index into states[]
015 protected:
016     unsigned read_pair();  // Read (CLK << 1) | DT
017
018 public: RotarySwitch(GPIO& gpio,int clk,int dt,unsigned mask=0xF);
019     int read();            // Returns +1, 0 or -1
020 };
```

In the `RotarySwitch` class, notice how the two switch contacts are handled by two `Switch` class instances named `clk` and `dt` (lines 012 and 013). Line 014 declares a variable `index`, which will be described shortly. The class method `read_pair()` is declared in the `protected` region so that when using class inheritance, you will have access to it (line 016). The constructor in line 018 is almost the same as `Switch`, except that you have two GPIO numbers provided for `clk` and `dt`.

Listing 8-5 shows the `read_pair()` method code. It reads a bit from switches `clk` and `dt` and returns them as a pair of bits for convenience. Lumping them together in this way provides some programming convenience.

Listing 8-5. RotarySwitch::read_pair() Method in rotary.cpp

```
029 /*
030  * Protected: Read switch pair
031  */
032 unsigned
033 RotarySwitch::read_pair() {
034   return (clk.read() << 1) | dt.read();
035 }
```

Listing 8-6 shows the more interesting code. To read the encoder, a small table is used, declared in lines 008 and 009. These identify the next bit pairs expected when rotation is progressing clockwise.

Listing 8-6. The RotarySwitch::read() Method for Interpreting the Rotary Encoder in rotary.cpp

```
05 /*
006  * State array for CLK & DT switch settings:
007  */
008 static unsigned states[4] =
009     { 0b00, 0b10, 0b11, 0b01 };
010
...
```

```
038 /*
039  * Read rotary switch:
040  *
041  * RETURNS:
042  * +1      Rotated clockwise
043  *  0      No change
044  * -1      Rotated counter-clockwise
045  */
046 int
047 RotarySwitch::read() {
048     unsigned pair = read_pair();
049     unsigned cw = (index + 1) % 4;
050     unsigned ccw = (index + 3) % 4;
051
052     if ( pair != states[index] ) {
053             // State has changed
054             if ( pair == states[cw] ) {
055                     index = cw;
056                     return (pair == 0b11 || pair == 0b00) ? +1 : 0;
057             } else if ( pair == states[ccw] ) {
058                     index = ccw;
059                     return (pair == 0b11 || pair == 0b00) ? -1 : 0;
060             }
061    }
062
063    return 0;        // No change
064 }
```

As the comments document (lines 041 to 044), the `RotarySwitch::read()` method returns a signed value.

- +1 when the control has rotated clockwise
- -1 if in the reverse
- 0 if there was no event to report

The first operation is to read from the pair of switches (line 048). This bit pair is debounced thanks to the work of the `Switch` class that was used to manage them. The values `cw` and `ccw` are the computed next `index` values into array `states` for clockwise and counterclockwise (lines 049 and 050).

Line 052 checks to see whether the value read (variable `pair`) has changed from the last time. If so, an event has happened, and line 054 tests to see whether you went clockwise. If so, the value of `index` is updated in line 055 and a `+1` or `0` is returned. The `+1` is returned, however, only if the control is now at one of the detent positions where `pair` reads as `0b00` or `0b11`. Otherwise, you pretend that nothing happened by returning 0. Lines 057 to 059 perform the same function for counterclockwise. Failing all else, line 063 returns 0 to indicate that nothing interesting happened.

Listing 8-7 shows how the class `Flywheel` inherits from class `RotarySwitch`. This allows you to leverage what you've already built, while adding the flywheel effect on top. The constructor in line 019 for `Flywheel` is the same as for `RotarySwitch`. Like before, the `Flywheel::read()` method returns the same values.

Listing 8-7. Class Flywheel for Rotary Control with Flywheel Effect, in flywheel.hpp

```
011 /*
012  * Class for a (pair) Rotary switch:
013  */
014 class Flywheel : public RotarySwitch {
015     timespec       t0;     // Time of last motion
016     double         ival;   // Last interval time
017     int            lastr;  // Last returned value r
018
019 public: Flywheel(GPIO& gpio,int clk,int dt,unsigned mask=0xF);
020     int read();            // Returns +1, 0 or -1
021 };
```

To implement the `Flywheel` class, you need a fine set of timing functions. Listing 8-8 shows some local functions that are used by the class. The function `get_time()` is used to get time with nanosecond precision. It should be understood, however, that the time returned in the structure `timespec` may not be that accurate but is provided in seconds (member `tv_sec`) and nanoseconds (member `tv_nsec`).

Listing 8-8. The Local Static Functions Used by the Flywheel Implementation, in flywheel.cpp

```
009 static inline void
010 get_time(timespec& tv) {
011   int rc = clock_gettime(CLOCK_MONOTONIC,&tv);
012   assert(!rc);
013 }
014
015 static inline double
016 as_double(const timespec& tv) {
017   return double(tv.tv_sec) + double(tv.tv_nsec) / 1000000000.0;
018 }
019
020 static inline double
021 timediff(const timespec& t0,const timespec& t1) {
022   double d0 = as_double(t0), d1 = as_double(t1);
023
024   return d1 - d0;
025 }
026
```

```
027 Flywheel::Flywheel(GPIO& agpio,int aclk,int adt,unsigned amask)
028 : RotarySwitch(agpio,aclk,adt,amask) {
029
030   get_time(t0);
031   lastr = 0;
032 };
```

The function as_double() in lines 015 to 018 convert the timespec value into a double value, which is more convenient to compute with. The routine timediff() in lines 020 to 025 compute the time difference in seconds.

The constructor in lines 027 to 032 initializes the inherited RotarySwitch class (line 028) and initializes t0 with a start time.

Listing 8-9 is the main star of this chapter, illustrating the extra code used to implement the flywheel effect. Line 035 replaces the RotarySwitch::read() method for this class with new code and tricks.

Listing 8-9. The Meat of the Flywheel Class in flywheel.cpp

```
034 int
035 Flywheel::read() {
036     static const double speed = 15.0; // Lower values change faster
037     timespec t1;                      // Time now
038     double diff, rate;                // Difference, rate
039     int r, m = 1;                     // Return r, m multiple
040
041     r = RotarySwitch::read();        // Get reading (debounced)
042     get_time(t1);                     // Now
043     diff = timediff(t0,t1);          // Diff in seconds
044
045     if ( r != 0 ) {                   // Got a click event
046             lastr = r;                // Save the event type
047             ival = ( ival * 0.75 + diff * 1.25 ) / 2.0 * 1.10;
048             if ( ival < 1.0 && ival > 0.00001 ) {
049                     rate = 1.0 / ival;
050             } else rate = 0.0;
051             t0 = t1;
052             if ( speed > 0.0 )
053                     m = rate / speed;
054     } else if ( diff > ival && ival >= 0.000001 ) {
055             rate = 1.0 / ival;        // Compute a rate
056             if ( rate > 15.0 ) {
057                     if ( speed > 0.0 )
058                             m = rate / speed;
059                     ival *= 1.2;      // Increase interval
060                     t0 = t1;
061                     r = lastr;        // Return last r
062             }
063     }
```

```
064
065     return m > 0 ? r * m : r;
066 }
```

Line 041 uses the original `RotarySwitch::read()` call to read the value of `+1`, `0`, or `-1` depending upon the rotary control. But the `Flywheel` version of `read()` will perform some additional processing.

Line 042 assigns the current time in nanoseconds to `t1`. Using the start time `t0`, the elapsed time from the last event is computed in line 043 (variable `diff`). This difference in time is a floating-point value in seconds but includes the fractional difference in nanoseconds. If the control has been sitting *idle* for a long time, this difference may be very long for the first time (perhaps minutes or hours). However, *successive* events will have time differences of perhaps of 10 milliseconds.

Line 045 tests to see whether you got an event in variable `r`. If there was rotation, `r` will be nonzero. The rotation event is recorded for later use in line 046 (variable `lastr`). Then a weighted average is computed from the prior interval and current difference in line 047 (stored in variable `ival`). An additional 10 percent is added to the value by multiplying by `1.10`. The rationale for this adjustment will be revealed shortly, including why lines 048 to 053 compute a `rate` value and a multiple `m`.

Lines 045 to 053 just described operate when there is a rotational event—either the control has clicked over, clockwise or clockwise. But what happens after quickly rotating the knob and releasing it? Or slowing down?

When there is no rotational event in line 045, control passes to line 054 to test whether the current time difference (`diff`) is greater than the last computed interval `ival`. Recall that the weighted value of `ival` had 10 percent added to it (at the end of line 047). As the rotational rate increases, the time difference between events gets smaller (`diff`). So, as long as this time interval decreases or remains about the same, the generated flywheel events are suppressed. In fact, the rate of rotation would have to drop below 10 percent of the current weighted average before the flywheeling applies. By doing this, expected events that occur on time are handled *normally*. Only when expected next events are *late* do you consider synthesizing events in the flywheeling code.

Line 054 also insists that the last weighted interval time (`ival`) is greater than `0.000001`. This is a rate-limiting action on the flywheeling effect. Line 055 computes a rate, which is the inverse of time interval (`ival`). Only when the rate rises above `15.0` do you synthesize more flywheel events (line 056). The variable `speed` is hard-coded with the value `15.0` (line 036). It is used to compute the multiple `m` from the rate using division (lines 053 and 058). If you change the value of `speed` to zero, there will be no multiplying effect.

The effect of the multiple of `m` is to increase the value returned from a simple `+1` to, say, a `+8`, when the rotational rate is high. This applies to line 065. When the rotational rate slows, the multiple has less effect, and the control responds in smaller increments.

In the flywheeling section in line 059, you compute a next interval time (`ival`). Here the interval time has 20 percent added to it so that it slows and becomes less likely to synthesize another event. This reduces the next `rate` calculation in line 056. As `ival` is increased, eventually the `rate` value drops below `15.0` and then ceases to synthesize any more events. Lines 060 and 061 reset the timer `t0` and set the returned `r` to the `lastr` value saved. Line 065 brings it all together by applying multiple `m` and `r`.

Main Routine

To knit this all together, let's review the main program for `fm2.cpp`, which uses the `Flywheel` class. Because the tricky stuff is located inside the class implementation code, the main programs used in this chapter are essentially the same. Listing 8-10 presents `fm2.cpp`.

Listing 8-10. The Main Program for fm2.cpp

```
#include "flywheel.hpp"

/*
 * Main test program:
 */
int
main(int argc,char **argv) {
    GPIO gpio;              // GPIO access object
    int rc, counter = 1000;

    // Check initialization of GPIO class:
    if ( (rc = gpio.get_error()) != 0 ) {
            fprintf(stderr,"%s: starting gpio (sudo?)\n",strerror(rc));
           exit(1);
    }

    puts("FM Dial with Flywheel Effect.");
    printf("%5.1lf MHz\n",double(counter)/10.0);

    // Instantiate our rotary switch:
    Flywheel rsw(gpio,20,21);

    // Loop reading rotary switch for changes:
    for (;;) {
           rc = rsw.read();
            if ( rc != 0 ) {
                  // Position changed:
                  counter += rc;
                  if ( counter < 881 )
                          counter = 881;
                  else if ( counter > 1079 )
                          counter = 1079;
                  printf("%5.1lf MHz\n",double(counter)/10.0);
             } else {
                  // No position change
                  usleep(1);
             }
    }
```

```
050
051     return 0;
052 }
```

The GPIO class is instantiated as gpio in line 019, and errors are checked in lines 023 to 026. The Flywheel class is instantiated in line 032, with the gpio class passed in as part of the constructor.

The main loop consists of lines 035 to 049. If you read a rotational event, then lines 038 to 044 are executed. Otherwise, a delay of usleep(1) is performed in line 047. On the Raspberry Pi 3 this takes about 68µsec and serves only to pass control to some other process needing the CPU. According to htop(1), this uses about 27 percent of one core. You can increase the sleep time to something above 68µsec to reduce this overhead. But if you overdo the delay, the control will become less responsive.

Summary

In this chapter you discovered rotary encoders and applied the principles of contact debouncing to a reliable read position. From this foundation, you saw code to sense rotary motion and apply flywheeling to make the control more effective. This refashions the humble rotary control as a powerful user input for the Raspberry Pi.

Bibliography

[1] "Keyes KY-040 Arduino Rotary Encoder User Manual." Henrys Bench. 2015. Accessed August 13, 2016. <http://henrysbench.capnfatz.com/henrys-bench/arduino-sensors-and-input/keyes-ky-040-arduino-rotary-encoder-user-manual/>.

CHAPTER 9

More Pi Inputs with 74HC165

When you consider how many GPIO pins have special functions, you might find that there aren't always enough inputs to choose from. This can be an issue when many of the alternate functions are being used (SPI, I2C, UART, PWM, and so on). One way to cost effectively expand the number of Pi inputs is to attach one or more 74HC165 CMOS chips. These are available from `digikey.com` for as little as a dollar. Figure 9-1 illustrates an example of the 16-pin dual inline package (DIP) chip.

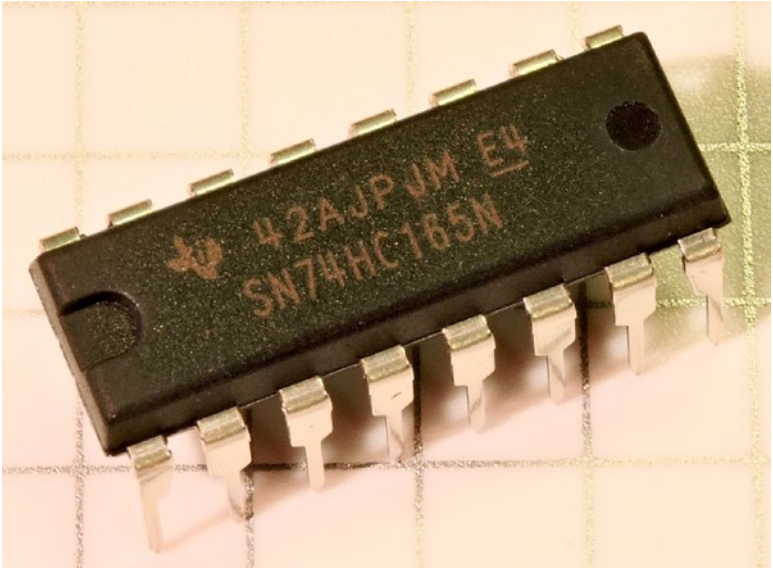

Figure 9-1. *The 74HC165 16-pin DIP*

74HC165 Pinout

The 74HC165 is a CMOS device that will operate nicely with the Pi's 3.3V system. When ordering, make sure that the part number includes the logic family "HC" designation in it. The plain 74165 part, for example, is a 5V part, which will not work at 3.3 volts. Still other

© Warren Gay 2017
W. Gay, *Custom Raspberry Pi Interfaces*, DOI 10.1007/978-1-4842-2406-9_9

logic families will be unsuitable for other reasons. You should also be aware that you want to order the DIP form of the chip (shown in Figure 9-1). Otherwise, your chip will not fit your breadboard and be too small for soldering.

Figure 9-2 illustrates the schematic pinout of the device. Different manufacturers label the pins differently even though the pins serve the same purpose. The labels used by NXP have been added around the outside of the chip in the figure, which correspond with the ones I chose to use in this chapter.

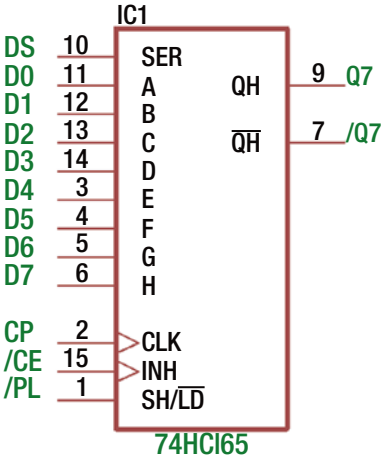

Figure 9-2. *74HC165 pinout*

Table 9-1 summarizes the functions of the pins for this device (additional input pins D1 through D6 have been omitted).

Table 9-1. *Summary of Pin Functions*

Pin	NXP Label	Other Designations	Description
15	/CE	INH	Chip enable (inhibit)
1	/PL	SH/LD	Parallel load (shift/load)
2	CP	CLK	Clock pulse
9	Q7	QH	High bit (noninverted)
7	/Q7	QH	High bit (inverted)
10	DS	SER	Serial input data
11	D0	A	Least significant parallel input bit
6	D7	H	Most significant parallel input bit

For your benefit, realize that /CE indicates the same thing as $\overline{CE}$. The leading slash or the overhead bar indicates that the signal is "active low." The $\overline{CE}$ (chip enable) input is active when it is low. The same pin is labeled by other manufacturers (like Fairchild Semiconductor, for example) as the INH (inhibit). Since the INH is active high (no bar overhead), the end result is the same; that is, the chip is *not* enabled when the INH signal is in the high state. These are just two different sides of the same logic function.

Some chips are sufficiently complex that they need a function table to clarify the interaction of the various inputs and outputs. The following section includes a function table adapted from the Fairchild datasheet.

Function Table

Table 9-2 helps you view the behavior of functions at a glance. When looking at this for the first time, you should study it long enough to understand it. These charts are invaluable and will save you time in the future.

Table 9-2. *74HC165 Function Table*

Inputs					Internal Outputs		Output
/PL	/CE	CP	DS	Parallel			Q_H
				A…H	Q_A	Q_B	
L	X	X	X	a…h	a	b	h
H	L	L	X	X	Q_{A0}	Q_{B0}	Q_{H0}
H	L	↑	H	X	H	Q_{AN}	Q_{GN}
H	L	↑	L	X	L	Q_{AN}	Q_{GN}
H	H	X	X	X	Q_{A0}	Q_{B0}	Q_{H0}

The first obvious characteristic of this table is that symbols like L, H, and X are used. Table 9-3 summarizes the meaning of these symbols. These are fairly self-explanatory, but I'll describe them in detail.

Table 9-3. *Legend for Table 9-2*

Symbol	Meaning
H	High level, steady state
L	Low level, steady state
X	Irrelevant (don't care)
↑	Signal transition from low to high level
Q_{A0}, Q_{B0}, Q_{H0}	The level of Q_A, Q_B or Q_H, respectively, before the indicated steady-state input conditions were established
Q_{AN}, Q_{GN}	The level of Q_A or Q_G before the most recent ↑ transition of the clock, indicating a 1-bit shift

The first row of Table 9-2 shows the state of input pin /PL as being "L" (low). The remaining inputs /CE, CP, and DS are shown as an X. This means that while /PL is active (low), these other inputs have no effect (are irrelevant). Under the "Parallel" column, you see that inputs "a" through "h" modify the internal state of internal latches Q_A and Q_B (and implied Q_C to Q_H). Here I adopted the Fairchild convention for the input labels for association with the outputs (these are D0 through D7 inputs using NXP labels). Finally, note that Q_H is actually an output pin. This is the highest-order bit in the shift register that is made available externally to the chip. So, whenever /PL is applied, the data presented at H (D0) will appear at pin Q_H. None of the other inputs participates in this operation (including the chip enable).

For the remaining rows of the function table, the state of /PL is high. Except for the last row, you see rows 2, 3, and 4 have /CE as enabled (L). When the clock (CP) is low (L), you see that the state of the DS shift input pin is marked as an X and thus has no relevance in this state. The output pin Q_H shows Q_{H0} in the table, which simply means that it retains the last latched value.

The third and fourth rows of the function table show an up arrow for the clock (CP). This means that when the clock signal goes from a low state (L) to high (H), the transition causes the shift register to shift. The state of the input pin DS is copied to the first shift register latch Q_A, while the other values from Q_A to Q_G get copied to the next bit position. The prior value of Q_H is lost in this operation (shifted out).

The last row of the function table is also important. When the /PL signal and the /CE are inactive (not L), you see that the remaining inputs are marked with X. In this state, the CP, DS, and inputs D0 (A) through D7 (H) have no effect on the state of the chip.

Breadboard Experiment

Figure 9-3 illustrates the circuit for wiring up to your Raspberry Pi. You can safely use the Pi's 3.3V power since the CMOS chip uses very little current (much less than 1mA). Don't forget to connect the Pi's ground to pin 8 of the chip to complete the power circuit.

Note that the unused input SER is connected to ground. You could alternatively connect this to the supply, but it must be connected. CMOS inputs should always be established, whether low or high, and never left "floating."

To use the supplied software (in the `hc165` subdirectory) without modification, use the GPIO connections shown. If you're willing to modify the C++ code, then you can use different GPIO pins. The connections to inputs D0 through D7 in the schematic are completely arbitrary. The schematic is wired to cause the value of 0x82 to be received by the test program. Bits D1 (B) and D7 (H) are connected to the +3.3V supply so that those bits will be read as 1 bits, while all the remaining pins will read as 0. Attach buttons or switches or change the wiring if you like after you have proven that it works.

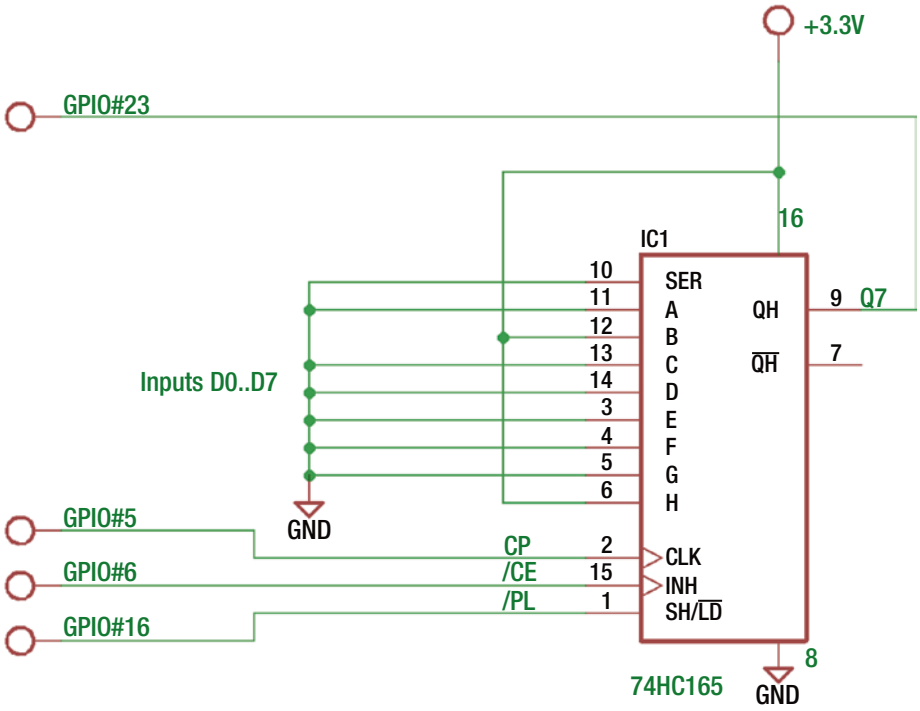

Figure 9-3. *Breadboard schematic*

Figure 9-4 illustrates my wired breadboard using Dupont wires. The wiring looks like a rat's nest only because all data inputs of a CMOS device need to be wired to either ground or the supply. Most of the wires are connecting the parallel inputs to ground. One can never have too many Dupont wires!

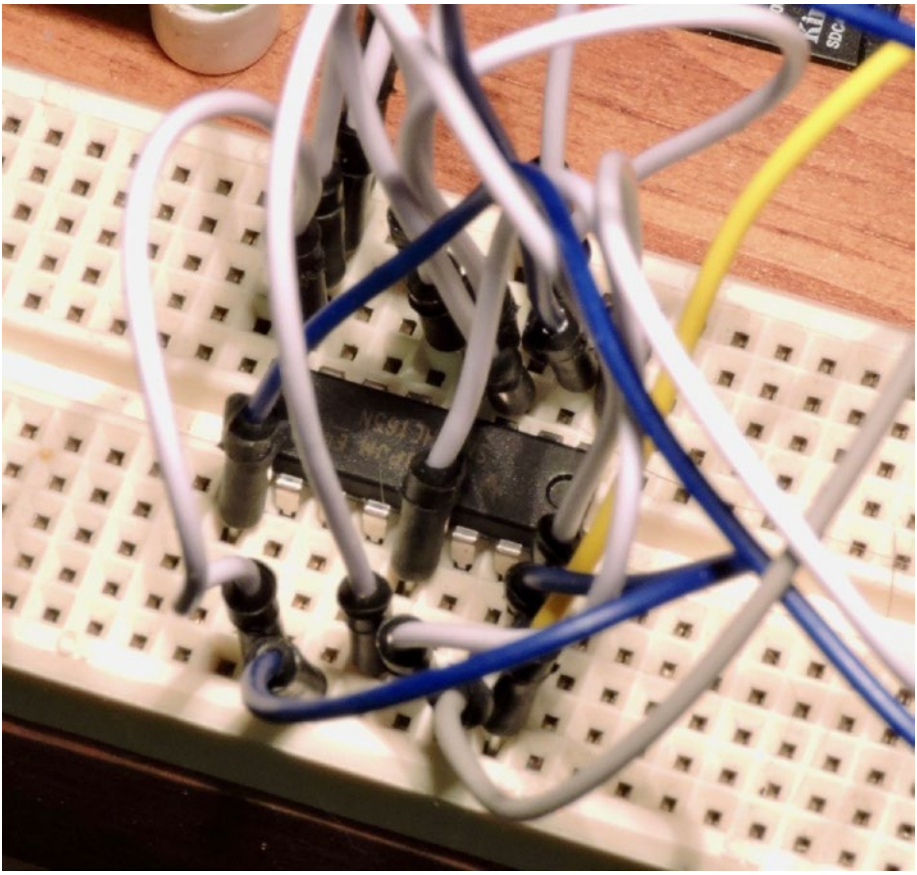

Figure 9-4. *Breadboard wiring*

Program

The program found in the `hc165` subdirectory is a little illustration program that can be used to test your breadboard setup, partially illustrated in Listing 9-1. Lines 010 through 013 can be changed to match the GPIO pins that you have chosen to use. Line 015 sets the `usleep(3)` sleep time to 1µsec. In practice, it will take considerably longer, requiring about 65µsec on the Pi 3 (70µsec on Pi 2). This is because of the overhead of entering the kernel and returning.

You can take advantage of the librpi2 library to save time and effort. The main program for `hc165.cpp` configures the GPIOs as inputs and outputs, as required (not shown here).

Listing 9-1. Partial Rendering of hc165.cpp

```
010 static int hc165_pl = 16;   // Outputs
011 static int hc165_ce = 6;
012 static int hc165_cp = 5;
013 static int hc165_q7 = 23;   // Input
014
015 static unsigned usecs = 1;
016 static GPIO gpio;
017
018 static void
019 nap() {
020     usleep(usecs);
021 }
022
023 static void
024 strobe(int pin) {
025     nap();
026     gpio.write(hc165_pl,0);         // Load parallel
027     nap();
028     gpio.write(hc165_pl,1);         // Loaded
029     nap();
030 }
031
032 static unsigned
033 hc165_read() {
034     unsigned byte = 0;
035
036     strobe(hc165_pl);               // Load parallel
037     gpio.write(hc165_cp,0);         // CP = low
038     gpio.write(hc165_ce,0);         // /CE = low
039     nap();
040
041     for ( int x=0; x<8; ++x ) {
042             byte <<= 1;
043             byte |= !!gpio.read(hc165_q7);  // Read Q7
044             gpio.write(hc165_cp,1);         // Shift
045             nap();
046             gpio.write(hc165_cp,0);         // Complete clock pulse
047             nap();
048     }
049     gpio.write(hc165_ce,1);
050     return byte;
051 }
```

Lines 023 to 030 show the `strobe()` function. All it does is set the /PL line low and then high again with some nap times in between. If the signals change too quickly, the CMOS chip will be unable to keep up.

The `hc165_read()` function returns an unsigned value representing the shifted-out inputs from the 74HC165. The first operation is to strobe the input data into the chip using the `strobe()` function. Then the clock signal is set to low in preparation of the shift to come. The /CE is also set low to enable the chip's operation.

The loop in lines 041 through 048 successively reads the bit from Q7 and shifts it into the value `byte`. Once this has been done eight times, the /CE is set high again, and the value is returned.

Use the `make` command to compile the code.

```
pi@rpi3:~/src/hc165 $ make
g++ hc165.o -o ./hc165 -L/home/pi/rpi2016/hc165/..//lib -lrpi2 -lrt -lm
sudo chown root ./hc165
sudo chmod u+s ./hc165
```

Since direct use of the GPIO ports requires root access, the `make` procedure gives the executable `hc165` root `setuid` access. This saves you from having to use `sudo` in order to run the program.

After your breadboard is ready, run the program, as follows:

```
pi@rpi3:~/src/hc165 $ ./hc165
byte = 82
pi@rpi3:~/src/hc165 $
```

If you wired your breadboard as shown in Figure 9-3, you should see the program report "byte = 82" (hex). Otherwise, other values will reflect the state of your shift register inputs.

Logic Analyzer View

Figure 9-5 shows a logic analyzer capture of an input operation. From this you can visualize the /PL going low to clock the data into the shift register. You see /CE go low to activate the chip, and on the rising edge of the clock (CP), you can see data changes in Q7. The highest bit b7 is shifted out first, ending with least significant bit b0. When all is complete, the /CE goes high again.

The actual bit read operation in line 043 of the program occurs just before the clock (CP) goes high. This leaves plenty of time for the shift register data output to stabilize.

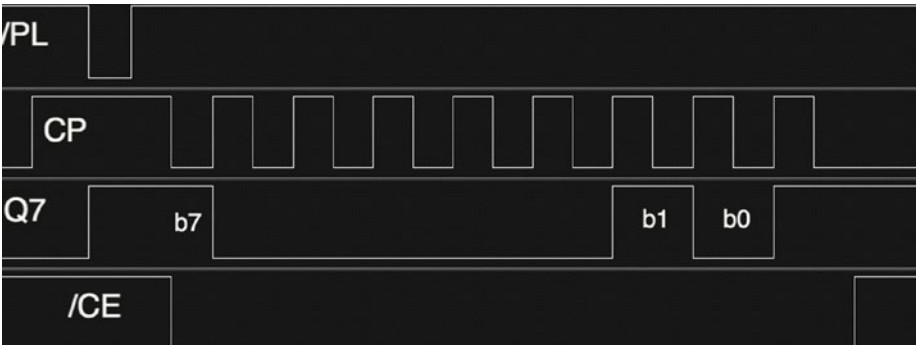

Figure 9-5. *Logic analyzer capture*

You have used the call `usleep(3)` with an argument of 1 to cause the program to give up the CPU and come back as early as the kernel can (the call indicates to sleep for 1µsec). However, there is overhead just going into the kernel so that by the time it has done so, it is time to return. The time of each `usleep(3)` call amounted to about 65µsec.

Profit and Loss

Let's take a moment to take stock of what you gained and lost using the 74HC165 chip. To gain eight more input pins, you had to allocate three output and one input GPIOs. This is an improvement of 50 percent. But what did you lose? More time is involved.

Referring to Figure 9-5, you can see that there is about 10 time periods involved in reading in the 8 inputs. The program used a `usleep(3)` call for 1µsec, but that turned into about 65µsec because of kernel overhead. If you assume those numbers, you then require the following:

$$10 \times 65 = 650 usec$$

This means each read of eight bits requires about 0.7ms. By computing the inverse of this, you can determine the maximum number of reads per second, as follows:

$$\frac{1000ms}{0.65ms} = \frac{1{,}538reads}{second}$$

For many applications like reading a switch or a push button, this is quite acceptable. But other applications may need a higher sampling rate.

Even More Inputs

What if you need more than eight additional inputs? This is easily accomplished by adding another 74HC165, without requiring additional GPIOs. The schematic in Figure 9-6 shows how this can be wired to accomplish this.

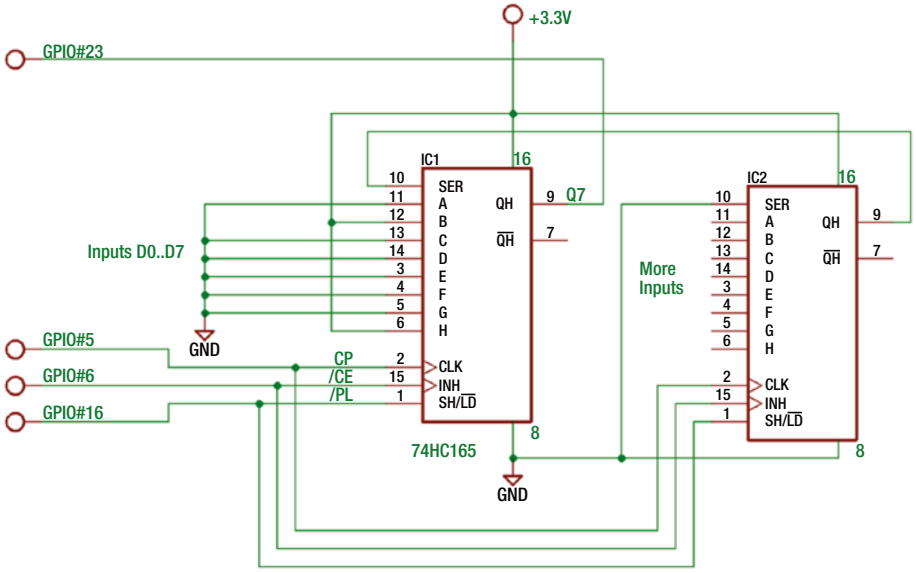

Figure 9-6. *Expanding the use of 74HC165 to 16 inputs*

You can see that the /CP, /CE, and /PL connections are simply parallel-connected to the same pins of IC2. The only thing new is that the Q7 (QH) output from IC2 now connects to IC1's SER input (pin 10). This allows the data shifted out of IC2 to shift into IC1, as the bits are being read by the Pi. Performing 16 shifts instead of 8, the Pi will be able to read all 16. How does this impact your read times?

The bit periods involved will be approximately 10+8, or 18 periods.

$$18 \times 65usec = 1.170ms$$

For about 80 percent more time, you get eight more bits of data. This arrangement will allow you to do about 855 reads per second.

The code change required is minor; you simply change the loop in line 041 to loop for 16 times instead of 8. See Listing 9-2 for the change.

Listing 9-2. Change Required to Read 16 Bits

```
041     for ( int x=0; x<16; ++x ) {
042             byte <<= 1;
043             byte |= !!gpio.read(hc165_q7);  // Read Q7
044             gpio.write(hc165_cp,1);         // Shift
045             nap();
046             gpio.write(hc165_cp,0);         // Complete clock pulse
047             nap();
048     }
```

Other Bit Counts

There is no law that says that you must read a multiple of eight bits of input data. For example, using one 74HC165 chip, your application might require only six inputs. In this case, line 041 of Listing 9-2 could be changed to loop only six times instead of eight. Let's calculate the impact of that change.

$$2+6=8\ time\,periods$$

Assuming 65µsec for each time period, this results in a read time as follows:

$$8\times 65=0.520ms$$

This translates to the following:

$$\frac{1000ms}{0.52ms}=\frac{1{,}923reads}{second}$$

By customizing your I/O to match your data needs, you can improve the I/O bandwidth of your interface.

Combining GPIOs

Sometimes you can economize your GPIO usage when combining with other peripherals. Chapter 10 looks at adding outputs with the use of the 74HC595 chip. It too will require a /CE (chip enable), a clock pulse (CP), and a data latching signal (/PL). If you share the /CE, CP, and /PL GPIO signals with both your input and output devices, you would need only one more GPIO to send data out. The input and output bits can be handled at the same time (on separate GPIOs) so that your I/O transfer rate is not affected. This topic will be explored further in Chapter 10.

Chip Enable

If the GPIOs are dedicated only to your 74HC165 chip, you can make a further optimization: simply ground the /CE input. Referring to Table 9-2, notice that the parallel load (/PL) signal operates regardless of the state of the /CE input. If you ground the /CE signal going into the 74HC165, the chip is *always* enabled. This allows you to reduce the required GPIO signals to just /PL, CP, and the Q7 lines. If, on the other hand, you share these GPIOs with multiple devices, then the /CE signal remains necessary to select the chip to activate.

As an experiment, disconnect GPIO #6 from pin 15 (/CE). Then connect pin 15 (/CE) to ground. When you run the `./hc165` program (unmodified), it should still function as before. This will prove that the /PL input is independent of the /CE input.

As a further experiment, comment out all of the /CE (GPIO #6) references in the program and try again. If the modifications are correct, you should be able to repeat the read of the device successfully with only three GPIO signals.

CD4012B

The 74HC165 is not the only chip available. There is also the CMOS CD4012B device, which can run from 3.3V and requires very little current. The pinout differs, but the operation is similar.

Summary

This chapter has demonstrated that for as little as a dollar you can exchange four GPIOs for eight inputs. If you need even more inputs, you can obtain eight more for a dollar more. You learned about datasheet "function tables" and became familiar with the signals that drive the 74HC165 shift register. The next chapter will introduce the 74HC595 chip for adding GPIO outputs to your Pi.

CHAPTER 10

More Pi Outputs with 74HC595

Like input GPIOs in Chapter 9, the application designer may require more GPIO outputs. Because this 74HC595 part is a member of the "HC" logic family, it is able to operate with a V_{CC} ranging from 2.0 to 6.0 volts. This is perfect for operating under the 3.3V Raspberry Pi and costs less than a dollar at digikey.com. Figure 10-1 illustrates a chip that I've used.

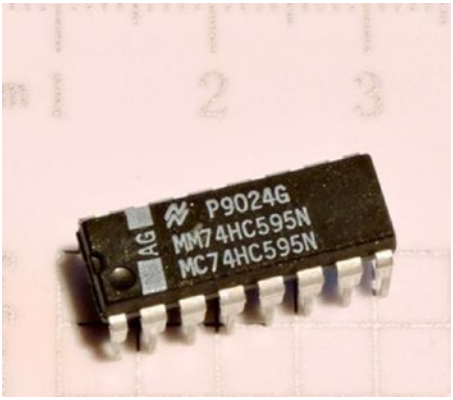

Figure 10-1. *74HC595 serial in, parallel out shift register*

Note that the HCT family is designed for TTL input levels. I focus on the HC devices in this chapter because the device must interface to the CMOS-based Raspberry Pi.

74HC165 Pinout

Figure 10-2 provides a schematic pinout for the 74HC595 chip. Once again, different manufacturers use different labels for the various pins.

© Warren Gay 2017
W. Gay, *Custom Raspberry Pi Interfaces*, DOI 10.1007/978-1-4842-2406-9_10

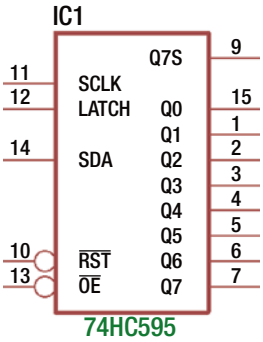

Figure 10-2. *74HC595 pinout*

Table 10-1 documents the pins, the schematic labels, the NXP labels (not shown in Figure 10-2), and the pin functions for this chip.

Table 10-1. *74HC595 Pins and Functions*

Pin	Schematic Label	NXP Label	Function
11	SCLK	SHCP	Shift clock
14	SDA	DS	Shift (in) data
12	LATCH	STCP	Latch (clock pulse)
10	RST	MR	(Master) reset
13	OE	OE	Output enable
15	Q0	Q0	Output Q0 (least significant bit)
7	Q7	Q7	Output Q7 (most significant bit)
9	Q7S	Q7S	Serial data output (for chaining)

Function Table

Table 10-2 is an adaptation of the NXP-provided table, except that it uses the labels provided by the EAGLE software in Figure 10-2. In addition to the legend provided in Chapter 9's Figure 9-3, there are two new symbols: NC, which means "No Change," and Z, which means "high impedance" (essentially disconnected from the circuit).

Table 10-2. *Function Table for 74HC595*

Controls				Input	Output	
SCLK	STCP	OE	MR	DS	Q7S	Qn
X	X	L	L	L	L	NC
X	↑	L	L	L	L	L
X	X	H	L	L	L	Z
↑	X	L	H	D	Q6S	NC
X	↑	L	X	X	NC	QnS
↑	↑	L	X	X	Q6S	Qns

When the MR (reset) signal goes low (in the first line of the table), you can see that there are no changes on the outputs (Qn = NC), but it does clear the internal shift register. This is true whether the OE (output enable) signal is active or not (lines 2 and 3).

Line 3 shows you, however, that when the OE signal is *inactive* (high), the shift register outputs Q0 through Q7 (Qn) go into a high impedance (Z) state. In other words, these outputs become disconnected from the circuit that they are wired to.

Line 4 applies when MR is inactive (H), OE is active (L), and the signal SCLK (↑) goes from low to high. Here I used "D" to indicate a data bit is copied into the shift register bit Q0 (as ON Semiconductor does). The remaining shift registers' cells are shifted to the next position. This results in the Q7 bit being replaced with the prior state of Q6. Note that the parallel latched outputs Q0 through Q7 remain unchanged (NC).

While OE remains active (L) in line 5, the rising signal STCP (↑) causes the shift register bits to appear externally at Q0 through Q7. This latching operation has no effect on the internal shift register contents, which is why output Q7S shows no change (NC).

The last line of the function table indicates that you can shift a data bit into the internal shift register and simultaneously latch that result so that it is externally visible. This can save a little bit of time in the output cycle.

Breadboard Experiment

Figure 10-3 illustrates the breadboard circuit for the 74HC595 chip. In this arrangement, you hardwire the OE pin to ground. This means that the outputs Q0 through Q7 will always be enabled. The clock pulses are wired to GPIO#5 as in the previous chapter. The LATCH signal is active when there is a low to high transition.

You can take advantage of the /PL line as used in Chapter 9. Sharing that GPIO is possible because the input 74HC165 loads data while this line is low. The output 74HC595 will latch its outputs when the signal goes from low to high. You can, of course, use a different GPIO for this purpose if your application demands it. I'll discuss GPIO sharing in more detail later.

Finally, you need GPIO #24 for passing data into the 74HC595 shift register from the Pi. You can use different GPIO assignments, but this is what the supplied source code will use for this experiment.

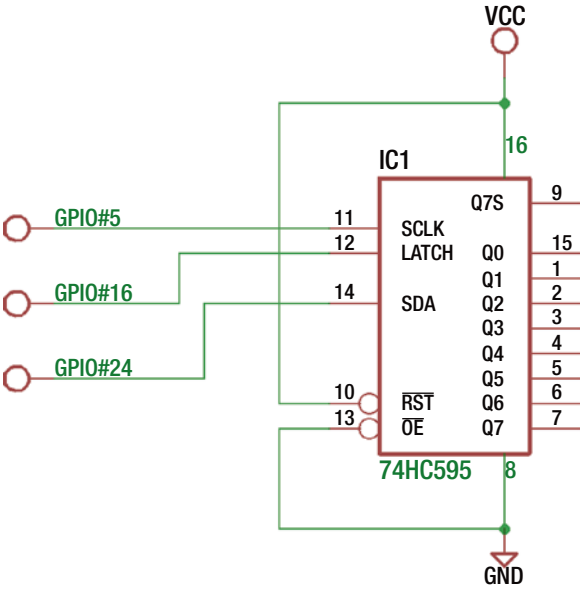

Figure 10-3. *Breadboard circuit for 74HC595 output*

Lastly, this experiment ties the RST input to the +3.3V supply. This reset input is active low, so you tie it high so that it doesn't become active for this experiment.

Unlike the 74HC165 input chip, you have no remaining inputs to wire to ground or the supply. CMOS outputs can be left without connections.

In the subdirectory `hc595`, you will find the `Makefile` and program `hc595.cpp`. Change to that directory and type `make`.

```
$ cd ./hc595
$ make
g++ -c -std=c++11 -Wall -Wno-deprecated -Wno-narrowing -I. -I/usr/local/
include/librpi2 -g -O0 hc595.cpp -o hc595.o
g++ hc595.o -o ./hc595 -L/home/pi/rpi2016/hc595/..//lib -lrpi2 -lrt -lm
sudo chown root ./hc595
sudo chmod u+s ./hc595
```

Once again, note that the program `hc595` has been given `setuid` permission so that you won't need to run it with `sudo`. Root permission is required to access the GPIO registers directly. Listing 10-1 shows the program area of interest.

Listing 10-1. Program hc595.cpp

```
008 #include "gpio.hpp"
009
010 static int hc595_latch      = 16;  // Outputs
011 static int hc595_sclk = 5;
```

```
012 static int hc595_sda = 24;
013
014 static unsigned usecs = 1;
015 static GPIO gpio;
016
017 static void
018 nap() {
019   usleep(usecs);
020 }
021
022 static void
023 hc595_write(unsigned data) {
024   unsigned b, temp = data;
025
026   gpio.write(hc595_sclk,0);             // SCLK = low
027   gpio.write(hc595_latch,0);            // LATCH = low
028   nap();
029
030   for ( int x=0; x<8; ++x ) {
031         b = !!(temp & 0x80);            // b = d7 (msb first)
032          temp <<= 1;
033          gpio.write(hc595_sda,b);       // SDA = data bit
034          nap();                         // Wait
035          gpio.write(hc595_sclk,1);      // SCLK low -> high
036          nap();
037          gpio.write(hc595_sclk,0);      // Complete clock pulse
038          nap();
039    }
040    gpio.write(hc595_latch,1);           // Latch data to outputs
041    nap();
042    printf("Wrote %02X to 74HC595\n",data);
043 }
```

The main program (not shown) configures the GPIOs as outputs. If no command-line arguments are given, a call is made to `hc595_write(0xD5)` to transmit that bit pattern to the output chip. If other command-line parameters are provided, those values are written instead.

The `hc595_write()` routine starts by making sure that the SCK and LATCH signals are set to the low level (lines 026 and 027), followed by a delay. Line 024 has copied the data value into variable `temp` for use in the loop.

In the `for` loop of lines 030 to 039, this `temp` value is sampled at bit 7 and stored in variable `b`. The double `!!` operator just converts a nonzero result (in `b`) into the value 1 and otherwise to the value 0. Then line 032 shifts the variable `temp` one position to the left for the next loop iteration.

Line 033 writes the data bit value in `b` out to GPIO `hc595_sda` (GPIO #24). This establishes the high or low value to be shifted into the output register. Line 034 delays so that the signal transitions don't happen too fast for the chip. Line 035 raises the clock signal (SCLK) from low to high. This will clock the data bit into the output chip. Line 037 returns the clock signal low again after a delay.

After all eight bits of data are shifted into the output chip, the LATCH signal is activated in line 040. Recall that it is the transition from low to high that makes the shift register chip's output available on the chip output pins (Q0 through Q7). There is one more delay following this operation so that if the output routine gets called back to back, sufficient time will exist between signal changes.

Experiment Run

The schematic in Figure 10-3 did not have anything shown connected to the outputs Q0 through Q7. The experiment can be run without any connections if you have a DMM to measure the output levels with. Simply take a meter reading, and if the value is a high level, note a 1 bit.

If you want a visual indication, you can attach LEDs with a dropping resistor in series. Depending upon the manufacturer, the absolute maximum output of each pin is 25mA (Fairchild). The chip is further limited by its maximum power dissipation, when every output is driving full current. The Fairchild datasheet indicates 600mW as the absolute maximum power dissipation. If you limit each output to a maximum of 22mA, you just fit within that parameter.

$$22mA\,x\,8\,pins\,x\,3.3volts = 580mW$$

Most LEDs will light with considerably less than this. Google *LED voltage drop* and you will find that many red LEDs will require about 1.8 volts. Taking this into account, you can compute the dropping resistor required.

$$R_{LED} = \frac{V_{CC} - V_{LED}}{I_{LED}}$$

Assuming a modest current of 10mA for the LED and substituting allows you to compute the series dropping resistor.

$$R_{LED} = \frac{3.3V - 1.8V}{10mA} = 150\Omega$$

Figure 10-4 illustrates, in schematic form, the circuit necessary to attach to the output pins for Q0 through Q7. If you don't mind moving the Dupont wires around, you can get by with just one LED.

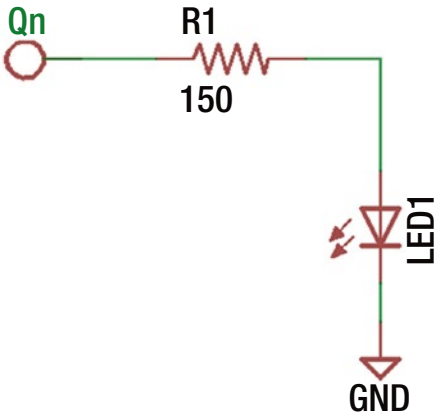

Figure 10-4. *LED hookup with series dropping resistor*

Running the program with no command-line arguments will output the hexadecimal value of D5.

```
$ ./hc595
Wrote D5 to 74HC595
```

The `hc595` program reports the value written to the 74HC595 chip. With this hex value (11010101 in binary), you should observe the following parallel data output:

- Q7 is high.
- Q6 is high.
- Q5 is low.
- Q4 is high.
- Q3 is low.
- Q2 is high.
- Q1 is low.
- Q0 is high.

Obviously, if you measure something else, then something is incorrect.

Input and Output

Let's now look at combining the output 74HC595 with the input 74HC165 chips, using shared GPIO lines. Figure 10-5 illustrates the schematic circuit. IC1 is the 74HC595 that I've been discussing in this chapter, while IC2 is the added 74HC165 chip that was examined in Chapter 9.

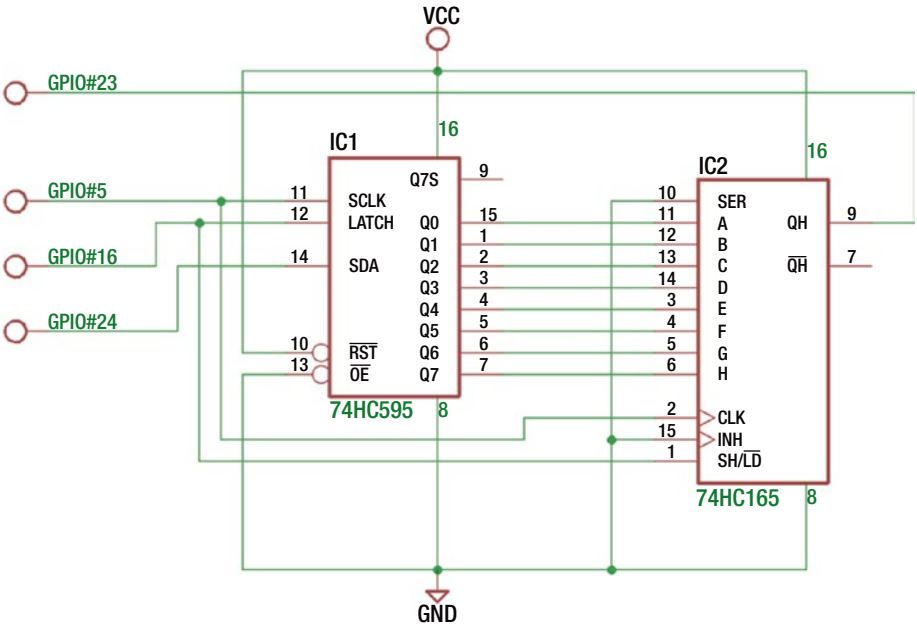

Figure 10-5. *Combined output with input breadboard circuit*

In this experiment, you've connected the outputs Q0 through Q7 to the inputs A through H of the 74HC165 chip. By doing this, your software should be able to output a value and then read it back as verification. The chip enable (INH) of IC2 has been wired to ground, since there is no need to select a chip. As before, the unused input SER of IC2 has also been tied to ground to eliminate any floating inputs.

The circuit shares GPIO #5 and GPIO #16 with both chips. GPIO #5 supplies the clock pulse, while GPIO #16 supplies the LD signal for IC2 and the LATCH symbol for IC1. The unshared GPIOs include #23 used for reading data, while GPIO #24 is used to supply data to IC1.

Listing 10-2 shows the interesting aspects of the code. Lines 010 to 013 allocate the GPIOs being used. Pin `gpio_qh` is the serial data in from the 74HC165, while `gpio_sda` is the serial data going out to the 74HC595. The `gpio_latch` and `gpio_sclk` lines are shared between the two chips.

Listing 10-2. Program hc165_595.cpp

```
010 static int gpio_qh          = 23;  // Input
011 static int gpio_latch       = 16;  // Outputs
012 static int gpio_sclk        = 5;
013 static int gpio_sda         = 24;
014
015 static unsigned usecs = 1;
016 static GPIO gpio;
017
018 static void
019 nap() {
020   usleep(usecs);
021 }
022
023 static unsigned
024 io(unsigned data) {
025   unsigned b, ib, temp = data;
026   unsigned idata = 0u;
027
028   gpio.write(gpio_sclk,0);             // SCLK = low
029   nap();
030
031   for ( int x=0; x<8; ++x ) {
032         ib = gpio.read(gpio_qh);       // Read 74HC165 QH output
033         idata = (idata << 1) | ib;     // Collect input bits
034
035         b = !!(temp & 0x80);           // Set b to output data bit
036         temp <<= 1;
037
038         gpio.write(gpio_sda,b);        // write data bit
039         nap();                         // Wait
040          gpio.write(gpio_sclk,1);      // SCLK low -> high (shift data)
041         nap();
042         gpio.write(gpio_sclk,0);       // Complete clock pulse
043         nap();
044   }
045   gpio.write(gpio_latch,0);            // Set new inputs
046   nap();
047   gpio.write(gpio_latch,1);            // Latch data to outputs
048   nap();
049   printf("Read %02X, Wrote %02X\n",idata,data);
050
051   return idata;
052 }
```

The function `io()` has been written to send and receive data. The data sent out is manipulated in temporary variable `temp` in lines 035 and 036. The data being read is accumulated in temporary variable `idata` in lines 032 and 033. At the conclusion, the values read and written are reported in line 049.

The main program invokes `io()` with values ranging from 0x00 through 0xFF and then the two additional values 0x00 and 0x01.

```
$ ./hc165_595
Read 01, Wrote 00
Read 02, Wrote 01
Read 00, Wrote 02
Read 01, Wrote 03
Read 02, Wrote 04
Read 03, Wrote 05
Read 04, Wrote 06
...snip...
Read FB, Wrote FD
Read FC, Wrote FE
Read FD, Wrote FF
Read FE, Wrote 00
Read FF, Wrote 01
```

The first value read (and reported) should be disregarded. It is reporting the current value in the shift register, which may be garbage the first time around. The second value also should be disregarded since this will represent the uninitialized value of the output register, as you will see.

Looking at the general pattern, you can see that the input values are two cycles behind what was last written. Here's an example:

```
Read FC, Wrote FE
```

Here you can observe that the read value was hexadecimal FC, while the currently written value is FE. The reason for this is subtle. Let's break down the operations involved.

1. The LD signal of the 74HC165 goes low to capture its inputs, which is wired to the 74HC595 outputs Q0 through Q7.
2. When the LD signal returns high, the 74HC165 shift register holds 0xFC to send back to the Pi (this was the current 74HC595 output value).
3. However, in step 2, a low to high transition clocks in the new outputs for the 74HC595 outputs (which the 74HC165 chip has not yet seen).
4. The clock line shifts in eight more bits to the output IC1 chip, while the Pi reads the captured data in IC2.

Consequently, the Pi will read 0xFC, while just having written out 0xFE. The output value 0xFE will not be seen on the IC1 outputs until the next cycle's step 2. These details can get confusing. Figure 10-6 illustrates my own breadboard experiment.

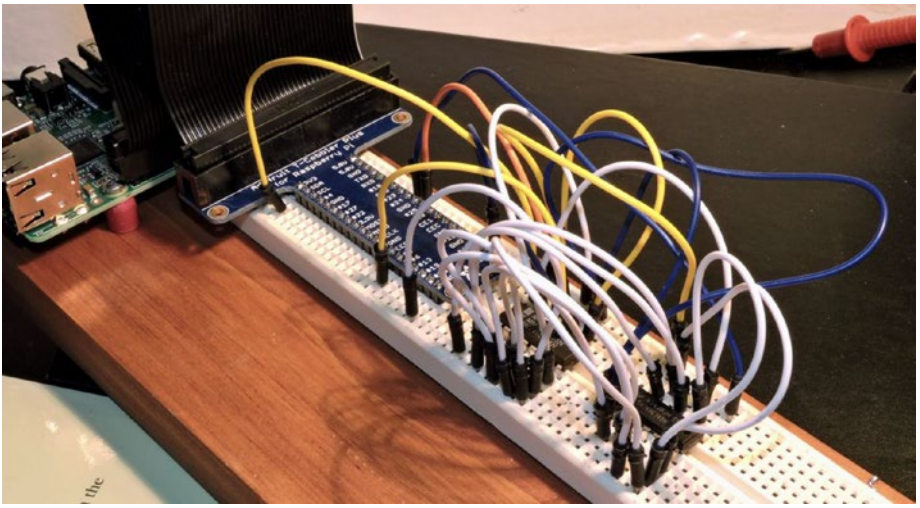

Figure 10-6. *74HC595 and 74HC165 breadboard experiment*

Additional Outputs

Additional outputs can be had by adding another 74HC595 chip. Figure 10-7 shows the schematic for doing so. The SDA and LATCH signals are attached in parallel, and IC2's data input comes from IC1's output Q7S (pin 9).

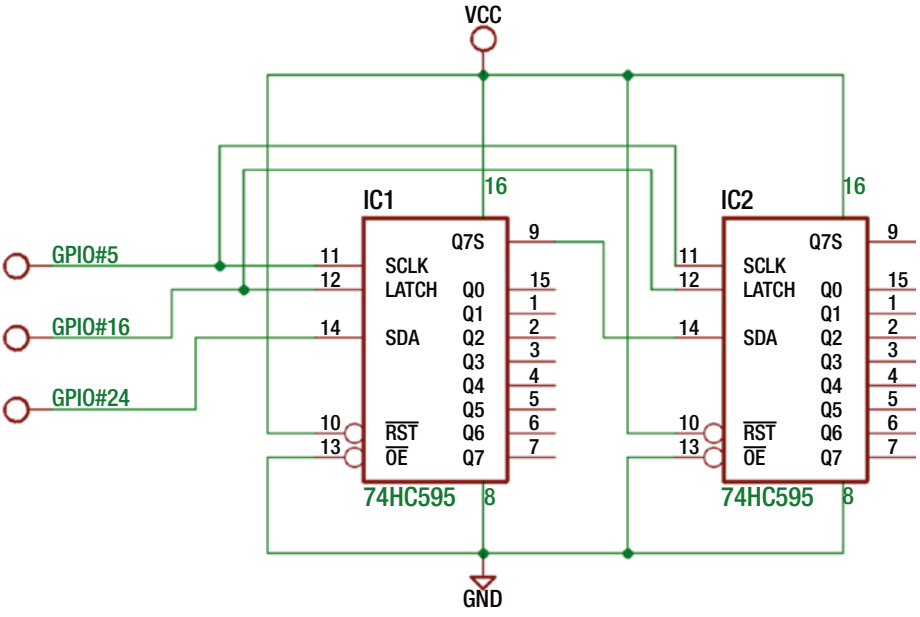

Figure 10-7. *Two 74HC595 chips used for 16-bit output*

Profit and Loss

The breadboard experiment in Figure 10-5 was a demonstration of eight outputs and eight inputs. These required four GPIO pins to manage it. This is a 400 percent improvement in GPIO ports. The cost, however, is the extra time required to shift in/out the data. The Figure 10-5 experiment illustrated that the total time was reduced by performing input and output simultaneously.

Summary

This chapter demonstrated how you can easily add GPIO outputs to your Pi for the cost of about $1. Even more outputs can be had with additional chips. Finally, you exercised both input and output chips simultaneously to gain 16 I/O pins using only four allocated GPIO ports.

Beyond the Pi's GPIO ports used, the 74HC165 and 74HC595 chips required no software configuration, making them simple to use. The next chapter will introduce another way to add I/O pins to your Pi. But this will require software configuration.

CHAPTER 11

MCP23017 I/O Port Extender

While it is possible to add input and output ports with various CMOS chips, sometimes more flexibility is required. The pin functions of chips like the 74HC165 and 74HC595 are fixed as input or output. Sometimes it is desirable to be able to configure them as required like the Raspberry Pi's own GPIO ports. This chapter will examine the Microchip MCP23017 peripheral chip, which can offer any combination of 16 GPIO ports (in DIP form), each of which can be individually configured as input or output. Figure 11-1 illustrates the chip sitting on a breadboard.

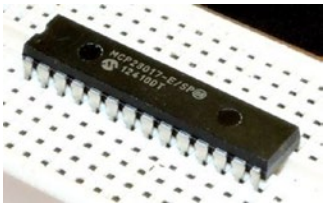

Figure 11-1. *An MCP23017 chip on the breadboard*

MCP23017

Microchip's MCP23017 can be purchased for as little as $1.99; it communicates with the Pi over the I2C bus. Because the I2C bus requires only two power lines and two data lines, it needs only a four-conductor ribbon cable and thus can cover some distance. Figure 11-2 shows the schematic pinout.

© Warren Gay 2017
W. Gay, *Custom Raspberry Pi Interfaces*, DOI 10.1007/978-1-4842-2406-9_11

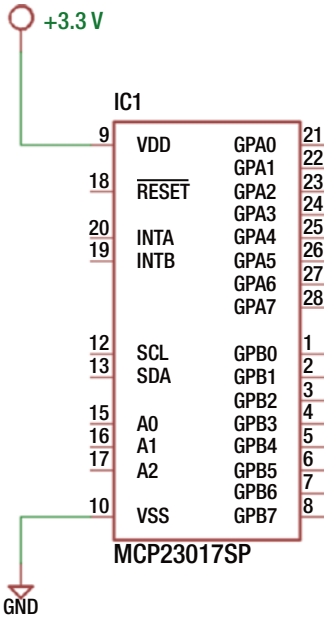

Figure 11-2. *MCP23017 pinout*

The active low RESET pin can be tied to the supply if not required, since a software reset is still possible. The signal forces a chip reset like the name implies.

The SCL and SDA are the two I2C communication connections. Chip inputs A0 through A2 allow you to configure the I2C address for the chip. Grounding them gives the chip address 0x20. Table 11-1 summarizes the addresses.

Table 11-1. *MCP23017 I2C Addresses*

A2	A1	A0	Address (Hex)
0	0	0	0x20
0	0	1	0x21
0	1	0	0x22
0	1	1	0x23
1	0	0	0x24
1	0	1	0x25
1	1	0	0x26
1	1	1	0x27

The INTA and INTB output pins are optional signals that can provide interrupt notification to the Raspberry Pi over one or two GPIO ports. The remaining pins GPA0 through GPA7 and GPB0 through GPB7 provide 16 more GPIO ports, which are under software configuration control.

Wiring

Figure 11-3 shows the wiring necessary to attach the MCP23017 to the Raspberry Pi. On the breadboard, simply wire the T-Cobbler connections marked SCL and SDA to pins 12 and 13, respectively.

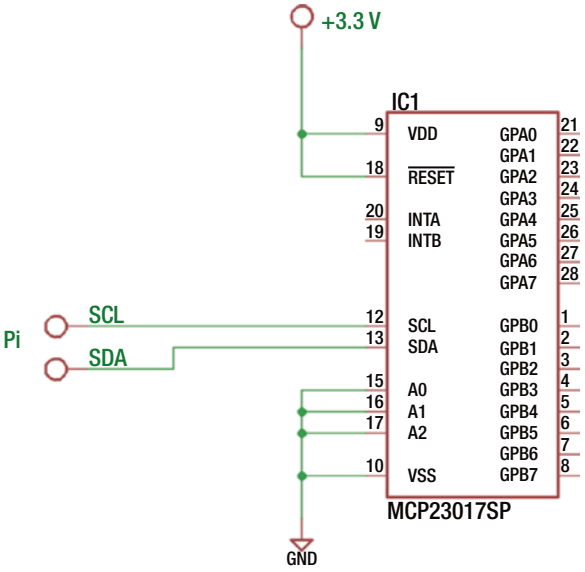

Figure 11-3. *MCP2301 wired to the Raspberry Pi*

Make certain that the RESET pin is also wired to the supply so that it doesn't become active. For simplicity and agreement with the provided software, connect inputs A0 through A2 to ground. This configures the chip to respond to I2C address 0x20. Signals INTA and INTB are outputs and can be ignored for the moment.

After reset, the MCP23017 chip automatically configures all the GPAx and GPBx pins as inputs, without a pull-up resistor. This is not ideal because this leaves the CMOS inputs floating. But for this initial experiment, the chip should tolerate it.

If you haven't already done so, install `i2c-tools`.

```
$ sudo apt-get install i2c-tools
```

Once installed, then list your I2C buses.

```
$ i2cdetect -l
```

If nothing is displayed, then you need to make sure that the I2C drivers get loaded. The new kernels now use the `/boot/config.txt` file to enable I2C support. Edit the file so that the `i2c` line is uncommented as follows:

```
# Uncomment some or all of these to enable the optional hardware interfaces
dtparam=i2c_arm=on
#dtparam=i2s=on
#dtparam=spi=on
```

Save your changes and reboot to try again.

```
sudo /sbin/shutdown -r now
```

After the reboot, list the I2C buses again.

```
$ i2cdetect -l
i2c-1 i2c 3f804000.i2c I2C adapter
```

From this, you now see `i2c-1` is available as expected. Now probe for your MCP23017 device. The command-line argument 1 shown next indicates to scan bus `i2c-1`.

```
$ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: 20 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

If all was successful, you should see a "20" listed among the hyphens. This tells you that the device is reachable on the `i2c-1` bus.

Output GPIO Experiment

With your breadboard wired according to Figure 11-3, you can perform a GPIO output test. Enter the subdirectory `mcp23017` in the source code and type `make` to build the programs there. Once that is done, you can run the output test.

```
$ ./mcp_out
GPIOA = 0x0B
GPIOB = 0xC1
```

This program will write the hexadecimal value 0x0B to GPIO port A and 0xC1 to GPIO port B. You'll examine the source code later.

With your DMM or an LED and dropping resistor (refer to Figure 10-4 in Chapter 10), probe the MCP23017 GPIO output pins. With this program run, you should observe the values listed in Table 11-2. Pay special attention to the chip pin numbers when wiring or taking readings. The pins for GPIOA and GPIOB are physically laid out in reverse order from each other.

Table 11-2. *Output Readings of the MCP23017 After Running mcp_out*

GPIO A Pin	Port	Result	GPIO B Pin	Port	Result
28	GPA7	Low	8	GPB7	High
27	GPA6	Low	7	GPB6	High
26	GPA5	Low	6	GPB5	Low
25	GPA4	Low	5	GPB4	Low
24	GPA3	High	4	GPB3	Low
23	GPA2	High	3	GPB2	Low
22	GPA1	Low	2	GPB1	Low
21	GPA0	High	1	GPB0	High

Figure 11-4 illustrates my breadboard setup. One nice aspect of I2C is the wiring simplicity. The plastic DIP (PDIP) form of the chip consists of 28 pins. The pin arrangement is somewhat unusual in that the V_{DD} (+3.3V) goes to pin 9, while pin 10 (V_{SS}) is the ground connection. Be sure that the RESET pin is wired high or it will sporadically reset or stay in reset mode. The remaining white wires shown in Figure 11-4 ground A0 through A2 so that the I2C address is established as hex 0x20.

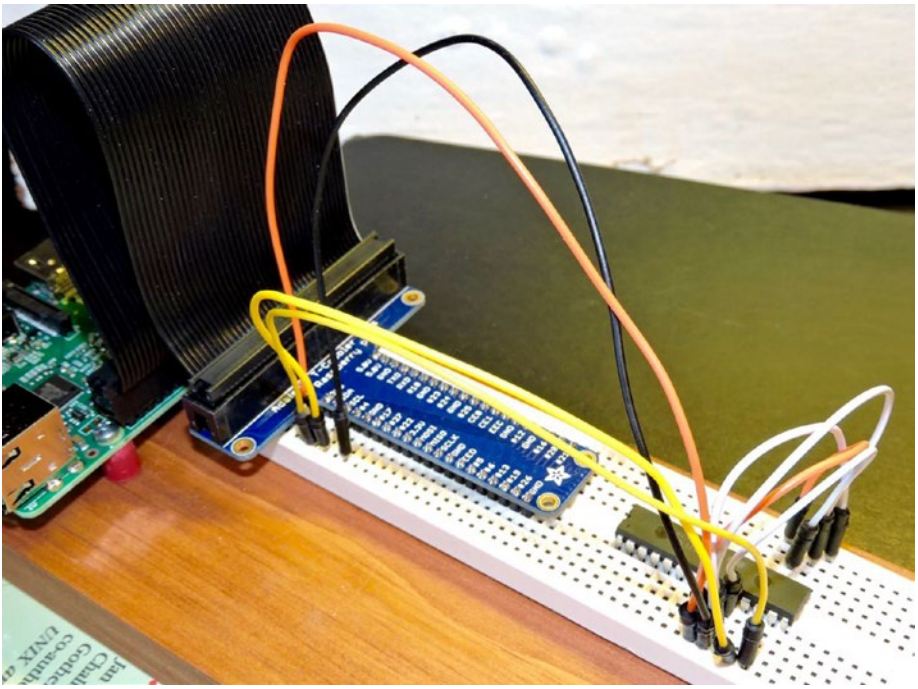

Figure 11-4. *MCP23017 breadboard setup*

Input Experiment

With no change to the breadboard circuit, you can now run an input experiment. This is one area that the I2C peripheral excels: it can software reconfigure the 16 GPIO pins as outputs or inputs. With the program compiled earlier (in the `mcp23017` subdirectory), you can just invoke `mcp_in`.

```
$ ./mcp_in
GPIOA = 0xFF, GPIOB = 0xFF
GPIOA = 0xBF, GPIOB = 0xFF
GPIOA = 0xFF, GPIOB = 0xFF
GPIOA = 0xDF, GPIOB = 0xFF
GPIOA = 0xFF, GPIOB = 0xFF
GPIOA = 0xDF, GPIOB = 0xFF
GPIOA = 0xFF, GPIOB = 0xFF
GPIOA = 0xFF, GPIOB = 0xFD
GPIOA = 0xFF, GPIOB = 0xFF
GPIOA = 0xFF, GPIOB = 0xFD
GPIOA = 0xFF, GPIOB = 0xFF
GPIOA = 0xFF, GPIOB = 0xFD
GPIOA = 0xFF, GPIOB = 0xFF
GPIOA = 0xFF, GPIOB = 0xFD
```

Initially, you will see only the following line:

```
GPIOA = 0xFF, GPIOB = 0xFF
```

After you see this initial report, ground one end of a Dupont wire (to the Pi). With the other end of the wire, touch or insert the wire in various places for GPIO A or GPIO B. Be careful not to touch any other pins, however. As you do this, changes in the input port (or ports) will be reported on your console session. Sometimes there will be "stutter" output. This is because of the speed of the Pi and the contact bouncing or scratching that occurs.

When the joy of the experiment diminishes, press ^C (Control-C) to end the program.

Software Operations

The flexibility of the MCP23017 GPIO peripheral requires extra responsibility on the software side. To complicate things further, Microchip provides two methods for addressing *registers* within its chip. This is configured by the `IOCON.BANK` bit. This chapter assumes `IOCON.BANK=0`, including the software provided. This results in the registers being laid out as in Table 11-3.

Table 11-3. *MCP23017 Register Addresses*

Register Name	Hexadecimal Address	Register Description
IODIRA	00	I/O Direction for GPIO A
IODIRB	01	I/O Direction for GPIO B
IPOLA	02	Input Polarity for GPIO A
IPOLB	03	Input Polarity for GPIO B
GPINTENA	04	Interrupt on Change for GPIO A
GPINTENB	05	Interrupt on Change for GPIO B
DEFVALA	06	Default Compare for Interrupt GPIO A
DEFVALB	07	Default Compare for Interrupt GPIO B
INTCONA	08	Interrupt on Change Control GPIO A
INTCONB	09	Interrupt on Change Control GPIO B
IOCON	0A	Configuration Control
IOCON	0B	
GPPUA	0C	Pull-up Resistor for GPIO A
GPPUB	0D	Pull-up Resistor for GPIO B
INTFA	0E	Interrupt Flags for GPIO A

(*continued*)

Table 11-3. (*continued*)

Register Name	Hexadecimal Address	Register Description
INTFB	0F	Interrupt Flags for GPIO B
INTCAPA	10	Interrupt Capture for GPIO A
INTCAPB	11	Interrupt Capture for GPIO B
GPIOA	12	GPIO A
GPIOB	13	GPIO B
OLATA	14	Output Latch for GPIO A
OLATB	15	Output Latch for GPIO B

In this discussion, please keep in mind that the I2C and register addresses are two different entities. The register address allows you to access registers within the peripheral, while the I2C address allows you to select the peripheral device.

Essentially, Table 11-3 shows that the least significant bit in this mode selects GPIO A or B, with the register selection shifted left one bit. Hence, within the program, you make use of the following macros:

```
#define IODIR                   0
#define IPOL                    1
#define GPINTEN                 2
#define DEFVAL                  3
#define INTCON                  4
#define IOCON                   5
#define GPPU                    6
#define INTF                    7
#define INTCAP                  8
#define GPIO                    9
#define OLAT                    10

#define GPIOA                   0
#define GPIOB                   1

#define MCP_REGISTER(r,g) (((r)<<1)|(g))
```

Using these macros, you can select register `IODIR` for `GPIOB` using `MCP_REGISTER(IODIR,GPIOB)` in the code.

I2C Header Files

Under Raspbian Linux, you need certain header files for performing I2C I/O.

```
#include <sys/ioctl.h>
#include <linux/i2c.h>
#include <linux/i2c-dev.h>
```

These provide structure definitions and macros necessary to issue your I2C I/O operations.

Opening the I2C Driver

When the `i2cdetect -l` command was run, you saw that the `i2c-1` bus was available. You access this bus driver with the path `/dev/i2c-1`. Consequently, you use the `open(2)` system call to gain access to the driver. Listing 11-1 is a code snippet from `mcp_out.cpp` showing the open call.

Listing 11-1. Opening the I2C Bus

```
036 static const char *i2c_device = "/dev/i2c-1";
037 static int i2c_fd = -1;

093 int
094 main(int argc,char **argv) {
095   int rc;
096
097   i2c_fd = open(i2c_device,O_RDWR);
098   if ( i2c_fd == -1 ) {
099         fprintf(stderr,"%s: opening %s\n",
100               strerror(errno),
101               i2c_device);
102         exit(1);
103  }
```

Line 036 defines the character string `i2c_device` path to be opened. The opened file descriptor is returned in variable `i2c_fd` if successful, or the value –1 is returned if there is a failure. The file descriptor acts as a handle to the Linux driver. Lines 098 through 103 report the error, if the open is unsuccessful.

I2C Write

Writing to an I2C device under Linux is fairly straightforward, once you've seen how it is done. Listing 11-2 shows the function `i2c_write`, which is used to write 1 to 2 bytes to the MCP23017 peripheral chip.

Listing 11-2. The I2C Write Routine

```
062 static int
063 i2c_write(int addr,int reg,int ab,uint8_t byte) {
064   struct i2c_rdwr_ioctl_data msgset;
065   struct i2c_msg iomsgs[1];
066   uint8_t reg_addr = MCP_REGISTER(reg,ab);
067   uint8_t buf[2];
068   int rc;
```

```
069
070   buf[0] = reg_addr;
071   buf[1] = byte;
072
073   iomsgs[0].addr = unsigned(addr);
074   iomsgs[0].flags = 0;
075   iomsgs[0].buf = buf;
076   iomsgs[0].len = 2;
077
078   msgset.msgs = iomsgs;
079   msgset.nmsgs = 1;
080
081   rc = ioctl(i2c_fd,I2C_RDWR,&msgset);
082   return rc < 0 ? -1 : 0;
083 }
```

The argument `addr` is the I2C address of the MCP23017 (in this case 0x20). The argument `reg` selects the MCP23017 register that you want to target, while `ab` selects GPIO A or GPIO B. The value to be written is provided in argument `byte`.

The Linux structure `i2c_rdwr_ioctl_data` is used to pass your request to the driver. The array of `i2c_msg` structures allows you to define the I2C operations required. In this case, you want to perform only one write, so the array consists of a single structure.

Line 066 computes the peripheral register address using the `MCP_REGISTER` macro. The data for the I2C message is created in lines 070 and 071. The message is the MCP23017 register you want to change (`buf[0]`) and then the value to assign to it (`buf[1]`).

Lines 073 to 076 describe the I2C operation. Line 073 tells the I2C driver the peripheral's I2C address (0x20). Line 074 indicates that this is a *write* operation because it is missing the flag `I2C_M_RD`, which is defined as a Linux macro. The data buffer address is provided in line 075, while the length of the data in bytes is described in line 076. These lines describe everything necessary for one I2C write message.

Because there can be several I2C messages performed at once, lines 078 and 079 are necessary. Line 078 describes how to locate the first I2C operation (described by struct `i2c_msg`). Line 079 in this case will inform the driver that only one I2C message is to be processed.

Line 081 is where you pass this assembled request to the kernel driver. If the operation fails for any reason, the return code `rc` will have the value –1 assigned to it. The first argument is the open file descriptor acting as a handle to the driver. The second argument, `I2C_RDWR`, indicates that you want to perform an I2C I/O operation. The last argument is a pointer to the "message set" to be performed.

I2C Read

Reading from an I2C device is almost the same as writing, but many devices need to know which peripheral register to read from. For this reason, you use a write and then read with the MCP23017 chip. Listing 11-3 shows the read routine used in the program `mcp_in.cpp`.

Listing 11-3. I2C Read Routine

```
062 int
063 i2c_read_data(int addr,int ab,uint8_t& byte) {
064   struct i2c_rdwr_ioctl_data msgset;
065   struct i2c_msg iomsgs[2];
066   uint8_t txbuf[1];
067   int rc;
068
069   txbuf[0] = MCP_REGISTER(GPIO,ab);
070
071   iomsgs[0].addr = iomsgs[1].addr = unsigned(addr);
072   iomsgs[0].flags = 0;            // Write
073   iomsgs[0].buf = txbuf;
074   iomsgs[0].len = 1;
075
076   iomsgs[1].flags = I2C_M_RD;     // Read
077   iomsgs[1].buf = &byte;          // Pass back data byte
078   iomsgs[1].len = 1;
079
080   msgset.msgs = iomsgs;
081   msgset.nmsgs = 2;
082
083   rc = ioctl(i2c_fd,I2C_RDWR,&msgset);
084   return rc < 0 ? -1 : 0;
085 }
```

In this routine you see that the setup of the message structures is almost the same. Line 062, however, declares an array of two `i2c_msg` structures, because you need to perform two operations.

Lines 070 through 074 set up a write of 1 byte, which simply contains the peripheral register that you want to read (define in line 069). This write operation will simply write the register to be selected to the MCP23017 device prior to the read operation.

Lines 076 through 078 set up a read of 1 byte into the variable `byte`. Here the value `byte` is passed using a C++ *reference* type. This allows the changed value to be passed back to the caller as a read byte.

Line 081 informs the driver that there are two I/O operations and ties everything together in the variable `msgset`. If the call to `ioctl(2)` in line 083 is successful, the peripheral register will be written to the device followed by a read. The sequence causes the indicated peripheral register to be read and returned.

Configuration

The programs `mcp_out.cpp` and `mcp_in.cpp` configure the MCP23017 peripheral through a series of I2C writes in the main program. Listing 11-4 shows a code snippet from the main program `mcp_in.cpp`.

Listing 11-4. Main Program Configuration in mcp_in.cpp

```
129  rc = i2c_write_both(i2c_addr,IOCON,0b01000100);  // MIRROR=1,ODR=1
130  assert(!rc);
131
132  rc = i2c_write_both(i2c_addr,GPINTEN,0x00);  // No interrupts enabled
133  assert(!rc);
134
135  rc = i2c_write_both(i2c_addr,DEFVAL,0x00);  // Clear default value
136  assert(!rc);
137
138  rc = i2c_write_both(i2c_addr,OLAT,0x00);     // OLATx=0
139  assert(!rc);
140
141  rc = i2c_write_both(i2c_addr,IPOL,0b00000000);  // No inverted polarity
142  assert(!rc);
143
144  rc = i2c_write_both(i2c_addr,GPPU,0b11111111);  // Enable all pull-ups
145  assert(!rc);
146
147  rc = i2c_write_both(i2c_addr,IODIR,0b11111111);  // All are inputs
148  assert(!rc);
```

Each of these writes uses a routine called `i2c_write_both`. This convenience routine simply performs a call to `i2c_write` of the same values, for GPIO A and GPIO B, reducing the amount of code. Line 147, for example, configures the MCP23017 register `IODIR`. When the bits in this register are written as 1, they configure the corresponding GPIO pin as an input. You can find more information about configuration in the Microchip datasheet (you can Google *MCP23017 datasheet* to find it).

Interrupt Capability

Earlier you saw a demonstration of reading MCP23017 GPIO inputs. But if the inputs are attached to push buttons, for example, how does your program stay informed without constantly polling the peripheral? You could continuously perform I2C read operations as the `mcp_in.cpp` program does, but this keeps the I2C bus very busy.

Microchip provides a feature in the MCP23017 peripheral to provide one or two interrupt signals for input signal *changes*. You may configure it for separate GPIO A and B interrupts, or you may configure it to use one pin for both. Pins 19 and 20 of the chip provide the interrupt signals.

The next demonstration will use one pin (INTA pin 20) to signal a change of GPIO input, whether for GPIO A or B. To do this, you must the set the peripheral's configuration.

- `IOCON.MIRROR = 1` (INTA represents both GPIO A and B).
- `IOCON.ODR = 0` (when true, the INTA is in "open drain" configuration)

- IOCON.INTPOL = 0 (you're going to make the INTA pin active low; this applies only when ODR=0).
- GPINTEN.BPINTx bits 0 through 7 enable interrupts for each pin (1=enabled)
- INTCON.IOCx bits 0 through 7 are either:
  - Compared to the DEFVAL register (when set to 1)
  - Compared to self (when set to 0)
- DEFVAL.DEFx bits 0 through 7 set the compare value for interrupts (optionally)

The DEFVAL settings apply only if you enable DEFVAL in IOCON.IOCx. For example, if IOCON.IOC2 for GPIOB is set to 1, then DEFVAL.DEF2 is used to compare against the GPIO B input 2. If the values differ, an interrupt is generated.

You're going to use INTCON.IOCx=0 so that the interrupt occurs on any change in the input signal. In this configuration, the DEFVAL bit setting has no influence. You can think of this configuration as comparing the input against its last known value.

The program being used for this experiment is named mcp_int.cpp. By default, this program assumes you have connected GPIO#5 to the MCP23017 INTA pin 20. If you need to change this to something else, then this is the line you need to edit:

```
041 static int gpio_inta = 5;  // GPIO for INTA signal
```

The main routine configures the peripheral for interrupts as described earlier. This is illustrated in Listing 11-5.

Listing 11-5. MCP23017 Interrupt Configuration

```
144   rc = i2c_write_both(i2c_addr,IOCON,0b01000000);  // MIRROR=1,
                                                          ODR=0,INTPOL=0
145   assert(!rc);
146
147   rc = i2c_write_both(i2c_addr,IODIR,0xFF);  // All are inputs
148   assert(!rc);
149
150   rc = i2c_write_both(i2c_addr,INTCON,0x00);  // Interrupts compare to self
151   assert(!rc);
152
153   rc = i2c_write_both(i2c_addr,GPINTEN,0xFF);  // All interrupts enabled
154   assert(!rc);
155
156   rc = i2c_write_both(i2c_addr,IPOL,0x00);  // No inverted input polarity
157   assert(!rc);
158
159   rc = i2c_write_both(i2c_addr,GPPU,0xFF);  // Enable all pull-ups
160   assert(!rc);
```

The comments indicate the configuration values being set. For simplicity, `mcp_int.cpp` sets all GPIO pins as inputs. Once all the configuration is performed, the main program enters its main loop, shown in Listing 11-6.

Listing 11-6. Main Loop for Interrupt Processing

```
162   {
163         uint8_t gpioa, gpiob, inta, intf_a = 0, intf_b = 0;
164
165          rc = i2c_read_data(i2c_addr,GPIOA,GPIO_,gpioa);
166          assert(!rc);
167
168          rc = i2c_read_data(i2c_addr,GPIOB,GPIO_,gpiob);
169          assert(!rc);
170
171          printf("GPIOA 0x%02X INTFA 0x%02X, GPIOB 0x%02X INTFB 0x%02X\n",
172                gpioa, intf_a,
173                gpiob, intf_b);
174
175          for (;;) {
176                inta = gpio.read(gpio_inta);
177                if ( inta != 0 ) {
178                      // No interrupt
179                      usleep(1);
180                      continue;
181                }
182
183                // Process interrupt
184
185                rc = i2c_read_data(i2c_addr,GPIOA,INTF,intf_a);
186                assert(!rc);
187
188                rc = i2c_read_data(i2c_addr,GPIOB,INTF,intf_b);
189                assert(!rc);
190
191                rc = i2c_read_data(i2c_addr,GPIOA,INTCAP,gpioa);
192                assert(!rc);
193
194                rc = i2c_read_data(i2c_addr,GPIOB,INTCAP,gpiob);
195                assert(!rc);
196
197                printf("GPIOA 0x%02X INTFA 0x%02X, GPIOB 0x%02X INTFB
                   0x%02X\n",
198                      gpioa, intf_a,
199                      gpiob, intf_b);
200          }
201    }
```

Lines 165 to 173 simply report the current state of the MCP23017 inputs prior to entering the loop. The loop, starting in line 175, polls Raspberry Pi GPIO (#5) to see whether the chip is reporting an interrupt. If the INTA line is high, there is no interrupt being reported, so a short sleep is performed and the loop is restarted (lines 177 to 181).

If the Raspberry Pi GPIO (#5) is low, this indicates that one or both of the MPC23017 ports has registered a change on an input pin. Lines 185 to 189 read the INTF register for GPIO A and GPIO B. This register sets a 1 bit where an input value has changed. Lines 191 to 195 read the interrupt capture register for GPIO A and B. These registers reflect the input line states at the time of the interrupt. Lines 197 to 199 simply report the changes.

Figure 11-5 shows how to wire the circuit for the interrupt test. The only new connection is the connection going from the Raspberry Pi's GPIO #5 to the INTA pin 20 of the peripheral.

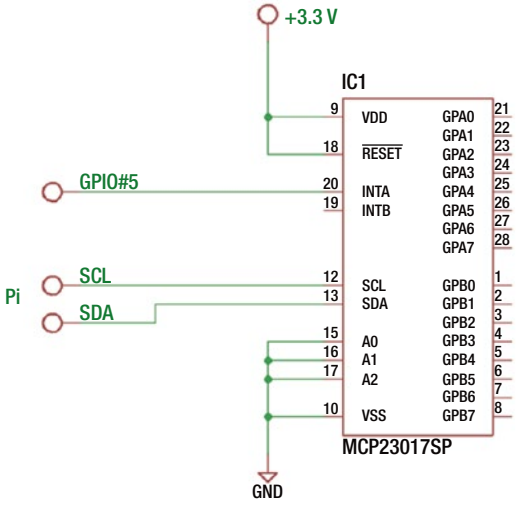

Figure 11-5. *MCP23017 wired for interrupt signaling*

After wiring according to Figure 11-5, run the program `mcp_int`. All chip pins are configured as input with a pull-up resistor so that they will read high with no connections to them. While the program was running in the following session, I touched a ground wire to GPA5 (bit 5 of GPIOA on pin 28) in the example session shown here:

```
$ ./mcp_int
GPIOA 0xFF INTFA 0x00, GPIOB 0xFF INTFB 0x00
GPIOA 0xDF INTFA 0x20, GPIOB 0xFF INTFB 0x00
GPIOA 0xFF INTFA 0x20, GPIOB 0xFF INTFB 0x00
GPIOA 0xFF INTFA 0x20, GPIOB 0xFF INTFB 0x00
GPIOA 0xFF INTFA 0x20, GPIOB 0xFF INTFB 0x00
```

The first line is simply the initial reported setting. The program will wait there until you do something (like grounding an input). The second line reports that GPIOA captured the hex value DF as part of the change (in the first line reported as FF). The INTFA register reports 20, indicating the bit that changed (bit 5). This confirms that the GPA5 went from a 1 bit to a 0 bit.

Because of the rate of changes, sometimes you will register changes but not see them in the capture register. In the example session, INTFA continues to register the fact that bit 5 changed but showed no change in the captured value (FF). Because of contact bouncing (of my Dupont wire), the chip registered a change, but the state of the input changed back to a 1 bit by the time the capture was made.

Interrupt Profit and Loss

What did you gain from the use of the INTA pin? This pin is intended by Microchip for use as a real interrupt signal for a microcomputer. Here you attached it to the Pi's GPIO #5 and polled that signal for changes. So, what did you gain?

While you still utilized polling, you polled a single Pi GPIO, which is far cheaper than continuously sending I2C read commands to the bus. By polling a Pi GPIO, you freed up available bandwidth on the I2C bus, which might be used for other devices. The I2C bus was used only when needed.

Summary

While the 74HC165 and 74HC595 parts were easy to use, they were not reconfigurable. The MCP23017, however, offers 16 fully software configurable GPIO pins. The price of this is the additional software complexity. The chip also provides additional interrupt input processing that is absent in the simpler 74HC165 and 74HC595 parts. Finally, the ability to connect several devices to the I2C bus makes the MCP23017 a flexible peripheral, whether the device is local or remote to the Raspberry Pi.

CHAPTER 12

MPD/MPC Hardware Controls

Raspberry Pi computers are attractive for embedded styled solutions due to their small size and low cost. This chapter will draw on earlier chapters to allow you to build a stand-alone embedded music player box.

To expedite the software effort, the MPD/MPC software will be used, which is easily installed on your Pi. The Music Player Daemon (MPD) component is the process that runs in the background to stream the music. The companion command-line control program `mpc` gives you control of it. Using this combined with a small LCD display, rotary control, and a digital volume control, you can build a small embedded music playing box. The rotary control will choose music selections (or Internet radio stations), the LCD will display the selections chosen, and the volume control sets the volume at the twist of a knob.

Audio Preparation

If you're already a user of MPD/MPC on your Raspberry Pi, you can skip this section. Otherwise, let's take a moment to prepare for and install some software.

Raspbian Linux is always being improved upon and updated. If you haven't done this in a while, it is probably useful to do some updating and upgrade now.

```
$ sudo apt-get update
```

This will update a number of packages that are currently installed. You may also want to upgrade Raspbian with this:

```
$ sudo apt-get upgrade
```

If you choose to skip the upgrade step but run into problems with the MPD/MPC software later, then you might want to try again after upgrading.

The next step is to install the MPD software.

```
$ sudo apt-get install mpd mpc
```

The MPD package is the daemon that runs in the background and actually performs the streaming of music files or Internet radio stations. The MPD package contains the command-line program `mpc`, which allows you to control MPD through various

© Warren Gay 2017
W. Gay, *Custom Raspberry Pi Interfaces*, DOI 10.1007/978-1-4842-2406-9_12

command-line options. There will be a number of package dependencies, which will automatically install as part of this. This process is fairly painless.

Next you need to make some decisions about where your sound will be coming out of. I am using the audio output jack of the Pi 3, but others may prefer to play audio to their HDMI device instead (which is the default). Depending upon what you decide, you may have to configure your audio system to get it operational.

There are a couple of helpful commands available to aid you in testing your audio output. One is the `aplay` command. If you use the `find` command as follows, you can find several WAV files to try:

```
$ find /usr/share -name '*.wav'
```

Or you could just try to play the file shown next using the `aplay` command (see the following text about `alsa-utils` if `aplay` is not installed):

```
$ aplay /usr/share/scratch/Media/Sounds/Electronic/ComputerBeeps1.wav
```

If everything is working, you should be able to hear audio. If you don't hear anything yet, don't panic. It might be working but is muted or at a low volume setting. The easiest way to check this is by using the `alsamixer` command.

```
$ alsamixer
```

If it isn't already installed, you may need to install it with this:

```
$ sudo apt-get install alsa-utils
```

This will bring up a screen that will allow you to view the available sound controls and change the volume. Figure 12-1 shows my Pi's simple setup.

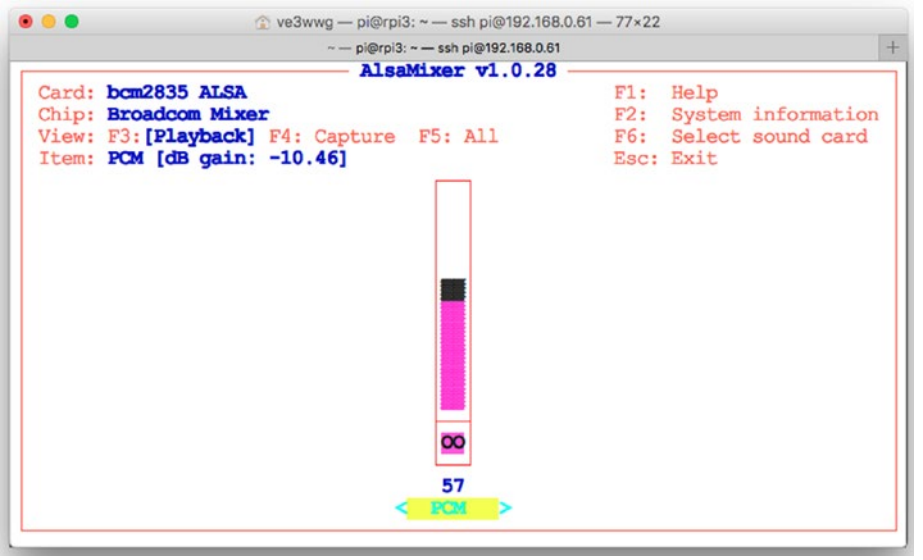

Figure 12-1. *The alsamixer command screen*

In Figure 12-1, I have only one sound device, labeled "PCM." The volume shown is 57 percent, and it can be increased by pressing the up arrow key. To exit the program, press the Escape key.

Some Raspberry Pi owners have more discriminating tastes in their audio fidelity and may use devices like the HiFiBerry DAC+ or similar. Because of the broad scope of configuration for audio devices for the Pi, you may have to reach out to forums or seek help through Google.

MPD/MPD

Once you have your audio functional, it is time to try out MPD/MPC. After installing MPD, you should have the file `/etc/mpd.conf`, which can be tailored to your needs. If you're running a simple setup like mine, you may want to edit it so that you have an audio device like this:

```
audio_output {
      type         "alsa"
      name         "My ALSA Device"
      device       "hw:0,0"    # optional
}
```

After making MPD configuration changes, you should restart the daemon.

```
$ sudo service mpd restart
```

This should get your earphone jack working as a sound source. There is some additional Pi-specific help at [2]. The configuration for MPD can be quite involved, but fortunately the help available on the Internet is plentiful.

Make sure that the daemon is running, either automatically started or manually started. To check that it is running, you can do this:

```
$ ps aux | grep mpd
mpd 530  0.0  2.6 138076 25020 ? Ssl  18:24   0:00 /usr/bin/mpd --no-daemon
```

If you don't see it running, you can manually start it with this:

```
$ sudo service mpd start
```

The next step is to make sure you have media that can be played. For demonstration purposes, I uploaded a few Mark Knopfler MP3 files to my Pi (a good place for them is in your `~pi/Music` directory). Once you have some music media, you need a *playlist*. To check your current playlist, use the `mpc` command.

```
$ mpc playlist
```

When you're setting this up for the first time, the playlist will be empty. I will be using MP3 music files for this chapter, but you can use streaming Internet services as well. But for simplicity, I advise that you use MP3 files initially.

■ **Note** `mpc help` will list helpful information for all of the `mpc` subcommand options.

If you have a playlist already and you want to start over, you can issue the `mpc` command.

```
$ mpc clear
```

This will clear your playlist, which can be verified with the following:

```
$ mpc playlist
```

The general way that you add a resource to your playlist is with the following command:

```
$ mpc add URI
```

There seems to be no wildcard way to add several MP3 files, so you must use the shell to help.

```
$ for f in ~/Music/*.mp3 ; do mpc add "file://$f" ; done
```

This causes the bash shell to loop over all MP3 files in the `~/Music` directory, issuing the `mpc add` command for each file (note the added `"file://"` in front of the `$f` variable). The text `"file://$f"` should always be in double quotes so that spaces in the file name do not cause problems.

Once you have done this, you should be able to display your playlist.

```
$ mpc playlist
Mark Knopfler - 5.15 A.M.
Mark Knopfler - All That Matters
...
Mark Knopfler - Postcards from Paraguay
Dire Straits & Mark Knopfler - Sailing To Philadelphia
Mark Knopfler - Song for Sonny Liston
Mark Knopfler - Stand Up Guy
Mark Knopfler - Sucker Row
Dire Straits & Mark Knopfler - The Long Road (Theme From ''Cal'')
Mark Knopfler - The Trawlerman's Song
Dire Straits & Mark Knopfler - What It Is
Mark Knopfler - Whoop De Doo
Dire Straits & Mark Knopfler - Why Aye Man
```

Make sure your volume is established at some reasonable level. This can be controlled through the `mpc` command, which you will leverage in the demonstration program.

```
$ mpc volume
volume: 81%
$ mpc volume 85
volume: 85%   repeat: on    random: off   single: off   consume: off
```

The first command just queries the current volume setting, while the second sets the volume to 85 percent.

To start a particular selection playing, you can indicate the track using a 1-based selection number. The following command starts the fifth playlist item:

```
$ mpc play 5
Mark Knopfler - Boom, Like That
[playing] #5/23   0:00/5:49 (0%)
volume: 85%   repeat: on    random: off   single: off   consume: off
```

The status of the MPD daemon can be displayed with the `status` subcommand (which you'll also use in the demonstration program).

```
$ mpc status
Mark Knopfler - Boom, Like That
[playing] #5/23   1:23/5:49 (23%)
volume: 80%   repeat: on    random: off   single: off   consume: off
```

Finally, the MPD daemon can be told to stop.

```
$ mpc stop
```

Many more subcommands and options are available, which can be viewed with the `mpc help` command. I've listed only the bare essentials as they apply to the demonstration program.

Hardware Setup

Before the demonstration C++ code is presented, let's look at the hardware setup and test each of the components involved. For this project, you have the following hardware elements connected to the Pi 3:

- 16×2 LCD from Chapter 4
- Rotary control from Chapter 8
- Potentiometer from Chapters 6 and 7

These controls are allocated to the following `mpd` functions:

- The LCD shows the current selection playing, as well as the selection being chosen when the rotary control is moved.
- The rotary control allows you to select any selection from your playlist. Choosing selection 0 causes the `mpd` daemon to stop playing.
- The potentiometer is used to set the MPD volume control, from 0 to 100 percent volume.

Figure 12-2 shows my own breadboard setup. The rotary control in the center has the round knob, while the volume control is to the right and can be seen with a chicken-head knob. The ADC PCB for the volume control is at the upper near right of the breadboard in the figure. The PCB to the right of that is the 3V to 5V level converter for the I2C connection to the LCD.

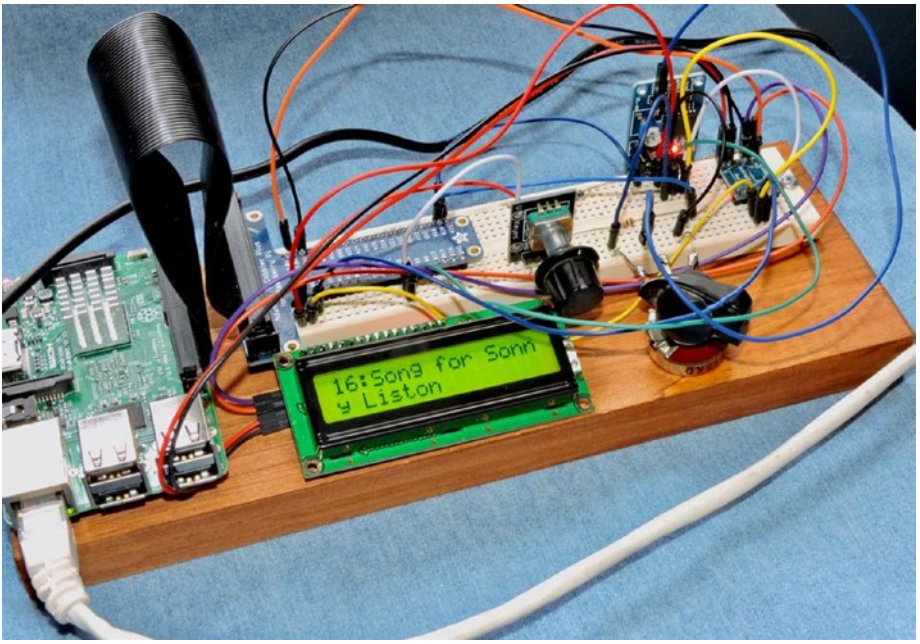

Figure 12-2. *MPC hardware setup: LCD, rotary control, and volume control (with chicken-head knob)*

While there is much more you could add to this arrangement like a pause push button, and so on, this example was kept basic to make the code simpler to explain. This also leaves an opportunity for you to expand this, adding fast-forward and reverse controls, for example.

The rotary control was wired as used in Chapter 8. Likewise, the LCD wiring was the same as in Chapter 4. Finally, the ADC is wired as presented in Chapter 6. The potentiometer is wired so that the wiper arm goes to the ADC input AIN3 (don't forget to remove the jumper P6). The other ends of the potentiometer are wired to +3.3 volts and ground (review Chapter 7's Figure 7-4).

Test Volume Control

The volume control is perhaps the simplest to test first, so wire that up and test it with the `readadc` program presented in Chapter 6.

```
$ cd ./pcf8591
$ ./readadc
230
```

Adjust the knob and check for a change in successive readings. If the control seems backward, reverse the outer +3.3V and ground connections to the potentiometer.

Test Rotary Control

To make sure you have the rotary control wired up correctly, check it with the `yl040c` program presented in Chapter 8.

```
$ cd ./YL-040
$ ./yl040c
Monitoring rotary control:
Step +1, Count 1
Step +1, Count 2
Step +1, Count 3
Step +1, Count 4
Step -1, Count 3
Step -1, Count 2
Step -1, Count 1
Step -1, Count 0
Step -1, Count -1
^C
```

If you see event messages like these as you rotate the control, then it is operational.

Test LCD

Finally, make sure the 16×2 device is operational. Use the `lcd` program presented in Chapter 4.

```
$ cd ./pcf8574
$ ./lcd
```

If the display is operational, you should see the display shown in Figure 4-10.

■ **Caution** Don't rush the hookup of the LCD device because 5V levels are involved. Make sure the level converter is properly connected before wiring up the Pi's I2C signals SDA and SCL.

The mpcctl Program

Now that you know the hardware controls are working, you can now compile and run the `mpcctl` demonstration program. If the program is not yet compiled, do so now.

```
$ cd ./mpd
$ make
```

If you had to choose different GPIO ports and/or I2C addresses for your hardware, you must customize these in the source code `mpcctl.cpp`. If, for example, your volume control requires changes, edit the following lines:

```
023 static const char *i2c_device = "/dev/i2c-1";
...
025 static uint8_t adc_i2c_addr = 0x48;     // I2C Addr for pcf8591 ADC
026 static uint8_t adc_i2c_ainx = 3;        // Default to AIN3
```

If your rotary control used different GPIO numbers than the ones I've used, edit the following line, changing the values 20 and 21:

```
305     Flywheel rsw(*gpio,20,21);          // Flywheeling control
```

Finally, if your LCD I2C address is different, then edit the value `0x27` in the following line:

```
408     LCD1602 lcd(0x27);                  // LCD class
```

Main Program

This application is written in C++ to take advantage of its library resources (STL) but using only the most basic C++ features to keep the program smaller. The `mpcctl` program is organized using the following threads of control:

- The main program
- Thread 1, the rotary control
- Thread 2, `mpd` status update
- Thread 3, LCD display
- Thread 4, volume control

It is possible to poll all of these controls in one thread, but the code becomes more difficult from a design and readability standpoint. There are also some operations such as loading a playlist that can be time-consuming and can make the controls unresponsive. Having each control managed by its own thread makes the software design simpler. However, the inter-thread interaction requires some special care. As so often is the case, there are many trade-offs.

The STL makes the creation of a thread simple, as you'll see in Listing 12-1. It requires only that you include the STL thread header file and use the `std::thread` type.

Listing 12-1. The mpcctl Main Program

```
016 #include <thread>
...
243 static void
244 update_status() {
...
269 }
...
276 static void
277 lcd_thread(LCD1602 *plcd) {
...
297 }
...

303 static void
304 rotary_control(GPIO *gpio) {
...
372 }
...
378 static void
379 vol_control() {
...
399 }
...
405 int
406 main(int argc,char **argv) {
407     GPIO gpio;              // GPIO access object
408     LCD1602 lcd(0x27);      // LCD class
409     int rc;
410
411     // Check initialization of GPIO class:
412     if ( (rc = gpio.get_error()) != 0 ) {
413         fprintf(stderr,"%s: starting gpio (sudo?)\n",strerror(rc));
414         exit(1);
415     }
416
417     // Initialize LCD
418     if ( !lcd.initialize() ) {
419         fprintf(stderr,
                "%s: Initializing LCD1602 at I2C bus %s, address 0x%02X\n",
420             strerror(errno),
421             lcd.get_busdev(),
422             lcd.get_address());
423         exit(1);
424     }
425
426     i2c_fd = open(i2c_device,O_RDWR);
```

```
427     if ( i2c_fd == -1 ) {
428         fprintf(stderr,"Warning, %s: opening %s\n",
429             strerror(errno),
430             i2c_device);
431     }
432
433     lcd.clear();
434     puts("MPC with Custom Controls");
435     lcd.set_cursor(false);
436
437     std::thread thread1(rotary_control,&gpio);
438     std::thread thread2(update_status);
439     std::thread thread3(lcd_thread,&lcd);
440     std::thread thread4(vol_control);
441
442     thread1.join();
443     thread2.join();
444     thread3.join();
445     thread4.join();
446
447     return 0;
448 }
```

The first thing the `main` program does is to instantiate the `GPIO` class (`gpio`). In the constructor, some code runs that requires root access (hence the need to run the program as `setuid`).

```
406 main(int argc,char **argv) {
407     GPIO gpio;              // GPIO access object
```

To check on the success of the constructor, you must check for it later with a call to the `GPIO::get_error` method.

```
411     // Check initialization of GPIO class:
412     if ( (rc = gpio.get_error()) != 0 ) {
```

Line 412 checks to make sure that the GPIO access was granted. Otherwise, the rotary control will not function.

Immediately after instantiating the `GPIO` class in line 407, the `LCD1602` class is instantiated as `lcd` with I2C address `0x27`. Line 418 checks that the `lcd` object initialized successfully.

```
408     LCD1602 lcd(0x27);      // LCD class
...
418     if ( !lcd.initialize() ) {
```

For the benefit of the volume control, the I2C bus defined at line 023 is opened and saved as `i2c_fd` in line 024.

```
023 static const char *i2c_device = "/dev/i2c-1";
024 static int i2c_fd = -1;
...
426     i2c_fd = open(i2c_device,O_RDWR);
427     if ( i2c_fd == -1 ) {
```

The volume control has been coded as an optional device. A warning is issued if the PCF8591 ADC is not found.

```
433     lcd.clear();
434     puts("MPC with Custom Controls");
435     lcd.set_cursor(false);
```

These lines simply clear the LCD and turn off the LCD cursor.

The main program then creates four threads of control.

```
437     std::thread thread1(rotary_control,&gpio);
438     std::thread thread2(update_status);
439     std::thread thread3(lcd_thread,&lcd);
440     std::thread thread4(vol_control);
```

The function `rotary_control` is passed the address of the instantiated `GPIO` object, since it will need it to read two inputs. Likewise, function `lcd_thread` also receives a pointer to the instantiated `LCD1602` object. The other two threads, `update_status` and `vol_control`, receive no arguments.

At this point, these four thread functions run in parallel within the same process memory space. Since the Pi 2 and 3 have four cores, it is possible that these functions truly execute simultaneously at times. An `htop` display in Figure 12-3 illustrates the executing threads and their CPU utilization.

```
  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 3874 root       20   0 39580  1140  1004 S  6.7  0.1  0:02.18 ./mpcctl
 3875 root       20   0 39580  1140  1004 S  6.2  0.1  0:02.08 ./mpcctl
  534 mpd        20   0  136M 29816 21048 S  2.9  3.1  3:16.62 /usr/bin/mpd --no-daemon
  691 mpd        20   0  136M 29816 21048 S  2.4  3.1  2:54.09 /usr/bin/mpd --no-daemon
 3918 pi         20   0  4604  3260  2248 R  1.4  0.3  0:00.40 htop
  692 mpd        -51  0  136M 29816 21048 S  0.5  3.1  0:18.48 /usr/bin/mpd --no-daemon
  690 mpd        20   0  136M 29816 21048 S  0.5  3.1  0:02.43 /usr/bin/mpd --no-daemon
 3876 root       20   0 39580  1140  1004 S  0.0  0.1  0:00.06 ./mpcctl
  431 root       20   0  2564  1776  1492 S  0.0  0.2  0:00.04 /sbin/dhcpcd -q -b
 3880 root       20   0 39580  1140  1004 S  0.0  0.1  0:00.02 ./mpcctl
 3877 root       20   0 39580  1140  1004 S  0.0  0.1  0:00.01 ./mpcctl
```

Figure 12-3. *Sample htop display of the mpcctl program executing*

Modifying the `usleep(3)` calls in the code will increase or decrease the CPU utilization. With *increased* sleep times, the CPU overhead *decreases* but normally at the cost of responsiveness of the controls.

The remainder of the main program is a series of joins.

```
442     thread1.join();
443     thread2.join();
444     thread3.join();
445     thread4.join();
```

These block the main thread from continuing until those threads terminate. With the `mpcctl.cpp` code as written, this never happens. This program could be enhanced to set a shutdown flag for all the threads and have them terminate. Then the main program would successfully join and exit. That is left as an exercise for you.

When you want the program to exit, just press Control-C. This will kill all threads and the main program simultaneously.

Rotary Encoder Thread

Listing 12-2 shows the rotary encoder thread. The pointer to the main program's `GPIO` object is passed into the thread by a pointer `gpio`.

Listing 12-2. The Rotary Encoder Thread

```
303 static void
304 rotary_control(GPIO *gpio) {
305     Flywheel rsw(*gpio,20,21);      // Flywheeling control
306     std::string title;              // Current playing title
307     time_t last_update, last_sel;
308     std::vector<std::string> playlist;
309     int pos = 0, of = -1, last_play = -1;
310     int rc;
311
312     // Initialize:
313     load(playlist);
314     get_mpc_status(title,pos,of);
315     if ( pos >= 0 )
316             last_play = pos;
317     last_update = last_sel = time(0);
318
319     for (;;) {
320         rc = rsw.read();        // Read rotary switch
321         if ( rc != 0 ) {        // Movement?
322             // Position changed:
323
```

```
324             if ( current_pos > -1 ) {
325                 pos = current_pos;      // Now playing a different tune
326                 current_pos = -1;
327             }
328
329             if ( playlist.size() < 1 ) {
330                 put_msg("(empty playlist)");
331             } else  {
332                 pos = (pos + rc);
333                 if ( pos < 0 )
334                     pos = -1;
335
336                 if ( pos >= int(playlist.size()) )
337                     pos = playlist.size() - 1;
338
339                 if ( pos >= 0 ) {
340                     std::stringstream ss;
341                     ss << (pos+1) << '>'
                            << song_title(playlist[pos].c_str());
342                     std::string text = ss.str();
343
344                     put_msg(text.c_str());
345                 } else  {
346                     put_msg("(Stop)");
347                 }
348                 last_sel = time(0);
349             }
350         } else  {
351             // No position change
352
353             time_t now = time(0);
354             if ( now - last_update > 60 ) {
355                 // Everything 60 seconds, update the
356                 // playlist in case it has been changed
357                 // externally
358                 load(playlist);
359                 last_update = now;
360             } else if ( now - last_sel > 1 ) {
361                 // Start playing new selection
362                 if ( last_play != pos ) {
363                     mpc_play(title,pos,of);
364                     last_play = pos;
365                 }
366                 last_sel = time(0);
367             }
368
```

```
369             usleep(200); // Don't eat the CPU
370         }
371     }
372 }
```

The `Flywheel` class is instantiated in line 305, using GPIO pins #20 and #21.

```
305     Flywheel rsw(*gpio,20,21);      // Flywheeling control
```

As part of the thread initialization, it performs the function `load()` to load the playlist into the `std::vector<std::string>` container named `playlist`. Then the current title (if any), position (`pos`), and number of playlist entries (`of`) are returned through pass-by-reference arguments to the call to `get_mpc_status()`. All of this information comes from executing the `mpc` command using a pipe (more about this later).

```
313     load(playlist);
314     get_mpc_status(title,pos,of);
```

The main loop of this thread begins here:

```
319     for (;;) {
320         rc = rsw.read();        // Read rotary switch
321         if ( rc != 0 ) {        // Movement?
322             // Position changed:
```

Here you read the rotary switch in line 320 and check to see whether there were any rotation events (variable `rc`). If so, you enter the block of code starting with line 322.

There is a somewhat difficult interaction between what is playing now and the rotary control. If the control has not been touched for one or more tracks, you could be at a selection point beyond what you last recorded in the local variable `pos`. So, you share the variable `current_pos` between the thread `update_status()` and the `rotary_control()` threads.

```
029 static volatile int current_pos = -1;   // Last song playing status
```

Note the `volatile` attribute for `current_pos`. Without this attribute, compiler optimization in one thread may not see a change made by another thread because of holding the value in a register. In this program, you don't use a mutex for this value, since the read or write of an `int` value is atomic on *this* platform. Otherwise, this is considered bad practice and would not be permitted in mission-critical software. In this program, you want to avoid threads interlocking more than is necessary because that affects the responsiveness of the controls.

With that background, you arrive at the following code:

```
324             if ( current_pos > -1 ) {
325                 pos = current_pos;      // Now playing a different tune
326                 current_pos = -1;
327             }
```

The thread examines `current_pos`, and if it's found to be greater than or equal to zero, it knows that a new selection has been established by the other thread. When this happens, you reset the local variable `pos` to that value and perform the rotary adjustment based upon that. Otherwise, line 332 just adjusts the selection as per usual.

```
332                 pos = (pos + rc);
```

With the rotary adjustment made, you update the LCD display.

```
341                     ss << (pos+1) << '>'
                            << song_title(playlist[pos].c_str());
342                     std::string text = ss.str();
343
344                     put_msg(text.c_str());
```

The 16×2 LCD is rather limited for space, and the rotary control thread shares the display with the `update_status()` thread. That thread shows the current title playing. In the display, the > is used for rotary control updates.

```
18>Sucker Row
```

The now-playing display uses a colon (:).

```
18:Sucker Row
```

When there is no rotary control movement, the `else` block starting at line 350 is performed. The code updates its playlist every minute with the code in lines 354 to 359. The main drawback to this is that this can cause a pregnant pause in reaction to rotary control movement, should that begin during this load. If you have no plans to update the playlist external to `mpcctl`, then this code block can be commented out.

The next block checks to see whether the position has changed. If so, it tells the MPD to start playing a new selection (line 363). This is tricky because you don't want to issue play commands while the control is still rotating. So, the code checks to see whether at least one second has passed in line 360, without further rotary events.

```
360             } else if ( now - last_sel > 1 ) {
361                 // Start playing new selection
362                 if ( last_play != pos ) {
363                     mpc_play(title,pos,of);
364                     last_play = pos;
365                 }
366                 last_sel = time(0);
367             }
```

Finally, at the end of the rotary control thread, you give up the CPU for a short period of time (200μsec) to allow other processes to run.

```
369             usleep(200); // Don't eat the CPU
```

Lowering this value increases the rotary control responsiveness but eats more CPU cycles. Lowering this time reduces CPU overhead, but the flywheeling effect is usually the first casualty.

LCD Thread

The purpose of this thread is to update the LCD display. It must arbitrate the requests of different threads wanting to display. The LCD thread operates in a lazy loop, as shown in Listing 12-3.

Listing 12-3. The Lazy LCD Thread

```
276 static void
277 lcd_thread(LCD1602 *plcd) {
278     LCD1602& lcd = *plcd;           // Ref to LCD class
279     std::string local_title;
280     bool chgf;
281
282     for (;;) {
283             mutex.lock();
284             if ( disp_changed ) {
285                     disp_changed = false;
286                     chgf = local_title != disp_title;
287                     local_title = disp_title;
288             } else  {
289                     chgf = false;
290             }
291             mutex.unlock();
292
293             if ( chgf ) // Did info change?
294                     display(lcd,local_title.c_str());
295             else    usleep(50000);
296     }
297 }
```

This thread is passed a pointer to the `LCD1602` class through argument `plcd`. This is then assigned to the reference variable in line 278 so that access is as if the class was declared locally as `lcd`. This isn't required but is more convenient than working with a pointer.

The lazy loop starts in line 282. For safe access to the title to be displayed, a mutex named aptly as `mutex` is locked in line 283.

```
031 static std::mutex mutex;                    // Thread lock
032 static volatile bool disp_changed = false;  // True if display changed
033 static std::string disp_title;              // Displayed title
...
282     for (;;) {
283         mutex.lock();
```

```
284         if ( disp_changed ) {
285             disp_changed = false;
286             chgf = local_title != disp_title;
287             local_title = disp_title;
288         } else  {
289             chgf = false;
290         }
291         mutex.unlock();
```

Since `disp_title` is an object of type `std::string`, you must use a mutex for multithreaded access to it. The `std::string` object will be calling upon `malloc(3)` and `realloc(3)` for buffer management, so this is not something that should be interrupted.

So, line 283 blocks until exclusive access is granted to the calling thread. Then the Boolean value `disp_changed` is checked, and when true, the new `disp_title` value is copied to local variable `local_title`. The local variable `chgf` is also set to true in this case (line 286). Otherwise, `chgf` is set to false (line 289). At the end of the block, the mutex is unlocked so that other threads can acquire the lock (line 291).

When the flag `chgf` is true, you then update the LCD display using a helper function named `display()` in line 294. Otherwise, the thread sleeps for 50ms.

```
293             if ( chgf ) // Did info change?
294                     display(lcd,local_title.c_str());
295             else    usleep(50000);
```

Listing 12-4 shows the helper display routine.

Listing 12-4. The Helper Function display()

```
221 static void
222 display(LCD1602& lcd,const char *msg) {
223     char line1[17], line2[17];
224
225     puts(msg);
226
227     strncpy(line1,msg,16)[16] = 0;
228     lcd.clear();
229     lcd.set_cursor(false);
230     lcd.putstr(line1);
231
232     if ( strlen(msg) > 16 ) {
233         strncpy(line2,msg+16,16)[16] = 0;
234         lcd.moveto(2,0);
235         lcd.putstr(line2);
236     }
237 }
```

The purpose of this routine is to split the text supplied in `msg` into two 16-character segments in buffers `line1` and `line2`. Line 225 just copies `msg` to standard output as is. But lines 227 to 230 extract `line1` for the LCD display.

If there is a line 2, lines 232 to 236 are performed to display it.

MPC Status Thread

Another of the executing threads is the one that checks the currently playing track. It is presented in Listing 12-5.

Listing 12-5. The Update Now Playing Status Thread

```
243 static void
244 update_status() {
245     time_t now;
246     std::string title;
247     int cpos = 0, epos = -1;
248
249     for (;;) {
250         now = time(nullptr);
251
252         // While status info is stale
253         while ( now - changed > 1 ) {
254             changed = now;
255
256             if ( get_mpc_status(title,cpos,epos) ) {
257                 if ( cpos >= 0 ) {
258                     std::stringstream ss;
259
260                     ss << (cpos+1) << ':' << song_title(title.c_str());
261                     put_msg(ss.str().c_str());
262                 } else {
263                     put_msg(title.c_str());
264                 }
265             }
266         }
267         usleep(10000);
268     }
269 }
```

This thread executes a loop starting in line 249. It checks the current time in line 250 and then determines how many seconds have elapsed since the last change (variable `changed` in line 253). If there has been at least one second elapsed since the last change, the helper routine `get_mpc_status()` is called in line 256. Arguments `title`, `cpos`, and `epos` are passed by *reference* and are updated by the function call. If a non-negative `cpos` value is returned, you send a "now playing" message to the LCD by calling `put_msg()`. If the value of `cpos` is negative, it is a message like "Stopped" and is displayed in line 263.

The helper routine put_msg() is used to provide the message to the display thread safely.

```
090 static void
091 put_msg(const char *msg) {
092
093     mutex.lock();                           // Lock mutex
094     if ( strcmp(msg,disp_title.c_str()) != 0 ) {
095             disp_title = msg;               // Save new message
096             disp_changed = true;            // Mark it as changed
097     }
098     changed = time(0);                      // Hold off status update
099     mutex.unlock();
100 }
```

The mutex is locked in line 093, and a check is made in line 094 to see whether the message is any different than the one you have already. If it differs, then disp_title is set to the new message and the value disp_changed is set to true. The time held in variable changed is updated to the current time.

Volume Control Thread

The last main component is the volume control thread. This is another lazy thread, which quietly determines whether the potentiometer has changed in value. The thread is presented in Listing 12-6.

Listing 12-6. Volume Control Thread Within mpcctl.cpp

```
378 static void
379 vol_control() {
380     int pct, last_pct = -1;
381
382     for (;;) {
383         pct = read_volume();
384         if ( pct != last_pct ) {
385             std::stringstream ss;
386
387             ss << "mpc volume " << pct << " 2>/dev/null 0</dev/null";
388             FILE *pfile = popen(ss.str().c_str(),"r");
389             char buf[2048];
390
391             while ( fgets(buf,sizeof buf,pfile) )
392                 ;       // Read and discard output, if any
393             pclose(pfile);
394             last_pct = pct;
```

```
395         } else {
396             usleep(50000);  // usec
397         }
398     }
399 }
```

The main loop starts in line 382, checking the potentiometer reading. The value returned is in percent (variable `pct`). If this value has changed, then the code in lines 385 to 394 are performed to change the `mpd` volume. The string stream `ss` (line 385) is used to construct a command of the following format:

```
mpc volume <pct> 2>/dev/null 0</dev/null
```

where the percentage value is supplied in place of `<pct>` in the previous line. A read pipe is opened in line 388.

```
388             FILE *pfile = popen(ss.str().c_str(),"r");
```

For non-C++ folks, the `ss.str()` converts the string stream to a `std::string`. The appended method `.c_str()` converts that to a C string that `popen(3)` can use. The second argument indicates that you will read (`"r"`) from the command pipe. While there is no data of interest in this case, you read and discard any lines of data in lines 391 and 392. If you were to prematurely close the pipe, this might kill the `mpc` command that you started with a `SIGPIPE` signal.

Finally, line 393 closes the pipe using `pclose(3)`. Don't make the mistake of using `fclose(3)` for an open pipe. Since `popen(3)` performs a `fork(2)` to create a new process (the `mpc` command), one of the `wait` system calls must be called to clean up the process status held by the kernel. The `pclose(3)` performs this necessary housekeeping. Failure to do this will result in zombie processes.

Program Summary

Much effort was put into keeping this demonstration program small, but you can see how complexity collects like a magnet. Yet, by breaking the process into four threads, some of the complexity is reduced, with the individual functions becoming four relatively simple threads. The alternative would be a complicated state machine.

Let's summarize how `mpcctl.cpp` controls the Music Player Daemon using the `mpc` command and the `popen(3)` library function (for lines not shown earlier, view the source file `mpcctl.cpp`):

- `mpc volume <pct>` (line 388), set volume
- `mpc status` (line 159), query status and "now playing"
- `mpc playlist` (line 173), load the playlist into `mpcctl`
- `mpc play <selection>` (line 205), start the playing of a selection
- `mpc stop` (line 206), stop playing

The alternative to using the `mpc` command, which communicates with the `mpd` daemon process, is to use the libmpd library. But the investment in code development for that approach is much higher.

Summary

This chapter presented an embedded Raspbian application involving a hardware rotary selection control, a small dedicated LCD, and a volume control. This is just the beginning of what is possible. For example, you could add a pause push button and fast-forward/backward controls. The latter could be accomplished with another rotary control that includes a push button. Push to pause or rotate to go forward/backward in a given track.

For an application like this, it is easy to see how custom hardware controls are far superior to using a standard computer keyboard and mouse. Your largest challenge may lie in knowing when to stop adding to the custom controls design.

Bibliography

[1] "Music Player Daemon." Atom News. N.p., n.d. Web. 29 Oct. 2016. <https://www.musicpd.org/>.

[2] "Rpi Music Player Daemon." ELinux.org. N.p., n.d. Web. 31 Oct. 2016. <http://elinux.org/Rpi_Music_Player_Daemon>.