8

ARDUINO I²C PROGRAMMING

This first chapter on I²C programming will start by discussing the Arduino platform, since it's probably safe to say that more lines of I²C code have been written for the Arduino than for any other.

This chapter covers the following information:

- An introduction to basic I²C programming
- A discussion of the Wire programming model that the Arduino library and IDE uses
- Arduino I²C read and write operations
- Accessing multiple I²C ports on various Arduino devices

This book tends to use Arduino *sketches* (programs) as the basis for generic examples, so a good understanding of Arduino I²C programming will be invaluable as you continue through subsequent chapters.

THE MCP4725 DIGITAL-TO-ANALOG CONVERTER

This book uses the MCP4725 DAC to demonstrate programming various control devices (SBCs), since the MCP4725 is easy to program and understand. It has the following features:

- 12-bit resolution
- On-board nonvolatile memory (EEPROM)
- External A0 address pin
- Normal or power-down mode
- Single-supply operation: 2.7 V to 5.5 V
- Standard (100 kbit/sec), fast (400 kbit/sec), and high (3.4 Mbit/sec) speeds
- Eight available I^2C addresses (though any given MCP4725 IC supports only two different addresses, a full range of eight addresses is possible since there are four different variants of the MCP4725, each supporting a different pair of addresses)

For Part III of this book, there are two I^2C operations of interest: writing a 12-bit digital value to the DAC and reading the current DAC output and EEPROM values from the chip.

The MCP4725 will respond to one of the following I^2C addresses: 0x60/0x61, 0x62/0x63, 0x64/0x65, or 0x66/0x67. An address pin on the MCP4725 provides the LO bit (bit 0) of this address. Bits 1 and 2 are determined by the particular IC you purchase. For example, the Adafruit MCP4725 breakout board uses an IC that responds to addresses 0x62 and 0x63; the SparkFun variant responds to addresses 0x60 and 0x61. If you purchase boards from Adafruit and SparkFun, you can put four of these boards on the same I^2C bus without having to resort to using an I^2C multiplexer. (There is a sneaky way to hook more of these boards to the same bus by using the address selection bit as a "chip select" line; see *https://mitchtronic.blogspot.com/2017/03/addressing-multiple-mcp4724s-in-same.html* for an example.) If you want to use chips with addresses 0x64/0x65 or 0x66/0x67, you could search for various boards on Amazon or build your own breakout board.

Both Adafruit and SparkFun have made their boards open hardware via the Creative Commons license, so you could build these boards and substitute in the appropriate MCP4725 IC. Note that these designs use surface-mounted parts and are not easy to assemble by hand. Check out the Adafruit and SparkFun designs at *https://github.com/sparkfun/MCP4725_Breakout/tree/v14* and *https://github.com/adafruit/Adafruit-MCP4725-PCB*. For more information about the MCP4725, see Chapter 15.

8.1 Basic I²C Programming

In Chapter 2, you learned that an I²C transmission begins with the output of a start condition followed by an address-R/W byte, followed by zero or more bytes of data, and, finally, end with a stop condition. The controller places these data bytes on the I²C bus, either by bit banging or by some hardware registers.

The only parts of this transmission that are common to all I²C devices are the start condition, the very first address byte, and the stop condition. Any bytes the controller transmits after the address byte until a stop condition comes along are specific to the peripheral responding to the address in the address byte.

The MCP4725 supports several command formats based on data you transmit immediately after the address byte. The programming examples in this part of the book will use only one of those commands: the *Fast Mode Write command*. This command requires 3 bytes on the I²C bus, as shown in Table 8-1.

Table 8-1: Fast Mode Write Command

First byte	Second byte	Third byte
Address	HO DAC value	LO DAC value
`aaaa aaax`	`0000 hhhh`	`1111 1111`

In Table 8-1, the `aaaa aaa` bits are the MCP4725 address. These will be `1100cba` where bits `c` and `b` are hard-coded into the IC itself and `a` comes from the address line on the chip. This corresponds to addresses 0x60 through 0x67. (Keep in mind that the I²C protocol shifts these address bits one position to the left and expects the R/W bit in bit 0. For this reason, the address byte will actually contain the values 0xC0 through 0xCF, depending on the IC address and the state of the R/W line.) The `hhhh 1111 1111` bits are the 12 bits to write to the digital-to-analog conversion circuitry. The HO 4 bits of the second byte must contain zeros (they specify the Fast Mode Write command and power-down mode). Assuming a 5-V power supply to the chip, the 3-byte sequence *0xC0, 0x00, 0x00* (the 3 bytes from Table 8-1) will write the 12-bit value 0x000 to the DAC at address 0x60, which will cause 0 V to appear on the DAC's output. Writing the 3-byte sequence *0xC0, 0x08, 0x00* will put 2.5 V on the output pin. Writing the 3-byte sequence *0xC0, 0x0F, 0xFF* will put 5 V on the analog output pin. In general, a value between 0x000 and 0xFFF (linearly) maps to a voltage between 0 V and 5 V on the DAC analog output. All you need is some way of placing these 3 bytes on the I²C bus.

Whereas the DAC uses the HO 4 bits of the second byte to specify the command (0b0000 is the Fast Mode Write command), the DAC read command is simpler still. The R/W bit in the address byte is all the MCP4725 needs to determine how to respond. It responds by returning 5 bytes: the

first is some status information (which you can ignore until Chapter 15, where I discuss the MCP4725 in detail), the second byte contains the HO 8 bits of the last value written to the DAC, and the third byte contains the LO 4 bits of the last value written in bits 4 through 7 (and bits 0 through 3 don't contain any valid data). The fourth and fifth bytes contain some status information and the 14 bits held in the on-chip EEPROM (see Chapter 15 for more information about the EEPROM).

How you place bytes on the I^2C bus and how you read data from the I^2C bus entirely depends on the system, library functions, and operating system (if any) you're using. This chapter discusses I^2C on the Arduino; therefore, we're going to consider how to read and write data on the I^2C bus using the Arduino library code.

8.2 Basic Wire Programming

The Arduino library responsible for I^2C communication is the Wire library. I^2C communication functions are not built into the Arduino language (which is really just C++ with some default include files). Instead, you have to enable the Arduino I^2C library code by including the following statement near the beginning of your program's source file:

```
#include <Wire.h>
```

Note that *Wire.h* must have an uppercase *W* on certain operating systems (Linux, in particular).

The *Wire.h* header file creates a singleton class object named `Wire` that you can use to access the class functions. You do not have to declare this object in your programs; the header file automatically does this for you. The following sections describe the various available `Wire` functions.

8.2.1 Wire Utility Functions

The `Wire.begin()` function initializes the Arduino Wire (I^2C) library. You must call this function once before executing any other functions in the Wire library. The convention is to call this function in the Arduino `setup()` function.

Without a parameter, `Wire.begin()` will initialize the library to work as a controller device on the I^2C bus. If you specify a 7-bit integer as an argument, this will initialize the library to operate as a peripheral device on the I^2C bus.

The `Wire.setClock()` function allows you to change the I^2C clock frequency, supplied as an integer parameter. This call is optional; the default clock speed is 100 kHz. Most Arduino boards will support 100,000 or 400,000 as the argument. A few high-performance boards might support 3,400,000 (high-speed mode). A few will also support 10,000 (low-speed mode on the SMBus).

Keep in mind that all peripherals and CPU(s) on the I^2C bus must be capable of supporting the clock speed you select. That is, you must set a clock speed that is no faster than the slowest peripheral on the bus.

8.2.2 Wire Read Operations

The `Wire.requestFrom()` function reads data from an I^2C peripheral device. There are two forms of the `Wire.requestFrom()` function call:

```
Wire.requestFrom( address, size )
Wire.requestFrom( address, size, stopCond )
```

In each of these calls, `address` is the 7-bit peripheral address, `size` is the number of bytes to read from the device, and the optional `stopCond` argument specifies whether the function issues a stop condition (if true) after receiving the bytes. If false, then the function sends a restart condition. If the optional `stopCode` argument is not present, the function uses `true` as the default value (to issue a stop condition after receiving the data).

NOTE *The Arduino library maintains a 32-byte buffer for incoming I^2C data reads. Because `Wire.requestFrom()` reads all incoming data before returning to its caller, an I^2C peripheral can transfer a maximum limit of 32 bytes in one operation using this call.*

Once the controller receives the data from the peripheral, an application can read that data using the `Wire.read()` and `Wire.available()` functions. The `Wire.available()` function returns the number of bytes left in the internal receive buffer, while the `Wire.read()` function reads a single byte from the buffer. Typically, you would use these two functions to read all the data from the internal buffer using a loop such as the following:

```
while( Wire.available() )
{
    char c = Wire.read(); // Read byte from buffer

    // Do something with the byte just read.
}
```

There is no guarantee that the peripheral will actually transmit the number of bytes requested in the call to the `Wire.requestFrom()` function—the peripheral could return *less* data. Therefore, it is always important to use the `Wire.available()` function to determine exactly how much data is in the internal buffer; don't automatically assume it's the amount you requested.

The peripheral determines the actual amount of data it returns to the controller. In almost all cases, the amount of data is fixed and is specified in the datasheet for the peripheral (or by the peripheral's design). In theory, a peripheral could return a variable amount of data. How you retrieve such data is determined by the peripheral's design and is beyond the scope of this chapter.

To read data from a peripheral device, a controller must transmit the peripheral address and an R/W bit equal to 1 to that peripheral. The `Wire.requestFrom()` function handles this. After that, the peripheral will

transmit its data bytes. The Arduino controller will receive those bytes and buffer them to be read later. Note, however, that the full read operation takes place with the execution of the `Wire.requestFrom()` function.

8.2.3 Wire Write Operations

A controller can write data to a peripheral using the `Wire.beginTransmission()`, `Wire.endTransmission()`, and `Wire.write()` functions. The `beginTransmission()` and `endTransmission()` functions bracket a sequence of write operations.

The `Wire.beginTransmission()` function takes the following form:

```
Wire.beginTransmission(address)
```

where `address` is the 7-bit peripheral address. This function call builds the first byte of the data transmission consisting of the address and a clear R/W bit.

There are three forms of the `Wire.write()` function:

```
Wire.write( value )
Wire.write( string )
Wire.write( data, length )
```

The first form appends a single byte to an internal buffer for transmission to the peripheral. The second form adds all the characters in a string (not including the zero-terminating byte) to the internal buffer for transmission to the peripheral. The third form copies some bytes from a byte array to the internal buffer (the second argument specifies the number of bytes to copy).

NOTE *In addition to its aforementioned 32-byte buffer for incoming I²C data reads, the Arduino library maintains a 32-byte buffer for outgoing I²C writes. Although you can have multiple calls to the various write functions between a `Wire.beginTransmission()` call and a `Wire.endTransmission()` call, the cumulative length must be 32 bytes or less.*

The `Wire.endTransmission()` function takes the address byte and data bytes from the internal buffer and transmits them over the I²C bus. This function call takes two forms:

```
Wire.endTransmission()
Wire.endTransmission( stopCond )
```

The first form transmits the data in the internal buffer and follows that transmission with a stop condition. The second form uses the single Boolean argument to determine whether it should send a stop condition (`true`) after transmitting the data (the function transmits a restart if `stopCond` is `false`).

Remember that the actual data transmission does not take place until the execution of the `Wire.endTransmission()` function call. The other calls simply build up an internal buffer for later transmission.

8.2.4 Wire Peripheral Functions

The Arduino functions up to this point have assumed that the Arduino is acting as an I^2C bus controller device. You can also program an Arduino to act as a peripheral device. The Arduino library provides two functions for this purpose:

```
Wire.onReceive( inHandler )
Wire.onRequest( outHandler )
```

In the first function, `inHandler` is a pointer to a function with the following prototype: `void inHandler( int numBytes )`. In the second, `outHandler` is a pointer to a function with the following prototype: `void outHandler()`.

The Arduino system will call `outHandler` whenever the (external) controller device requests data. The `outHandler` function will then use the `Wire.beginTransmission()`, `Wire.endTransmission()`, and `Wire.write()` functions to transmit data from the peripheral back to the (external) controller. The `inHandler` function will use the `Wire.begin()`, `Wire.available()`, and `Wire.read()` functions to retrieve data from the controller device.

8.3 Arduino I^2C Write Example

The program in Listing 8-1 demonstrates using the I^2C bus to talk to a SparkFun MCP4725 DAC breakout board. This program was written for and tested on a Teensy 3.2, though it should work with any compatible Arduino device (with slightly different timings).

The program generates a continuous triangle wave by continuously incrementing the DAC output from 0x0 to 0xfff (12 bits) and then decrementing from 0xfff back to 0x0. As you will see in the oscilloscope output, this program produces a triangle wave with slightly less than a 2.4-second period (around 0.42 Hz) when running on my setup (your mileage may vary). This frequency is determined by the amount of time it takes to write 8,189 12-bit values to the DAC. Since each transmission requires 3 bytes (address, HO byte and command, and LO byte), plus start and stop condition timings, it takes around 35 bit times at 100 kHz (10 µsec per bit time) to transfer each value.

```
// Listing8-1.ino
//
// A simple program that demonstrates I2C
// programming on the Arduino platform.

#include <Wire.h>

// I2C address of the SparkFun MCP4725 I2C-based
// digital-to-analog converter.

#define MCP4725_ADDR 0x60

void setup( void )
```

```
{
    Serial.begin( 9600 );
    delay( 1000 );
    Serial.println( "Test writing MCP4725 DAC" );
    Wire.begin(); // Initialize I2C library
}

void loop( void )
{
    // Send the rising edge of a triangle wave:

    for( int16_t dacOut = 0; dacOut < 0xfff; ++dacOut )
    {
        // Transmit the address byte (and a zero R/W bit):

      ❶ Wire.beginTransmission( MCP4725_ADDR );

        // Transmit the 12-bit DAC value (HO 4 bits
        // first, LO 8 bits second) along with a 4-bit
        // Fast Mode Write command (00 in the HO 2 bits
        // of the first byte):

      ❷ Wire.write( (dacOut >> 8) & 0xf );
        Wire.write( dacOut & 0xff );

        // Send the stop condition onto the I2C bus:

      ❸ Wire.endTransmission( true );

        // Uncomment this delay to slow things down
        // so it can be observed on a multimeter:
        // delay( 5 );
    }

    // Send the falling edge of the triangle wave:

    for( int16_t dacOut = 0xffe; dacOut > 0; --dacOut )
    {
        // See comments in previous loop.

        Wire.beginTransmission( MCP4725_ADDR );
        Wire.write( (dacOut >> 8) & 0xf );
        Wire.write( dacOut & 0xff );
        Wire.endTransmission( true );

        // Uncomment this delay to slow things down
        // so it can be observed on a multimeter:
        // delay( 5 );
    }
}
```

`Wire.beginTransmission()` initializes the Wire package to begin accepting data for (later) transmission on the I^2C bus ❶. The `Wire.write()` function copies data to transmit to the internal `Wire` buffers for later transmission

on the I²C bus ❷. After that, `Wire.endTransmission()` instructs the device to actually begin transmitting the data in the internal `Wire` buffers onto the I²C bus ❸.

Figure 8-1 shows one of the DAC 3-byte transmissions appearing on the I²C bus during the execution of the program in Listing 8-1 (this particular transmission was writing 0x963 to the DAC).

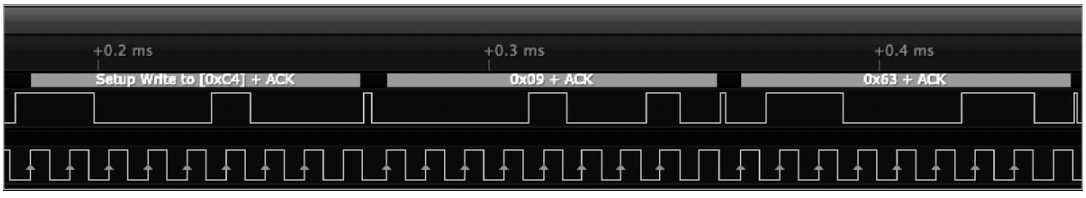

Figure 8-1: Sample I²C output during triangle wave transmission

As you can see in the oscilloscope output appearing in Figure 8-2, it takes approximately 2.4 seconds for a full cycle (one rising edge and one falling edge) of the triangle wave. Using the logic analyzer, I was able to determine that each 3-byte transmission took slightly less than 300 µsec, which roughly matches what you see on the oscilloscope output in Figure 8-2. Note that the timing between transmissions isn't constant and will vary by several microseconds between transmissions. This means 300 µsec is not a hard transmission time for 3 bytes.

The maximum frequency this software can produce based on a 100-kHz bus speed is approximately 0.4 Hz. To produce a higher frequency value, you would need to run the I²C bus at a higher clock frequency (for example, 400 kHz) or reduce the number of values you write to the DAC per unit time (for example, you can double the frequency by incrementing the loop counter by two rather than one).

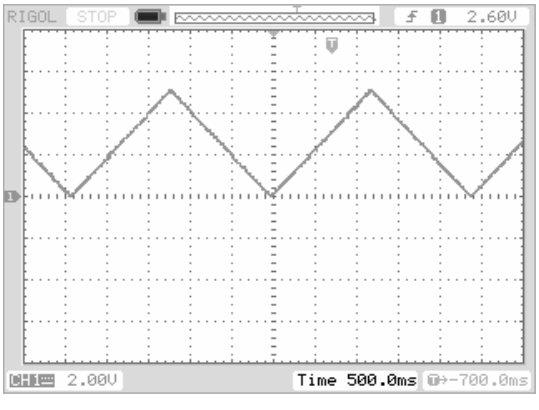

Figure 8-2: Triangle wave output from MCP4725

The code in Listing 8-1 gives up the I²C bus after each DAC transmission. If there were other controllers on the same bus talking to different peripherals, this would further reduce the maximum clock frequency of the triangle

wave (not to mention that it would add some distortion to the triangle wave if there were many pauses in the output sequence to the DAC). In theory, you could prevent this distortion by refusing to yield the I²C bus during the transmission; however, given the vast number of transmissions required here, the only reasonable solution to producing an undistorted triangle wave would be to ensure that the MCP4725 was the only device on the I²C bus.

8.4 Arduino I²C Read Example

Fundamentally, a DAC is an (analog) output-only device. You write a value to the DAC registers and an analog voltage magically appears on the analog output pin. Reading from a DAC doesn't make much sense. That said, the MCP4725 IC does support I²C read operations. A read command returns 5 bytes.

To read a value from the MCP4725, simply place the device's address on the I²C bus with the R/W line high. The MCP4725 will respond by returning 5 bytes: the first byte will be status information, the next two will be the last DAC value written, and the last pair of bytes will be the EEPROM value. The EEPROM stores a default value to initialize the analog output pin when the device powers up, before any digital value is written to the chip. See Chapter 15 for more details.

The program in Listing 8-2 demonstrates an I²C read operation.

```
// Listing8-2.ino
//
// This is a simple program that demonstrates
// I2C programming on the Arduino platform.
//
// This program reads the last written DAC value
// and EEPROM settings from the MDP4725. It was
// written and tested on a Teensy 3.2, and it also
// runs on an Arduino Uno.

#include <Wire.h>

// I2C address of the SparkFun MCP4725 I2C-based
// digital-to-analog converter.

#define MCP4725_ADDR 0x60

#define bytesToRead (5)
void setup( void )
{
    int     i = 0;
    int     DACvalue;
    int     EEPROMvalue;
    byte    input[bytesToRead];

    Serial.begin( 9600 );
    delay( 1000 );
    Serial.println( "Test reading MCP4725 DAC" );
```

```
    Wire.begin();  // Initialize I2C library

    Wire.requestFrom( MCP4725_ADDR, bytesToRead );
    while( Wire.available() )
    {
        if( i < bytesToRead )
        {
            input[ i++ ] = Wire.read();
        }
    }

    // Status byte is the first one received:

    Serial.print( "Status: " );
    Serial.println( input[0], 16 );

    // The previously written DAC value is in the
    // HO 12 bits of the next two bytes:

    DACvalue = (input[1] << 4) | ((input[2] & 0xff)  4);
    Serial.print( "Previous DAC value: " );
    Serial.println( DACvalue, 16 );

    // The last two bytes contain EEPROM data:

    EEPROMvalue = (input[3] << 8) | input[4];
    Serial.print( "EEPROM value: " );
    Serial.println( EEPROMvalue, 16 );

    while( 1 ); // Stop

}

void loop( void )
{
    // Never executes.
}
```

The following is the output from the program in Listing 8-2. Note that the output is valid only for my particular setup. Other MCP4725 boards may have different EEPROM values. Furthermore, the previous DAC value output is specific to the last write on my particular system (this was probably the last output written from Listing 8-1, when I uploaded the program in Listing 8-2 while the previous program was running).

```
Test reading MCP4725 DAC
Status: C0
Previous DAC value: 9B
EEPROM value: 800
```

The only thing interesting in this output is that I had programmed the MCP4725's EEPROM to initialize the output pin to 2.5 V on power-up (the halfway point with a 5-V power supply).

8.4.1 Arduino I²C Peripheral Example

The previous two sections described read and write operations from the perspective of a controller device. This section describes how to create an Arduino system that behaves as an I²C peripheral device. In particular, the source code appearing in Listing 8-3 simulates an MCP4725 DAC device using a Teensy 3.2 module. The Teensy 3.2 has an on-board, 12-bit DAC connected to pin A14. Writing a value between 0x000 and 0xfff produces a voltage between 0 V and +3.3 V on that pin. The code in Listing 8-3 associates `rcvISR` (and ISR) with the data received interrupt. When data arrives, the system automatically calls this routine and passes it the number of bytes received on the I²C bus.

The `rcvISR` interrupt service routine (ISR) fetches the bytes transmitted to the peripheral from the controller, constructs the 12-bit DAC output value from those bytes, and then writes the 12 bits to the DAC output (using the Arduino `analogWrite()` function). Once the output is complete, the code waits for the next transmission to occur. Just like a debug and test feature, this program writes a string to the `Serial` output every 10 seconds so you can verify that the program is still running.

```
// Listing8-3.ino
//
// This program demonstrates using an
// Arduino as an I2C peripheral.
//
// This code runs on a Teensy 3.2
// module. A14 on the Teensy 3.2 is
// a true 12-bit, 3.3-V DAC. This program
// turns the Teensy 3.2 into a simple
// version of the MCP4725 DAC. It reads
// inputs from the I2C line (corresponding
// to an MCP4725 fast write operation)
// and writes the 12-bit data to the
// Teensy 3.2's hardware DAC on pin A14.

#include <Wire.h>

// I2C address of the SparkFun MCP4725 I2C-based
// digital-to-analog converter.

#define MCP4725_ADDR 0x60

// Interrupt handler that the system
// automatically calls when data arrives
// on the I2C lines.

void rcvISR( int numBytes )
{
    byte LObyte;
    byte HObyte;
    word DACvalue;
```

```
    // Expecting 2 bytes to come
    // from the controller device.

    if( numBytes == 2 && Wire.available() )
    {
        HObyte = Wire.read();
        if( Wire.available() )
        {
            LObyte = Wire.read();

            DACvalue = ((HObyte << 8) | LObyte) & 0xfff;
            analogWrite( A14, DACvalue );
        }
    }
}

// Usual Arduino initialization function:

void setup( void )
{
    Serial.begin( 9600 );
    delay( 1000 );
    Serial.println( "I2C peripheral test" );

    // Initialize the Wire library to treat this
    // code as an I2C peripheral at address 0x60
    // (the SparkFun MCP4725 breakout board):

    Wire.begin( MCP4725_ADDR );

    // Set up the Teensy 3.2 DAC to have
    // 12-bit resolution:

    analogWriteResolution(12);

    // Define the I2C interrupt handler
    // for dealing with incoming I2C
    // packets:

    Wire.onReceive( rcvISR );
}

void loop( void )
{
   Serial.println( "MCP4725 emulator, waiting for data" );
   delay( 10000 ); // Delay 10 seconds

}
```

I connected the SCL, SDA, and Gnd pins of two Teensy 3.2 devices together (using a Teensy and an Arduino also works). On one of the units, I programmed the DAC output code similar to that found in Listing 8-1. On the other, I programmed the code in Listing 8-3. I put an oscilloscope on the A14 pin on the Teensy running the peripheral code (Listing 8-3). The

output appears in Figure 8-3. Note that the peaks on the triangle waves are between 0.0 V and 3.3 V (rather than 0 V and 5 V in Figure 8-2) because the Teensy is a 3.3-V device.

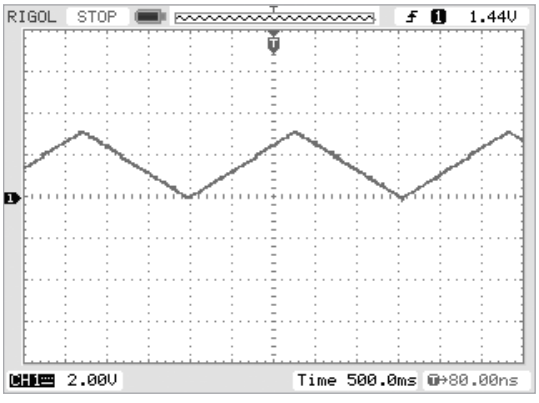

Figure 8-3: Triangle wave output from the Teensy 3.2 A14 pin

Figure 8-4 shows a small section of the output when some clock stretching occurs.

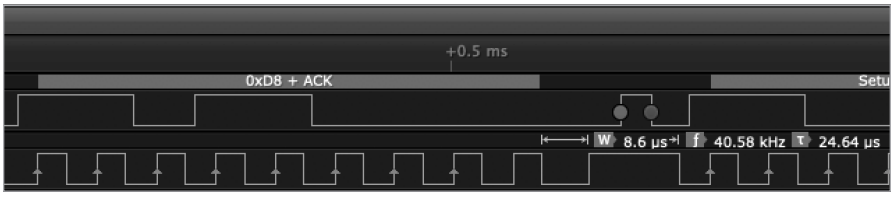

Figure 8-4: A stretched clock signal reduces the triangle wave frequency.

As you can see in Figure 8-4, the clock is stretched to 8.4 µsec after the transmission of the byte.

8.5 Multiple I²C Port Programming

The standard Arduino library assumes that only a single I²C bus is on the board (based on the hardware of the Arduino Uno). Many Arduino-compatible boards provide multiple I²C buses. This allows you to spread your I²C devices across multiple buses, allowing them to run faster, or to, perhaps, include two devices with the same address without having to resort to using an I²C bus multiplexer.

The standard Arduino library does not support multiple I²C buses; however, devices that do provide them will often provide some special library code that lets you access the additional I²C buses in the system. The Arduino convention when there are multiple instances of a device is to use a numeric suffix after the name to designate a particular device. In the case of the I²C bus, those device names are `Wire` (for the first, or 0th, port), `Wire1`, `Wire2`, and so on.

For example, to write a sequence of bytes to the second I²C port, you might use code like the following:

```
Wire1.beginTransmission( 0x60 );
Wire1.write( (dacOut << 8) & 0xf );
Wire1.write( dacOut & 0xff );
Wire1.endTransmission( true );
```

The mechanism for achieving this is hardware and system specific. Check the documentation for your particular SBC to see how this is done.

8.6 Chapter Summary

The Arduino library provides the `Wire` object to support I²C bus transactions. This chapter described the basic `Wire` functions available in the Arduino library, including those to initialize the I²C library, choose the I²C clock frequency, initiate a read from an I²C peripheral, read peripheral data placed in the internal buffer, initialize a buffer for transmission to a peripheral, and more.

This chapter also included several real-world examples of I²C communication using the SparkFun MCP4725.

FOR MORE INFORMATION

To learn more about Wire programming on the Arduino, you should first stop at the Arduino Wire library reference page: *https://www.arduino.cc/en/Reference/Wire*.

Of course, you can find about a bazillion different websites with Arduino I²C programming examples. A quick web search for "Arduino I²C examples" will probably turn up more hits than you are willing to read.

Simon Monk's book, *Programming Arduino Next Steps: Going Further with Sketches,* 2nd edition (McGraw-Hill Education TAB, 2018), contains a chapter on Arduino I²C programming.

Adafruit tutorials on Arduino I²C programming: *https://learn.adafruit.com/circuitpython-basics-i2c-and-spi/i2c-devices*

SparkFun I²C tutorials on Arduino I²C programming: *https://learn.sparkfun.com/tutorials/i2c/all*

Datasheets for the MCP4725 DAC: *https://cdn-shop.adafruit.com/datasheets/mcp4725.pdf*

More information about the Teensy 3.2: *https://www.pjrc.com/store/teensy32.html*