Application Information

3D Linear or 2D Angle Sensing with ALS31300 and ALS31313 Hall-Effect ICs

By Wade Bussing and Robert Bate,
Allegro MicroSystems

Abstract

This application note describes the use of the ALS31300 and ALS31313 3D linear Hall-effect sensor integrated circuits (IC) from Allegro MicroSystems for 3D linear sensing and 2D angle sensing applications. References throughout this application note to the ALS31300 also apply for the ALS31313, except that the ALS31300 is provided in a 10-contact DFN package, and the ALS31313 is provided in a TSSOP-8 package.

Detailed examples include converting register contents to gauss for linear sensing, and combining data from two axes to calculate an angle for rotational angle sensing. Other sections describe the process of reading and writing registers on the ALS31300 via the I²C interface, application schematics, and the associated Arduino example code. See Appendix A for full source code, including an Arduino .ino sketch file. The Arduino .ino sketch file is also available on Allegro's Software Portal.

Introduction

The ALS31300 3D linear Hall-effect sensor IC provides users with an accurate, low-cost solution for non-contact linear and angular position sensing. With its I²C interface, the ALS31300 provides convenient access to angle and linear information from multiple sensors on a single bus (see Figure 1).

Examples listed in this application note make use of the "Teensy" 3.2 microcontroller (https://www.pjrc.com/teensy/teensy31.html) and Arduino (https://www.arduino.cc/) software environment. While this document focuses on implementation using the Teensy 3.2, the practices and example code translate directly to other Arduino boards.

I²C Overview

The I²C bus is a synchronous, two-wire serial communication protocol which provides a full-duplex interface between two or more devices. The bus specifies two logic signals:

1. Serial Clock Line (SCL) output by the Master.
2. Serial Data Line (SDA) output by either the Master or the Slave.

The block diagram shown below in Figure 1 illustrates the I²C bus topology.

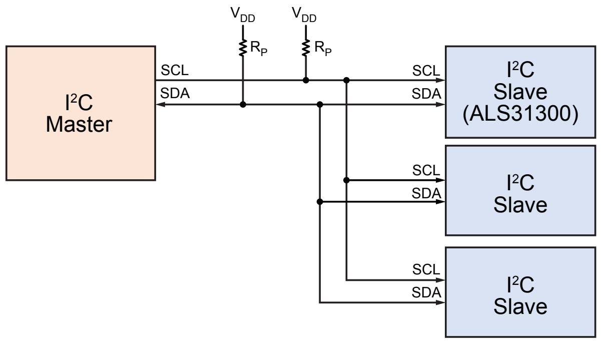

Figure 1: I²C bus diagram showing Master and Slave devices

Data Transmission

The transmission of data over I^{2}C is composed of several steps outlined in the sequence below.

1. Start Condition: Defined by a negative edge of the SDA line, initiated by the Master, while SCL is high.
2. Address Cycle: 7-bit Slave address, plus 1 bit to indicate write (0) or read (1), followed by an Acknowledge bit.
3. Data Cycles: Reading or writing 8 bits of data, followed by an Acknowledge bit. This cycle can be repeated for multiple bytes of data transfer. The first data byte on a write could be the register address. See the following sections for further information.
4. Stop Condition: Defined by a positive edge on the SDA line, while SCL is high.

Except to indicate Start or Stop conditions, SDA must remain stable while the clock signal is high. SDA may only change states while SCL is low. It is acceptable for a Start or Stop condition to occur at any time during the data transfer. The ALS31300 will always respond to a Read or Write request by resetting the data transfer sequence.

The clock signal SCL is generated by the Master, while the SDA line functions as either an input or open-drain output, depending on the direction of data transfer. Timing of the I^{2}C bus is summarized in the timing diagram in Figure 2. Signal references and definitions of these names can be found in the ALS31300 datasheet.

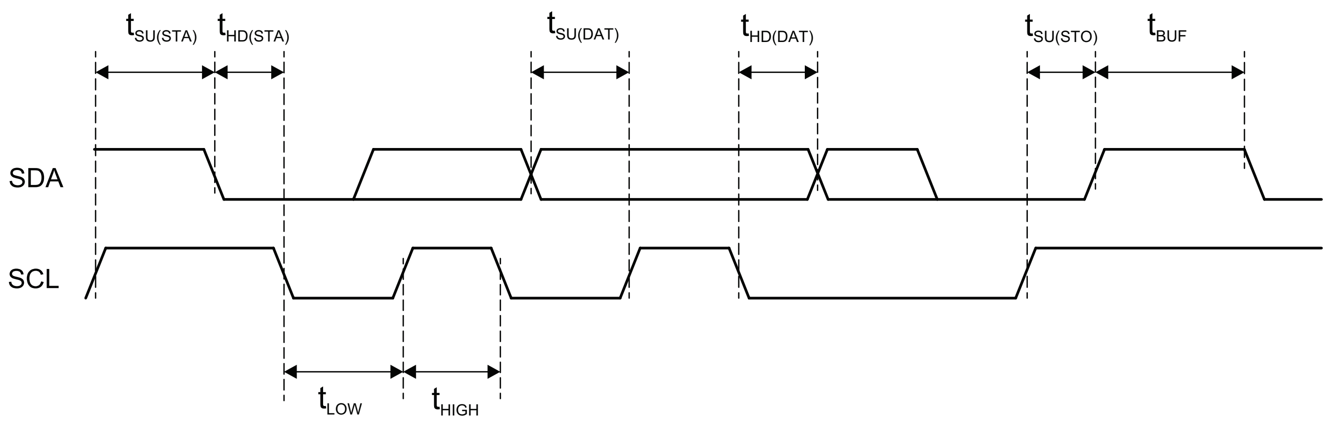

Figure 2: I^{2}C Input and Output Timing Diagram

I^{2}C Bus Speeds

Common I^{2}C bus speeds are 100 kbps *standard mode* and the 10 kbps *low-speed mode*, but arbitrarily low clock frequencies are also allowed. Recent revisions of the I^{2}C protocol can host more nodes and run at faster speeds including 400 kbps *fast mode* and 1 Mbps *fast mode plus* (Fm+) which are all supported by the ALS31300. Note the spec outlines an additional 3.4 Mbps *high-speed mode* that is not supported by ALS31300.

Implementation of I^{2}C with the ALS31300

The ALS31300 may only operate as a Slave I^{2}C device, therefore it cannot initiate any transactions on the I^{2}C bus.

The ALS31300 will always respond to a Read or Write request by resetting the data transfer sequence. The state of the Read/Write bit is set low (0) to indicate a Write cycle and set high (1) to indicate a Read cycle. The Master monitors for an Acknowledge bit to confirm the Slave device (ALS31300) is responding to the address byte sent by the Master. When the ALS31300 decodes the 7-bit Slave address as valid, it responds by pulling SDA low during the ninth clock cycle. When a data write is requested by the Master, the ALS31300 pulls SDA low during the clock cycle, following the data byte to indicate that the data has been successfully received. After sending either an address byte or a data byte, the Master must release the SDA line before the ninth clock cycle, allowing the handshake process to occur.

The default slave address for the ALS31300 is 110xxxx, where the four LSB bits are set by applying different voltages to the address pins ADR0 and ADR1. Both address pins have been set to ground for this demonstration as shown by the schematic in Figure 11. For information on selecting other I^{2}C Slave addresses, refer to the ALS31300 datasheet. With both pins grounded, the default I^{2}C Slave address is 96.

Write Cycle Overview

The write cycle to access registers on the ALS31300 are outlined in the sequence below.

1. Master initiates Start Condition
2. Master sends 7-bit slave address and the write bit (0)
3. Master waits for ACK from ALS31300
4. Master sends 8-bit register address
5. Master waits for ACK from ALS31300
6. Master sends 31:24 bits of data
7. Master waits for ACK from ALS31300
8. Master sends 23:16 bits of data
9. Master waits for ACK from ALS31300
10. Master sends 15:8 bits of data
11. Master waits for ACK from ALS31300
12. Master sends 7:0 bits of data
13. Master waits for ACK from ALS31300
14. Master initiates Stop Condition

The I²C write sequence is further illustrated in the timing diagrams below in Figure 3.

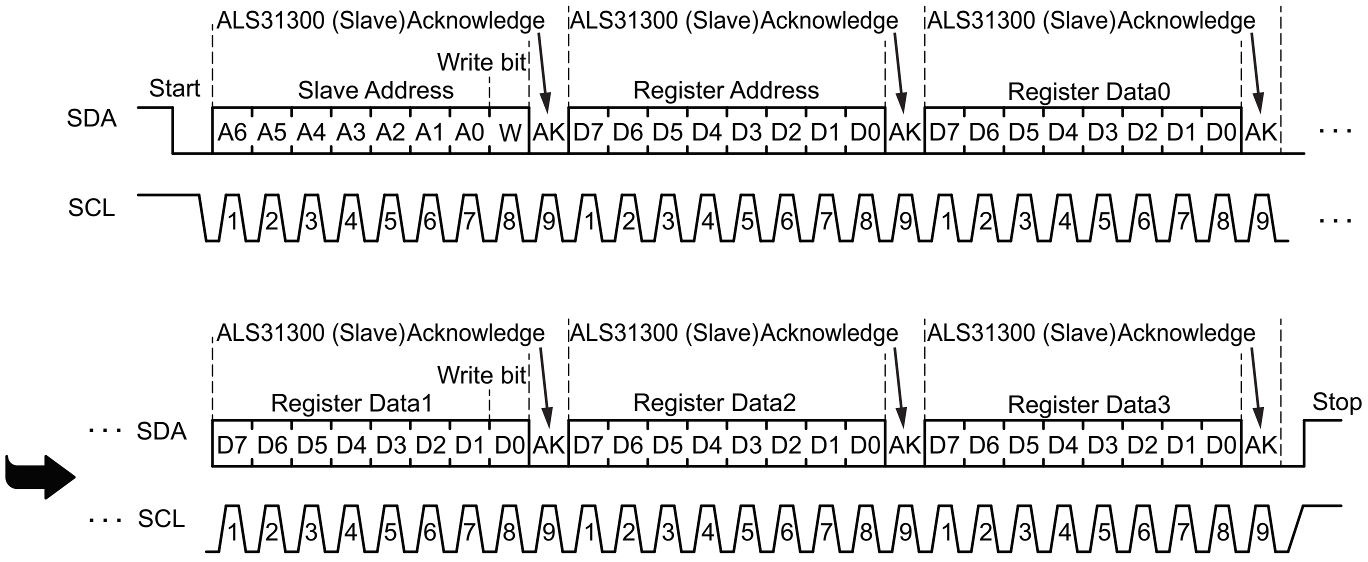

Figure 3: I²C Write Timing Diagram

Customer Write Access

An access code must be sent to the device prior to writing any of the volatile registers or EEPROM in the ALS31300. If customer access mode is not enabled, then no writes to the device are allowed. The only exception to this rule is the SLEEP bit, which can be written regardless of the access mode. Furthermore, any register or EEPROM location can be read at any time regardless of the access mode.

To enter customer access mode, an access command must be sent via the I²C interface. The command consists of a serial write operation with the address and data values shown in Table 1. There is no time limit for when the code may be entered. Once the customer access mode is entered, it is not possible to change access modes without powercycling the device.

Table 1: Customer Access Code

Access Mode	Address	Data
Customer Access	0x35	0x2C413534

Read Cycle Overview

The read cycle to access registers on ALS31300 is outlined in the sequence below.

1. Master initiates Start Condition
2. Master sends 7-bit slave address and the write bit (0)
3. Master waits for ACK from ALS31300
4. Master sends 8-bit register address
5. Master waits for ACK from ALS31300
6. Initiate a Start Condition. This time it is referred to as a Re-start Condition
7. Master sends 7-bit slave address and the read bit (1)
8. Master waits for ACK from ALS31300
9. Master receives 31:24 bits of data
10. Master sends ACK to ALS31300
11. Master receives 23:16 bits of data
12. Master sends ACK to ALS31300
13. Master receives 15:8 bits of data
14. Master sends ACK to ALS31300
15. Master receives 7:0 bits of data
16. Master sends NACK to ALS31300
17. Master initiates Stop Condition

The I²C read sequence is further illustrated in the timing diagrams below in Figure 4.

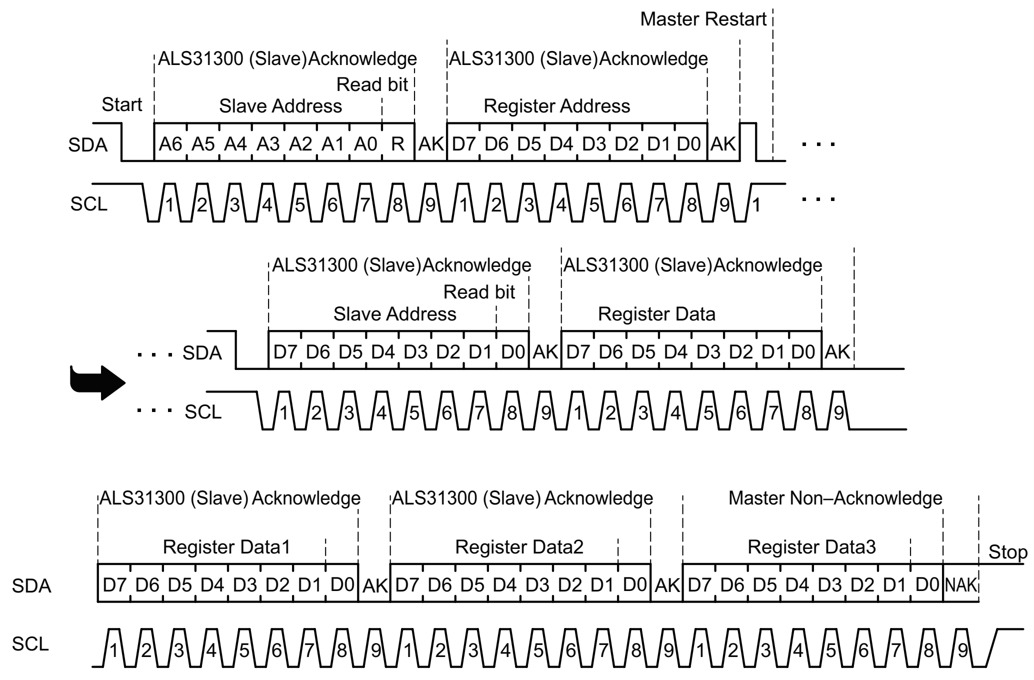

Figure 4: I²C Read Timing Diagram

The timing diagram in Figure 4 shows the entire contents (bits 31:0) of a single register location being transmitted. Optionally, the I²C Master may choose to replace the NACK with an ACK instead, which allows the read sequence to continue. This case will result in the transfer of contents (bits 31:24) from the following register, address + 1. The master can then continue acknowledging, or issue the not-acknowledge (NACK) or stop after any byte to stop receiving data.

Note that only the initial register address is required for reads, allowing for faster data retrieval. However, this restricts data retrieval to sequential registers when using a single read command. When the Master provides a not-acknowledge bit and stop bit, the ALS31300 stops sending data. If non-sequential registers are to be read, separate read commands must be sent.

I²C Readback Modes for X, Y, Z and Temperature Data

The ALS31300 I²C controller has several modes to make the process of repeatedly polling X, Y, Z and Temperature data convenient. These options include Single Mode, Fast Loop Mode and Full Loop Mode.

Single Mode

A single write or read command to any register—this is the default mode and is best suited for setting fields and reading static registers. If desired, this mode can be used to read X, Y, Z and Temperature data in a typical serial fashion, but fast or full loop read modes are recommended for high-speed data retrieval.

Fast Loop Mode

Fast Loop Mode offers continuous reading of X, Y, Z and temperature values, but is limited to the upper 8 bits of X, Y, Z and upper 6 bits of Temperature. This mode is intended to be a time-efficient way of reading data from the IC at the expense of truncating resolution. The flow chart in Figure 5 depicts Fast Loop Mode.

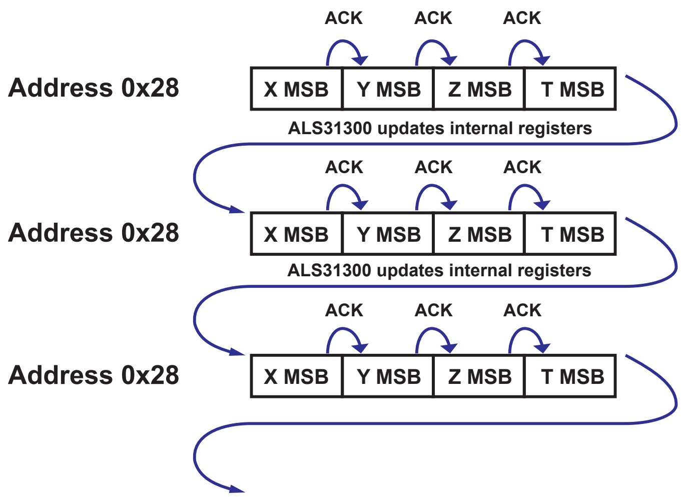

Figure 5: Fast Loop Mode

Full Loop Mode

Full Loop Mode provides continuous reads of X, Y, Z and Temperature data with full, 12 bit resolution. This is the recommended mode for users wanting a higher data rate for X, Y, Z and Temperature with full resolution. The flow chart in Figure 6 depicts Full Loop Mode.

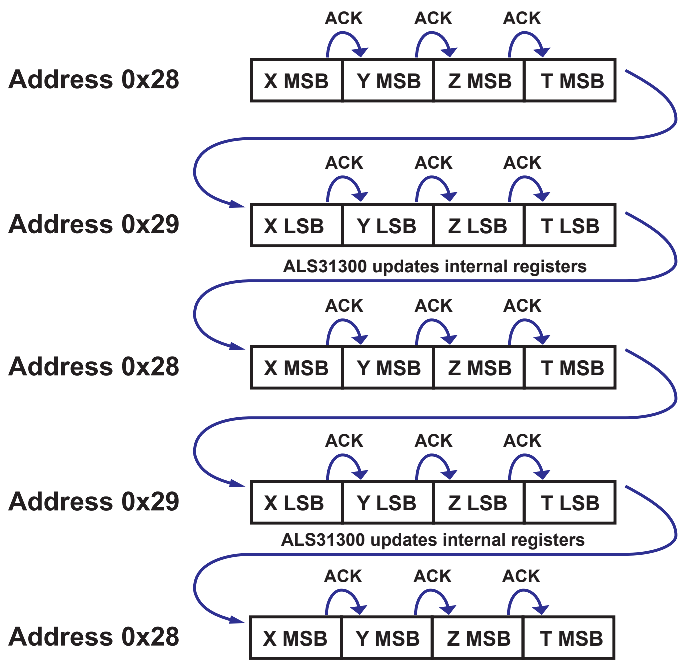

Figure 6: Full Loop Mode

The Looping modes are further described below in Table 2.

Table 2: ALS31300 Looping Read Modes

Code (Binary)	Mode	Description
00	Single	No Looping. Similar to Default I²C.
01	Fast Loop	X, Y, Z and Temperature fields are looped. 8 MSBs for X, Y, and Z; 6 MSBs for Temperature are looped.
10	Full Loop	X, Y, Z and Temperature fields are looped. Full, 12-bit resolution fields are looped.
11	Single	Same as code 0.

To set the read loop mode, set bits 3:2 at address 0x27 to the desired code value per Table 2.

Magnetic Field Strength Registers

The magnetic field strength registers contain data that is proportional to the measured magnetic field in each of the three axes as seen by the ALS31300. The register addresses and bit fields for X, Y, and Z magnetic data are described in Table 3. The orientations of X, Y and Z Axes are defined in Figure 7.

The MSBs and LSBs of each axis must be concatenated to resolve full 12-bit magnetic field data. Refer to Appendix A for example code of various techniques to poll and concatenate the magnetic data from ALS31300.

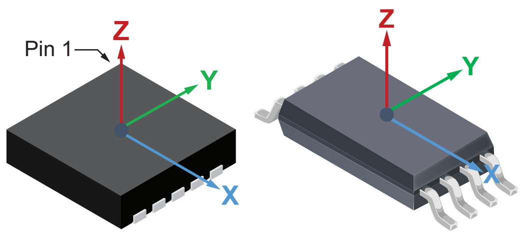

Figure 7: Magnetic Axes of ALS31300 DFN Package (left) and ALS31313 TSSOP Package (right) (not to scale)

Table 3: Magnetic Field Strength Registers

Address	Bits	Name	Description	R/W
0x28	31:24	X Axis MSBs	8 bit signal proportional to upper 8 bits of field strength in X direction.	R
	23:16	Y Axis MSBs	8 bit signal proportional to upper 8 bits of field strength in Y direction.	R
	15:8	Z Axis MSBs	8 bit signal proportional to upper 8 bits of field strength in Z direction.	R
0x29	19:16	X Axis LSBs	4 bit signal proportional to lower 4 bits of field strength in X direction.	R
	15:12	Y Axis LSBs	4 bit signal proportional to lower 4 bits of field strength in Y direction.	R
	11:8	Z Axis LSBs	4 bit signal proportional to lower 4 bits of field strength in Z direction.	R

Temperature Sensor Registers

Temperature Registers of ALS31300 are described below in Table 5.

Table 4: Temperature Registers

Address	Bits	Name	Description	R/W
0x28	5:0	Temperature MSBs	6 bit signal proportional to upper 6 bits of temperature.	R
0x29	5:0	Temperature LSBs	6 bit signal proportional to lower 6 bits of temperature.	R

Calculating Measured Field

For this example the full-scale range of ALS31300 is 500 gauss, giving a sensitivity of 4 LSB/gauss.

The process begins with a full 8 byte read of MSB and LSB registers to construct a 12 bit, 2's complement signed value. All data must be read in a single 8 byte read when combining registers, or the result will be the combination of two separate samples in time. The 12 bits of data are combined per Table 5.

Table 5: MSB and LSB Combination Data

BIT	11	10	9	8	7	6	5	4	3	2	1	0
DATA	MSB Data								LSB Data			

Assume a full 8 byte read returns the following binary data for a single axis:

MSB = 1100_0000

LSB = 0110.

The combined data {MSB; LSB} = 1100_0000_0110. The decimal equivalent = –1018, which can be converted to gauss by dividing by the device sensitivity (4 LSB/gauss).

$$gauss = -1018\ LSB \div 4\ LSB/G = -254\ gauss$$

Calculating Angle Using Two Axes

The angle of an applied field can be calculated using magnetic data from two axes of the ALS31300 and a four quadrant arc tangent function. For this example, a disc magnet is magnetized diametrically. The drawings in Figure 8 show reference orientations of puck magnets and their poles compared to the X, Y, and Z axes of the ALS31300. In the left orientation, the magnet is rotating around the Z axis, as indicated by the white arrow, while sensing magnetics with X and Y. In the orientation on the right, the magnet is rotated about the Y axis, while using X and Z channels to sense. The third orientation can be used with a magnet rotating around the X axis and sensing with Y and Z.

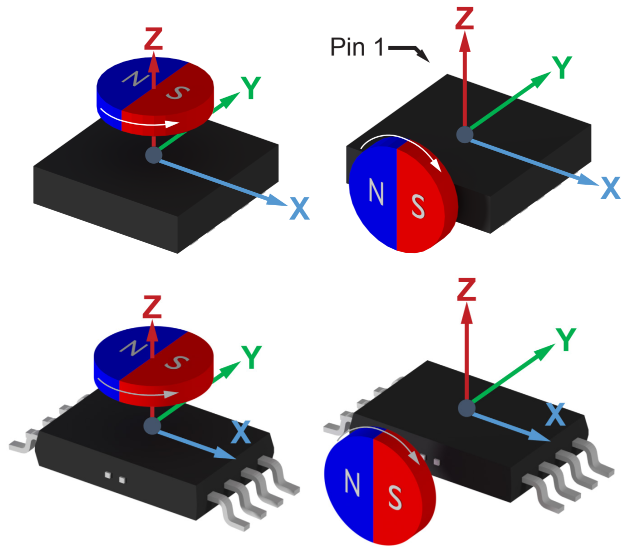

Figure 8: Diametric magnets and signal axes of the ALS31300 (top) and ALS31313 (bottom)

Standard inverse tangent functions, i.e. tan()$^{-1}$, return angle values ranging from –90° to 90°. For this application, it is important to use a four quadrant arc tangent function to return an angle from –180° to 180°. This function also avoids issues with dividing by 0. Four quadrant inverse tangent functions are listed in Table 6.

Table 6: Four Quadrant Arc Tan Function Calls

Program	Function	Description
MATLAB	atan2(Y,X)	4 quadrant tan^{-1}. Result in radians.
	atan2d(Y,X)	4 quadrant tan^{-1}. Result in degrees.
ARDUINO	atan2(Y,X)	4 quadrant tan^{-1}. Returns double.
	atan2f(Y,X)	4 quadrant tan^{-1}. Returns float.
C#	Atan2(Y,X)	4 quadrant tan^{-1}. Returns double.

Refer to Appendix A for full Arduino source code on calculating angle for XY, XZ, and YZ axes combinations.

The conversion process can be summarized into 3 major steps, which are listed below and identified on the scope plot in Figure 9. "Single" mode (Table 2) is used to simplify this example.

1. Read request is initiated by the Master.
2. Transmission of 8 bytes of data from the Slave.
3. Conversion of magnetic vector data to an angle value.

The read request (box 1) consists of a write to the device indicating which registers will be read. The device responds (box 2) with 8 bytes of data, (8 MSB of X, Y, Z and 6 MSB of temperature, followed by 4 LSB of X, Y, Z and 6 MSB of temperature).

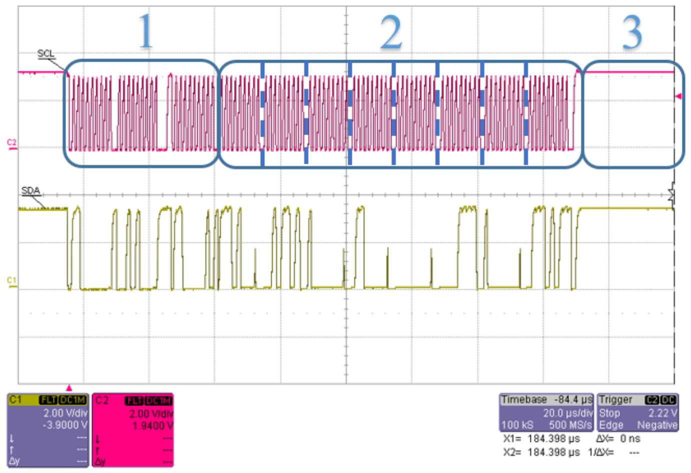

Figure 9: I^{2}C read of 8 data bytes in No loop mode. Registers 0x28 and 0x29.

Angle Calculation Timing

The total time to complete an angle calculation using the ALS31300 will be application specific, but predominantly dominated by the processing abilities and speed of the user's microcontroller. Other factors include the loop mode of the ALS31300 (Table 2) and communication frequency of the I^{2}C interface. Timing examples in this document assume a Teensy 3.2 microcontroller running at 72 MHz and configured I^{2}C communication speed of 1 MHz (Fast Mode+). Note that the Teensy 3.2 Fast Mode+ I^{2}C mode operates around 720 kHz.

The example in Figure 9 is a simple way to read data from the ALS31300, but it is not the fastest. The overhead of initiating a read (Box 1 in Figure 9) can be eliminated after the first request through the use of loop modes on the ALS31300.

The scope plot in Figure 10 shows the angle conversion flow with ALS31300 set in Full Loop Mode. The boxes 1, 2, and 3 still correspond to the same steps from Figure 9.

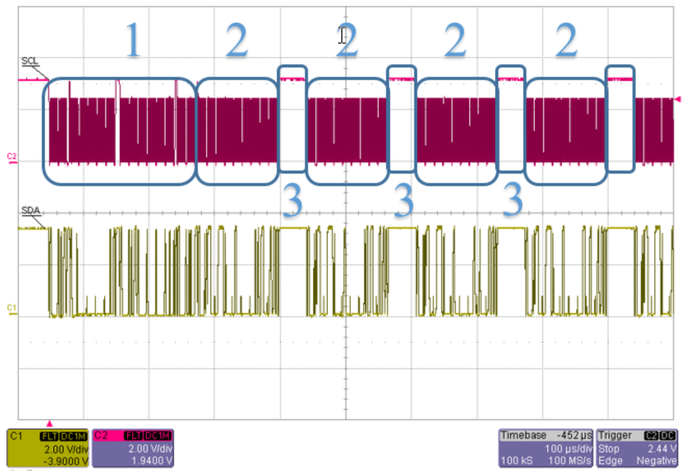

Figure 10: I²C read of 8 data bytes in Full Loop Mode. Registers 0x28 and 0x29.

Notice that box 1 occurs only once, but is slightly longer than the No Loop example in Figure 9. In Full Loop Mode, the read request consists of a write to the device indicating which register will be read, followed by a read/write to set full loop mode. Refer to full source code in Appendix A on how to implement no loop, fast loop and full loop read modes.

The repeated pause in box 3 shows the time it takes Teensy 3.2 microcontroller to perform the atan2f(x,y) function. The average duration of the atan2f(x,y) function on the Teensy 3.2 at 72 MHz is 30 µs, while the transmit time of 8 data bytes is 120 µs. Using the Teensy 3.2 and ALS31300 in Full Loop Mode, a fresh angle value can be calculated every 150 µs.

Application Schematics

Refer to the image in Figure 11 showing the application schematic used for the ALS31300 throughout this document.

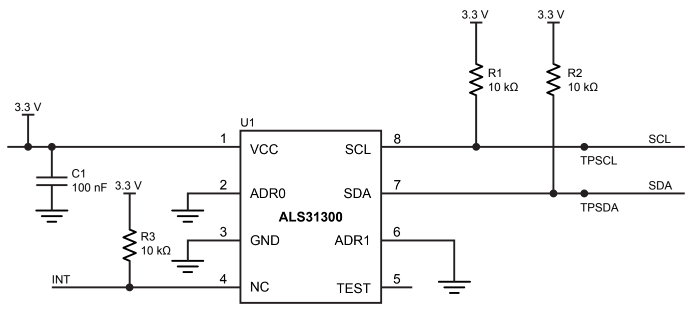

Figure 11: Application Schematic for ALS31300

Supporting circuitry for the Teensy 3.2 microcontroller is shown in the schematic in Figure 12.

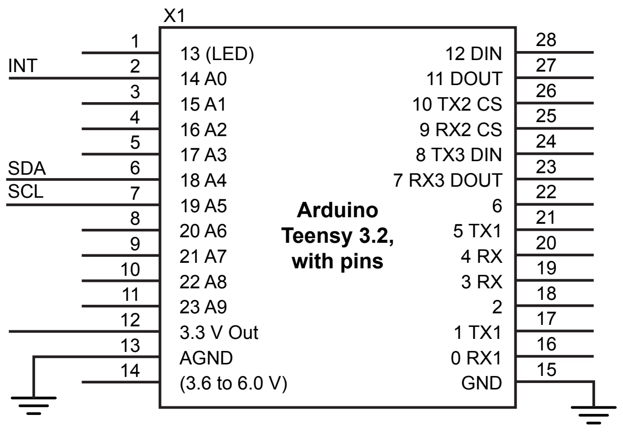

Figure 12: Teensy 3.2 Application Schematic

Refer to the nets labeled "SDA" and "SCL" in Figure 11 and Figure 12 indicating these connections between the two schematics. Note, the pin locations of SDA and SCL on the Teensy micro are user-selectable, but must be declared in software. Refer to the source code in Appendix A where the SDA and SCL pins are declared.

Conclusion

The ALS31300 and ALS31313 are a highly versatile, micropower 3D Hall-effect sensor ICs. The IC can be used for multi-axis linear position, or angular position sensing applications, and can be configured for operation in high resolution (12 bit) or medium resolution (8 bit) mode. The I²C bus is highly configurable, and can operate at bus speeds from 1 Mbps down to < 10 kbps, with a pullup voltage range of 1.8 to 3.3 V. The IC also includes a temperature sensor that can read over the I²C interface.

The Arduino .ino sketch file used with this application note is available on Allegro's Software Portal. Register for the "ALS31300" device to view the source code.

The ALS31300 datasheet is available at https://www.allegromicro.com/en/products/sense/linear-and-angular-position/linear-position-sensor-ics/als31300.

The ALS31313 datasheet is available at https://www.allegromicro.com/en/products/sense/linear-and-angular-position/linear-position-sensor-ics/als31313.

APPENDIX A: Full Arduino Source Code for ALS31300 and Teensy 3.2

The snippet below shows full Arduino source code used alongside this application. Example functions include I^{2}C initialization, reading from the ALS31300 in single, fast, and full loop modes, writing data to the ALS31300 using I^{2}C and calculating angle and gauss using magnetic data from the ALS31300.

The full .ino Arduino sketch is available on Allegro Microsystem's Software portal under the ALS31300 device tab. To register with Allegro's software portal and view the ALS31300 source code, visit https://registration.allegromicro.com/login.

```
/*
 *    Example source code for an Arduino to show
 *    how to communicate with an Allegro ALS31300
 *
 *    Written by K. Robert Bate, Allegro MicroSystems, LLC.
 *
 *    ALS31300Demo is distributed in the hope that it will be useful,
 *    but WITHOUT ANY WARRANTY; without even the implied warranty of
 *    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 */
#include <Wire.h>
#include <math.h>

// Return values of endTransmission in the Wire library
#define kNOERROR 0
#define kDATATOOLONGERROR 1
#define kRECEIVEDNACKONADDRESSERROR 2
#define kRECEIVEDNACKONDATAERROR 3
#define kOTHERERROR 4

// led blinking support
bool ledState = false;
int ledPin = 13;
unsigned long nextTime;

int deviceAddress = 96; // Address of the ALS31300
// SCL Pin = 19
// SDA Pin = 18

//
// setup
//
// Initializes the Wire library for I2C communications,
// Serial for displaying the results and error messages,
// the hardware and variables to blink the LED,
// and sets the ALS31300 into customer access mode.
//
void setup()
{
    // Initialize the I2C communication library
    Wire.begin();
    Wire.setClock(1000000);    // 1 MHz

    // Initialize the serial port
    Serial.begin(115200);
    // If using a Arduino with USB built in, uncomment the next line,
    // this allows the errors in Setup to be seen
    // while (!Serial);

    // Setup hardware and variables for code which blinks the LED
    nextTime = millis();
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, LOW);

    // Enter customer access mode on the ALS31300
    uint16_t error = write(deviceAddress, 0x24, 0x2C413534);

    if (error != kNOERROR)
    {
        Serial.print("Error while trying to enter customer access mode. error = ");
        Serial.println(error);
    }
}

// loop
//
// Every half second, read the ADCs of the ALS31300 and display
// the values and toggle the state of the LED.
//
void loop()
{
    // only perform the reading of the ALS31300 and the
    // toggling the state of the LED every half second
    if (nextTime < millis())
    {
        nextTime = millis() + 500L;

        // Uncomment which reading style is desired
        readALS31300ADC(deviceAddress);
        //readALS31300ADC_FastLoop(deviceAddress);
        //readALS31300ADC_FullLoop(deviceAddress);

        // Blink the LED
        ledState = !ledState;
        digitalWrite(ledPin, ledState);
    }
}

//
// readALS31300ADC
//
// Read the X, Y, Z values from Register 0x28 and 0x29
// eight times. No loop mode is used.
//
void readALS31300ADC(int busAddress)
{
    uint32_t value0x27;

    // Read the register the I2C loop mode is in
    uint16_t error = read(busAddress, 0x27, value0x27);
    if (error != kNOERROR)
    {
        Serial.print("Unable to read the ALS31300. error = ");
        Serial.println(error);
    }

    // I2C loop mode is in bits 2 and 3 so mask them out
    // and set them to the no loop mode
    value0x27 = (value0x27 & 0xFFFFFFF3) | (0x0 << 2);

    // Write the new values to the register the I2C loop mode is in
    error = write(busAddress, 0x27, value0x27);
```

```
if (error != kNOERROR)
{
    Serial.print("Unable to read the ALS31300. error = ");
    Serial.println(error);
}

for (int count = 0; count < 8; ++count)
{
    // Write the address that is going to be read from the ALS31300
    Wire.beginTransmission(busAddress);
    Wire.write(0x28);
    uint16_t error = Wire.endTransmission(false);

    // The ALS31300 accepted the address
    if (error == kNOERROR)
    {
        // Start the read and request 8 bytes
        // which are the contents of register 0x28 and 0x29
        Wire.requestFrom(busAddress, 8);

        // Read the first 4 bytes which are the contents of register 0x28
        uint32_t value0x28 = Wire.read() << 24;
        value0x28 += Wire.read() << 16;
        value0x28 += Wire.read() << 8;
        value0x28 += Wire.read();

        // Read the next 4 bytes which are the contents of register 0x29
        uint32_t value0x29 = Wire.read() << 24;
        value0x29 += Wire.read() << 16;
        value0x29 += Wire.read() << 8;
        value0x29 += Wire.read();

        // Take the most significant byte of each axis from register 0x28 and combine it with the least
        // significant 4 bits of each axis from register 0x29, then sign extend the 12th bit.
        int x = SignExtendBitfield(((value0x28 >> 20) & 0x0FF0) | ((value0x29 >> 16) & 0x0F), 12);
        int y = SignExtendBitfield(((value0x28 >> 12) & 0x0FF0) | ((value0x29 >> 12) & 0x0F), 12);
        int z = SignExtendBitfield(((value0x28 >> 4) & 0x0FF0) | ((value0x29 >> 8) & 0x0F), 12);

        // Display the values of x, y and z
        Serial.print("Count, X, Y, Z = ");
        Serial.print(count);
        Serial.print(", ");
        Serial.print(x);
        Serial.print(", ");
        Serial.print(y);
        Serial.print(", ");
        Serial.println(z);

        // Look at the datasheet for the sensitivity of the part used.
        // In this case, full scale range is 500 gauss, other sensitivities
        // are 1000 gauss and 2000 gauss.
        // Sensitivity of 500 gauss = 4.0 lsb/g
        // Sensitivity of 1000 gauss = 2.0 lsb/g
        // Sensitivity of 2000 gauss = 1.0 lsb/g

        float mx = (float)x / 4.0;
        float my = (float)y / 4.0;
        float mz = (float)z / 4.0;

        Serial.print("MX, MY, MZ = ");
```

```
            Serial.print(mx);
            Serial.print(“, “);
            Serial.print(my);
            Serial.print(“, “);
            Serial.print(mz);
            Serial.println(“ Gauss”);

            // Convert the X, Y and Z values into radians
            float rx = (float)x / 4096.0 * M_TWOPI;
            float ry = (float)y / 4096.0 * M_TWOPI;
            float rz = (float)z / 4096.0 * M_TWOPI;

            // Use a four quadrant Arc Tan to convert 2
            // axis to an angle (which is in radians) then
            // convert the angle from radians to degrees
            // for display.
            float angleXY = atan2f(ry, rx) * 180.0 / M_PI;
            float angleXZ = atan2f(rz, rx) * 180.0 / M_PI;
            float angleYZ = atan2f(rz, ry) * 180.0 / M_PI;

            Serial.print(“angleXY, angleXZ, angleYZ = “);
            Serial.print(angleXY);
            Serial.print(“, “);
            Serial.print(angleXZ);
            Serial.print(“, “);
            Serial.print(angleYZ);
            Serial.println(“ Degrees”);
        }
        else
        {
            Serial.print(“Unable to read the ALS31300. error = “);
            Serial.println(error);
            break;
        }
    }
}

//
// readALS31300ADC_FastLoop
//
// Read the X, Y, Z 8 bit values from Register 0x28
// eight times quickly using the fast loop mode.
//
void readALS31300ADC_FastLoop(int busAddress)
{
    uint32_t value0x27;

    // Read the register the I2C loop mode is in
    uint16_t error = read(busAddress, 0x27, value0x27);
    if (error != kNOERROR)
    {
        Serial.print(“Unable to read the ALS31300. error = “);
        Serial.println(error);
    }

    // I2C loop mode is in bits 2 and 3 so mask them out
    // and set them to the fast loop mode
    value0x27 = (value0x27 & 0xFFFFFFF3) | (0x1 << 2);

    // Write the new values to the register the I2C loop mode is in
```

```
error = write(busAddress, 0x27, value0x27);
if (error != kNOERROR)
{
    Serial.print("Unable to read the ALS31300. error = ");
    Serial.println(error);
}

// Write the register address that is going to be read from the ALS31300 (0x28)
Wire.beginTransmission(busAddress);
Wire.write(0x28);
error = Wire.endTransmission(false);

// The ALS31300 accepted the address
if (error == kNOERROR)
{
    int x;
    int y;
    int z;

    // Eight times is arbitrary, there is no limit. What is being demonstrated
    // is that once the address is set to 0x28, all reads will be from 0x28 until the
    // register address is changed or the loop mode is changed.
    for (int count = 0; count < 8; ++count)
    {
        // Start the read and request 4 bytes
        // which is the contents of register 0x28
        Wire.requestFrom(busAddress, 4);

        // Read the first 4 bytes which are the contents of register 0x28
        // and sign extend the 8th bit
        x = SignExtendBitfield(Wire.read(), 8);
        y = SignExtendBitfield(Wire.read(), 8);
        z = SignExtendBitfield(Wire.read(), 8);
        Wire.read();    // Temperature and flags not used

        // Display the values of x, y and z
        Serial.print("Count, X, Y, Z = ");
        Serial.print(count);
        Serial.print(", ");
        Serial.print(x);
        Serial.print(", ");
        Serial.print(y);
        Serial.print(", ");
        Serial.println(z);

        // Convert the X, Y and Z values into radians
        float rx = (float)x / 256.0 * M_TWOPI;
        float ry = (float)y / 256.0 * M_TWOPI;
        float rz = (float)z / 256.0 * M_TWOPI;

        // Use a four quadrant Arc Tan to convert 2
        // axis to an angle (which is in radians) then
        // convert the angle from radians to degrees
        // for display.
        float angleXY = atan2f(ry, rx) * 180.0 / M_PI;
        float angleXZ = atan2f(rz, rx) * 180.0 / M_PI;
        float angleYZ = atan2f(rz, ry) * 180.0 / M_PI;

        Serial.print("angleXY, angleXZ, angleYZ = ");
        Serial.print(angleXY);
```

```
            Serial.print(“, “);
            Serial.print(angleXZ);
            Serial.print(“, “);
            Serial.print(angleYZ);
            Serial.println(“ Degrees”);
        }
    }
    else
    {
        Serial.print(“Unable to read the ALS31300. error = “);
        Serial.println(error);
    }
}

//
// readALS31300ADC_FullLoop
//
// Read the X, Y, Z 12 bit values from Register 0x28 and 0x29
// eight times quickly using the full loop mode.
//
void readALS31300ADC_FullLoop(int busAddress)
{
    uint32_t value0x27;

    // Read the register the I2C loop mode is in
    uint16_t error = read(busAddress, 0x27, value0x27);
    if (error != kNOERROR)
    {
        Serial.print(“Unable to read the ALS31300. error = “);
        Serial.println(error);
    }

    // I2C loop mode is in bits 2 and 3 so mask them out
    // and set them to the full loop mode
    value0x27 = (value0x27 & 0xFFFFFFF3) | (0x2 << 2);

    // Write the new values to the register the I2C loop mode is in
    error = write(busAddress, 0x27, value0x27);
    if (error != kNOERROR)
    {
        Serial.print(“Unable to read the ALS31300. error = “);
        Serial.println(error);
    }

    // Write the address that is going to be read from the ALS31300
    Wire.beginTransmission(busAddress);
    Wire.write(0x28);
    error = Wire.endTransmission(false);

    // The ALS31300 accepted the address
    if (error == kNOERROR)
    {
        int x;
        int y;
        int z;

        // Eight times is arbitrary, there is no limit. What is being demonstrated
        // is that once the address is set to 0x28, the first four bytes read will be from 0x28
        // and the next four will be from 0x29 after that it starts all over at 0x28
        // until the register address is changed or the loop mode is changed.
    for (int count = 0; count < 8; ++count)
    {
        // Start the read and request 8 bytes
        // which is the contents of register 0x28 and 0x29
        Wire.requestFrom(busAddress, 8);

        // Read the first 4 bytes which are the contents of register 0x28
        x = Wire.read() << 4;
        y = Wire.read() << 4;
        z = Wire.read() << 4;
        Wire.read();    // Temperature and flags not used

        // Read the next 4 bytes which are the contents of register 0x29
        Wire.read();    // Upper byte not used
        x |= Wire.read() & 0x0F;
        byte d = Wire.read();
        y |= (d >> 4) & 0x0F;
        z |= d & 0x0F;
        Wire.read();    // Temperature not used

        // Sign extend the 12th bit for x, y and z.
        x = SignExtendBitfield((uint32_t)x, 12);
        y = SignExtendBitfield((uint32_t)y, 12);
        z = SignExtendBitfield((uint32_t)z, 12);

        // Display the values of x, y and z
        Serial.print("Count, X, Y, Z = ");
        Serial.print(count);
        Serial.print(", ");
        Serial.print(x);
        Serial.print(", ");
        Serial.print(y);
        Serial.print(", ");
        Serial.println(z);

        // Convert the X, Y and Z values into radians
        float rx = (float)x / 4096.0 * M_TWOPI;
        float ry = (float)y / 4096.0 * M_TWOPI;
        float rz = (float)z / 4096.0 * M_TWOPI;

        // Use a four quadrant Arc Tan to convert 2
        // axis to an angle (which is in radians) then
        // convert the angle from radians to degrees
        // for display.
        float angleXY = atan2f(ry, rx) * 180.0 / M_PI;
        float angleXZ = atan2f(rz, rx) * 180.0 / M_PI;
        float angleYZ = atan2f(rz, ry) * 180.0 / M_PI;

        Serial.print("angleXY, angleXZ, angleYZ = ");
        Serial.print(angleXY);
        Serial.print(", ");
        Serial.print(angleXZ);
        Serial.print(", ");
        Serial.print(angleYZ);
        Serial.println(" Degrees");
    }
}
else
{
    Serial.print("Unable to read the ALS31300. error = ");
```

```
        Serial.println(error);
    }
}

//
// read
//
// Using I2C, read 32 bits of data from the address on the device at the bus address
//
uint16_t read(int busAddress, uint8_t address, uint32_t& value)
{
    // Write the address that is to be read to the device
    Wire.beginTransmission(busAddress);
    Wire.write(address);
    int error = Wire.endTransmission(false);

    // if the device accepted the address,
    // request 4 bytes from the device
    // and then read them, MSB first
    if (error == kNOERROR)
    {
        Wire.requestFrom(busAddress, 4);
        value = Wire.read() << 24;
        value += Wire.read() << 16;
        value += Wire.read() << 8;
        value += Wire.read();
    }

    return error;
}

//
// write
//
// Using I2C, write 32 bit data to an address to the device at the bus address
//
uint16_t write(int busAddress, uint8_t address, uint32_t value)
{
    // Write the address that is to be written to the device
    // and then the 4 bytes of data, MSB first
    Wire.beginTransmission(busAddress);
    Wire.write(address);
    Wire.write((byte)(value >> 24));
    Wire.write((byte)(value >> 16));
    Wire.write((byte)(value >> 8));
    Wire.write((byte)(value));
    return Wire.endTransmission();
}

//
// SignExtendBitfield
//
// Sign extend a right justified value
//
long SignExtendBitfield(uint32_t data, int width)
{
    long x = (long)data;
    long mask = 1L << (width - 1);

    if (width < 32)
    {
        x = x & ((1 << width) - 1); // make sure the upper bits are zero
    }

    return (long)((x ^ mask) - mask);
}
```

APPENDIX B: TEENSY PINOUT FLYER

The pinout flyer below ships with each Teensy 3.2. It is also available from PJRS at the following link: https://www.pjrc.com/teensy/card7a_rev1.pdf.

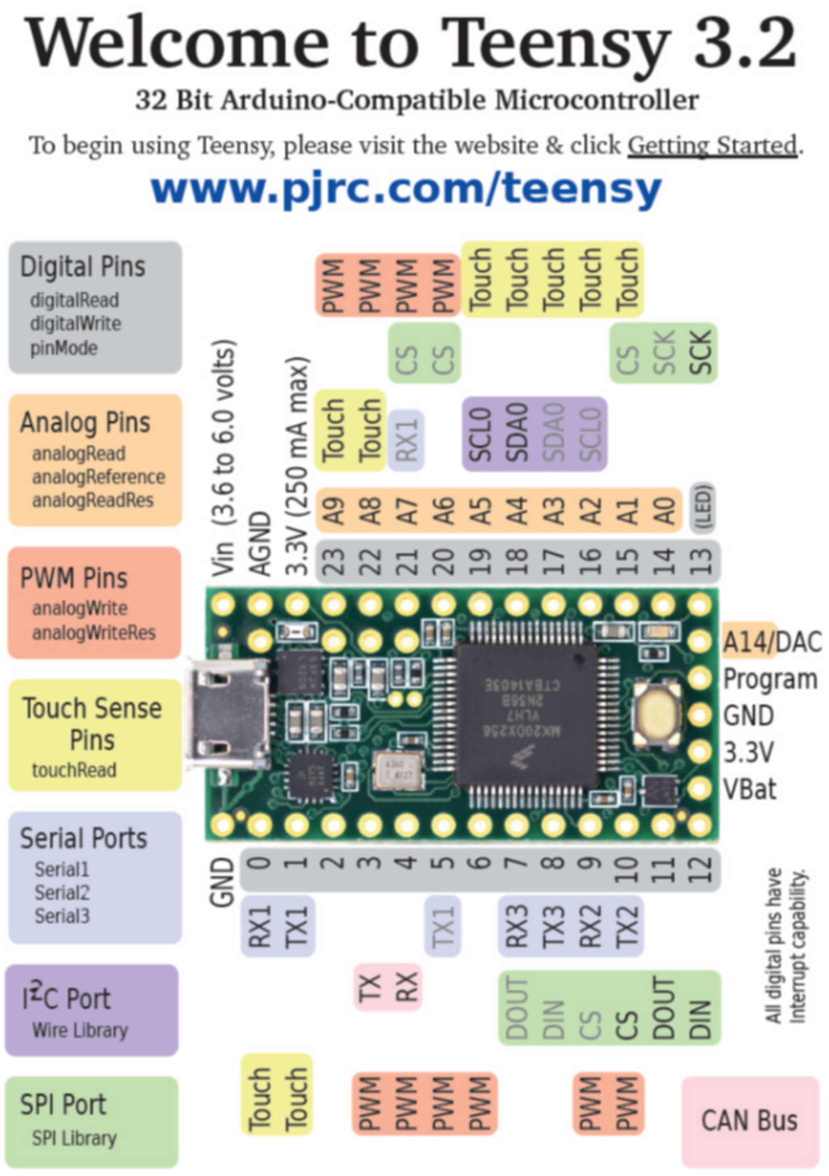

Figure 13: Pinout Flyer for TEENSY 3.2

Revision History

Number	Date	Description
–	July 31, 2017	Initial release
1	August 8, 2018	Added ALS31313 part references; updated Customer Access Code address (page 3).
2	April 1, 2019	Updated Write Cycle Overview section (page 3) and Read Cycle Overview section (page 4).
3	April 10, 2020	Minor editorial updates
4	March 27, 2024	Fixed broken links (page 8)