Lorenz Attractor

Bryan Takemoto

Goal:
Program the Atmega328 to provide basic support for floating-point operations. Then connect a MCP4725 DAC to the I2C port of the Atmega328. Finally, the MCP4725 will output a meaningful voltage to the oscilloscope which is interpreted as the solution to a floating-point problem.

Deliverables:
The purpose of this project is to demonstrate floating-point capability of the Atmega328 by computing the Lorenz attractor which is a non-linear ordinary differential equation proposed by Edward Lorenz. The solution will then be visually graphed onto the oscilloscope via MCP4725. Because of the limitation of the oscilloscope, the graph will be in 2D (XY) even though the Lorenz attractor is 3D (XYZ).

I. Literature survey

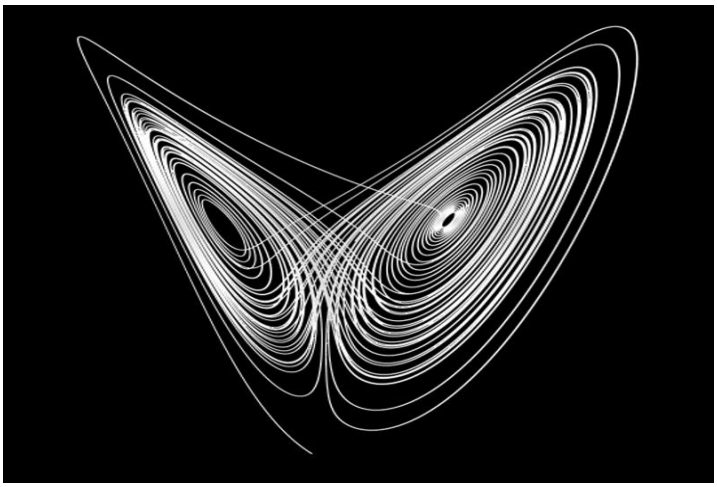
Mathematical computation has been the primary motivation for the development of processors. A lot of mathematical computation will most likely require a MCU [microcontroller] to evaluate an equation where data is fed into its input and outputs a solution to the user. Every MCU can compute an equation that accepts integer values and outputs an integer solution. But these are simple, ubiquitous functions expected on any ordinary MCU. Unfortunately, there are plethora of problems that require fractional values. Neglecting these fractional parts will return unwanted solutions to the user. In the case of this project, the Lorenz attractor is a system of equations that requires precise solutions, thus truncating fractional parts are unacceptable.

The Lorenz attractor is a unique equation because it exhibits a nonlinear behavior. The nonlinear property is defined as a system's output change will not be proportional to the change of its input. Furthermore, Lorenz attractor is famously known for its chaotic set of solutions. The chaotic property of the Lorenz attractor causes the system to be extremely sensitive to any change of its initial condition. In other words, any loss in accuracy such as truncating the fractional part during the computation will lead to inevitable errors with values that will either explode into massive numbers or deadlocks into a pattern (programming Lorenz attractor using integral datatype only). The Lorenz attractor appears in some models for lasers, dynamos, thermosyphons, brushless DC motors, electric circuits, chemical reactions, and osmosis. For Edward Lorenz, a meteorologist, the attractor was derived as model for convection in the earth's atmosphere. The following derivatives are the Lorenz equations:

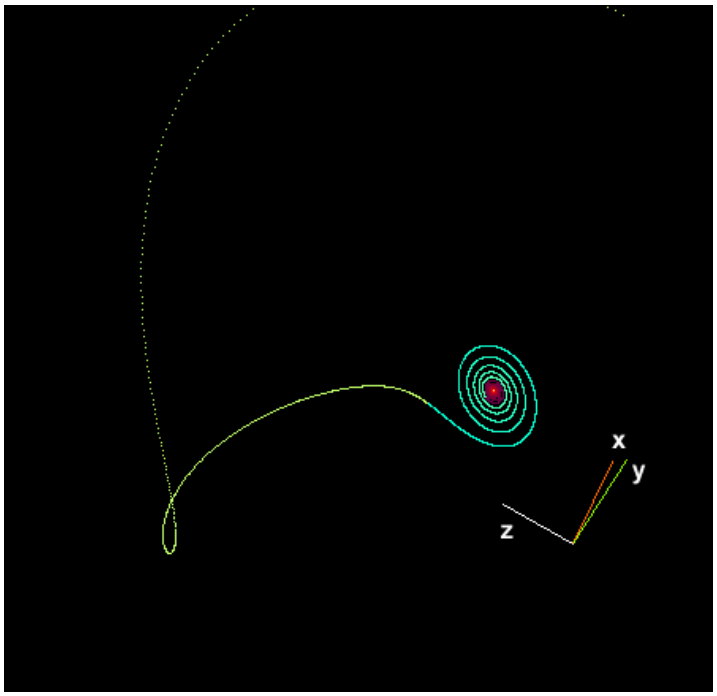
$$\frac{dx}{dt} = \sigma\,(y - x)$$
$$\frac{dy}{dt} = x\,(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

The constants σ, ρ, and β are the parameters that will affect the graph. Edward Lorenz used the following values: σ = 10, ρ = 28, and β = 8/3. These values should trace out a butterfly shaped graph on a 3D Cartesian.

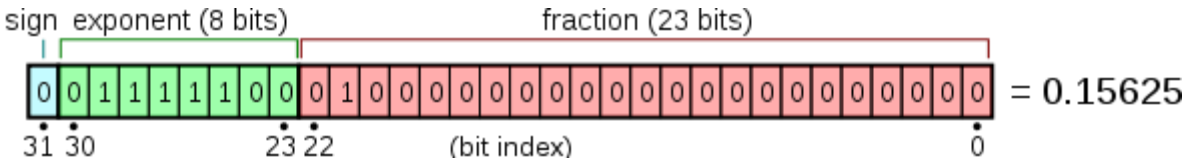
coordinate as shown below.



Note that the predefined constants guarantee a chaotic solution while others may or may not produce one. The constant of interest is ρ [rho] or also known as the Rayleigh number. Adjusting ρ will affect the chaos of the trace. Small values of ρ will most likely produce a stable system and trace out a pitchfork bifurcation as shown below instead of the butterfly as shown before.



To implement the Lorenz attractor, IEEE 754 single-precision will be used to represent a 32-bit floating-point value. The choice of 32-bit resolution is sufficient to prevent the Lorenz attractor from exploding. Theoretically, using 64-bit IEEE double-precision is a more logical choice to reduce the errors that a 32-bit single-precision has. The issue with 64-bit is that there is a lot more overhead compared to 32-bit, thus isn't feasible for an 8-bit MCU. The following diagram represents the fields for a 32-bit floating-point value.



There are three fields: sign, exponent, and fraction [mantissa]. Explaining each field and how to convert a decimal into floating-point numbers will take several pages which is beyond the scope of this report. But, doing a simple Internet search on IEEE 754 single-precision will offer numerous tutorials. It should be noted that arithmetic operations used in this project will be addition, subtraction, and multiplication. Addition and subtraction are straightforward to implement but multiplication adds the most overhead to the Atmega328 with several different algorithms available. Multiplication comes into play when two mantissas are multiplied during a floating-point multiplication. A naïve solution is to brute force the computation by looping a running sum variable. Brute force is only acceptable if the multiplication is trivial such as 8-bit multiplicand and multiplier. Unfortunately for this situation, 25-bit multiplication is heavy for brute force (potentially over 33,000,000 clock cycles if values are large enough). The algorithm of choice for this project will be the Booth multiplication. With this algorithm, in the worst-case scenario is no longer bounded linearly but now constantly because Booth multiplication only shifts a fixed number of times for every computation (depending on how many bits for either multiplicand or multiplier).

Once the float-pointing functions are implemented into the Atmgea328, the MCP4725 will be interfaced via I2C to the Atmega328. The MCP4725 is an economical DAC (Digital to Analog Converter) with a 12-bit resolution [0 – 4095]. Using an MCP4725 with an oscilloscope, the MCP4725 will graph the coordinates of the Lorenz attractor onto the oscilloscope. Since the Lorenz attractor also has negative values, an H-bridge circuit is required to reverse the polarity. H-bridge circuit is a preferable choice to represent negative values, but a more convenient solution is to use 2048 as the reference point of 0. Anything below 2048 is negative and above is positive. Since the Lorenz attractor is a 3D graph, two MCP4725s are required to graph with any choice of the two axes (XY is used for this case). It is possible to program two MCP4725s on a single bus using I2C but because the MCP4725 package has a preset built-in address, it isn't possible unless modification to both MCP4725s are done by soldering and cutting out the pull-up resistors. Instead of soldering and again for convenience, another Atmega328 and MCP4725 is added to the circuit. A simple inter-process communication via ports will synchronize both Atmega328s and MCP4725s to work in unison (one for X and other for Y).

II. Components

A. *Atmega328*

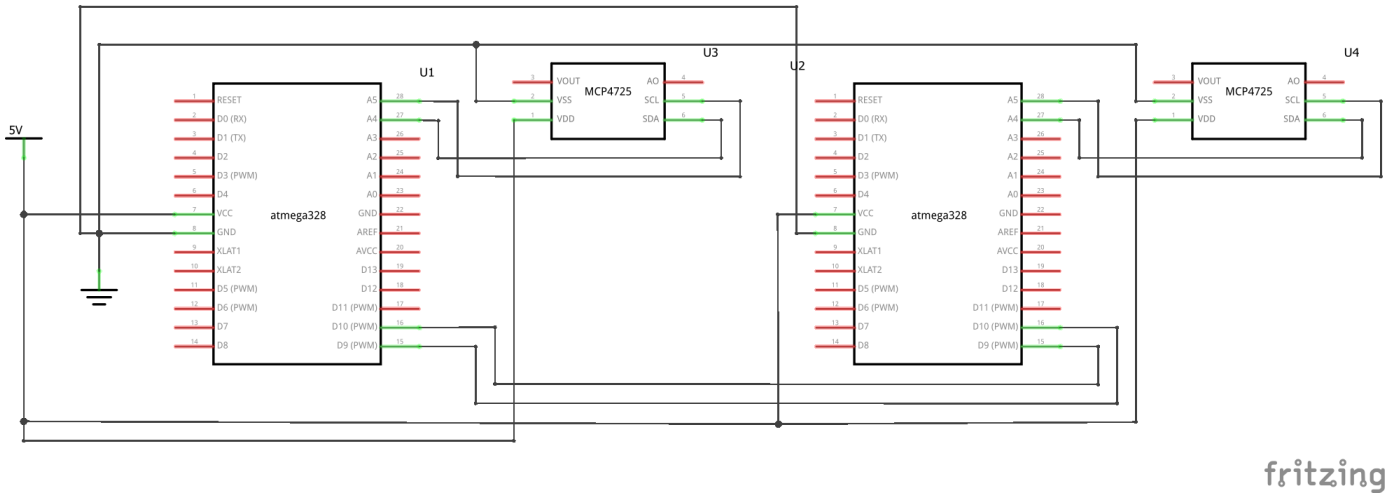
Atmega328 is part of a family of AVR microcontrollers developed by Atmel. These AVR microcontrollers use a modified Harvard architecture 8-bit RISC. The Atmega328 has 32 general purpose working registers, 32 KB programmable flash, 1 KB EEPROM, 2 KB SRAM, 23 general purpose I/O, counters/timers, ADC, PWM, USART, I2C, Watchdog Timer, and SPI support. For this project, only the I2C and two I/O pins are used.

B. *MCP4725*

The MCP4725 is a low-power, high accuracy 12-bit DAC with a built-in EEPROM. It is programmed via I2C interface command. A 12-bit value is written to the MCP4725's register to adjust the output voltage. Writing to the EEPROM is optional if the user wishes to save the last value written to the MCP4725 before powered off. Note that writing to EEPROM is significantly slower than just writing to the registers of the MCP4725. Therefore, avoid writing to the EEPROM for optimal operations. The output of the voltage depends on what 12-bit value written to the register and the reference voltage. Caution must be exercised

when picking a reference voltage to prevent damage to the MCP4725 or misinterpret the values. MCP4725 accepts a range of 2.7V to 5.5V but preferred values are 3.3V or 5V. If the voltage source is stable, using 3.3V will give you the most precise result but for this project, 5V is used instead. There are three speeds that the MCP4725 can operate: standard (100kbps), fast (400 kbps), and high-speed (3.4 Mbps). The MCP4725 can interchange between normal and power-down mode.

III. Schematics



IV. Implementation

- Atmega328
  - Implement 32-bit floating-point functions: addition, subtraction, and multiplication
  - Implement I2C functions to interact with the MCP4725
  - Implement a function that converts 32-bit floating-point into a binary value for the MCP4725 to output (since the values are small, scale it up at least by multiplying 32)
  - First Atmega328 should be program to output X and second Atmega328 for Y.
- MCP4725
  - Libraries for I2C can be found on the Internet but it is also simple to write
  - Reference voltage should be 5V
  - 2048 should be the reference point for 0
  - Connect output voltage to oscilloscope
  - Only write to the registers (EEPROM will slow down the Lorenz trace on oscilloscope)

V. Conclusion

The implementation of floating-point functions enables the ATmega328 to perform more complex operations. This allowed the ATmega328 to compute the Lorenz attractor consistently without producing large numbers or incorrect patterns. Once the solution is available, it can be converted into an analog value by using the MCP4725 DAC and graphed onto to the oscilloscope to verify the result. Ultimately, having floating-point functionality opens more opportunities for the application of ATmega328 with Lorenz attractor being one of the many possible usage.

VI. Code

Main:

```
#define F_CPU 16000000UL

#include <avr/io.h>
```

```
#include <stdint.h>
#include <util/delay.h>

// Operations [float_add]
#define SUB   1
#define ADD   0

// Adjust DAC prescalar output
#define SCALE 5 // 64

// Lorenz constants
#define SIGMA 0x41200000    // 10
#define RHO   0x41e00000    // 28
#define BETA  0x402aaaab    // 8/3 [2.666...]

// Lorenz derivatives
volatile uint32_t dt = 0x3c23d70a; // 0.01
volatile uint32_t dx = 0x00000000; // Cleared
volatile uint32_t dy = 0x00000000; // Cleared
volatile uint32_t dz = 0x00000000; // Cleared

// Coordinates
volatile uint32_t x = 0x3f800000;  // x = 1.0
volatile uint32_t y = 0x3f800000;  // y = 1.0
volatile uint32_t z = 0x3f800000;  // z = 1.0

// Floating Point Operators
uint32_t float_add(uint32_t A, uint32_t B, uint8_t OP);
uint32_t float_mult(uint32_t M, uint32_t R);
uint16_t float_digital(uint32_t Num);

// I2C Functions
void i2c_init(void);
void i2c_write(unsigned char data);
void i2c_start(void);
void i2c_stop(void);

void dac(unsigned int input);

volatile uint32_t j = 0;
volatile uint32_t k = 27;
volatile uint16_t num_out;

int main(void)
{
  uint32_t temp;    // Temporary variable
  uint32_t osc_out; // Value to output to the DAC
  uint16_t i;     // Iterative variable

  DDRB = (1 << 1);  // PB1

  i2c_init();     // Initialize the I2C

  _delay_ms(5000);
  while (1)
  {
    for(i = 0; i < 1000; i++) {
      // dx/dt = SIGMA(y-x)
      dx = float_add(y, x, SUB);
      dx = float_mult(SIGMA, dx);
      dx = float_mult(dx, dt);
```

```
      // dy/dt = x(RHO-z)-y
      dy = float_add(RHO, z, SUB);
      dy = float_mult(dy, x);
      dy = float_add(dy, y, SUB);
      dy = float_mult(dy, dt);

      // dz/dt = xy-(BETA)z
      temp = float_mult(x, y);
      dz = float_mult(BETA, z);
      dz = float_add(temp, dz, SUB);
      dz = float_mult(dz, dt);

      x = float_add(x, dx, ADD);   // Update the change to X
      y = float_add(y, dy, ADD);   // Update the change to Y
      z = float_add(z, dz, ADD);   // Update the change to Z

      osc_out = float_digital(x);  // Output X
      PORTB = (1 << 1);
      while((PINB & 0x4) != 0x4);
      PORTB = 0x0;
      dac(osc_out);
    }
  }
}
```

Function for floating-point addition/subtraction:

```
// Floating point addition/subtraction
uint32_t float_add(uint32_t A, uint32_t B, uint8_t OP)
{
  // Save function arguments
  uint32_t a = A;
  uint32_t b = B;
  uint8_t sub = OP;

  // Operand a
  uint8_t exp0 = a >> 23;            // Extract the exponent field of a
  uint32_t mant0 = (a & 0x007FFFFF) | 0x00800000; // Extract the mantissa field of a

  // Operand b
  uint8_t exp1 = b >> 23;            // Extract the exponent field of b
  uint32_t mant1 = (b & 0x007FFFFF) | 0x00800000; // Extract the mantissa field of b

  // Final result
  int16_t exp = 0;    // Final exponent
  uint32_t final = 0;// Result to be returned
  uint32_t mant = 0;   // Final mantissa

  // Temporary variables
  uint32_t temp = 0;

  // Check if the operation is subtraction
  if(sub > 0)
  b ^= 0x80000000;

  // Adjust and compute the exponent
  if(exp0 > exp1)          // exp(a) > exp(b)
  {
    temp = exp0 - exp1;       // shift amount = exp(a) - exp(b)
    exp = exp0;
    mant1 = mant1 >> temp;     // Adjust B
  }
```

```
  else if(exp1 > exp0)       // b > a
  {
    temp = exp1 - exp0;      // shift amount = exp(b) - exp(a)
    exp = exp1;
    mant0 = mant0 >> temp;     // Adjust A
  }
  else
  {
    exp = exp0;          // Equal exponents
  }

  // Same signs [a + b] or [(-a) + (-b)]
  if((a & 0x80000000) == (b & 0x80000000))
  {
    final |= (a & 0x80000000);   // Set sign
    mant = mant0 + mant1;    // Add the two mantissas

    // Normalize the mantissa
    if(mant > 0x00FFFFFF)
    {
      mant = mant >> 1;    // Shift mantissa to adjust the floating point
      exp += 1;         // Increment the exponent
    }
  }
  // Different signs
  else
  {
    // [a - b]
    if((b & 0x80000000) == 0x80000000)
    {
      // Two's complement
      mant1 = ~mant1;
      mant1 += 1;
      // Perform the subtraction
      mant = mant0 + mant1;
    }
    // [b - a]
    else
    {
      // Two's complement
      mant0 = ~mant0;
      mant0 += 1;
      // Perform the subtraction
      mant = mant0 + mant1;
    }

    // Check if the value is negative, if so, absolute value the mantissa and set sign bit to 1
    if(mant > 0x00FFFFFF)
    {
      mant = ~mant;
      mant += 1;
      // Set sign as negative
      final |= 0x80000000;
    }

    // Normalize the mantissa
    if(mant > 0)
    {
      while(mant < 0x00800000)
      {
        mant = mant << 1;
        exp -= 1;
```

```
      }
    }
    else
    exp = 0;
  }

  // Overflow case [Largest value]
  // Exponent cannot be larger than 254 [with bias of +127]
  if(exp > 254)
  {
    final |= 0x7FFFFFFF;
    return final;
  }
  // Underflow case [Smallest number]
  // Exponent cannot be smaller than 0 [with bias of +127]
  else if(exp < 0)
  {
    final &= 0x80000000;
    return final;
  }

  mant &= 0x007FFFFF;  // Remove implicit 1
  temp = exp;
  temp = temp << 23;   // Shift the exponent into the correct exponent field
  final |= temp;     // Insert exponent into final
  final |= mant;     // Insert mantissa into final

  return final;
}
```

Function for floating-point multiplication:

```
// Floating point multiplication
uint32_t float_mult(uint32_t M, uint32_t R)
{
  // Save function arguments
  uint32_t a = M;
  uint32_t b = R;

  // Operand a
  uint8_t exp0 = a >> 23;            // Extract the exponent field of a
  uint32_t mant0 = (a & 0x007FFFFF) | 0x00800000; // Extract the mantissa field of a

  // Operand b
  uint8_t exp1 = b >> 23;            // Extract the exponent field of b
  uint32_t mant1 = (b & 0x007FFFFF) | 0x00800000; // Extract the mantissa field of b

  // Final result
  int16_t exp = 0;   // Final exponent
  uint32_t final = 0;// Result to be returned

  // Booth Multiplier Variables
  uint8_t i = 0;
  uint64_t A = 0;
  uint64_t S = 0;
  uint64_t P = 0;

  // Compute the sign
  final |= (a & 0x80000000) ^ (b & 0x80000000); // Xor the sign bits

  // Compute the initial exponent
```

```
  exp = (exp0 + exp1) - 127;           // Add the exponents and subtract 127 [rid the redundant
bias]
  if(exp > 0x00FF)               // Early check for overflow
  return final |= (0x7FFFFFFF);

  // Multiply the mantissas using Booth's algorithm
  // m = mant0    x = 25 bits
  // r = mant1    y = 25 bits
  // A = S = P = x + y + 1 = 51 bits [USE LONG INT]

  // Fill register A with m
  A = mant0;
  A = A << 26;

  // Two's complement of m
  mant0 = ~mant0;
  mant0 += 1;

  // Fill register S with -m
  S = mant0;
  S = S << 26;
  S &= 0x0007FFFFFFFFFFFF;

  // Fill register P with r
  P = mant1 << 1;

  for(i = 0; i < 25; i++)
  {
    if((P & 3) == 1)
    {
      P += A;
    }
    else if((P & 3) == 2)
    {
      P += S;
    }
    P &= 0x0007FFFFFFFFFFFF;   // Mask out the overflow

    // Shift negative number
    if(P >= 0x0004000000000000)
    {
      P = P >> 1;
      P |= 0x0004000000000000;
    }
    // Shift positive number
    else
    P = P >> 1;
  }
  P = P >> 1;   // One last shift to complete the Booth algorithm

  // Normalize the result
  while(P > 0x00007FFFFFFFFFFF)
  {
    P = P >> 1;
    exp += 1;
  }
  while(P < 0x00003FFFFFFFFFFF && P > 0)
  {
    P = P << 1;
    exp -= 1;
  }
```

```
  // Truncate
  while(P >= 0x0000000001000000)
  P = P >> 1;

  // Overflow case [Largest value]
  // Exponent cannot be larger than 254 [with bias of +127]
  if(exp > 254)
  {
    final |= 0x7FFFFFFF;
    return final;
  }
  // Underflow case [Smallest number]
  // Exponent cannot be smaller than 0 [with bias of +127]
  else if(exp < 0)
  {
    final &= 0x80000000;
    return final;
  }

  mant0 = exp;
  mant0 = mant0 << 23;    // Extract the final exponent
  final |= mant0;       // Insert the exponent into final
  mant0 = P;
  mant0 &= 0x007FFFFF;    // Extract the final mantissa
  final |= mant0;       // Insert the mantissa into final
  return final;
}
```

Function to convert floating-point into a representable integer for MCP4725:

```
uint16_t float_digital(uint32_t Num)
{
  uint32_t a = Num;

  uint16_t final = 0;
  int8_t temp = (a >> 23) - 127;
  uint64_t mant = (a & 0x007FFFFF) | 0x00800000;

  // Denormalize the floating point
  while(temp != 0)
  {
    if(temp > 0)
    {
      temp--;
      mant = mant << 1;
    }
    else if(temp < 0)
    {
      temp++;
      mant = mant >> 1;
    }
  }
  mant = mant << SCALE;  // Scale the value

  mant = mant >> 23;
  final |= mant & 0x0000000000000FFF;

  if((a & 0x80000000) == 0x80000000)
  final = 0x07FF - final;  // Negative
  else
  final = 0x07FF + final;  // Positive
```

```
  return final;
}
```

Functions for I2C and MCP4725:

```
// DAC function
void dac(unsigned int input)
{
  i2c_start();
  i2c_write(0b11000000);
  i2c_write(0b01000000);
  i2c_write(input >> 4);
  i2c_write(input);
  i2c_stop();
}

// I2C initialization
void i2c_init(void)
{
  TWSR = 0x00;                    // Set prescalar to 0
  TWBR = 0x48;                    // SCL = 50 KHz (Fosc = 8 MHz)
  TWCR = (1 << TWEN);                 // Enable TWI
}

// Write to the I2C
void i2c_write(unsigned char data)
{
  TWDR = data;                    // Data to be transmitted
  TWCR = (1 << TWINT) | (1 << TWEN);         // Use TWI module and write
  while((TWCR & (1 << TWINT)) == 0);         // Wait for TWI to complete
}

// I2C start condition
void i2c_start(void)
{
  TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN); // Transmit START condition
  while((TWCR & (1 << TWINT)) == 0);         // Wait for TWI to complete
}

// I2C stop condition
void i2c_stop(void)
{
  TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWSTO); // Transmit STOP condition
}
```

REFERENCES

Atmgea328 Datasheet: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf
MCP4725 Datasheet: https://www.sparkfun.com/datasheets/BreakoutBoards/MCP4725.pdf
IEEE 754: https://en.wikipedia.org/wiki/Single-precision_floating-point_format
Floating-point Arithmetic: http://www.toves.org/books/float/
Lorenz System: https://en.wikipedia.org/wiki/Lorenz_system
Lorenz Attractor: http://paulbourke.net/fractals/lorenz/
Booth Multiplication: https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm