TITLE: MAX30105 USING I2C

GOAL:

- Implement max30105 sensor interfaced by I2C communication
- Receive data from the sensor and display on via Data Visualizer

DELIVERABLES:

The project is intended to implement the MAX30105 sensor using I2C in AVR in order to read, monitor, and display heart rate of the user.

LITERATURE SURVEY:

There are around 80 million households in the USA that own dogs or cats. This means ~67% of all households in the USA have a cat or dog. Unfortunately, approximately 7.6 million dogs and cats enter animal shelters, which mean they have wandered out of their homes. This leads to a problem for pet owners, especially ones with traditional pet doors. These pet doors allow the pet with access to the outside and back home. However, most pet doors do not have a method of tracking the whereabouts of the animals. Even simple notifications indicating whether the pet is inside or outside the house do not exist. With the use of an IMU and Wi-Fi module, we can design such a product that will allow pet owners to worry less and monitor their pet's whereabouts.

Heart rate monitors are monitoring devices that allows a user to measure their own heart rate in real time, or even record it for later examination. These days, it is most often used by people involved in various forms of physical exercise to maintain how "hard" they should exercise and to keep a certain pace to maximize their workout. However, there's a lot more to being able to read a heart rate rather than just keeping track of how many calories one loses. In the medical world, monitoring a heart rate allows for observation of any irregularities of a patient's pulse such as those who have atrial fibrillation. With the use of a microcontroller and the MAX30105 sensor, a person can design a simple heart rate monitor.

The MAX30105 can perform multiple functions such as a smoke detector, particle detection, or even a simple heart rate monitor. In this case, a MAX30105 sensor is used to create a basic heart rate monitor. This sensor allows a person to read their heart own heart rate in beats per minute, average beats per minutes, and possibly even more if given the functions. The way it works is that the sensor uses a built-in photon detector and its LEDs to detect reflected light which works well with something like oxygenated blood. Every time the blood reflects the light, it symbolizes a pulse and the sensor will send this data to the Atmega328p microcontroller where it performs several functions in order to calculate specific values such as the beats per minute. The microcontroller will then transmit and output the data to a terminal using USART.

The I2C/TWI interface is used to communicate with the MAX30105. This interface allows multiple devices to be connected via master-slave connections. The master is allowed to send data and request data and can communicate will all devices connected to the SDA/SCL line. Slaves can only communicate with the master. Communications occur when the SCL is high and

stops when a STOP signal is received. Communications with multiple devices is achieved by addressing each device by their device address (usually hardwired, sometimes can be re-written). If multiple devices with the same address are present, the first device to respond to the master's request will be communicated with. The device address consists of 7 bits, with the last bit representing either READ or WRITE operations. The I2C reads 9 bits, with the 9^{th} bit being a STOP bit. Once a device is contacted, the master, depending on the 8^{th} bit, can READ data from or WRITE to the slave.

The I2C/TWI interface is used to communicate with the MAX30105 sensor. This allows multiple devices to connect in master-slave communications. The master device is permitted to send and receive data and is capable of communicating with all devices that are connected to the SDA/SCL line. Slave devices are only able to communicate with the master device. If SCL is high, communications occur and will stop once given a stop signal. Multiple device communication is achieved only by addressing each device by their address. To explain in a simpler context, consider a classroom with a teacher and students. The teacher holds the authority and can speak as they wish. When they ask a question, and multiple students raise their hands, they call the student by their name and the student will provide an answer. The device address consists of seven bits. The last bit is reserved to represent whether it is a read or write operation. I2C reads nine bits in total. The eighth bit determines the master device reads data from or writes to the slave. The ninth and last bit represents the stop bit.

There are 2 versions of reading in I2C, acknowledged and not acknowledged. It is represented by the ninth bit as well. If it is set, the master will continue to read data from the slave. However, if not acknowledged, then the slave returns control to the master after stopping communication. Similarly, the write does the same only stopping when all bits are written to the register.

COMPONENTS:

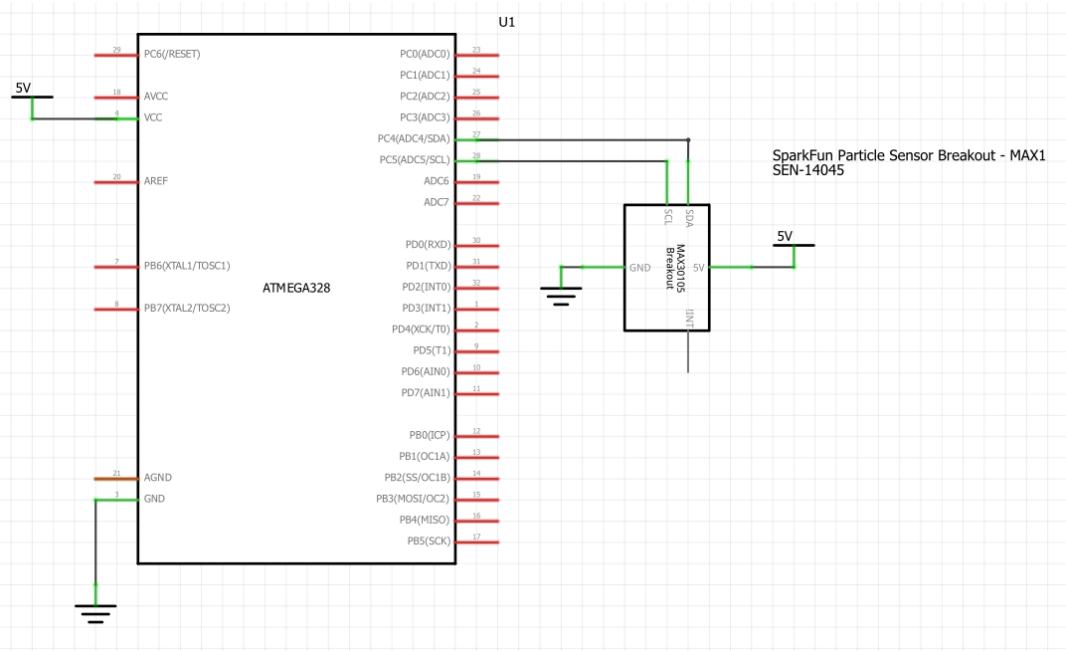
MAX30105 Sensor

The MAX30105 Particle Sensor is a sensor capable of sensing distance, heart rate, particle detecting, and even the blinking of an eye to name a few. It has 3 LEDs and a sensitive photon detector. The concept is to blink the various LEDs and detect whatever is reflected. Based on the reflective signature, it can detect and differentiating the various types of particles or materials such as oxygenated blood, or smoke.

ATmega328p

The ATmega328p is an 8 bit microcontroller by Atmel. This microcontroller works by programming in either C or assembly. The code written will allow the users to determine what pins they want to set as input and output. This microcontroller consists of 32 general purpose registers. The ATmega328p consists of 6 PWMs, 3 timer/counters, up to 1kB EEPROM, 2kB SDRAM, I2C interface, SPI interface, 10-bit ADCs, and an 8MHz clock. The ATmega328p, however, is limited in the amount of pins can be used as input and output. Also, only one 16 bit timer exists on the chip and only one USART module is included.

SCHEMATICS:



IMPLEMENTATION:

- MAX30105 interfaced via I2C to receive commands from the microcontroller and to send data back to the microcontroller
    - The I2C communication is implemented via a header file adapted from g4lvanix via Github.
    - USART and the built in FTDI module of the Atmega328p was used to ensure correct data was being transmitted.
- ATmega328p was used to link the MAX30105 Sensor with the data visualizer.
    - The ATmega328p read data from the sensor to perform calculations and transmit data to the data visualizer.
    - In order to perform the calculations, the necessary headers were translated from Arduino code into AVR C code.

SNAPSHOTS/SCREENSHOTS*: (only links - do not embed images or videos in the document)

Snapshot of physical implementation included in zip file of project.

MAX30105 Presentation Video:

https://www.youtube.com/watch?v=IcXRsYNxXWQ&list=PL1tZawtBXQFvtoDnjwl1pYpSboMNeCssi

MAX30105 Operation Video:

https://www.youtube.com/watch?v=L-zqoyLWR3Y&index=1&list=PL1tZawtBXQFvtoDnjwl1pYpSboMNeCssi

CODE:

Main

```
/*Begining of Auto generated code by Atmel studio */
#include <Arduino.h>

/*End of auto generated code by Atmel studio */

#include <Wire.h>
#include "MAX30105.h"
#include "heartRate.h"
//Beginning of Auto generated function prototypes by Atmel Studio
//End of Auto generated function prototypes by Atmel Studio
const byte RATE_SIZE = 4; //Increase this for more averaging. 4 is good.
byte rates[RATE_SIZE]; //Array of heart rates
byte rateSpot = 0;
long lastBeat = 0; //Time at which the last beat occurred

float beatsPerMinute;
int beatAvg;

MAX30105 particleSensor;

void setup()
{
  Serial.begin(9600);
  Serial.println("Initializing...");

  // Initialize sensor
  if (!particleSensor.begin(Wire, I2C_SPEED_FAST)) //Use default I2C port, 400kHz speed
  {
    Serial.println("MAX30105 was not found. Please check wiring/power. ");
    while (1);
  }

  //Setup to sense a nice looking saw tooth on the plotter
  byte ledBrightness = 0x1F; //Options: 0=Off to 255=50mA
  byte sampleAverage = 4; //Options: 1, 2, 4, 8, 16, 32 //4 on Ex. 5
  byte ledMode = 2; //Options: 1 = Red only, 2 = Red + IR, 3 = Red + IR + Green
  int sampleRate = 400; //Options: 50, 100, 200, 400, 800, 1000, 1600, 3200 //400 on Ex.
5
  int pulseWidth = 411; //Options: 69, 118, 215, 411
  int adcRange = 4096; //Options: 2048, 4096, 8192, 16384

  particleSensor.setup(ledBrightness, sampleAverage, ledMode, sampleRate, pulseWidth,
adcRange); //Configure sensor with these settings

  //Arduino plotter auto-scales annoyingly. To get around this, pre-populate
  //the plotter with 500 of an average reading from the sensor

  //Take an average of IR readings at power up
  const byte avgAmount = 64;
  long baseValue = 0;
  for (byte x = 0 ; x < avgAmount ; x++)
  {
    baseValue += particleSensor.getIR(); //Read the IR value
  }
```

```
  baseValue /= avgAmount;

  //Pre-populate the plotter so that the Y scale is close to IR values
  /*for (int x = 0 ; x < 500 ; x++)
    Serial.println(baseValue);*/
}

void loop()
{
        long irValue = particleSensor.getIR();

        if (checkForBeat(irValue) == true)
        {
                //We sensed a beat!
                long delta = millis() - lastBeat;
                lastBeat = millis();

                beatsPerMinute = 60 / (delta / 1000.0);

                if (beatsPerMinute < 255 && beatsPerMinute > 20)
                {
                        rates[rateSpot++] = (byte)beatsPerMinute; //Store this reading in
the array
                        rateSpot %= RATE_SIZE; //Wrap variable

                        //Take average of readings
                        beatAvg = 0;
                        for (byte x = 0 ; x < RATE_SIZE ; x++)
                        beatAvg += rates[x];
                        beatAvg /= RATE_SIZE;
                }
        }
    Serial.print("IR=");
    Serial.print(irValue);
    Serial.print(", BPM=");
    Serial.print(beatsPerMinute);
    Serial.print(", Avg BPM(RR Interval)=");
    Serial.print(beatAvg);

    if (irValue < 50000)
    Serial.print(" No finger?");

    Serial.println();
  //Serial.println(particleSensor.getIR()); //Send raw data to plotter
}
```

MAX30105.c

```
#include <avr/io.h>
#include <stdio.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include "i2c_master.h"
#include "MAX301XX.h"

typedef int bool;
#define TRUE 1
#define FALSE 0

// Status Registers
static const uint8_t MAX30105_INTSTAT1 =        0x00;
static const uint8_t MAX30105_INTSTAT2 =        0x01;
static const uint8_t MAX30105_INTENABLE1 =              0x02;
static const uint8_t MAX30105_INTENABLE2 =              0x03;

// FIFO Registers
static const uint8_t MAX30105_FIFOWRITEPTR =    0x04;
static const uint8_t MAX30105_FIFOOVERFLOW =    0x05;
static const uint8_t MAX30105_FIFOREADPTR =     0x06;
static const uint8_t MAX30105_FIFODATA =        0x07;

// Configuration Registers
static const uint8_t MAX30105_FIFOCONFIG =              0x08;
static const uint8_t MAX30105_MODECONFIG =              0x09;
static const uint8_t MAX30105_PARTICLECONFIG =  0x0A;    // Note, sometimes listed as
"SPO2" config in datasheet (pg. 11)
static const uint8_t MAX30105_LED1_PULSEAMP =   0x0C;
static const uint8_t MAX30105_LED2_PULSEAMP =   0x0D;
static const uint8_t MAX30105_LED3_PULSEAMP =   0x0E;
static const uint8_t MAX30105_LED_PROX_AMP =    0x10;
static const uint8_t MAX30105_MULTILEDCONFIG1 = 0x11;
static const uint8_t MAX30105_MULTILEDCONFIG2 = 0x12;

// Die Temperature Registers
static const uint8_t MAX30105_DIETEMPINT =              0x1F;
static const uint8_t MAX30105_DIETEMPFRAC =     0x20;
static const uint8_t MAX30105_DIETEMPCONFIG =   0x21;

// Proximity Function Registers
static const uint8_t MAX30105_PROXINTTHRESH =   0x30;

// Part ID Registers
static const uint8_t MAX30105_REVISIONID =              0xFE;
static const uint8_t MAX30105_PARTID =                  0xFF;    // Should always be 0x15.
Identical to MAX30102.

// MAX30105 Commands
// Interrupt configuration (pg 13, 14)
static const uint8_t MAX30105_INT_A_FULL_MASK =         0b10000000;
static const uint8_t MAX30105_INT_A_FULL_ENABLE =       0x80;
static const uint8_t MAX30105_INT_A_FULL_DISABLE =      0x00;

static const uint8_t MAX30105_INT_DATA_RDY_MASK = 0b01000000;
static const uint8_t MAX30105_INT_DATA_RDY_ENABLE =     0x40;
```

```
static const uint8_t MAX30105_INT_DATA_RDY_DISABLE = 0x00;

static const uint8_t MAX30105_INT_ALC_OVF_MASK = 0b00100000;
static const uint8_t MAX30105_INT_ALC_OVF_ENABLE =     0x20;
static const uint8_t MAX30105_INT_ALC_OVF_DISABLE = 0x00;

static const uint8_t MAX30105_INT_PROX_INT_MASK = 0b00010000;
static const uint8_t MAX30105_INT_PROX_INT_ENABLE = 0x10;
static const uint8_t MAX30105_INT_PROX_INT_DISABLE = 0x00;

static const uint8_t MAX30105_INT_DIE_TEMP_RDY_MASK = 0b00000010;
static const uint8_t MAX30105_INT_DIE_TEMP_RDY_ENABLE = 0x02;
static const uint8_t MAX30105_INT_DIE_TEMP_RDY_DISABLE = 0x00;

static const uint8_t MAX30105_SAMPLEAVG_MASK =  0b11100000;
static const uint8_t MAX30105_SAMPLEAVG_1 =     0x00;
static const uint8_t MAX30105_SAMPLEAVG_2 =     0x20;
static const uint8_t MAX30105_SAMPLEAVG_4 =     0x40;
static const uint8_t MAX30105_SAMPLEAVG_8 =     0x60;
static const uint8_t MAX30105_SAMPLEAVG_16 =    0x80;
static const uint8_t MAX30105_SAMPLEAVG_32 =    0xA0;

static const uint8_t MAX30105_ROLLOVER_MASK =   0xEF;
static const uint8_t MAX30105_ROLLOVER_ENABLE = 0x10;
static const uint8_t MAX30105_ROLLOVER_DISABLE = 0x00;

static const uint8_t MAX30105_A_FULL_MASK =     0xF0;

// Mode configuration commands (page 19)
static const uint8_t MAX30105_SHUTDOWN_MASK =   0x7F;
static const uint8_t MAX30105_SHUTDOWN =        0x80;
static const uint8_t MAX30105_WAKEUP =                  0x00;

static const uint8_t MAX30105_RESET_MASK =              0xBF;
static const uint8_t MAX30105_RESET =                   0x40;

static const uint8_t MAX30105_MODE_MASK =               0xF8;
static const uint8_t MAX30105_MODE_REDONLY =    0x02;
static const uint8_t MAX30105_MODE_REDIRONLY =  0x03;
static const uint8_t MAX30105_MODE_MULTILED =   0x07;

// Particle sensing configuration commands (pgs 19-20)
static const uint8_t MAX30105_ADCRANGE_MASK =   0x9F;
static const uint8_t MAX30105_ADCRANGE_2048 =   0x00;
static const uint8_t MAX30105_ADCRANGE_4096 =   0x20;
static const uint8_t MAX30105_ADCRANGE_8192 =   0x40;
static const uint8_t MAX30105_ADCRANGE_16384 =  0x60;

static const uint8_t MAX30105_SAMPLERATE_MASK = 0xE3;
static const uint8_t MAX30105_SAMPLERATE_50 =   0x00;
static const uint8_t MAX30105_SAMPLERATE_100 =  0x04;
static const uint8_t MAX30105_SAMPLERATE_200 =  0x08;
static const uint8_t MAX30105_SAMPLERATE_400 =  0x0C;
static const uint8_t MAX30105_SAMPLERATE_800 =  0x10;
static const uint8_t MAX30105_SAMPLERATE_1000 = 0x14;
static const uint8_t MAX30105_SAMPLERATE_1600 = 0x18;
static const uint8_t MAX30105_SAMPLERATE_3200 = 0x1C;
```

```
static const uint8_t MAX30105_PULSEWIDTH_MASK = 0xFC;
static const uint8_t MAX30105_PULSEWIDTH_69 =   0x00;
static const uint8_t MAX30105_PULSEWIDTH_118 =  0x01;
static const uint8_t MAX30105_PULSEWIDTH_215 =  0x02;
static const uint8_t MAX30105_PULSEWIDTH_411 =  0x03;

//Multi-LED Mode configuration (pg 22)
static const uint8_t MAX30105_SLOT1_MASK =            0xF8;
static const uint8_t MAX30105_SLOT2_MASK =            0x8F;
static const uint8_t MAX30105_SLOT3_MASK =            0xF8;
static const uint8_t MAX30105_SLOT4_MASK =            0x8F;

static const uint8_t SLOT_NONE =                      0x00;
static const uint8_t SLOT_RED_LED =                   0x01;
static const uint8_t SLOT_IR_LED =                           0x02;
static const uint8_t SLOT_GREEN_LED =                 0x03;
static const uint8_t SLOT_NONE_PILOT =                0x04;
static const uint8_t SLOT_RED_PILOT =                 0x05;
static const uint8_t SLOT_IR_PILOT =                  0x06;
static const uint8_t SLOT_GREEN_PILOT =         0x07;

static const uint8_t MAX_30105_EXPECTEDPARTID = 0x15;

uint8_t i2caddr = 0x57; //7-bit I2C i2caddr

#define I2C_SPEED_STANDARD        100000
#define I2C_SPEED_FAST            400000

uint8_t revisionID;
uint8_t activeLEDs;

//The MAX30105 stores up to 32 samples on the IC
//This is additional local storage to the microcontroller
#define STORAGE_SIZE 4 //Each long is 4 uint8_t s so limit this to fit on your micro
struct Record
{
  uint32_t red[STORAGE_SIZE];
  uint32_t IR[STORAGE_SIZE];
  uint32_t green[STORAGE_SIZE];
  uint8_t  head;
  uint8_t  tail;
} sense; //This is our circular buffer of readings from the sensor

uint8_t requestFrom(uint8_t address)
{
    i2c_start(address);
    uint8_t rc = i2c_write((address<<1) | 1); // set read bit
    return rc;
}
/*
uint8_t requestFrom(int i2caddr)
{
    return requestFrom( (uint8_t) i2caddr);
}
// Added for compatibility with the standard Wire library.
uint8_t requestFrom(int i2caddr, int quantity)
{
    return requestFrom( (uint8_t) i2caddr);
```

```
    // Ignore 'quantity', since SoftI2CMaster::requestFrom() just sets the start of read
adresses,
    // so it's the same for any number of uint8_t s.
    (void)quantity;
}
// Added for compatibility with the standard Wire library.
uint8_t requestFrom(uint8_t i2caddr, uint8_t quantity)
{
    return requestFrom( (uint8_t) i2caddr);

    // Ignore 'quantity', since SoftI2CMaster::requestFrom() just sets the start of read
adresses,
    // so it's the same for any number of uint8_t s.
    (void)quantity;
}
*/

volatile uint64_t millis_prv = 0;

void millis_init()
{
 TCCR0A = 0;
 // set timer0 with CLKio/8 prescaler
 TCCR0B = _BV(CS01) | _BV(CS00);
 // clear any TOV1 Flag set when the timer overflowed
 TIFR0 &= ~TOV0;
 // set timer0 counter initial value to 0
 TCNT0 = 0x0;
 // enable timer overflow interrupt for Timer0
 TIMSK0 = _BV(TOIE0);
 // clear the Power Reduction Timer/Counter0
 PRR &= ~PRTIM0;
}

// TIMER0 interrupt handler
ISR(TIMER0_OVF_vect)
{
 // reset the counter (overflow is cleared automatically)
 TCNT0 = (uint8_t)(0xFF - ((F_CPU/8)/1000)); // use CLKio/8 prescaler (set CS0n
accordingly above)
 millis_prv++;
}

// return elapsed time in milliseconds
uint64_t millis()
{
 return millis_prv;
}

void begin(uint8_t address) {

 i2c_init();
 i2c_start(address);
  // Step 1: Initial Communication and Verification
  // Check that a MAX30105 is connected
  if (!readPartID() == MAX_30105_EXPECTEDPARTID) {
    // Error -- Part ID read from MAX30105 does not match expected part ID.
```

```
    // This may mean there is a physical connectivity problem (broken wire, unpowered,
etc).
    return i2c_stop;
  }

  // Populate revision ID
  readRevisionID();

  return TRUE;
}

//
// Configuration
//

//Begin Interrupt configuration
uint8_t getINT1(void) {
  return (readRegister8(i2caddr, MAX30105_INTSTAT1));
}
uint8_t getINT2(void) {
  return (readRegister8(i2caddr, MAX30105_INTSTAT2));
}

void enableAFULL(void) {
  bitMask(MAX30105_INTENABLE1, MAX30105_INT_A_FULL_MASK, MAX30105_INT_A_FULL_ENABLE);
}
void disableAFULL(void) {
  bitMask(MAX30105_INTENABLE1, MAX30105_INT_A_FULL_MASK, MAX30105_INT_A_FULL_DISABLE);
}

void enableDATARDY(void) {
  bitMask(MAX30105_INTENABLE1, MAX30105_INT_DATA_RDY_MASK, MAX30105_INT_DATA_RDY_ENABLE);
}
void disableDATARDY(void) {
  bitMask(MAX30105_INTENABLE1, MAX30105_INT_DATA_RDY_MASK,
MAX30105_INT_DATA_RDY_DISABLE);
}

void enableALCOVF(void) {
  bitMask(MAX30105_INTENABLE1, MAX30105_INT_ALC_OVF_MASK, MAX30105_INT_ALC_OVF_ENABLE);
}
void disableALCOVF(void) {
  bitMask(MAX30105_INTENABLE1, MAX30105_INT_ALC_OVF_MASK, MAX30105_INT_ALC_OVF_DISABLE);
}

void enablePROXINT(void) {
  bitMask(MAX30105_INTENABLE1, MAX30105_INT_PROX_INT_MASK, MAX30105_INT_PROX_INT_ENABLE);
}
void disablePROXINT(void) {
  bitMask(MAX30105_INTENABLE1, MAX30105_INT_PROX_INT_MASK,
MAX30105_INT_PROX_INT_DISABLE);
}

void enableDIETEMPRDY(void) {
  bitMask(MAX30105_INTENABLE2, MAX30105_INT_DIE_TEMP_RDY_MASK,
MAX30105_INT_DIE_TEMP_RDY_ENABLE);
}
void disableDIETEMPRDY(void) {
```

```
  bitMask(MAX30105_INTENABLE2, MAX30105_INT_DIE_TEMP_RDY_MASK,
MAX30105_INT_DIE_TEMP_RDY_DISABLE);
}

//End Interrupt configuration

void softReset(void) {
  bitMask(MAX30105_MODECONFIG, MAX30105_RESET_MASK, MAX30105_RESET);

  // Poll for bit to clear, reset is then complete
  // Timeout after 100ms
  unsigned long startTime = millis();
  while (millis() - startTime < 100)
  {
    uint8_t response = readRegister8(i2caddr, MAX30105_MODECONFIG);
    if ((response & MAX30105_RESET) == 0) break; //We're done!
    _delay_ms(10); //Let's not over burden the I2C bus
  }
}

void shutDown(void) {
  // Put IC into low power mode (datasheet pg. 19)
  // During shutdown the IC will continue to respond to I2C commands but will
  // not update with or take new readings (such as temperature)
  bitMask(MAX30105_MODECONFIG, MAX30105_SHUTDOWN_MASK, MAX30105_SHUTDOWN);
}

void wakeUp(void) {
  // Pull IC out of low power mode (datasheet pg. 19)
  bitMask(MAX30105_MODECONFIG, MAX30105_SHUTDOWN_MASK, MAX30105_WAKEUP);
}

void setLEDMode(uint8_t mode) {
  // Set which LEDs are used for sampling -- Red only, RED+IR only, or custom.
  // See datasheet, page 19
  bitMask(MAX30105_MODECONFIG, MAX30105_MODE_MASK, mode);
}

void setADCRange(uint8_t adcRange) {
  // adcRange: one of MAX30105_ADCRANGE_2048, _4096, _8192, _16384
  bitMask(MAX30105_PARTICLECONFIG, MAX30105_ADCRANGE_MASK, adcRange);
}

void setSampleRate(uint8_t sampleRate) {
  // sampleRate: one of MAX30105_SAMPLERATE_50, _100, _200, _400, _800, _1000, _1600,
_3200
  bitMask(MAX30105_PARTICLECONFIG, MAX30105_SAMPLERATE_MASK, sampleRate);
}

void setPulseWidth(uint8_t pulseWidth) {
  // pulseWidth: one of MAX30105_PULSEWIDTH_69, _188, _215, _411
  bitMask(MAX30105_PARTICLECONFIG, MAX30105_PULSEWIDTH_MASK, pulseWidth);
}

// NOTE: Amplitude values: 0x00 = 0mA, 0x7F = 25.4mA, 0xFF = 50mA (typical)
// See datasheet, page 21
void setPulseAmplitudeRed(uint8_t amplitude) {
  writeRegister8(i2caddr, MAX30105_LED1_PULSEAMP, amplitude);
```

```
}

void setPulseAmplitudeIR(uint8_t amplitude) {
  writeRegister8(i2caddr, MAX30105_LED2_PULSEAMP, amplitude);
}

void setPulseAmplitudeGreen(uint8_t amplitude) {
  writeRegister8(i2caddr, MAX30105_LED3_PULSEAMP, amplitude);
}

void setPulseAmplitudeProximity(uint8_t amplitude) {
  writeRegister8(i2caddr, MAX30105_LED_PROX_AMP, amplitude);
}

void setProximityThreshold(uint8_t threshMSB) {
  // Set the IR ADC count that will trigger the beginning of particle-sensing mode.
  // The threshMSB signifies only the 8 most significant-bits of the ADC count.
  // See datasheet, page 24.
  writeRegister8(i2caddr, MAX30105_PROXINTTHRESH, threshMSB);
}

//Given a slot number assign a thing to it
//Devices are SLOT_RED_LED or SLOT_RED_PILOT (proximity)
//Assigning a SLOT_RED_LED will pulse LED
//Assigning a SLOT_RED_PILOT will ??
void enableSlot(uint8_t slotNumber, uint8_t device) {

  uint8_t originalContents;

  switch (slotNumber) {
    case (1):
      bitMask(MAX30105_MULTILEDCONFIG1, MAX30105_SLOT1_MASK, device);
      break;
    case (2):
      bitMask(MAX30105_MULTILEDCONFIG1, MAX30105_SLOT2_MASK, device << 4);
      break;
    case (3):
      bitMask(MAX30105_MULTILEDCONFIG2, MAX30105_SLOT3_MASK, device);
      break;
    case (4):
      bitMask(MAX30105_MULTILEDCONFIG2, MAX30105_SLOT4_MASK, device << 4);
      break;
    default:
      //Shouldn't be here!
      break;
  }
}

//Clears all slot assignments
void disableSlots(void) {
  writeRegister8(i2caddr, MAX30105_MULTILEDCONFIG1, 0);
  writeRegister8(i2caddr, MAX30105_MULTILEDCONFIG2, 0);
}

//
// FIFO Configuration
//
```

```
//Set sample average (Table 3, Page 18)
void setFIFOAverage(uint8_t numberOfSamples) {
  bitMask(MAX30105_FIFOCONFIG, MAX30105_SAMPLEAVG_MASK, numberOfSamples);
}

//Resets all points to start in a known state
//Page 15 recommends clearing FIFO before beginning a read
void clearFIFO(void) {
  writeRegister8(i2caddr, MAX30105_FIFOWRITEPTR, 0);
  writeRegister8(i2caddr, MAX30105_FIFOOVERFLOW, 0);
  writeRegister8(i2caddr, MAX30105_FIFOREADPTR, 0);
}

//Enable roll over if FIFO over flows
void enableFIFORollover(void) {
  bitMask(MAX30105_FIFOCONFIG, MAX30105_ROLLOVER_MASK, MAX30105_ROLLOVER_ENABLE);
}

//Disable roll over if FIFO over flows
void disableFIFORollover(void) {
  bitMask(MAX30105_FIFOCONFIG, MAX30105_ROLLOVER_MASK, MAX30105_ROLLOVER_DISABLE);
}

//Set number of samples to trigger the almost full interrupt (Page 18)
//Power on default is 32 samples
//Note it is reverse: 0x00 is 32 samples, 0x0F is 17 samples
void setFIFOAlmostFull(uint8_t numberOfSamples) {
  bitMask(MAX30105_FIFOCONFIG, MAX30105_A_FULL_MASK, numberOfSamples);
}

//Read the FIFO Write Pointer
uint8_t getWritePointer(void) {
  return (readRegister8(i2caddr, MAX30105_FIFOWRITEPTR));
}

//Read the FIFO Read Pointer
uint8_t getReadPointer(void) {
  return (readRegister8(i2caddr, MAX30105_FIFOREADPTR));
}


// Die Temperature
// Returns temp in C
float readTemperature() {
  // Step 1: Config die temperature register to take 1 temperature sample
  writeRegister8(i2caddr, MAX30105_DIETEMPCONFIG, 0x01);

  // Poll for bit to clear, reading is then complete
  // Timeout after 100ms
  unsigned long startTime = millis();
  while (millis() - startTime < 100)
  {
    uint8_t response = readRegister8(i2caddr, MAX30105_DIETEMPCONFIG);
    if ((response & 0x01) == 0) break; //We're done!
    _delay_ms(10); //Let's not over burden the I2C bus
  }
  //TODO How do we want to fail? With what type of error?
  //? if(millis() - startTime >= 100) return(-999.0);
```

```
  // Step 2: Read die temperature register (integer)
  int8_t tempInt = readRegister8(i2caddr, MAX30105_DIETEMPINT);
  uint8_t tempFrac = readRegister8(i2caddr, MAX30105_DIETEMPFRAC);

  // Step 3: Calculate temperature (datasheet pg. 23)
  return (float)tempInt + ((float)tempFrac * 0.0625);
}

// Returns die temp in F
float readTemperatureF() {
  float temp = readTemperature();

  if (temp != -999.0) temp = temp * 1.8 + 32.0;

  return (temp);
}

// Set the PROX_INT_THRESHold
void setPROXINTTHRESH(uint8_t val) {
  writeRegister8(i2caddr, MAX30105_PROXINTTHRESH, val);
}


//
// Device ID and Revision
//
uint8_t readPartID() {
  return readRegister8(i2caddr, MAX30105_PARTID);
}

void readRevisionID() {
  revisionID = readRegister8(i2caddr, MAX30105_REVISIONID);
}

uint8_t getRevisionID() {
  return revisionID;
}


//Setup the sensor
//The MAX30105 has many settings. By default we select:
// Sample Average = 4
// Mode = MultiLED
// ADC Range = 16384 (62.5pA per LSB)
// Sample rate = 50
//Use the default setup if you are just getting started with the MAX30105 sensor
void setup(uint8_t powerLevel, uint8_t sampleAverage, uint8_t ledMode, int sampleRate,
int pulseWidth, int adcRange) {
  softReset(); //Reset all configuration, threshold, and data registers to POR values
/*
  powerLevel = pL;
  sampleAverage = sA;
  ledMode = lM;
  sampleRate = sR;
  pulseWidth = pW;
  adcRange = aR;
*/
```

```
  //FIFO Configuration
  //-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
  //The chip will average multiple samples of same type together if you wish
  if (sampleAverage == 1) setFIFOAverage(MAX30105_SAMPLEAVG_1); //No averaging per FIFO
record
  else if (sampleAverage == 2) setFIFOAverage(MAX30105_SAMPLEAVG_2);
  else if (sampleAverage == 4) setFIFOAverage(MAX30105_SAMPLEAVG_4);
  else if (sampleAverage == 8) setFIFOAverage(MAX30105_SAMPLEAVG_8);
  else if (sampleAverage == 16) setFIFOAverage(MAX30105_SAMPLEAVG_16);
  else if (sampleAverage == 32) setFIFOAverage(MAX30105_SAMPLEAVG_32);
  else setFIFOAverage(MAX30105_SAMPLEAVG_4);

  //setFIFOAlmostFull(2); //Set to 30 samples to trigger an 'Almost Full' interrupt
  enableFIFORollover(); //Allow FIFO to wrap/roll over
  //-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

  //Mode Configuration
  //-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
  if (ledMode == 3) setLEDMode(MAX30105_MODE_MULTILED); //Watch all three LED channels
  else if (ledMode == 2) setLEDMode(MAX30105_MODE_REDIRONLY); //Red and IR
  else setLEDMode(MAX30105_MODE_REDONLY); //Red only
  activeLEDs = ledMode; //Used to control how many uint8_t s to read from FIFO buffer
  //-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

  //Particle Sensing Configuration
  //-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
  if(adcRange < 4096) setADCRange(MAX30105_ADCRANGE_2048); //7.81pA per LSB
  else if(adcRange < 8192) setADCRange(MAX30105_ADCRANGE_4096); //15.63pA per LSB
  else if(adcRange < 16384) setADCRange(MAX30105_ADCRANGE_8192); //31.25pA per LSB
  else if(adcRange == 16384) setADCRange(MAX30105_ADCRANGE_16384); //62.5pA per LSB
  else setADCRange(MAX30105_ADCRANGE_2048);

  if (sampleRate < 100) setSampleRate(MAX30105_SAMPLERATE_50); //Take 50 samples per
second
  else if (sampleRate < 200) setSampleRate(MAX30105_SAMPLERATE_100);
  else if (sampleRate < 400) setSampleRate(MAX30105_SAMPLERATE_200);
  else if (sampleRate < 800) setSampleRate(MAX30105_SAMPLERATE_400);
  else if (sampleRate < 1000) setSampleRate(MAX30105_SAMPLERATE_800);
  else if (sampleRate < 1600) setSampleRate(MAX30105_SAMPLERATE_1000);
  else if (sampleRate < 3200) setSampleRate(MAX30105_SAMPLERATE_1600);
  else if (sampleRate == 3200) setSampleRate(MAX30105_SAMPLERATE_3200);
  else setSampleRate(MAX30105_SAMPLERATE_50);

  //The longer the pulse width the longer range of detection you'll have
  //At 69us and 0.4mA it's about 2 inches
  //At 411us and 0.4mA it's about 6 inches
  if (pulseWidth < 118) setPulseWidth(MAX30105_PULSEWIDTH_69); //Page 26, Gets us 15 bit
resolution
  else if (pulseWidth < 215) setPulseWidth(MAX30105_PULSEWIDTH_118); //16 bit resolution
  else if (pulseWidth < 411) setPulseWidth(MAX30105_PULSEWIDTH_215); //17 bit resolution
  else if (pulseWidth == 411) setPulseWidth(MAX30105_PULSEWIDTH_411); //18 bit resolution
  else setPulseWidth(MAX30105_PULSEWIDTH_69);
  //-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

  //LED Pulse Amplitude Configuration
  //-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
  //Default is 0x1F which gets us 6.4mA
  //powerLevel = 0x02, 0.4mA - Presence detection of ~4 inch
```

```
  //powerLevel = 0x1F, 6.4mA - Presence detection of ~8 inch
  //powerLevel = 0x7F, 25.4mA - Presence detection of ~8 inch
  //powerLevel = 0xFF, 50.0mA - Presence detection of ~12 inch

  setPulseAmplitudeRed(powerLevel);
  setPulseAmplitudeIR(powerLevel);
  setPulseAmplitudeGreen(powerLevel);
  setPulseAmplitudeProximity(powerLevel);
  //-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

  //Multi-LED Mode Configuration, Enable the reading of the three LEDs
  //-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
  enableSlot(1, SLOT_RED_LED);
  if (ledMode > 1) enableSlot(2, SLOT_IR_LED);
  if (ledMode > 2) enableSlot(3, SLOT_GREEN_LED);
  //enableSlot(1, SLOT_RED_PILOT);
  //enableSlot(2, SLOT_IR_PILOT);
  //enableSlot(3, SLOT_GREEN_PILOT);
  //-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

  clearFIFO(); //Reset the FIFO before we begin checking the sensor
}

//
// Data Collection
//

//Tell caller how many samples are available
uint8_t available(void)
{
  int8_t numberOfSamples = sense.head - sense.tail;
  if (numberOfSamples < 0) numberOfSamples += STORAGE_SIZE;

  return (numberOfSamples);
}

//Report the most recent red value
uint32_t getRed(void)
{
  //Check the sensor for new data for 250ms
  if(safeCheck(250))
    return (sense.red[sense.head]);
  else
    return(0); //Sensor failed to find new data
}

//Report the most recent IR value
uint32_t getIR(void)
{
  //Check the sensor for new data for 250ms
  if(safeCheck(250))
    return (sense.IR[sense.head]);
  else
    return(0); //Sensor failed to find new data
}

//Report the most recent Green value
uint32_t getGreen(void)
```

```
{
  //Check the sensor for new data for 250ms
  if(safeCheck(250))
    return (sense.green[sense.head]);
  else
    return(0); //Sensor failed to find new data
}

//Report the next Red value in the FIFO
uint32_t getFIFORed(void)
{
  return (sense.red[sense.tail]);
}

//Report the next IR value in the FIFO
uint32_t getFIFOIR(void)
{
  return (sense.IR[sense.tail]);
}

//Report the next Green value in the FIFO
uint32_t getFIFOGreen(void)
{
  return (sense.green[sense.tail]);
}

//Advance the tail
void nextSample(void)
{
  if(available()) //Only advance the tail if new data is available
  {
    sense.tail++;
    sense.tail %= STORAGE_SIZE; //Wrap condition
  }
}

//Polls the sensor for new data
//Call regularly
//If new data is available, it updates the head and tail in the main struct
//Returns number of new samples obtained
uint16_t check(void)
{
  //Read register FIDO_DATA in (3-uint8_t  * number of active LED) chunks
  //Until FIFO_RD_PTR = FIFO_WR_PTR

  uint8_t  readPointer = getReadPointer();
  uint8_t  writePointer = getWritePointer();

  int numberOfSamples = 0;

  //Do we have new data?
  if (readPointer != writePointer)
  {
    //Calculate the number of readings we need to get from sensor
    numberOfSamples = writePointer - readPointer;
    if (numberOfSamples < 0) numberOfSamples += 32; //Wrap condition

    //We now have the number of readings, now calc uint8_t s to read
```

```
    //For this example we are just doing Red and IR (3 uint8_t s each)
    int bytesLeftToRead = numberOfSamples * activeLEDs * 3;

    //Get ready to read a burst of data from the FIFO register
    i2c_start(i2caddr);
    i2c_write(MAX30105_FIFODATA);
    i2c_stop();

    //We may need to read as many as 288 uint8_t s so we read in blocks no larger than
I2C_BUFFER_LENGTH
    //I2C_BUFFER_LENGTH changes based on the platform. 64 uint8_t s for SAMD21, 32
uint8_t s for Uno.
    //Wire.requestFrom() is limited to BUFFER_LENGTH which is 32 on the Uno
    while (bytesLeftToRead > 0)
    {
      int toGet =  bytesLeftToRead;
      if (toGet > I2C_BUFFER_LENGTH)
      {
        //If toGet is 32 this is bad because we read 6 uint8_t s (Red+IR * 3 = 6) at a
time
        //32 % 6 = 2 left over. We don't want to request 32 uint8_t s, we want to request
30.
        //32 % 9 (Red+IR+GREEN) = 5 left over. We want to request 27.

        toGet = I2C_BUFFER_LENGTH - (I2C_BUFFER_LENGTH % (activeLEDs * 3)); //Trim toGet
to be a multiple of the samples we need to read
      }

      bytesLeftToRead -= toGet;

      //Request toGet number of uint8_t s from sensor
      requestFrom(MAX30105_ADDRESS);

      while (toGet > 0)
      {
        sense.head++; //Advance the head of the storage struct
        sense.head %= STORAGE_SIZE; //Wrap condition

        uint8_t  temp[sizeof(uint32_t)]; //Array of 4 uint8_t s that we will convert into
long
        uint32_t tempLong;

        //Burst read three uint8_t s - RED
        temp[3] = 0;
        temp[2] = i2c_read_nack();
        temp[1] = i2c_read_nack();
        temp[0] = i2c_read_nack();

        //Convert array to long
        memcpy(&tempLong, temp, sizeof(tempLong));

              tempLong &= 0x3FFFF; //Zero out all but 18 bits

        sense.red[sense.head] = tempLong; //Store this reading into the sense array

        if (activeLEDs > 1)
        {
          //Burst read three more uint8_t s - IR
```

```
          temp[3] = 0;
          temp[2] = i2c_read_nack();
          temp[1] = i2c_read_nack();
          temp[0] = i2c_read_nack();

          //Convert array to long
          memcpy(&tempLong, temp, sizeof(tempLong));

                tempLong &= 0x3FFFF; //Zero out all but 18 bits

                sense.IR[sense.head] = tempLong;
        }

        if (activeLEDs > 2)
        {
          //Burst read three more uint8_t s - Green
          temp[3] = 0;
          temp[2] = i2c_read_nack();
          temp[1] = i2c_read_nack();
          temp[0] = i2c_read_nack();

          //Convert array to long
          memcpy(&tempLong, temp, sizeof(tempLong));

                tempLong &= 0x3FFFF; //Zero out all but 18 bits

          sense.green[sense.head] = tempLong;
        }

        toGet -= activeLEDs * 3;
      }

    } //End while (bytesLeftToRead > 0)

  } //End readPtr != writePtr

  return (numberOfSamples); //Let the world know how much new data we found
}

//Check for new data but give up after a certain amount of time
//Returns TRUE if new data was found
//Returns i2c_stop if new data was not found
bool safeCheck(uint8_t maxTimeToCheck)
{
  uint32_t markTime = millis();

  while(1)
  {
       if(millis() - markTime > maxTimeToCheck) return(FALSE);

       if(check() == TRUE) //We found new data!
         return(TRUE);

       _delay_ms(10);
  }
}

//Given a register, read it, mask it, and then set the thing
```

```
void bitMask(uint8_t reg, uint8_t mask, uint8_t thing)
{
  // Grab current register context
  uint8_t originalContents = readRegister8(i2caddr, reg);

  // Zero-out the portions of the register we're interested in
  originalContents = originalContents & mask;

  // Change contents
  writeRegister8(i2caddr, reg, originalContents | thing);
}

//
// Low-level I2C Communication
//
uint8_t readRegister8(uint8_t address, uint8_t reg) {
  uint8_t data;
  i2c_start(address);
       i2c_write(reg);
       data = i2c_read_nack();
       i2c_stop();
       return data;
}

void writeRegister8(uint8_t address, uint8_t reg, uint8_t value) {
  i2c_start((address<<1) | 1);
  i2c_write(reg);
  i2c_write(value);
  i2c_stop();
}
```

heartRate.c

```
#include <stdint.h>
#include "heartRate.h"

typedef int bool;
#define TRUE 1
#define FALSE 0

int16_t IR_AC_Max = 20;
int16_t IR_AC_Min = -20;

int16_t IR_AC_Signal_Current = 0;
int16_t IR_AC_Signal_Previous;
int16_t IR_AC_Signal_min = 0;
int16_t IR_AC_Signal_max = 0;
int16_t IR_Average_Estimated;

int16_t positiveEdge = 0;
int16_t negativeEdge = 0;
int32_t ir_avg_reg = 0;

int16_t cbuf[32];
uint8_t offset = 0;

static const uint16_t FIRCoeffs[12] = {172, 321, 579, 927, 1360, 1858, 2390, 2916, 3391,
3768, 4012, 4096};

//  Heart Rate Monitor functions takes a sample value and the sample number
//  Returns TRUE if a beat is detected
//  A running average of four samples is recommended for display on the screen.
bool checkForBeat(int32_t sample)
{
  bool beatDetected = FALSE;

  //  Save current state
  IR_AC_Signal_Previous = IR_AC_Signal_Current;

  //This is good to view for debugging
  //Serial.print("Signal_Current: ");
  //Serial.println(IR_AC_Signal_Current);

  //  Process next data sample
  IR_Average_Estimated = averageDCEstimator(&ir_avg_reg, sample);
  IR_AC_Signal_Current = lowPassFIRFilter(sample - IR_Average_Estimated);

  //  Detect positive zero crossing (rising edge)
  if ((IR_AC_Signal_Previous < 0) & (IR_AC_Signal_Current >= 0))
  {

    IR_AC_Max = IR_AC_Signal_max; //Adjust our AC max and min
    IR_AC_Min = IR_AC_Signal_min;

    positiveEdge = 1;
    negativeEdge = 0;
    IR_AC_Signal_max = 0;

    //if ((IR_AC_Max - IR_AC_Min) > 100 & (IR_AC_Max - IR_AC_Min) < 1000)
```

```
    if ((IR_AC_Max - IR_AC_Min) > 20 & (IR_AC_Max - IR_AC_Min) < 1000)
    {
      //Heart beat!!!
      beatDetected = TRUE;
    }
  }
  //  Detect negative zero crossing (falling edge)
  if ((IR_AC_Signal_Previous > 0) & (IR_AC_Signal_Current <= 0))
  {
    positiveEdge = 0;
    negativeEdge = 1;
    IR_AC_Signal_min = 0;
  }
  //  Find Maximum value in positive cycle
  if (positiveEdge & (IR_AC_Signal_Current > IR_AC_Signal_Previous))
  {
    IR_AC_Signal_max = IR_AC_Signal_Current;
  }
  //  Find Minimum value in negative cycle
  if (negativeEdge & (IR_AC_Signal_Current < IR_AC_Signal_Previous))
  {
    IR_AC_Signal_min = IR_AC_Signal_Current;
  }

  return(beatDetected);
}
//  Average DC Estimator
int16_t averageDCEstimator(int32_t *p, uint16_t x)
{
  *p += ((((long) x << 15) - *p) >> 4);
  return (*p >> 15);
}
//  Low Pass FIR Filter
int16_t lowPassFIRFilter(int16_t din)
{
  cbuf[offset] = din;

  int32_t z = mul16(FIRCoeffs[11], cbuf[(offset - 11) & 0x1F]);

  for (uint8_t i = 0 ; i < 11 ; i++)
  {
    z += mul16(FIRCoeffs[i], cbuf[(offset - i) & 0x1F] + cbuf[(offset - 22 + i) & 0x1F]);
  }

  offset++;
  offset %= 32; //Wrap condition

  return(z >> 15);
}
//  Integer multiplier
int32_t mul16(int16_t x, int16_t y)
{
  return((long)x * (long)y);
}
```

REFERENCE:

ATmega328p Datasheet: http://www.atmel.com/images/atmel-8271-8-bit-avr-microcontroller-atmega48a-48pa-88a-88pa-168a-168pa-328-328p_datasheet_complete.pdf

MAX30105 Datasheet: https://www.mouser.com/ds/2/256/MAX30105-967526.pdf

MAX30105 Libraries:
https://github.com/sparkfun/SparkFun_MAX3010x_Sensor_Library/tree/master/src

I2C Library: https://github.com/g4lvanix/I2C-master-lib

USART Libraries: http://jump.to/fleury