Abstract

The overarching goal of this project was to create an LED fan, a device that produces an image of some sort by flashing leds rapidly as the fan rotates. We purchased a motor, an LED strip, a slip ring, a motor hub, a magnet and a Hall effect sensor. Using these components and some raw materials, we constructed a fan with an LED strip attached to one end of the blade. We then wrote Verilog code to control the led strip and C code to control the motors and tell the FPGA when the LEDs should pulse and reset. This allowed us to not only fulfill the project requirement of creating an interesting device that uses the FPGA, ATSAM, and new hardware, but also do a project that we felt produced a really cool end product.

Introduction

Motivation

This project was motivated by us thinking about a problem that would allow us to do some machining as well as possibly some sort of visualization with music. This ended up being relatively complicated after some initial research, so it was pared down to machining some hardware that would also have some visualization. The idea of an LED fan was proposed and after research and discussions with Professor Harris, we moved forward with this idea. We think LED fans are a really cool product and initially we weren't sure just how difficult it would be, but believed that we could finish in the time allotted.

Block Diagram

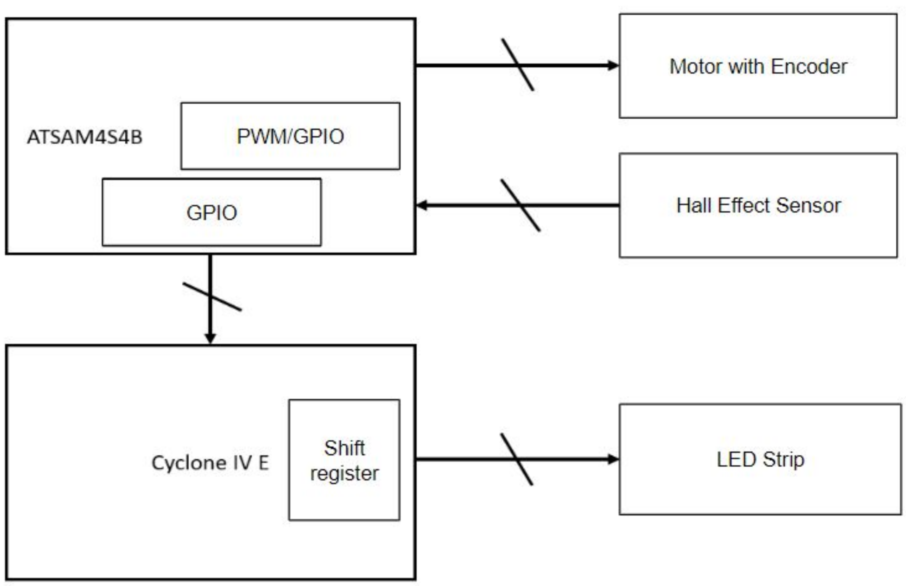


Figure 1: Overall System Block Diagram

Overview

In terms of hardware, the parts we had to work with were a motor with a 25 mm outer diameter, a motor hub for the 4mm output shaft of the motor, and a slip ring with a 22mm outer diameter with a flange. First, we machined a block of aluminum to the dimensions shown in Appendix A. This served as the main housing for the motor, giving it the height needed as well as rigidity to prevent any sort of large scale vibrations damaging our set up. We then machined the holes into the front plate shown in Appendix B. This front plate has the same screw pattern as the front of the motor, allowing us to use the front plate to connect the motor housing securely to the motor. We then created the aluminum post shown in Appendix C which served as a support for the slip ring. The holes at the top of the post served to connect to the laser cut piece shown in Appendix D which directly connects the flange of the slip ring. The fan itself was also machined by laser cutting the simple profile shown in Appendix E, with the hole pattern in the center of the fan blades matching that on the motor hub. The full assembly, sans the motor, slip ring, motor hub and fan blades is shown in Appendix F

New Hardware

LED Strip

The LED strip we chose was a Pololu addressable RGB LED strip [1] that we cut to 10 LEDs. The strip required a VCC between 5V and 3.5V. We expected each LED to draw a maximum of 40 mA (at maximum R, G, and B intensities), so with 10 LEDs all on we expected to draw 400 mA of current.

The FPGA controls the led strip using the SK6812 single line digital communication protocol. The full communication protocol is linked in Appendix G, but the key part of the protocol is displayed in figure 2.

10. The data transmission time (TH+TL=1.25μs±600ns):

T0H	0 code, high level time	0.3μs	±0.15μs
T0L	0 code, low level time	0.9μs	±0.15μs
T1H	1 code, high level time	0.6μs	±0.15μs
T1L	1 code, low level time	0.6μs	±0.15μs
Trst	Reset code，low level time	80μs	

11. Timing waveform:

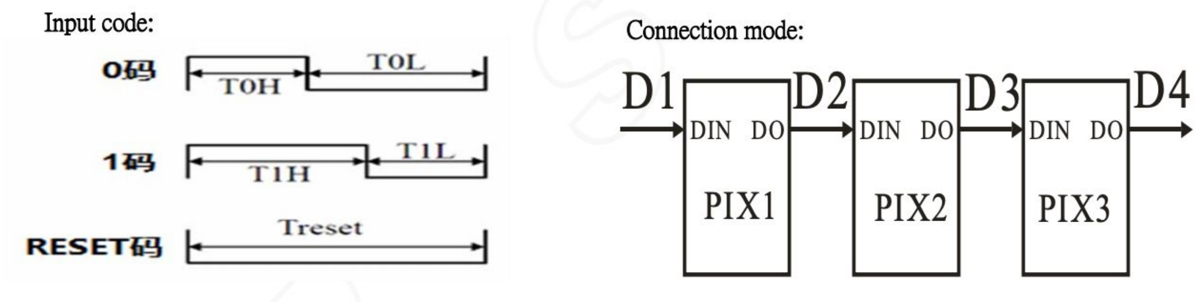


Figure 2: SK6812 Communication Protocol

As can be seen from the figure, if we want to send a 0, we need to send high for 0.3 μs, followed by low for 0.9μs. If we want to send a 1 instead, we need to send high for 0.6 μs followed by low for 0.6μs. This leads itself rather intuitively to the FPGA design of having the bit current bit being sent control a multiplexer that contains in it either a 4’b1000 or 4’b1100, and have those bits shifted out on 0.3μs intervals. In terms of the bits themselves, an image is first created in Photoshop with 100x10 pixel dimensions. This is exported as a jpeg of maximum size. The jpeg is then processed by a python script which separates the image into individual R,G, and B channels, and then takes each channel and converts it from a 10x100 matrix to a 1x1000 vector. The zeroth position is first pixel in the first column and it goes column by column, meaning positions 0-9 are the first column, 10-19 are the second column and so on. These vectors is exported as a .txt file and loaded into memory by Quartus. This results in 3 separate memory banks of size 8x1000 with each bank containing the bit values for a single color channel. These bits are shifted out with the appropriate waveforms as shown above. The process is described in more detail in the FPGA Design section.

Gearmotor

To spin the LED fan at sufficient speeds, we decided to use a brushed DC gearmotor. We selected a Pololu low-power brushed DC gearmotor with a 4.4:1 gearbox and a built-in quadrature encoder [2].

The quadrature encoder on the motor has 48 total counts per revolution (posedge and negedge of two encoder outputs). Thus, one encoder output has 12 posedge counts per revolution. With 4.4 revolutions per one full shaft rotation, this means that one encoder output will have 52.8 counts per revolution.

To supply sufficient power to the Gearmotor, a motor driver circuit was used, with help from Sparkfun's PWM motor tutorial [3]. A TIP31A NPN transistor [4] was used, and current was controlled with a base resistor of 680 Ohms. The motor driver circuit ensures that the PWM signal from the ATSAM drives enough current to the motor to overcome its stall current. The circuit is shown in Figure 3 in the Schematics section.

With experimentation, it was found that the motor needed an initial spin or initial application of 5 V dc to consistently overcome the stall current before the PWM signal could drive the motor alone. Thus, a SPST switch button was placed between the base and the collector of the transistor. Each time the circuit is powered, the button is pressed briefly to connect the motor to 5V dc and start it.

Hall Effect Sensor and Magnet

To be able to detect the absolute position of the LED fan, we decided to use a Hall effect sensor and a Magnet. The sensor we picked was a Melexis US5881LUA Hall switch [5], and the magnet we picked was a 1 mm thick, 12 mm diameter neodymium magnet [6]. The hall switch required 5VCC and GND for power. It was south-pole polarized. When no magnetic field was detected, the hall switch output 500mV; when a south-polarized magnetic field was detected, the hall switch output 0mV. From testing, it was found that the hall switch triggered low when the south pole of the magnet was within 4 mm of the south-pole-side of the hall switch.

Two magnets were stacked and taped to the opposite acrylic fan blade as the LED strip, to provide a counterweight to the LED strip. The hall switch was placed on the base of the mount below and behind the fan blade so that as the fan swung the magnets would be close enough in proximity to the hall switch to trigger it on each rotation.

As shown in Figure 3 in the Schematics section, a non-inverting gain operational amplifier [7] was used to raise the output levels of the hall switch up to 3.3V logic required by the ATSAM, which received the resulting signal through GPIO. The TL081CP dual-rail op amp was used, requiring +-5Vdc [8]. A gain of 3.3/0.5=6.6 was required, so a 660 kOhm resistor and a 100 kOhm resistor were used for R2 and R1, respectively.

Schematics

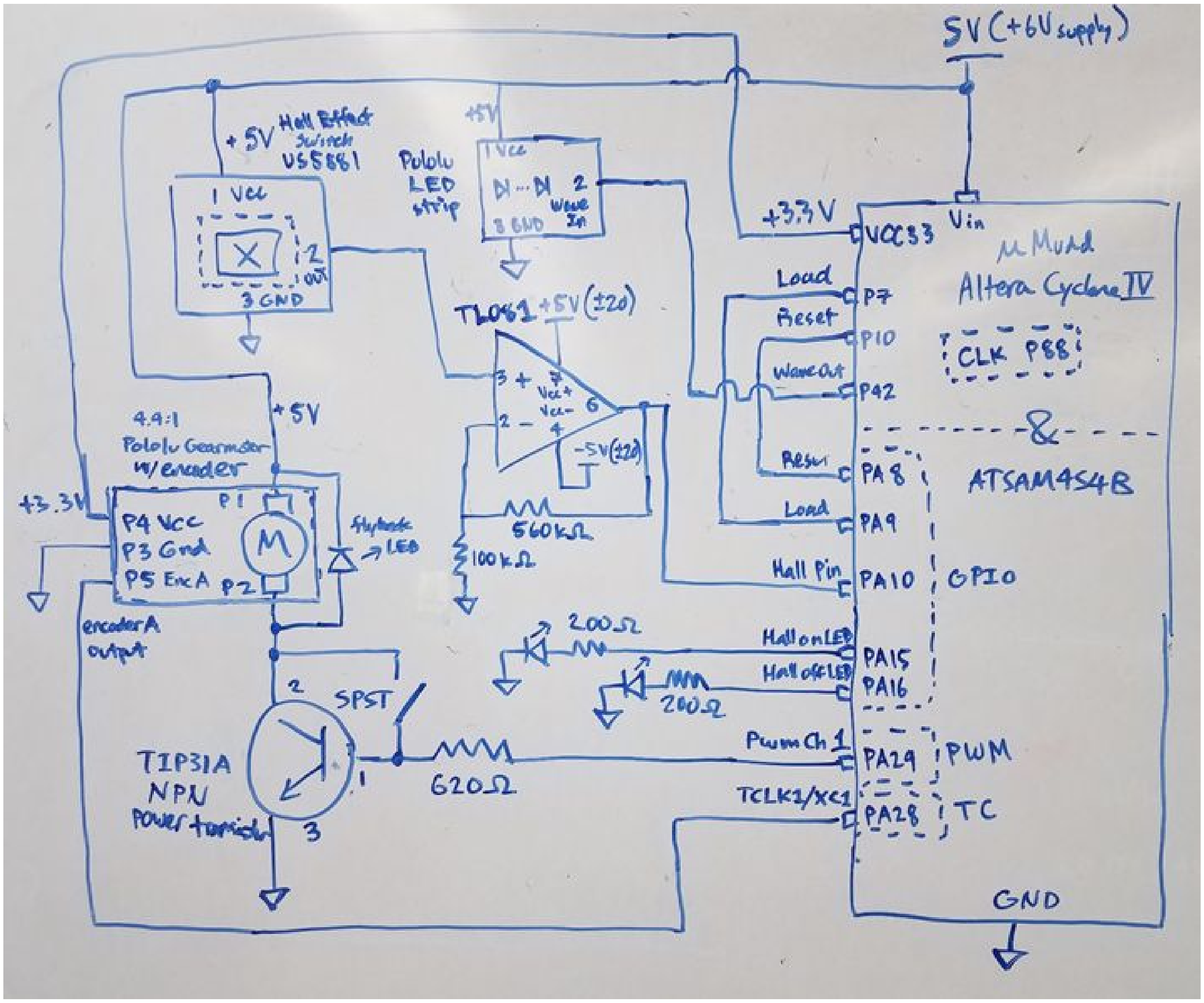


Figure 3: External Hardware Schematic

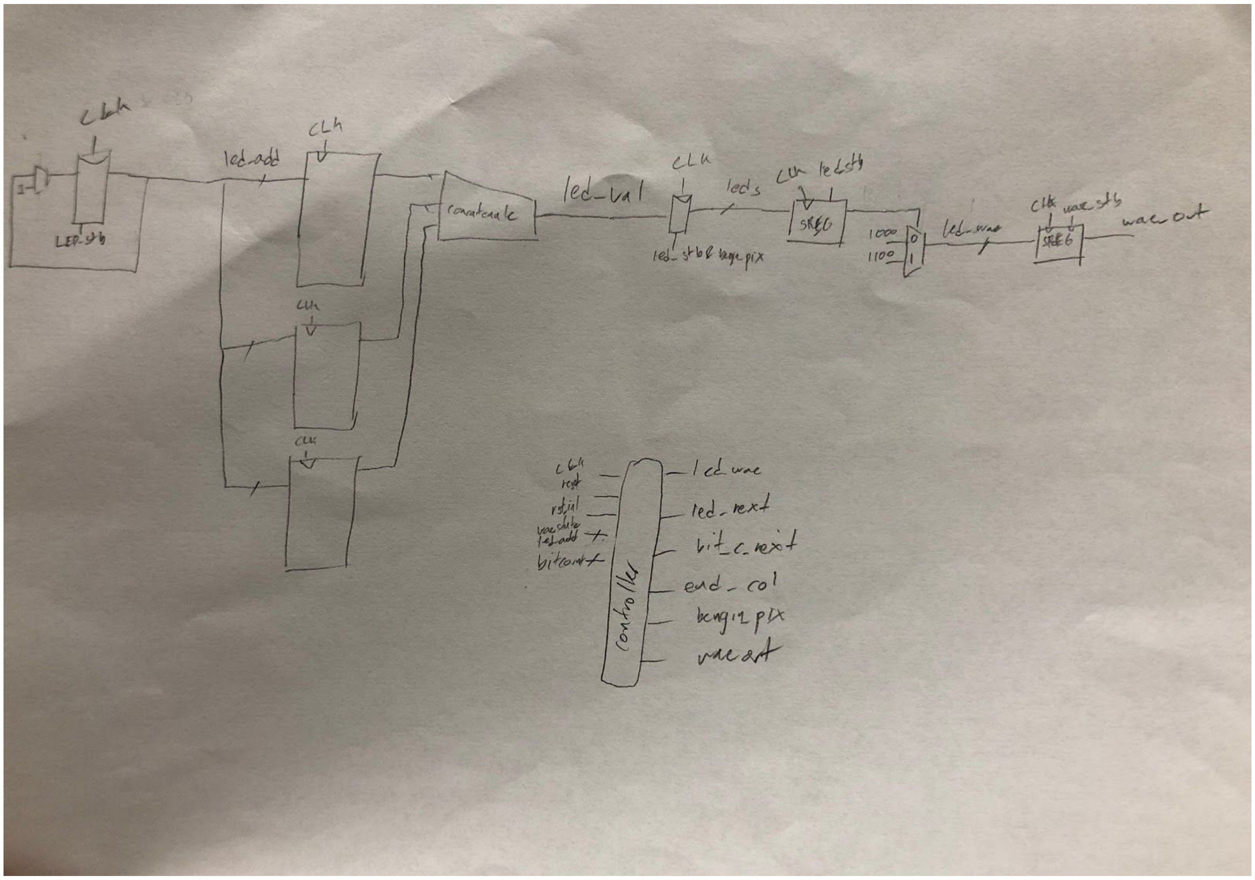


Figure 4: FPGA Logic Schematic

Microcontroller Design

The ATSAM was responsible for controlling the motor speed, obtaining a sense of the motor's speed from the motor's onboard encoder, and sending appropriately timed control signals to the FPGA.

For controlling the motor speed, the Pulse Width Modulation (PWM) peripheral was configured and used. At the beginning of the project, we anticipated a need to use multiple peripherals, so the main addition to the existing PWM header file was functionality for activating multiple PWM channels. In the final implementation, PWM channel 1 was used to control the motor. This corresponds to pin PA24 (see Schematics section). PWM channel 1 was initialized to output a 10,000Hz/60 period signal at a 32/60=53.33% duty cycle, sufficient for driving the motor at a consistent speed.

For obtaining a sense of motor speed from the motor's encoder, the ATSAM's Timer Counter (TC) peripheral was utilized. The accessible TC0 peripheral and its three timer counter channels TC0, TC1, and TC2 were activated and configured in various modes. TC0 was kept in default waveform mode to enable precise and accurate control of delays. TC1 and TC2 were configured in capture mode to count positive edges of selected clock signals. TC1 was configured to select external clock XC1, which was connected to the encoder output of the motor. TC1 would count the edges of the encoder; each time the Hall effect sensor signal (hall pin low) was detected and the LED strip had completed one full rotation, the ATSAM would check the counter value to verify the encoder had recorded a full revolution, then TC1's counter would be reset to 0. TC2 was configured to select internal clock MCK/128, a 312500Hz internal clock signal. TC2's counter value would also be recorded and reset each time the Hall effect signal (hall pin low) was detected. A function would take the recorded counter value of one full rotation and known MCK/128 frequency, and calculate the motor's RPM.

Finally, the ATSAM was responsible for sending appropriately timed control signals to the FPGA to ensure that the LED strip was updated and reset correctly. The ATSAM was configured with two GPIO output pins to send 100us reset and load pulses. These signals were timed using TC's microsecond delay function. The ATSAM would pulse reset initially to communicate to the FPGA to prepare to send the first set of LED control data to the LED strip. Afterwards, the main while loop would appropriately pulse load to send a set of control data and delay between pulses for the LED strip to rotate to the next position for the next set of data.

To ensure a fully displayed message, The delay between load pulses had to be configured for the expected RPM of the LED fan. The motor and the LED strip were run with an estimated delay between load pulses, and the behavior of the resulting image was observed. If the image was not fully displayed, the delay between load signals was too long; if the image did not take up the full fan, the delay between load signals was too short. In this way, a suitable delay between load pulses was found to be 800us.

To ensure a static image, absolute position had to be measured. The main while loop would check for the Hall effect signal (hall pin low) signifying a full rotation with the magnet oriented down and the LED strip oriented up. Once that was detected, the reset signal would be sent, ensuring that LED data would be sent sequentially starting from each time the LED strip was oriented up.

Overall, the control scheme we used was open loop control. Given additional time, closed loop control could have been performed with the encoder output. This is further discussed in the Results and Conclusions section.

FPGA Design

The FPGA logic takes 8 bits at a time from each memory bank and concatenates it into GRB order, resulting in a 24 bit long vector. This vector is shifted out one bit at a time every 1.2 μs. The bit, as described above, controls whether a 4'b1000 or 4'b1100 logic is output. The first bit of this is shifted out every 0.3 μs. Once 24 bits have been shifted out, the GRB bits are updated from memory and the process repeats. Once this process has occurred 10 times, as in the values for 10 leds have been sent out, an internal reset is raised high and the output pin is pulled low while other pieces of logic hold their value. This all waits until the "load pin", an input for the FPGA and an output from the microcontroller is forced high. This triggers the sending of another 10 leds bit data.

Results

The final implementation of the LED fan was successful. The LED fan, controlled by the FPGA and the microcontroller on the uMudd board, successfully displayed words of up to 11 letters legibly and consistently.

In terms of how the microcontroller controlled delays for reset and load, open loop control was performed and the correct delays were manually tuned. In the final implementation, open loop control proved to be sufficient to display visible, stationary, and consistent images/text.

Some glitches were encountered. Each time the LED fan is initially turned on, it takes at least 30 seconds for the motor speed to stabilize to about 670 RPM, or ~11 revolutions per second, the speed for which the 800 us delay between loads is tuned for. The motor speed does not remain precisely at 670 RPM however, so when it speeds up faster than expected the microcontroller sends more loads more than expected per revolution; when it is slower than expected, the microcontroller leaves out several loads at the end of the image. The result is that, for displaying our “FLIP-FLOP” text, when the motor is slightly too fast, an additional “L” is partially displayed between the “FL” and “IP” of “FLIP”; when the motor is slightly too slow, the “I” in “FLIP” at the end of the image is not displayed or only partially displayed. Overall, the RPM ended up being consistent enough after a certain period of the LED fan running to correctly display “FLIP-FLOP”.

With additional time, closed loop control could be performed so that the LED fan would adjust according to the RPM of the motor. The calculated RPM from TC2’s counter value could be used to calculate the time it would take for a full rotation. Then, the total number of loads per revolution and the RPM could be used together to calculate precisely how much delay would be required between loads for any RPM.

We issued a few issues that we couldn’t resolve in time for the project demo. The first was relatively simple, as described in the LED strip section, the images to be displayed were generated in Photoshop. Although these images were exported as maximum size jpegs, there were still some compression issues with a few black pixels, especially those around areas that had significant patches of vibrant color. They would gain a very small RGB value, which wouldn’t matter on a conventional screen with conventional pixels, but due to the intensity of the led strip and the relatively few leds visible, these small imperfections were visible. These could have been filtered out during the Python phase, but there was not enough time to do this as other issues were being ironed out at the same time.

The larger issue that was encountered with the FPGA concerned the updating of the LED bit values. For the value for the first led was repeated on each column of LEDs each sequence, which ended up cutting off the very last LED value. This had to due with timing issues of incrementing the led memory address after the load pin was set. It was a subtle issue that we spent some time trying to resolve at the end of our time but ultimately gave up on in order to make sure other portions of our project were working more smoothly.

Conclusions

Overall, we were able to control an LED fan with the FPGA and microcontroller on the uMudd board to successfully display readable words. In total, the team spent 30-40 hours on the final project, including research, coding, testing and simulating, machining, and fine-tuning/troubleshooting. The header files written by Christopher Ferrarin '20 and Kaveh Pezeshki '21 were indispensable, as was the ATSAM4S Family Datasheet [9]. We were able to build off the existing functionality of the header files, and add functionality for multiple PWM channels and enable capture mode for TC channels.

References

[1] "Addressable High-Density RGB 72-LED Strip, 5V, 0.5m (SK6812)." Pololu LED Strip Product Information, Pololu Corporation 2019. Web. https://www.pololu.com/product/2531

[2] "4.4:1 Metal Gearmotor 25Dx63L mm LP 6V with 48 CPR Encoder." Pololu Gearmotor Product Information, Pololu Corporation 2019. Web. https://www.pololu.com/product/4821

[3] Recktenwald, Gerald. "Basic DC Motor Circuits." Sparkfun DC Motor Tutorial. Sparkfun.com. Web. https://cdn.sparkfun.com/assets/resources/4/4/DC_motor_circuits_slides.pdf

[4] "Complementary Silicon Plastic Power Transistors." TIP31 Series Datasheet, ON Semiconductor, Sept 2015. Web. https://www.onsemi.com/pub/Collateral/TIP31A-D.PDF

[5] "US5881 Unipolar Hall Switch". Melexis Hall Switch Datasheet, Melexis, June 2019. Web. https://www.mouser.com/datasheet/2/734/US5881-Datasheet-Melexis-953437.pdf

[6] "Gravitech MAG-1." Mouser Magnet Product Information, Mouser Electronics, Inc, 2019. Web. https://www.mouser.com/ProductDetail/992-MAG-1

[7] "Op Amp Non-Inverting Amplifier: Operational Amplifier Circuit." electronics-notes.com, 2019. Web. https://www.electronics-notes.com/articles/analogue_circuits/operational-amplifier-op-amp/non-inverting-amplifier.php

[8] "TL08xx JFET-Input Operational Amplifiers." TL08xx Series Datasheet, Texas Instruments, May 2015. Web. http://www.ti.com/lit/ds/symlink/tl082.pdf

[9] "SAM4S Series - Atmel | SMART ARM-based Flash MCU." ATSAM4S Family Datasheet. Atmel, Jun 2015. Web. http://pages.hmc.edu/harris/class/e155/ATSAM4S_Family_Datasheet.pdf

Bill of Materials

LED strip	https://www.pololu.com/product/2531
motor hub	https://www.pololu.com/product/1081
motor w/ encoder	https://www.pololu.com/product/4801

slip ring	https://www.adafruit.com/product/736
Hall effect sensor	https://www.mouser.com/ProductDetail/482-5881LUAAAA000BU
rare earth magnet	https://www.mouser.com/ProductDetail/992-MAG-1

Software

C Code - finalProject.c

```
// finalProject.c
// rzhang@g.hmc.edu, mnara@g.hmc.edu 20 November 2019
//
// motor control with PWM

////////////////////////////////////////////////
// #includes
////////////////////////////////////////////////

#include <stdio.h>
#include "SAM4S4B.h"

////////////////////////////////////////////////
// Constants
////////////////////////////////////////////////

#define HALL_PIN         PIO_PA10
#define HALL_PIN_ON      PIO_PA15
#define HALL_PIN_OFF     PIO_PA16
#define RESET_PIN        PIO_PA8
#define LOAD_PIN         PIO_PA9
//#define LOAD_BUTT_PIN PIO_PA25

////////////////////////////////////////////////
// Helper Functions
////////////////////////////////////////////////
// calculate time of rev in seconds
double calculateTimeOfRev(uint32_t count){
  return count/312500.0;
}
// calculate RPM from time of rev in seconds
int returnRPM(double timeOfRev){
  int temp = 60*1.0/timeOfRev;
  return temp;
}

////////////////////////////////////////////////
// Main
////////////////////////////////////////////////

int main(void) {
  // initialize peripherals
  samInit();
```

```
  pioInit();

  tcDelayInit();
  // "clock divide" = master clock frequency / desired baud rate
  // the phase for the SPI clock is 1 and the polarity is 0

  // initialize a 10,000Hz/60 = 166.67Hz frequency, 32/60=53.33% duty cycle PWM
on channel 1
  // pwm channel 1 corresponds to pin PA24
  pwmInit(1, 10000, 60,32); // 7 of 8 is high enough, 1 of 8 is too low

  // with tachometer and encoder output, measured resulting RPM of fan
  // RPM measured to be 605-615RPM, with everything attached to fan

  //tcCaptureMode(TC_CH1_ID);

  //initialize GPIO pins
  pioPinMode(HALL_PIN, PIO_INPUT);
  pioPinMode(HALL_PIN_ON, PIO_OUTPUT);
  pioPinMode(HALL_PIN_OFF, PIO_OUTPUT);
  //initialize reset pin and load pin
  pioPinMode(RESET_PIN, PIO_OUTPUT);
  pioPinMode(LOAD_PIN, PIO_OUTPUT);

  // begin by setting reset and load low
  pioDigitalWrite(RESET_PIN, PIO_LOW);
  pioDigitalWrite(LOAD_PIN, PIO_LOW);
  tcDelayMillis(1000);

  // burst reset pin
  pioDigitalWrite(RESET_PIN, PIO_HIGH);
  tcDelayMicroseconds(100);
  pioDigitalWrite(RESET_PIN, PIO_LOW);
  // wait
  tcDelayMicroseconds(2000);

  // main while loop
  // sends load signal
  // reads hall effect sensor and resets accordingly
  // reset maintains persistence of vision's absolute position

  int TC_CH1_Counter; // for calculating rpm
  int TC_CH2_Counter; // for calculating rpm
  double timeOfRev;
  int RPM;
  while(1){
    pioDigitalWrite(LOAD_PIN,PIO_HIGH);
    tcDelayMicroseconds(100);
```

```
    pioDigitalWrite(LOAD_PIN,PIO_LOW);
    // wait for fan to complete one full revolution
    // corresponding to 4.4 rotations of the motor
    // 1 rotation of the motor - 12 posedges of the motor encoder

    tcDelayMicroseconds(765);
    // wait for the current counter to reach the previous counter value
    // while(tcReadChannel(TC_CH1_ID)<TC_CH1_Counter);

    // manually tuned 885us delay between loads
    // coupled with fan speed of 605-615 RPM, checked with tachometer

    if(pioDigitalRead(HALL_PIN)==PIO_LOW){ // if hall effect is detected, a
rotation has passed
      pioDigitalWrite(HALL_PIN_ON,PIO_HIGH);
      pioDigitalWrite(HALL_PIN_OFF,PIO_LOW);
      TC_CH1_Counter=tcReadChannel(TC_CH1_ID); // update rotation counter value
      TC_CH2_Counter=tcReadChannel(TC_CH2_ID);
      timeOfRev = calculateTimeOfRev(TC_CH2_Counter);
      RPM = returnRPM(timeOfRev);
      pioDigitalWrite(RESET_PIN, PIO_HIGH);    // burst reset
      tcDelayMicroseconds(100);
      pioDigitalWrite(RESET_PIN, PIO_LOW);
      TC0->TC_CH[1].TC_CCR.SWTRG = 1; // Reset counter
      TC0->TC_CH[2].TC_CCR.SWTRG = 1; // Reset counter
    }
    pioDigitalWrite(HALL_PIN_ON,PIO_LOW);
    pioDigitalWrite(HALL_PIN_OFF,PIO_HIGH);
  }
}
```

C Code - SAM4S4B_pwm.h

```
/* SAM4S4B_pwm.h
 *
 * cferrarin@g.hmc.edu
 * kpezeshki@g.hmc.edu
 * 2/25/2019
 *
 * Contains base address locations, register structs, definitions, and functions for the PWM
 * (Pulse Width Modulation Controller) peripheral of the SAM4S4B microcontroller. */

#ifndef SAM4S4B_PWM_H
#define SAM4S4B_PWM_H

#include <stdint.h>
#include "SAM4S4B_sys.h"
#include "SAM4S4B_pio.h"

////////////////////////////////////////////////////////////////////////////////////////////////
//////
// PWM Base Address Definitions
////////////////////////////////////////////////////////////////////////////////////////////////
//////

#define PWM_BASE (0x40020000U) // PWM Base Address


////////////////////////////////////////////////////////////////////////////////////////////////
//////
// PWM Registers
////////////////////////////////////////////////////////////////////////////////////////////////
//////

// Bit field struct for the PWM_CLK register
typedef struct {
    volatile uint32_t DIVA : 8;
    volatile uint32_t PREA : 4;
    volatile uint32_t      : 4;
    volatile uint32_t DIVB : 8;
    volatile uint32_t PREB : 4;
    volatile uint32_t      : 4;
} PWM_CLK_bits;

// Bit field struct for the PWM_CMR register
typedef struct {
    volatile uint32_t CPRE : 4;
    volatile uint32_t      : 4;
    volatile uint32_t CALG : 1;
    volatile uint32_t CPOL : 1;
    volatile uint32_t CES  : 1;
    volatile uint32_t      : 5;
    volatile uint32_t DTE  : 1;
    volatile uint32_t DTHI : 1;
    volatile uint32_t DTLI : 1;
    volatile uint32_t      : 13;
} PWM_CMR_bits;
```

```
// Channel struct for each of the PWM peripheral's 4 channels
typedef struct {
    volatile PWM_CMR_bits PWM_CMR;       // (PwmCh_num Offset: 0x0) PWM Channel Mode Register
    volatile uint32_t     PWM_CDTY;      // (PwmCh_num Offset: 0x4) PWM Channel Duty Cycle 
Register
    volatile uint32_t     PWM_CDTYUPD;   // (PwmCh_num Offset: 0x8) PWM Channel Duty Cycle 
Update Register
    volatile uint32_t     PWM_CPRD;      // (PwmCh_num Offset: 0xC) PWM Channel Period 
Register
    volatile uint32_t     PWM_CPRDUPD;   // (PwmCh_num Offset: 0x10) PWM Channel Period Update 
Register
    volatile uint32_t     PWM_CCNT;      // (PwmCh_num Offset: 0x14) PWM Channel Counter 
Register
    volatile uint32_t     PWM_DT;        // (PwmCh_num Offset: 0x18) PWM Channel Dead Time 
Register
    volatile uint32_t     PWM_DTUPD;     // (PwmCh_num Offset: 0x1C) PWM Channel Dead Time 
Update Register
} PwmCh;

// Channel struct for each of the PWM peripheral's 8 comparison options
typedef struct {
    volatile uint32_t PWM_CMPV;      // (PwmCmp Offset: 0x0) PWM Comparison x Value Register
    volatile uint32_t PWM_CMPVUPD;   // (PwmCmp Offset: 0x4) PWM Comparison x Value Update 
Register
    volatile uint32_t PWM_CMPM;      // (PwmCmp Offset: 0x8) PWM Comparison x Mode Register
    volatile uint32_t PWM_CMPMUPD;   // (PwmCmp Offset: 0xC) PWM Comparison x Mode Update 
Register
} PwmCmp;

#define PWM_CMP_NUMBER 8
#define PWM_CH_NUMBER 4
// Peripheral struct for the PWM peripheral
typedef struct {
    volatile PWM_CLK_bits PWM_CLK;       // (Pwm Offset: 0x00) PWM Clock Register
    volatile uint32_t   PWM_ENA;       // (Pwm Offset: 0x04) PWM Enable Register
    volatile uint32_t   PWM_DIS;       // (Pwm Offset: 0x08) PWM Disable Register
    volatile uint32_t   PWM_SR;        // (Pwm Offset: 0x0C) PWM Status Register
    volatile uint32_t   PWM_IER1;      // (Pwm Offset: 0x10) PWM Interrupt Enable Register 1
    volatile uint32_t   PWM_IDR1;      // (Pwm Offset: 0x14) PWM Interrupt Disable Register 1
    volatile uint32_t   PWM_IMR1;      // (Pwm Offset: 0x18) PWM Interrupt Mask Register 1
    volatile uint32_t   PWM_ISR1;      // (Pwm Offset: 0x1C) PWM Interrupt Status Register 1
    volatile uint32_t   PWM_SCM;       // (Pwm Offset: 0x20) PWM Sync Channels Mode Register
    volatile uint32_t   Reserved1[1];
    volatile uint32_t   PWM_SCUC;      // (Pwm Offset: 0x28) PWM Sync Channels Update Control 
Register
    volatile uint32_t   PWM_SCUP;      // (Pwm Offset: 0x2C) PWM Sync Channels Update Period 
Register
    volatile uint32_t   PWM_SCUPUPD;   // (Pwm Offset: 0x30) PWM Sync Channels Update Period 
Update Register
    volatile uint32_t   PWM_IER2;      // (Pwm Offset: 0x34) PWM Interrupt Enable Register 2
    volatile uint32_t   PWM_IDR2;      // (Pwm Offset: 0x38) PWM Interrupt Disable Register 2
    volatile uint32_t   PWM_IMR2;      // (Pwm Offset: 0x3C) PWM Interrupt Mask Register 2
    volatile uint32_t   PWM_ISR2;      // (Pwm Offset: 0x40) PWM Interrupt Status Register 2
    volatile uint32_t   PWM_OOV;       // (Pwm Offset: 0x44) PWM Output Override Value 
Register
    volatile uint32_t   PWM_OS;        // (Pwm Offset: 0x48) PWM Output Selection Register
```

```
    volatile uint32_t   PWM_OSS;       // (Pwm Offset: 0x4C) PWM Output Selection Set Register
    volatile uint32_t   PWM_OSC;       // (Pwm Offset: 0x50) PWM Output Selection Clear
Register
    volatile uint32_t   PWM_OSSUPD;    // (Pwm Offset: 0x54) PWM Output Selection Set Update
Register
    volatile uint32_t   PWM_OSCUPD;    // (Pwm Offset: 0x58) PWM Output Selection Clear Update
Register
    volatile uint32_t   PWM_FMR;       // (Pwm Offset: 0x5C) PWM Fault Mode Register
    volatile uint32_t   PWM_FSR;       // (Pwm Offset: 0x60) PWM Fault Status Register
    volatile uint32_t   PWM_FCR;       // (Pwm Offset: 0x64) PWM Fault Clear Register
    volatile uint32_t   PWM_FPV;       // (Pwm Offset: 0x68) PWM Fault Protection Value
Register
    volatile uint32_t   PWM_FPE;       // (Pwm Offset: 0x6C) PWM Fault Protection Enable
Register
    volatile uint32_t   Reserved2[3];
    volatile uint32_t   PWM_ELMR[2];   // (Pwm Offset: 0x7C) PWM Event Line 0 Mode Register
    volatile uint32_t   Reserved3[11];
    volatile uint32_t   PWM_SMMR;      // (Pwm Offset: 0xB0) PWM Stepper Motor Mode Register
    volatile uint32_t   Reserved4[12];
    volatile uint32_t   PWM_WPCR;      // (Pwm Offset: 0xE4) PWM Write Protect Control
Register
    volatile uint32_t   PWM_WPSR;      // (Pwm Offset: 0xE8) PWM Write Protect Status Register
    volatile uint32_t   Reserved5[7];
    volatile uint32_t   PWM_TPR;       // (Pwm Offset: 0x108) Transmit Pointer Register
    volatile uint32_t   PWM_TCR;       // (Pwm Offset: 0x10C) Transmit Counter Register
    volatile uint32_t   Reserved6[2];
    volatile uint32_t   PWM_TNPR;      // (Pwm Offset: 0x118) Transmit Next Pointer Register
    volatile uint32_t   PWM_TNCR;      // (Pwm Offset: 0x11C) Transmit Next Counter Register
    volatile uint32_t   PWM_PTCR;      // (Pwm Offset: 0x120) Transfer Control Register
    volatile uint32_t   PWM_PTSR;      // (Pwm Offset: 0x124) Transfer Status Register
    volatile uint32_t   Reserved7[2];
    volatile PwmCmp     PWM_CMP[PWM_CMP_NUMBER]; // (Pwm Offset: 0x130) 0 .. 7
    volatile uint32_t   Reserved8[20];
    volatile PwmCh      PWM_CH[PWM_CH_NUMBER]; // (Pwm Offset: 0x200) ch = 0 .. 3
} Pwm;

// Pointer to a Pwm-sized chunk of memory at the PWM peripheral
#define PWM ((Pwm*) PWM_BASE)


///////////////////////////////////////////////////////////////////////////////////////////
//////
// PWM Definitions
///////////////////////////////////////////////////////////////////////////////////////////
//////

// Constants relating to clock speed tuning in pwmInit()
#define PWM_CLK_PRE_MAX 11
#define PWM_CLK_DIV_MAX 255

// The specific PIO pins and peripheral function which PWM uses, set in pwmInit()
// these are the same as NPCS, MISO, MOSI, and SPCK for SPI. does it just multiplex? this
seems bad.
// changed from the old ones
#define PWM_CH0_PIN PIO_PA23//PIO_PA11
#define PWM_CH1_PIN PIO_PA24//PIO_PA12
#define PWM_CH2_PIN PIO_PA13
```

```
#define PWM_CH3_PIN PIO_PA14
#define PWM_FUNC    PIO_PERIPH_B

// Values which "channelID" can take on in several functions
#define PWM_CH0 0
#define PWM_CH1 1
#define PWM_CH2 2
#define PWM_CH3 3

// Values which the CPOL bit in the PWM_CMR register can take on
#define PWM_CMR_CPOL_LOW  0 // Output waveform starts at a low level
#define PWM_CMR_CPOL_HIGH 1 // Output waveform starts at a high level

// Values which the CPRE bits in the PWM_CMR register can take on
#define PWM_CMR_CPRE_MCK     0
#define PWM_CMR_CPRE_MCK2    1
#define PWM_CMR_CPRE_MCK4    2
#define PWM_CMR_CPRE_MCK8    3
#define PWM_CMR_CPRE_MCK16   4
#define PWM_CMR_CPRE_MCK32   5
#define PWM_CMR_CPRE_MCK64   6
#define PWM_CMR_CPRE_MCK128  7
#define PWM_CMR_CPRE_MCK256  8
#define PWM_CMR_CPRE_MCK512  9
#define PWM_CMR_CPRE_MCK1024 10
#define PWM_CMR_CPRE_CLKA    11
#define PWM_CMR_CPRE_CLKB    12

// Writing any other value in this field aborts the write operation of the WPEN bit.
// Always reads as 0.
#define PWM_WPCR_WPKEY_PASSWD (0x50574DU << 8)


//////////////////////////////////////////////////////////////////////////////////////////////
//////
// PWM User Functions
//////////////////////////////////////////////////////////////////////////////////////////////
//////

/* Enables the PWM peripheral and initializes its frequency, period, and duty cycle.
 * Requires pioInit().
 *    -- freq: the desired frequency of the PWM clock in Hz
 *    -- period: the desired frequency of the PWM waveform in number of clock periods
 *    -- dutyCycle: the desired duty cycle of the PWM waveform in number of waveform periods
 * Note: the actual frequency of the PWM waveform is given by freq / period, where
 * 0 < period < (2^16 = 65536). The higher the period, the more resolution for the duty cycle,
 * which is given by Duty Cycle = dutyCycle / period, where 0 < period < (2^16 = 65536). Note
that
 * 15.319 Hz <= freq <= 4 MHz based on allowable clock divisions. The alignment defaults to
 * left-aligned, and so is not set. */
void pwmInit(int channelID, int freq, uint16_t period, uint16_t dutyCycle) {
    pmcEnablePeriph(PMC_ID_PWM);
    pioInit();

    switch (channelID) {
        case PWM_CH0: pioPinMode(PWM_CH0_PIN, PWM_FUNC); break;
        case PWM_CH1: pioPinMode(PWM_CH1_PIN, PWM_FUNC); break;
```

```
        case PWM_CH2: pioPinMode(PWM_CH2_PIN, PWM_FUNC); break;
        case PWM_CH3: pioPinMode(PWM_CH3_PIN, PWM_FUNC); break;
    }

    PWM->PWM_DIS |= (1 << channelID); // Disables PWM while setting values

    // Finds prescaler and linear divider values
    uint32_t preScl[PWM_CLK_PRE_MAX] = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024};
    uint32_t preSclIndex = 0;
    uint32_t linDiv;
    while (preSclIndex < PWM_CLK_PRE_MAX) {
        linDiv = MCK_FREQ / preScl[preSclIndex] / freq;
        if (linDiv <= PWM_CLK_DIV_MAX) break;
        preSclIndex++;
    }
    // check: channel 0, clk A, channel 1, clk b
    // Sets the clock if a configuration can be found. Otherwise, disables the clock.
    if (channelID == PWM_CH0){
      if (preSclIndex < PWM_CLK_PRE_MAX) {
          PWM->PWM_CLK.PREA = preSclIndex;
          PWM->PWM_CLK.DIVA = linDiv;
      } else {
          PWM->PWM_CLK.DIVA = 0;
      }

      PWM->PWM_CH[channelID].PWM_CMR.CPRE = PWM_CMR_CPRE_CLKA; // Set base clock speed
    }else if (channelID == PWM_CH1){
      if (preSclIndex < PWM_CLK_PRE_MAX) {
          PWM->PWM_CLK.PREB = preSclIndex;
          PWM->PWM_CLK.DIVB = linDiv;
      } else {
          PWM->PWM_CLK.DIVB = 0;
      }

      PWM->PWM_CH[channelID].PWM_CMR.CPRE = PWM_CMR_CPRE_CLKB; // Set base clock speed
    }

    PWM->PWM_CH[channelID].PWM_CMR.CPOL = PWM_CMR_CPOL_HIGH; // Set waveform polarity

    PWM->PWM_CH[channelID].PWM_CPRD = period; // Set period
    PWM->PWM_CH[channelID].PWM_CDTY = dutyCycle; // Set duty cycle

    PWM->PWM_ENA |= (1 << channelID); // Enable PWM after setting values
}

#endif
```

C Code - SAM4S4B_tc.h

```
/* SAM4S4B_tc.h
 * edited by rzhang@g.hmc.edu mnara@hmc.edu December 2019
 *
 * cferrarin@g.hmc.edu
 * kpezeshki@g.hmc.edu
 * 2/25/2019
 *
 * Contains base address locations, register structs, definitions, and functions for the TC
(Timer
 * Counter) peripheral of the SAM4S4B microcontroller. */

#ifndef SAM4S4B_TC_H
#define SAM4S4B_TC_H

#include <stdint.h>
#include "SAM4S4B_sys.h"
#include "SAM4S4B_pmc.h"
#include "SAM4S4B_pio.h"

////////////////////////////////////////////////////////////////////////////////////////////////
//////
// TC Base Address Definitions
////////////////////////////////////////////////////////////////////////////////////////////////
//////

#define TC0_BASE   (0x40010000U) // TC0 Base Address
#define TC1_BASE   (0x40014000U) // TC1 Base Address


////////////////////////////////////////////////////////////////////////////////////////////////
//////
// TC Registers
////////////////////////////////////////////////////////////////////////////////////////////////
//////

// Bit field struct for the TC_CCR register
typedef struct {
    volatile uint32_t CLKEN   : 1;
    volatile uint32_t CLKDIS  : 1;
    volatile uint32_t SWTRG   : 1;
    volatile uint32_t         : 29;
} TC_CCR_bits;

// Bit field struct for the TC_CMR Capture Mode register
typedef struct {
    volatile uint32_t TCCLKS  : 3;
    volatile uint32_t CLKI    : 1;
    volatile uint32_t BURST   : 2;
    volatile uint32_t LDBSTOP : 1;
    volatile uint32_t LDBDIS  : 1;
    volatile uint32_t ETRGEDG : 2;
    volatile uint32_t ABETRG  : 1;
    volatile uint32_t         : 3;
    volatile uint32_t CPCTRG  : 1;
```

```
    volatile uint32_t WAVE    : 1;
    volatile uint32_t LDRA    : 2;
    volatile uint32_t LDRB    : 2;
    volatile uint32_t         : 12;
} TC_CMR_CM_bits;

// Bit field struct for the TC_CMR Waveform Mode register
typedef struct {
    volatile uint32_t TCCLKS  : 3;
    volatile uint32_t CLKI    : 1;
    volatile uint32_t BURST   : 2;
    volatile uint32_t CPCSTOP : 1;
    volatile uint32_t CPCDIS  : 1;
    volatile uint32_t EEVTEDG : 2;
    volatile uint32_t EEVT    : 2;
    volatile uint32_t ENETRG  : 1;
    volatile uint32_t WAVESEL : 2;
    volatile uint32_t WAVE    : 1;
    volatile uint32_t ACPA    : 2;
    volatile uint32_t ACPC    : 2;
    volatile uint32_t AEEVT   : 2;
    volatile uint32_t ASWTRG  : 2;
    volatile uint32_t BCPB    : 2;
    volatile uint32_t BCPC    : 2;
    volatile uint32_t BEEVT   : 2;
    volatile uint32_t BSWTRG  : 2;
} TC_CMR_bits;

// Bit field struct for the TC_SR register
typedef struct {
    volatile uint32_t COVFS   : 1;
    volatile uint32_t LOVRS   : 1;
    volatile uint32_t CPAS    : 1;
    volatile uint32_t CPBS    : 1;
    volatile uint32_t CPCS    : 1;
    volatile uint32_t LDRAS   : 1;
    volatile uint32_t LDRBS   : 1;
    volatile uint32_t ETRGS   : 1;
    volatile uint32_t         : 8;
    volatile uint32_t CLKSTA  : 1;
    volatile uint32_t MTIOA   : 1;
    volatile uint32_t MTIOB   : 1;
    volatile uint32_t         : 13;
} TC_SR_bits;

// Bit field struct for the TC_BMR register
typedef struct {
    volatile uint32_t TC0XC0S : 2;
    volatile uint32_t TC1XC1S : 2;
    volatile uint32_t TC2XC2S : 2;
    volatile uint32_t         : 2;
    volatile uint32_t QDEN    : 1;
    volatile uint32_t POSEN   : 1;
    volatile uint32_t SPEEDEN : 1;
    volatile uint32_t QDTRANS : 1;
    volatile uint32_t EDGPHA  : 1;
    volatile uint32_t INVA    : 1;
```

```
    volatile uint32_t INVB    : 1;
    volatile uint32_t INVIDX  : 1;
    volatile uint32_t SWAP    : 1;
    volatile uint32_t IDXPHB  : 1;
    volatile uint32_t         : 2;
    volatile uint32_t MAXFILT : 6;
    volatile uint32_t         : 6;
} TC_BMR_bits;

// Channel struct for each of the 3 TC channels, Capture Mode
typedef struct {
    volatile TC_CCR_bits TC_CCR;        // (TcChannel Offset: 0x0) Channel Control Register
    volatile TC_CMR_CM_bits TC_CMR;        // (TcChannel Offset: 0x4) Channel Mode Register
    volatile uint32_t    TC_SMMR;       // (TcChannel Offset: 0x8) Stepper Motor Mode Register
    volatile uint32_t    Reserved1[1];
    volatile uint32_t    TC_CV;         // (TcChannel Offset: 0x10) Counter Value
    volatile uint32_t    TC_RA;         // (TcChannel Offset: 0x14) Register A
    volatile uint32_t    TC_RB;         // (TcChannel Offset: 0x18) Register B
    volatile uint32_t    TC_RC;         // (TcChannel Offset: 0x1C) Register C
    volatile TC_SR_bits  TC_SR;         // (TcChannel Offset: 0x20) Status Register
    volatile uint32_t    TC_IER;        // (TcChannel Offset: 0x24) Interrupt Enable Register
    volatile uint32_t    TC_IDR;        // (TcChannel Offset: 0x28) Interrupt Disable Register
    volatile uint32_t    TC_IMR;        // (TcChannel Offset: 0x2C) Interrupt Mask Register
    volatile uint32_t    Reserved2[4];
} TcCh_CM;

// Channel struct for each of the 3 TC channels, Waveform Mode
typedef struct {
    volatile TC_CCR_bits TC_CCR;        // (TcChannel Offset: 0x0) Channel Control Register
    volatile TC_CMR_bits TC_CMR;        // (TcChannel Offset: 0x4) Channel Mode Register
    volatile uint32_t    TC_SMMR;       // (TcChannel Offset: 0x8) Stepper Motor Mode Register
    volatile uint32_t    Reserved1[1];
    volatile uint32_t    TC_CV;         // (TcChannel Offset: 0x10) Counter Value
    volatile uint32_t    TC_RA;         // (TcChannel Offset: 0x14) Register A
    volatile uint32_t    TC_RB;         // (TcChannel Offset: 0x18) Register B
    volatile uint32_t    TC_RC;         // (TcChannel Offset: 0x1C) Register C
    volatile TC_SR_bits  TC_SR;         // (TcChannel Offset: 0x20) Status Register
    volatile uint32_t    TC_IER;        // (TcChannel Offset: 0x24) Interrupt Enable Register
    volatile uint32_t    TC_IDR;        // (TcChannel Offset: 0x28) Interrupt Disable Register
    volatile uint32_t    TC_IMR;        // (TcChannel Offset: 0x2C) Interrupt Mask Register
    volatile uint32_t    Reserved2[4];
} TcCh;

#define TC_CH_NUMBER 3 // Number of TC channels
// Peripheral struct for a TC peripheral (TC0, Capture mode)
typedef struct {
    TcCh_CM             TC_CH[TC_CH_NUMBER]; // (Tc Offset: 0x0) channel = 0 .. 2
    volatile uint32_t TC_BCR;             // (Tc Offset: 0xC0) Block Control Register
    volatile TC_BMR_bits TC_BMR;             // (Tc Offset: 0xC4) Block Mode Register
    volatile uint32_t TC_QIER;            // (Tc Offset: 0xC8) QDEC Interrupt Enable Register
    volatile uint32_t TC_QIDR;            // (Tc Offset: 0xCC) QDEC Interrupt Disable
Register
    volatile uint32_t TC_QIMR;            // (Tc Offset: 0xD0) QDEC Interrupt Mask Register
    volatile uint32_t TC_QISR;            // (Tc Offset: 0xD4) QDEC Interrupt Status Register
    volatile uint32_t TC_FMR;             // (Tc Offset: 0xD8) Fault Mode Register
    volatile uint32_t Reserved1[2];
    volatile uint32_t TC_WPMR;            // (Tc Offset: 0xE4) Write Protect Mode Register
```

```
} Tc_CM;

// Peripheral struct for a TC peripheral (TC1, Waveform Mode)
typedef struct {
    TcCh              TC_CH[TC_CH_NUMBER]; // (Tc Offset: 0x0) channel = 0 .. 2
    volatile uint32_t TC_BCR;              // (Tc Offset: 0xC0) Block Control Register
    volatile TC_BMR_bits TC_BMR;              // (Tc Offset: 0xC4) Block Mode Register
    volatile uint32_t TC_QIER;             // (Tc Offset: 0xC8) QDEC Interrupt Enable Register
    volatile uint32_t TC_QIDR;             // (Tc Offset: 0xCC) QDEC Interrupt Disable
Register
    volatile uint32_t TC_QIMR;             // (Tc Offset: 0xD0) QDEC Interrupt Mask Register
    volatile uint32_t TC_QISR;             // (Tc Offset: 0xD4) QDEC Interrupt Status Register
    volatile uint32_t TC_FMR;              // (Tc Offset: 0xD8) Fault Mode Register
    volatile uint32_t Reserved1[2];
    volatile uint32_t TC_WPMR;             // (Tc Offset: 0xE4) Write Protect Mode Register
} Tc;

// Pointers to Tc-sized chunks of memory at each TC peripheral
#define TC0 ((Tc*) TC0_BASE)
#define TC1 ((Tc*) TC1_BASE)


//////////////////////////////////////////////////////////////////////////////////////////////
//////
// TC Definitions
//////////////////////////////////////////////////////////////////////////////////////////////
//////

// Clock speeds for the 5 TC clocks used in generating delays
#define TC_CLK1_SPEED (MCK_FREQ / 2)
#define TC_CLK2_SPEED (MCK_FREQ / 8)
#define TC_CLK3_SPEED (MCK_FREQ / 32)
#define TC_CLK4_SPEED (MCK_FREQ / 128)
#define TC_CLK5_SPEED 32000

// Values which TCCLKS bits can take on in TC_CMR
#define TC_CLK1_ID 0
#define TC_CLK2_ID 1
#define TC_CLK3_ID 2
#define TC_CLK4_ID 3
#define TC_CLK5_ID 4
#define TC_XC0_ID  5
#define TC_XC1_ID  6
#define TC_XC2_ID  7

// Arbitrary block IDs used to easily find a channel's block
#define TC_BLOCK0_ID 0
#define TC_BLOCK1_ID 1

// The specific PIO pins and peripheral function which are needed for TC
// external clock input pins
#define TC_TC0_TCLK0 PIO_PA4
#define TC_TC0_TCLK1 PIO_PA28
#define TC_TC0_TIOA2 PIO_PA26
#define TC_FUNC    PIO_PERIPH_B

// Values which "channelID" can take on in several functions
```

```
#define TC_CH0_ID 0
#define TC_CH1_ID 1
#define TC_CH2_ID 2
#define TC_CH3_ID 3
#define TC_CH4_ID 4
#define TC_CH5_ID 5

// Values which the WAVESEL bits can take on in TC_CMR
#define TC_MODE_UP        0 // The counter increases then resets low once it caps out
#define TC_MODE_UPDOWN    1 // The counter increases then decreases once it caps out
#define TC_MODE_UP_RC     2 // The counter increases then resets low when an RC match occurs
#define TC_MODE_UPDOWN_RC 3 // The counter increases then decreases when an RC match occurs

// Writing any other value in this field aborts the write operation of the WPEN bit.
// Always reads as 0.
#define TC_WPMR_WPKEY_PASSWD (0x54494Du << 8)


///////////////////////////////////////////////////////////////////////////////////////////////
//////
// TC Helper Functions
///////////////////////////////////////////////////////////////////////////////////////////////
//////

/* Returns the TC block ID that corresponds to a given channel.
 *    -- channelID: a TC channel ID, e.g. TC_CH3_ID
 *    -- return: a TC block ID, e.g. TC_BLOCK1_ID */
int tcChannelToBlock(int channelID) {
    return channelID / 3;
}

/* Returns a pointer to the given block's base address.
 *    -- block: a TC block ID, e.g. TC_BLOCK1_ID
 *    -- return: a pointer to a Tc-sized block of memory at the block "block" */
Tc* tcBlockToBlockBase(int block) {
    return (block ? TC1 : TC0);
}

/* Given a channel, returns a pointer to the corresponding block's base address.
 *    -- channelID: a TC channel ID, e.g. TC_CH3_ID
 *    -- return: a pointer to a Tc-sized block of memory at the block "block" */
Tc* tcChannelToBlockBase(int channelID) {
    return tcBlockToBlockBase(tcChannelToBlock(channelID));
}


///////////////////////////////////////////////////////////////////////////////////////////////
//////
// TC User Functions - Timer/Counter (Lower Level; See Delay Functions Below)
///////////////////////////////////////////////////////////////////////////////////////////////
//////

/* Initializes the TC peripheral by enabling the Master Clock to TC0 and TC1. */
void tcInit() {
    // enable the two peripherals, three channels each for 6 channels
    pmcEnablePeriph(PMC_ID_TC0);
    pmcEnablePeriph(PMC_ID_TC1);
```

```
    pmcEnablePeriph(PMC_ID_TC2);
               // enable external clock input pins
               pioPinMode(TC_TC0_TCLK0, TC_FUNC);
               pioPinMode(TC_TC0_TCLK1, TC_FUNC);
    pioPinMode(TC_TC0_TIOA2, TC_FUNC);
}

/* Enables a TC channel and configures it with the desired clock and mode.
 *    -- channelID: a TC channel ID, e.g. TC_CH3_ID
 *    -- clock: a TC clock ID, e.g. TC_CLK3_ID
 *    -- mode: a TC mode ID, e.g. TC_MODE_UP_RC */
void tcChannelInit(int channelID, uint32_t clock, uint32_t mode, uint32_t wave) {
    Tc* block = tcChannelToBlockBase(channelID);
    int chInd = channelID % TC_CH_NUMBER;
    block->TC_CH[chInd].TC_CCR.CLKEN   = 1;     // Enable clock
    block->TC_CH[chInd].TC_CMR.TCCLKS  = clock; // Set clock to desired clock
    block->TC_CH[chInd].TC_CMR.WAVE    = wave;     // Waveform mode
    if(wave){
      block->TC_CH[chInd].TC_CMR.WAVESEL = mode;  // Set counting mode to desired mode
    }
}

/* Enables TC channel 1 for external clock 1.
 *    */
void tcExtInitCh1() {
    Tc* block = tcChannelToBlockBase(TC_CH1_ID);
    int chInd = TC_CH1_ID % TC_CH_NUMBER;
    block->TC_BMR.TC1XC1S = 0;     // Enable external clock XC1
    block->TC_CH[chInd].TC_CCR.CLKEN   = 1;     // Enable clock?
    block->TC_CH[chInd].TC_CMR.TCCLKS  = TC_XC1_ID; // Set clock to XC1
    block->TC_CH[chInd].TC_CMR.WAVE    = 0;     // Capture mode ??
}

/* Reads the current value of the counter of a given channel.
 *    -- channel ID: a TC channel ID, e.g. TC_CH3_ID
 *    -- return: the value (32-bit unsigned integer) in channel "channelID"'s counter */
uint32_t tcReadChannel(int channelID) {
    Tc* block = tcChannelToBlockBase(channelID);
    int chInd = channelID % TC_CH_NUMBER;
    return block->TC_CH[chInd].TC_CV;
}

/* Resets the counter of a given channel to zero, at which point it continues counting.
 *    -- channel ID: a TC channel ID, e.g. TC_CH3_ID */
void tcResetChannel(int channelID) {
    Tc* block = tcChannelToBlockBase(channelID);
    int chInd = channelID % TC_CH_NUMBER;
    block->TC_CH[chInd].TC_CCR.SWTRG = 1;
}

/* Sets the value of the RC compare register for a given channel, relevant to certain TC
modes.
 *    -- channel ID: a TC channel ID, e.g. TC_CH3_ID
 *    -- val: the value (32-bit unsigned integer) to write to the RC register */
void tcSetRC_compare(int channelID, uint32_t val) {
    Tc* block = tcChannelToBlockBase(channelID);
    int chInd = channelID % TC_CH_NUMBER;
```

```
    block->TC_CH[chInd].TC_RC = val;
}

/* Checks whether an RC match has occurred since the last call to tcCheckRC_compare().
 *    -- channel ID: a TC channel ID, e.g. TC_CH3_IC
 *    -- return: 1 if an RC match has occurred since the last read; 0 if it hasn't */
int tcCheckRC_compare(int channelID) {
    Tc* block = tcChannelToBlockBase(channelID);
    int chInd = channelID % TC_CH_NUMBER;
    return block->TC_CH[chInd].TC_SR.CPCS;
}


//////////////////////////////////////////////////////////////////////////////////////////////
//////
// TC User Functions - Delay Unit (Higher Level)
//////////////////////////////////////////////////////////////////////////////////////////////
//////

/* Configures TC Channel 0 to perform delays using the fastest clock and RC compares. Does not
 * require the user to call any lower-level functions such as tcInit(). */
void tcDelayInit() {
    tcInit();
    // enable channel 0 clock MCK/2, reset when RC match occurs
    // enable channel 2 capture mode clock MCK/128
    // enable channel 1 capture mode clock XCK1 TCLK1
    tcChannelInit(TC_CH0_ID, TC_CLK1_ID, TC_MODE_UP_RC, 1);
    tcChannelInit(TC_CH2_ID, TC_CLK4_ID, TC_MODE_UP_RC, 0); // need a slower clock
    tcExtInitCh1();
}

/* Delays the system by a specified number of microseconds
 *    -- duration: the number of microseconds to delay
 * Note: This works up to (2^16 - 1 = 65535) us. Using the fastest available clock,
TC_CLK1_ID,
 * we achieve a resolution of 0.5 us. Also note that this doesn't use the lower-level
functions
 * in order to optimize speed; ideally, it would be written in assembly language for further
 * optimization. Requires that tcDelayInit() be called previously.
 *
 * CAUTION: If master clock speed is NOT the default 4 MHz, the constant in the line with
 * comment "set compare value", which is currently 2, must be changed to TC_CKL1_SPEED / 1e6.
*/
void tcDelayMicroseconds(uint32_t duration) {
    // switch to TC1 because TC0 will be used for capture mode speed measure?
    TC0->TC_CH[0].TC_CCR.SWTRG = 1; // Reset counter
    TC0->TC_CH[0].TC_RC = duration * 20; // Set compare value
    while(!(TC0->TC_CH[0].TC_SR.CPCS)); // Wait until an RC Compare has occurred
}

/* Delays the system by a specified number of milliseconds
 *    -- duration: the number of milliseconds to delay
 * Note: The dependence on a "for" loop makes this code less efficient than tcDelayMicros(),
and
 * so should be avoided for durations shorter than 65 milliseconds, in which case
tcDelayMicros()
 * is the better option. Requires that tcDelayInit() be called previously. */
```

```
void tcDelayMillis(int duration) {
    for (int i = 0; i < duration; i++) {
        tcDelayMicroseconds(1000);
    }
}

/* Delays the system by a specified number of seconds
 *    -- duration: the number of seconds to delay
 * Note: the dependence on nested "for" loops and function calls makes this code extremely
 * inefficient, and so should be avoided for durations shorter than a minute. Requires that
 * tcDelayInit() be called previously. */
void tcDelaySeconds(int duration) {
    for (int i = 0; i < duration; i++) {
        tcDelayMillis(1000);
    }
}

#endif
```

Python Code

```
import numpy as np
import cv2

FILE_IN = 'test_strip_7.jpg'

def main():

   img = cv2.imread(FILE_IN, 1); #load image as rgb no alpha channel
    arr1 = np.array(img)
   arr2 = np.swapaxes(arr1,0,2)

   b = arr2[0:1]
   g = arr2[1:2]
   r = arr2[2:3]

   b = b.flatten()
   g = g.flatten()
   r = r.flatten()

   np.savetxt('ts7/rvals.txt', r, fmt='%2.2x')
   np.savetxt('ts7/bvals.txt', b, fmt='%2.2x')
   np.savetxt('ts7/gvals.txt', g, fmt='%2.2x')

def toHex(a):
   vhex = np.vectorize(hex)
   b = vhex(a)
   return b



if __name__ == '__main__':
   main()
```

Verilog

```
module ledstrip (input  logic clk, reset,
                input logic load,
                output logic wave_out);

logic rst_internal; // holds pins low while waiting for load pin to
go high

logic end_col; // pulse that goes high after values for 10 leds are
sent

logic [31:0] led_counter; // counter that demarks each individual led
logic led_stb; // strobe for the beginning of each led

logic[4:0] bit_counter, bit_c_next; // keeps track of how many bits
have been sent

logic begin_pix; //pulses high when a pixel starts

logic led_bit; // actual bit value being sent, gets decoded into
waveform

logic [3:0] led_wave; // value that holds the waveform for a 1 or 0
for the leds

logic [31:0] wave_counter; // counter that should iterate 4 times per
led
logic wave_stb; //strobe that pules 4 times per led

logic [1:0] wave_state; //whole number iterator 1-4 during each led

logic [9:0] led_add, led_next; // current memory address for the
r/g/b memory
logic [7:0] red_val, grn_val, blu_val; // memory output for the r/g/b
memory
logic [23:0] led_val, leds; // led pixel values and holder for next
set of leds

          //block to create clocks and pulses
          always_ff @(posedge clk)
               begin
```

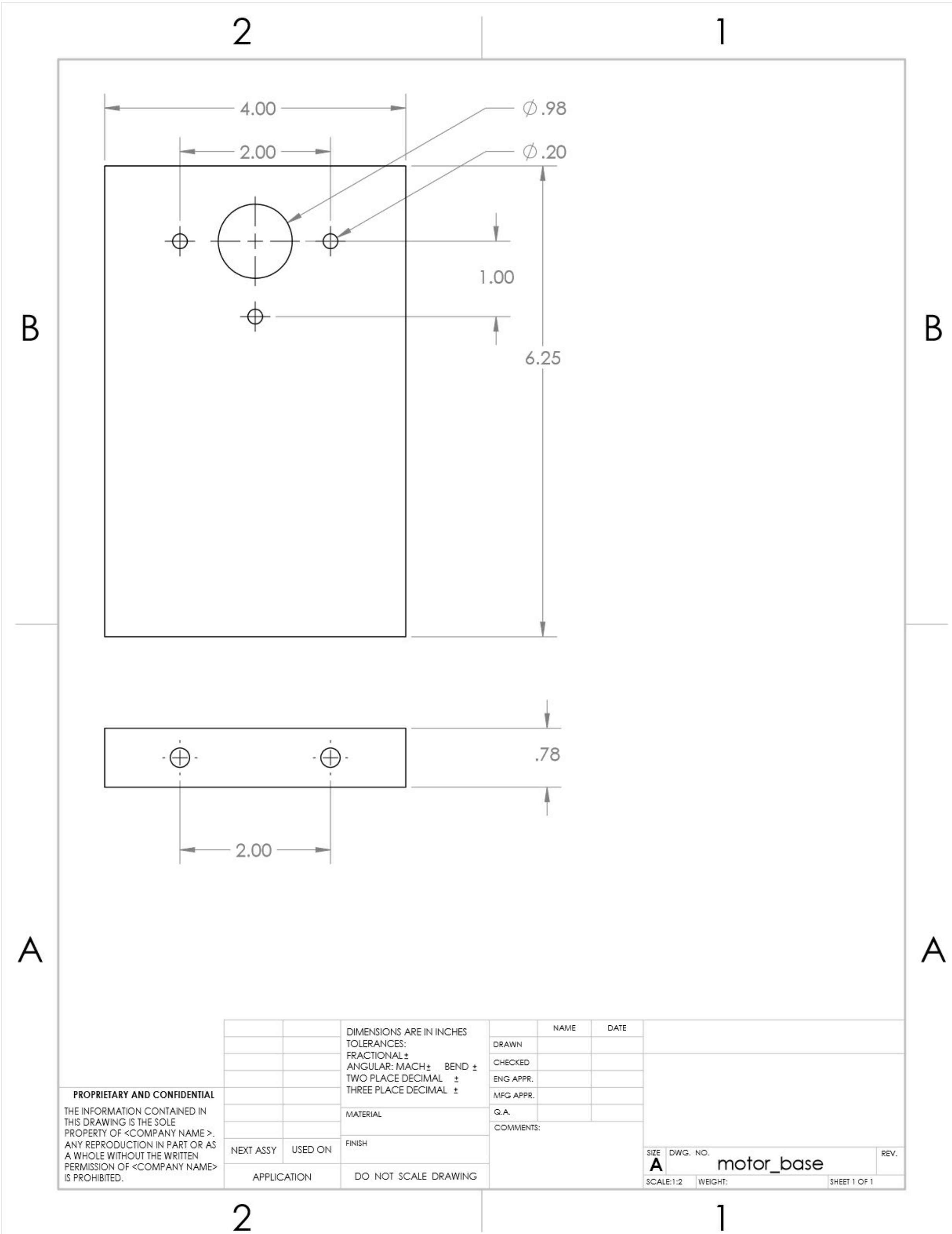
```
                    // update wave counter
                    if (reset | rst_internal) wave_counter <= 1'b0;
                    else {wave_stb, wave_counter} <= wave_counter +
32'h15555555;

                    // counter to generate clock with period 1.2
microseconds
                    if (reset | rst_internal) led_counter <=
32'hFFFFFFFF;
                    else {led_stb,led_counter} <= led_counter +
32'h05555555;

                    // counter to increment the led address
                    if (reset) led_add <= 1'b0;
                    else if(begin_pix & led_stb & !rst_internal)
led_add <= led_next;

                    // wave state control
                    if (led_stb) wave_state <= 2'b11;
                    else if (wave_stb) wave_state <= wave_state -
1'b1;

                    // internal reset control
                    if (end_col & !load) rst_internal <= 1'b1;
                    else if (load | reset) rst_internal <= 1'b0;

                    // update led values;
                    if (reset | rst_internal)
                         begin
                              leds = led_val;
                              led_bit = leds[0];
                         end
                    else if (begin_pix & led_stb) leds = led_val;
                    else if (led_stb) {led_bit, leds} = {leds,
reset};
               end

          //updating the bit counter and led values on neg edge to
prevent race condition
          always_ff @(negedge clk)
               begin
                    // update the bit counter
                    if (reset | rst_internal) bit_counter = 1'b0;
```

```
                    else if (led_stb) bit_counter = bit_c_next;

               end

          //creating string of all the led values
          assign led_val = {grn_val, red_val, blu_val};

          //modules for memory to access the predetermined values
          rpixvals rpv(clk, led_add, red_val);
          gpixvals gpv(clk, led_add, grn_val);
          bpixvals bpv(clk, led_add, blu_val);

          //controller to set the flags needed for logic in the ff
blocks
          controller con(clk, reset, rst_internal, led_bit,
wave_state, led_add, bit_counter, led_wave, led_next, bit_c_next,
end_col, begin_pix, wave_out);

endmodule

module controller(input  logic clk, reset, rst_internal,
                                input  logic led_bit,
                                input  logic [1:0] wave_state,
                                input  logic [9:0] led_add,
                                input  logic [4:0] bit_counter,
                                output logic [3:0] led_wave,
                                output logic [9:0] led_next,
                                output logic [4:0] bit_c_next,
                                output logic end_col,
                                output logic begin_pix,
                                output logic wave_out);

          always_comb
               begin
                    // actual wave out logic
                    if(reset | rst_internal) wave_out = 1'b0;
                    else wave_out = led_wave[wave_state];

                    // check if at end of file
                    if (led_add == 10'b1111101000) led_next = 1'b0;
                    else led_next = led_add + 1'b1;
```

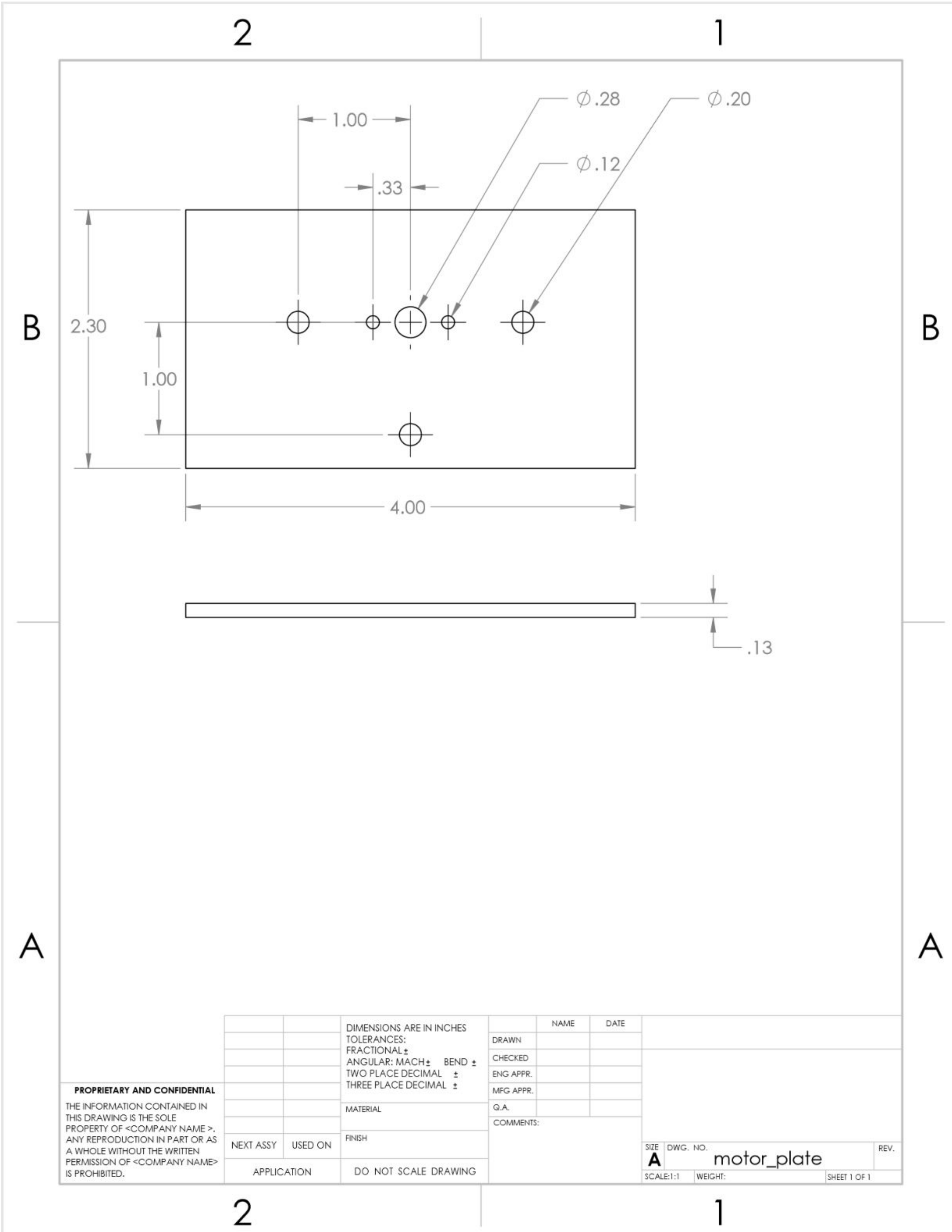
```
                    //check if finished with column, should
activate reset code
                    if (led_add % 10      == 0 & led_add > 0 &
wave_out == 1) end_col = 1'b1;
                    else end_col = 1'b0;

                    //set the begin pixel high at the beginning of
each pixel
                    if (bit_counter == 1'b0) begin_pix = 1'b1;
                    else begin_pix = 1'b0;

                    //see if we've sent all 24 bits, reset to zero
if we have
                    if (bit_counter == 5'b10111) bit_c_next = 1'b0;
                    else bit_c_next = bit_counter + 1;

                    if (led_bit) led_wave = 4'b1100;
                    else led_wave = 4'b1000;
               end
endmodule

module rpixvals(input logic clk,
                         input  logic [9:0] a,
                         output logic [7:0] y);

          // sbox implemented as a ROM
          logic [7:0] rvals[0:999];

          initial   $readmemh("rvals.txt", rvals);

          always_ff @(posedge clk)
               y <= rvals[a];

endmodule

module gpixvals(input logic clk,
                         input  logic [9:0] a,
                         output logic [7:0] y);

          // sbox implemented as a ROM
          logic [7:0] gvals[0:999];

          initial   $readmemh("gvals.txt", gvals);
```

```
        always_ff @(posedge clk)
            y <= gvals[a];

endmodule

module bpixvals(input logic clk,
                        input  logic [9:0] a,
                        output logic [7:0] y);

        // sbox implemented as a ROM
        logic [7:0] bvals[0:999];

        initial   $readmemh("bvals.txt", bvals);

        always_ff @(posedge clk)
            y <= bvals[a];

endmodule
```

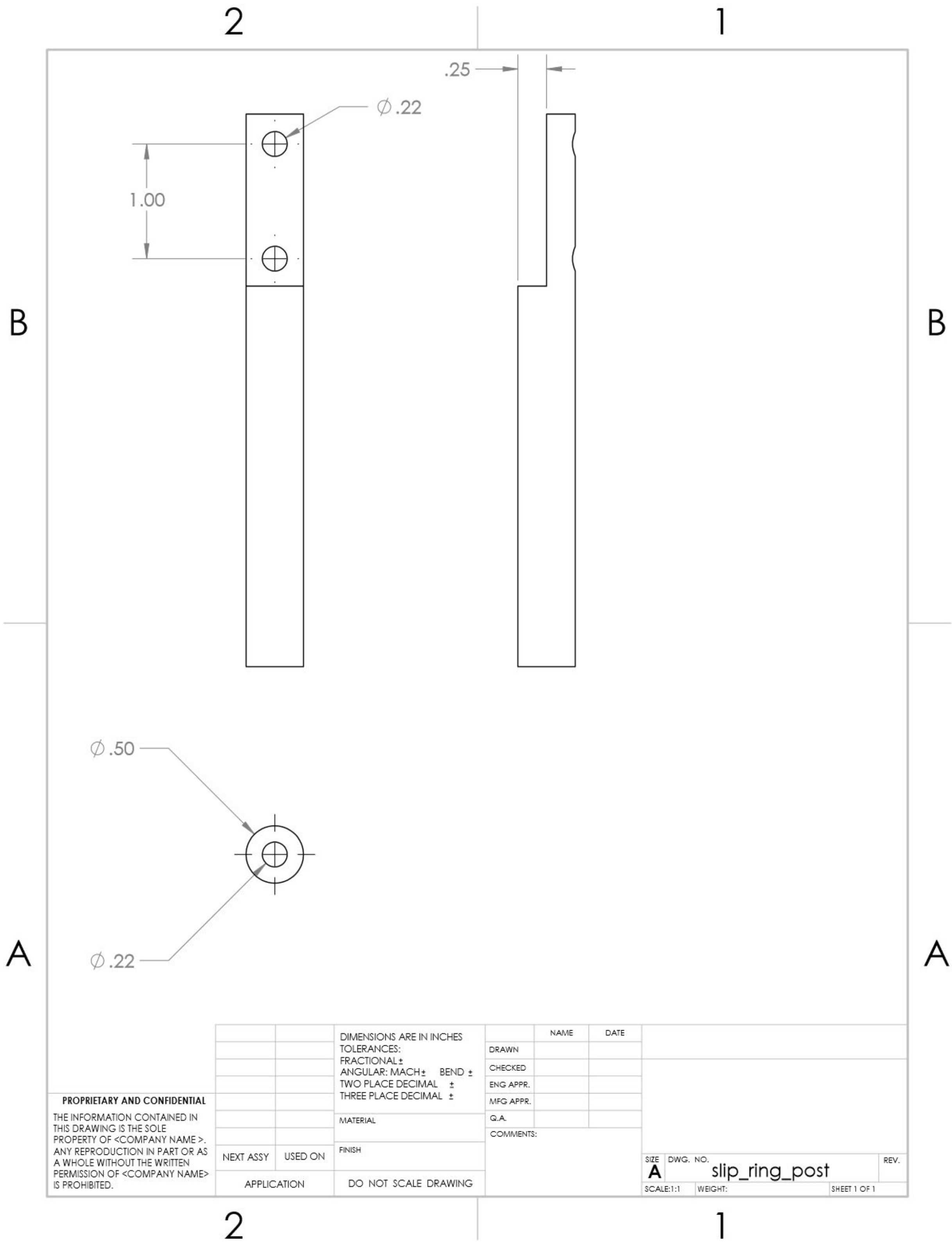
Appendix A


2
1
4.00
2.00
∅.98
∅.20
1.00
6.25
B
B
.78
2.00
A
A
DIMENSIONS ARE IN INCHES
TOLERANCES:
FRACTIONAL±
ANGULAR: MACH± BEND ±
TWO PLACE DECIMAL ±
THREE PLACE DECIMAL ±
NAME
DATE
DRAWN
CHECKED
ENG APPR.
MFG APPR.
Q.A.
COMMENTS:
PROPRIETARY AND CONFIDENTIAL
THE INFORMATION CONTAINED IN THIS DRAWING IS THE SOLE PROPERTY OF <COMPANY NAME>. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF <COMPANY NAME> IS PROHIBITED.
MATERIAL
FINISH
NEXT ASSY
USED ON
APPLICATION
DO NOT SCALE DRAWING
SIZE
A
DWG. NO.
motor_base
REV.
SCALE:1:2
WEIGHT:
SHEET 1 OF 1

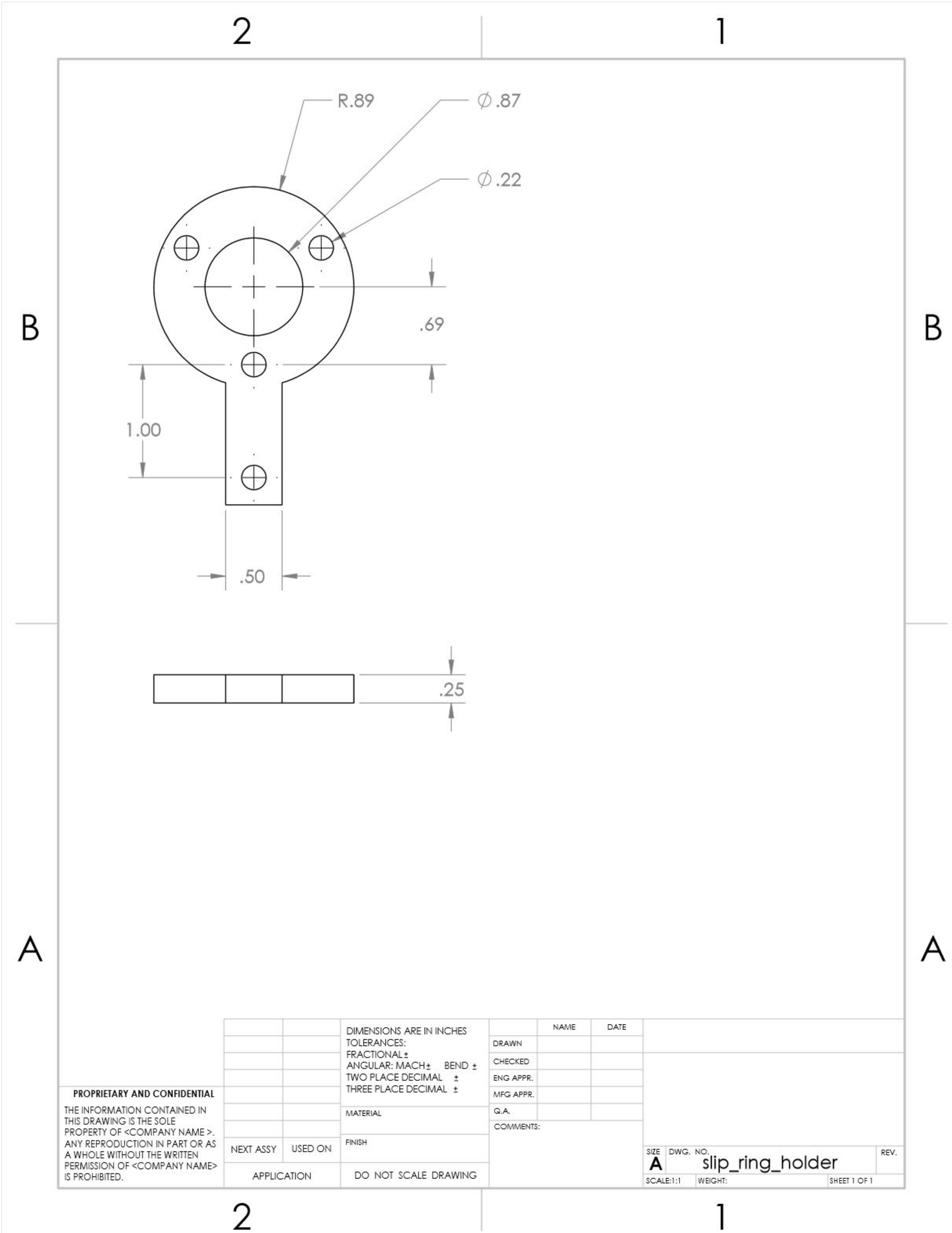
Appendix B


2
1
B
A
∅.28
∅.20
∅.12
1.00
.33
2.30
1.00
4.00
.13
DIMENSIONS ARE IN INCHES
TOLERANCES:
FRACTIONAL ±
ANGULAR: MACH ± BEND ±
TWO PLACE DECIMAL ±
THREE PLACE DECIMAL ±
NAME
DATE
DRAWN
CHECKED
ENG APPR.
MFG APPR.
Q.A.
COMMENTS:
MATERIAL
FINISH
PROPRIETARY AND CONFIDENTIAL
THE INFORMATION CONTAINED IN THIS DRAWING IS THE SOLE PROPERTY OF <COMPANY NAME>. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF <COMPANY NAME> IS PROHIBITED.
NEXT ASSY
USED ON
APPLICATION
DO NOT SCALE DRAWING
SIZE
A
DWG. NO.
motor_plate
REV.
SCALE:1:1
WEIGHT:
SHEET 1 OF 1

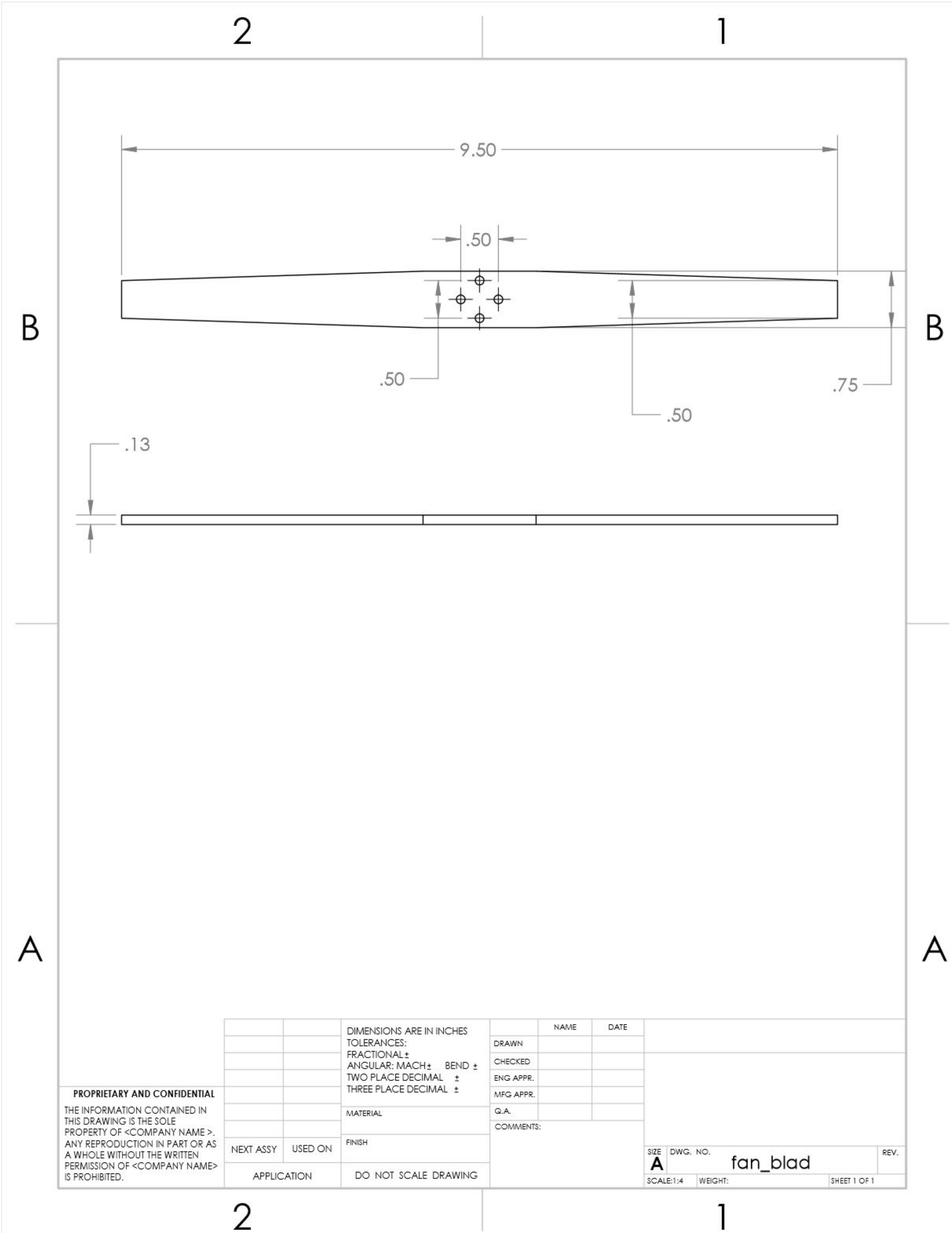
Appendix C


2
1
.25
∅.22
1.00
B
B
∅.50
∅.22
A
A
DIMENSIONS ARE IN INCHES
TOLERANCES:
FRACTIONAL ±
ANGULAR: MACH ± BEND ±
TWO PLACE DECIMAL ±
THREE PLACE DECIMAL ±
NAME
DATE
DRAWN
CHECKED
ENG APPR.
MFG APPR.
Q.A.
COMMENTS:
PROPRIETARY AND CONFIDENTIAL
THE INFORMATION CONTAINED IN THIS DRAWING IS THE SOLE PROPERTY OF <COMPANY NAME>. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF <COMPANY NAME> IS PROHIBITED.
MATERIAL
FINISH
NEXT ASSY
USED ON
APPLICATION
DO NOT SCALE DRAWING
SIZE
A
DWG. NO.
slip_ring_post
REV.
SCALE:1:1
WEIGHT:
SHEET 1 OF 1
2
1

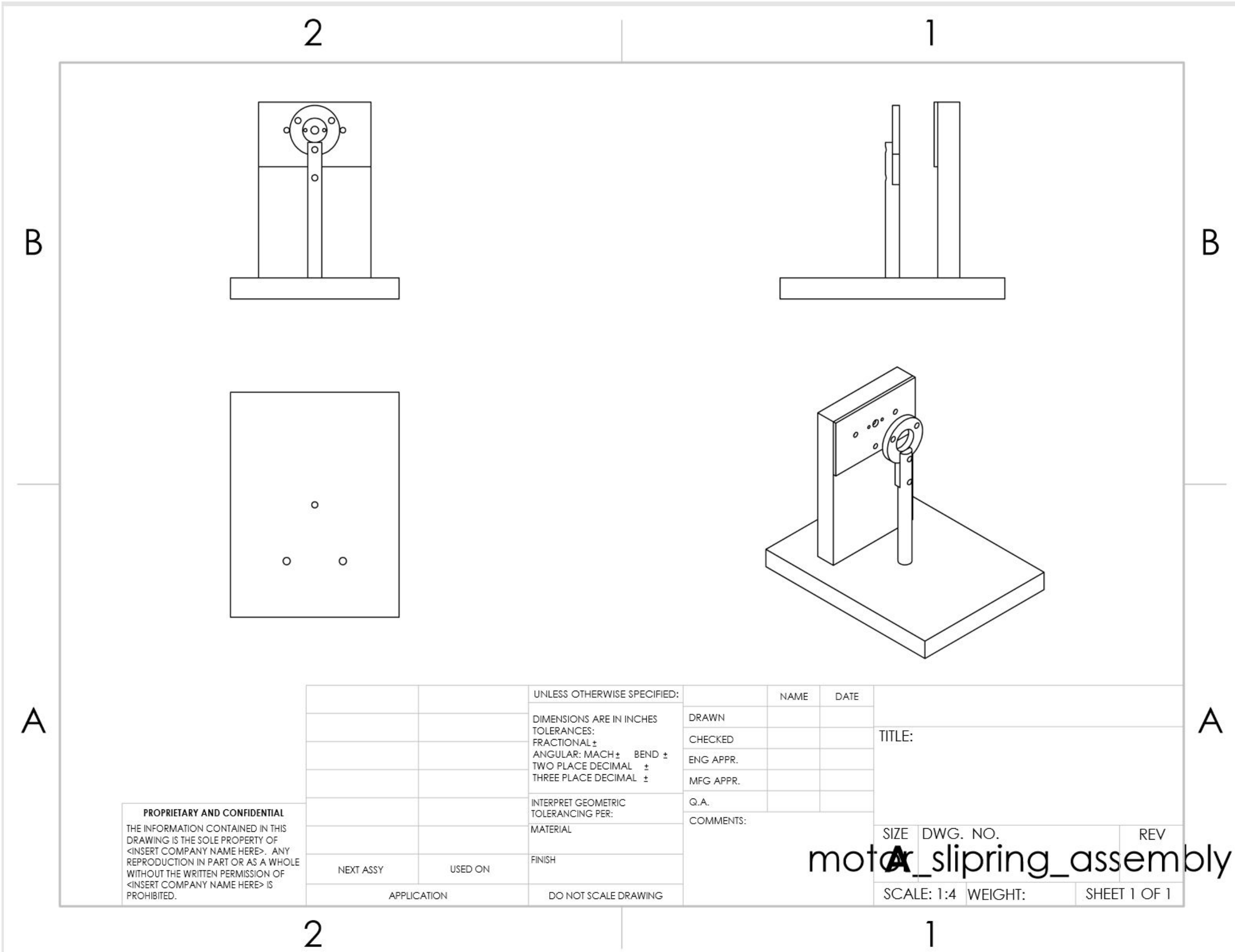
Appendix D


2
1
R.89
∅.87
∅.22
.69
1.00
.50
.25
B
A
PROPRIETARY AND CONFIDENTIAL
THE INFORMATION CONTAINED IN THIS DRAWING IS THE SOLE PROPERTY OF <COMPANY NAME>. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF <COMPANY NAME> IS PROHIBITED.
NEXT ASSY
USED ON
APPLICATION
DIMENSIONS ARE IN INCHES
TOLERANCES:
FRACTIONAL ±
ANGULAR: MACH ± BEND ±
TWO PLACE DECIMAL ±
THREE PLACE DECIMAL ±
MATERIAL
FINISH
DO NOT SCALE DRAWING
NAME
DATE
DRAWN
CHECKED
ENG APPR.
MFG APPR.
Q.A.
COMMENTS:
SIZE
A
DWG. NO.
slip_ring_holder
REV.
SCALE:1:1
WEIGHT:
SHEET 1 OF 1

Appendix E


2
1
B
A
9.50
.50
.50
.75
.50
.13
DIMENSIONS ARE IN INCHES
TOLERANCES:
FRACTIONAL±
ANGULAR: MACH± BEND ±
TWO PLACE DECIMAL ±
THREE PLACE DECIMAL ±
NAME
DATE
DRAWN
CHECKED
ENG APPR.
MFG APPR.
Q.A.
COMMENTS:
MATERIAL
FINISH
NEXT ASSY
USED ON
APPLICATION
DO NOT SCALE DRAWING
PROPRIETARY AND CONFIDENTIAL
THE INFORMATION CONTAINED IN THIS DRAWING IS THE SOLE PROPERTY OF <COMPANY NAME>. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF <COMPANY NAME> IS PROHIBITED.
SIZE
A
DWG. NO.
fan_blad
REV.
SCALE:1:4
WEIGHT:
SHEET 1 OF 1

Appendix F


2
1
B
B
A
A
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN INCHES
TOLERANCES:
FRACTIONAL ±
ANGULAR: MACH ± BEND ±
TWO PLACE DECIMAL ±
THREE PLACE DECIMAL ±
INTERPRET GEOMETRIC TOLERANCING PER:
MATERIAL
FINISH
DO NOT SCALE DRAWING
NEXT ASSY
USED ON
APPLICATION
NAME
DATE
DRAWN
CHECKED
ENG APPR.
MFG APPR.
Q.A.
COMMENTS:
TITLE:
SIZE
DWG. NO.
REV
A
motor_slipring_assembly
SCALE: 1:4
WEIGHT:
SHEET 1 OF 1
PROPRIETARY AND CONFIDENTIAL
THE INFORMATION CONTAINED IN THIS DRAWING IS THE SOLE PROPERTY OF <INSERT COMPANY NAME HERE>. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF <INSERT COMPANY NAME HERE> IS PROHIBITED.
2
1

Appendix G

https://www.pololu.com/file/0J1233/sk6812_datasheet.pdf