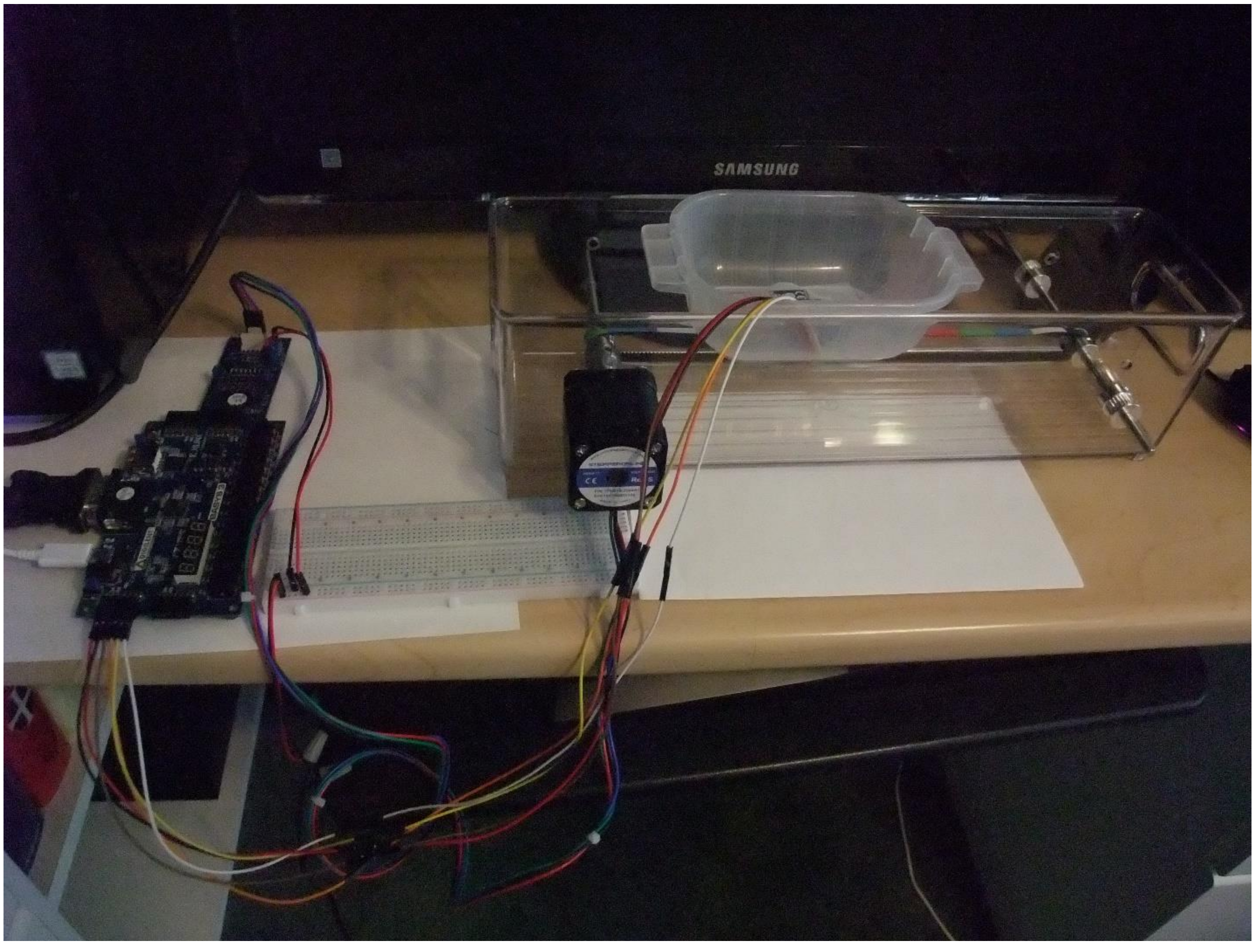
FINAL PROJECT REPORT
COLOR OBJECT COUNTER



University of CINCINNATI

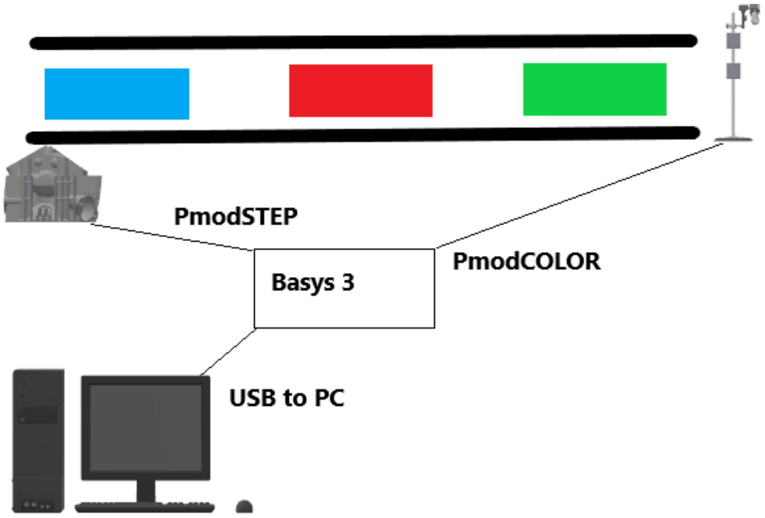
EECE 6014C **Doyle B. Johnson**

Project Title:

Color Object Counter

Project Description:

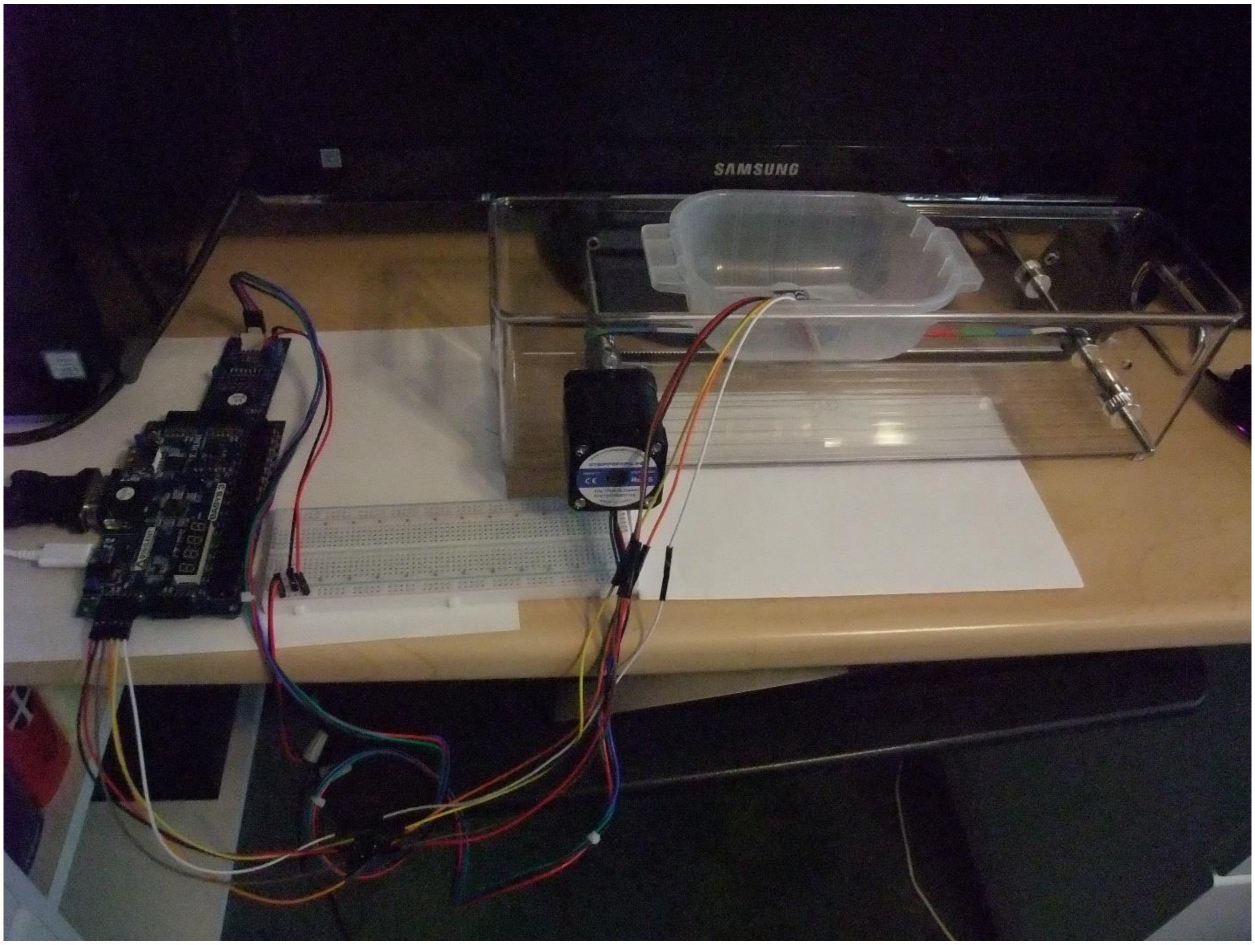
My project simulates a factory environment in which objects of different colors are counted as they pass by a color sensor (Digilent PmodCOLOR add-on) on a conveyor belt. The design includes a motor control to advance the conveyor belt a set distance each time interval to allow an accurate color measurement of each object. The system counts the objects of different colors and displays the respective counts on the terminal screen.



Parts List:

Parts List	Quantity
Digilent Basys 3 board	1
Digilent PmodCOLOR module	1
Digilent PmodSTEP module	1
Stepperonline 17HS19-2004S1 (motor)	1
5v power supply [800ma]	1
Extension pin wires	6
USB cable [type A to micro-B]	1
Plastic bin [10x10x36cm]	1
Plastic bin [4x9x13 cm]	1
Steel rod [5x100 mm]	1
Steel rod [5x200 mm]	1
Shaft coupling [5mm internal dia.]	1
Timing pulleys [20 teeth]	4
Rubber belt [6x560 mm loop]	1
Paper strip w/ color blocks [6x560 mm]	1

Hardware Design:



The design is based on a Basys 3 board, using a Microblaze core on the Xilinx FPGA, along with IP modules for the PmodCOLOR, PmodGPIO, GPIO, Timer, UART, and Interrupt Controller.

The PmodGPIO block is used to connect the PmodSTEP module (which does not have a separate IP module), and the basic GPIO module is used to connect the push-buttons.

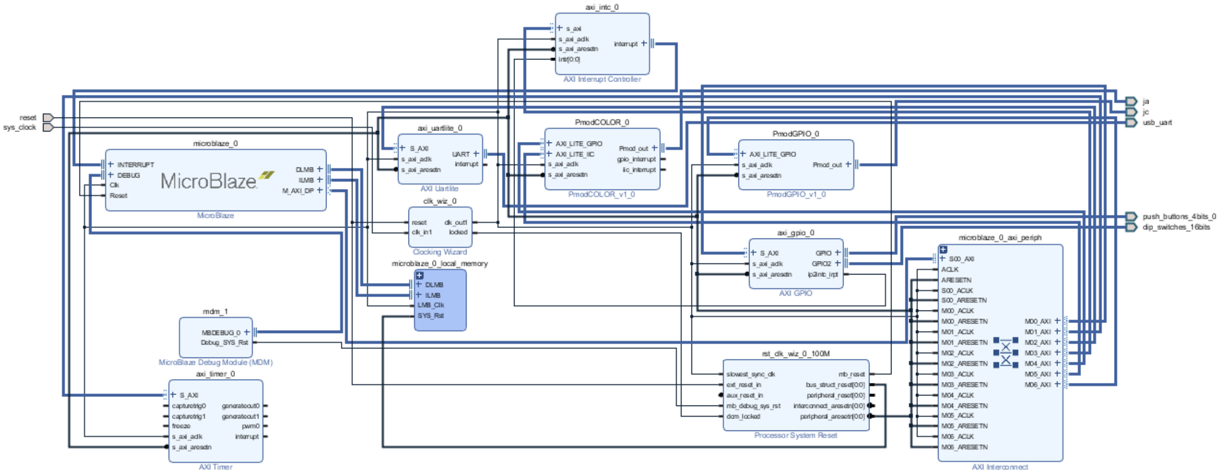
The Interrupt Controller is needed in order to capture push-button I/O to stop the system from running. I would have liked to reset the system, but due to complexities of resetting the Microblaze core, and peripherals, I instead opted to pause the system and allow the user to either resume operation or exit the system.

```
void ResetBtnInterruptHandler(void *CallbackRef)
{
	int answer;
	XGpio *GpioPtr = (XGpio *)CallbackRef;
	xil_printf("System paused. \r\n");
	xil_printf("Press 'r' to resume or 'x' to eXit system. \r\n");
	answer = getchar();
	while ((getchar()) != '\r');
	if (answer == 'r') {
```

```
                xil_printf("Resume system operation.\r\n");

                XGpio_InterruptClear(GpioPtr, GlobalIntrMask);  // clear interrupt
        }
        else {
                xil_printf("Exiting the system.");
                COLOR_SetLED(&pmodCOLOR, 0);
                exit(0);
                }
        IntrFlag = 1;

}
```

The Interrupt Controller block must be manually connected to the axi_gpio module and to the Micoblaze block. Specifically, a manual connection must be made from the ip2intc_inpt pin on the axi_gpio_0 block to the ins[0:0] pin on the axi_intc_0 block. The interrupt output pin on the axi_intc_0 block is then wired to the INTERRUPT pin on the Microblaze block. This provides the interrupt signal from the push-buttons to enable interrupt processing.

The Vivado Block Design is shown below (enlarged version after software code at end of report):

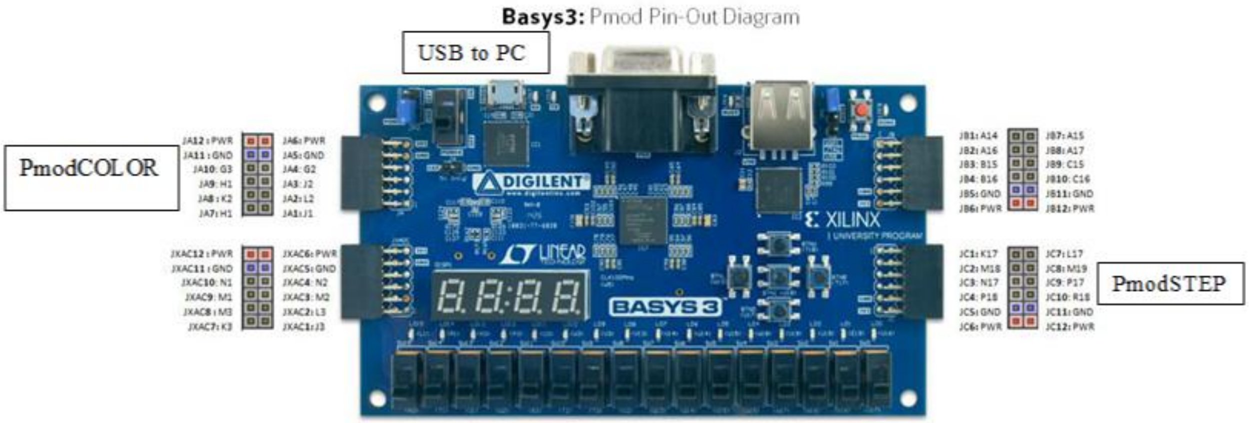


The hardware design includes an HDL wrapper, and is exported for use by the Vitis SDK to program the Xilinx FPGA on the Basys 3 board.

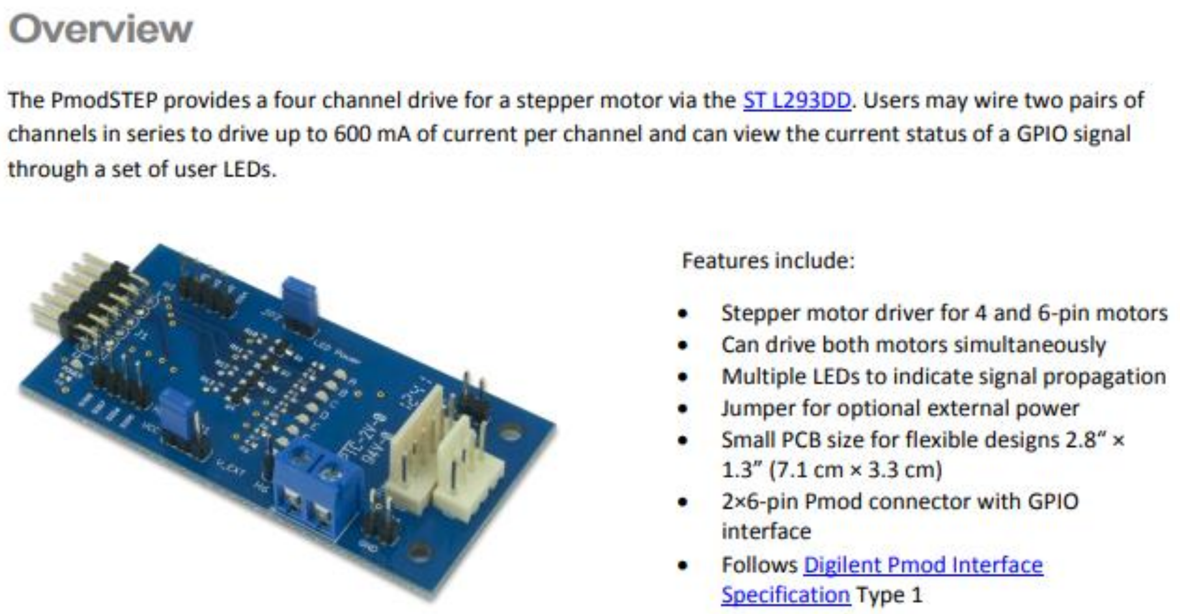
Input/Output:

The design is implemented using the Digilent Basys 3 board. The PmodCOLOR module connects to the JA Pmod header, and the PmodSTEP module (motor control) connects to the JC Pmod header. The PmodCOLOR module has its own dedicated Pmod module in the block design IP. The PmodSTEP module, however, just uses the board’s GPIO interface.

In addition, the system USB cable connects to a PC to provide a serial I/O terminal interface to display the system output (object counts).



PmodSTEP



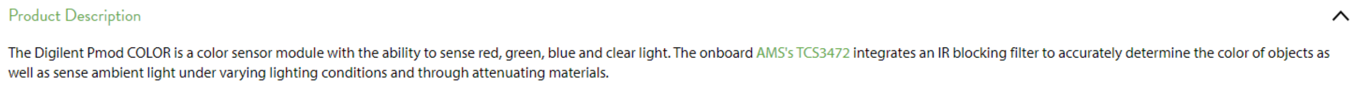
Overview

The PmodSTEP provides a four channel drive for a stepper motor via the ST L293DD. Users may wire two pairs of channels in series to drive up to 600 mA of current per channel and can view the current status of a GPIO signal through a set of user LEDs.

Features include:

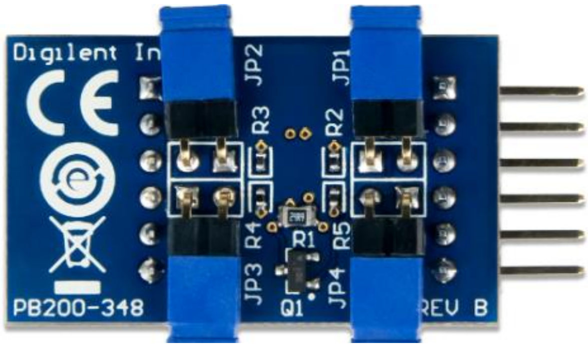
- Stepper motor driver for 4 and 6-pin motors
- Can drive both motors simultaneously
- Multiple LEDs to indicate signal propagation
- Jumper for optional external power
- Small PCB size for flexible designs 2.8" × 1.3" (7.1 cm × 3.3 cm)
- 2×6-pin Pmod connector with GPIO interface
- Follows Digilent Pmod Interface Specification Type 1

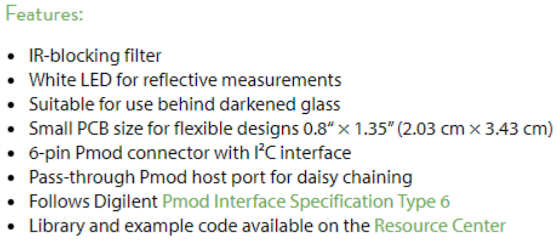
PmodCOLOR



Product Description

The Digilent Pmod COLOR is a color sensor module with the ability to sense red, green, blue and clear light. The onboard AMS's TCS3472 integrates an IR blocking filter to accurately determine the color of objects as well as sense ambient light under varying lighting conditions and through attenuating materials.




Features:

- IR-blocking filter
- White LED for reflective measurements
- Suitable for use behind darkened glass
- Small PCB size for flexible designs 0.8" × 1.35" (2.03 cm × 3.43 cm)
- 6-pin Pmod connector with I^2C interface
- Pass-through Pmod host port for daisy chaining
- Follows Digilent Pmod Interface Specification Type 6
- Library and example code available on the Resource Center

Wiring Diagram of Project:

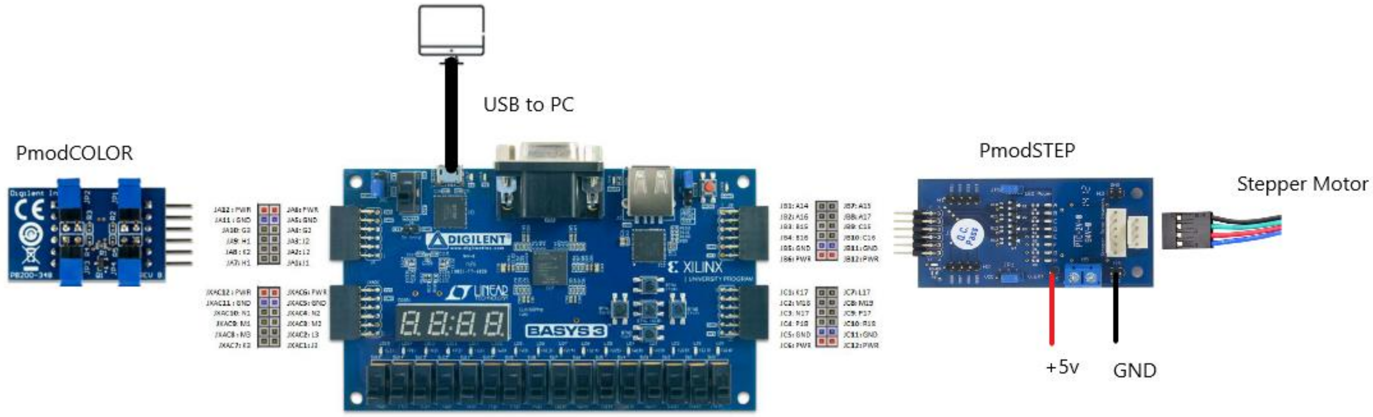


Photo of Basys 3 board, with PmodSTEP, 5V power supply to the PmodSTEP module, and header pins to connected from JA header to PmodCOLOR module:

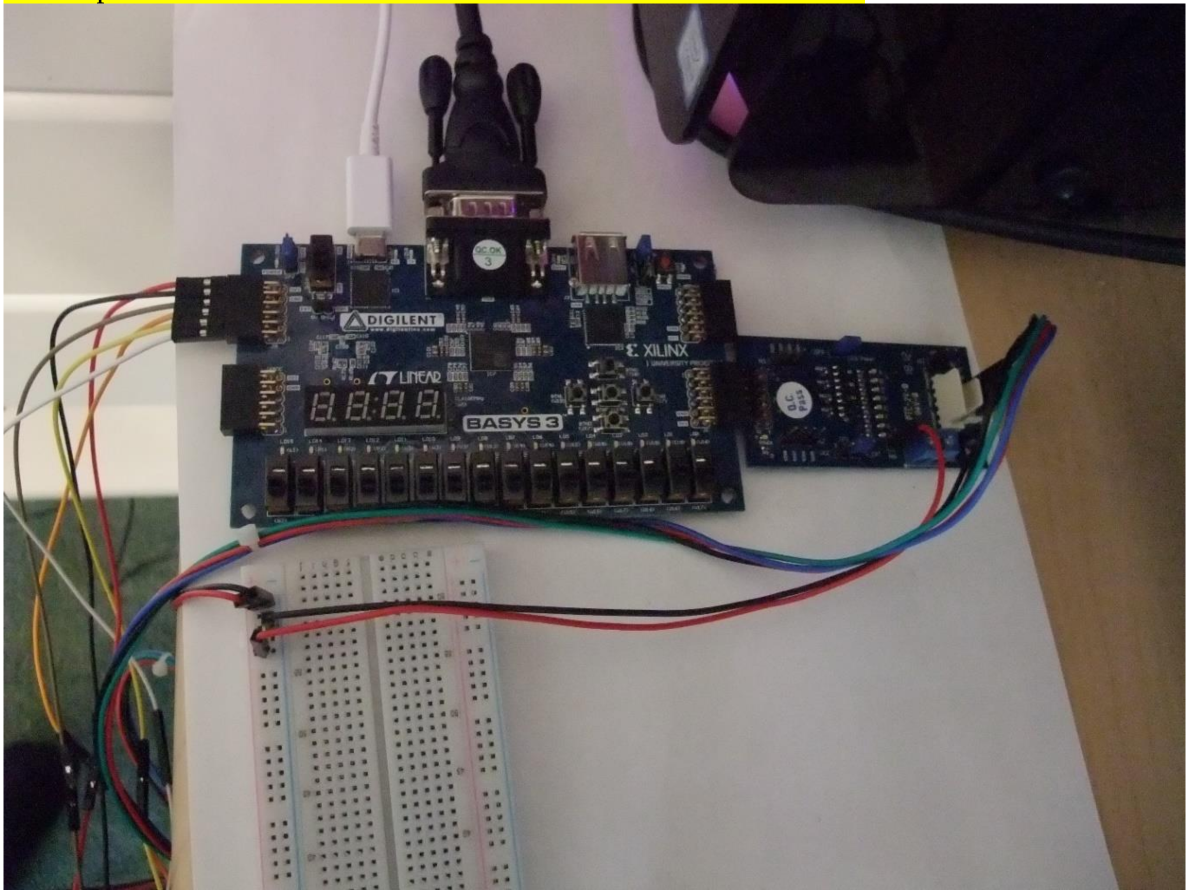
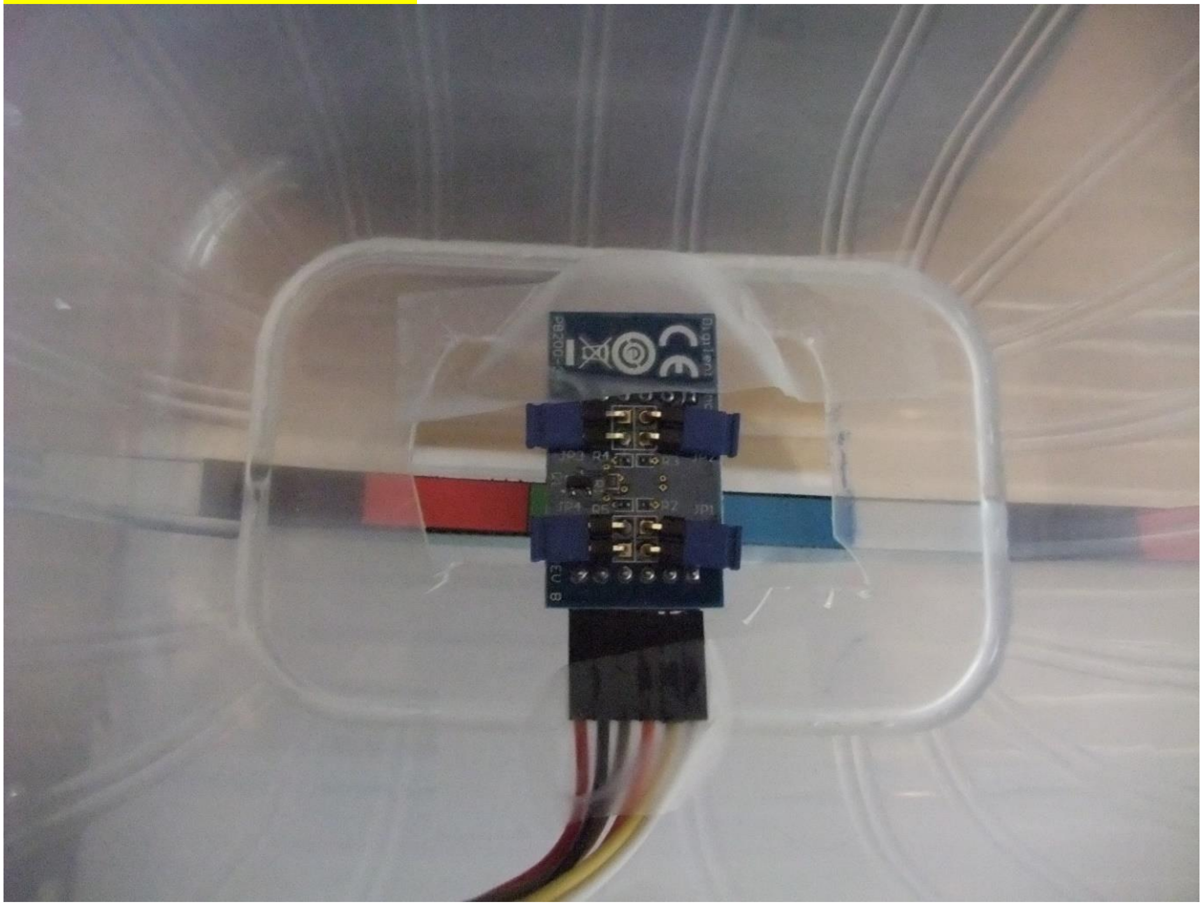


Photo of PmodCOLOR module:



Stepper Motor:

I initially tried to use the Digilent stepper motor available from the Digilent website:

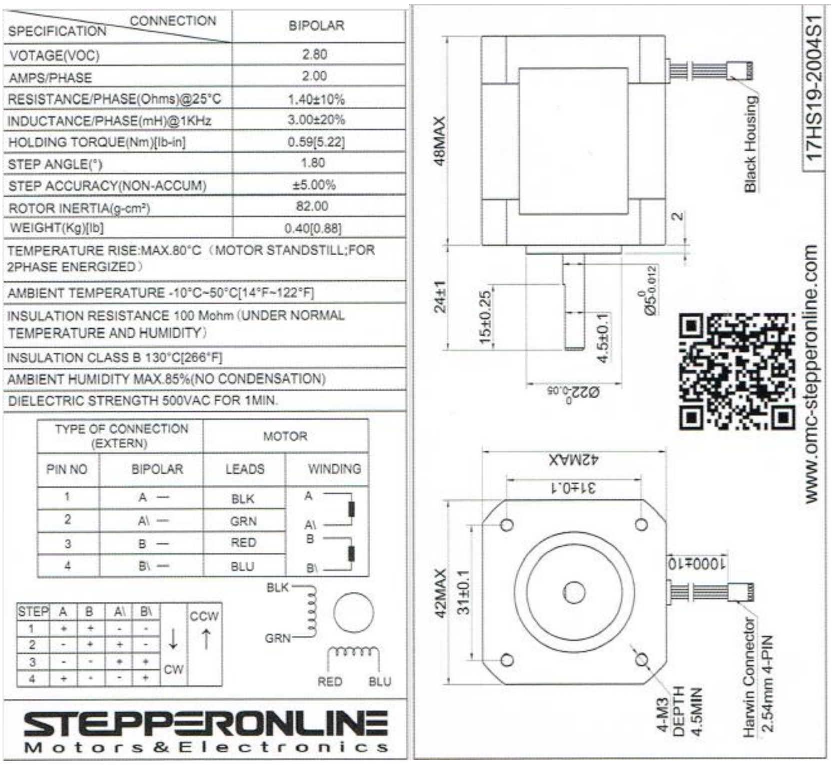
https://store.digilentinc.com/stepper-motor/

However, this stepper is not very precise. After testing, it was not possible to accurately rotate the motor 360°. Specifically, setting the motor to "203" steps (one full rotation) caused the motor to lose about 90° after about 100 rotations. Similarly, "204" steps (one full rotation) resulted in a 90° gain after 100 rotations.

As a result, I bought a larger stepper motor. The Stepperonline model 17HS19-2004S1 is much more accurate. After extensive testing, it had repeatable position control. For purposes of this project I stepped it 25 steps each interval, which translates to 2cm of linear movement of the belt.

https://www.amazon.com/STEPPERONLINE-Stepper-Bipolar-Connector-compatible/dp/B00PNEQKC0

Photo of stepper motor:



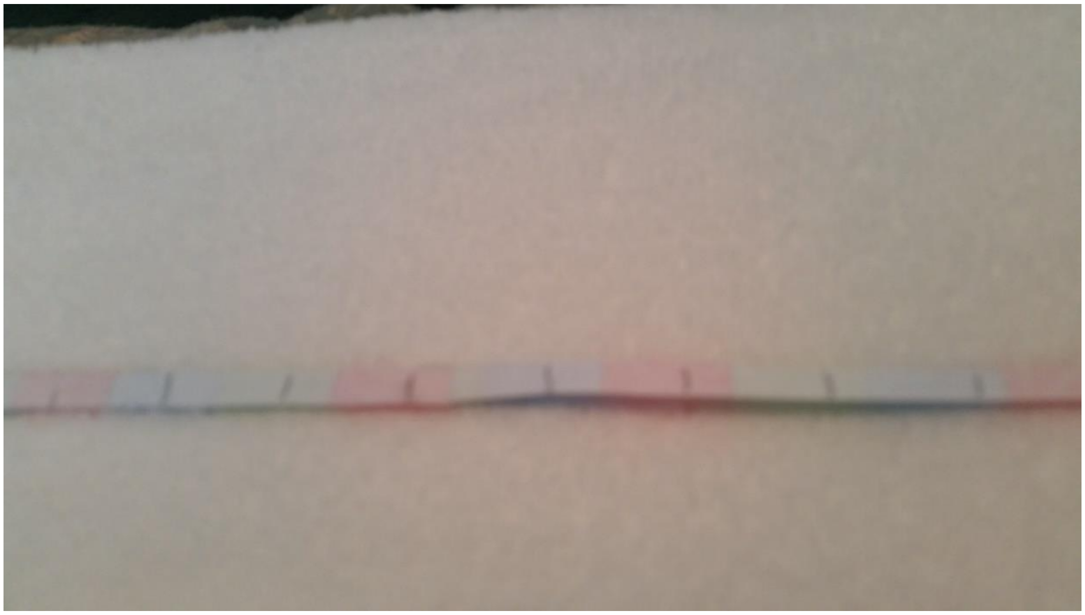
SPECIFICATION \ CONNECTION	BIPOLAR
VOTAGE(VOC)	2.80
AMPS/PHASE	2.00
RESISTANCE/PHASE(Ohms)@25°C	1.40±10%
INDUCTANCE/PHASE(mH)@1KHz	3.00±20%
HOLDING TORQUE(Nm)[lb-in]	0.59[5.22]
STEP ANGLE(°)	1.80
STEP ACCURACY(NON-ACCUM)	±5.00%
ROTOR INERTIA(g-cm²)	82.00
WEIGHT(Kg)[lb]	0.40[0.88]
TEMPERATURE RISE:MAX.80°C（MOTOR STANDSTILL;FOR 2PHASE ENERGIZED）	
AMBIENT TEMPERATURE -10°C~50°C[14°F~122°F]	
INSULATION RESISTANCE 100 Mohm（UNDER NORMAL TEMPERATURE AND HUMIDITY）	
INSULATION CLASS B 130°C[266°F]	
AMBIENT HUMIDITY MAX.85%(NO CONDENSATION)	
DIELECTRIC STRENGTH 500VAC FOR 1MIN.	

TYPE OF CONNECTION (EXTERN)		MOTOR	
PIN NO	BIPOLAR	LEADS	WINDING
1	A —	BLK	A
2	A\ —	GRN	A\
3	B —	RED	B
4	B\ —	BLU	B\

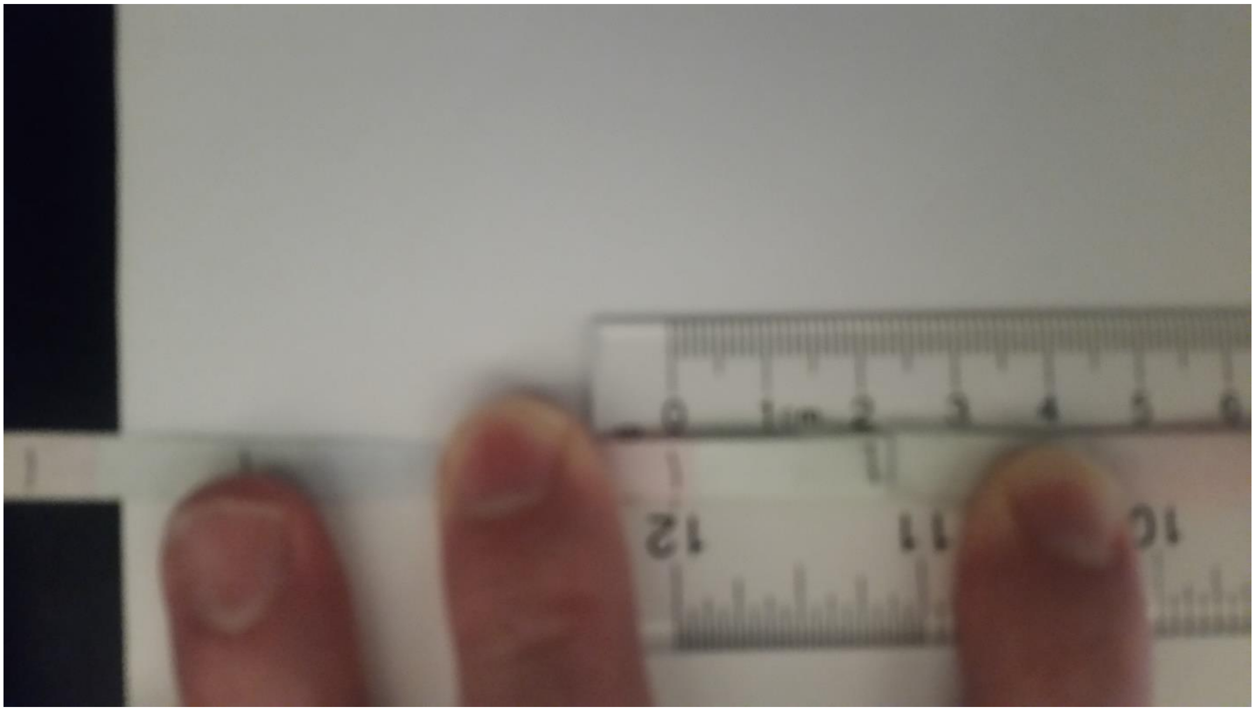
STEP	A	B	A\	B\	CW	CCW
1	+	+	-	-	↓	↑
2	-	+	+	-		
3	-	-	+	+		
4	+	-	-	+		

The stepper motor is stepped by activating two wires at the same time. This is controlled in the software, as described below.

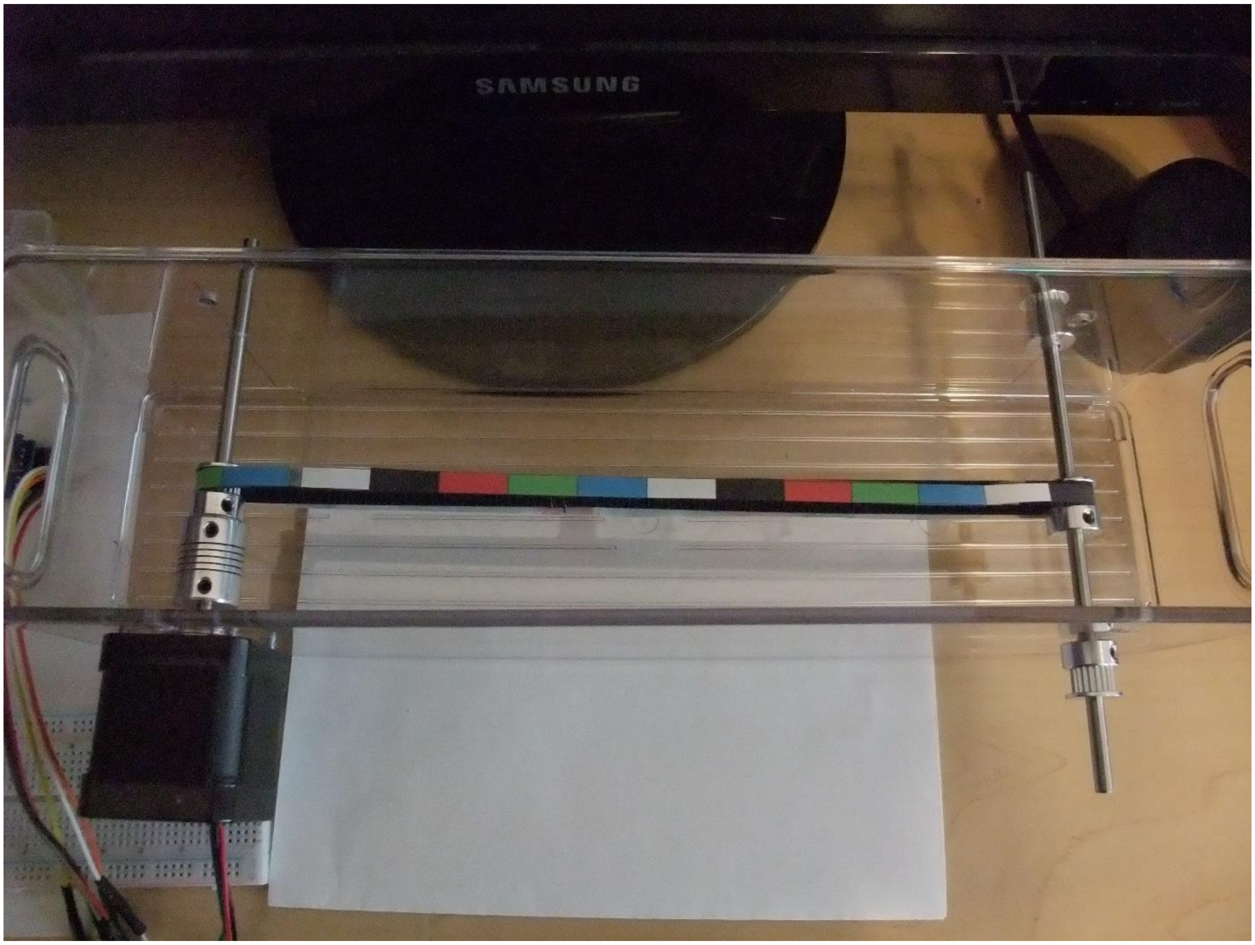
The stepper is very accurate. As shown in the video demonstration, I marked a white loop of paper (reverse side of color loop) on the belt, and measured the distance between the marks:



The marks measure consistently 2cm apart around the loop.


0
1cm
2
3
4
5
6
12
11
10

Conveyor Belt



The conveyor belt is constructed from standard notched printer belt material. This is commonly used for 3D printers. The stepper motor is connected to a shaft with a notched pully. The belt loops around a free-turning pully on the opposite end. A clear plastic box is used to mount the hardware in place.

A paper strip of color blocks is taped onto the belt. Each color block is 2cm long, such that the center-to-center distance is 2cm. The belt is 56 cm, so there are 28 color blocks. The stepper motor advances the belt 2cm each interval so that the color sensor is positioned in the middle of each color block.

The color strip has the following colors:

Red:	6
Green:	6
Blue:	6
Black:	5
White:	5

Color Sensor and Color Science:

The PmodCOLOR module sensor outputs 16-bit RGB colors. The RGB color space is normally defined as 0-255, where (0,0,0) is black and (255,255,255) is white.

The sensor is a continuous reading sensor and the data can be read using the library function:

```
color_data = COLOR_GetData(&pmodCOLOR);
```

where pmodCOLOR is the instantiation of the module.

For purposes of this project, I read each color reading 100 times and averaged the result in order to average out the low-bit fluctuations.

```
         redTotal = redTotal + color_data.r;
        greenTotal = greenTotal + color_data.g;
        blueTotal = blueTotal + color_data.b;

         // average color data over SAMPLE-SIZE readings
         redAverage = redTotal / SAMPLE_SIZE;
         greenAverage = greenTotal / SAMPLE_SIZE;
         blueAverage = blueTotal / SAMPLE_SIZE;
```

However, I had two major problems with the color sensor. First, the sensor is not very accurate and despite using the calibration libraries available from Digilent, the color sensor would not produce consistent readings even from the same color sample.

Second, the sensor is an RGB sensor, but the only print output I can produce is CMYK. Using a CMYK inkjet printer it is simply not possible to print, for example, true RGB red (255,0,0), green (0,255,0) or blue (0,0,255). The printed red color appears more orange, but the sensor can still detect it as more red. Blue and green are more problematic, however. Without a source of being able to print true RGB colors, detecting intermediate colors, like yellow (255,255,0) is even more difficult.

Extensive online research failed to produce any easy solutions. Color matching between RGB and CMYK is a known problem, and commercial software programs and commercial printers can be used to provide a better color match. However, there is really no way to correct the colors using a consumer inkjet printer with standard software.

As a result, I limited the color blocks to red, green, blue, white and black. However, even these limited colors were challenging to detect accurately.

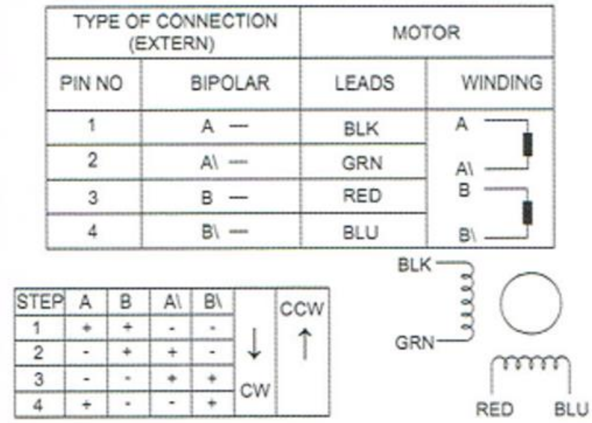
In conclusion, the Digilent PmodCOLOR sensor seems designed more for experimentation and toys, than industrial or scientific use. This makes sense given its relative low cost. The inability to print true RGB primary colors further exacerbates the problems. As a result, I was not able to produce consistent color counts, as shown below:

Software:

Program code written in C executes on the Microblaze core to control the system operation.

Digilent provides libraries for the PmodCOLOR module, and Xilinx provides libraries for the Microblaze Interrupt Controller. However, Digilent does not provide any library modules, or examples, for using the PmodSTEP module to control a stepper motor.

As a result, I had to research how to drive the PmodSTEP module, and how to control a stepper motor. The Stepperonline stepper motor provides the following instructions for controlling the motor:



TYPE OF CONNECTION (EXTERN)		MOTOR	
PIN NO	BIPOLAR	LEADS	WINDING
1	A —	BLK	A
2	A\ —	GRN	A\
3	B —	RED	B
4	B\ —	BLU	B\

STEP	A	B	A\	B\		CCW
1	+	+	-	-	↓ CW	↑
2	-	+	+	-		
3	-	-	+	+		
4	+	-	-	+		

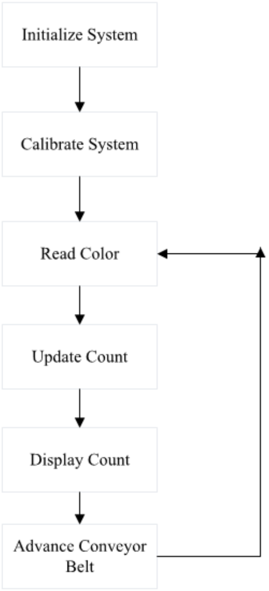
I implemented this control flow using the “GPIO_setPin” library call:

```
GPIO_setPin(&pmodSTEP, 5, 1);
 GPIO_setPin(&pmodSTEP, 7, 1);
 GPIO_setPin(&pmodSTEP, 6, 0);
 GPIO_setPin(&pmodSTEP, 8, 0);
 DelayMSec(5);
 GPIO_setPin(&pmodSTEP, 5, 0);
 GPIO_setPin(&pmodSTEP, 7, 1);
 GPIO_setPin(&pmodSTEP, 6, 1);
 GPIO_setPin(&pmodSTEP, 8, 0);
 DelayMSec(5);
 GPIO_setPin(&pmodSTEP, 5, 0);
 GPIO_setPin(&pmodSTEP, 7, 0);
 GPIO_setPin(&pmodSTEP, 6, 1);
 GPIO_setPin(&pmodSTEP, 8, 1);
 DelayMSec(5);
 GPIO_setPin(&pmodSTEP, 5, 1);
 GPIO_setPin(&pmodSTEP, 7, 0);
 GPIO_setPin(&pmodSTEP, 6, 0);
 GPIO_setPin(&pmodSTEP, 8, 1);
 DelayMSec(5);
```

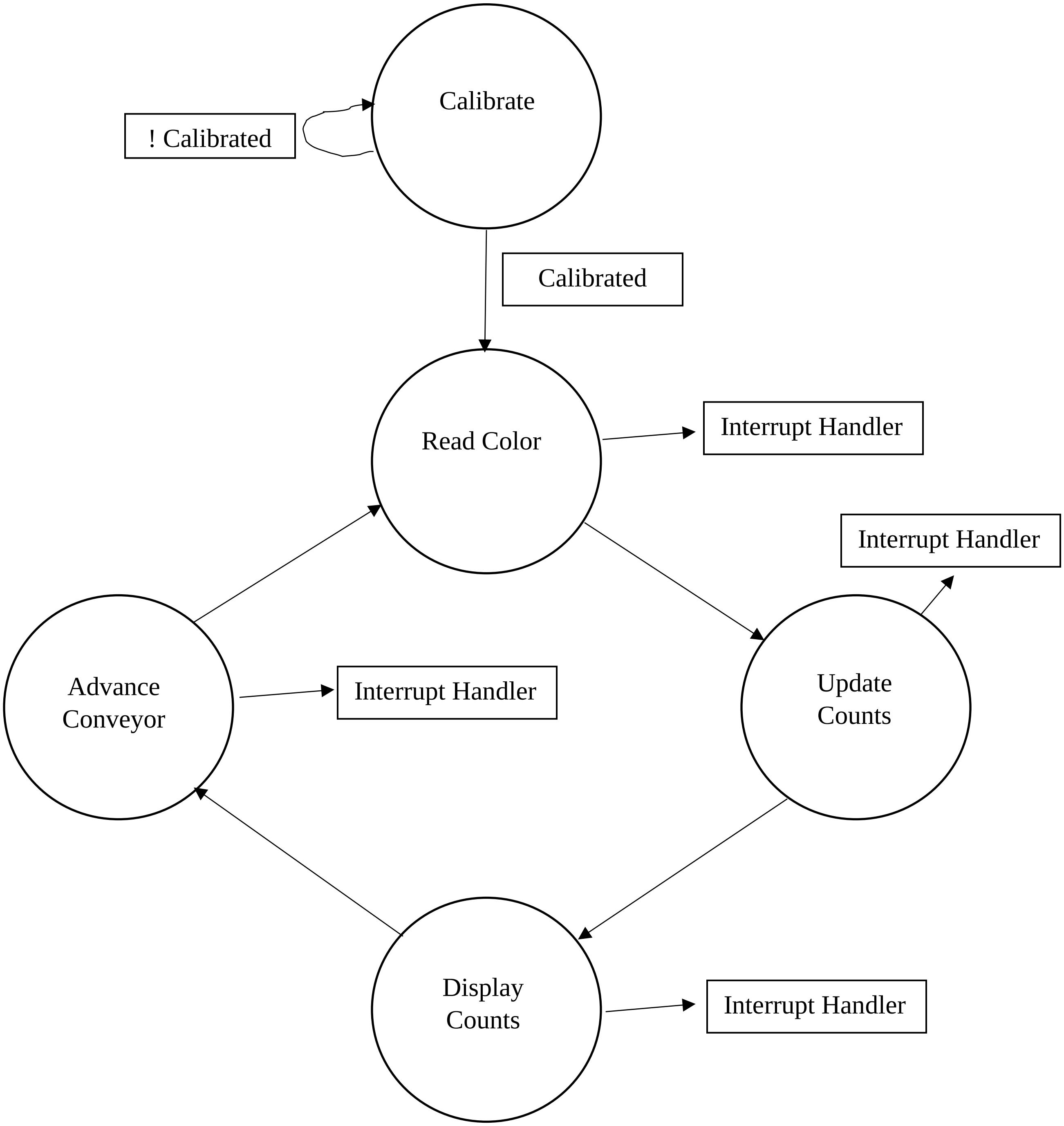
I used the PmodCOLOR example library files to perform the color calibration. In addition, I used the Xilinx interrupt library files to generate and process an interrupt. An interrupt is generated to pause the system when one of the push-buttons is pushed, and then the user can select to resume normal operation or exit the system.

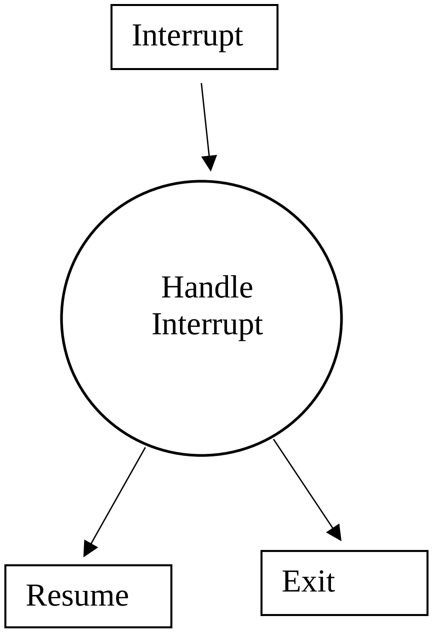
Specifically, the system code flowchart is:

Flowchart:



State Machine:




Interrupt
Handle Interrupt
Resume
Exit

Test Cases:

Acceptance tests for components and use cases:

Motor test:

PmodSTEP advances motor a fixed amount at set intervals

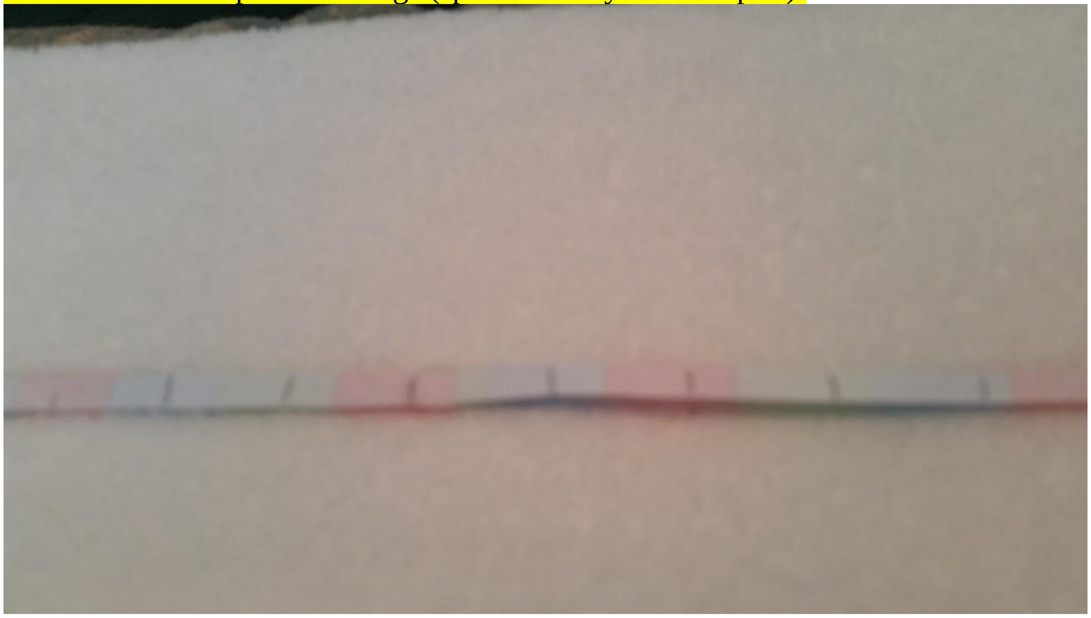
Test motor control and measure time and distance of conveyor movement

Test result:

I can control the stepper motor with excellent precision. Initially, I tested the Digilent stepper motor. However, it lacked positional accuracy, even after just a few revoltions.

After switching from the Digilent stepper motor to the Stepperonline motor, I can accurately control the motor. As shown in the test video (and the marked paper strip), the stepper motor can advance the belt exactly 2cm each interval and maintain this level of precision indefinitely.

Photo of manual space markings (spaced evenly at 2cm apart):



Calibration test:

After color calibration, system should accurately identify different colors:

Test different colors and verify correct identification

Test result:

I am using the color calibration functions from the PmodCOLOR Vivado library files. However, the color sensor is not particularly accurate. The PmodCOLOR sensor seems designed more for experimentation and toys, and not industrial or scientific use. This makes sense given its relative low cost. Consistent color matching is a challenge with this sensor. As a result, I limited the system operation to red, green, blue, white and black.

For the tests, I ran the system for approximately 2 minutes (3.5 loops of the color strip). The expected numbers vary based on exactly when the system was paused, and what the starting position of the loop was.

Summary of test results below (photos below):

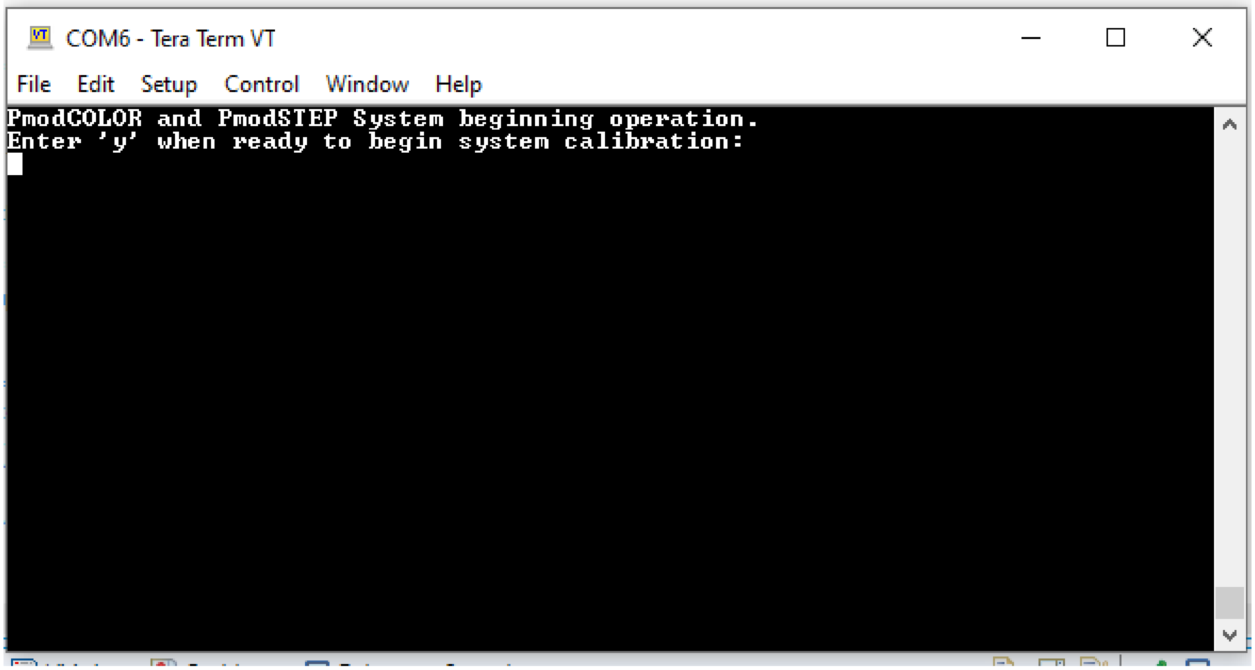
Color	Test #1		Test #2		Test #3		Test #4		Test #5	
	Exp.	Cnt.	Exp.	Cnt.	Exp.	Cnt.	Exp.	Cnt.	Exp.	Cnt.
Red	22	20	21	20	21	16	22	17	22	19
Green	22	16	22	19	22	19	22	20	21	16
Blue	22	25	21	24	22	30	22	29	21	27
White	17	14	17	18	18	12	19	18	17	20
Black	17	25	17	17	17	23	18	19	18	17
Total:	100		98		100		103		99	

“Exp.” = expected count based on starting and ending position of the loop

“Cnt.” = output on display on system

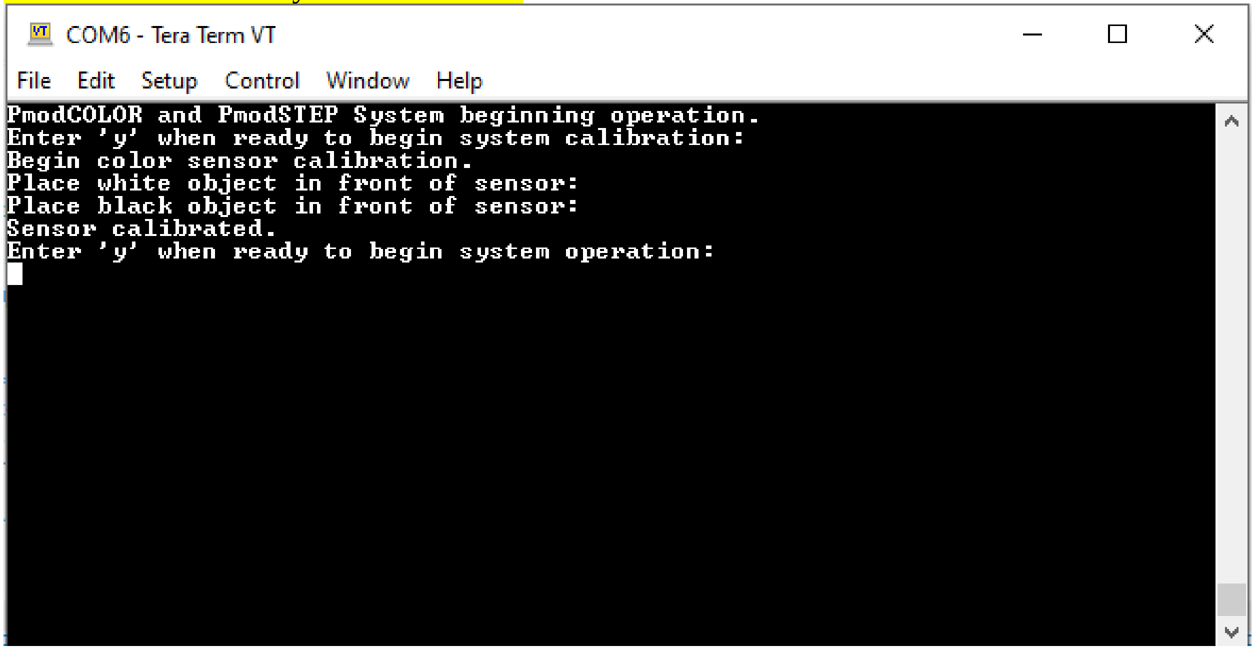
Color	Total Expected	Total Count	Error
Red	108	92	(16%)
Green	109	90	(18%)
Blue	108	135	+25%
White	88	82	(7%)
Black	87	101	+16%
Total:	500	500	

Screen-shot of initial system calibration:



```
PmodCOLOR and PmodSTEP System beginning operation.
Enter 'y' when ready to begin system calibration:
```

Screen-shot of initial system calibration:



```
PmodCOLOR and PmodSTEP System beginning operation.
Enter 'y' when ready to begin system calibration:
Begin color sensor calibration.
Place white object in front of sensor:
Place black object in front of sensor:
Sensor calibrated.
Enter 'y' when ready to begin system operation:
```

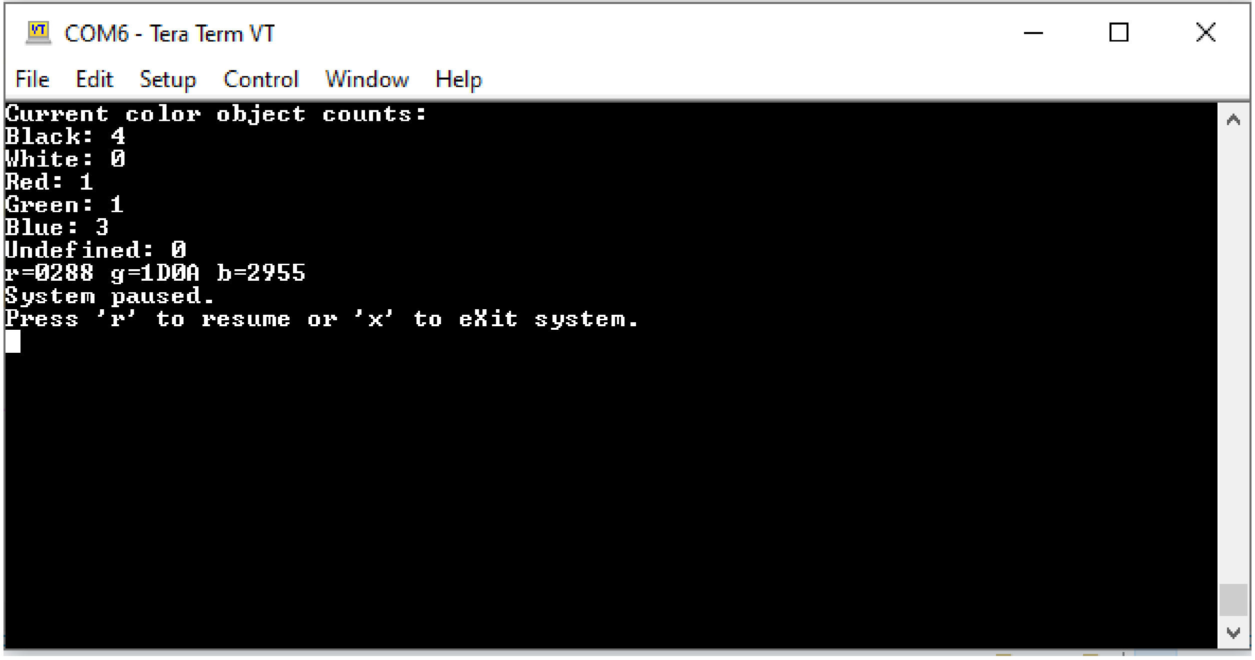
Interrupt test:

Activate push-button
System should immediately stop conveyor, count values, and display
User can then resume or exit the system

Test result:

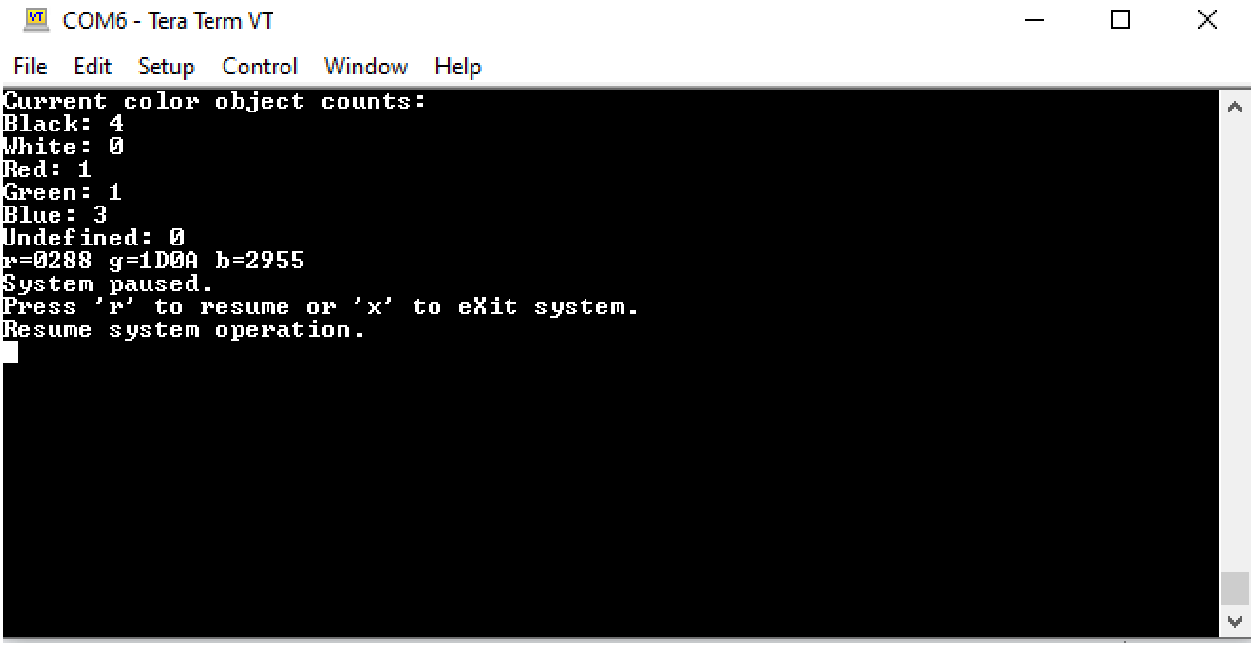
The push-buttons have been connected to the interrupt controller and a push-button event interrupts the processing regardless of operation state. Confirmed in video demonstration.

Screen-shot of system interrupt:



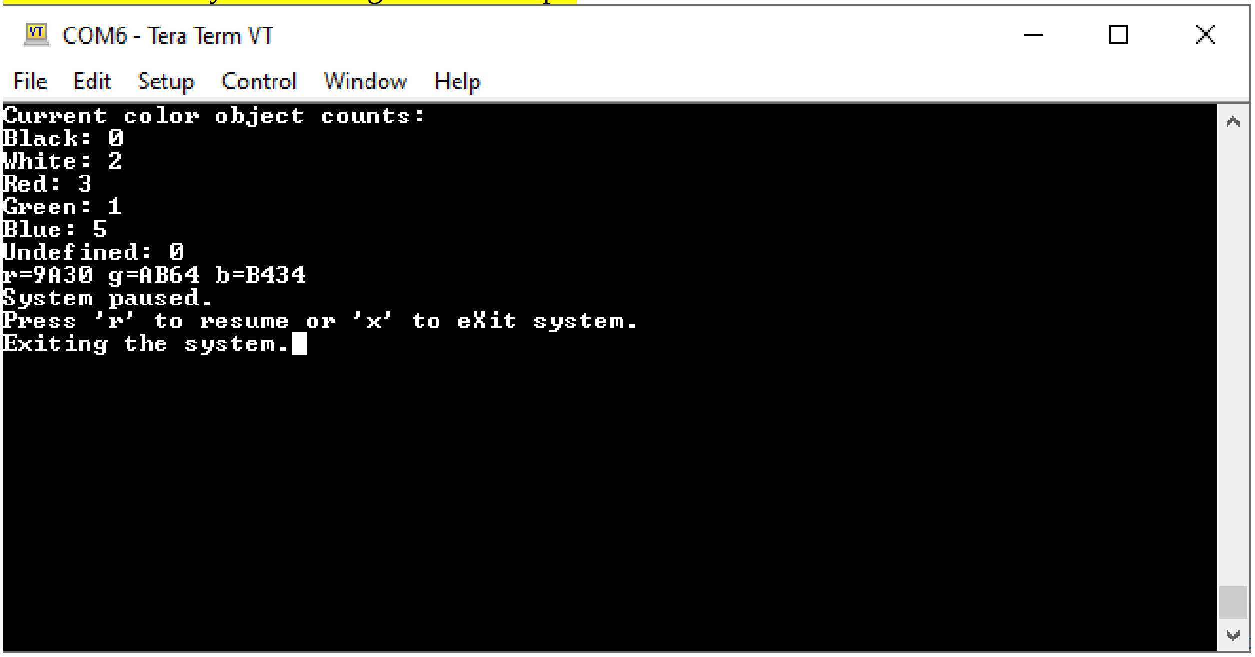
```
Current color object counts:
Black: 4
White: 0
Red: 1
Green: 1
Blue: 3
Undefined: 0
r=0288 g=1D0A b=2955
System paused.
Press 'r' to resume or 'x' to eXit system.
```

Screen-shot of system resuming system operation:



```
Current color object counts:
Black: 4
White: 0
Red: 1
Green: 1
Blue: 3
Undefined: 0
r=0288 g=1D0A b=2955
System paused.
Press 'r' to resume or 'x' to eXit system.
Resume system operation.
```

Screen-shot of system exiting after interrupt:



```
Current color object counts:
Black: 0
White: 2
Red: 3
Green: 1
Blue: 5
Undefined: 0
r=9A30 g=AB64 b=B434
System paused.
Press 'r' to resume or 'x' to eXit system.
Exiting the system.
```

Operation test:

In normal operation, system should advance conveyor and update color (object) counts on screen

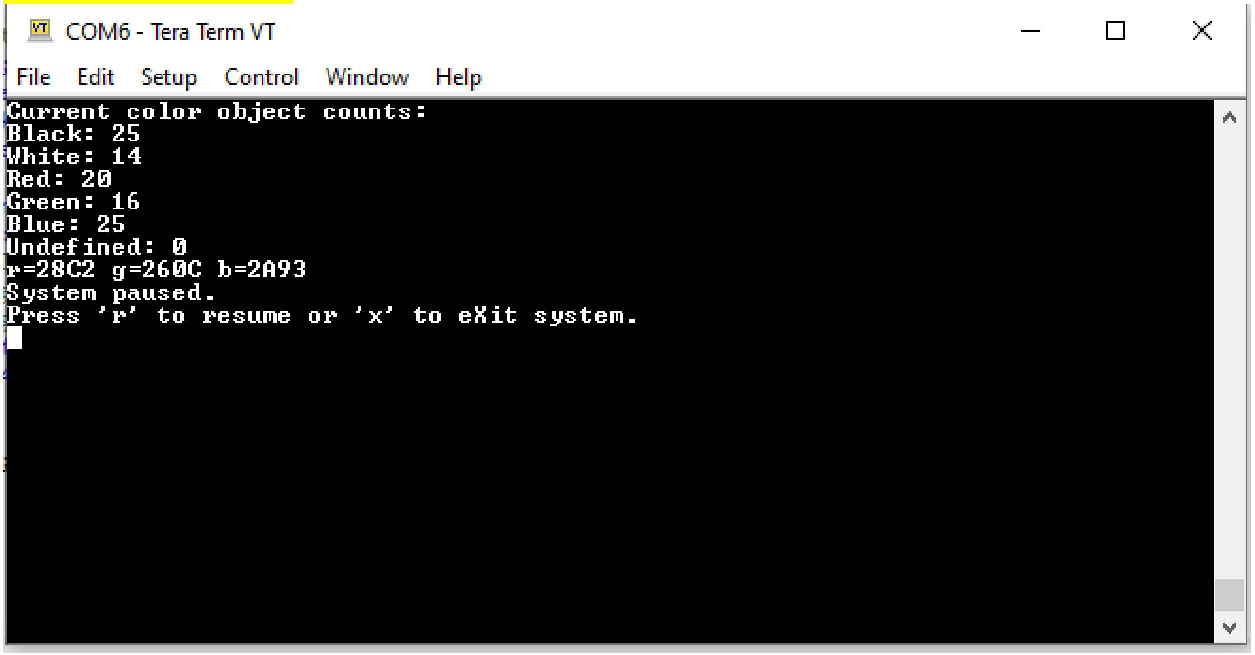
Test with different color combinations and compare output count with known inputs

Confirm conveyor advances at correct time and moves correct distance

Test result:

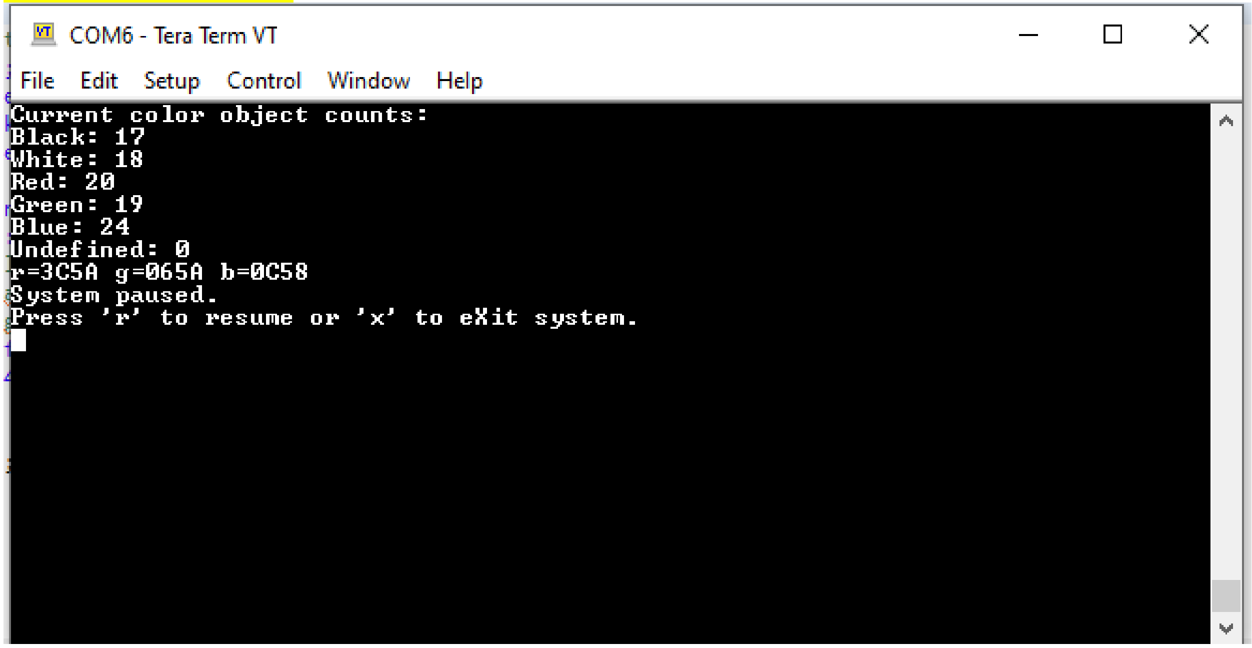
The count and display functions are working correctly. The stepper motor very accurately advances the belt. The push-button interrupts are captured every time and properly handled. Repeated interrupts are handled. Unfortunately, the color capture is not accurate as noted above.

Screen-shot of Test#1:



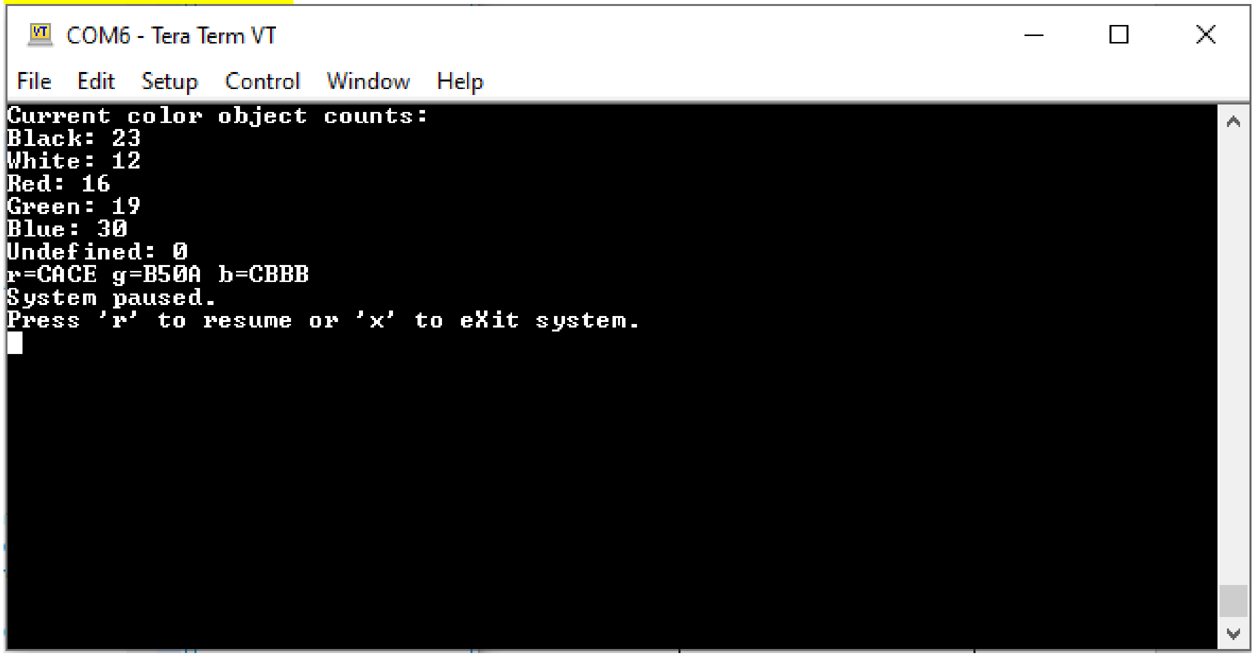
```
Current color object counts:
Black: 25
White: 14
Red: 20
Green: 16
Blue: 25
Undefined: 0
r=28C2 g=260C b=2A93
System paused.
Press 'r' to resume or 'x' to eXit system.
```

Screen-shot of Test#2:



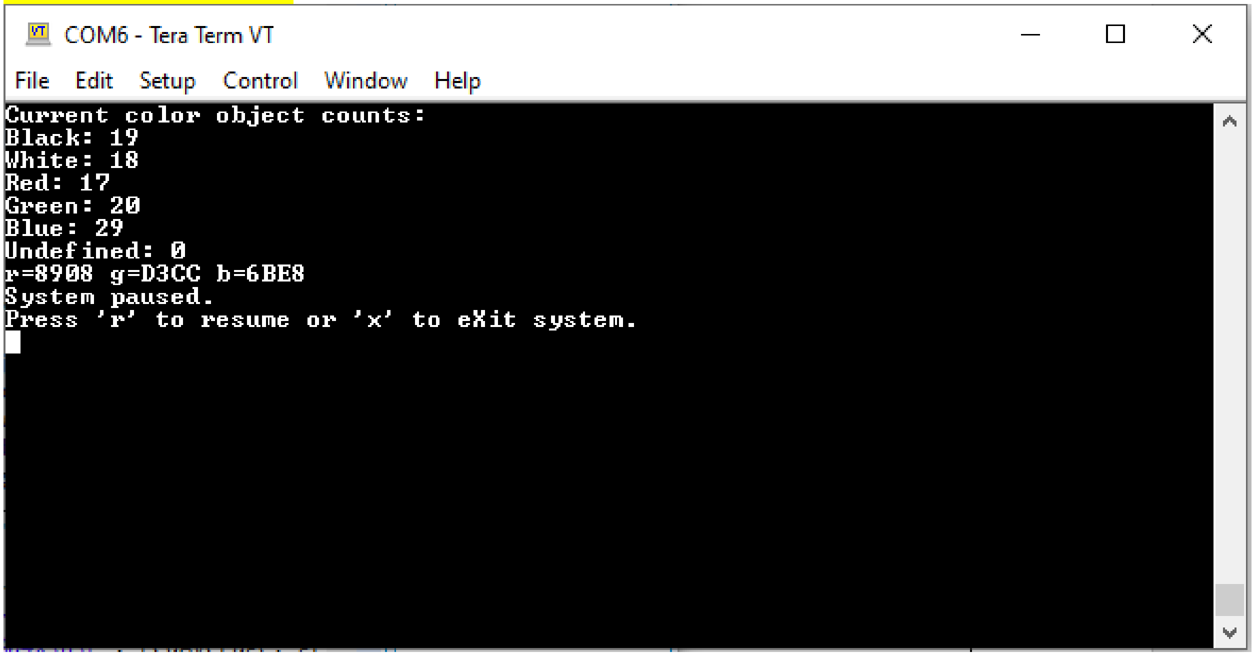
```
Current color object counts:
Black: 17
White: 18
Red: 20
Green: 19
Blue: 24
Undefined: 0
r=3C5A g=065A b=0C58
System paused.
Press 'r' to resume or 'x' to eXit system.
```

Screen-shot of Test#3:



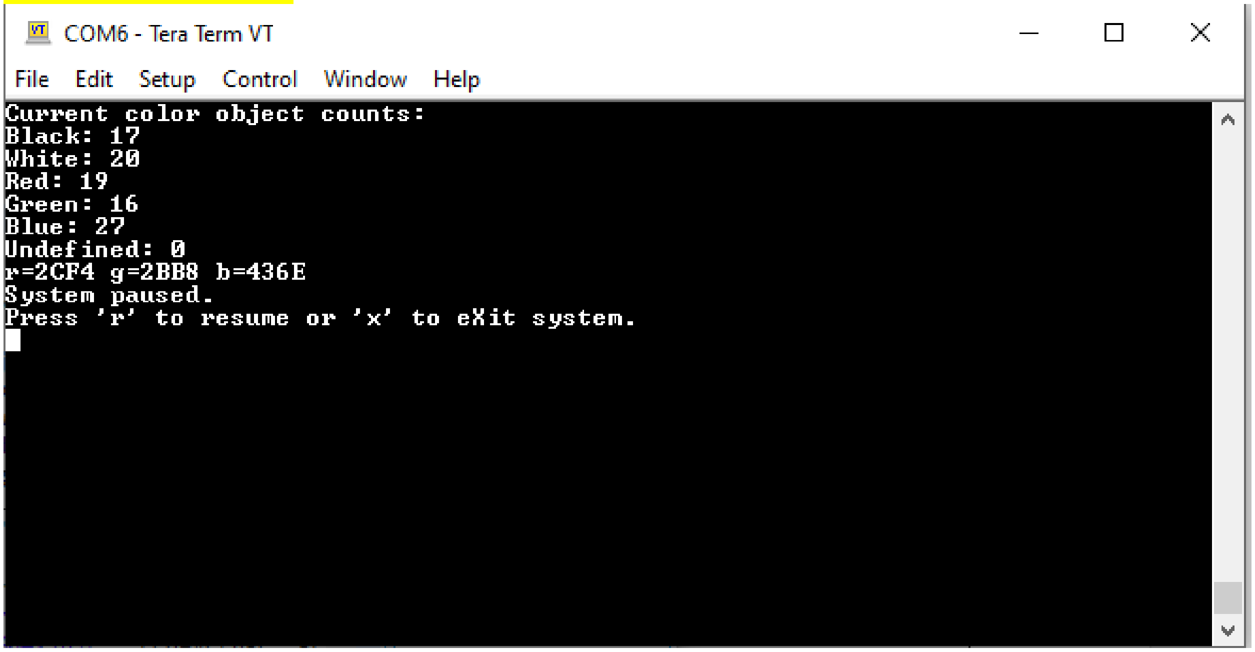
```
Current color object counts:
Black: 23
White: 12
Red: 16
Green: 19
Blue: 30
Undefined: 0
r=CACE g=B50A b=CBBB
System paused.
Press 'r' to resume or 'x' to eXit system.
```

Screen-shot of Test#4:



```
Current color object counts:
Black: 19
White: 18
Red: 17
Green: 20
Blue: 29
Undefined: 0
r=8908 g=D3CC b=6BE8
System paused.
Press 'r' to resume or 'x' to eXit system.
```

Screen-shot of Test#5:



```
Current color object counts:
Black: 17
White: 20
Red: 19
Green: 16
Blue: 27
Undefined: 0
r=2CF4 g=2BB8 b=436E
System paused.
Press 'r' to resume or 'x' to eXit system.
```

Conclusion/Lessons Learned:

This project resulted in a working color object counter. A precisely controlled stepper motor advanced a conveyor belt having a paper color block strip. A color sensor reads the color and updated a color count. A push-button event interrupts normal processing and pauses the system. The user has the option to resume operation or exit the system. The color counts are displayed on the serial term screen.

However, the color sensor does not produce consistent color readings. In addition, it is difficult to provide true RGB input colors. As a result, the color counts are not accurate.

I learned several important technical lessons from this project:

1. Interrupt handling for the Microblaze core using the Interrupt Controller IP module
2. Controlling I/O to a Pmod module using standard GPIO (i.e. manually setting values to individual pins)
3. Stepper motor control by activating select input coils on the stepper motor
4. Color science RGB vs. CMYK

Project source code:

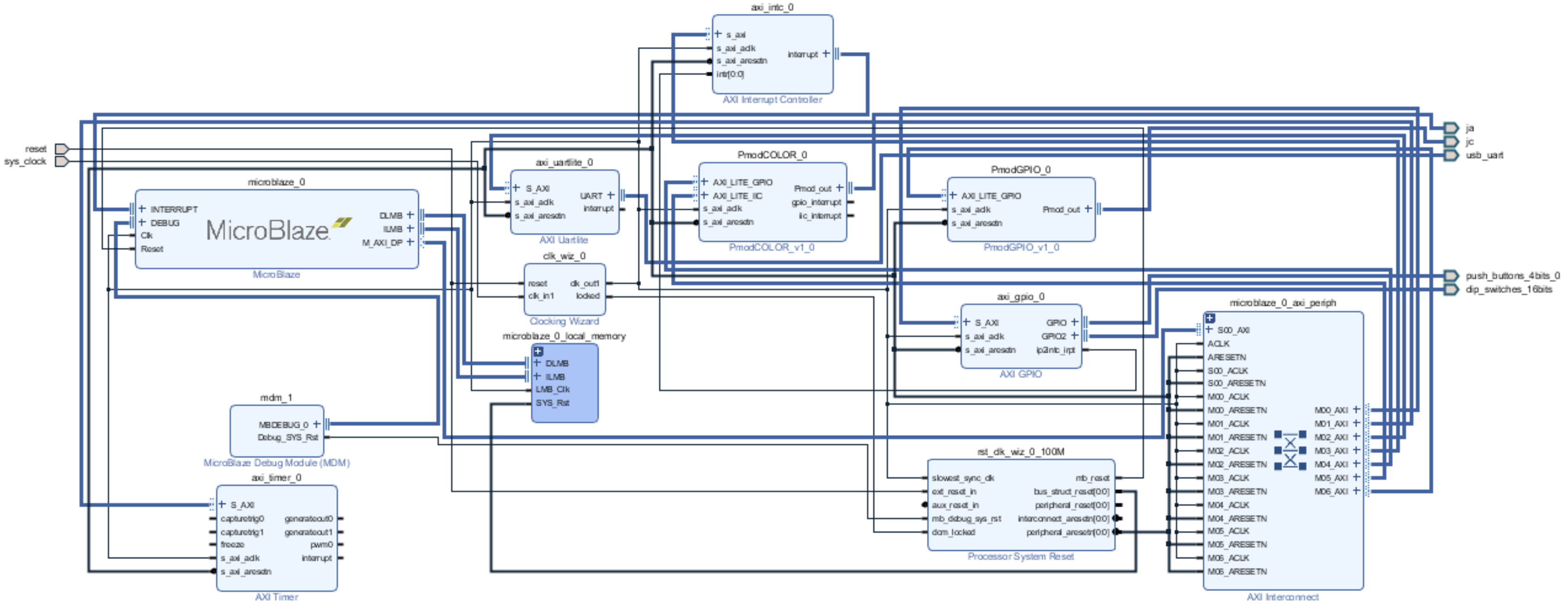
```
/*
 *
 * EECE 6014C Final Project
 * Doyle B. Johnson
 *
 * Color Object Counter using PmodCOLOR and PmodSTEP modules
 *
 * Digilent PmodCOLOR and Xilinx XGpio interrupt library routines incorporated herein
(as noted below).
 * Digilent does not provide any library support for the PmodSTEP module.
 *
 */

#include "PmodGPIO.h"
#include "PmodCOLOR.h"
#include "xil_cache.h"
#include "xil_exception.h"
#include "xparameters.h"
#include "xtmrctr.h"
#include "xgpio.h"
#include "xintc.h"
#include <stdio.h>
#include <stdlib.h>

// the following definitions are used to set GPIO (push-button) interrupt to stop
system
#define GPIO_DEVICE_ID 0
#define INTC_GPIO_INTERRUPT_ID   XPAR_INTC_0_GPIO_0_VEC_ID
#define GPIO_CHANNEL1 1

#define INTR_DELAY  0x00FFFFFF

#define INTC_DEVICE_ID    XPAR_INTC_0_DEVICE_ID
#define INTC        XIntc
#define INTC_HANDLER      XIntc_InterruptHandler

// sample size used to average several color readings
#define SAMPLE_SIZE 100


typedef struct {
   COLOR_Data min, max;
} CalibrationData;

COLOR_Data color_data;
CalibrationData calib_data;

// create instances IP modules
PmodGPIO pmodSTEP;
PmodCOLOR pmodCOLOR;
XTmrCtr TimerCounter;
XGpio Gpio;
INTC Intc;
```

```
int i;
static u16 GlobalIntrMask;
static volatile u32 IntrFlag;

// function prototypes
void SystemInit();
void SystemCalibration();
void SystemOperation();
CalibrationData SystemInitCalibrationData(COLOR_Data firstSample);
void SystemCalibrate(COLOR_Data newSample, CalibrationData *calib);
COLOR_Data SystemNormalizeToCalibration(COLOR_Data sample, CalibrationData calib);
void ActivateSTEP();
void DelayMSec();
void ResetBtnInterruptHandler(void *CallBackRef);
int XGpioInterruptSetup(INTC *IntcInstancePtr, XGpio *InstancePtr,
                    u16 DeviceId, u16 IntrId, u16 IntrMask);
void GpioDisableIntr(INTC *IntcInstancePtr, XGpio *InstancePtr,
                    u16 IntrId, u16 IntrMask);


// millisecond timer for delays
void DelayMSec(long int t)
{
t=t * 100000;
long int count_time;

        XTmrCtr_Start(&TimerCounter, 0);

        do // delay
        {
        count_time = XTmrCtr_GetValue(&TimerCounter, 0);
        }
        while (count_time< t );

      XTmrCtr_Reset(&TimerCounter, 0);

} // DelayMSec


// activate stepper motor to advance conveyor
void ActivateSTEP(int t)
{
      int i;

    // need to fire the coils in the proper order to control stepper motor
      for(i=0; i < t; i++)
      {
                    GPIO_setPin(&pmodSTEP, 5, 1);
                    GPIO_setPin(&pmodSTEP, 7, 1);
                    GPIO_setPin(&pmodSTEP, 6, 0);
                    GPIO_setPin(&pmodSTEP, 8, 0);
                    DelayMSec(5);
                    GPIO_setPin(&pmodSTEP, 5, 0);
                    GPIO_setPin(&pmodSTEP, 7, 1);
                    GPIO_setPin(&pmodSTEP, 6, 1);
                    GPIO_setPin(&pmodSTEP, 8, 0);
                    DelayMSec(5);
                    GPIO_setPin(&pmodSTEP, 5, 0);
                    GPIO_setPin(&pmodSTEP, 7, 0);
                    GPIO_setPin(&pmodSTEP, 6, 1);
                    GPIO_setPin(&pmodSTEP, 8, 1);
                    DelayMSec(5);
                    GPIO_setPin(&pmodSTEP, 5, 1);
                    GPIO_setPin(&pmodSTEP, 7, 0);
                    GPIO_setPin(&pmodSTEP, 6, 0);
                    GPIO_setPin(&pmodSTEP, 8, 1);
                    DelayMSec(5);
       }

}


// function to initialize timer, GPIO(stepper), XGpio(push-button interrupt) and
COLOR modules
void SystemInit() {

   XTmrCtr_Initialize(&TimerCounter, 0);
   XTmrCtr_SetOptions(&TimerCounter, 0, 0x0);

   XGpio_Initialize(&Gpio, GPIO_DEVICE_ID);
   XGpioInterruptSetup(&Intc, &Gpio, GPIO_DEVICE_ID, INTC_GPIO_INTERRUPT_ID,
GPIO_CHANNEL1);

   GPIO_begin(&pmodSTEP, XPAR_PMODGPIO_0_AXI_LITE_GPIO_BASEADDR, 0x00);

   COLOR_Begin(&pmodCOLOR, XPAR_PMODCOLOR_0_AXI_LITE_IIC_BASEADDR,
         XPAR_PMODCOLOR_0_AXI_LITE_GPIO_BASEADDR, 0x29);

   COLOR_SetENABLE(&pmodCOLOR, COLOR_REG_ENABLE_PON_MASK);
   DelayMSec(5);
   COLOR_SetENABLE(&pmodCOLOR,
         COLOR_REG_ENABLE_PON_MASK | COLOR_REG_ENABLE_RGBC_INIT_MASK);
   DelayMSec(5);

}


// calibrate color sensor
void SystemCalibration() {

   xil_printf("\e[1;1H\e[2J");
   xil_printf("PmodCOLOR and PmodSTEP System beginning operation.\r\n");
   xil_printf("Enter 'y' when ready to begin system calibration:\r\n");
      while (getchar() != 'y') {
   xil_printf("Enter 'y' when ready to begin system calibration:\r\n");
      }
      while ((getchar()) != '\r');

   COLOR_SetLED(&pmodCOLOR, 1);  // turn on PmodCOLOR bright LED
   xil_printf("Begin color sensor calibration.\n\r");
   xil_printf("Place white object in front of sensor:\n\r");
   DelayMSec(5000);
   for(i=0; i < SAMPLE_SIZE; i++)
   {
   color_data = COLOR_GetData(&pmodCOLOR);
   calib_data = SystemInitCalibrationData(color_data);
   }
   xil_printf("Place black object in front of sensor:\n\r");
   DelayMSec(5000);
   for(i=0; i < SAMPLE_SIZE; i++)
   {
   color_data = COLOR_GetData(&pmodCOLOR);
   calib_data = SystemInitCalibrationData(color_data);
   }
   xil_printf("Sensor calibrated.\n\r");
   xil_printf("Enter 'y' when ready to begin system operation:\r\n");
   while (getchar() != 'y') {
          xil_printf("Enter 'y' when ready to begin system operation:\r\n");
   }
   while ((getchar()) != '\r');
   xil_printf("System operation begin:");

}


/*
 * main system loop:
 * read color data, normalize color data, average color data
 * classify color, count color, and output to display
 * advance stepper motor to position for next color object
 *
 */
void SystemOperation() {

   int redTotal = 0, blueTotal = 0, greenTotal = 0;
   int redAverage = 0, blueAverage = 0, greenAverage = 0;
   int black_count = 0, white_count = 0, red_count = 0, green_count = 0, blue_count =
0;
   int undefined_count = 0;

   while (1) {
           DelayMSec(500);
             redTotal = 0;
             blueTotal = 0;
             greenTotal = 0;

             // read color data and calibrate
             for(i=0; i < SAMPLE_SIZE; i++)
             {
             color_data = COLOR_GetData(&pmodCOLOR);
             SystemCalibrate(color_data, &calib_data);
             color_data = SystemNormalizeToCalibration(color_data, calib_data);
           redTotal = redTotal + color_data.r;
           greenTotal = greenTotal + color_data.g;
             blueTotal = blueTotal + color_data.b;
             }

          // average color data over SAMPLE-SIZE readings
          redAverage = redTotal / SAMPLE_SIZE;
          greenAverage = greenTotal / SAMPLE_SIZE;
          blueAverage = blueTotal / SAMPLE_SIZE;

         if ((redAverage < 0x0200) && (greenAverage < 0x0200) && (blueAverage <
0x0200))
                black_count++;
          else if ((redAverage > 0xAAAA) && (greenAverage > 0xAAAA) && (blueAverage >
0xAAAA))
                white_count++;
          else if ((redAverage > blueAverage) && (redAverage > greenAverage))
                red_count++;
          else if ((blueAverage > redAverage) && (blueAverage > greenAverage))
                blue_count++;
          else if ((greenAverage > redAverage) && (greenAverage > blueAverage))
                green_count++;
          else
                undefined_count++;


          // display results to display screen
          xil_printf("\e[1;1H\e[2J");
          xil_printf("Current color object counts:\n\r");
          xil_printf("Black: %d\n\r", black_count);
          xil_printf("White: %d\n\r", white_count);
          xil_printf("Red: %d\n\r", red_count);
          xil_printf("Green: %d\n\r", green_count);
          xil_printf("Blue: %d\n\r", blue_count);
          xil_printf("Undefined: %d\n\r", undefined_count);
          xil_printf("r=%04x g=%04x b=%04x\n\r", redAverage, greenAverage,
blueAverage);

          // set number of step loops for stepper motor to advance conveyor
          ActivateSTEP(25);

   }

}

/*
 *
 * The following three color calibration routines were derived
 * from the Digilent PmodCOLOR library example:
 * vivado-library-v2019.1-
1/ip/Pmods/PmodCOLOR_v1_0/drivers/PmodCOLOR_v1_0/examples/main.c
 *
 * availabe at:
 * https://github.com/Digilent/vivado-
library/releases?_ga=2.23246778.1017225569.1586268606-304264148.1586268606
 *
 */

// initial color calibration
CalibrationData SystemInitCalibrationData(COLOR_Data firstSample) {
   CalibrationData calib;
   calib.min = firstSample;
   calib.max = firstSample;
   return calib;
}


// update color calibration
void SystemCalibrate(COLOR_Data newSample, CalibrationData *calib) {
   if (newSample.c < calib->min.c) calib->min.c = newSample.c;
   if (newSample.r < calib->min.r) calib->min.r = newSample.r;
   if (newSample.g < calib->min.g) calib->min.g = newSample.g;
   if (newSample.b < calib->min.b) calib->min.b = newSample.b;

   if (newSample.c > calib->max.c) calib->max.c = newSample.c;
   if (newSample.r > calib->max.r) calib->max.r = newSample.r;
   if (newSample.g > calib->max.g) calib->max.g = newSample.g;
   if (newSample.b > calib->max.b) calib->max.b = newSample.b;
}


// normalize color data from 16-bit sensor input
COLOR_Data SystemNormalizeToCalibration(COLOR_Data sample,
      CalibrationData calib) {
   COLOR_Data norm;
   norm.c = (sample.c - calib.min.c) * (0xFFFF / (calib.max.c - calib.min.c));
   norm.r = (sample.r - calib.min.r) * (0xFFFF / (calib.max.r - calib.min.r));
   norm.g = (sample.g - calib.min.g) * (0xFFFF / (calib.max.g - calib.min.g));
   norm.b = (sample.b - calib.min.b) * (0xFFFF / (calib.max.b - calib.min.b));
   return norm;
}

/*
 *
 * The following two interrupt functions were derived from
 * Xilinx xgpio library file:
 * xgpio_intr_tapp_example.c
 *
 * available at:
 *
https://github.com/Xilinx/embeddedsw/blob/master/XilinxProcessorIPLib/drivers/gpio/ex
amples/xgpio_intr_tapp_example.c
 *
 * substantially modified for the present application
 */

/******************************************************************************/
/**
*
* This function performs the GPIO set up for Interrupts
*
```

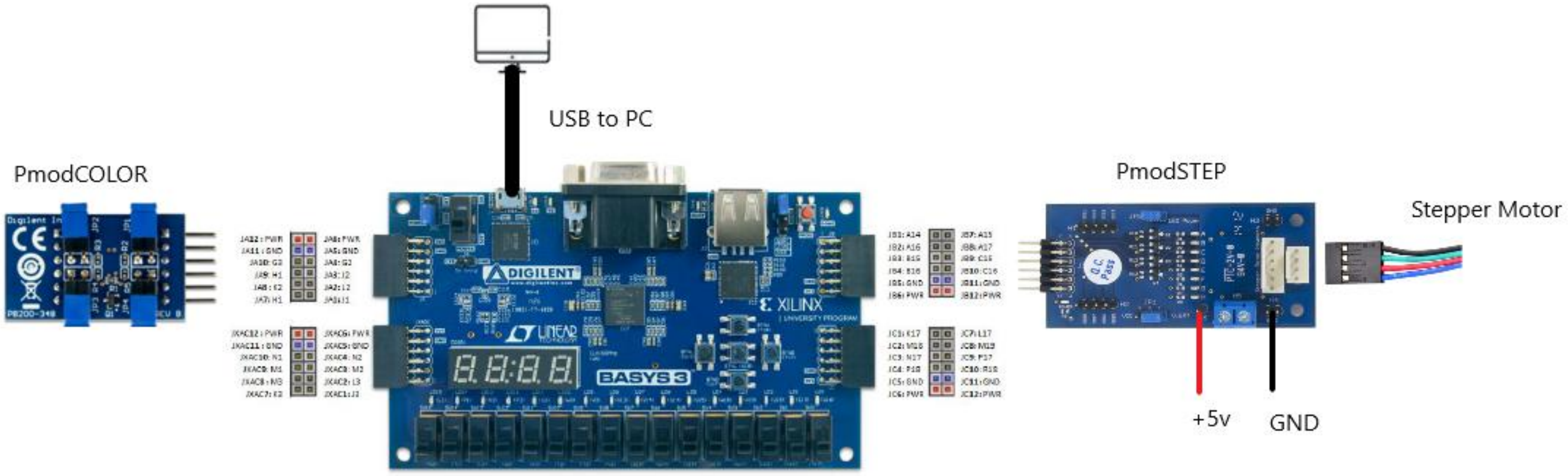
```c
* @param     IntcInstancePtr is a reference to the Interrupt Controller
*            driver Instance
* @param     InstancePtr is a reference to the GPIO driver Instance
* @param     DeviceId is the XPAR_<GPIO_instance>_DEVICE_ID value from
*            xparameters.h
* @param     IntrId is XPAR_<INTC_instance>_<GPIO_instance>_IP2INTC_IRPT_INTR
*            value from xparameters.h
* @param     IntrMask is the GPIO channel mask
*
* @return    XST_SUCCESS if the Test is successful, otherwise XST_FAILURE
*
* @note            None.
*
******************************************************************************/
int XGpioInterruptSetup(INTC *IntcInstancePtr, XGpio *InstancePtr,
                    u16 DeviceId, u16 IntrId, u16 IntrMask)
{
        int Result;
        GlobalIntrMask = IntrMask;

        /*
         * Initialize the interrupt controller driver.
         * Specify the device ID from xparameters.h
         */
        Result = XIntc_Initialize(IntcInstancePtr, INTC_DEVICE_ID);
        if (Result != XST_SUCCESS) {
                return Result;
        }

        /* Connect interrupt service routine */
        XIntc_Connect(IntcInstancePtr, IntrId,
                     (Xil_ExceptionHandler)ResetBtnInterruptHandler, InstancePtr);

        /* Enable the interrupt vector at the interrupt controller */
        XIntc_Enable(IntcInstancePtr, IntrId);

        /*
         * Start the interrupt controller such that interrupts are recognized
         * and handled by the processor
         */
        Result = XIntc_Start(IntcInstancePtr, XIN_REAL_MODE);
        if (Result != XST_SUCCESS) {
                return Result;
        }

        /*
         * Enable the GPIO channel interrupts so that push button can be
         * detected and enable interrupts for the GPIO device
         */
        XGpio_InterruptEnable(InstancePtr, IntrMask);
        XGpio_InterruptGlobalEnable(InstancePtr);

        /*
         * Initialize the exception table and register the interrupt
         * controller handler with the exception table
```

```
       */
      Xil_ExceptionInit();
      Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                     (Xil_ExceptionHandler)INTC_HANDLER, IntcInstancePtr);

      /* Enable non-critical exceptions */
      Xil_ExceptionEnable();
      return XST_SUCCESS;
}

/****************************************************************************/
/**
*
* Interrupt handler for push-button event. Pause system and either resume or exit.
*
* @param     CallbackRef is the Callback reference for the handler.
*
****************************************************************************/
void ResetBtnInterruptHandler(void *CallbackRef)
{
      int answer;
      XGpio *GpioPtr = (XGpio *)CallbackRef;
      xil_printf("System paused. \r\n");
      xil_printf("Press 'r' to resume or 'x' to eXit system. \r\n");
      answer = getchar();
      while ((getchar()) != '\r');
      if (answer == 'r') {
             xil_printf("Resume system operation.\r\n");

             XGpio_InterruptClear(GpioPtr, GlobalIntrMask);  // clear interrupt
      }
      else {
             xil_printf("Exiting the system.");
             COLOR_SetLED(&pmodCOLOR, 0);
             exit(0);
             }
      IntrFlag = 1;

}


// program main
int main(void) {
   SystemInit();
   SystemCalibration();
   SystemOperation();
   return 0;
}
```

Vivado Block Design



Wiring Diagram


USB to PC
PmodCOLOR
PmodSTEP
Stepper Motor
+5v
GND
JA12 : PWR
JA11 : GND
JA10: G3
JA9: H2
JA8 : K2
JA7: H1
JA6: PWR
JA5: GND
JA4: G2
JA3: J2
JA2: L2
JA1: J1
JXADC12 : PWR
JXADC11 : GND
JXADC10: N1
JXADC9: M1
JXADC8 : M3
JXADC7: K3
JXADC6: PWR
JXADC5: GND
JXADC4: N2
JXADC3: M2
JXADC2: L3
JXADC1: J3
JB1: A14
JB2: A16
JB3: B15
JB4: B16
JB5: GND
JB6: PWR
JB7: A15
JB8: A17
JB9: C15
JB10: C16
JB11: GND
JB12: PWR
JC1: K17
JC2: M18
JC3: N17
JC4: P18
JC5: GND
JC6: PWR
JC7: L17
JC8: M19
JC9: P17
JC10: R18
JC11: GND
JC12: PWR
DIGILENT
XILINX
BASYS 3