Aerial Collision Avoidance System Final Report

Bradley Bergerhouse
Austin Wenzel
Nelson Gaske

Advisor: Dr. Malinowski

ABSTRACT

The Aerial Collision Avoidance System is designed to implement collision avoidance on a quadcopter platform using minimal sensor input. The project consists of wireless communication system, IR range-finders, a real-time Linux system, and a quadcopter aerial platform. The quadcopter is controlled by a computer connected to the system wirelessly. The Linux system minutely alters given commands to avoid obstacles based on ranging sensor inputs. The system is also designed to keep the aerial platform in place or land it safely in case of communication link breakdown.

TABLE OF CONTENTS

INTRODUCTION

Quadcopters maintain the ability to move rapidly and agilely through the air much like a plane while maintaining the helicopter's ability to hover and move at low airspeeds. By equipping such a copter with distance sensors and cameras an autonomous aerial vehicle is created which can avoid nearby obstacles regardless of velocity. Information gathered by these sensors will then be used to navigate the quadcopter through spaces too tight for other styles of autonomous aerial vehicles.

GOALS

Since this is the first time an aerial robotic project is attempted in our ECE department it was hard to foresee how much work can be accomplished. Original project was aiming for a complete autonomous quadcopter platform. However, in the course of research and development it turned out that these goals were not obtainable in the time allotted. The following list is the original goals of the project:

- To develop a quadrocopter platform
- Autonomously navigate through narrow passages using onboard and stationary sensors
- Avoid obstacles using video and other sensor feedback
- Implement backup fly-by-wire controls for safety and testing

After numerous discoveries it was determined that the time frame for the original project was much too short. As such, the goals were revised with goals that were possibly obtainable within the time frame.

- Use various sensors to detect obstacles
- Autonomously avoid obstacles in the path
- Eliminate collisions due to human error

The revised goals indicate the system was more complicated and nuanced than originally anticipated. After encountering difficulties it was determined that time was best spent heavily documenting the process of attempting to meet the revised goals in lieu of the original goal specifications. The documentation left is a complete set of materials and resources believed to be necessary and sufficient for the completion and extension of the original goals in future projects.

BLOCK DIAGRAM AND SYSTEM DESCRIPTION

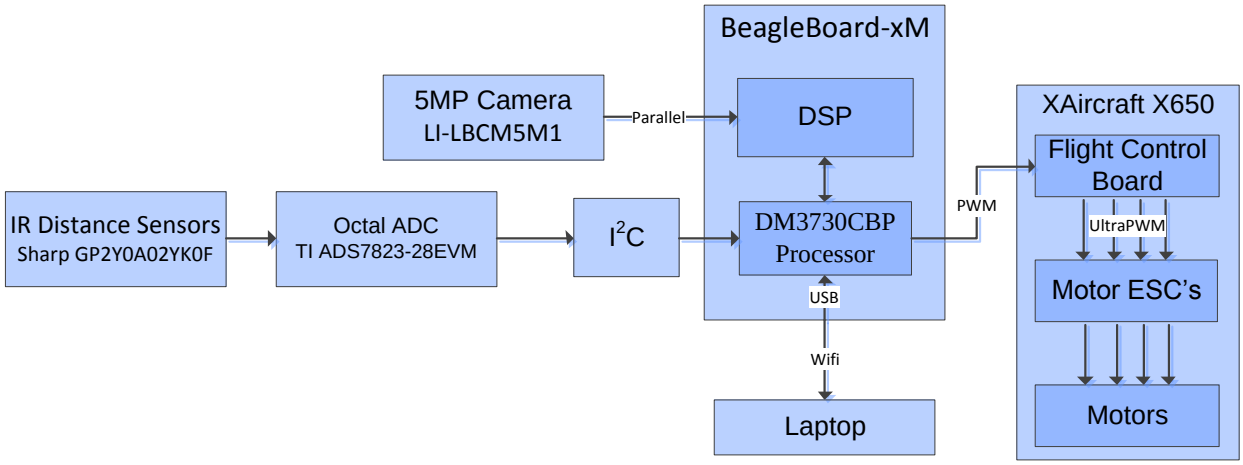


Figure 1 System Block Diagram

There are various subsystems, including the XAircraft X650 quadcopter platform, BeagleBoard-xM, IR distance sensors, and camera for visual input.

The entire project is based on the XAircraft X650 quadcopter as the flying platform. This choice was made based on its lifting capabilities and cost effectiveness. The platform contains a flight controller, attitude heading reference system (AHRS) , electronic speed controllers (ESC) and brushless motors. The flight controller receives input usually from a remote control transmitter/receiver combo, but for this project it receives input from the BeagleBoard directly. This relationship can be seen in the right of Figure 1. The AHRS is responsible for stability of the platform by transmitting attitude and heading information to the flight controller. The flight controller then transmits control signals to the ESC's which drive the brushless motors.

The BeagleBoard-xM is a high end hobbyist embedded computing solution. It contains a TI OMAP processor, which includes integrated ARM and DSP cores. The BeagleBoard is the main component of this project and can be seen in the center of Figure 1. It has many input and output ports including USB, audio, HDMI, PS/2, and serial. In addition to its many ports, it runs Linux which the teams familiarity with increased productivity throughout the project.

Infrared distance sensors are used to aid in obstacle avoidance. The interface to the BeagleBoard can be seen on the left of Figure 1. The decision criteria for the sensors were predominately based on effective distance, as the project needed sensors that functioned from ~15-150cm. The Sharp GP2Y0A02YK0F sensor was chosen primarily based on meeting the range requirements and low cost.

A Leopard Imaging 5 megapixel camera is used for image processing in combination with the BeagleBoard. The camera was chosen because the BeagleBoard has a header specifically design for this style camera. The 5 megapixel model creates a high resolution image providing large amounts of information for image processing.

HARDWARE

The following sections describe the hardware subsystems used in this project. The major hardware components are shown in the block diagram section above and consist of: platform power distribution, sensors, and the BeagleBoard. The list below indicates equipment either purchased or modified for this project in particular.

- XAircraft X650 Quadcopter Platform
- BeagleBoard-xM
- Leopard Imaging 5MP Camera
- Belkin Wireless USB network card
- IR distance sensors (Sharp GP2Y0A02YK0F)
- Octal ADC TI ADS7823-28EVM
- Sparkfun Logic Level Converter (1.8V to 5V)
- Battery (11.1V 3S LiPo)
- 5V Switching Power Regulator

In addition to materials used for the platform and project itself, a number of laboratory devices were used in the building and testing of individual project components. The list below shows the particular pieces of equipment used in the lab for testing and debugging purposes.

- **EQ-2690** Tektronix TDS 2024B Oscilloscope
- **EQ-2385** Agilent E3630A DC Bench supply
- **EQ-2140** Fluke 45 DMM
- **EQ-2700** Agilent 33220A Waveform Generator

PLATFORM POWER DISTRIBUTION

The quadcopter platform requires a typical 11.1V battery, so a battery was chosen with an output voltage of 11.1V and a high capacity. Since the project has numerous subsystems, a 5000mAh 3 cell Lithium Polymer battery was chosen to power the project. The platform flight controller and ESCs run directly off of the 11.1V battery, while the rest of the subsystems require 5V. To accomplish this, a switching regulator was borrowed from another project and rebuilt to be lighter with a smaller footprint. The regulator provides a stable 5V output for the rest of the platform subsystems. The circuit we used was one of the test circuits for the LM2576HV-ADJ part, with one addition. A potentiometer was included to tune the output voltage to precisely 5V, because the BeagleBoard needs a consistent supply of 5V. The circuit for the switching regulator can be seen in Appendix A. Figure 2 below displays how power is distributed across the platform.

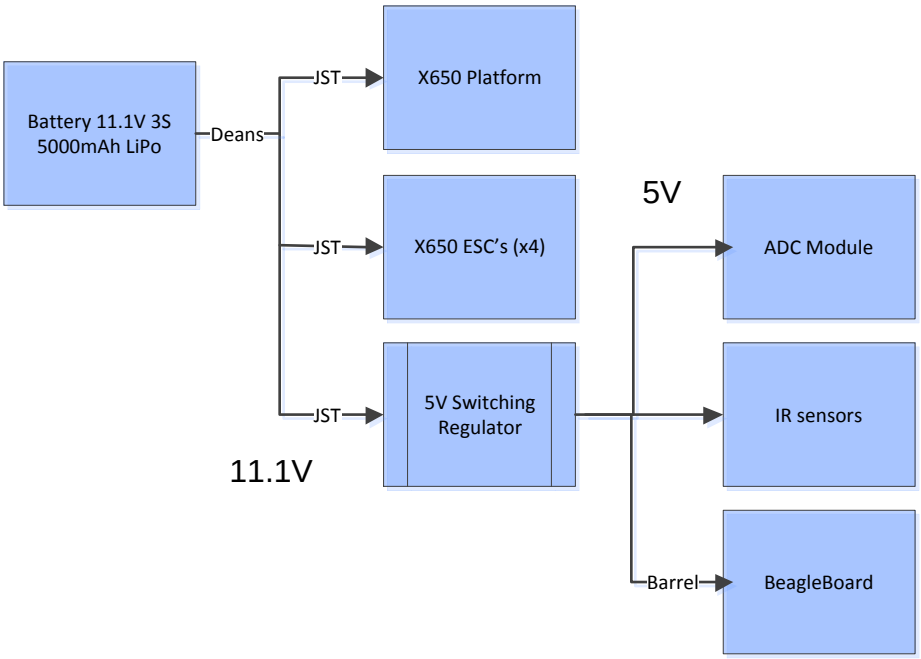


Figure 2 Platform Power Distribution

SENSORS

The sensors used for this project include:

- 5MP camera (LI-LBCM5M1)
- IR distance sensors (Sharp GP2Y0A02YK0F)

The purpose of the camera is to provide feedback on how fast and far the platform has traveled through image processing. This task can be performed by capturing images below the platform where "markers" will be placed along the platforms flight-path. These "markers" can be dots of specific color that the software would recognize. The camera plugs directly into a socket on the Beagleboard, where it communicates over a parallel protocol, and is to be positioned on the platform so it is pointing downward. The image resolution is reduced to a smaller resolution, such as 800x600, to perform image processing quicker.

Six Sharp GP2Y0A02YK0F Infrared distance sensors are used to determine distances in various directions. The following sections will detail the layout of the sensors, communication protocol, distance calculation, and the software system.

The IR sensors are placed in the positive and negative directions of Euclidian 3-space. These directions correspond to north, south, east, west, up, and down relative to the center of the quadcopter's compass. This arrangement is used to give the quadcopter a basic view of its surroundings. The Inter-integrated circuit (I2C) communication protocol is used to communicate with the sensors through an ADC.

I2C FOR SENSOR INPUT

I2C is present on both the BeagleBoard and ADC board. It provides adequate communication bandwidth and accessibility, making it the obvious choice for data acquisition. The ADC is connected to the BeagleBoard over the traditional 3-wire I2C setup (GND, SCL, SDA), with a level translator in between to translate the logic level from BeagleBoard (1.8V) to ADC (5V) levels. Each sensor is connected to the ADC through a separate channel and the ADC acts as a multiplexer to read multiple channels over a single I2C bus. The level translators go in between the BeagleBoard and ADC. Here is the arrangement of the translators on the I2C bus:

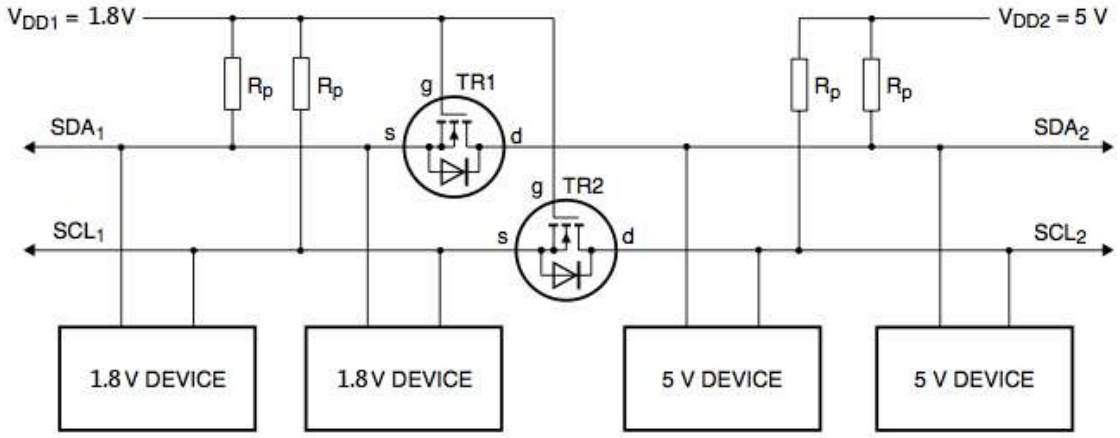


Figure 3 Level shifting circuit

Figure 3 is only a generic level shifting circuit; the circuit actually used can be found in Appendix B. A part from Sparkfun [1] was utilized that provided the exact circuit needed for I2C translation for cheaper than could be made individually for use with this project.

SENSOR DISTANCE CALCULATION

Below in Figure 4 it can be seen that sensor output voltage is a nonlinear function. The nonlinear function is difficult to use when calculating discrete sensor values. The solution is to linearize the relationship between distance and an arbitrary function of voltage. A common method of linearizing IR range sensor data is through the use of a power function as in the code line below from Physical Computing [2]:

```
float distance = 12343.85 * pow(analogRead(sensorPin), -1.15)
```

As is apparent above, the distance calculation utilizes a floating point power function with a non-integer power. Normally in large systems, the power function is acceptable however the BeagleBoard's processor does not contain a floating point unit and must emulate floating point calculations in software. Due to the performance penalty of emulating floating point power functions the standard linearization function above is unsuitable and should not be used in a system expecting near real-time performance.

In lieu of the linearization above, this project uses a linear piecewise interpolation between experimental data points. The interpolation can be seen in the Figure 4 below as the straight lines connecting subsequent data

points. The accuracy of this method can be increased linearly with respect to the number of data points used in the interpolation, allowing additional accuracy with minimal code maintenance should it be required in the future.

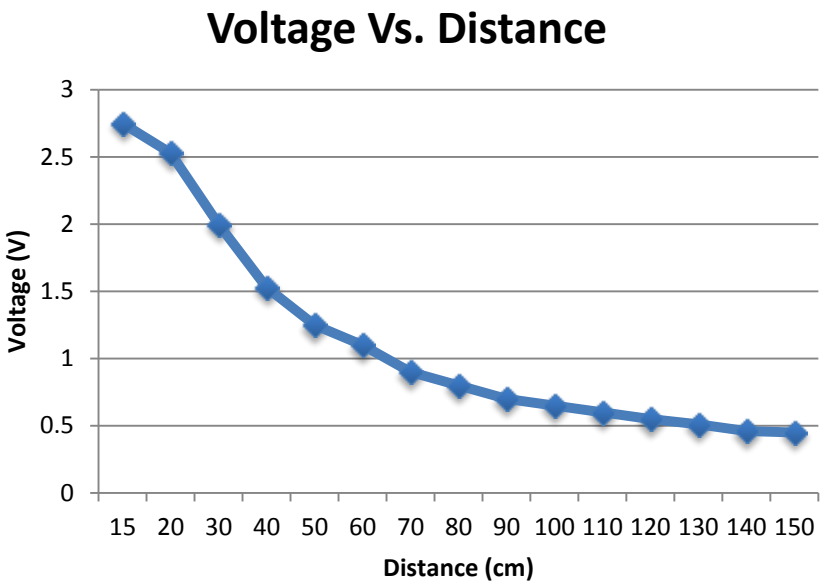


Figure 4 Sensor Output vs. Measured Distance

SENSOR CODE

The code that was used to communicate with the sensors can be found in Appendix C. It is used to acquire data from each sensor and compute the distance to the nearest object, if available. The sensor data is then placed in a shared memory location.

BEAGLEBOARD OMAP 3530

The Beagleboard was chosen for this project because it had the TI OMAP 3530 processor chip onboard. The OMAP 3530 processor controls all peripherals, subsystems, and memory connections on the Beagleboard using the L3 and L4 busses. The main subsystems that are used in this project are the ARM and DSP cores. The ARM core is used to control the I2C and the PWM functionalities and the DSP core is used for capturing images and image processing.

The OMAP processor is capable of running uC Linux or a lightweight regular Linux distribution. When running a full-fledged Angstrom Linux distribution with a graphical interface, the BeagleBoard consumes 1.3A at 5V. Additionally the BeagleBoard weighs in at 66 grams, which is very light considering the processing power available. These were the main reasons the BeagleBoard was chosen for this project.

DSP HARDWARE

The DSP core on the BeagleBoard is primarily used for processing digital signals like images or sound at a very fast rate. Finding out how the camera and DSP core interact was a challenging task. First of all, the camera

module sends in pixel data in parallel to the camera image signal processor or ISP on the OMAP 3530 chip. The camera ISP is connected to the DSP core through the L3 bus. The L3 bus allows for other interconnections from the DSP core to other systems such as the ARM core or to memory.

The camera ISP supports up to 12-bits of RAW RGB data which can be used with the 5 MP leopard imaging camera. The RAW data is transferred in parallel, when in SYNC mode, through signal cam_d0 to cam d11 where one pixel is sampled every cam_pclk rising edge. This process is shown below in Figure 5 where cam_hs and cam_vs are used to identify active pixels. The RAW data can be sent to the DSP core to be processed then be sent to memory.

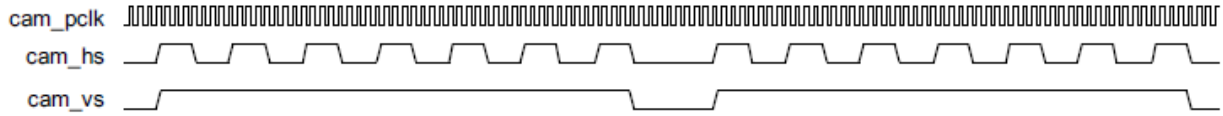


Figure 5 Camera ISP clock timing

SOFTWARE

The software subsystems in the project form a complex and comprehensive set of functions and routines including: reading the sensors, pulse width modulation, and image processing. This section details the Linux kernel providing the foundational processing structures, PWM output generation for flight control, I2C sensor data acquisition, and color distance calculation.

LINUX KERNEL AND XENOMAI

Linux is the chosen operating system (OS) for this project to provide high level abstractions of hardware and data management. This OS was chosen due to its efficiency when running on an embedded system and the presence of many default device drivers necessary for operation. Initially the Angstrom Linux distribution with kernel 2.6 was chosen to provide an initial on-board development environment to explore the software requirements to operate the hardware in the necessary fashion. The Angstrom distribution provided a test bed to determine rough power and timing requirements.

The platform currently uses a Linux kernel 2.6 patched with Xenomai – a real-time subsystem for Linux – which decreased power draw by 80% (mostly due to lack of graphical output and fewer processes running) and increased timing accuracy by three orders of magnitude. The current power draw is around .2A and the kernel has a timing accuracy to the microsecond. The power consumption is achieved by disabling video output on the HDMI port and Xenomai running the kernel in a hypervisor to create a pseudo real-time OS. The precision timing functions provided by a real-time environment are necessary for the PWM functionality and communication with the platform.

The Xenomai system supports a number of Application Programming Interfaces (APIs) called Skins. Each Skin is a complete set of functions emulating a proprietary API. The particular Skin used in this project is the Real Time Application Interface (RTAI) as it is a default and commonly used Skin. The particular function used to generate timings for PWM signals is **rt_sleep(RTIME delay)** which suspends a process for **delay** nanoseconds. Additionally the Xenomai patch integrates into the default threading system in Linux which spawns non-real-time

threads. The spawned threads however can become nearly real-time through the use of precision timing functions which allow the thread to interrupt kernel code at the end of the timing.

PWM OUTPUT

The industry standard RC PWM consists of a number of pulses occurring with a frequency of 50Hz. The pulses vary from -100%PWM to +100%PWM indicating a high voltage time of 1ms to 2ms respectively. The remainder of the 20ms period is low voltage. The 0% PWM then becomes a pulse with a 1.5ms high time followed by 18.5ms of low voltage. Figure 6 below demonstrates the shape of the RC pulses.

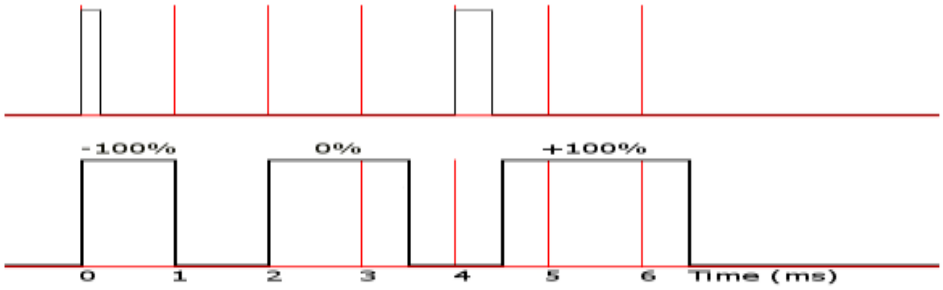


Figure 6 RC PWM timing

An aerial platform such as a quadcopter requires 4 communication channels for total control and a fifth channel for gear/mode. The five channels required are pinned out to five General Purpose Input Output (GPIO) pins on the BeagleBoard xM's expansion header. Each pin is controlled by an individual software thread in charge of timing the switching. This functionality is split between an initialization/control file <PWM.c> (Appendix D) and the pin control file <PWM_Funcs.c> (Appendix E).

PWM.c contains the initialization code using the Linux device drivers. These are used because the initialization is not a time sensitive process and the time taken for the driver to properly initialize the pin will have no effect on the final output. The program then initializes a shared memory segment mapped to the physical location of the GPIO control regions. The memory and some control information is then passed to the generation function spawned as a new thread. The generation function uses direct memory access via the shared segment to control the pin state which is the updated by the CPU automatically.

The pin voltage for a logic high is 1.8V and low voltage is 0V. The quadcopter platform relies on a logic level input of 0V low to 5V high. To facilitate the conversion a set of logic translators identical to those used for the I2C bus are employed to push the 1.8V up to 5V while maintaining the 0V low voltage. Figure 3 demonstrates this conversion where the SDA and SDL lines will be replaced with PWM channels.

DSP SOFTWARE

Determining which algorithm would be used for image processing was a very crucial decision for this project. At first, the idea of canny edge detection was brought up where the edges of the “markers” would be marked and compared to previous images. But that idea was changed to doing corner detection on the “markers” and comparing the corner points of previous images.

After extensive research, the Smallest Univalue Segment Assimilating Nucleus, or SUSAN, corner detection would be used to perform edge detection. The SUSAN corner detection algorithm takes the pixel in question, known as the nucleus pixel, and places a circular mask around the nucleus pixel. All pixels in the mask are

compared to the nucleus pixel and if that pixel's brightness matches the nucleus pixel's brightness it is marked in the mask as the "USAN". **Invalid source specified.** Using the USAN, corners can be determined by marking where the edges of the USAN meet. This is ideal because every pixel doesn't need to be tested as the nucleus to find the corners; this will decrease processing time and processing usage.

Once the algorithm for image processing on the DSP was selected, code needed to be written and compiled for use on the DSP core. This proved to be a challenging task since there was a lack of knowledge when it came to interfacing with the DSP and the online documentation pertaining to this task are scattered. It was found out that the code for the DSP needed to be cross compiled on a host computer and then the compiled program is transferred to the Beagleboard for the DSP to run it. The first compiler that was tried was c6run since the Linux community had a guide to setting it up on the Beagleboard. Unfortunately, the host kernel version was not compatible with the version of c6run installed and there were missing required files. After that compiler failed and no quick solution was available it was decided to try another compiler since time was running out. The next compiler that was tried was CodeSourcery which needed a registration in order to obtain the software. But the email that provided a link to the software download never came and time was short.

There were factors that caused delays in the project which caused the DSP part of the project to fall behind schedule. First, going into this project there was a lack of knowledge in the team pertaining to DSP chips and how to program them. Another factor that caused delay was learning how the camera module and DSP interfaces with each other. This was crucial research so that when the cross compiler was properly set up on the Beagleboard writing the image processing code could begin immediately. Getting a working cross compiler on the Beagleboard and host computer was the next issue that caused the most delay in the project. This was supposed to be a quick step but took the most time due to compatibility issues with the host computer. Due to the lack of time, having the camera module directly interact with the DSP core was not possible with the given time left.

Instead, it was suggested by our advisor to use Video4Linux APIs to capture an image and to use a color distance algorithm as a proof of concept that it was possible to capture images on the Beagleboard and process them. The Video4Linux APIs utilize the DSP core by bringing in the data from the camera to the DSP core which then stores it into a memory buffer where the ARM core receives the data to be written. Since Video4Linux doesn't allow for direct processing of pixels on the DSP core the color distance algorithm is processed on the ARM core before the data was written.

Color distance algorithm was used because it is a color filter algorithm that calculates a pixel's closeness to the selected color by using a distance equation. It does this by first calculating the difference of each color component (the following code selects the color red).

$$DR = R0 - 255;\ DG = G0 - 0;\ DB = B0 - 0;$$

Next it calculates the distance using the difference of the color components and the max value for the colors (255).

$$DISTANCE = 255 - FLOOR(SQRT(DR * DR + DG * DG + DB * DB));\ IF(DISTANCE < 0)\{DISTANCE = 0;\}$$

Then it updates the pixel to the distance calculated and the weighted sums for the center, width, and height.

$$R0 = DISTANCE;\ G0 = DISTANCE;\ B0 = DISTANCE;$$
$$SUMH = SUMH + ROW * DISTANCE;\ SUMW = SUMW + COL * DISTANCE;\ SUMD = SUMD + DISTANCE;$$

This color distance process for the code is shown below in Figure 7.

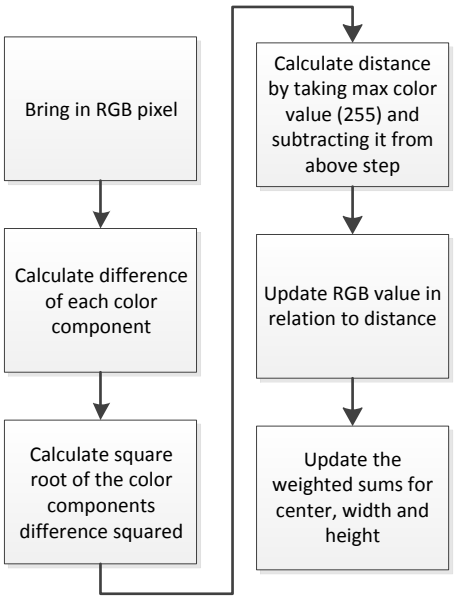


Figure 7 Color Distance Process

COLOR DISTANCE RESULTS

The given code above filters all other colors except the color red. So when the code is ran on an image all pixels that are closer to the RGB value of 255,0,0 will appear "close" or be brighter and all pixels that are far away from the RGB value of 255,0,0 will appear "far" or darker. This is shown below in figure 8 where the left image is unfiltered and the right image is filtered with the color distance algorithm.

Figure 8 Color Distance Results

PROGRESS TOWARDS COMPLETION

Currently the system is able to collect sensor information and accurately detect obstacles. The collision avoidance algorithm has not yet been started but the hardware and software systems are in place for an algorithm to be developed. Finally, the elimination of collisions due to human error is contingent upon obstacle avoidance and is therefore not yet achieved.

Despite not meeting the majority of goals for the project, significant progress has been made towards a platform which can avoid obstacles prior to human intervention. The following sections will discuss the progress achieved and not achieved in a number of key facets of the project.

SYSTEM INTEGRATION

The lack of prior work led to spending approximately half of project work time evaluating system components. Due to the magnitude of this project, special care was taken in evaluating all platform options and other components. Some components had a long lead time forcing the project to begin later than anticipated.

System integration work began in earnest upon receipt of the platform and other components. To date, the IR sensors have been mounted and interfaced with the BeagleBoard. However, a proper mounting method for the BeagleBoard has not been determined. Rotor guards have been fashioned to protect both people and the platform from collision. All systems currently can run off of the on board battery with the use of the switching regulator. The PWM outputs are ready to be interfaced with the flight controller upon completion of a collision avoidance algorithm.

SOFTWARE FUNCTIONS

The BeagleBoard offers great power and flexibility, which is the reason why the board was selected. However, the board excels at no particular function. Linux was chosen as the operating system on the platform to provide convenient interfaces for hardware devices, as well as the team's familiarity with the OS.

Angstrom Linux is the default distribution for the BeagleBoard-xM so it was the distribution of choice, providing full hardware support for the board. Angstrom was used as a test bed for software code with the intention of moving code to a headless version of Linux at a later date to conserve power and increase computational speed. The Xenomai real-time framework was chosen after determining that the timing for PWM generation was required to be more accurate than could be provided by a non-real-time kernel. Differences in kernel revisions ensured that the process of patching a particular kernel with Xenomai would be long and arduous, but ultimately completed. Unfortunately Xenomai does not officially support the BeagleBoard-xM revision C that the project used and as such, no hardware external to the processor is functional.

PWM generation was originally a script which toggled pin outputs using the default GPIO hardware drivers provided by the Linux kernel. The waveforms generated by the script had a substantial amount of jitter and the script was migrated to a C program. However the C program suffered from similar amount of jitter due to there being no real-time functionality in the default kernel. The C program running identical code under the Xenomai framework performs exceptionally well and reduced the jitter the jitter to acceptable levels.

I2C is functioning and sensor data can be acquired through C code that runs on the BeagleBoard. The I2C chain works and the level shifter works flawlessly. A few problems were encountered along the way with how the Linux kernel communicates over the I2C bus. These problems were rectified by using a logic analyzer to monitor the I2C bus traffic and diagnose the problem.

With the aid of the Xenomai real-time framework, the project has achieved real-time performance during PWM generation. The I2C code will also be able to run under Xenomai as real-time code, allowing for data to be acquired on a periodic basis. Communication with the DSP core, when complete, will also be initiated on a periodic basis, allowing for images and their information to be processed with minimal CPU overhead. As a whole, once every system component is fully assembled and integrated, there should be no issues running the platform control applications on a real-time periodic basis with accurate timing. $1/50^{th}$ of a second would be a baseline for obstacle avoidance because of the 50Hz pulse rate of the RC PWM pulses. This time is the minimum possible response time due to the pulse rate of the RC PWM pulses.

FUTURE WORK

Although this project has not been completed, there is plenty of room for future work on the platform. This team was unable to mount the subsystems to the platform; this would be the next logical step for a new team. A majority of the time for this project was spent learning how to utilize the following functionality of the BeagleBoard that goes beyond standard Linux-based computing platforms: real-time RC PWM output, running code on an integrated DSP core, and I2C interfacing. Real-time PWM generation was initially believed to be an almost trivial script to toggle GPIO pins on the expansion header of the BeagleBoard, unfortunately the real-time PWM generation proved to be a difficult task. The difficulty to get a non-real-time Linux system to reliably output signals for platform control was no small feat. The discovery of the Xenomai real-time framework saved the PWM portion of the project and eventually allowed the project to adequately output PWM signals. Another time-consuming problem was trying to run code on the DSP core. Since the BeagleBoard has an onboard DSP core, it was believed that interfacing code for image and signal processing would be a simple process, but it was not. There was no discovered way to directly interface to the DSP core, which provided a serious problem for the image processing portion of the project. Hopefully a new version of the Linux kernel that runs on the BeagleBoard will fix the DSP to ARM interface and easily allow code to run on the DSP core. The final problem the team encountered that went above a typical Linux platform was I2C interfacing. The I2C libraries included with Angstrom suffered from a lack of documentation that gave the team a rough time developing code to interface over the I2C bus. Although Linux is the team's ideal embedded OS, the BeagleBoard provided the aforementioned problems that took up a significant amount of time.

The documentation completed as part of the project should alleviate some of the problems of learning the new hardware and its limits. This team wanted to utilize the DSP to perform image processing necessary for this project. It is recommended for future groups to find a working cross compiler and complete the DSP implementation.

Once the hardware subsystems are mounted to the platform, the next step would be to integrate the software components presented in this project into a single program. The DSP code could then be loaded into the DSP core from the single program to provide image processing services required for object detection. It would be reasonable to expect the next project to completely finish the above software and hardware integration to fully provide an aerial platform to the electrical engineering department. Fly-by-wire controls replacing the standard

transmitter/receiver combination with a laptop communicating with the BeagleBoard over a wireless communication protocol could be implemented as a test application for the quadcopter platform.

Following the integration described above, the platform will provide the department with an excellent source of future project topics including: aerial navigation and mapping, 3-dimensional localization, and aerial drones. These topics will be filled with new and exciting projects for future teams and provide further opportunity to expand the project possibilities at Bradley University.

REFERENCES

[1] Sparkfun. [Online]. http://www.sparkfun.com/products/8745

[2] Eric. (2011, April) Physical Computing. [Online]. http://dm.risd.edu/courseblogs/7026-s11/how-to-linearize-sharp-ir-sensor-data/

[3] (2010) XAircraft Wiki. [Online]. http://www.xaircraft.com/wiki/X650(en)

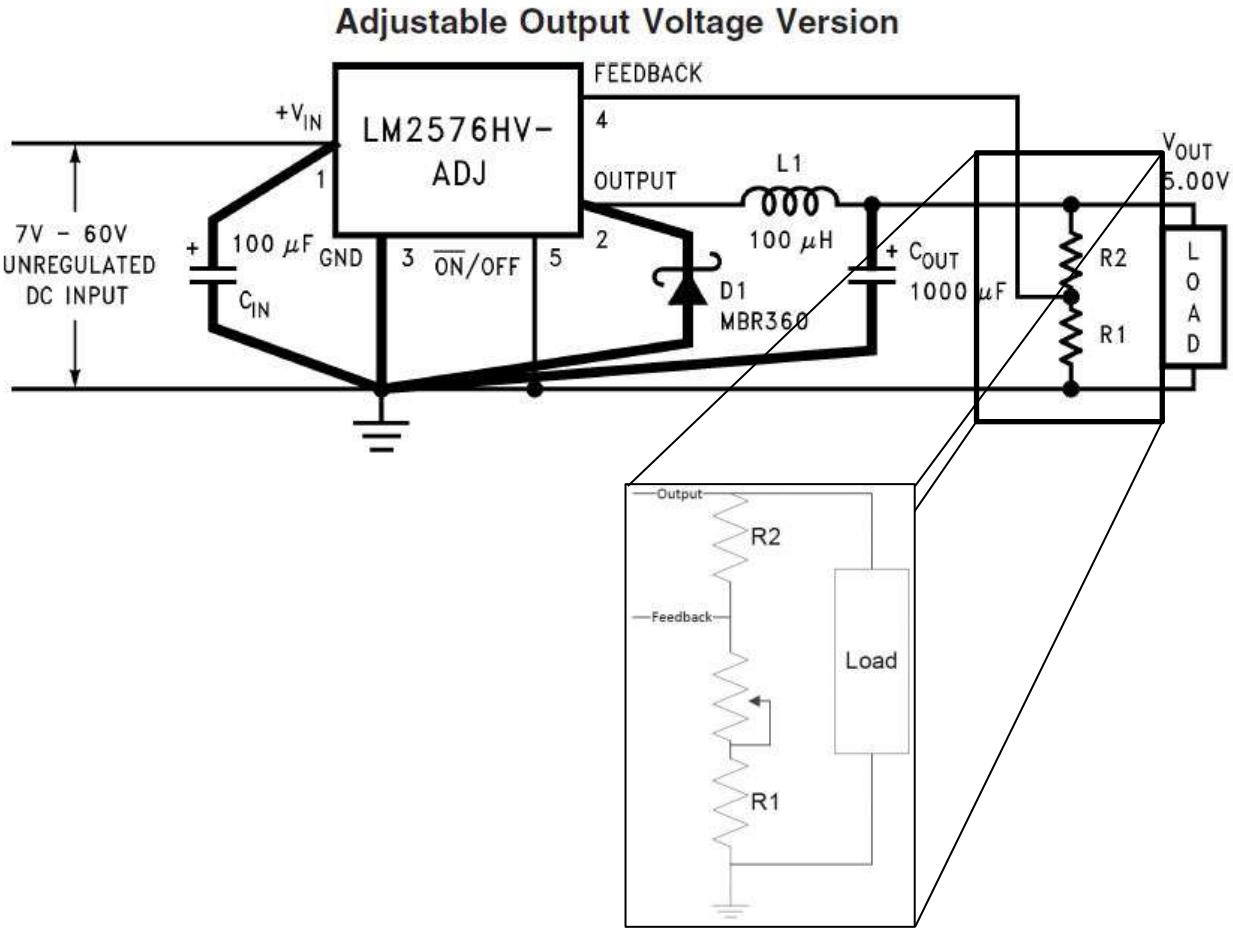
[4] (2006, December) SHARP. [Online]. http://www.sharpsma.com/webfm_send/1487

[5] (2010, April) BeagleBoard. [Online]. http://beagleboard.org/static/BBxMSRM_latest.pdf

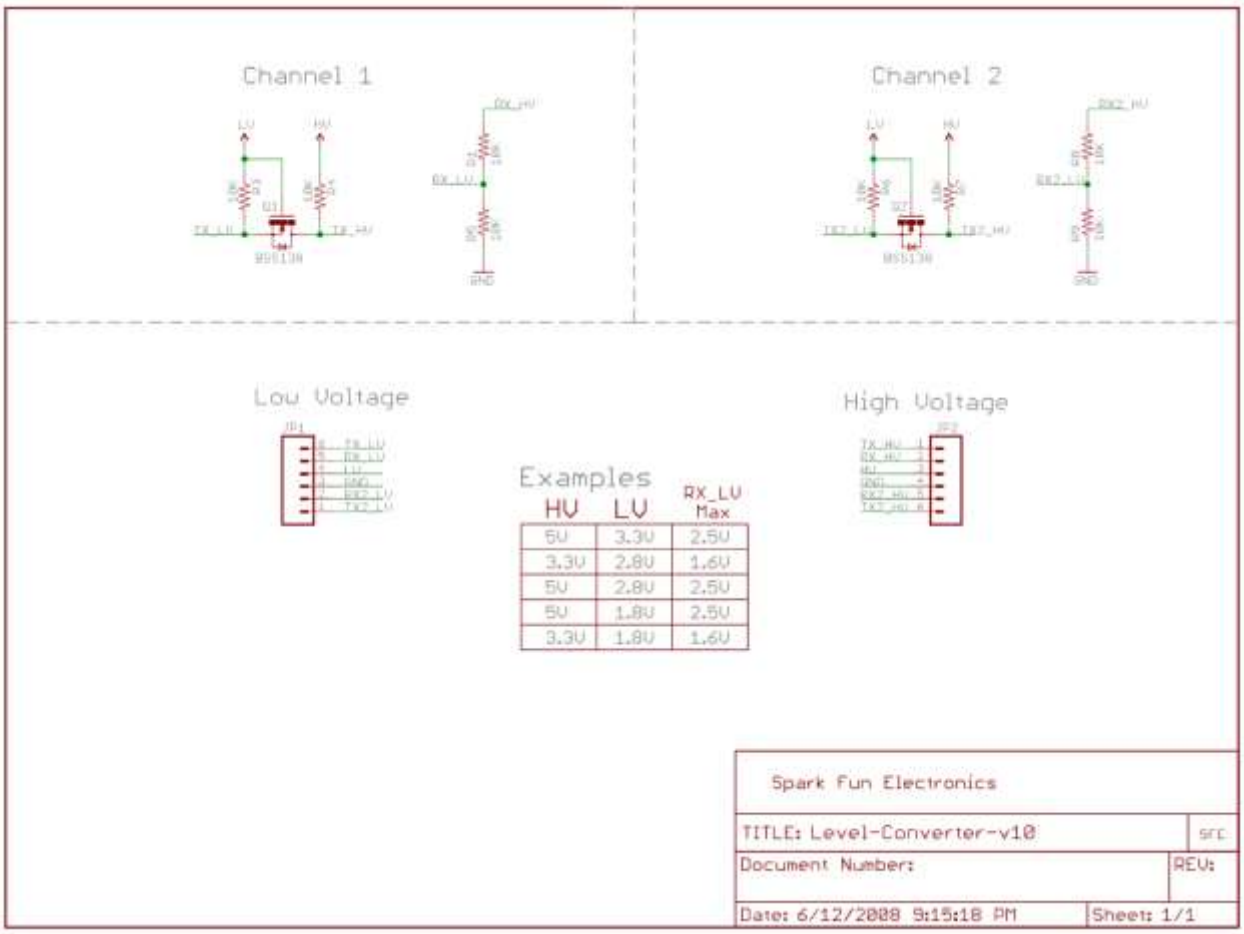
[6] (2004, June) TI. [Online]. http://www.ti.com/lit/ug/slau124/slau124.pdf

[7] Leopard Imaging. [Online]. https://www.leopardimaging.com/uploads/LI_LBCM5M1_Camera_Board.pdf

APPENDIX A: SWITCHING POWER REGULATOR



APPENDIX B: I2C TRANSLATOR



APPENDIX C: I2CTEST

```
//#include <glib.h>
//#include <glib/gprintf.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>#include <stdlib.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
double lookup (float input);
int main(void) {
    int file;
    char filename[40];
    const char *buffer;
    int addr = 0x4b;//0b00101001;        // The I2C address of the ADC


    sprintf(filename,"/dev/i2c-2");
    if ((file = open(filename,O_RDWR)) < 0) {
        printf("Failed to open the bus.");
        /* ERROR HANDLING; you can check errno to see what went wrong */
        exit(1);
    }

    if (ioctl(file,I2C_SLAVE,addr) < 0) {
        printf("Failed to acquire bus access and/or talk to slave.\n");
        /* ERROR HANDLING; you can check errno to see what went wrong */
        exit(1);
    }

    char buf[10] = {0};
    float data;
    char channel;
      int i;

while(1){
    for(i = 0; i<6; i++) {
            buf[0] = 0b10010110;
      if (write(file,buf,1) != 1) {
        /* ERROR HANDLING: i2c transaction failed */
        printf("Failed to write to the i2c bus.\n");
        buffer = strerror(errno);
        printf(buffer);
        printf("%d\n\n",i);
            }
            switch(i){
                  case 0:
                        buf[0] = 0b10000100;
                        break;
                  case 1:
                        buf[0] = 0b11000100;
                        break;
```

```
                    case 2:
                          buf[0] = 0b10010100;
                          break;
                    case 3:
                          buf[0] = 0b11010100;
                          break;
                    case 4:
                          buf[0] = 0b10100100;
                          break;
                    case 5:
                          buf[0] = 0b11100100;
                          break;
                }if (write(file,buf,1) != 1) {
            /* ERROR HANDLING: i2c transaction failed */
            printf("Failed to write to the i2c bus.\n");
            buffer = strerror(errno);
            printf(buffer);
            printf("%d\n\n",i);
                }
            // Using I2C Read
                buf[0] = 0b10010111;
                write(file,buf,1);
            if (read(file,buf,2) != 2) {
                /* ERROR HANDLING: i2c transaction failed */
                printf("Failed to read from the i2c bus.\n");
                buffer = strerror(errno);
                printf(buffer);
                printf("\n\n");
            } else {
                data = (float)((buf[0] & 0b00001111)<<8)+buf[1];
                data = data/4096*5;
                channel = i;//((buf[1] & 0b00110000)>>4);
                printf("Channel %02d Data:  %02f cm: %01f\n",channel,data,
lookup(data));
            }
        }
          printf("\n\n\n\n");
        //unsigned char reg = 0x10; // Device register to access
        //buf[0] = reg;
        sleep(1);
}
}
double lookup (float input){
if (input >= 2.8)
      return 0;
else if(input >= 2.53)
      return ((15-20)/(2.75-2.53))*(input-2.53) + 15;
else if(input >=2)
      return ((20-30)/(2.53-2))*(input-2) + 20;
else if(input >=1.53)
      return ((30-40)/(2-1.53))*(input-1.53) + 30;
else if(input >=1.25)
      return ((40-50)/(1.53-1.25))*(input-1.25) + 40;
else if(input >=1.1)
      return ((50-60)/(1.25-1.1))*(input-1.1) + 50;
else if(input >=.9)
```

```
      return ((60-70)/(1.1-.9))*(input-.9) + 60;
else if(input >=.8)
      return ((70-80)/(.9-.8))*(input-.8) + 70;
else if(input >=.7)
      return ((80-90)/(.8-.7))*(input-.7) + 80;
else if(input >=.65)
      return ((90-100)/(.7-.65))*(input-.65) + 90;
else if(input >=.6)
      return ((100-110)/(.65-.6))*(input-.6) + 100;
else if(input >=.55)
      return ((110-120)/(.6-.55))*(input-.55) + 110;
else if(input >=.51)
      return ((120-130)/(.55-.51))*(input-.51) + 120;
else if(input >=.46)
      return ((130-140)/(.51-.46))*(input-.46) + 130;
else if(input >=.45)
      return ((140-150)/(.46-.45))*(input-.45) + 140;
else
      return 155;
}
```

APPENDIX D: PWM.C

```
/*
Beagleboard PWM Driver Program
-PWM.c                  Startup/Initialization
-PWM_Funcs.c      PWM Generation Routine

=======PWM.c=======
Copyleft Bradley Bergerhouse
             Bradley University
             March 22, 2012
             bbergerhouse@mail.bradley.edu
*/
#include <sys/shm.h>
#include <sys/mman.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include"PWM_Funcs.c"

/*    Pin 1 1.8V
      Pin 2 5V
      Pin 3 139         Y
      Pin 4 144         N
      Pin 5 138         Y
      Pin 6 146         N
      Pin 7 137         Y
      Pin 8 143         N
      Pin 9 136         Y
      Pin 10      145         N
      Pin 11      135         Y
      Pin   12    158         Y
      Pin 13      134         Y
      Pin 14      162         Y
      Pin 15      133         Y
      Pin 16      161         Y
      Pin 17      132         Y
      Pin 18      159         Y
      Pin 19      131         Y
      Pin 20      156         Y
      Pin 21      130         Y
      Pin 22      157         Y
      Pin 23      183         N
      Pin 24      168         N
      Pin 25      REGEN
      Pin   26    nRESET
      Pin 27      GND
      Pin   28    GND

      the GPIO pins consist wholely of pins
```

```
      on the 5th and 6th gpio chips.  These
      chips service CPU pins 128 to 191.

      we will use Chip 5 which services pins
      128 to 159.  PWM channels will be assigned
      accordingly.
*/
enum CHANNEL
{
      PWM_T,
      PWM_R,
      PWM_A,
      PWM_E,
      PWM_G
};

/*
Some useful #defines for information
used to generate the signals.

NUMCHAN           - Number of PWM channels
DELAY       - Delay between a data change
                  and pin update under Angstrom
PWMPERIOD   - Total time (ns) in a pulse period
ZEROPCT           - Time logic high for 0% PWM
PHUNDRED    - Time logic high for +100% PWM
NHUNDRED    - Time logic high for -100% PWM
*/
#define NUMCHAN         5
#define     DELAY       250000
#define PWMPERIOD       20000000
#define ZEROPCT   2000000 - DELAY
#define PHUNDRED  2500000 - DELAY
#define NHUNDRED  1500000 - DELAY
int main()
{
    int i=0,j,k,l, segID, md;
      char str[125];
      char* mem;

      /*Channels in order
        0-T   1-R   2-A   3-E   4-G*/
      /*create shared memory segment for communication*/
      segID = shmget(IPC_PRIVATE, NUMCHAN + 1, S_IRUSR | S_IWUSR);
      mem = (char *) shmat(segID, NULL, 0);

      time_info PWM[NUMCHAN] = {{PWM_T, ZEROPCT, PWMPERIOD - ZEROPCT, 144,
mem},
                                          {PWM_R, ZEROPCT, PWMPERIOD -
ZEROPCT, 138, mem},
                                          {PWM_A, ZEROPCT, PWMPERIOD -
ZEROPCT, 146, mem},
                                          {PWM_E, NHUNDRED, PWMPERIOD -
NHUNDRED, 137, mem},
                                          {PWM_G, ZEROPCT, PWMPERIOD -
ZEROPCT, 143, mem}};
```

```
	pthread_t PWM_Thread[NUMCHAN];

	/*open gpio ports for writing
	  do this through the kernel driver because
	  this is not a timing specific function*/

	j = open("/sys/class/gpio/export", O_WRONLY);
	for(i = 0; i < NUMCHAN; ++i) {
		sprintf(str, "%d", PWM[i].GPIO);
		puts(str);
		write(j, str, 3);
		sprintf(str, "/sys/class/gpio/gpio%d/direction", PWM[i].GPIO);
		k = open(str, O_WRONLY);
		write(k, "out", sizeof("out"));
		close(k);
	}
	close(j);

	puts("Exported channels.");
	/*start a thread for each PWM channel*/
	/*in the future a pointer to the specific
	  pin's address should be sent as part of the PWM struct*/
	for(i = 0; i < NUMCHAN; ++i)
		pthread_create(&PWM_Thread[i], NULL, pwm, (void*)&PWM[i]);

	puts("Started Threads.  Mapping Memory");

	md = open("/dev/mem", O_RDWR | O_SYNC);
	volatile ulong *gpio;
	gpio = (ulong*) mmap(NULL, 0x10000, PROT_READ | PROT_WRITE, MAP_SHARED,
md, 0x49050000);
	if(gpio == MAP_FAILED)
		puts("Map Failure.");

	close(md);
	puts("Memory Mapped");
	while(1)
    {
		sleep(1);
		/*gpio 5 pwm values
		0000 0000 0001 1111 1010 1000 1001 0110 - 2074774
		0000 0000 0001 1111 1011 0100 0100 1110 - 2077774
		0000 0000 0001 1111 1010 1100 0111 1110 - 2075774
		changing bits
		0000 0000 0000 0000 0001 1100 1111 1000*/
		printf("GPIO Chip 5 : %o\n", gpio[0x6038/4]);
    }

	shmdt(mem);
	shmctl(segID, IPC_RMID, NULL);
}
```

APPENDIX E: PWM_FUNCS.C

```
/*
Beagleboard PWM Driver Program
-PWM.c                  Startup/Initialization
-PWM_Funcs.c      PWM Generation Routine

=======PWM_Funcs.c=======
Copyleft Bradley Bergerhouse
             Bradley University
             March 22, 2012
             bbergerhouse@mail.bradley.edu*/
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#define NUMCHAN         5

/*gpio[0x6038/4] == gpio chip 5*/

typedef struct
{
      char channel;
    long int time_h;
    long int time_l;
      int GPIO;
      char* mem;
      ulong* gpbf;
}time_info;

/*
Rudimentary lock/unlock functions for the shared memory segment

By reading the memory at m[NUMCHAN] it is possible to determine
which channel has "locked" the memory.  This lock system relies
on good coding practices of locking/unlocking suring modifications

m[c] whether or not the channel {c = 0 to NUMCHAN - 1} has locked
a the segment.  This is currently not used in any fashion but may
be useful in the future.
*/
void lock(int c, char * m) {
      m[NUMCHAN] = c + 1;
      m[c] = 1;
}

void ulock(int c, char * m) {
      m[c] = 0;
      m[NUMCHAN] = 0;
}
```

```
/*
PWM generation function using Direct Memory Access for improved speed


*/
void *pwm(void* pwm_times)
{
    int fs, cp_pin;
    int i=0;
      ulong * buf;
      ulong mask = 1;
    char ch;
      char str[125];
    struct timespec pwm_delay;

      puts("");
      pwm_delay.tv_sec=0;
    time_info *pwm_ptr = (time_info*)pwm_times;
      printf("Entering Thread : %d:%d\n", pwm_ptr->channel, pwm_ptr->GPIO);
      cp_pin = pwm_ptr->GPIO % 32;
      mask = mask << cp_pin;
      buf = pwm_ptr->gpbf;

      sprintf(str, "/sys/class/gpio/gpio%d/value", pwm_ptr->GPIO);
    fs = open(str, O_WRONLY);
      pwm_delay.tv_sec=0;
      printf("Starting PWM on Pin : %d\n", pwm_ptr->GPIO);

    while(1)
    {
            //while(pwm_ptr->mem[5] != 0);
            //lock(pwm_ptr->channel, pwm_ptr->mem);
            buf[0x6090/4] = (buf[0x6090/4] & ~mask);
        write(fs, "0", 1);//change to a direct memory
            //ulock(pwm_ptr->channel, pwm_ptr->mem);
        pwm_delay.tv_nsec=pwm_ptr->time_l;
        nanosleep(&pwm_delay,NULL);

            //while(pwm_ptr->mem[5] != 0);
            //lock(pwm_ptr->channel, pwm_ptr->mem);
            buf[0x6094/4] = (buf[0x6094/4] | mask);
            write(fs, "1", 1);//change to a direct memory
            //ulock(pwm_ptr->channel, pwm_ptr->mem);
        pwm_delay.tv_nsec=pwm_ptr->time_h;
        nanosleep(&pwm_delay,NULL);
    }
    close(fs);
}
```