Final Documentation

Team IC-U

IDD

Intellegent Detection and Deterrence

EE Senior Design
5/9/2011

N. Bosler, T. Florencki, O. Omusi, M. Wohlwend

Table of Contents

1. Introduction

Security systems use many different inputs to alert the owners, but can do very little as far as actively deterring intruders and determining their identities. Also, many systems use low quality video cameras to capture movement but cannot capture descriptive features of the intruder. This project seeks to create a system that can both act as a physical deterrent and capture hi-resolution images of the intruder.

The system requirements have changed throughout the project proposal, high level design, and low level design. The main change dealt with the unit that was used to do the video processing. It became evident that the embedded ARM was not the right choice for this project. The reasons for this will be discussed in the computer subsystem. Instead, it was decided that a motherboard and microprocessor would take the place of the ARM. The motherboard is used to process the video and the microcontroller is used to update the position of the servos, fire the gun and camera, and account for the various user inputs.

The other system requirements are:

- Accurate motion sensing algorithm in a somewhat controlled environment. This needs to output a definite 'target' to the overall system processor.
- Ability to view VGA video output of 'seeker' camera in order to monitor motion sensing algorithm in real time. The VGA output will allow for an external monitor to be connected to the system so that the user may see the input from the 'seeker' camera in addition to seeing the algorithm's 'target selection'.
- Motion of gun/hi-res camera tied to location of detected motion from incoming video feed. The system takes the 'target' as defined by the motion sensing algorithm and aligns the gun to that 'target'.
- Trigger control of the gun by the system when motion is detected
- Control of hi-res camera image capture. The images being output from the hi-res camera need to be captured and saved using either storage on the overall system or its own dedicated storage unit.
- Removable digital storage of hi-res camera images. Images saved by the camera system need to be permanently stored on removable storage to be easily accessible to the user.
- Overall user control of the system to select mode of operation (i.e. to simply detect or to detect and deter)
- Ability to toggle the use of high resolution image capture and paintball gun fire. This functionality allows the user to set their desired security parameters and decide at any time which of the two features to enable.
- Ability to hit a moving target at a distance of up to 40 feet

Most of the system requirements were met. Trouble arose when trying to find an appropriate high resolution camera. The only possible cameras that were found that could be electrically triggered were of professional grade quality. These solutions were too expensive to implement. Luckily, such a high quality camera was obtained from the graphics department in the College of Engineering. Due to the heavy weight of this camera, we were not able to mount it on top of the gun as originally planned. As a result, the high resolution camera was placed next to the mount on a flat surface. As a proof of concept, a high resolution point and shoot digital camera was still mounted onto the gun turret.

The system creates a solution to the detection and deterrence problem by the integration of a 'seeker' camera and a combination gun/high-resolution camera attached to two servo motors for control. The incoming video feed from the seeker camera (which remains stationary) will be analyzed for motion within the frame by a motherboard computer. The location of motion from the seeker cam will then translate to movement of the gun and high-resolution camera through the use of two servo motors. The system can then either fire the gun and/or capture images with the high-resolution camera. The computer will interface via the RS-232 standard with a microprocessor, which will then communicate with the gun, camera, servos, and user inputs to the system. Switches allow the user to turn the system on, control whether the servos get updated, and enable gun firing. Once the microprocessor obtains the position and firing information from the computer, it uses pulse-width modulation to control both servos and digital input and output pins to read the user commands and fire the gun and camera.

The system consists of a case, which is designed to be placed inside the user's home. The case contains the power supply, the computer, and the microprocessor board. An XLR cable, which is used to send power to the mount, and a standard serial cable, which is used to send the signals, is designed to be routed outside to the mount. These two cables connect into an auxiliary board, which distributes the signals to the servos, gun and camera.

The design meets expectations. We were able to successfully hit a moving target (at a relatively constant speed) in a controlled environment, as well as take high resolution images of the intruder. The processor on the motherboard was not able to fully handle the video processing so the motion tracking algorithm had to be stripped down a little bit to account for this. In doing so, we lost the ability to prioritize between two targets if both are in the 'seeker' camera frame. Also, the new algorithm became more sensitive, so threshold values had to be increased. The limitations imposed by the processor also make it hard to account for targets moving with a non-constant velocity. A faster processor will allow for a more robust motion tracking algorithm that takes all of these problems into account.

Overall, the project represents a successful first version. Improvements can be made, of course, and will be discussed in the future enhancements section.

2. Detailed Project Description

2.1 System Theory of Operation

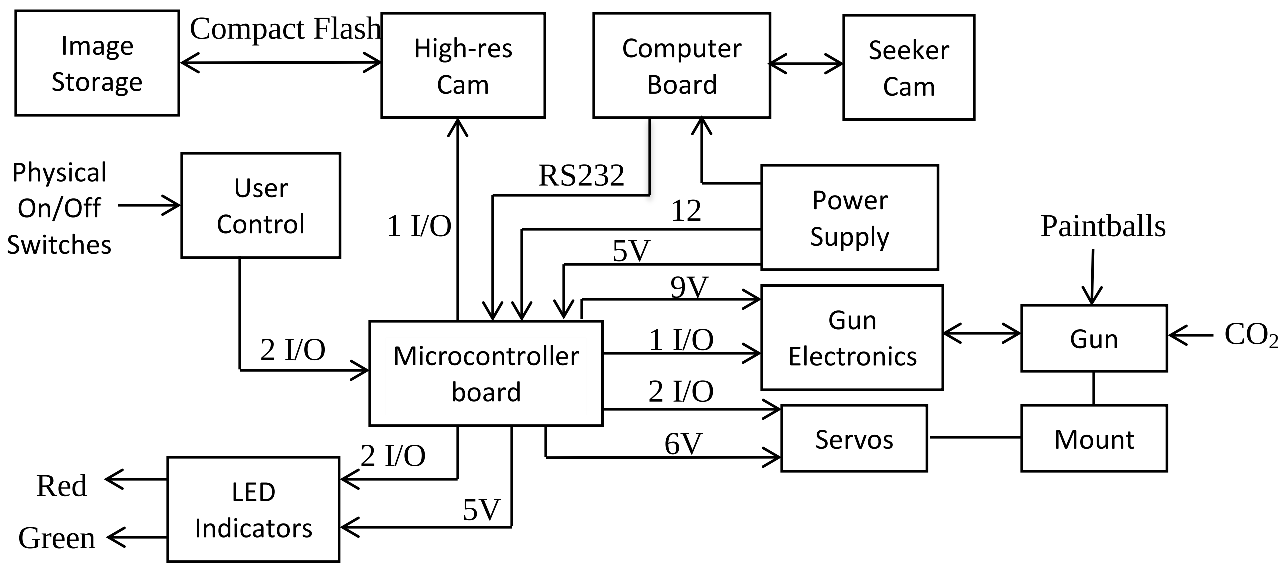
All the processing of the video feed from the web camera is done on the computer board. The running program on the computer is written in C and uses the OpenCV library. It grabs each frame from the video feed and compares it to the previous frame it received. After applying some image filters to the compared images, the program gets contours that are the edges of the moving object and bounds them with rectangles. A rectangle size threshold is added so that very small and irrelevant movements in the environment are ignored. By finding the average of all the centers of all the rectangles in the frame, the program finds the centroid of all the moving edges which should be a good approximation of the center of the moving object. The program also takes into account the fact that the video processing adds a delay in the system that prevents real time processing. To compensate for this, the program adds a lead to the target location by looking at the target's last four locations and linearly approximating where the target will be next. The program also uses the size of the motion-bounding rectangle to determine whether or not to fire at the target using a predetermined threshold.

Having calculated what pixels on the screen best represent the target's location, the program calculates the angular location of the target at 20 feet (practical operation distance). The angular locations in two planes and the gun-fire command are then transmitted over the computer serial port (asynchronous serial RS232 communication) one after another to the microcontroller board at 9.6 kbps.

The microcontroller receives the serial data and immediately stores it in an array when the receive interrupt flags go up. The main program running on the microcontroller then reads through the array and updates the location values and fire command. The location values are translated into the appropriate pulse widths for servo control and are used to load count registers on timer1 and timer3. Timer0 is used to generate a 20ms time base for servo control. Every 20ms, timer0 sets two pins high and timers 1 and 3 which both have shorter periods (based on the loaded values) each set one of these pins low when their interrupt overflow flags go off. This generates a PWM signal on both pins which is sent to the servos and moves them to the designated target location. The gun is triggered by a pulse generated using the time base from timer0. This pulse has a period of 0.2 seconds and a 50% duty cycle. The camera is also triggered using a pulse generated from timer0 with a period of 0.8 seconds and 50% duty cycle.

The system includes three user-controlled switches that change the operation of the system. The first button is used to turn on the entire system (the computer, microcontroller, gun and servos all share the same power source). The second switch puts the system in "detection mode". When it is turned on, the gun/camera combination begins tracking moving objects when the motion program is running on the computer. This also enables the camera to take pictures but the gun is disabled. Green LEDs on the system case and on the turret light up to indicate "detection mode". The third switch is used to put the system in "deterrence mode". This is the same as detection mode except that here the gun is enabled to fire at objects that are above the chosen size threshold for targets. Red LEDs on the system case and on the turret light up to indicate this state.

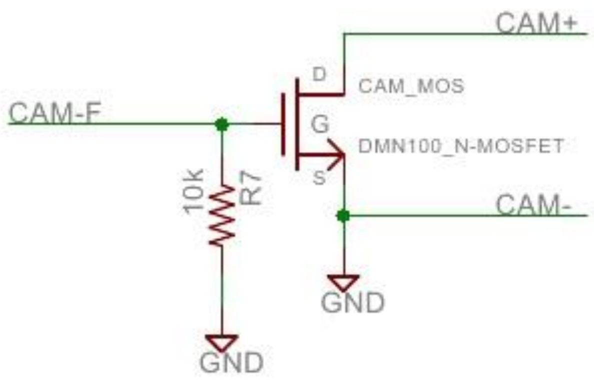
2.2 System Block Diagram



2.3 Detailed Subsystem Operation

High-Resolution Camera

The high-resolution camera has a resolution of 12 mega pixels and the optical zoom is dependent on the lens that is used. Currently, the zoom is 3x. These settings provide a detailed image of the moving object. The camera can be externally triggered by I/O pins on the microcontroller. Compact flash is the storage mechanism. The figure below shows the circuit that controls the firing of the camera. CAM-F is connected to an I/O pin and CAM+ and CAM- are connected to two leads of a standard 3.5mm connector. The other end of this connector is fed into the camera. Shorting of these two leads causes the camera to fire. When CAM-F is set



high, CAM+ and CAM- short, and the camera fires. The means by which this is connected in the actual system is discussed in the board layout and mount subsections. The code used to fire will be discussed in the section on microcontroller code.

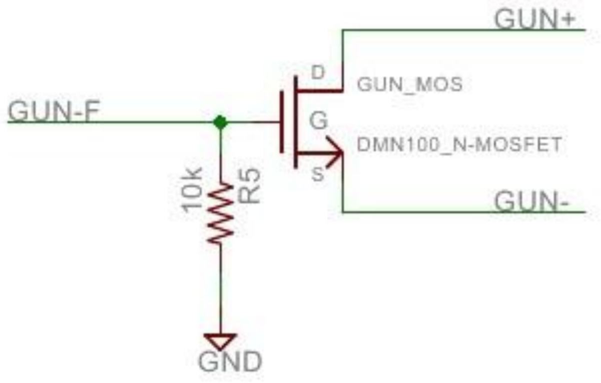
This subsystem was tested by writing code that simultaneously set CAM-F high and turned on an LED on the microcontroller board. To make sure this subsystem worked, it was verified that every time the LED turned on, the camera also was able to take a picture.

'Seeker' Camera

The seeker camera is a low resolution (VGA) webcam, reducing the amount of processing power needed to implement the motion-sensing algorithm. It feeds its information into the computer. In addition, it is small so that it is not in the way of other components. The 'seeker' camera supports UVC drivers which allows for communication with the Linux operating system. The mounting of this camera will be discussed in the mount subsection.

Paintball Gun

The paintball gun has an electrical firing mechanism so that it can be easily triggered by setting a digital output pin. It will also need to be supplied with the necessary CO2 and paintballs. The firing circuit for the gun is shown below. GUN-F is connected to a digital I/O. GUN+ and GUN- are connected to the switch of the electric trigger. When GUN-F is set high, GUN+ and GUN- will short (simulating the pulling of the trigger), and the gun will fire. The means by which this is connected in the actual system is discussed in the board layout and mount subsections. The code used to fire will be discussed in the section on microcontroller code.



This subsystem was tested by writing code that simultaneously set GUN-F high and turned on an LED on the microcontroller board. To make sure this subsystem worked, it was verified that every time the LED turned on, the gun also fired.

Microcontroller Board

The purpose of this board is multi-faceted. It needs to provide power for the servo motors, paintball gun, and all external LED indicators. The board accepts input from the user in the form of switches to enable/disable certain functions of the whole system and also accepts input from the computer containing instructions for positioning of the servo motors and whether or not to fire the gun and/or camera. In addition, it must be able to interpret the instructions from the computer and use this information to provide the necessary signals to the servo motors, paintball gun, camera, and LED indicators.

This subsystem interfaces with almost every other aspect of the entire system. It accepts data from the computer, provides the signals for the LED indicators on the Gun Mount, accepts user inputs from the case, houses the microcontroller and its associated circuitry, powers and sends signals to fire to the paintball gun, sends signals to take pictures to the camera, and powers and provides pulse-width modulation for the servo motors.

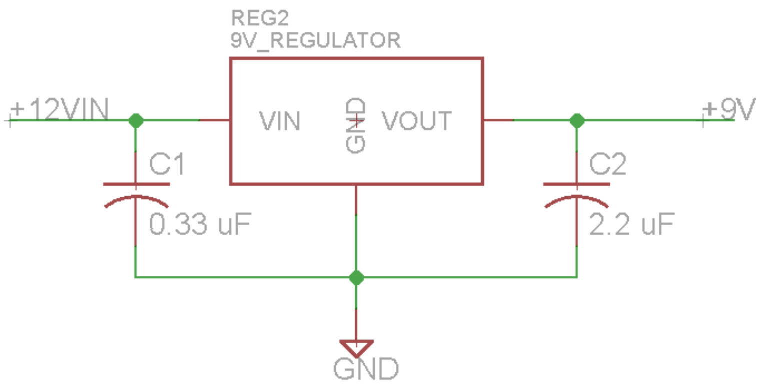
Microcontroller

We chose the PIC18F4620 processor as the brains for our microcontroller board. There were a few different criteria that we were looking for in a microcontroller that led us to pick this particular one. The power requirements were simple. We knew that our board was going be supplied with a stable +5V from the computer's power supply unit, so it would be most convenient to have a microcontroller that could also operate at +5V. One of the most important criteria for picking the microcontroller was that it needed to have at least one USART for our serial communication. As far as input/output pins, we needed a baseline of 8 I/O pins, not including those needed for programming the microcontroller and those associated with the USART. Lastly, we decided that it would be best to pick a microcontroller within the same family as that used in the kit board so that we could use the same programming module to program our microcontroller. With this particular criterion in mind, most of the processors in this family met our requirements, and so we decided to pick one of the 'lower end' models that still did what we wanted. Price was not an issue in our decision because we were able to request limited free samples through the manufacturer.

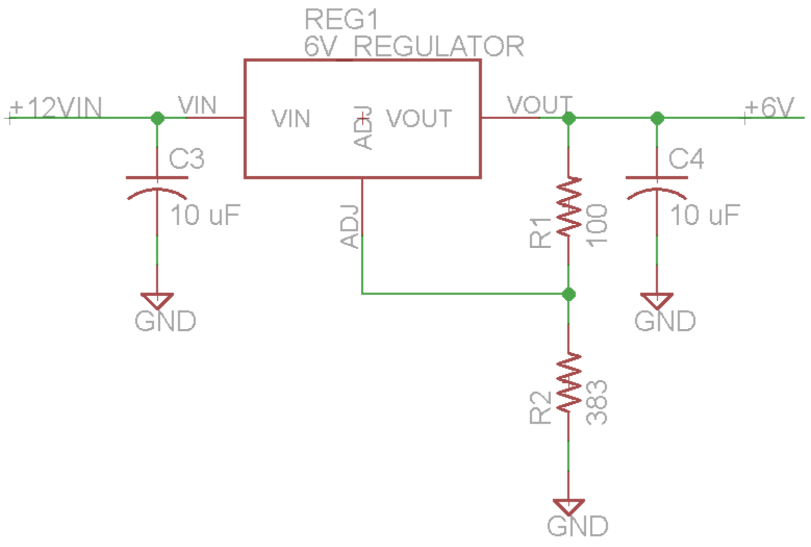
Providing Power

The power input to this subsystem was in the form of three Molex 8981 connectors coming from the Power Supply Unit (PSU) of the computer. Three connectors were used instead of one primarily to be able to reduce the amount of cable clutter from the PSU as it has three molex connectors. This was a convenient way to bring power to the system as the PSU provides +5V, +12V, and GND through this connector. The +5V was used to operate the microcontroller, while the +12V was regulated to the necessary +9V to power the gun and +6V to power the servo motors.

The +9V was obtained using a fixed-voltage +9V regulator capable of sourcing 1A of current. In initial testing of the gun, a maximum of 0.5A was drawn from the source, which is why we decided to pick a regulator capable of 1A. The circuitry associated with this regulator is simply called for by the data sheet to keep the voltage output stable.



The +6V was obtained using a variable-voltage regulator capable of sourcing 7A of current. The stall-current of each servo motor was rated to be 3A. We decided to err on the side of safety and pick a part capable of providing the current should something go wrong with the gun mount. Due to the high current capacity of this part, it is a through-hole component. Since this regulator could provide a variable voltage output, it required more circuitry than the +9V regulator. Resistors R1 and R2 were used to set the output voltage of the regulator. As with the other regulator, the data sheet called for input and output capacitors to help keep the output voltage stable. However, the output required a tantalum capacitor.



Power was transmitted from our regulators to the appropriate subsystem through use of an XLR cable and the auxiliary board (see section). We decided to use XLR, a connector commonly used for audio purposes, because it has three connections (for +6V, +9V, and GND) and locks in place to provide a sturdy connection between board and cable.

This functionality of the microcontroller board was tested by connecting the input power to a lab variable power supply. We set the supply current low so that our system would be protected in case the regulator(s) would fail. Once the open-circuit voltage (no load) was verified and no current was being drawn (which would suggest a short), the variable power supply was disconnected and the system tested with the system power supply unit through the molex connectors.

Sending Signals

To route signals from the microcontroller to the appropriate subsystem, we wired them to a D-sub 9 connector. This was an appropriate connector to use because it contains the exact number of pins as signals that we want to send to the auxiliary board and screws in place to ensure a solid connection between board and cable.

In addition to signals being sent via output ports on the board, we decided to add a bank of board-mounted LEDs, which would give us flexibility in using our board for prototyping. Setting different LEDs could allow us to more-easily debug microcontroller code. The LED bank was connected to Port D from the microcontroller and could be disabled by removing the jumper to +5V. The resistor was chosen to limit the current through the LED to 10mA. We connected the cathode of the LEDs to the output pin on the microcontroller so that when we set the pin low, the LED will be on. The benefit of this is that the microcontroller does not need to supply the current required to power the LED.

As this microcontroller had more than our required number of input/output ports, we decided to allow ourselves connectivity to a particular subsystem output through two different output ports on the microcontroller. For example, the output signal to fire the gun could come from either pin A1 or pin D1 from the microcontroller. To do this, we connected both ports A and D to their own jumper bank and the output to the D-sub 9 connector to the other ends of both the port A and D jumpers. Thus, to operate the system from port A, all of the jumpers should be on the port A jumper bank with none on port D. This is operated likewise for operating on port D. Only one of these pins should be connected to the output at any time. The benefit of this is redundancy. If something were to accidentally happen to one of the pins on the microcontroller (blown out), we could easily change the jumper and the port in code then resume operation. As this is a prototype board, we felt this would be useful, but would not be needed in a production board.

To protect the microcontroller, we added a buffer between the pin from the microcontroller and the D-sub 9 connector for the camera fire and gun fire signals. This consisted of tying the microcontroller pin to the gate of an n-channel MOSFET connected to the two ends of the signals. This provided a simple way to help protect our microcontroller from any unexpected surges.

To test whether or not signals could be sent from our microcontroller, we set random pins on Port D high and low. We then connected the jumper to power the LEDs and looked at the LED bank to see if they matched how we set the pins from Port D.

Receiving Signals

The two user-controlled switches each provide the system with enable/disable commands. In order to get relevant information about the position of the switches, the board sent a +5V, micro controller pin, and GND to each of the three-way switches. We used a three-way switch so that the input to the micro controller would always be either +5V or GND and would never be left floating.

As with the output signals, the input signals were tied to two different ports on the microcontroller, controlled by a jumper.

To test the digital reception of signals, we programmed the microcontroller to change the output of the LED bank based on what the input to the microcontroller. We connected the user-controlled switches to the input pins on the microcontroller, which would directly connect either +5V or GND to the input pin. We then made sure that the LED bank reflected changes made on the user-controlled switches.

RS232 Communication

As mentioned in the interface section, the RS232 output from the computer uses a voltage swing of +12V and -12V. These voltages are too high for our microcontroller and would damage it if applied directly to the pins. Because of this, we used a MAX232 chip to essentially bring those voltages to +5V and GND so that our microcontroller could read the signals.

The only associated circuitry with this part involved adding datasheet-mandated capacitors.

To test the RS232 communication, we connected the output of the com port on the computer to the input on the microcontroller board. We had the computer set to send new serial information every two seconds. The microcontroller was set to read the data from the RS232 and output it to the LED bank. We could then look at the LEDs to make sure that the data matched up with what we were sending.

Auxiliary Board

The purpose of this board was to allow for simple cable connections between the case and the subsystems outside (gun, camera, servos). It accepts the XLR and D-sub 9 connectors and splits the signals up so that each subsystem can use them. This allowed us to run only two cables from the main board outside instead of individual cables for each signal, a much cleaner solution.

Since the auxiliary board only ‘split’ incoming signals from the D-sub 9 connector and XLR to their appropriate destinations, testing simply involved measuring the resistance from input pin to output pin to ensure that connections were routed correctly.

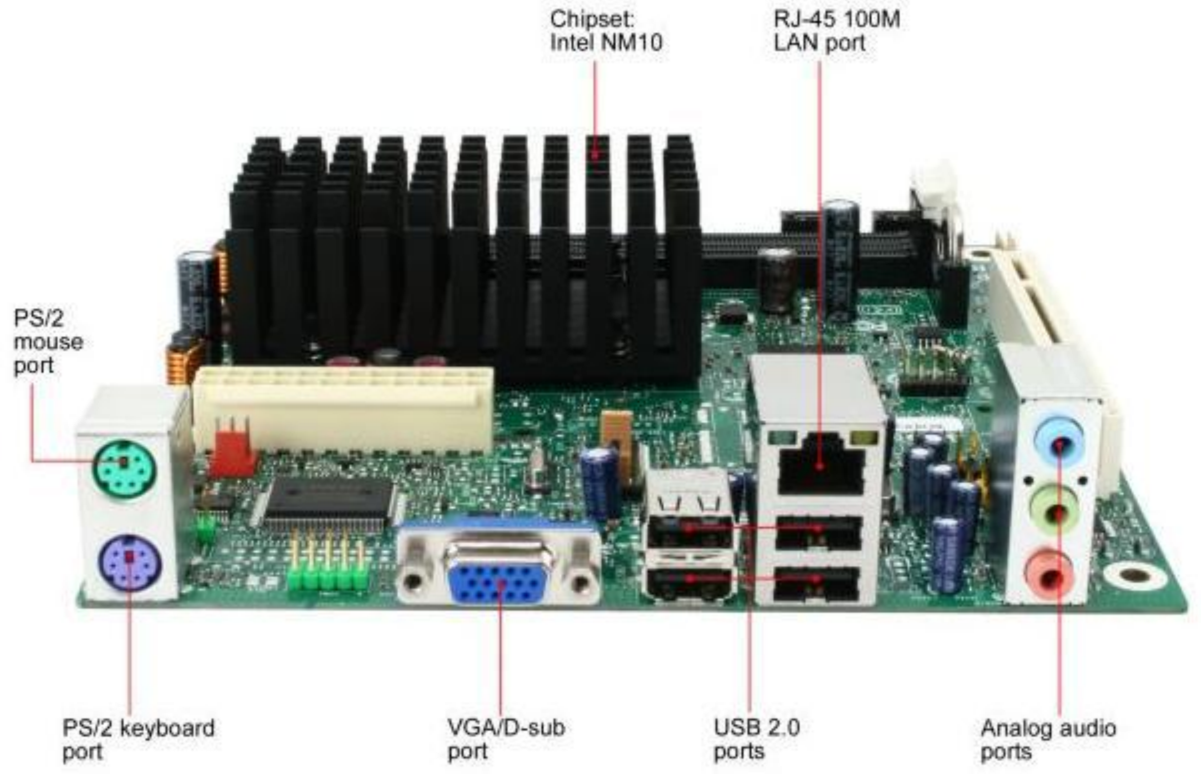
Miscellaneous

The board has connectivity for a programmer, which is necessary for allowing us to transfer our code from the computer to our actual micro controller. This is also tied to a reset switch, which is useful when testing the micro controller.

Decoupling capacitors were added near the power input to the micro controller to further protect it from any voltage variation.

Computer

In all previous documentation, it was stated that an embedded ARM would be used to do the video processing and to send control signals to gun, camera, and servos. However, it became evident that the ARM would not be able to satisfy the system requirements. The Ubuntu operating system, as well as the openCV library that is used to do the video processing was not able to be successfully installed on the ARM. In addition to this, the coding necessary to implement interrupts was beyond the scope of this project. Interrupts are necessary to control both servos. Lastly, the ARM has a 200 MHz processor, and it became evident that this would not be fast enough to process the video input. Instead, it was decided that an Intel Motherboard/CPU combination would be used to do the video processing. When picking the replacement, a small form factor, a faster processor, and serial capabilities were the criteria that was used. The Intel Atom D410 Intel NM10 Mini ITX Motherboard/CPU Combo were chosen based on these criteria. A picture is shown below.

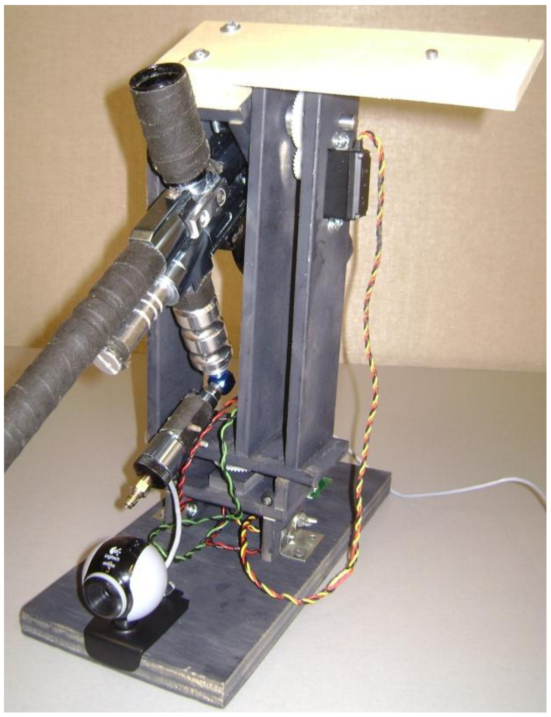


The Intel Motherboard/CPU combo, with a 1.6 GHz processor, interprets the video input from the ‘seeker’ cam. Serial communication between the computer and the microcontroller board sets the position of the servos and fires the gun and camera. This will be discussed later in further detail in the system interface section. The ‘seeker’ camera is connected to the computer via a USB interface. The mounting of this computer will be discussed in the case subsection. The motion tracking algorithm is listed in the video code subsection. A link to the data sheet is listed in the Appendix.

Gun Mount

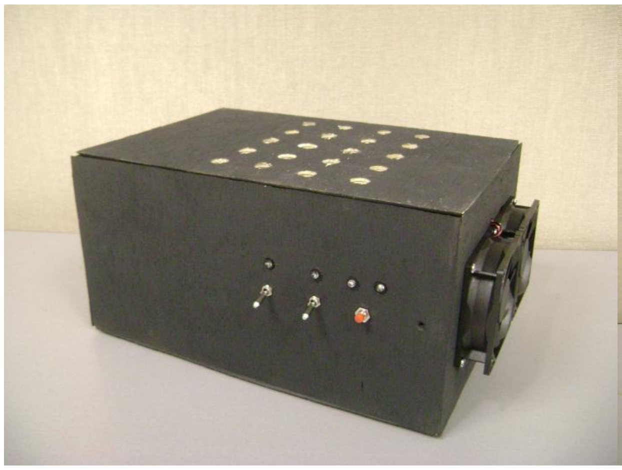
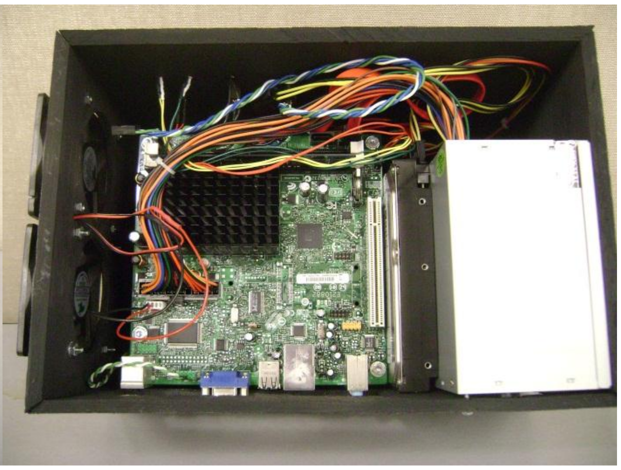
The purpose of the gun mount was to provide a mounting scheme for the paintball gun, high-resolution camera, and seeker camera that is able to provide pan and tilt motion for the gun and high-resolution camera. The seeker camera should remain stationary on the mounting system. The mount should also provide mounting spots for the servo motors. The movement of the servo motors should also be fast and strong enough to turn the mount at a speed appropriate for following a moving target. This subsystem needs to be sturdy enough so that the firing of the paintball gun will not cause any damage to the system. The mount should provide a mounting spot for the auxiliary board. Finally, the mount should also contain indicator LEDs to warn somebody approaching it as to the status of the system.

This subsystem interfaces with many of the mechanical portions of the project. It interfaces with the microcontroller board by housing the auxiliary board and displaying the LED indicators, which reflect the status of the user-controlled switches. The paintball gun, high-resolution camera, seeker camera, and servo motors are mounted on this subsystem.



This mounting scheme was realized by using two rotating axes- the phi direction (pan) at the bottom of the system and the theta direction (tilt) near the top. The theta direction movement involved a long axis to which the gun cradle was attached. This axis went through holes in the mount and allowed to rotate freely. There was also a gear attached to the axis which gave an overall gear ratio of axis rotation to servo motor of 2:1. This was done to increase the torque output of the servo motors so that there would be no issues in being able to move the weight of the mount. Doing this decreased the speed response but this was acceptable because of the high speed of the servos. The phi direction movement was realized similarly. The gun cradle and theta direction movement hardware had an axle connected to it facing down. This then was secured in the base of the mounting scheme and its rotation tied to the other servo motor through the same gearing ratio as the theta direction. The high-resolution camera was mounted onto an extension of the gun cradle so that it would experience the same theta and phi motion as the gun. The seeker camera was attached to the base of the mount so that it would not experience any movement.

Case

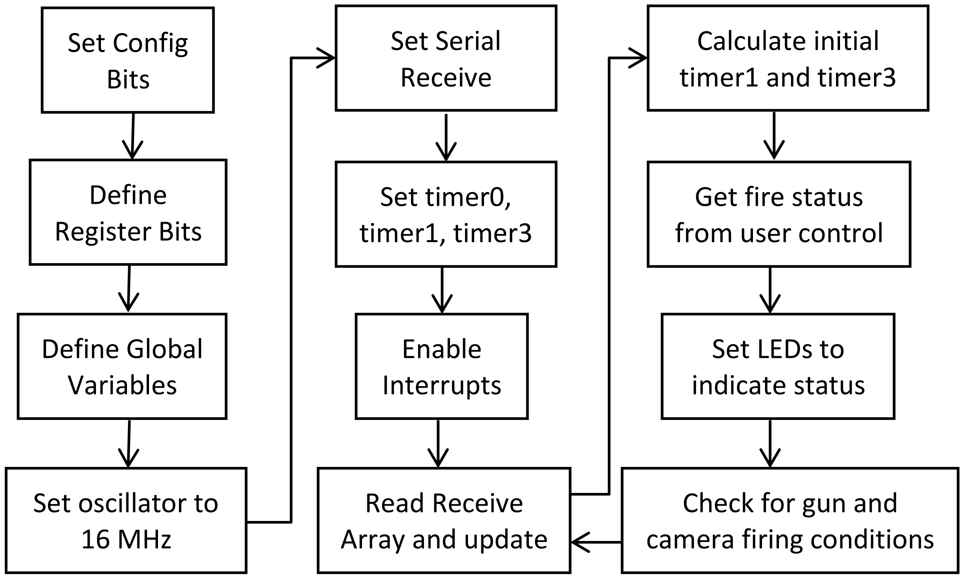



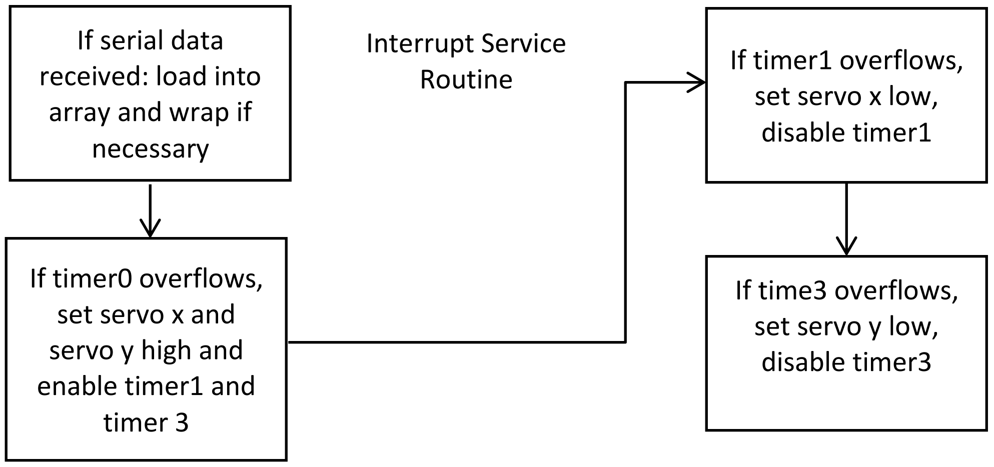
The case is constructed out of wood and houses the computer board, microcontroller board, hard drive, and power supply. The case is designed to be as small as possible for easy placement in a user's home. The front of the case has the user interface panel that includes the switches and button that determine system operation. This front panel is designed for ease of use and to avoid user operation error. The red button turns on the power to the system and is tied to the power on LED and HDD access LED. The track on switch enables the servos and camera and is tied to a green LED both on the case and on the mount. The fire enable switch is connected to a red LED on the case and the mount. On the rear of the case, there is the input/output panel that connects to the cables that send the necessary signals to the mount. Included on this panel are the development/debug ports that allow the user to connect a monitor, mouse, and keyboard to troubleshoot the system. The 250W power supply provides the necessary power for the computer and the electronics at the mount. To protect components from overheating two standard computer case fans are mounted to the outside to provide cooling air flow.

To test the functionality of the case, everything was wired up and the button and switches tested first on a DMM to insure currents were safe for the LEDs. Then each button and switched was connected and checked to make sure the LEDs lit at the proper time. This included making sure the LEDs on the gun mount matched those on the case.

Microcontroller code

The microcontroller code was developed using Source Boost IDE. The main functions of this code was to take asynchronous serial RS232 information coming from the computer and use that information to control the servos, paintball gun and the high resolution camera. It was necessary that this code be entirely interrupt driven because the information coming into the microcontroller and the servos have to be updated in real-time. The flow chart included below shows how that is achieved.


Set Config Bits
Define Register Bits
Define Global Variables
Set oscillator to 16 MHz
Set Serial Receive
Set timer0, timer1, timer3
Enable Interrupts
Read Receive Array and update
Calculate initial timer1 and timer3
Get fire status from user control
Set LEDs to indicate status
Check for gun and camera firing conditions


If serial data received: load into array and wrap if necessary
If timer0 overflows, set servo x and servo y high and enable timer1 and timer 3
Interrupt Service Routine
If timer1 overflows, set servo x low, disable timer1
If time3 overflows, set servo y low, disable timer3

The serial data from the computer is stored in an array within the interrupt service routine (ISR) as soon as the interrupt flag for this goes off. The storage structure is a character array with a length of 6 because the computer sends ‘x’ followed by the value for the horizontal plane servo, ‘y’ followed by the value for the vertical plane servo and ‘z’ followed by the state of the gun. In the semaphore to this ISR in the main code, the variables for the servos and the firing status are updated with the values from the array.

The maximum clock frequency from the internal oscillator of the microcontroller was 8MHz. To get the most speed out of our system, the 4X phase-locked loops (PLL) were activated to increase our oscillator frequency. This was done by setting the appropriate values in the OSSTUNE register. 16 MHz (4 MHz X 4) was the oscillator frequency used instead of 32MHz (8 MHz X 4) because with the latter prescales would have been required on all the timers as opposed to the former with which just one prescale is required on timer0.

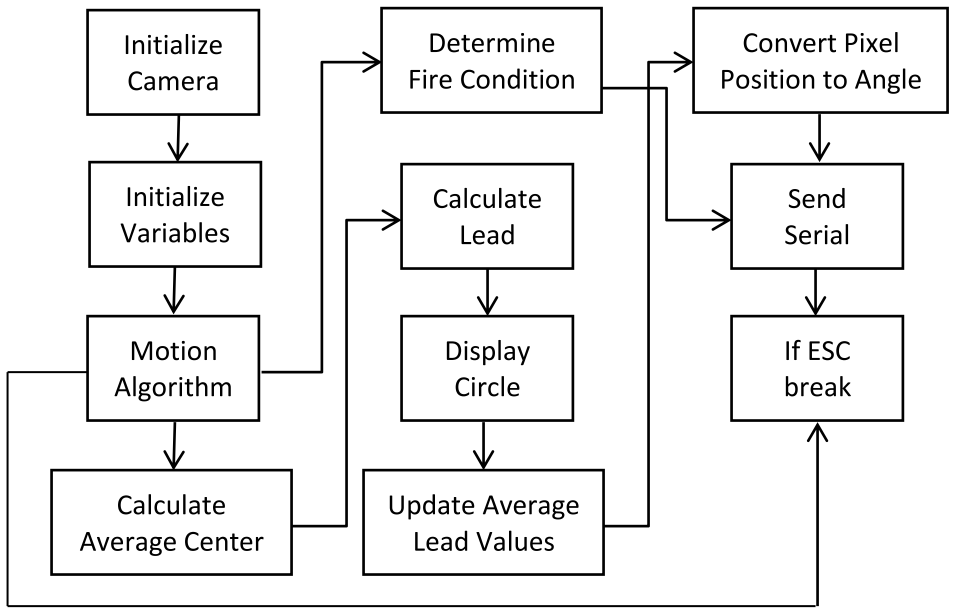
Three timers are used to create the PWM signals for the servos: timer0, timer1 and timer3. All three are 16-bit timers running off the internal clock and timer0 has a prescale of 1:8. Timer0 creates the 20ms (50Hz) period for the PWM and timers 1 and 3 create the pulse widths that correspond to the receive values. This is done by taking the receive values and using a simple calculation (see code in Appendix A) to find the count value that should be loaded into the 16-bit count registers to get a pulse width that corresponds to the position of the target. Timer0 overflow is also used to generate pulses that fire the camera and the gun. The camera is fired every 800ms (40 overflows of timer0) with a 50% duty cycle and the gun is fired every 200ms (10 overflows) with the same duty cycle.

The user switch function is encoded here also. The “detect mode” switch is connected to pin E0 and the code reads whether or not porte.0 is 1 (switch is on) or 0 (switch is off). It also uses these values to light up the LEDs on the turret and the system case. The same is true for the “deter mode” switch which is connected to pin E1.

The PWM signals were tested by initializing the values that the code would expect from the computer and connecting the USB analyzer to the output pins to see if the PWM signals were as expected. The serial communication was tested on its own by connecting the microcontroller to the computer, sending values and then having the microcontroller display the values on the bank of LEDs on the board. The final test was then to put these two functions together. Values were sent over the serial communication and the output of the microcontroller was checked to see if it matched the expected values.

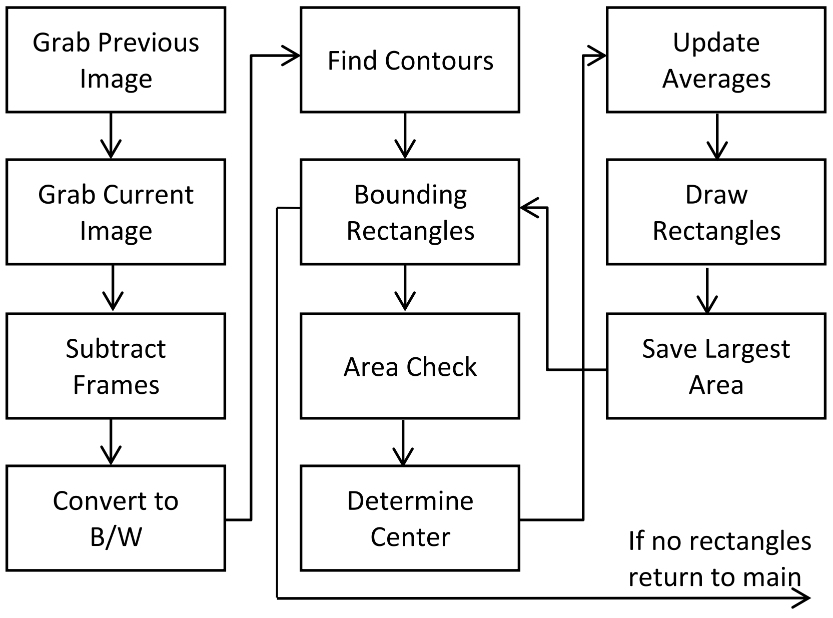
Overall System Code

The overall system code is explained in the following flow chart:



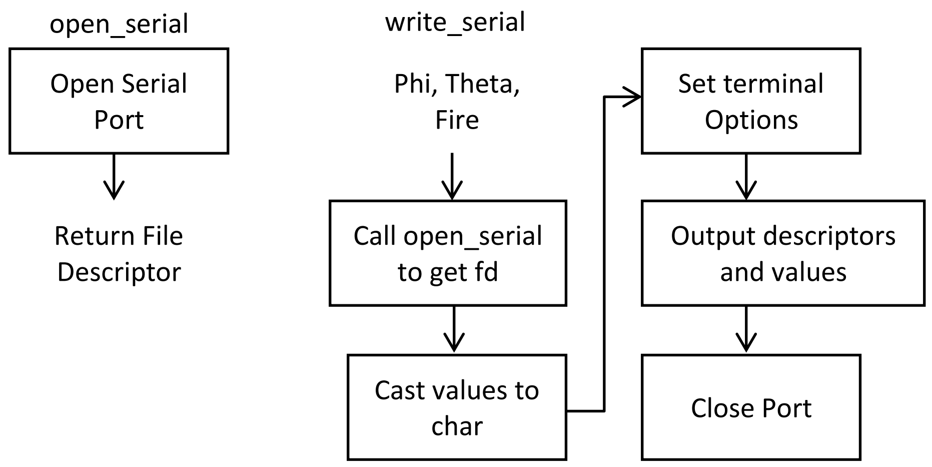
This is all coded in C using the OpenCV libraries for the image processing. This is developed to operate in a Linux environment as it uses Linux specific commands for the serial communication. From the flow chart, the code begins with mounting the camera and setting the necessary variables for the rest of the code. This then feeds an infinite loop that contains the necessary computations and algorithms. The motion algorithm will be explained later. Its function is to determine all the centers of movement and from those, further computation can be done to calculate average center. The center is calculated using the average of all the individual centers and then this is passed to the calculate lead. Before the lead is calculated the largest area of motion is compared to a set threshold to determine if the target should be shot at. If it is large enough the fire condition is sent to the write serial function. The calculate lead function takes the current average center and compares it to three previous centers and multiplies it by a factor to account for the lag of the system. This factor was determined through testing and will be further explained at the end of this section. Next the code will display a target circle to the user based off this lead. The previous values are updated for the next iteration through the code and the new center based off the lead is converted to a degree. The pixel center needs to be translated into an angle so the servos can understand it. This is done through the necessary trigonometry based off the distance between the camera and the target. Finally these converted values are passed to the send serial function and sent to the microcontroller. The code checks for the user quit button, and then repeats.

The motion algorithm is explained with the following flowchart:



The motion algorithm compares the current frame to the previous frame and determines the changes. This is done by a simple frame-by-frame comparison. This type of algorithm was chosen because of its relative simplicity and fast execution. The algorithm starts by updating the previous and current images. An example of a starting image is presented in figure X. To determine the areas that have changed a simple absolute difference function is used to subtract the current image from the previous image. An example of the difference image is present in figure X. This difference image is still in color and in order for it to be processed it must be in black and white. The output image after being converted is shown in figure X. This image is passed through the cvGetContours function to connect the individual pixels into continuous blobs. Then rectangles are drawn around these contours if they are above a minimum threshold and displayed to the user as shown in figure X. The averages are computed in the main code, but each iteration of the contours function updates the sums of centers to give the main code the necessary information to compute the averages. This section of code also determines the largest rectangles area and saves it for comparison to the firing threshold.

The serial functions are explained below with the following flowchart:

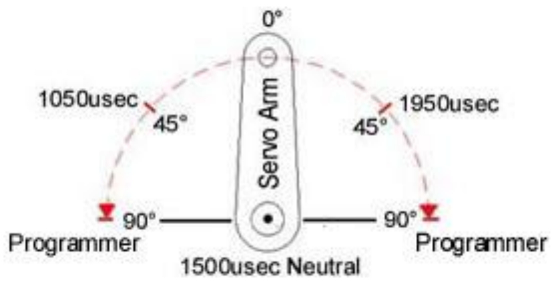


The function open_serial is very simple. It calls the open function to return the file descriptor for COM1 port. The write_serial function takes the phi, theta, and fire conditions of the system from the main code and sends them over the RS232 communication line to the microcontroller. It initially calls the open_serial function to get the file descriptor for the port. Casting the passed theta and phi values to characters is necessary to put them into the write function. Next the terminal option values are set to configure for a baud rate of 9600, set local mode, and remove the parity bit. Now the code is ready to output the data string to the microcontroller. It passes the x first as a descriptor, phi, y as a descriptor, theta, z as a descriptor, and finally the fire condition. Provided there are no errors, the function closes the port and returns to the main code.

In testing of the system, it was quickly realized that the hardware was not fast enough to support the algorithm as written. This prompted a rewrite of the code to its current form as well as the addition of the leading algorithm to account for the delay that still existed. It was originally hoped that converting the code from python to C would have a dramatic increase in speed. However it was found that while the translation did help, it did not fix the speed enough. The original algorithm used a moving average as the comparison frame, but in an attempt to increase speed this was dropped in favor of the now implemented frame-by-frame approach. Also removed from the original algorithm was functions that expanded the individual pixels to create larger areas of motion. By removing these functions the code lost the ability to prioritize between targets. Also in testing it was observed that the program has a memory leak and does not properly release memory. This problem is currently unresolved.

Servos

Digital high torque servos were purchased from www.servocity.com. Digital servos were chosen due to their faster response and generally higher torque ratings. These servos did not need a programmer to function properly. They were rated at 168 oz-in of torque and 0.18 sec/60° when operated at 6V. To ensure there was enough torque for the system, a larger gear was connected to the servo gear for a 2:1 gear ratio that doubled the torque and halved the speed and the distance moved. Our web cam had a horizontal angular range of 50° but the stock came with only a 90° range which would be insufficient after the 2:1 gearing. To make up for this, the servos purchased had 180° rotation as opposed to the stock servos. This provided sufficient torque, range of motion and speed for the complete turret system.

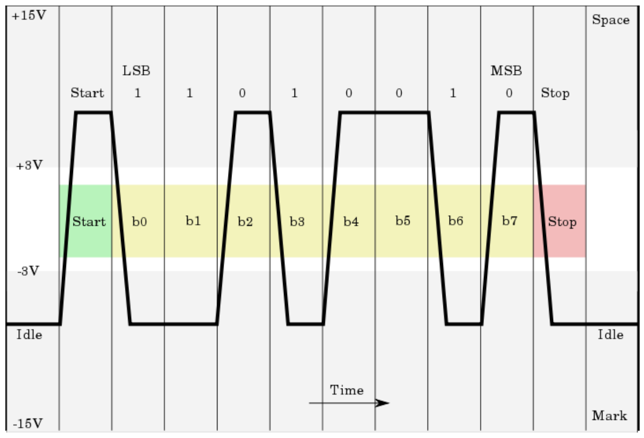


Control of the servos required PWM signals with a period of 20ms and pulse widths ranging from 0.6ms to 2.4ms. The signals were generated from the microcontroller and then transmitted to the auxiliary board via a serial cable. The servos were plugged into the auxiliary board.

The servos were powered via an XLR cable connected to the microcontroller board. The microcontroller board takes the 12V from the system power unit and regulates it to 6V which the servos run at. The current drawn by the servos varied but never peaked because the servos did not stall.

2.4 Interfaces and Sensors

The only interface used in this project was RS232. This communication protocol uses bit-by-bit sending of bits based on a preset time base. We used this form of communication to transmit data from the computer to the microcontroller which contained servo positions and whether or not to fire the gun and camera. We picked RS232 because it is a relatively simple protocol and many microcontrollers have built-in support for reading and writing to it. The computer uses ±12V to transmit the data, but the microcontroller only accepts inputs of GND and +5V. To get around this, we used the MAX232 component on the board to translate the voltage levels between the computer and microcontroller.



3. System Integration Testing

3.1 Testing Integrated Subsystems

It was very important to calibrate the position of the gun with respect to position of the web camera. To do this, the frame of the camera was positioned to be in line with a dry erase board. A ruler was used to determine the middle of the web camera, which was a point on the dry erase board. A laser pointer was mounted onto the gun so that the dot of the pointer represented the view of the paintball gun barrel. Serial commands were manually sent to the microcontroller via the command terminal of the computer so that the laser pointer was in line with the middle of the frame, as marked on the white board. The microcontroller code was then adjusted to account for the difference between the web camera middle and the mount middle determined by the servos.

The integrated set of subsystems was tested outside in a controlled environment. The frame of the web camera was positioned so that the background was a stationary wall. This provides a constant background image as well as increased safety because it prevents anything else from entering the video frame.

A piece of wood with a black square mounted to it was walked across the frame of reference. A black square was added to provide a contrast, and thus a target, for the web camera to identify. Multiple trials were run in attempt to perfect the algorithm used to lead the moving target. Threshold values were also changed to eliminate noise.

3.2 Testing Verifies Overall System Requirements

Verifying that the design requirements was fairly simple. If the black target was consistently hit during the trial, the system was deemed successful. Once this happened, another trial was conducted to verify the results. The progression of this testing is shown in a video on the group website (http://seniordesign.ee.nd.edu/2011/Design%20Teams/IC_U/IC_U_Base.html)

4. User's Manual

4.1 Installing the Product

Installation of the system is designed to be as simple and painless as possible. The case is to be mounted in the building and connected to power. If the user would like they can connect the case to a monitor and keyboard for debugging purposes. The key part of installing the system is mounting the gun mount properly. For the system to work the mount must be solidly screwed in or bolted to minimize extra movement. The cables connecting the case and mount must be connected to provide the necessary signals to the servos and gun.

4.2 Setting up the Product

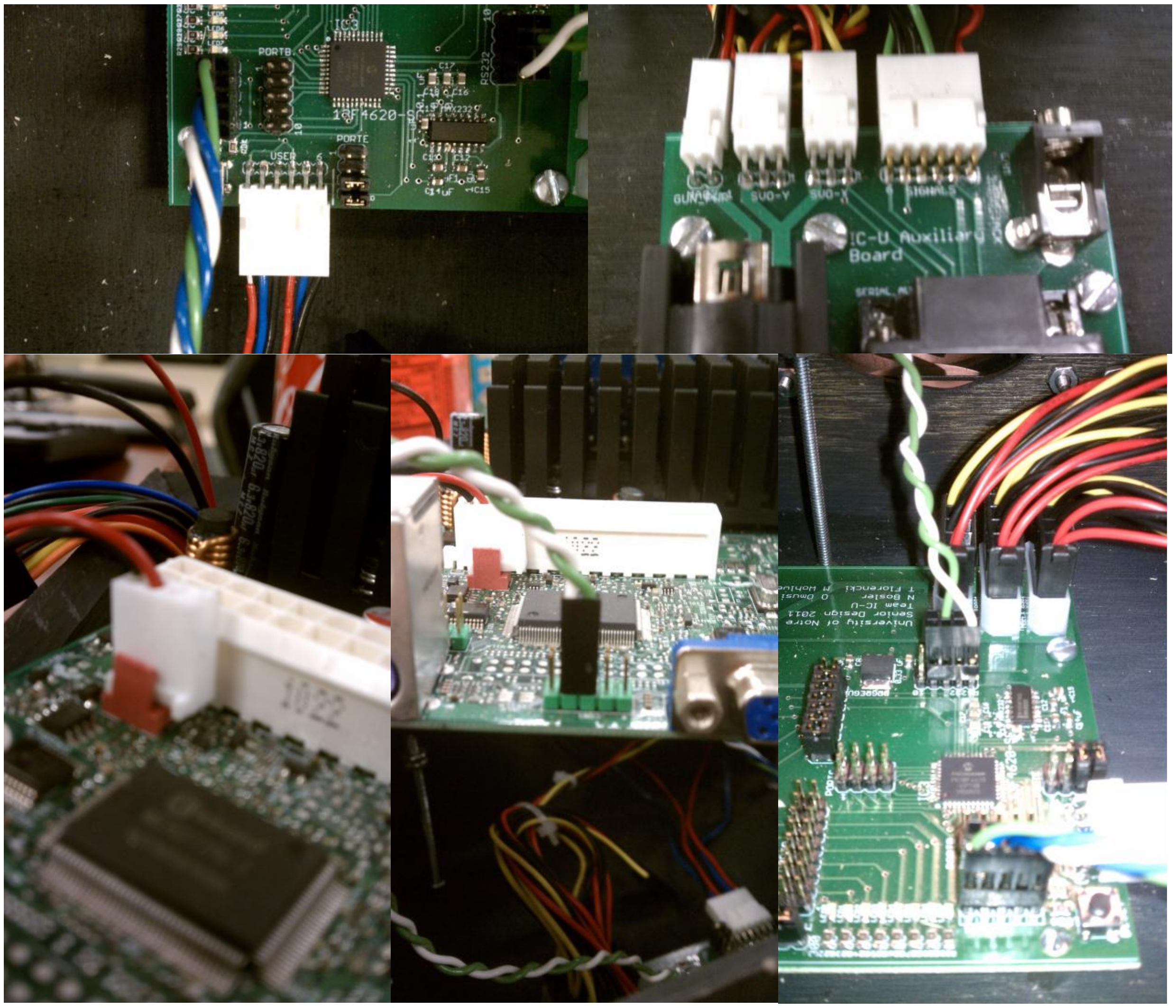
In order to set up the system all the user needs to do is update the effective distance if it is not 30 ft. Provided this is the distance needed, all the user needs to do is power on the system and then set the control switches to the desired operation.

4.3 How to Determine if the Product is Working

While ensuring that your product is working properly is important, safety should always be your top priority when using this system. To safely test this system after it has been installed and set up, the CO2 should not be hooked up to the gun. The first step is to turn on the system and ensure that the system standby (green light) and gun fire switch (red light) are turned on. The next step is to approach the gun mounting system and ensure that it both tracks and fires (you will hear a clicking noise from the gun trying to fire the gun without CO2) as it should. Once system operation has been established, turn off both the system standby and gun fire switches before replacing the CO2 connection to the gun.

4.4 Troubleshoot the Product

If the system is not operating as you expect, the first thing to check is to ensure that all of the cables are connected properly and the power supply switch is turned on. Also, be sure to check that the user-controlled switches are set to the status that they should be. The next thing to check is that none of the internal connections have come loose. To do this, please refer to the following pictures.



5. To-Market Design Changes

There are numerous changes that would be made to this system in order for the product to be taken to market. All of these changes will improve the functionality of the system.

The major change that needs to be made deals with the processing power of the motherboard. As mentioned previously, the motion tracking algorithm had to be stripped down to increase the responsiveness of the system. Before doing this, a delay of about two seconds existed. Stripping the code made the system more sensitive to noise and made it lose its ability to distinguish between two targets. For example, if two people were in the frame and remained at a constant distance apart, the gun would shoot in between the two targets. Also, the target had to be led more than would be necessary with a faster processor. We feel that a processor with a speed above 2.5 GHz would eliminate the slow response time and allow us to return to the more robust motion tracking algorithm. More processing power would also allow the system to respond better to a target with a non-constant speed. Leading the target would still be necessary, however.

Wireless or Ethernet communication should be added to provide a more robust software user interface. This communication would allow the user to view the video feed and control the system settings from a computer.

The mounting system should be made out of a material other than wood, such as metal or plastic. Currently, the axles and gears suffer a fair amount of slip as the servos move quickly. This slip negatively affects the accuracy of the system. Constructing the mount out of a different material will increase the strength of the system, allowing the high resolution camera to be mounted and increasing the precision of the servos, and thus the system as a whole.

The video code should be updated so that it runs upon system start-up. This will eliminate the need for a monitor, keyboard, and mouse. These accessories could still be used to troubleshoot the system but would not be absolutely necessary. This makes system operation easier for the user.

The top for the case should be fastened onto the case to add robustness and stability.

A feature should be added to the mount, such as sonar technology, that will allow the system to determine how far away the target is from the gun and a camera. Currently, the motion tracking algorithm is optimized for a single distance. This distance is used in the trigonometric calculations performed to determine the serial position information sent to microcontroller. If the target is not at the optimum distance, the calculations will be incorrect, effecting the accuracy of the system. Adding distance detection will allow the system to eliminate this problem.

Lastly, an infrared web camera should be used. Currently, the web camera used would not be able to function in an area of extreme darkness, severely limiting the applications for which a system like this could be used. Adding infrared capabilities will make the system relevant in an unlit nighttime situation.

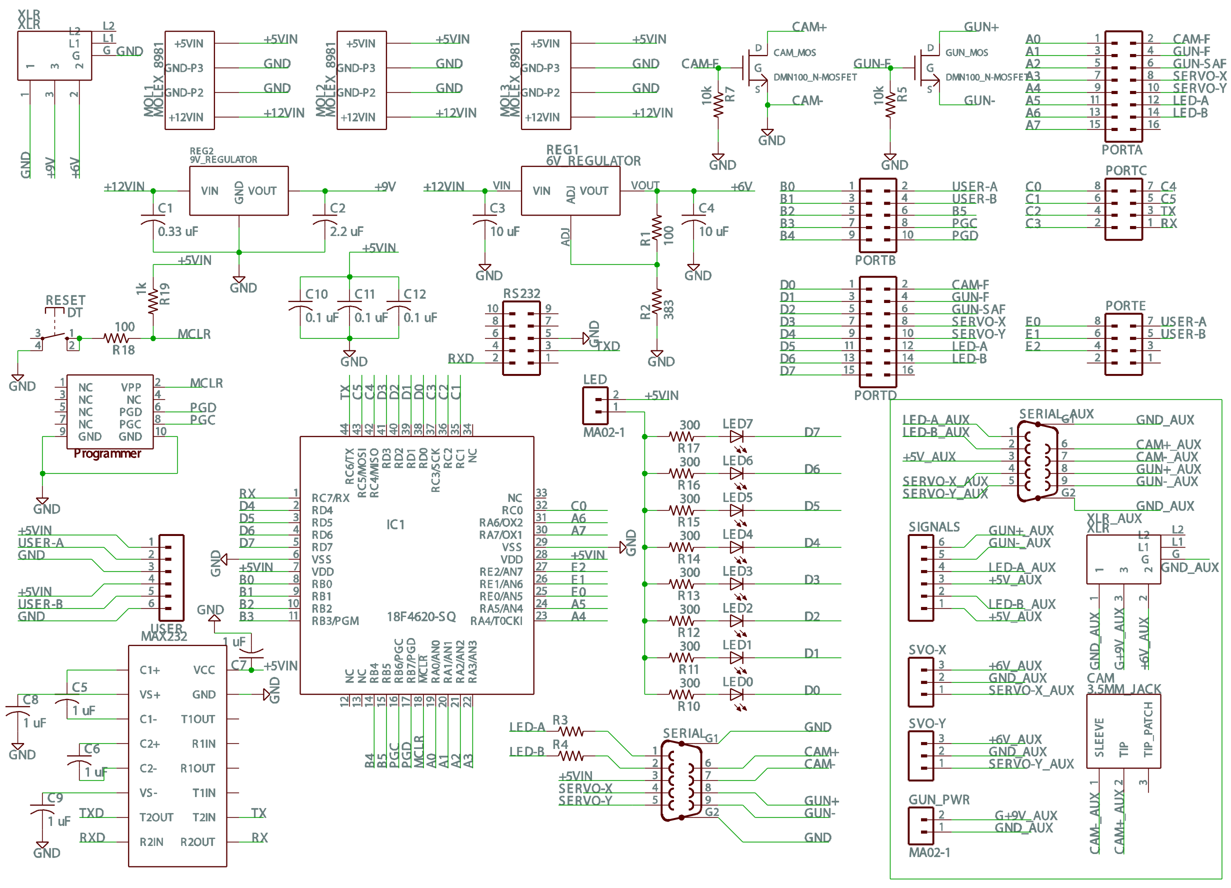
6. Conclusions

Through the use of existing technologies Team IC-U hopes to integrate two useful security features into one system. Integrating the paintball gun allows a security system to fight back against intruders and actively deter crimes from occurring. Using the high resolution camera to identify intruders, the system also helps in the prosecution of the intruder. The system increases safety for its users and their household or business.

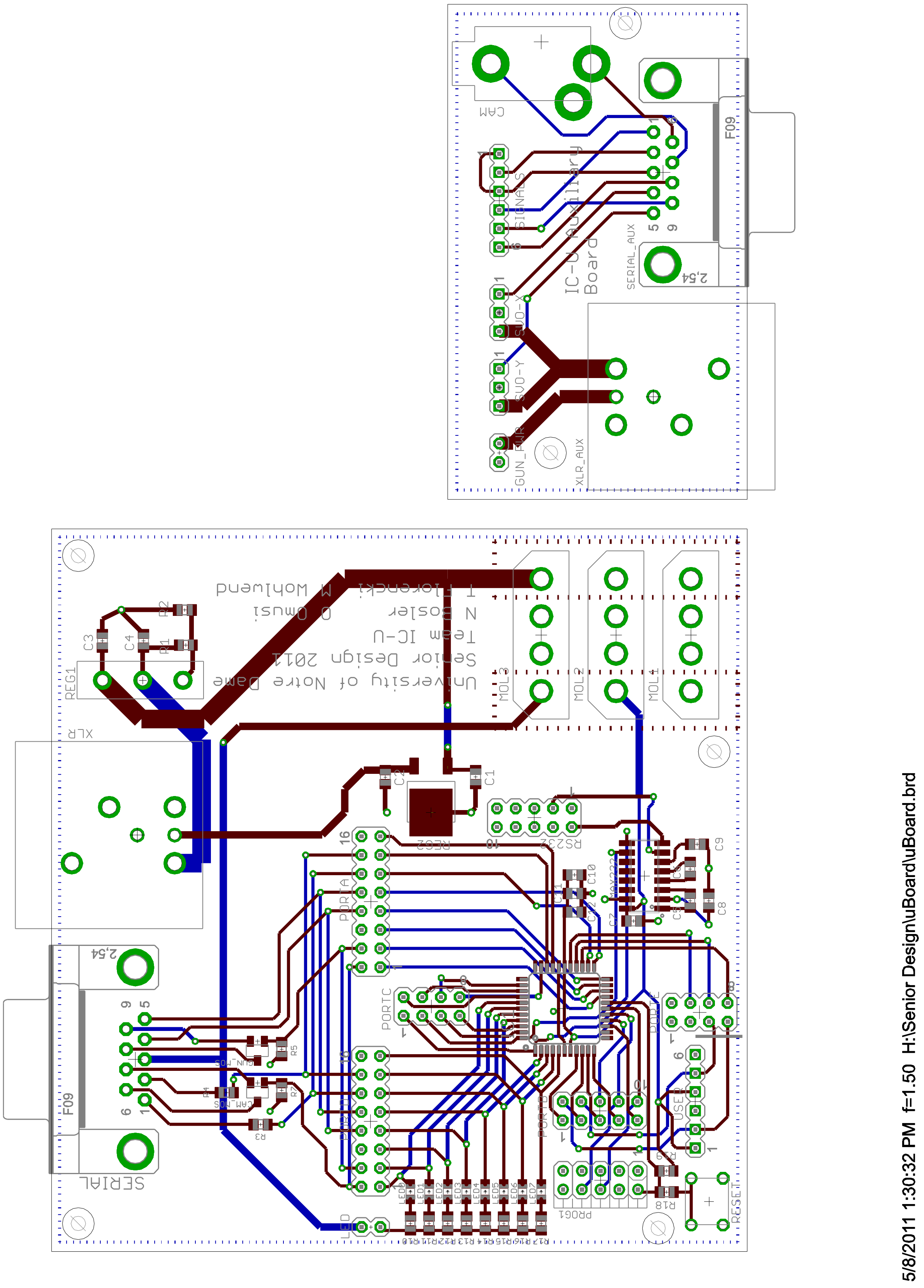
The key challenges of developing this system were the motion sensing algorithm and communication between each system. The overall system is successful in being able to accurately detect and follow motion perceived in the seeker camera, thus proving its worth and potential for retail sale. This in conjunction with its low cost of parts can make it a competitive product in the market.

7. Appendices

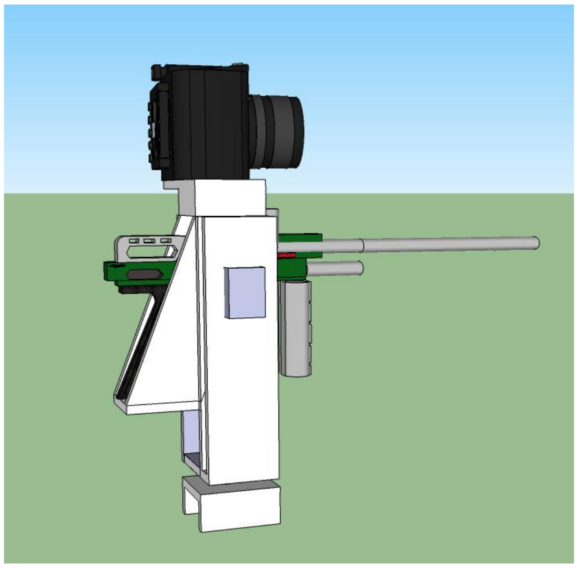
7.1 Board Design and Schematic

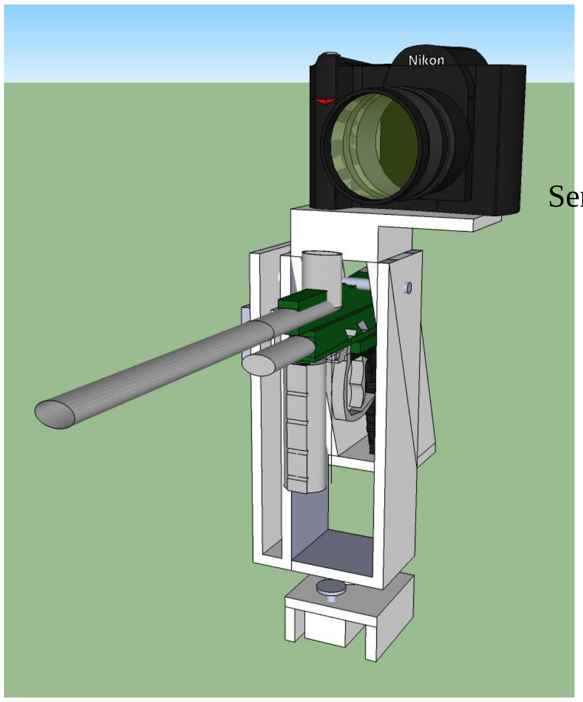


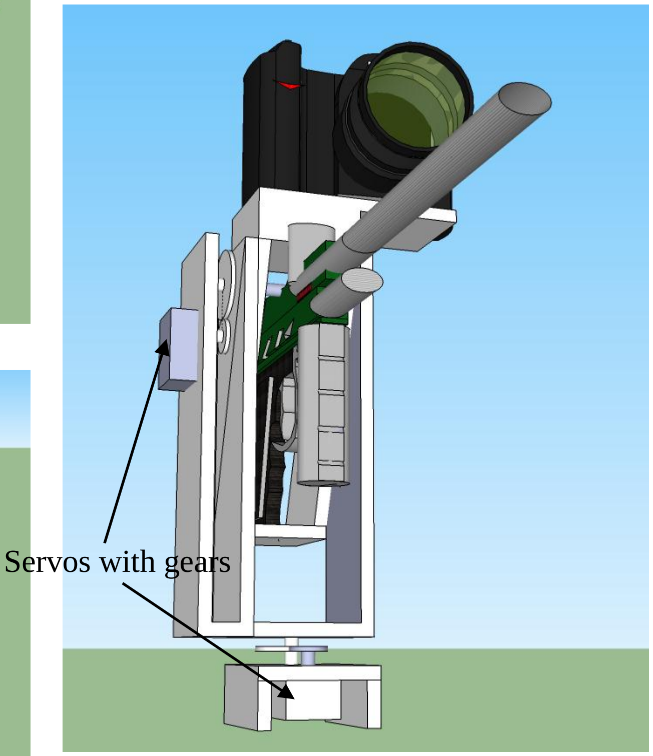
5/8/2011 1:43:26 PM C:\Program Files\EAGLE-5.10.0\projects\uBoard_poster.sch (Sheet: 1/1)


IC-U Auxiliary Board
CAM
SIGNALS
SERIAL_AUX
F09
SVO-X
SVO-Y
GUN_PWR
XLR_AUX
University of Notre Dame
Senior Design 2011
Team IC-U
N Bosler T Florencki
O Omusi M Wohlwend
REG1
XLR
SERIAL
MOL3
MOL2
MOL4
REG2
RS232
PORTA
PORTB
PORTC
PORTD
PORTE
USER
PROG1
RESET
LED
5/8/2011 1:30:32 PM f=1.50 H:\Senior Design\uBoard\uBoard.brd

7.2 Original Mount Design







7.3 Links and References

Links to important data sheets:
Intel Atom D410 Intel NM10 Mini ITX Motherboard/CPU Combo:
http://downloadmirror.intel.com/18358/eng/D410PT_TechProdSpec.pdf
Max232DR:
http://focus.ti.com/lit/ds/symlink/max232.pdf

PIC18F4620:
http://ww1.microchip.com/downloads/en/DeviceDoc/39626e.pdf

References:

http://www.embeddedarm.com/products/board-detail.php?product=TS-7300
http://www.servocity.com/
http://www.pyroelectro.com/tutorials/servo_motor/theory.html
http://www.paintballsentry.com/Products.htm
http://appdelegateinc.com/blog/2010/08/02/motion-tracking-with-a-webcam/
http://www.codeproject.com/KB/audio-video/Motion_Detection.aspx
http://www.ideasonboard.org/uvc/

7.4 Microcontroller Code

```
#include <system.h>

#pragma DATA _CONFIG1H, _OSC_INTIO67_1H
#pragma DATA _CONFIG2H, _WDT_OFF_2H
#pragma DATA _CONFIG4L, _LVP_OFF_4L & _XINST_OFF_4L
#pragma DATA _CONFIG3H, _MCLRE_ON_3H

#pragma CLOCK_FREQ 16000000

#define toggle1 latd.0 = 1; nop(); latd.0 = 0;

volatile bit tmr0if@INTCON.2;
volatile bit tmr0ie@INTCON.5;
volatile bit tmr1if@PIR1.0;
volatile bit tmr1ie@PIE1.0;
volatile bit tmr3if@PIR2.1;
volatile bit tmr3ie@PIE2.1;
volatile bit gie@INTCON.7;
volatile bit peie@INTCON.6;
volatile bit ipen@RCON.7;
volatile bit spen@RCSTA.7;
volatile bit cren@RCSTA.4;
volatile bit brgh@TXSTA.2;
volatile bit sync@TXSTA.4;
volatile bit brg16@BAUDCON.3;
volatile bit rcie@PIE1.5;
volatile bit rcif@PIR1.5;

bool flag0 = false;
bool flag1 = false;
bool flag3 = false;
bool flag_rx = false;
char data;
char array[6];
int data_count = 0;
char value_x = 50;
char value_y = 40;
char value_z = 0;
unsigned short count_x;
unsigned short count_y;
char upper_x;
char lower_x;
char upper_y;
char lower_y;
int gun = 0, cam = 0;
bool fireg = false, firec = false;


void set_serial_receive (void)
{
	//Set baud rate
	spbrgh = 0;
	spbrg = 0x19;

	//Enable asynchronous
	spen = 1;		//serial port enable
	sync = 0;		//set asynchronous mode
	cren = 1;		//continuous receive enable
	brg16 = 0;		//disable 16-bit baud rate
	brgh = 0;		//low speed baud rate
	trisc = 0x80;	//Set C7 to input for serial RX, others to outputs

	//Enable interrupts
	gie = 1;
	peie = 1;
	rcie = 1;
}

void interrupt(void)
{
	if (rcif)
	{
		flag_rx = true;
		data = rcreg;
		array[data_count] = data;
		if (data_count < 5)
			data_count++;
		else
```

```
                    data_count = 0;
        }

        if (tmr0if)
        {
                flag0 = true;
                tmr0h = 0x63;
                tmr0l = 0xC0;
                tmr0if = 0;
                lata.3 = 1;
                lata.4 = 1;
                tmr1ie = 1;
                tmr1if = 0;
                tmr1h = upper_x;
                tmr1l = lower_x;
                tmr3ie = 1;
                tmr3if = 0;
                tmr3h = upper_y;
                tmr3l = lower_y;
                gun++;
                cam++;
        }

        if (tmr1if)
        {
                flag1 = true;
                lata.3 = 0;
                tmr1ie = 0;
                tmr1if = 0;
        }

        if (tmr3if)
        {
                flag3 = true;
                lata.4 = 0;
                tmr3ie = 0;
                tmr3if = 0;
        }
}

void main(void)
{
        osccon = 0b01101000;//set 4MHz speed
        osctune.6 = 1;//activate PLL for 4X speed (16MHz)

        set_serial_receive();
        adcon1 = 0x0F; //set all pins digital
```

```
	trise = 1;
	trisa = 0;
	trisd = 0;
	latd = 0xFF;

	t0con = 0b10000000;//timer0: internal  clock, prescale of 1:8
	tmr0h = 0x63;
	tmr0l = 0xC0;

	t1con = 0b00000001;//timer1: internal  clock, no prescale
	t3con = 0b01000101;//timer3: internal  clock, no prescale

	//set interrupt bits
	peie = 1;
	ipen = 1;
	tmr0ie = 1;
	gie = 1;

	while(1)
	{
		//Get count
		count_x = 65536 - (4000+40*(value_x+8));
		upper_x = count_x >> 8;
		lower_x = count_x & 0xFF;

		count_y = 65536 - (4400+40*(value_y-6));
		upper_y = count_y >> 8;
		lower_y = count_y & 0xFF;

		if (porte.0 == 1) //USER-A (green)
		{
			latd.3 = 0; //board led
			lata.5 = 0; //turret led
		}
		else
		{
			latd.3 = 1; //board led
			lata.5 = 1; //turret led
		}

		if (porte.1 == 1) //USER-B (red)
		{
			latd.4 = 0; //board led
			lata.6 = 0; //turret led
		}
		else
		{
				latd.4 = 1; //board led
				lata.6 = 1; //turret led
			}

			if (flag_rx == true)
			{
				flag_rx = false;
				for(int i=0; i<6; i++)
				{
					if (array[i] == 'x' && porte.0)

					{
						if (i<5)
							value_x = array[i+1];
						else
							value_x = array[0];
					}
					else if (array[i] == 'y' && porte.0)
					{
						if (i<5)
							value_y = array[i+1];
						else
							value_y = array[0];
					}
					else if (array[i] == 'z')
					{
						if (i<5)
							value_z = array[i+1];
						else
							value_z = array[0];

						if (value_z == 101)
							{
							firec = true;
							fireg = false;
							}
						else if (value_z == 102)
							{
							firec = false;
							fireg = true;
							}
						else if (value_z == 103)
							{
							firec = true;
							fireg = true;
							}
						else
```

```
                                                        {
                                                        firec = false;
                                                        fireg = false;
                                                        }
                                        }
                                }
                        }

                        if (flag0 == true)
                        {
                                flag0 = false;
                        }
                        if (flag1 == true)
                        {
                                flag1 = false;

                        }
                        if (flag3 == true)
                        {
                                flag3 = false;

                        }
                // Check for gun-firing conditions
                        if (gun <= 4 && porte.0 && porte.1 && fireg)
                                {
                                lata.1 = 1;
                                latd.1 = 0;
                                }
                        else
                                {
                                lata.1 = 0;
                                latd.1 = 1;
                                }
                        if (gun >= 9)
                                gun = 0;

                        // Check for cam-firing conditions
                        if (cam <= 19 && porte.0 && firec)
                                {
                                lata.0 = 1;
                                latd.0 = 0;
                                }
                        else
                                {
                                lata.0 = 0;
                                latd.0 = 1;
                                }
```

```
		if (cam >= 39)
			cam = 0;
	}
}
```

7.5 Computer C code

```
//-----------------------------------------------//
//
// track.c
//
// University of Notre Dame EE Senior Design 2011
// Team IC-U
//
//-----------------------------------------------//

#include <stdlib.h>
#include <stdio.h>
#include <cv.h>		// openCV functions
#include <highgui.h> // openCV GUI functions
#include <string.h>  // String function definitions
#include <unistd.h>  // UNIX standard function definitions
#include <fcntl.h>   // File control definitions
#include <errno.h>   // Error number definitions
#include <termios.h> // POSIX terminal control definitions
#include <math.h>    // Standard math functions

int open_port(void);
void write_serial(int x, int y, int fire);

int main(void)
{
	// Initialize memory for cvFindContours
	CvMemStorage *storage = cvCreateMemStorage(0);

	// Open webcam
	CvCapture* capture = cvCaptureFromCAM( CV_CAP_ANY );
	if( !capture ) {
		fprintf(stderr, "Cannot initialize webcam\n");
		getchar();
		return -1;
	}

	// Create a window in which the captured images will be presented
	cvNamedWindow( "motion", CV_WINDOW_AUTOSIZE );


	// Capture first frame to get size
```

```
IplImage *frame = cvQueryFrame(capture);
CvSize frame_size = cvGetSize(frame);
IplImage *grey_image = cvCreateImage(frame_size, IPL_DEPTH_8U, 1);
IplImage *difference = cvCloneImage(frame);
IplImage *previous_image = cvCloneImage(frame);

// Initializations
IplImage *color_image = 0;
IplImage *display_image = 0;
int thresh = 500, min_move = 50;
int fire = 100;
float pi = 3.14159;
int xtot, ytot, count;
CvPoint center_pt;
int area, previous_area;
CvRect bound_rect;
int distance = 20;
float theta, phi;
center_pt.x = 320;
center_pt.y = 240;
int xminus1 = 0, xminus2 = 0, xminus3 = 0;
int xtarget = 0, ytarget = 0;
CvPoint target;
// Compute total view of camera based off distance
float half_x = distance * tan(25*pi/180), half_y = distance * tan(20*pi/180);


while (1)
{

        // Grab previous image
        if (color_image != 0)
                previous_image = cvCloneImage(color_image);

        // Capture frame from webcam
        color_image = cvQueryFrame(capture);
        display_image = cvCloneImage(color_image);

        // Minus the current frame from the previous image
        cvAbsDiff(color_image, previous_image, difference);

        // Convert the image to grayscale.
        cvCvtColor(difference, grey_image, CV_RGB2GRAY);

        // Convert the image to black and white.
        cvThreshold(grey_image, grey_image, 70, 255, CV_THRESH_BINARY);
```

```
		// Find contours of the image
		CvSeq *contour = NULL;
		cvFindContours(grey_image, storage, &contour, sizeof(CvContour),
CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE, cvPoint(0,0));

		// Clear averages
		xtot = 0;
		ytot = 0;
		count = 0;
		previous_area = 0;

		for( ; contour != 0; contour = contour->h_next ) {
			// Find bounding rectangle around contour
			bound_rect = cvBoundingRect(contour, 0);
			area = bound_rect.width * bound_rect.height;


			if (area > min_move) { // if movement is above min threshold
				// Determine center of rectangle
				int x = bound_rect.width / 2 + bound_rect.x;
				int y = bound_rect.height / 2 + bound_rect.y;

				// Display Rectangles
				CvPoint pt1 = {bound_rect.x, bound_rect.y};
				CvPoint pt2 = {bound_rect.x + bound_rect.width, bound_rect.y +
bound_rect.height};
				cvRectangle(display_image, pt1, pt2, CV_RGB(255,0,0), 1, 8, 0);

				// Save info for average
				xtot = xtot + x;
				ytot = ytot + y;
				count++;

				// Find largest area of motion
				if (area > previous_area)
					previous_area = area;
			}
		}

		// if motion was detected find averaged center
		if (count != 0) {
			center_pt.x = xtot / count;
			center_pt.y = ytot / count;
		}
		// determine if motion is large enough to shoot at
	if (previous_area > thresh)
			fire = 103;
		else
			fire = 100;

		// Add a lead to the moving target
		xtarget = center_pt.x + (10)*(center_pt.x - xminus3)/3;
		ytarget = center_pt.y;
		target.x = xtarget;
		target.y = center_pt.y;

		// Display targeting circle to screen
		cvCircle(display_image, target, 10, CV_RGB(255, 100, 0), 1, 8, 0);
	cvCircle(display_image, target, 6, CV_RGB(255, 100, 0), 1, 8, 0);
	cvCircle(display_image, target, 2, CV_RGB(255, 100, 0), 1, 8, 0);
		cvCircle(display_image, target, 15, CV_RGB(100, 100, 255), 3, 8, 0);
		cvShowImage("motion", display_image);

		// Grab updated averaging values
		xminus3 = xminus2;
		xminus2 = xminus1;
		xminus1 = center_pt.x;

		// Convert pixel position to angle at given distance
		int x = 100-(10*xtarget)/64;
	int y = 80-ytarget/6;

	theta = atan(((x*2*half_x)/100 - half_x)/distance);
	int xdeg = ((theta*180/pi) + 25)*2;

	phi = atan(((y*2*half_y)/80 - half_y)/distance);
	int ydeg = ((phi*180/pi) + 20)*2 + 5; // Add 5 degree vertical offset

		// Send serial info
	if (center_pt.x != 0 && center_pt.y != 0)
			write_serial(xdeg,ydeg,fire);
		else
			write_serial(50,40,100);

		// Check for escape
		if ( (cvWaitKey(10) & 255) == 27 ) break;

	}

	// Clean up memory
```

```
        cvReleaseMemStorage( &storage );
        cvDestroyWindow("motion");
        cvReleaseCapture(&capture);
        return 0;

}

int open_port(void) {

        int fd; // File descriptor for the port

        fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NDELAY);
        if (fd == -1)
        {
                // Could not open the port.
                perror("open_port: Unable to open /dev/ttyS0 - ");
        }
        else
                fcntl(fd, F_SETFL, 0);
        return (fd); // Return file discriptor
}

void write_serial(int phi, int theta, int fire) {
        int port;
        port = open_port(); // Get file descriptor

        // Cast input int to char for output
        char PHI = (char) phi;
        char THETA = (char) theta;
        char FIRE = (char) fire;

        struct termios options;

    // Get the current options for the port
    tcgetattr(port, &options);

    // Set the baud rates to 9600
        cfsetispeed(&options, B9600);
        cfsetospeed(&options, B9600);

    // Enable the receiver and set local mode
    options.c_cflag |= (CLOCAL | CREAD);

        // Set no parity
        options.c_cflag &= ~PARENB;
        options.c_cflag &= ~CSTOPB;
        options.c_cflag &= ~CSIZE;
        options.c_cflag |= CS8;

    // Set the new options for the port...
    tcsetattr(port, TCSANOW, &options);

        // Output to port
        int n;
        n = write(port, "x", 1);
        n = write(port, &PHI, 1);
        n = write(port, "y", 1);
        n = write(port, &THETA, 1);
        n = write(port, "z", 1);
        n = write(port, &FIRE, 1);
        if (n<0)
                fputs("Write failed\n", stderr);
        close(port);
}
```