# Gaze Tracking Algorithm

## Introduction to Gaze Tracking:

The premise behind Gaze tracking is to track the line of the sight of the person. Eye Gaze tracking has been an active research topic for decades because of its potential usages in Human Computer Interaction (HCI). A popular application of gaze tracking is to allow disabled users with controlling common computer task such as clicking or moving the cursor.

Gaze tracking is commonly done using glint-pupil vector tracking. This method uses an infrared light to illuminate the eye and increase contrast in the eye for ease of tracking pupil and other useful features of the eye. Glint is the reflection of the light off of the retina that appears as a bright spot on the cornea [1] [2]. It is used as a reference point in the eye because the glint location remains stationary with respect to the eye as the direction of gaze changes. If the infrared illumination is coaxial with the optical path, then the eye acts as a retroreflector as the light reflects of the retina creating a bright pupil effect

If the illumination source is offset from the optical path, then the pupil appears dark because the reflection from the retina is directed away from the camera. The bright and dark pupil effects enhance the contrast of features in the eye that make the algorithm more accurate. This method tracks the pupil, and ultimately the vector between the pupil and glint. This vector is then mapped (mapping is obtained during calibration process) to location in the screen [2].

While the glint-pupil vector tracking is proved to be very accurate in determining gaze of user’s eyes, it is computationally intensive. Since our algorithm must be running real time on a microprocessor, speed of computation is a huge constraint. Thus we decided to merely track the pupil to find the direction of user’s gaze. Furthermore, while experimenting with using infrared light to illuminate the eye proved to make the algorithm accurate, issues with IRB approval prohibited us from directing infrared light at the eye. This is discussed in great detail in section 5.

## Overview of Approach

To determine the direction of user’s gaze, we aim to track the pupil of the user. To implement gaze tracking, we take advantage of a few characteristics of human eyes and more specifically of the pupil. Since the pupil is the darkest region of the human eye, we base our algorithm in analyzing dark regions in each frame. Secondly, we expect the pupil to be circular and thus look for circular region. Finally, we expect the pupil to have moved in continuous fashion, e.g the position of the pupil does not change significantly in small amount of time.

To obtain images of the user’s eyes, we use a commercial Logitech QuickCam Chat Web Camera shown in Figure 1. The camera is rated to output up to 30 frames per second with frame resolution of 640x480.



Figure 1: Logitech QuickCam Chat Web Camera

To obtain image data from the web camera, OpenCV 2.0 is used. Once we have obtained the image data, we begin implementing our image processing algorithm that is outlined in Figure 2.

We begin by Thresholding the image which finds pixels in the image that have intensity value below a calibrated threshold value. From the selected pixels, we find connected regions. For a connected region to be passed as the pupil, we have three requirements: 1) connected region has approximately the size of the user’s pupil that is determined from calibration, 2) the region is approximately circular, and 3) continuity condition: the centroid of the candidate region should remain close to previously found centroid. If these requirements are not met, then the threshold parameter is varied until they are met or until we have reached maximum number of iterations that indicate that the user is blinking. Once such region that meets these requirements is found, the region is cleaned up in the “Remove Tails” stage and its centroid is found.



Figure 2: Overview of Image Processing Algorithm

## Algorithm Implementation

Since our algorithm is being implemented in real time, computational efficiency is extremely important. To maximize the efficiency of our algorithm, we organize our data in the following data structures shown in Table 1 :

|  |  |  |
| --- | --- | --- |
| Data Structure Name: | Data Structure Type | Data Stored |
| imageData | Three dimensional array of bytes | The RGB pixel values for each coordinate in a given frame. |
| crPointList | Two dimensional array of points | Each row contains the coordinates of a connected region. Only regions that meet the area requirement are stored. |
| crBinary | Three dimensional array of binary Value | Binary(i,x,y) = 1 if the coordinate (x,y) is in region i, and 0 otherwise |
| crMap | Three dimensional array of integers | Contains the integer index which maps a point from the matrix form of the region to an element in crPointList. |
| crSize | Array of integers | Elements are the sizes of the connected regions. |
| crCount | Integer | Number of connected region stored. |

Table 1: Data structures

|  |  |
| --- | --- |
| Module Name: | threshold() |
| Inputs: | imageData, initial threshold |
| Outputs: | List of points that satisfy threshold criteria |
| Functional Description: | Scans each pixel in the region of interest in a frame and checks to see which pixels are dark enough to belong to the pupil. This process is repeated until a region (computed with getConnectedRegions()) with an area close to a reference area is found, or until a maximum number of iterations has been reached. If the maximum number of iterations is reached and no suitable regions are detected, identify the user as blinking. |

|  |  |
| --- | --- |
| Module Name: | getConnectedRegions() |
| Inputs: | List of dark points identified in threshold() |
| Outputs: | crPointList, crSize, crCount,crBinary |
| Functional Description: | Uses a stack based implementation of the flood fill algorithm to identify connected regions of dark points. |

|  |  |
| --- | --- |
| Module Name: | findUnityRatio() |
| Inputs: | crBinary, crCount |
| Outputs: | Aspect ratio for each connected region in CR, index of the connected region with aspect ratio nearest to one |
| Functional Description: | Computes the ratio of the longest horizontal and longest vertical lengths. The connected region with the aspect ratio closest to one is identified as the pupil. |

|  |  |
| --- | --- |
| Module Name: | removeAberrations() |
| Inputs: | crPointList, crMap, crSize, Index indicating chosen region |
| Outputs: | Updated crPointList and crSize |
| Functional Description: | Computed the number of pixels in each row of the connected region and find the mean and standard deviation of the pixel counts. Remove rows that have pixel counts that fall out of a certain number of standard deviations away from the mean. Repeat the process in the vertical direction. |

|  |  |
| --- | --- |
| Module Name: | computeCentroid() |
| Inputs: | crPointList, crSize, Index indicating chosen region |
| Outputs: | Coordinates of the centroid |
| Functional Description: | Sum the coordinates of all points belonged to the pupil region and divide by the total number of points. The result is the coordinate of the centroid. |

|  |  |
| --- | --- |
| Module Name: | generateCursorCommand() |
| Inputs: | Reference centroid coordinates, current centroid coordinates |
| Outputs: | Cursor command code (integer values ranging from 0 to 5) |
| Functional Description: | Compares the reference centroid coordinates to the current centroid coordinates. If the difference between the two coordinates exceeds a threshold value for 10 consecutive frames, then the cursor command output will be changed accordingly. Otherwise, the previous cursor command is output. |

|  |  |
| --- | --- |
| Cursor Command | Direction of Movement |
| 0 | No change |
| 1 | Right |
| 2 | Left |
| 3 | Up |
| 4 | Down |
| 5 | Left Click |

Table 2: Translation between cursor command and direction

## Lighting Configuration

After testing our image processing algorithm in Capstone without a controlled lighting environment, we saw that reflection and glares from the fluorescent light on the pupil confused our algorithm. To combat this issue, we have a lamp that point to the back wall in capstone and away from the user. This resolved most of our issues with reflections, though the conditions are still not ideal for accurate gaze tracking. An image of the lamp we used is shown in Figure 3.



Figure 3: Configuration of lamp in Capstone

## Working Example

For better understanding of our algorithm and its main steps, we will demonstrate the result of each step of our algorithm on the image shown in Figure 4 obtained from out webcam:



Figure 4: Original Image

Suppose that from a calibration process, the initial threshold value was chosen to be 34 and the reference pupil area was selected to be 554. After applying thresholding and coloring pixels that have intensity value less than the threshold value, we obtain the result shown in Figure 5.



Figure 5: After Thresholding: Where biggest connected region is colored yellow and the second largest colored red

Note that two connected regions that have area area relatively close to the reference pupil area are the eyebrow and the pupil. The eyebrow is computed to have 2416 pixels and the pupil has 1024 pixels. Since both connected regions have areas that fall outside of the acceptable deviance from the reference pupil area, we do not consider them as candidates for the pupil. Since both regions have area greater than reference pupil area, out algorithm decreases the threshold value to 31. The result after adjusting the threshold value is shown in Figure 6.



Figure 6: Image after adjusting threshold value

The area of each connected region in the processed image is:

Eyebrow 1: 564

Eyebrow 2: 196

Pupil: 512

Note that the first and third connected regions have area within the acceptable range to be considered as the pupil. To distinguish between these two connected regions, we compute the aspect ratio of both connected regions:

Eyebrow Aspect ratio = 0.49

Pupil Aspect Ratio = 1.07

The pupil aspect ratio is within the acceptable range but the eyebrow aspect ratio is not. Thus our algorithm correctly identifies the pupil as the region of interest. Finally, we remove aberrations of the pupil and obtain the processed image shown in Figure 7 with the centroid of the region denoted as intersection of the two lines:

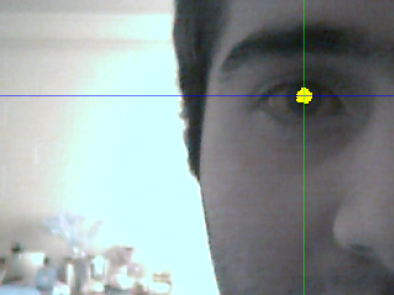


Figure 7: Final Processed Image

# Command Interpretation

The output of the gaze tracking algorithm is a command that indicates the direction of user’s gaze. The cursor command value ranges from 0 to 5. The interpretation of cursor commands to an associated cursor response is shown in Table 3.

|  |  |  |
| --- | --- | --- |
| Cursor Command | Direction of Movement | Vector |
| 0 | No change | (0,0) |
| 1 | Right | (1,0) |
| 2 | Left | (-1,0) |
| 3 | Up | (0,-1) |
| 4 | Down | (0,1) |
| 5 | Left Click | N/A |

Table 3: Translation of cursor command to vector for direction of movement

Since the human eyes often move involuntary, we require that the output of our algorithm is a particular cursor command for ten consecutive processed frames before executing cursor movement or click. Once this requirement has been met, cursor movement is implemented using Windows API (Application Programming Interface).

For simplicity, the implementation of cursor movements/clicking is done in Python. The cursor commands 0-4 received from the processing module is associated with a vector as demonstrated in Table 3. For such cursor commands, the new cursor position is obtained by the following expression:

where the parameter, speed, is adjustable by the user. Finally, the cursor command “5” corresponds to a left mouse click. This was done with Windows API call.

# Calibration:

## Why Do We Calibrate?

Since characteristics of the human eyes vary between users, the user is guided through a calibration process where the characteristics of their eyes are obtained. Parameters that are obtained through calibration stage are:

1. Initial Threshold Value: the initial value that is used by algorithm to locate pixels that have intensity value below the initial threshold value.
2. Region of Processing: a user adjustable box that would encapsulate the eye.
3. Minimum change in distance from reference pupil to associate with change in direction of gaze.
4. Reference pupil centroid when user is looking at the center of screen.
5. Reference pupil area: the number of pixels of the centroid when user is looking at the center of screen.

Note that parameters 1 and 2 are manually adjusted by the user whereas the others are computed by the gaze tracking algorithm

## Calibration GUI

For ease of adjusting calibration parameters, we designed an interactive calibration process, in which the user will be able to modify algorithm parameters via a GUI on the computer.

In the first calibration process, the user is asked to manually adjust the threshold value until the overlay image has his/her pupil colored red. The user will also adjust the boundaries of processing region to enclose only the pupil. Note that the adjusted calibration parameters are communicated to the beagle bone (for more details on this communication, please refer to section 4 ). The beagle bone will process the image by marking the pixels with intensity value below a threshold value as red. The processed calibration frames from the camera will be transferred via Ethernet on the beagle bone to the computer so that the user can view the effect of these parameters in real time and choose suitable values (for more details on this communication, please refer to section 4.1).

Once the user is satisfied with the result, they will click next on the GUI, which will trigger the Host computer to send an integer to the beagle bone to let it know to process the frame to find two calibration parameters, the reference centroid and the reference pupil area. In the second step of the algorithm, the user is asked to adjust the boundaries of the processing region until it encloses the whole eye.

Finally, in the last stage of calibration, the user is asked to gaze at a black dot that appears in the top of the screen. Once done, user presses a key that results to an integer being sent to the beagle bone via Xbee. Once this is recognized by the Beagle Bone, the most current frame is processed to find the relative change in number of pixels from the current centroid to the reference centroid. This process is repeated with black dot in the bottom of screen, right of screen, and left of screen. This stage modifies the parameters associated with how far the user must look in a particular direction before being considered gazing in that direction.

The Calibration GUI was implemented in python using the Tkinter, a standard python GUI interface. Screen captures of a few steps of calibration process are shown in the following figures.

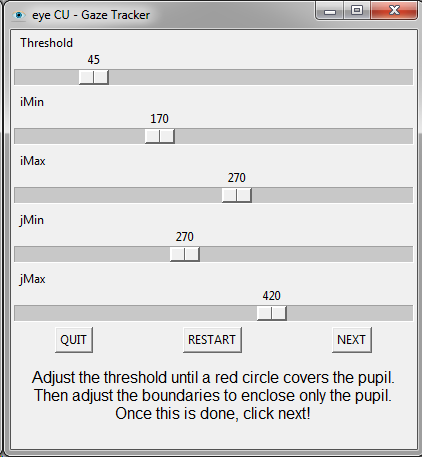


Figure 8: First Step of the calibration GUI

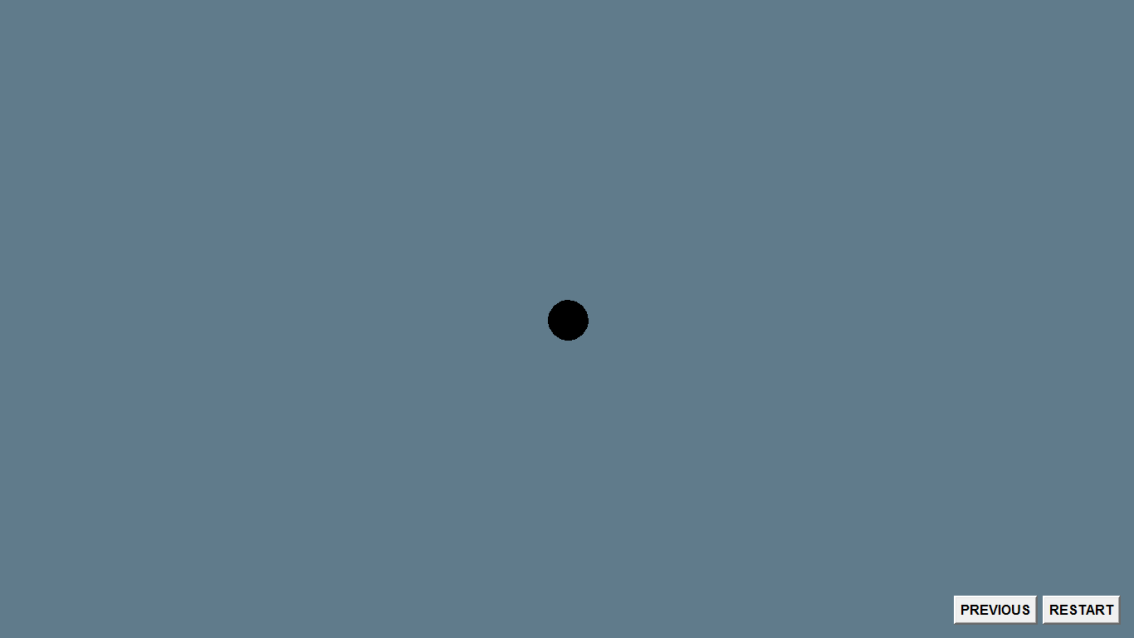


Figure 9: Third step of calibration with black dot at the center of screen

# Software Interfaces:

## Host Computer and Beagle Bone Communication: Transferring Frames

The processed frame by the beagle bone is transferred via USB on the Beagle Bone to the Host Computer. The type of transfer is bulk transfer. The transfer rate is approximately a frame per second. Though this rate is rather slow, it is not an issue because this is only used for debugging purposes. The code for video transfer on the host computer side as well as the beagle bone can be found in appendix INSERT!!!!

## Beagle Bone and MSP430 Communication

To communicate cursor commands and calibration data between the MSP430 board and the Beagle Bone, we are using XBEEs. Interfacing to an XBEE is straightforward; all you have to do is use a standard UART (universal asynchronous receiver transmitter) interface. On each of the processors chosen we have several UART ports built in for us to use. Programmatically communicating using XBEEs is just like talking over a serial port, the XBEE modules handle all of the wireless transmission.

## MSP430 and Host Computer Communication

Communication between the host computer and the MSP430 board is another UART serial interface. This time we have a FTDI chip that converts UART to USB for transmission. On the host computer side there is a USB to serial driver. In this way we can communicate with the board using a serial module in any programming language. For our project, we decided to use python, and the pySerial module. The python program uses a blocking read to look for data on the serial port. Once the cursor command is received, the host computer processes the command. The communication is two way, so that we can send calibration data back to the Beagle Bone.

**JTAG Connector**

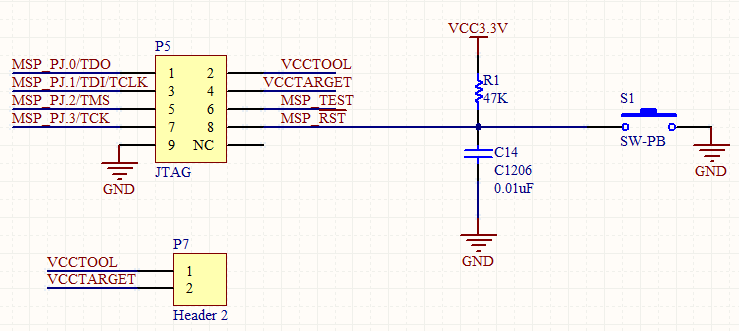


Figure 10: JTAG Programmer Connector Schematic

In order to program the MSP430, we needed a way to load code. The MSP430F5438A requires a 4-wire JTAG programming protocol. Using the Texas Instruments SLAU278H JTAG configuration, a schematic of this interface was developed as shown in Figure 10 [3].s The MSP-FET430UIF was used as the JTAG programmer for the MSP430F5438A through Code Composer Studio version 5. Note that the MSP-FET430UIF cannot be used to program the MSP430F5438A in Code Composer Studio version 4 due to an incompatible firmware update for the programmer. This information is currently not well documented in the literature provided by Texas Instruments; however, information on this issue may be found on the TI E2E Community Support Forums. As shown in Figure 10, the four pins, TDO, TDI, TMS, and TCK are used to program the MSP430F5438A. Additionally, the two pins, VCC TOOL and VCC TARGET are used to either power the programmer via USB power supplied by the host computer or powering from the target MSP430F5438A board. Depending on the user’s requirements, the user may choose to power the programmer off of the board or via USB through a jumper connection on the P7 header.

The programmer also has a connection to the /RST pin of the processor for resetting the processor after programming. The reset switch provides a quick way to re-run code on the MSP430F5438A in the event that the processor gets hung-up, or to simply reset the processor. In order to reset the processor, /RST is brought to ground. When the processor is not in reset, and in order to maintain normal operation, /RST is pulled up to VCC. The reset circuit consisting of R1, C14, and the momentary push button is called a button debounce circuit. This circuit prevents spikes of high and low voltages corresponding to high and low input to the reset pin of the processor. This debounce circuit relies principally on the capacitor which serves to resist sudden voltage changes on the line.

**RS232 Interface**

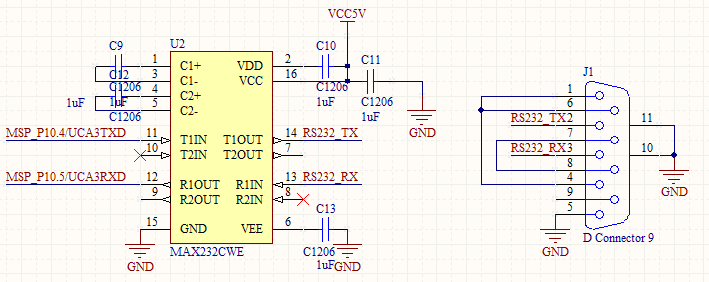


Figure 11: RS232 Interface Schematic

As a back-up in case there was an issue with the USB interface with the MSP430F5438A we employed an RS232 interface as shown in Figure 11. This would provide an additional interface between the MSP430F5438A and the host computer for XBee RX and TX data. As you can see in the schematic, the MAX232 is powered externally by 5V but it requires 10V to operate which is supplied internally through dual charge-pump DC-DC voltage converters. The first charge pump on the MAX232 takes in 5V doubling the voltage to approximately 10V. While the second charge pump takes the +10V, inverts the input, outputting -10V. The charge pump, capacitor values, and pin configuration for typical operation can be found in the datasheet in the appendix for the MAX232. Furthermore, for interfacing a host computer over serial, a DB9 serial connector is necessary as shown in Figure 11.

**USB Circuit**

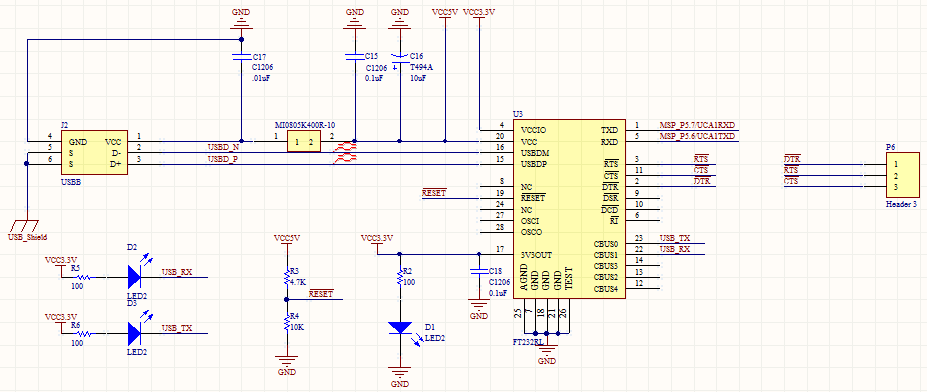


Figure 12: USB Interface Schematic

Figure 12 displays the USB Interface Schematic for the custom MSP430 board. The FT232RL chip from FTDI is a USB to serial interface allowing the MSP430F5438A to communicate XBee RX and TX data over a USB connection with the host computer. The design of this circuit was facilitated using an application note by FTDI on hardware design guidelines for FTDI ICs [4]. As shown in Figure 12, the FT232RL is in a self-powered or bus powered configuration, drawing power from the USB-B connector connected to the host computer. Additionally, the FT232RL has a +3.3V output from an integrated LDO regulator which the datasheet advises should be decoupled to ground using a 100nF capacitor. The pin VCCIO is the power supply for the UART interface and CBUS group pins. The MSP430F5438A datasheet recommends a maximum of +3.6V at any pin, the VCCIO pin on the FT232RL must be set to +3.3V in order to ensure recommended operating conditions have been met. Since the FT232RL outputs +3.3V at pin 17, VCCIO could be powered by the FT232RL. Due to an oversight, VCCIO was tied to +5V so the PCB trace had to be removed and a white wire was added to connect pin 17 to VCCIO. Note that Figure 12 shows the proper configuration.

You can see in Figure 12 that a ferrite bead is connected in series with the USB power supply to reduce EMI noise from being radiated down the USB cable to the USB host from the FT232RL and associated circuitry. Obviously, the value of the ferrite bead depends on the total current drawn. A 1.5A 40Ω ferrite bead was employed by Laird Technologies. Furthermore, decoupling capacitors C15 and C16 were also added to minimize EMI emission and improve ESD immunity. The signals D- and D+ were routed as a differential pair in order to reject common-mode noise. This means that D- and D+ had very equal lengths in order to maintain the timing of signals. In addition, the overall length of these two signals was made to be as short as possible.

As a visual indication that the FT232RL was correctly powered, a power green LED was connected to the 3V3OUT pin. Additionally, since any of the CBUS I/O pins can be configured to drive an LED, LEDs were placed on CBUS0 and CBUS1 as visual indicators when data is being transmitted or received.

In Figure 12, /RESET on the FT232RL is connected to voltage divider consisting of R3 connected to 5V, in series with R4 connected to ground. This circuit serves to hold the IC in reset while a USB cable is not connected to the peripheral.

**XBee Circuit**

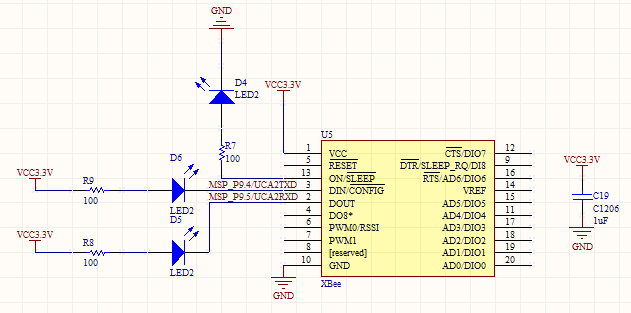


Figure 13: XBee Schematic

The XBee Series 1 chip antenna wireless transceiver was used to send as well as receive data from the XBee on the Beagle Bone to the host computer and vice versa. This XBee is controlled by the MSP430F5438A through a UART on the MSP.

The primary purpose of the LEDs on ON, DIN, and DOUT of the XBee is for debugging purposes. Additionally, a decoupling capacitor was placed across power and ground on the XBee. In an attempt to reduce reflections from the wireless transmitter, no components were placed immediately beneath the XBee in order to prevent turning a component into a disruptive antenna.

NEED KHASHI TO ADD INFO ABOUT XBEE CONFIGURATION!!!

**Microprocessor**

The design of the XBee transceiver board centers around the processor. This processor needed to be able to arbitrate data between an XBee and the host computer. The processor had to be reliable, powerful enough to arbitrate data, easy to develop with from both a software and hardware perspective as well as readily available. Since Capstone provides MSP430 development boards by Olimex and Texas Instruments offers free samples of most MSP430s we reasoned that we could begin development immediately on the MSP430 architecture. We chose the MSP430F5438A processor because we needed a processor with at least two UARTs, one for XBee communication and one for USB communication with the host computer. In addition, we wanted to add an RS232 serial interface as a backup in case the USB interface did not operate as desired. Therefore we needed a processor with at least three UARTs. Texas Instruments only has MSP430s with either two or four UARTs but not three so we decided to not sacrifice a UART and go with four. The MSP430F5438A has four UARTs, in addition to coming in a 100 pin LQFP package.

**MSP430F5438A**

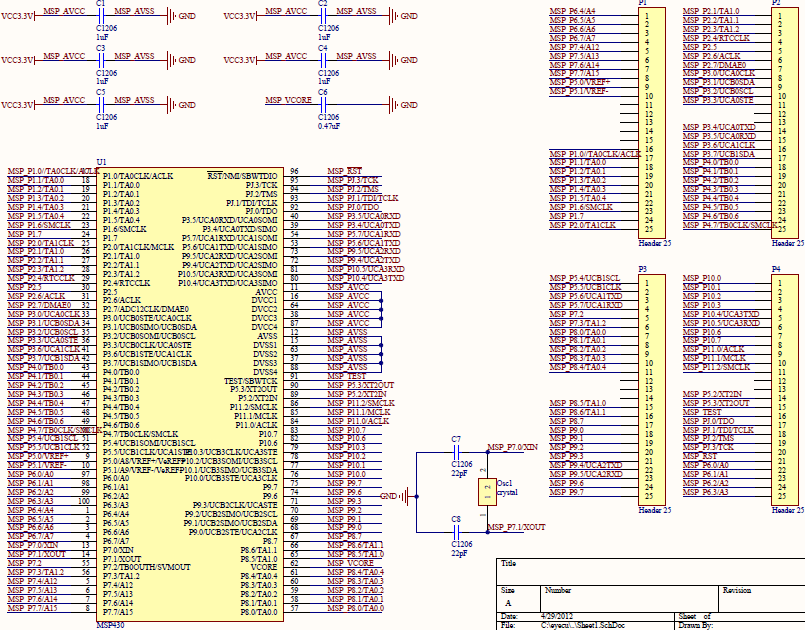


Figure 14: MSP430F5438A Processor Pin Breakout

The processor we are using on the MSP430 board is a MSP430F5438A. A schematic of the processor can be seen in Figure 14. As you can see in the schematic, the MSP430 runs off of 3.3V which is generated by the USB circuit. The MSP430 is supported by an external oscillator circuit with a 32.768kHz crystal. The MSP430F5438A has an internal 32.768kHz oscillator; however, it is recommended to use an external oscillator for time sensitive applications.

**Power Supply**

The power supply is designed to supply power to the Beagle Bone. It utilizes a linear regulator which takes in an input voltage range of 6V to 15V and regulates this voltage to 5V. The power supply is powered by a 6.6V lithium iron phosphate battery (LiFePO4). Figure 15 shows the LiFePO4 battery used in the power supply.



Figure 15: 6.6V LiFePO4 Battery

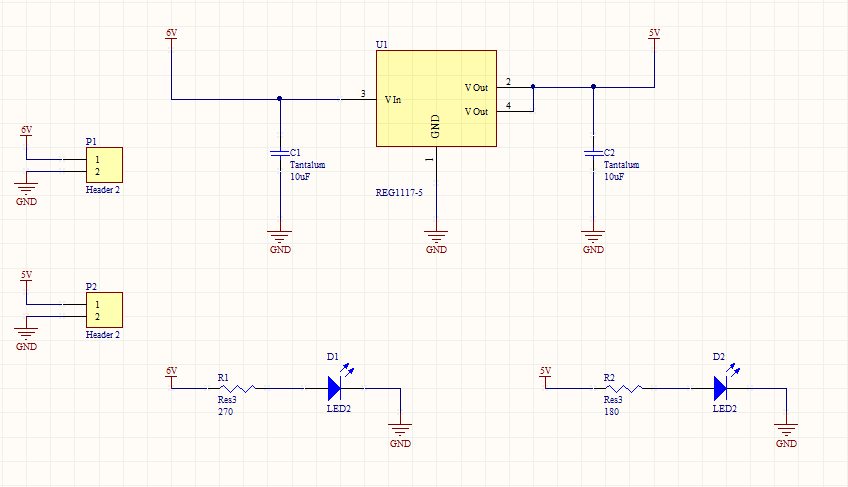


Figure 16: Power Schematic Revision 1

The first revision of the power system, shown in Figure 16, utilizes a single linear regulator with two decoupling capacitors. Status LEDs are added to the Vin and Vout rails of the power supply to provide a visual indication of an input voltage present and output voltage prepared for use. Figure 17 shows the first revision of the power supply PCB. P1 is the input supplied by the lithium iron phosphate battery via a Deans Ultra Plug. P1 is the output of the power supply and uses a DC barrel jack to connect to the Beagle Bone.

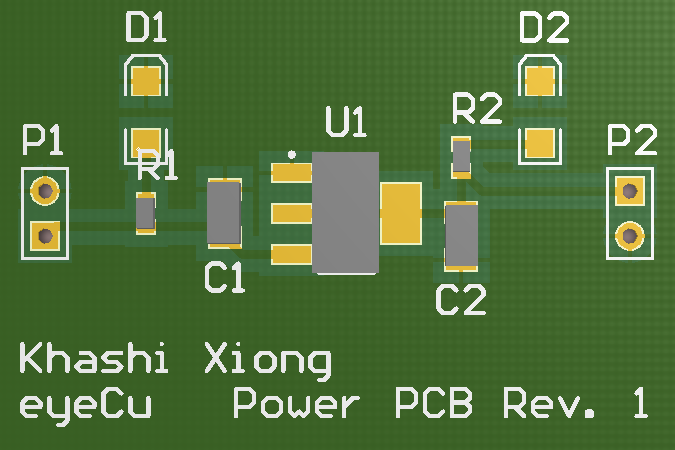


Figure 17: Power Supply PCB Revision 1

The second revision of the power supply utilizes the same circuit as the first revision power supply, but the second revision power supply also includes a recharging circuit capable of recharging the lithium iron phosphate battery. The charge controlling integrated circuit is a BQ2057 from Texas Instruments and utilizes a PNP transistor, decoupling capacitors, and a charging LED. When the battery is fully charged the charging LED turns off. Figure 18 shows the second revision power supply with battery recharging circuit.

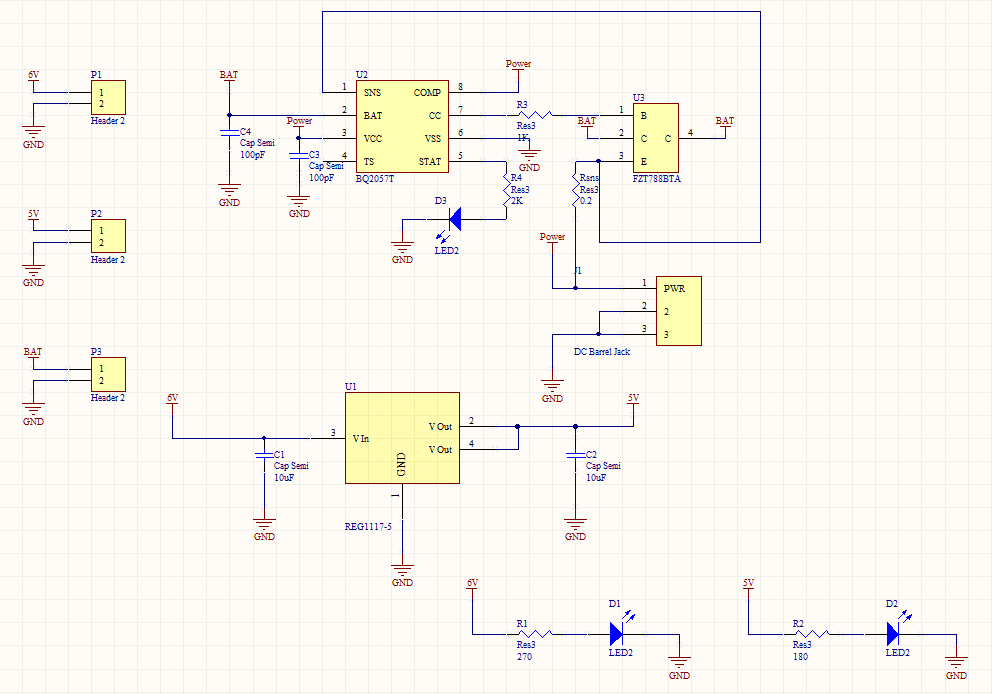


Figure 18: Power Schematic Revision 2

Figure 19 shows the second revision PCB. Power for the recharging circuit takes an input DC power supply through J1 and outputs through a Deans Ultra Plug via P3.

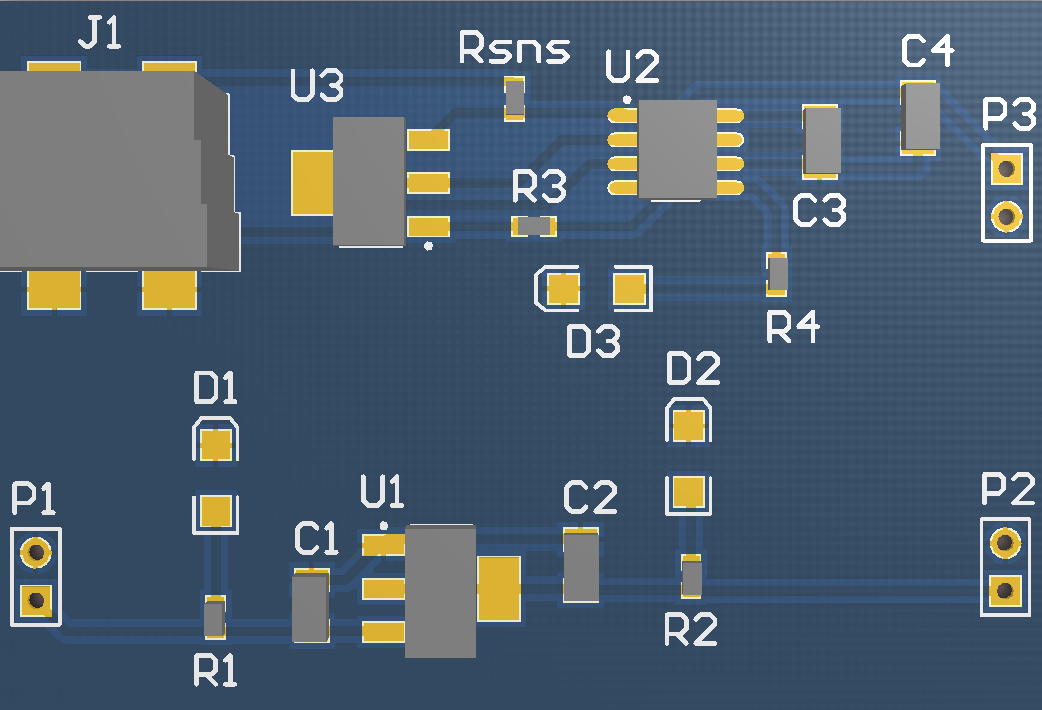


Figure 19: Power PCB Revision 2

Figure 20 shows the flowchart for the recharging circuit.

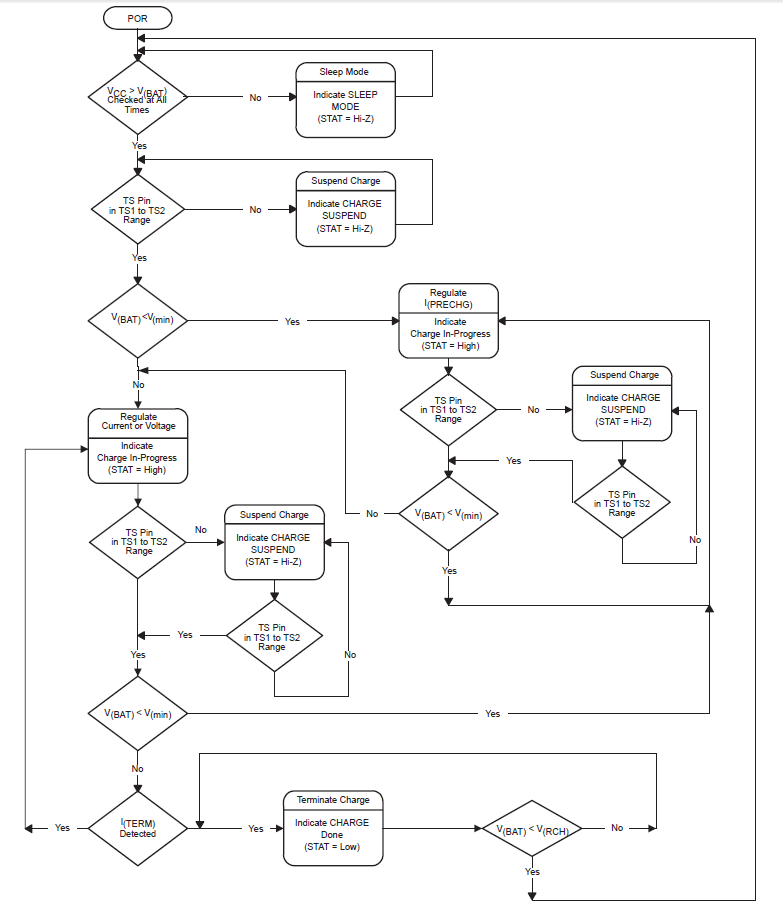


Figure 20: Recharging Circuit Flowchart

# Institutional Review Board (IRB):

# Bibliography:

|  |  |
| --- | --- |
| [1] | L. Dongheng, D. Winfield and D. Parkhurst, "Starburst: A hybrid algorithm for vide-based eye tracking combining feature-based and model-based approaches," *Computer Vision and Pattern Recognition - Workshops, 2005. CVPR Workshops. IEEE Computer Society Conference on,* p. 79, 2005. |
| [2] | E. Guestrin and E. M, "General theory of remote gaze estimation using pupil center and corneal reflections," *Biomedical Engineering, IEEE Transactions,* vol. 53, no. 6, pp. 1124-1133, 2006. |
| [3] | Texas Instruments Incorporated, "MSP430 Hardware Tools: User's Manual SLAU278H," 2009, Revised 2011. |
| [4] | Future Technology Devices International Ltd., "Application Note AN\_146 USB Hardware Design Guidelines for FTDI ICs," Future Technology Devices International Limited, 2010. |