NC STATE UNIVERSITY

# Team NoName Final Report

Firefighting Drone Challenge

William Simon
Scott Schachner
Jeremy Pattishal
Jesse Dannenberg

Date Submitted: 5/8/2015

## Introduction

The goal of this challenge is to develop the technologies required for drones to assist in firefighting operations. In the considered scenario there is a building on fire and many rooms are quickly filling up with smoke. Traditionally firefighters have to scan the building for victims, room by room, by crawling under the smoke while looking for potential hazards. With a drone capable of entering each room and able to scan a room accurately for a human body, the drone can remove the firefighter from a potentially dangerous situation.

*Image 1: The Final Drone*

# Hardware

## Airframe

Turnigy Talon Quadcopter frame

- This is a carbon quadcopter frame that provided the foundation of what would become the firefighting drone. This simple kit is equipped with four 10 inch hollow carbon rods, and two small carbon plates to act as the core structural components of the drone.
- The quadcopter kit also has a motor mount for each of the four carbon arms, and connection brackets for the center plates and carbon arms.

Additional Carbon Rods

- Additional 1 cm thick carbon rods were implemented as a means of adding rigidity to the airframe as well as providing mounting space for the 10 ultrasonic sensors.
- These carbon rods connect each of the four drone arms to one another meeting each at a 45 degree angle.

Aluminum Corner Brackets

- Four aluminum corner bracket were milled from solid pieces of metal. The holes were then drilled through each bracket; one hole large enough to fit the main arms of the drone and two smaller holes drilled at 45 degree angles until they intersect the larger hole in order to place the additional smaller carbon rods from one corner to another.

Additional Carbon Fiber Plates

- Additional 2 mm thick carbon fiber plates were used to make longer legs for the drone when the middle section is extended for fitting of all the extra components.
- The new legs were simply cut in a triangular shape for simplicity then attached in the same fashion as the original legs.

Brass Connection Rods

- Various length brass rods each with a female threaded end and a male threaded end were used extensively in the construction of the center column. The rods connect carbon plates and attach various electrical components to the main body of the airframe.

## Electronic Components

**Component specs and manuals are included under the Downloads section of the website.

BeagleBone Black Microcontroller (BBBK)

- The BBBK was chosen for its versatility in embedded Linux applications. It was used for the purposes of having efficient execution and control with software for near real-time performance.

KK2 Flight Controller and KK2 Programming Board

- The KK2 is the chosen flight controller for the copter due to its versatility, ease of use, and general familiarity. The programming board was used to change the flying mode of the KK2 board, as it has multicopter support options.

Turnigy 9X 9Ch Transmitter (Receiver included)

- This is the interface between pilot and drone. The receiver can receive up to 9 channels, however our drone only uses 5.

Ultrasonic Sensors

- The chosen sensors for obstacle avoidance and collision detection implementation for safe and easy flight mechanics. HC-SR04 was the model.

IMU - Razor 9DOF

- The IMU provides an accelerometer, gyroscope, and magnetometer for precise orientation measurements. This is useful for smoother control.

Logic Level Converter

- This is required to shift 5V signals to 3.3V signals and vice-versa, as the BBBK is only 3.3V tolerant.

Cameras

- A Logitech C920 webcam was used for high fidelity streaming purposes for FPV (first person view) remote piloting.
- A Seek Thermal camera was used for a video feed that would be unperturbed by smoke and locate victims and potential danger spots before firefighters enter the building.

Wifi Adapter

- Using an AP (access point) configurable Wi-Fi card, a connection was established between a client (most often a laptop) and the drone for display of video feeds.
- The AP was also used for programming the BBBK.

Motors and ESCs

- Brushless motors are used to drive the propellers. ESCs (electronic speed controllers) are fed PWM (pulse-width modulation) signals, which are translated into a DC source for the motors. Longer pulse within period = higher voltage = higher rpm.

Custom PCB

- Designed to simplify wiring to/from the BBBK. This is not a necessary component, it just made many lives easier.

Power Setup / Batteries

- One 4000mAh 4S LiPo battery became the final power source. The distribution method is described below in the electrical wiring section.
- We used a Turnigy HV SBEC 5A Switch Regulator to drop the voltage coming off the battery to 5 volts, which can be used to power the BBBK, ultrasonics, KK2 and receiver.

## Initial Design Plan

Initially, we planned on building a pentacopter as seen in Image 2. We progressed all the way to flight testing phase when we realized that the wasted thrust from the center prop overlapping with the smaller props as well as the central tower was so much that we were unable to achieve anymore flight than ground effect. We were forced to switch to the final version of the copter as seen above.

*Image 2: Initial Pentacopter Design*

## Final Airframe Design

The airframe is very simple. From the base of the Turnigy quadcopter, the landing gear legs are extended to 6 inches in order to compensate for the addition of parts in the center tower. The final design does not add any more plates for mounting any components in the tower, instead the use of brass connection rods to extend the bottom plate was employed. This gives ample space for wiring and a lightweight design. The hollow carbon fiber arms were also cut in half in order to fit through a doorway. Without cutting the arms, the craft would have never been able to fit through a typical door. The addition of carbon fiber rods for the mounting of ultrasonic sensors were also used. Aluminum holders were used to hold the rods in place; this also served to stiffen the airframe from twisting under full payload.

### Lessons Learned

After constructing the pentacopter we found that the current configuration created way too much drag for the craft to lift off the ground more than an inch or two. The center column of electronics and wires essentially blocked all of the thrust provided by the main lifting center propeller. In an effort to actually get the drone airborne the decision was made to convert it into a quadcopter configuration. Without extending the arms the four outer motors were replaced with larger (type of motor) motors with 12 inch propellers. Since the drone arms were too small for the larger propellers; we had to stagger the height of the motors by an inch which made it particularly hard to calibrate the drone for flight. In the future it would be best to set up the firefighting drone in a quadcopter configuration with every motor level with one another and without any propeller overlap. This could be achieved with higher power motors, notably the same ones used by Team Hindenburg. The motors and propeller setup would be able to fit in the same footprint as our modified drone without propeller overlap. The need for the shorter main arms is to fit within a door-frame.

# Electrical Wiring

## Introduction

Wiring was one of the biggest headaches of this project, and a part that underwent a number of adjustments. Wiring for ultrasonic sensors, a Razor IMU, and a Turnigy receiver will be covered below. Note that the specifics for the physical wiring are left somewhat up to the builder, as each group used a different method for attaching everything to the BBBK. Our group went through several failing methods before falling back on Team Hindenburg's PCB, which we modified for our purposes. If the reader decides to go with this method, note that we used the 1st version of the PCB, which had soldering points for logic level converters on the board. We did not use these converters, but the open wires allowed us to modify the board by jumping the sensor inputs to the pins we needed them on as opposed to Team Hindenburg's pins. This PCB may or may not be included in their report.

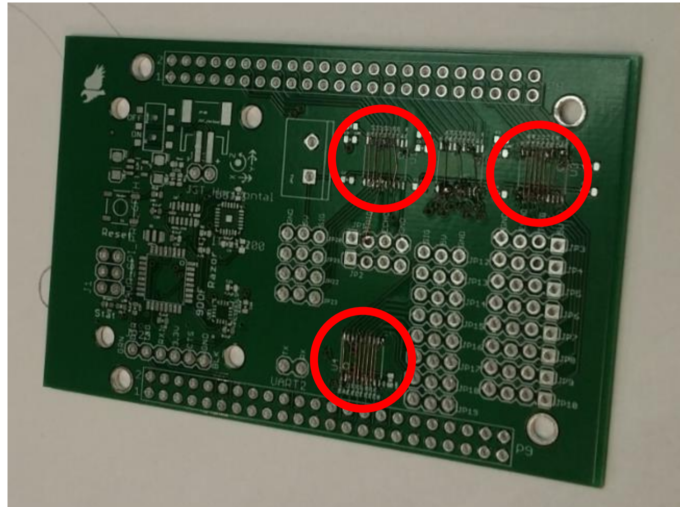

*Image 3: Hindenburg's PCB with Our Modifications*

# Sensor Wiring

## Introduction

Sensors integration was the most complicated part of the project, taking up the first half of the semester and going through several iterations as will be detailed below. The sensors included on the drone are 10 HC-SR04 ultrasonic sensors and 1 9DOF Razor IMU.

## HC-SR04 Sensor

The HC-SR04 is an ultrasonic sensor. It fires a burst of ultrasonic sound, which bounces off of objects and returns to the sensor. The delay from firing to return is measured to calculate the distance from the object. The sensor requires very specific timing to measure accurately, and it is difficult for a non-preemptive OS such as Arch Linux to measure accurately. Therefore, our team utilized one of the BBBK's Programmable Realtime Unit (PRU) CPUs to trigger and measure the sensors. The PRUs use ARM assembly code to operate, with only five nanoseconds per line of code, allowing very accurate measurements. The timing diagram for the HC-SR04 can be seen in Figure 1. One very important note about the HC-SR04 is that it is designed to

*Image 4: HC-SR04 Ultrasonic Sensor*

be used with 5V tolerant systems, which the BBBK is not. Therefore, special care must be taken when wiring the sensors to the board, which will be discussed in further detail later in this report.

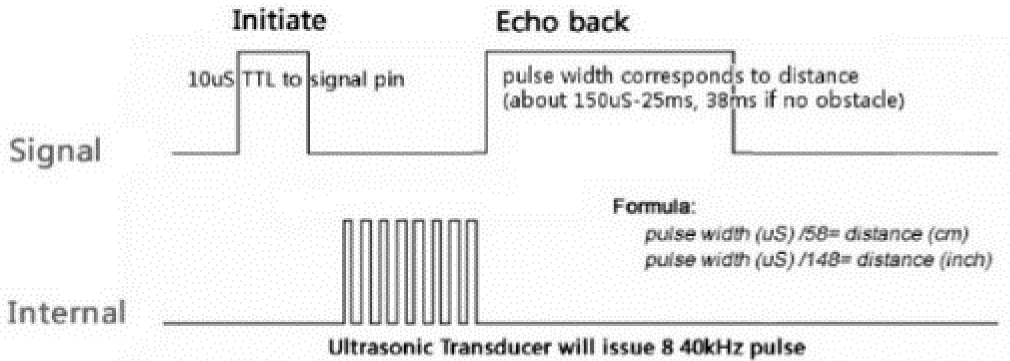


*Figure 1: HC-SR04 Timing Diagram*

## Initial Wiring Plan

Before testing began, we laid out where we thought we would like to wire our sensors to the BBBK, as in Figure 2.

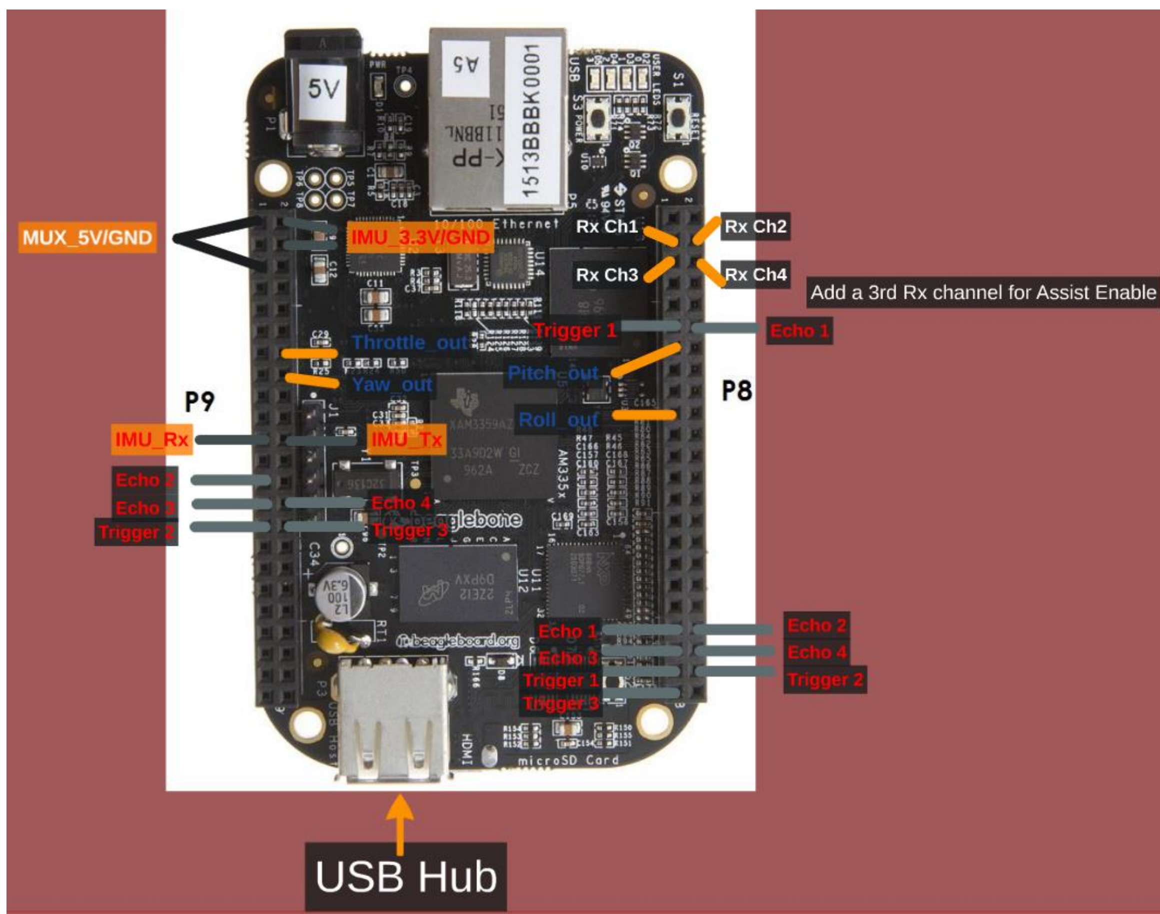


*Figure 2: Initial Wiring*

We initially planned to multiplex our echoes so as to reuse PRU pins, as a large portion of the pins were in use by the BBBK HDMI port. However, we quickly realized that this layout would cause unnecessary delays in measurement, since not all ultrasonics could be measured at once. This, coupled our learning how to disable the HDMI, led to a change in our wiring layout before we reached the testing phase.

## Special Note on Triggering

Each team struggled with the proper way to trigger the ultrasonic sensors. Multiple methods were tried, with different levels of success. Methods that were NOT successful are as follows. DO NOT try to trigger all sensors from one pin. Triggering the sensor draws approximately 2mA from the BBBK, and the BBBK pins are rated for 4mA or 6mA, depending on the pin. Firing all sensors from one pin will destroy the pin. DO NOT try to trigger the sensors too fast without a current limiting resistor. Another team had trouble with this, so I do not have the exact numbers or rate of fire, however our theory is that since the HC-SR04 is technically rated for a 5V trigger, and we are only providing 3.3V, the current draw is too high to sustain a fast fire rate. Our method of triggering, which works, is detailed later in this section.

## Final Wiring Layout

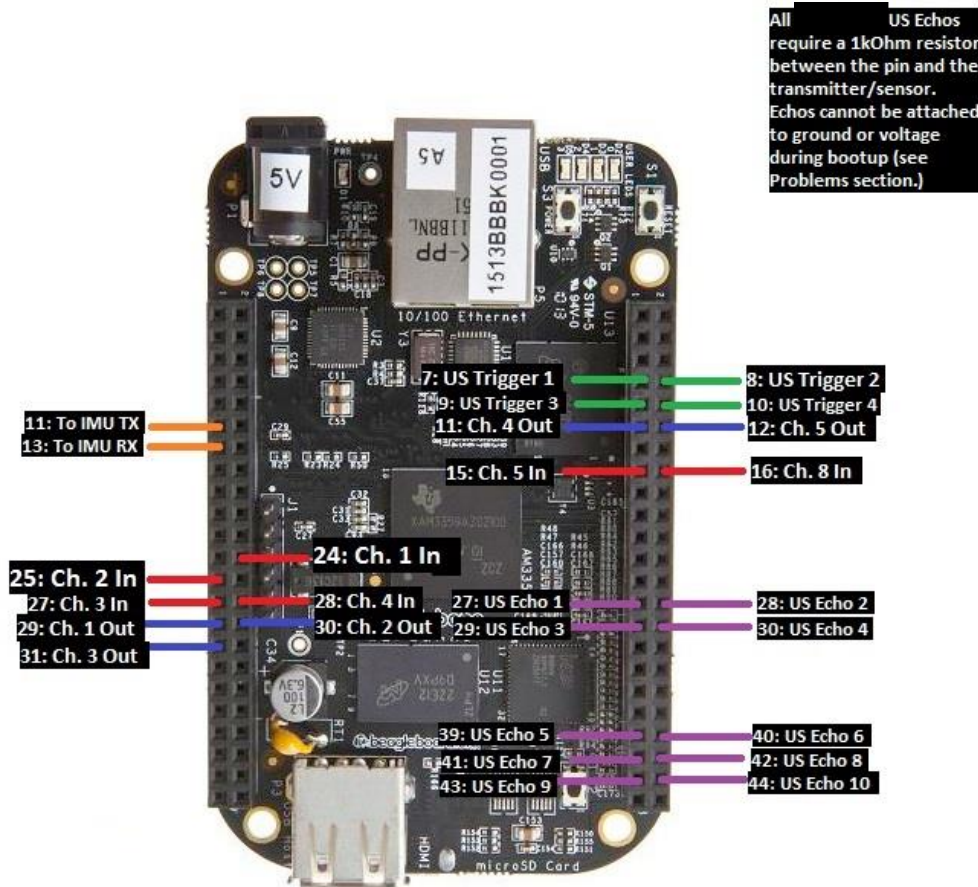


*Figure 3: Final BBBK Wiring*

Figure 3 is our final wiring layout. As can be seen, each sensor has a dedicated echo pin on the PRU. We have 4 Trigger pins, wired to standard GPIOs, which can also be controlled via PRU, albeit at a much slower rate. However, trigger signals are not required to be accurate, only greater than 10 microseconds, so there is no problem using GPIOs. Triggers 1 and 2 each trigger 3 sensors, while Triggers 3 and 4 trigger 2 each. This is to avoid the triggering problems described above. The pins used have been checked to verify that they can supply the 6mA needed to trigger the sensors. Note that while it does not matter which sensors are wired where for the echo pins, we used the wiring layout in Table 1 for the triggers. We chose this layout so that in the event that we had sensor interference from firing all sensors at once, we could fire one trigger at a time and have sensors facing different directions so as to have the least amount of interference as possible. For example, if Trigger 1 were fired, the north, southwest and southeast triggers are facing away from each other, and as such will not interfere.

| Trigger Pin | Sensor Orientation |
|---|---|
| 1 | North, SW, SE |
| 2 | South, NW, NE |
| 3 | East, West |
| 4 | Top, Bottom |

*Table 1: Triggering Layout*

## Wiring the Echoes

Wiring the echo pins properly is very important for not damaging the BBBK. The HC-SR04 is designed to be used with 5V tolerant boards. Since the BBBK pins are 3.3V tolerant, wiring a sensor directly to the board is a great way to fry it. Several methods were tested by different teams when wiring the sensors. Logic level converters were tested by a different team and ultimately thrown out, as the delay made the measurement very inaccurate. They switched to voltage dividers, but did not have time to test them extensively, so we cannot endorse that method either. Our team's initial solution that we carried through to the final design was using a simple 1kOhm current limiting resistor in series with the pin, as seen in Figure 4. We never had any problems with this layout as long as the system is powered up correctly. System powering will be discussed later in this report.

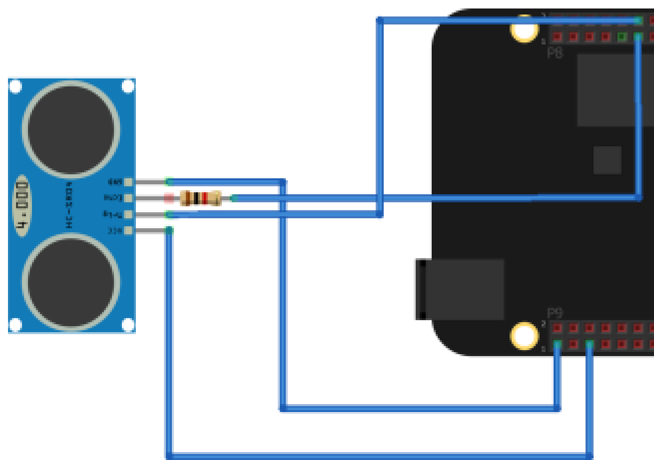


*Figure 4: HC-SR04 Echo Wiring*

# Transmitter/Receiver Wiring

## Introduction

While sensors integration is necessary to get distance and movement data to the BBBK, setting up the transmitter and receiver is equally important to allow the BBBK to read the user's inputs and modify them as necessary with the sensor data. On a standard drone, the user wires the receiver directly to the KK2 flight board. For our project however, we wired the receiver to the BBBK, which reads in the users inputs, modifies them as necessary given the sensor data, and outputs the modified signals to the KK2.

## Turnigy 9X

For this project, we used the Turnigy 9X 9 Channel transmitter/receiver combo. The receiver receives PPM signals from the transmitter, which it converts to PWM signals similar to what is seen in Figure 5. These signals are typically interpreted by the KK2, but we set up the BBBK's PRU 0 to measure the length of time each signal is high and output this number to shared ram, which can then be read and modified by the main program. The wiring layout can be seen in image 3 above.

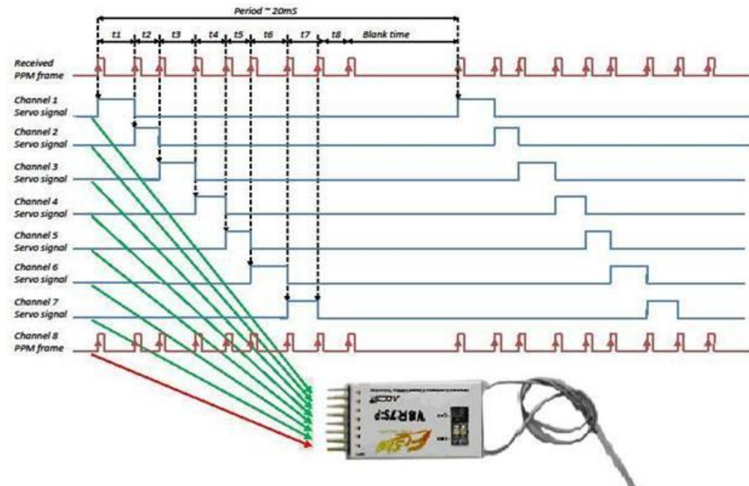


*Figure 5: PPM to PWM Reciever*

## Receiver Input Wiring

Unlike the ultrasonic sensors, the transmitter is 3.3V tolerant and can be attached directly to the BBBK. It must be powered by a 5V line, as well as grounded. System powering is covered later in this report.

## BBBK Output Wiring

PWM output pins are then wired to the KK2 Flight Board. Initially, we wired out KK2 directly to the BBBK. This may have been a fatal mistake however, as the KK2 is expecting a 5V input. Ultimately, we added a logic level converter in between the BBBK and the KK2 as seen in Figure 6. This worked out well in limiting the current draw on the BBBK.

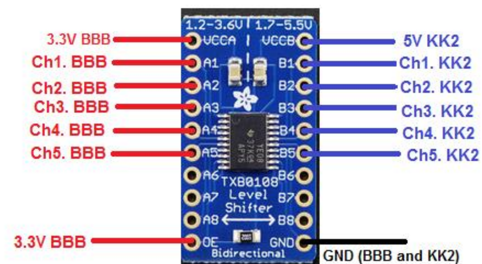


*Figure 6: Logic Level Converter Wiring*

# Control Coding

## Introduction

Control coding utilizes the ultrasonic sensors, the IMU and the receiver inputs to modify user input and send updated signals from the output pins to the KK2 control board. The objective of control coding is to use the IMU to prevent drift in from drafts, and use the ultrasonic sensors to prevent crashes. Our team only completed part of this goal, implementing control coding for 4 sensors, north, south, east and west. We are currently also reading data from the IMU, but not using it to modify the outputs. The code also uses the channel 8 input signal to switch between modified and unmodified signals. All program files can

be found in the Github linked in the Downloads section of our website. This Github also has a readme specifying the proper methods for getting the code up in running on the BBBK. Note that code has only been tested for Arch Linux ARM.

## PWM I/O Programming

The receiver inputs and BBBK outputs to the KK2 are both PWM signals as discussed in the previous section. These signals are nearly identical to the echo inputs from the ultrasonic sensors, and therefore are decoded and encoded on PRU 0. The PRU code is PWMInOut.p. The program uses a timer to measure how long each signal is high and saves this data into a shared memory that can be accessed by both PRUs and the primary CPU and main control program. The memory layout can be seen in Table 2. The modified memory locations contain the PWM signals that have been modified by control code. The software mux is also located in this program. When the program reads the value of channel 8, it checks to see if it is above or below the median time. If the signal is high for over the median time, the switch is high, assist is on, and the program will generate PWM output signals from the modified memory locations. Otherwise, the output signals are generated from the same memory locations that the input signals were saved to, essentially just passing them through without modifying them. If the main program crashes for any reason, the user can flip the assist switch to take direct control of the drone. This solution does not solve complete BBBK failure however, and should therefore be replaced with a hardware solution. More details on code functionality can be found in the annotations in the files themselves.

## Ultrasonic Sensor Coding

Ultrasonic sensor triggering and measuring is controlled entirely by PRU 1 on the BBBK. The PRU code is loaded via PruInit.py and sensor measurements taken by the PRU are read from memory in US.py. The PRU code itself is in Ultrasonic.p. There is a delay of .03 seconds before each trigger to allow the sensors to reset and the previous sound bursts to die away, resulting in a frame rate of approximately 33 measurements a second per sensor. This frame rate can be easily increased by decreasing the delay before the trigger, however we felt that this frame rate was more than enough for our goals while limiting the amount of sound interference and avoiding the triggering problem discussed above. The PRU stores the values of the ultrasonic measurements to a shared memory that can be accessed by both PRUs and the primary CPU and code. The memory layout can be seen Table 2.

| Signal | Offset from Shared Ram Start (0x10000) |
|---|---|
| Ch. 1 In | 0x10200 |
| Ch. 2 In | 0x10210 |
| Ch. 3 In | 0x10220 |
| Ch. 4 In | 0x10230 |
| Ch. 5 In | 0x10240 |
| Ch. 6 In | 0x10250 |
| Modified Ch 1. Out | 0x10200 |
| Modified Ch 2. Out | 0x10200 |
| Modified Ch 3. Out | 0x10200 |
| Modified Ch 4. Out | 0x10200 |
| Modified Ch 5. Out | 0x10200 |
| Echo 1 | 0x10000 |
| Echo 2 | 0x10010 |
| Echo 3 | 0x10020 |
| Echo 4 | 0x10030 |
| Echo 5 | 0x10040 |
| Echo 6 | 0x10050 |
| Echo 7 | 0x10060 |
| Echo 8 | 0x10070 |
| Echo 9 | 0x10080 |
| Echo 10 | 0x10090 |

*Table 2: Shared Memory Mapping*

More details on code functionality can be found in the annotations in the files themselves.

### Modifying the Output Signals

Control coding can be found in the PWM.py file. Our control coding is relatively simple, since we did not have a lot of time to work on it. We are currently only using the North, South, East and West ultrasonic sensors and a simple proportional algorithm to modify code. PWM.py contains annotations on the specifics of the control code.

## System Powering and Boot Up

### Practical Problems

Powering the BBBK and sensors systems can be tricky for a few reasons. Pins 27-46 are SYSBOOT pins, which means they are integral during the board boot process. If there is any signal over these pins, the BBBK will probably not boot properly. Also, supplying current to a pin before the BBBK is booted will most likely result in damage and destruction of the board. These problems make powering order very important on the drone. No team found one foolproof solution, however, we found a fairly reliable system.

### Switches

Our team used 3 switches to ensure proper boot order. Note in the figure below that switch 3 disconnects the sensors and receiver entirely from the system. This is because even a ground connection can mess with the SYSBOOT pins. Although the KK2 is not connected to SYSBOOT pins, it too requires a switch. If the KK2 does not receive a PWM signal at its inputs upon power up, it will enter a failed state and must be turned off and on again. Integrating a switch is much easier than manually disconnecting the power wire each time this must be done. Given more time, we would like to replace the switches with MOSFETS so they can be controlled from software.

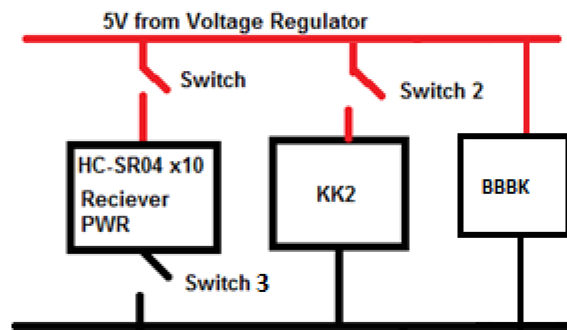

*Figure 7: Switch Wiring*

### Booting from MicroSD

An important aspect of the boot up sequence is boot order. Normally, the system first attempts to boot from eMMC, then from microSD card. However, if there is an error with the SYSBOOT pins, this boot order can be corrupted and the BBBK may hang. We therefore found it useful to boot only from microSD. This is accomplished by installing Arch Linux onto a microSD card and wiping the eMMC memory via the line

```
sudo dd if=/dev/zero of=/dev/mmcblk1 bs=1024 count=1024
```

This way, no matter what the boot sequence is, the BBBK will always fall back to the microSD. We found that this combined with the switch system provided reliable booting. Team Hindenburg claims to have found a more reliable method, so it may be useful to read their report on the subject.

# IMU Integration

## Introduction

During the planning phase of the project, we wanted a method of detecting and resisting drafts that could be caused by a fire inside a building. To do this, we would need a way to measure drift. The class settled on the 9DOF Razor IMU from Sparkfun for its ease of use.

## IMU Wiring

The IMU is one of the easiest components to wire to the BBBK. It is already 3.3V tolerant, so no special wiring is needed to integrate it. Simply power the board from the 3.3V rail, ground it, and attach the Tx and Rx pins as specified in Figure 3 above.

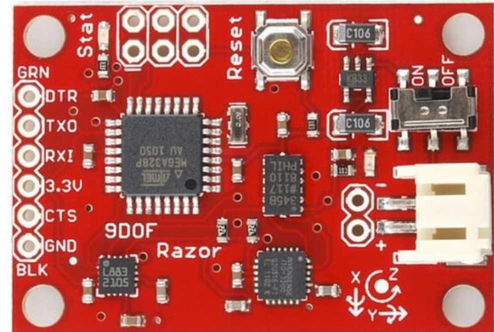


*Image 5: 9DOF Razor IMU*

## IMU Control Coding

Unfortunately, the IMU was never actually used for controls, as we had planned. Time was very limited and this was a low priority on our list. In terms of base functionality, it is unnecessary; however, it could add some very interesting capabilities to the drone with the included gyroscope, magnetometer, and accelerometer.

Although it wasn’t used, if wired IMU readings will be outputted to the GUI when the Main.py program in the Github is run. These signals are read in the imu.py file.

# Video Integration

## Introduction

This section will walk through the necessary steps to acquire a video stream from the Logitech C920 webcam and the Seek Thermal Camera over the Wi-Fi connection.

*In order for these to work, the devices must be plugged into the Beaglebone via USB and properly recognized. This can be verified by checking the output of the command: `# lsusb`

## Logitech Stream Setup

*Image 6: Logitech C920*

For the implementation of a video stream, we will be using a service known as MJPG-Streamer. This service will allow us to send the feed of the camera's frames to an IP address for anyone with access to the Wi-Fi network to see.

First, you'll need to download the MJPG-Streamer package along with its dependencies:

```
# pacman –S mjpg-streamer libjpeg imagemagick gcc libv4l-dev
```

To get all the necessary mjpg-streamer files, download the folder "mjpg-streamer-code-182.zip" from our <Downloads> page.

Using WinSCP, move the zip file you just downloaded into a folder of your creation that you will navigate to whenever you wish to start the streaming service. Next, using 7zip, extract the contents of the folder with the following commands:

```
# pacman –S p7zip
```

```
# 7za x mjpg-streamer-code-182.zip
```

If the file "videodev2.h" is not under /usr/include/linux/, run the following command:

```
# sudo ln –s /usr/include/linux/videodev2.h /usr/include/linux/videodev.h
```

Now that everything is installed, you can start the service simply by executing a command. The following command is what our team used for a 60fps 640x280 resolution stream over the IP of alarm. This command (as is) must be executed inside the directory named "mjpg-streamer" from the .zip file contents:

```
# ./mjpg_streamer -i "./input_uvc.so -r 640x280 –f 60" –o "./output_http.so –w ./www" &
```

The ampersand (&) tacked onto the end simply lets the process run unmonitored in the background. To be able to type commands after this one, hit ctrl+C. The stream is available in several different browser modes/windows for you to choose from at http://alarm:8080 when mjpg-streamer is running.

To stop the stream, the system must be powered off if the ampersand was used.

Other helpful documentation for mjpg-streamer: http://skillfulness.blogspot.com/2010/03/mjpg-streamer-documentation.html

## Seek Thermal Stream Setup

*Image 7: Seek Thermal Camera*

Get required packages:

```
# pacman –S python2 python2-pillow opencv python2-numpy python2-pyusb
```

Download the python file “processVideo.py” from our <Downloads> page and place into the directory of your choice.

Running this program from here will NOT work correctly because there is no configured display port for the program to show frames. Running the program using:

```
# python2 processVideo.py
```

Should return something like “Gtk Warning: cannot open display”. To fix this, the program needs to have a remote display to project the frames to, i.e. a server. If this is not the error you receive, it is most likely unable to compile due to a missing package. Make sure they line up with all the imported modules at the top of the python program.

In our setup, we use X11 forwarding to send the compressed frames over to the server for decompression, processing, and display to the user. This is completed by using an XWin Server as part of Cygwin/X.

To get XWin Server on your computer, follow the instructions at this link: http://x.cygwin.com/docs/ug/setup.html#setup-cygwin-x-installing

To open XWin Server, navigate to the root directory (folder = cygwin). Under this folder, go into “bin” and look for “run.exe”. If this doesn’t work, try typing “xwin” into your start menu search bar and see if it comes up there. Just a search for “xwin” on the correct drive should return the correct result.

Once you open up XWin Server, ssh into the Beaglebone:

```
$ ssh –XY root@alarm
```

The –X enables X11 forwarding on the server's side. The –Y forces authentication credentials. On the Beaglebone, X11 forwarding must be enabled in a different way, although the change will last until you change it back. To do this, using WinSCP, navigate to `/etc/ssh` and open the "`sshd_config`" file in a text editor (Notepad++ is highly recommended). Once opened, scroll down until you see the lines below (line 102):

```
#X11Forwarding no
#X11DisplayOffset 10
#X11UseLocalhost no
```

Change these lines to:

```
X11Forwarding yes
#X11DisplayOffset 10
X11UseLocalhost no
```

Additional packages are required for X11 forwarding:

```
# pacman -S xorg xauth
```

The "`xauth`" package is absolutely necessary for this method of streaming, otherwise the server you're using will see the client (Beaglebone) as restricted and will be unable to capture frames from the camera.

Now you finally have everything you need to correctly run the program. This will open up a display on the server's machine (your laptop, in this case) showing a video feed from the thermal camera. To run the program, as before, navigate to the directory where processVideo.py is stored and type:

```
# python2 processVideo.py
```

Typing ctrl+C will close the program.

From boot, once in the correct directory, the sequence for viewing the stream will be:

1) Start XWin Server

2) `$ ssh -XY root@alarm`

3) `# python2 processVideo.py`

*Image 8: Seek Thermal Visual Output with Bounding*

# Wi-Fi Configuration

NOTICE: This guide assumes the use of a TP-LINK Wifi Adapter, model no. TL-WN7200ND. If you do not have the exact model, this exact same process outlined below should work as long as your adapter has a Ralink chipset (prefix = RT) and has access point capability. It also assumes **Arch Linux ARM** is the distro of Linux in use. These instructions, and indeed all of the instructions on the Github are only tested for Arch.

## Before Wireless Capability

To connect your Beaglebone to the internet (so that you can download packages), follow the steps below:

1) Plug the Beaglebone into an internet capable router via Ethernet cable
2) Plug the 5V power adapter into the Beaglebone
3) Check the list of devices connected to your network. If you're using a NETGEAR router, this can be done by visiting routerlogin.net. On this list, you'll see a device labeled as "ALARM." This is the Beaglebone! It should have an IP address next to it that will be something like 198.162...
4) In putty (or something like it) use the IP address (or "alarm" should work to, in the same field) to establish a connection.
5) `login as: root`
6) `password for root: root`
7) Good to go!

User Guide by TP-Link:

http://www.tp-link.com/resources/document/TL-WN7200ND_V1_User_Guide_1910010859.pdf

## Driver Details

When you first connect the wifi adapter via the micro-USB cable, the LED will most likely begin to blink slowly. If this occurs, it means that the driver has been installed correctly and the device is recognized. If this happens, you're good to go! The driver required for this adapter is native to the Arch distribution (distro) our team is using, so this *should* be plug and play. If you're using our setup, it will be.

*Image 9: TP-Link Wi-Fi Adapter*

To check driver status, typing

```
# dmesg | grep usbcore
```

Should return something like "`usbcore: registered new interface driver rtl8187`" if it worked correctly.

OPTIONAL: Using the Adapter as a Router for an Internet Connection when in SSH

This mode may be helpful at first when you don't have to worry about portability. If you don't like working while being connected to a router, skip this section. The usefulness of this method as opposed

to just using the AP (Access Point) comes from the fact that you don't have to enable internet sharing amongst your ports.

With this setup, you plug the Beaglebone's Ethernet port into a router, and the adapter into the USB port of the Beaglebone. This will enable you to connect to your Beaglebone wirelessly through the preexisting wifi network you have setup for that router (using the same SSID and password).

This configuration was set up using steps entirely from the ArchWiki at this link:

https://wiki.archlinux.org/index.php/Wireless_network_configuration#Manual_setup

Get required packages:

```
# pacman –S iw wireless_tools wpa_supplicant
```

Before continuing to the next step, check the name of the wifi interface. By default, it will most likely be wlan0, which is used in the commands below. To check this interface, type the command below and read the output. Interface names *usually* start with a 'w'.

```
# ip link
```

Set the adapter to ad-hoc mode so the adapter can share a peer-to-peer connection with your router. Then, type the following commands to set the interface up and match it to your personal network:
**Notice the subtle difference between the '`1(one)`' and the '`l(L)`', the top portion of the one is bent downward.

```
# iw dev wlan0 set type ibss
```

```
# ip link set wlan0 up
```

```
# wpa_supplicant –D nl80211,wext –I wlan0 –c <(wpa_passphrase
"Your_SSID" "Your_key") –B
```

"`–B`" allows the process to run in the background so you can do other things while connected to the internet.

```
# dhcpcd wlan0
```

The LED on the adapter should now be blinking quickly to indicate data transmission/reception.

```
# iw dev wlan0 link
```

The output of this will show you whether the link is actually up or not.

## How to Setup a Customizable Access Point

Get required packages:

```
# pacman –S hostapd dnsmasq iptables brctl dhclient
```

A detailed guide for setup and configuration, as well as customization options, from installation, can all be found here:

https://github.com/oblique/create_ap

Our method utilized the WPA + WPA2 passphrase:

```
# create_ap wlan0 eth0 MyAccessPoint MyPassPhrase
```

"eth0" is there to set where the wlan0 interface will get internet connectivity, if one is available. Typing the line above will run the script. The script will not run again when you restart unless you type that line again. In order to set it to start at boot, type the following:

```
# systemctl enable create_ap
```

To configure what AP is being created by the script running on boot, using a program such as WinSCP, edit the file "create_ap.service" found in the bin under ~/root and change the "ExecStart" line. The command already there should look similar to the examples on the github page.

Your AP will only allow you to connect to the Beaglebone wirelessly and independent of any internet connection or router. To connect your AP to the internet, the Beaglebone will need to be connected via the Ethernet port to an internet-connected device with internet sharing enabled (laptop). You do not need an internet connection to SSH to the Beaglebone.

To enable Ethernet port internet sharing on a Windows computer, simply right-click the wifi icon in the bottom right corner and open the "Network and Sharing Center." On the left side of that window, select to "Change adapter settings." You should be in the Network Connections window now, under Control Panel\Network and Internet\Network Connections. Next, right-click on the adapter built into your laptop with the wifi connection on it and open "Properties." Navigate to the "Sharing" tab and check the box next to enable Ethernet connection sharing. Your AP should now have internet access while your Ethernet cable is connected and you will be free to download packages.

# Conclusions

## Strong Points of this Design

- Our drone had functional control coding running through the PRUs. 4 sensors were implemented, with the modular nature of the code making it very easy to add more. All 10 sensors were at least tested to insure that they were firing and reading correctly and simultaneously. Only one other team had control coding in a somewhat usable state by the end of the semester, and ours was the only drone that had control coding solely through the BBBK.
- Our drone had excellent Wi-Fi enabled video with a range of over 100 feet and through multiple stone walls at 60 frames a second with almost no latency. Ultimately we were not able to test the functionality to its limits because we implemented the code so late in the semester; however, no other team came close to touching our capabilities via Wi-Fi (we are ignoring the illegal analog radio system used.)
- Our drone had usable infrared, capable of detecting and bounding heat sources such as humans and bowls of hot water. No other team as far as I know of had infrared to a usable state.

## Points of Improvement

- Easily our biggest problem while working on this drone was our team's lack of experience in drone design. We simply did not know how to design an effective and sturdy drone, as well as tune it properly. In the future we would like to be able to consult or work directly with someone knowledgeable in the field.