Research and Development of a Low-Cost, Highly-Accessible Open-Source Telepresence System

Prepared by:
Remington Stanley Bullis

Faculty Advisors:
Dr. Randy Hoover
Associate Professor, Department of Electrical and Computer Engineering

Dr. Thomas Montoya
REU Site Director, Department of Electrical and Computer Engineering

Dr. Alfred Boysen
Professor, Department of Humanities

Program Information:
National Science Foundation
Grant NSF (EEC-1359476)

Research Experience for Undergraduates
Summer 2016

South Dakota School of Mines and Technology
501 E St. Joseph Street
Rapid City, SD 57701

Table of Contents

Abstract

The widespread adoption of smartphones availability of low-cost headsets and controllers provides and unprecedented access to virtual reality robotics, something that has yet to be leveraged economically in either the maker community or commercial robotics industry. This widespread access to virtual reality, the ever-expanding capabilities of open-source software, and the increasing role telepresence systems play in businesses and the military might have a common junction point in an open-source virtual-reality telepresence solution. This research aims to find out if such an inexpensive virtual reality telepresence system can be built on open-source software and popular, community-supported hardware. The process followed involved defining from a high level a potential manifestation of such a system: a “client” system composed of a smartphone, inexpensive virtual-reality headset, and Bluetooth controller would communicate over the Internet via a dedicated app to a “remote” side, composed of a land-mobile robot (driven by the Bluetooth controller) onto which is mounted a head-tracking stereo-vision gimbal. After a formal definition is selected, a more specific series of the subsystems are defined and individual requirements elucidated, culminating in a list of “best selection” hardware and software for the construction of the system. A proof-of-concept prototype system is constructed and discussion about the limitations and merits of this approach is presented. It is concluded that a low-cost, virtual-reality telepresence system based on open-source software and community-supported hardware is feasible not only in theory, but also practically. Such a system fills a gap in the open-source robotics community and has grand potential commercially.

1. Introduction

1.1 Background

To fully understand the vision of this research some background information is needed, specifically in the areas of virtual reality, telepresence systems, and the open-source software movement.

Virtual Reality

Since the 1980s virtual reality, the idea that we can transport ourselves to another, computer-generated “reality” through the use of technology has been promised as only a few years away, waiting on only the next technological baby step. Only in the last few years, however, has convincing virtual reality been a technical possibility. In light of this several companies are developing commercial products limited to use with high-end gaming PCs and some consoles. The largest potential user base for VR, surprisingly, is actually the average smartphone user. Modern smartphones are flush with sensors such as magnetometers, gyroscopes, and accelerometers, all of which can be used to provide a convincing virtual reality illusion. Google has proven the viability of a smartphone-based VR system with its release of low-cost headset called the Google Cardboard, into which a user slips their phone, opens up the Google Cardboard app, selects an experience, and is then transported to a convincing virtual reality illusion.

Telepresence

A telepresence system allows a user to have presence of some form in a remote location. A prime example of simple telepresence is video conferencing: people are able to “meet” and collaborate via video from anywhere in the world. Sometimes telepresence systems also allow a user to physically affect the remote environment: surgeons use remotely-operated robotic systems to perform surgery and save lives over vast distances. In militaries around the world ordnance disposal robots are used to safely defuse explosive devices, minimizing risk to human life.

Open Source Software

In the late 1990s the open-source software movement grew out of the free-software movement, and holds the core ideas that open-sourced software is free for anyone to use, dissect, distribute, and modify. Open-source software is developed by a community of (usually) unpaid developers collaboratively in an open format, and can result in very high quality code providing all the functionality of commercial software. This ideology has fundamentally changed the world of software development, providing tools for both common and highly-specific tasks as well as opening the doors for anyone to make and distribute their own software free of charge. The open-source movement has shifted out of the purely digital and into the physical as well, with many hobby 3D printers, milling machines, microcontroller boards, and single-board computers being produced with the same mindset. (Open Source Initiative)

1.2 Objectives

The vision of this project is quite exciting, and best showcased with a potential use case: A prospective college student would like to tour a far-off university but is unable to afford the travel for a shoes-in-the-quad tour. Instead of travelling, he purchases a cheap Bluetooth controller and VR headset for his smartphone off the Internet for less than $20. Upon receipt, he connects the controller to his phone, slips his phone into the headset, and starts the university's "virtual reality" tour app. Once inside the app the user is able to connect to and control a robot on the university's campus. This robot is mobile, and has a gimballed camera system on it allowing the user to turn his head (with phone in headset) with the robot's cameras turn to match. Real-time head tracking provides a realistic sense of presence, and remote control via the user's Bluetooth controller provides the capacity to tour a campus or department just like any other student.

The above system is highly idealized, and potentially unobtainable. This research set out to investigate the feasibility of a system like described above constructed within realm of the hobby roboticist. Is it technically possible to build a low-cost, virtual-reality telepresence system built around community-supported hardware, open-source software, and a smartphone? If so, what would such a system look like? From these questions we set the objectives of this research.

Distilled down, there are two main objectives. The first is to investigate the feasibility of a virtual-reality telepresence system built around community-supported hardware, open-source software, and a smartphone. The second is to produce a prototype of this potential system to vet the functionality and practicality of all the involved parts.

2. Broader Impact

The impact of a system such as this would take two forms: one in the maker community and the other in practical use by educational systems and other institutions. Community-produced robotic designs, software, and hardware are a foundational pillar of the maker community. But while full-faceted plans for many complex systems like 3D printers and hexapod robots are available, nobody has yet to combine virtual reality and Internet-connected robots within the maker sphere. This research can provide a data point within this increasingly-potent development climate.

Institutions, specifically those dealing with education, and some businesses could find tremendous value in a low-cost telepresence solution. University campus tours almost universally require a trip to campus. For many prospective students these “boots on the ground” tours necessitate a multiple-day trip and not-insignificant financial burden. Some universities are now adopting “virtual tours,” allowing prospective students to view 360 degree imagery of set points on campus through their web browsers, or even virtual reality headsets through dedicated apps. While much more accessible than an in-person tour, virtual tours cannot put their users physically in the environment they are hoping to experience. This system could provide an excellent middleground: users would be able to physically explore (and potentially interact with) a college campus in real time, all with low cost to the university and marginal hardware investment by the touree. Similarly, the technology could be adapted for use in remote museum tours, quasi on-site troubleshooting or inspection, or even in the real estate industry to allow buyers a remote “tour” of a prospective home.

3. Procedure

The procedure followed for this particular project naturally settled into three distinct, yet interconnected purviews: hardware, software, and the construction of a physical prototype. The requirements of each subsystem in those sections will be dictated by the overarching design topology of the proposed system, something that must be decided upon before any hardware or software can be pinned down.

3.1 Whole-System Structure

A formal definition of the telepresence system and its functional parts is needed to start directing investigation and hardware selection. The proposed system would have two separate systems, connected over the Internet. A client/user side which includes the user’s phone in a virtual-reality headset, a Bluetooth controller of some kind connected to said phone, and an app running that connects to the “remote” side via IP. The remote side would consist of a small land-mobile robot with a stereo-vision camera system mounted on top. The cameras would track the user’s head movement through the app running on the user’s phone, streaming stereo video back in real time to simulate the user being in the remote location. The robot’s movement would be controlled by the user’s Bluetooth controller, also via the app. In this way the user could control a remote’s movement and “look around”, providing a passable illusion of presence in a remote location.

Ideally this system would be assembled with primary emphasis on economic accessibility. Since the software to be used is open-source and free to use, any cost associated with implementing such a system would come from hardware: the headset and controller from the client side, and the robot from the remote side. The majority of the cost would be levied by the remote robot, which contains a moving base, cameras, and some systems to connect to the Internet and control camera movement. In selecting hardware focus must be put on minimizing expense yet maintaining essential functionality.

3.2 Remote Robot Design Topology

One of the key challenges in building a robot is selecting how every part of the robot will interact. What controls what? For simple robots such as line followers or maze solvers, a microcontroller can handle all tasks required quite satisfactorily. For complex robots a computer is required to handle difficult calculations and process incoming data, but these computers can't usually interface with hardware directly. Instead, they rely on special hardware controllers communicating via serial or USB to handle hardware interfacing. Such controllers are generally quite expensive and perform one task: a servo controller controls servos, a motor controller spins wheels, and a digital output bank blinks lights.

The robot in this telepresence system must stream video in real time and control various motors to drive or track the user's head motion. The second is clearly within the purview and capacities of microcontroller, but the first is much more of a challenge for low-cost MCUs. A small, efficient single board computer would be much better suited to the task and provide easier network interfacing than a microcontroller. However, any computer running an operating system built upon a non-realtime kernel will struggle with real-time tasks, such as generating a PWM waveform or handling external interrupts. These things are what microcontrollers were designed for!

Ideally this remote robot would have the benefits of both a MCU and a computer. A master-slave relationship would provide this, relegating a MCU to hardware interaction and a single-board computer handling all software and networking duties. A communication link would allow the single-board computer to communicate with the MCU, simply issuing commands and letting the MCU handle hardware interaction.

This "distributed control" topology allows the robot to leverage the strengths of both computing elements. It also allows for platform agnosticism: code can be updated on either side of the communication link separately and each side can be modified without affecting the other. This allows for a Raspberry Pi to be swapped out for an Odroid C2 running the Raspberry Pi's software with no update to the MCU code. Or, conversely, the Raspberry Pi can be swapped to a larger, more capable robot with no adjustments to code provided the communication protocol with the new platform is identical. Because of these reasons it was decided to have a microcontroller handle the hardware bits of the remote robot, and a single-board computer to control the MCU via serial and handle video streaming.

3.3 Software Analysis and Selection Criteria

The open-source community is flush with extremely specific software for *most* purposes imaginable. Given the entirety of software development from the past twenty-five years to sift through, a set of baseline requirements for software used in this project were outlined.

Overarching software requirements:

- The software's source code must be open to the public and licensed under some open-source license
- Software must be relatively modern and undergoing usage currently
- Software must have a community to provide support

Compounded with the above criteria, each software subsystem is subject to its own additional requirements. They are as follows:

Video Streaming:

- Minimal latency
- Variety of streaming format options
- Must provide a solution for receiving the stream on a mobile device

Mobile Platform and Development Tool:

- Widely used and easily accessible by the majority of college-aged individuals

- Low barrier to entry for software development
- Development platform should have a large, mature community and robust support

Embedded Development Toolchain:

- Simple, easy-to-use toolchain for writing, compiling, and flashing code to MCUs
- Support for feature expansion

Streaming Platform Operating System:

- Lightweight to allow for minimal operating overhead
- Minimal configuration

3.4 Hardware Market Audit and Selection Criteria

The international market for hobby-level robotics is massive and can provide a product for any niche. In order to find the best solution for this project, a series of baseline hardware requirements were set in place to narrow down the playing field.

Overarching hardware requirements:

- Hardware must be inexpensive
- Hardware must be widely available
- Hardware (or included technology) must be community supported
- Hardware must be reliable
- If 3D printed, parts should be printable on a small (14x10x10cm) print bed

If the above weren't stringent enough, each hardware subsystem is privy to its own additional restrictions:

Remote Robot Chassis:

- Provide simple locomotion
- Must have an integrated power system
    - Battery, regulators, harness, switch
- Large platform for mounting accessories

Head Tracking System:

- Provide pan, roll, and tilt within the range of normal human head movement
- Mount cameras a natural eye span apart
- Cameras must be low-cost and easy to interface

Remote Control and Headset:

- Inexpensive
- Controller able to connect to modern smartphones

Hardware Controller:

- Provide simple hookup for power and devices
- Variety of communication protocols
- Wide variety of available accessories to expand the robot's capabilities if desired

Streaming Platform:

- Low power consumption
- Simple network connectivity
- Enough processing power to stream VR compatible video at 30fps

3.5 Prototype Construction Procedure

Assembly of the prototype was aimed to be achievable with only hand tools and a supply of readily-available fasteners such as M3 screws and zip ties. Printed parts were to be constructed on a heavily-modified PrinterBot Simple 1401 3D printer, and designed using the OnShape browser-based CAD suite. OnShape provides Git-like version tracking, such that the evolution and refinement of a part over time is documented and any revision is available publicly online.

4. Results

The results of this research will be analyzed with a per-subsystem approach similar to that seen in the above section.

4.1 Hardware Selections

Remote Robot Chassis Selected: RobotGeek Geekbot Core

Purchasing individual parts, constructing a custom frame, and wiring a custom power system was decided against in the interest of keeping costs low (multiple suppliers means multiple shipping charges), and minimizing the time input to get the robot rolling. Many hobby robotics/electronics companies provide basic, and economical, “rolling chassis” kits online. Surveying the websites of Pololu Robotics, RobotGeek, Sparkfun Electronics, and Trossen Robotics reveals a bevy of chassis available, but only Trossen Robotics offers a kit meeting the requirements laid out in the prior section.

The Trossen Robotics Geekbot Core Kit provides a smart 4400 mAh LiPo battery (handles balancing, over/undervolting internally) and requisite charger, a wiring harness with power switch and barrel jack, two continuous-rotation servo motors and accompanying wheels, and an open, unobstructed build platform 10.5” in diameter. Provision of a self-babysitting battery keeps a separate battery charger out of the picture, minimizing cost and reducing the risk of dangerous battery mishandling. The servo motors used to locomote are simpler to drive than a direct-drive DC motor system and allow for anything with a standard hobby servo interface to control the chassis. This kit is not only economical, but also appears to be the best selection available for this use case.

Head Tracking System: Custom Gimbal + USB Webcams + Google Cardboard

In order to provide a believable virtual reality viewing system we need two things: a range of motion to mimic the human head's, and cameras set roughly an eyewidth apart. The United States Army was kind enough to survey its soldiers and found their mean interpupillary distance to be 63.5mm apart; for ease of production an interpupillary distance of 65mm was used. Providing stereo cameras with a range of motion similar to that of the human head requires three degrees of freedom: pan, for the left-right looking motion; tilt, for the up-down looking motion; and roll, for accurate reproduction of the human head's tendency to "twist" when at the edges of motion range.

Since machine time, material costs, and high-precision actuators are prohibitively expensive for low-cost projects such as this, it was decided to design a gimbal providing pan, roll, and tilt using standard hobby servos that could be 3D printed on most low-cost 3D printers. The printed design seen below gives a full 180° of pan, roll, and tilt to two Raspberry Pi cameras, and is easily extendible to any other desired camera configuration. The servos used are of a Dynamixel form factor and were supplied by RobotGeeks. Drawings of the gimbal parts can be found in Appendix A.

The cameras selected for use in this design are left intentionally vague. USB webcams were selected over options such as Raspberry Pi Cameras or Firefly MVs, as they are ubiquitous, cheap, and supported across most hardware platforms sporting a USB port. In testing a Logitech CM525 was used, though any modern USB webcam should provide indistinguishable functionality. Naturally the gimbal design will have to be modified to accompany different webcam's physical dimensions, but a universal stereo webcam gimbal is not the focus of this research.

Remote Control and Headset Selected: Budget BT Controller + Google Cardboard

The market for smartphone game controllers has seen a marked boom in recent years. Because of this, a plethora of cheap controllers, usually connecting to the target device via Bluetooth, have flooded the market. With the challenge of connecting a controller to a smartphone strapped to one's head, Bluetooth provides an excellent and platform-agnostic solution. Any controller with at least one analog stick should be sufficient, and many Bluetooth controllers meeting this requirement are available for less than $10 online.

There are a variety of virtual-reality focused headsets available for use with smartphones on the market. Samsung, Zeiss, and even Mattel have feature-filled plastic headsets available, but these are quite expensive in comparison to the new breed of headsets: foldable cardboard and plastic lenses. Google was the first to introduce such a headset, and other manufacturers were quick to adapt and build upon the idea. Now these headsets can be purchased pre-cut for less than $10, and provide all the functionality needed for this telepresence system. (Metz 2015)

Hardware Controller Selected: Arduino Uno

Industry's increased dependency on microcontrollers for even simple tasks has introduced the average electronics enthusiast to more MCU options than ever before. However few of the options from AVR, Microchip, STM, and others fit the relatively simple requirements of this project, enjoy a large community of users, feature a free and robust development/programming toolchain, and provide expandability like the AVR-based Arduino microcontroller boards. These boards provide RS232 serial, I2C, SPI, digital and analog input/output, and a wide variety of interfaceable hardware. As five servos are all that must be driven in the current design, an Arduino Uno is more than enough microcontroller for the job. More importantly, this platform is the go-to for hobbyists and artists in need of microcontroller capabilities. As such the Arduino community provides information ranging in scope from first-run tutorials to in-depth analysis of silicon-level code optimizations; this depth of knowledge will be of immediate use when code inevitably stops flashing correctly or weird compiler errors crop up.

Streaming Platform Selected: Raspberry Pi 3

The single-board computer industry has exploded in recent years and the hardware available has seen an exponential increase in features. The Raspberry Pi has been and continues to be the most popular single board computer on the market, and most definitely has the largest user base and community support. The main competitor and only other comparably-loved SBC available is the Odroid C2. Analyzing the hardware specifications listed in Table 1 clearly shows the Odroid C2 winning the horsepower war with twice as much RAM, a faster CPU, and a more capable GPU. However, for the scope of this project the maximum capability is not needed; ease of use trumps maximal performance. Because the Raspberry Pi offers a more robust and developed community of users and information, it was chosen over the Odroid C2.

Another significant factor in the choice to use the Raspberry Pi 3 is its inclusion of onboard WiFi. The Odroid C2 does not provide in-built WLAN capabilities, and necessitates the use of a USB dongle to gain wireless connectivity. Such a dongle would be yet another piece of hardware to purchase, and another thing to potentially break or lack driver support. The Raspberry Pi provides a simple connectivity solution, greatly simplifying wireless operation.

Table 1

Raspberry Pi 3 and Odroid C2 Hardware Comparison

	Raspberry Pi 3	**Odroid C2**
CPU	4 x ARM Cortex A53 @ 1.2GHz	4 x ARM Cortex A53 @ 2.0GHz
GPU	Broadcom VideoCore IV	Mali 450
RAM	1 GB DDR2	2 GB DDR3
Networking	10/100 Ethernet, 2.4GHz 802.lln Wireless, Bluetooth 4.1 Classic/LE	Gigabit Ethernet
Storage	microSD	microSD, eMMC
Connectivity	40-pin GPIO, HDMI, 3.5mm audio/video, 4 x USB2.0, Ethernet, CSI, DSI	40-pin GPIO, Ethernet, HDMI, 4 x USB2.0 (1 USB OTG), IR receiver

4.2 Software Selections

Video Streaming Software Selected: Gstreamer

Initial research into the topic of real-time video streaming revealed that a UDP/IP based, minimally encoded video stream would provide the lowest latency from frame capture to display on the receiving end and (usually) the lowest CPU load on the transmitting device. To choose which piece of software would be most optimal a series of latency tests were performed using the most popular webcam/video solutions currently available across many Linux distributions: Motion, FFMpeg, and GStreamer. These tests were run a video resolution of 640x480 pixels, 30 frames per second, MJPEG encoded with a Logitech C525 webcam as the input device streamed from a Raspberry Pi 3 to a laptop via a wifi connection. The results of this test can be seen in Table 2.

Gstreamer provided the lowest latency streaming by a wide margin, coming in at 105 milliseconds. Clearly, in the interest of minimal latency GStreamer is the solution of choice. It also provides options for encoding video in multiple codecs, some with GPU-encoding support, including h.264 for use with Real Time Streaming Protocol-enabled connections to provide minimal latency with high-definition video. ("Overview of available plugins" 2016)

Table 2

Streaming Software Latency Test Results (Average of 3 Frames)

FFMPEG	**Motion**	**GStreamer**
2380ms	742ms	105ms

Mobile Platform and Development Tool Chosen: Android + Android Studio

In the context of this project we need three things from a mobile operating system: wide market share, an easy-to-use solution for producing the virtual-reality app, and support for receiving the video stream sent from our transmitting robot. Currently there are three "main" players in the mobile operating system market: Android, iOS, and Windows Phone. Of these Android has by a wide margin the largest market share globally at ~80%. Developing the mobile "receiving" VR app for Android would provide access to this technology to the largest volume of people. Android is also the only platform for which GStreamer currently provides an SDK, allowing for the same low-latency streaming seen above to be received on a mobile Android device. (GStreamer 2016)

There are a multitude of app-development IDEs for Android, with the main no-charge solutions being Android Studio and Eclipse. Android Studio is an IDE built on an open-source version of the popular IntelliJ IDEA Java IDE and is provided directly by Google, the developers of Android. Eclipse is an open-source software development IDE providing support for dozens of languages, providing Android support via the “Android Development Tools for Eclipse” plugin. Of the two, Android Studio is by far the more used within the app development community and provides an excellent turnkey solution for Android app development. While both Eclipse and Android Studio can be used, Android Studio was chosen for its ease of setup and use, as well as the massive banks of community support dedicated to developing apps specifically within Android Studio.

Embedded Development Toolchain Chosen: Arduino IDE

Open-source support for AVR microcontroller programming has been around for over a decade, but only recently have code editor and programmer/flasher combos supporting Arduino boards by the open-source community. There are many options available to the hobby roboticist wanting to develop for an Arduino: plugins for Atmel Studio, Eclipse, Codeblocks, Visual Studio, and even Gedit are available, and several Arduino-specific IDEs have been developed. Of the Arduino-focused IDEs the most prominently used is, by far, Arduino’s own Arduino IDE. This is the platform the majority of code used on Arduino boards is developed on and provides a very straightforward way to write, compile, and flash code to Arduino boards. Additionally, this tool provides an easy way to access and include feature-expanding libraries from inside the IDE.

Streaming Platform Operating System Chosen: Raspian Jessie Lite

With the popularity of the Raspberry Pi platform comes a wide variety of available operating systems, some officially supported by the Raspberry Pi Foundation and others solely by the community. It was elected to focus solely on the officially supported operating systems provided or endorsed by the Raspberry Pi Foundation, as these operating systems are what the vast majority of Raspberry Pi owners will run on their devices and what the majority of community support will be centered around.

The Raspberry Pi Foundation currently provides one operating system, based on Debian Jessie, in two flavors: Raspian Jessie and Raspian Jessie Lite. Raspian Jessie Lite removes the LXDE desktop and other GUI software found in Raspian Jessie, providing a lightweight, command-line operating system. Other options endorsed by the Raspberry Pi foundation but supported by other entities include Ubuntu MATE, Snappy Ubuntu Core, Windows 10 Internet of Things Core, and RISCOS. A GUI interface is useless (and unnecessarily taxing) for a robot that will be interfaced with primarily via SSH, and we must be able to run code developed for the Linux kernel. These restrictions bring our focus to two choices: Raspian Jessie Lite and Snappy Ubuntu Core. Of these two operating systems Raspian Jessie Lite provides a larger community support base, both in collective knowledge and software availability. (Raspberry Pi Foundation 2016)

4.3 Prototype

A practical demonstration of all the subsystems present in the vision of this project would be ideal, but impractical given the short timeframe of this research. Instead a segmented approach to construction was taken: first construct, then get the chassis moving via Arduino, after that get joystick control via USB, next joystick control via TCP/IP with video streaming, and finally an Android app combining remote control and video streaming. The last goal was never achieved due to time constraints, but full joystick control and video streaming over IP was accomplished.

Physical Construction

For constructing the prototype a GeekBot Basic was used as it provides an Arduino-compatible board out of the box, among other unnecessary accessories like LEDs and a buzzer. On the top platform and centered was mounted gimbal assembly, and the Raspberry Pi 3 mounted in front of the gimbal. The Arduino Uno was mounted on the lower platform near the rear of the bot. With the addition of a five-volt regulator from Pololu Robotics and another battery for long stints away from the mains, the prototype's power system was complete.

The gimbal was assembled using only M2 screws and a spare servo horn included with the Geekbot servos. Servos are physically retained in their receptacles through friction, minimizing hardware usage. Attachment of the gimbal and Raspberry Pi to the frame was achieved with spare hardware from the construction of the GeekBot kit. In the "final" form of the prototype a Logitech C525 webcam was zip-tied to the top of the tilt gimbal. Note that a final, "fully featured" robot would include two webcams spaced to match the average user's interpupillary distance. It was deemed unnecessary to add a second camera during the testing of the bot, as any system that would work with one camera would work just as well with two.

All motors and servos were connected to the Arduino, as well as some of the accessories included in the GeekBot Basic kit for testing and debugging purposes. The Arduino and webcam were connected to the Raspberry Pi via USB, to which power was applied through the onboard micro-USB socket.

It is important to note the relatively minimal cost of this prototype remote robot. The hardware used in construction amounted to around $340, certainly within the reach of most hobbyists and educational institutions.

Software Construction

There are two pieces of software which had to be written for the prototype: a control server and a control client. The control server would take input from a computer-connected joystick (in this case an Xbox 360 controller), parse that input to commands for the robot, and send those the remote-control server running on the Raspberry Pi. This remote-control server would then pass these commands to the Arduino via a serial connection. Note that this remote control scheme is used only to handle the control side of the prototype; video streaming is handled by GStreamer via command line through a script. A fully-featured and modular Arduino "sketch" was written to control all of the GeekBot's functions and control the gimbal's movement. All of this code can be examined in Appendix B. It should be noted that a preliminary Android application was produced but due to time constraints was only developed to the stage of making a TCP connection to the remote-control server on the Raspberry Pi.

Testing

The prototype was tested over hardwired and WiFi connections and provided full remote control of the robot's mobility and gimbal as well as a 100ms-latency 640x480 resolution MJPEG stream over vast distances on campus. As a practical demonstration of this, the robot was placed in the basement of the McLaury building on SDSMT's campus and driven remotely from the Electrical Engineering/Physics building via WiFi many times.

5. Discussion

The prototype constructed was deemed a success and proof-of-concept, proving that most hardware and software dots could connect to produce a partially-fleshed facsimile of the vision mentioned at the start of this document. But, while the remote robot built is quite functional it also suffers from some debilitating limitations.

Physically, the stereo-vision gimbal is mounted very close to the ground: roughly calf height on average-height person. This is quite suboptimal for presenting the illusion of presence in a remote location, as the user would be seeing everything from nearly ground level. Another particularly problematic limitation is that of controllable range. Currently the remote robot's traversable range is limited directly by the WiFi access point's signal strength. Practically this meant only half or less of SDSMT's McLaury building could be explored at one time; such a system would not, ignoring a network of very unusual zero-handoff access points, be able to support campus tours or even full-building tours.

From a mechanical perspective, this platform provides for most needs of a virtual-reality telepresence system quite well. The gimbal provides a full 180 degrees of pan, tilt, and roll while also being very smooth and quick to respond. Locomotion speed maxes out around the average walking speed of an adult, allowing for easy obstacle avoidance and fine maneuvering where necessary. Hard, flat surfaces presented no challenge to the robot, but offroading was unsuccessful. With the addition of another camera this mobile platform provides all the physical capabilities needed for testing and further development of software.

All the software used worked rather well. At no point were the software tools selected insufficient. GStreamer provided a 100ms-latency 640x480 stream with surprising reliability over WiFi, and offers many options for expandability in the future. Both GStreamer and the client/server remote-control software are command-line tools, and as such need some experience with command-line Unix-based systems to operate. Luckily this can be simplified with the addition of startup scripts for both the client and server as used on the prototype and seen in Appendix B.

Another challenge is the IP address of each "side" must be known to the other, i.e. the robot must know the phone's IP, and the phone the robot's. This is currently accomplished by manually finding the external IP of each device via ssh and then updating the respective startup scripts. When used as envisioned, a system such as this would have no knowledge of the either side's IP address, introducing what might be the largest challenge this project faces: finding and connecting the client and remote halves of the system.

The lynchpin part of this proposed telepresence system, the app, never made it past very basic network functionality due to time constraints. Without a functional Android application this system as envisioned is impossible. However, it is possible to construct such an application using Android, Android Studio, and GStreamer's Android SDK; it will just take time and expertise in mobile development.

Cost has also been kept surprisingly low. The prototype constructed came in at just under $350, despite extra cost from an upgraded GeekBot Basic kit. In theory a system based on all the hardware selected above (with the addition of a $15 5-volt regulator) could be acquired for ~$320, including the cardboard headset and Bluetooth controller. This is phenomenally cheap compared to commercial telepresence solutions, even more so when the technology involved is considered.

Despite the many shortcomings of this prototype system it’s important to note that the original objective has been accomplished: it is possible to construct a low-cost, virtual-reality telepresence system on open-source software and community-supported hardware. The constructed prototype, while not a fully-functioning solution as in the original vision, is proof that a rolling robot with a head-tracking system can be controlled over IP while streaming low-latency video using tools available to any hobby roboticist.

6. Conclusion

6.1 Summary

In this research, the feasibility of low-cost, virtual-reality telepresence system was investigated. After finding that all of the necessary software was available in the open-source community and hardware available inexpensively, a prototype system was constructed to test the proposed system. The prototype system achieve remote control of locomotion and a head-tracking gimbal over TCP/IP, and a low-latency video stream over UDP/IP allowing the robot to be driven remotely from anywhere in the world. While successful, the prototype revealed many areas requiring additional investigation and work.

6.2 Future work

Remote Robot Construction

As mentioned above the remote robot's camera gimbal is unsatisfactorily close to the ground. A solution that raised the gimbal to the height of a human head would be optimal, but such a design might introduce further challenges such as camera sway or oscillation during movement. It would also be prudent to design a mobile base capable of traveling over intermediate terrain, such as rough sidewalks, brick paths, or flat grass. However, both of these changes could add considerable expense in producing the remote robot; clearly solutions are available but it remains to be seen what the costs would be and whether they would be prohibitively high.

Connectivity

With a system connected by WiFi comes the range limits of WiFi access points. Barring the terrific expense of installing institution-wide zero-handoff WiFi access points, a cellular solution seems most potable. 4G LTE networks in the United States are able to supply latency as low as 67 milliseconds, and around 8 Mb/s of bandwidth. In theory this seems capable of supporting the proposed telepresence system from anywhere with LTE coverage, but more investigation is needed to determine the practical use of cellular connections in low-latency robotics. (Gibbs 2016)

Connections between user and remote sides are also difficult to establish, as elucidated in the discussion prior. A potential solution would be to follow a system similar to multiplayer video games: maintaining a master server to through which all client-robot connections are managed. Such a system mandates a centralized and permanent connection point, which would have to be hosted by some committed institution or organized group. Surely such a system is possible but whether it's the best choice for connection negotiation should be investigated further.

Software Ease of Use

Currently all the software used on the prototype is command line only, limiting the scope of use to someone with knowledge of text-only Linux. Optimally the tools necessary would be available online and GUI-based, possibly presented on a screen attached to the remote robot. It's also possible that boot-ready images for the Raspberry Pi 3 could be provided, similar to the GNU Radio-prepped images prepared by Gareth Hayes for amateur radio enthusiasts. Easy access to and operation of the software required for this system to function is lacking and could stand a thorough analysis. (Hayes 2014)

Mobile Application

The mobile application the proposed system hinges on has not been produced, but on paper appears possible. Obviously the development of an application to take controller input, track head movement, and connect to a remote robot is a next step following the conclusion of this research. Such work probably falls outside the realm of research and into the realm of product development, however.

References

1. Gibbs, C. (2016, February 02). With average speeds of 12.26 Mbps, T-Mobile is closing the LTE gap with Verizon, OpenSignal finds. Retrieved August 3, 2016, from http://www.fiercewireless.com/wireless/average-speeds-12-26-mbps-t-mobile-closing-lte-gap-verizon-opensignal-finds

2. GStreamer SDK has reached its goal. (n.d.). Retrieved August 2, 2016, from http://gstreamer.com/

3. Hayes, G. (2014, October 28). Raspberry Pi Image with GNU Radio (HackRF DVB-T). Retrieved August 3, 2016, from http://garethhayes.net/gnu-radio-for-raspberry-pi/

4. Metz, C. (2015, June 1). The Inside Story of Google's Bizarre Plunge Into VR. Retrieved

August 2, 2016, from http://www.wired.com/2015/06/inside-story-googles-unlikely-leap-cardboard-vr/

5. Open Source Initiative. (n.d.). History of the OSI. Retrieved August 4, 2016, from https://opensource.org/history

6. Overview of available plug-ins. (n.d.). Retrieved August 1, 2016, from https://gstreamer.freedesktop.org/documentation/plugins.html

7. Raspberry Pi Downloads - Software for the Raspberry Pi. (n.d.). Retrieved August 1, 2016, from https://www.raspberrypi.org/downloads/

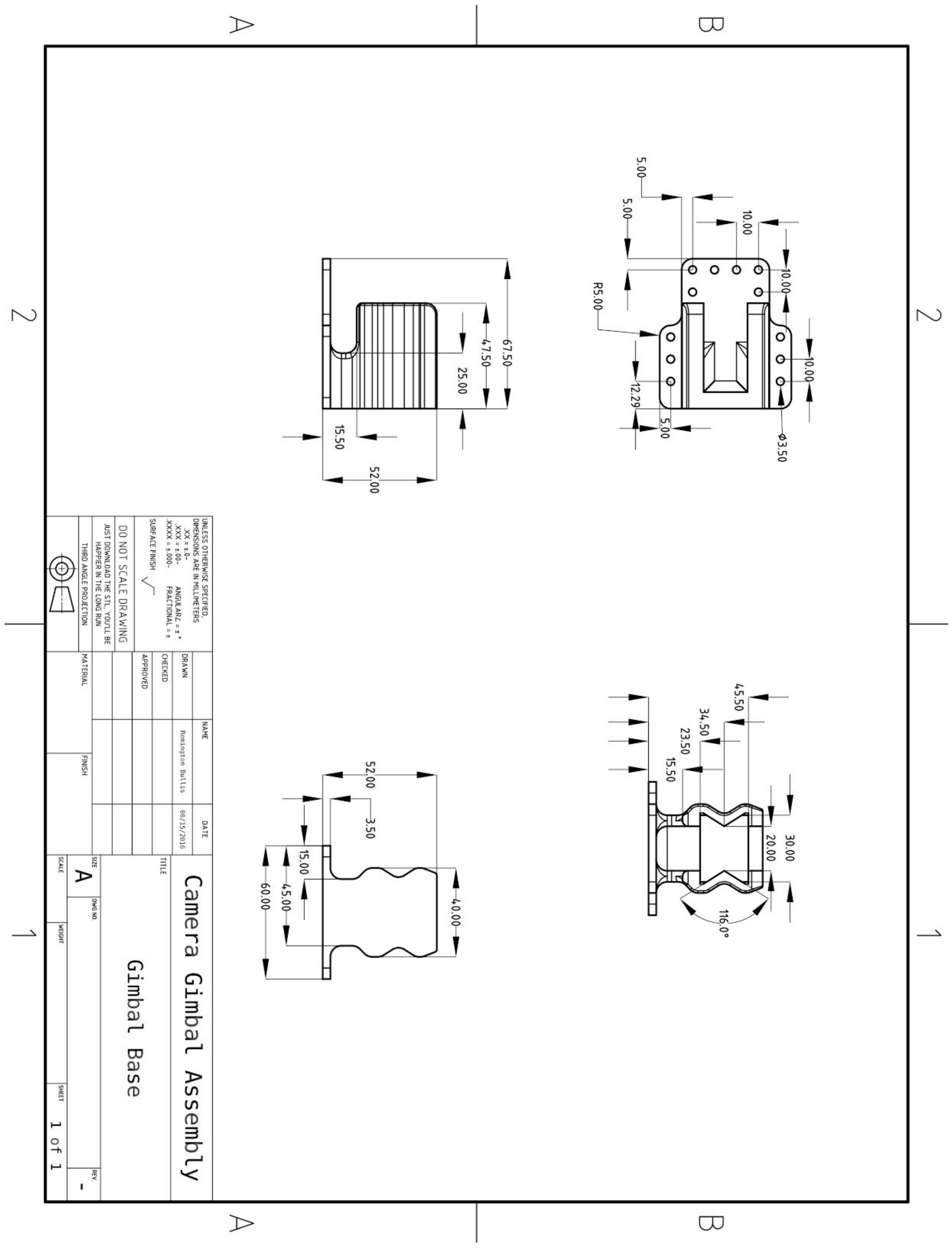
Acknowledgements

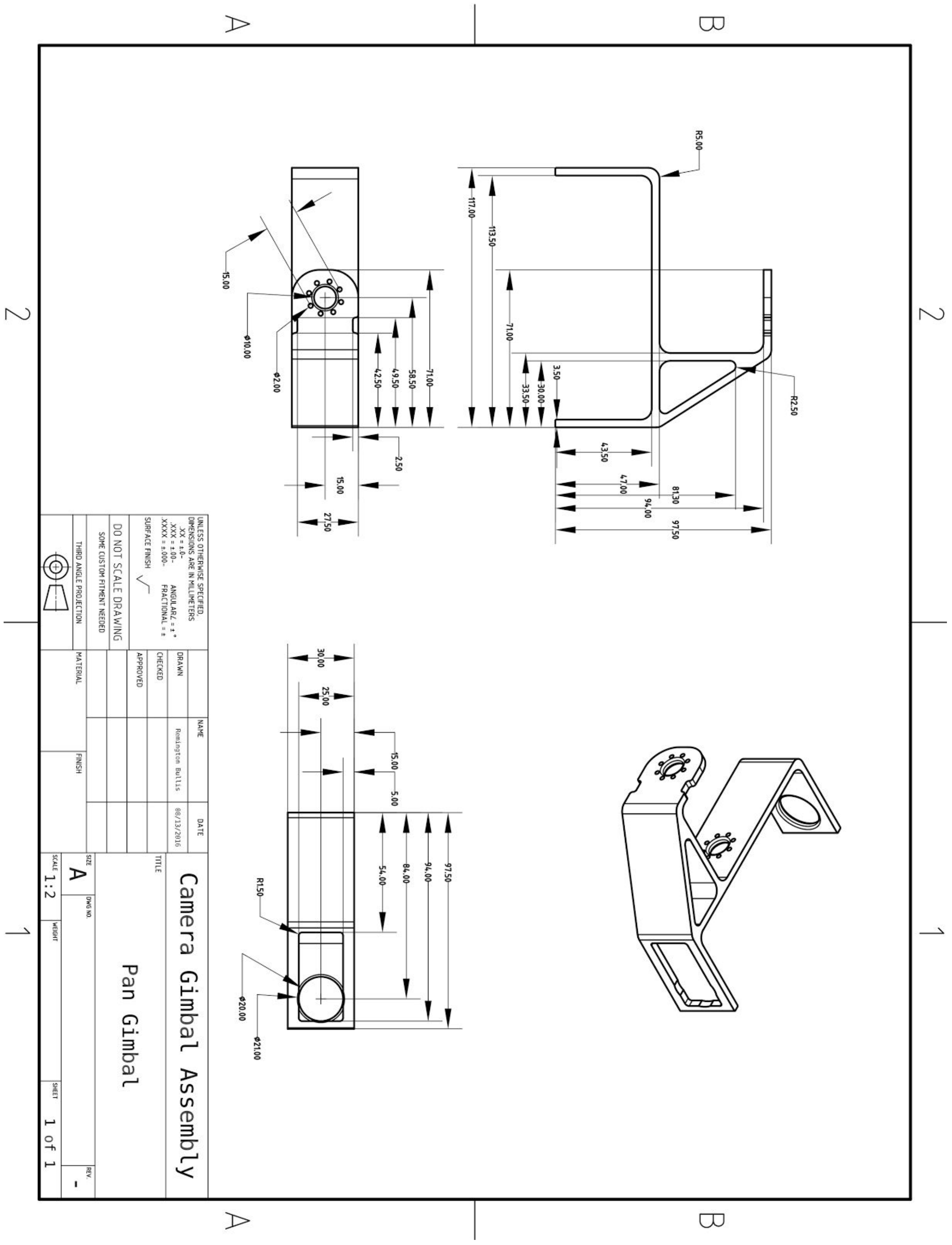
This work was made possible by the National Science Foundation REU Site: Bringing Us Together, Improving Communications and Lives (Award Number EEC-1359476).

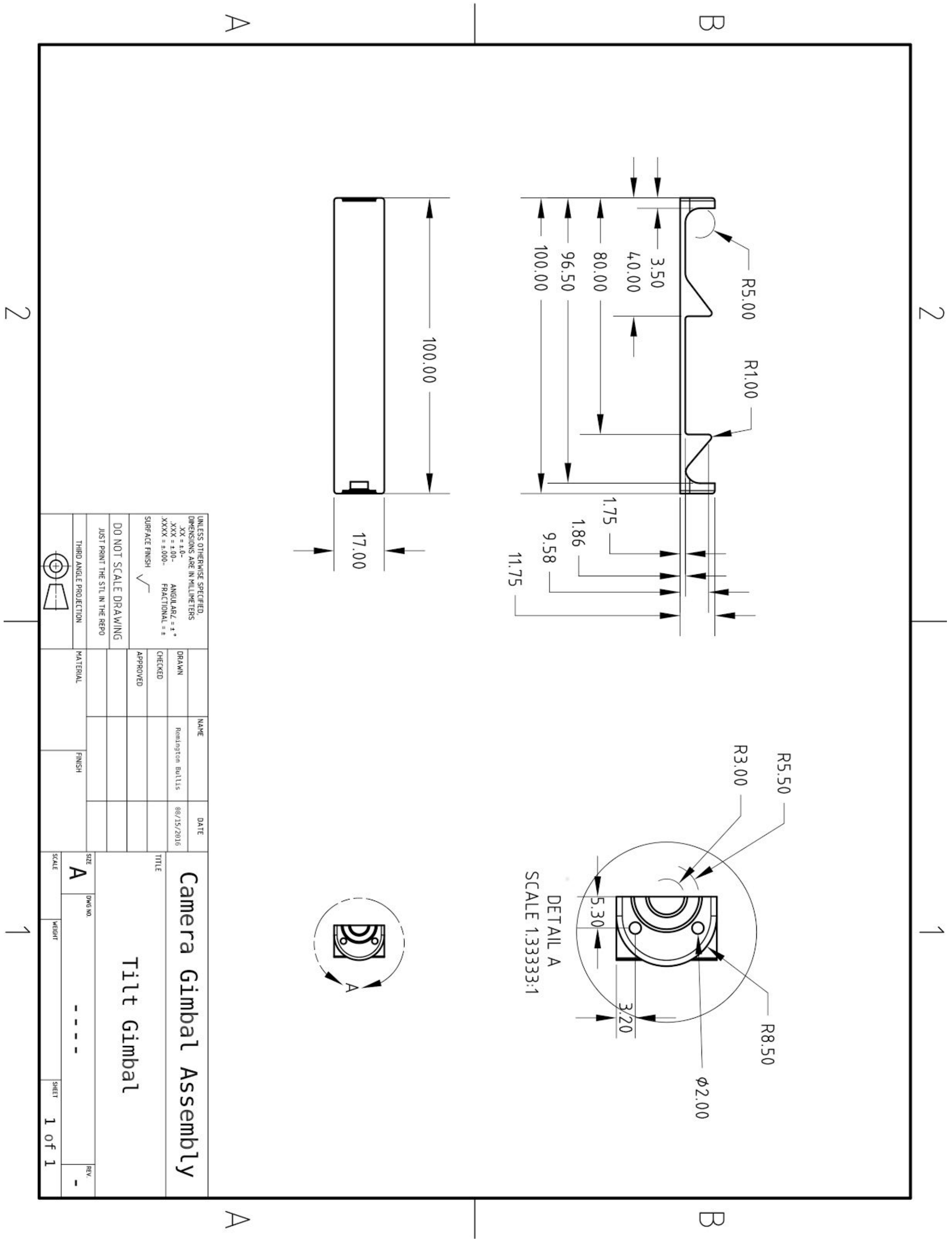
Many thanks to Dr. Thomas Montoya for heading the REU program, Dr. Randy Hoover for his guidance and support, Dr. Jeffrey McGough for providing lab space and hardware, and Dr. Alfred Boysen for keeping the students on track to excel throughout the course of this program.

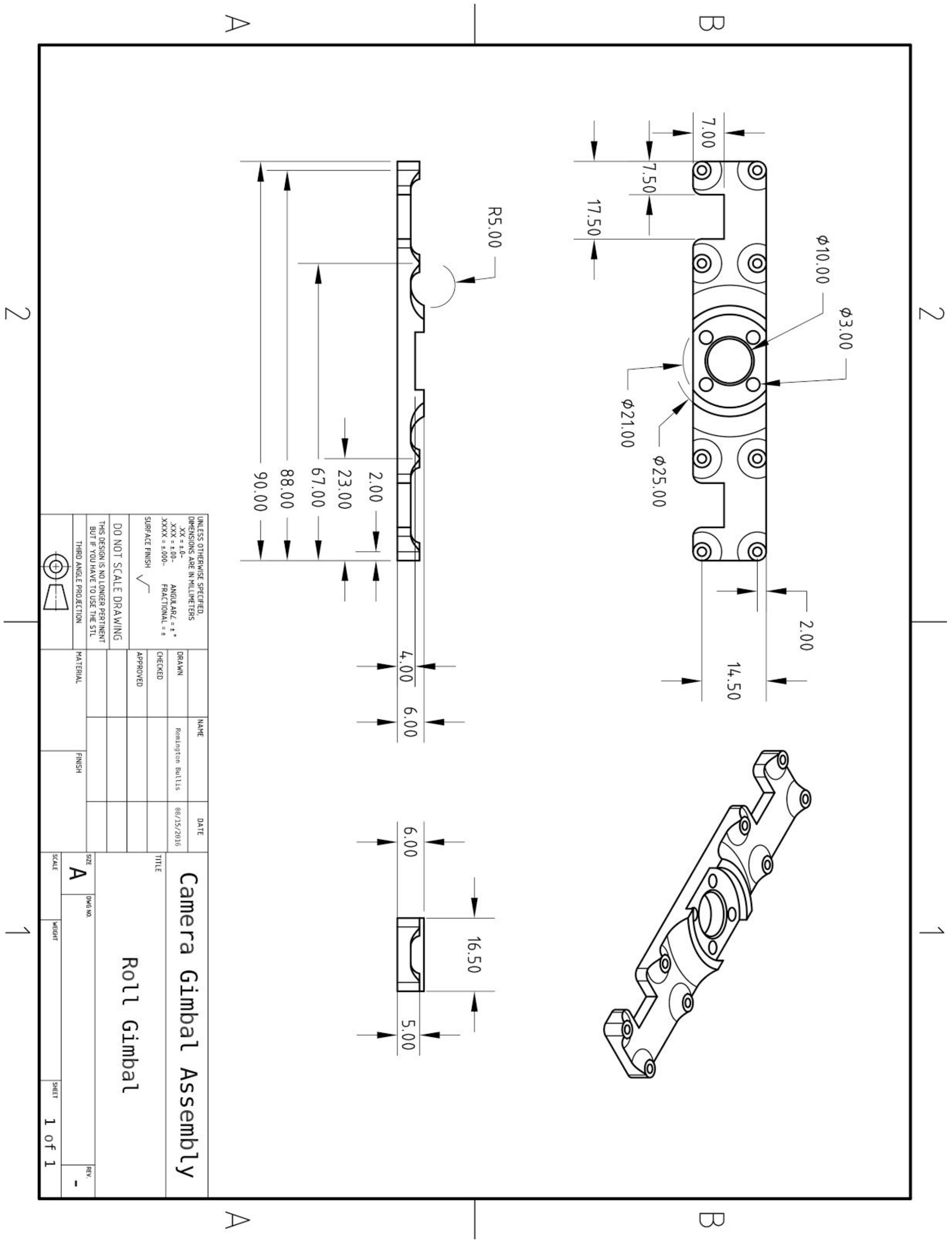
Appendix A: Gimbal Drawings

This appendix contains drawings for each gimbal piece. In practice, these were printed on a PrinterBot Simple 1401 (early 2014 model) modified to use GT2 timing belt, expanding the build area from 10 x 10 x 10cm to 14 x 10 x 10cm. These parts were designed to be printable inside of that volume. Most parts were designed with minimal tolerance in order to minimize the number of fasteners required for the gimbal assembly. This means, even on a dimensionally accurate printer, that some fitting will be required post-print. All of these parts (both drawings and printable STL files) can be found in the author's git repo at https://gitlab.com/rembullis/open-source-telepresence-reu-2016.


UNLESS OTHERWISE SPECIFIED, DIMENSIONS ARE IN MILLIMETERS
.XX = ±.0-
.XXX = ±.00-
.XXXX = ±.000-
ANGULAR∠ = ±°
FRACTIONAL = ±
SURFACE FINISH
DO NOT SCALE DRAWING
JUST DOWNLOAD THE STL, YOU'LL BE HAPPIER IN THE LONG RUN
THIRD ANGLE PROJECTION
NAME
DATE
DRAWN
Remington Bullis
08/15/2016
CHECKED
APPROVED
MATERIAL
FINISH
TITLE
Camera Gimbal Assembly
Gimbal Base
SIZE
A
DWG NO.
REV.
-
SCALE
WEIGHT
SHEET
1 of 1
A
B
1
2
67.50
47.50
25.00
15.50
52.00
5.00
5.00
10.00
10.00
10.00
R5.00
12.29
5.00
ø3.50
45.50
34.50
23.50
15.50
30.00
20.00
116.0°
52.00
3.50
15.00
45.00
60.00
40.00


A
B
2
1
R5.00
117.00
113.50
71.00
3.50
30.00
33.50
R2.50
43.50
47.00
81.30
94.00
97.50
15.00
Ø10.00
Ø2.00
42.50
49.50
58.50
71.00
2.50
15.00
27.50
30.00
25.00
15.00
5.00
54.00
84.00
94.00
97.50
R1.50
Ø20.00
Ø21.00
UNLESS OTHERWISE SPECIFIED,
DIMENSIONS ARE IN MILLIMETERS
.XX = ±.0-
.XXX = ±.00-
.XXXX = ±.000-
ANGULAR∠ = ±°
FRACTIONAL = ±
SURFACE FINISH
DO NOT SCALE DRAWING
SOME CUSTOM FITMENT NEEDED
THIRD ANGLE PROJECTION
NAME
DATE
DRAWN
Remington Bullis
08/13/2016
CHECKED
APPROVED
MATERIAL
FINISH
TITLE
Camera Gimbal Assembly
Pan Gimbal
SIZE
A
DWG NO.
REV.
-
SCALE
1:2
WEIGHT
SHEET
1 of 1


R5.00
3.50
4.00
80.00
96.50
100.00
R1.00
100.00
17.00
1.75
1.86
9.58
11.75
R5.50
R3.00
5.30
3.20
R8.50
ϕ2.00
DETAIL A
SCALE 1.33333:1
A
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN MILLIMETERS
.XX = ±.0-
.XXX = ±.00-
.XXXX = ±.000-
ANGULAR∠ = ±°
FRACTIONAL = ±
SURFACE FINISH
DO NOT SCALE DRAWING
JUST PRINT THE STL IN THE REPO
THIRD ANGLE PROJECTION
NAME
DATE
DRAWN
Remington Bullis
08/15/2016
CHECKED
APPROVED
MATERIAL
FINISH
Camera Gimbal Assembly
TITLE
Tilt Gimbal
SIZE
A
DWG NO.
- - - -
REV.
-
SCALE
WEIGHT
SHEET
1 of 1
A
B
1
2


A
B
2
1
7.00
7.50
17.50
R5.00
ϕ10.00
ϕ3.00
ϕ21.00
ϕ25.00
2.00
23.00
67.00
88.00
90.00
2.00
14.50
4.00
6.00
6.00
16.50
5.00
UNLESS OTHERWISE SPECIFIED,
DIMENSIONS ARE IN MILLIMETERS
.XX = ±.0-
.XXX = ±.00-
.XXXX = ±.000-
ANGULAR∠ = ±°
FRACTIONAL = ±
SURFACE FINISH
DO NOT SCALE DRAWING
THIS DESIGN IS NO LONGER PERTINENT
BUT IF YOU HAVE TO USE THE STL
THIRD ANGLE PROJECTION
NAME
DATE
DRAWN
Remington Bullis
08/15/2016
CHECKED
APPROVED
MATERIAL
FINISH
TITLE
Camera Gimbal Assembly
Roll Gimbal
SIZE
A
DWG NO.
REV.
-
SCALE
WEIGHT
SHEET
1 of 1

Appendix B: Code and Startup Scripts

This appendix contains all of the code written for the prototype construction as well as the scripts used to start up the client and remote robot. The C files (and requisite headers) used in the prototype's construction can be found here. Note that the Arduino code is *not* included, but can be found at the author's git repo along with the code presented in this appendix: https://gitlab.com/rembullis/open-source-telepresence-reu-2016.

networking.h:

```
#ifndef __NETWORKING_H
#define __NETWORKING_H


#include <sys/socket.h>
#include <errno.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>

/*
   SCARY. This header provides a simple way to connect and make TCP/IP connections.
*/

#define DEFAULT_PORT 5000

/*
   This ...thing sets up a TCP client connection to a server, given the IP as chars and a port.
*/
int start_client(char *ip, int port)
{
   int sockfd = 0;
   struct sockaddr_in serv_addr; // Bog standard struct for IP things

   if(ip == NULL) // Try giving me a *real* IP, thanks!
                        return -1;

   sockfd = socket(AF_INET, SOCK_STREAM, 0); // Request a socket from the OS
   if(sockfd <= 0)
      return -2;

   serv_addr.sin_family = AF_INET; // Gimme TCP
   if(port <= 0)
      serv_addr.sin_port = htons(DEFAULT_PORT); // Set the port to supplied if port > 0
   else
      serv_addr.sin_port = htons(port);

   if(inet_pton(AF_INET, ip, &serv_addr.sin_addr) <= 0) // Get a correctly-formatted IP address into the struct.
      return -3;

   if(connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)   // Try to make a connection to the IP
      return -4;
                                          //  generate above as the medium.
```

```
    return sockfd;
}

/*
    This thing, similar to the above, starts the server on the robot side. Beware.
*/
int start_server(int port)
{
    int serverfd = 0, connfd = 0;
    struct sockaddr_in serv_addr;

    serverfd = socket(AF_INET, SOCK_STREAM, 0); // The SOCKET fd

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY); // Connection setup crap
    if(port <= 0)
        serv_addr.sin_port = htons(DEFAULT_PORT);
    else
        serv_addr.sin_port = htons(port);

    bind(serverfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)); // Yeah I wanna listen on this port and IP
    listen(serverfd, 10);                              // Telling the OS to listen to only 10 connections at once!

    return serverfd;
}

/*
    This kills the server (socket)
*/
int stop_server(int serverfd)
{
    close(serverfd);
    return 0;
}

/*
    This actually acquires the connection with the remote client, returning the stream fd when a connection is made.
*/
int get_stream(int serverfd)
{
    int streamfd = -1;

    do
    {
        streamfd = accept(serverfd, (struct sockaddr*)NULL, NULL); // SOMEONE ANYONE PLEASE RESPOND
    }
    while(streamfd < 1);

    return streamfd;
}
```

```
/*
   Kills the stream, no diff from above but in name for keeping calls straight
*/
int close_stream(int streamfd)
{
   close(streamfd);
   return 0;
}



#endif
```

---

controller.h:

```
#ifndef __CONTROLLER_H
#define __CONTROLLER_H

#include <stdio.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include "360_controller_map.h"

/*
   This file provides a basic "controller" struct. Not much more, as it's fleshed out in
   the "geekbot_controller" header.
*/

#define EVENT_BUTTON         0x01    // Button state update
#define EVENT_AXIS           0x02    // Axis state update
#define EVENT_INIT           0x80    // Non-input event, to be ignored

#define MAX_BUTTON_COUNT        16
#define MAX_AXIS_COUNT          27   // 8 for 360, 27 for PS3

struct event {
   unsigned int time;          // Timestamp of when this event came in, in millis
   short value;                // The event's associated value
   unsigned char type; // The event type, if button or axis
   unsigned char id;   // The event ID, tells which button or axis when used with a joystick
};
struct event jsevent;

/*
   This is a struct to make handling controllers simpler. It provides a simple "object" to pass
    around to functions.
*/
struct js_controller {
   char controller_location[128];          // Location to open the controller from, think /dev/input/js0
   int controller_id;                      // Wierd name, is actually the file descriptor with opened. Named as such to easily
differentiate between multiple controllers
   unsigned char button[MAX_BUTTON_COUNT];  // Array of button values, as defined above
   short axis,[MAX_AXIS_COUNT];                              // Array of axis values, as defined above
   int output_id;                                                       // An OUTPUT TARGET file descriptor.
Could be useful if using two controllers sending to two devices.
};

/*
   This function "starts" the controller, opening its location and returning the fd
*/
int start_controller( struct js_controller *controller )
```

```
{
   controller->controller_id = open( controller->controller_location, O_RDONLY | O_NONBLOCK );
   if( controller->controller_id < 0 )
   {
      printf( "Unable to open controller at %s", controller->controller_location );
         return -1;
   }
   return controller->controller_id;
}

/*
   Checks the controller for any buffered inputs, updates the controller data accordingly
*/
int update_controller( struct js_controller *controller )
{
   int temp = controller->controller_id;
   if(controller->controller_id < 0)
   {
      printf("Trying to update a non-existent controller!\n");
      return -1;
   }
   while( read(temp, &event, sizeof(event)) > 0 )
   {
      event.type &= ~EVENT_INIT; // ignore non-input events
      if ( event.type == EVENT_AXIS )
         controller->axis[event.id] = event.value; // update the corresponding axis/button with its new value
      if( event.type == EVENT_BUTTON )
         controller->button[event.id] = event.value;
   }
   return 0;
}

/*
   Returns the respective button state
*/
unsigned char get_button_state( unsigned char button_id, struct js_controller *controller )
{
   return controller->button[button_id];
}

/*
   Returns the respective axis state
*/
short get_axis_state( unsigned char axis_id, struct js_controller *controller )
{
   return controller->axis[axis_id];
}

#endif
```

---

geekbot_controller.h

```
#ifndef __GEEKBOT_CONTROLLER_H
#define __GEEKBOT_CONTROLLER_H

#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include "easy_serial.h"
#include "controller.h"
#include "360_controller_map.h"
#include "networking.h"

/*
   This file fleshes out the baseline "controller" struct for use specifically with the geekbot.
   Obviously the attempted modular solution is a bit half-baked, but it works.
*/

#define BAUDRATE           115200 // The baudrate the robot will communicate on, obviously they should match.

#define FLAG_DRIVE_DRIVE   0x45
#define FLAG_DRIVE_TURN    0x37
#define FLAG_DRIVE_LEFT    0x36
#define FLAG_DRIVE_RIGHT   0x35

#define FLAG_OUT_LIGHTS    0x30
#define FLAG_OUT_BUZZER    0x29

#define FLAG_CAM_PAN_HOR       0x28
#define FLAG_CAM_PAN_VERT      0x27
#define FLAG_CAM_ROLL          0x26

#define DEADZONE_HORIZONTAL_MIN  -6000
#define DEADZONE_HORIZONTAL_MAX   6000

#define DEADZONE_VERTICAL_MIN  -6000
#define DEADZONE_VERTICAL_MAX   6000

/*
   This bounds any input to within the range of a short
*/
short bound_short(int num)
{
   if( num > SHRT_MAX )
      return SHRT_MAX;
   else if(num < SHRT_MIN )
      return SHRT_MIN;
   else
      return num;
}
```

```
/*
   Maps a value on an input range to an output range. Looks similar to the Arduino code, yeah?
*/
short map(long x, long in_min, long in_max, long out_min, long out_max)
{
   return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}

/*
   Specifically maps to the range of a short, like above. Not *really* necessary but nice to have.
*/
short short_map(long x)
{
   return (x - SHRT_MIN) * (2*SHRT_MAX - 2*SHRT_MIN) / (SHRT_MAX - SHRT_MIN) + 2*SHRT_MIN;
}

/*
   This is the function called when a signal is caught. SIGTERMS are *not* handled, be aware.
*/
void sig_handler(int signo)
{
   if (signo == SIGINT)
   {
      printf("Received SIGINT, exiting.\n");
         exit(1);
   }
   else
      printf("UNKNOWN SIGNAL\n");
   return;
}

/*
   Attempts to attach the sig_handler func as the signal handler callback.
*/
int sig_setup()
{
   if (signal(SIGINT, sig_handler) == SIG_ERR)
   {
       printf("\ncan't catch SIGINT\n");
       return 1;
   }
   return 0;
}

/*
   Sends a "command" composed of a flag and short value. Flags are defined above.
*/
int send_command( unsigned char flag, short value, int send_port )
{
   unsigned char out_flag  = flag, i;
   short out_value = value;
   int n  =  write(send_port, &out_flag, 1);
   n +=  write(send_port, &out_value, sizeof(out_value));
   return 0;
```

```
}

/*
   This function takes in an array of "old" button presses and compares the values against the "new" values stored in
the input controller's array.
   Upon detection of a new button press, the respective function inside the switch is called. There's NO doubt in my
mind that there's a better way
   to do this, but this works for now.
*/
int execute_button_updates( unsigned char *old_button_values,
                                                  struct js_controller *controller )
{
   int button_id, pushed;
   for(button_id = 0; button_id < MAX_BUTTON_COUNT; button_id++)
   if(controller->button[button_id] != old_button_values[button_id])
   {
      old_button_values[button_id] = controller->button[button_id];
      pushed = controller->button[button_id];
      switch(button_id)
      {
         case BUTTON_A:
            break;
         case BUTTON_B:
                                    break;
         case BUTTON_X:
                                    break;
         case BUTTON_Y:
                                    break;
         case BUTTON_LEFT_BUMPER:
            if(pushed)
               send_command(FLAG_CAM_ROLL, SHRT_MIN, controller->output_id);
            else
               send_command(FLAG_CAM_ROLL, 0, controller->output_id);
            break;
         case BUTTON_RIGHT_BUMPER:
            if(pushed)
               send_command(FLAG_CAM_ROLL, SHRT_MAX, controller->output_id);
            else
               send_command(FLAG_CAM_ROLL, 0, controller->output_id);
            break;
         case BUTTON_LEFT_CLICK:
            if(pushed)
               send_command(FLAG_OUT_BUZZER, 1, controller->output_id);
            else
               send_command(FLAG_OUT_BUZZER, 0, controller->output_id);
            break;
         case BUTTON_RIGHT_CLICK:
            if(pushed)
               send_command(FLAG_OUT_LIGHTS, 1, controller->output_id);
            else
               send_command(FLAG_OUT_LIGHTS, 0, controller->output_id);
            Break;

         case BUTTON_START:
            break;
         case BUTTON_BACK:
            break;
         case BUTTON_XBOX:
            break;
         default:
            printf("Button not supported.\n");
            break;
      }
   }
   return 0;
}

 /*
         Check the above function, same idea.
 */
int execute_axis_updates( short  *old_axis_values,
                                             struct js_controller *controller )
{
   int axis_id;
   short value;
   for(axis_id = 0; axis_id < MAX_AXIS_COUNT; axis_id++)
      if(controller->axis[axis_id] != old_axis_values[axis_id])
         {
            old_axis_values[axis_id] = controller->axis[axis_id];
            value = controller->axis[axis_id];
            switch(axis_id)
            {
               case AXIS_LEFT_STICK_VERTICAL:
               case AXIS_LEFT_STICK_HORIZONTAL:;
               static  short left = 0, right = 0, new_right, new_left;
                  int temp  = controller->axis[AXIS_LEFT_STICK_VERTICAL] +
controller->axis[AXIS_LEFT_STICK_HORIZONTAL]; // Some ghetto code for arcade drive
                  new_right = temp/2;
                  temp = controller->axis[AXIS_LEFT_STICK_VERTICAL] -
(controller->axis[AXIS_LEFT_STICK_HORIZONTAL]);
                  new_left = temp/2;
                  if( new_left != left )
                  {
                     send_command(FLAG_DRIVE_LEFT, new_left, controller->output_id);
                     left = new_left;
                  }
                  if( new_right != right )
                  {
                     send_command(FLAG_DRIVE_RIGHT, new_right, controller->output_id);
                     right = new_right;
                  }
                  break;
               case AXIS_RIGHT_STICK_VERTICAL:
                  send_command(FLAG_CAM_PAN_VERT, value, controller->output_id);
                  Break;


               case AXIS_RIGHT_STICK_HORIZONTAL:
                  send_command(FLAG_CAM_PAN_HOR, value, controller->output_id);
                  break;
               case AXIS_LEFT_TRIGGER:
                  value = map(value, SHRT_MIN, SHRT_MAX, 0, -SHRT_MAX);
                  send_command(FLAG_CAM_ROLL, value, controller->output_id);
                  break;
               case AXIS_RIGHT_TRIGGER:
                  value = map(value, SHRT_MIN, SHRT_MAX, 0, SHRT_MAX);
                  send_command(FLAG_CAM_ROLL, value, controller->output_id);
                  break;
               default:
                  printf("Axis not supported.\n");
                  break;
            }
         }
   return 0;
}

#endif
```

---

easy_serial.h:

```
#ifndef __EASY_SERIAL_H
#define __EASY_SERIAL_H

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <sys/ioctl.h>
#include <getopt.h>

/*
   This file provides a simple, easy-to-use interface
   for using serial ports.
*/


/*
   This function just takes ina  location, and a baud rate, and
   then attempts to open the port for async serial read/write
*/
int serial_port_init(const char* serialport, int baud)
{
   struct termios toptions;
   int fd;

   fd = open(serialport, O_RDWR | O_NOCTTY | O_NDELAY);
   if (fd == -1)  {
      printf("Can't open serial port!");
      return -1;
   }

   speed_t brate = baud;

   switch(baud) {
      case 4800:   brate=B4800;   break;
      case 9600:   brate=B9600;   break;
      case 19200:  brate=B19200;  break;
      case 38400:  brate=B38400;  break;
      case 57600:  brate=B57600;  break;
      case 115200: brate=B115200; break;
   }
   cfsetispeed(&toptions, brate);
   cfsetospeed(&toptions, brate);

   // 8N1
   toptions.c_cflag &= ~PARENB;
```

```
    toptions.c_cflag &= ~CSTOPB;
    toptions.c_cflag &= ~CSIZE;
    toptions.c_cflag |= CS8;
    // no flow control
    toptions.c_cflag &= ~CRTSCTS;

    toptions.c_cflag |= CREAD | CLOCAL;  // turn on READ & ignore ctrl lines
    toptions.c_iflag &= ~(IXON | IXOFF | IXANY); // turn off s/w flow ctrl

    toptions.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG); // make raw
    toptions.c_oflag &= ~OPOST; // make raw
    toptions.c_cc[VMIN]  = 0;
    toptions.c_cc[VTIME] = 20;
    return fd;
}

/*
    Dump anything buffered to junk
*/
void clearPort(int port)
{
    int n = 1;
    char nothing = 0;
    while(n > 0)
        n = read(port, &nothing, 1);
    return;
}

#endif
```

---

geekbot_client.c:

```
#include <stdio.h>
#include "geekbot_controller.h"

/*
   This is the actual client run code. It takes three args: -j, -p, -a. These correspond to the
   joystick location (ala /dev/input/js0), the outgoing port,a nd the outgoing address.
*/

// It's  a super lazy function to handle cmd line args!
int handle_args(int argc, char **argv, char *joystick, char *ip, int *port)
{
   int i;
   for(i = 1; i < argc; i++)
   {
      if(!strcmp(argv[i], "-j"))
         strcpy(joystick, argv[i+1]);
      else if(!strcmp(argv[i], "-p"))
         *port = atoi(argv[i+1]);
      else if(!strcmp(argv[i], "-a"))
         strcpy(ip, argv[i+1]);
      i++;
   }
}

int main(int argc, char **argv)
{
   if(sig_setup()) // Assign the sighandler
      return 1; //GET OUT

   char server_address[64] = "127.0.0.1"; // Defaults for the server address, joystick loc, and port.
   char joystick[64] = "/dev/input/js0";
   int  port = -1;

   if(argc > 7)
   {
      printf("Too many arguments! Exiting."); // Mess with the best, die like the rest.
      return 1;
   }

   handle_args(argc, argv, joystick, server_address, &port); // Laaaaaaaazy....

   printf("Trying to connect to %s:%i...\n", server_address, port);
   int robot_ip = start_client(server_address, port); // In this case robot_ip is a file descriptor, so bear with me.
   if(robot_ip < 1)
   {
      printf("Unable to connect, returned fd = %i\n", robot_ip); // Stuff’s broke.
      return 2;
   }

   printf("Connected to server!\nrobot_ip=%i\n", robot_ip);
```

```
    short        old_axis_values[MAX_AXIS_COUNT] = {0};
    unsigned char old_button_values[MAX_BUTTON_COUNT] = {0}; // These are archives for past values.
    struct js_controller xbox; // A controller struct, for a 360 controller though it's agnostic.

    strcpy(xbox.controller_location, joystick); // Update the controller.

    xbox.output_id = robot_ip; // Tell it where to spit commands.

    if(start_controller(&xbox) < 0) // Try to open the controller desperately.
    {
        printf("Controller no go, exiting...\n"); // /kills self
        return 1;
    }
    printf("Controller started, output_id = %i\n", xbox.output_id); // Things are going great fam.

    while(1)
    {
        update_controller(&xbox);                                         // Check for new controller info
        execute_button_updates(old_button_values, &xbox);       // Use the new controller and axis values to do things
        execute_axis_updates(old_axis_values, &xbox);
        usleep(1000);                                      // This keeps the computer working for longer than 30
seconds.
    }
}
```

---

geekbot_server.c:

```
#include <stdio.h>
#include "networking.h"
#include "easy_serial.h"

/*
   Similar to the client run code, this takes two args: -p and -d.
   The first is the (IP) port to listen on, the second is the serial port
   location (ala /dev/ttyUSB0).
*/

#define COMMAND_LENGTH 3
#define BAUDRATE 115200

// Struct for holding a full "command"
typedef struct command_struct{
   unsigned char flag; // Tells what to do with...
   short value;        // This information.
} command;

/*
   This incredibly complicated (snort) function composes a two-byte int (a short, on most x86) from two bytes
*/
short make_int(unsigned char highbyte, unsigned char lowbyte)
{
   short val = 0;
   val |= highbyte;
   val <<= 8;
   val |= lowbyte;
      return val;
}

/*
   Reads n bytes from fd into dest. Crazy!
*/
int read_n_bytes(int fd, unsigned char *dest, unsigned int n)
{
   int read_bytes = 0;
   int num;
   while (read_bytes < n)
   {
      num = read(fd, dest+read_bytes, n-read_bytes);
      if (num < 1 )
      {
         printf("Unable to read from file, might be dead.\n");
         return 1;
      }
      read_bytes += num;
   }
   return 0;
}
```

```
/*
        Provides an easy way to grab one command from a file.
*/
int get_command(int fd, command *cmd)
{
   unsigned char temp[3] = {0};
   if(read_n_bytes(fd, temp, COMMAND_LENGTH))
      return 1;
   cmd->flag = temp[0];
   cmd->value = make_int(temp[2], temp[1]);
   return 0;
}

/*
   Sends a command, composed of a flag followed by a short value
*/
int send_command( unsigned char flag, short value, int send_port )
{
   unsigned char out_flag  = flag, i;
   short out_value = value;
   int n  =  write(send_port, &out_flag, 1);
      n +=  write(send_port, &out_value, sizeof(out_value));
   return 0;
}

/*
   A *very* lazy function to handle command line args.
*/
int handle_args(int argc, char **argv, char *serial_port, int *port)
{
   int i;
   for(i = 1; i < argc; i++)
   {
      if(!strcmp(argv[i], "-d"))
         strcpy(serial_port, argv[i+1]);
      else if(!strcmp(argv[i], "-p"))
         *port = atoi(argv[i+1]);
      i++;
   }
}

int main(int argc, char **argv)
{
   if(argc < 3)
   {
      printf("You must provide an IP port on which to listen for connections (-p), and a TTY (-d) for comms!\n");
      return 1;
   }

   char serial_port[64] = "/dev/ttyUSB0"; // Default ports, both IP and serial
   int port = 5000;
   handle_args(argc, argv, serial_port, &port); // Lazy AF
```

```
    int serialfd = serial_port_init(serial_port, BAUDRATE); // Attempt to start the serial connection
    if(serialfd < 1)
    {
        printf("Unable to open serial port, exiting...\n"); // Seppuku
        return 1;
    }

    printf("Serial port opened...\n");
    int serverfd = start_server(port); // Scary function call to "start the server", whatever that means.
    if(serverfd < 1)
    {
        printf("Error starting server: %i\n", serverfd);    // Suicide by code
        return 2;
    }

    printf("Server started fd = %i, waiting for connection on TCP port %i\n", serverfd, port);
    int streamfd = get_stream(serverfd); // If everything goes according to plan, the prog just sits here while waiting
for a TCP bytestream from a client.

    if(streamfd < 1)
    {
        printf("Error getting stream from server: %i\n", streamfd); // Stuff's wrecked
        return 3;
    }

    command cmd; // Crazy, a variable!
    printf("Connected streamfd = %i, waiting for input from client\n", streamfd); // Things are working.
    while(1)
    {
        if(get_command(streamfd, &cmd) == 0)
            send_command(cmd.flag, cmd.value, serialfd); // Pump that command to the Arduino
        else
        {
            close_stream(streamfd); // If the stream can't be read, this loop is run. It closes the stream, and waits til a new
stream is available.
            do
            {
                sleep(1);
                printf("Connection lost, waiting for new connection...\n");
                get_stream(serverfd);
            }
            while(streamfd < 1);
            printf("Successfully reconnected.\n"); // Hooray, you only screwed up a little bit.
        }
        usleep(1 * 1000); // Help the scheduler do other computery things. If this isn't there one core gets pegged worse
than a log cabin.

    }
            return 0;
}
```

---

client_startup.sh:

```
#!/bin/bash

LOCAL_IP=$(ip route get 8.8.8.8 | awk 'NR==1 {print $NF}')  # This grabs the current IP used to contact the
internet
ROBOT_IP=10.42.0.15                          # This is the remote device's IP. Obiously you need to know this.
CONTROL_PORT=5000
# The port on which RC commands will be sent
STREAM_PORT=5001
# The port on which the video stream will be received.

sudo pkill gst-launch-1.0 # Kill any instances of these things that are running
sudo pkill geekbot_client
echo "Trying to find stream...\n"
sudo gst-launch-1.0 -vv -e udpsrc port=$STREAM_PORT ! jpegdec ! xvimagesink &  # Split off the stream viewer
echo "Starting control client.\n"
sudo ~/your_path_here/geekbot_client -j $JOY -a $ROBOT_IP -p $CONTROL_PORT # Start the RC client, try to
connect and drive

sudo pkill gst-launch-1.0 # Once the RC client exits, the stream and client are killed (or re-killed)
sudo pkill geekbot_client
```

server_startup.sh:

```
#!/bin/bash

LOCAL_IP=$(ip route get 8.8.8.8 | awk 'NR==1 {print $NF}') # Grab the device's internet-connected IP
VIEWING_IP=151.159.143.56       # This is the CLIENT IP. You need to know this
CONTROL_PORT=5000                       # Port on which to look for RC connections
STREAM_PORT=5001                        # Port on which to broadcast the stream
SERIAL=/dev/ttyUSB0             # The serial port location

sudo pkill gst-launch-1.0 # Safety first
sudo pkill geekbot_server
echo "Starting stream.\n"
sudo gst-launch-1.0 v4l2src device=/dev/video0  ! "video/x-raw,width=640,height=480,framerate=30/1" ! jpegenc !
udpsink host=$VIEWING_IP port=$STREAM_PORT &
echo "Starting control server.\n" # The above line starts the stream broadcasting as a fork
sudo ~/your_path_here/geekbot_server $CONTROL_PORT # This just waits for an RC connection and then drives

sudo pkill gst-launch-1.0 # Kill the things when done
sudo pkill geekbot_server
```