Chloe, Part I

A Robotic Exploration into Digital Companions
Part I: The Body

Melissa Kronenberger

ITGM 736: Physical Interactive Media

Fall 2013

I Design Abstract

When we use the term, 'interaction design,' we are not limited to discussing video games and web sites. In fact, interaction designers can be found in many disciplines, from other entertainment-specific fields like theme park design, to broader and more varied subjects like marketing. Interaction design is relative to information systems, the service industry, education, communication, and any other system in which an exchange and an experience both occur.

What follows is my exploration into the realms of physical interactive media, through the design and development of an interactive toy named *Chloe*. As a toy, Chloe will take the form of a six-legged robot, or 'hexapod.' As in an interactive entity, she will be driven by an on-board 'brain' in the form of an embedded system.

To further the research and development of my *Agon & Alea Thesis*, Chloe will be used to gain insight into how how physical toys can become the artificial companions of their owners. After my previous experience with augmented reality and computer vision, Chloe will be used to evaluate the role sound can play in enhancing a bonding experience. It is expected that Chloe will become responsive to sound at the end of *Chloe, Part II.*

In *Chloe, Part I*, I will be hunting down and applying the information I need in order to bring the toy's physical form to life. The end goal of *Part I* is to assemble Chloe's physical body, set up the development environment, communicate successfully with the chosen embedded system, and, finally, use this setup in order to operate Chloe's legs.

II Table of Contents

III Introduction

“As I sit here with my new BeagleBone cupped in my hands, all white with two long strips of black ports, I can’t help but be intimidated. I have never worked with a piece of hardware like this before, and I’m scared. The sheer amount of knowledge I will have to assimilate in a short period of time will be staggering.

“Then again, I already know I can do it. First of all, I've worked on harder projects. Secondly, I have worked on projects with even less preliminary knowledge. And thirdly, I am surrounded by people with engineering majors. I expect the latter to actually be the least helpful; I'm known to have a temper when frustrated.

“Where was I? Ah yes. As I sit here with my new BeagleBone, my embedded system of choice, I cannot help but wonder at the power packed into such a tiny thing. *This* is the board that will bring to life my very first robot, and that? That is going to be *neat*. I think I will name her Chloe.”

1 Project Overview

Chloe, Part I is the first of a three-part project that will deal with the construction, design, and programming of a small interactive robot. This robot will be brought to life by a small embedded system that will act as the toy's brain. At the conclusion of *Chloe, Part III*, the toy will be able to walk and respond to a player's voice.

2 Project Purpose

This report details my experience developing with physical interactive media for the very first time. Its primary purpose is to enable me as an interactive designer. As a result, I will have the guts and intuition necessary to design for upcoming technology, or lead a team of toy developers.

Secondly, because *Chloe* already fits the mold of being an artificial character, I will be using her to further my thesis research into how players can bond with and derive emotional pleasure from interactive characters. Specifically, I will use Chloe to evaluate how vocal interactions can be used to create emotional pleasure.

3 Report Structure

As a personal exploration and thesis tool, I have elected to write the majority of *Chloe* in the style of a first-person present tense narrative; yet I have organized this narrative into a formal report.

The decision to use a present-tense narrative form reflects the experience of working on Chloe, which feels often like detective work. I have come to describe my learning and research process as, 'hunting for clues,' to what not only my solutions, but even my *questions* should look like.

In the future I intend to use this report as a tool for solving complex problems; and I feel it is important to preserve not only my results, but also my problem solving technique, stumbling blocks,

and my general pacing.

The decision to organize this information in a formal report stems from a need to create a *useful* tool, both for myself and for other designers who might come after me. With the organization and completeness imposed by a formal structure, it will become possible for other designers to replicate my process, compare their solutions, and make sense of my results. It also ensures that I make a thorough record for my own purposes.

The report will be broken into the following sections: Introduction, Background, Project Management, System Design, Implementation and Debugging, Conclusions, and Appendixes.

Introduction will specify the motivations and goals of the project. Background will explain information about the project I have chosen, including a list of parts. It will detail how I chose the project and what my logical, artistic, and personal reasoning was for doing so. Project Management will describe my methodology both in terms of scheduling and recording my progress, while briefly touching on how these project management techniques have become part of my personal design methodology.

System Design will highlight how the project I have chosen will come together, and how the list of parts will combine in order to form the finished toy. The longest section of the paper will be Implementation and Debugging, which will read as a linear narrative starting from the construction of the robot's legs and chassis, and terminating when I am able to move the toy's various appendages.

4 Project Motivation

Now that I already have my board, I suppose I should record my project motivation.

The very first thing I knew about my project was that I wanted to create some kind of toy. I wanted it to move, even if it had only two wheels. I quickly summoned up a simple project within my skull: My toy would be a little two-wheeled alligator that drove about randomly until it encountered a person's finger, whereupon it would chase after that finger in an attempt to bite it.

This was in keeping with the general lean of my thesis. I am a game designer, and I am working on creating emotionally compelling interactive characters that can converse with their players and offer entertainment and pleasure through emotional bonding. I also had extensive experience with computer vision as a result of using augmented reality in my thesis, and I believed that this project was both feasible and natural for me. It was a bonus that our teacher requested we do something 'cool.'

I was only a few hours into researching robot kits and do-it-yourself guides when I finally stumbled upon something very old and very familiar to me: A hexapod.

For the uninitiated, a hexapod is a six-legged robot that looks something like a spider. It is more difficult to drive than any wheeled robot, but it's considerably easier to control than a quadruped or biped. The reason I say hexapods were 'old' and 'familiar' to me is because I was suddenly hit by a massive flashback. I *already knew* what a hexapod *was*.

You see, I had researched hexapods before, in my life; several times, even. As if a little box in the attic of my life had been opened, I suddenly remembered researching the six-legged monsters thoroughly and being fascinated by them. As I remembered this, I started remembering other things, too.

When I was a child, I'd asked for a 'real live robot' for Christmas. After much deliberation, my mother had father purchased a quadruped robot kit. My father and I had accidentally broken it mid-assembly, and it had been retired to the bottom of a toy bin, but scarcely a year before entering this major I had gone home for Christmas, dug the little guy out, and put it all together with a jury-rigged gear. It had never responded to voice commands, but it had walked. I'd been amazed.

That had started my fascination with hexapods. So why hadn't I built one? Why had hexapods been retired to a dusty memory-attic box? The answer: cost.

In America, hexapods were cost-prohibitive for an unemployed kid. Now in Hong Kong, with Shenzhen just across the border, I suddenly found myself confronted with an online listing for an economically viable little six-legged beauty.

I fell in love immediately; that is the solid truth behind my motivation.

I have been told that it is only through our interests that we can truly be inspired. Well, the moment I first saw the hexapod, a very different, surprisingly simple, and remarkably well-defined project plan rolled into my head.

1. I was going to use that hexapod chassis to construct an artificial pet.
2. It was *sound*, not vision, that this pet needed to process. I do not communicate visually with my cat; I communicate by touch and sound. How could I have missed something so simple for the entirety of my thesis? I needed to test this!
3. I would program it to walk, to turn towards me, and to dance in response to my voice.
4. I would use it in the future as a test platform for my thesis's 'bonding' engine.
5. I would design its behavior, and later its appearance, based on the peacock jumping spider to make it cute and more approachable to a general audience.
6. I would program it to respond to its name, unlike my cat.
7. *Her* name was *Chloe*.

I had been too afraid of tackling a 'difficult' project and I'd instead resigned myself to doing an uninteresting one! What did a little snapping alligator robot have to tell me about my thesis or research interests? Nothing! It was *only* a hardware project.

Chloe, on the other hand, is tied to my thesis, my interests, and my past; she is feasible and lies just within the boundaries of what I can push myself to achieve in the allotted time; she will be incredibly difficult, incredibly satisfactory, and incredibly valuable to build; and she is most certainly cool. She is the right project, at the right time, for the right reasons, with the right motivation.

She is Chloe. The act of building her provides its own motivation.

5 Objectives and Aims

Chloe, Part I is responsible for meeting X sets of objectives.

Firstly, *Chloe, Part I* must work towards fulfilling the course requirements set forth by my ITGM 736: Physical Interactive Media course. The course requirements are defined as such: students must learn to use physical input devices, develop physical interactive media prototypes, research the best software and hardware solutions for a personal project, develop a personal project using custom software and hardware, produce short video documentation of their projects for festivals and portfolio reels, and write research reports.

Secondly, *Chloe, Part I* must work towards the individual requirement of the project as set forth in the syllabus; in this case, Project A. This includes respecting the time table set forth in the course syllabus for the project pitch, prototype, documentation, and presentation.

I have also identified a number of objectives that I intend to meet before I can move on to *Chloe, Part II.* The chassis and legs must be fully assembled, and the full range of servos must be accessible. I will not consider *Chloe, Part* I to be truly complete until I have successfully driven the full range of servos through on-board code, and gotten the robot to stand on its own. This project must set the stage for *Chloe, Part II*, which means the robot must be in such a state as it can signal it understands vocal commands.

My overarching objective for *Chloe* is to give me the confidence and familiarity necessary to start other physical interactive media projects. My learning objectives are focused on gaining a broad understanding of the discipline and the various steps of the development pipeline. In this way I not only gain an entry-point to learning more about the discipline, but I also develop the understanding I will need to communicate with hardware engineers and low level programmers in the future.

At the end of *Chloe*, I should be able to visualize the process for translating almost any idea for a physical interactive media project into a design and then into a fully developed product.

On a personal level, I am eager to learn more about Linux, using terminal commands to communicate with the computer, communication protocols, what all those tiny boards do and what they are made of, how to use wires and pins to make a physical prototype, how electricity can be used to create such complex computation devices, what's currently 'out there' available for designers to play with, and how to program a miniature computer.

6 Minimum Requirements

Chloe, Part I's minimum requirements can be derived from taking into account objectives that were established through in-class discussions with the teacher concerning project scope. These objectives are the most concrete and also the most minimal. Chloe must have at least one assembled joint that includes a servo. The servo must be hooked up to my board of choice, which must be able to access and drive the servo.

IV Background

The course syllabus recommends using an Arduino kit that comes with support for Microsoft Kinect. The kit comes with an exercise book, and takes much of the guesswork out of designing for physical systems. More importantly, Arduino boards are specifically built for designers. They carefully eliminate the need to learn many 'engineering' concepts in order to get to meaningful functional prototypes.

Back before I'd rediscovered hexapods, it seemed like I should go with an Arduino board. On the other hand, I didn't really want to work with Kinect. Would an Arduino meet my needs? What would happen to me if I chose to go off the beaten path and purchased a different board?

I had some other options, at least. You see, I had some experience with augmented reality, a point I mentioned above when I illustrated my 'alligator' project. I had recently worked with Quallcomm's augmented reality toolkit, Vuforia. Due to that I knew Quallcomm made processors for mobile phones, which meant Quallcomm probably made tiny boards for designers. I knew that Quallcomm planned to release pointcloud technology for their snapdragon processors in the near future, which to me meant that any Quallcomm board would be able to do some serious visual processing.

I started to research, and I quickly found that Quallcomm *did* make highly powerful boards called DragonBoards, but they were way outside my price range. Their competitors seemed to be Texas Instrument's processors in PandaBoards and BeagleBoards, which I observed were more affordable. Still, I had no way of knowing whether I should stick with my teacher's suggestion of Arduino, pursue one of these other boards, or look for an altogether unrelated board from some other company.

I needed some way to make a good decision. Something better than long hours of research, dozens of online reviews, or countless forum posts...

I needed a brain to pick.

1 Human Resources

'The Captain,' for that is how I shall be referring to him, loves RC toys.

He spends most of his days playing around with tiny boards and building bizarre prototypes. He, along with a large number of eclectic artists, engineers, and technical artists, make up my immediate peer group; the perfect environment in which for me to design and build my very first robot.

The Captain, however, is worthy of particular note. He has met and worked with my instructor, which gives him unique insight into the direction my class will be heading. In fact, The Captain also has several years of teaching experience under his belt, so he knows how to talk to a novice.

Perfect. He will probably end up being my most important resource.

Equally valuable as a human resource is my teacher. With a Masters degree in Engineering (among other things), and plenty of experience in education, my teacher is a wellspring of knowledge. The fact that I have an engineer on each side of the fence, both in class and out, means that every question I ask leaves me in a thorough matrix of immediately valuable and highly targeted information. Where one might only mention the use of a certain protocol, the other can explain its

history and application; where the other might suggest an objective, the one can send me on a search for key terms that will help me understand how to meet that objective. Neither would voluntarily give me explicit directions; and at least one is always on hand to answer the omnipresent question: is this going to electrocute me?

My talented peer group means that I have a wide variety of allies to call on as secondary and tertiary human resources.

If I eventually decide to skin Chloe to resemble a real Jumping Spider, I have two peers who are currently working with cloth and who may be willing to let me borrow their sewing equipment and their time.

I also share my class time with an undergraduate student who is cross-listed in the same basic course. The fact that we are working on similar projects allows us to discuss nearby component stores, online resources, and the difficulties we face while prototyping or dealing with engineers.

a A Note on Human Resources

The most valuable resources we humans have when starting new projects is each other. No amount of internet surfing, catalog shopping, or spec sheet hopping can replace the amazing pattern-matching of the human mind. So it is that my most important (and often most neglected) advice to myself when starting a new project is to query the expertise of the people around me.

This is often incredibly difficult. First of all, as a fairly independent person, I want to do everything myself. This is even evident in how I will make my own textures, music, sound effects, and sprites, when I could be using public domain works, a bad habit that often wastes time and slows down development progress.

Secondly, I do not want to inconvenience other people or set myself up for disappointment by asking them to help me out with projects they have no incentive to work on. With students this comes out more frequently, as the demands of coursework frequently override everything else, including good intentions and promises made at the beginning of a school semester.

Thirdly, it is not easy to ask good questions or get good answer. If I am a novice to the discipline of interface design and I try to speak with an interface designer, the suggestion I receive is going to be colored by the designer's current area of research, the blogs they've read over the last week, their misunderstanding of my question, my own inability to ask the question properly, and the fact that they don't remember what it's like to be a layman.

Lastly, the project I'm working on should necessarily be my very own. Allowing another individual to help me could prevent me from learning important concepts. The very last thing I wanted was to lose control of my learning process.

So it becomes important to my design process to write this note on human resources, to define the line between sloth and synergy, and to offer counsel to all future incarnations of myself on what it means to exist as part of a community.

No one designs, researches, or writes in a vacuum. We are all part of a greater community, and I must rely on one another and build off of each other members' expertise to advance what the human race is capable of. There is no shame in my 'reinventing the wheel' when I'm learning skills at the core of

my domain. But when it comes to obtaining a *breadth* of knowledge, I cannot get far if I try to learn each and every piece of information depth first.

A friendly expert is able to help orient us, point us down a good path towards our goal, teach us new tricks, help us parse terminology, and in general expand our understanding. In fact, frequently an expert's most important utility is in helping us understand the options available to us, or to identify the ground on which we are currently standing.

2 Deciding on the Board

The first thing I told The Captain was that I was interested in creating some kind of toy. I told him about the Arduino kit and the Dragonboard, and then confessed that while I wanted to be the one to pick my own board and define my own project, I didn't know what to look for on the board spec sheets. I had no frame of reference; they were radically different from any other hardware I'd ever purchased.

The Captain had me explain my basic idea to him, and the provided me with my very first foothold into the realm of physical computing.

The thing I had to pay attention to, he identified, was memory. Embedded systems were very small compared to personal computers, and frequently had little memory. He said that Arduino boards in particular were memory poor, and that vision libraries were memory hungry.

I was confused by this revelation, because I knew the recommended kit for the class used an Arduino board to talk to a Microsoft Kinect. He was able to quickly interpret that (most likely) the board in the kit would remain connected to the PC, which would then do the heavy lifting of processing the Kinect input. Either way, he knew enough about vision libraries to explain the Ardunio's paltry memory wouldn't be able to handle my custom project.

Aha! A stepping stone. When I heard how widely boards could vary in memory, I realized that each was customized for a specific purpose. Aftewards I was able to look at the Raspberry Pi and realize the little board was specialized to be a miniature computer. In fact, many of the 'mobile phone' developer boards, like Quallcomm's DragonBoard, had on board GPUs and interfaces for connecting screens. I did not need any of this functionality.

What then did *I* need? How did tiny boards communicate with wheels and motors?

Turning back to The Captain, I explained my findings and asked him how boards could be used to drive motors. He responded by ushering me up to his computer and quickly showing me a video in which a single wire was used to communicate with a digital servo. I had never heard the word 'servo' before except as a word fictional Transformers used in lieu of 'hands.'

A servo was a motor. What was an actuator? A port? A parallel port? A serial port? Why was the Raspberry Pi so cheap? On question lead to another, and quickly I was on a path far away from being confused over Arduinos and well on the way to identifying the board I needed. A discussion of all of these attributes led me to go back and examine the DragonBoard's competition.

Both PandaBoard and BeagleBoard were described in a similar way to the Raspberry PI; they were low-cost mobile development platforms, heavily immersed in an active open source community. The PandaBoard and BeagleBoard were still outside my price range and far too large for a toy. I was a little nervous. I didn't really know how to make a decision just yet. I hadn't yet seen the hexapod and I didn't quite know what I wanted. Perhaps, I thought, going with the Ardunio kit was best. I'd at least be able to do interesting things with augmented reality, and that was familiar...

To ensure I would be able to do what I wanted, I began researching robot tutorials and per-assembled kits. I found guides that could run robots on Arduino boards, and was thrilled that no matter what board I worked with, I was going to be able to build a robot.

The next day The Captain had me looking for a certain board which had gone inexplicably missing. He wanted me to look at it because he felt it would be useful for my project, and described it as white. After searching high and low, left and right, here and there, near and far, I opened up a box and pulled out the board, only to find a second board hidden beneath it. White, with two long strips of black ports, and without an HDMI port in sight, was a lovely little BeagleBoard BeagleBone. This was the smaller and more focused cousin of the larger BeagleBoard, andhad previously gone undiscovered in my research.

I was online within minutes, researching the strange white block in my hand. With its philosophy of 'bring your own peripherals', the BeagleBone was a credit card-sized, low-cost, powerful little computer. It could do almost anything, depending on how I wrote my software and connected devices. Where the average Arduino had kilobytes of on board memory, the Bone had megabytes. I didn't have to be an expert in robotics to understand how significant a jump that was; that was the difference between retaining a handful of string and running a full-blown 3D iPad application.

Even though I didn't truly understand the little board just yet, how it worked, or whether or not its other attributes were aligned towards my purposes, I knew well enough not only to trust my own research (the BeagleBone was a smaller and more mobile competitor to DragonBoard, and I had just found a ton of literature that described the BeagleBone as an excellent multipurpose board for beginning projects) but also The Captain's expertise. As he explained the board's serial ports to me later that day, I realized just how how adaptable the little machine was, and that it was well-suited to meet my needs. Trusting his judgment, I adopted the board as my own and brought it in to see my teacher.

My teacher was the other side of the expert human resources coin. He had many questions about the board; questions I couldn't answer but which provided me insight into what sorts of questions were important to ask. He was concerned the lack of Arduino's super-convenient IDE would force me to spend too much time on unnecessary 'engineering' components, but with access to both experts I was pretty sure I could handle myself. Plus, that sounded like a direct challenge, and goodness knows I love a challenge. He also wanted to know if the ports were parallel ports or serial, and if they could drive analog motors or digital ones. I had no idea on the second question, and my understanding was still too fragile to answer the first, but I was pretty sure the board had serial ports.

While the teacher gave a lecture on computer components like diodes and resistors, his questions

made me simultaneous nervous and yet brave; All of these words were so incredibly new to me, and yet I was starting to get a basic handle on each of them. Realizing that I had absorbed and retained new knowledge made me excited.

That was the evening I re-discovered the hexapod, and realized that almost everything I'd imagined about my project was wrong. The only thing that was absolutely *right* was the tiny little board sitting in the palm of my hand. The BeagleBone's serial ports were going to have no problem controlling so many legs or running a complex movement algorithm. It was also going to have no problems handling a detailed sound processing library. There was no time to be fooling around with uncertain memory, complex work-around, or risking the unacceptable possibility that I might have to plug the robot in and restrain its mobility.

I finally knew what my needs were, and the BeagleBone was going to meet them. Perfect. I could wrangle with some difficult low-level programming and computer concepts in exchange for the ability to build my robot. Cradled in the palm of my hand was Chloe's new brain; the most robust, versatile, and affordable brain I could give her.

Time to get my hands dirty; there was a great deal I needed to learn about my board in a very short period of time; and all those acronyms weren't going to decode themselves.

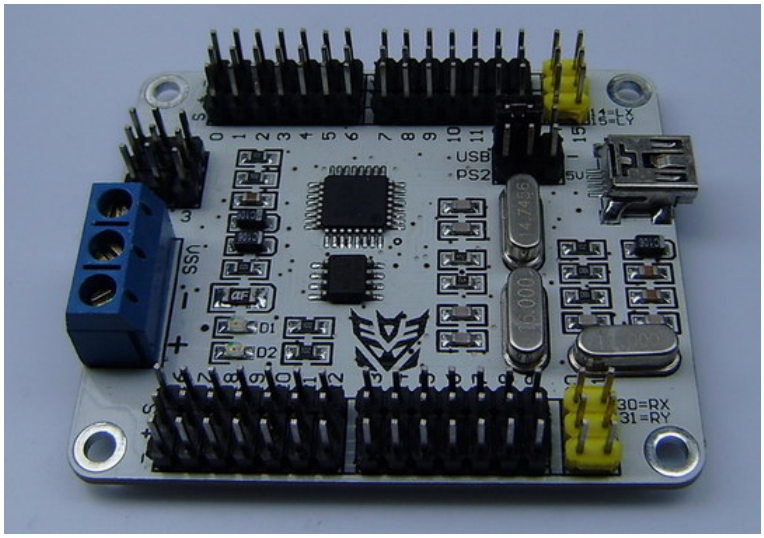
3 List of Parts

1. BeagleBoard BeagleBone

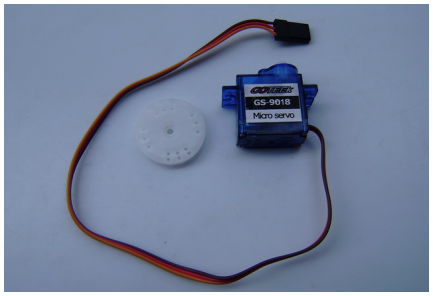
1. MicroUSB power cord & communication
2. White, Revision A5
3. 3.3 Volt system
4. Thorough documentation, community, and guides at www.beagleboard.org
5. Linux distribution

2. Control board for direct interface with servos

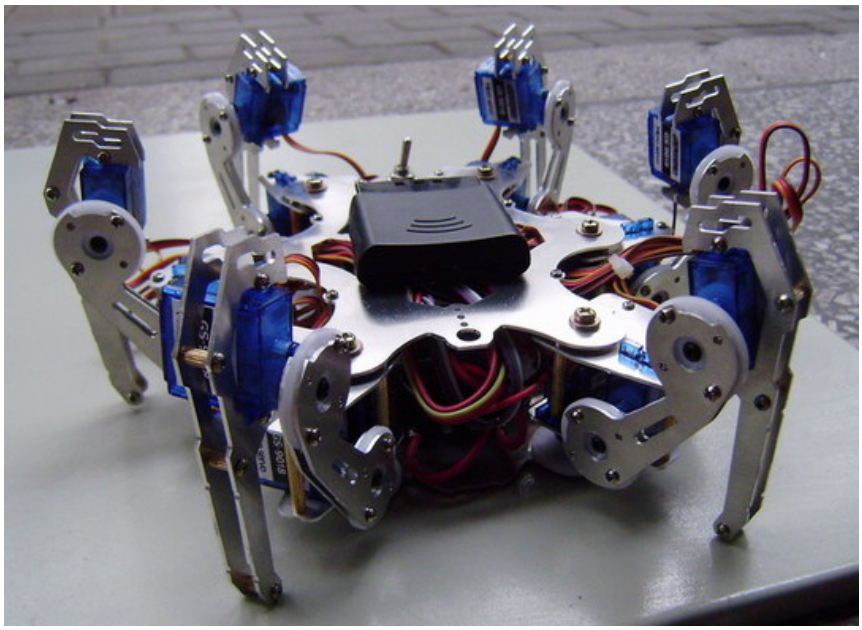
1. Printed documentation in Chinese
2. 5 Volt system
3. UART driven, capacity of 32 analog servos
4. MicroUSB power cord & communication
5. Specialized for RC



3. Robot kit and servos



1. 18 analog servos, 3 per leg
    1. 5 Volt
2. Chasis
    1. Top and bottom pieces for central chassis
    2. 12 lower leg pieces, 2 per leg
    3. 12 upper leg pieces, 2 per leg
    4. 12 shoulder pieces, 2 per leg
    5. Assorted screws, nuts, cylindrical bearings, support bars
3. Printed documentation in Chinese



4. Windows PC

1. Windows 7
2. Running Oracle Virtual Machine
    1. Running Ubuntu Desktop Linux distribution for ease of communication with board
3. Eclipse IDE

4 Additional Research

As I didn't want to make any decisions I'd later regret, I spent a day after discovering Chloe's board and body before committing to purchasing either. I used these days to gather information, and learned

about competing or related products.

For example, In studying Beaglebone's 'open hardware' I came across the Raspberry Pi, which in some lights could be seen as competition to the BeagleBone. They are some of the neatest, smallest, and least expensive computers a person could get their hands on. However, Pi's are meant to replace true desktops. They have a GPU and are meant to be connected to screens. There exists an add-on module with serial ports and a joystick which makes them valuable as hand held gaming devices, but they are not designed necessarily for driving actuators.

Getting the correct robot kit (or building one from scratch) was also a matter of solemn importance. A hexapod robot (or an octopod, which would also be interesting) requires a certain number of legs. These robots need motors that can support multiple degrees of freedom for two joints per leg. These motors are also called, as I am learning, ‘servos.’ A perfect 'spider leg' would have three joints per leg, but two seems to be the norm for hexapods. These legs are the most important components to the robot kit, as there really isn’t anything else to the physical robot aside from the frame itself, unless one starts to consider various sound, ultrasound, vision, or touch sensors. In a self-made kit, the frame, or 'chassis' would have to be fabricated by a company in Hong Kong. The servos would have to be purchased independently.

While I eventually went back to the very first robot kit I fell in love with, I looked at many other options before making my purchase. For instance, I found a set of servos being advertised for Hexapod robots at 3-5 USD a piece. This was too expensive for me.

I also looked at the Arduino Alumin hexapod kit; however I was already disliking Arduino at this point in time, and the legs of this bot seemed fragile and susceptible to bending or breakage. One of the reasons I picked Chloe's chassis was for the double tip of her lower legs.

After examining several similar kits, including ones covered in sensors with legs that were designed to sense contact, I realized that the customizable, basic nature of Chloe's kit was what made it desirable. I didn't need the distraction of all that additional functionality, and I would personal purchase any piece I did need. I certainly didn't need an ultrasound detector just yet; I'd busy myself with getting

her to walk.

In my research I stumbled upon perhaps one of the most important of my discoveries, which was the eerily realistic octopod *Robugtix*, made by the Amoeba Robotics Ltd. Robugtix was advertized as running on a proprietary Inverse Kinematics Engine, which was what made it move so smoothly and realistically. This will end up being a vital ingredient to *Chloe, Part III* once basic motion and sound have been taken care of.

V Project Management

1 Documentation as a Design Methodology

For me, documenting my design process is a vital part of my design methodology that helps me combat negative emotions like anxiety while providing me with an in-depth look at my progress and the time it takes for me to complete tasks. It is because of this thorough documentation that I was able to reconstruct my project log for the documentation phase. In fact, I would argue that all previous documents I've ever written had been 'tampered with' by virtue of the fact that I wrote them only with my end results in mind. This particular document is different, however, in that it preserves much of the essence and feeling of what my exact development process was like.

2 Project Schedule

Chloe, Part I is held to a rigorous schedule laid out by the tight time frame of the ten-week quarter here at SCAD. The syllabus allows for more time to work on *Part I* than it will for both *Part II* and *III,* however a significant chunk of this time was spent deciding on a board and learning the raw basics of how to interact with it.

Here is the schedule for *Part I* as per the course syllabus:

24 September – Project A assigned

27 September – Due: Project Ideas

3 October – Due: Project Pitch (formal)

8 October – Due: Prototype

10 October – Due: Documentation

15 October – Due: Presentation (Midterm)

3 Project Log

17 September – Day one and I am getting an early start thinking about my project. Why wait? I'm a complete layman when it comes to physical components, and I need all the head-start I can get.

18 September – I explain my project idea to The Captain after preliminary research.

19 September – I meet the BeagleBone for the very first time.

24 September - I bring the BeagleBone in to class; it's the second class period and I feel ahead. Then again, I did have a week to find the board I wanted. I'm pretty confident this is the board I want, but the teacher asks a lot of questions I'm not sure how to answer.

Towards evening I see the hexapod for the very first time, and I know that's the project I want to work on. I'm sold.

26 September – I am studying every pin and port of my board. It was an adventure just trying to figure out how to start the Ethernet connection with it the first time! The board contains schematics and

other fun stuff. I do an Internet search on every acronym and write detailed explanations of my findings so that I can remember the relevant ones. Afterward, I use my own words to explain what I've found to The Captain in a few text messages, and he augments my explanations, validates them, and helps with some terms I could not find.

27 September – I thoroughly explore two plazas north of Sham Shui Po's Cheung Sha Wan. After an exhaustive search, I manage to locate three- and only three- shops that sell the types of boards and components I or my classmate would need for our projects. I am very happy to have found them!

28 – I am still learning everything I can from my board.

30 September – Robot parts arrive! I have been working through the 'getting started' material available for my board. I am still studying schematics, reading documentation, and all-in-all probably getting *much* too prepared before getting my hands dirty. Now that the robot parts are here, I'm hoping they will motivate me! Unfortunately, I have no instructions for assembly. I still take them out and experiment with the motors.

1 October – I install a Virtual Machine so I can run Linux on my computer. I start it up and start installing everything I'm going to need. The Ubuntu download alone takes forever. I stay up long into the night trying to install a different Linux distribution (Ubuntu) on my board and then successfully communicate with it. It takes forever for me to understand what I need to do to get embedded Ubuntu on the machine, but when I do it ends up paving the way for me to understand a whole lot of other things about my board.

2 October – The captain helps demonstrate how my servos will work by hooking them up to an RC controller. I am a novice to Linux, so doing things like setting up eclipse is crazy.

3 October – Instruction for assembly of robot still have not arrived; I begin assembling the robot anyway using two images of the bot and eyeballing it to guess at which screws are used where, and which pieces had to be assembled in which orientation. I start with the shoulders because they are easy to see. I realize the robot's name is and in fact has been Chloe in my head for quite some time. I play with Linux in the evenings.

4 October – I assemble the shoulders and legs independently, but I do not yet join them together. The screws for joining the servos to their respective disks will slightly distort the plastic, so I do not want to install the legs until I have the chassis fully assembled and I have made some decisions about what I want the range of the legs to be. It occurs to me that it is up to me how to install the legs, and that my choices will determine the range of motion.

5 October – The instructions arrive. I've done everything right. What a relief. I assemble the central chassis. I've realized there are two different orientations for the final leg segment; the servo can face 'down' or 'up.' The images and instructions for the robot all have the orientation built 'down,' but from what I can see this causes the top end of the leg to bang into the shoulder of the robot. I reconfigure one leg. I am concerned the leg may not be as strong in this configuration, so I will need to wait until I can test it before reconfiguring or installing the other legs.

6 October – After frustrated wrestling with the BeagleBone for days trying to figure out why my program wasn't running, I finally managed to execute a C program and get it's LEDs to light up as a result. Yahoo!

7 October – I get the legs to move for the very first time. I am *so happy*.

8 October – I present my demo of the legs moving in class, and finish complete assembly of the robot in class. Amusingly, despite the pains I took to ensure I didn't install the legs 'incorrectly,' the fact that I built most of this robot with no instructions, and the careful calibration I did to ensure that all the legs has the exact same range of motion, I actually installed the legs while the chassis was upside down. Alas. Well, maybe I like it better this way!

9 October – I write the documentation for the presentation on the 10th

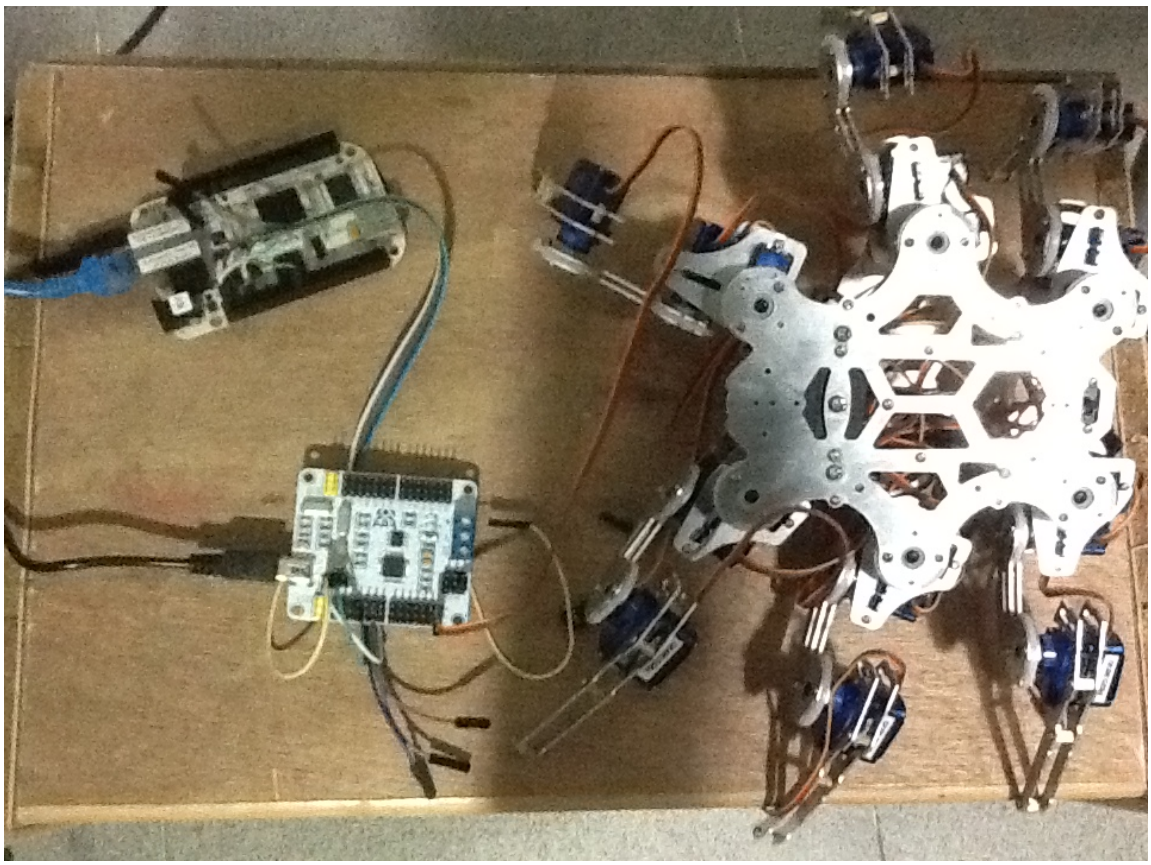
10 October – I present and submit the documentation.

11 October to 15 October – Debug problems with leg movement, set up the pinmux so that I can access the legs with code instead of just terminal commands, and get Chloe to stand for the very first time.

VI System Design

Chloe's system design is relatively simple at this stage. At the conclusion of *Part I,* she will exists at the intersection of five components. These are:

- The virtual Linux development machine, which can communicate with her via SSH through terminal and test her capabilities, program for her, send her code, and execute her on-board applications.
- The BeagleBone, which serves at her brain. The BeagleBone is a stand-alone computer, and will eventually be able to run without the need of the development machine. Several documents detailing the BeagleBone's schematics will be included at the end of the document in an Appendix. The BeagleBone currently runs an on-board embedded version of Ubuntu, which was installed in lieu of its default Angstrom distribution.
- The control board. The controller board takes care of a number important functions for the BeagleBone, acting as a 'shield' between the main board and the peripherals. It is basically a communications hub that takes in instructions from the BeagleBone and supplies them to the peripherals (the servos) as needed. It also allows me to translate between the BeagleBone's 3.3 volt communications and the servos, which expect 5 volts. The control board has no real



operating system or brains to it.

- The chassis (+ legs). The chassis or body of the robot, in addition to its legs, is designed to serve as housing for the servos and boards, and also as the structural components of the spider.
- The servos. The eighteen servos are the muscles and joints of the robot. There are eighteen servos which are connected to and powered by the control board. They are an integrated part of the legs, holding together other structural elements at the joints. The servos cannot be removed without disassembling the robot's legs.

The software behind Chloe's behavior at this point is current unremarkable, consisting only of a single C++ document, and is expected to expand immediately and considerably upon the start of *Part II,* and hopefully before the final presentation of Part I when Chloe successfully stands for the first time.

1 Implementation and Debugging

a Introductory Exploration of the Board and Its Abilities

I plug the BeagleBone in. At first only the power LED comes on, and Windows tries helplessly to find a driver for it. After a small duration of time in which the USER LEDS refuse to respond, I slip the MicroSD card into the board. Suddenly the machine comes to life, and “BeagleBone” shows up as a drive under My Computer. Another thing worth noting is that I am at school, and the computers here freak out feeling unsafe access might be occurring. I push “OK” and things seem to be working. In order to proceed I will need to install the BeagleBone drivers; so I log out and back in with system admin rights.

I am currently trying to get the ‘browsing’ capability to work. I don’t really understand it, but apparently I’m supposed to be able to make an Ethernet connection or network over my USB? The file on board my Beaglebone says that I need to eject the device to get this to work, even though the Getting Started guide says this is only for old system files. I guess even though I have the new driver on my development machine, this little blurb is referring to the files already on board the Beaglebone? Alright! I ejected it and now it says we have an Ethernet connection. Headed to 192.168.7.2

I am now watching the slides on board the BeagleBone and looking up terms. My teacher recommends Debian because it is an old well-established distribution oriented towards servers. He says this makes it more desirable than say Ubuntu or Fedora, as they are more targeted towards an end user desktop experience. QNX is a real-time (handles interrupts from hardware input immediately instead of queuing them) operating system, very stable, that is used in mission critical operations. QNX also has a tiny footprint. It is most suited for small, pure hardware operations.

What operating system should I choose to work with? Angstrom very small and scales down to devices with 4MB storage. It has less storage space, and smaller memory footprint, but doesn’t perfectly mirror standard behavior. There is less documentation for it, and debugging can be a problem. Debian was recommended by my sister

If I stick with Angstrom, I get access to some useful tools for developing on my board. It turns out there is a Cloud9 IDE which would allow me to use Bonescript (a variant of Javasript) to program directly from the web browser onto my board. This is similar to the system Arduino uses and certainly user friendly.

However, my past experiences in life have all pointed to the conclusion that the native language of engineers is indisputably C/++, and that if one wants access to the funnest and juiciest libraries available, the easiest way is to speak plain C. Marshaling between the languages can end up being a real pain. Fortunately, I know after a quick survey that the libraries I'll need are indeed in C. As C is the language I'm most intimately familiar with, and I have no particular interest in Javascript, the choice in language is easy.

The documentation also mentions that it is particularly easy to cross-compile using Linux. I don't know what that means, but it tells me there's a workaround to using Window through the program

Netbeans. According to the documentation, Netbeans allows users to write the code on desktop, save it in a location accessible to the Beagle, and then automatically compile it on the Bonee itself using ssh and the built in compiler on the BeagleBone's OS.

Er. What? Lots of terms, there. I mean I understand the majority of all of them, or at least I've been exposed to them. I've been exposed to SSH through using Git, and I sort of knew there was a built-in compiler on the BeagleBone... Maybe I should make myself a Linux machine to avoid this 'Netbeans' thing? It says I can use GDB for remote debugging over ssh. Since I don't know exactly what that means, I'll just have to make a mental check under, "I can use either Linux or Windows to debug programs on my board."

Looking at schematics of the board, I see things I don't understand. What is a PMIC Expansion Header? This is the only strip of black ports I don't really understand, so I ask the internet about it. PMIC stands for Power Management IC. According to the internet, these are integrated circuits which manage power requirements of these host system. They are often included in battery operated devices such as mobile phones and portable media players. Common voltages are 5V, 3.3V, 1.8V. There's a wonderful diagram of the chip itself on BeagleBone wiki, but I still have no idea what it really means.

The documentation and wiki mentions I need or have or have to update a kernel. That word is still foreign to me, something I dodged around while installing MacOS on my laptop and which I still don't really understand.

As I move to the next slide, I'm greeted with a long list of what I think are pins. Interesting! These are how I communicate with motors and sensors, right? According to this documentation, I have two sets of 46 pin headers. These pins are divided into P9 and P8 or Expansion A and Expansion B respectively, which amusingly means the higher number is the earlier letter, and the lower number is the latter letter alphabetically.

Long ways, on each side of the headers, I see the numbers 1 and 2 on one side, and 45 and 46 on the other. I think this means that the pins are named 1-45 and 2-46 and one side is odd and the other is even. Yes! That is the case. And the numbers across from one another between P9 and P8 don't necessarily have any relationship. These 'headers' are just a place to attach all the wires; they're not organized by purpose or function.

Let's see if I can research these different pin types and learn a little bit. What's the first type of pin the documentation mentions I have? 2XI2C pins (That's the latter 'I', not the number '1', for reference). By referencing the internet, I learned that it's named this way because it's an inter-integrated circuit bus (two 'I's). It is apparently ssed for attaching low-speed peripherals to an embedded system, or communication between components on same board.

Looking at my board, I can see I have connections labeled I2C1_SCL, I2C2_SCL, I2C1_SDA, and I2C2_SDA. These also have a pin number and a connection number. What do the acronyms mean? I search farther. Apparently any I2C connection is made up of two primary wires, a SCL and an SDA. These stand for Serial Clock Line (or Master) and Serial Data Line (or Slave) respectively.
So, this is a serial port. Alright, I get that. What is it used for, motors? According to this it is useful

where simplicity and low manufacturing cost are more important than speed, such as reading configuration data, system management, monitor colors, volume, reading hardware monitors and sensors like thermostats, reading real-time clicks, and turning on and off the power supply of a component. One of their big advantages is that a micro-controller can control a huge network of over a hundred devices with just two general purpose I/O pins and some software.

Not motors then, I guess? Ah! After a quick online check, I find a 'servo driver module' which I assume is some kind of control board. This would connect to my BeagleBoard and allow an I2C pin to control a large number of servos. The Captain has already helped me place the order for my hexapod and control board, and now I think I know sort of how my control board will work. The board even has a wire for control, power, and ground.

By now I think I know a bit about I2C ports so I look at the next thing on my list of understanding: 5xUART. Hmm. What is that, University of Art? Unlikely. It stands for universal asynchronous receiver/transmitter. Its job is to translates data between parallel and serial forms. Data format and transmission speeds are configurable. It is used for serial communications over a computer or peripheral device serial port.

Eh. I heard the word 'peripheral,' but I also get the sensation that this has something to do with networking or wireless. As the online articles don't mirror one another, it's difficult for me to tell what's being said about the UART that also applied to the I2C, or vise versa. For instance, it says the UART Takes bytes and transmits in sequential fashion; then at the destination a second UART re-assembles the bits into complete bytes. If I'm reading this right, that means that a UART can't talk to anything other than another UART; which means whatever you're talking to has to have the same interface installed. Okay. Is that... odd?

A UART either has some hardware or software that does a lot more than the I2C as well. According to what I've read, the UART contains a clock generator, input and output shift registers, transmit/receive control, read/write control logic, and possibly some buffers and FIFO memory. It can detect errors and mind some of its own business so that the CPU can spend more time on critical real time tasks. Again, while all this sounds present, all I can hear in my head is 'serial' and 'how exactly is this different from IC2?

I'm lucky The Captain explained what 'Half Duplex' and 'Full Duplex' were while we were walking yesterday, because those terms have just cropped up. According to this, a UART can be Simplex, Half Duplex, or Full Duplex, and it is used when compatible interfaces are required for things. The examples it gives, like RS232 and RS485, confuse me. I guess you don't learn a lot when you enumerate your options and facts; you really need to see something in action to be able to use it, or have it explained in a narrative way relevant to your interests. It does mention that UARTs function well in electrically noisy environments and over 'long' distances. I'll try and keep that in mind.

Based on what I've seen, it looks like both UART and I2C can be used to control peripherals like servo motors, and a quick search proves that this assumption is correct. I can also find UART control boards. At this point I realize that you can use any signals for anything you want; the questions are 'what do you need to do?' 'what quality do you need? and 'what are the lowest cost means available to you for doing it at that quality?' It seems to that UART connections either exist in or can be used with direct

connections, wireless, or USB.

I finally find a forum which is able to explain to me on additional big different between UART and I2C. UART is design for peer to peer communication, whereas I2C is for general broadcast. In short, in order for UART to talk to numerous peripherals, it needs a 'shield' or control board between it and the peripherals to decode and split the signals. I2C, however, can talk directly to a very large number of peripheral devices, as long as they all have a means to receive its input. However I2C, while designed to reduce number of pins and gate count, isn't in as widespread use as UART, and many more devices are UART compliant.

I think I finally get the difference between the two. I just saw a servo with a circuit embedded into its side. The servo itself would be able to decode a signal from I2C without a control board in the way. However if my board has no integrated circuit, a control board is necessary anyway. That's why my hexapod kit comes with one.

There are some other pins, but now I'm starting to get the hang of how these work. They're just different standards, different protocols that have been created to facilitate different things, and which all have different devices created for them and different adoption rates.

For example, SPI stands for Serial Peripheral Interface. It operates in full duplex mode, which means it's good for two way communication, and takes 4 wires. It has higher throughput than I2C or SMBus, and protocol Flexibility. It also has simple hardware interfacing, lower power requirements, and doesn't need precision or a master clock. It also doesn't need a unique address, but unfortunately it requires lots of pins, can only handles short distances, and is prone to noise spikes with no error-checking to fix it.

Then there is the CAN, which was really hard to research because 'can' is a very common word. It stands for controller area network and is a vehicle bus designed to allow microcontrollers to communicate each other within a vehicle without a host computer. It uses a message-based protocol; specifically for automotive applications, aerospace, maritime, industrial automation, and medical equipment. It assures message delivery, non-conflicting messages, time of delivery, low cost, EMF noise resilience, redundant routing, and other characteristics. Based on what I'm reading, it's incredibly robust- which must come with a tradeoff somewhere.

Although this sounds like the type of thing that could control a robot, I don’t see any examples of it working with servos. From what I can see it is consistently used with vehicles and medical applications, which means it is present here more out of a need to make the BeagleBone widely useful for a large variety of design applications.

Lastly there is also a large number of GPIO ports, which I am to understand mean general purpose input and output.

2 Moving On From Pins and Headers

By now I am starting to build up a vocabulary of terms that I can use to describe my board and its

capabilities. I hardly understand everything just yet, but I'm tired of reading and parsing through and conducting searching on documentation. This phase of my relationship with the board is over and I'm ready to move on. I shouldn't undervalue what I just did over the last few days, however, because the things I can now talk about in relation to my board have expanded tenfold. Each confusing term and lack of clarification gave me more questions to ask of The Captain and the teacher during our breaks, and each added further to my understanding.

Looking at all the pins helped me understand a lot more about my board. It also allowed me to understand what I was getting from my 'shield,' that is, the control board for my servos. I2C shields were more expensive which is probably why they were not packaged with my kit. What I have in hand now is a UART-driven shield with the ability to drive up to 32 servos.

Now that I have the robot's physical body, I have to get the board to a development state. That means installing the operating system of my choice, setting up the development IDE, and going through one programming loop so that I can get an LED blinking of my own accord on screen. Because I am choosing to work with sound processing, I will not be using the Angstrom operating system, but rather I will move to Ubuntu.

I ended up choosing Ubuntu at the Captain's recommendation. He explained that it would be possible to move the desktop GUI, if it was even bundled with the embedded distribution for my board, and explained that anything I could run on Debian, I would be able to run on Ubuntu; but there would be more libraries and other resources available for my use because Ubuntu is more prolific.

Still there are a few more things I need to understand before I can dive in; because I don't know how to dive in! I need to know what it MEANS to start developing on the board. For instance, I need an operating system. How do I install one? There are also some slides of important information on the board that I should hold on to, because installing Ubuntu will render these things unavailable to me.

3 The Boot Process

The next intimidating slide I've come across and need to understand is the “Typical BeagleBone boot process” slide. It's very fortunate that I have experience with installing Mac OS on a Dell laptop! Basically every computer needs a booter. This is a very low level program that does little more than offer preliminary instructions on what to do when the computer is turned on.

a Step 1

According to this slide, step 1 of the boot process is: ROM loads u-boot SPL (MLO). Not sure what all the acronyms are for, but based on step 2, this is a prelude to the booter (and not the booter itself). The default location is /media/mmcblk0p1/MLO, which is gobbly gook I only need if I have to go track it down and fix it or install a different tool. “It performs external DRAM configuration,” means that it basically makes the system memory available for use.

b Step 2

This step says that u-boot SPL (MLO) loads u-boot, default location at /media/mmcblk0-1/u-boot.img. Img files are, to my understanding, bootable, and they're what I'll be writing with Win32DiskImager, a

program recommended to me by an earlier BeagleBone slide.

c Step 3

This step says u-boot executes default environment/commands. I am guessing this is basically a configuration step, based on the fact that the default location of what it's talking about is /media/mmcblk0-1/uEnv.txt, which suggests a text or config file. It says “By default, 'uenvcmd' variable is executed. I don't know what that means.

d Step 4

Commands load kernel. The way The Captain describes an operating system is as follows: the kernel, the shell, and the GUI. The kernel, as he describes in layman's fashion, handles the main and most base-level 'loop' that keeps the entire system running. Like main(); in a C++ program or my actionscript gameLoop();s . The shell he describes as the part that is able to interpret commands, such as the terminal in a Mac; commands like ssh, git, mkdir, cd, l, etc. The primary GUI are the visual components that separate the user from having to interact directly with the shell, and more or less provides mouse support.

e Step 5

This step is 'kernel reads the root file system,' which I interpret to mean that this is the step in which the kernel determines what to do with the rest of its life. This is where programs are scheduled for execution. Wow, that sounds so morbid out of context...

4 “Baseline Software...”

I'm starting to get slightly nausous with too many new terms. Looking at the linux page of the BeagleBone documentation, I think the author wanted to inform me on the details of every tiny nitty gritty step that could be used to cobble together a Linux distribution on the BeagleBone from scratch, but all he succeeded in doing was making me feel like he'd given me an ulcer. This documentation is *way* too heavy for a getting started guide. I suppose the author assumed only an expert would want to change their distribution from Angstrom? Why? Just because I'm not familiar with Linux or embedded systems don't mean I like to go with the flow, here!

I see the wiki linux page contains examples for installing a toolchain, building a BeagleBone MLO and u-boot, building aBeagleBone3.8-rcX kernel, and building a kernel module (which they made a hello world with). But I have no idea what any of that means.

a u-boot

It is hard to know whether or not to capitalize 'u-boot' when it starts a sentence, as it is a proper noun whose first letter is intentionally left lowercase. Well, anyway, the first piece of software is u-boot itself, which can be obtained from a git repository. Something that should be noted is that u-boot hasn't had a 'release' since 2009, but it's had commits made to its project as little as 10 days ago, suggesting more support is being added, and bugs are being fixed. There is a repository for BeagleBone specific u-boot.

b Linux Kernel

Sounds important! I probably already have this, as Angstrom is a Linux distribution, but it's still a good idea to keep track of where the Beagleboard-patched kernel is.

c OE-core based BSP

You've got me, it's not clear to me what exactly this is. The link the documentation gives leads to a place that says: “Layer containing TI hardware support metadata” or “meta-ti.” TI, or Texas Instruments, is the provider of the BeagleBpme processor. Unfortunately that's not exactly enough to make an informed description of what this is.

OE means “Open Embedded” which is apparently in collaboration with “Yocto Project,” which is a Linux Foundation project, and which I am also seeing in the same sentence as “Yocto BSP layer for Beagleboard.org platforms.”

Aha! I found a blurb about Yocto: The advisory board includes members from several key silicon vendors, embedded Linux suppliers, and the OpenEmbedded community, including representation from TI, Intel, Mentor Graphics, MontaVista, TimeSys, and Dell.

Now I've found a definition for BSP: Board Support Package development tools for embedded platforms. Alright, so let's work backwards. There is a community called Open Embedded which created a support package for the embedded system I'm using: The BeagleBone, and this 'BSP' takes the form of a layer of metadata that describes the Texas Instruments hardware on the board.

d Mentioning “Linaro”

Linaro is something the slides mentioned just before explaining Yocto. Its blurb says that Linaro is a not-for-profit engineering organization consolidating and optimizing open source Linux software and tools for the ARAM architecture. The slide says that it provides a compiler, debugger, profiler, kernel, and middleware, and validation of efforts done with Ubuntu and Android.

Aha! I have found out what Linaro does that I should care about. I went to downloads and saw “Linaro Engineering Builds” which led to an FAQ about what a “LEB” (Linaro Engineering Build) was. According to this, LEBs are full system builds of popular Linux open-source products that come with Linaro improvements per-integrated; currently products supported by the LEB program are Android, OpenEmbedded, and Ubuntu.

It goes on to stay that Linaro provides engineering builds that come with significantly different priorities and requirements than the product and end-user focused distribution business. As such they focus heavily on middleware, kernel, and hardware enablement topics, and deprioritize safe upgrade paths, security, and stability.

There are two ways to get a Linaro build on your hardware, says this additional getting started site under Linaro → Engineering. The fastest way is to grab a milestone image and “dd” it to an SD card. (What does that mean? Direct download? Dragon dance?) These milestone images are pre-built using common combinations of hardware packs and root file systems. Ubuntu Desktop images include x11-base hardware packs, with the exception of Beagle. There is in fact a Beagle Ubuntu Desktop build

under Oneiric → Ubuntu-Desktop. Customization is explained on their wiki.

However, the “BeagleBoard-xM is one of their release platforms,” but not the BeagleBone. This explains too why Linaro offered a Beagle Ubuntu Desktop distribution for an embedded system; the BeagleBoard-xM has more functionality than the BeagleBone.

VII Switching the Operating System

By now I have done a great job of inventorying what the BeagleBone has to tell me about itself; however I still don't feel much closer to understanding what I should do next with the board. Feeling slightly lost, I know that the next big step is probably to get my Linux distribution of choice on the board. This will be a big step in my learning process, and a difficult one, but I think it's important. I talk to The Captain and he agrees.I start off with an online search...

1 It Starts...

I've installed Mac OS on an old Dell Laptop before, so I've had my share of Terminal-related difficulties and hacking. Still, starting off with the idea that I want to install Ubuntu on an embedded system is like jumping straight into the middle of an ocean with no idea which direction is north. What do I need? Where do I start? I understand some of the terminology, such as 'booter,' but every resource I look at seems to conflict with every other resource.

Do I need to compile my own toolchain of... what? Huh? What does that even mean? Do I have to update the booter? How? Does the booter come with the distribution? Why do I have to package the booter with the kernel with the... isn't there an installation for this? How would I possibly run the installation? Do I do it with the USB plugged in? With it plugged in and ejected so the Ethernet connection is running? With just the SD card plugged in through a reader? Do I have on board MMC memory like the BeagleBone Black? What do all these commands do?

Mm. Yeah, this is going to be a painful learning experience, I can tell.

On the other hand, I was prepared for that. So while biting down on my temper, I begin the arduous task of combing through and comparing strange new information. I try to get at the root of what it is I'm trying to do, and getting frustrated that this often conflicts with what tutorials are attempting to teach me. Each one really does have *something* to teach me about my task, but each also comes with plenty of garbage I need to weed out. With the search query “BeagleBone Ubuntu Installation -black” still bringing up plenty of BeagleBone black guides, I have to make guesses about what applies to me and what doesn't.

One of the first things I realize I have to do in order to lessen my headaches is to get Ubuntu-Desktop installed on my development machine. I can do this through a virtual machine, so I go and download Oracle VM VirtualBox to help me.

Part of the reason I chose this route is that most everyone technologically savvy enough to be installing Ubuntu on their BeagleBones is already running some distribution of Linux, and so most of the tutorials are postured from that angle. I can find Angstrom tutorials from the Windows host machine perspective, but it is too difficult for me to tell what in these tutorials applies to me and what doesn't from this stage.

Another reason I choose to use the virtual machine is that familiarizing myself with Linux on one end of the conversation may help me with Linux on the other end. A third reason is that talking to the

BeagleBone on Windows requires more workarounds and side programs, and talking to the BeagleBone on Linux is easier and more straightforward.

And, lastly of all, one final reason I choose to install the virtual machine is because I feel like I have absolutely no idea what I'm doing, but I need to start somewhere, and I have a tutorial sitting right in front of me that starts off telling me to install one.

2 "Screen"

Things do not go as I had hoped. For hours I debug why it is the BeagleBone is unsuccessful in virtualizing two USB ports. At first I assume the two USB connections I see are the two ports, and I get confused about why the second one isn't 'working.' Then I realize one of the USB connections is supposed to manifest as *two* all on its own. I type in the command "ls dev/ttyUSB*" over and over again, with Ethernet mode on and off, having plugged and unplugged and toggled various things, but instead of getting a USB0 and a USB1, I get only the USB0. Typing out the tutorial's command 'screen' on this USB0 yields no results. I'm stuck.

I try a number of other tutorials, jumping around the net and surfing for answers to countless questions I barely know how to ask. Because I don't know what I'm doing, I'm not even sure if my board has MMC memory or if it has a user button or if it can be flashed. I end up following the tail end of a non-applicable tutorial, back-tracking, side-cycling, and all sorts of other fun travel-related words.

I'm frustrated, and I go back and take a deep breath and start looking at the tutorials from square one. I give up on the ones from the wiki page, which cause me no end of stress. They tell me I need to update bootloaders and pack them with kernels and... yeah we'll save that for a rainy day; I'm not sure what exactly is happening on that instruction page, but deep down I'm pretty sure it's telling me the long-winded, round-about, 'customized' way of doing things (Linux is a lot of do-it-yourself), not the way I truly need right now. At the end of my patience, there are a thousand routes to go down, and this long, complicated, and poorly-described (for a beginner) tutorial is not the one for me.

3 Retrospect – Past Tense

It turns out I was right. In looking back over a tutorial I *thought* required me to already get past the 'screen' instructions, I realize that the instructions are *most likely* working with an SD card that has been plugged in via a simple card reader. The reason I couldn't tell this earlier was because I was unfamiliar with basic Linux commands. As I'd never seen 'apt-get' before, and didn't know what packages were, I couldn't recognize that the tutorial's very first command was downloading some useful tools onto Ubuntu-Desktop, not interfacing directly with the embedded machine.

At last I recognize the tutorial for what it iss. The instructions are for BeagleBone Black and involve flashing MMC memory, but I have read somewhere that the Black can boot from MMC or Flash memory; and to me that means my good old fashion vanilla BeagleBone might be able to boot from a card prepared in this fashion. I just have to hope that what the tutorial teaches me to load into the card, exists in such a state as it doesn't *need* to be in MMC memory to be bootable.

After struggling with trying to figure out what distribution to try and where to download it from, I eventually manage to get a version downloaded and installed onto the SD card. Eureka! I plug the SD card into my board and plug the board in. Everything shows up differently! The installation succeeded! The very last instruction on thetutorial was to ssh to the board...

Uh oh. What now?

4 Finding Out the IP Address, in Retrospect

I think I must have blacked out, because I have no idea how I eventually figured out what my board's IP address was. I currently know that typing in ssh ubuntu@192.168.7.2 gets me to where I need to be. The Captain suggests that the BeagleBone's Angstrom distribution configured the IP address into the part of the board responsible for setting up the virtualized USB ports and Ethernet connection, and that the new Ubuntu distribution simply didn't overwrite this IP address.

Therefore, it is likely that the IP address that worked for me was in fact the same IP address I had originally used to find the BeagleBone in the browser, and that when I accidentally navigated to this location, and realized the BeagleBone was there, I tried ubuntu@192.168.7.2 and realized it worked, and then was overcome by an exhaustion that wiped all memory of this from my mind.

After hearing this explanation, I went and checked to find that the BeagleBone had indeed always been at 192.168.7.2. I make a mental note: Do not program past 2:00 am unless undesirable consequences.

However there is likely another way to find out my board's IP address should it ever change in the future. By typing in ifconfig on the virtual machine end of things, I can see that my et1 connection has an inet address of 192.168.7.1 and a Bcast of 192.168.7.3. This leaves only 192.168.7.2 as a space suitable for my board to reside in. Mystery solved!

After I ssh-ed successfully to the board, I tried unsuccessfully to ping www.google.com until at last I realized it was time to plug in the ethernet cable and that my USB did not provide access to the internet, despite the fact that I could connect to my board via internet browser.

5 Reflection on Installation

With me at such an entry level to all of this, a lot of what I have to do is by educated guess. Anyone who has worked with physical devices or Linux could have made these decisions instantaneously, without thinking, but for me it's like treading through a thick soup. So many questions are unanswered and so many 'obvious' decisions have to be made with little more than intuition. And in many cases, intuition is wrong!

However, each time I make a guess, I learn.

By the time of my recording these events, I can now follow any similar tutorial for any physical device without batting an eye. I know that 'bootable' means 'bootable' and that the only question is whether the architecture of the processor can support the kernel's low-level instructions.

Basically, the kernels for PC processors and embedded systems running on ARM architecture are different, and those differences are what differentiate between desktop and embedded distributions of Linux.

The difference distributions of Ubuntu are managed by individuals in the open source arena, people who love boards and love technologies and pack up all the ingredients a given board needs in order to run a Linux distribution on them.

VIII Stage Two: Cross-Compiling

I found an article which goes into lovely detail about why cross-compiling is necessary, and now that I understand how PC processors and ARM architecture machines are quite different, I can digest the

basic premise that 'normal' compiling breaks down programs into instruction sets for PC processors, and I need to set up everything to compile to an instruction set for my embedded machine. I can also comprehend that while it would certainly be possible to compile on board the BeagleBone itself, my PC has a lot more processing power to get the job done faster.

1 Setting Up the IDE

The IDE I will be working with is Eclipse, which I initially believed was unfortunate as I have no great love for Java and I much prefer the tools available through Visual Studio. Perhaps I associated Java with a bunch of confused programmers who never successfully learned C++ in my undergraduate classes and believed Java was the proper language to write triple A games in. Or maybe I had mentally associated Eclipse with Dev-C++, the ancient and monstrous creature I was forced to use during my undergraduate degree for writing basic C programs.

Whatever the case, my trepidation was unfounded. It is clear after downloading the program that Eclipse has both aged well and continued to mature, and while it might not be my familiar Virtual Studio, Code Blocks, or Mono Develop (Hey I have to work in Unity3D a lot), it is nevertheless quite a respectable, acceptable, and plesant-to-gaze-upon creature. I have resolved, therefore, to like it.

There were also some useful commands I have to enter now that the installation is done. I must “sudo apt-get install gcc-arm-linux-gnueabi” to get the correct compiler, following shortly on the heels of a “sudo apt-get install eclipse eclipse-cdt g++ gcc.”

I then go through the steps provided by my lovely tutorial, which details how to use the different view to talk remotely to my BeagleBoard and then work with code on my machine. I set up my debug configuration and my compiler settings (using the arm-linux compiler instead of the default gcc and c++)

2 Here we go again!

The first time I try to follow the tutorial, I can't seem to get the errors to stop. I have a feeling this has to do with the options I picked when setting up the program. See the tutorial told me to select 'Hello World' and whatever setup I wanted. However while I was looking at the interface, I saw their was an option for 'Cross Compile' under almost every option but the 'Hello World' one. I clicked on the Hello World program anyway, and no matter what I do it just won't compile.

It has trouble no matter what I #include at the top, be it iostream for cout or the corresponding .h for printf. All I want to do is print some text to the screen! Based on the fact that it can't 'resolve' the symbols, I think it's failing to find the libraries. Still, this shouldn't be so hard. That's it, I'm switching. I'm going to create another project, a blank C++ for Cross Compiling, and then I'll drag the main.ccp text file into it!

Alright, this is going much better. I'm not getting any errors on my side anymore. I've created a second project just as I said I would and I pulled over the main.cpp. There were a few bugs here and there but I've smoothed them out. It doesn't seem like my debugger wants to work, however. It keeps complaining that its suffering errors, failing to launch... arg.

Browsing through my board in frustration in the remote view gives me the first glimpse of my compiled program nestled safely in the confines of my /home/ubuntu/ directory. Wait a minute! How

did it get there? Aha! It is the remote debugger that is broken, not the compilation!

I SSH to my ubuntu directory and quickly try to run the executable, but I get a strange error. The computer says it cannot execute binary file. What? Why? At first I assume this is a problem back with iostream, and I go back and try to debug. But as the hours go by I wonder if this is a problem with the compiler. I start looking for all the libraries in my Linux installation, and where they're located. I look to see they're being properly included in my build. I search the internet for bugs, but for some reason I'm just not finding anything.

3 It Actually Takes a Very Long Time To Solve This

It actually takes a very long time for me to solve this problem. On the way I learn all sorts of commands. I learn cat, and file, for example, but these fail to tell me any information. File tells me that I have a 32-bit ARM, dynamically linked non-stripped binary executable. As I look across the internet it tells me all sorts of things, like that the most common cause of my error is that I'm failing to compile a 32-bit program from a 64-bit system. My PC is 64-bit and so is my Linux distribution, so for awhile I believed this. Then I realized it didn't make any sense.

Secondly the internet was concerned that I had created a non-ARM distribution, but armed with the knowledge that I hadn't failed to product the 32-big program, and that adding the string-m32 to my builds hadn't changed a thing, I was able to more quickly and easily disregard this as inapplicable to me. Arg. I try nonsense. I include things that are already included. I try small variations. I click multiple different parsers.

The surprise ended up being the 'dynamic.' Once, in an attempt to get things working, I had included the flag -static at the end of my build. However, doing so caused my terminal window to hang and eclipse to crash. When I finally rebooted everything, I rand the program and it said there was a segmentation fault.

Exhausted, five minutes later I couldn't remember that this 'segmentation fault' was the only indication I'd had that a program had actually run successfully, and that it hadn't run succesfully in any other build. Later I was compelled to try the -static flag again. Again it crashed everything, and again I got mad, until after hours of watching the build fail to launch I got the second 'Segmentation Fault' error. Wait a minute.

If I'm getting a segmentation fault, that means I'm accessing out of bounds memory. That's *weird*, given that I'm using a printf and cout statement interchangeably, but that means my program is *running.* Hold up, hold up, back track! What does that mean!?

I learned to use an important command, readelf, in a different way. For some reason all these binaries are in 'ELF' format, and while I don't know what that means, I can tell basically what 'readelf' is going to do. Before I had been using readelf -a; now I use readelf -l in conjunction with the online resource that told me about the static/dynamic problem. I see a line the tutorial told me I would see:

Requesting program interpreter: /lib/ld-linux.so.3.

I quickly navigate to that location and lo and behold it does not exist! But guess what does exist? /lib/ld-linux-arm.so.3. Holy crap! How do I link this? I find a command called 'ln' and use “sudo ln (target)(symbolic)” or “sudo ln `ld-linux-arm.so.3 /lib/ld-linux.so.3`” and I recompile my program dynamically.

Segmentation fault. It ran! Also, I think it is having a segmentation fault because I typed int main and didn't return anything. Man, is that all it really would have taken for me to notice the program was running earlier? Arg.

Wow. I succesfully cross-compiled, despite not even knowing what 'cross-compile' meant more than 24 hours ago. I feel great.

4 Onto the LEDS

At first I'm not sure where one begins to light LEDs. I ask around and I find a bunch of people lighting external LEDs. Okay, that's not what I need. Keep going. Eventually I started asking BeagleBone specific questions, realizing that there has to be something in the hardware, no doubt, that know exactly where those LEDs are. After my teacher's lecture on Linux distributions, I'm starting to understand there's a component of every distribution that's specific to the architecture; no doubt that's where my lights are.

I find an Angstrom distribution that puts one of the lights at /sys/devices/platform/leds-gpio/leds/BeagleBone::usr0/. I manage to find them at a similar location, and find /sys/devices/ocp.3/gpio-leds.7/leds/BeagleBone:green:user0 /

This can't be the only way to locate files; I hunted and pecked through every directory, and half of them are named the same thing. Let's see. Ah! I can use find -name “BeagleBone:green:usr2” 2>/dev/null (where that last part destroys error messages and logging) to find the LEDS. Surprisingly, there's a more direct route: /sys/class/leds/BeagleBone:Green:usr2.

This is useful information for me to remember. After all, the Angstrom and Ubuntu distributions are still both Linux at their core, and so anything that applies to one should be vaguely applicable to the other.

After finding the LEDs, I 'm trying to change their brightness as a tutorial tells me I can. Not working. Grr.

Whoa. My moment of understanding comes from elinux.org in an exercise about flashing an LED. The website states, “The easiest way to do general purpose I/O (gpio) on the Beagle is through a terminal window and a shell prompt. In Linux, almost everything is treated as a file, even things that aren't files.” Elsewhere I see this called a sysfs interface. Suddenly I get it; I can send commands to my board by writing into files! And that means something like an fwrite had to be used from C to accomplish the same task, right? Yes, I'm starting to get this!

Finally I try a sudo -s, followed by an echo 1>brightness. Then I see a better command, echo 1 | sudo tee brightness. Both work! Someone explains on a forum why my sudo echo 1 > brightness didn't

work; apparently sudo gives permission for echo to do its writing to buffer thing, but not for the pushing to the file to actually occur. Or *something* like that. The 'tee' I don't understand, but it gets me past that restriction.

I toggle the LEDs on and off. I am so victorious.

5 LEDS in C(++)

The LEDs now make sense to me, and I'm ready to use a sample fwrite tutorial to toggle them. I write up the program pretty fast. First I make a duplicate of my Hello World program, which runs. I delete it and continue; I even use strings and other C++ conventions to pretty it up. Maybe I'm feeling cocky, but a voice in my head tells me I might regret all my Object Oriented Programming if it doesn't run, and...

...and it doesn't run. Okay, what's wrong? I'm not compiling any differently than I was before. "Failed to execute MI command target-select-remote. Packet reply was too long." That doesn't mean anything. I play with some values and try to get the program to obey me, but it doesn't work. I get another error: "Filesystem input or output error. "

I try to put the mystery of these errors together, but they don't make sense to me: "readchar: Got EOF" or "remote side has terminated connection. GDBserver will reopen the connection." An online tutorial suggested to do a manual debugging launcher, but after trying that I just got more frustrated and more errors.

I begin stripping down my program to see what part was offending the board, but I can't find anything. I to run as root on the board, and see that sometime in all the hubbub, a LEDHello program has finally appeared, though I'm not sure when or how. Heh, maybe it happened on the second thing I tried! It isn't executable. Chmod didn't seem to want to let me change the permissions either. I wonder if perhaps the chmod line in the compiler settings is what messed everything up? It's the only thing different from my last program.

Well, if it's the only thing different, I guess it *must* be the offending line, right? I delete it. Ahhh, a big red error I'd been glimpsing for awhile rears its head after I delete the chmod line: warning: Architecture rejected target-supplied description. What on earth does that mean? I navigate to the little program, which notifies me that it definitely believes itself to be ARM. Wait! Is there a possibility that this is just the debugger failing to connect? That the two machines are having a 32-bit 64-bit or communications fight or something?

Perhaps my 'non-executable-executable' is actually executable? Perhaps its executable permissions just weren't properly set because

of my botched chmod line!

Yes! Victory! Okay, so the debugger is absolutely failing to hook up to the remote end, but at the moment I'm not sure it's my biggest problem. I can always try to solve that one later. The thing I should mention is that when I failed to chmod -x ./LEDHello, I wasn't sure what to do. I ended up using the remote SSH file exploration GUI in Exclipse to navigate to LEDHello. Right clicking on it gave me access to permissions, and nothing was set to executable (which explains why LEDHello was showing up white instead of the green I associate with executables in the terminal).

Now, even though the debugger is failing to hook up, I've navigated to LEDHello over the SHH on the terminal and ran ./LEDHello successfully. HUZZAH.

Hmm, it failed to open the files; I'm guessing that's because an odd character snuck in. I bet it has to do with '/' in a string. Oho, it turns out the problem is actually my char(i) for turning a uint to a char. Looks like I have to do a more complex conversion than that, as casting "1" to a unit yields a value of 49 and I bet 0 is up in the high fifties. That means I can't use this technique. I know of another way, involving string streams, but I don't really like it... Perhaps I should rest for now.

6 The LEDS Shall Work

I begin debugging on October the 7th by trying to figure out what was wrong with my NumberToString function. The autocomplete features aren't very helpful, so after a moment of internet searching I realize I need to add in a #include<sstream>. Now I'm able to convert numbers to strings, yay!

But My program still won't run. It doesn't think the file exists- ah, I accidentally put 'BeagleBone:Green:usr' in both the directory and file prefix! Hmm but now it still claims its failing to open the file.

It turns out that "r+", while working for other people everywhere, isn't what this program wanted. I change it to rw+ and now the file opens.

I discovered this by writing a function called smallTest in which I simply opened a file and reported back my success or otherwise. The first thing I decided was to test whether or not the C++ type 'string' was letting me down. I started by opening a file specified by a string variable or even a char* like the tutorials had done, I went with a straight string literal, figuring this would eliminate the largest

Unfortunately, my file would still not open. With the literal removing most room for error on that front, the only other input I could change was the "r+." Changing this to "rw+" reported a successful opening of the file. Seeing this, I copied my findings to the other functions and got them all to successfully open their own files.

Victory? No! Strangely enough, the LEDs still continue not to respond.

Using the terminal, ssh, and linux commands, I echo about on the trigger and brightness files, and learn to use the linux command "cat" in order to see what is inside the files. I turn off the triggers manually by echoing in the string literal "none" and now I'm musing over why the program can't do

the same thing..

Fopen had found the needed files and reported no failure in opening them... I think this means I simply *must* have found the correct files. But then why can'tI write to them? Was this a permissions thing? I try some sudo ./LEDHello! No dice. Alright, I'll set permissions aside from now and focus on something else.

Now there are all sorts of things could be wrong, but I'm going to sit for a moment and try to think of the 'simplest' most programmatic and easy-to-debug solution I could think of. What is the most likely reason for all this hullabaloo?

Probably the fwrite statements, which naturally come after the fopen statements. I've tweaked them to suit my own needs, so perhaps I've failed them somehow.

I browse about on the fwrite statements, and do some debugging, such as printf-ing out what the size of my strings are for disabling the LED triggers. I'm concerned they either include or fail to include some kind of EOF I ought to be worrying about. It turns out they have no EOF; they contain their characters and nothing more.

Now the original fwrite statement I'd examined in the tutorial had seemed to be all backwards, plugging in numbers in strange places (like the size of the string in the place where size(char) should have gone, and size(char) where the length of the string should have gone). I'd tried to fill in the fields in a more sensible way, using a string.c_str(), a sizeof(char), and a string.length(). But what if these functions are letting me down? I'm not trusting C++'s string to behave nicely with C's gritty underbelly.

The fact that the samples I found used char* instead of string gave me a place to start. Their 'r+' hadn't served me, but perhaps char* would provide some insight. There was also something else strange I'd done to the fwrite for toggling the LED's brightness. Hadn't I tried to write an integer into the file?

I go back into smallTest and ome upon a realization. I have been writing 0 and 1 into my LED files as *integers*, but using the Linux command 'file' on the brightness file reveales that they are ASCII text files, not binary files. I realize I probably ought to be printing in a character string of size one, not an integer!

I can't really debug the triggers anymore unless I unplug and replug the board in. Easy enough one supposes, but I decide to focus on the more observable problem of turning the LEDs on and off now that the triggers are disabled.

Now I use the low-level char* msg = “1” in my smallTest function . I still don't trust strings, and I'm not sure if the default functions they come with can be trusted.

Nothing. Drat. I change the fopen command to “w+” and throw in some printf statements.

It works.

Holy crap! The light is on! Eureka!

I do some basic debugging and fill in a whole bunch of new information in my other functions, and am able to successfully turn on all the LED's by tossing in a “1.”

Wahoo! Oops, I have to turn a 1 into a “1” and a 0 into a “0” or I won't be able to turn the lights off. I rewrite the setLED function to include a new NumberToString to do this transition, and I update the trigger function as well despite being unable to test it.

Hey, none of the lights turn off. Why? Is this a string thing? Is .length() letting me down? I printf it out

and search the web a bit. No, it looks good. Disgrunted, I typed out a big printf statement in which I print the new state as a string, then sizeof(char), then string.length(), and so forth.

The lights turn off. What? How? I didn't change anything! I run the program again, and the lights turn on and back off.

Suddenly, a light turnes on in my head as well. Wait a minute! It wasn't the “w+” that had fixed my program! It was the debug printf statements! Years of experience programming with Actionscript and grading C projects made of nothing but simple cout statement smack me upside the head. There was some kind of *timed event* going on in the background, independent of the linear and chronological nature of the code, that was arbitrarily completing or not completing depending on how fast the program was running.

The debug statements had slowed the program down- through luck, more than anything- and permitted the timed event to execute. And in this case, the 'timed event' were buffers.

You see, most print statement in C and C++ write to some kind of buffer or stream, and then wait patiently to be 'flushed' so that they can display themselves. I realized that this must have been the same case with fwrite(). That meant that fwrite() *was* correctly receiving all the data it needed to write, and that's why no errors were being thrown. It was writing *to a buffer*, and then I was fclose()ing the file before the buffer had time to flush. No error thrown. No output witnessed. No crash. A silent but deadly espionage.

A quick google search brings me to the fflush()function. I throw it in before the fclose() and take out the printf statements. I cross my fingers...

Victory. The program runs without a hitch, the LEDs turn on and off .

Hey you know what, now that I think about it, I really need to start up a GIT repository in my Linux Virtual Machine. Victory! I even managed to use Import->Existing Projects into Workspace and not check “Copy files” and now I have the git repository set up in a folder called MyDragon way outside of workspace where I can git them. I will probably transfer this file into that repository XD.

7 The Legs

The last thing I have to get done before demo day is getting those leg servos to twitch. I dont' have to get them running on C++ code yet, but I at least have to access them through the board. This is hard and takes a lot of searching, because as far as I can tell the serial ports don't work like the LEDs. Rather than being files with a set state, they're more like the streams I was working with the other day. They're channels. Information slides down them and disappears into the other side.

First I have to hook everything up physically though.

I make sure The Captain is on hand before I start trying to put anything together, and he makes sure I am using the images in my Chinese instructions correctly. Putting the robot's body together was so easy I barely felt the need to document it; all I had to do was screw things into place. Yet these boards sort of scare me. How do I connect components? What can I accidentally damage? What order do I plug things in? I carefully attach the control board to the BeagleBone and then the control board to the servo. The Captain promises me I will hurt neither myself nor the board.

In looking around the internet and talking to The Captain, I realize that the tty files under /dev/ are the tools I need to talk to Chloe's leg servos. I know I have to send a 'string' into those files, and I'm pretty

sure I can just echo into them. Bewildered but hopeful, I try to echo a “#0 P500 T500\r\n” > ttySO command into the ttyS0 file. It doesn't work, and I'm actually slightly disappointed. I try every servo number and every ttyS* number. Nothing works.

Darn. I search the internet and start getting frustrated with lots of complex terminology trying to explain really round-about ways for implementing physical components on generic or perhaps self-made boards. Arg, I don't know what question to ask. I start seeing how the BeagleBone Black is different from the White, but I'm not sure if my own internal kernel is upgraded to the same level as the Black or not. What I *do* see is a lot of people complaining that ttyO0 and ttyO1 are all missing. I also see someone else saying not to use ttyS0 to talk to the serial ports (S stands for serial, I believE) but to use ttyO0. Odd. I try ttyO0.

Something's been bothering me. Why are there only 4 ttyS devices, if there are 5 UARTs? Later I realize this might be because one of the UARTs is wired to the USB, but in trying to get more information about which tty is my UART1, I stumble across something that suggests any 'real' serial ports are usually archived under a directory with a long 'code'. Anything under a folder that says 'serial' probably doesn't exist, and is just there by default.

What does that mean? Is it true for my BeagleBone? Hmm. I get frustred and go in circles for a bit. What the heck is ttyS0? TtyO0? Are they missing because of a revision thing? Where are they now? Ugh this is awful, trying to figure out what's going on. I know the basics are that I need to read to a file. Let's see what I can do!

After much digging and wading through long lists of complicated explanations, I managed to find what I was looking for- or at least part of it. It's involves taking what I find at /lib/firmware/ named BB-UART1-00A0.dtbo (and 2, 4, and 5) and sending its results to /sys/devices/bone_capemgr.8/slots

The line that liberated me was at the very end of a very long-winded tutorial about the Device Tree and building 'capes. 'After a long discussion in pinmuxing (I still don't know what that word means) and capes (nor that) and all sorts of other wonderful topics, it gave me a splendid little line of yum yum that looked like this: `# echo BB-UART5 >/sys/devices/bone_capemgr*/slots`

Liberation.

Suddenly I have a ttyO1 port. I run it again for BB-UART1 and BB-UART2, and I get a ttyO2 port. At this point I am concerned that UART1 is routed to the USB and I have wired up my board to work with UART2. I push in an attempt at sending a command. Nothing. I try it again with slight variations. Nothing.

Frustrated, I seem to remember I have to initialize serial connections, something about baud rate. I look for some samples on how to accomplish this and find the line: stty -F /dev/ttyUSB0 9600. I try filling in my own parameters and write: stty -F /dev/ttyO2 115200

I don't get anything exciting back.

I try echo -en “#0 P500 T500\r\n” > ttyO2

I hear the servo move.

The demo is done, and I am victorious.

The important thing to realize now is that I have to get ttyO02 initialized from the code. Somehow. That's the only way I can stream to it, terminos to it, or whatever input/output method I pick. The nuances of moving the leg left or right or whatever are nothing compared to figuring out how to talk to it in the first place. And the answer to that seems to be dealing with pinmux.

IX Conclusions

Chloe, Part I is in its final stages of documentation, wrap up, presentation, and finalization for *Chloe, Part II*. It is difficult to draw any thesis-related conclusions at this point in time, as I've basically initialized the body on which the actual personality will build. However it's clear to see that I've already met a large number of learning outcomes.

Looking back at learning outcomes for the course:

1. Students will learn to use physical input devices such as switches, sensors, and cameras
    1. This is to be handled in *Part II*
2. Students will develop physical interactive media prototypes
    1. Done
3. Students will research independently the best software and hardware solutions for a personal project
    1. Done and done! I selected the board and robotics kit with input from numerous sources, I am working on picking a battery that suits my needs, I am familiar with various components stores all over my area, and I most certainly have gotten my own independently researched software solution up and running.
4. Students will develop a personal project using custom software and hardware
    1. Done and done; I am putting together a board and a kit that were not explicitly made for one another or shipped together, and I will late be adding additional sound capture devices. The software I'm programming to run on the board is entirely my own.
5. Students will product short video documentation of their projects for festivals and portfolio reels
    1. I have produced short videos documenting Chloe's first movements, as well as gathered together photographs of her early life pre-assembly. I will be able to use these to produce my end video documentation.
6. Students will write research reports on key artists and designers working in the interactive field
    1. Yet to be determined

Right now I'm dissatisfied that Chloe still cannot stand on her own, but I know I have the resources to get her upright prior to the start of *Part II*. Clearly I need to be able to accurately transmit output before I will be ready to receive input. I do not believe getting the library to work will be one of my larger challenges; because despite the difficulties of learning a new API, there are nowhere near as many unanswered ambiguities about software implementation in my mind as there are about hardware implementation.

I gauge that I am currently on track to complete *Chloe*'s project as described, and I look forward to seeing her stand.

X References

- Installing Ubuntu on the BeagleBone. (n.d.). Retrieved October 9, 2013, from http://zpriddy.com/installing-ubuntu-on-the-beaglebone/
- Arduino Aluminium Hexapod Spider Six 3DOF Legs Robot Frame Kit @GoodLuckBuy.com. (n.d.). Retrieved October 9, 2013, from http://www.goodluckbuy.com/arduino-aluminium-hexapod-spider-six-3dof-legs-robot-frame-kit.html
- BeagleBoard.org - community supported open hardware computers for making. (n.d.). Retrieved October 9, 2013, from http://www.beagleboard.org/
- BeagleBoard.org - faq. (n.d.). Retrieved October 9, 2013, from http://beagleboard.org/support/faq
- BeagleBoard.org - Getting Started. (n.d.). Retrieved October 9, 2013, from http://www.beagleboard.org/Getting%20Started
- beagleboard/linux · GitHub. (n.d.). Retrieved October 9, 2013, from https://github.com/beagleboard/linux
- BeagleBoardBeginners - eLinux.org. (n.d.). Retrieved October 9, 2013, from http://elinux.org/BeagleBoardBeginners
- BeagleBoardPinMux - eLinux.org. (n.d.-a). Retrieved October 9, 2013, from http://elinux.org/BeagleBoardPinMux#UART2
- BeagleBoardPinMux - eLinux.org. (n.d.-b). Retrieved October 9, 2013, from http://elinux.org/BeagleBoardPinMux
- BeagleBone - eLinux.org. (n.d.). Retrieved October 9, 2013, from http://elinux.org/BeagleBone
- Beaglebone + Ubuntu | MitchTech. (n.d.). Retrieved October 9, 2013, from http://mitchtech.net/beaglebone-ubuntu/
- Beaglebone and the 3.8 Kernel. (n.d.). Retrieved October 9, 2013, from

https://docs.google.com/document/d/17P54kZkZO_-JtTjrFuVz-Cp_RMMg7GB_8W9JK9sLKfA/pub

- BeagleBone Black – Ubuntu Precise 12.04 LTS, Ubuntu Raring 13.04, Debian Wheezy 7.0.0 Images for the BeagleBone Black – armhf.com. (n.d.). Retrieved October 9, 2013, from http://www.armhf.com/index.php/boards/beaglebone-black/
- BeagleBone Getting Started Guide. (n.d.). Retrieved October 9, 2013, from http://www.lvr.com/beaglebone.htm
- bradfa/beaglebone_pinmux_tables · GitHub. (n.d.). Retrieved October 9, 2013, from https://github.com/bradfa/beaglebone_pinmux_tables
- EBC Exercise 10 Flashing an LED - eLinux.org. (n.d.). Retrieved October 9, 2013, from http://elinux.org/EBC_Exercise_10_Flashing_an_LED
- embedded - Driving Beaglebone GPIO through /dev/mem - Stack Overflow. (n.d.). Retrieved October 9, 2013, from http://stackoverflow.com/questions/13124271/driving-beaglebone-gpio-through-dev-mem
- FAQs | Raspberry Pi. (n.d.). Retrieved October 9, 2013, from http://www.raspberrypi.org/faqs
- Fire Bird V Atmega2560 Hexapod Robotic Research Platform - Buy Fire Bird V Atmega2560 Hexapod Robotic Research Platform Product on Alibaba.com. (n.d.). Retrieved October 9, 2013, from http://www.alibaba.com/product-free/123806154/Fire_Bird_V_ATMEGA2560_Hexapod_Robotic.html
- gcc - Cross-Compiling for an embedded ARM-based Linux system - Stack Overflow. (n.d.). Retrieved October 9, 2013, from http://stackoverflow.com/questions/12512101/cross-compiling-for-an-embedded-arm-based-linux-system
- Getting started with Linaro: Software, Documents & Discussion. (n.d.). Retrieved October 9,

2013, from http://www.linaro.org/engineering/getting-started

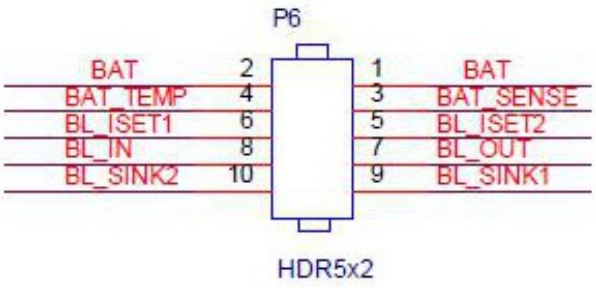
- Getting UART2 (/dev/ttyO1) Working on BeagleBone Black | Pignology News. (n.d.). Retrieved October 9, 2013, from http://blog.pignology.net/2013/05/getting-uart2-devttyo1-working-on.html
- How to copy files using SSH [Archive] - The macosxhints Forums. (n.d.). Retrieved October 9, 2013, from http://hintsforums.macworld.com/archive/index.php/t-29244.html
- How to make your first robot | Let's Make Robots! (n.d.). Retrieved October 9, 2013, from http://letsmakerobots.com/start
- HowToIdentifyADevice/Serial - Debian Wiki. (n.d.). Retrieved October 9, 2013, from https://wiki.debian.org/HowToIdentifyADevice/Serial
- Intrinsyc Software — Qualcomm DragonBoard™. (n.d.). Retrieved October 9, 2013, from http://shop.intrinsyc.com/collections/snapdragon-1
- kirillv/cpp-inverse-kinematics-library · GitHub. (n.d.). Retrieved October 9, 2013, from https://github.com/kirillv/cpp-inverse-kinematics-library
- LED Control Sample Code. (n.d.). Retrieved October 9, 2013, from http://www.lvr.com/code/led_control.c
- linux - Two-way C++ communication over serial connection - Stack Overflow. (n.d.). Retrieved October 9, 2013, from http://stackoverflow.com/questions/11677639/two-way-c-communication-over-serial-connection
- No such file or directory error. (n.d.). Retrieved October 9, 2013, from http://ubuntuforums.org/showthread.php?t=2031471
- OfficeofCTO/HardFloat/LinkerPathCallApr2012 - Linaro Wiki. (n.d.). Retrieved October 9, 2013, from https://wiki.linaro.org/OfficeofCTO/HardFloat/LinkerPathCallApr2012

- onboard LED control of beaglebone | LinkedIn. (n.d.). Retrieved October 9, 2013, from http://www.linkedin.com/groups/onboard-LED-control-beaglebone-1474607.S.119144212
- piranha32/IOoo · GitHub. (n.d.). Retrieved October 9, 2013, from https://github.com/piranha32/IOoo
- Platform | Pandaboard. (n.d.). Retrieved October 9, 2013, from http://pandaboard.org/content/platform
- Setting up Eclipse on the Beaglebone for C++ Development | derekmolloy.ie. (n.d.). Retrieved October 9, 2013, from http://derekmolloy.ie/beaglebone/setting-up-eclipse-on-the-beaglebone-for-c-development/
- SourceForge.net: Robotics Library. (n.d.). Retrieved October 9, 2013, from http://sourceforge.net/apps/mediawiki/roblib/index.php?title=Robotics_Library
- SpiderBot (Hexapod) by mind - Thingiverse. (n.d.). Retrieved October 9, 2013, from http://www.thingiverse.com/thing:1603
- T8 - Robugtix™. (n.d.). Retrieved October 9, 2013, from http://www.robugtix.com/t8/
- T8 robot tarantula gives everyone the willies. (n.d.). Retrieved October 9, 2013, from http://www.gizmag.com/robugtix-t8-robot-tarantula/28168/
- Ubuntu Manpage: ost::TTYStream - TTY streams are used to represent serial connections. (n.d.). Retrieved October 9, 2013, from http://manpages.ubuntu.com/manpages/jaunty/man3/ost_TTYStream.3.html
- Using Eclipse to Cross-Compile for BeagleBone Black - Michael Leonard. (n.d.). Retrieved October 9, 2013, from http://www.michaelhleonard.com/cross-compile-for-beaglebone-black/
- Using Eclipse to Cross-compile, Part 1: Install a Toolchain. (n.d.). Retrieved October 9, 2013, from http://www.lvr.com/eclipse1.htm

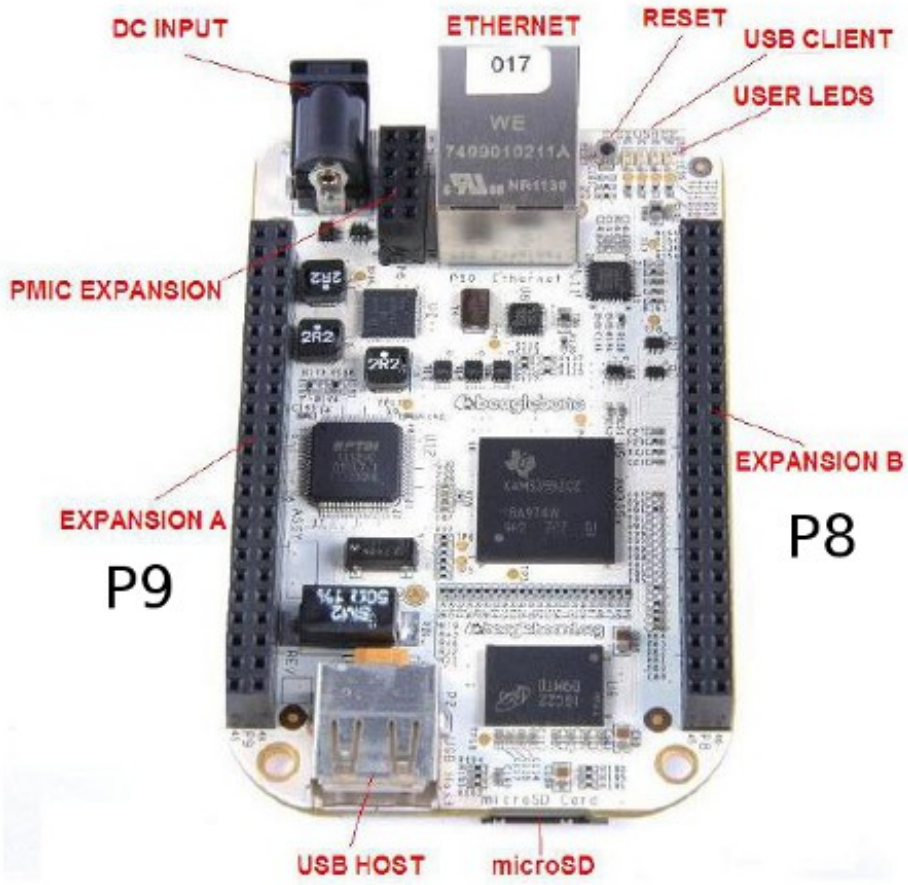
- Using the user leds - BeagleBone. (n.d.). Retrieved October 9, 2013, from http://beaglebone.cameon.net/home/using-the-user-leds

XI Appendix A: Schematics

1 BeagleBone



PMIC Expansion Header



Board Connector and Indicators

Expansion Header P9 Pinout

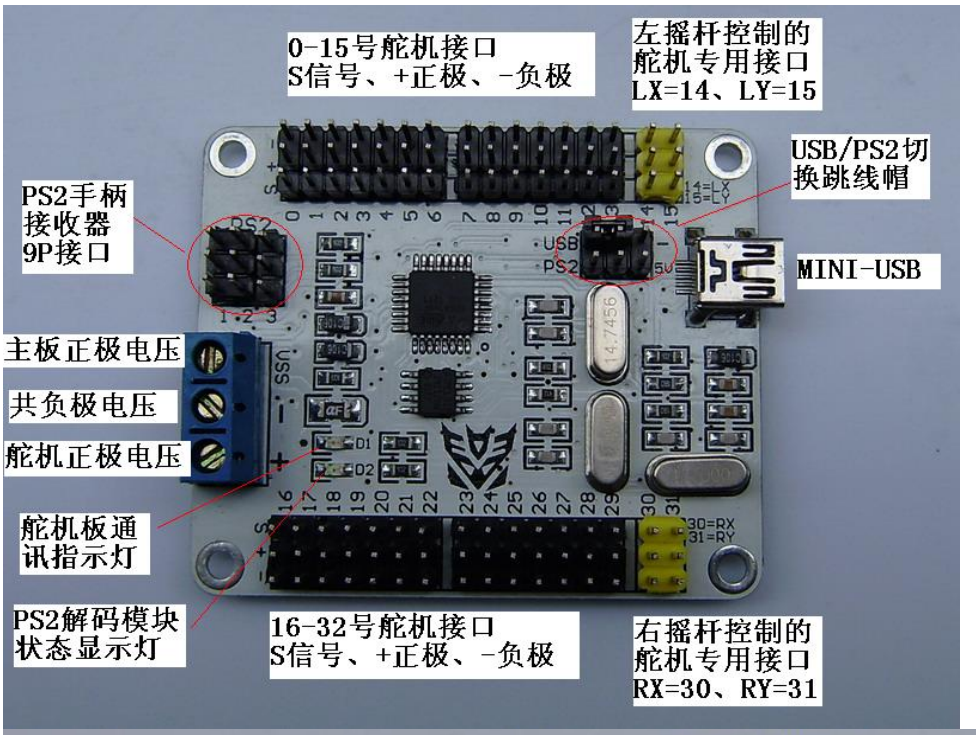
SIGNAL NAME	PIN	CONN		PIN	SIGNAL NAME
	GND	1	2	GND	
	VDD_3V3EXP	3	4	VDD_3V3EXP	
	VDD_5V	5	6	VDD_5V	
	SYS_5V	7	8	SYS_5V	
PWR_BUT*		9	10	A10	SYS_RESETn
UART4_RXD	T17	11	12	U18	GPIO1_28
UART4_TXD	U17	13	14	U14	EHRPWM1A
GPIO1_16	R13	15	16	T14	EHRPWM1B
I2C1_SCL	A16	17	18	B16	I2C1_SDA
I2C2_SCL	D17	19	20	D18	I2C2_SDA
UART2_TXD	B17	21	22	A17	UART2_RXD
GPIO1_17	V14	23	24	D15	UART1_TXD
GPIO3_21	A14	25	26	D16	UART1_RXD
GPIO3_19	C13	27	28	C12	SPI1_CS0
SPI1_D0	B13	29	30	D12	SPI1_D1
SPI1_SCLK	A13	31	32	VDD_ADC(1.8V)	
AIN4	C8	33	34	GNDA_ADC	
AIN6	A5	35	36	A5	AIN5
AIN2	B7	37	38	A7	AIN3
AIN0	B6	39	40	C7	AIN1
CLKOUT2	D14	41	42	C18	GPIO0_7
	GND	43	44	GND	
	GND	45	46	GND	

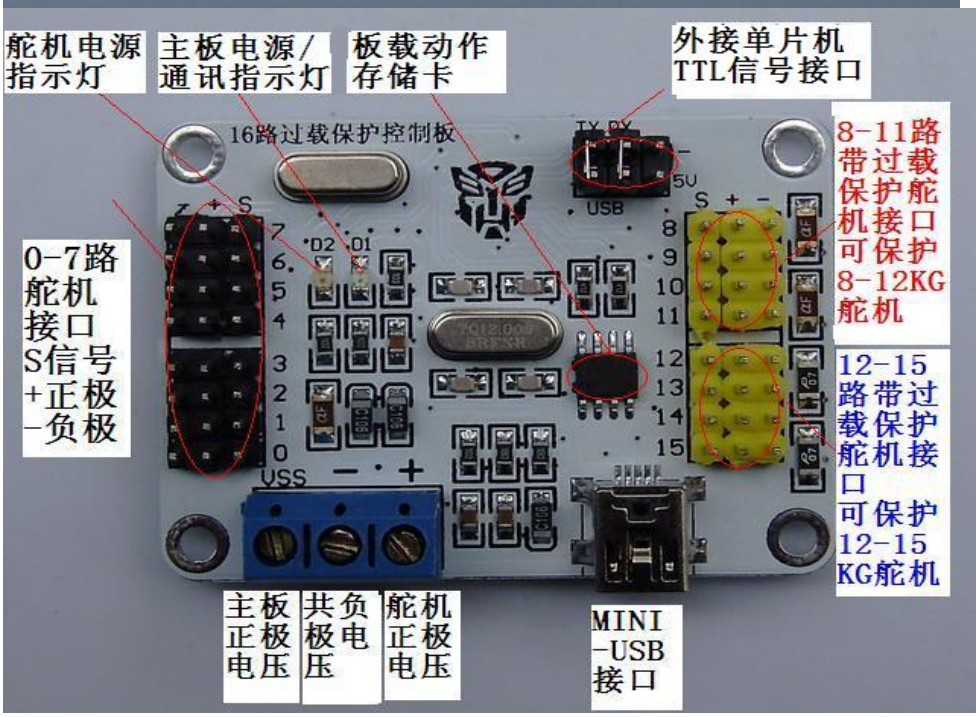
Expansion Header P8 Pinout

SIGNAL NAME	PROC	CONN		PROC	SIGNAL NAME
	GND	1	2	GND	
GPIO1_6	R9	3	4	T9	GPIO1_7
GPIO1_2	R8	5	6	T8	GPIO1_3
TIMER4	R7	7	8	T7	TIMER7
TIMER5	T6	9	10	U6	TIMER6
GPIO1_13	R12	11	12	T12	GPIO1_12
EHRPWM2B	T10	13	14	T11	GPIO0_26
GPIO1_15	U13	15	16	V13	GPIO1_14
GPIO0_27	U12	17	18	V12	GPIO2_1
EHRPWM2A	U10	19	20	V9	GPIO1_31
GPIO1_30	U9	21	22	V8	GPIO1_5
GPIO1_4	U8	23	24	V7	GPIO1_1
GPIO1_0	U7	25	26	V6	GPIO1_29
GPIO2_22	U5	27	28	V5	GPIO2_24
GPIO2_23	R5	29	30	R6	GPIO2_25
UART5_CTSN	V4	31	32	T5	UART5_RTSN
UART4_RTSN	V3	33	34	U4	UART3_RTSN
UART4_CTSN	V2	35	36	U3	UART3_CTSN
UART5_TXD	U1	37	38	U2	UART5_RXD
GPIO2_12	T3	39	40	T4	GPIO2_13
GPIO2_10	T1	41	42	T2	GPIO2_11
GPIO2_8	R3	43	44	R4	GPIO2_9
GPIO2_6	R1	45	46	R2	GPIO2_7

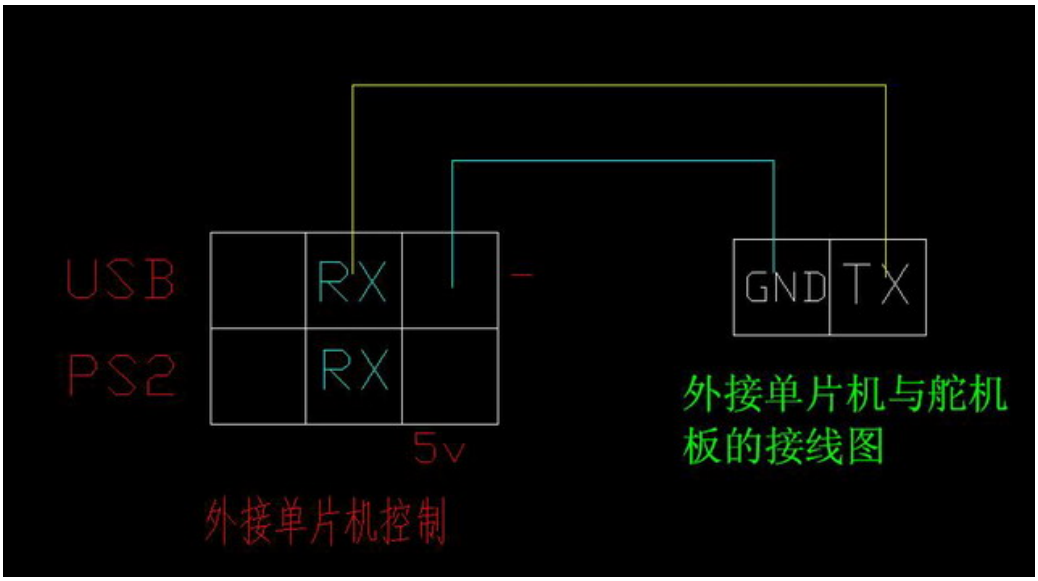
2 Control Board

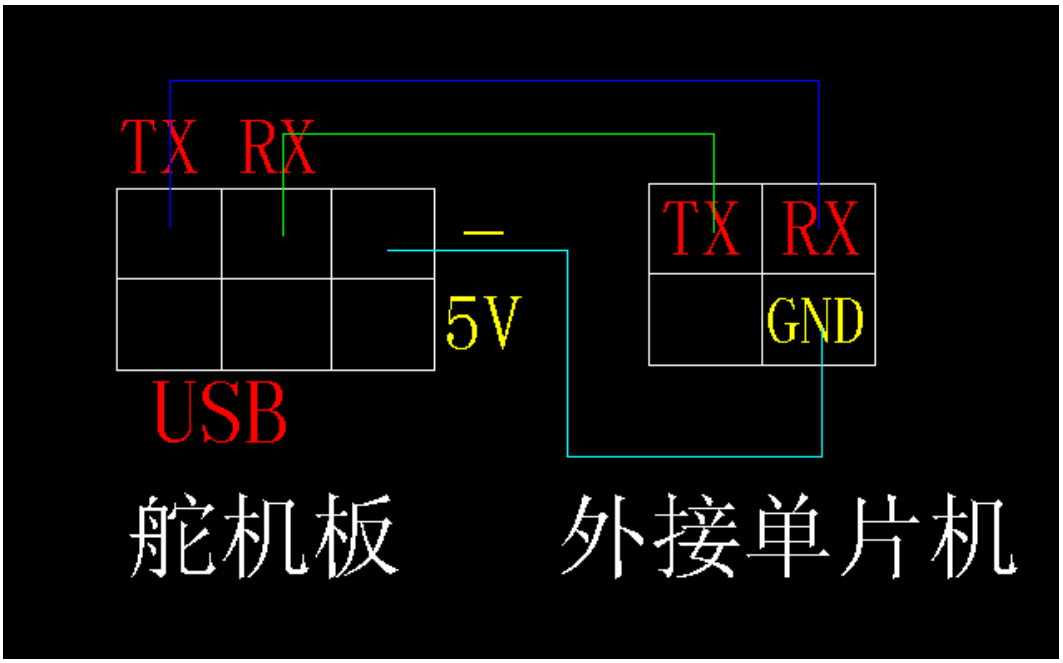
a Schematics





Connections





c Communication

<ch> P <pw> S <spd>... # <ch> P <pw> S <spd> T <time>
<ch> =舵机号, 0 - 31.
<pw> =脉冲宽度单位 微秒 , 范围500 - 2500.
<spd> =移动速率 us/s 每秒移动脉宽数针对一个舵机有效
<time> =移动到指定位置使用的毫秒数 (Optional)
例子 "#5 P1600 S750 "
移动舵机号5 到脉宽 1600us 速率为每秒移动脉宽750微秒
"#5 P1600 #10 P750 T2500 "
移动舵机号5 到脉宽 1600us 移动舵机号10 到脉宽 750us 使用时间为2500ms
注： T 可以对前面所有舵机有效除了有S的舵机号
#5 P1600 #10 P750 #12 P1700S500 T2500 5号和10舵机是使用2.5S完成移动12舵机看它以速率500us/s实际使用时间确定

注意外接单片机、串口助手，通讯代码需加回车 \r\n
如 舵机控制：#5 P1600 T500\r\n
动作组控制：PL0 SQ1 SM100\r\n

外接单片机或者ARDUINO时，运行动作组执行指令说明
运行动作组
PL <p> SQ <s> [SM <m>] [IX <i>] [ONCE]
PL 0 指定动作场景 必须指定
SQ <s> 指定动作组编号 s, 0 – 127 不指定为0
SM <m> 指定速度比m, –200- 200 不指定为100
IX <i> 指定启动动作组开始步编号i , 0 - 255。 不指定为0
ONCE 指定执行动作一次。 不指定为循环运行
范例说明
在动作场景中运行动作组5， 100%速度正向运行。
PL 0 SQ 5
改变动作场景中的速度，以50%速度反向运行。
PL 0 SM -50
暂停动作场景 (设置速度为0)
PL 0 SM 0
改变动作场景的速度为200%正向运行。
PL 0 SM 200
停止动作场景
PL 0
在动作场景中运行动作组15， 开始步编号为2,以70%的速度反向运行, 只运行一次
PL 0 SQ 15 IX 2 SM -70 ONCE
ARDUINO控制舵机板范例：

```
void setup()
{
```

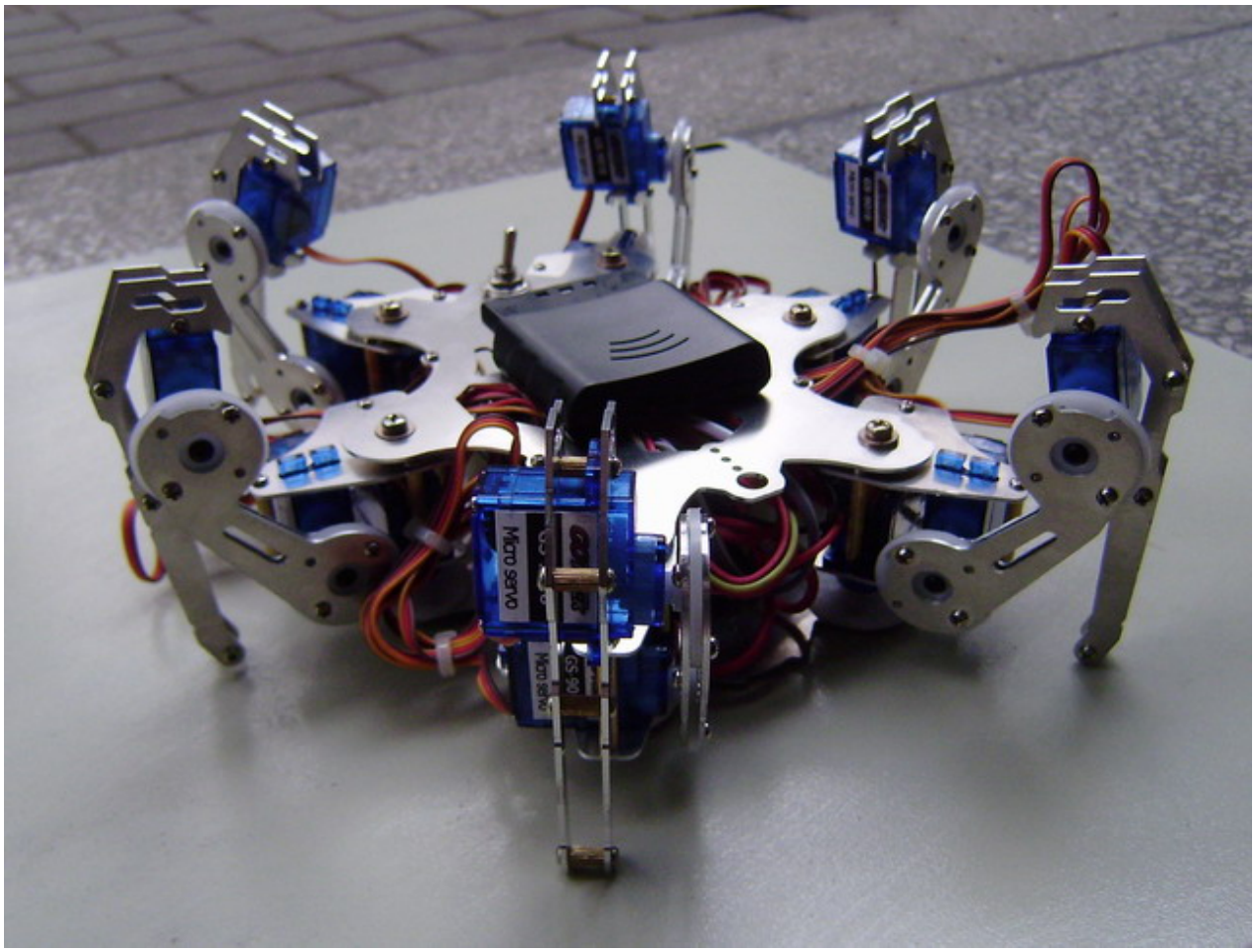
```
Serial.begin(115200);//波特率锁定在115200，不能修改
}
void loop()
 {
  Serial.println("PL0");//先停止以前的动作组
  delay(100);//延时
  Serial.println("PL0 SQ1 SM100 ");//以100%速度运行动作组1
  delay(500);//延时500MS，以保证该动作组运行完成
  Serial.println("#5 P1600 T500");//5号舵机用500MS的时间运行到P1600的位置
  delay(500);// 延时500MS，以保证该舵机运行到指定位置
}
```

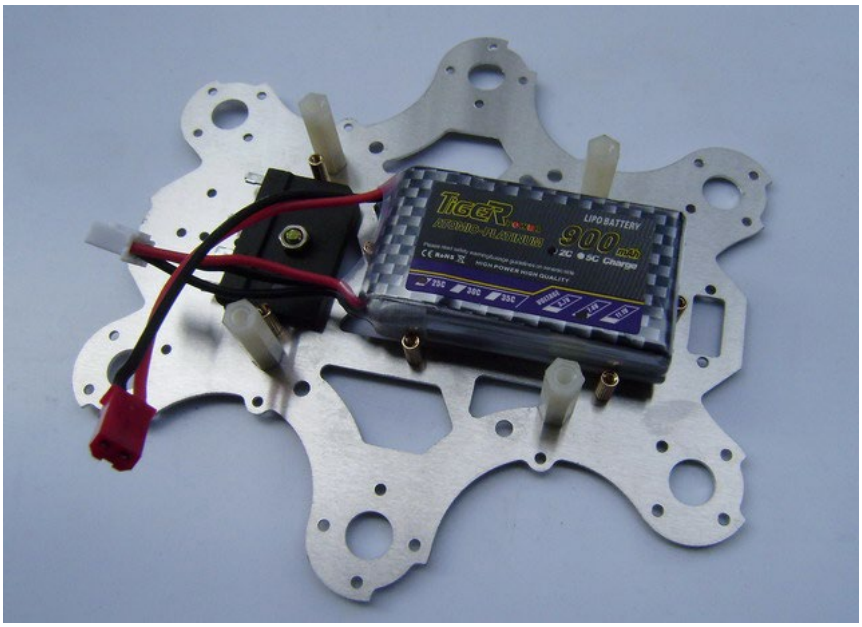
注意外接单片机、串口助手，通讯代码需加回车 \r\n

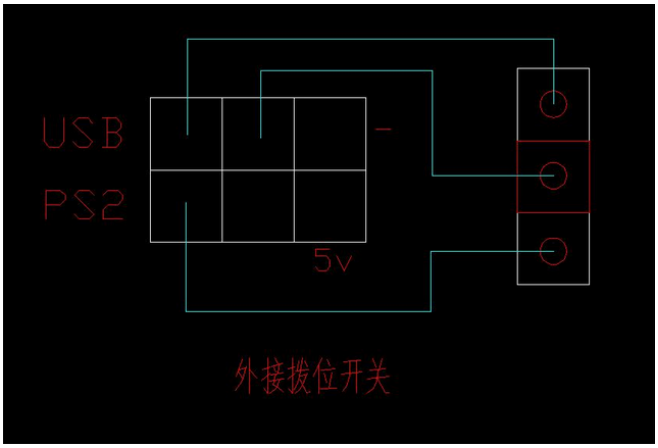
如 舵机控制：#5 P1600 T500\r\n

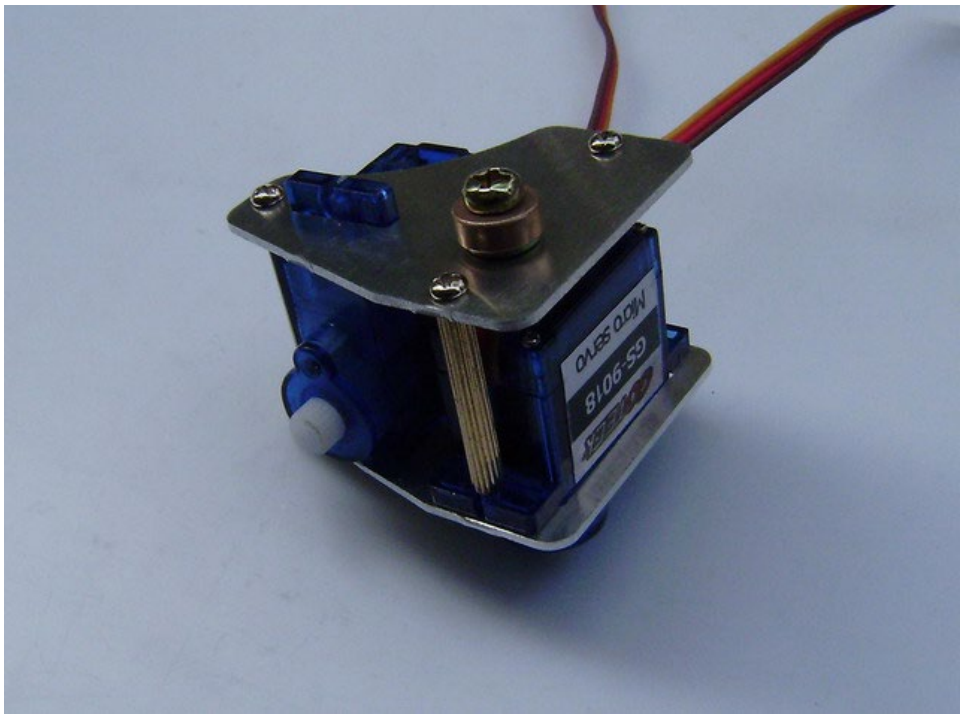
动作组控制：PL0 SQ1 SM100\r\n

3 Vital Robot Images






USB
PS2
-
5v
外接拨位开关


CAUSE CLOWN
BH06-SG