Individual Lab Report 1

Nathaniel Chapman

Team B: Space Jockey
with Brian Boyle, Ardya Dipta Nandaviri, Songjie Zhong
October 10th, 2013

1. Individual Progress

In the weeks since the Conceptual Design review, the bulk of my effort in this class has been directed towards getting Tasks 5 & 6 out the door. Due to my broad brush background. I have found that my primary role in the team has shifted to the effort of integrating and verifying the work of my teammates.

1.1 Task 5

For this task, I received the initial schematic and board designs from Dipta. I then verified and corrected some of the parts packages and libraries, calculated the amp loads and trace widths used for the board, and cleaned up and organized the board layout and schematics to be visually cleaner. Once done with that, I coordinated with Songjie to generate the CAD drawings of the board, helped Brian with the analysis, and then collected the information and turned it in.

1.2 Task 6

For task 6, we set out to incorporate support for a servo, stepper, and DC motor, as well as an infrared sensor, potentiometer, force sensor, and motor encoder. Our experimental layout can be seen in figure 1. I took the lead on integrating the Arduino code and developing the embedded software. To do this, I developed a hardware-agnostic driver interface and a serial packet format to allow for client side control of motors and for smooth client-host communications. I also developed the stepper motor driver, and provided the fancy laser-cut potentiometer lever seen in our testing setup (which was a leftover from a previous class). I also set up a shared Github repository for the project, and managed branching and merging of code.

To develop our system software, we each started by working individually to develop the basic functions necessary to drive our assigned sensors or actuators. It was my responsibility to then take each person's driver code, wrap it up in a nice container class, and attach it to our communications and controls protocols.

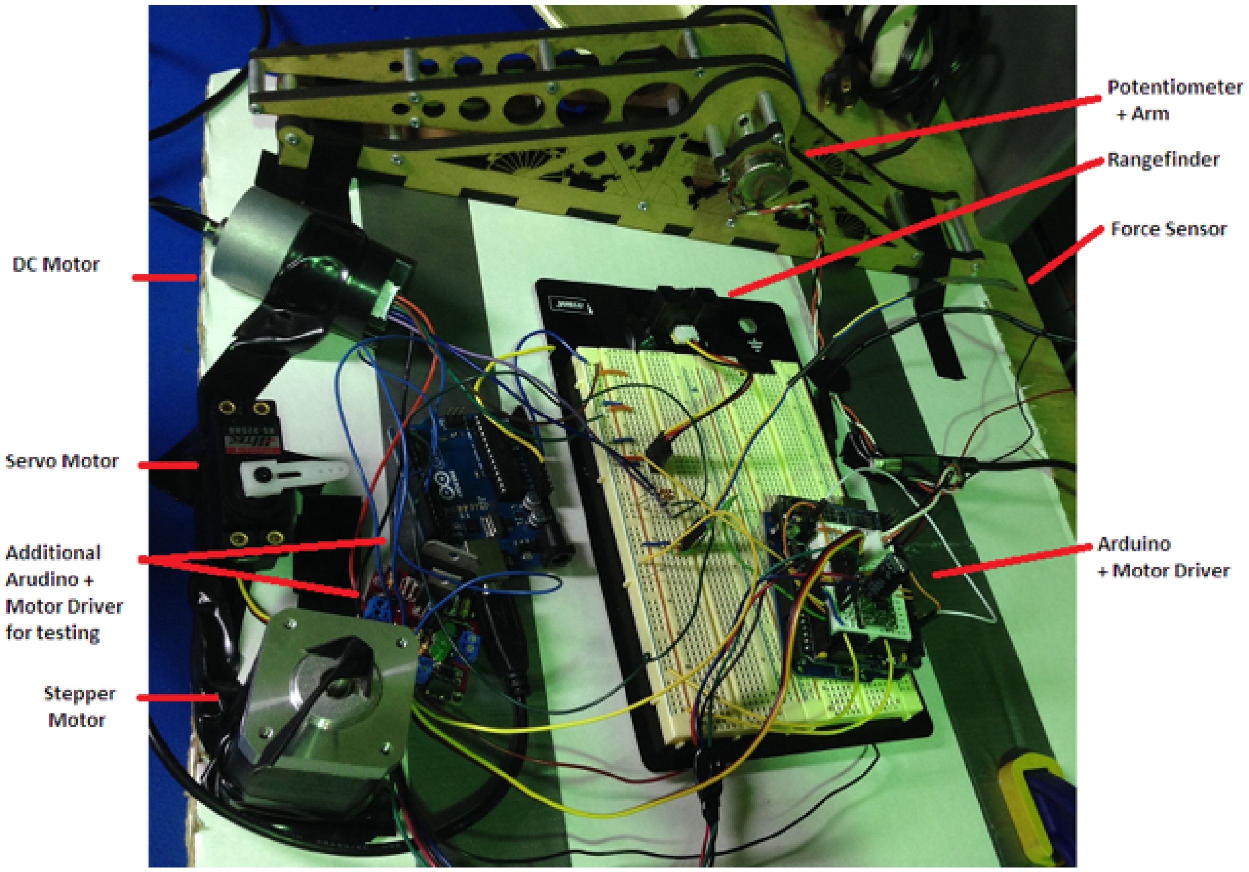

Figure 1: Task 6 Experimental Layout

1.3 Task 6 – Arduino Software Architecture

To provide for easy communication and control between the sensors, motors, and computer interface. I implemented two hardware abstractions: InputChannel and OutputChannel. An InputChannel simply stores a 16-bit value loaded from one of the sensors, or from the serial interface from the computer. An OutputChannel is attached to a motor controller, LED or other output device, and interprets that same scaled 16-bit value to implement control of its device. OutputChannels also provide an attachment method where they can automatically pull their control signal from any one of the supplied InputChannels. They also have their value updated in round-robin fashion every pass through the main loop of the Arduino program, though specific drivers could easily be set up to be updated on an interrupt-driven basis for timing-critical devices as well. In this manner, one InputChannel may drive arbitrarily many OutputChannels, and OutputChannels are continually updated even if the Arduino is disconnected from its host computer.

The overall system has 17 input channels, and 8 output channels, these numbers are completely arbitrary, and could be expanded to power larger systems if needed. The lowest input channel is always set to 0, providing a simple “off” state to paired devices (similar to /dev/null on Unix machines). The lower 8 input channels (1-8) are connected to live hardware inputs, as well as a few debugging signals that are being generated on the Arduino. The upper 8 input channels (9-16) are “virtual channels” whose values are set by the computer over the serial link. In this way, serial control and direct hardware control can both be handled by the motors without any additional driver support. A diagram of the basic Input/Output mapping mechanism is provided in figure 2 below.

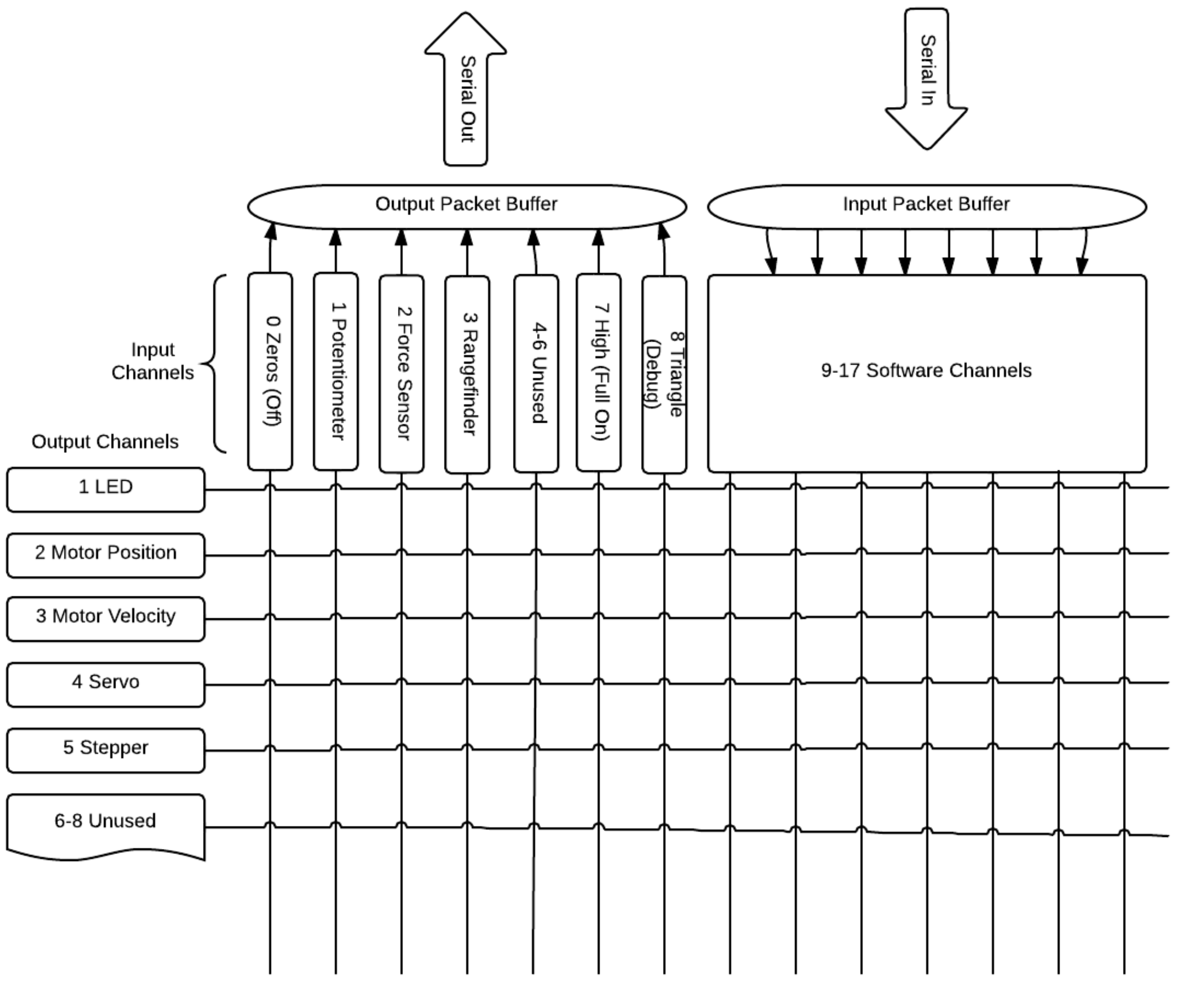

Figure 2: A functional diagram of the hardware abstraction scheme employed on the Arduino side of Task 6.

Serial communication is packet-based, using a 24 byte packet. The packet has 16 bit head and tail characters (0xDEAD and 0xBEEF, for easy debugging), and another 16 bit checksum as the last value in the packet body. The first two bytes of the packet body are 8-bit control signals. By default, they are set to (0, 0), which represents a no-op; but any pair (A, B) means pair InputChannel[A] to OutputChannel[B]. If they are set to (0xFF, 0xFF), this signifies a transmission error of the previous packet (bad checksum), and requests a re-transmit, although this feature is not fully implemented at the present time. The bulk of the packet body is a set of eight 16 bit channel values. Note that the format of both status (Arduino → PC) and control packets (PC → Arduino) are identical. Status packets transmit the live sensor (and debug) channel data, and Control packets set the values of the “virtual” InputChannels (channels 9-17). Thus, to directly control an actuator from the GUI, the computer sends a control packet tying that output channel to an arbitrary “virtual” input channel, and then sets the value of that channel in the same packet. A diagram of the packet structure can be seen in figure 3. For the current number of channels, every channel value is sent in every packet (status packets are sent every 50 ms, control packets are only sent when user input is received), however, if the system is scaled up greatly, it may be necessary to implement a different packet format and addressing scheme to deal with limited serial bandwidth.

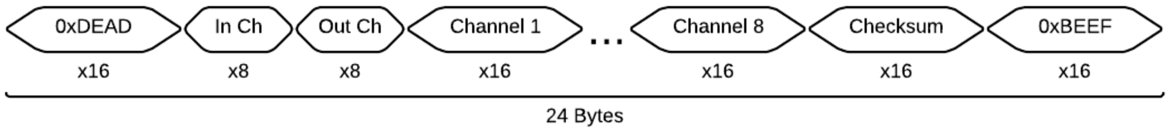

Figure 3: The packet structure employed for serial communications.

The way in which input channel values are interpreted is handled differently for each actuator driver. All channels are normalized to the range 0 → 0xFFFF, but the individual actuator drivers can scale this input as they need. This control scheme also proves to be fairly flexible for devices that require multiple control modes (e.g. position and velocity control modes for a DC motor), one driver may provide multiple OutputChannel objects that control the different modes of operation, and then handle priority or interleaving of control modes internally. For task 6, this was to be implemented on our DC motor controller, however, we ran into issues when integrating that particular driver (due to some C++ name space conflicts, and other bugs) and were unable to be completed before our demo. However, this technique could be used in the future to implement hybrid control of our robotic manipulators (by providing separate channels for torque and position along each axis) or other similar tasks.

The GUI for task six was developed in Python, due to it's convenient libraries, and easy support for c-like structs, which allowed us to use the same basic packet specification on both sides. The GUI was developed by Brian Boyle, and will be discussed in more detail in his ILR.

2 Challenges / Issues

2.1 General Challenges

As a group, one of our biggest challenges we've faced is communication. A few of our teammates are not native English speakers, and this means we have to take additional steps to ensure everyone is on the same page after each meeting, but we are slowly learning how to work around that.

I personally am also responsible for some of our more noteworthy communication failures. In Task 6, I did not do a good job of communicating the channel architecture up front, or explaining how to structure each individual's code with that in mind, and this really came back to bite us. It greatly added to the scale of my task in integrating each individual's code, and eventually led to the delays and setbacks that prevented us from being able to successfully incorporate and demonstrate DC motor control during the lab time.

2.2 Task 6

In task 6, we also faced several technical problems, our first stepper driver board was burnt out when we received it, leading to several hours of fruitless driver debugging. Also, our DC motor jammed during operation, and needed to be replaced. Also of particular note was one killer bug that arose during our final software integration: I introduced a null-pointer reference when writing my OutputChannel update code. On a PC, this would have led to a segmentation fault, but on an Arduino, it just results in very strange emergent bugs and mangled memory bits that appear to shift around the code with each compile (which they essentially do, as the compiler's memory map changes each time you comment/uncomment a section for debugging). In the end, after about 7 hours of frantic debugging till 5 in the morning, I was able to figure it out, and I now know to watch out for that, which is good to know!

3 Efforts By Team Members

3.1 Task 5

Dipta did the initial schematic design and board layout, I fixed the package definitions, wire traces, and cleaned up the visual design, Songjie did the CAD model of the final board, and Brian performed the analysis.

3.2 Task 6

Dipta wrote the DC Motor, Encoder, and Infrared rangefinder individual code, he also wrote the PID controller for Position and velocity control of the DC Motor, Songjie wrote the Force and Potentiometer sensor drivers, and Brian wrote the GUI and PC code. My contribution is discussed above.

3.3 General Project Effort

Songjie and Dipta have been working with Metin Sitti to begin investigating the viability of dry-adhesive feet for our project. Thus far, we have faced several scheduling difficulties in trying to meet with him, but we were finally able to meet with him today, and will be able to start manufacturing and testing dry adhesive components for our project soon, so that we can determine our foot actuator design. I have been taking the lead on managing and incorporating the various class assignments is developing the GUI prototype for our final system, which will help us further define our project use case, and anticipate difficulties with robot guidance and autonomy.

4 Future Plans

For the next several weeks, our focus as a team will be in implementing the bench squid for our project and developing a prototyping of of our robot's legs. My particular role in that will be to coordinate and research parts for the squid, and begin ordering components. This will require me to coordinate with Songjie to figure out torque specs for our servos, and to work with Dipta on the Power Distribution Board design, so that we can provide the right amounts of power for all devices (Servos, microcontroller, camera, and radio). First priority for me will be the final selection of a microcontroller platform and camera solution, alongside the tentative selection of battery type and voltage so we can plan the power board accordingly.

Appendix A – Source Code Listing

The source Code from Task 6 is submitted alongside this PDF. What follows is a brief description of each file.

Individual	This folder contains the individual drivers made by team mates.
ForceSensorDC	Songjie's prototype controlling the DC motor with the force sensor
MotorSpeedCtl	Dipta's PID DC motor velocity controller
MotorPositionCtl	Dipta's PID DC motor position controller
Sensor_motor	Dipta's prototype controlling the Servo with the IR rangefinder
Stepper_Driver	My initial prototype of the Pololu Stepper Motor driver
Task6	This folder contains the final integrated lab code,
GUI.py	Brian's python GUI implementation
SendPacket.py	A simple test script for communications testing
Task6.ino	The main Arduino file for our implementation
Pinout.h	A shared header declaring all of our device-wide pin uses
Comtypes.h*/.cpp*	This library contains the Packet and Channel classes, as well as some channel wrapper implementations around the Arduino Servo Library and basic LED functions
Motor.h/.cpp	The DC motor position and velocity control driver, incomplete
Stepper.h/.cpp	The stepper motor driver