CSCE 436/836: Embedded Systems
Lab 2: Basic Communication, Debugging, Thruster Control, and Sensor Reading

Instructor: Carrick Detweiler
carrick _at_ cse.unl.edu
University of Nebraska-Lincoln
Spring 2011

Started: Feb 8, 2011
Lab 2 Due: Feb 22, 2011

1 Overview

In this lab we will implement basic serial communication, thruster control, and analog sensor reading. We will also learn about techniques for debugging embedded systems. The serial communication will enable simple keyboard-based control of the hovercraft. We will control the thrusters by implementing PWM control of the power lines of the motors. The output of the thrusters will be controlled by sending serial port commands. Finally, we will read the analog sensors on the hoverboard. This includes the gyro on the 3.3V Atmel and the current and voltage feedback on the 5V Atmel.

Note that at the start of the lab we have only covered the serial communication topics in the lecture portion of class. Over the next two weeks we will cover most of the concepts needed to complete the rest of this lab. However, the lab instructions are fairly detailed, so you can work ahead of what we are covering in class or you can wait if you would prefer. Regardless, use the readings listed on the course website and the Atmel datasheet as a reference in lab.

Finally, as always, please read through the whole lab before starting. You may want to complete some sections of the lab before others. In particular, we will start using the 5V Atmel in this lab, but this is not described until towards the end of the lab.

2 Materials

In this lab you will need the following supplies in addition to those you already have:

- Five 2-56 by 3/4 inch socket head screws
- Five 2-56 plastic lock nuts
- Velcro

3 Safety

The safety instructions included here are a reminder from the previous labs. Please read through them again to remind yourself of safe operating practices when working with the Hoverboard and Hovercraft. Please let the instructor know if you have any questions or concerns.

In this lab we will start controlling the thrusters from the Hoverboard. Make sure that all fingers, hair, clothing, and other items stay well clear of the thrusters at all times. This is especially true when you first start working with them as they may turn on unexpectedly as you develop your control code. Also remember that you should turn off the thrusters when they are not in use as they may be damaged if operated for an extended period of time.

Please be extremely careful with the hoverboard. Make sure that you keep any metallic objects far away from the hoverboard. This includes rings, necklaces, coins, pens, etc. Also, never place the hoverboard on anything conductive as there are exposed elements on the back of the hoverboard. You should also never

have liquids near the hoverboard. Be careful when connecting cables and wires to the hoverboard to make sure that you connect it to the right port and in the right orientation.

We will be using high-power, two cell Lithium-Polymer (LiPo) batteries to power the hoverboard. Please be extremely careful with these batteries and recall the safety information we discussed in class and lab. In particular, only use the designated chargers in the lab to charge the batteries. The hoverboards will not turn on if the voltage on the battery is too low, however, you should also be aware of the use of your battery and never leave it connected to the hoverboard when not in use. A fully charged battery will have a voltage of 8.4V. A battery that is about half charged will have a voltage of approximately 7.4V, and a nearly discharged battery will have a voltage of around 7.0V. A voltage below 6.0V can be dangerous, especially if you try to recharge it. If this happens, please let the instructor know immediately. With care, it is possible to revive an over discharged battery if it is done quickly (but putting it on the charger is not the way to do it and it is dangerous).

Unlike programming a normal computer, it is possible that a bug in your code could physically damage the hoverboard or other devices. Take care, especially when programming I/O pins as setting the wrong state on the wrong pin could damage the processor or peripherals.

4 Getting Started [5 pts.]

To start this lab, we will mount the hoverboard on the hovercraft. Also, you will need to download the code templates from the course website.

4.1 Mounting Hoverboard on Hovercraft

You have been provided with 3/4 inch 2-56 socket-cap screws and lock nuts. Screw these into the mounting holes, you may need to move, cut, or adjust the electrical tape you used to cover the base of the hoverboard (make sure you it is still covered). The screws will stick out significantly. Then, choose a location for your hoverboard. It is possible to adjust the position later, but you should try to minimize the number of times you move it as it will result in additional holes in your hovercraft base.

When choosing a location to mount your hoverboard, consider the location of the battery, how thruster power wires will be routed to connect to the hoverboard, and the weight distribution needed to balance the hoverboard (with the battery weighing significantly more than the hover board). Once you have decided on a position for the hoverboard, simply press the screws down into the foam base. You can remove the hoverboard at any time by just pulling it up. You should be careful when removing and inserting the hoverboard so that you don't enlarge the holes too much. While the fit may be loose, the length of the screws should be sufficient to prevent unwanted motion of the board.

You have also been giving some Velcro. You can use this to affix the battery to the board. If your battery does not have Velcro on it already, please make sure to attach the fuzzy side of the Velcro to the battery.
Question: Where did you mount your hoverboard and battery? Why?

4.2 Base Code

Download the code templates from the course website. As always, you should look over all of the code so that you have a sense of what needs to be done. Look for `STUDENT CODE` comments, which indicate where you need to fill in code.

I recommend that you integrate some of your code from lab1 in with the code from lab2. In particular, it will be useful to be able to use the LEDs and button, so copy that code to your lab2 directory and modify the corresponding SConstruct file to include any additional source files you have added. With this base configuration, you should make sure that you can compile your code without errors or warnings. You may also find the Morse code implementation useful, although we will quickly move to using serial communication.

The code provided for the lab is for the 3.3V processor, which we will start off programming. We will also program the 5V processor in this lab. While the code will be very similar for the 5V Atmel, it will

be slightly different. The programming and build scripts, however, are the same, so you can directly copy these for the 5V processor. For this lab, you are only being provided with a very small amount of code. The design of the majority of the code will be left up to you for most of the lab.

5 Serial Communication [15 pts.]

In this part of the lab we will setup the two UART ports on the 3.3V Atmel processors. We will communicate over the serial port by using the USB to serial converter that you use to program the boards. This USB adapter allows communication with a 3.3V UART. Later in this section, you will learn how to communicate over the serial port from the computer. You will want to read this section before you start actually programming too much as you will need to connect from the computer to the hoverboard to see if your UART code is working.

Look at the hoverboard datasheet and identify the pins that are used for the serial ports. Make a note of this and also identify the two serial port connectors on the actual hoverboard.

In the sample code there are a couple of functions that you need to complete in `uart.c`. We will discuss each of these functions briefly, look at the code template comments for additional details. Section 5.2 contains activities to do once the serial port is working, but it also contains some techniques that you can use to debug your system if you are having problems.

`void uartInit(uint8_t port, uint32_t baud)`

This is the main initialization function. Depending on the value of `port`, it will configure either UART0 or UART1. The USB to serial converter we are using typically operates with 8-bits, no-parity, and 1 stop bit. This function should setup these parameters and enable the serial port operation of the ports. Carefully read the section on the UARTs in the Atmel datasheet to help set the correct parameters.

This function also sets the baud rate of the port. This should probably just be a call to `uartSetBaud(...)`, it is included as a parameter here so that the default baud rate for the UART is clear upon initialization.

`void uartSetBaud(uint8_t port, uint32_t baud)`

This function sets the baud rate of the specified `port`. Read the datasheet to determine how to set the passed baud rate. Recall that there is a normal and double speed mode. Depending on the desired baud rate, one of these may have a lower error rate. It is not required, but optionally, this function can choose which mode to use. If you do not do this, you should set it such that a baud rate of 19200 has a low error as this is the baud rate we will typically use. Also remember that the 3.3V processor operates at 8MHz.

`void uartSendByte(uint8_t port, char c)`

Implement this function that sends the specified byte, `c`, over the specified UART `port`.

`uint8_t uartReceiveByte(uint8_t port, char *c)`

Implement this function that checks to see if there is a byte to receive over the specified `port`. If a byte has been received, it should be stored in the location pointed to by `c` and return 1 to indicate a byte has been received. If there is no available byte, it should return immediately and return 0.

This function is said to be non-blocking as it does not wait if there is no data available. This is typically better as it allows other code to execute if there is nothing available. Blocking code can introduce bugs into embedded systems as it will get stuck in an infinite loop if a hardware or other error prevents communication. You should never use blocking code, unless you have a specified timeout that will escape from the blocked code once a certain amount of time has passed.

5.1 Communicating from the Netbooks

You can communicate with the Hoverboard using any serial terminal program such as minicom, seyon, or something else (even hyperterminal in windows). Here I will tell you how to use `GNU screen`. This is one of my favorite programs. It allows you to have multiple terminal windows open and is also able to communicate over serial ports. First make sure you have it:

```
sudo apt-get install screen
```

Next, you can connect to a serial port by giving the command:

```
screen /dev/ttyUSB0 19200
```

which will connect to serial port `/dev/ttyUSB0` at a baud rate of 19200. Adjust the baud rate if you set your Hoverboard to communicate at a different speed.

To exit `screen` you need to send the command `ctrl-a k` (that is hold down the control key and press `a`, let go of the control key and press `k`). If you forget to do this (and instead "x" or kill the window), `screen` will remain running and attached to the serial port in the background. You can reconnect to it by typing `screen -r`.

Note that in order to program the board, you will have to `ctrl-a k` the `screen` so that the `program` script can use the serial port.

5.2 UART Communication

The first thing to do once you have written the UART functions is to verify that it works. You can test the sending and receiving from the serial ports separately. Modify the main loop code to start out by initializing both serial ports to a baud rate of 19200. Next, test the sending on both serial ports by sending a short (and different) string (multiple characters) over both serial ports that repeats continuously. You can send a string over UART0, for instance, by executing the code:

```
uartSendByte(0,'t');
uartSendByte(0,'e');
uartSendByte(0,'s');
uartSendByte(0,'t');
```

You should make sure to add a pause after sending each string. This is important because if you send continuously over a serial port it is possible (especially if there is some error in the clk speed) that the receiver will have trouble synchronizing with the start and stop bits.

Question: Did your serial port sending code work? If not, what problems did you have?

If you receive garbage over the serial port and not the string you expected, you probably have a problem somewhere setting the baud rate. If you don't receive anything at all, try testing the receiver first (below), which will tell you if you at least have the baud rate correct. If the receiver works, it is likely you forgot to set the TX pin to output.

To test the receiver, write some code that lets you turn on and off the LEDs depending on which character you send. Define different keys that turn on or off each of the LEDs. Try sending these characters and see if it works.

Question: What keys did you use to turn on and off the LEDs? Did this work the first time, if not what was the problem?

Now combine the sender and receiver. Keep the LED toggling code, but add to it so that it echos back the same character that you sent if it is one of the LED toggle commands and sends back a '?' if that key does not correspond to a known keyboard command.

Question: Experiment with standard baud rates (those shown in the table at the end of the UART section in the Atmel datasheet). Do all of them work for communication with the computer? If some do not work, why is that?

It is useful to be able to send strings without having to type each individual character. Add a function that lets you send a string over the UARTs. Refer to the Morse Code file from Lab1 if you are unsure how this should be implemented.

Printing predefined strings is useful, but it is much better if you can print formatted strings in printf-style. It is possible to configure the system such that `printf` will work to print over the serial port. However, we will start by learning how to print formatted strings. It is relatively easy if we use the function `snprintf(...)`. We are using `snprintf` instead of `sprintf`. This takes the additional argument "n", which indicates the size

of the buffer being written into. Never ever, **ever** use `sprintf` as this one very common method of ensuring that your code will have buffer overflow bugs, as `sprintf` assumes that the buffer you pass it is of infinite size.

If you are unfamiliar with `snprintf` you can google it or look at the man page (from the terminal, type `man snprintf`). Using this function requires including `stdio.h` in the .c file. The idea is that you can create a temporary buffer to write the string into and then print it using the string sending code you just wrote (perhaps using `strlen(...)` to find the length of the string). Try printing out over the serial port some of the variables that are changing in your code. You may want to create a convenience function that enable easy printing of integers as decimal number or hex numbers.

Question: To go to a new line in the output, you can send a `\r\n`. `\r` is a carriage return and `\n` is a newline. Describe what happens if you only send one or the other.

If you look at the `program` script that we use, you will see that at the start the first thing it does is send the string "reboot" over the serial port. Add a basic parser to your code that will reboot the board if it receives that specific string. You will notice in there is a `reboot()` command in `main.c` that you can use to reboot the board. This means that you won't have to press the reboot button, you can just start the program script (unless your code is stuck somewhere and doesn't parse the reboot command).

Question: There are a number of ways you can implement parsing the serial input and looking for the reboot command. Describe how you implemented the parser for the reboot command and discuss why you choose to do it this way. In particular, address how you can also receive and process other serial commands while looking for the reboot string. Does your implementation allow you to use the characters in the string reboot for other commands?

6 Debugging [15 pts.]

Debugging embedded systems is much more challenging than debugging programs written on a PC. In this section we will learn some techniques for debugging. We will apply these techniques to figure out how some code that someone else wrote works, even though we do not have the source code.

6.1 Outputting Information

The most basic method for debugging is to use an LED to show where the program is in its execution. Simply turning on a green LED when the code is executing is a helpful tool. LEDs can also be used to obtain relatively precise information on the timing of events. By turning on an LED only when the function you are interested in is executing, you can see how long that function takes and obtain some information on when it is executing. If it is a very short event you may need to use an oscilloscope to see when it is actually occurring. Alternatively, you can only toggle the LED on every Nth execution of the function you are interested in by keeping track of a global counter. One thing to keep in mind, however, is that turning on and off LEDs does incur a slight overhead as it takes some clock cycles to turn them on and off.

LEDs are useful, but we only have two on each processor so it can take a long time to debug complex sections of code. We can also use the serial port to aid in debugging by printing out information. However, printing strings over the serial port can take a significant amount of time. So if timing is important, printing debug information over the serial port can change the program timing.

One useful technique is to output the values of configuration registers over the serial port. This allows you to verify that they are set as you expected (and that some other part of the code did not overwrite registers you thought you set properly). Try printing out the values of the DDR registers and verify that they are set to the expected values.

Question: How long does it take to send the string `Hello, World!\r\n` over the serial port if the baud rate is 9600? What about 19200 or 115200?

6.2 Help from the Compiler

The compiler gives us a number of sources of information that can help debug programs. First and foremost are the warnings and errors the compiler produces. You have to fix errors in order to compile the code. However, you also should pay attention to any compiler warnings (the `SConstruct` file enables warnings from `gcc` by passing the `-Wall` flag). If there are warnings, you should fix them, even if your code works as you expect. Often the warnings will indicate some potentially undefined behavior that could cause problems in the future. Also, if you always ensure that there are no compiler warnings, if you see one, you will know there might be a problem.

`scons` will only recompile files that have changed since the last compile. Sometimes it is useful to recompile everything to make sure there are no outstanding warnings or other problems. You can tell `scons` to delete any compiled files by issuing the command `scons -c` (this is like doing a `make clean`). You can then recompile everything and look for warnings.

The `SConstruct` file is setup so that when you compile, the file `main.lss` is also generated. This file contains the assembly code that is generated by compiling the code. You can look at this file to see the assembly that is generated from the `C` code you write. At the top of this file is also the memory map. There are a number of entries in this memory map. They are:

- `.text` goes into the processor's flash. This is the actual code that is executed.
- `.bss` goes into the processor's RAM. This is for any globally defined, but not initialized variables.
- `.data` goes into the processor's RAM. This is for any globally defined variables that are initialized.
- `.stab` and `.stabstr` which are just used for debugging information, they are not loaded into the processor when you program it

Note that some of these sections may not be present if they are not used at all (in particular the `.bss` and `.data` sections). By summing `.bss` and `.data` you can determine how much of the processor's RAM has been used, excluding memory used by the stack or heap (which you should not be using).

Question: In Section 5.2 you included `stdio.h` in order to use `snprintf`. This triggered the compiler to include some code from the stdio library. How much flash and memory does including this require?

When you look at `main.lss` you can see that shows the `C` code you wrote with the assembly. Sometimes the `C` code and assembly do not quite line up (or the assembly does not make sense when compared to the `C` code). This is because compiler optimization has been turned on. This means the compilers will rearrange and modify code in order to speed operation. If you look at the `SConstruct` file you will see that `-Os` flag is passed to the compiler. This enables size optimization. What this means is that the compiler will optimize the code to try and make it smaller, while also trying to make it execute faster. Making it smaller is useful to ensure that we have sufficient space in the flash for the code. Alternatively you can disable optimization by passing `-O0` to the compiler (that is a dash, capital "oh" and a zero). There are also levels 1, 2, and 3 for optimization. Sometimes it is useful to disable optimization by using `-O0`. This will make the assembly code correspond much more directly to the `C` code. This may allow you to spot bugs in the assembly that would be difficult to find with optimization enabled[1]. Be aware that changing the optimization level can drastically change the speed of program execution and the program size. In general, it is good to keep `-Os` enabled.

Question: For the different optimization levels, what is the resultant code size in flash for your code base?

6.3 Applying Debugging Techniques

In the 3.3V Atmel code templates there is a file `keycode.h` and `keycode.o`. The file `keycode.o` is the precompiled object file, you do not have access to the original `keycode.c` file that generated the object file. The author of the original code was testing some new code he wrote for the embedded system that controlled his car's steering. Unfortunately, there was a bug in the code so he drove off a cliff. The keycode file was used by an old project (an encryption keypad) to generate some keys to encrypt a client's data. The keypad

[1] In addition, sometimes optimizations can cause unexpected behaviors. The most common is that the compiler will remove code that it does not think or needed. For instance, loops that do not seem to do anything. This can cause a problem if you wanted to use that loop for a delay.

broke and now we need to make a new one to allow them to recover their keys. The .h and .o files were recovered for the project, but not the .c files and unfortunately the author did not comment the code well (at all) and functions are not named well.

Start by determining the amount of code and memory that the `keycode` code uses. You can do this by modifying the `SConstruct` file to include the object file or not and then examine `main.lss`. Note that when you look at `keycode` code in `main.lss` there are not any comments to help determine what is going on. This is because the author either did not comment the code at all or did not include debug information when compiling (hopefully it is the later, but judging by the .h file it might not be).

Question: How much code space and memory does `keycode` use?

Now look at `keycode.h` where the functions are defined. In `keycode.h` you will notices that there are 4 functions `startOne()`, `startTwo()`, `startThree()`, and `startFour()`. These are presumably functions that need to be run at the start to initialize the system. Unfortunately, it turns out that starting them in this sequence does not work.

Some initial debugging has found that if you call one of these functions out of order, the function will not return and the board will reboot. If a function is called in the correct order, it will return, but the board will still reboot shortly after. We believe that the full start sequence must be called in order to prevent rebooting.

Question: Determine the correct sequence to call these functions. How did you go about doing this? Hint: using LEDs or the serial port may be useful.

There are two additional functions `delaySmall()` and `delayBig()`. We think that these may be used by some of the other functions to perform delays.

Question: Approximately how many milliseconds do `delaySmall()` and `delayBig()` delay for? How did you go about determining this? You should only need your board and a stop watch to figure this out.

Finally, there are two functions `encodeChar(char c)` and `encodeInt32(uint32_t i)`. Once the startup sequences have been called these functions will operate. For the client we need to recover the encoding of the character 'c' and 'd' and the 32 bit integer 369288000. For some reason, these functions do not return values, instead, it looks like the `encodeChar(...)` function stores its result in the memory location `KEYCODE_CHAR_MEMORY_LOCATION` and `encodeInt32(...)` stores its result in `KEYCODE_INT32_MEMORY_LOCATION`. You can read the memory location by casting the integer memory location into a memory pointer of the correct size and then dereferencing that pointer.

Question: What are the encoded values for the characters 'c' and 'd' and the 32 bit integer 369288000? Explain how you determined this.

Question: How long does do the two encode functions delay for? Answer this in terms of milliseconds and in terms of the delay functions.

6.4 Additional Methods

There are numerous other methods to debug embedded systems. You will discover many yourself as you program microcontrollers more. You should also be aware of JTAG interfaces that are external devices that connect to a microcontroller and allow line debugging, insertion of break points, and the reading of registers. Not all processors support JTAGs and they are often processor-specific and can be expensive. They are, however, very useful for debugging systems. It is difficult, however, to use them to debug real-time events.

Another technique is to use periodic interrupts (we will learn about these later) to examine the state of different variables or registers (you can even use it to sample the current size of the stack). This can be useful, but again can interfere with real-time events.

7 PWM Motor Control [15 pts.]

In this section we will configure the pulse width modulator (PWM) pins on the 3.3V Atmel to control and vary the power to the thrusters. Start by looking at the hoverboard schematic. Notice that the PWM pins are connected to a number of N-channel mosfets. These are like switches connected to the ground lines of

the motors. If we turn them on, the motors will turn on at full speed (do not do this). We cannot vary the voltage or current that goes to the motors (that would be a complicated circuit) as we did with the power supply in lab to control the motor speed. But we can use PWM to switch the mosfets fast. Because motors act as inductors, if we switch fast enough, the voltage the motor sees will be proportional to the duty cycle of the switching. This means we can vary the duty cycle of the switching to change the speed of the motor.
Question: Notice the resistors connected to the mosfets. What do you think they do? Is the mosfet on when the control pin is high or low?

Create functions (perhaps in a new .c file) to initialize and control the PWM pins. Note, you should limit the maximum speed the thrusters can operate at to about 75% of maximum. Impose this limit in the software you write. Higher power operation may reduce the lifetime of the motors. Also remember that you should not leave the motors on longer than you need to (and at most around 10 minutes).

You should configure the PWM duty cycle to be around 500Hz, remember that the processor operates at 8MHz. This duty cycle is fast enough to provide smooth operation for the motors. It is also slow enough so that the mosfets are not switching too fast. Mosfets consume more energy (heat up) the faster they switch, so it is better to drive them at a lower switching frequency. If you set the frequency too high, you may burn the mosfets. You will need to read the datasheet carefully to figure out how to control each of the PWM pins (note that each of the timers is slightly different).
Question: Describe some of the differences between the different timer/PWM ports on the Atmel.

Now add to your code so that you can control each of the thrusters by sending different key strokes. You may want to have a "faster" and "slower" key for each thruster that increments/decrements the thruster speed by 5 or 10% with each key press. Alternatively, you could make it such that it only thrusts when a key is held down. This will give you primitive control over the hovercraft.
Question: What PWM percentage do you need to use to have lift with the lift thruster?
Question: Describe the key mapping you used and your experience in operating the hovercraft with these key mappings.

8 Analog to Digital Converter [15 pts.]

We will now implement code to read analog signals on the analog to digital (A2D or ADC) pins on the 3.3V Atmel processor. As always, you should start by reading the section on the ADC in the Atmel datasheet and reviewing the hoverboard schematic. Most of the ADC pins on the 3.3V Atmel are free pins that can be used to connect to external analog sensors. The last two pins, however, are connected to the analog outputs of the gyroscope. We will read these in this section.

The ADC has 10 bit resolution. Configure the ADC to use AVCC as the voltage reference, with single-ended voltage input (as opposed to using it in differential input mode), and in single conversion mode. You will also need to make sure that the ADC clock rate is correct. Do not worry about using the ADC noise canceler at this point. Note that since the ADC gives 10 bit resolution and the registers are only 8 bits, you have to read two different registers (in a particular order) and then shift the result to fit into a 16 bit integer.
Question: What is the actual ADC clock rate that you used?
Question: Given that the ADC has 10 bit resolution, what is the resolution of the ADC in mV?

8.1 Reading the Gyroscope

The gyro gives a relatively stable mid-voltage reading when no rotations are occurring. When you rotate in one direction the voltage drops and it increases if you rotate in the other direction. In addition to the ZOUT-1X output, there is also a 4X output. This is the same signal as the 1X output, except the signal change due to rotation is amplified 4 times. Try reading both of these channels (and printing the value over the serial port) while keeping the board still and also while rotating each direction.
Question: Find the datasheet for the gyro. What is maximum rotational rate that can be measured? What is the mV per degree per second for the two outputs? What about converted into bits per degree per second as read by the ADC?

Note that you may need to limit the rotational thrust on your hovercraft if the rotation rate exceeds that which can be measured by the gyro.

When the gyro is not rotating, it produces a stable voltage that is nominally around 1.23V (as per the datasheet). This number will vary slightly for each gyro. Determining this number accurately is critical because we are interested in integrating the output from the gyro (so that we get an absolute angle). If there is a small error in the initial offset, the integrated error will grow quickly over time. It is impossible to remove all offset error, but we will try to remove most of it. Create a function that averages a number of readings from the gyro channels to find the offset [2].

Question: In addition to reporting the offset determined by your averaging function, plot the readings of the gyro channels over time when the gyro is not moving. You can record data from your serial port to a file by doing `cat /dev/ttyUSB0 > datafile.txt`. Is there noise in the signal? Is it different on the two channels? Note, you can also increase the serial port baud rate to increase the number of samples you can capture.

Question: Now record a similar sequence of data with one of the thrusters active (but not rotating the board). Is there more noise?

Question: Now record a sequence of data when you are using the thrusters to rotate your hovercraft in both directions. You can script a thrust sequence (starting, stopping, changing direction, stopping, etc) that will start with the press of the button to make collecting data easier. Analyze and describe this data.

Question: Create a function that averages a few successive readings (enough to remove most of the noise you notices in the signal when not moving). Print this on the serial port every few seconds and record data for a number of minutes without moving the hovercraft. Is the offset stable over time?

9 Setting up the 5V Atmel Processor [15 pts.]

Now that you are experts in configuring Atmel processors, you should setup the 5V Atmel processor on the hoverboard. The sample code does not contain any files for the 5V processor. You should create a new directory for the 5V processor code and copy the build (you may need to modify the list of sources) and programming scripts. Most of the code should be very similar. Ideally, you could create a set of common code that will be shared between the two processors and only customize a few particular aspects that are different. In this section there are brief instructions reminding you of the differences between the processor configurations.

9.1 LEDs and Buttons

You should configure the LEDs and buttons on the 5V Atmel processor. Note that they are on different pins than on the 3.3V processor.

9.2 Serial Ports

The 5V Atmel processor only has a connector for one UART (UART0). The other is dedicated to a radio connector slot. We will use this later in the semester to have radio control of the hoverboard so that you can operate it remotely. So you only need to configure UART0, although you are also free to configure UART1. The main difference between the 5V and 3.3V Atmel is that the 5V processor runs at 20MHz.

Notice in the schematic that the TX pins on the serial ports have diodes and pull-up resistors. This allows the connection of 3.3V UART devices to the 5V processor by only allowing the output of 3.3V on the TX line.

9.3 Analog to Digital Converter

The 5V Atmel ADC has current monitors and battery voltage monitors as well as a number of free ADC pins. The configuration of the ADC will be very similar to that of the 3.3V processor. The main difference

[2] Note that the gyro offset can also be impacted by operating temperature and other factors that may change over time.

is the clock speed and the fact that AVCC is 5V instead of 3.3V.

9.4 Monitoring Current and Voltage

Look at the first page of the hoverboard schematic. In the lower left there are two INA196 current monitor chips. These are amplifiers that multiply the voltage drop over the small value resistors (R32 and R33) that all of the current must pass through. U6 monitors the current used by the thrusters and U7 monitors the current used by the rest of the board.
Question: What is the maximum current that can be measured by both current monitors? Hint: first find the gain of the INA196 and then note that the maximum voltage the current monitors can output is 5V.
Question: Given that the Atmel has 10 bit ADC resolution, what is the minimum change in current that it can detect?

Create functions that output the current measured by both current monitors.
Question: Can you detect the change in current when you turn on or off an LED? What if you hold the reset button on the other processor down?

Overall, it is probably best not to use more than 10 or 15A continuously for all the thrusters combined. Use the LEDs on the 5V processor to provide visual feedback as to how much current is being used. Perhaps the green light could be on if it is less than 10A and the red light could turn on if you use more than 15A.
Question: Using the current monitor on the thrusters determine and report how much current the thrusters use for different PWM levels.

The 5V Atmel also has a voltage monitor circuit using R31 and R34 which form a voltage divider. Write code to convert the raw ADC reading into a voltage. This will give you the ability to monitor the battery voltage and, perhaps, shut down thrusters if the battery voltage drops too low.
Question: What is the maximum battery voltage the 5V Atmel could read?
Question: (Optional) Record and plot the battery voltage readings with the thrusters off and with one or more on at different levels.

10 To Hand In

You should designate one person from your group as the point person for this lab (each person needs to do this at least once over the semester). This person is responsible for organizing and handing in the report, but everyone must contribute to writing the text. You should list all group members and indicate who was the point person on this lab. Your lab should be submitted by email before the start of class on the due date. A pdf formatted document is preferred.

Your lab report should have an introduction and conclusion and address the various questions (highlighted as ***Question:***) throughout the lab in detail. It should be well written and have a logical flow. Including pictures, charts, and graphs may be useful in explaining the results. There is no set page limit, but you should make sure to answer questions in detail and explain how you arrived at your decisions. You are also welcome to add additional insights and material to the lab beyond answering the required questions. The clarity, organization, grammar, and completeness of the report is worth **20 points** of your lab report grade.
Question: Please include your code with the lab report. Note that you will receive deductions if your code is not reasonably well commented. You should comment the code as you write it, do not leave writing comments until the end. I recommend structuring your code such that for every section of this lab, a different function is called from the main loop. Then you will be able to comment out just one function to change between sections in the lab and you won't have to delete working code from previous parts of the lab.
Question: For everyone in your group how many hours did each person spend on this part and the lab in total? Did you divide the work, if so how? Work on everything together?
Question: Please discuss and highlight any areas of this lab that you found unclear or difficult.