Lab #2: Writing a Driver

In this exercise, you will write a simple device driver module in a C++ framework. You should write and test this driver carefully, as you will use it in later projects. The exercise will take one week; you will work in teams of two. Skills to practice in this lab include the following:

- Writing input/output code for a microcontroller using Special Function Registers
- Working with a touch of `class`(es)
- Documenting code with Doxygen
- Testing your code thoroughly to ensure that it is reliable

1 The Project

Write a PWM based motor driver class. Design it so that the same driver class can be used to drive different motors; for example, it must be able to operate either of the two motor drivers on the ME405 board, and (here's the challenge) it must be possible to instantiate two motor drivers at the same time, each driving a different motor. The format of the lines used to create the motor driver objects should be as follows, with as many parameters as needed:

```
my_motor_driver* p_motor_1 = new my_motor_driver (param1A, param2A, param3A);
my_motor_driver* p_motor_2 = new my_motor_driver (param1B, param2B, param3B);
```

We have two objects, which in this example are pointed to by `p_motor_1` and `p_motor_2`, instantiated from the *same* class; the difference between the two objects is in the constructor parameters. *Hint: Some parameters are pointers to I/O registers, for example* `&DDRC` *which is of type* `volatile uint8_t*`. Then to control two drivers:

```
p_motor_1->set_power (100);
p_motor_2->set_power (-220);
```

Your `set_power()` method should take as its parameter a **signed** 16-bit number. Positive numbers cause torque in one direction, negative in the other. This configuration works well for closed-loop motor control in future labs. Also implement dynamic "braking" (actually damping), in which the driver shorts the motor's leads together, by providing a `brake()` method. You can choose to either brake at a given strength by giving a parameter to `brake()` or to always brake at full braking torque with `void brake (void);`

You will use your motor drivers in later class projects. Therefore, you should test the drivers carefully with a test program which makes the motor driver do all the things it will have to do: forward and reverse motion, braking, and freewheeling. Consider using a potentiometer or two, along with your A/D converter driver from Lab 1, as the user input to to control the motor speeds. We may be able to scrounge up some used potentiometers in lab – they seem to come and go from quarter to quarter.

It's easiest to make a test program in which the A/D converter object and both of the motor driver objects are instantiated in `task_brightness.cpp`. You can print things by using the `DBG()` macro or use the "`*p_serial << things;`" style in `task_brightness.cpp`.

As you begin Lab 2, you should make a copy of your source files from Lab 1. However, you should **not** copy your entire Lab 1 directory. Instead, create a new set of empty directories (including `lib` and the ones under it) for Lab 2, then copy the source files (`*.h`, `*.c`, `*.cpp`, etc.) from your Lab 1 directories into your new Lab 2 directories. Then use your Subversion program to add the Lab 2 files. Then when you commit your changes, the Subversion repository will have copies of both labs' files and you can work on your Lab 2 files while leaving your Lab 1 files for reference (and for the grader if needed).

Rather than trying to write your motor drivers from scratch, you should make copies of `adc.h` and `adc.cpp`, then change the names and comments. Make sure to change the comments as soon as you've copied the files; don't wait until later. Changing the name in the macro `#ifndef _AVR_ADC_H_` to another name is very important.

Your grade will be determined by how well you do your work and how much you've done – kind of like some judged sports (degree of difficulty multiplied by a quality score). Quality means well commented code which has been thoroughly tested, good documentation of the code, and meaningful, documented testing. The grade will include points for code quality (neat and commented), code function (it works), thorough testing, and clear Doxygen comments. The quality of the **memo** you write will also be a component of the grade.

2 References

There is a schematic and layout for the ME405 circuit boards at
`http://wind.calpoly.edu/ME507/lab_board405.shtml`

Information about the ME405 software library is at
`http://wind.calpoly.edu/ME405/doc/`

Information about the FreeRTOS real-time operating system can be found at
`http://www.freertos.org/a00106.html`

The library used with our AVR compiler is `AVR-libc`; its reference is at
`http://www.nongnu.org/avr-libc/user-manual/modules.html`

You should use your favorite search engine to dig up the datasheet for the ST Microelectronics VNH3SP30 motor driver chip. The data sheet shows how signals from the microcontroller make the motor driver chip work. The connections on the circuit board from the microcontroller to the two VNH3SP30 chips are shown in the following table.

	VNH3SP30 Pin	ATmega1281 Pin	Function
Motor 1	INA	Port C pin 0	Mode select bit A
	INB	Port C pin 1	Mode select bit B
	DIAGA/B	Port C pin 2	Output enable and diagnostics
	PWM	Port B pin 6	PWM control, pin OC1B
Motor 2	INA	Port D pin 5	Mode select bit A
	INB	Port D pin 6	Mode select bit B
	DIAGA/B	Port D pin 7	Output enable and diagnostics
	PWM	Port B pin 5	PWM control, pin OC1A

3 Deliverables

On the due date, you'll demonstrate your program in lab and turn in the following:

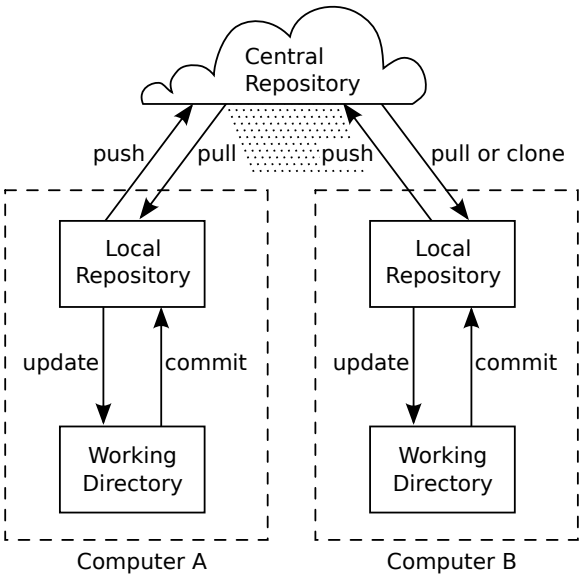
- A one page memo describing what your program does. The memo should discuss the results of your motor tests in some detail.
- A printout of the Doxygen documentation for your A/D converter class. Please print code and Doxygen output double-sided if you're using the lab printers.
- Printouts of the source files you have written or substantially modified. This certainly includes the `*.cpp` and `*.h` files for the motor driver class you created. Please do not turn in printouts of other files which you didn't write or modify.
- Please make a zip file of your source code, name the zip file with the last names of your team members (as in `JonesSmith.zip`) and email the zip file to your instructor at `me@me.me.calpoly.edu`.

4 Using Mercurial Version Control

You're probably accustomed to using a cloud service to hold your data and synchronize the latest copy of each file among many computers. This method works acceptably for some purposes, but it's not sophisticated enough to support software development. When you're writing software, it's helpful to have other abilities in addition to simple file synchronization:

- The ability to manually control when and how synchronization takes place
- The ability to compare the current version of a file to previous versions and quickly spot the differences
- The ability to *revert* to a previous working version of your program if changes cause it to quit working
- The ability to create *branches* in your software development history for trying out new and sometimes crazy designs, then *merge* branches back into your main software development if they turn out to be good ideas
- The ability to *merge* changes made by two or more lab partners who have worked on the same file at the same time

There are many version control systems which provide the aforementioned abilities. Examples are CVS, Subversion, Mercurial, and Git. In ME405 we will be using Mercurial, which is more modern and flexible than CVS and Subversion while being a bit simpler and easier to use than the very powerful (overkill for our uses) Git.



A diagram showing the organization of a Mercurial version control system is shown above.