

Lab #1: Writing a Driver

0 Introduction

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
- Managing sets of source files (`*.cpp`, `*.h`, `Makefile`, etc.) using Mercurial

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:

```
motor_drv* p_motor_1 = new motor_drv (param1A, param2A, param3A);  
motor_drv* p_motor_2 = new motor_drv (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 0, as the user input to control the motor speeds. We may be able to scrounge up some used potentiometers.

It's easiest to make a test program in which the A/D converter object and both 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 1, you should make a copy of your source files from Lab 0. However, you should **not** copy your entire Lab 0 directory. Instead, create an empty directory for Lab 1, then copy the source files (`*.h`, `*.c`, `*.cpp`, etc.) from your Lab 0 directory into your new Lab 1 directory.

When you've prepared the Lab 1 directory, set up Mercurial to track your files. Go up to your `labs` directory (the one which contains the `lab0`, `lab1`, and `lib` subdirectories – right?) and initialize a Mercurial repository. Then add the source code files (`*.h`, `*.c`, `*.cpp`, `Makefile`, etc.) to your Mercurial set. This process is discussed in Section 4.

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 is 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>

Look up the datasheet for the ST Microelectronics VN3SP30 motor driver chip. The connections on the circuit board from the microcontroller to the two VN3SP30 chips are shown in the following table.

	VN3SP30 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

* DIAGA/B pins must be driven by AVR pins set as inputs with pullup resistors turned on.

3 Deliverables

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

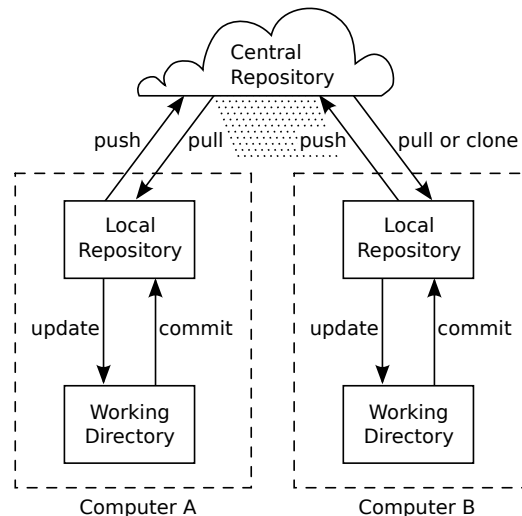
- A one page memo describing what your program does. The memo should discuss the results of your motor tests in some detail. It should also contain a link to your source code on the Mercurial repository.
- A printout of the Doxygen documentation for your A/D converter class (class documentation only, not file documentation). 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.
- Make sure your latest source code has been uploaded to the Mercurial repository at <http://wind.calpoly.edu/hg/mechaXX>.

4 Using Mercurial Version Control

Mercurial is somewhat like a cloud server in that it holds a copy of your files somewhere. It has special abilities which are important for software development also:

- The ability to manually *control* synchronization
- 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 and manage *branches* in your development process for trying out new designs
- The ability to *merge* changes made by two or more teammates who have worked on the same files 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 easier to use than Git, which is overkill for us.



A diagram showing the organization of a Mercurial version control system is shown above. It might represent the situation when two lab partners, Alice and Bönsputt, are working on the same project, each on her or his own computer. There are two important processes shown in the diagram:

- A local working copy of all source files is kept on each computer. Whenever a substantial set of changes has been made, the programmer will **commit** those changes to the local repository on that same computer, thus creating a new revision of the project. The local repository holds all previous versions of the project's files.

- Files from the local repository can be **pushed** to or **pulled** from a central repository which is usually kept on a remote computer.

4.1 Working with Versions Locally

Mercurial can be used to keep a history of all the versions of a project's files on a local computer. If Alice is always working alone, she will first **create** a local repository by entering the command

```
hg init
```

into a terminal window. This will create a hidden directory called `.hg` in which the current revision of Alice's files *as well as all previous committed revisions* will be saved. Next, Alice will **add** files to the set of files which Mercurial will track. This is done with the `add` command, using specific file names or wildcards in whatever combination is convenient:

```
hg add Makefile
hg add *.h *.c *.cpp
hg add lib/*
```

In this example, we assume that `lib` is a subdirectory of the current directory and that Alice needs to have Mercurial track all files in that subdirectory. The `add` command does not save anything; it only adds files to the list of files that Mercurial will track. To record the current state of all added files into the local repository, Alice types ¹

```
hg commit -m 'The initial revision of my program'
```

The string in single quotes following the `-m` is a log message which should say what changes were made between the previous revision and the one currently being saved. When committing, Mercurial creates a new revision number and stores a *change set* – the differences between the current revision of the files and the previous revision. For the first commit, all added files are copied into the local repository. Mercurial can use the originally committed files plus change sets to reconstruct the state of all tracked files as they were committed with any revision number.

Having convenient access to all one's previous versions can save a great deal of time when modifications to a program cause it to quit working. One of the easiest ways out of such a mess is to **clone** a previous revision into a new directory and work there. When Alice is working with revision 13, modifications to the egg timer driver cause the fuel injector timing to malfunction – a very difficult error to diagnose. She then navigates to the directory *above* her project directory named `my_dir`, probably by typing `cd ..`, and runs

```
hg clone -r 12 my_dir rev_12
```

This causes a copy of the files in directory `my_dir` to be made in a new directory called `rev_12`, except the files in `rev_12` are as they were when the

previous revision, number 12, was committed (that's what `-r 12` means). Now Alice has a working program again and can try another method to implement those egg timer modifications.

4.2 Synchronizing with a Central Repository

A Mercurial server can be thought of as a “Dropbox™ server” which knows how to work with source code. Most communication with the server is done by **pushing** and **pulling** change sets to and from a central repository on the server. Before anything can be pushed or pulled, however, the local repository must be synchronized with the central server.

- Many projects begin with an existing set of files on a server. If there is a pre-existing copy of the files already set up, one clones a local copy of the files on the server and modifies that local copy:

```
hg clone http://wind.calpoly.edu/hg/mecha42
```

This creates a new directory under the directory in which the terminal was working when the `clone` command was run.
- Sometimes, one begins with an empty central repository copy and must add a set of files which were not previously in the central repository. In this case, it's still easiest to clone a copy of the empty central repository from the server, then put the files into the local working copy thus cloned, creating revision 0.
- If one has an existing local Mercurial repository whose contents are to be newly added to a central repository, one must *merge* the repositories – see below.

Once the central and local repositories have been properly linked, the pushing and pulling can commence. To send the latest committed version of her files to the central repository, Alice types into her favorite terminal

```
hg push
```

and the central repository's files are updated with the *tip* revision from the local repository. The tip is the most recently committed version of the files. At some point, Bönsputt (remember him?) is ready to join the project using his home computer, on which none of the project files are present. Bönsputt needs to *clone* the project files to his computer, so he `cd`'s his terminal to a directory under which the project directory tree can be kept and types

```
hg clone http://wind.calpoly.edu/hg/mecha42
```

This action makes a new local repository and puts the latest version of the files *which have been pushed to the central repository* onto Bönsputt's computer.

But Alice is still working on the project on *her* computer at the same time. What happens when both programmers work on the project separately, then try to commit and push their changes to the central repository?

4.3 Branches and Merging

Mercurial is designed to handle the maintenance of source code which is being worked on by a community of developers; community work implies imperfect coordination and conflicts. A common type of conflict occurs when two or more partners have worked on the same project – or worse, the same files – and made different changes. When each partner commits a set of changes to a different copy of the project, two *branches* have been created. The definition of branches is that each branch is a linear path of change sets which at some point diverged from the same parent, as branches on a tree diverge from a common trunk.² The command `hg branch` shows the name of the current branch, and the command `hg branch name` assigns a name to the current branch. You don't have to use the `hg branch` command if you have no need for branch names.

When a team has created multiple branches, the branches will at some point need to be *merged* into a single program which incorporates the changes from all the branches – or at least the good ones. If a branch turns out to be bad, it can just be abandoned; archive it in case there's something good in there, then clone a good version of the project from another branch.

Merging branches becomes necessary when one team member attempts to push changes to the central repository but the other member has already pushed different changes. Alice attempts to push changes when Bönsputt has already pushed different changes and gets a message similar to:

```
pushing to http://centralHgServer.net
searching for changes
abort: push creates new remote head 0f249283915d!
(pull and merge or see "hg help push" for details ... )
```

Alice should therefore pull Bönsputt's changes from the central repository:

```
pulling from http://centralHgServer.net
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 7 changes to 2 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
```

As the messages indicate, the conflicting changes need to be merged into a single set of files – the team's current program. Alice merges:

```
hg merge
merging myfile.lol
was merge successful (yn)?
```

If the teammates' modifications were all in different files or on different lines of the same files, merging is usually automatic. If the same lines in the same files were changed in different ways, merging requires human

intervention; Mercurial automatically pops up a merge program such as `meld` or `kdiff3` – usually a GUI program – which helps Alice to manually reconcile the changes. Sometimes complex changes require extra fussing around and the use of the `hg resolve` command.

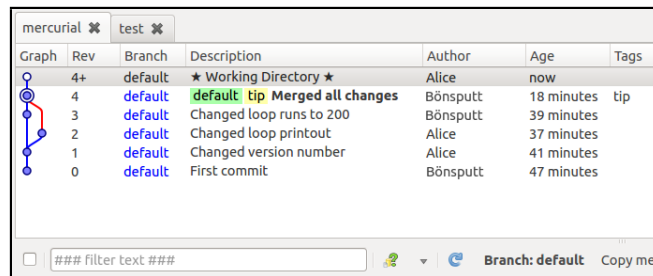
When the merging is complete, Alice should make sure the program can still be compiled and run. Then she commits the merged version in the usual way:

```
hg commit -m 'Committing merged changes'
```

and then pushes the merged, committed version to the central repository:

```
hg push http://centralHgServer.net
pushing to http://centralHgServer.net
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 12 changes to 4 files
```

A graphical view of a project's revision history which includes the creation and merging of two branches is shown below.



Graph	Rev	Branch	Description	Author	Age	Tags
4+	4	default	★ Working Directory ★	Alice	now	
	4	default	default: tip Merged all changes	Bönsputt	18 minutes	tip
	3	default	Changed loop runs to 200	Bönsputt	39 minutes	
	2	default	Changed loop printout	Alice	37 minutes	
	1	default	Changed version number	Alice	41 minutes	
	0	default	First commit	Bönsputt	47 minutes	

A popular Mercurial GUI is TortoiseHg, which runs on Windows™ and better systems. TortoiseHg integrates itself with the system's file manager, adding symbols to file icons so that one can see at a glance the states of the files (modified, saved, *etc.*). There is also a TortoiseHg "Workbench" which shows a graphical diagram of a project's revision history, allows commits and pushes and pulls without using the command line, and gives many options with a right-click of the mouse. The graphical revision history view shown above was part of a TortoiseHg Workbench window.

4.4 Other Things

A handy feature of Mercurial is the ability to easily create an archive containing all the files in a given revision set. It can be used as follows:

```
hg archive my_project.zip
```


This creates an archive file in the popular `zip` format, ready to save to a USB drive or email to colleagues. It is also possible to use links on a Mercurial server's web page to get compressed files of the latest version on the server in `zip`, `gz`, or `bz2` format.

If a file called `.hgignore` is created in a project directory, Mercurial will not display or ask about matching files. It can even ignore `.hgignore`:

<code>syntax: glob</code>	<i>Specifies plain syntax, not fancy regular expressions³</i>
<code>.hgignore</code>	<i>Ignore the ignore file</i>
<code>junk_directory/*</code>	<i>Ignore everything in the given directory</i>
<code>*.elf</code>	<i>Ignore binary files whose names end with <code>.elf</code></i>
<code>*.hex</code>	<i>Ignore binary files whose names end with <code>.hex</code></i>
<code>*~</code>	<i>Ignore automatic backup files that end with a tilde</i>

There are dozens of other features in Mercurial, far too many to list in a short introduction such as this. In addition to web resources, one can ask the program itself for assistance by typing

```
hg help
```

Whenever you use any revision control system, make sure your commit messages are more meaningful than these shown by Munroe at `xkcd.com`:

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE.	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAHAHAHAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT
MESSAGES GET LESS AND LESS INFORMATIVE.

Notes

¹When one runs the `hg commit` command *without* the `-m` followed by a log message, Mercurial will automatically run a text editor with which the user should enter a log message.

²A shrubbery, whose trunks grow from common roots, just confuses the analogy. Ignore it.

³Regular expressions are a sophisticated and very useful way to specify complex patterns in file names and other text, and they're worth learning.