# Lab #0: What AVR

## 0 Purpose

The objective of this lab is to help you become familiar with C++, the AVR processor, and some of the tools you will be using in this course. The exercise will take one week, and you will work in teams of two or three as directed by the instructor.

If you have written programs in assembly language, you've probably worked with a single source code file that contained almost your entire program. C++ is used very differently; your programs will consist of many files which are linked together. You need to learn to use an efficient set of tools for managing projects that consist of many files, tens to hundreds per project. You will also be working on teams; the tools you learn here will help you to work efficiently with others to lighten the burden of development.

## 1 Hints

Here are some things about which it may help to know in advance:

- The computers in the lab do not run Windows™, they run Linux (the currently used version is Ubuntu 14.04 LTS, "Trusty Tahr.")
- These computers are currently set up for standalone use; in other words, each one has a complete set of tools and its own copy of all your files. This means that when you save files and settings on one computer, your stuff **will not be on the other computers** in the lab unless you copy it. You will soon learn to use tools which overcome this limitation very easily. A nice thing about the standalone computers is that even when the network and/or servers go down, you can still get work done.
- All the tools used in this course are free software. That means if you'd like to work at home sometimes, you can download everything you need; the tools are freely available for Linux, Unix™, Mac OS™, and even Microbe-soft™operating systems.

## 2 Setup and Familiarization

The following procedure is intended to get you up to speed quickly with the programming tools in the lab.

**(a)** Wake up a PC in the lab and log on. You will be given an account and password by the instructor. Familiarize yourself with the desktop. We have LibreOffice (a version of OpenOffice), which is a near clone of that other office suite; there are many other programs which work similarly to those to which you're accustomed. You can customize the applications in the launcher bar; those settings will remain on the particular computer you've customized. You can also change the screen background and launcher icon size if you like (right-click on the background).

**(b)** You're probably used to doing most file moving, copying, etc. with a file manager, so let's try one out next. Use the home folder icon on the launcher bar. By default, files are opened with double clicks but you can change this to single-click if you prefer. You can customize sets of bookmarks in a panel on the left, and your bookmarks won't be erased every night. Use Ctrl-T to open more tabs in the file manager; you can drag and drop files onto tabs.

**(c)** Start up a command shell. Use the "Dash Home" button at upper left or press Alt-F2, then type terminal (you probably only need to type term and the system will figure out what you want). Learning just a few commands will allow you to make use of the command shell:

| | |
|---|---|
| `pwd` | Print Working Directory, *i.e.* Where am I? |
| `cd` *dir* | Move to directory *dir*; `cd ..` moves one up |
| `ls` | List all the files in the current directory |
| `dir` *or* `ls -l` | A detailed list of files, *in color!* |
| `mkdir` *newdir* | Make a new directory named *newdir* |
| `rm` *file* | Remove (delete) a file |
| `make` | Compile your program into machine code |
| `make install` | Compile program, then download it to the AVR chip |
| `make clean install` | Clear out all old compiled code, re-compile everything with the latest version, and download it all |
| `make help` | Show some of the things the `make` utility can do |

**(d)** Download a zip file containing Lab 0 starter code from the course Poodle page. Make a subdirectory under your home directory such as (`/home/mechaXX/labs`) to hold your programs. Double-click on the zip file in a file manager, then extract the contents `lab0` into your source code directory. You should now have lots of files in three directories `lab0`, `lib`, and `task_comm`. These three directories should be contained in your `labs` directory.

**(e)** You'll need to edit source files – not just one, but several in a project. Kate, the "K Advanced Text Editor", is an excellent file editor with line numbers, syntax coloring (which works for C/C++ and many other languages), the ability to open and close groups of files in sessions, and more. Open all the *.h and *.cpp files in your directory with Kate. When they're open, use Sessions → Save from the menu to save the file list. Have a look at all these files. Soon they'll make sense! *Really.* To see line numbers, use the F11 key to turn line numbering on, or better yet, use menu item Settings → Configure Kate... to turn line numbers on by default. You can use a terminal inside Kate for running make; click the Terminal tab at the bottom of the Kate window and stretch the terminal window with the "⇕" pointer if needed.

**(f)** Go to the terminal window and type `make clean all`. You should see a whole bunch of messages which end something like the following (with some different file names):

```
Compiling:  ../lib/serial/emstream_pointer.cpp --> build/emstream_pointer.o
Compiling:  ../lib/serial/emstream_uint16_t.cpp --> build/emstream_uint16_t.o
Compiling:  ../lib/serial/emstream_uint32_t.cpp --> build/emstream_uint32_t.o
Compiling:  ../lib/serial/emstream_uint64_t.cpp --> build/emstream_uint64_t.o
Compiling:  ../lib/serial/emstream_uint8_t.cpp --> build/emstream_uint8_t.o
Compiling:  ../lib/serial/rs232int.cpp --> build/rs232int.o
Library-ing:  (*.o) --> build/lib_me405.a
Compiling:  main.cpp --> build/main.o
Compiling:  task_user.cpp --> build/task_user.o
Compiling:  task_brightness.cpp --> build/task_brightness.o
Compiling:  adc.cpp --> build/adc.o
adc.cpp:89:10:  warning:  unused parameter channel [-Wunused-parameter]
   uint16_t adc::read_oversampled (uint8_t channel, uint8_t samples)
            ^
adc.cpp:89:10:  warning:  unused parameter samples [-Wunused-parameter]
Linking:  build/main.o build/task_user.o build/task_brightness.o build/adc.o
build
     text     data     bss     dec     hex filename
    15878      186    6070    2213    5676 build/lab0.elf
```

There are compiler warnings. Most warnings indicate problems or things that need to be finished; these warnings will go away when you complete your assignment. In general, make sure your finished code compiles without warnings.

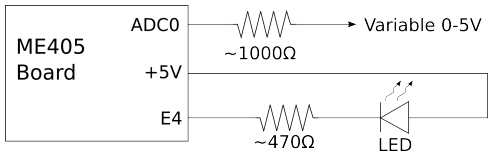
**(g)** To download your program to the AVR processor, first connect the programmer as described in Section 5.2. Then run make install and you should get:

```
avrdude -p m1281 -c usbtiny -V -Uflash:w:example.hex
avrdude:  AVR device initialized and ready to accept instructions
Reading | ################################################## | 100% 0.00s
avrdude:  Device signature = 0x1e9307
avrdude:  NOTE: FLASH memory has been specified, an erase cycle will be
performed
                To disable this feature, specify the -D option.
avrdude:  erasing chip
avrdude:  reading input file "example.hex"
avrdude:  input file example.hex auto detected as Intel Hex
avrdude:  writing flash (4056 bytes):
Writing | ################################################## | 100% 1.10s
avrdude:  4056 bytes of flash written
avrdude:  safemode:  Fuses OK
avrdude done.  Thank you.
```

If there are problems, ask your instructor for help. Sometimes typing an error message surrounded by quotation marks into a search engine can lead to a solution. You're unlikely to be the first person to have a given problem.

**(h)** To see if the program is running, use a "dumb terminal" connected to the microcontroller through a USB-serial connection (see Section 6). Run a terminal program, press some keys in it. "WTF" means "What's That Function?", by the way.

**(i)** Connect an LED from Port E, Pin 4 of the ATmega1281 to the +5V output; also connect a variable voltage supply to A/D channel 0, through a resistor for safety.




*Note: Don't leave out the resistors, and remember to connect ground wires.*

**(j)** At this point you are able to edit program files, compile the code, download it to the microcontroller, and see the result of running the program. You're ready to proceed to the main part of this exercise.

# 3 Project Procedure

For this project, you are to take on the role of a consulting engineer who is working on code for a customer. Your customer has a program (the example program which controls the brightness of an LED) and has a very basic closed-source A/D driver. Your customer needs a reliable, well written **and documented**, open source version.

**(k)** As the program is supplied to you, it reads the A/D converter and uses the result to control the brightness of an LED. Your assignment is to replace the obfuscated and incomplete code that runs the A/D converter with properly written, portable (among AVR processors), well commented code. The code which makes the A/D converter work is in files adc.h and adc.cpp. You are asked to do the following:

- Fix the constructor `adc::adc()` so that it is clear and readable. This pretty much means that you need to re-write the code in there. The code in the constructor should turn the A/D power on (*i.e.* enable it), select the reference voltage source (AVCC with external capacitor at AREF pin is recommended), and set the clock prescaler to a division factor of 32. Use the ATmega1281 data sheet from www.atmel.com as a reference; read through Section 26, "ADC – Analog to Digital Converter" for information about how to make the A/D work.
- Fix the code in the method `adc::read_once()` so that it is clear and readable. This means re-writing it too. To do a conversion, you must first set the multiplexer to select a channel; we only need channels 0 through 7 in single-ended mode. Make sure you don't mess up other bits in the ADMUX register when you change the channel bits. Then start a conversion, wait for the conversion to be complete (*Hint: watch bit* ADSC), get the results, and return those results from the function.
- Write code for method `adc::read_oversampled()` so that it works. It's perfectly fine for this method to call the `read_once()` method so that you don't have to write the A/D reading code over. *Note: Watch out for overflow in your variables! If you add up too many samples, you might overflow a 16-bit number. Limiting the number of samples which can be averaged is an acceptable solution to this problem, as long as the limit is reasonable (not one!).*

- Put code in `operator << (emstream&, adc&)` to make it useful. It is recommended to print the contents of the A/D control registers such as ADMUX and ADCSRA in binary, then show the current reading on one or more channels.

**(l)** All your code must be properly commented. This means you must have meaningful Doxygen comments for every method. You should edit the file doxygen.conf so that the project name and version number are correct (you may pick a version number as you like). Run make doc to produce the documentation. Then open the file in directory doc/rtf under your source file directory in a word processor. Delete the title page, indices, and so on; include just the documentation for class adc as part of your report (see Section 4 below). *Note: In the past, students have often printed out their Doxygen documents and turned them in without even looking at them. The grader can tell when the documents say things such as* Example Project Name – CHANGE THIS NOW. *Please submit more dignified documentation.*

**(m)** Test your A/D code. Give varying voltages to the input at A/D channel 0; record the input to the A/D and the output number. Compute and graph the error between expected and actual A/D output for different input voltages. Does it make sense to compute the percentage error, with $N_{ACT}$ and $N_{EXP}$ being actual and expected A/D outputs for a given input voltage, as $\%ERR = (N_{ACT} - N_{EXP})/N_{EXP}$?

***No.*** The error should be compared to the full scale reading, not the current reading, or small errors in small readings will show up as huge percentage errors.

Note that it is better (even for grades) to find and report bugs than to have hidden bugs in your code.

# 4 Deliverables

Next week, you'll demonstrate your program at the beginning of the lab session and turn in the following:

- A short (about one page, and definitely not more than one page) memo from you to your customer. The memo should describe what your code does and show evidence that you have tested it, such as a graph showing the A/D accuracy. Including short snippets of code in your memo, if they help explain something important, can be good; format all such code snippets in a monospaced font such as `Courier`.
- A printout of the Doxygen documentation for *only* the file(s) which you changed. Only print the **class documentation**, not the file documentation. It should show that you've edited the Doxygen comments in this file to show what your program now does. This should be printed **double sided** to save some trees.
- A printout of your file adc.cpp and any other .h and .cpp files whose content you changed. Using the color printers in the lab for these is nice; make sure to print the code **double sided**. Check that you've set the font size for code printouts so that most lines of code don't need to wrap around onto extra lines.

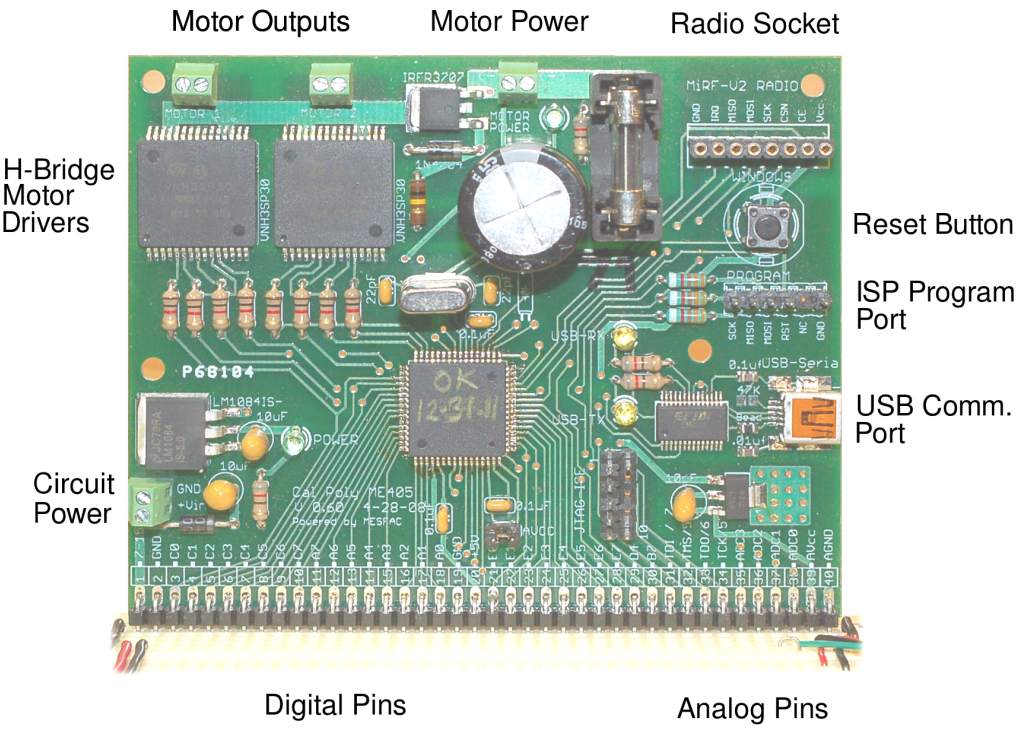
# 5 The ME405 Microcontroller Board

## 5.1 Overview

We use custom microcontroller circuit boards in ME405. These boards have been designed in Cal Poly's ME Department for use in mechatronics courses and Senior projects. They are comparatively simple to work with, fairly robust, and relatively easy to repair when "Learn by Doing" becomes "Learn by Breaking Stuff." Of course, you are expected to treat these boards with care and not damage them unnecessarily. The most important features of the board are the following:

- An Atmel ATmega1281 microcontroller with 128KB of flash memory for your programs, 8KB of RAM for data, and lots of peripherals such as A/D and timer/counters
- Two VNH3SP30 motor controllers, each capable of controlling a DC motor with more than 10 amps at up to 24V
- A USB-serial interface through which the microcontroller communicates with a PC
- Reverse supply voltage protection and voltage regulators which supply separate power to the motors, the digital circuitry, and the analog circuitry
- ISP (In-System Programming) capability through both SPI and JTAG connectors
- 40 pins on one side of the board, for mounting the board in a breadboard and connecting many types of external devices

A photograph of an ME405 board version 0.60, with labels for some of the most important connections about which you need to know, is shown below.



## 5.2 Minimal Quick Start Connections

The following procedure is used to get an ME405 board up and running in its most basic configuration.

- You should begin with an ME405 board which is plugged into a breadboard.
- Connect circuit power for the microcontroller. You can connect the board using the green screw terminals marked GND and +Vin, or you can run wires on your breadboard which go to the leftmost two pins labeled 7–15V and GND. If using a lab power supply, the supply should be set to 7 volts and its current limit should be set to about 0.2 amps. Make sure to set the current limit; with the default current limit, the board **will not work.** The board can also be powered by a common 9 volt battery, a 7.2V NiMH pack, or a 9 to 12 volt DC "wall wart" plug-in power supply.
- To program the microcontroller, first connect an ISP programmer to the PC using a USB cable. A green light in the programmer's black box should illuminate **(Fiat lux!).** Then connect the programmer to the 5-pin (six pins in a row with one missing) header marked PROGRAM. [1] The ISP programmer is used **only** for downloading programs to the microcontroller.

  An alternative programmer is a parallel port programming cable. If using this type of cable, look in the file called `Makefile` for the programmer specification:

```
# This define is used to choose the type of programmer from the
following options:
# bsd - Parallel port in-system (ISP) programmer using SPI interface on
AVR
# jtagice - Serial or USB interface JTAG-ICE mk I clone from ETT or
Olimex
# bootloader - Resident program in the AVR which downloads through
USB/serial port
# PROG = bsd
# PROG = jtagice
# PROG = bootloader
PROG = usbtiny
```

  ...and make the `bsd` programmer active by removing the `#` character from in front of the `PROG = bsd` line and putting a `#` character in front of the `PROG = usbtiny` line.
- In order to see what the microcontroller is doing, attach a mini-USB cable to the board's USB communications port marked USB-Serial with the "I" sometimes cut off. Then attach this cable to a PC, laptop, or netbook. Open up a serial port terminal on the PC (see Section 6 below for information about serial terminal programs). The microcontroller can then "talk" to the PC through this connection. In the future, we may use wireless modules for the same purpose.

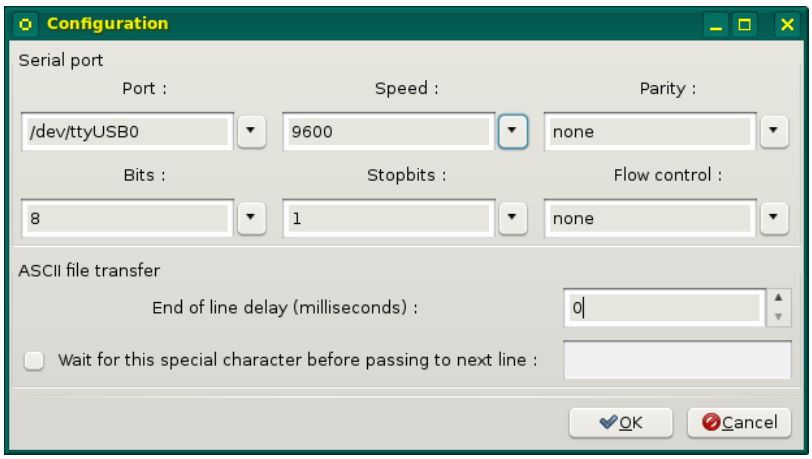
[1] It is possible to install a bootloader on the microcontroller and use the USB communications port or even a wireless radio to download programs; however, doing so is not as quick or convenient as using an ISP programmer.

# 6 Terminal Emulators

Terminal emulators are programs which act like the text-only computer consoles from the days when computers with the power of an AVR took up entire rooms. They are still very efficient ways to communicate with small computers. Windows™ users may be familiar with the program "Hyperterminal." A much better terminal program is PuTTY, which is free, open source, and easy to find with a search engine. On our ME 405 Linux computers, we commonly use GTK Term, PuTTY, and Minicom. You are welcome to use any of these or another program if you find a better one.

## 6.1 GTK Term

This terminal emulator program is the easiest to use. Start it with the Ubuntu dashboard, where typing gtkterm gets you an icon called Serial port terminal. To set GTK Term's options, choose the menu item `Configuration` → `Serial Port...` and make sure the contents of the resulting dialog box look like the following:




Use menu option `Configuration` → `Save Configuration...` to save this configuration with the name `default` so that each time GTK Term is started, it will use these settings.

GTK Term isn't very good at saving serial data to a file. Minicom works better for this. If you need to save data but don't need to watch it in a terminal, you can also type a command such as cat /dev/ttyUSB0 > myfile.txt into the command console.

## 6.2 PuTTY

This program is sort of like a fancy version of GTK Term which also does Secure Shell logins to other computers and other things we don't need. Probably the best thing about PuTTY is that it has both Windows™and Linux/Unix versions.

When you start PuTTY, choose the category Serial in the box at left; enter /dev/ttyUSB0 for the serial line. Leave the speed and bits at 9600, 8, and 1. Both parity and flow control must be set to None. Click the Open button and PuTTY is up and running. You can save your setup as a "session" before clicking Open so that if you start PuTTY with the command putty -load *session*, then instead of making you fill out a dialog box, PuTTY will start up immediately with the session settings you saved as *session*.

### 6.3 Minicom

This program runs in a terminal window, and it can also be used on computers which are so old they don't run graphical interfaces. It is started by typing `minicom` in a command console. It displays a text screen which looks like the following:

```
 Welcome to minicom 2.3

OPTIONS: I18n
Compiled on Oct 24 2008, 06:47:12.
Port /dev/ttyUSB0

Press CTRL-A Z for help on special keys









CTRL-A Z for help | 9600 8N1 | NOR | Minicom 2.3 | VT102 | Offline
```

Minicom commands are entered by typing `Ctrl-A` followed by a letter. The most important combination is `Ctrl-A Q` for quit (the "Q" can be upper or lower case). You also need to use `Ctrl-A O` to adjust program options. When you first start Minicom, the options might not be set properly. Type `Ctrl-A O` and you should see the main options menu, shown below left. Choose `Serial port setup` and make sure that the options are configured as shown below right:

```
+-----[configuration]------+        +----------------------------------------------+
| Filenames and paths      |        | A -    Serial Device      : /dev/ttyUSB0       |
| File transfer protocols  |        | B - Lockfile Location     : /var/lock          |
| Serial port setup        |        | C -   Callin Program      :                    |
| Modem and dialing        |        | D -  Callout Program      :                    |
| Screen and keyboard      |        | E -    Bps/Par/Bits       : 9600 8N1           |
| Save setup as dfl        |        | F - Hardware Flow Control : No                 |
| Save setup as..          |        | G - Software Flow Control : No                 |
| Exit                     |        |                                                |
+--------------------------+        |    Change which setting?                       |
.                                   +----------------------------------------------+
```

If you change any options, make sure to choose `Save setup as dfl` before you exit the `[configuration]` menu. This saves your settings as the default so the next time you start Minicom, these settings will be in effect.

One other helpful capability of Minicom is recording serial output in a file. The command `Ctrl-A L` is used to save data to a file. Remember to press `Ctrl-A L` *again* when it's time to finish capturing data, then choose `close`. This will ensure that all your data is saved in the file.