# CURIE Academy, Summer 2014
# Lab 2 Notes: Computer Engineering – Software Perspective

Prof. Christopher Batten
School of Electrical and Computer Engineering
Cornell University

The field of computer engineering is at the interface between hardware and software and seeks to balance the tension between application requirements and technology constraints. In Lab 1, you explored the field of computer engineering from a hardware perspective by gradually building a simple binary adder. In this lab, you will explore the field of computer engineering from a software perspective by gradually building a mobile robot control application. These lab notes provide a brief survey of background information relevant to understanding the purpose and context for the lab.

As illustrated in Figure 1, computer systems can be viewed as a stack of abstraction and implementation layers from applications at the highest layer to technology at the lowest layer. In these lab notes we will briefly discuss the application, algorithm, programming language, operating system, and instruction set layers as they relate to our mobile robot control application. In the actual lab session, you will have an opportunity to put what you have learned into practice. We will focus on some layers more than others, but by the end of this lab you should have a good understanding of how computer engineers can leverage these layers to design software for future computing systems.

## 1. Application: Mobile Robot Control

Our target application is to build a mobile robot which can quickly and autonomously navigate a relatively well-known environment to find a specific target. This kind of autonomous mobile robot control is becoming increasingly common in a variety of contexts including landmine detection, planetary exploration, and underwater surveying. Figure 2 shows the specific environment that our robots will be navigating in this lab. The environment is 4′×3′ with light-colored plywood floors and 6″ tall

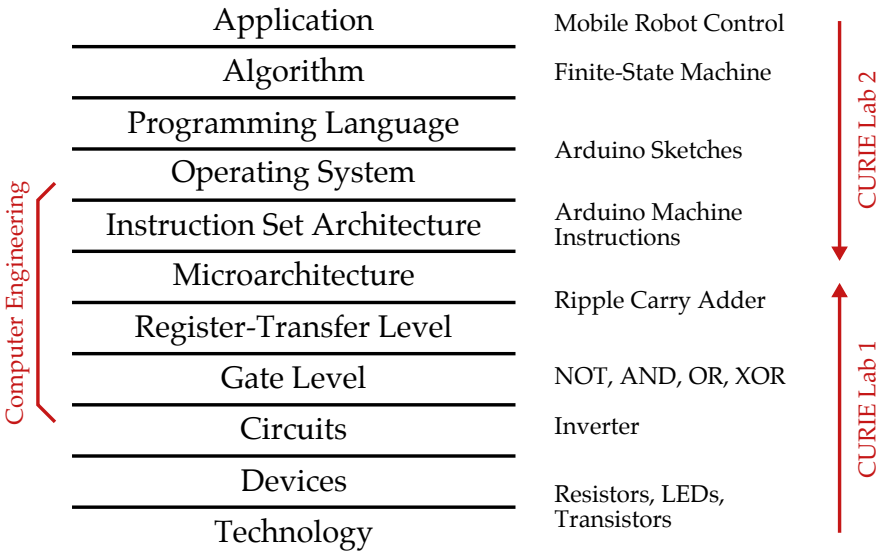


**Figure 1: Computer Systems Stack**

walls. Although you can assume that the robots start in the indicated quadrant, you cannot rely on the robot starting in a specific location or with a specific orientation. The target for the lab will be a 12″×12″ black square on the floor in the indicated location.

At the highest level, a mobile robot control application uses sensors to observe the environment and actuators to interact with the environment (see Figure 3). In this lab, we will use a variety of sensors including mechanical bump switches, analog grayscale sensors, and infrared range sensors. The primary actuators will be the two drive motors which enable the robot to move around in its environment. There are two primary classes of controllers: an *open loop controller* relies primarily on predetermined information about its environment instead of using sensors; while a *closed loop controller* continually interprets sensor data to make decisions on how to use its actuators and to observe whether or not the actuators are indeed having the desired effect. For example, an open loop controller might have the robot move in a predetermined pattern to move around obstacles, but if the robot accidentally bumped into an obstacle it would be stuck. In contrast, a closed loop controller would use sensors to determine whether or not the robot has accidentally bumped into an obstacle and if so work to navigate around the obstacle. Closed loop controllers are much more robust since sensors and actuators are far from perfect: sensors are noisy, actuators are inaccurate, and both can sometimes fail. In this lab, you must use a closed loop controller since you do not know the initial location and orientation of your robot.

Figures 4–6 illustrate the mobile robotic platform we will be using in this lab in more detail. There are two mechanical bump switches mounted on the front of the robot to help the robot detect obstacles, and there is an analog grayscale sensor on the bottom of the robot to help the robot detect the target. An infrared range sensor is mounted on the robot, although this is only used in the optional extensions at the end of the lab. There is a stack of three circuit boards on the top of the robot. The lowest board is an Arduino board which includes a small computing system called a microcontroller. The middle board is a motor-driver board which helps the Arduino control the drive motors. The top board is the prototyping board which has two LEDs, a button, a potentiometer, and space for a small solderless breadboard. The solderless breadboard is used for connecting the sensor cables to the robot. Figure 6 illustrates the various pins and components on the prototyping board. You should avoid using pins which are shaded in the figure since these are either not relevant to the lab, or are already in use internally. You will notice that several other pins are already wired up for you.

The robot has two separate battery packs: the single 9V battery powers the stack of circuit boards, while the five AA batteries power the drive motors. To power the circuit boards you need to insert

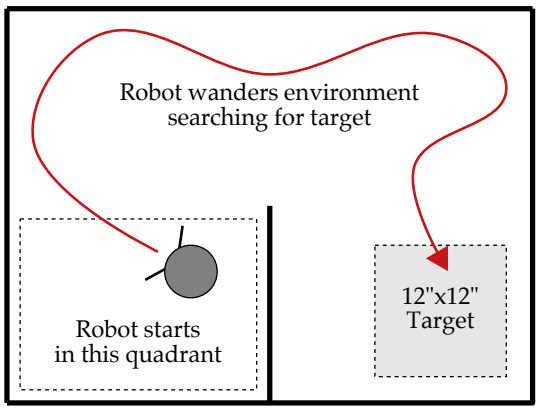


**Figure 2: Environment for Mobile Robot**

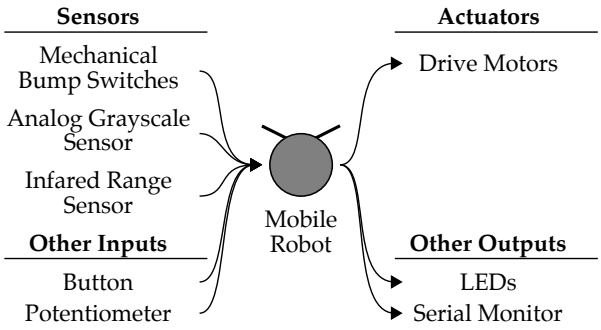


**Figure 3: Sensors and Actuators**

the barrel plug connector into the Arduino board. To power the drive motors you need to flip the drive motor switch on the top of the robot into the ON position. We have included a wooden block which the robot can sit on; this is useful for testing so that you can experiment with controlling the drive motors and the analog grayscale sensor without having the robot actually move. **Please note that if at anytime you need to immediately stop the robot from moving simply flip the drive motor switch into the OFF position!**

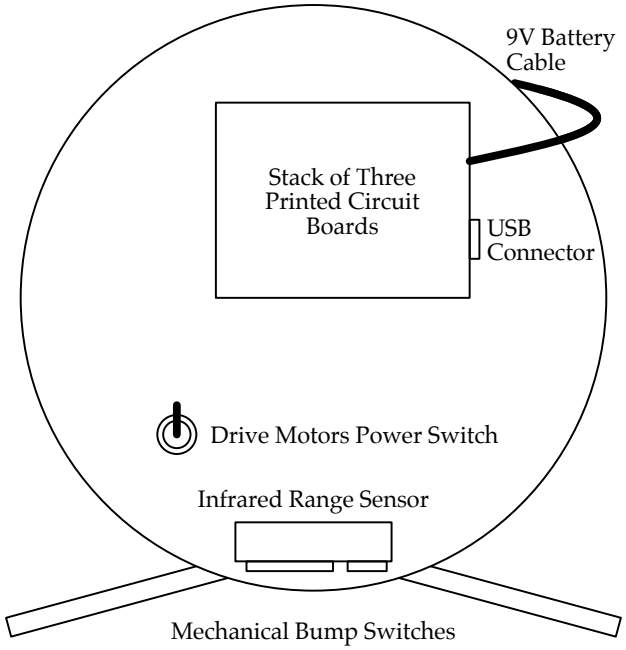


**Figure 4: Top-View of Mobile Robotic Platform**

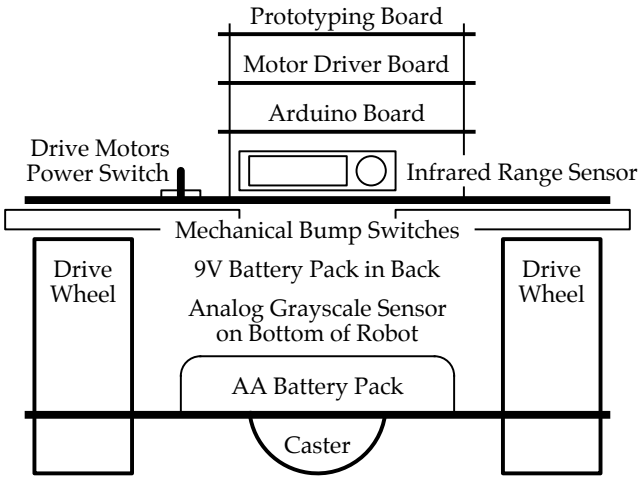


**Figure 5: Front-View of Mobile Robotic Platform**

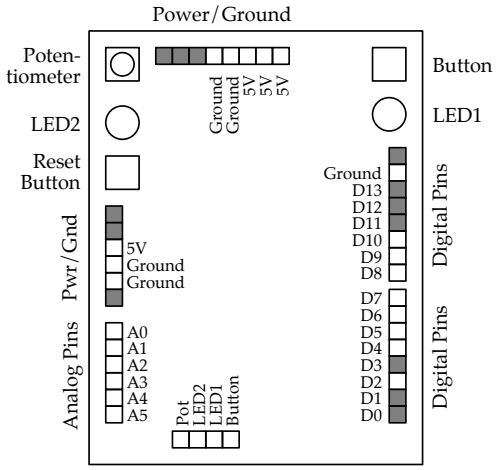


**Figure 6: Prototyping Board**

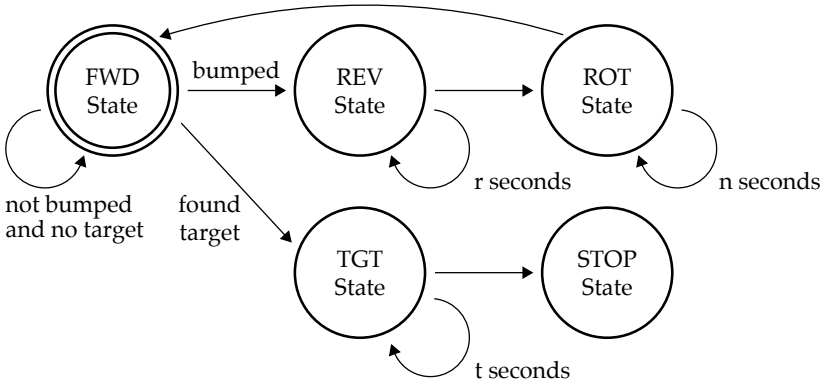


**Figure 7: Finite-State Machine Algorithm for Wander-to-Target Behavior**

## 2. Algorithm: Finite-State Machines

At the application level, we have specified the high-level goal of controlling a mobile robot to navigate an environment in search of a target and we have outlined more details on the specific environment, target, and mobile robotic platform. We now need to refine this high-level description into an actual algorithm. An algorithm is a step-by-step procedure for implementing the desired application. There are many ways to represent algorithms, but for this lab we will use a specific representation known as a *finite-state machine* (FSM) which uses a set of states and state transitions to concretely express an algorithm. For our mobile robot control application, the robot will set actuators based on its current state and use the sensor inputs to determine when to transition to a new state.

Figure 7 illustrates the five-state FSM we will be using in this lab. The robot starts in the FWD state where it simply moves forward. The robot remains in the FWD state until either it senses an obstacle using its mechanical bump switches or it senses the target using its analog grayscale sensor. If the robot senses an obstacle, it transitions into the REV state where it will move in reverse for a certain fixed amount of time. Once the robot has backed away from the obstacle, it then transitions into the ROT state where it rotates in place for a fixed amount of time. Finally, the robot returns to the FWD state to begin exploring in a new direction. If the robot senses the target, it transitions into the TGT state where it spins in a circle to indicate that it has successfully found the target. After a fixed amount of time the robot transitions into the STOP state where it remains until the algorithm is reset.

## 3. Programming Language and Operating System: Arduino Sketches

Now that we have refined our application into an algorithm, we can implement this algorithm as a program in a specific programming language. There are many programming languages each with different advantages and disadvantages, but for this lab we will use a C-like language that is provided as part of the Arduino programming framework. Figure 8 illustrates an example Arduino program which adds two numbers together and displays the result on the computer attached to the Arduino.

by blinking an LED $n$ times where $n$ is the sum. Note that in the Arduino documentation, an Arduino program is often called a "sketch". In the lab, you will gradually develop a more complicated program which implements the mobile robot control application using a finite-state-machine algorithm.

A program consists of a sequence of statements; these statements are executed one at a time by the microcontroller to ultimately execute the program. Note that comments start with two forward slashes (i.e., `//`) and are ignored by the Arduino. You can feel free to exclude them although com-

```
1  // Global constants for pin assignments
2
3  int pin_led1 = 4;
4
5  // The setup routine runs once when you press reset
6
7  void setup() {
8    Serial.begin(9600);               // Setup the serial terminal
9    pinMode( pin_led1, OUTPUT );      // Configure pin_led1 as digital output
10 }
11
12 // Global variables
13
14 short input_a = 2;
15 short input_b = 2;
16 short sum     = 0;
17
18 // The loop routine runs over and over again
19
20 void loop() {
21
22   // Do the addition
23   sum = input_a + input_b;
24
25   // Print the result to the serial monitor
26   Serial.println( sum );
27
28   // Blink LED sum times
29   for ( int i = 0; i < sum; i++ ) {
30     digitalWrite( pin_led1, HIGH );  // Turn on the LED
31     delay(500);                      // Wait 0.5 seconds
32     digitalWrite( pin_led1, LOW );   // Turn off the LED
33     delay(500);                      // Wait 0.5 seconds
34   }
35
36   // Wait four seconds
37   delay(4000);
38 }
```

**Figure 8: Example Arduino Program**

ments are important part of effective software engineering. An Arduino program can be divided into four sections. The first section (Lines 1–3) is where we create global names for pin assignments. The second section (Lines 5–10) is the `setup` routine. The statements in the `setup` routine execute only once at the very beginning of the program. The third section (Lines 12–16) is where we define global variables. We can think of a variable as a named "box" where we can store values. So in this example we have three "boxes" named `input_a`, `input_b`, and `sum`. Variables also have types which indicate what kind of values can be stored in the box; and in this case the type of all three global variables is `short`. The type `int` means the variable can store integer values and will probably be the most common type you use in your programs. The fourth and final section (Lines 18–38) is the `loop` routine. The statements in the `loop` routine execute repeatedly over and over again. Throughout this lab, we will provide you the necessary code for the first two sections; you will be responsible for focusing on the `loop` routine and possibly adding global variables where necessary.

The statement on Line 3 is a variable assignment. It specifies that the variable named `pin_led1` should be assigned the value `4`. From then on, whenever we use the name `pin_led1` it will be interpreted as the value `4`. Effectively, what we are saying that LED1 on the prototyping board is wired up to digital input `4` on the Arduino board.

The statement on Line 9 is a call to a routine which tells the Arduino to setup the serial monitor on the host computer attached to the Arduino through a USB cable. We can use the serial monitor to view results sent back from the Arduino. The statement on Line 10 tells the Arduino that pin number `pin_led1` (i.e., pin number `4`) should be configured as a digital output pin.

Line 23 does the actual addition, and Line 26 uses a routine to send the sum back to the host computer. Lines 28–34 use a `for` loop to blink an LED `sum` times. The statement on Line 30 is a call to a routine which tells the Arduino microcontroller to write a logic low value to the pin number `pin_led1`. The statement on Line 31 is a call to a routine which tells the Arduino microcontroller to wait for the given number of milliseconds before continuing to the next statement.

Normally, a program interfaces with the hardware through the operating system; the operating system acts as a master referee which controls which program can access which hardware at which time. The Arduino microcontroller is so simple that it actually does not have a real operating system.

## 4. Instruction Set Architecture: Arduino Machine Instructions

Processors cannot directly execute the sequence of high-level statements which make up the program in the previous section. Instead, processors execute a sequence of very simple *machine instructions*. Figure 9 illustrates a very simple architecture with a processor to compute on data and two kinds of memory to store data: *main memory* is slow but can store a large amount of data, while *registers* are fast but can only store a small amount od data. This simple architecture supports three kinds of machine instructions: *load instructions* move values from memory into registers; *store instructions* move values from registers into memory; and *arithmetic instructions* perform simple arithmetic on values stored in registers.

We use a special piece of software called a *compiler* to transform the sequence of high-level statements in Figure 8 into a sequence of machine instructions the processor can understand. The Arduino programming framework provides a convenient graphical user interface to write and compile Arduino programs and then upload them to the Arduino-based mobile robotic platform. Figure 11 illustrates the Arduino programming framework's graphical user interface which includes a series of icons on the toolbar. You will primarily be using the first two icons. The *checkmark* icon is used to compile your program into machine instructions and the *right-arrow* icon is used to upload these machine instructions to the robot.

Figure 10 shows the actual machine instructions generated by the compiler for Line 21 in Figure 8. There are four machine instructions: the first two machine instructions load values from main memory into registers, the third machine instruction adds these two values together, and the final machine instruction stores the sum back out to main memory. The processor would likely use a ripple-carry adder similar to the one you developed in Lab 1 to implement the add machine instruction.

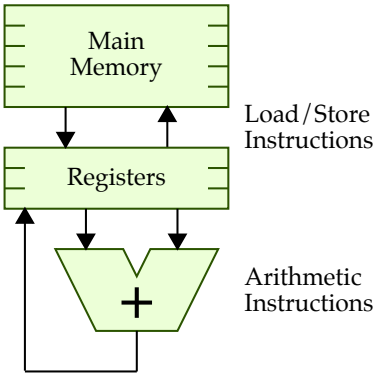


**Figure 9: Simple Architecture**

```
...
00000100 <loop>:
 100: push r28
 102: push r29

 # load two values from main memory into two registers
 104: lds  r24, 0x0103
 108: lds  r25, 0x0102

 # do the actual addition
 10c: add  r24, r25

 # store the sum from a register back into main memory
 10e: sts  0x0104, r24
...
```

**Figure 10: Machine Instructions for Line 21 in Figure 8**

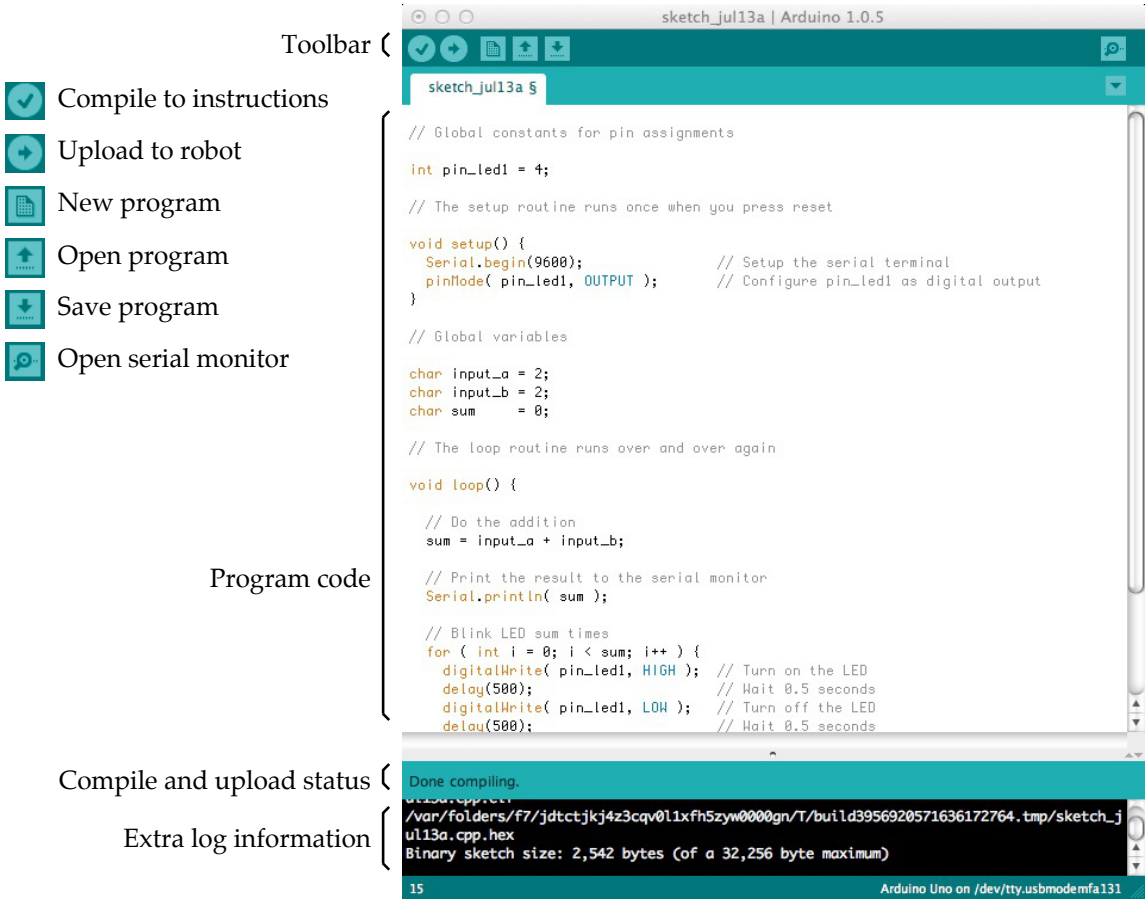


**Figure 11: Arduino Programming Framework**