# ASSIGNMENT 4: THE 68HC11

## Introduction

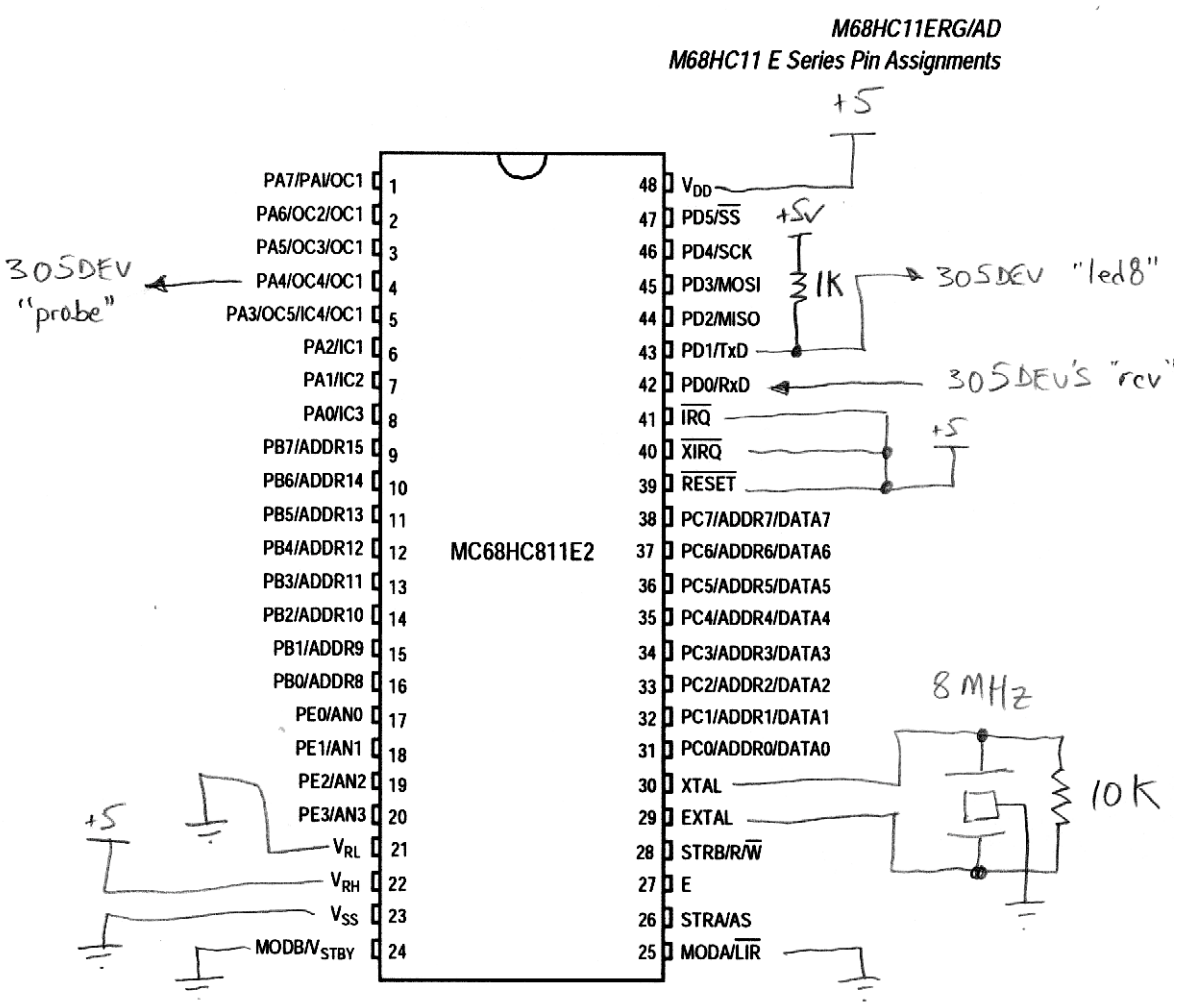
In this assignment, you will:

• construct a minimal 68HC11 system on your breadboard, and (2) use the serial port of a computer to load programs into the HC11's internal 512 bytes of RAM (making use of the serial bootstrap feature of the CPU).

• load a known-working program into the HC11 to verify that your wiring setup and serial line configuration is working.

• modify and write new programs for the HC11 and use your setup to load them into the HC11 for testing and debugging.

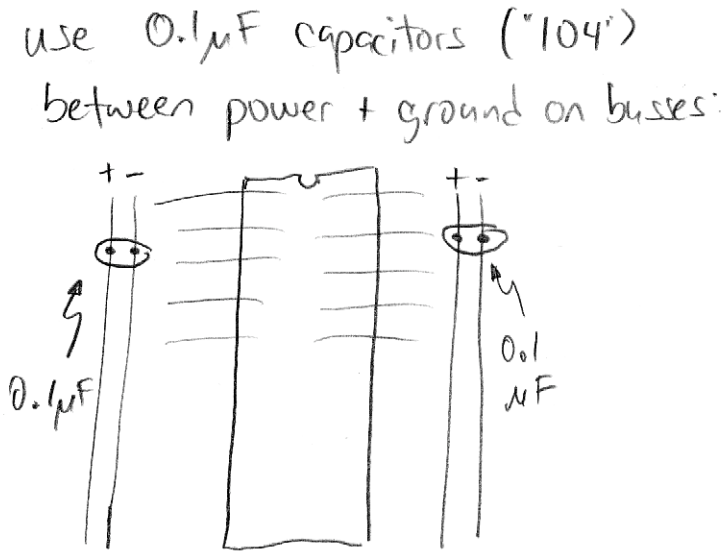
## Hardware Setup: HC11 Wiring



The previous drawing shows how the 68HC11 chip should be wired. Here are a few notes of interest:

- Power is applied via Vdd (pin 48); ground is applied at Vss (pin 23).
- The pins ~RESET, ~XIRQ, and ~IRQ must all be tied high.
- The analog input references should be connected: $V_{RL}$ to ground, and $V_{RH}$ to +5v.
- The MODA (pin 25) and MODB (pin 24) lines must be tied to ground. This put the HC11 into the serial bootstrap mode when it is powered on or reset.
- There is a 1K pullup resistor on the HC11's TxD output. This is necessary because when the HC11 starts up in bootstrap, the pin can only assert ground, and must be pulled high to produce a logic one ("wired OR mode").
- Please observe the following notes on the ceramic resonator:
    - The ceramic resonator has 3 pins, and is connected to the XTAL, EXTAL, and ground.
    - **Mount the resonator near to the XTAL and EXTAL lines, and keep all wires to the resonator as short as possible.**
    - The outer two pins of the resonator go to XTAL and EXTAL (either way—there's no polarity).
    - **Wire the center pin of the resonator directly to Vss (ground) of the microprocessor.** Then run a wire from the processor Vss to the ground bus.
    - You must add a 10K resistor in parallel with the outer two pins of the resonator to get the oscillation frequency to stabilize.

**Hardware Setup: Power Wiring**




When wiring up the power busses, make sure to insert one 0.1$\mu$F capacitor (they're marked "104") into each power-ground bus. The diagram above illustrates this. The purpose of these capacitors is to smooth out fluctuations in the power supply, making sure the HC11 operates properly.

## Software Setup

The procedure for loading a boot program into the 68HC11 is the following.

1. Turn on the HC11 with MODA and MODB held low. Your circuit (if wired properly) does this. The chip automatically powers on in serial boot mode.

2. Open a serial connection to the HC11 with parameters 1200 baud, no parity, 8 data bits, 1 stop bit (1200-N-8-1).

3. Write a 255 to the serial port. When received by the HC11, this is used to synchronize the baud rate.

4. Write 512 bytes of data to the serial port. These data are loaded into the HC11's internal RAM, and then execution automatically jumps to address 0.

Java routines are provided for perfoming the steps above. You should either (a) install a Java Development Kit (JDK) on your computer, or (b) use the JDK already installed on the computers in the OS 306 Engaging Computing Lab.

Instructions for installing the JDK or using the existing one are provided in a separate on-line document. Please see `http://www.cs.uml.edu/~fredm/courses/91.305/javasetup.shtml` for instructions.

## HC11Boot.java

At end of this handout is the initial test program `HC11Test.java`. This program is also available from the course web site.

All this program does is load a tiny, three instruction program into the HC11:

```
      ldaa 0x10
      staa 0x1000
loop: jmp loop
```

Here is what this HC11 program does. First, it loads the value 0x10 into register A. Then, it stores this value to address 0x1000. Address 0x1000 is a special address—it is the location of the PORTA data register. Data that are written to this address appear as ones and zero on the PORTA pins. The effect of writing 0x10 to this address is that bit 4 of PORTA gets set to a one. Note from the HC11 datasheet that its default (power-on) value is zero. Thus, when the program runs, Port A, bit 4 (which happens to be pin 4 of the HC11) will change from zero to a one. Hooray!

The third line of the program creates an infinite loop, jumping to itself endlessly. The CPU must be doing something; it never just stops. So if you want your program to terminate, you should keep it busy by having it loop in a known fashion. Otherwise it will just plow along, executing whatever code it happens to find in memory.

## Serial.java

The HC11Boot.java program uses a class that is defined in the file Serial.java. This provides the Serial object that implements rudimentary communication through a serial port.

## PROBLEM 4–1: BOOT YOUR 68HC11.

You don't have to write any code here, you just have to put all the pieces just described together and get it working for yourself.

Build up the HC11 circuit as described. Connect the HC11's Port A4 output (pin 4) to the dev board probe display.

Get Java running on a PC.

Compile the `HC11Test.java` and `Serial.java` files.

Plug the UML305DEV board into the serial port of the PC. Note whether you are using COM1 or COM2.

Power on your UML305DEV board, thereby powering up the HC11 in the serial bootstrap mode.

Run `HC11Test.java` (telling it whether to use COM1 or COM2). It will then communicate with the HC11 and install the little test program into the microprocessor.

Witness the Port A4 output (pin 4 of the HC11) go from zero to one, as evidenced by the probe indicator going from green to red.

Congratulations, your 68HC11 is alive!

If you got the PA4 line to turn red, you're ready to proceed. If not, you'll need to debug.

First make sure you don't have any wiring errors.

If this doesn't result in happiness, try to find either or both of (a) a known working HC11 setup, so you can test your development computer configuration, and (b) a known working computer setup, so you can test your board better.

If that fails, come to office hours/lab!

## PROBLEM 4–2: BEEPING.

The following small HC11 program, `beep.s`, creates an oscillation on the Port A4 pin:

```
;;; beep.s
;;; toggles Port A4 with delay loops between each toggle,
;;; creating an oscillation in the audible range.

        ldx #0x1000             ; point at register base
loop:   bset 0,x,#0x10          ; set bit 4 in PORTA register
declp1: deca                    ; decrement reg A
        bne declp1              ;   and loop till it hits zero.
        bclr 0,x,#0x10          ; clear bit 4 of PORTA
declp2: deca                    ; decrement reg A
        bne declp2              ;   till it hits zero.
        bra loop                ; now recycle at the beginning.
```

The program works by setting and clearing bit 4 in the Port A register, making use of the bit set and bit clear instructions (`bset` and `bclr`). In between each set and clear, there is a delay that results from decrementing the A register till it hits zero.

Using the `as6811` assembler (available from the Resources: Software area of the course web site), assemble the program. Type at the prompt:

```
as6811 -l beep.s
```

so that the assembler produces a listing file. From the listing file (named `beep.lst`), look at the resultant object code. For instance, the first code line of the file assembles to:

```
0000 CE 10 00             5         ldx #0x1000        ; point at register base
```

The first column of numbers, “0000,” is the location that this code assembled to. The second set of numbers “CE 10 00,” is the object code for this instruction. The next number, “5,” is the line number of this statement in the source file.

Examine the rest of the file so that you can find the object code.

Now, create your own copy of the `HC11Test.java` file. Rename it to `BeepTest.java` or similar.

Open the file in your favorite editor, and replace the Java class name `HC11Test` with your new filename (e.g., “`BeepTest`”). You should also replace the class name in the print statement in the `setup` method, though of course this doesn't affect functionality.

Now, replace the object code loaded into the `buf` array with the object code from your assembled `beep.lst` file. The first few lines should now look like:

```
        buf[i++] = (byte)0xce;
        buf[i++] = (byte)0x10;
        buf[i++] = (byte)0x00;
```

Continue in this fashion, copying the whole of the HC11 beep program into the Java source file.

Now, compile the Java file, turn on your HC11 board, and run the Java program to download the HC11 code.

When the Java program has finished, the code should be running on your HC11. If you plug the output from the Port A 4 line (pin 4 of the HC11) into the 305DEV probe line, you should see both the red and green LEDs on at the same time (actually, they're flashing back and forth, but too fast for you to see).

Now, plug the PA4 line into the piezo input. You should hear a tone!

**Problem 4–2a.** Your HC11 has an 8 MHz oscillator. This is divided by 4 to result in a 2 MHz instruction clock (the E clock). In the 68HC11 Reference Manual there is a mapping from instructions to cycles; for example, the LDX #0x1000 instruction takes 3 cycles, or 1.5 microseconds, to execute.

What is the frequency of the tone that the program generates? **Your answer should be a frequency in cycles per second (Hz), along with a justification for how you got it.**

**Problem 4–2b.** Re-write the program to generate precisely 1000 Hz. Assemble it, copy the assembled object code into a new instance of the loader program, and run it to make sure it works. Hint: employ instructions that don't do any useful work except take up time. **Turn in a printout of your working .lst file, plus a copy of the .java file you used to download it to the HC11.**

**Problem 4–2c.** Which tone is louder—the original tone, or the 1000 Hz tone? Note: this is somewhat subjective, but take a stab at it. **Turn in the answer by frequency, e.g., "xxxx Hz seems louder."**

## BOOTLOAD

In Problem 4-2, you hand-copied the HC11 program you were interested in running into an instance of the "HC11Boot" source file. This is painstaking and annoying (though it does serve the function of making you deal with the specific opcode bytes of your HC11 program).

For the rest of the assignment, we will use a different process for getting your code into the HC11:

1. Write your HC11 code in to a source file with a ".s" suffix.
2. Run the as6811 assembler on your .s file, using the -ol flag. This will produce a "relocatable" file, with a .rel suffix, that contains the object code bytes to be downloaded.
3. Use the new BootLoad.java program to download the .rel file to your HC11.

Following is a walk-through.

Start off by downloading BootLoad.java and ReadRel.java from the course website.
Compile these files with Serial.java:

```
javac BootLoad.java ReadRel.java Serial.java
```

Next, assemble an HC11 source file with the flag to produce the .rel file. For instance, assemble the beep.s program from earlier:

```
as6811 -ol beep.s
```

This should produce a beep.rel program, with contents like this:

```
XH2
H 1 areas 1 global symbols
S .__.ABS. Def0000
A _CODE size 11 flags 0
T 00 00 CE 10 00 1C 00 10 4A 26 FD 1D 00 10 4A
R 00 00 00 00
T 00 0D 26 FD 20 F2
R 00 00 00 00
```

The lines starting with "T" contain the object code. The first two bytes in these lines are the address where the code goes; the remainder of the bytes on these lines is the object code itself.

Now we are ready to download the beep code into the HC11. Do with with the new BootLoad:

```
java BootLoad beep COM1
```

The program will then download the code from beep.rel into the HC11, producing the following output:

```
Using serial port COM1
Read 17 bytes of code from file beep.rel
Serial port is open.  Turn on HC11 and press <ENTER> to continue...
Writing to serial line...
done.
```

OK. Now we're ready to more productively code on the HC11.

## PROBLEM 4–3: INTRO TO SERIAL COMMUNICATIONS

In this exercise, we will learn how to transmit ASCII data over the serial line from the HC11, and receive and display it on the PC.

The following program `serialxmit.s` (available on the course website) will cause the HC11 to repeatedly transmit the ASCII character set beginning with code 0x21 (the exclamation point, '!') and ending with code 0x5a (the capital letter 'Z'):

```
;;; serialxmit.s
;;; counts from 0x21 to 0x5a on 68HC11 serial port and then repeats

base=   0x1000
scsr=   0x102e                  ; serial comms status reg
scdr=   0x102f                  ; serial comms data reg

        ldx #base               ; pointer to register base
loop:   ldaa #0x21              ; start with ASCII 0x21 to transmit
xmit:   staa <scdr,x            ; transmit reg a
        brclr <scsr,x,#0x40,.   ; loop here until byte transmitted
        inca                    ; inc char to be transmitted
        cmpa #0x5B              ; see if it's beyond ASCII 0x5a
        bne xmit                ; no, send the next one
        bra loop                ; reset char to 0x21
```

Let's look at the program in some detail.

The program begins by installing the value 0x1000 into the X register (the label `base` was previously assigned as 0x1000). This allows indexed-by-X instructions to be used to talk to the serial control registers.

Next, the value 0x21 is loaded into the A register. This will be the first byte that is transmitted.

At the next line, beginning with the label `xmit`, the value in the A register is written to the `scdr` (serial communications data register). This will cause the value in A to be written out the serial port. Note that the indexed-by-X addressing mode is used. This requires a one-byte value which is added to the value in X to form the extended address. The less-than-symbol < is an operator in the assembler which takes the low byte of a label's value. Hence, the low byte of the `scdr`, or 0x2f, is taken and added to the 0x1000 in X to point to the `scdr` register.

To avoid overrunning the register, it is necessary to check a "busy" bit in the `scsr` (the serial communications status register). The next instruction does this. This is the "branch-if-bit-clear" instruction, which uses an indexed-by-X addressing mode. The `<scsr` when added to X points to the status register. The `#0x40` operand indicates we are interested in testing the 2nd highest bit in this register (which, if you refer to the HC11 documention, is the Transmit Complete flag, with a 1 indicating it is complete). Finally, the dot at the end of the instruction is the address to which to jump. In the assembler syntax, the dot has a special meaning, which is: the address of the beginning of this instruction. Thus, the `brclr` will literally jump to itself if it bit is clear, which means that the serial transmit is still underway. Hence, we will be stuck here in this instruction until the transmit is done.

The next instructions increment the A register and loop in the appropriate way. After being incremented, the A register is compared with the value 0x5b (one higher than the last value we want transmitted). The compare value performs a subtraction (A – 0x5b, in this case) and then throws the result away. But the compare does set the condition code bits, and next we test if the result was "not equal to zero," using the `bne` instruction. If it wasn't zero, we want to transmit this next value of A, so we jump to the `xmit` label. If it *was* zero, then A must have been 0x5b, and we don't want to transmit that, so we jump to the `loop` label, which resets A to 0x21 before transmitting again.

The `serialxmit.s` program is available on the course website. Assemble it to produce the `.rel` object file, and then download it to the HC11 using `BootLoad`.

Now, you need to run a terminal emulator on your PC in order to see what the code is doing. You may either (1) use a standard terminal emulator on your PC (e.g., HyperTerminal), with serial settings 1200-N-8-1 and no flow control, or (2) compile and run `BootTerm.java` (available on the course website). This is a version of the `BootLoad` program that, after booting the HC11, runs a tiny terminal program with the correct settings.

Regardless of the terminal emulator method, when you have it all working, you should see the following on your screen:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-./0123456789:
;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRST
UVWXYZ!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-./01234
56789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN
OPQRSTUVWXYZ!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-.
/0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

etc.

Does it work? Make sure you have the HC11's serial transmit line (pin 43 of the chip) plugged into the "xmt" port of your dev board, not LED8!

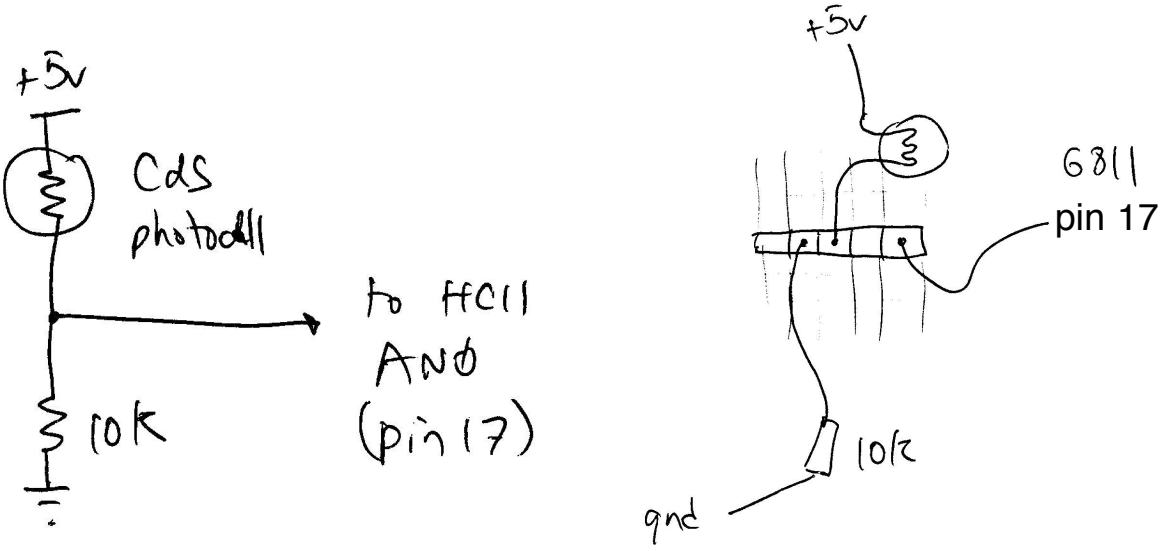
There is nothing to turn in for this exercise, but you can't really go on to the next problem until you have this working.

## PROBLEM 4–4: SERIAL LINE BARGRAPH LIGHT SENSOR

In this exercise, you will construct a voltage divider using a light-sensitive photocell, and wire its output into one of the HC11 analog-to-digital (A/D) inputs. Using the techniques presented in `serialxmit.s`, you will write a program that repeatedly displays the converted analog reading as a bargraph of 0 to 15 asterisks.

### How To

The photocell is wired in a "voltage divider" circuit as shown. The diagram on the left is the electrical schematic; on the right is a visual wiring guide.




The voltage divider generates an output voltage that is a function of the ratio of the two resistances in the legs of the converter. In this instance, the photocell is in the upper leg. Its resistance decreases with increasing light, causing a higher voltage result in this case.

### Sample Code

The file `analogdemo.s` demonstrates how to initialize the HC11's analog-to-digital converter and make a conversion. The core functionality in the following code snippet:

```
        bset <OPTION,X,#0x80    ; set high bit of option -> turn on A/Ds

loop:   ldaa #0                 ; select channel 0
        staa ADCTL              ; start conversion
donelp: ldaa ADCTL
        bpl donelp              ; if high bit set, it's done
        ldaa ADR1               ; grab result!
```

The full sample program repeatedly makes analog-to-digital conversions and prints to the serial line an ASCII representation of their hex value.

### How to Go About It

To verify that your hardware is functioning properly, it is suggested that you first build the circuit and run the supplied `analogdemo.s` demo program. The demo program will repeatedly make analog

readings and transmit their value, represented as a single ASCII character, out the serial line.

### Features

Your solution must:

- demonstrate the use of subroutines. Make sure to initialize the stack pointer at the beginning of your code.
- repeatedly print the bar graph of 0 – 15 asterisks based on the high nybble of the converted value, with the linefeed and carriage return characters between each bar. You should see a sideways scrolling bar graph on your terminal emulator when it is working.

For extra credit, you can generate a graph of 0 – 31 asterisks using the upper five bits of the converted value.

### What to Turn In

Hand in your assembled .lst file and a screen capture of your program in action.

### If You Don't Have a Photocell

If your kit does not contain a photocell, you can use a potentiometer (also known as a volume knob, available in lab). This is a three terminal resistor with a center tap. Wire the one of the outer terminals to +5v, the other outer terminal to ground, and the variable center tap terminal to the HC11's analog input.

## PROBLEM 4–5: LIGHT-CONTROLLED OSCILLATION

### ***FOR HONORS/GRAD STUDENTS OR FOR EXTRA CREDIT***

Using the techniques shown in this assignment and the previous one, create a program that will oscillate, with the frequency of oscillation continuously varying based on the value of the converted photocell reading.

Scale the frequency of oscillation so that it varies over the audible range of about 100 Hz to about 5 kHz as the photocell value varies between 0 and 255. (You may choose as to whether small readings correspond to low frequencies or high frequencies.)

### What to Turn In

Turn in your program, and an analysis that shows the frequency that corresponds to analog readings of 0, 128, and 255.

## HC11Boot.java

```
// HC11Boot.java: bootstrap program for loading code into HC11
// Fred Martin / UML CS / last modified Mon Sep 29 08:35:40 2003
//
// put the bytes you want installed into HC11 internal RAM
// into "buf" object beginning at buf[0].

class HC11Boot {
     private static Serial s;

     public static void main(String[] argv) {
          setup(argv);     // select serial port and init Serial object
           System.out.println("Using serial port " + s.getPortName());

          // make 512 byte buffer and fill with zeroes
          byte[] buf= new byte[512];
         int i;
          for (i=0; i<512; i++) buf[i] = 0;

          // this 3-instruction program will turn on bit 4 of PORTA (PA4)
          // and then loop endlessly
         i=0;
          buf[i++] = (byte)0x86;   // RAM loc 0 -- LDAA with 0x10
         buf[i++] = (byte)0x10;  //          1          0x10 (bit 4 on)
          buf[i++] = (byte)0xb7;   //          2 -- STAA extended to 0x1000
         buf[i++] = (byte)0x10;  //          3          0x10
         buf[i++] = (byte)0x00;  //          4          0x00  (the PORTA reg)
          buf[i++] = (byte)0x7e;   //          5 -- JMP extended to 0x0005
         buf[i++] = (byte)0x00;  //          6          0x00
         buf[i++] = (byte)0x05;  //          7          0x05 (loop the JMP!)

           System.out.println("Writing to serial line...");
         try {
              s.put(0xff);          // send baud rate detect byte
              s.put(buf);           // dump 512 bytes to serial port
          } catch (Exception e) {
              e.printStackTrace();
         }
           s.flush(); // make sure all chars get out before exiting
           System.out.println("done.");
    }

     private static void setup(String[] args) {
          if (args.length == 0) {
                System.out.println("Usage: HC11Boot serial-port");
             s = new Serial();
              String[] ports=s.portsAvailable();
              int nPorts= ports.length;
               System.out.println("Available ports are:");
              for (int i=0; i< nPorts; i++) {
                   System.out.println("   " + ports[i] + " ");
             }
               System.out.println();
              System.exit(0);
         }
          s = new Serial(args[0]);
    }
}
```