Microcontroller Workshop
Prerequisites

- A working knowledge of the Arduino Environment, and it’s C/C++ Programming Language
- A working knowledge of basic electronics and circuit assembly (breadboarding).

Course Materials

- A Computer
- Your previous summer camp work
This is a whole lot of stuff to cover, but please, don't hesitate to stop and ask question. I don't know your level of expertise, and where I'm leaving out information that you would like, or need to know. I also don't know where I'm delivering overly redundant information, so please, give me feedback while I'm up here.
Files and Resources

• All files and resources are already available online!
• Navigate to the Wiki (http://wiki.lvl1.org), and click on “Workshops” in the left-hand menu
• These resources will change, as some of them aren’t as good as they could be!
Most of this circuitry is unneeded a lot of the time. Most of the discrete components (resistors, capacitors, etc) are a result of high quality power management built into the arduino: You can plug in any power source you want, and the board will figure out which one to use.
Why Arduino?

- Easy USB interface for programming Microcontroller
  - Bootloader
  - FTDI Chip
- Relabeled pins and expandable form factor
- Easy to use IDE and software library

This easy USB interface eliminates the need for an expensive programmer to burn code onto the microcontroller.

No need to know anything about power sources and power management, easy to stack pre-designed shields on top.
Flexibility:
The ability to design your own software or firmware without interference from the Arduino IDE or Software Libraries.
The ability to choose your editor, choose how your source is compiled and linked together, and the ability to more easily add and extend your software libraries.
The ability to branch into more advanced programming and debugging.

Thirst for Knowledge:
Everything we’re doing today can be done on the Arduino. You’re taking this next step because you want to learn more about how Microcontroller systems work, because you want to build your own projects which use these bare chips in interesting ways, because you don’t want to forever rely on pre-written software libraries and components which have been tested and used in the same boring ways a hundred thousand times. Because expanding your knowledge beyond arduino sets you up for learning how these systems work in general, freeing you from any platform dependence.

Why Un-Arduino?

• Cost
  – Arduino Clones: $15-$50
  – AtMega 328P in Single Quantities: $3
  – AtTiny45V in Single Quantities: $2

• Size
• Flexibility
• The all-consuming, zombie-like thirst for knowledge
Power Regulator:
This isn’t an ideal solution, but it works for the purposes of this workshop. A power regulator can be bodged together with a 9 volt and another $5 in components.

We don’t’ have a reset switch: No particular reason, I just don’t anticipate the need for this sort of functionality in this workshop. It’s easy enough to disconnect and reconnect power, should the need arise.

What are we replacing the Training Wheels with?

- **Bootloader -> Flashing Raw Code**
  - Using the AVR’s ISP functionality, we’ll burn code right on the chip
- **I/O -> Datasheet Comprehension**
  - We don’t need no stinking labeled pins.
- **Clock Source -> No Clock Source**
  - The AVR has an 8MHz Oscillator on-board.
  - AtTiny45 has an internal PLL
These differences aren’t appreciable!

- With under $10 in components, you can throw together an Arduino-compatible breadboard circuit, and use the Arduino IDE.
- The Arduino is popular because it packages everything together, along with an easy-to-use bootloader, SDK, and form factor.
In the Beginning...
Microcontroller Principals
Flash:
Like hard disk storage— Nonvolatile, and kind of slow. Your program gets stored here.

RAM:
Like, well, RAM— Volatile, fast as hell. Your variables get stored here.

EEPROM:
A harder hard disk— Really nonvolatile, but slow. Easier to access at runtime.

Memory-Mapped Peripherals:
On most microcontrollers (and most computers in general), peripherals like serial ports, ADCs, etc. are accessed on the RAM bus. You write to memory locations in order to change functionality.
As a result of the Harvard architecture, in order to store and read data from Flash, you need to execute specific instructions. Since flash is fairly slow, this isn’t a very fast way of going about things. It can offload huge amounts of static data, however. Unless you’re specifically using an instruction to access flash program memory, you’re using RAM.

The AVR Architecture

- We are using the AtTiny line of chips
  - One of many AVR Processory Families
  - Tinys designed for single use, simple installations
  - Megas design for use in larger, more complex systems
  - Code is portable within subfamilies.
  - Code is mostly portable within families.
  - Code is somewhat portable between families.
- AtMega use a “Harvard Architecture” core
  - Flash and RAM Memory busses are accessed separately.
  - I/O and Peripherals are accessed by writing data to RAM addresses.
Peripherals Available

- 4kb of Flash Memory
- 256b of EEPROM
- 256b of RAM
- 4 PWM outputs
- 4 ADC inputs
- 1 USI (TWI, USART, I2C, etc)
This isn’t quite like the AVR model, but close.
Datasheet Comprehension

Available:
http://www.atmel.com/dyn/resources/prod_documents/doc
2586.pdf
Critical Datasheet Pages

- Page 2, Pinout

Minimum Items we need to get this thing working correctly: VCC and GND.
For connecting a choosing resistors for the LED, we do not want to exceed reasonable ratings. From the curve on the left, we see that we can sink roughly 20 ma of current, and still have a pretty good “low” level output. From the table on the right, we see that we cannot exceed 40 ma of current on an IO pin before blowing up our chip.
AVR-GCC compiles your C or C++ code into actual op-codes which are useful to the AVR. Each file in your project gets compiled into a separate output file. AVR-LD comes along after the fact, and analyzes the collection of output files (according to your make script), and links them together, along with any included libraries, into a single output file. This output file is a 1:1 image of the code you will be putting on your AVR. This output file includes the flash memory space (.text), initialized variables (.data), and uninitialized variables (.bss).

AVRdude is the AVR downloader/uploader. It is able to read and write the memories of most AVR chips, with most of the programmers which exist today. We use this program to update code, modify AVR “fuses,” as well as program the EEPROM.

AVR-LibC is a set of Libraries which implements the C Standard Library for the AVR family of microcontrollers. These libraries also provide the headers required to correctly interface with the memory mapped peripherals on the AVR series of chips, as well as a few additionally useful functions and macros.

Your favorite text editor will be used to program C files for the AVR, as well as make files. A nice, simple editor for windows is “Programmer’s Notepad,” and for Linux, you’ve got a standard range of evangelical choices. Nano, Vim, Emacs, etc.
Test the Programmer!

- Without plugging the programmer into your breadboard, test it on the computer with the following command:
  
  `Avrdude --port /dev/ttyUSB0 --reset --wp 15 --configfile /path/to/configfile`

- You should get [Insert Valid Response here], indicating that your programmer is properly soldered and AVRDude is properly configured.

-P: Port
-p: Part
-c: Programmer
Test the Toolchain!

• Now we’re going to write our first program for the AVR, and compile it.

```c
#include <avr/io.h>
#include <util/delay.h>

#define F_CPU 16500000UL //16.5 MHz

int main(void)
{
    DDRB = 0b00000010; //Port B Pin 1 is an output.
    while(1)
    {
        PORTB = 0b00000010; //Turn LED off.
        _delay_ms(500);
        PORTB = 0b00000000; //Turn LED on.
        _delay_ms(500);
    }
}
```

Parts of the program:
#include <avr/io.h>-- The pound sign tells the compiler that this is something that should be handled by the C pre-processor, a parser which handles inline replacements, and other duties. This tells the C preprocessor to replace the include line with the contents of avr/io.h. The brackets indicate that this file is located in the standard compiler include path. Avr/io.h, in this case, contains all the memory location definitions, ram location definitions, and other common, shared avr functions.

Delay.h includes some simple delay loops, delay_us and delay_ms, delaying for a millisecond and a microsecond, respectively. Delay_ms can generate a delay of up to 6 seconds. These delays require that F_CPU be defined in order to determine CPU speed.

DDRB– Data Direction Register for Port B. This sets pins as inputs or outputs. 1=Output, 0= input. In order to avoid potential electrical problems, all ports which are not being used, or are connected to unknown or unused inputs should be configured as inputs.

While(1)-- In embedded programming, the main function of a C file MUST NOT RETURN, since programs on Microcontrollers have nothing to return to. On a computer, C programs return to the operating system (or the program which
requested the execution of the C program), but on a microcontroller, these few lines of code ARE the operating system. AVR-GCC and AVR-LD do a good job of detecting main functions which return these days, and add a while(1) loop at the end in most cases, but in some cases, this can result in undefined behavior which requires a reset at best, and destroys your hardware at worst.

PORTB—This sets the value of port be. The 0b indicates that the byte will be in binary. 0x is hex. Port B is an 8 bit register (All registers on the AVR are 8 bits. Registers with more than 8 bits will, instead, be divided into high and low registers.)

_Delay_ms
A makefile is the glue that tells the compiler, linker, and other programs how to turn the C code of your program into something which can be uploaded to the AVR. With one sourcefile, the makefile need not be complicated, but for multiple sourcefiles and includes, the makefile can be very complicated. Make script sourcery is beyond the scope of this class. I’ll give you a makefile and explain the basic parts.
Test the Toolchain!

- Navigate to the directory you’ve been storing all this stuff in, using a command prompt or terminal.
- Type ‘make’
- Something like this should happen:
The .hex file is the actual program that will get uploaded to your board. This is a 1:1 image of the code to be burned onto the chip. If you look, the .lss file is an assembly file, with your C code interleaved, so you’re able to see what assembly instructions your code gets turned into.

Test the Toolchain

- Now look at all the files in your program directory!
- Whose didn’t work?
Wait a second...

- I tricked you.
- From the Datasheet:

**8.2.1 Default Clock Source**

The device is shipped with internal RC oscillator at 8.0MHz and with the fuse CKDIV8 programmed, resulting in 1.0MHz system clock. The startup time is set to maximum and time-out period enabled. (CKSEL = '0010', SUT = '10', CKDIV8 = '0'). The default setting ensures that all users can make their desired clock source setting using any available programming interface.
Decoded: Access the hfuse in read mode, output to the console ‘-’ in hex format. Note, this is a capital -U
Access the Low Fuse in write mode, change it to 0xE2 (hex). Now we change the mode to ‘w’ for write, type the code in hex format, and declare the input mode as ‘m’ for ‘Immediate,’ meaning that we’ve typed in the value we want.
C++: A bit of a lie. There are lots of good reasons to program a microcontroller in C++, and if you’re good at it, and know which parts of C play well with microcontrollers, and which don’t, there’s no reason not to use C++, and anyone that tells you otherwise is a big fat liar. I know C well, and everyone that’s writing libraries for AVR stuff programs them in C (in general).

Program starts at main() and goes until the end! Everything else is all about how you’ve compiled and linked your program together. Main cannot accept args.

In AVR C, access to functions, peripherals, and libraries is not performed through structs. Although the C headers for the Xmega are changing this, this is not true of any of the atmega devices.
How Your Code is Compiled

- First, the C preprocessor scans the code for preprocessor directives, and performs the desired operations.
- Next, the compiler compiles your code into valid AVR Assembly and Hex code.
- Finally, the linker links your written code files together with each other, along with any library functions you’ve made use of.
AVR LibC

- AVR LibC implements the standard C library.
- This library includes:
  - Standard Object Types
  - Common Mathematical Functions
  - Standard IO Manipulation
  - Dynamic Memory Allocation
  - String Manipulation
- Also includes a number of AVR Exclusive functionality:
  - Bootloader Support
  - Interrupt Mappings
  - Fuse Settings
  - Program Space Manipulation
  - Power Management
  - Handy Macros
Some Exceptions: Look them up and talk about them.

Standard Object Types and Integer Sizes

- The AtMega chips are 8 bit Microcontrollers. This means that all data is manipulated and calculated in 8 bit increments.
- Operations on 8 bit data is faster.
- Use the minimum number of bits for a given variable.
  - Default type is int. This is 16 bits.
  - For most objects, uint8_t is fine. Unsigned integer, 8 bits.
  - Floats are BAD, take up a lot of space, and are difficult to perform math on.
In order to print and do math on floats and doubles, you have to configure some things in your makefile. Specifically, you need to uncomment the linker flags which enable the math and printf_float libraries, then do a make clean/make. As an aside, make clean is the common make command to clean up all the thus-far made files.

Some Types

- Int8_t (-127-128)
- Uint8_t (0-255)
- Int16_t (-32767-32768)
- Uint16_t (0-65535)
- Int32_t (-2147483647-2147583658)
- Uint32_t (0-4294967295)
- Int64_t (-9.22337204 × 10^{18}, 9.22337204 × 10^{18})
- uint64_t (0-1.844x10^{19})
- Float (32 bits)
- Double (64 bits)
Dynamic Memory Allocation

• Avoid it! AVR implementations ofMallocare slow, and performing dynamic memory allocation can cause big problems!
  – Corrupting other data in the system
  – Causing the program to jump to a random location in memory
  – Causing undefined behavior which could destroy the system
Program Space Manipulation

- You can store read-only data in Program Memory. This reduces RAM consumption, and is a great idea for static strings, and large amounts of unchanging data.
- The <avr/prgmspace.h> header provides lots of easy macros for accessing this space.
- Take more time to access from Flash than from RAM.
Handy Macros

• You’ll often find yourself wishing to change only one bit of a data register.

• In order to set a bit, use “register |= _BV(location)”.
  – Equivalent to register |= 1 << location
  – Or-Equal replaces only the bit you want set, leaving the others undisturbed.

• In order to clear a bit, use “register &= ~(~_BV(location))”.
  – Equivalent to register &= 0 << location
Handy Macros

- Similarly, you’ll often want your program to wait for a specified external condition.
- `loop_until_bit_is_set(register, location)`
- `loop_until_bit_is_clear(register, location)`
The AVR LibC Reference

• More Info Here:
  http://www.nongnu.org/avr-libc/
If. This is in order to debounce the circuit. The nested while structure ensures that no action will take place until the button is released.

Masking: Often, we’ll only be interested in one or two bits of a given register. In order to remove spurious data, and make compare operations easier, we mask off the bits we’re interested in. In this case, we’re only interested in the last bit of the PINB register. In order to mask this bit off, we bitwise AND the register with a bitmask. This sets all bits except the first one to zero. If the first bit is zero, it remains zero, and if the first bit is 1, it remains 1.
Interrupts free you from the necessity of polling for a switch press in any given period. While polling for something to happen, you must poll fast enough that the external event is actually caught if it occurs. Interrupts allow you to discover these external triggers non-synchronously, with reference to the program.

In AVR LibC, interrupt service routines are the bits of code which are operated upon when an interrupt is triggered. The way interrupts work: When an interrupt is triggered, the CPU finishes the operation it’s in the middle of, then saves all the registers to SRAM, and disables interrupts. Then, the code in the interrupt service routine is executed. Upon exiting, the registers are loaded back from SRAM, interrupts are re-enabled, and execution begins where it left off. As a result, it’s a good idea to keep your service routines short, so that another interrupt isn’t triggered and forgotten while you’re servicing the existing one. This service routing is very short: It simply toggles when the service routine is entered as a result of a switch release.

This particular interrupt is the pin change interrupt. Each general IO pin on the AVR can generate a pin-change interrupt. This interrupt is triggered any time the pin state changes, from high to low, or low to high. These interrupts are banked, however. First, each pin change interrupt must be enable in its own register, then, the bank of pin change interrupts must be enabled. The service routine operates on the bank of interrupts. For example, if Pin Change Interrupts 0-7 are enables, then service routine 0 is called whenever ANY of pins 0-7 are changed. It is up to the service routing to inspect which pin has actually changed.

The volatile keyword is VERY important. It tells the compiler that the variable toggleStatus might change at any time, asynchronous of the program. If the compiler notices that there isn’t code that might change this variable, it will optimize reads to the variable. This is a very important keyword in dealing with interconnected systems that interact with the real world, as in microcontrollers. As a debugging hint, if you find that certain variables aren’t reading as modified when they should be, the problem probably relates to a missing volatile keyword. If inserting a long delay in the middle of a bit of code fixes the problem, it’s definitely related to the volatile keyword.

```c
#include <avr/io.h>
#include <avr/interrupt.h>

volatile uint8_t toggleStatus = 0;
ISR(PCINT0_vect)
{
    // Only do something if we're changing from a 0 to a 1 (releasing the switch)
    if((PINB & 0x01) == 1)
    {
        toggleStatus = !(uint8_t)toggleStatus;
    }
}

int main(void)
{
    // 0 is input, 1 is output.
    DDRB = 0b00000001;
    PCICR = 0b00000000; //Enable Pin Change Interrupts 0-7
    PCMSK0 = 0b00000001; //Enable Pin Change Interrupt 0
    sei(); //Enable Interrupts

    while(1)
    {
        if(toggleStatus == 0)
        {
            PORTB ^= _BV(0);
        }
        else
        {
            PORTB ^= _BV(1);
        }
    }
```
Printf

- Primary source of debugging for the enterprising hacker
- Other debuggers do a better and more verbose job
  - AVR Dragon paired the GDB
- This is easy and cheap
By default, Printf only performs inline replacements with integer types. In order to get printf working with floating point types, you need to include math and floating point printf libraries. This can easily be done with the Makefile I’ve given you. The ADC example includes this replacement already done.
We need to define F_CPU again this time, for the pre-processor UBRR Calculation (although in this case I’ve just put a value in).
We also need to define a baud. Baud rate is related to the clock rate of the CPU. Unfortunately, an 8mhz CPU is not a perfect match for any baud rate. As a result, we’ll need to look at the tables on wormfood.net in order to figure out which baud rates give us acceptable error rates. For 8mhz, 9600 baud works well, with an error rate of only .2 percent.
The UBRR calculation can be made with the preprocessor, using the given equation, or simply looked up from that big table. In this case, I’ve just looked it up. The UBRR is the only input into the USART baud rate generator.

This is the first time we’ve seen function prototypes. In most standards of C, functions must be declared before they are used. This is that declaration.

The Static file is the configuration for printf. The FDEV_SETUP_STREAM does all the hard work of configuring a file stream on an 8 bit microcontroller for us. We give it the name of our output function, the name of the input function (in this case, none), and a mode in which this file will operate. Now, any bits thrown at this file will pass through the uart_putchar function.

In the main loop, we configure our data registers. This time, PORTD, pin 2 must be configured as an output, for serial transmission. Furthermore, the UBRR must be loaded. Since the UBRR is more than 8 bits, it is split into high and low portions. The high portion must be shifted down 8 bits, and then loaded into the register, while the low portion can be loaded directly. The upper bits will simply be truncated. Finally, we enable receiving and transmission by putting 1s in the appropriate register.

Finally, we set stdout as a pointer to our file stream which we configured earlier. Stdout is a file included with stdio.h, which acts as the standard output file. When printf is used without declaring an output file, it defaults to stdout.

Our putchar routine simply places a valid character in the UDRO register. This register is the single, generic holding register for all data coming into and out of the uart. The AVR takes care of actually inserting this byte on the wire. The first if statement replaces any newline character with a windows-compatible equivalent. The loop_until statement waits for the CPU to confirm that it is ready to receive more data.

Getchar accomplishes approximately the same thing. If a character has not been placed in the UDRO register as a result of a write operation, it is returned.
How to Debug using printf

• Is my code reaching a certain point?
  – Printf(“I’m here!\n”);
• Is a variable being set to a particular value?
  – Printf("Variable1: %X Variable2: %X\n", variable1, variable2);
• Want to stop a program until you do something?
  – While(uart_getchar != ‘n’) {}
Disadvantages to Printf Debugging

• You’ll inevitably miss a debug statement somewhere
• Printing takes a LONG TIME, and can slow your code down considerably.
  – Can’t be used in the middle of really time sensitive code
• Print lines take a LOT of Ram. This can be ameliorated by storing your debug strings in program memory.
Meat and Potatoes

- We are going to be using a third party library, V-USB to program our Caps Lockers
- This third party library implements all the nasty, computer-science USB stuff for us, including signaling, interrupts, endpoints, and so on.
Meat and Potatoes

```c
for (;;) {
    /* main event loop */
    wd_setEvent();
    usBFull();

    /* A USB keypress cycle is defined as a scancode being present in a report, and
    then absent from a later report. To press and release the Caps Lock key, instead of
    holding it down, we need to send the report with the Caps Lock scancode and
    then an empty report. */
    if (usbIsInterruptReady() && reportCount < 2) /* we can send another key */
        buildReport();
    usbfSendInterrupt(reportBuffer, sizeof(reportBuffer));
}

timerFull();
```
Meat and Potatoes

```c
static void buildReport(void){
    uchar key = 0;
    if(reportCount == 0){
        key = 0x39;
    }
    reportCount++;
    reportBuffer[0] = 0;
    /* no modifiers */
    reportBuffer[1] = key;
}
```
Congratulations!

You’re now an AVR Programmer
Advanced Topics
Advanced Topics

• The How, Why, and Where of the Bootloader
• I2C Interfacing
• Using the ADC
• Internet Connectivity
• Motors
• Comparison of AVR Chips
This doesn’t refer to the actual ability to read Flash memory: No matter what, code within the boot loader cannot read flash memory in the program second, only erase or write. This refers to the ability to read while operating in the NRRW section. While writing or erasing to the RWW section, code can be read from the NRRW section. While writing or erasing to the NRRW section, the CPU is frozen until the write or erase operation completes.

The Bootloader

- AVR Flash Memory is divided into two sections, Read-While-Write (RWW), and Non-Read-While-Write (NRRW)
- Bootloader MUST reside in the NRRW section
- Bootloader can be 256b, 512b, 1k or 2k
- Bootloader can write and erase any location in Flash, including itself
- Bootloader has access to all faculties of the AVR Chip

Th
The Bootloader

- The bootloader code can be started from an application, or on reset.
- Bootloader is responsible for jumping into application memory or initiating reset.
- Application Code cannot modify flash memory, but it may start the bootloader.
  - Clever placement of variables in memory can result in the ability to pass variables into the bootloader code.
The Arduino Bootloader

• This bootloader emulates an STK500 serial Programmer.
• The bootloader is initialized when the chip comes out of reset.
• If the first character received over the serial port is a special character, the bootloader begins. It simply writes the data received (in hex format) to memory, starting at flash address 0x0000.
• If no character is received, or a non-special character is received, the bootloader resets into Application Flash
How to Write Your Own Bootloader

• The bootloader is located in the last 256b, 512b, 1k, or 2k of flash memory.
  – It’s up to you to keep track of this! If you mess up, you’ll put bootloader code in Application flash, and get really wacked up behavior.
• Use an __attribute__ tag in C to declare a new memory section
  – void boot(void) __attribute__ ((section (".bootloader")));
• Use a linker flag to tell the linker where to put this section
  – -Wl,-section-start=.bootloader=0x1E000
• Make sure your bootloader code compiles to less whatever limitation you’ve set.
I2C Interfacing

• I2C chips are some of the most common, and easy to use components available for interfacing with small Microcontrollers
  – IO Expanders, ADCs, DACs, EEPROMs, RTCs, etc.
• The I2C bus is a two-wire, single-duplex bus
• The AVR supports 100kHz and 400kHz data rates
• The AVR can address up to 127 slave devices
The I2C interface is actually pretty complicated to implement in a manner which is consistent with standards, and won’t cause hard locks on your CPU. As an advantage, once you get this code working once, you can reliably interface with any I2C compliant devices. If you don’t understand the code I’ve provided today, don’t panic: I’ll post some simpler, easier to break code later.
Using the ADC

- The ADC on the AVR is complex, and robust
- 6 channel, 10 bit ADC
- AVcc or 1.1v Bandgap conversion references
- Up to 15 Ksps at maximum resolution
ADC Registers of Note

- **ADCSRA: ADC Control and Status Register A**
  - Bits 2:0, Prescaler value. The input clock to the ADC is divided by a value provided here. In order to provide maximum resolution, the ADC must be clocked between 50 and 200 khz.
  - Bit 6, ADC Start Conversion
  - Bit 7, ADC Enable

- **ADMUX: ADC Multiplexer Selection**
  - Bits 3:0, ADC input selection
  - Bits 7:6, Voltage Reference Selection

- **ADCL and ADCH, the ADC result registers**
ADC Usage

• Configure the prescaler to the desired frequency
  – A table in the datasheet gives an idea of the conversion times you can expect.
• Configure the correct voltage reference
  – If using AREF as a reference, you cannot use AVCC or the 1.1v reference
• Configure the correct ADC pin in the ADC multiplexer
• Start the conversion
• Wait for the conversion to complete
ADC Notes

• You can configure manual conversions, or successive conversions. Successive conversions complete faster after the initial configuration, but must be reconfigured if another analog pin read is desired.

• The Internal Bandgap reference will remain steady under all VCC conditions, while AVCC and AREF can fluctuate if you’re not very careful. This can cause problems.

• If you haven’t taken measures to minimize noise, the lower 2 bits of your ADC conversion will likely be junk.
ADC Example

- The AVR ADC can be used as a simple, inaccurate temperature sensor.
- Manufacturing processes ensure that this sensor is completely non-linear, and must be calibrated on a chip-by-chip basis.
- Regardless, an example of this usage is included.
- In order to get this example working properly, connect AVCC to VCC, and AREF to ground with a small capacitor.
Serial or USB to computer is the cheapest. An FTDI chip and support electronics comes in under $10. If you’re using some wizardry, you can write your own USB interface for even less. Serial level converters are dirt cheap. Now all you have to do is figure out the software on the computer.

ENC28J60 chips are cheap, $5 each. They require a lot of support electronics. They generate MAC level packets on the wire, and receiver them. They operate at 10mbps. Modules can be found for under $30 with everything on board. This chip is the basis of the Arduino Ethernet Shield. You have to write your own UDP or TCP stack, as well as handle ARP and other Ethernet Standard protocols.

Wiznet chips are cheap, but forget about designing a support board without a lot of money and experience. Just buy a $25 module. These bad boys run at 10/100 mbps, and include a TCP/IP stack on board. These are probably the best way to go right now, unless you’d prefer the additional flexibility of generating your own MAC packets.

There are hundreds of other microcontrollers out there with built in MAC circuitry which might be worth considering if your project will be doing any serious interaction with networked devices. Notably, some of the lesser ARM controllers.
Motors

• Servos
  – Use one of the PWM channels on the AVR. Easy to generate PWM signals using the timer functionality.

• DC motors
  – Operate through an H-bridge with appropriate flyback diodes. Logic levels can easily turn these motors on and off.
  – PWM through the H-bridge to control speed

• Stepper Motors
  – Stepper Motor drivers are cheap ($10-$15), and are a very easy way to interface with a variety of stepper motors with high accuracy, and high power. These can be driven with logic level inputs.
The atmega chips are currently the workhorse line of 8-bit microcontrollers. Of these, there are three lines which are in common use among hackers and hobbyists. The most common is the 48/88/168/328 line. These come in small form factor, breadboardable DIP packages, and include a reasonable amount of memory, as well as a number of useful peripherals. We’ve been looking at these during the course of this workshop.

The 164/324/644/1284 line are the bigger brothers to the previous line. They also come in DIP packages, but have more pins, and are wider. This line expands upward to include more RAM, more Flash, and more eeprom, along with more peripherals, like more UARTs, more counters, and more IO, along with the ability to be programmed and debugged through JTAG.

The 640/1280/1281/2560/2561 line is the biggest brother of all of these. These chips are only available in 64 or 100 pin surface mount packages, but include yet more flash and more ram, more timers, more PWM channels, more ADC channels, more UARTs, along with the ability to do JTAG debugging.

The atTiny line use less power, have fewer pins, and are smaller, all around. In general, they have less flash and less ram, and may run at lower clock speeds. They range in pin count from 6 to 28. The atTiny 13 is the smallest in common use, with 8 pins, 1k of Flash, and 64 bytes of RAM, along with a 4 channel, 10 bit ADC.

The 25/45/85 are slightly larger and more capable, at 8 pins, they have 128 or 256 bytes of RAM, timers with PWM, ADC, and a universal serial interface.

The 24A/44/84 are larger still, at 14 pins. They have more timers, more ADCs, on board temperature sensor, and an external debug interface.

The 48/88 are the largest, at 28 pins. They are almost equivalent to the smaller atMegas, with less RAM.

The xmega are new chips from Atmel. These chips only come in surface mount flavors. They are ridiculously capable, a hybrid of 8 and 16 bit microcontrollers. The smallest chips include 16k Flash, 2k RAM, 32mhz clock speed, 12 bit ADC, 4 counters, 2 uarts, 2 spi, and 2 i2c.

The largest include 384k of Flash, 32k of RAM expandable to 16MB, DMA, multiple 12 bit ADCs, four DACs, eight UARTs, and AES/DES crypto engines.
Acknowledgements

- The Membership of LVL1, for inspiring and helping
- Sparkfun, for excellent tutorials and prototyping products
- Limor Fried aka Lady Ada, for developing an excellent cheap AVR programmer
Resources

- WinAVR: http://sourceforge.net/projects/winavr/files/
- AVR LibC: http://www.nongnu.org/avr-libc/
- AVR Fuse Calculator: http://www.engbedded.com/fusecalc
- Sparkfun Forums: http://forum.sparkfun.com/
- AVR Freaks: http://www.avrfreaks.net/
- Alternative to WinAVR: http://www.makehackvoid.com/group-projects/mhvavrtools

- Kernighan and Ritchie’s “The C Programming Language” is one of the best C books ever written. Over 20 years old, and still definitive, it’s worth picking up. ISBN: 978-0131103627
Prototyping Resources

- Evil Mad Scientist sells a great minimalist target board
- The Teensy USB development board is a great board to program in C without the Arduino environment. Small, cheap and capable.
- The Bus Pirate, available from Seeed Studios for $30, is a great way to gain insight into how the other microchips you’re using will communicate with the AVR chip.
- The Open Logic Sniffer, from Seeed Studios for $45, is a great logic probe. With up to 32 channels and 70MHz + speeds, this open source logic probe is nicer than most $250 logic probes.
- The AVR Dragon is a great programmer and debugger, which can be easily paired with the Gnu Debugger to perform more robust debugging of code. It can program and AVR chip out there, including the new Xmegas. At $50, it’s a great investment.
It’s my hope that you’ve gotten a little more out of this workshop than just how to work with AVRs. Hopefully, I’ve set you guys up on a road to understanding how microcontrollers work in general. If an AVR chip isn’t enough, you might want to consider these options:

Every ARM chip out there is so much more powerful than the AVR chips, it’s almost hard to believe. The Cortex M0 is the smallest chip available right now, with similar peripheral interfaces and speed. It’s designed for the same task, but with much more complexity, and much more capability. The M3 is the big brother of this chip, and includes lots of flash, ram, and other built in peripherals that are difficult to summarize. Many include USB and Ethernet functionality. These chips are difficult to break into, however, because there is no single simple toolchain for the ARM processors.

The 32 bit AVRs are a little easier to program for. They aren’t nearly as capable as the ARM chips, but they fill a niche. Programming them is radically different from programming an 8 Bit microcontroller, however, so don’t use these chips hoping for a familiar environment.

The Xmegas are still AVRs, and still programmed in largely the same way, but they are way more capable than anything else in the AVR family. Consider these if some of the AVRs discussed earlier aren’t good enough.

Finally, PICs definitely serve a purpose. The free toolchains available for PIC have drastically increased their usability in the past 5 years, and Microchip has a mindboggling large array of different PICs to choose from. Don’t not use a PIC out of zealotry for a particular chip. The only reason I stick with AVRs is because I don’t want to invest money in another toolchain.
Questions?