Circuits

Information and tutorials on building circuits, ranging from radios to microcontrollers

- Arduino in Assembly (First Post)

Arduino in Assembly (First Post)

What is this?

Hello! This is my first post for the foxgirl laboratory blog. We'll be going over different projects as I experiment with projects and learn how they work! I'm doing this purely for the science (and engineering) as I want to learn more how different aspects of the universe work. For those interested in who I am, I'm a trans woman who writes software for fun. I would like to be able to write software which I get paid for, so if any companies are hiring for that position, let me know if you'd like to hire me.

First, we'll set some rules for how this blog will work. This is NOT a copy/paste tutorial. Chances are likely, the copy/pasted code and designs will not work, or at the very least, won't work as it should. I'll be writing this as I learn, so mistakes will be left in, with corrections added and noted later on (possibly even in different chapters). I'll be writing in a mixture of freeform and structured writing. Also, I'll be uploading files I believe I have permission to use (either from copyleft licensing or fair use laws under the United States jurisdiction). For files which I'm not sure I can upload, I'll link to them instead.

I plan on focusing more on the hardware side of computing in the near future. Currently, I have a 6502 microprocessor and other parts which are similar to the parts used by Ben Eater's Hello 6502 tutorial. The parts I have can be seen at this Mouser project link. If you haven't ordered from Mouser before, do note, they primarily sell to manufacturing companies, so Mouser will ask you what your company does as part of the Know Your Customer guidelines put in place by the US Export Administration Regulations. I told them that I was wanting to know more about how a computer works on a hardware level, will be following along with Ben Eater's 6502 tutorials, and will be posting about them on this site you are reading right now. They'll ask after you place your first order, so don't wait until Thursday morning to make the order, else you'll be waiting for far longer than 2 days before they start processing the order. Also, make sure you don't use the products for anything illegal like making weapons of mass destruction or anything else which would violate human rights.

In the future, I'll be looking into photonic circuits. Currently, I have no idea how to get a photonic chip.

When writing this blog, I'll be assuming you, the reader, have at least some experience with higher level languages (such as C), GNU/Linux, the shell, git, reading documentation, binary in different formats such as hexadecimal, and basic mathematics such as algebra.

Arduino

For this first post, we're going to program an Atmega2560 Arduino to blink the internal LED on/off every second in AVR assembly. Do note, the compilers are more clever than you, and they work best when you don't try to out clever them, so for any production programs, use a higher level language like C unless you can absolutely verify your assembly is faster in that specific portion of a program. For the sake of this project, the point isn't to make the most efficient code in the world, it's to learn how computers work at a lower level. When asking for help with getting proof of life on my Arduino after flashing it with assembly, I was sent a link to this Atmega328P kit. The kit will be useful for more in depth learning how Atmel microprocessors work.

When starting off with this project, you'll want both an Arduino (based) Atmega2560 and a USBasp ISP cable. The firmware on the USBasp ISP cable is out of date, so it may have trouble flashing bigger projects, and we'll be looking into how the customers in the reviews flashed the updated firmware to the ISP cable in the future.

The ISP cable is required to flash assembly directly to the Arduino as the binaries we are creating will not have the Optiboot bootloader which the Arduino IDE tacks onto sketches (Arduino's programs). The Arduino IDE assumes the Optiboot bootloader is already installed on the Arduino when it uploads sketches. If it isn't, like it won't be after flashing the blink program, then you'll need to use the IDE to flash the bootloader back to it using the ISP cable.

The Atmega2560 runs at a clock cycle of 16 MHz or 16,000,000 cycles per second. CPUs operate on cycles and each instruction takes a specific number of cycles to execute. The number of cycles is documented per instruction on the ISA datasheet. The Instruction Set Architecture or ISA is a description of how the code should work. It doesn't always work that way, hence the prevalence of Illegal Instructions on older processors. You can read more about Illegal Instructions on this reverse engineering blog.

Arduino Blinker

In order to compile the blink program, you'll need an AVR compatible assembler such as AVRA. I installed my copy of AVRA from the Arch User Repository or AUR. AVRA is based off of the official Microchip AVR Assembler. The documentation can be found on Microchip's website. The Atmega2560 is 8 bit. You can find all sorts of documentation for the Atmega2560 on microchip's website. For flashing the binaries to the Arduino and also reading the binaries back, you'll need avrdude. You can find documentation for avrdude in the git repo. avrdude is also what the Arduino IDE needs to flash the bootloader back when you are done with this project.

When compiling the program, I use the command below to output both the hex file avrdude flashes as well as a list file which compares the binary opcodes to the assembly instructions. The AVR

microprocessor is based on the RISC architecture.

```
avra blink.asm -o blink.hex -l blink.list
```

When flashing the program to the Arduino Mega 2560, I use the following command and specify the device is the Atmega2560, as well as specify the programmer uses USBasp, and specify the location of final output from the assembler. The hex format allows flashing bytes only where they need to be flashed instead of having to pad out the file to get the bytes in the correct positions.

```
avrdude -p atmega2560 -c USBasp -U flash:w:blink.hex
```

I'll be using ports to write to the internal LED pin. The Internal LED is on pin D13 (digital 13) of the Arduino 2560. Pin D13 is on Port B as can be seen on the Arduino Mega 2560 pinout datasheet.

There's also a pinout datasheet on the Atmega2560 microprocessor itself. The text, PB7, which is attached to the D13 section of the datasheet specifies that the pin is on bit 7 of port B. Such that bit 7 means the highest bit of a byte which is mapped from 7 to 0. This would look like binary 10000000 on if bit 7 was the only pin set to output. Also, I should note, the Atmega2560 is little endian. This means that when a number or other data takes more than one byte to represent it, the part that's read first will be the smallest portion of the number, What this means is, hex `0x01 0x00` would be the number 1, whereas `0x00 0x01` would be 256. One byte can go from 0 to 255, and then we tack on the 1 on a second byte to signify value 256. More information on how endianness works can be found here. There's also a hex converter which accounts for endianness by ScadaCore.

When using ports, we need to set all of the pins corresponding to that port at once, even if we don't plan on using them. When a bit is set to 1, it means output, and when set to 0, it means input. We set the pin modes by writing to data direction register B also known as DDRB. We can write to the pins by writing a value to port B or PORTB and we can read from the pins by reading from pin B or PINB.

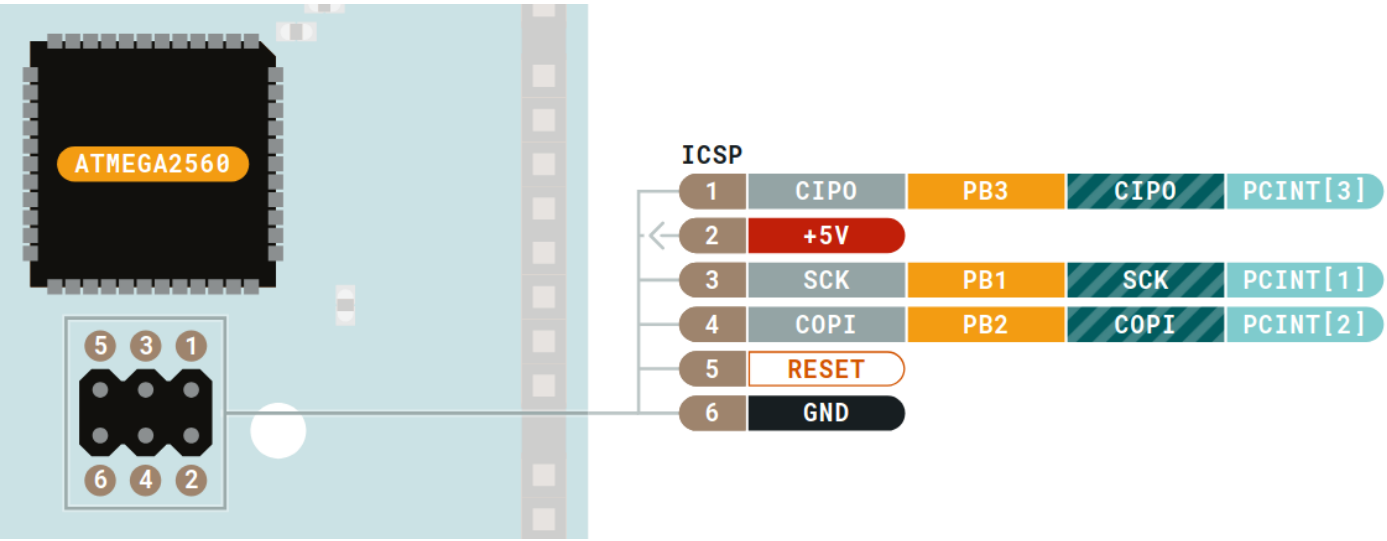
We will be writing the below program in AVR assembly.

```
void setup() {
  // pinMode(LED_BUILTIN, OUTPUT);
  DDRB = 0b10000000;
}

void loop() {
  // digitalWrite(LED_BUILTIN, LOW);
  PORTB = 0b00000000;
  delay(1000);
```

```
  // digitalWrite(LED_BUILTIN, HIGH);
  PORTB = 0b10000000;
  delay(1000);
}
```

Before we start writing the assembly, we want to make sure we know how to connect the USBasp programmer to the Arduino. There's two ICSP pins on an Atmega2560. The one we want will be a grid of 2x3 pins immediately next to the Atmega2560 microprocessor. I've included a screenshot below to show the pins and its pinout. The screenshot came from the Arduino Mega 2560 pinout datasheet. You can find all of the Atmega 2560 datasheets on Arduino's site. The USBAsp uses the older terms for these pins, whereas the Arduino uses the newer terms. You'll want to connect the MISO pin on the USBAsp to the CIPO pin on the Arduino. You can see how the terms have changed on this Arduino ISP page. This connection will be useful for flashing both the Arduino bootloader when you are done with the project, as well as any custom binaries you write with no bootloader.



When we first write our assembly file, we will want to specify a target processor. This will allow the compiler to know if an instruction is valid for our target as well as determine what opcodes to convert the assembly to and to verify if the code stays within the range of the address bus of that particular processor. A list of assembler directives will be able to help explain how each of the directives works. The directives themselves aren't assembly and won't be stored as opcodes themselves, only used for figuring out how to assemble the program. Since we are targeting the Atmega2560, we specify the target processor with the below directive.

```
.device Atmega2560
```

The Atmega2560 and other AVR devices maps it's memory out using the Harvard Architecture. The Harvard Architecture uses separate address busses for different types of memory. This will allow RAM, Flash, and EEPROM to use the same addresses without affecting each other. The other main architecture is the Von Neumann Architecture which has the different memories and other addressable components on the same address bus. My future Hello 6502 will use the Von Neumann

Architecture. The memory map specifics can be found on the Atmega Microcontroller datasheet. I'm still working out the specifics of how much can actually be used for each memory. I'll have to empirically test what I can now that I've performed the math which I believe to be correct.

The application data or code will start at address 0 on the Atmega2560 microprocessor. I've marked it as `0x000000` to hint at the available address space to us, although the address space is much smaller than `0xFFFFFF`. The origin directive tells the compiler where to expect the code to be installed on the device according to the address bus. In this particular case, the code will be installed at address 0. In the future 6502 project, the code will be installed at `0x8000`, although it'll actually be on address 0 of the eeprom. We'll go into more detail on using A15 for chip enable on a future post (assuming I stick with the plan). avrdude will write the code to the correct locations for you from the hex file.

```
.org 0x000000
```

The define directive will allow us to create labels for registers which are self explanatory. There are 32 8-bit general purpose registers for the AVR architecture, and we are only using 4 of them for this blink program, so we don't need to keep track of if anything else has written to it or create more than one label for a register (for each purpose the register is assigned). The equals directive allows assigning data to a label for future reference. Normally we would import a pregenerated file with standard data labels, however, I just used the Atmega Microcontroller datasheet to grab the relevant information. I've underlined the values below in blue. I'm not sure what the hex values in parenthesis refer to at this time. The F_CPU value, I pulled from converting the 16 MHz clock to Hertz.

13.4.5 PORTB – Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

13.4.6 DDRB – Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

13.4.7 PINB – Port B Input Pins Address

Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

```
.def    rTemp = r16
.def    rCounterInner = r17
.def    rCounterMiddle = r18
```

```
.def    rCounterOuter = r19


;// I've modified the comments to use // instead of ; so the syntax highlighting looks correct.
;// As this blog software doesn't support assembly of any kind, I've been marking it as the Dart language
.equ    F_CPU = 16000000  ;// Cycles per second (doesn't actually change the oscillator clock rate)
.equ    PINB = 0x03  ;// Input - Port B Input Pins Address
.equ    DDRB = 0x04  ;// Direction - Port B Data Direction Register (bit value 1 means output, bit value 0 means input)
.equ    PORTB = 0x05  ;// Output - Port B Data Register
```

Now we've entered the actual code. The comments should explain what the individual lines do, and I'll explain the bigger functions here. The setup function below sets the pins on DDR B to either be input or output. We can't write values to the I/O space directly, so we load the value into a register, then we copy the value from the register into the I/O space. I'm not familiar with the design decisions of microprocessor manufacturers, so I don't know what the hardware mechanism is which makes it require copying to a general purpose register first before copying to the I/O space.

```
setup:
    ldi rTemp,0b10000000  ;// load's pin directions into temp register (pins 7-0, left to right) - 1 cycle
    out DDRB,rTemp  ;// Writes temp register contents to DDR E - 1 cycle (`in` is also 1 cycle)
```

This is the main loop which runs indefinitely. The labels don't actually define a separation of code into functions, they just mark addresses with easy to remember names. Since setup doesn't jump anywhere before the end of the section, it'll just continue executing into the code marked with the start label. The code turns all pins on Port B off (only pin 7 on this port is actually in output mode anyway). It then calls a delay function which we write later on in the code, then it turns pin 7 back on and calls delays again before jumping back to the address at the start label. `rcall` will automatically grab the address of the instruction after the call and push it onto the stack before jumping to the labeled address. `ret` will automatically pop the address saved by `rcall` off of the stack and jumps to the saved address. On the AVR architecture, the stack starts from higher addresses and moves backwards to the lower addresses when pushing onto the stack. The stack is placed into either Internal or External SRAM (as defined by the program or if not defined, defaults to the last address of the Internal SRAM). Looking at the schematic and pinout of the Arduino Mega 2560, an external SRAM is not built in to the Arduino Mega 2560 board, however, the pins for the external SRAM are able to be used. The Atmega pins are AD7 to AD0 for the lower and A15 to A8 with Arduino pins PA0 to PA7 and PC0 to PC7 respectively.

```
start:
    ldi rTemp, 0b00000000  ;// 1 cycle
    out PORTB,rTemp  ;// Writes temp register contents to Port B - 1 cycle


    rcall delay  ;// 2 to 4 cycles (dependent on hardware)
```

```
    ldi rTemp, 0b10000000  ;// 1 cycle
    out PORTB,rTemp  ;// 1 cycle

    rcall delay  ;// 2 to 4 cycles (dependent on hardware)

    jmp start  ;// 3 cycles
```

This is the delay function. It sets up the counters under the delay label, and then it starts decrementing the counters. Each register can only hold 8 bits which can go up to 256 values. We use an outer, middle, and inner loop to get the number of loops needed to last a second. At... THIS WILL BE CONTINUED AT A LATER DATE WHEN I'M NOT TIRED

Note: There is a problem with the math to calculate how many cycles it takes to last a second. Either the clock is faster than 16 MHz or the math is wrong.

```
delay:
    ;// F_CPU (16 MHz or 16,000,000) / Number of cycles per loop (16 right now) / inner loop counter (255) /
middle loop counter (255) = outer loop counter (15.378~)
    ldi rCounterInner, 255  ;// 1 cycle
    ldi rCounterMiddle, 255  ;// 1 cycle
    ldi rCounterOuter, 32  ;// 1 cycle

delay_loop:
    ;// Decrement Counter by 1
    dec rCounterInner  ;// 1 cycle

    nop  ;// 1 cycle
    nop  ;// 1 cycle
    nop  ;// 1 cycle

    ;// Inner counter
    ;// If counter not 0, jump back to start of loop
    cpi rCounterInner, 0  ;// 1 cycle
    brne delay_loop  ;// If false, takes 1 cycle, if true, takes 2 cycles

    ;// Middle Counter
    dec rCounterMiddle  ;// 1 cycle
    cpi rCounterMiddle, 0  ;// 1 cycle
    brne delay_loop  ;// If false, takes 1 cycle, if true, takes 2 cycles
```

```
;// Outer Counter
dec rCounterOuter  ;// 1 cycle
cpi rCounterOuter, 0  ;// 1 cycle
brne delay_loop  ;// If false, takes 1 cycle, if true, takes 2 cycles

ret  ;// 4 to 6 cycles (dependent on hardware)
```