I like playing with my Arduino Uno board and its graphical development environment. It’s a tool that makes it easy to create programs and hides many details, but that leaves me wanting to look beneath, to understand the details that are normally hidden. I felt the urge to work closer to the hardware, stepping away from the default library and the Java IDE and using the compiler directly from the command line. In particular, using C to program the Arduino means usually being able to create smaller programs, and with more fine grained control of what happens. C is adopted worldwide to program for small microprocessors because it gives a good trade-off between development effort and program efficiency, and because of its history there are well-optimized libraries, extensive guides and ways to solve problems. So if you find that Arduino language creates programs that are too big or too slow but you want to squeeze the performance out of your board, or you want a more modular approach, moving to C could be the right choice.

Fortunately all the tools are there, because the Arduino IDE uses them under the hood. In my particular case, since I develop on a Linux machine, Arduino uses the avr-gcc compiler and the avrdude uploading tool. I can use these tools to develop a program with pure C code, instead of the Arduino language, and upload that program on the board. The Arduino IDE preferences contains verbosity options that have the effect of printing the commands that are run while the program is compiled and uploaded. This was very useful to understand what the graphical user interface is doing, which turns out is a common workflow for C and C++ builds. We cam mimic this flow, build our program from C calling the avr-gcc command with the right options and upload it running avrdude with the right options. The diagram below shows the “toolchain” flow from writing C code to uploading to Arduino board.

To make it simple I implemented the classic blink program that toggles the output pin connected to the on-board LED. avr-gcc toolchain doesn’t know the Arduino Uno board layout, it only knows the microcontroller that is mounted on it, which is the Atmega328p, so we need to read/write directly the hardware I/O registers of that particular chip, and look at its pinout to understand the pin names and where they go: everything can be found in the Atmega328p datasheet. Instead from the Arduino Uno schematics we can find out where the pins are connected, for example we can find that the LED is connected to the PB5 pin of the Atmega328p chip, so that’s the pin we need to control. Now we need to write the code that toggles the PB5 pin. The AVR compiler is complemented with a C library: avr-libc, which contains useful functions and headers to access the functionalities and I/Os of AVR chips. It also makes it easy to write complete C programs without using assembly language. Inside the avr-libc manual and the Atmega328p datasheet there are many examples on how to toggle IOs, and with them I prepared the following code in a file called “led.c“:

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

#define BLINK_DELAY_MS 1000

int main (void)
{
 /* set pin 5 of PORTB for output*/
  DDRB |= _BV(DDB5);

 while(1) {
   /* set pin 5 high to turn led on */
     PORTB |= _BV(PORTB5);
       _delay_ms(BLINK_DELAY_MS);

  /* set pin 5 low to turn led off */
    PORTB &= ~_BV(PORTB5);
      _delay_ms(BLINK_DELAY_MS);
  }
}
```

The Port “B” of the microcontroller can be changed bit by bit with special instructions called “sbi” and “cbi“. In C we use the bitwise “|=” and “&=” assignment operators, which usually read and write a variable, but the compiler recognizes those kind of accesses generating optimized assembly in case of bit-wise operations, and there is no read operation involved. The “_BV” macro together with PORTB5 is used to build a value that contains one on the bit that corresponds to PB5. The main function contains an infinite loop that raises and lowers the bit 5 of PORTB register and between these operations waits for one second using the library function “_delay_ms“. If you installed avr-gcc on your Linux machine, the ports definitions and useful macros (like “_BV“) can be found in the “/usr/lib/avr/include/avr/iom328p.h” and “/usr/lib/avr/include/avr/sfr_defs.h” headers, or in the directory where the library has been installed.

The compiler is able to create an ELF executable program that contain machine code and other information such as program section memory layout and debug information. In order to compile a program and upload it to the Arduino Uno, We need to create an IHEX file and use the avrdude tool to load it inside the Flash. The tool to convert the ELF into IHEX in our case is avr-objcopy. Now it’s time to run the command lines that build and upload the LED blink program. Most of the parameters and options of the avr-gcc and the avrdude tools for the Uno board can be found in the “hardware/arduino/boards.txt” file from inside the Arduino IDE installation directory, and some other information is present in the avrdude manual. The commands that I used to compile and upload the “led.c” code above are:

```
$ avr-gcc -Os -DF_CPU=16000000UL -mmcu=atmega328p -c -o led.o led.c
$ avr-gcc -mmcu=atmega328p led.o -o led
$ avr-objcopy -O ihex -R .eeprom led led.hex
$ avrdude -F -V -c arduino -p ATMEGA328P -P /dev/ttyACM0 -b 115200 -U flash:w:led.hex

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.00s

avrdude: Device signature = 0x1e950f
avrdude: NOTE: FLASH memory has been specified, an erase cycle will be performed
 To disable this feature, specify the -D option.
 avrdude: erasing chip
 avrdude: reading input file "led.hex"
 avrdude: input file led.hex auto detected as Intel Hex
 avrdude: writing flash (88 bytes):

Writing | ################################################## | 100% 0.02s

avrdude: 88 bytes of flash written

avrdude: safemode: Fuses OK

avrdude done.  Thank you.
```

The first command line takes the C source file and compiles it into an object file. The options tell the compilerto optimize for code size,what is the clock frequency (it’s useful for delay functions for example) and which is the processor for which to compile code. The second commands links the object file together with system libraries (that are linked implicitly as needed) into an ELF program. The third command converts the ELF program into an IHEX file. The fourth command uploads the IHEX data ito the Atmega chip embedded flash, and the options tells avrdude program to communicate using the Arduino serial protocol, through a particular serial port which is the Linux device “/dev/ttyACM0“, and to use 115200bps as the data rate.

After the commands are done the code is uploaded and the led starts blinking. That went well at the first try, mostly due to the fact that the tools have good support

for the Arduino.

It is possible to list the ELF program in terms of assembly instructions (disassemble the program) by using this command:

```
$ avr-objdump -d led >led.lst
```

It is also possible to do a backup of the Atmega328p Flash content by using this command:

```
$ avrdude -F -V -c arduino -p ATMEGA328P -P /dev/ttyACM0 -b 115200 -U
flash:r:flash_backup.hex:i
```

Note that this guide is written for Linux machines, but it could be adapted to work on Windows and Mac. In particular the compilation that builds the ELF program file should be roughly the same, but the upload part could be very different.