Howto make an Arduino fast enough to...

Willem Maes

May 1, 2018

0.1 Introduction

Most problems student face when using the Arduino in their projects seems to be the speed. There seems to be a general opinion that an Arduino is to slow to do some decent measurements. We will see that the microntroller used is fast enough but that the code generated by the Arduino IDE (Integrated Development Environment) makes an Arduino verry slow. As you'll see throughout this article, fast code is not only important for doing calculations, but even more so for IO operations. If you venture in to measurements and control this will become even more clear since much of the work done by the microcontroller results in IO. Usually a faster feedback loop means better performance.

Throughout this article, I'll be focusing on the Arduino Uno since that seems to be the most common board out there, although much of this article should also apply to the other boards as well.

0.2 Why Arduinos are Slow

0.2.1 Clock Speed

First of all, you are only as fast as your clock (disregarding multi-core processors), which the Arduino Uno defaults to using a 16Mhz crystal. What that means is the ATmega microcontroller can execute up to 16 million instructions per second. Now, 16 million instructions per second may sound like a lot (and it is, sort of), but when you consider what all an Arduino needs to do to execute even simple operations, it really isn't that much. For many projects, the clock cycles are shared between things like calculations, I2C communication, reading and writing to pins and registers, and many more operations.

Even then, seemingly simple commands can take up quite a bit of clock cycles, such as setting a digital pin to high. This is one of the simplest IO operations you can perform on an Arduino, but it actually takes a very long time (over 50 clock cycles!) because of the amount of code used in the

```
digitalWrite();
```

method, which I'll address in the next section. So a faster clock would allow you to execute the instructions at a faster pace.

0.2.2 Safety Checks and Validation

So, outside of the clock speed, why are Arduinos slow?

Well, it mostly has to do with some of the standard method calls and objects we use throughout our code. Here are just a few of the main culprits:

```
digitalWrite();
digitalRead();
pinMode();
```

Many of these methods suffer from the same drawbacks, so let's take a look at the code for one of the most commonly used methods,

```
 digitalWrite();
```

```
void digitalWrite(uint8_t pin, uint8_t val)
{
    uint8_t timer = digitalPinToTimer(pin);
    uint8_t bit = digitalPinToBitMask(pin);
    uint8_t port = digitalPinToPort(pin);
    volatile uint8_t *out;

    if (port == NOT_A_PIN) return;

    // If the pin that support PWM output, we need to turn it
    off
    // before doing a digital write.
    if (timer != NOT_ON_TIMER) turnOffPWM(timer);

    out = portOutputRegister(port);

    uint8_t oldSREG = SREG;
    cli();

    if (val == LOW) {
        *out &= ~bit;
    } else {
        *out |= bit;
    }

    SREG = oldSREG;
}
```

As you can see, there is quite a bit going on here.All of this code is executed every time you do a **digitalWrite()**. But shouldn't it be much simpler? All we really need to do is set the pin high or low in a register, right? Turns out that the Arduino creators decided it was more important to add safety checks and validation to the code than it was to make the code fast. After all, this is a platform targeted more towards beginners and education than it is to power users and CPU-intensive applications.

The first few lines use the **pin** parameter to find the corresponding **timer , bit** and **port** for the given pin. The **port** is actually just a memory-mapped register, which controls multiple pins. To only turn on or off the pin we want, we need to determine which bit of the register our pin corresponds to, which is what the **digitalPinToBitMask()** function does.

Once we've found the **timer, bit,** and port, we check to make sure it is in fact a valid **pin**. This line isn't required for **digitalWrite()** to do its job, but it acts as a safety net for more inexperienced programmers (and even experienced ones). We'd hate to be writing to the wrong memory location and corrupt the program.

The

```
1  if (timer != NOT_ON_TIMER) ...
2
```

line is there to make sure we end any previous PWM usage of the pin before we write a "constant" high or low. Many of the pins on Arduinos can also be used for PWM output, which requires a timer to operate by timing the duty cycles. If needed, this line will turn off the PWM. So again, to ensure we don't see any weird behavior, this is a safety check meant to help the user.

And finally, in the last few lines we're actually setting the given value to the given port.

The safety checks do slow down the execution quite a bit, but it also makes debugging much easier. This way when something goes wrong, you're less likely to get weird behavior that will leave you scratching your head. There is nothing more frustrating to have seemingly logical code and to not get the expected result.

Since you're working directly with hardware and don't have an operating system to keep you safe, problems can be hard to find.

If speed isn't your goal, then I'd highly recommend you continue to use these method provided by Arduino. There is no point in exposing yourself to unnecessary risk if it doesn't help you reach your end goal.

You should also know that the method calls aren't always slow because of the amount of code it executes, but a contributing factor could be because of the physical limitations of the device. For example, **analogRead()** takes about 100 microseconds per call due to the resolution it provides and clock

it's supplied. A lower ADC resolutions would decrease the time each call takes. However, even then, while the hardware is ultimately the limiting factor here, the Arduino code does conservatively set the ADC max sample rate to only 9600Hz (while capable of around 77Khz). So, Arduinos are much slower than they need to be, almost always because of design choices and trade-offs.

0.3 How to Speed up Arduino

To be clear, we aren't actually making Arduino faster, rather, we're making the code more efficient. I point out this distinction because using these tricks won't give us a faster clock (although we can speed up the clock, which I'll touch on later), it will just execute less code. This is an important distinction because having a faster clock provides us other benefits, like having more precise timers, faster communication, etc.

Also, keep in mind that by using the code below you're making some trade-offs. The programmers who developed Arduino weren't just lousy coders who couldn't write fast code, they consciously made the decision to add validations and safety checks to methods like **digitalWrite()** since it benefits their target customers. Just make sure you understand what can (and will) go wrong with these kinds of trade-offs.

Anyway, on to the code.

0.3.1 Digital Write

Now, I'm not going to show you how to speed up every method, but many of the same concepts from here can be applied to other methods like **pinMode()**. The minimum amount of code you need to write to a pin is:

```
1 #define CLR(x,y) (x&=(~(1<<y)))
2 #define SET(x,y) (x|=(1<<y))
```

Yup, that's it.

As you can see, we get right to the point in these macros. To use them, you'll have to reference both the port and bit position directly instead of conveniently using the pin numbers. For example, we'd be using the macro like this:

```
1 SET(PORTB, 0);
2
```

This would end up writing a **HIGH** value to pin 8 on your Arduino Uno. It's a little bit rough, but much faster. This also means we're more prone to doing something wrong, like reference a non-existent port, write over an active PWM, or a number of other things.

The macro gives us quite a boost, using up an estimated 2 cycles (8Mhz frequency), whereas **digitalWrite()** uses up a whopping 56 cycles (285Khz frequency).

0.3.2 Analog Read

Arduino provides an convenient way to read analog input this using the **analogRead()** function. Without going into much details the **analogRead()** function takes 100 microseconds leading to a theoretical sampling rate of 9600 Hz.

The following piece of code takes 1000 samples using the **analogRead()** and calculates some statistics.

```
void setup()
{
  Serial.begin(115200);
  pinMode(A0, INPUT);
}

void loop()
{
  long t0, t;

  t0 = micros();
  for(int i=0; i<1000; i++) {
    analogRead(A0);
  }
  t = micros()-t0;  // calculate elapsed time

  Serial.print("Time per sample: ");
  Serial.println((float)t/1000);
  Serial.print("Frequency: ");
  Serial.println((float)1000*1000000/t);
  Serial.println();
  delay(2000);
}
```

This code gives 112us per sample for a 8928 Hz sampling rate.
So how can we increase sampling rate?

Table 1: Prescale register

Prescale	ADPS2 ADPS1 ADPS0	Clock freq (MHz)	Sampling rate (KHz)
2	0 0 1	8	615
4	0 1 0	4	307
8	0 1 1	2	153
16	1 0 0	1	76.8
32	1 0 1	0.5	38.4
64	1 1 0	0.25	19.2
128	1 1 1	0.125	9.6

0.3.3 Speedup the analogRead() function

We now need a little more details. The ADC clock is 16 MHz divided by a 'prescale factor'. The prescale is set by default to 128 which leads to 16MHz/128 = 125 KHz ADC clock. Since a conversion takes 13 ADC clocks, the default sample rate is about 9600 Hz (125KHz/13).

Adding some macro's and a few lines of code in the setup() function we can set an ADC prescale to 16 to have a clock of 1 MHz and a sample rate of 76.8KHz.

```
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))//macro to
    clear bit in special function register
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))//macro to set
    bit in special function register
```

```
void setup()
{
  sbi(ADCSRA, ADPS2);
  cbi(ADCSRA, ADPS1);
  cbi(ADCSRA, ADPS0);
  ...

```

The real frequency measured with the test program is 17us per sample for a 58.6 KHz sampling rate.

The following table shows prescale values with registers values and theoretical sample rates. Note that prescale values below 16 are not recommended but just try.

0.3.4 AnalogRead with Interrupts

A better strategy is to avoid calling the **analogRead()** function and use the 'ADC Free Running mode'. This is a mode in which the ADC continuously

converts the input and throws an interrupt at the end of each conversion. This approach has two major advantages:

1. Do not waste time waiting for the next sample allowing to execute additional logic in the loop function.

2. Improve accuracy of sampling reducing jitter.

In this new test program I set the prescale to 16 as the example above getting a 76.8 KHz sampling rate.

```
int numSamples=0;
long t, t0;

void setup()
{
  Serial.begin(115200);

  ADCSRA = 0;             // clear ADCSRA register
  ADCSRB = 0;             // clear ADCSRB register
  ADMUX |= (0 & 0x07);    // set A0 analog input pin
  ADMUX |= (1 << REFS0);  // set reference voltage
  ADMUX |= (1 << ADLAR);  // left align ADC value to 8 bits from
    ADCH register

  // sampling rate is [ADC clock] / [prescaler] / [conversion
    clock cycles]
  // for Arduino Uno ADC clock is 16 MHz and a conversion takes
    13 clock cycles
  //ADCSRA |= (1 << ADPS2) | (1 << ADPS0);    // 32 prescaler
    for 38.5 KHz
  ADCSRA |= (1 << ADPS2);                     // 16 prescaler
    for 76.9 KHz
  //ADCSRA |= (1 << ADPS1) | (1 << ADPS0);    // 8 prescaler for
    153.8 KHz

  ADCSRA |= (1 << ADATE); // enable auto trigger
  ADCSRA |= (1 << ADIE);  // enable interrupts when measurement
    complete
  ADCSRA |= (1 << ADEN);  // enable ADC
  ADCSRA |= (1 << ADSC);  // start ADC measurements
}

ISR(ADC_vect)
{
  byte x = ADCH;  // read 8 bit value from ADC
  numSamples++;
}

```

```
void loop()
{
  if (numSamples>=1000)
  {
    t = micros()-t0;  // calculate elapsed time

    Serial.print("Sampling frequency: ");
    Serial.print((float)1000000/t);
    Serial.println(" KHz");
    delay(2000);

    // restart
    t0 = micros();
    numSamples=0;
  }
}
```

0.3.5 AnalogRead to RAM before Serial.print

If after every **analogRead()** follows a **serial.Print()** we are going to lose a lot of time. A solution for this problem is to write the analog values to RAM. To do this we fill up an array of integers with the analog values we measure. When this array is full we stop measuring and start writing all values with **serial.Print()** to our terminal program. The max number of values (10 bit) we can write to RAM in this way is about 600 for the Arduino UNO.

```
// defines for setting and clearing register bits

#ifndef cbi
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
#ifndef sbi
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif


//let's say you want to read up to 500 values
const unsigned int numReadings = 500;
unsigned int analogVals[numReadings];

void setup()
{

    // set prescale to 16
sbi(ADCSRA,ADPS2) ;//cbi means clear bit
cbi(ADCSRA,ADPS1) ;//sbi means set bit
cbi(ADCSRA,ADPS0) ;
```

```
22
23 Serial.begin(115200);
24 }
25
26
27
28 void loop()
29 {
30
31  for (int i=0; i <= numReadings; i++){
32    analogVals[i] = analogRead(A0);
33    i++;
34    analogVals[i] = analogRead(A1);}
35
36
37
38
39
40 for (int i=0; i <= numReadings; i++){
41 Serial.print(analogVals[i]);
42 Serial.print(",");}
43
44
45
46 Serial.println(";");
47
48
49 }
```

Using this method with prescaler 4 I was able to perform adc conversions in less then 6 us per sample, or a sample frequency of 166 Khz.

0.3.6 Serial

Unfortunately for some tasks, like if you need to use serial communication, there isn't a whole lot you can do to improve the speed, but there are some optimization you can watch out for.

Serial communication is commonly used for sending debugging or status information to the desktop IDE, which means you probably have **Serial.println()** statements all throughout your code. It's easy to forget about these statements after development, so if you're looking for a speed boost and don't need to debug anymore, try removing all the **println()** calls and removing **Serial** from the code altogether. Just having it initialized (and not even using **Serial.println()**) means you're wasting a lot of cycles in the TX and RX interrupts. In one case, it was measured that just having **Serial**

enabled slowed down the **digitalWrite()**s by about 18percent. That's a lot of wasted cycles due to dead code.

0.3.7 Clock Speed

Although I've mostly focused on the software improvements you can make, don't forget that there is always the "simple" improvement of speeding up the clock. I say it this way because it isn't really a simple plug-and-play improvement after all. In order to speed up the clock on an Arduino, you need to insert a new crystal in to the board, which may or may not be difficult depending on your soldering skills.

Once you've put in a new crystal oscillator, you still need to update the bootloader to reflect the change, otherwise it won't be able to receive code over the serial port. And lastly, you'll need to change the F_CPU value to the proper clock speed. If you upped the clock to 20Mhz (the fastest clock the ATmega is rated for), for example, then you'll need to modify a few files in the Arduino IDE:

In **preferences.txt**, change

- from: $build.f_cpu = 16000000L$
- to: $build.f_cpu = 20000000L$

In the **makefile**, change

- from: $F_CPU = 16000000$
- to: $F_CPU = 20000000$

According to some people the ATmega328 can be overclocked to 30Mhz, but I don't recommend it =)

0.4 Conclusion

Hopefully you found something in this article that you can easily apply to your projects, or at least I'm hoping it will encourage you to browse the Arduino source code and the microcontroller's data sheet to find optimizations of your own. The Arduino is a very capable board, but it can be capable of so much more.

If you want to learn more on ADC Free Running mode and tweaking ADC register you can look at the following pages:

https://sites.google.com/site/qeewiki/books/avr-guide/analog-input

http://www.instructables.com/id/Girino-Fast-Arduino-Oscilloscope/

http://www.instructables.com/id/Arduino-Audio-Input/

Most information in this article was based on:

Arduino Forum - Faster Analog Read

http://yaab-arduino.blogspot.be/2015/02/fast-sampling-from-analog-input.html

http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P$_D$*atasheet.pdf*