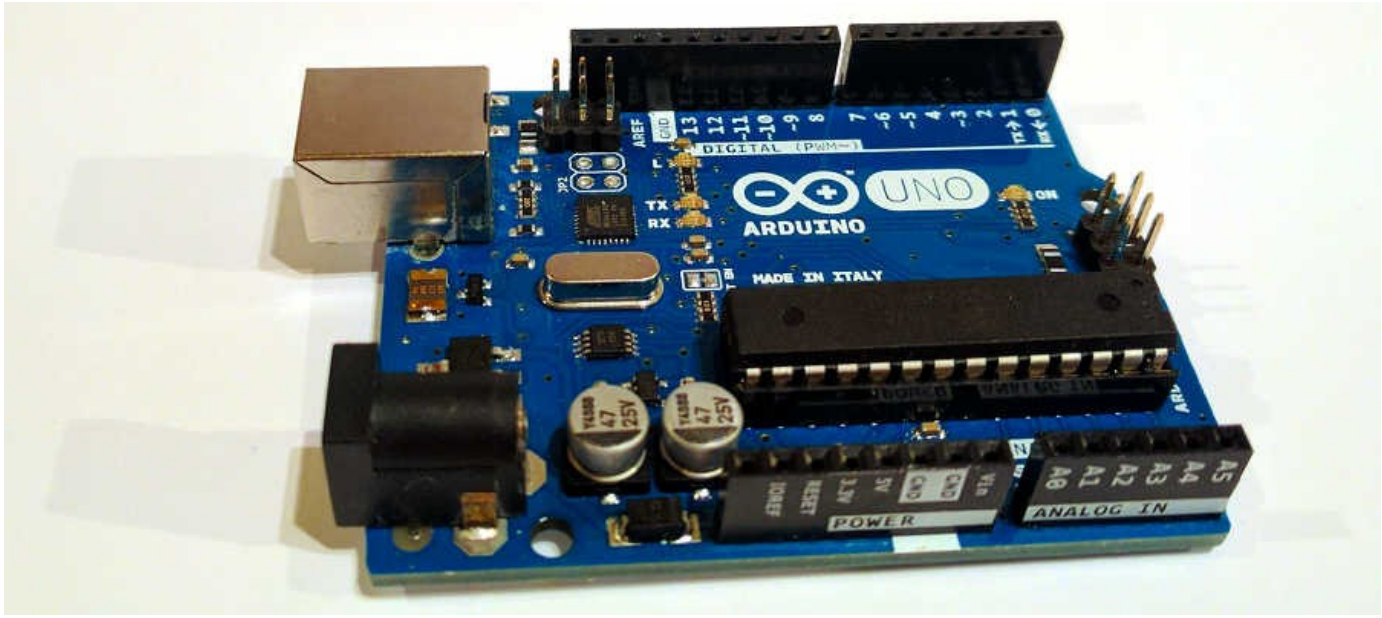
Peter Beard.

Writing mostly about computers and math.

- Blog
- Projects
- Contact

Arduino Power Saving

2017-03-31 06:42:36



A picture I took of my little power hog. Some rights reserved: cc by-sa

I've got a little Arduino project that I run off a 9V battery and I've noticed that it chews through batteries a lot faster than I expected it to — sometimes a battery lasts less than 24 hours. To figure out what was going on I hooked the project up to my power supply and discovered that it draws about 65 mA at 9V, which is a little more than half a watt. There are some LEDs connected to the board but even without those the Arduino Uno all by itself draws 49 mA, which seems way too high to me. I did a little research and testing and I was able to get it down to 36 mA without changing the hardware at all and I hope to get it lower than that with a different battery.

There are two main software solutions to the power consumption problem: putting the CPU to sleep and reducing the clock speed. The Arduino Uno uses an ATmega328P microcontroller, so I took a look at the datasheet to see what sleep modes were available and what I could do with the clock speed.

Sleep Modes

The ATmega328P has six sleep modes but only five of these are supported in the C header (`avr/sleep.h`):

- Idle (SLEEP_MODE_IDLE)
- ADC Noise Reduction (SLEEP_MODE_ADC)
- Power-save (SLEEP_MODE_PWR_SAVE)
- Standby (SLEEP_MODE_STANDBY)
- Power-down (SLEEP_MODE_PWR_DOWN)

Idle mode is the default mode where pretty much everything will wake the CPU and use power. The next mode, ADC noise reduction mode, disables the I/O clocks and is mainly useful for improving the performance of the MCU's on-board ADC. The last three modes are pretty similar in that they all disable pretty much everything the processor can do.

The power-save mode allows the CPU to wake from a timer interrupt, so it's useful if you want the Arduino to periodically wake up and do some work before going back to the low power state. The standby mode keeps the main clock running so it can wake up more quickly (6 cycles) but it can only wake from external interrupts since it disables the timers. When in the power-down mode, the AVR disables everything except the external interrupts so there's no other way to wake it and it takes a while to come back up — this is the lowest power mode.

The sleep mode can be changed by including the avr-libc header `avr/sleep.h` in your project. There are six functions in this header but you really only need to use two unless you're doing something fancy. So, to put the Arduino to sleep you can do something like this:

```
#include <avr/sleep.h>

void setup()
{
  set_sleep_mode(SLEEP_MODE_POWER_DOWN); // Set the sleep mode
  sleep_mode(); // Go to sleep
}
```

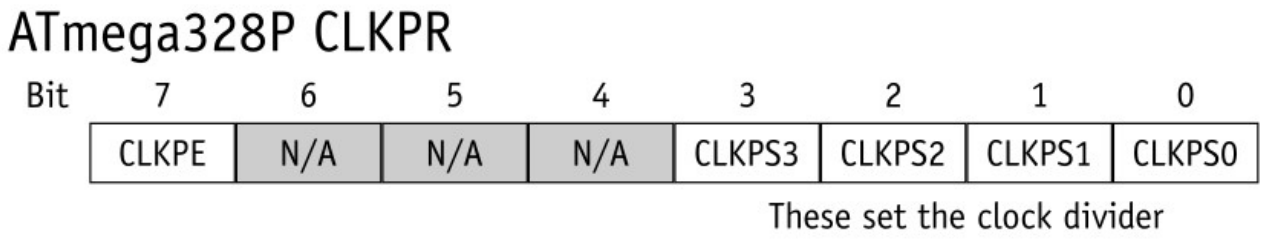
That's it. The Arduino will now sleep pretty much indefinitely since we didn't configure an interrupt or anything else that would wake it up. Once in power-down mode, the only things that will wake the CPU are:

- An external reset
- A watchdog timer interrupt/reset
- A serial address match
- An external level interrupt
- A pin change interrupt

Once the CPU wakes it will continue execution from where it left off, servicing interrupts and so on until it hits another sleep instruction. Since my project has a big button on top I was able to use power-down mode and wake on a pin change interrupt.

Clock Speed

The clock speed on the Uno board is controlled by an external oscillator but the microcontroller has a clock prescaler that can be adjusted at run time. There's a special register (CLKPR) that can be written to change the clock divider. CLKPR is an 8-bit register where the most significant bit is an enable bit (CLKPCE) and the four least significant bits indicate the value of the divider (CLKPS), like this:



For the ATmega328P, the divider is just 2^n where n is the value of the last four bits of CLKPR (CLKPS). For example, setting these bits to 0011 will set the clock divider to 8, which is 2^3.

The compiler knows what CLKPR means so we can use it like any other C variable. We have to set the enable bit before we can set the clock divider so it always requires two assignments like this:

```
void setup()
{
  CLKPR = 0x80;
  CLKPR = 0x08;
}
```

Setting CLKPR to 0x80 sets the enable bit and then we set the bottom half of the register to 1000 so we get a divider of 2^8 or 256. The default clock speed for the Uno is 16 MHz, so with a divider of 256 that means the microcontroller is running at 62.5 kHz. The datasheet says that values above 8 are reserved, so this is the slowest we can get the clock to go. Still, for my project that works out fine since I don't care much about speed as long as it's fast enough to service interrupts in real time.

How Much Power Can Be Saved?

So, having explained the two main ways to save power, let's take a look at some numbers. For these results I used an Arduino Uno connected to 5V at the 5V pin on the board. The microcontroller was programmed to run variations of this program (current was measured with the LED off):

```
#include <avr/sleep.h>
#define LED_PIN 13

void setup()
{
  pinMode(LED_PIN, OUTPUT);
  digitalWrite(LED_PIN, LOW);
}

void loop()
{
  digitalWrite(LED_PIN, HIGH);
  delay(250);
  digitalWrite(LED_PIN, LOW);
  delay(2000);
}
```

Sleep Modes

Mode	Current Draw (mA)	Power Consumption (mW)
Idle	49	245
ADC Noise Reduction	37	185
Power-save	33	165
Standby	31	155
Power-down	30	150

Pretty good; we use about 39% less power in power-down mode than in idle mode. Even the highest-power state (ADC noise reduction) saves 24% so the power saving modes are definitely worth a look.

Clock Speed

For the clock speed tests I ran the processor in idle mode since the power saving mode has a bigger effect on power consumption than the clock speed. I tried using other power save modes but it didn't make any difference as the main thing that matters is whether the clock is running or not, not how fast it's running.

CLKPR	Effective Clock Speed	Current Draw (mA)	Power Consumption (mW)
0x00	16 MHz	49	245
0x01	8 MHz	43	215
0x02	4 MHz	42	210
0x03	2 MHz	37	185
0x04	1 MHz	37	185
0x05	500 kHz	37	185
0x06	250 kHz	37	185
0x07	125 kHz	37	185
0x08	62.5 kHz	37	185

So it looks like the minimum power consumption is 185 mW for idle mode, regardless of clock speed. Still, for projects that have to run in idle mode you can save about 24% by clocking down the processor.

Voltage-Dependent Power Consumption

One final interesting thing to note is that the Arduino board uses less power at lower voltages. I measured these values by supplying power right at the 5V pin but the little barrel jack adapter I built showed the same effect from the other side of the voltage regulator.

Voltage (V)	Current (mA)	Power (mW)
5	37	185
4.75	31	147
4.5	30	135
4.25	30	128
4	24	96
3.75	24	90
3.5	18	63
3.25	18	59
3	18	54
2.75	12	33

The microcontroller started to lock up below 2.75 V but it's still pretty incredible. An 82% reduction in power consumption is really good, but the trade-off is clock speed and stability. These results are at 62.5 kHz since the processor will operate at a lower voltage when the clock speed is slower. You can do something similar at higher clock speeds but the effect is most pronounced at the bottom of the range. Still, since my project doesn't need to be fast I'm thinking about building a little 3 or 4 V regulator just to save a few more milliwatts.

There are a few other things you can do to save power like disabling the watchdog timer and the brown-out reset, but overall the power saving changes are pretty simple and you can save a fair amount of power just by underclocking the processor or configuring a power save mode.