Chapter 6 - Optimizations

Arduino Internals

by Dale Wheat

Apress © 2011 *Citation*

Recommend? yes no

Electronic Measurements

To measure things electrical and electronic, you need a variety of *test and measurement equipment*. For example, you can use a *volt meter* to measure voltage. To measure current, an *ammeter* is used. These tools are good for measuring static or slowly changing levels. For more complex measurements, you use more complex tools.

The Arduino as Test Equipment

What test equipment is available on *your* bench? Please don't feel like you must immediately go out and buy thousands of dollars of test gear (unless that's an option and this was the ~~excuse~~ justification you've been waiting for—if so, go nuts!). You may be surprised by what you can do with what's right in front of you.

For some baseline readings on Arduino performance, you can use the Arduino to measure itself. These measurements can be used for *relative* performance increases but not *absolute* readings, because the Arduino must necessarily spend a small amount of time tending to the measurements themselves.

Open the Arduino software, and type in the short sketch from Listing 6-7. If you're super-lazy, rename the `stop_time` variable to `j` and omit the comments. It's really a very short sketch.

Listing 6-7: The Arduino Measuring the Performance of the Arduino

```
// arduino_performance

void setup() {
  Serial.begin(9600); // initialize serial port to 9600 bps
  Serial.println("Arduino performance test begins now.");
}

extern volatile unsigned long timer0_millis;

void loop() {

  unsigned long i = 0; // test value
  unsigned long stop_time; // in milliseconds

  // calculate stop time (current time + 1 second)
  stop_time = millis() + 1000;

  while(timer0_millis < stop_time) i++; // count!

  // report performance results:
  // number of loop iterations in one second
  Serial.print(i);
  Serial.println(" loops in one second.");

  while(1); // and stop here
}
```

Compile the sketch, and upload it to your Arduino. After the upload is complete, open the Serial Monitor window. Make sure the communication rate is set to 9600. Opening the Serial Monitor window toggles the Data Terminal Ready (DTR) line on the serial port, which should reset your Arduino and start the test over again. If your Arduino lacks the Auto Reset function, you have to coordinate the pressing of the on-board Reset button after opening the Serial Monitor. In either case, you should be greeted by the program self-announcement, "Arduino performance test begins now." (or whatever text, if any, you decided to include there), followed one second later by the test-result message.

Compiled with Arduino 0022 and running on an Arduino Uno, this performance test reports 836,003 loops in one second. The Arduino Mega 2560 reports 835,901 loops in the same time period. That's a lot of loops!

What do these numbers tell you? These preliminary test results tell you how quickly the Arduino can do *absolutely nothing useful*, while measuring the time it takes to do absolutely nothing useful. This is similar to the BogoMIPS metric used to measure the relative speed of CPUs running the Linux kernel (see `http://tldp.org/HOWTO/BogoMips`).

As Ralph Tenny once said, "If you can't measure the frequency, measure the period," reminding you of the reciprocal nature of period and frequency. Technically, you're not measuring the time it takes for each iteration of the empty loop. You're measuring the number of times the loop executes in a given time period (the frequency) and using the reciprocal of the frequency to estimate the loop execution time.

Note that you're using the direct `timer0_millis` trick again to keep track of the Arduino's concept of the current time. Because you're not rewriting the `timer0_millis` value anywhere in this sketch, you don't abandon the usefulness of the `millis()` function, as you did previously in the Blink optimization, even though it isn't being used elsewhere. This little optimization almost doubles the number of do-nothing loops that can be performed by the Arduino in a given second by eliminating the overhead associated with the `millis()` function call and reading the millisecond count directly.

Because a stock Arduino Uno uses a 16MHz oscillator (either a quartz crystal or a ceramic resonator, depending on the batch build date), it can execute up to 16,000,000 instructions per second. Because of the highly optimized machine code that is created for the `while()` loop that is doing all the counting, the loop executes very quickly. It takes 5 cycles to increment the 32-bit loop counter, `i`; 8 cycles to compare the `timer0_millis` millisecond counter with the captured `stop_time` variable; and 2 cycles to jump back to the beginning of the loop, for a total of 15 cycles.

If the Atmel AVR was doing nothing else, it should have been able to run this loop over a million times in any given second. However, this sketch depends heavily on the fact that Timer0 is running and generating interrupts on a regular basis; otherwise the loop would never end because there would be no incrementing of the millisecond timer. The *average* cycle time is 16,000,000 cycles per second ÷ 836,003 loops per second, or just over 19 cycles per loop.

You can now use this cycle count as your baseline reading. Any additional time spent in the `while()` loop can be calculated by offsetting the total time by this baseline reading.

Let's see how long it takes to turn the LED on and off again using the `digitalWrite()` function. You can safely omit the configuration of the LED output pin, because the resulting blink rate will be too high to see with your eyes, anyway. Modify the `while()` loop to look more like Listing 6-8.

Listing 6-8: Measuring the Performance of the `digitalWrite()` Function

```
while(timer0_millis < stop_time) {
  digitalWrite(13, HIGH); // LED on
  digitalWrite(13, LOW); // LED off
  i++; // count!
}
```

Compiled with Arduino 0022, the Arduino Uno can execute 112,653 of these loops in one second, whereas the Arduino Mega 2560 can execute only 67,013 of the same loops. The wide variation is explained by the larger number of I/O pins that the Arduino Mega 2560 must account for in the `digitalWrite()` function.

For the Arduino Uno, these numbers indicate that it takes 16,000,000 cycles per second ÷ 112,653 loops per second, or just over 142 cycles per loop. You subtract out the baseline reading of 19 cycles, leaving 123 cycles per loop. This gives you an average of 66 cycles for each of the two `digitalWrite()` functions. This translates into 4,125 nanoseconds.

The Arduino Mega 2560, however, takes a leisurely 110 cycles, on average, for each call to the `digitalWrite()`

function, which can also be stated as 6,875 nanoseconds.

Remember that substitution of the single-instruction bit-manipulation for the `digitalWrite()` function reduced the time to set or clear a single bit to two cycles or just 125 nanoseconds. Let's test that idea by replacing the `digitalWrite()` function calls with `bitSet()` and `bitClear()` macros, as shown in Listing 6-9.

Listing 6-9: Replacing the `digitalWrite()` Function Calls with Macros

```
while(timer0_millis < stop_time) {
  bitSet(PORTB, 5); // LED on
  bitClear(PORTB, 5); // LED off
  i++; // count!
}
```

For the Arduino Uno, the simple substitution increases the loop count to 690,609 loops in a single second. This corresponds with a cycle count per loop of approximately 23, or just 4 cycles over the baseline reading. Because both the `SBI` and `CBI` instructions take exactly two cycles to execute, this is exactly what you expected. It would also appear to confirm the validity of your test.

The Arduino Mega 2560 also shows a similar boost in performance after the substitution, turning in a goodly 682,028 count, remaining right around the average of 23 cycles per loop, which is only 4 cycles above the baseline reading. Remember to set and clear bit 7 of `PORTB` on the Arduino Mega 2560, although in this example it doesn't really make any difference because the blinking of the LED is too fast to see.

Applying a final optimization to this loop, let's substitute the on-again-off-again commands with a single bit-toggle operation. See Listing 6-10.

Listing 6-10: Using the Bit-Toggling Capabilities of the AVR to Increase Performance

```
while(timer0_millis < stop_time) {
  bitSet(PINB, 5); // toggle LED
  i++; // count!
}
```

This loop runs 756,381 times in a second, which corresponds to 21 clock cycles per loop. Once the baseline overhead is removed, you can see that only two cycles were needed to toggle the LED *really quickly*, and that lines up exactly with your understanding of the `SBI` instruction, which needs only two cycles to do its job.

As Fast As Possible

It's not especially useful to blink an LED at this ridiculously high rate, but it would be nice in lots of other applications to send and receive data as fast as possible. Let's determine the highest toggle rate you can get on the I/O pin using software alone. To do this, you must abandon the Arduino Serial Monitor as your test instrument and try something else.

Take a look at Listing 6-11. It's a simple program that looks like it ought to be really ripping along.

Listing 6-11: Toggling an I/O Pin as Fast as Possible

```
void setup() {
  bitSet(DDRB, 5); // PB5/D13 is connected to an LED
}

void loop() {
  bitSet(PINB, 5); // toggle LED
}
```

Because the Arduino is quite busy at the moment, it can no longer help you measure itself. This is where you bring in the Big Iron, a piece of test equipment specifically designed to measure high frequencies and display waveforms: the oscilloscope.

An oscilloscope generally displays a graph-like image that represents a voltage level in the Y-dimension over time, which is represented by the X-dimension. Figure 6-3 is an example screen capture from a relatively low-cost oscilloscope manufactured by Rigol, displaying the output of D13 on an Arduino Uno.

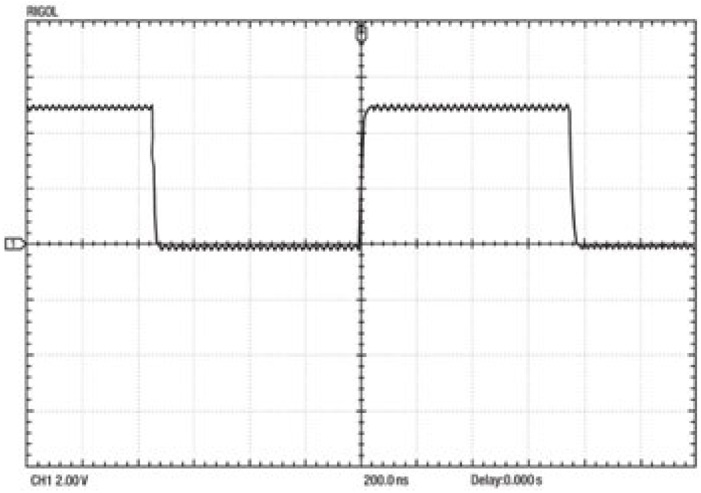

Figure 6-3: Digital signal waveform output from the sketch in Listing 6-11

In addition to displaying a picture of the waveform for you to see, the oscilloscope also provides several other useful measurements, including frequency, as well as the period of both the high-level and low-level portions of the signal. Additional readings can also be selected on most digital storage oscilloscopes (DSOs). See Figure 6-4.

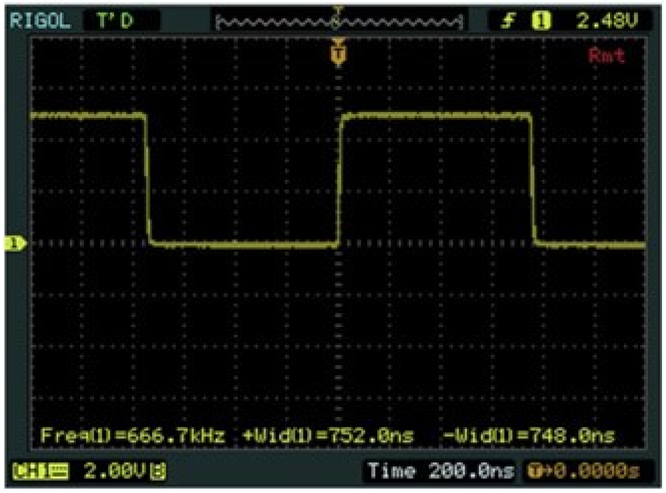

Figure 6-4: Screen capture from an oscilloscope showing additional measurements

The scope would appear to be reporting an output frequency of 666.7KHz, with an on time of 750ns and an off time of 748ns. This would lead you to believe that each pass through the `loop()` function was taking 12 cycles. What's going on here?

The answer is something that's easy to forget when programming the Arduino: the `loop()` function is being called repeatedly and ad infinitum by the hidden `main()` function. The apparent slowness of this signal is due to the calling overhead of the `loop()` function itself.

Here is a breakdown of what's happening in this sketch:

1. `main()` calls `loop()`: four cycles on the Arduino Uno, five on the Arduino Mega 2560.
2. `loop()` executes the `SBI` instruction: two cycles.
3. `loop()` returns to `main()`: four cycles on the Arduino Uno, five on the Arduino Mega 2560.
4. `main()` jumps back to the beginning of itself: two cycles.

As you can see, the problem is even worse on the Arduino Mega 2560. Due to its larger address space, it takes one cycle longer to push and pop the larger addresses (22-bit vs. 16-bit) to and from the stack when performing subroutine calls. The same is true for interrupt handling.

To eliminate this slowdown, all you have to do is wrap the `bitSet()` macro in a loop of your own devising. See Listing 6-12.

Listing 6-12: A Faster Loop for Toggling an I/O Pin

```
void loop() {
  while(1) {
    bitSet(PINB, 5); // toggle LED
  }
}
```

This sketch produces a much faster output signal; see Figure 6-5. The entirety of this sketch, after the I/O port has been properly configured as an output, consists of two machine-language instructions: `SBI` and `RJMP`. The first instruction toggles the I/O line, and the second instructions jumps back to the first instruction.

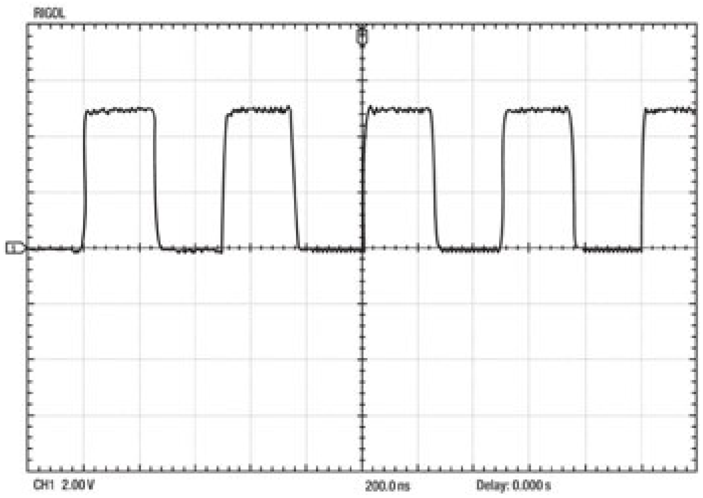

Figure 6-5: A much faster output waveform

Unfortunately, this signal isn't stable. Watching the oscilloscope display for even a short period of time reveals an occasional glitch occurring. Also, the frequency and period readings tend to jump around a little. What is causing this?

You left the millisecond timer running, and it's generating an interrupt about 1,000 times every second. During the time that it takes for the interrupt service handler to execute, the I/O line *does not toggle*. You can fix this by adding the following statement to the `setup()` function:

```
noInterrupts(); // no!
```

This function call, like so many others, is yet another macro. It turns out to be a single machine-language instruction, `CLI`, which clears the `I` (for Interrupt) bit in the CPU's status register, `SREG`. This, in turn, globally disables all interrupts from being acknowledged. Because you don't want or need interrupts in this example, this is a good solution to the problem.

Slowing It Down

So now you know that you can get the humble Arduino to output a 2MHz signal using software alone. That's pretty fast! It can't do much else at the same time, assuming you want a nice, stable signal. Let's find out how slow it can go.

Lowering the operating speed of the Arduino is the best way to lower its power requirements. If possible, you can also lower the operating voltage, which also saves power.

There are two easy ways to change the operating frequency of your Arduino. The first is to replace the quartz crystal or ceramic resonator, choosing a new component with a lower frequency. The other option is to change the operating frequency in software, using the clock prescale register (`CLKPR`). Let's try the software method first. You can always ~~mangle~~ improve your Arduino's hardware later, if you feel the need to do so.

The `CLKPR` register controls the division of the CPU clock according to the values in Table 6-3, assuming your Arduino is currently clocked by a 16MHz oscillator.

Table 6-3: Clock Prescale Register Divider Values
Open table as spreadsheet

CLKPR Value	Division Factor	Resulting Frequency	clock_prescale_set() parameter
0	1	16,000,000	`clock_div_1`
1	2	8,000,000	`clock_div_2`
2	4	4,000,000	`clock_div_4`
3	8	2,000,000	`clock_div_8`
4	16	1,000,000	`clock_div_16`
5	32	500,000	`clock_div_32`
6	64	250,000	`clock_div_64`
7	128	125,000	`clock_div_128`
8	256	62,500	`clock_div_256`

Changing the CPU's clock affects other peripherals as well, including the analog-to-digital Converter (ADC) and general-purpose I/O ports. The serial port, however, isn't affected, so you can still use common communication rates, even when your CPU is slowed down.

There is a special write sequence to update the `CLKPR` register. Bit 7, called `CLKPCE`, must be set to `1` while all the other bits are set to `0` to start the sequence. Then the actual prescaler value must be written to the `CLKPR` register within four clock cycles. To ensure that this takes place in time, it's wise to disable interrupts for the duration of the write sequence.

Listing 6-13 demonstrates two things. The first is that serial communication still runs at the proper rates. The second is that the LED is blinking very slowly, when it ought to be a high-speed blur.

Listing 6-13: Slowing the CPU Clock

```
void setup() {
  Serial.begin(9600);
  Serial.println("Normal serial communication at 9600 bps");

  pinMode(13, OUTPUT); // D13 is connected to an LED

  noInterrupts(); // disable interrupts temporarily
  CLKPR = 1<<CLKPCE; // enable clock prescaler write sequence
  CLKPR = 8; // select clock divisor of 256
  interrupts(); // re-enable interrupts
}

void loop() {
  digitalWrite(13, HIGH); // LED is on
  delay(10); // 0.01 second delay
  digitalWrite(13, LOW); // LED is off
  delay(10); // 0.01 second delay
}
```

The Arduino's CPU is now running at 62.5KHz instead of the original 16MHz. The LED is on for 2.56 seconds and then off for the same time period. The serial port still works as you would expect.

The `clock_prescale_set()` function, declared in the `avr/power.h` header file, does the same thing.

Further Power Reductions

Two more methods are available to you for reducing the power requirements of your Arduino. The first is to put the processor to sleep when it would otherwise just be waiting around for something to happen. You can save additional power by shutting down unused peripherals.

The AVR architecture allows for several depths of sleep to occur. You can think of them as ranging from lightly napping to heavy slumber. To access these sleep modes, you can use some functions from the `avr-libc` library. Add the following line to the beginning of your sketch:

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

This is a header file that is *not* normally included automatically by the Arduino compilation process, so you need to explicitly add it when you want to use the sleep functions provided by the `avr-libc` library.

The lightest form of sleep for the AVR is known as *idle mode*. In this mode, the CPU stops executing instructions, which saves a lot of power. All the remaining peripherals, however, continue to run at full speed, thereby consuming some power.

Interrupts are covered in more detail in Chapter 7. For now, you already have a free interrupt configured and running for you: the Timer0 interrupt used to trigger the millisecond counter. The sketch shown in Listing 6-14 demonstrates a method for putting the CPU to sleep while waiting for the timer to increment. This effectively puts the chip to sleep for about 99% of the time: a big savings in power.

Listing 6-14: Putting the CPU to Sleep

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

extern volatile unsigned long timer0_millis;

void setup() {
  pinMode(13, OUTPUT); // an LED is attached to D13
}

void loop() {
  while(timer0_millis < 1000) {
    set_sleep_mode(SLEEP_MODE_IDLE); // select "lightly napping"
    sleep_mode(); // go to sleep
  }
  timer0_millis = 0; // reset millisecond counter
  bitSet(PINB, 5); // toggle LED
}
```

Bonus: Digital Signal Probe

A crude *digital signal probe* for use with your Arduino can be built using only a short piece of 22-gauge wire. You can use smaller-diameter wire, but it tends to fall out of the expansion header sockets. Larger diameter wire or repurposed paper clips aren't recommended because they tend to deform the internal spring structure of the header pins, rendering them potentially unreliable in the future.

To build a probe, follow these steps:

1. Take a 6" piece of 22-gauge solid, insulated wire and strip about a quarter (1/4) of an inch of insulation from both ends. If you want to get fancy, you can solder header pins to a similar length of stranded wire of any small gauge, but only a simple wire of some sort is needed for this experiment.

2. Compile and upload the example Blink sketch to your Arduino:

   - File > Examples > 1.Basics > Blink
   - Sketch > Verify / Compile

- File > Upload to I/O Board

3. Verify that the LED is indeed blinking properly.
4. Using the code editor, change all references to pin 13 to some other pin number, such as 2. The code should now look like this:

```
void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(2, OUTPUT);
}

void loop() {
  digitalWrite(2, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(2, LOW);    // set the LED off
  delay(1000);              // wait for a second
}
```

5. Save this sketch with a different name; don't overwrite your example sketches!
6. Compile and upload the sketch to your Arduino. Verify that the LED is no longer blinking.
7. Install one end of the wire in the expansion header pin socket marked 13.
8. Install the other end of the wire in the expansion header pin socket marked 2.
9. Verify that the LED is now blinking.

Why does this work? Remember that you changed all the references to pin 13 to pin 2 (or some other convenient pin of your choosing). Because D13 was *not* configured as an output in the `setup()` function, it remains in its default state as an *input*. As an input, it doesn't mind at all if another output is connected to it. The LED and its associated current-limiting resistor remain electrically attached to pin 13, but pin 13 no longer drives it. Any signal you now connect to D13 will show you whether it's high (LED on) or low (LED off).

Try probing the other pins in the expansion headers. Because the modified Blink example sketch didn't specifically configure any other pins as outputs, they normally show up as low (the LED remains unlit). Pay special attention to the relative brightness of the LED as you probe the 5V line and the 3.3V line. Which one is brighter? Why?

More extra credit questions: Why does probing the RESET line cause the Arduino to reset? Why is the RX line always high, even when nothing is being received on the serial port?

You can always use the programmable LED as a digital signal probe as long as pin D13 is left as an input.

Avoid probing the Vin pin if your Arduino is powered by an external power supply of greater than 6V. The LED won't mind, but D13 *certainly* will.

Previous Next