Chapter 6 - Optimizations

Arduino Internals

by Dale Wheat

Apress © 2011 *Citation*

Recommend? yes no

Previous Next

Shrink Blink

Let's see what you can do with your old friend, Blink, found in File ➤ Examples ➤ 1.Basics ➤ Blink. Compiled for the Arduino Uno using Arduino 0022, the resulting binary sketch size is 1,018 bytes (of a 32,256-byte maximum). The same sketch, when compiled for the Arduino Mega 2560, results in a binary sketch size of 1,588 bytes (of a 258,048 byte maximum). The extra size can be attributed to the initialization of the larger number of available peripherals aboard the ATmega2560 chip along with the additional low-level functions needed to talk to them.

How Blink Works

When you look at it a certain way, the Blink example sketch only does three things:

1. Configure digital pin 13 as an output.
2. Change the state of the output pin.
3. Pause for a *human-perceptible* amount of time, and then repeat from step 2.

Because Blink is an example sketch in the 1.Basics category, it makes no attempt at optimization. Its goal is be easy for both the Arduino compiler and the beginning Arduino programmer to understand. It says what it does and does what it says, which is always a good thing. In furtherance of this plainness and simplicity, Blink spells out the discrete commands to turn on the LED, wait, turn off the LED, wait, repeat.

Turning the LED on and off can certainly be accomplished with the code as presented in the Blink sketch, but it can also be looked at, in a more abstract sense, as *toggling* the LED output. If it's on, turn it off, and vice versa. You use this concept shortly to cut the size of the `loop()` function in half.

Measuring Space-Saving Optimizations

Edit the Blink example sketch using the Arduino code editor. In the `setup()` function, add two forward slashes (`//`) to the left of the `pinMode()` function call. In the Arduino IDE, this modification turns the color of the `pinMode()` statement from orange (executable code) to gray (program comments).

Everything following the *single-line-comment* marker (`//`) until the end of the line is considered a program comment and ignored by the compiler. You could have deleted the entire line, but you might want it back shortly for more experiments, so just comment it out for now. This is a useful technique for making small changes to code to compare behaviors, before and after.

Compile the code to see what happens. On the Arduino Uno, the binary sketch size drops from 1,018 bytes to 934 bytes, just by commenting out the `pinMode()` function call in the `setup()` function: a space savings of 84 bytes. For the Arduino Mega 2560, the resulting binary sketch size is now 1,504 bytes, showing an identical reduction of 84 bytes. This is explained by the omission of the `pinMode()` function call, containing 8 bytes, as well as the code for the function itself, containing 76 bytes. Because there was no other reference to the `pinMode()` function in the program, the compiler wisely omitted the instructions that perform the actual task from the final binary sketch. This form of space-saving optimization has already been done for you, but there is more that you can and should do to get the most out of the limited program memory space.

Code Analysis

The call to the `pinMode()` function required a total of eight bytes of program memory. The `CALL` instruction itself takes four bytes, or two instruction words. Remember that all AVR machine-language instructions are a multiple of 16 bits in length, and each byte contains 8 bits, so it always takes at least 2 bytes of program memory for each machine-language instruction. The subroutine `CALL` instruction is two program words long. The shorter Relative Call (`RCALL`) instruction could have been used here, because the relative distance from the calling program to the subroutine happens to be within the 2KB program word limit of the `RCALL` instruction. In a larger program, however, the `CALL` instruction may be required. There is no simple way to tell this compiler to use the shortest instruction possible in every case.

The other four bytes in the `pinMode()` function call are taken up by the two eight-bit parameters that are passed to the function: the pin number (`13`) and the direction (`OUTPUT`). Each parameter is loaded into a predefined register before the actual function call is performed. Each of these *constant value* parameters is loaded into a register using the Load Immediate (`LDI`) instruction, which is one program word (two bytes) long. The compiler knows if you're using constants (as opposed to variables) so it can easily decide to take this shortcut; this is another optimization provided by the compiler that is already working for you.

You may be thinking at this point that you have completely disabled the humble Blink sketch by failing to configure the one output that it needs to perform its simple function. This isn't entirely true. Try compiling and uploading the sketch to your Arduino, minus the `pinMode()` function call, and observe what happens.

In a brightly lit room or sunlit area, you most likely won't see anything happening. However, this isn't precisely the case. Try observing your Arduino in a darkened room, or wait for the sun to go down. You see that the LED attached to D13 is indeed blinking, albeit *very* dimly. This is because writing a `HIGH` or a `LOW` value to a digital pin that is configured as an *input* causes the built-in pull-up resistor to be activated or deactivated. Even though you didn't explicitly configure D13 as an input, this is the *default state* of all general-purpose I/O pins after reset. The LED is being powered by the very small amount of current flowing from V_{CC} through the pull-up resistor, which is typically only a small fraction of a milliamp.

Although a dimmer LED saves power, it's not saving you any program space, per se. Let's replace the functionality of the `pinMode()` function so that D13 can be configured to be the output it always dreamed of being.

Life Without pinMode()

Setting the direction of an I/O pin is a very straightforward procedure on the Atmel AVR. Recall from Chapter 3 that each of the general-purpose I/O pins belongs to a group called a *port*, and each of the bits within a given I/O port can be either an input or an output. The direction of each of the individual pins is controlled by the bits within the *data direction register* (`DDRx`) associated with the I/O port.

Arduino hides all this complexity from you and numbers the digital pins starting at zero. The analog pins are likewise numbered. Arduino programmers don't need to worry about ports and addresses and bit positions. You just look at the Arduino I/O Board, find the pin you want to use, and use the number printed on the printed circuit board (PCB). It's quite handy.

Because you know that D13 is PB5 (Port B, bit 5) on the Arduino Uno and PB7 (Port B, bit 7) on the Arduino Mega 2560, you can directly set the single bit in Port B's data direction register (`DDRB`, bit 5 or 7, depending) and get the same result as the `pinMode()` function call, without all the program overhead.

If you have an Arduino Uno or other ATmega328-based Arduino, add the statement `bitSet(DDRB, 5);` to the `setup()` function of the example Blink sketch. If you're using an Arduino Mega 2560, add `bitSet(DDRB, 7);` to the `setup()` function. For the Arduino Uno, your `setup()` function should now look like Listing 6-1.

Listing 6-1: Directly Setting the Direction Bit of an I/O Port Without Using the `pinMode()` Function

```
void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  //pinMode(13, OUTPUT);
  bitSet(DDRB, 5); // D13 (PB5) is now an output
}
```

Compile and upload to your Arduino. You should have a properly blinking LED again, but the binary sketch size should be only 936 bytes. The `bitSet(DDRB, 5);` statement is reduced to a single instruction (two bytes) by the compiler—and it works.

> **Tip** Because you've replaced a *somewhat* human-readable program statement, `pinMode(13, OUTPUT)`, with complete bafflegab and abracadabra, it's more important than ever to use clear, explanatory comments in your code.

Abbr. & Shrtcts

There are several things you need to know and understand about the little code modification just discussed. First, the original designers of the C programming language were pretty ~~lazy~~ *efficient* when it came to typing. If you've ever spent much time working with a Teletype ASR-33, you may begin to understand why. It wasn't exactly a pleasant experience. The designers came up with many shortcuts to help reduce the amount of typing required to perform most functions.

The second thing you should know about the replacement configuration statement is that it isn't truly a function call, even though it looks like one. The `bitSet()` function is really just a macro definition, defined in the `wiring.h` header file that is included into your sketch, along with the other Arduino-supplied code, before being compiled into the final binary sketch, ready to upload. This function takes two parameters, `value` and `bit`. These parameters are then substituted into a prewritten mathematical formula, `((value |= (1UL << (bit)))`. When the substitution is complete, the resulting statement effectively reads as `DDRB |= 1<<5`, which means "Set bit 5 in DDRB."

The `wiring.c` and `wiring.h` files also provide several other bit-manipulation macros that often can be reduced to a single instruction, including `bitClear()`, `bitRead()`, and `bitWrite()`. See the Arduino Reference page (choose Help ➤ Reference in the Arduino software) for more information about these functions.

The `DDRB |= 1<<5;` statement is a good example of the possible compactness of the C language. Technically, this is an *assignment* statement. The `|=` operator is called an *augmented* (or *compound*) *assignment operator*. It's *augmented* because it's a composite of the assignment operator `=` and the bitwise, inclusive-or function, represented by the vertical bar character `|`. The statement could have been written `DDRB = DDRB | (1<<5);`. Spelled out, this translates as "Read the value that is currently in the `DDRB` register, bitwise-or this value with the constant value 1<<5, and then store the result back in the `DDRB` register."

The term `DDRB` is used twice in this statement: once as a source (as one of the *operands*) and once as a destination. On the left side of the equals sign (`=`) is the destination, or *lvalue*. On the right side is the source, or *rvalue*. In either case, the `DDRB` term represents an address in I/O space. It's defined in the `avr/io.h` header file, which is included automatically by the Arduino software. Actually, the `avr/io.h` header file is just a generic header file that looks at which part has been defined as being used: for example, `__AVR_ATmega2560__` is a symbol that is defined in the code when the Arduino Mega 2560 board is selected by choosing Tools ➤ Board.

With all this predefinition of terms having already been put in place, you can refer to `DDRB` like any other variable in the program. It can be read, written, or modified just as any other numeric value can be.

Binary Notation

The `1<<5` notation needs some explaining. This is an example of the *left-handed bitwise shift* operator at work. The `1` on the left side is a constant, representing a single, digital 1 in the least significant binary digit. The `5` tells the operator how many times to shift the bits leftward. The corresponding *right-handed bitwise shift operator* is, you guessed it, written `>>`. The 5 is used because you're interested in bit 5 of the `DDRB` register, which corresponds with the fifth pin of the Port B general-purpose I/O port, PB5.

Shifting a bit to the left in a binary number is the same as multiplying it by two. If you shift a decimal base 10) number to the left (and add a zero to the now-empty space on the right), it's the same as multiplying that number by 10. Binary numbers are exactly the same as decimal numbers, except they're base 2 instead of base 10. Inside the computer chip, all numbers (and all data, really) are represented by binary numbers.

So instead of `1<<5`, you could use decimal `32` (`DDRB = 32;`), hexadecimal `0x20`, octal `040`, or binary `0b00100000`. Unfortunately, none of these other (perfectly accurate) representations readily communicate to the reader that you're interested in bit 5 here, and only bit 5. The binary representation `0b00100000` comes close but is more error-prone and takes much longer to type. The compiler understands them all, and they end up as the right bits in the right places after all the dust settles.

Couldn't you write `1<<5` directly into the `DDRB` register and be done? In this trivial example, yes. You don't care about the values of the other bits in `DDRB`, and writing a single bit (bit 5) clears all the other bits as an unintentional byproduct. Because this is a trivial example, it doesn't matter. Setting or not setting any of the other bits in that particular register doesn't affect the desired outcome.

Using the `bitSet()` macro instead of either the direct assignment statement or the augmented assignment statement makes the source code clearer in its intent, without incurring any sort of size penalty or code bloat. The compiler reduces it to the smallest code possible. When it's within the range of the *bit-manipulation* instructions `SBI` and `CBI`, it uses those. Otherwise, it cobbles together something brief.

If the `1<<5` notation just isn't your thing, you can use the Arduino-provided `bit()` macro, which does the same thing. For example, `1<<5` can be written `bit(5)`.

Further Analysis

As just mentioned, you could simply write the desired value directly into the `DDRB` register, treating it just like any other variable. You'd do so with a statement such as `DDRB = (1<<5);`. This works because, in this example, you don't especially care if you clobber all the other bits in the `DDRB` register. You only care if the fifth bit gets set.

One interesting note about writing a value directly to the register (instead of the read/modify/write procedure used with the augmented assignment operator) is that the binary sketch size is now 938 bytes, or 2 bytes *bigger* than the preferred method, using the `bitSet()` macro. This is because the compiler uses two machine-language instructions to perform this direct write: one to load a constant value into a register, and another to write that value to the I/O port. There is no Load Immediate (`LDI`) instruction that directly addresses the I/O space. The binary sketch size on the Arduino Mega 2560, likewise, is 1,506 bytes, one machine-language instruction longer than the more compact version.

How, exactly, does the compiler perform this bit of magic? The answer lies in the *highly optimized* AVR-specific libraries that are provided with the Arduino software. They're part of the AVR port of the GNU Compiler Collection (`avr-gcc`) and collected together into the `avr-libc` library. The compiler determines that only a single bit is being manipulated and uses the exact machine-language instruction that was created specifically for this job: Set Bit in IO Register (`SBI`). The `SBI` instruction performs the complete read/modify/write process in two clock cycles but only uses a single word of program memory to do it.

That's not all you can optimize in terms of space in this example. Let's further use your understanding of the internals of the Atmel AVR to optimize the blinking of the LED.

Easy Toggling

As discussed previously regarding the example sketch Blink, the LED is explicitly turned on and then explicitly turned off again, using two different forms of the `digitalWrite()` function. The first function call turns on the LED by writing a `HIGH` level to pin D13, and the second function call turns off the LED by writing a `LOW` level to the same pin.

Also mentioned previously is that this on-again, off-again cycle can be more abstractly viewed as a *toggling* of the output pin. Luckily, the AVR knows how to toggle the output of its pins using a special write sequence, using an otherwise unused address.

The Input Pins Address (`PINx`) is normally only ever *read* by the processor, to determine the actual logic levels present on the general-purpose I/O pins.

> **Tip** Don't make the mistake of trying to *read* the `PORTx` register. This is the output port data register only and only indicates what value, if any, was last written to the output port. To read inputs from the outside world, you *must* read from the `PINx` register.

When the AVR design engineers decided to add *pin-toggling* capabilities to the newer AVR chips, they didn't have any extra I/O addresses to use, so they cleverly reused an address that had only been used previously for inputs. Writing a `1` to a bit in the `PINx` register *inverts* or toggles the output level of the corresponding output pin for that port. Note that this *doesn't work* on older AVR devices, such as the ATmega8, ATmega16, and ATmega32. When in doubt, check the datasheets.

To toggle the LED, you just need to write a single `1` to the correct bit position in the `PINB` register. You don't need to preserve any of the other bits in this register, because they technically don't exist.

Replacing both of the `digitalWrite()` function calls with `bitSet(PINB, 5);` results in a binary sketch size of only 658 bytes for the Arduino Uno and 870 bytes for the Arduino Mega 2560. Please remember to use `7` instead of `5` when using the Arduino Mega 2560, because D13 is mapped to PB7 and not PB5.

This results in a slimming of 278 more bytes. The `digitalWrite()` function is handy, but it comes with a price.

Further Reduction

A closer look at the resulting code reveals a duplication of efforts. Whereas the original version of Blink's `loop()` function contained four program statements to perform the discrete steps involved, the new version needs only two: the first to change the LED state, and the second to wait a short period of time. Because the `loop()` function automatically repeats itself, it turns on the LED after one pass and then turns off the LED on the next pass.

Eliminating the now-redundant lines of code produces a binary sketch size of 644 bytes, a further saving of 14 bytes of program memory.

Wasting Time More Efficiently

What's left to optimize in this sketch? The only function calls you haven't modified so far are the `delay()` function calls that produce the human-perceptible delay you need in order to be able to detect the blinking of the LEDs with the naked eye.

Just to establish a lower boundary, comment-out the remaining `delay()` function call within the `loop()` function. When compiled, this produces a binary sketch size of 454 bytes. This is the lower boundary for the binary sketch size because no code for the delays has been included at all, and it will take *something* to replace that functionality. Let's look at a couple of alternatives.

First, let's try a simple `for()` loop to kill some time. A `for()` loop is a special kind of *control* (that is, looping) structure. A `for()` loop contains a place for some initialization code, a place for a test to see if the loop should continue, and a place for some code to perform every loop. These code snippets are included as the parameters of the `for()` statement and are separated by semicolons.

Add the program statement `for(int i = 0; i < 32000; i++);` somewhere inside the `loop()` function. It really doesn't matter where you place this statement in such a simple example program.

Suspiciously, after compilation, the binary sketch size is *still* 454 bytes. Uploading the sketch to your Arduino produces perhaps puzzling results. The LED comes on and appears to stay on, but it looks a little dimmer than before. In reality, it's blinking away madly, more than half a million times per second. This is much too fast for the human eye to detect and looks like a blur. What happened?

Optimization happened, that's what. The compiler spied your empty `for()` loop, which contained no executable statements. The compiler then decided that the `for()` loop wasn't needed in the final program. Therefore, the compiler completely omitted the `for()` loop, thinking it had done a Good Thing. Remember, the compiler takes *optimization* very seriously and doesn't understand you wanting to waste time like that.

Add the keyword `volatile` just before the integer type declaration `int` to prevent this optimization. This produces a 498-byte binary sketch (a good sign, because it's bigger) and, better yet, a rapidly yet visibly flashing LED. The `volatile` keyword instructs the compiler to make no assumptions about this particular variable, including assumptions that might have lead to optimizations.

The parameter passed to the Arduino's built-in `delay()` function is the desired delay time expressed in milliseconds. The parameter selected in your `for()` loop is just a big number and doesn't directly correspond to milliseconds. By giving up this accuracy, you save 146 bytes of program memory. This is figured by offsetting the 190 bytes you saved by omitting the `delay()` function by the 44 bytes taken up by the empty `for()` loop.

Using Lower-Level Code

Another approach to adding a perceptible delay is to use the already-ticking Timer0 and its associated interrupt handler, `TIMER0_OVF_vect`, which increments an unsigned, long integer value called `timer0_millis`.

> **Caution** If you ~~mess with~~ improve the built-in Arduino code that uses the timers, you risk losing the proper function of the built-in timing functions, such as `delay()`, `millis()`, and so on.

To refer to this new variable in your sketch, you need to add an external variable declaration. The variable `timer0_millis` is *defined* in the `wiring.c` source code, but the compiler forgets all about it when it has completed the compilation of that source file and moved on to the next file to be compiled. You remind the compiler about this variable by adding the external variable *declaration* `extern volatile unsigned long timer0_millis;` somewhere in your sketch but *outside* either of your two functions, `setup()` and `loop()`. This declaration needs to be placed somewhere in your sketch *before* the first reference is made to the variable. In other words, it has to be defined before it can be used. Also note that this declaration must match the definition in the other file exactly. You can't say it's an `int` in one place and a `byte` in another, or the compiler will become confused.

Now replace the empty `for()` loop with the following two program statements inside the `loop()` function:

```
while(timer0_millis < 1000); // wait 1 second
timer0_millis = 0; // reset millisecond timer
```

The first statement is an empty `while()` loop that waits for `timer0_millis` to equal or exceed the value 1,000. Because `timer0_millis` should increment at the rate of one count per millisecond, this loop, excluding any code overhead, should take approximately one second to complete. It won't be *exactly* one second, but it will be very close. When the `while()` condition is satisfied, execution continues with the next program statement, which resets the millisecond timer count.

Do take note of the fact that this counter-resetting method completely nullifies the intended operation of the `millis()` function, which is to report the cumulative number of milliseconds since the Arduino sketch began to execute. Because this is a trivial example, and all you want is a one-second delay, you can take this liberty.

Compile and upload this sketch to your Arduino. This alternative delay results in a binary sketch size of 496 bytes on the Arduino Uno and 708 bytes on the Arduino Mega 2560, even smaller than your empty `for()` loop delay method. Had the lower-level Arduino code not already initialized Timer0 and provided an interrupt routine, you would be required to do so yourself, and the code would be correspondingly larger. As it is, you have creatively used the existing software to do your bidding, at the expense of the original intent of the `millis()` function.

In summary, you've replaced three function calls and cut the binary sketch size by more than half. In truth, there's still more to cut, because a lot of unused functionality is waiting to be exploited in a plain-vanilla Arduino sketch.

Previous Next