Member-only story

MICROCONTROLLERS

AVR basics: reading and writing GPIO pins

Putting a microcontroller to use means taking control of its I/O pins. Here's how you do it.

Mansfield-Devine · Follow

Published in Machina Speculatrix · 6 min read · Mar 12, 2025

Once you've set up a pin, or a whole port's worth of pins, as inputs or outputs on your AVR microcontroller (as covered in my previous AVR piece), it's time to start writing and reading values.

Writing values

Let's assume you've set the pins on Port B of an Atmel Atmega 328P as outputs. The next step is to actually set the pins high or low. As with setting the pin direction, you set values (0 or 1) for each pin by writing a value to an eight-bit register. For outputs we use the PORTx registers. So for Port B, that would be `PORTB`.

The pins in each port are numbered 0–7 and we set them high or low by writing a 1 or 0 to the associated bit in the register. For example, to set pin 3 high, we need to make sure that the '3' bit of `PORTB` is a one. The '3' bit is actually the fourth from the right. The bits are numbered from 0 going from right (least significant bit, LSB) to left (most significant bit, MSB). So the '3' bit is actually represented by the decimal value 8 because in binary that's

`00001000`. So you could just write that to `PORTB`. But as we learned in the article on writing to the Data Direction Register, there's a better way to do this — through the use of ORing a value with the existing register value.

And as usual, to help us with this there are some handy macros we can use to refer to the pins. In fact, you can take your pick. For pin 3 on Port B you can use `PORTB3` or `PB3`. (You could also, technically, use `PINB3` — these macros all have the same value. But the PINxx ones are intended for use with reading inputs). My preference is to use `PB3`. There is an argument for using `PORTB3` as it makes it explicit you're setting outputs, but to me it looks confusing — as though we're setting a port rather than a pin. So here's an example:

```
DDRB |= (1 << DDB3); // set pin 3 of Port B as output
PORTB |= (1 << PB3); // set pin 3 of Port B high
PORTB &= ~(1 << PB3); // set pin 3 of Port B low
PORTB |= (1 << PORTB3); // set pin 3 high again
```

If you're not familiar with why we use the shift left operator (<<) and the ORing (|) and ANDing (&), make sure you read the post on setting the Data Direction Register.

There's another useful operation too — the ability to toggle a bit using the Exclusive-OR (XOR) operator, represented by the caret symbol.

```
PORTB ^= (1 << PB3);   // toggles the state of the bit
```

Let's walk through how this works. As before, the `1 << PB3` operation creates the binary value we need: `00001000`. We XOR this with the current value of the register. Let's think about all the bits in our value `(1 << PB3)` *except for* the bit we're trying to set (ie, `PB3`). All these other bits in the set value are zeroes. If a corresponding bit in the register is a 1 then the result of using XOR is a 1, so the bit is unchanged. If the bit in the register is a 0 then XORing that with 0 is also 0, so again it's unchanged. So that's good — none of the bits other than the one we're addressing is affected.

Now let's look at the bit we're trying to toggle. It's a 1 in our set value `(1 << PB3)`. If the corresponding bit in the register is also a 1 then the result of 1 XOR 1 is 0, so the bit is flipped. If the bit in the register is a 0, then the result of 0 XOR 1 is 1, so again it's flipped. Toggle achieved.

Reading values

Reading a value is slightly different, partly because we need to do something with the value we find. We use the PINx registers for reading, so for Port B this would be `PINB`.

Essentially, the method is to AND the value in the input register with a value that represents the pin we're testing.

Let's assume we want to read the input for pin 4 on Port B. That pin is represented by the macros `PINB4` or `PB4`, take your pick. We'll use the former version to be explicit we're reading inputs.

First, you'll need to have set up pin 4 as an input by setting its bit in the Data Direction Register to 0:

```
DDRB &= ~(1 << PINB4);
```

To find out if pin 4 is high or low, we need to AND the current value of the register with `00001000`. To get the latter value we use: `1 << PINB4`. So the method is:

```
PINB & (1 << PINB4)
```

This will tell us if pin 4 of the input register is high or low. But think for a moment about what the actual result of that operation is. If pin 4 is low, the

would be 32 and so on.)

This is important because, like I said, you need to do something with the result of this operation. It's possible that all you need to do is test whether the result is 0 (meaning 'low') or any positive value (meaning 'high'), which is easy. But if you do anything more complex, be careful. For example, let's say you assign the result to a variable.

```
uint8_t pinValue = PINB & (1 << PINB4);
```

Remember that the value of pinValue won't be a 1 or 0. It'll be 0 or 16. If you want a 1 or 0 you could use:

```
uint8_t pinValue = (PINB & (1 << PINB4)) >> PINB4;
```

Or you could use the result directly — perhaps in an 'if' statement:

```
if ( (PINB & (1 << PINB4)) == (1 << PINB4) ) {
    // pin is high
} else {
    // pin is low
}
```

Hiding the details

Maybe all this business with shifting bits around looks clumsy to you. Personally, I think it's usefully explicit — I see `1 << PINB4` and the meaning "move a 1 into the PINB4 position" is clear. And when you start getting into other registers, such as those controlling the SPI and I2C (two wire) interfaces, the use of macro labels with these operations helps clarify what the program is doing.

However, some people like to abstract away some of this stuff inside macros and functions. In the Atmel libraries, the `avr/sfr_defs.h` file (which is automatically included from `avr/io.h` which in turn you tend to include in pretty much all your programs) includes some handy macros for setting 'special function registers' (hence the 'sfr' abbreviation in the following examples). These include:

```
bit_is_set(sfr, bit)
bit_is_clear(sfr, bit)
loop_until_bit_is_set(sfr, bit)
loop_until_bit_is_clear(sfr, bit)
```

So to use them you just pass the register and the bit for the relevant pin. The first two return a 'non-zero' result if the condition you're testing for is true and 0 if false. The other two are fairly self-explanatory.

Should you want to create your own functions there is one gotcha. The macro labels for the registers are often not the addresses of the registers themselves but *pointers* to those addresses. For example, you might be tempted to write a function something like this:

```
void setBit (uint8_t register, uint8_t pin) {
    register |= (1 << pin); // *** THIS DOESN'T WORK
}

setBit(PORTB, PB3);
```

That's not going to work. Instead you need to use a pointer in the function and pass the register by reference. You also need to use 'volatile' to prevent a compiler error.

```
void setBit (volatile uint8_t * register, uint8_t pin) {
    *register |= (1 << pin);
}
```

```
setBit(&PORTB, PB3);
```

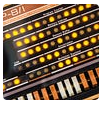
You can find all the AVR-related articles here.

I've created a GitHub repo for supporting files to accompany this AVR series of articles. You can find it here: https://github.com/mspeculatrix/AVR_8bit_Basics/

Steve Mansfield-Devine is a freelance writer and photographer. You can find photography portfolio at Zolachrome, buy his books and e-books, or follow him on Bluesky or Mastodon.

You can also buy Steve a coffee. He'd like that.

Microcontrollers | Avr | Programming | Technology | Embedded Systems



Published in Machina Speculatrix

21 Followers · Last published 5 hours ago

Follow

Electronics, robotics, home automation, hacking and more. The lab notebook of an amateur meddler who likes playing with things until they work—or blow up.

Written by Mansfield-Devine

38 Followers · 2 Following

Follow

Freelance writer & photographer, tech journalist and electronics botherer.

Responses (2)

Alex Mylnikov

What are your thoughts?

gBird
2 days ago

It's been awhile since I programmed at this level. That's for the brain stirring! 🧠🥄

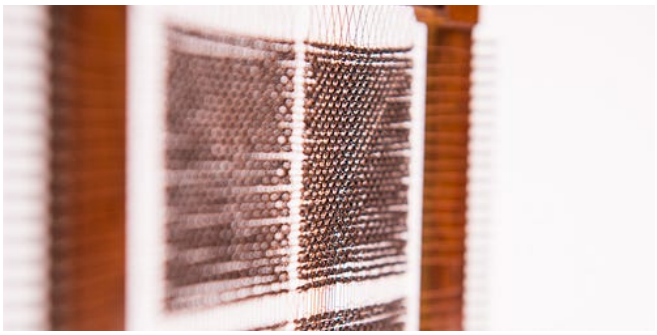
Reply

Nadeem Chuhan
3 days ago

Interesting article

Reply

More from Mansfield-Devine and Machina Speculatrix

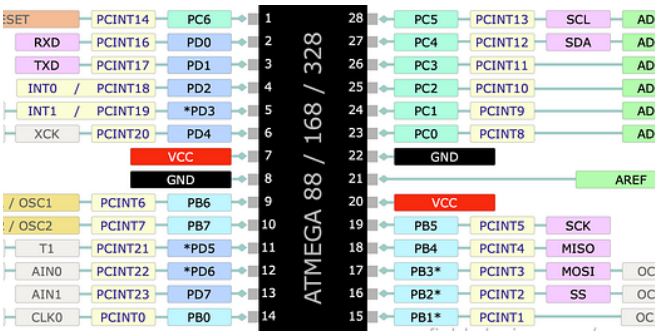


In Machina Speculatrix by Mansfield-Devine

Retro to the core

Today, computer memory comes buried inside anonymous chips. But there was a tim...

Mar 4 412 10

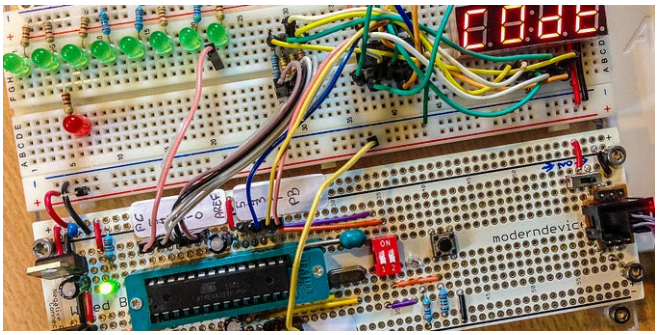


In Machina Speculatrix by Mansfield-Devine

AVR basics: registers, defines, ports & pins

The fundamental operations of a microcontroller are reading from and writing...

Mar 10 6



In Machina Speculatrix by Mansfield-Devine

The joy of AVR 8-bit microcontrollers

Microcontrollers are becoming insanely fast and capable. But there's something about th...

In Machina Speculatrix by Mansfield-Devine

About me

You might be wondering, who the hell is this? Let me explain.

Mar 1 100 Mar 3 24 2

See all from Mansfield-Devine See all from Machina Speculatrix

Recommended from Medium

In Wall Street Gradient by Shubhransh Rai

This is How USA will Zero the National Debt

US Economy—21 Trillion, US National Debt—36 Trillion, the math isn't mathing here.

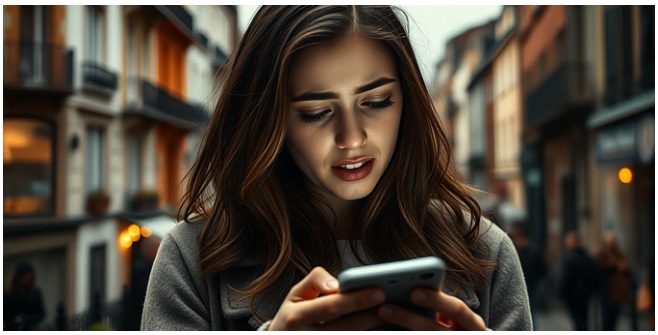
Feb 25 1.2K 45

In The Preamble by K.W. Hampton, PhD, MPA

What Google and Meta's Leaked Internal Memos Reveal About...

They're scared as sh*t!

Mar 3 3.5K 118



Mallick Speaks

When Rent Becomes a Creepy Bargain: The Sex-for-Rent Surge i...

Hey, imagine you're a 20-something international student, fresh off the plane in...

Mar 4 745 34

Giorgio Provinciali

The Massacre Continues – Live from the Kursk Front

Live report

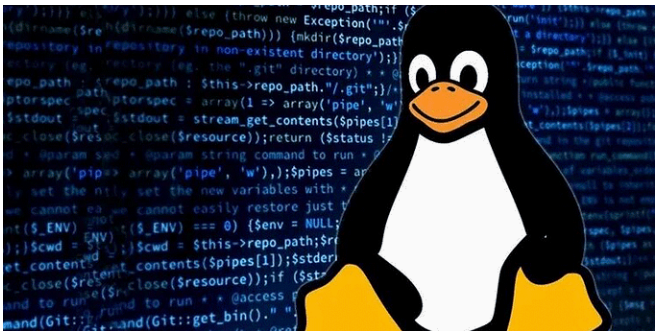
4d ago 6.6K 23

In Code Like A Girl by Nidhi Jain 👩‍💻

9 Lessons from a Principal Engineer That Made Me a Better...

The small but important missing pieces that no one talks about

Mar 10 1K 23



In InfoSec Write-ups by Frost

Powerful Linux Tricks That Will Change Your Life

If you've ever worked in a Linux environment, you know how powerful and versatile it can...

Mar 12 120

See more recommendations