AVR basics: reading and writing GPIO pins

Machina 29/04/2017 4 Comments

Once you've set up a pin, or a whole port's worth of pins, as inputs or outputs, it's time to start writing and reading values.

> `This series is part of my learning process and I hope it will help others who, like me, are embarking on projects such as programming AVR chips. The way I learn things is to write about them as I go. Feel free to correct my mistakes. FYI, I program for AVR ATMEGAs using AtmelStudio 7 on Windows 10. I use the Atmel-ICE programmer to get the code on to the chip. If your toolchain is different you may need to take that into account. The macros mentioned here come from the standard Atmel libraries.`

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 to left. So the '3' bit is actually represented by the decimal value 8 because in binary that's 0b00001000. So you could just write that to PORTB. But as we learned in the post 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: 0b00001000. 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 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 0b00001000. 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 result is 0. That's simple enough. But if pin 4 is high, the result won't be 1 – it's a 1 shifted 'PINB4' times to the left, or 0b00010000. In decimal, that's 16. (If you were doing a similar operation for pin 3 the result would be 8, for pin 5 it 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 meaning "move a 1 into the PINB4 position", so the code is explicit and 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);
```

AVR Basics   Atmega, AVR, C, GPIO, microcontroller, programming

Related posts

» Zolatron 64: Writing an OS for a 6502 homebrew computer
» Zolatron 64: 6502 bare metal random
» Zolatron 64: being persistent
» Zolatron 64: The start of an operating system
» Elliott 405 cheat sheet