ATTiny Port Manipulation (Part 1): PinMode() and DigitalWrite()

by myless8

First off, this is written with Arduino and the Arduino IDE in mind. I will be referencing various Arduino sources and datasheets for AVR chips.

Secondly, what does this really mean?

Manipulation of port registers allow for lower-level and faster manipulation of the i/o pins of Arduino and on any ATTiny microchip.

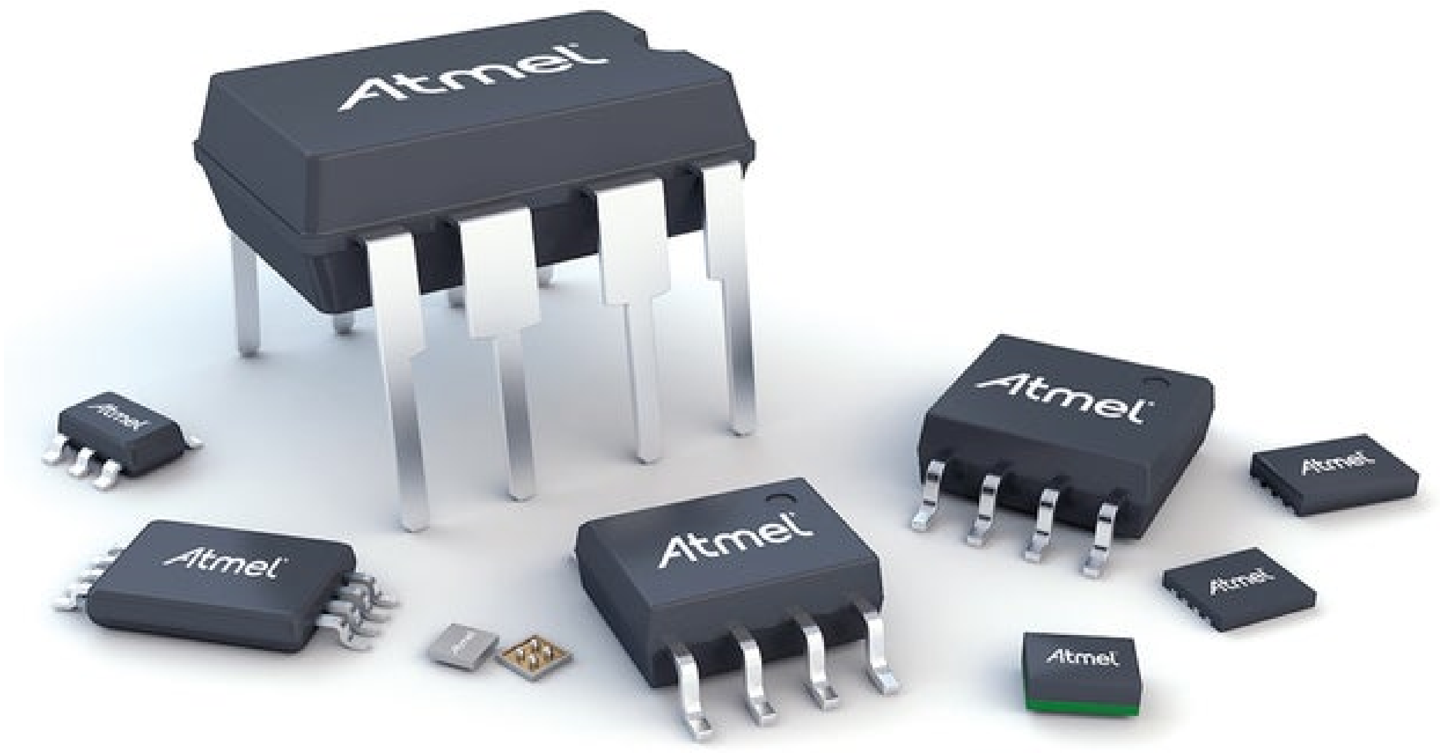
Why would you want to do this?

The Arduino IDE and its coding is extremely user friendly. That is, it makes programming easier with the use of functions to accomplish tasks. In this case, pinMode() and digitalWrite(). These write commands to a certain pin. What you do not see is *how* the functions do this.

The short version is that they go through their own lines of code to do what you want it to. And this takes time. In Micro controller time, it can be pretty significant if you are working on a particularly time sensitive project. Direct port manipulation is roughly 40 times faster at writing a command to a pin than the digitalWrite() function. Additionally, port manipulation saves on memory in the chip as it ends up being fewer lines of code being programmed into the controller/chip. Although, if this is your main reason for learning port manipulation, many would simply say that you are using the wrong chip.

If this is so beneficial, why doesn’t everyone do this?

Most will eventually wander into this area in their tinkering careers. But the biggest problem is, that this can be very intimidating and confusing to anyone new without someone sitting next to them explaining it. It gets into binary mathematics (not that you really need to understand that part), new commands you will not have seen up to this point, and it forces you to really learn to read understand a datasheet for a new microchip.



Step 1: Getting Familiar With a Datasheet

The following is a link to the datasheet for the ATTiny85. This is a chip I am familiar with.

ATTiny85 Datasheet

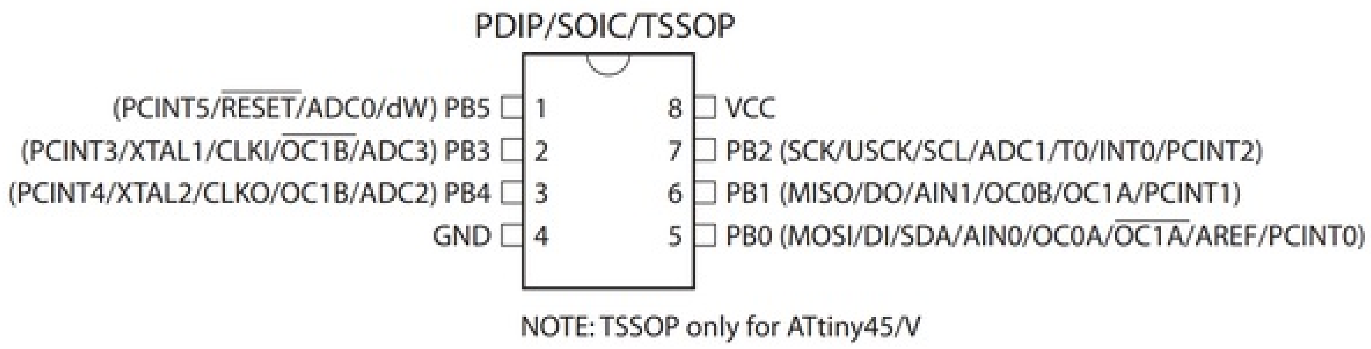
The first thing to know is the difference between Pins and Ports. A Pin is the physical metal part coming from the chip and the number ascribed to it given its location that chip.

A Port refers to a Pin given its function within a register.

This brings us to the term you have to learn, register. A register is a location in a store of data, used for a specific purpose and with quick access time. In this case, it's a binary series that you can write 1s or 0s to in order to assign different functionalities to a port.

If you look at the first figure on page 2 of the link above, you will see the typical ATTiny85 version that would be used by the average tinkerer. You can see that it has 8 pins, most of which have multiple labels attached to them.

For example, Pin 2 is labelled as PCINT3, CLKI, ADC3, and PB3. This means that Pin 2 can be used as Pin Change Interrupt 3 (PCINT3), Clock In (CLKI), Analog to Digital Converter 3 (ADC3), and PORT B 3 (PB3). Pin 2 is always Pin 2 but also Port 3 in the PB register, for example.

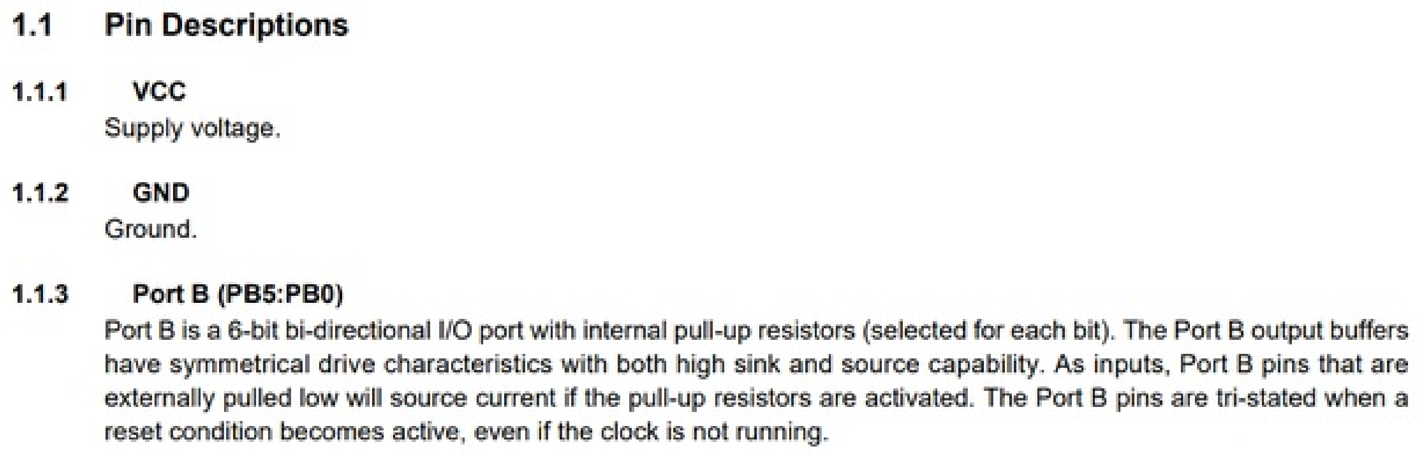

PDIP/SOIC/TSSOP
(PCINT5/$\overline{RESET}$/ADC0/dW) PB5 1
(PCINT3/XTAL1/CLKI/$\overline{OC1B}$/ADC3) PB3 2
(PCINT4/XTAL2/CLKO/OC1B/ADC2) PB4 3
GND 4
8 VCC
7 PB2 (SCK/USCK/SCL/ADC1/T0/INT0/PCINT2)
6 PB1 (MISO/DO/AIN1/OC0B/OC1A/PCINT1)
5 PB0 (MOSI/DI/SDA/AIN0/OC0A/$\overline{OC1A}$/AREF/PCINT0)
NOTE: TSSOP only for ATtiny45/V

Step 2:

For us, we want to know which ports are used as input/output (I/O) pins. We find this in the Pin Description section on Page 2. So now we know we are interested in the pins labelled as PB-something, in this case, PB0 - PB5.

Writing 1s and 0s to different registers will affect what a pin will do. If you were to use the digitalWrite() on Pin 3, what it would do is find the register that can label the pin to PB4, find the bit in that register, and then write it, while also making sure that it is not serving any of the other possible functions attached to it. This is why digitalWrite() takes the time that it does to do what it does.

When it comes to using digitalWrite(), you always need a pinMode() function to call it an output first. This is done with yet a different register.



1.1 Pin Descriptions

1.1.1 VCC
Supply voltage.

1.1.2 GND
Ground.

1.1.3 Port B (PB5:PB0)
Port B is a 6-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port B output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port B pins that are externally pulled low will source current if the pull-up resistors are activated. The Port B pins are tri-stated when a reset condition becomes active, even if the clock is not running.

Step 3:

The answers to all the questions we have about this begin on page 53: 10. I/O Ports.

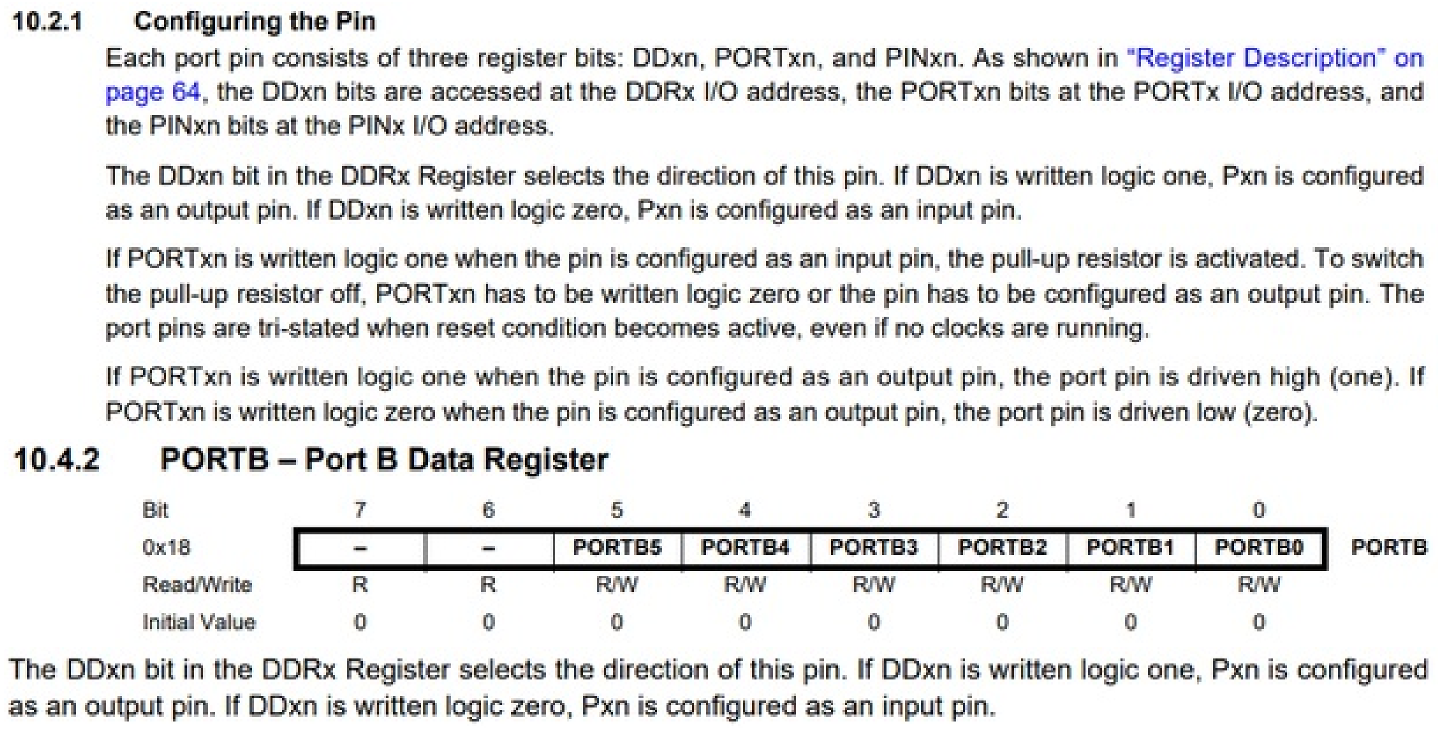
Scrolling down to 10.2.1 Configuring the Pin (in image 1), you see a mess of abbreviations that is enough to turn off anyone who does not understand what they mean.

Let's take a look at the acronyms here so we can understand them better.
First up, DDxn, which refers to DDRB, or the Port B Data Direction Register. Then, PORTxn, which refers to PORTB, or the Port B Data Register. And, PINxn, which refers to PINB, or the Port B Input Pins Address (which we will not address in this Instructable). All of these are shown on page 64.

The second image above is the **D**ata **D**irection **R**egister for port **B**.
In the top row, you have the bit numbers. These are effective location designations. Binary is read from right to left, hence the backwards numbering. There are 8 bits in this register, from 0 to 7.
The second row, is the name of the ports in the register. Note that there are two bits that don't have names. They do not do anything. By the end, these are the names you will use when referencing the various bits in a register.
The third row tells you if you can read the bit, write the bit, both, or neither. Bit 6 and 7 do not have a port name, do not do anything, and therefore cannot be written to. All other bits can be read from and written to.
The fourth row tells you what each bit initializes to. In this case, they are all 0s.

There are a total of 6 usable bits, going from 5 to 0, or 0 to 5. Either is acceptable. You may see it referred to like this: DDRB[5,0] or DDRB[0,5].

Shown on the bottom of page 54, it states that a 0 for anyone of these writes the port (and therefore its corresponding pin) as an input. A 1 writes the port as an output. So this is the first thing we want to do in replacing a digitalWrite() command, by replacing the pinMode() command. Before showing you what the actual coding looks like, we will skip to the next register to continue with understanding what to do.



10.2.1 Configuring the Pin

Each port pin consists of three register bits: DDxn, PORTxn, and PINxn. As shown in "Register Description" on page 64, the DDxn bits are accessed at the DDRx I/O address, the PORTxn bits at the PORTx I/O address, and the PINxn bits at the PINx I/O address.

The DDxn bit in the DDRx Register selects the direction of this pin. If DDxn is written logic one, Pxn is configured as an output pin. If DDxn is written logic zero, Pxn is configured as an input pin.

If PORTxn is written logic one when the pin is configured as an input pin, the pull-up resistor is activated. To switch the pull-up resistor off, PORTxn has to be written logic zero or the pin has to be configured as an output pin. The port pins are tri-stated when reset condition becomes active, even if no clocks are running.

If PORTxn is written logic one when the pin is configured as an output pin, the port pin is driven high (one). If PORTxn is written logic zero when the pin is configured as an output pin, the port pin is driven low (zero).

10.4.2 PORTB – Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x18	–	–	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The DDxn bit in the DDRx Register selects the direction of this pin. If DDxn is written logic one, Pxn is configured as an output pin. If DDxn is written logic zero, Pxn is configured as an input pin.

Step 4:

This labels the **PortB** data **R**egister. This says that each pin within this register is initialized to 0. A 0 writes the port as LOW. A 1 writes the port as HIGH. This is how to replace the digitalWrite() function itself! Shown above is at the top of page 51:

Alright, finally here. Let's talk about the actual coding now!
First off, we want to add Image 4 as a line of code at the top of our program so that our commands are understood in the Arduino IDE.

As mentioned above, we first need to replace pinMode(). This is done with the DDRB port register. We do this with a "DDRB =" command. For example, let us say you want to write Pin 2 on the ATTiny85 to be an output. So we need look at the datasheet at the diagram again and see that Pin 2 is labelled as PB3. Then we look at the DDRB to see that PB3 is the 4th bit (reading from right to left) in the Register. We need this to be a 1. So we say:

DDRB = 0b00001000;

(Where "0b" tells the computer you are writing a number in byte format.)

And that's it! Pin 2 is set as an output.

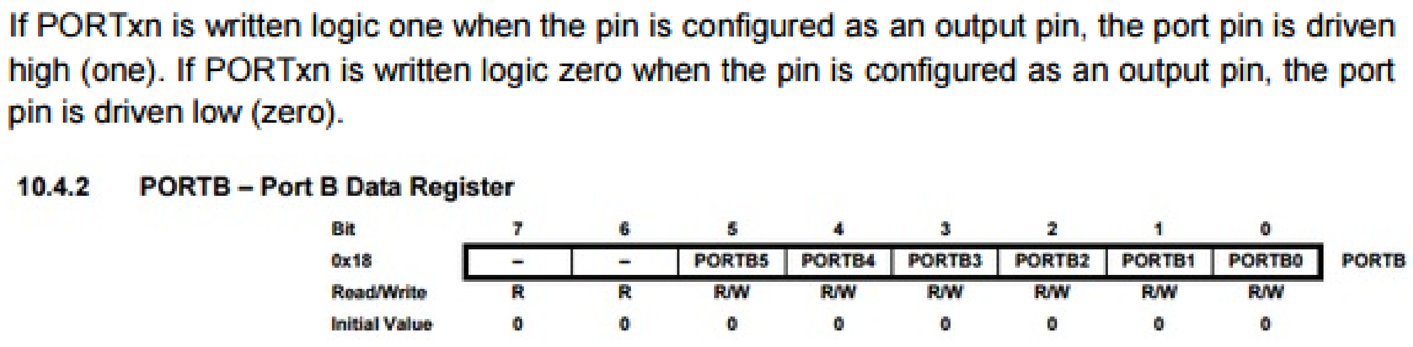
Now let us write the pin as HIGH.

Again, we know from the figure above from the datasheet that Pin 2 is PB3. Now we need to reference the PORTB Register and find out which bit writes to it. We see it is also the 4th bit (from right to left) in the Register. We need that to be a 1. So we say:

PORTB = 0b00001000;

Pin 2 is now set to HIGH.

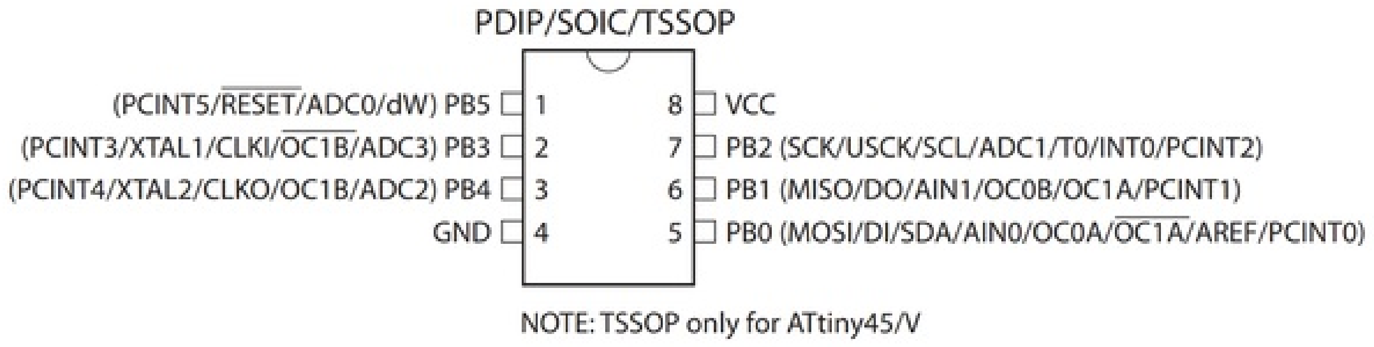
Be sure to look at the next Step!

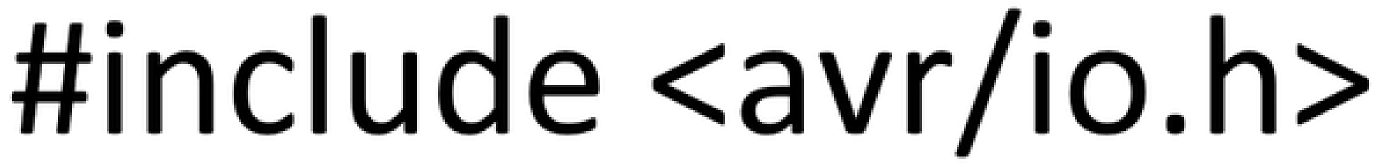


If PORTxn is written logic one when the pin is configured as an output pin, the port pin is driven high (one). If PORTxn is written logic zero when the pin is configured as an output pin, the port pin is driven low (zero).

10.4.2 PORTB – Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x18	–	–	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	


PDIP/SOIC/TSSOP
(PCINT5/RESET/ADC0/dW) PB5 1
(PCINT3/XTAL1/CLKI/OC1B/ADC3) PB3 2
(PCINT4/XTAL2/CLKO/OC1B/ADC2) PB4 3
GND 4
8 VCC
7 PB2 (SCK/USCK/SCL/ADC1/T0/INT0/PCINT2)
6 PB1 (MISO/DO/AIN1/OC0B/OC1A/PCINT1)
5 PB0 (MOSI/DI/SDA/AIN0/OC0A/OC1A/AREF/PCINT0)
NOTE: TSSOP only for ATtiny45/V



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

Step 5:

This, though, is not actually how you want to write these commands…

The problem with writing commands like this is that when you do, you are not only writing a 1 to that 4th spot, but you are also writing a 0 to all the other spots as well. If you are writing a program that switches multiple pins on and off at different times, keeping track of how to write the command each time can grow increasingly confusing and can definitely lead to errors!!!

Be happy though, there is a way around this, and it is even easier on the eyes than what we have above. We replace

DDRB = 0b00001000;

with

DDRB |= (1 << PB3);

This is how you will want to write a pinMode() with port manipulation. The "|=" is a bitwise OR assignment. You need only worry about what this does an less so how it does it. But 1 << PB3 means that you are going to write a 1 to the PB3 port in the Register, with the << specifically meaning it will look for the PB3 spot wherever it is while ignoring all other bits in the Register.

The one big thing to know here. You can only write a 1 to a bit in a Register in this fashion such that you can only go from 0 to 1 with the |= operator. This is because the whole process is a shorthand method for a binary math operation based on comparisons for the values in the Register.

So when we apply this same logic to the PORTB command, it looks like this:
PORTB |= (1 << PB3);
And again, the same rule above applies here as well. Which then begs the question, "How do I turn a pin back to LOW if I cannot write a zero, or if I can only write a 1?"

This is accomplished with a different operator. PORTB &= ~(1 << PB3);

The "&=" is a bitwise AND assignment and the "~" is a bitwise NOT assignment. This is, again, more binary math stuff happening here. But what you need to know is that using this method will *unwrite* a 1 in the spot you designate and turn it back into a 0.

Step 6:

Here is an example of a sketch of mine that uses port manipulation.

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

void setup(){
DDRB |= (1 << PB3);   //replaces pinMode(PB3, OUTPUT);
DDRB |= (1 << PB4);     //replaces pinMode(PB4, OUTPUT);
}

void loop()
{
 delay(random(600000, 900000));
 byte state = random(0, 2);
 switch(state)
  {
   case 0:
   PORTB |= (1 << PB3);   //replaces digitalWrite(PB3, HIGH);
   delay(20);
   PORTB &= ~(1 << PB3);  //replaces digitalWrite(PB3, LOW);
   break;

   case 1:
   PORTB |= (1 << PB4);   //replaces digitalWrite(PB4, HIGH);
   delay(20);
   PORTB &= ~(1 << PB4);  //replaces digitalWrite(PB4, LOW);
   break;
  }
}
```

Step 7:

One of the benefits to pinMode() and digitalWrite() is that you can use variables inside them so you don't have to go through your entire code if you want to change one pin for another. We can do this with our port manipulations as well with #define. It effectively labels a port/register reference to a variable. It is used as follows:

#define [variable name] [port label]

The above sketch is now rewritten with definitions.

```
#include <avr/io.h>
#define LED_PIN PB1
#define BUZZER_PIN PB3

void setup()
{
 DDRB |= (1 << LED_PIN);
 DDRB |= (1 << BUZZER_PIN);
}

void loop()
{
 delay(random(600000, 900000));
 byte state = random(0, 2);
 switch(state)
  {
   case 0:
   PORTB |= (1 << LED_PIN);
   delay(20);
   PORTB &= ~(1 << LED_PIN);
   break;
   case 1:
   PORTB |= (1 << BUZZER_PIN);
   delay(20);
   PORTB &= ~(1 << BUZZER_PIN);
   break;
  }
}
```

For those who are curious, this sketch goes to a prank toy that plugs into a USB at the back of someone's computer. At random between 10 and 15 minutes, it either blinks a LED or buzzes a buzzer.

First page:
"For example, Pin 3 is labelled as PCINT3, CLKI, ADC3, and PB3. This means that Pin 3 can be used as..."
I think you mean Pin 2 (as 'Pin' refers to a physical pin).

I've read though this many times and have never caught that error. Thanks for pointing it out!
The Instructable has been updated to reflect that correction.

who can help me?

[illegible]

Hi. Gee, man, You really throw me back to my days writing in Assam. I wrote a dozen's of progr's for a microcontroller ST62xx. It was (IS) the beaty of them powerful commands when used in a appropreat way. Let's have an ex. :
1. Having inputs on a "port" A.
2. some of these may be config. as outputs and some as inputs.
3. Now we want to eliminate the "debounce" affect of them inputs.
Let's assume 4 of them port pins to be inputs having the annoying behavioure of a "bounce" of a switch.
Now in ass. it's very simple to:

4. Read the whole Port
5. ”AND” the port to a register, (0010 1001) to the Input
For the ”Debounce” check I only need one check if ”any” of the pins is changed.
Then I’ll make a deboune check on the whole register, not the need of to check every pin
Check me on Instructables ” kiselin”

I'm sorry I missed your comment for the last six weeks! I will definitely be checking your Instructables out!

Thank to your explanation i manage to made a 7-segment hexadecimal counter (using two serial shift registers to have 16 digital outputs).
https://www.youtube.com/watch?v=NAh5o-Gnx9E

That's awesome! I'd love to see your code. Mad props, my dude.

very informative, thanks for sharing

As you mentioned I almost fell out of my seat when I looked at the datasheet but now I understand. Thank you so much.

very helpful thanks!

Thank you so much!!

thank u so much, now I understand.

You're welcome. I tried to make it as straight forward as I could in a way that made sense to someone new, like myself in the process of writing this.

It was hard, given that I had to keep bouncing around in the datasheet to explain why things are the way they are.

Did u know why on a Attiny85 the PB5 does not accept the same code than the other 5?
I mean I just make a normal "Blink" sketch with your info and it works on PB0, PB1, PB2, PB3 and PB4 but NOT on PB5... I supose the RESET function is around this...

You'll notice on page 60 in the datasheet, Alternate Functions of Port B, that PB5 is also the reset pin, as you noted. At the bottom, its states: RESET: External Reset input is active low and enabled by unprogramming (“1”) the RSTDISBL Fuse.

What this means is that, you have to specifically disable the reset disable fuse (outside of programming), to override the reset function and use it as an I/O pin.

HOWEVER! Once you do this, you can no longer program the chip using an ISP (FTDI breakout, Arduino as ISP, the MKII, etc) as all of these require the reset pin to program the chip. Calling it a fuse means this is not something that changes within the program, and does not go back to default when power is turned off. So, if you disable the reset, it is completely disabled. You can change it back with a High Voltage Serial Programmer, but that is another topic entirely.

Your best bet is just not to use PB5 as an I/O pin. If you need another pin, you should just use a chip with more pins.

thank u again.

What is confuse to me is this:

I upload a Digispark loader to a new 85 bare chip using Arduino UNO as ISP with a software made on Brasil (http://www.hackeduca.com.br/) , and now I can program it directly via USB with a so called "development board" like this: http://www.ebay.com/itm/Development-Programmer-Boa.... , ok now I can not use the PB5, no problem... But now I can not program it using ISP, it only works via the development board, why??

So I can disable the reset at this time and use it?

The chip will still work and be programmed with the "development board"?

When the chip is new and empty the reset pin is enabled?

Hi Sigifredo, I am the developer from (http://www.hackeduca.com.br/attiny-com-o-bootloade...

If you want to "reset" the reset pin you will need a high voltage (12 volts) programmer, otherwise, only the development board.

You can create one by yourself it's not expensive.

The way the ATTiny85 is designed is that pin 1 has the primary function of being the Reset Pin before being anything else. That is because this is as fuse bit that says it is. Read the following article on fuses:

http://www.ladyada.net/learn/avr/fuses.html

Changing a fuse requires a different program and interface. It's not like a normal bit in a register.

When it comes to programming a chip, the ISP needs to be able to be able to reset the chip using the reset pin. If you change that fuse to disable to reset function, a normal ISP CANNOT reprogram the chip anymore. It is effectively bricked at that point.

As the article above says:

"Reset Disable

This fuse turns the **Reset** pin into a normal pin instead of a special pin. If you turn this fuse on you cant program the chip using **ISP** anymore. I would suggest you never set this fuse unless you *really* mean to.

By default, chips that come from the factory have this turned off (that is, Reset is enabled)," you can change it back, but its an involved process that requires a lot more research.

Finally I understand!! Thank you for this write up!!.