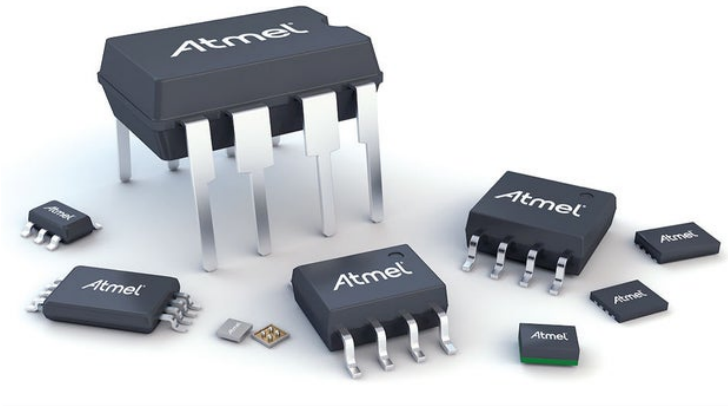
ATTiny Port Manipulation (Part 1.5): DigitalRead

by myless8

ATTiny Port Manipulation (Part 2): analogRead()

This is a continuation of the first Instructable I wrote titled, ATTiny Port Manipulation (Part 1): pinMode() and digitalWrite(). Though there is some crossover information between that one and this one, this is still written assuming you understand the basics of port manipulation within registers. We are now going to delve into digitalRead(). We will again be using the ATTiny85 as our example chip, though all methods learned are usable on all ATTiny microchips. The following is a link to the datasheet.

ATtiny85 datasheet



Page 2: Figure 1-1. Pinout of ATtiny13A: 8-PDIP/SOIC is the chip version we will be referencing.
Even after knowing how to replace pinMode() and digitalWrite(), it is slightly less intuitive as to how to read the state of a register port and use it in code. We will address one good way of reading the state of a digital input pin that is simple and reliable.

This Instructable will be shorter than its predecessor but will be just as useful. So without holding out on you any more than necessary, let's get on with it.

Supplies:

ATtiny85 datasheet

Step 1:

We recall from the previous Instructable, ATtiny Port Manipulation (Part 1): pinMode() and digitalWrite(), that for this particular chip has a single register for GPIO pins. Image 1 shows which pins are associated with Port B.

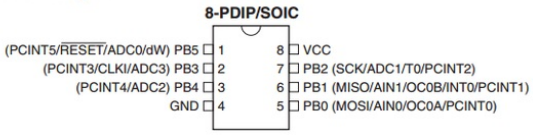
overview of the features associated with these pins. Most of it is pretty technical and of no real use to you. The part of interest to you first is at the bottom of page 50, Configuring the Pin, as shown in Image 3. We find that we will be dealing with Port B and the registers associated with it: DDRB - Data Direction Register for Port B PORTB - Port B Data Register PINB - Port B Input Pins Address

Image 2 shows that PB0-PB5 are indeed I/O (input/output) pins.

Chapter 10: I/O Ports begins on page 49 with an

Each of these are their own registers with details shown on pages 57 - 58 under 10.4: Register Descriptions. Image 4 shows these registers in detail. Next we will learn what is important about them and begin discussing an example sketch.

Figure 1-1. Pinout of ATtiny13A



1.1.3 Port B (PB5:PB0)

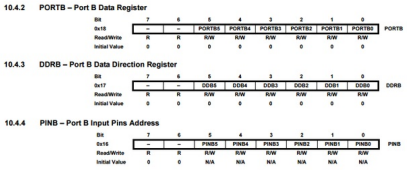
Port B is a 6-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port B output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port B pins that are externally pulled low will source current if the pull-up resistors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

Port B also serves the functions of various special features of the ATtiny13A as listed on page 55.

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 57, 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.



Step 2:

Lets start by conceiving of a sample sketch where we might use some input pins for some application. A simple enough example would be to have 2 buttons that will act as +/- buttons for an audio project. One button will be the +1, the other for -1. We will have an initial value and change it with these buttons.

Looking at image 1, we can decide to use any two pins from pins 2, 3, 5, 6, or 7. (We would not want to use PB5 on pin 1 as it is default RESET only and changing this is extremely undesirable). We decide to use pins 2 and 3. These are port pins PB3 and PB4, respectively.

By looking again at the Configuring the Pin section (Image 2), the last sentence states that if DDxn is written logic zero, Pxn is configured as an input pin. We know from the previous Instructable that these pins all have default values of 0. This can be seen in Image 4, 10.4.3 DDRB Initial Value. PB3 and PB4 are the 3rd and 4th bit, by index, in the register (from right to left).

We do not need to actually write any code to configure these pins as inputs, but let us do so anyways. So far, we are working only in the void setup(). Also, do recall that we need

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

at the top so all port manipulation commands are understood.

```
#include <avr/io.h>
void setup(){
DDRB &= ~(1 << DDB3);
DDRB &= ~(1 << DDB4);
}
```

Recall that '|=' will write a '1' to the specified spot and '&= ~' will unwrite a '1' to a spot.

So, in this case, we have verified that the two ports are 0, configuring the pins as inputs.

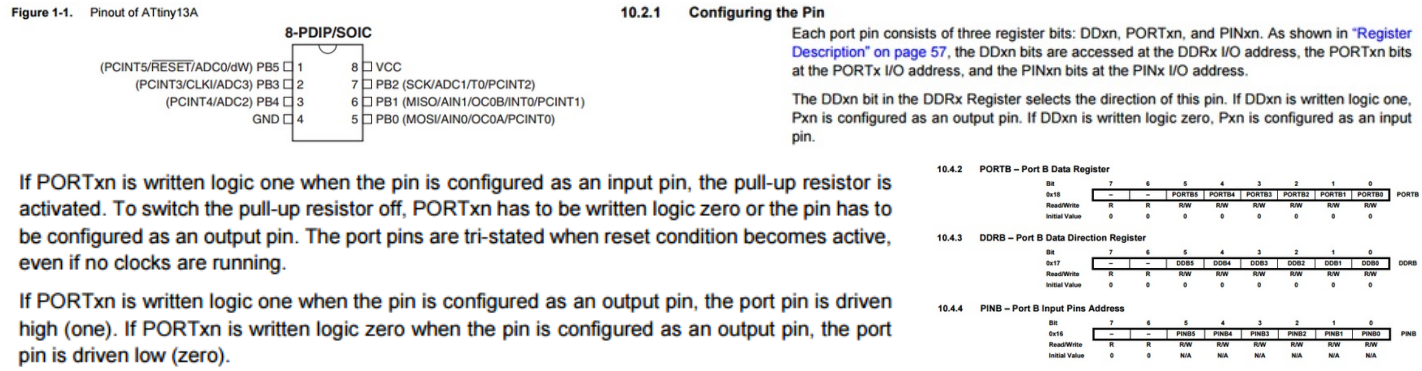


Figure 1-1. Pinout of ATtiny13A

8-PDIP/SOIC

(PCINT5/RESET/ADC0/dW) PB5 1 — 8 VCC
(PCINT3/CLKI/ADC3) PB3 2 — 7 PB2 (SCK/ADC1/T0/PCINT2)
(PCINT4/ADC2) PB4 3 — 6 PB1 (MISO/AIN1/OC0B/INT0/PCINT1)
GND 4 — 5 PB0 (MOSI/AIN0/OC0A/PCINT0)

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 57, 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	

10.4.3 DDRB – Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x17	–	–	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
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	

10.4.4 PINB – Port B Input Pins Address

Bit	7	6	5	4	3	2	1	0	
0x16	–	–	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	N/A	N/A	N/A	N/A	N/A	N/A	

Step 3:

We need to take an aside and address something right away when it comes to reading digital input pins. They are very sensitive when it comes to picking up a signal.
The issue that arises is called a Floating Pin and can be observed by creating a simple sketch on your Arduino, as below, WITHOUT ATTACHING ANYTHING to pin 3.

```
int inputPin = 3;
int x;

void setup() {
pinMode(inputPin, INPUT);
Serial.begin(9600);
}

void loop() {
x = digitalRead(inputPin);
Serial.println(x);
delay(100);
}
```

Running the program and viewing the Serial Monitor will reveal that you randomly switch between seeing 0 and 1.

Why is this?

Again, pins configured this way are very sensitive to electromagnetic radiation coming from all of its surroundings. In a digitalRead() scenario, the Arduino does not recognize analog values. It only reads 0 or, more than 0 (which means 1). Even though the pin is *effectively* receiving 0V and we'd expected it to read as 0, the radiation around it randomly gets detected and read as some value higher than 0, which in turn, means 1 to your Arduino.
The solution is to use either a pull-up resistor or a pull-down resistor. These are not special resistors of any kind. They only get their name from how they are used. They are high in value, typically 10k ohm.
If you want to use a pull-down resistor, your 10k ohm, and connect the input pin to GND. If you want to use a pull-up

resistor, you take the same 10k ohm resistor and connect your input pin to 5V, instead.

The reason for doing this is, when you are not trying to send data to the input pin, like pressing the button in our example, you are tying the pin to either 5V or GND so that the reading is stable. Your button, upon being pressed, would then change the signal, which is what gets referenced in your coding as the button being pushed.

As we will use in our example in the following steps, the easiest way to use the button is to tie it to 5V with a pull-up resistor and have the button attached to GND. This is because, as can be seen in Image 1, our port pins have built-in pull-up resistors inside the chip, we need only activate them to make use of them.

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.

Port B also serves the functions of various special features of the ATtiny13A as listed on page 55.

Step 4:

So, we just discussed that we will have our buttons use their internal pull-up resistors. The buttons will then be attached to GND. So circuit would be represented by the following:
PB3 -> button -> GND
PB4 -> button -> GND

The way we setup our internal pull-up resistors can be seen in Image 1. This is the second half of the 10.2.1 Configuring the Pin section at the top of page 51. If you recall the PORTB register is responsible for configuring OUTPUT pins as high or low. But this is an INPUT pin, so setting it PORTB3 and PORTB4 to 1 will instead activate their pull-up resistors.

This is represented in the following code, which is the same as before, with the addition of what is needed for the pull-up resistors.

```
#include <avr/io.h>
void setup() {
DDRB &= ~(1 << DDB3);    //set PB3 as input
DDRB &= ~(1 << DDB4);    //set PB4 as input
PORTB |= (1 << PORTB3);  //activate internal pull-up resistor for PB3
PORTB |= (1 << PORTB4);  //activate internal pull-up resistor for PB4
}
```

Moving on...

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).

Step 5: Binary Logic (an Aside)

This is the majority of what we need to do. But now we need to take another aside and discuss, briefly, a binary logic table. I know, I know. That sounds awful, but it is surprisingly straightforward and i'll keep it short.
We need to understand what the & symbol means in Binary Logic.
...It means Logical AND. *sigh* simple enough, right?

The 'logical' part in it means that it is used specifically to compare two things. Bits, to be precise. The idea is that you feed two bits in and get one bit out. This can be represented by a truth table. The left side represents the inputs; the right side tells you the outputs.

0 0 | 0
0 1 | 0
1 0 | 0
1 1 | 1

The short version is that, you can ONLY get 1 as an output if you gave it two 1s as inputs to begin with. You will understand why this is necessary in the following steps.

Step 6:

Alright, lets get to this. Let us quickly revisit the existing code we already have with a few additions. Those additions will be the variables x, lastx, y, and lasty, to hold our digitally read data; val and lastVal to hold our changing value.

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

byte x;
byte lastx;
byte y;
byte lasty;
int val;
int lastVal;

void setup() {
DDRB &= ~(1 << DDB3);    //set PB3 as input
DDRB &= ~(1 << DDB4);    //set PB4 as input
PORTB |= (1 << PORTB3);  //activate pull-up resistor for PB3
PORTB |= (1 << PORTB4);  //activate pull-up resistor for PB4
}
```

When it comes to reading input pins, its state is always being updated, even while you're not reading it. What happens when you read it, is you are taking a snapshot of its state at that point in time.

Now we get to talk about code that is used within the void loop(). It is as simple as the following:

```
void loop() {
x = PINB;
y = PINB;
delay(50);
}
```

Confused yet? Why did I set both variables to the same thing when they are supposed to be different values?

This is where the previous step will come into play. We need to understand what I actually just did. Let's assume that when this takes place, neither button is being pushed. What would x and y actually be; what do their values really look like?

PINB is an entire Register with 8 bits of data and we just saved it twice to two different variables. So the variable data actually looks like the following:

x = B00011000
y = B00011000
*the 'B' tells you the data in in Byte format and that the 0s and 1s represent a binary byte of data.
If you recall PB3 and PB4 both have their pull-up resistors activated and their default status of the buttons when NOT pushed is 1.

So...now what?

Step 7:

Right now, we have:
x = B00011000
y = B00011000

These are the snapshots of the states of the port pins at the time you saved them to their variables.
What we are interested in is isolating individual bits of data from within these snapshots for analysis. This is done with our & (logical *and*) comparison.

We do this by comparing each byte to another custom byte of our own making in order to accomplish our goal.
When we want to read PB3, we will compare it to B00001000.
When we want to read PB4, we will compare it to B00010000.
The reason we do this is because we are comparing each bit from our snapshot to the bit in the corresponding spot from our manually generated bytes.

Lets look closer at PB3 as shown in Image 1.

The first row is the data we saved to x. The second row is our manually chosen byte to compare x to. The third row is the result of comparing x and our manual byte.

The result will be yet another byte that can only have a 1 only in the spot where BOTH bytes had a 1. Because our custom byte only has a 1 in the spot corresponding to PB3s location in its register, we will only ever get a 1 in our output if PB3 is also a 1. This is how we isolate PB3s value. The same goes for PB4.

You do this by running the following lines of code:

```
x = x & B00001000;
y = y & B00010000;
```

Now if we look at the actual values for x and y now, we have:

x = B00001000;
y = B00010000;
We have effectively cherry picked the values from the proper register bits. But so what? Did this accomplish anything of value? Very much so!
If we consider the scenario where a single button is pushed, lets say on PB3. That would pull PB3 to LOW and we would get this instead:
x = B00000000;

y = B00010000;

We have successfully isolated a digital read from a specific pin!

We can now use this as something in the code! But how exactly?
It is nice to know that even the the data is of the form B******** where the *'s can be either 0's or 1's, Arduino is cool enough that it will seamlessly switch between binary and decimal values. So with y = B00010000 the Arudino can also understood it as being binary value 16. Or as we will reference it, larger than 0.

The following is currently updated code for our example:

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

byte x;
byte lastx;
byte y;
byte lasty;
int val;
int lastVal;

void setup() {
DDRB &= ~(1 << DDB3) //set PB3 as input
DDRB &= ~(1 << DDB4); //set PB4 as input
PORTB |= (1 << PORTB3); //activate pull-up resistor for PB3
PORTB |= (1 << PORTB4); //activate pull-up resistor for PB4
Serial.begin(9600);
}

void loop() {
x = PINB;   //set x equal to input register B
y = PINB;   //set y equal to input register B
x = x & B00001000;  //isolate value of PB3
y = y & B00010000;  //isolate value of PB4
if (x == 0 && x <> lastx) //if x is equal to 0, PB3 button was pushed. then...
{
val += 1;   //add 1 to x
lastx = x;   //lastx
}
if (y == 0 && y <> lasty) //if y is equal to 0, PB4 button was pushed. then...
{
val -= 1;   //subtract 1 from y
lasty = y;   //update lasty
}
if (val <> lastVal)  //if val has changed then...
{Serial.Println(val);}  //print value of val
lastVal = val;   //update lastVal
delay(5);   //short delay
}
```

PB3	B	0	0	0	1	1	0	0	0
our byte	B	0	0	0	0	1	0	0	0
& outcome	B	0	0	0	0	1	0	0	0

Thank you very much

OP here. Instructable was updated to fix a few errors and update the final code so pressing and holding a button doesn't explode the value of val.
Also updated to the use of ATtiny85 instead of ATtiny13A.