



## ATTiny Port Manipulation (Part 2): AnalogRead()



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 analogRead().

We will again be using the ATTiny13A as our example chip, though all methods learned are usable on all ATTiny microchips. The following is a link to the datasheet.

[Attiny85 Datasheet](#)

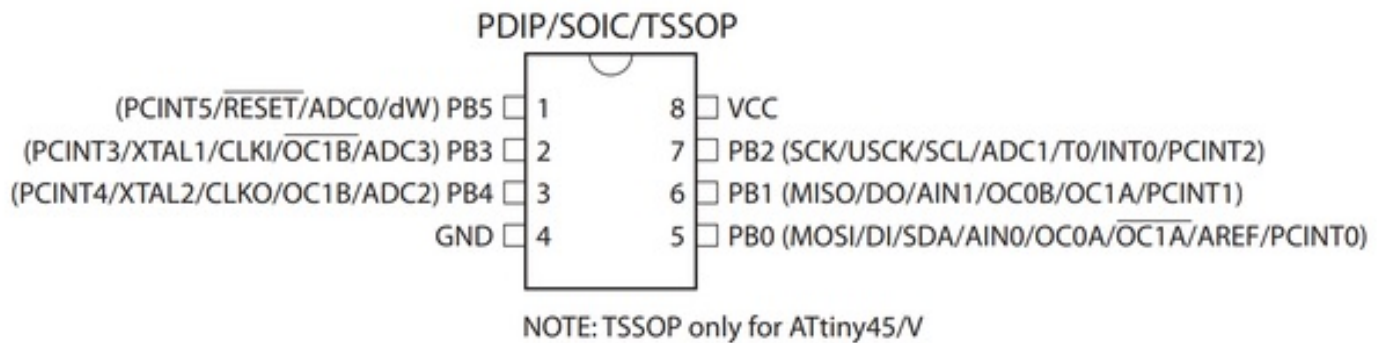
Page 2: **Figure 1-1. Pinout of ATtiny25/45/85: PDIP/SOIC/TSSOP** is the chip version we will be referencing (Image 1). Now bear in mind that, it is not as simple to send bit information to the ATTiny as it is getting it from. You can have the chip send you a 5V signal, then you, outside the system, can use that signal for whatever. To light a LED, switch a transistor, run a motor, etc... The chip does not need to know, so you do not have to tell it. Going the other way around gets more complicated.

You have to tell the chip, that it is going to receive data to begin with, tell it when to receive the data, how to interpret the data, and what to do with that data. The following is a quick list of the order of operations of how to receive analog data through the Analog to Digital Converter (ADC). We will then go through almost every step, so that by the end, you will be a master of analogRead() using Port Manipulation.

1. Include the avr/io library so all port commands are understood.
2. Define some macros (optional but extremely useful).
3. Set the voltage reference.
4. Choose which pin will be Analog Input.
5. Choose precision level and left or right adjusting.
6. Enable the Analog to Digital Converter.
7. Set analog read type.
8. Select prescaler.
9. Start the conversion.
10. Retrieve the data (or possibly sets of data).
11. Use data.
12. Repeat from step 10.

This may seem a little intense, but the journey will be a very rewarding one. It will give you a much better understanding of the inner workings of the Arduino IDE and how all AVR chips really work on the inside.

So, without stalling, let's get started!



## Step 1: Steps 1 and 3: Understanding the Analog to Digital Converter and Reference Voltages

### Step 1:

Use the following code line at the top of your sketch so all port commands are understood:

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

### Step 2:

This will not be included in this Instructable.

### Step 3:

To start, we need to understand what the Analog to Digital Converter (ADC) really does. It takes an analog signal and converts it to a digital one. (duh! Right?) An analog signal can hold any value and will fluctuate continuously. Whereas a digital signal will only hold discrete values.

What this means for us, is that you can send a signal with, literally any value (between its high and low points). Example: between 1 and 10, an analog signal could be 1, 3.6, 4.3333, etc. Whereas a digital signal is discrete, it might only hold values to plus or minus a half. Example: 1, 2.5, 5, 7.5, etc.

The ADC will take your analog signal and convert it to the digital version. 4.3333 would change to 4.5 instead, for example.

We learned in Part 1 how to find out which pins are attached to which ports and how to find the corresponding registers. According to the pinout diagram (Image 1) we want the ports marked ADC. There are 4 of them. These are your analog pins:

Pin 1 is ADC0

Pin 7 is ADC1

Pin 3 is ADC2

Pin 2 is ADC3

Our ATtiny chips will take a signal from 0 to 5.5V on these pins and convert it from 0 to 255 or from 0 to 1023 (depending on the resolution you want). We will assume 0 – 255 for the remainder of this Instructable.

In order to do this, it uses a formula:

$$\text{ADC} = (\text{Vin} * 1024) / (\text{Vref})$$

**ADC** is the converted value that you are reading

**Vin** is the signal voltage signal being converted

**Vref** is the reference voltage

In order to convert your signal into a discrete value, the chip needs to compare your voltage to a different already known voltage. There are typically internal voltages usable on the chip itself; usually in the form of 3.3V or 5V. You can also use an external reference voltage, that can hold any voltage you want (with a maximum of 5V). On your Arduino, this is what the AREF (Analog Reference) pin is for. This is useful when you are using the ADC for a peripheral device attached to the Arduino that uses its own power source. It helps to avoid translation errors between devices (though this is a topic for another time).

According to the Datasheet, our chapter on ADC starts on page 82.

Note that I will be going in a different order than that chapter in the datasheet. That is because I am writing in the order you will use the information within the Arduino IDE; so bear with me.

**Chapter 17, section 6** (17.6.2) is where we find ADC Reference Voltage (Image 2).

We see that we only have two choices for an internal reference voltage at 1.1V or 2.56V. The internal reference voltage is always worth taking note of when working with any chip.

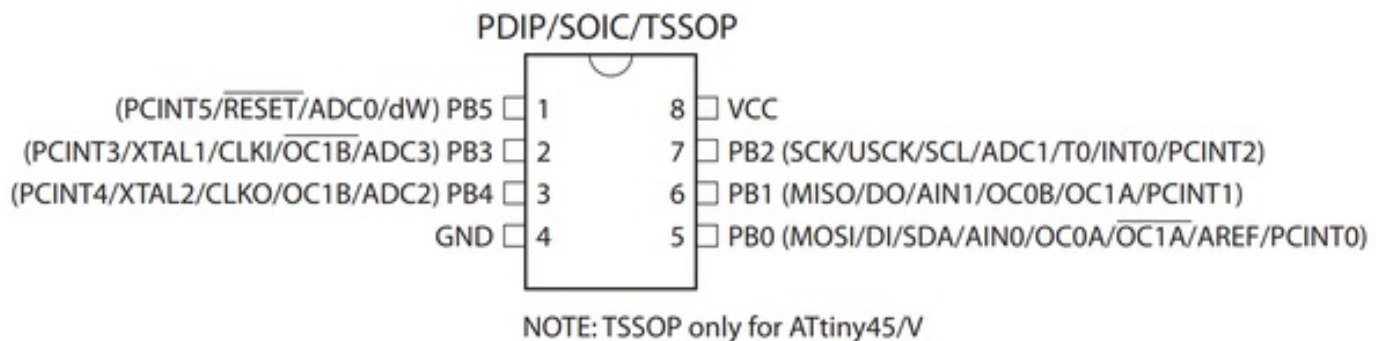
Then skipping to **17.13 Register Description**, we find the registers. Looking through, we see that the Voltage Reference Selection bits are within the ADMUX (ADC Multiplexer Selection) Register (Image 3).

We are looking at the REFS0 and REFS1 (Reference Selection 0 and 1) bits which happen to be bits 6 and 7 in the ADMUX Register. When it says that you can use Vcc as your reference voltage, it means it will use the voltage level that the chip is currently being powered with. This chip operates anywhere between 1.8V and 5.5V, so the Vcc can be anywhere within that range. We will assume that you are running the chip with a 5V source. This is a good reference voltage to use. However, we are going to use the internal voltage reference as our reference voltage. The register tells us that the bits are initialized to 0, which means we need to change REFS1 it to a 1.

This is done with the following code line in your void setup():

```
ADMUX |= (1 << REFS1); //use internal reference voltage of 1.1V
```

Now, let's move on.



### 17.6.2 ADC Voltage Reference

The voltage reference for the ADC ( $V_{REF}$ ) indicates the conversion range for the ADC. Single ended channels that exceed  $V_{REF}$  will result in codes close to 0x3FF.  $V_{REF}$  can be selected as either  $V_{CC}$ , or internal 1.1V / 2.56V voltage reference, or external AREF pin. The first ADC conversion result after switching voltage reference source may be inaccurate, and the user is advised to discard this result.

### 17.13.1 ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0	
0x07	REFS1	REFS0	ADLAR	REFS2	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits 7:6, 4 – REFS[2:0]: Voltage Reference Selection Bits**

These bits select the voltage reference ( $V_{REF}$ ) for the ADC, as shown in Table 17-3. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSR is set). Whenever these bits are changed, the next conversion will take 25 ADC clock cycles. When differential channels and gain are used, using  $V_{CC}$  or an external AREF higher than ( $V_{CC} - 1V$ ) as a voltage reference is not recommended as this will affect the ADC accuracy.

**Table 17-3.** Voltage Reference Selections for ADC

REFS2	REFS1	REFS0	Voltage Reference ( $V_{REF}$ ) Selection
X	0	0	$V_{CC}$ used as Voltage Reference, disconnected from PB0 (AREF).
X	0	1	External Voltage Reference at PB0 (AREF) pin, Internal Voltage Reference turned off.
0	1	0	Internal 1.1V Voltage Reference.
0	1	1	Reserved
1	1	0	Internal 2.56V Voltage Reference without external bypass capacitor, disconnected from PB0 (AREF) <sup>(1)</sup> .
1	1	1	Internal 2.56V Voltage Reference with external bypass capacitor at PB0 (AREF) pin <sup>(1)</sup> .

## Step 2: Step 4: Choosing Your Analog Read Pins

When it comes to choosing pins for output signals, it is easy to do and you can control as many as are available on the chip you are using. Using pins as inputs is a little different. This is because the chip has only one analog to digital conversion unit. This means that only one pin can be used at a time.

**14.6.1 Channel Inputs** gives us an overview of this (Image 1).

It is saying that you have to choose one pin, or *channel*, at a time for each conversion. This is pretty easy to do within the Arduino IDE. We will cover this later on. For this Instructable we will assume we are only using one analog input.

To choose which analog input we want to use we have to reference the ADMUX Register again (Image 2).

(Image 3) is seen at the top of page 93.

For this Instructable we are going to be reading analog data from ADC3.

This means we need to write both MUX1 and MUX0 to 1's.

This is done with the following code lines. You would do this within the void setup().

```
ADMUX |= (1 << MUX0); //combined with next line...
ADMUX |= (1 << MUX1); //sets ADC3 as analog input channel
```

### 17.6.1 ADC Input Channels

When changing channel selections, the user should observe the following guidelines to ensure that the correct channel is selected:

In Single Conversion mode, always select the channel before starting the conversion. The channel selection may be changed one ADC clock cycle after writing one to ADSC. However, the simplest method is to wait for the conversion to complete before changing the channel selection.

In Free Running mode, always select the channel before starting the first conversion. The channel selection may be changed one ADC clock cycle after writing one to ADSC. However, the simplest method is to wait for the first conversion to complete, and then change the channel selection. Since the next conversion has already started automatically, the next result will reflect the previous channel selection. Subsequent conversions will reflect the new channel selection.

#### 17.13.1 ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0	
0x07	REFS1	REFS0	ADLAR	REFS2	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

**Table 17-4. Input Channel Selections**

MUX[3:0]	Single Ended Input
0000	ADC0 (PB5)
0001	ADC1 (PB2)
0010	ADC2 (PB4)
0011	ADC3 (PB3)

### Step 3: Step 5: Choosing Your Readout Options

*This step is very involved and explains in great detail how 10-bit resolution is resolved. If it seems to be too much, you can skip to the TL;DR section at the end which is a very brief synopsis of everything before it, which explains the single line of code that it builds up to.*



You may have noticed or asked the questions concerning the resolution of the ADC. On page 1 it says that this chip features 10-bit resolution (image 1).

For those who may not know. Each 1 and 0 is a *bit* of data. There are 8 bits to a *byte*. Each register contains 8 bits or exactly 1 byte. This is just a computer thing overall, and is not subject to change. So, this begs the question of how this chip features 10-*bit* resolution when a register only holds 1 *byte*. The answer is that the 10 bits are held between 2 registers, 8 in one, 2 in another. These registers are the ADCH and ADCL, (Analog to Digital Converter High Register and Analog to Digital Converter Low Register).

If you want to have 10 bits of resolution, the binary data is held over both registers with right adjusting. Both registers need to be read in the proper order, stored, combined, then used as the analog data you want. I will take a moment to quickly go over binary sequences and how binary is read. If you already understand how binary works, you can skip the section between the lines.

-----  
A number represented in binary is the sum of a series of 1's and 0's where each position in the sequence represents a power of 2. To find the decimal value of that binary number, you simply add up the powers of 2 in each location represented with a 1. A binary sequence can be of any length, but we will continue using 1 byte to represent the data.

Refer to (Image 2)

Row 1 is the number 98 shown below in binary.

Row 2 shows the values at each position within the sequence shown as powers of 2.

Row 3 shows the values in row 2 as whole numbers.

Row 4 shows that only positions with 1's have their corresponding values added to the overall number the binary sequence represents. In this case,  $64 + 32 + 2 = 98$ .

We say that the position all the way on the right is the first digit in the number then work left from there. This is why all the registers start at the right and work left.

-----  
This is significant when you are choosing 8-bit resolution for your result. This option is selected with the ADLAR (ADC Left Adjust Result) bit within the ADMUX Register (Image 3).

The reason this is all important becomes clear looking at the ADCL and ADCH (Image 4).

The ADLAR bit shifts the bits in the registers.

It is worth noting that no matter which you want, 8-bit or 10-bit resolution, your chip reads 10 bits worth of resolution which can represent numbers between 0 and 1023. 8-bit resolution can only represent numbers between 0 and 255. You just choose how much of that data you care about. Let's imagine that you took an analog value that the ADC tells you is 855 in binary. The bits from right to left look like this: 1101010111.

With ADLAR = 0, or right adjusted, between the two registers, your data looks like (Image 5).

The blank column represents separation between the registers. The most significant bits, ADC8 and ADC9 are held in the ADCH, and if you want an accurate representation of your analog value, you need these bits. This requires you to read both registers, combine them, then take that value from there.

With ADLAR = 1, or left adjusted, between the two registers, your data looks like (Image 6).

The blank column represents separation between the registers. In this case, the least significant bits are held in the ADCL, and to get an accurate representation of your analog value, you do not need them.

\*When you ignore 2 bits of data, you end up with a completely different binary number within a different range. Though the end result is still valid.\*

This can be shown using percentages. The highest value that can be held within 10 bits, is 1023, or, it can hold 1024 different numbers. 855 is our number. 855 is 83.5% of 1024. If we took only the 8 bits available from ADLAR = 0, we would

have 01010111, or 87, which is 34% of 256 (where 256 is the number of values held by an 8-bit sequence). Clearly 83.5 is not equivalent to 34.

But when looking at the 8 bits from  $ADLAR = 1$ , we would have 11010101, which is 213. This is 83.2% of 256. Notice that 83.5 is a lot closer to 83.2 than to 34. You end up losing less than half a percent of accuracy on your reading. Pretty good, huh?

Okay, take a moment to digest... What does this mean?

I know, I know. I'm finally getting to that.

### TL;DR:

*The ADC gathers 10-bit resolution, which is too big for a single register. Data is separated between two registers, ADCH and ADCL (2 bits in the first, 8 in the second), which is read in that order, (though data must be retrieved in the opposite order). The ADLAR bit in the ADMUX register, from 0 to 1, changes the adjustment from right to left. This shifts the binary sequence between the registers, such that instead of the 2 most significant bits alone in one register, you have the two least significant bits alone in the other (8 bits in the first, 2 in the second) where only ADCH needs to be read.*

The ADC gives you 10-bit resolution, regardless. 10-bit resolution is great, but you have to read 2 registers, so you'd have to get to 2 sets of data, combine, then use. Which is more complicated than necessary. 8-bit resolution is good enough and much easier. But! To get just 8-bit resolution means you have to ignore two bits of data that is gathered regardless. To ignore the proper bits without misrepresenting data, you have to change the adjustment from right to left, then only read the ADCH.

To do this, you need ADLAR to equal 1.

The following line of code accomplishes this:

```
ADMUX |= (1 << ADLAR); //Left Adjust the ADCH and ADCL registers
```

PHEW!!

Okay, now... You will recall from Part 1, that the above format for writing 1's and 0's to a register is the better way of doing so. I am now about to tell you otherwise.

Sorta...

As of right now, we have the following code lines for our project, all within the void setup().

```
ADMUX |= (1 << REFS0); //sets reference voltage to internal 1.1V
ADMUX |= (1 << MUX0); //combined with next line...
ADMUX |= (1 << MUX1); //sets ADC3 as analog input channel.
ADMUX |= (1 << ADLAR); //left adjusts for 8-bit resolution
```

The thing is, because the void setup() only happens once, and all four commands are to the same register, we can revert back to the original format for writing 1's and 0's to a register to combine all 4 commands into 1. The following code is an example of this:

```
ADMUX = 0b10100011; //sets 1.1V IRV, sets ADC3 as channel, left adjusts
```

This is only acceptable within the void setup() as the code is executed only one time.

The other *better* method is a must within the void loop(). Otherwise you end up writing 0's to other spots as well, and it horribly complicates things.

- **Peripheral Features**

- 8-bit Timer/Counter with Prescaler and Two PWM Channels
- 8-bit High Speed Timer/Counter with Separate Prescaler
  - 2 High Frequency PWM Outputs with Separate Output Compare Registers
  - Programmable Dead Time Generator
- USI – Universal Serial Interface with Start Condition Detector
- 10-bit ADC
  - 4 Single Ended Channels
  - 2 Differential ADC Channel Pairs with Programmable Gain (1x, 20x)
  - Temperature Measurement
- Programmable Watchdog Timer with Separate On-chip Oscillator
- On-chip Analog Comparator

0	1	1	0	0	0	1	0
$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
128	64	32	16	8	4	2	1
0	64	32	0	0	0	2	0

- **Bit 5 – ADLAR: ADC Left Adjust Result**

The ADLAR bit affects the presentation of the ADC conversion result in the ADC Data Register. Write one to ADLAR to left adjust the result. Otherwise, the result is right adjusted. Changing the ADLAR bit will affect the ADC Data Register immediately, regardless of any ongoing conversions. For a complete description of this bit, see ["ADCL and ADCH – The ADC Data Register" on page 137](#).

### 17.13.3.1 ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
0x05	–	–	–	–	–	–	ADC9	ADC8	ADCH
0x04	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

### 17.13.3.2 ADLAR = 1

Bit	15	14	13	12	11	10	9	8	
0x05	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
0x04	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

	1	1		0	1	0	1	0	1	1	1	
ADCH	ADC9	ADC8		ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	$2^9$	$2^8$		$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	



	1	1	0	1	0	1	0	1		1	1	
ADCH	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2		ADC1	ADC0	ADCL
	2 <sup>9</sup>	2 <sup>8</sup>	2 <sup>7</sup>	2 <sup>6</sup>	2 <sup>5</sup>	2 <sup>4</sup>	2 <sup>3</sup>	2 <sup>2</sup>		2 <sup>1</sup>	2 <sup>0</sup>	

## Step 4: Steps 6 and 7: Enabling the ADC and Choosing Analog Read Method

### Step 6:

This one is easy. In order to use the ADC, you need to enable it. This is done with the ADEN (ADC Enable) bit in the ADCSRA (ADC Control and Status Register A) Register. This is seen in 14.12.2 (Image 1).

The ADEN bit is the last bit in the register, and we want this to be a 1. So we input the following code in the void setup().

```
ADCSRA |= (1 << ADEN); //enables the ADC
```

### Step 7:

At this point, we need to decide which kind of data collection method we want. There are 3 different ways.

1. Single conversion (which we will concentrate on in this Instructable).
2. Free running mode.
3. Conversion using interrupts (we will not discuss this one).

Single conversion is basically how the analogRead() function works as used in the Arduino IDE. Specifically, it starts the conversion (which we will get to shortly) gathers the data, then stops the conversion. And then allows for data usage.

In free running mode, the ADC is always updating its data from the analog input channel. Then whenever you collect the data, it pulls the values from the registers at that moment.

Conversion using interrupts is as it sounds. But again, we will touch on this in a different Instructable.

The bits for this are also found in the ADCSRA. But the default setting is single conversion mode. And so, no change to the ADCSRA is needed for this example.

The effective difference between single conversion and free running mode in practice is:

In single conversion mode, you have to restart conversion within the ADC each time you wish to use it within the void loop().

In free running mode, you need only start conversion within the ADC once in the void setup(). Then just grab data as needed from the ADCH in the void loop(). Though, with this method, the ADC is always active and using power.

## 17.13.2 ADCSRA – ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0	
0x06	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

## Step 5: Step 8: Selecting Your Prescaler

This is pretty technical area to try and teach. So, we are going to go with the *because I said so* methodology. Early in the ADC Chapter we see the following paragraph (Image 1).

What this means is that the *ADC* clock needs to run slower than the *Chip* frequency clock. In our case, we are dealing with a 1 MHz chip clock. It says we want to have the ADC clock between 50 and 200. The easiest value to get to is 125 kHz.

To do this, we use the Prescaler bits, or the ADPS (ADC Prescaler Select) in the ADCSRA register., as shown in (Image 2).

What *this* means, is that by setting the bits, you define a division factor. This takes the Chip frequency and divides it by the division factor to determine ADC Clock Speed. For us, in this example, we want to divide our 1MHz by 8 to get to the 125kHz that we want for the best resolution sample.

This is done by writing 011 to ADPS[2,0].

This is done with the following lines of code in the void setup()

```
ADCSRA |= (1 << ADPS1); //with next line...
ADCSRA |= (1 << ADPS0); //set division factor-8 for 125kHz ADC clock
```

ADPS2 is initialized to 0, so no command needs to be written to it.

Now, at this point, we have the following lines of code for the ADCSRA in the void setup():

```
ADCSRA |= (1 << ADEN); //enables the ADC
ADCSRA |= (1 << ADPS1); //with next line...
ADCSRA |= (1 << ADPS0); //set division factor-8 for 125kHz ADC clock
```

Again, as the previous section showed, we can combine these commands into a single command using the old method as follows:

```
ADCSRA = 0b10000011; //turn on the ADC, keep ADC single conversion mode
//and set division factor-8 for 125kHz ADC clock
```

There are two more registers for ADC commands, we will use neither of them. So at this point, we are ready to move onto actually using and pulling data out of the ADC in the void loop().

So, without further ado.

## 14.5 Prescaling and Conversion Timing

By default, the successive approximation circuitry requires an input clock frequency between 50 kHz and 200 kHz to get maximum resolution. If a lower resolution than 10 bits is needed, the input clock frequency to the ADC can be higher than 200 kHz to get a higher sample rate.

### • Bits 2:0 – ADPS[2:0]: ADC Prescaler Select Bits

These bits determine the division factor between the system clock frequency and the input clock to the ADC.

**Table 14-4.** ADC Prescaler Selections

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

## Step 6: Steps 9 - 12: Collecting and Using Your Data

Collecting everything we have done so far, we have the following code as it would appear in the Arduino IDE.

```
#include <avr/io.h> //allows for register commands to be understood

void setup() {
  ADMUX = 0b01100011; //sets 1.1V IRV, sets ADC3 as input channel, and
    //left adjusts
  ADCSRA = 0b10000011; //turns on the ADC, keeps ADC in single conversion mode
} //and sets division factor to 8 for 125kHz ADC clock
```

Now we get to finally jump into the void loop() and collect some analog data. To do this you need to start a conversion. Remember, we are in single conversion mode. How to do this is shown in (Image 1).

We need to set the ADSC (ADC Start Conversion) bit in the ADCSRA register to 1. This is accomplished with the following code line in the void loop():

```
ADCSRA |= (1 << ADSC); //start conversion
```

The thing that may have you questioning is how to stop the conversion. Fortunately, the conversion stops itself. When you write a 1 to this bit, the conversion starts, collects the data into the ADCH and ADCL registers, then writes itself back to a 0 when it is done. The data in the registers remains static until it is either read, or overwritten.

The data is now stored in the ADCH and ADCL. When using 10-bit resolution, we find we need to read the registers in proper order. Page 94 shows this in (Image 2).

This also says that, for this example, that when left adjusted (which it for our example) it is sufficient to simply read the ADCH. The easiest way to retrieve the data is to simply set a variable equal to the ADCH. The following is an example.

```
analogData = ADCH; //store data in analogData variable
```

What is nice about this, is that even though the data is gathered as a binary number, the Arduino IDE switches seamlessly between binary, hexadecimal, as well as decimal. If you were to have the Serial interface enabled, you can simply print the analogData variable and see a decimal value between 0 and 255.

The following is a complete code to show everything together with Serial interface enabled to display analog data from a potentiometer.

```
#include <avr/io.h> //allows for register commands to be understood
int analogData; //declare analogData variable

void setup() {
  ADMUX = 0b01100011; //sets 1.1V IRV, sets ADC3 as input channel,
    //and left adjusts
  ADCSRA = 0b10000011; //turn on ADC, keep ADC single conversion mode,
    //and set division factor-8 for 125kHz ADC clock
  ADCSRA = 0b10000011; //turn on the ADC, keep ADC single conversion mode
    //set division factor-8 for 125kHz ADC clock
  Serial.begin(9600); //start Serial Interface
}

void loop() {
  ADCSRA |= (1 << ADSC); //start conversion
  analogData = ADCH; //store data in analogData variable
  Serial.print("analogData: "); //print "analogData: "
  Serial.println(analogData); //print data in analogData variable
  delay(1000); //delay 1 second
}
```

You will be happy to see that the analog data will be displayed in decimal form without any instruction from the user.

The vast majority of this Instructable is written just to set up the void setup() and understanding all the details for doing so properly. After that, collecting the data is pretty simple.

Stay tuned for Part 3.

### • Bit 6 – ADSC: ADC Start Conversion

In Single Conversion mode, write this bit to one to start each conversion. In Free Running mode, write this bit to one to start the first conversion. The first conversion after ADSC has been written after the ADC has been enabled, or if ADSC is written at the same time as the ADC is enabled, will take 25 ADC clock cycles instead of the normal 13. This first conversion performs initialization of the ADC.

ADSC will read as one as long as a conversion is in progress. When the conversion is complete, it returns to zero. Writing zero to this bit has no effect.

When ADCL is read, the ADC Data Register is not updated until ADCH is read. Consequently, if the result is left adjusted and no more than 8-bit precision is required, it is sufficient to read ADCH. Otherwise, ADCL must be read first, then ADCH.



Very good and in depth tutorial. Unfortunately I found an inconsistency between the two instructions :

1) ADMUX |= (1 << REFS1); //use internal reference voltage of 1.1V and

2) ADMUX = 0b01100011; //sets 1.1V IRV, sets ADC3 as channel, left adjusts

As shown in the ADMUX table, REFS1 is bit 7, therefore the second instruction should be

ADMUX = 0b10100011; //sets 1.1V IRV, sets ADC3 as channel, left adjusts

Beside that, congratulations for your excellent tutorial.



Thank you for the comment, I really did put some effort into writing this.

And thank you for that observation!

I've tried updating this to reflect the correction, but it doesn't seem to be taking.

I've tried about 9 times... But yes, you are very correct, and as soon as it lets me fix it, it will be fixed.



Very good tutorial. When I compile the above sketch with the Arduino IDE, I do get " 'Serial' was not declared in this scope " . Any suggestions?



Without seeing your code, the best I can suggest is to make sure you are using your serial commands properly. They are also case sensitive.

Serial.begin(9600); //this line is void setup()

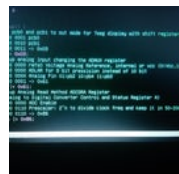
Serial.println();

Serial.print();



Thanks you for the detailed explanation, is good to understand why each line and not just copy and paste.

I haven't tested but I'm having a good time writing the code



Hey, I'm glad it made sense!

Let me know how things work out and if there are any issues with my code that you run into.



I made a 6 button input using the analog port and is working perfect. Thank you so much



I would love to see your setup and the code!

Keep on keeping on!



why is it ADMUX = 0x01100011; ? If it is binary, shouldn't it be 0b? (ADMUX = 0b01100011;)



Right, this is wrong in the article. "0x" is the prefix for a hex number. All direct initialization in the form of 0x and then 8 binary numbers should be in the form 0b and then the 8 binary numbers. Note this is not standard ANSI C, but it works in Atmel Studio.



The Instructable has been updated to show this correction. Thank you for the critique!



Very nice description (also the Part 1). Very well done, thank you!



Thanks for the tutorial !



Thank you. I hope it made sense and helped you out.



great !!, ... nice tutorial... cheers.



I appreciate the feedback.



Very good tutorial, congratulation!!!



Thank you, very much!