

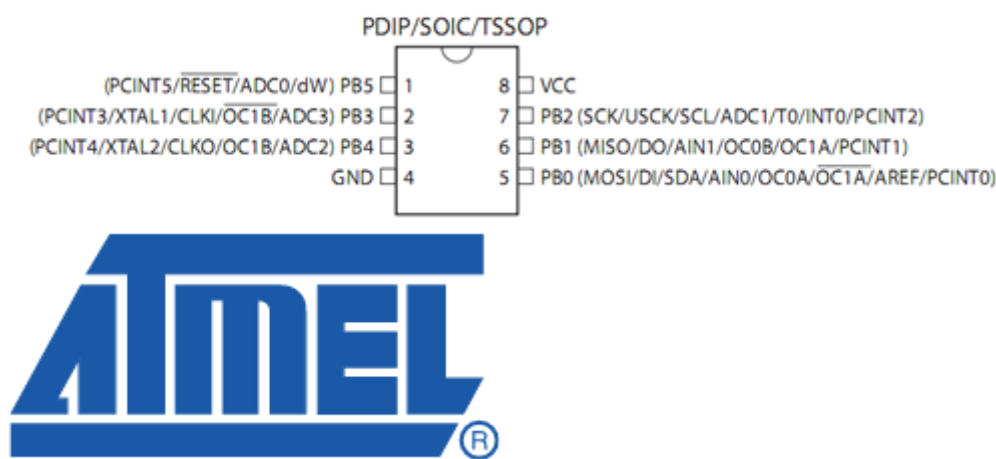
# ATtiny85 ADC

From wikipost  
Jump to navigationJump to search

This page provides a copy-and-paste working example of using the ADC (Analog-to-Digital Converter) in an Atmel ATtiny85 8-bit microcontroller.

With minimal adjustments this code should work on microcontrollers from the same family (such as the ATtiny25, ATtiny45, ATtiny24, ATtiny44 and ATtiny84).

Pinout ATtiny25/45/85



The main thing you will need to find out for your own microcontroller is the clock speed, as it determines the ADC sample rate. The ADC sample rate for this mcu needs to be between 50 - 200kHz and can be adjusted to fall within this range by means of setting the Prescaler bits in the ADCSRA register.

Example #1 code:

- assumes a clock speed of 8MHz on an ATtiny85
- uses 8-bit resolution (values from 0-255)
- uses ADC2 on pin PB4
- uses VCC as the reference voltage

```
void initADC()
{
    /* this function initialises the ADC

    ADC Prescaler Notes:
    -----

    ADC Prescaler needs to be set so that the ADC input frequency is between 50 - 200kHz.

    For more information, see table 17.5 "ADC Prescaler Selections" in
    chapter 17.13.2 "ADCSRA – ADC Control and Status Register A"
    (pages 140 and 141 on the complete ATtiny25/45/85 datasheet, Rev. 2586M-AVR-07/10)

    Valid prescaler values for various clock speeds

    Clock    Available prescaler values
    -----
    1 MHz    8 (125kHz), 16 (62.5kHz)
    4 MHz    32 (125kHz), 64 (62.5kHz)
    8 MHz    64 (125kHz), 128 (62.5kHz)
    16 MHz   128 (125kHz)
```

```

Below example set prescaler to 128 for mcu running at 8MHz
(check the datasheet for the proper bit values to set the prescaler)

*/

// 8-bit resolution
// set ADLAR to 1 to enable the Left-shift result (only bits ADC9..ADC2 are available)
// then, only reading ADCH is sufficient for 8-bit results (256 values)

ADMUX =
    (1 << ADLAR) | // left shift result
    (0 << REFS1) | // Sets ref. voltage to VCC, bit 1
    (0 << REFS0) | // Sets ref. voltage to VCC, bit 0
    (0 << MUX3) | // use ADC2 for input (PB4), MUX bit 3
    (0 << MUX2) | // use ADC2 for input (PB4), MUX bit 2
    (1 << MUX1) | // use ADC2 for input (PB4), MUX bit 1
    (0 << MUX0); // use ADC2 for input (PB4), MUX bit 0

ADCSRA =
    (1 << ADEN) | // Enable ADC
    (1 << ADPS2) | // set prescaler to 64, bit 2
    (1 << ADPS1) | // set prescaler to 64, bit 1
    (0 << ADPS0); // set prescaler to 64, bit 0
}

int main(void)
{
    initADC();

    while(1)
    {
        ADCSRA |= (1 << ADSC); // start ADC measurement
        while (ADCSRA & (1 << ADSC) ); // wait till conversion complete

        if (ADCH > 128)
        {
            // ADC input voltage is more than half of VCC
        } else {
            // ADC input voltage is less than half of VCC
        }
    }

    return 0;
}

```

## Reference Voltage

The reference voltage determines the upper limit of the ADC range. For example, if the ATtiny85 is powered with 5V and the ADC is configured to use Vcc as the reference voltage then the ADC is able to convert an (analogue) voltage between 0V and 5V. Within the whole range between 0V to 5V it is able to sample any voltage and convert that to an 8-bit (256 values) or 10-bit (1024 values) integer number. So if you were to provide 2.0V on pin PB4 you would get a value of 102 in 8-bit sampling (or 410 in 10-bit sampling).

According to the datasheet you can power your ATtiny85 from 1.8V to 5.5V (depending on frequency and model used). Specifying VCC as the reference voltage will then lead to a varying conversion value. For instance, powering your micro with 3.3V and putting the same 2.0V on PB4 will now result in a value of 155 in 8-bit sampling (or 621 in 10-bit sampling).

There is a simple provision in the ATtiny85 to make the varying VCC voltage (especially when powering directly off batteries) not affect the ADC readings. The chip provides a VCC-independent fixed 1.1V reference voltage. Check the REFS0 and REFS1 registers in the datasheet on how to enable these. You won't be able to measure our example 2.0V voltage, but a simple voltage divider will take care of that.

Alternatively a special-purpose reference voltage chip or linear voltage regulator can be added to your design to make sure the reference voltage doesn't vary too much.

## Sample Resolution

8-bit or 10-bit?

The sample resolution determines the final number that the ADC will store the result to. To refresh your memory, 8-bits are used to hold values from 0 to 255 and 10-bits are required to hold values from 0 to 1023. Since the ATtiny85 is an 8-bit micro it needs two registers to hold the final ADC value if you're using 10-bits sampling. In that case you will need to read the 'upper' and the 'lower' bytes, add them and treat them as an 16-bit integer in order not to lose any information.

Whether you need 8-bit or 10-bit sampling depends on your application. You will have a coarser granularity when sampling at 8-bits but it's easier to process and code and if your requirements do not demand very fine detection of ADC input voltages it may be all you need. One thing to keep in mind is the voltage range per sample value.

Minimum detectable voltage difference between steps (0-5V):

at 8-bit sampling: 19.53mV

at 10-bit sampling: 4.88mV

Example #2 code:

- assumes a clock speed of 1MHz on an ATtiny85
- uses 10-bit resolution (values from 0-1024)
- uses ADC2 on pin PB4 (pin 3)
- uses Vcc as the reference voltage

```
void initADC()
{
    /* this function initialises the ADC

    For more information, see table 17.5 "ADC Prescaler Selections" in
    chapter 17.13.2 "ADCSRA – ADC Control and Status Register A"
    (pages 140 and 141 on the complete ATtiny25/45/85 datasheet, Rev. 2586M-AVR-07/10)

    // 10-bit resolution
    // set ADLAR to 0 to disable left-shifting the result (bits ADC9 + ADC8 are in ADC[H/L] and
    // bits ADC7..ADC0 are in ADC[H/L])
    // use uint16_t variable to read ADC (instead of ADCH or ADCL)

    */

    ADMUX =
        (0 << ADLAR) | // do not left shift result (for 10-bit values)
        (0 << REFS2) | // Sets ref. voltage to Vcc, bit 2
        (0 << REFS1) | // Sets ref. voltage to Vcc, bit 1
        (0 << REFS0) | // Sets ref. voltage to Vcc, bit 0
        (0 << MUX3) | // use ADC2 for input (PB4), MUX bit 3
        (0 << MUX2) | // use ADC2 for input (PB4), MUX bit 2
        (1 << MUX1) | // use ADC2 for input (PB4), MUX bit 1
        (0 << MUX0); // use ADC2 for input (PB4), MUX bit 0

    ADCSRA =
        (1 << ADEN) | // Enable ADC
        (1 << ADPS2) | // set prescaler to 16, bit 2
        (0 << ADPS1) | // set prescaler to 16, bit 1
        (0 << ADPS0); // set prescaler to 16, bit 0
}
```

```

}
int main(void)
{
    initADC();

    uint8_t adc_lobyte; // to hold the low byte of the ADC register (ADCL)
    uint16_t raw_adc;

    while(1)
    {
        ADCSRA |= (1 << ADSC);          // start ADC measurement
        while (ADCSRA & (1 << ADSC) ); // wait till conversion complete

        // for 10-bit resolution:
        adc_lobyte = ADCL; // get the sample value from ADCL
        raw_adc = ADCH<<8 | adc_lobyte; // add lobyte and hibernate

        if (raw_adc > 512)
        {
            // ADC input voltage is more than half of the internal 1.1V reference voltage

        } else {

            // ADC input voltage is less than half of the internal 1.1V reference voltage

        }

    }

    return 0;
}

```

## Accuracy and Averaging

If possible, take multiple samples and average them out. Assuming that the ADC always provides a reliable value at the first attempt may lead to undesired consequences. For one of my voltage-datalogger applications the main ADC sampling loop consisted of:

- sample 5 readings
- sort these from high to low
- discard the top and bottom
- average the remaining three

Alternatively, the following function can be used to sample and average many readings into only a few lines of code:

```

// take 100x 8-bit samples and calculate a rolling average of the last 15 samples

float voltage_fl;          // real battery voltage (0-5V) with decimals
float adc_step=0.01953; // (0-5V over 256 values)
int sample_loop;

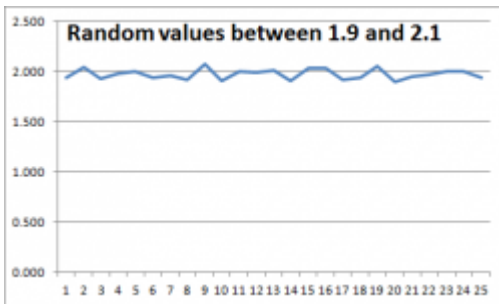
for (sample_loop=100; sample_loop > 0 ; sample_loop --)
{
    ADCSRA |= (1 << ADSC);          // start ADC measurement
    while (ADCSRA & (1 << ADSC) ); // wait till conversion complete

    voltage_fl = voltage_fl + (((ADCH * adc_step) - voltage_fl) / 15); // integrated last 15-sample rolling
    average
}

```

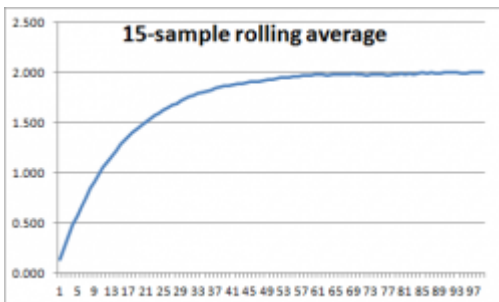
}

Does 100 samples sound like a bit much? Have a look here for what the averaging function does when it is given random values between 1.9 and 2.1.



A close-up of some of the random values



And this is what happens after running the averaging function 100 times.



Last 15 values are rolling averaged

As you can see it takes a while to settle. This is mainly caused by the number of rolling average samples (here 15) and the gradual 'building up' of the values when starting from 0. Ideally the very first ADC reading would not be zero, but to be on the safe side we have taken this precaution.

#### Resources:

-  File:Attiny85-summary.pdf ATtiny85 Summary Datasheet (30 pages, 692kB)
-  File:ATtiny85.pdf ATtiny85 Complete Datasheet (236 pages, 4.59MB)
- <http://www.microchip.com/wwwproducts/en/ATtiny85> Official ATtiny85 product page

#### See also:

- C and C++
- ATtiny84\_ADC
- ATtiny85\_PWM

Retrieved from "[http://www.marcelpost.com/wiki/index.php?title=ATtiny85\\_ADC&oldid=3205](http://www.marcelpost.com/wiki/index.php?title=ATtiny85_ADC&oldid=3205)"

- This page was last edited on 17 November 2019, at 15:34.