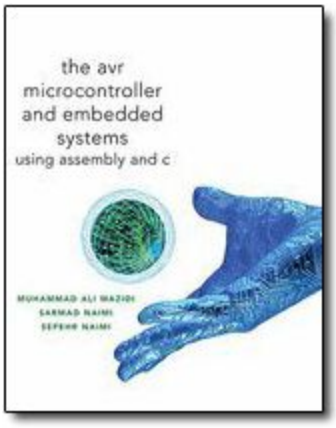
Analog-to-Digital Conversion



Chapter 13 ADC, DAC, and Sensor Interfacing

ATMEL 8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash
http://www.atmel.com/dyn/resources/prod_documents/doc8161.pdfasd
Chapter 23 “Analog-to-Digital Converter”

Table of Contents

References

1. 8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash - ATmega328P, Chapter 23 "Analog-to-Digital Converter", ATMEL document doc8161
2. AVR Freaks Design Note #021 "Using the Built-in ADC in AVR"
3. Successive Approximation ADC, Georgia State University, Department of Physics and Astronomy
4. Successive approximation ADC, Wikipedia

ATmega328P ADC Subsystem Features

- The ATmega48PA/88PA/168PA/328P features a 10-bit successive approximation ADC.
- 0.5 LSB Integral Non-linearity
- ± 2 LSB Absolute Accuracy
- **15 k samples** per second (kSPS) at Maximum Resolution and up to 76.9 kSPS (13 µs Conversion Time)
- 6 Multiplexed Single Ended Input Channels
- Temperature Sensor Input Channel
- Free Running or Single Conversion Mode
- Interrupt on ADC Conversion Complete

How It Works

What is a Successive Approximation ADC?

Illustrations from Tocci, Ronald J., Digital Systems, 5th Ed, Prentice-Hall, 1991.

A **successive approximation ADC** is a type of analog-to-digital converter that converts a continuous analog waveform into a discrete digital representation via a binary search through all possible quantization levels before finally converging upon a digital output for each conversion.

The successive approximation Analog to digital converter circuit typically consists of four chief subcircuits. Figure 1 is an example 4-bit ADC and will be used to illuminate how these four subcircuits work together to convert an analog value into a digital number.

1. A sample & hold **comparator** circuit acquires the analog input voltage (Vs) and compares it to the output of an internal DAC with input reference voltage (Vref). To keep the illustration as simple as possible, this reference voltage is not shown and may be assumed to be equal to 15 v. In our example 4-bit DAC the analog input voltage (Vs) is set to 7.2 volts.
2. The result of the comparison is sent to a successive approximation register (SAR). Identified as **control logic** and bits **D3** to **D0** in our simplified 4-bit DAC block diagram.
3. The internal DAC supplies the comparator with an analog voltage equivalent of the digital code output of the SAR for comparison with Vs.
4. The SAR subcircuit, a finite state machine, implements the algorithm defined in Figure 2.

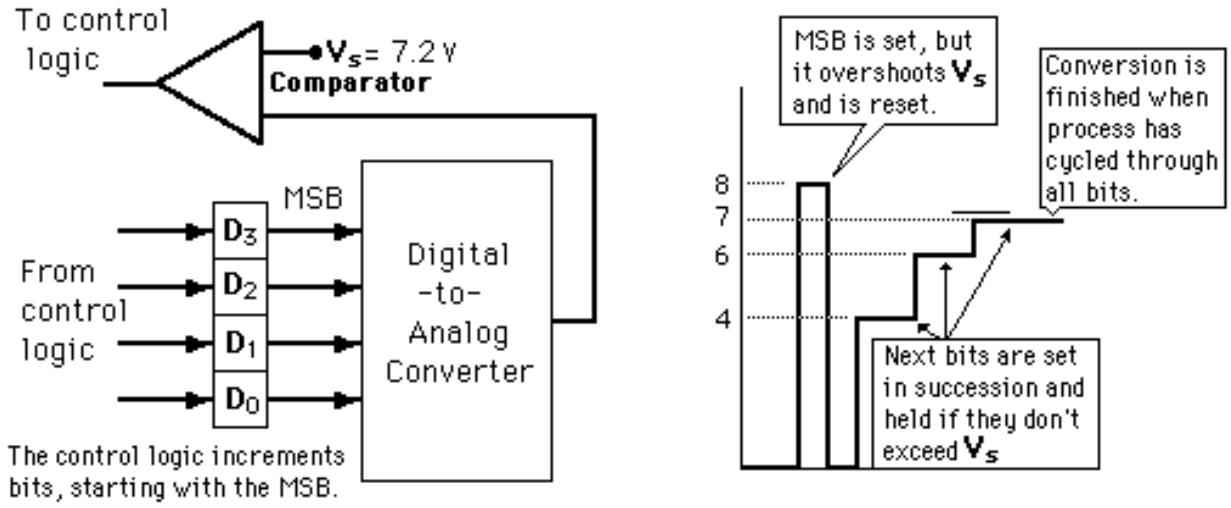


Figure 1 Simplified Block Diagram of a 4-bit ADC

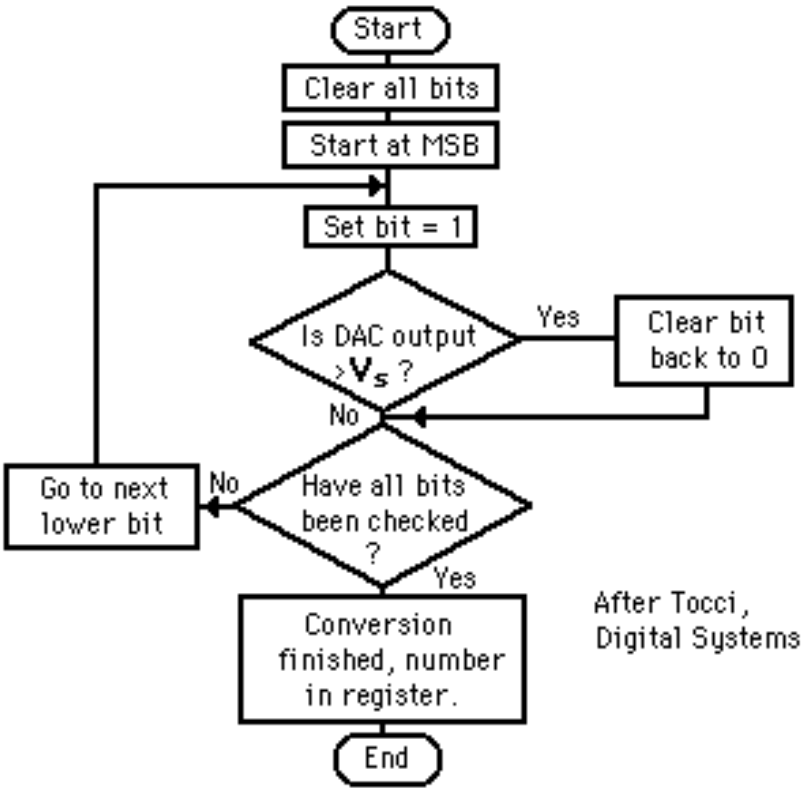


Figure 2 SAR Algorithm

Using the Built-in ADC in AVR

The Atmel ATmega328P datasheet provides everything you need to know to use the ADC subsystem of our AVR microcontroller. In the following sections I am going to focus on those topics which I consider to be most relevant when wanting to use the ADC:

- How to connect the pins related to the ADC (Voltage Reference).
- How to make an Analog to Digital conversion within the Arduino IDE.
- A Simple Analog to Digital Conversion (**`analogRead`**)
- The registers of the ADC (ADMUX, ADCSRA, and ADCH:ADCL). ADC registers ADCSRB and DIDR0 are left at default values and considered outside the scope of this introductory lesson.
- How to select an operating mode (Single Conversion and Free-Running)
- How to specify resolution/conversion speed (Sample Frequency).
- How to verify conversion complete (polling the ADSC bit).

I have tried to weave these topics into a single story centered around the Arduino Uno and the **`analogRead`** function. Consequently, section headings are more for future reference and for the most part can be ignored if you are reading the material from beginning to end.

Any good story must start with the big picture and ours is no exception. In Figure 3 we have the block diagram of the ADC subsystem of the AVR microcontroller.

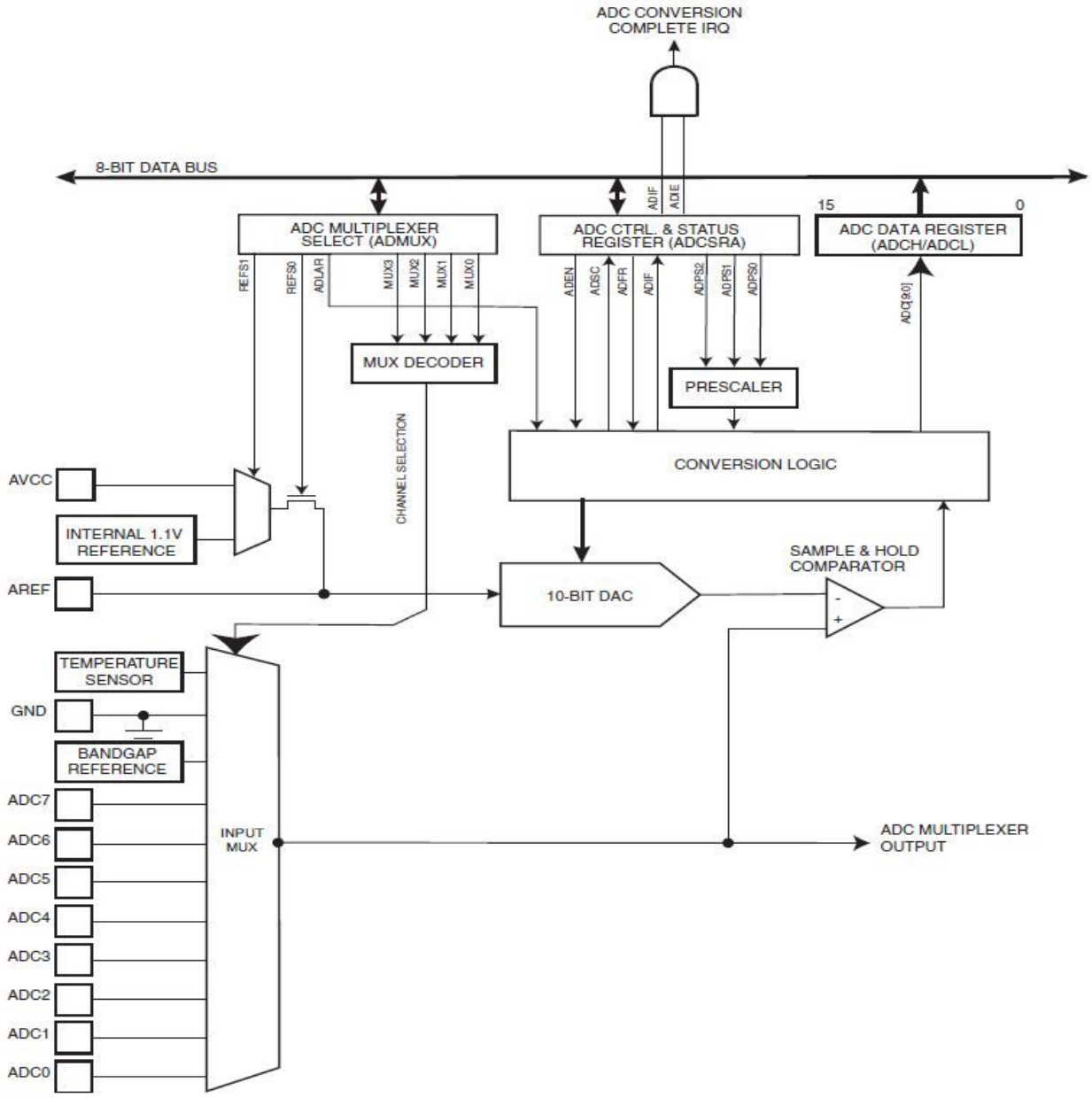


Figure 3 AVR Analog to Digital Converter Block Schematic Operation

Note: ADCSRA ADATE signal mislabeled as ADFR in Figure 3.

Voltage Reference (VREF)

While most of our story centers around how to work with the registers of the ADC. In this section I am going to talk about how the Arduino hardware handles the reference voltage for the 10-bit DAC (AVCC to VREF), how you can improve the noise immunity of your design, and finally how you can change the reference voltage (AVCC to Vref). If you choose to do the latter, make sure you read the warning (or have a spare ATmega328P on hand).

The minimum value of the 10-bit output of the ADC register (0x000) represents AGND and the maximum value (0x3FF) represents the voltage on the VREF line, within 1 LSB. You can think of VREF as normalizing the input voltage as defined by the following equation.

$$ADC = \frac{V_{IN} \cdot 1024}{V_{REF}}$$

> If you want to get into the details, I would recommend...
> An ADC and DAC Least Significant Bit (LSB)
> by *Adrian S. Nastase*

As shown in Figure 3, the reference voltage to the 10-bit DAC (VREF) can be sourced from AREF, AVCC, or an internal 1.1V reference voltage. We are using the Arduino and so are limited by how they have wired these pins. Comparing this schematic with Figure 23-9 "ADC Power Connections" in the Reference Data Sheet of the ATmega328P, we see that this is not the optimal wiring solution. As one example, the Arduino circuit is missing 10 µH inductor between AVCC and VCC. Two more examples are provided on the next page.

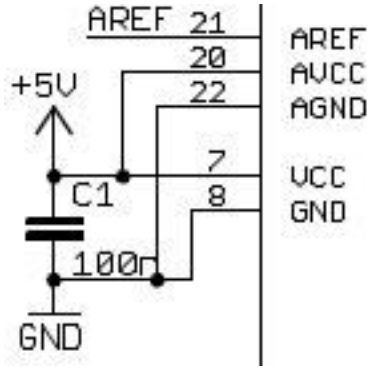


Figure 4 How the Arduino wires the ADC reference voltage pins

The source of the reference voltage is set by bits REFS1 and REFS0 in the ADC Multiplexer Select (ADMUX) register as defined in Table 1.

Bit	7	6	5	4	3	2	1	0	
(0x7C)	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The default setting for an **`analogRead`** call within the Arduino IDE sets these bits to REFS1 = 0, REFS0 = 1. Looking at Table 1 and Arduino schematic (Figure 4), we see that VREF = AVCC = 5v. For voltage reference mode 01_2 and 11_2 (Table 1) ATMEL recommends an external capacitor be connected to the AREF pin to improve noise immunity, without specifying the capacitor to be used. As seen in the Arduino schematic, the Arduino does not come with this

AREF capacitor. From a quick search of the web it appears that a capacitor value between 10nF to 100nF (**.01 µF** to 0.1 µF) is typically used. The only ATMEL source I could find recommends a "typical value" of 10 nF.

Table 1 Voltage Reference Selections for ADC

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal V_{ref} turned off
0	1	AV_{CC} with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 1.1V Voltage Reference with external capacitor at AREF pin

Changing the Reference Voltage

The Arduino provides a function named **`analogReference(uint8_t mode)`** which allows you to change the DEFAULT value of voltage reference source (REFS = 01_2).

```
void analogReference(uint8_t mode)
{
  // can't actually set the register here because the default setting
  // will connect AVCC and the AREF pin, which would cause a short if
  // there's something connected to AREF.
  analog_reference = mode;
}
```

Also included in the wiring.h header file are the three available values defined as constants (compare names below to Table 1).

```
#define EXTERNAL 0
#define DEFAULT  1
#define INTERNAL 3
```

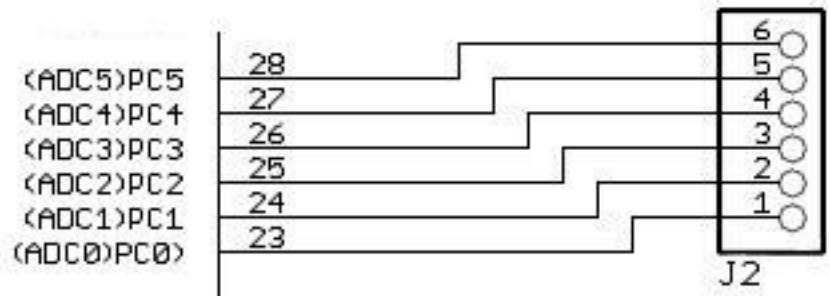
For example, if you want to reference an external voltage wired to the AREF pin you would add the following code to the setup section of your Arduino sketch. In this example, AREF is wired to the 3.3 V output provided by the Arduino.

```
analogReference(EXTERNAL);     // where Vref is 3.3V
```

WARNING: If you wire a voltage directly to AREF, when the Arduino changes the ATmega328P default setting of 00_2 to the Arduino's default value of 01_2, a short will exist between AVCC and AREF lines as shown in Figure 3. Even though your setup script will switch it back to its original safe 00_2 value the damage will already have been done to the ATmega328P (i.e., time to buy a new microcontroller). To protect the ATmega328P during this short period of time, add a **1K ohm** resistor between your reference voltage source and the AREF pin. During normal operation the voltage drop across this resistor should be negligible.

How to make an Analog to Digital conversion within the Arduino IDE

Converting an Analog signal into its digital equivalent is accomplished within the Arduino IDE using the **`analogRead(pins)`** function. The **`analogRead`** function takes a single argument "pin", identifying one out of the six Analog pins of the Arduino Uno to be read.



Using the **`analogRead`** function is demonstrated by the AnalogInput sketch (Open - Analog - AnalogInput)

```
/*
 Analog Input
 Demonstrates analog input by reading an analog sensor on analog pin 0 and
 turning on and off a light emitting diode(LED) connected to digital pin 13.
 The amount of time the LED will be on and off depends on
 the value obtained by analogRead().

 The circuit:
 * Potentiometer attached to analog input 0
 * center pin of the potentiometer to the analog pin
 * one side pin (either one) to ground
 * the other side pin to +5V
 * LED anode (long leg) attached to digital output 13
 * LED cathode (short leg) attached to ground

 * Note: because most Arduinos have a built-in LED attached
 to pin 13 on the board, the LED is optional.

 Created by David Cuartielles
 Modified 16 Jun 2009
 By Tom Igoe

 http://arduino.cc/en/Tutorial/AnalogInput

 */

int sensorPin = 0;    // select the input pin for the potentiometer
int ledPin = 13;      // select the pin for the LED
int sensorValue = 0; // variable to store the value coming from the sensor
```

```
void setup() {
  // set pin(s) to input and output
  pinMode(sensorPin + A0, INPUT);
  // declare the ledPin as an OUTPUT:
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // read the value from the sensor:
  sensorValue = analogRead(sensorPin);
  // turn the ledPin on
  digitalWrite(ledPin, HIGH);
  // stop the program for <sensorValue> milliseconds:
  delay(sensorValue);
  // turn the ledPin off:
  digitalWrite(ledPin, LOW);
  // stop the program for for <sensorValue> milliseconds:
  delay(sensorValue);
}
```

In the next section we will look at how the **analogRead** function works. Our story is about to become a lot darker.

A Simple Analog to Digital Conversion

Before we get into the details of how the Arduino **analogRead** function works let me give you a thumbnail sketch of the process. If you find yourself getting lost in the forest, you may want to reread the following paragraph. Figure 5 "ADC Registers" is provided here to help you through the mnemonic soup (ADSC, ADCSRA, etc.) contained in this summary paragraph.

> The Arduino **analogRead** function performs a simple analog conversion, where the ADC is triggered manually by setting the ADSC bit to logic one in the ADCSRA register. The ADSC bit will read as logic one as long as a conversion is in progress. When the conversion is complete, it returns to zero. The **analogRead** function polls this bit and returns the 10-bit result in the ADCH:ADCL register pair when conversion is complete.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
(0x7E)	DIDR0	–	–	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D	266
(0x7D)	Reserved	–	–	–	–	–	–	–	–	
(0x7C)	ADMUX	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	262
(0x7B)	ADCSRB	–	ACME	–	–	–	ADTS2	ADTS1	ADTS0	265
(0x7A)	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	263
(0x79)	ADCH	ADC Data Register High byte								265
(0x78)	ADCL	ADC Data Register Low byte								265

Figure 5 ADC Registers

Now, let's take a look under-the-hood at the **analogRead** function. I am assuming the use of the Arduino Uno. Consequently, to simplify the explanation of this function I have removed a

macro expansion required by the Arduino MEGA.

```
int analogRead(uint8_t pin)
{
    uint8_t low, high;

    // set the analog reference (high two bits of ADMUX) and select the
    // channel (low 4 bits).  this also sets ADLAR (left-adjust result)
    // to 0 (the default).
    ADMUX = (analog_reference << 6) | (pin & 0x0f);

    // without a delay, we seem to read from the wrong channel
    //delay(1);

    // start the conversion
    sbi(ADCSRA, ADSC);

    // ADSC is cleared when the conversion finishes
    while (bit_is_set(ADCSRA, ADSC));

    // we read ADCL first; doing so locks both ADCL
    // and ADCH until ADCH is read.  reading ADCL second would
    // cause the results of each conversion to be discarded,
    // as ADCL and ADCH would be locked when it completed.
    low = ADCL;
    high = ADCH;

    // combine the two bytes
    return (high << 8) | low;
}
```

Exercise 1: The C++ `sbi` instruction is now deprecated. Write one line of C++ code to replace this instruction.

The registers of the ADC

Although the ADC subsystem of the ATmega328P microcontroller contains six (6) registers (see Figure 5), only ADMUX, ADCSRA, ADCH, are ADCL are used to the configure the ADC as used by the Arduino `analogRead` function. We will look at ADCSRB and DIDR0 shortly.

ADC Multiplexer Selection Register Initialization

Assuming the unsigned 8-bit argument pin in the analogRead(pin) function call is zero (00_{16}), the C++ line...

```
ADMUX = (analog_reference << 6) | (pin & 0x0f);
```

...at the beginning when the Arduino bootstrap loader initializes the Analog subsystem of the ATmega328P microcontroller, the ADC ADMUX register is set to 0100_0000_2.

Bit	7	6	5	4	3	2	1	0	
(0x7C)	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	1	0	0	0	0	0	0	

Figure 6 ADC Multiplexer Selection Register Control

Let's take a closer look at each of these bits and what part they play in our story.

As originally covered in the "Voltage Reference" section of our lesson and summarized in Table 1 "Voltage Reference Selections for ADC", setting **`REFS1 = 0`** and **`REFS0 = 1`** means that our reference voltage VREF is equal to VCC, which we will assume is 5v.

The ADC generates a 10-bit result which is stored in the ADC Data Registers, ADCH and ADCL. By default, the result is presented right adjusted (**`ADLAR = 0`**), but can optionally be presented left adjusted by setting the ADLAR bit in ADMUX to logic one. For the analogRead function the result of the conversion process is right adjusted (DEFAULT value).

Exercise 2: Add unsigned 8 bit integer parameter `left_adjust` to the `analogRead` function (i.e, `analogRead(uint8_t pin, uint8_t left_adjust)`. Use left_adjust to set or clear the ADLAR bit in ADMUX. Your function should insure all undefined bits (bits 7 to 1) are set to zero.

Using Different Channels as seen from the Figure 3 schematics, the ADC can access multiple analog channels through the MUX. The ADMUX Register controls the MUX and the different input pins which can be directed to the sample-hold circuit. The sample-hold circuit keeps the sampled voltage level stable while the conversion is made, using successive approximation. The analogRead functions pin parameter allows the calling program to define which pin is to be read. The Arduino Uno uses the ATmega328P processor packaged in a PDIP (See Figure 1-1 "Pinout ATmega48PA/88PA/168PA/328P" in the data sheet). This limits us to 6 analog inputs (ADC5 to ADC0). For my example, the analog signal is read on pin ADC0. Consequently, the input to the MUX is set to 0 (**`MUX3 = 0, MUX2 = 0, MUX1 = 0, MUX0 = 0`**).

ADC Control and Status Register A Initialization

During initialization (see **`init()`** subroutine in **`main(void)`**), the Arduino sets ADCSRA to 1000_0111_2.

Bit	7	6	5	4	3	2	1	0	
(0x7A)	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	1	0	0	0	0	1	1	1	

Figure 7 How the ADC Control and Status Register A, is initialized by the **`init()`** subroutine

The bits of the ADCSRA register are used to set the operating mode and sample frequency of

the ADC. We will look at the "single conversion" and "free-running" modes of operation. These are the two most common operating modes, although there are others. For example you can program the ADC to sample an analog line at a fixed interval of time.

How to select an operating mode

Single Conversion Mode

Before starting a conversion the ADC must be enabled. This is done by setting the bit **`ADEN = 1`** in the ADCSRA Register. As illustrated in Figure 7 this is taken care of by Arduino **`init()`** subroutine.

The ADC can be run in two modes: Single Conversion mode or Free-running mode. In Single Conversion mode the conversion is initiated manually, by setting the **`ADSC`** bit in the ADCSR Register. Arduino's **`analogRead`** function uses the single conversion mode as shown in the following inline assembly instruction.

```
// start the conversion
sbi(ADCSRA, ADSC);
```

Free-Running Mode

One of the disadvantages of the Arduino IDE is that we trade flexibility for simplicity. To run the ADC in Free-Running mode it may be best to make your own custom version of `analogRead` or to write your own ADC C++ within a more powerful IDE like Eclipse, AVR Studio, or ATMEL Studio.

Bit	7	6	5	4	3	2	1	0	
(0x7B)	–	ACME	–	–	–	ADTS2	ADTS1	ADTS0	ADCSRB
Read/Write	R	R/W	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 8 ADC Control and Status Register A

Note: The *Analog Comparator Multiplexer Enable* ACME bit is not used by the ADC circuit and should be kept at its default value of 0.

In Free-Running mode the ADC is setup to start a new conversion immediately after the previous conversion is complete. To enable this feature, the *ADC Auto Trigger Enable* **`ADATE`** bit in ADCSRB (Figure 6) is set to one and *ADC Auto Trigger Source* ADTS[2:0] bits in ADCSRB (Figure 8) are set to zero. This selects the ADC interrupt flag **`ADIF`** as the trigger source. The ADC now operates in Free Running mode, constantly sampling and updating the ADC Data Registers.

The first conversion is started by writing a logical one to the ADSC bit in ADCSRA. If you use this mode, I would recommend enabling the I-bit in SREG and ADC Interrupt (**`ADIE`** = 1) bit in ADCSRA. Now your Interrupt Service Routine (ISR) can take action each time a conversion is complete (no more polling).

How to specify resolution/conversion speed (Sample Frequency)

The sample frequency of the AVR ADC is a function of the operating mode (single conversion or free-running), the ATmega328 system clock frequency (CK = 16 MHz), and the prescale setting (ADCSRA bits ADPS2 to ADPS0).

From when a single conversion is initiated to the result is in the ADCH:ADCL Registers 13 ADC cycles will pass as defined in Table 2.

Table 2 ADC Conversion Time

Condition	Sample & Hold (Cycles from Start of Conversion)	Conversion Time (Cycles)
First conversion	13.5	25
Normal conversions, single ended	1.5	13
Auto Triggered conversions	2	13.5

The "First conversion" line refers to when the ADEN bit in ADCSRA register is set. This additional time is required to initialize the analog circuitry including the 7-bit ADC prescaler (see Figure 9 "ADC Prescalar"). The Arduino sets the ADEN bit in the **`init()`** subroutine.

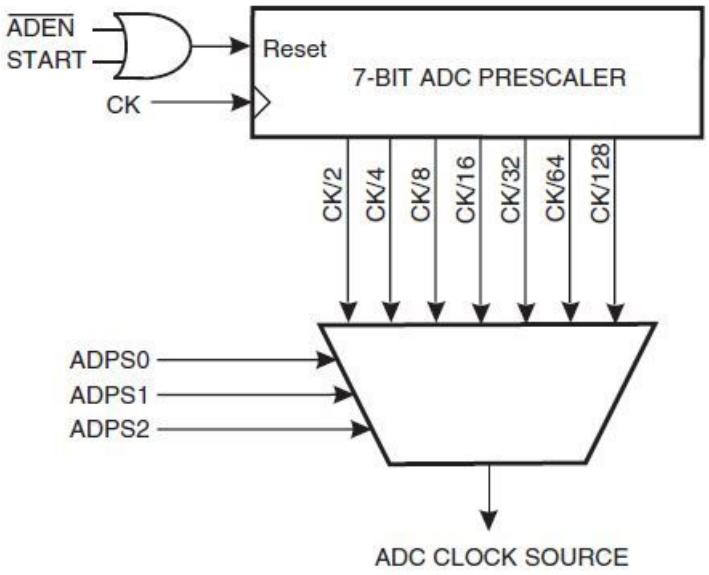


Figure 9 ADC Prescaler

Table 3 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

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 acceptable, the input clock frequency to the ADC can be higher than 200 kHz to get a higher sample rate. The Arduino **`init()`** routine sets prescaler bits ADPS2 to ADPS0 in the ADCSRA Register to all ones (see Figure 7). Which means that:

$f_{ADC} = f_{CK}/128 = 16\ \text{MHz}/128 = 125\ \text{kHz}$

Consequently, the ADC prescaler of the Arduino generates an ADC clock frequency of 125 kHz (less than 200 kHz), giving us a maximum resolution of 10 bits at a sample frequency of:

$f_{sample} = f_{ADC}/13 =$ **9.615 ksps (104 µs Conversion Time)**

What would happen if we set our prescaler to divide the system clock by 64? An ADC clock at 250 kHz violates the limit in ADC clocking for full 10 bit resolution (ADC clock 200 kHz for 10-bit resolution). Considering the Table 28-7 regarding “ADC Characteristics” Section 28.8 in the data sheet, the absolute accuracy would probably be 2 - 3 LSB. This means that the ADC can be considered to be a 9-bit ADC since the LSB is not reliable. This could be a reasonable trade in exchange for a sample frequency of 19.231 ksps (52 µs Conversion Time). Once again if you are thinking of pushing the limits, it may be best to write your own ADC C++ program in a more powerful IDE like Eclipse, AVR Studio, or ATMEL Studio.

How to verify conversion complete (polling the ADSC bit)

When we last left our Arduino **`analogRead`** routine, it had started the analog to digital conversion process by setting the ADEN bit in the ADCSRA Register to logic one.

```
// start the conversion
sbi(ADCSRA, ADSC);
```

The analogRead routine now polls the ADSC bit in the ADCSRA register whithin a C++ while

loop.

```
// ADSC is cleared when the conversion finishes
while (bit_is_set(ADCSRA, ADSC));
```

Once the conversion is complete, the ADC hardware subsystem of the ATmega328P clears the ADSC bit and stores the result in the ADCH:ADCL register pair.

Bit	15	14	13	12	11	10	9	8	
(0x79)	–	–	–	–	–	–	ADC9	ADC8	ADCH
(0x78)	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	

Figure 10 ADC Data Registers ADCH and ADCL

It is important to read ADCL first to ensure that valid data is read. By reading ADCL first, the ADCL and the ADCH Registers are “locked” and cannot be updated by the ADC until the ADCH has been read.

```
low = ADCL;
high = ADCH;
```

Our story is finally over, when the high order byte is shifted 8 places to the left and or'd with the low order byte. The 10 bit answer (zero extended to 16 bits) is now returned as a positive integer (16-bit signed).

```
// combine the two bytes
return (high << 8) | low;
```

DIDR0 – Digital Input Disable Register 0

Although it does not play a part in the Arduino IDE, your own design should initialize the DIDR0 register, as defined here. The 3DoT software does work with this register.

Bit	7	6	5	4	3	2	1	0	
(0x7E)	–	–	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D	DIDR0
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	

Figure 11 Digital Input Disable Register 0

When an analog signal is applied to the digital input buffer of a GPIO port pin, the buffer can consume an unnecessarily high rate of current as the input signal voltage stays within the undefined region between logic one and zero (see slew rate).

To solve this problem, the GPIO digital input buffer shared with the analog input buffer should be disabled by setting the corresponding bit in the *Digital Input Disable Register 0* DIDR0 register.

If you are wondering, ADC pins ADC7 and ADC6 do not have digital input buffers, and therefore do not require Digital Input Disable bits (i.e., ATmega328 only has 28 pins).

Exercise 3: What GPIO Port pin is shared with ADC input A4. Write C++ code to disable this digital input buffer.

Concluding Remarks

As we have seen the Arduino **`analogRead`** function provides a simple way for converting an analog signal into its digital equivalent. In EE470 “Digital Control” the experimentally measured conversion time the `analog_wire_spi.ino` script is 125 µsec (sample frequency = 8 ksps). From our discussion, we know that the conversion time of our Arduino script is 104 µsec. One thing we can take away from this finding is that over 83% of the time the script is waiting for the ADC subsystem.

Exercise 3: If you replaced the Arduino `analogRead` function in the `analog_wire_spi` script, with an interrupt driven routine and placed the ADC in Free-Running mode, what conversion time and sample frequency would you expect to achieve? Hint: see Table 2 “ADC Conversion Time.”

During our journey I hope you have also gained some insights into how we might turbo-charge the little Arduino. For example:

- Switch from Single Conversion to Free-Running mode and moving to an Interrupt driven solution, freeing the processor to do other tasks without sacrificing performance.
- Change your system clock frequency such that the ADC clock frequency can be set to 200 KHz. This will allow you to reach a sample rate of approximately **15 Kbps**.
- Overclock the ADC clock to 250 kHz yielding a sample frequency of 19.231 ksps (52 µs Conversion Time).