USC Viterbi School of Engineering

Ming Hsieh Department of Electrical Engineering

EE 109L - Introduction to Embedded Systems

Using the Atmel ATmega328P Analog to Digital Conversion Module

1 Introduction

The Atmel ATmega328P microcontroller used on the Arduino Uno has an analog-to-digital conversion (ADC) module capable of converting an analog voltage into a 10-bit number from 0 to 1023 or into an 8-bit number from 0 to 255. The ADC can convert signals at a rate of about 15 kSPS (samples per second.)

The sections below discuss the various conversion parameters that must be set in order to use the ADC module. All of these parameters are selected by setting or clearing bits in the module's registers.

2 Reference Voltages

In order to convert an analog voltage to a digital value on any ADC, the converter has to also be provided with the range of voltages it is expected to deal with. Otherwise it's like asking "How high is high?". The range is specified to the converter by supplying a low and a high reference voltage. The converter will then assign digital values linearly[1] between the minimum and maximum digital value as the input signal changes from the low reference to the high reference. If the input is the equal to (or below) the low reference, the output is all zeros. If it's equal to (or above) the high reference it outputs all ones, and if the input is midway between the low and high references the output would be the digital value in the middle of the numerical range.

On the ATmega328P the low reference is fixed at ground but it has the ability to select one of three sources to serve as the high reference.

AREF - This is a separate pin on the microcontroller that can be used to provide any high reference voltage you wish to use as long as it's in the range of 1.0V to V_{CC}. On the Uno it's wired to one of the black headers.

AVCC - (Use this one in the lab assignments.) This pin on the microcontroller provides power to the ADC circuitry on the chip. On the Uno it is connected to V_{CC}. This is the simplest option to use provided AVCC is connected to V_{CC}.

1.1V - The microcontroller has an internal 1.1V reference voltage that can be used.

3 Input Channel

The version of the ATmega328P used on the Arduino Uno has the ability to convert the analog signals from six different sources into digital values. The inputs to the ADC module appear on the Arduino board as connections A0 through A5. However since there is only one actual analog-to-digital converter in the microcontroller **only one channel** can be converted at a time. The choice of which of the six channels to use as input to the ADC is made by the program, and can be changed by the program whenever needed.

[1] ADCs can also be designed to do logarithmic conversions or other functions.

4 Conversion Rate

The ADC circuitry needs to have a clock signal provided to it in the range of 50kHz to 200kHz. There is no external ADC clock input on the chip but rather the clock is generated internally from same clock used to run the microcontroller. This CPU clock is usually too fast (16MHz on the Uno) so the chip includes an adjustable "prescaler" to divide the CPU clock down to something acceptable. The prescaler can be set to divide by a choice of divisors (2, 4, 8, 16, 32, 64, or 128) and it's up to the programmer to select one that results in a ADC clock within the acceptable range.

For example, if the ATmega328P is being used with a 10MHz CPU clock the prescaler can produce the following ADC clock frequencies.

Prescaler	ADC clock	
2	10MHz ÷ 2 = 5MHz	
4	10MHz ÷ 4 = 2.5MHz	
8	10MHz ÷ 8 = 1.25MHz	
16	10MHz ÷ 16 = 625kHz	
32	10MHz ÷ 32 = 312.5kHz	
64	10MHz ÷ 64 = 156.25kHz	← within range
128	10MHz ÷ 128 = 78.125kHz	← within range

As in the example above it's very possible that there may be more than one prescaler value that works and in that case either one can be used.

5 Registers

The interface to the ADC module from the software is through a group of registers. The descriptions below are very brief and only cover some of the functions of the registers. For full description see the ATmega328P datasheet. In the description below notation with brackets and a colon like "XYZ[2:0]" means the combination of the bits XYZ2, XYZ1 and XYZ0. For example if we say "XYZ[2:0] = 5", this means the binary value of 5 (101) has been assigned to the three XYZ bits (XYZ2 = 1, XYZ1 = 0, XYZ0 = 1).

ADMUX - Multiplexer Selection Register

The ADMUX register contains three fields of bits for controlling various aspects of the data conversion.

ADMUX

7	6	5	4	3	2	1	0
REFS1	REFS0	ADLAR		MUX3	MUX2	MUX1	MUX0

Bits 7:6 - Bits REFS1 and REFS0 control the selection of the high reference source.

REFS1	REFS0	High voltage selection	
0	0	AREF	
0	1	AVCC	← use this one in Lab 5
1	1	Internal 1.1V	

Bit 5 - The ADLAR bit determines whether to left adjust the result or right adjust it.

ADLAR	Conversion result
0	Right adjusted for 10 bit results
1	Left adjusted for 8 bit results

Bits 3:0 - The MUX3 through MUX0 bits control the selection of which of the six input lines to connect to the digitizing circuit.

MUX3	MUX2	MUX1	MUX0	Input to ADC module
0	0	0	0	Arduino A0 = Port C, bit 0
0	0	0	1	Arduino A1 = Port C, bit 1
0	0	1	0	Arduino A2 = Port C, bit 2
0	0	1	1	Arduino A3 = Port C, bit 3
0	1	0	0	Arduino A4 = Port C, bit 4
0	1	0	1	Arduino A5 = Port C, bit 5

ADCSRA - Control and Status Register A

The ADC module of the ATmega328P has two control and status registers, ADCSRA and ADCSRB. For basic ADC operations only the bits in the ADCSRA register have to be modified. For information on the ADCSRB register, see the ATmega328P data sheet.

ADCSRA

7	6	5	4	3	2	1	0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

This register has bits for initiating actions and reporting various conditions in the ADC module. The ones needed for this lab are these.

Bit 7 - The ADEN bit enables the ADC module. Must be set to 1 to do any ADC operations.

Bit 6 - Setting the ADSC bit to a 1 initiates a single conversion. This bit will remain a 1 until the conversion is complete. If your program using the polling method to determine when the conversion is compete, it can test the state of this bit to determine when it can read the result of the conversion from the data registers. As long as this bit is a one, the data registers do not yet contain a valid result.

Bit 3 - Setting the ADIE bit to a 1 enables interrupts. An interrupt will be generated on the completion of a conversion. The interrupt vector name is "`ADC_vect`". (Don't use this in Lab 5 unless directed otherwise.)

Bits 2:0 - The ADPS2, ADPS1 and ADPS0 bits selects the prescaler divisor value.

ADPS2	ADPS1	ADPS0	Prescaler value
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

ADCH and ADCL - Data Register (high and low bytes)

The results of a conversion are stored in the two bytes of the Data Register. The most significant bits of the result are stored in ADCH and the least significant bits are in ADCL. If you wish to use the full 10-bit conversion results, the ADLAR bit in the ADMUX register should be cleared to a zero. This causes the 10-bit conversion result to be right justified in the 16-bit combination of ADCH and ADCL (referred to in a program as ADC). The two most significant bits are in ADCH(bits 1 and 0) and the lower eight bits are in ADCL.

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

When using the 10-bit results (ADLAR=0), the results can be read in a program with code like this.

```
unsigned short x;

x = ADC;
```

This will result in the high order bits from ADCH being stored in the upper byte of the 16-bit variable, and the lower eight bits from ADCL being stored in the lower byte of the variable.

In many cases only an eight-bit conversion value is needed. For this situation, set the ADLAR bit in the ADMUX register to a one. The conversion results will be left justified with the eight most significant bits in ADCH. The 8-bit result can then be obtained by just reading the ADCH register and ignoring the contents of ADCL.

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

When using 8-bit results (ADLAR=1), the results can be read in a program with code like this.

```
unsigned char x;

x = ADCH;
```

6 Using the ADC

6.1 ADC Setup

The basic steps in setting up the ADC module to do conversions are these.

1. Configure the REFS[1:0] bits to select the high reference voltage to use. Using AVCC is recommended.
2. Set or clear the ADLAR bit depending on whether you are want to use 10 or 8 bit conversion results.
3. Configure the MUX[3:0] bits to select the input channel to be used.
4. Using the list of available prescaler values, determine what the ADC clock would be for each of them. Select one that gives an ADC clock in the range of 50kHz to 200kHz, and then configure the ADPS[2:0] bits to select this clock prescaler value.
5. Set the ADEN bit in ADCSRA to a one. This enables the ADC and you're now ready to initiate a conversion.

If using interrupts also do these steps.

6. Write an interrupt service routine (ISR) for the `ADC_vect` interrupt.
7. Set the ADIE bit in ADCSRA to a one to enable the module to interrupt.
8. Enable global interrupts with the `sei()` function call.

6.2 ADC Conversions Using Polling

To do conversions using polling, do these steps.

1. Set the ADSC bit in ADCSRA to a one. This starts a conversion.
2. Go into a loop checking the state of the ADSC bit. As long as it is still a one the conversion is in progress and your program should stay in the loop repeating the check. Once the ADSC bit becomes a zero the conversion is complete and you can break out of the loop.
3. Read the result from ADCH (8-bit) or ADC (10-bit).

6.3 ADC Conversions Using Interrupts

To do conversions using interrupts, do these steps.

1. Set the ADSC bit in ADCSRA to a one. This starts the first conversion.
2. Go into a loop doing nothing or perform some other task to wait until the interrupt occurs.
3. In the ISR, read the result from ADCH (8-bit) or ADC (10-bit). If you wish to start another conversion right away this can be done in the ISR by setting the ADSC bit again before returning from the ISR.