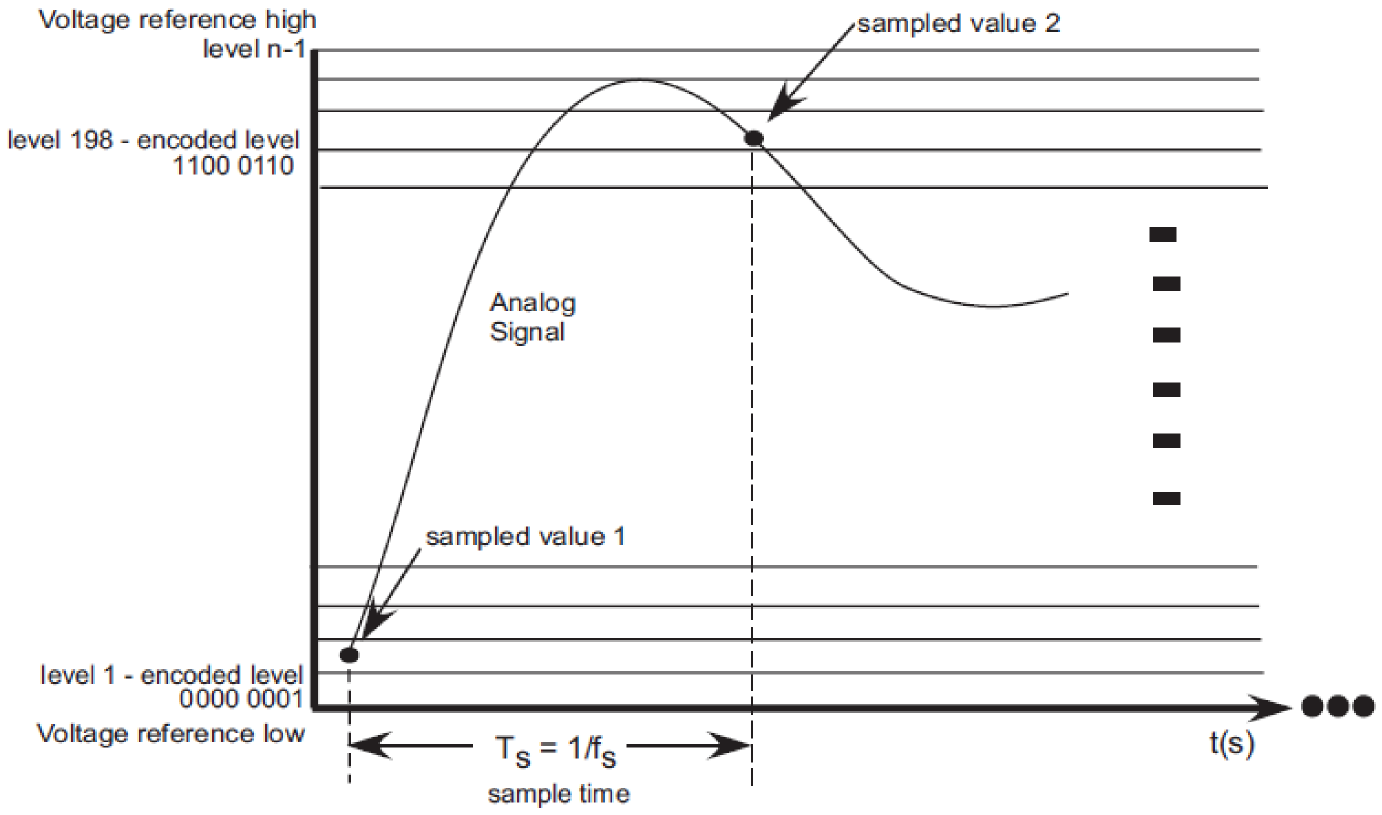
ARDUINO ADC PROGRAMMING

LEARNING OBJECTIVES

- By the end of this lecture, you will be able to:
  - Understand how to configure the Microcontroller Registers to perform ADC operations
  - Understand how to interface an Analog device to your microcontroller and perform the necessary ADC operation.

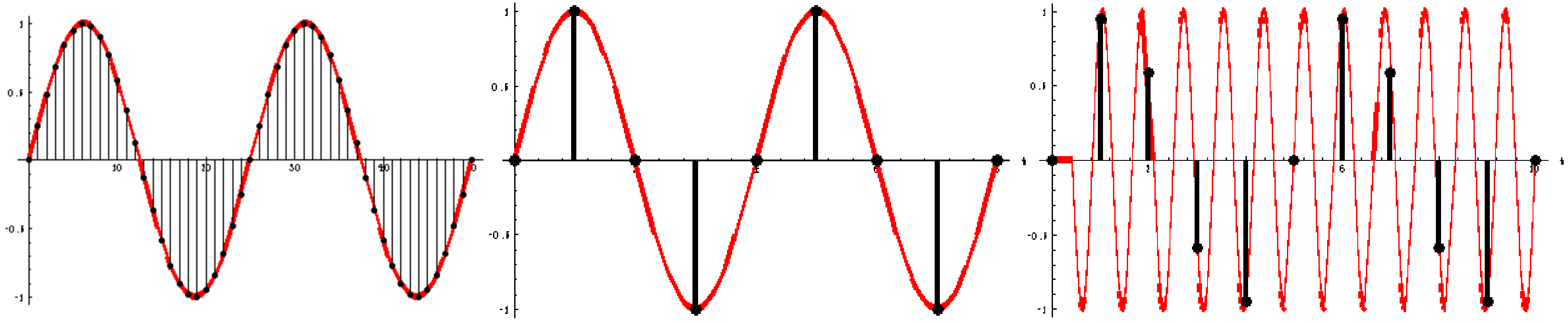
ADC KEY FEATURES

- 10-bit Successive Approximation ADC
- 13-260 us Conversion Time
- 8-channel analog multiplexer which allows eight Single-Ended voltage inputs



SAMPLING

- **Taking snapshot of the analog signal over time**
  - Minimize number of samples but still able to faithfully reconstruct the original signal from the samples
  - Rate of change of signal determines sampling frequency: periods
- **Nyquist Sampling Rate:** Must sample a signal at least twice as fast as the highest frequency content of the signal
  - Example: Human voice signal has frequency range 20Hz – 4 kHz
  - Signal should be sampled at least at 8 kHz



QUANTIZATION & ENCODING

NUS National University of Singapore
School of Computing

- Input voltage signals are typically mapped in the range 0-5 volts
- b bit allows to divide the input signal range into 2^b different quantization levels
- Increased quantization level improves the accuracy
- Quantized signal is encoded, i.e., quantization level is represented as a binary number
- **Resolution: voltage distance between two adjacent quantization levels**
  - Example: 5 volts range, 1 bit representation, 2.5 volts resolution

$$resolution = (voltage\ span)/2^b = (V_{ref\ high} - V_{ref\ low})/2^b$$

PROGRAMMING PROCEDURE

- **To program the ADC on the AVR, the following steps need to be taken:**
- 1. Activate power to the ADC:
  - **Write a "0" to bit 0 (ADC) of the Power Reduction Register PRR.**
- 2. Switch on the ADC:
  - **Write a "1" to bit 7 (ADEN) of the ADC Control and Status Register ADCSRA.**
- 3. Choose which channel you want to read from, and the reference voltage source.
  - **This is done in the ADC Multiplexer Register ADMUX.**

PROGRAMMING PROCEDURE

- 4. Start the conversion:
  - **Write a "1" to bit 6 (ADSC) of ADCSRA.**
- 5. Wait until the conversion ends.
  - **Poll bit 6 of ADCSRA until it becomes 0.**
- 6. Read in the converted value.
  - **Read in bits 7-0 from register ADCL, and combine with bits 9-8 from register ADCH.**
- 7. GOTO Step 4 until desired number of values are converted.

POWER REDUCTION REGISTER

- The Power Reduction Register is used to turn off power to parts of the AT328P, to conserve energy.

Name: PRR
Offset: 0x64
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
	PRTWI0	PRTIM2	PRTIM0		PRTIM1	PRSPI0	PRUSART0	PRADC
Access	R/W	R/W	R/W		R/W	R/W	R/W	R/W
Reset	0	0	0		0	0	0	0

- To turn on power, write a "0" to the bit corresponding to the device you want to switch on. In this case bit 0 (PRADC) corresponds to the ADC.

```
PRR&=0b11111110;
```

SWITCHING ON THE ADC

- Now that power is being supplied to the ADC, we must switch it on by writing a "1" to bit 7 of ADCSRA:

Name: ADCSRA
Offset: 0x7A
Reset: 0x00
Property: -

Bit 7 – ADEN: ADC Enable
Writing this bit to one enables the ADC. By writing it to zero, the ADC is turned off. Turning the ADC off while a conversion is in progress, will terminate this conversion.

Bit	7	6	5	4	3	2	1	0
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

- We also need to set a prescalar value.
  - This determines the sampling frequency, which is given by:

$$f_s = \frac{f_{clk}}{ps}$$

SWITCHING ON THE ADC

- The prescalar value *ps* is specified using bits *ADPS2:0* in ADSCRA, using the following table:

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

SWITCHING ON THE ADC

- The standard clock rate on the Arduino UNO is 16 MHz.
- We will use a pre-scale value of 0b111 (128), giving us a 16,000,000/128 = 125KHz sampling rate.
- For now we will ignore the ADSC, ADATE, ADIF and ADIE bits, setting these to 0.
- The C statement to set up ADSCRA is therefore:

```
ADCSCRA = 0b10000111;
```

SETTING UP THE ADMUX REGISTER

- The ADMUX register lets you choose your reference voltage, as well as which channel to convert:

Name: ADMUX
Offset: 0x7C
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
	REFS1	REFS0	ADLAR		MUX3	MUX2	MUX1	MUX0
Access	R/W	R/W	R/W		R/W	R/W	R/W	R/W
Reset	0	0	0		0	0	0	0

- The REFS1 and REFS0 bits tell the ADC which reference voltage to take, for conversion.

REFS[1:0]	Voltage Reference Selection
00	AREF, Internal V_{ref} turned off
01	AV_{CC} with external capacitor at AREF pin
10	Reserved
11	Internal 1.1V Voltage Reference with external capacitor at AREF pin

SETTING UP THE ADMUX REGISTER

- For the Uno, the AVcc is connected to Vcc through a capacitor, so we will use it as the reference source
  - REFS[1:0] = 0b01
- The conversion channel can be selected using this table:

Channel	MUX2	MUX1	MUX0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

SETTING UP THE ADMUX REGISTER

- For now, we will ignore the MUX3 and ADLAR bits and set these bits to '0'.
- To configure the ADMUX to use the AVcc and convert Channel 2, the setting is :

```
ADMUX=0b01000010;
```

STARTING THE CONVERSION

- Now that we've set everything up, we need to start the conversion. To do this we set the ADSC bit (bit 6) in ADSCRA to a 1.

```
ADCSRA |= 0b01000000;
```

- Loop until the ADSC bit returns back to 0, signalling end of conversion.

```
while(ADCSRA & 0b01000000);
```

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.

CLEARING THE ADIF & READING THE RESULT

- The ADIF bit is set when an ADC conversion completes and the Data Registers are updated. In polling mode (like this example), we need to explicitly clear the flag by **<u>Writing a '1'</u>** to the ADIF bit (bit 4) in the ADCSRA.

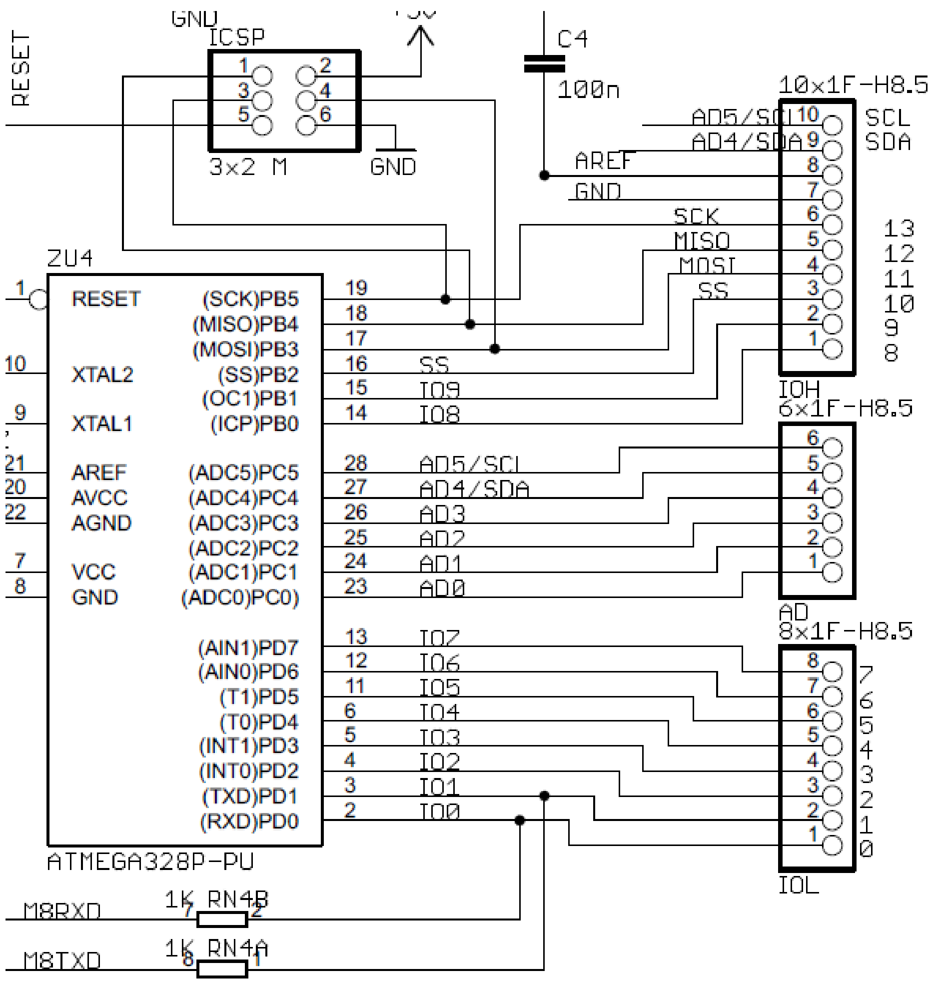
```
ADCSRA |= 0b00010000;
```

- We can then read the ADCL and ADCH registers to the get the converted value.
  - **<u>IMPORTANT</u>**: Reading from ADCH causes the ADC to over-write both ADCL and ADCH with new values.
  - ALWAYS read ADCL first or you will lose the data there!

```
loval=ADCL;
hival=ADCH;
adcval= hival * 256 + ADCL;
```

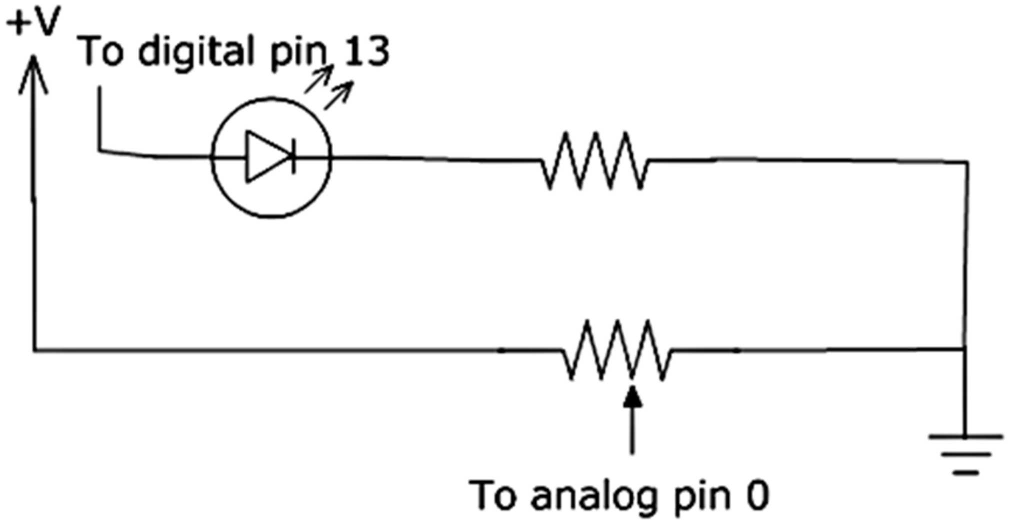
PIN MAPPING

- As shown in the reference circuit for the Arduino Uno, the mapping is direct
- ADC channel 0 maps to Arduino analog input 0, etc.



SEE IT IN ACTION!

- We will now write a program that reads in a voltage reading from Analog Pin 0 (ADC0, Port C Bit 0) , and uses the read value to control the rate of blinking of an LED connected to Digital Pin 13 (Port B Bit 5).



SEE IT IN ACTION!

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

void ledOn()
{
        PORTB|=0b00100000;
}

void ledOff()
{
        PORTB&=0b11011111;
}
```

SEE IT IN ACTION!

```
int main()
{
        unsigned adcvalue, loval, hival;

        // Set up ADC
        // Write 0 to PRR bit 0 to disable power reduction on the ADC
        PRR &= 0b11111110;

        // Enable ADC, don't start conversion, disable ADIF and ADIE and
        // set ADPS2-0 to 111 to set prescalar value of 128
        ADCSRA=0b10000111;

        // Set up ADMUX to convert from channel 0.
        ADMUX=0b01000000;

        // Set up LED on pin 13 (port B pin 5) for output.
        DDRB|=0b00100000;
```

SEE IT IN ACTION!

```
    while(1)
    {
            // Start ADC conversion by writing a 1 to ADSC bit.
            ADCSRA |= 0b01000000;
            // Wait for conversion to end.
            while(ADCSRA & 0x01000000);

            // Clear ADIF, Read ADC value
            ADSCRA |=0b00010000;
            loval=ADCL;
            hival=ADCH;
            adcvalue = hival*256+loval;

            ledOn();
            _delay_ms(adcvalue);
            ledOff();
            _delay_ms(adcalue);
    }
} // main
```

THE END!

- In your studio, you will first be implementing the Polling approach.
- Next, you will look at the Interrupt approach for the ADC.
- Good Luck!