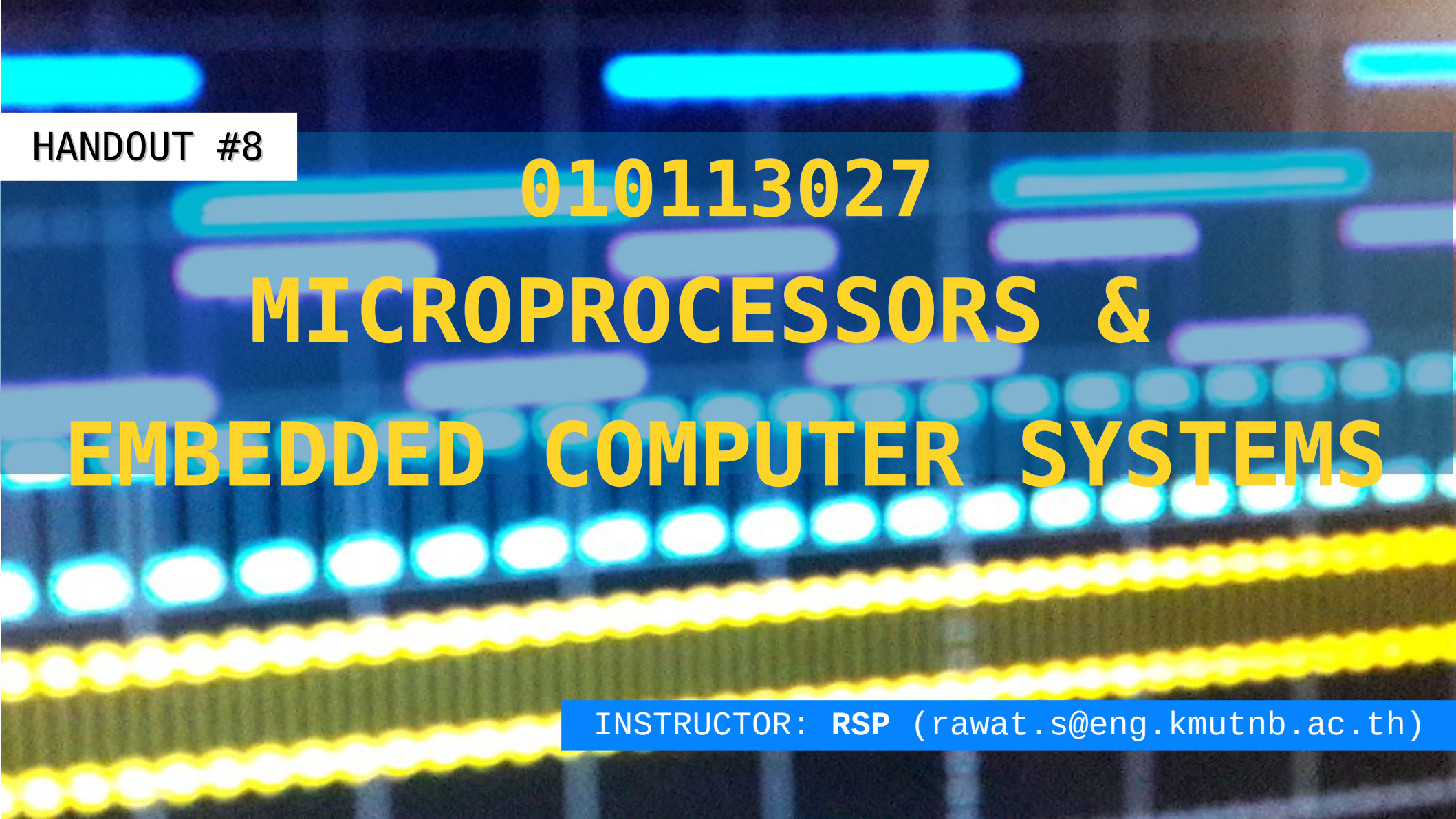


HANDOUT #8

010113027
MICROPROCESSORS & EMBEDDED COMPUTER SYSTEMS

INSTRUCTOR: **RSP** (rawat.s@eng.kmutnb.ac.th)

What We Will Learn...

- Basic Concept of a SAR-type ADC
- ADC (Analog to Signal Converter) Module inside AVR MCU
- ADC Operating Modes & Configurations
- ADC Voltage Reference Sources, ADC Bit Resolution, ADC Accuracy
- ADC Registers of ATmega328P
- AVR C Code Examples for Using the ADC Module
- Analog Comparator

ADC (Analog-to-Digital Converter)

- The **ADC** is a hardware module inside an **MCU** that converts an **analog voltage** into a **digital value**.
- There are **several ADC architectures**, such as **Flash ADC, Successive Approximation Register (SAR)** and **Sigma-Delta**.
- Among these, the **SAR-type ADC** is most widely used because it provides a good balance between conversion speed, resolution, hardware complexity, and power consumption.
- The **ADC** compares the **input voltage** (**VIN)** to a **reference voltage (VREF)** and outputs the following value:

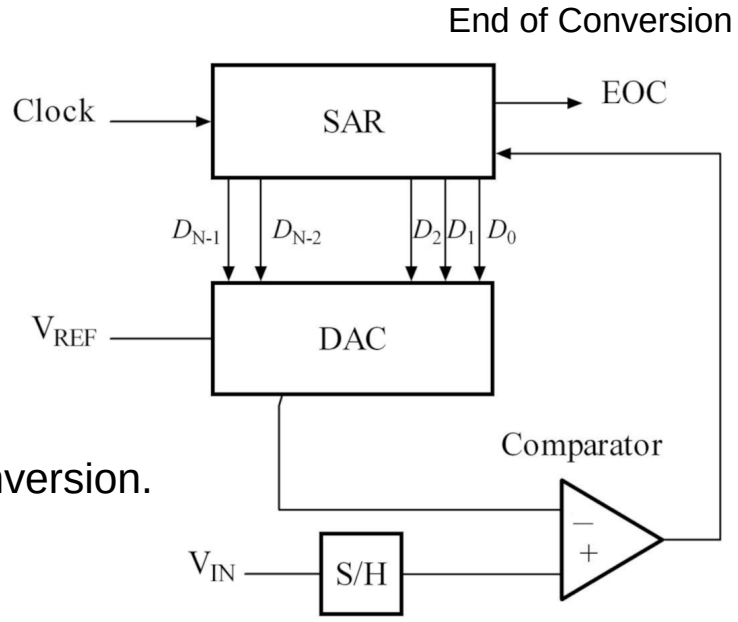
  ADC Value = 1024 x VIN / VREF and **0 ≤ VIN ≤ VREF**

  VREF = +5V → **ADC resolution** = 5V / 1024 = 4.883mV

- The **ATmega328P** does support only **single-ended analog inputs** and does **NOT support differential inputs**.

Successive-approximation ADC

- The core concept of a **SAR ADC** is to perform a **binary search** to find the digital code that best represents the input analog voltage with N-bit resolution.

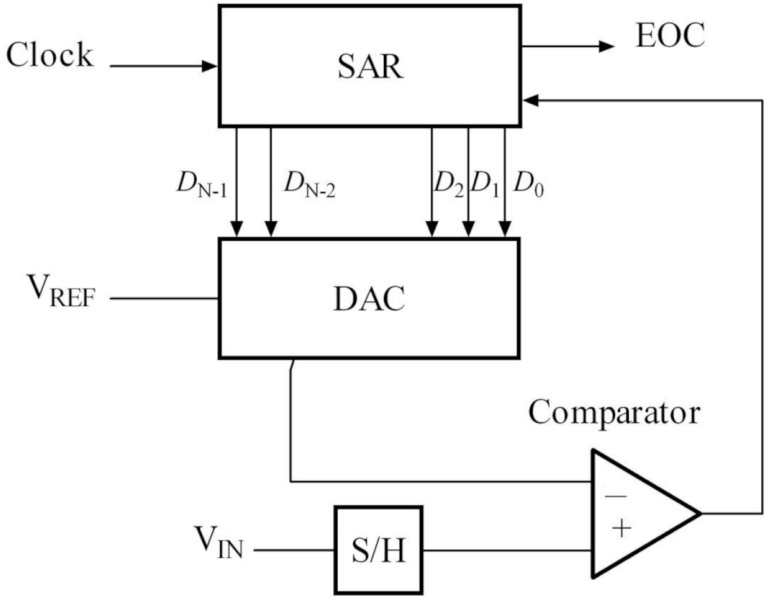


Key Components of a SAR ADC

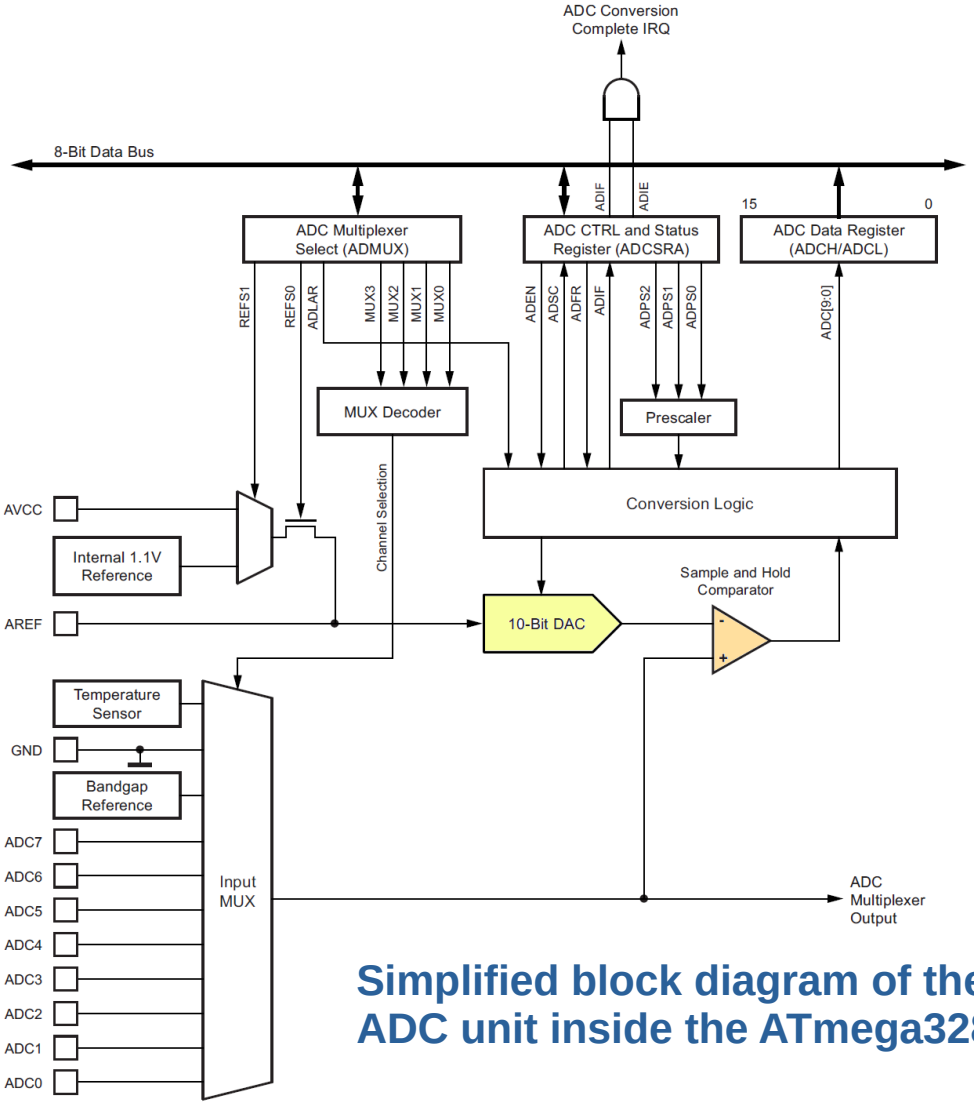
1. **Sample-and-Hold (S/H) Circuit**
   - Captures and holds the input voltage steady during conversion.
2. **Analog Voltage Comparator**
   - Compares the sampled input voltage to an internally generated reference voltage (from the DAC).
3. **Digital-to-Analog Converter (DAC)**
   - Converts digital codes to corresponding analog voltages for comparison.
4. **Successive Approximation Register (SAR) and Control Logic**
   - Controls the binary search process bit by bit.

Reference: https://en.wikipedia.org/wiki/Successive-approximation_ADC

ATmega328P - ADC Block Diagram



Simplified block diagram of a SAR ADC



Simplified block diagram of the ADC unit inside the ATmega328P

Binary Search in SAR ADC

1. Sampling

- The **ADC** samples the input analog voltage and holds it constant for conversion.

2. Initialize SAR

- The **SAR** register starts with the most significant bit (**MSB**) set to 1 and all lower bits cleared (0).

3. Generate Trial DAC Voltage

- The **DAC** converts the SAR binary code into an analog voltage **V_DAC**.
- Initial value: **V_DAC = VREF/2**.

4. Compare and Decide for MSB

- The comparator compares the sampled input with the **DAC** output:
- If the sampled voltage is higher than **V_DAC**, the **MSB** stays 1.
- If it is lower, the **MSB** is cleared.

5. Next Bit

- The SAR sets the next bit (next less significant) to 1 and repeats the comparison.
- This process continues one bit at a time, moving from **MSB** to **LSB**.

6. Finish After N Bits

- After **N** iterations, the **SAR register** holds a digital value that best approximates the input voltage.

ADC Features in the ATmega328P

- The **ATmega328P** uses a **10-bit SAR ADC**.
- It converts an input voltage (**0 to VREF**) into a digital value from **0 to 1023**.
- The conversion result is stored in the **ADC data registers ADCL (low byte)** and **ADCH (high byte)**, which together form a **16-bit register**.
- **Up to 8 analog input channels (ADC0–ADC7)** are available and selected through an **internal analog multiplexer**.
  - **ATmega328P in a 28-pin PDIP**: only **ADC0** to **ADC5** are available (can be used as digital I/O pins; share pins with **PORTC)**.
  - **ATmega328P in a 32-pin TQFP / QFN**: **ADC6 and ADC7** pins are analog only (no digital input buffers).
- The input channel is selected using **MUX[3:0]** bits in the **ADMUX** register.

ADC Registers

ADMUX – ADC Multiplexer Selection Register

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	

ADCSRA – ADC Control and Status Register A

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	0	0	0	0	0	0	0	0	

ADCSRB – ADC Control and Status Register B

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	

DIDR0 – Digital Input Disable Register 0

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	

ADC Reference Voltage Selection

- The **ADC** compares the input voltage with a **reference voltage** (**VREF**).
- For the **ATmega328P**, the voltage reference sources are:
  - Use the voltage supply for analog: **AVCC**
    - Typically connected to **VCC**, e.g., **5V** on **Arduino Uno / Nano**
  - Use the dedicated **AREF** pin (**external reference**), any voltage level up to **VCC**
  - Use the internal **1.1V reference**
    - Higher ADC resolution: 1.1V / 1024 = 1.0742 mV per step
- The **VREF source** is selected using the **REFS[1:0]** bits in the **ADMUX** register.
  - **REFS[1:0] = "00"**: select **AREF** pin
  - **REFS[1:0] = "01"**: select **AVCC** pin
  - **REFS[1:0] = "11"**: select **internal 1.1V**

ADC Registers

ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0
(0x7C)	**REFS1**	**REFS0**	**ADLAR**	**–**	**MUX3**	**MUX2**	**MUX1**	**MUX0**
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

Input Channel Selections

MUX3..0	Single Ended Input
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5
0110	ADC6
0111	ADC7
1000	ADC8(1)
1001	(reserved)
1010	(reserved)
1011	(reserved)
1100	(reserved)
1101	(reserved)
1110	1.1V (V_{BG})
1111	0V (GND)

Note: 1. For temperature sensor.

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

ADC Result Alignment & Clock Prescaler

Result Alignment

- The 10-bit result can be right-adjusted or left-adjusted in **ADCL/ADCH**.
- Controlled by the **ADLAR** bit in **ADMUX**.
- For right adjustment, read **ADCL** (low byte) first, then **ADCH** (high byte).

ADC Clock Prescaler

- For full 10-bit accuracy, the **ADC clock** should typically be **50–200 kHz**.
- With a **16MHz** CPU clock, choose a /128 prescaler → **125 kHz ADC CLK**.
- The **ADC prescaler** is selected using **ADPS[2:0]** bits in **ADCSRA**.

ADC Registers

ADCL and ADCH – The ADC Data Register

ADLAR = 0

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	

ADLAR = 1

Bit	15	14	13	12	11	10	9	8	
(0x79)	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
(0x78)	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	

DIDR0 Register

DIDR0 – Digital Input Disable Register 0

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	

- **Bits 7:6 – Res: Reserved Bits**

These bits are reserved for future use. To ensure compatibility with future devices, these bits must be written to zero when DIDR0 is written.

- **Bit 5:0 – ADC5D..ADC0D: ADC5..0 Digital Input Disable**

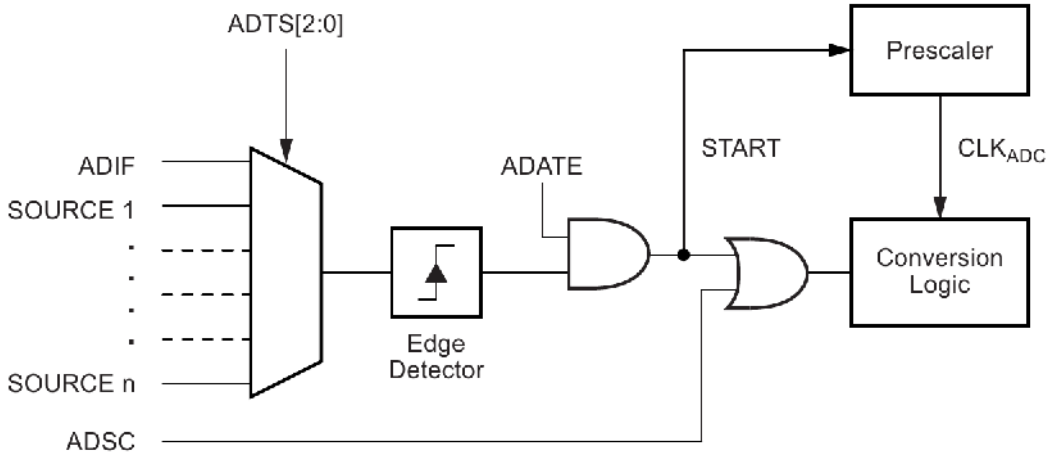
When this bit is written logic one, the digital input buffer on the corresponding ADC pin is disabled. The corresponding PIN register bit will always read as zero when this bit is set. When an analog signal is applied to the ADC5..0 pin and the digital input from this pin is not needed, this bit should be written logic one to reduce power consumption in the digital input buffer.

Note that ADC pins ADC7 and ADC6 do not have digital input buffers, and therefore do not require digital input disable bits.

The **Digital Input Disable Register** turns off the digital input buffer on pins that are being used as analog inputs. Disabling these buffers reduces power consumption and avoids noise injection into ADC measurements.

ADC Registers

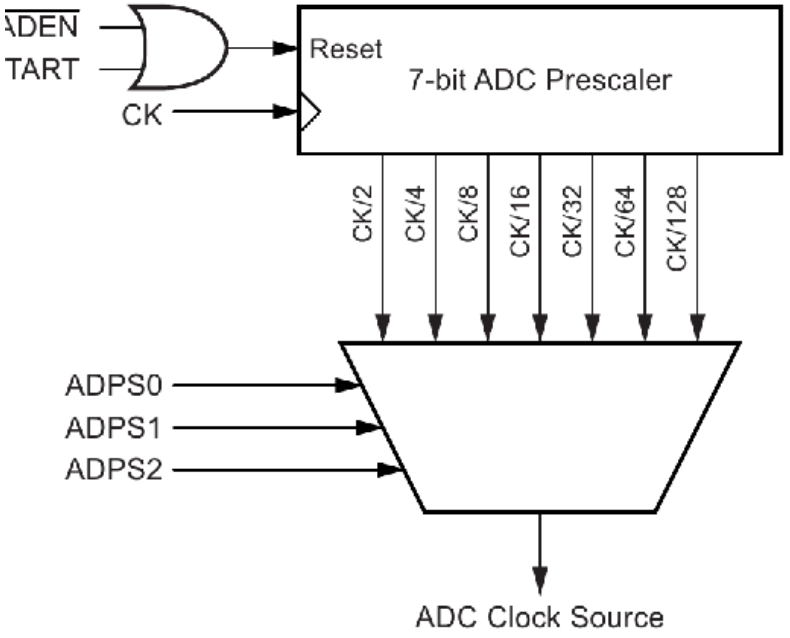
ADC Auto Trigger Logic



ADC Auto Trigger Source Selections

ADTS2	ADTS1	ADTS0	Trigger Source
0	0	0	Free running mode
0	0	1	Analog comparator
0	1	0	External interrupt request 0
0	1	1	Timer/Counter0 compare match A
1	0	0	Timer/Counter0 overflow
1	0	1	Timer/Counter1 compare match B
1	1	0	Timer/Counter1 overflow
1	1	1	Timer/Counter1 capture event

ADC Prescaler Selection



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

ADC Operation Modes

1) Single Conversion Mode

- Disable the power reduction ADC bit (**PRADC**)
- Start conversion by setting the **ADSC** bit.
- When **ADSC** clears, the conversion is complete.
- Read the result from the **ADCL / ADCH** registers.

2) Free-Running Mode

- The **ADC** continuously performs conversions automatically.
- Set the **ADATE** bit to 1 and set the **ADTS[2:0]** bits = **000**.

3) Auto Trigger (Timer/Events)

- Conversions can be triggered automatically by hardware events (e.g., **Timer0 / Timer1 Overflow or Compare Match**).
- Select the trigger source using **ADTS[2:0]** bits in **ADCSRB**.

ADC In Free-Running Mode

The **ADC Free-Running mode** uses an **interrupt flag trigger**.

- The ADC interrupt flag (**ADIF**) can be used as the **auto-trigger source**.
- When a conversion finishes, the next conversion starts immediately.
- ADC continuously samples the input (free-running mode).
- ADC data register (**ADCH:ADCL)** is updated after every conversion.
- First conversion must be started manually by setting **ADSC = 1**.
- Further conversions run automatically.
- Clearing **ADIF** is NOT required to start the next conversion.

ADC Register: ADCSRA

ADCSRA – ADC Control and Status Register A

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	0	0	0	0	0	0	0	0	

- **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 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.

- **Bit 5 – ADATE: ADC Auto Trigger Enable**

When this bit is written to one, auto triggering of the ADC is enabled. The ADC will start a conversion on a positive edge of the selected trigger signal. The trigger source is selected by setting the ADC trigger select bits, ADTS in ADCSRB.

ADC Register: ADCSRA

- **Bit 4 – ADIF: ADC Interrupt Flag**

This bit is set when an ADC conversion completes and the data registers are updated. The ADC conversion complete interrupt is executed if the ADIE bit and the I-bit in SREG are set. ADIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, ADIF is cleared by writing a logical one to the flag. Beware that if doing a read-modify-write on ADCSRA, a pending interrupt can be disabled. This also applies if the SBI and CBI instructions are used.

- **Bit 3 – ADIE: ADC Interrupt Enable**

When this bit is written to one and the I-bit in SREG is set, the ADC conversion complete interrupt is activated.

- **Bits 2:0 – ADPS2:0: ADC Prescaler Select Bits**

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

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

ADC Register: ADMUX

ADMUX – ADC Multiplexer Selection Register

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	

- **Bit 7:6 – REFS1:0: Reference Selection Bits**

These bits select the voltage reference for the ADC, as shown in Table 23-3. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set). The internal voltage reference options may not be used if an external reference voltage is being applied to the AREF pin.

Table 23-3. 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

- **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.

- **Bit 4 – Res: Reserved Bit**

This bit is an unused bit in the Atmel® ATmega328P, and will always read as zero.

ADC Register: ADMUX

- **Bits 3:0 – MUX3:0: Analog Channel Selection Bits**

The value of these bits selects which analog inputs are connected to the ADC.

Input Channel Selections

MUX3..0	Single Ended Input
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5
0110	ADC6
0111	ADC7
1000	ADC8[1]
1001	(reserved)
1010	(reserved)
1011	(reserved)
1100	(reserved)
1101	(reserved)
1110	1.1V (V_{BG})
1111	0V (GND)

Note: 1. For temperature sensor.

ADC Conversion Time

Single Conversion Mode (ADATE=0)

- Start with **ADSC=1**
- First conversion: **25 ADC cycles**
- Extra time is used for:
  - ADC circuitry startup
  - sample/hold initialization
- Subsequent conversions: **13 ADC cycles**
  - Set **ADSC** bit = 1 to restart the next ADC conversion.

Auto-Trigger Mode

- Select the trigger source and enable by setting **ADATE = 1**.
- Trigger event starts conversion automatically.
- Each conversion: **13.5 ADC cycles**.

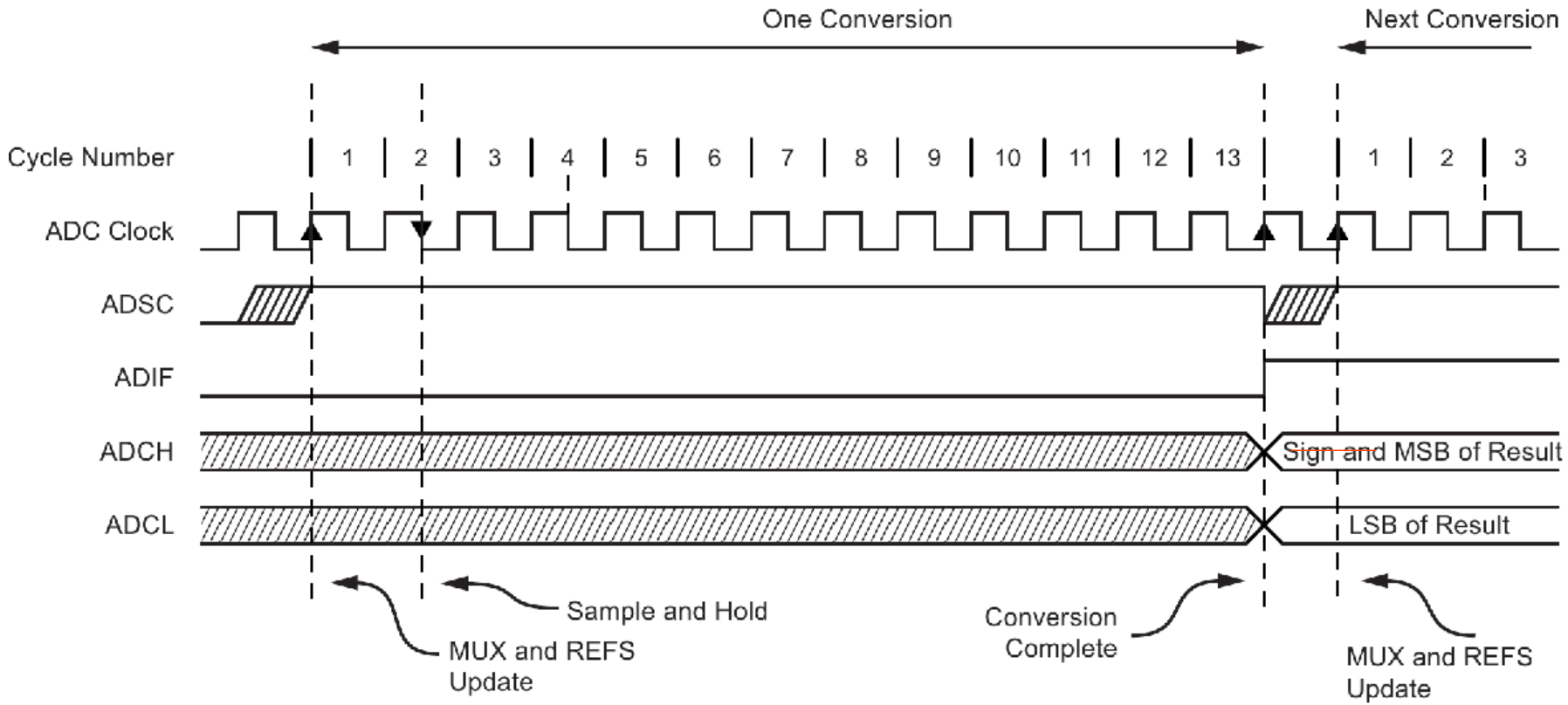
Free-Running Mode

- Set **ADATE=1** and **ADTS=000**.
- Select the **ADC interrupt flag** as the **trigger source**.
- Each (avg.) conversion: **13 ADC cycles**

Condition	Sample and 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

ADC Timing Diagram

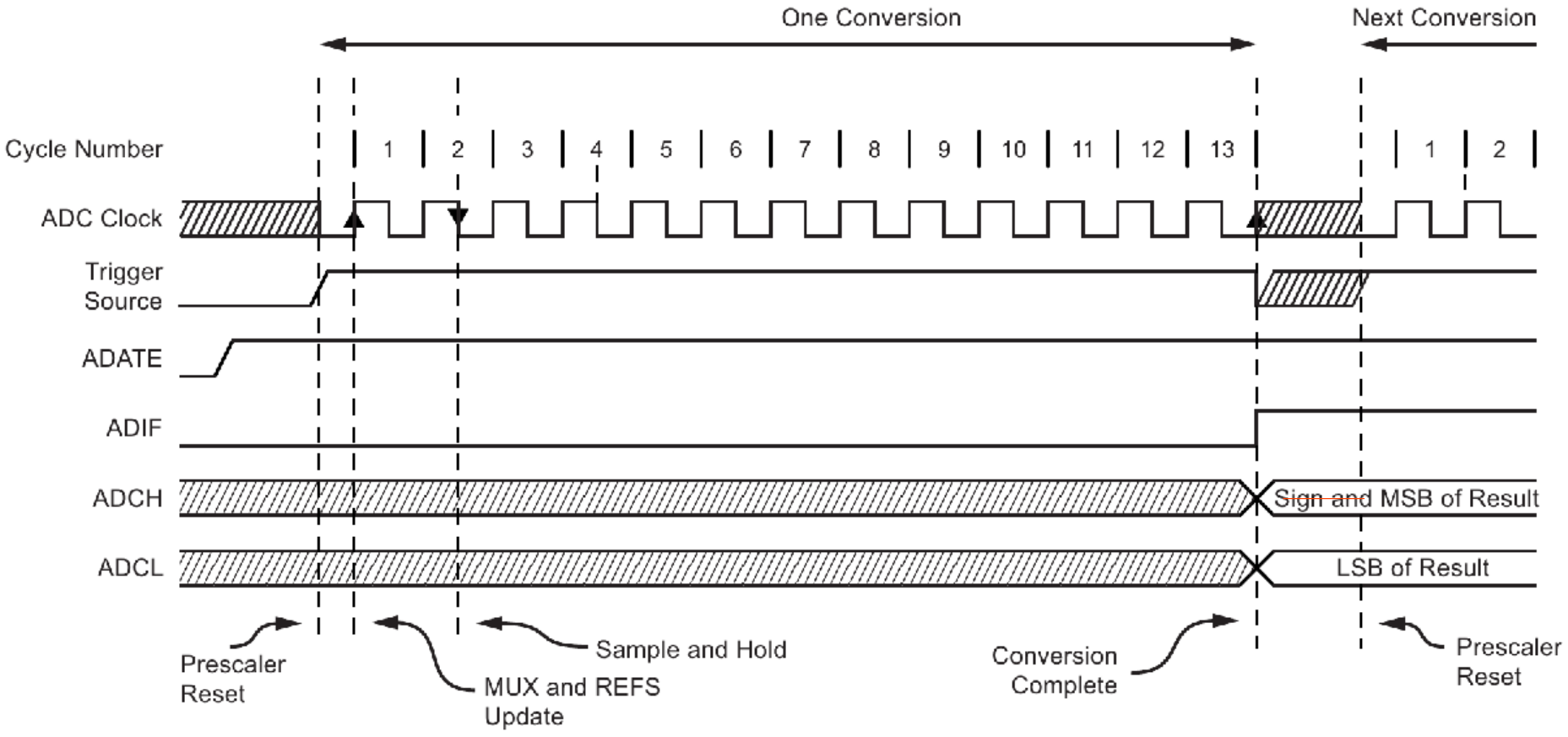
Single Conversion (Not the first conversion)



A normal conversion takes **13 ADC clock cycles**. The **first conversion** after the ADC is switched on (ADEN in ADCSRA is set) takes **25 ADC clock cycles** in order to initialize the analog circuitry.

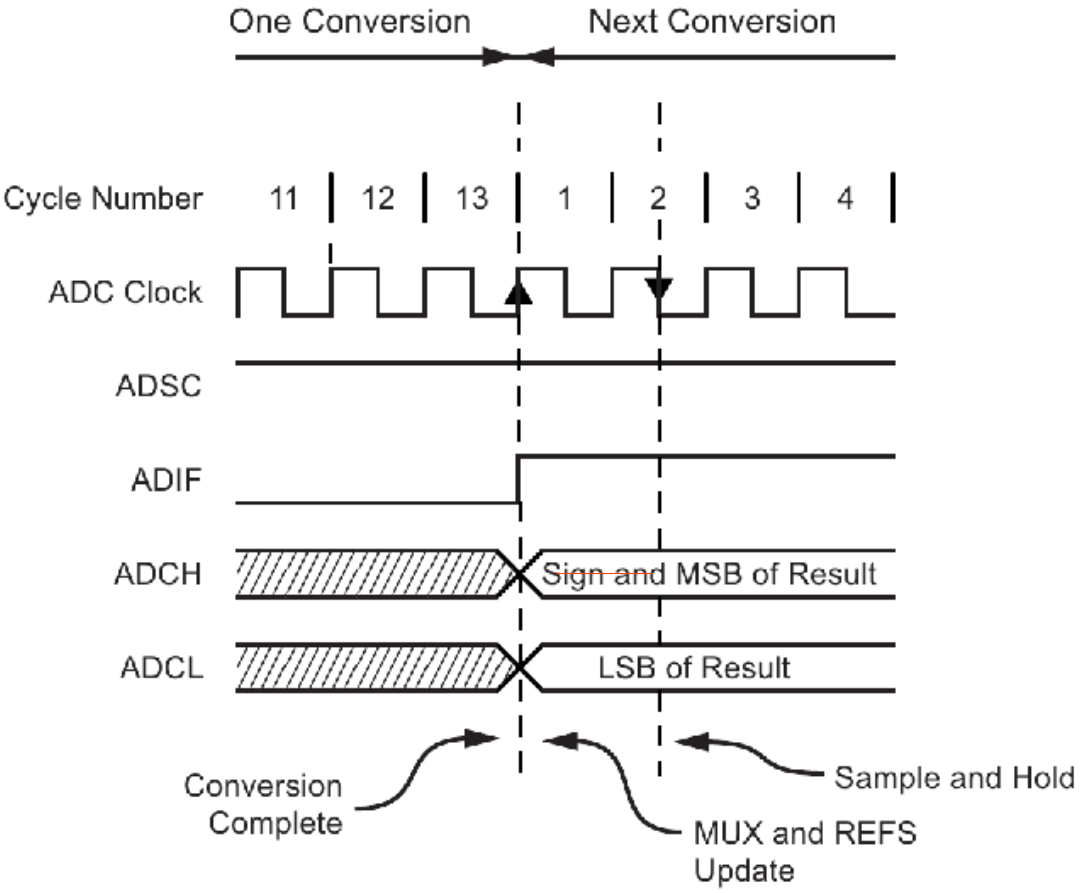
ADC Timing Diagram

Auto Triggered Conversion



ADC Timing Diagram

Free-Running Conversion



ADC Timing Diagram

- When an ADC channel is selected, the input voltage charges an internal **sample-and-hold capacitor** (≈14 pF) through the **internal analog multiplexer resistance**.
- The ADC is optimized for signal sources with an output impedance of **≤ 10 kΩ**, allowing the capacitor to settle within the acquisition time.
- Higher source impedance may prevent full settling of the sample-and-hold capacitor, resulting in conversion errors.
- When using a **potentiometer-based voltage divider** as an ADC input, a **10 kΩ potentiometer** is preferred over **100 kΩ** to ensure proper ADC sampling accuracy.
- High-impedance sources should be buffered with an op-amp to provide a low output impedance to the ADC.

ADC Accuracy

ADC Conversion

- N-bit ADC: **LSB = VREF / 2^N**
  - 10-bit ADC, VREF=5V, LSB = 4.88mV
  - 10 bits => 1024 codes (0..1023)
- This conversion should be ideally linear.

Offset Error

- The difference between the ideal input voltage and the actual input voltage required to trigger the first digital code transition.
- Positive or negative offset error.

Gain Error

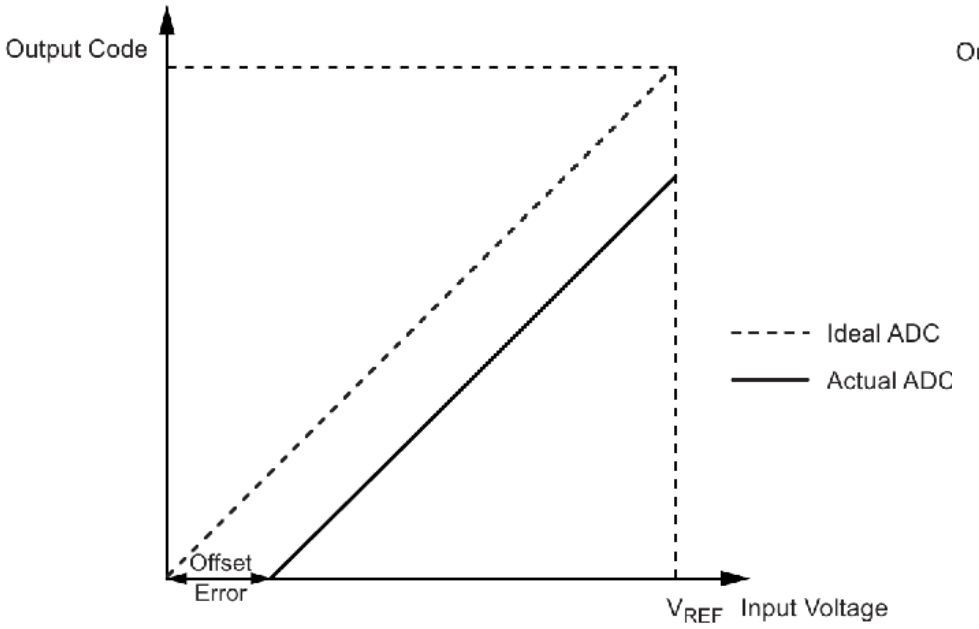
- ADC scale or slope is wrong.

Voltage Step Width

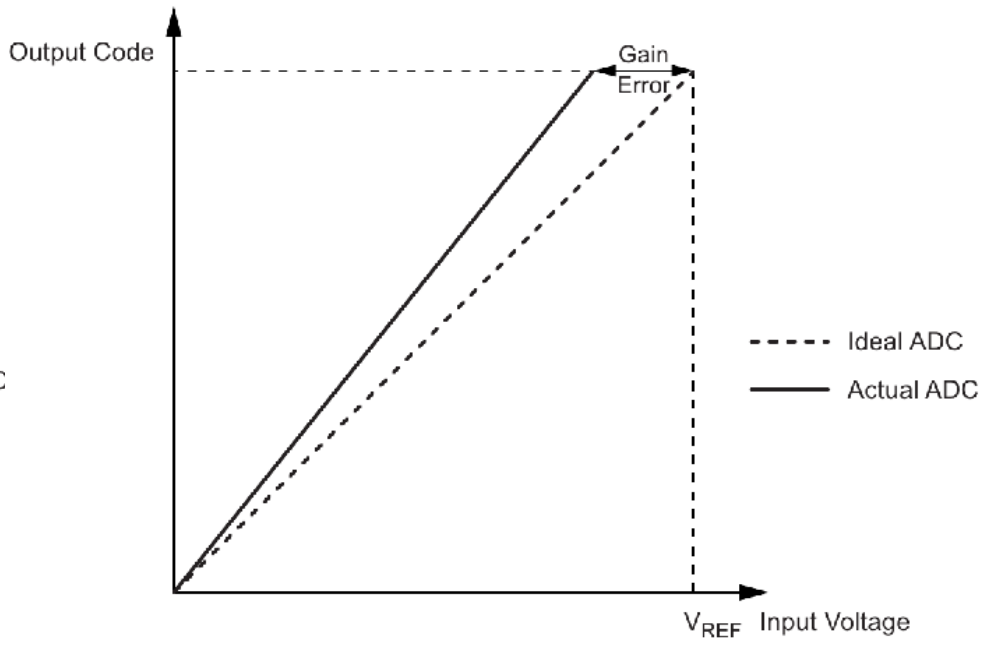
- Step width indicates how much input voltage must change before the ADC output value increments by 1.
- Step width = voltage difference between adjacent code transition points
  - **Ideal**: **VREF / 2^N = LSB**
  - **Non-ideal**: The step width may not constant (some steps too small, some too large).
  - **DNL (Differential non-linearity):**

    DNL = (actual step width / LSB) - 1
  - Ideal **DNL** = 0 LSB

ADC Accuracy

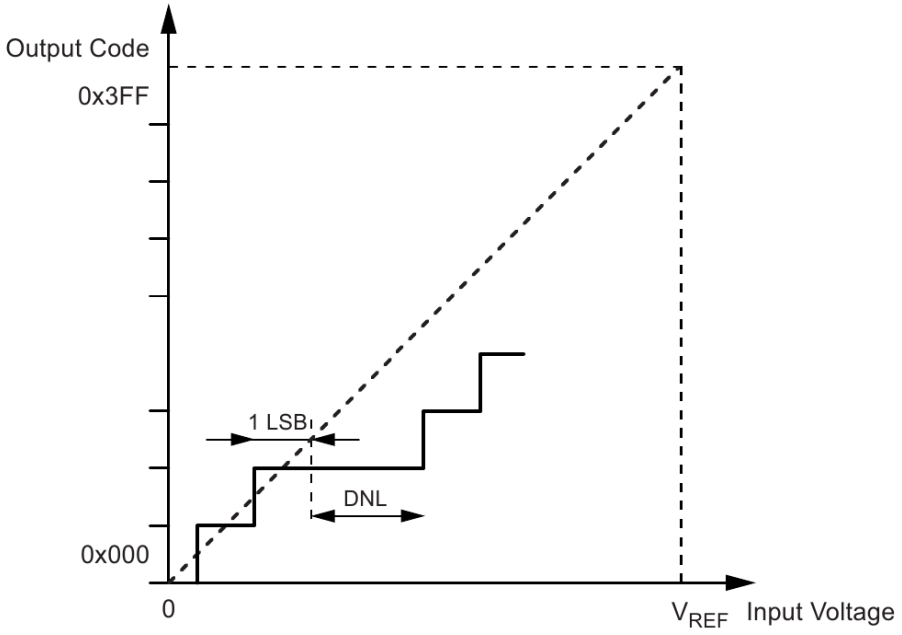


Offset Error

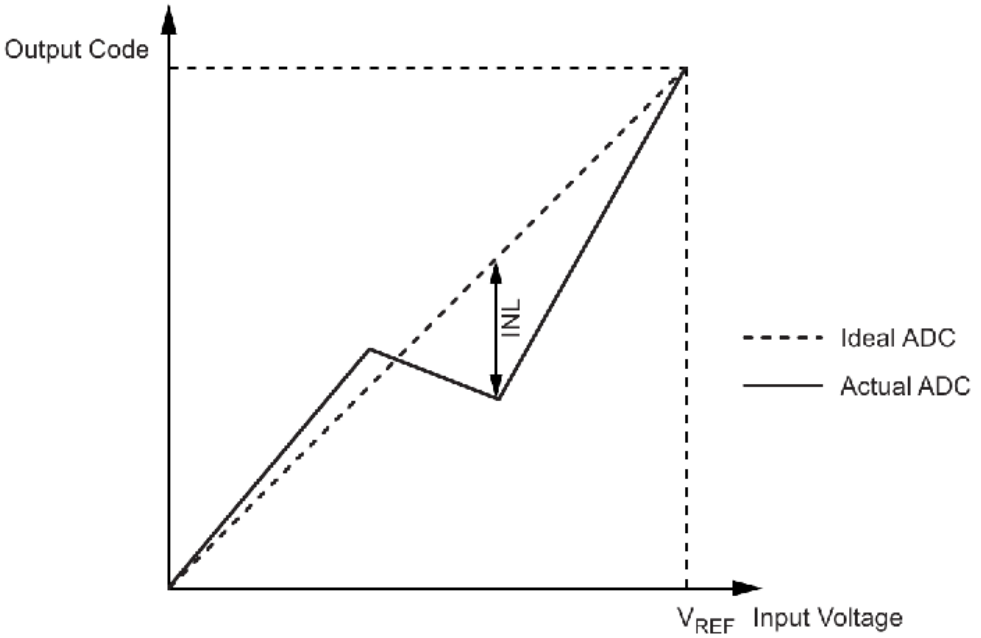


Gain Error

ADC Accuracy



Differential non-linearity (INL)



Integral non-linearity (INL)

- **INL** is the maximum deviation of the ADC's actual transfer function from an ideal straight line, measured in LSB.
- It shows how far the ADC transfer curve bends away from a straight line.

ADC Input Switching

- When **ADMUX** is changed, the **ADC input multiplexer** selects a new channel.

```
ADMUX = (ADMUX & 0xF0) | channel; // ADC0..ADC5 on Uno

ADMUX = (ADMUX & 0xF0) | 0; // Select ADC0
ADMUX = (ADMUX & 0xF0) | 1; // Select ADC1
...
```

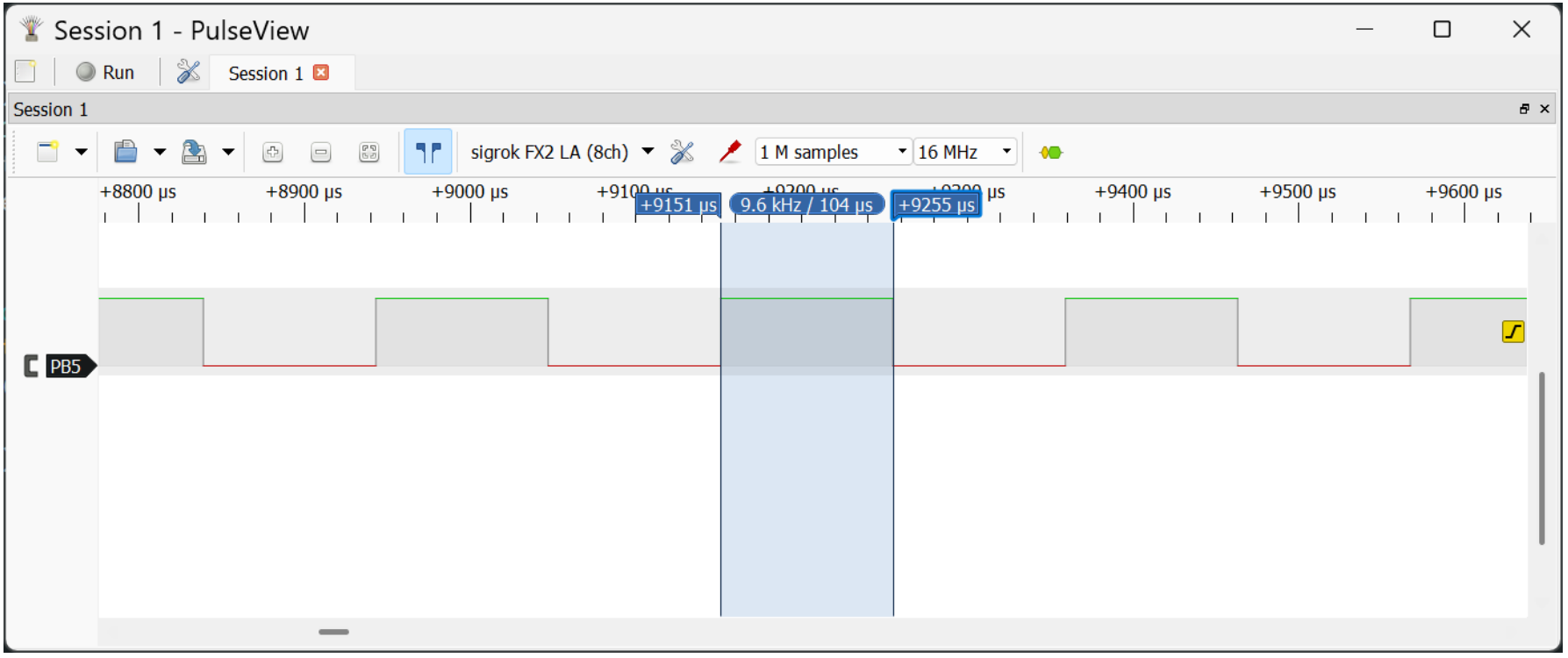
- The **sample-and-hold (S&H) capacitor** initially retains the previous channel's voltage.
- Because the capacitor must settle to the new input voltage through the source impedance, the **first conversion** may be inaccurate and should be discarded.

Code Example: Free-Running ADC + ISR

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

volatile uint16_t adc_value;

static void gpio_init(void) {
  // Configure PB5 as output
  DDRB  |= (1 << DDB5);     // PB5
output
  PORTB &= ~(1 << PORTB5);  // Output
LOW
}

ISR(ADC_vect) {
  PINB = (1 << PINB5); // Toggle PB5
  // Read ADCH:ADCL to clear flag
  adc_value = ADC;
}
```

```
static void adc_init(uint8_t channel) {
  // Note: ADLAR = 0 (default) right adjusted
  ADMUX = (1 << REFS0); // AVcc vref
  ADMUX = (ADMUX & 0xF0) | channel;
  ADCSRA = (1 << ADEN) |(1 << ADATE)|(1 << ADIE)
         | (1 << ADPS2)|(1 << ADPS1)|(1 <<
ADPS0);
  ADCSRB = 0;  // Use free running mode
  ADCSRA |= (1 << ADSC); // Start ADC conversion
}

int main(void) {
  gpio_init();
  adc_init(0);
  sei();
  while (1) {
  }
}
```

Output Signal Measurement



The pulse width measured by a USB logic analyzer is about **104 µs**.

The ADC clock frequency is **16 MHz / 128 = 125 kH**z, so one ADC clock period is **8 µs**.

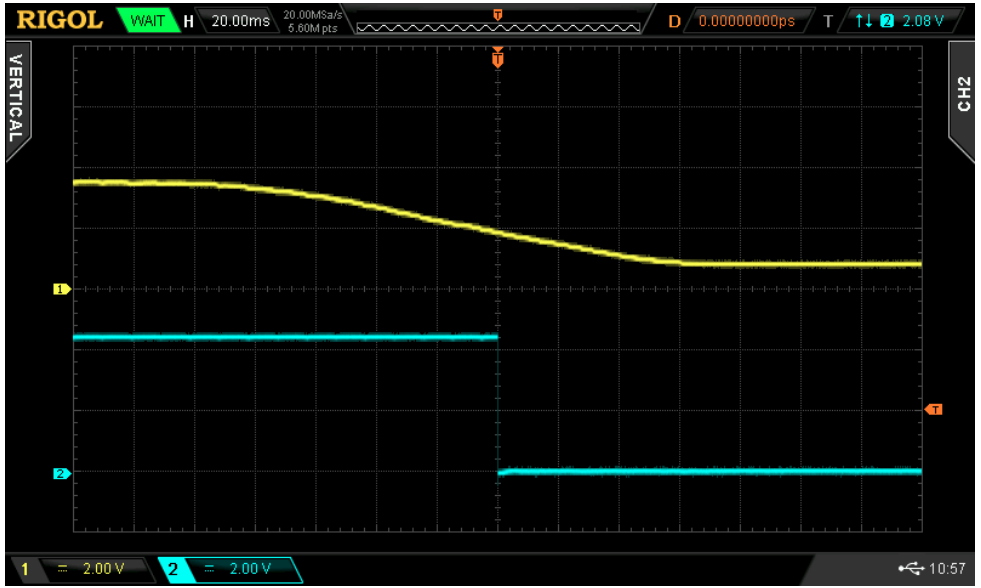
Therefore, the **average ADC conversion time** is **104 µs ÷ 8 µs** = **13 ADC clock cycles**.

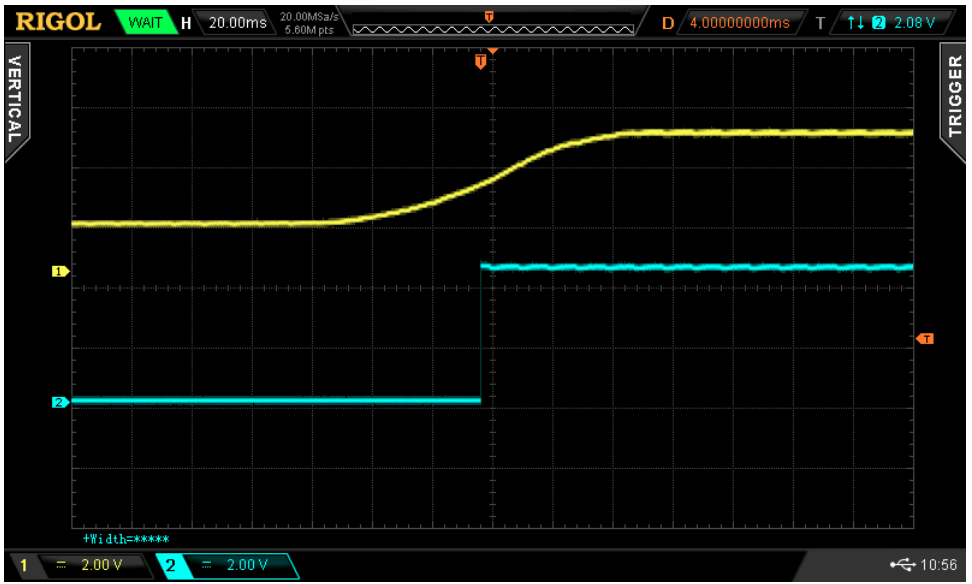
Code Example: Software Schmitt-Trigger Input

```
// Define the lower/upper threshold voltages
#define TH_LOW   100
#define TH_HIGH  (1023-TH_LOW)

ISR(ADC_vect) {
   adc_value = ADC;
   static uint8_t state = 0; // 0=LOW, 1=HIGH
   if (!state && adc_value > TH_HIGH) {
       PORTB |= (1 << PORTB5); // Output HIGH
       state = 1;
   }
   else if (state && adc_value < TH_LOW)  {
       PORTB &= ~(1 << PORTB5); // Output LOW
       state = 0;
   }
}
```

- In this example, the **ADC ISR** compares the current ADC value with predefined upper and lower thresholds.
- The output changes state only when the signal crosses these limits (**hysteresis behavior**).

Output Signal Measurement





CH1: Analog input
CH2: Digital output

C Code for USART (Asynchronous Mode)

```c
#ifndef F_CPU
#define F_CPU 16000000UL
#endif

#include <avr/io.h>

#define BAUD        115200UL
#define UBRR_VALUE  ((F_CPU/(8UL*BAUD)) - 1)

void usart_init(void) {
   UBRR0H = (uint8_t)(UBRR_VALUE >> 8);
   UBRR0L = (uint8_t)UBRR_VALUE;
   UCSR0A = (1 << U2X0);  // double speed
   UCSR0B = (1 << TXEN0); // TX enable
   UCSR0C = (1 << UCSZ01) | (1 << UCSZ00); // 8N1
}

void usart_tx(char c) {
   while (!(UCSR0A & (1 << UDRE0)));
   UDR0 = c;
}

void usart_print(const char *s) {
   while (*s) usart_tx(*s++);
}
```

Code Example: VCC Measurement

```
#ifndef F_CPU
#define F_CPU 16000000UL
#endif

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdio.h>

extern void usart_init(void);
extern void usart_tx(char c);
extern void usart_print(const char *s);

void init_adc(void) {
  // Select AVcc reference + 1.1V bandgap input
  ADMUX = (1 << REFS0) | 0x0E;
  // Enable ADC, prescaler 128
  ADCSRA = (1 << ADEN)
         |(1 << ADPS2)|(1 << ADPS1)|(1 << ADPS0);
  ADCSRA |= (1 << ADSC);  // Start ADC
  while (ADCSRA & (1 << ADSC)) ; // Dummy read
}
```

```
uint16_t read_vcc_mv(void) {
  ADCSRA |= (1 << ADSC); // Start ADC
  while (ADCSRA & (1 << ADSC)) ;
  uint16_t adc = ADC; // Read ADC value
  // Vcc (mV) = 1.1 * 1023 * 1000 / ADC
  uint32_t vcc = 1125300UL / adc;
  return (uint16_t)vcc;
}

int main(void) {
  char sbuf[16];
  usart_init();
  init_adc();
  while (1) {
    uint16_t value = read_vcc_mv();
    snprintf(sbuf, sizeof(sbuf),
             "VCC:%u\r\n", value);
    usart_print(sbuf);
    _delay_ms(100);
  }
}
```

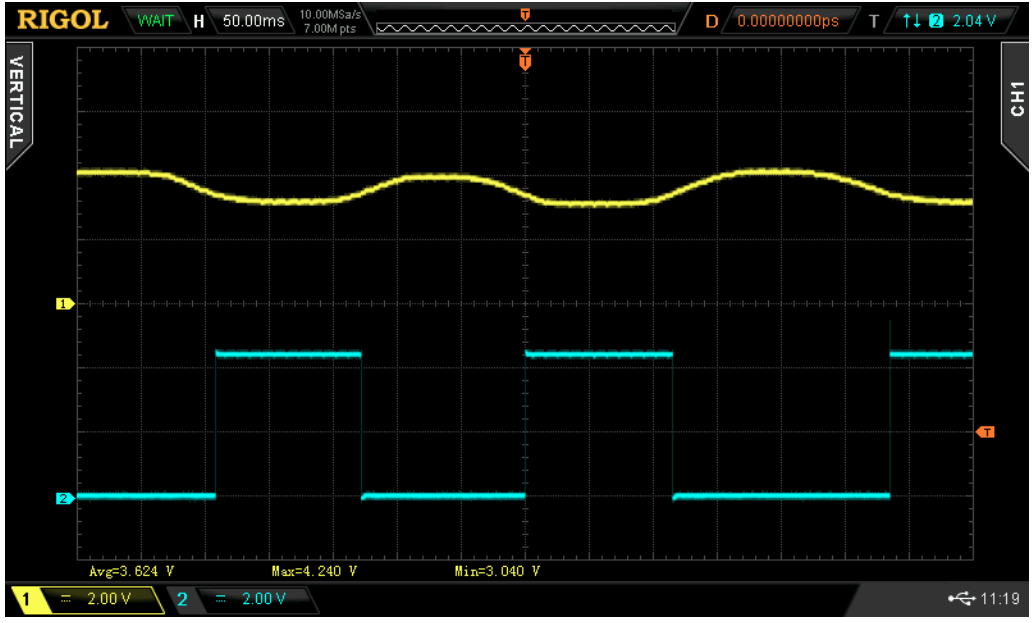
Code Example: Two-Channel Voltage Compare

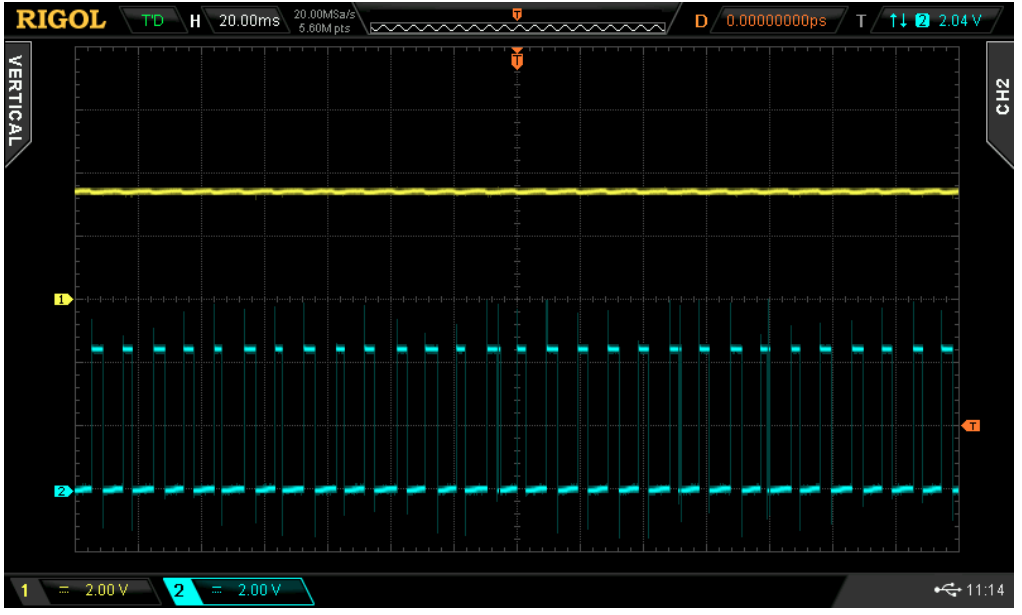
```
#include <avr/io.h>
#include <avr/interrupt.h>

volatile uint16_t adc_val[2];
volatile uint8_t  chan = 0;

typedef enum {S0=0, S1} state_t;
state_t state = S0;

static void gpio_init(void) {
  DDRB |= (1 << DDB5);
  PORTB &= ~(1 << PORTB5);
}

static void adc_init(void) {
  ADMUX = (1 << REFS0); // AVcc
  ADMUX = (ADMUX & 0xF0) | chan;
  ADCSRA = (1 << ADEN)
         | (1 << ADIE)
         | (1 << ADPS2)
         | (1 << ADPS1)
         | (1 << ADPS0);
  ADCSRB = 0;
  ADCSRA |= (1 << ADSC);
}
```

```
ISR(ADC_vect) {
  uint16_t value = ADC;
  switch (state) {
    case S0: // Skip ADC value
      state = S1; break;
    case S1:
      // Save the ADC value
      adc_val[chan] = value;
      // Change the ADC channel
      chan = (chan + 1) % 2;
      ADMUX = (ADMUX & 0xF0) | chan;
      if (chan==0) {
        // Update compare output
        if (adc_val[0] < adc_val[1])
          PORTB |= (1 << PORTB5);
        else
           PORTB &= ~(1 << PORTB5);
      }
      state = S0; break;
    default:
      state = S0;
  }
  ADCSRA |= (1 << ADSC);
}
```

```
int main(void) {
  gpio_init();
  adc_init();
  sei();
  while (1) {}
}
```

Output Signal Measurement





In this experiment, the **A0** pin is connected to an analog input signal, while the **A1** pin is connected to 3.3V. If the voltage at the **A1** pin is higher than the voltage at the **A0** pin, **PB5** is set high; otherwise, it is set low.

The **CH1** waveform represents the analog input applied to the **A0** pin. The voltage at the **A1** pin (**3.3V** reference) is not shown. The **CH2** waveform shows the **PB5** output, which corresponds to the comparison result.

If the voltage at the **A0** pin is very close to the voltage at the **A1** pin, noise (ripple) can cause the output to toggle frequently.

Code Example: Block Mode ADC Sampling

```
#ifndef F_CPU
#define F_CPU 16000000UL
#endif

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdlib.h>

#define BAUD 115200UL
#define UBRR_VALUE ((F_CPU/(8UL*BAUD)) - 1)

extern void usart_init(void);
extern void usart_print(const char *s);

#define ADC_SAMPLES 128
volatile uint16_t adc_buf[ADC_SAMPLES];
volatile uint8_t adc_index = 0;
volatile uint8_t block_done = 0;
```

```
void adc_init(void) {
  ADMUX =
    (1 << REFS0) | // AVcc reference
    (0 << MUX0);   // ADC0 channel
  ADCSRA =
    (1 << ADEN)  | // Enable ADC
    (1 << ADATE) | // Enable auto trigger
    (1 << ADIE)  | // Enter interrupt
    (1 << ADPS2) | // Set prescaler to 128
    (1 << ADPS1) |
    (1 << ADPS0);
  ADCSRB = 0x00; // Use free-running mode
}
```

Code Example: Block Mode ADC Sampling

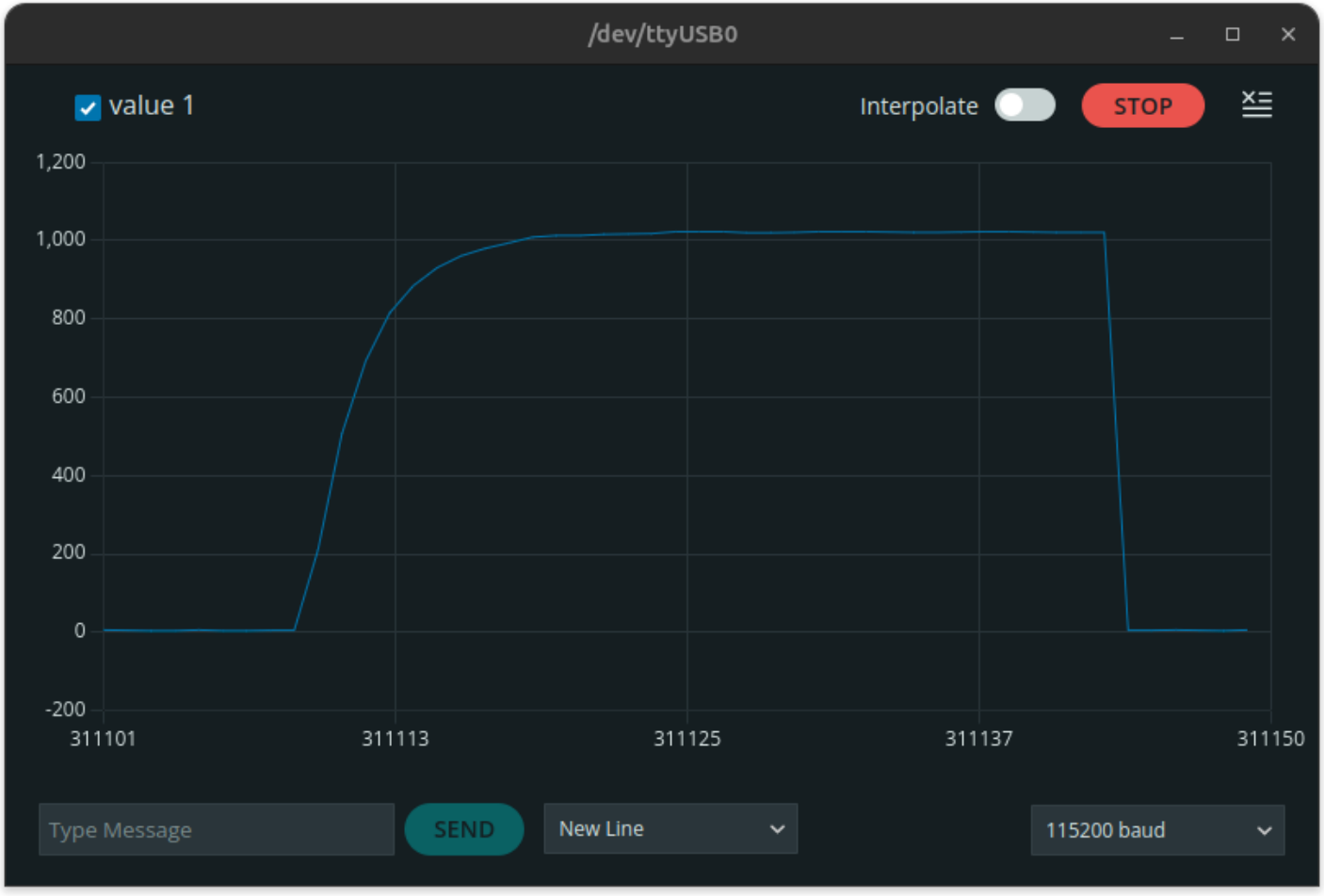
```
void adc_start(void) {
  adc_index = 0;
  block_done = 0;
  ADCSRA |= (1 << ADEN);
  ADCSRA |= (1 << ADSC); // Start ADC
}

void adc_stop(void) {
  ADCSRA &= ~(1 << ADEN); // Disable ADC
}
```

```
ISR(ADC_vect) {
  adc_buf[adc_index++] = ADC;
  if (adc_index >= ADC_SAMPLES) {
    adc_stop(); // Stop the ADC
    block_done = 1; // Sampling complete
  }
}
```

Code Example: Block Mode ADC Sampling

```
int main(void) {
  char txt[8];
  usart_init();
  adc_init();
  sei();
  while (1) {
    adc_start(); // Start the ADC input sampling
    // Wait until the sampling process is done.
    while (!block_done) ;
    for (uint8_t i = 0; i < ADC_SAMPLES; i++) {
      itoa(adc_buf[i], txt, 10);
      usart_print(txt);
      usart_print("\n");
    }
    usart_print("\n");
    _delay_ms(50);
  }
}
```

Arduino Serial Plotter



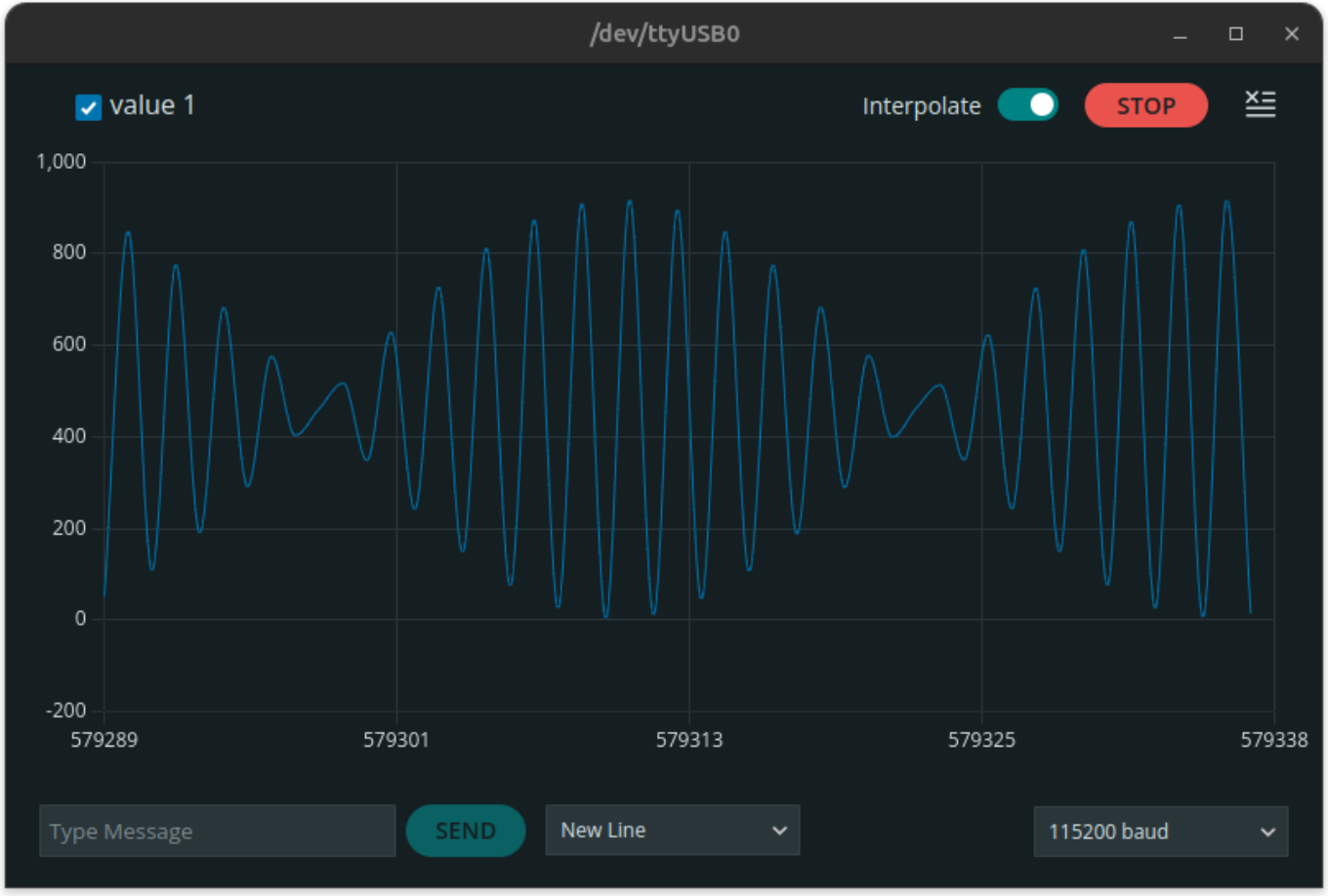
Arduino Serial Plotter

500Hz sine waveform with a DC offset (0V-4V range)

Arduino Serial Plotter

5kHz sine waveform with a DC offset (0V-4V range)

The signal frequency is too high and the **Nyquist criterion** (f_sample > 2*f_signal) is violated.



Code Example: Block Mode ADC Sampling

- In this example, the ADC is configured to run continuously (**free-running**).
- Samples are collected into a **fixed-size buffer (128 samples)**.
- Processing / transmission via USART occurs only after the block is complete.
- ADC sampling is then restarted for the next block.
- In free-running mode, the sampling rate is set only by and the CPU clock and the **ADC prescaler (/128)**.
  - F_CPU = 16MHz, prescaler=128, 13 ADC cycles per sample
    => **Sampling rate = 9.615k Samples/sec**
- It is better to use the ADC in **timer-triggered mode** (timer-triggered block sampling).

Code Example: Block Mode ADC Sampling

- We cannot send each **ADC sample** immediately because the **USART** bandwidth and timing are insufficient to keep up with the ADC without corrupting the sampling process.
- Each ADC sample is transmitted as a string of approximately 4–5 bytes.
- At a sampling rate of **9615 samples/s**:
  - 9615 samples/sec × 5 bytes per sample (string) = 48,075 bytes/s
  - Approximately 480,750 bits/s (assuming 10 bits per UART byte).
  - This required data rate is **much higher than** the available baud rate of 115200 bps.

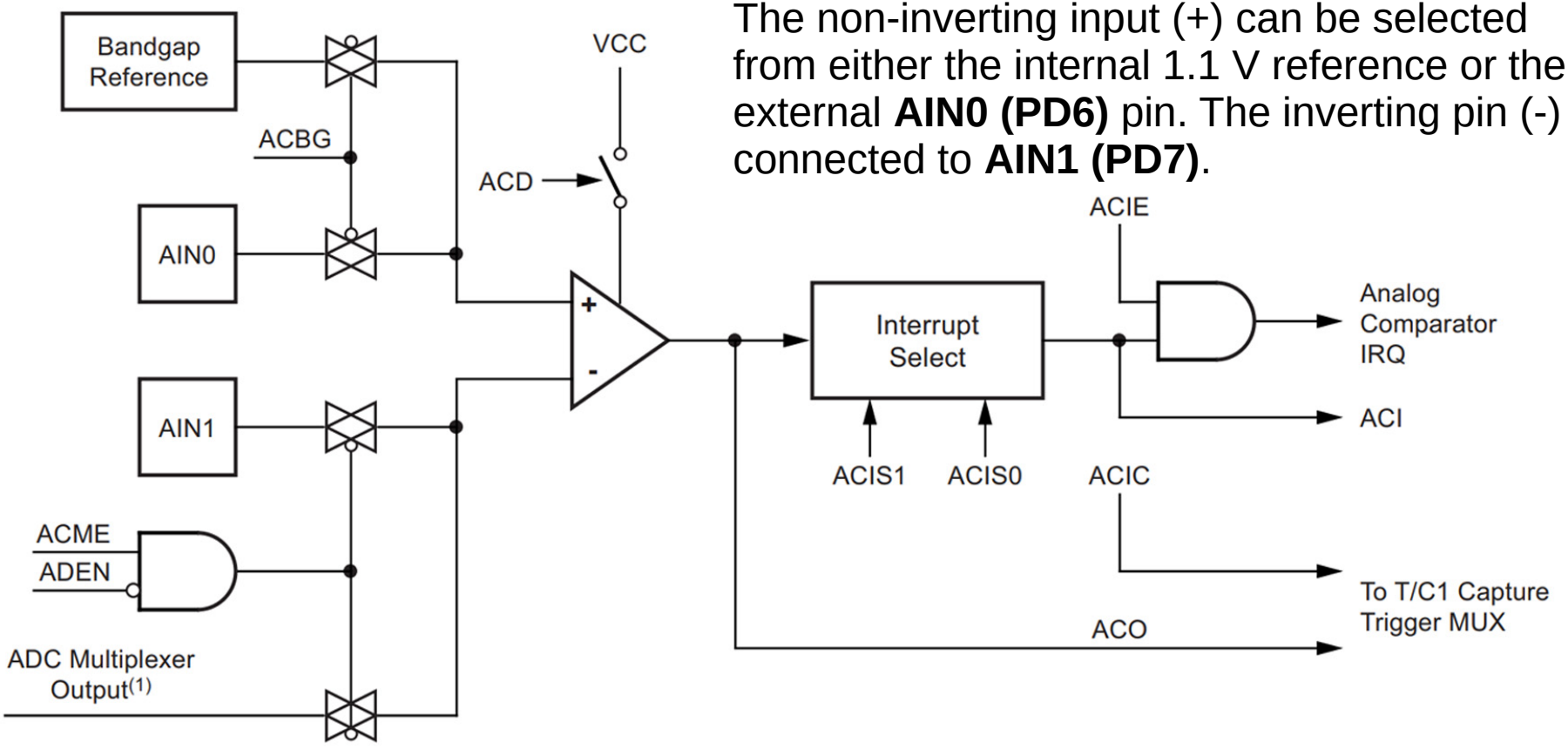
Code Example: Reduced Data Sampling

```
#define DOWN_SAMPLES (100)
// Only 1 out of every 100 samples is stored

volatile uint8_t cnt = 0; // sample count

ISR(ADC_vect) {
  uint16_t sample = ADC; // Read the ADC value
  if (cnt == DOWN_SAMPLES-1) {
     cnt = 0; // Reset the sample counter
     adc_buf[adc_index++] = sample; // Save the sample
  } else {
     cnt++; // Increment the sample counter
  }
  if (adc_index >= ADC_SAMPLES) {
    adc_stop(); // Stop the ADC
    block_done = 1; // Sampling complete
  }
}
```

Analog Comparator

On an ATmega MCU, the **analog comparator** has two inputs: a non-inverting input (+) and an inverting input (−).

The non-inverting input (+) can be selected from either the internal 1.1 V reference or the external **AIN0 (PD6)** pin. The inverting pin (-) is connected to **AIN1 (PD7)**.



Analog Comparator Register

ACSR – Analog Comparator Control and Status Register

Bit	7	6	5	4	3	2	1	0	
0x30 (0x50)	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	ACSR
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	N/A	0	0	0	0	0	

- **Bit 7 – ACD: Analog Comparator Disable**

When this bit is written logic one, the power to the analog comparator is switched off. This bit can be set at any time to turn off the analog comparator. This will reduce power consumption in active and idle mode. When changing the ACD bit, the analog comparator interrupt must be disabled by clearing the ACIE bit in ACSR. Otherwise an interrupt can occur when the bit is changed.

- **Bit 6 – ACBG: Analog Comparator Bandgap Select**

When this bit is set, a fixed bandgap reference voltage replaces the positive input to the analog comparator. When this bit is cleared, AIN0 is applied to the positive input of the analog comparator. When the bandgap referance is used as input to the analog comparator, it will take a certain time for the voltage to stabilize. If not stabilized, the first conversion may give a wrong value.

- **Bit 5 – ACO: Analog Comparator Output**

The output of the analog comparator is synchronized and then directly connected to ACO. The synchronization introduces a delay of 1 - 2 clock cycles.

Analog Comparator Register

- **Bit 4 – ACI: Analog Comparator Interrupt Flag**

This bit is set by hardware when a comparator output event triggers the interrupt mode defined by ACIS1 and ACIS0. The analog comparator interrupt routine is executed if the ACIE bit is set and the I-bit in SREG is set. ACI is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, ACI is cleared by writing a logic one to the flag.

- **Bit 3 – ACIE: Analog Comparator Interrupt Enable**

When the ACIE bit is written logic one and the I-bit in the status register is set, the analog comparator interrupt is activated. When written logic zero, the interrupt is disabled.

- **Bit 2 – ACIC: Analog Comparator Input Capture Enable**

When written logic one, this bit enables the input capture function in Timer/Counter1 to be triggered by the analog comparator. The comparator output is in this case directly connected to the input capture front-end logic, making the comparator utilize the noise canceler and edge select features of the Timer/Counter1 input capture interrupt. When written logic zero, no connection between the analog comparator and the input capture function exists. To make the comparator trigger the Timer/Counter1 input capture interrupt, the ICIE1 bit in the timer interrupt mask register (TIMSK1) must be set.

- **Bits 1, 0 – ACIS1, ACIS0: Analog Comparator Interrupt Mode Select**

These bits determine which comparator events that trigger the analog comparator interrupt.

ACIS1/ACIS0 Settings

ACIS1	ACIS0	Interrupt Mode
0	0	Comparator interrupt on output toggle.
0	1	Reserved
1	0	Comparator interrupt on falling output edge.
1	1	Comparator interrupt on rising output edge.

When changing the ACIS1/ACIS0 bits, the analog comparator interrupt must be disabled by clearing its interrupt enable bit in the ACSR register. Otherwise an interrupt can occur when the bits are changed.

Analog Comparator Registers

ADCSRB – ADC Control and Status Register B

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	

- **Bit 6 – ACME: Analog Comparator Multiplexer Enable**

When this bit is written logic one and the ADC is switched off (ADEN in ADCSRA is zero), the ADC multiplexer selects the negative input to the Analog Comparator. When this bit is written logic zero, AIN1 is applied to the negative input of the Analog Comparator.

Analog Comparator Multiplexed Input

ACME	ADEN	MUX2..0	Analog Comparator Negative Input
0	x	xxx	AIN1
1	1	xxx	AIN1
1	0	000	ADC0
1	0	001	ADC1
1	0	010	ADC2
1	0	011	ADC3
1	0	100	ADC4
1	0	101	ADC5
1	0	110	ADC6
1	0	111	ADC7

DIDR1 Register

DIDR1 – Digital Input Disable Register 1

Bit	7	6	5	4	3	2	1	0	
(0x7F)	–	–	–	–	–	–	AIN1D	AIN0D	DIDR1
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7..2 – Res: Reserved Bits**

These bits are unused bits in the ATmega328P, and will always read as zero.

- **Bit 1, 0 – AIN1D, AIN0D: AIN1, AIN0 Digital Input Disable**

When this bit is written logic one, the digital input buffer on the AIN1/0 pin is disabled. The corresponding PIN register bit will always read as zero when this bit is set. When an analog signal is applied to the AIN1/0 pin and the digital input from this pin is not needed, this bit should be written logic one to reduce power consumption in the digital input buffer.

Code Example: Voltage Comparator

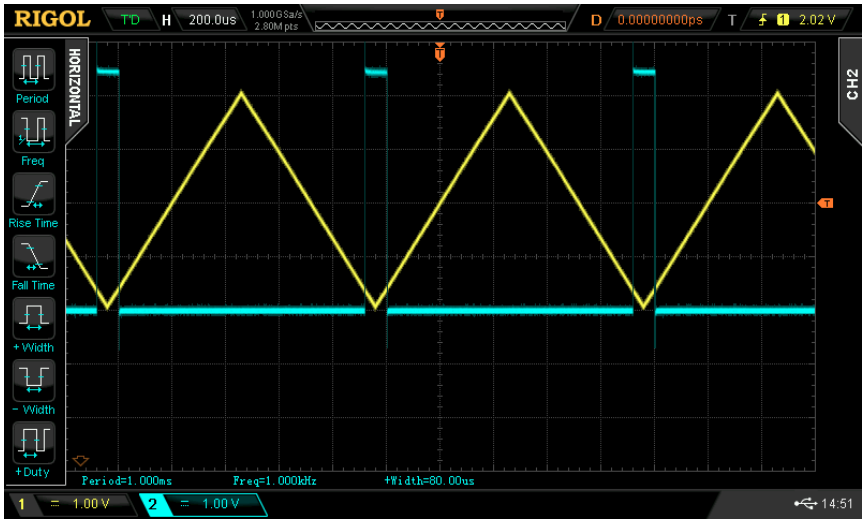
```
#include <avr/io.h>
#include <avr/interrupt.h>

void init_analog_comparator() {
  // Disable the digital input buffer on the AIN1/PD7 pin.
  DIDR1 |= (1 << AIN1D);
  // The AIN0/PD6 pin is not used.
  // The positive input (+0 of the comparator is connected to
  // the internal bandgap reference voltage (1.1V).
  // The AIN1/PD7 pin is connected to the negative input (-).
  ADCSRB &= ~(1 << ACME);
  // Enable Analog Comparator Interrupt on Both Edge.
  ACSR &= ~(1 << ACD);
  ACSR &= ~((1 << ACIS1) | (1 << ACIS0));
  ACSR |= (1 << ACBG);
  ACSR |= (1 << ACIE);
}
```

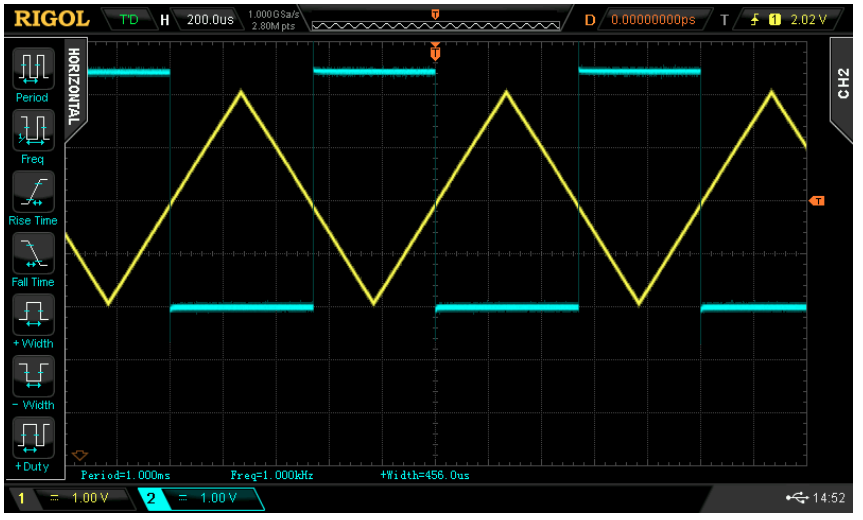
Code Example: Voltage Comparator

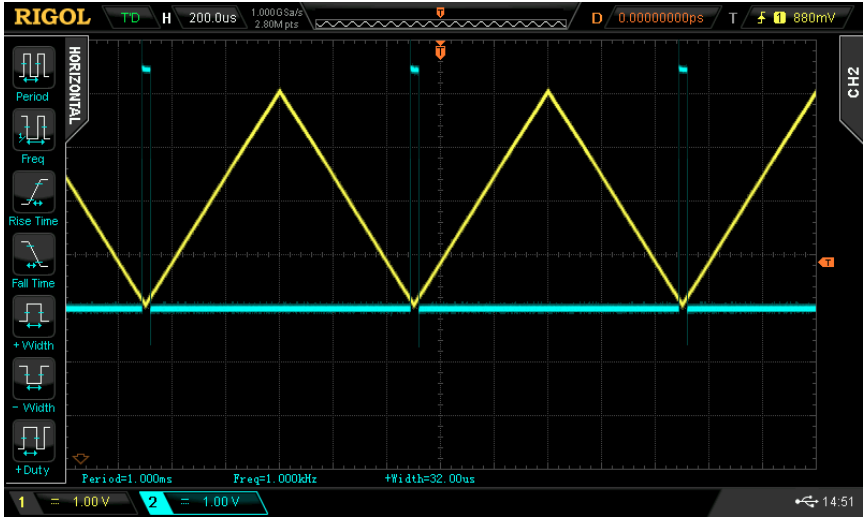
```
ISR(ANALOG_COMP_vect) {
  PINB |= (1 << PB5); // Toggle the PB5 output pin.
}

int main() {
  DDRB |= (1 << DDB5); // Set PB5 as output.
  init_analog_comparator(); // Initialize the analog comparator.
  sei(); // Enable global interrupts.
  while(1) {}
}
```

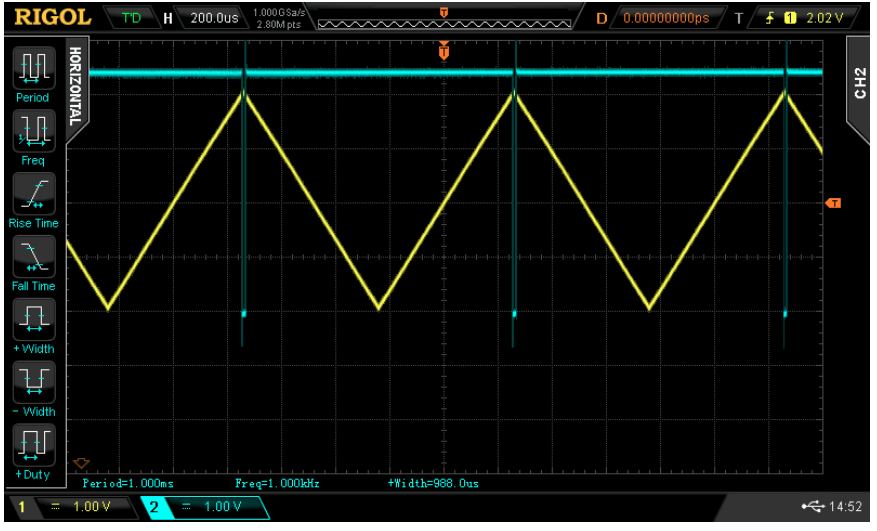
Signal Waveforms

Using the analog comparator to convert a triangular waveform to a pulse signal.









Code Example: ADC Auto-Trigger + Timer 1 Overflow Interrupt

```
#ifndef F_CPU
#define F_CPU 16000000UL
#endif

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/atomic.h>
#include <stdio.h>

extern void usart_init(void);
extern void usart_print(const char *s);

#define PERIOD     (2000-1)
#define LOAD_VALUE (65535-PERIOD)

volatile uint16_t adc_value = 0;
volatile uint8_t  valid = 0;

ISR(TIMER1_OVF_vect) {
  TCNT1 = LOAD_VALUE; // Reload Timer1 count
  PORTB |= (1<<PB5);  // Turn on LED
}
```

```
void timer1_init() {
  TCCR1A = TCCR1B = 0; // Use Normal mode
  // Load the initial value for Timer1 count
  TCNT1 = LOAD_VALUE;
  // Clear Timer1 overflow interrupt flag
  TIFR1  |= (1<<TOV1);
  // Enable Timer1 overflow interrupt
  TIMSK1 |= (1<<TOIE1);
  // Set up Timer/Counter1 in Normal mode
  // prescaler = 8
  TCCR1B |= (1<<CS11);
}
```

- The **Timer1** is configured to run in Normal Mode with Overflow interrupt (TOIE1) enabled.
- The **Timer1** counts from 0x0000 to 0xFFFF (65535).
- When it overflows (wraps from 0xFFFF to 0x0000), the TOV1 flag is set.
- The prescaler is set to 8 and the **Timer1** runs at 16MHz/8 = 2MHz.

Code Example: ADC Auto-Trigger + Timer 1 Overflow Interrupt

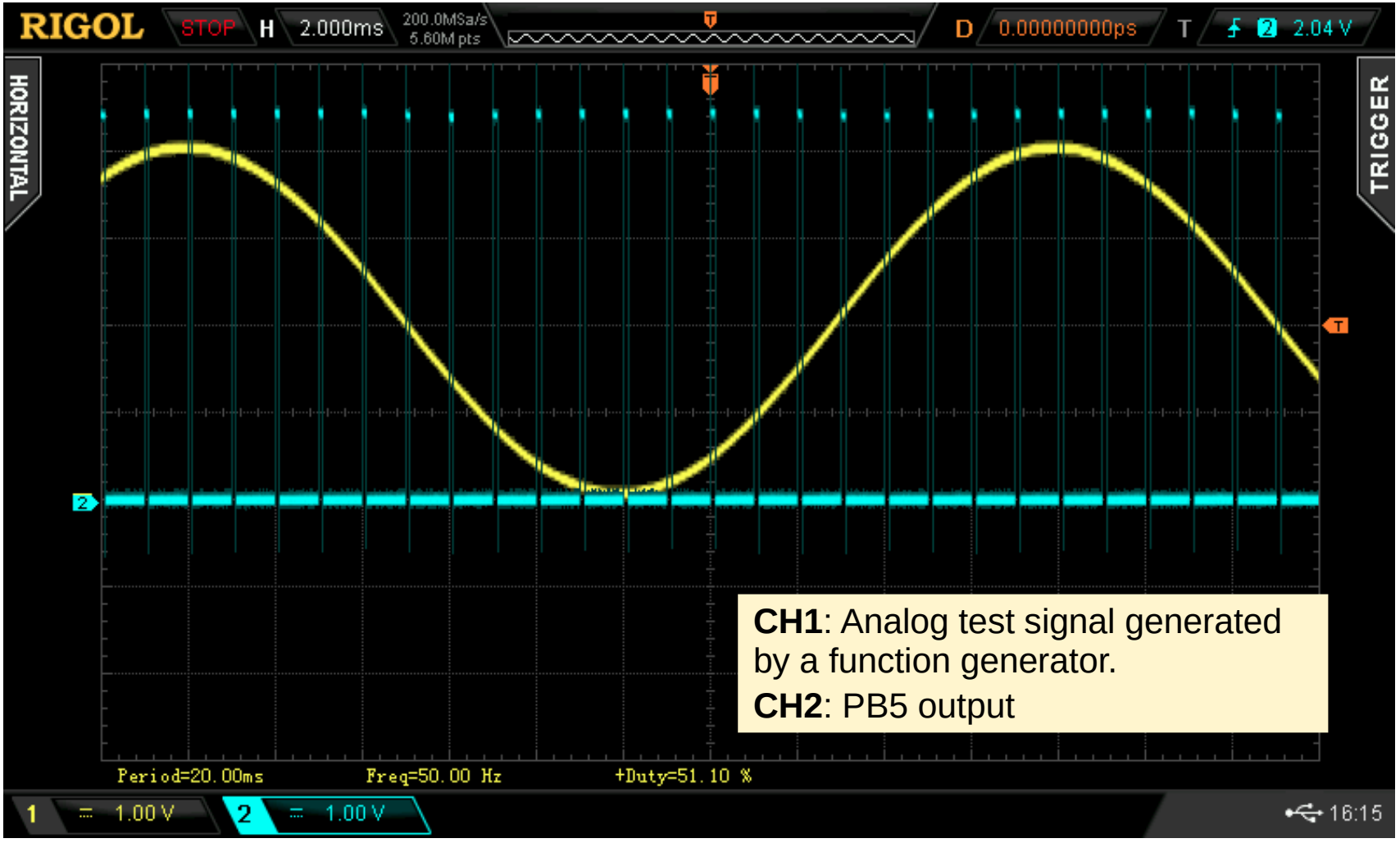
```
void adc_init() {
  // Set PC0/A0 as an input pin
  DDRC &= ~(1<<DDC0);
  // Disable Digital Input Buffer on A0
  DIDR0 |= (1<<ADC0D);
  // Set reference voltage to AVCC
  ADMUX |= (1<<REFS0);
  // Auto-Trigger Source: Timer1 Overflow
  ADCSRB = (1<<ADTS2) | (1<<ADTS1);
  // Enable ADC and set prescaler to 128
  ADCSRA = (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
     | (1<<ADIE) | (1<<ADEN) | (1<<ADATE);
}
```

```
// ISR for ADC conversion complete
ISR(ADC_vect) {
  adc_value = ADC; // Save the ADC value
  PORTB &= ~(1<<PB5); // Turn off LED
  valid = 1; // Set sample valid flag
}
```

Code Example: ADC Auto-Trigger + Timer 1 Overflow Interrupt

```
int main(void) {
  char buf[8];
  uint16_t value;
  DDRB |= (1<<PB5);
  usart_init();
  adc_init();    // Initialize ADC
  timer1_init(); // Initialize Timer1
  sei();         // Enable global interrupts
  while (1) {
    if (valid) {
      ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
        valid = 0;
        value = adc_value;
      }
      snprintf(buf, sizeof(buf), "%u\r\n", value);
      usart_print(buf);
    }
  }
  return 0;
}
```

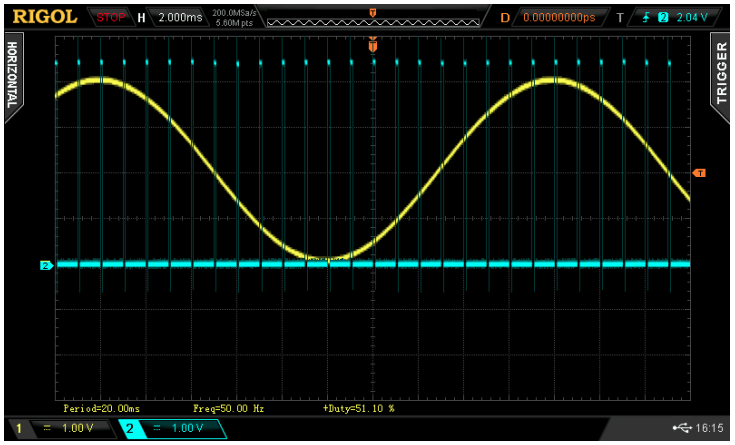
Signal Waveforms



CH1: Analog test signal generated by a function generator.

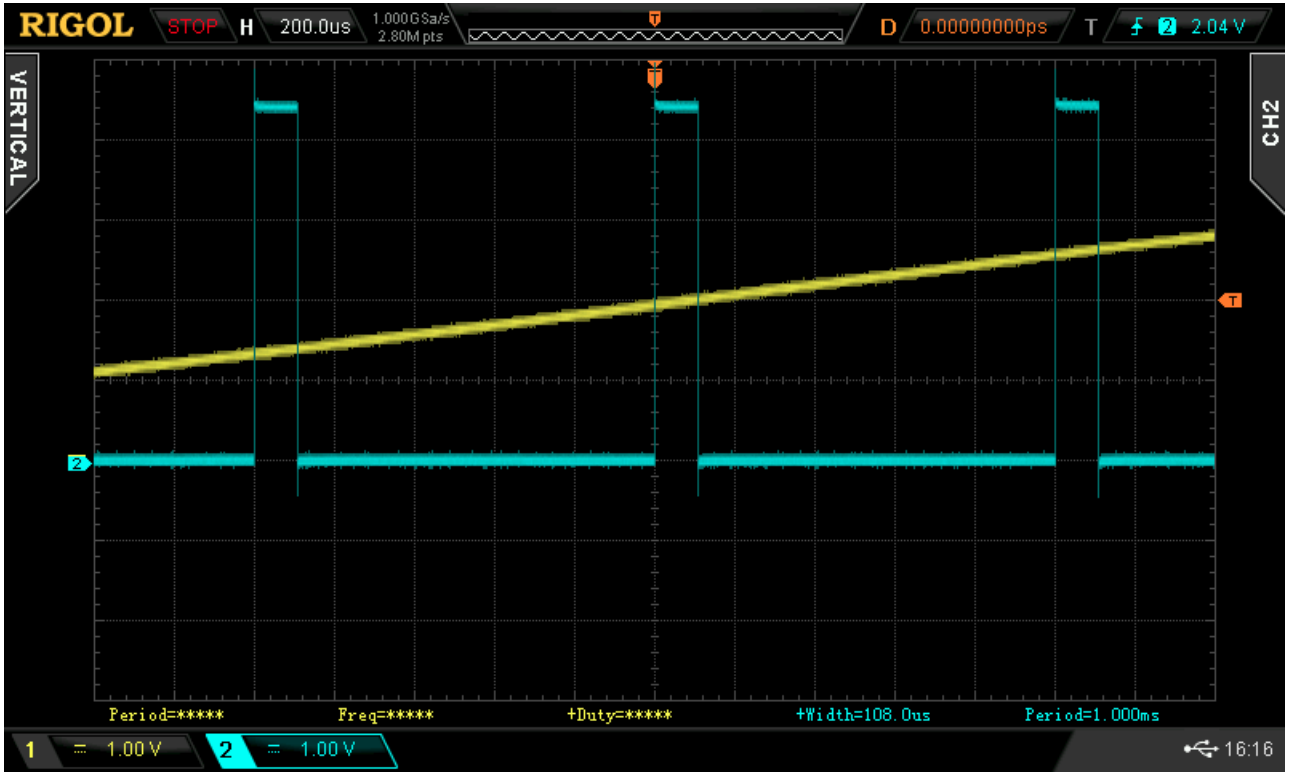
CH2: PB5 output

Signal Waveforms



CH1: Analog test signal generated by a function generator.

CH2: PB5 output



Each ADC conversion takes approximately 108 µs.

ADC conversions occur every 1 msec.