Inatel Instituto Nacional de Telecomunicações	RELATÓRIO 12	Data: / /
	Disciplina: E209	
	Prof: Yvo Marcelo Chiaradia Masseli Monitores: João Lucas/Luan Siqueira/Matheus/Vinícius	
Conteúdo: Microcontrolador Atmega328p		
Tema: Conversor A/D		
Nome: Francisco José Carvalho Junior	Matrícula:1628	Curso:GEC

OBJETIVOS:

- Utilizar as ferramentas de simulação para desenvolver programas para o Atmega328p.
- Desenvolver um programa de controle que faça uso do Conversor AD interno.
- Utilizar as entradas e saídas do Atmega328p com circuitos de aplicação.

Parte Teórica

Conversor Analógico/Digital (ADC)

O bloco de conversão A/D (analógico/digital) é um periférico cada vez mais comum nos microcontroladores. O conversor A/D realiza o processo de **amostragem**, **desratização** e **codificação** de uma tensão aplicada a um pino de entrada analógica do MCU. Quando uma conversão é iniciada, o valor instantâneo da tensão no pino é retido pelo bloco conversor. Esse valor de tensão é associado a uma palavra binária através do **método de aproximação sucessiva (SAR).** No caso do ATMega328p, a palavra binária tem **10 bits**, sendo armazenada uma parte no registro **ADCL**, e outra no **ADCH**. Dessa forma, a tensão de entrada é convertida utilizando as tensões de referência em valores digitais de **0 a 1023**.

Para controlar o processo de conversão A/D, os registros **ADMUX** e **ADCSRA** (em anexo) devem ser configurados para que o canal A/D do Atmega328p opere corretamente. O Atmega328p permite que sejam utilizadas diversas formas de conversão em função da aplicação. Nós utilizaremos a função mais simples que é a conversão única. Nesse método uma conversão é disparada quando o bit **ADEN (ADC ENABLE)** e o **ADSC (ADC Start Conversion)** são setados (ligados). Quando a conversão chega ao fim, a flag **ADIF** vai para 1, indicando que a leitura do valor convertido pode ser realizada.

Os pinos utilizados como entradas analógicas são aqueles identificados de **ADC0 a ADC5 (PC0 à PC5).**

Como a conversão é em 10 bits, a tensão de entrada é convertida seguindo a expressão:

Palavra_Digital **(ADCH + ADCL)** = (Vin * 1023) / V_{ref}, ou ainda,

Vin = (Palavra_Digital * V_{ref}) / 1023

A relação $V_{ref}/1024$ é conhecida como resolução do conversor AD, ou seja, a menor variação da tensão de entrada que provoca a alteração de 1 bit na palavra digital de saída.

Vamos considerar a tensão de alimentação VDD = 5V. Sabendo que a resolução do Atmega328 é de 5V/1024 = 4,88mV. Assim, caso o conversor retorne um valor digital lido de 430, a tensão existente na entrada analógica do microcontrolador seria de 2,0984 V.

Resumindo, para calcularmos o valor da tensão existente na entrada podemos usar uma simples regra de três:

```
5V ---------- 1023
 X ---------- valorLido
```

Porém, percebe-se que o valor lido facilmente será um valor de tensão não inteiro e sabemos que para o microcontrolador operar números nesse formato não é uma tarefa muito fácil, ou seja, para fazer operações com números do tipo **float**, ele gasta um tempo muito elevado. Assim, preferimos sempre trabalhar em milivolts, resultando em:

```
5000mV ---------- 1023
     X ---------- valorLido
```

A linha de programa que é capaz de calcular a tensão aplicada na entrada analógica do microcontrolador seria (considere que a variável que armazena o valor digital é *valorLido*):

```
tensao = (valorLido*5000)/1023;
```

Aqui vale ressaltar que se a linha tiver escrita exatamente como acima, pode - se gerar um resultado incorreto devido a limitação de armazenamento da variável "tensão". Vejamos, se **tensao** for do tipo **unsigned int**, ela não poderá armazenar o resultado de valorLido*5000, caso valorLido tenha seu valor máximo de 1023, uma vez que 1023*5000=5115000, o que ultrapassa o limite de armazenamento de uma variável do tipo **unsigned int**, que é 65535!

Por fim, é preferível fazer o seguinte:

```
unsigned long int aux;
unsigned int valorLido;
unsigned int tensao;
…
      //utiliza-se uma variável auxiliar para realizar o cálculo
      aux = valorLido*5000;
      //depois, divide-se por 1023
      aux = aux/1023;
      //retorna o resultado, transformando-o para o tipo unsigned int
      tensao = (unsigned int) aux;
```

A última linha utiliza o recurso denominado ***casting*** no qual um tipo de variável é convertido para o outro. No caso, de **unsigned long int** para **unsigned int**.

Como configurar o bloco ADC

Para fazermos uso do conversor AD, devemos configurá-lo. Primeiramente vamos configurar a tensão de referência (V_{ref}), o preescaler utilizado, e habilitar o ADC.

Para definir o V_{ref} devemos setar os BITs escolhidos no registro **ADMUX**, por exemplo, configurando V_{ref} = 5V:

```
ADMUX = (1 << REFS0); // 0b01000000
```

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

Para configurar o preescaler se faz uso do registro **ADCSRA:** (analisar User Guide (em anexo) para escolha)

```
ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0); // divisor 128
```

Table 23-5. 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

Por fim habilitamos o ADC:

```
ADCSRA |= (1 << ADEN);
```

Lembre-se que as tensões de referência deverão ser modificadas de acordo com o sensor em uso. As tensões podem ser modificadas ao longo do programa de acordo com o necessário.

Após as configurações principais devemos iniciar a parte de conversão do nosso AD. Para isso faremos uso dos registros **ADCSRA e ADMUX.**

Vamos informar qual a entrada queremos converter:

```
ADMUX = (ADMUX & 0xF8) | input;
```

Agora devemos começar a conversão, para isso é necessário setar o bit **ADSC.**

```
ADCSRA |= (1 << ADSC); // ADC Start Conversion
```

Um detalhe importante é que só podemos fazer a conversão de 1 canal por vez, ou seja, se tivermos 2 canais, devemos esperar um pouco para iniciar a conversão do outro.

Para uma boa conversão, devemos ter o entendimento que ruídos serão convertidos também, imagine a seguinte situação:

Você irá fazer uma conversão simples de um sensor que estava com tensão de 1,2V e no momento de converter um ruído 5 volts entra no se sistema por um breve período de tempo. Esse ruído irá gerar uma resposta no **ADC result (ADCL + ADCH)** de 1023 (5V) e não 245 (1,2V) que seria o correto. Para evitarmos esse tipo de erro devemos criar uma amostragem de, por exemplo, 100 leituras e tirar a média de todas elas, isso irá garantir que o valor lido está aproximado ao que realmente deveria ter sido lido.

Para criar essa amostragem podemos fazer uso de um **for().** Dentro desse for devemos iniciar a conversão, verificar se ela já terminou e armazenar o valor lido em uma variável de sua escolha. Lembre-se que como o valor armazenado pode ser muito maior que um inteiro (INT) é necessário criar que ela seja um **Long Int.**

Um código genérico para a conversão por amostragem será:

```
for (i = 0; i < NUMERO_DE_AMOSTRAS; i++) {
      ADCSRA |= (1 << ADSC);
      while (!(ADCSRA & (1 << ADIF)));
      var_temp = ADCL;
      var_temp += (ADCH << 8);
      var_armazenagem += var_temp;
}
```

Esse código acima espera a conversão AD terminar (linha WHILE). Internamente a flag **ADIF** é setada quando uma conversão foi finalizada e armazenada e resetada assim que a interrupção for tratada (ela é tratada mesmo sem um ISR).

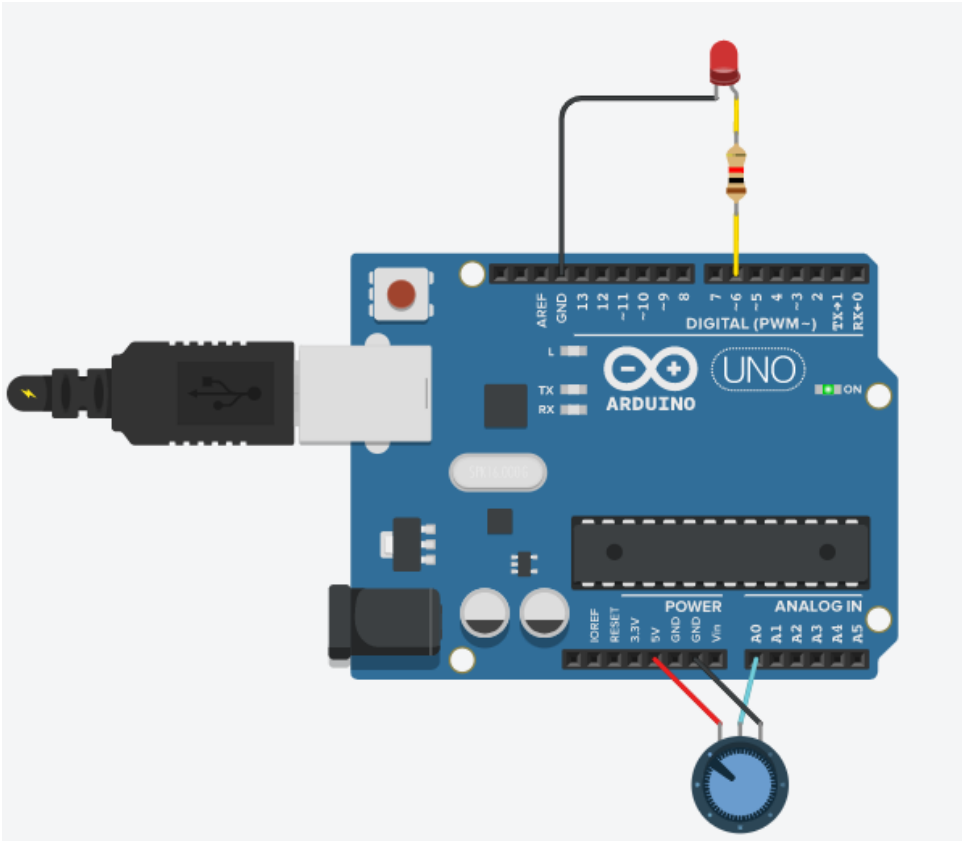
Parte Prática

Programa 1) Exemplo de ADC como voltímetro

O projeto é um sistema que realiza a medição da tensão na entrada analógica do microcontrolador e controla a potência do LED verde via PWM de acordo com o valor convertido (diretamente proporcional).

A tensão de entrada é aplicada através de um potenciômetro, variando de 0V até 5V. No programa é feito a conversão de binário para tensão. A dica aqui é trabalhar com a tensão em milivolts com o objetivo de evitar operações em ponto flutuante.

Para conectar o potenciômetro, deve-se conectar conforme a imagem abaixo.



```
// Valor de THRESHOLD
#define THRES 500
void ADC_init(void)
{
    // Configurando Vref para VCC = 5V
    ADMUX = (1 << REFS0);
    /*
        ADC ativado e preescaler de 128
        16MHz / 128 = 125kHz
        ADEN = ADC Enable, ativa o ADC
        ADPSx = ADC Prescaler Select Bits
        1 1 1 = clock / 128
    */
    ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
}
int ADC_read(u8 ch)
{
    char i;
    int ADC_temp = 0; // ADC temporário, para manipular leitura
    int ADC_read = 0; // ADC_read
    ch &= 0x07;
    // Zerar os 3 primeiros bits e manter o resto
    ADMUX = (ADMUX & 0xF8) | ch;
    // ADSC (ADC Start Conversion)
    ADCSRA |= (1 << ADSC); // Faça uma conversão
    // ADIF (ADC Interrupt Flag) é setada quando o ADC pede interrupção
    // e resetada quando o vetor de interrupção
    // é tratado.
    while (!(ADCSRA & (1 << ADIF)))
        ;                   // Aguarde a conversão do sinal
    for (i = 0; i < 8; i++) // Fazendo a conversão 8 vezes para maior precisão
    {
        ADCSRA |= (1 << ADSC); // Faça uma conversão
        while (!(ADCSRA & (1 << ADIF)))
            ;                   // Aguarde a conversão do sinal
        ADC_temp = ADCL;        // lê o registro ADCL
        ADC_temp += (ADCH << 8); // lê o registro ADCH
        ADC_read += ADC_temp;    // Acumula o resultado (8 amostras) para média
    }
    ADC_read = ADC_read >> 3; // média das 8 amostras
    return ADC_read;
}
void setup()
{
    //Configurando o PWM
    TCCR0A |= (1 << WGM01) | (1 << WGM00); // Modo fast PWM
    //TCNT0 = 0 -> PD6(OC0A) = 1
    //TCNT0 = OCR0A -> PD6(OC0A) = 0
    TCCR0A |= (1 << COM0A1); //TCNT0 = 0, PD6(OC0A) = 1
    TCCR0B |= (1 << CS00); // preescaler 1
    //Configurando INT0
    EICRA |= (1 << ISC01);
    EIMSK |= (1 << INT0);
    //Configurando pd6 saida
    DDRD |= (1 << PD6);
    sei();

}

void loop(){
    u16 adc_result0, adc_result1;
    unsigned long int aux;
    unsigned int tensao;
    DDRB = (1 << PB5); // PB5 como saída
    Serial.begin(9600);
    ADC_init(); // Inicializa ADC
    adc_result0 = ADC_read(ADC0D); // lê o valor do ADC0 = PC0
        _delay_ms(50);                  // Tempo para troca de canal
        adc_result1 = ADC_read(ADC1D); // lê o valor do ADC1 = PC1
        // condição do led
        if (adc_result0 < THRES && adc_result1 < THRES)
            PORTB |= (1 << PB5);
        else
            PORTB &= ~(1 << PB5);
        // Mostra o valor na serial
        Serial.print("ADC0: ");
        Serial.println(adc_result0);

        //OCR0A recebe valor do potenciometro
        OCR0A = adc_result0;

        //Serial.print("ADC1: ");
```

```
        //Serial.println(adc_result1);
        /*
            CÁLCULO TENSÃO
            aux = (long)adc_result0 * 5000;
            aux /= 1023;
            tensao = (unsigned int)aux;
            Serial.print("TENSAO0: ");
            Serial.println(tensao);
        */
        _delay_ms(1000);
}
```

Programa 2)

Modifique o programa anterior para que o controle da potência do LED seja feita de maneira inversamente proporcional.

Única parte do código que muda

```
void loop(){
    u16 adc_result0, adc_result1;
    unsigned long int aux;
    unsigned int tensao;
    DDRB = (1 << PB5); // PB5 como saída
    Serial.begin(9600);
    ADC_init(); // Inicializa ADC
    adc_result0 = ADC_read(ADC0D); // lê o valor do ADC0 = PC0
        _delay_ms(50);                  // Tempo para troca de canal
        adc_result1 = ADC_read(ADC1D); // lê o valor do ADC1 = PC1
        // condição do led
        if (adc_result0 < THRES && adc_result1 < THRES)
            PORTB |= (1 << PB5);
        else
            PORTB &= ~(1 << PB5);
        // Mostra o valor na serial
        Serial.print("ADC0: ");
        Serial.println(adc_result0);

        //OCR0A recebe valor do potenciometro inversamente proporcional
        OCR0A = abs(1023 - adc_result0);

        //Serial.print("ADC1: ");
        //Serial.println(adc_result1);
        /*
            CÁLCULO TENSÃO
            aux = (long)adc_result0 * 5000;
            aux /= 1023;
            tensao = (unsigned int)aux;
            Serial.print("TENSAO0: ");
            Serial.println(tensao);
        */
        _delay_ms(1000);
}
```

ANEXO) Código

```
// Valor de THRESHOLD
#define THRES 500
void ADC_init(void)
{
    // Configurando Vref para VCC = 5V
    ADMUX = (1 << REFS0);
    /*
        ADC ativado e preescaler de 128
        16MHz / 128 = 125kHz
        ADEN = ADC Enable, ativa o ADC
        ADPSx = ADC Prescaler Select Bits
        1 1 1 = clock / 128
    */
    ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
}
int ADC_read(u8 ch)
{
    char i;
    int ADC_temp = 0; // ADC temporário, para manipular leitura
    int ADC_read = 0; // ADC_read
    ch &= 0x07;
    // Zerar os 3 primeiros bits e manter o resto
    ADMUX = (ADMUX & 0xF8) | ch;
    // ADSC (ADC Start Conversion)
    ADCSRA |= (1 << ADSC); // Faça uma conversão
    // ADIF (ADC Interrupt Flag) é setada quando o ADC pede interrupção
    // e resetada quando o vetor de interrupção
    // é tratado.
    while (!(ADCSRA & (1 << ADIF)))
        ;                       // Aguarde a conversão do sinal
    for (i = 0; i < 8; i++) // Fazendo a conversão 8 vezes para maior precisão
    {
        ADCSRA |= (1 << ADSC); // Faça uma conversão
        while (!(ADCSRA & (1 << ADIF)))
            ;                           // Aguarde a conversão do sinal
        ADC_temp = ADCL;                // lê o registro ADCL
        6 ADC_temp += (ADCH << 8); // lê o registro ADCH
        ADC_read += ADC_temp;           // Acumula o resultado (8 amostras) para média
    }
    ADC_read = ADC_read >> 3; // média das 8 amostras
    return ADC_read;
}
int main()
{
    u16 adc_result0, adc_result1;
    unsigned long int aux;
    unsigned int tensao;
    DDRB = (1 << PB5); // PB5 como saída
    Serial.begin(9600);
    ADC_init(); // Inicializa ADC
    while (1)
    {
```

Registros principais (fonte: UserManual ATMega328p)

23.9.1 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. For a complete description of this bit, see Section 23.9.3 "ADCL and ADCH – The ADC Data Register" on page 219.

- **Bit 4 – Res: Reserved Bit**

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

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

The value of these bits selects which analog inputs are connected to the ADC. See Table 23-4 on page 218 for details. 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).

Table 23-4. 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.

23.9.2 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.

- **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.

23.9.3 ADCL and ADCH – The ADC Data Register

23.9.3.1 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	

23.9.3.2 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	

When an ADC conversion is complete, the result is found in these two registers.

When ADCL is read, the ADC data register is not updated until ADCH is read. Consequently, if the result is left adjusted and no more than 8-bit precision is required, it is sufficient to read ADCH. Otherwise, ADCL must be read first, then ADCH.

The ADLAR bit in ADMUX, and the MUXn bits in ADMUX affect the way the result is read from the registers. If ADLAR is set, the result is left adjusted. If ADLAR is cleared (default), the result is right adjusted.

- **ADC9:0: ADC Conversion Result**

These bits represent the result from the conversion, as detailed in Section 23.7 "ADC Conversion Result" on page 215.