

Microcontroladores - AVR

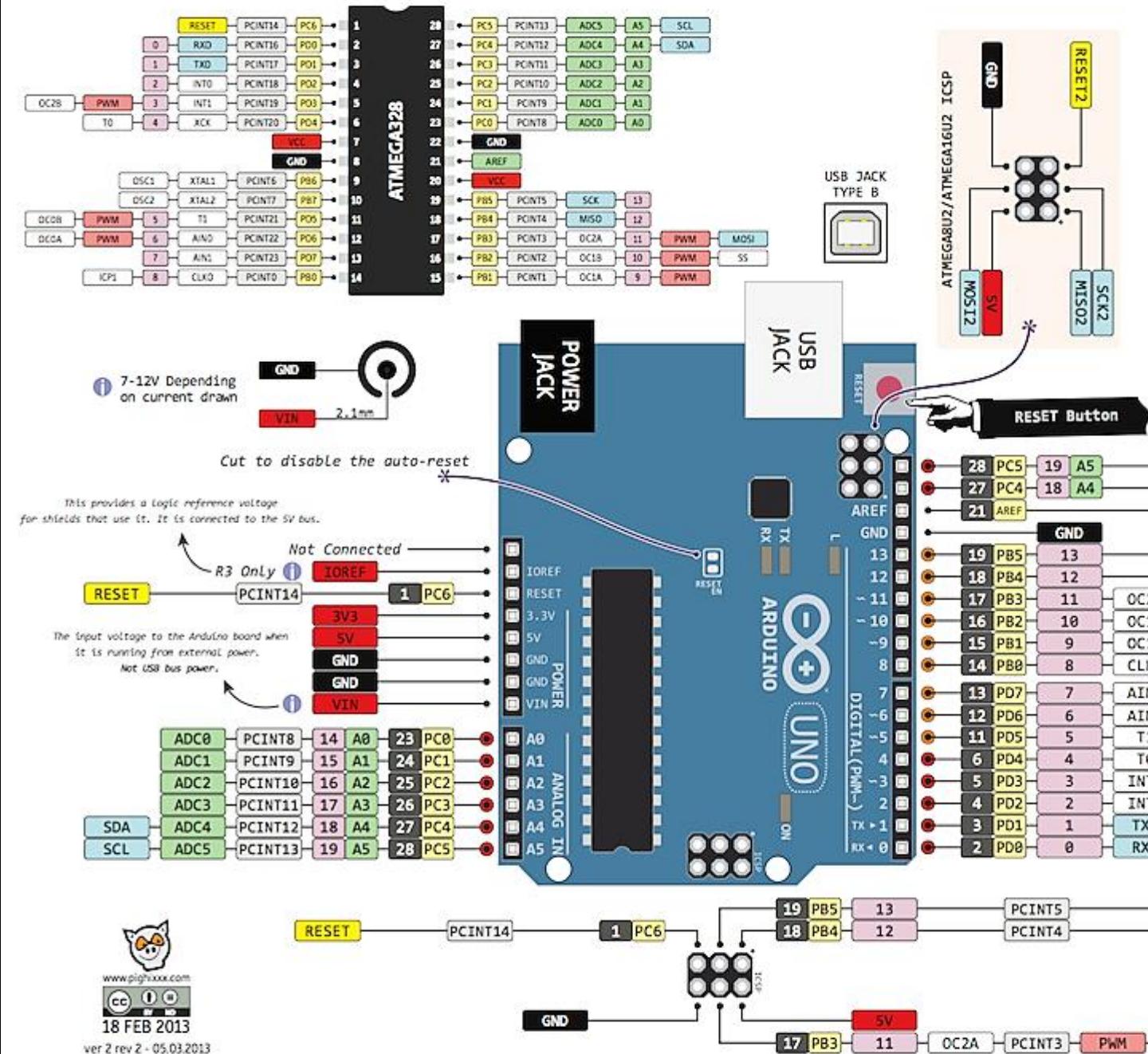
Introdução

Os microcontroladores AVR da fabricante ATMEL são de 8 bits e 32 bits.

O código fonte (programa – firmware) para o microcontrolador necessita ser escrito, compilado, depurado e gravado.

Todas estas tarefas são realizadas com o suporte de softwares adequados.

THE
DEFINITIVE
ARDUINO
UNO
PINOUT DIAGRAM

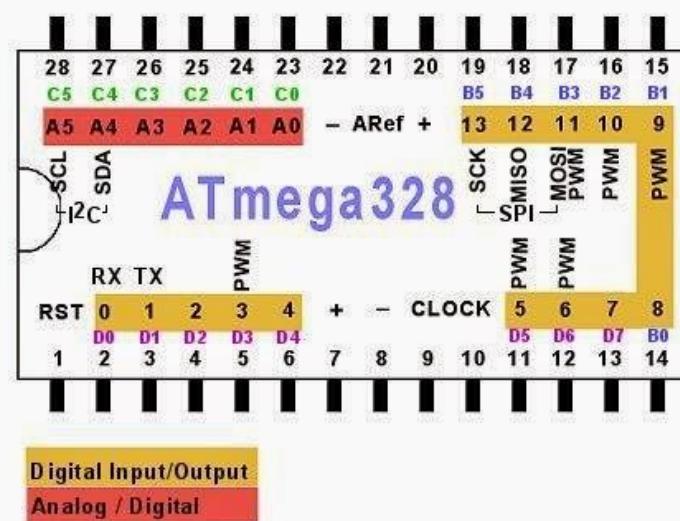


ATMEGA328p

(PCINT14/RESET)	PC6	1	28	PC5	(ADC5/SCL/PCINT13)
(PCINT16/RXD)	PD0	2	27	PC4	(ADC4/SDA/PCINT12)
(PCINT17/TXD)	PD1	3	26	PC3	(ADC3/PCINT11)
(PCINT18/INT0)	PD2	4	25	PC2	(ADC2/PCINT10)
(PCINT19/OC2B/INT1)	PD3	5	24	PC1	(ADC1/PCINT9)
(PCINT20/XCK/T0)	PD4	6	23	PC0	(ADC0/PCINT8)
VCC		7	22	GND	
GND		8	21	AREF	
(PCINT6/XTAL1/TOSC1)	PB6	9	20	AVCC	
(PCINT7/XTAL2/TOSC2)	PB7	10	19	PB5	(SCK/PCINT5)
(PCINT21/OC0B/T1)	PD5	11	18	PB4	(MISO/PCINT4)
(PCINT22/OC0A/AIN0)	PD6	12	17	PB3	(MOSI/OC2A/PCINT3)
(PCINT23/AIN1)	PD7	13	16	PB2	(SS/OC1B/PCINT2)
(PCINT0/CLKO/ICP1)	PB0	14	15	PB1	(OC1A/PCINT1)

Correspondência entre os pinos do ATMEGA e do Arduino

Arduino	ATmega328	Arduino	ATmega328	Arduino	Atmega328
<i>Analog In</i>	PORTC		PORTD		PORTB
A0	PC0	0	PD0	8	PB0
A1	PC1	1	PD1	9	PB1
A2	PC2	2	PD2	10	PB2
A3	PC3	3	PD3	11	PB3
A4	PC4	4	PD4	12	PB4
A5	PC5	5	PD5	13	PB5
		6	PD6		
		7	PD7		



PORTB: É uma porta bidirecional de 8 bits, com resistores *pull-up* internos, selecionáveis para cada pino.

As funções alternativas para os pinos da PORTB são:

- XTAL
- SPI
- Comparadores de saída (*Output Compare*) para temporizadores.

PORTC: É uma porta bidirecional de 7 bits, com resistores pull-up internos, selecionáveis para cada pino.

As funções alternativas para os pinos do PORTC são:

- Entradas analógicas (ADC)
- I2C.

PORTD: É uma porta bidirecional de 8 bits, com resistores pull-up internos, selecionáveis para cada pino.

As funções alternativas para os pinos do PORTD são:

- Porta serial do USART.
- Interrupções externas INT0 e INT1.
- Comparadores de saída para temporizadores.

AVcc: É o pino de tensão para o conversor analógico para digital (ADC).

AREF: pino de referência análogo para o ADC

Características

ATmega328 Features	
No. of Pins	28
CPU	RISC 8-Bit AVR
Operating Voltage	1.8 to 5.5 V
Program Memory	32KB
Program Memory Type	Flash
SRAM	2048 Bytes
EEPROM	1024 Bytes
ADC	10-Bit
Number of ADC Channels	8
PWM Pins	6
Comparator	1
Packages (4)	8-pin PDIP32-lead TQFP28-pad QFN/MLF32-pad QFN/MLF
Oscillator	up to 20 MHz
Timer (3)	8-Bit x 2 & 16-Bit x 1

Características

Enhanced Power on Reset	Yes
Power Up Timer	Yes
I/O Pins	23
Manufacturer	Microchip
SPI	Yes
I2C	Yes
Watchdog Timer	Yes
Brown out detect (BOD)	Yes
Reset	Yes
USI (Universal Serial Interface)	Yes
Minimum Operating Temperature	-40 C to +85 C

- 131 Instruções poderosas, a maioria executada em um único ciclo de relógio.
- Um banco de 32x8 registros de uso geral.
- Até 20 MIPS (Milhões de instruções por segundo) a 20 MHz.
- Um multiplicador de hardware on-chip de 2 ciclos.
- Memória de programa FLASH de 32 KB, programável dentro do sistema.
- Memória SRAM interna de 2 KBytes.
- Memória EEPROM de 1 KByte.
- 2 Temporizadores / Contadores de 8 bits.
- 1 Temporizador / Contador de 16 bits.
- 6 canais PWM.
- 6 canais analógicos para o ADC.
- 1 porta serial USART.
- 1 interface serial SPI.
- 1 interface serial de 2 fios, compatível com I2C.
- 1 temporizador de vigilância.
- 1 Um comparador analógico *on-chip*.
- Interrupções.
- Vários modos de baixo consumo

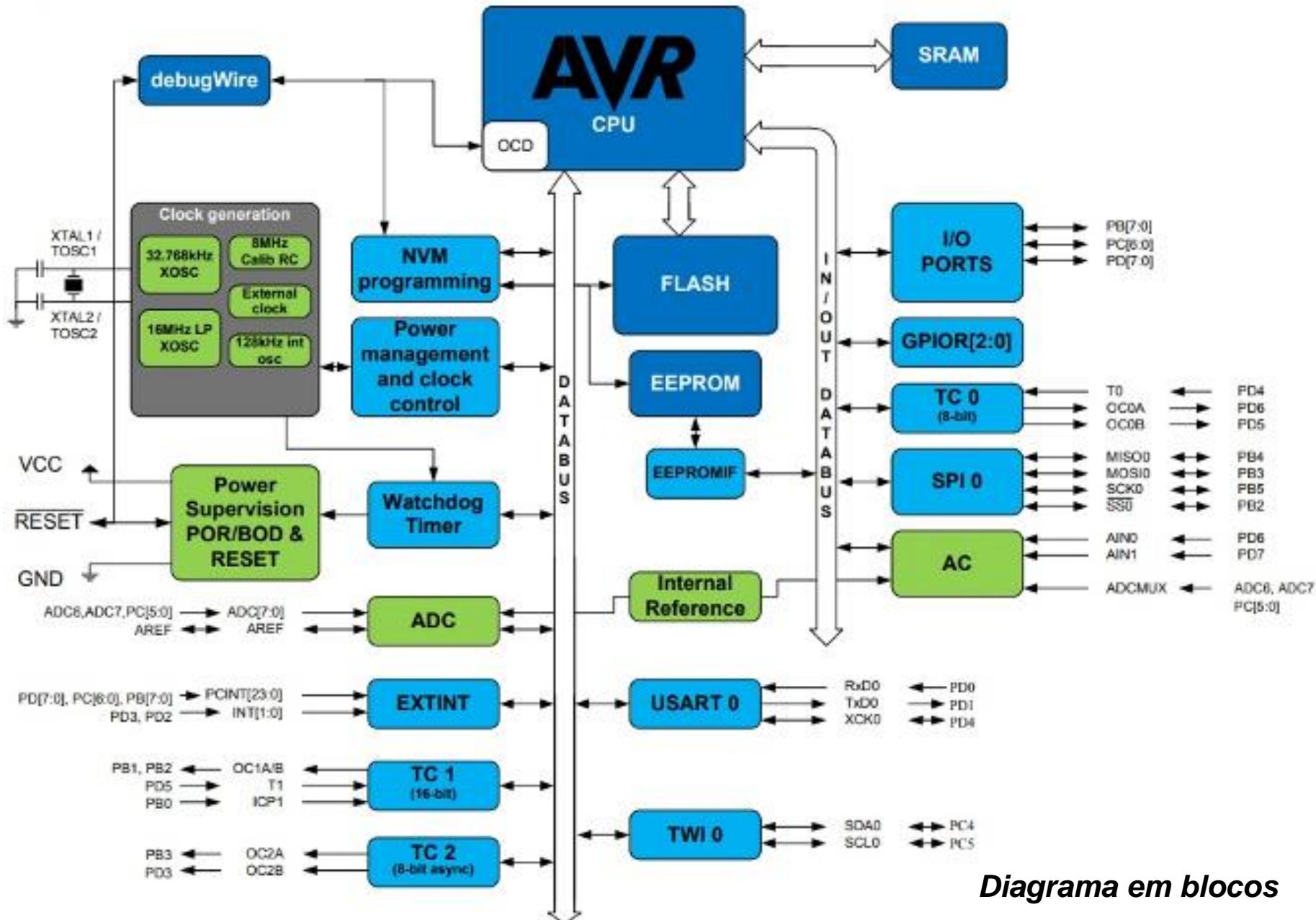
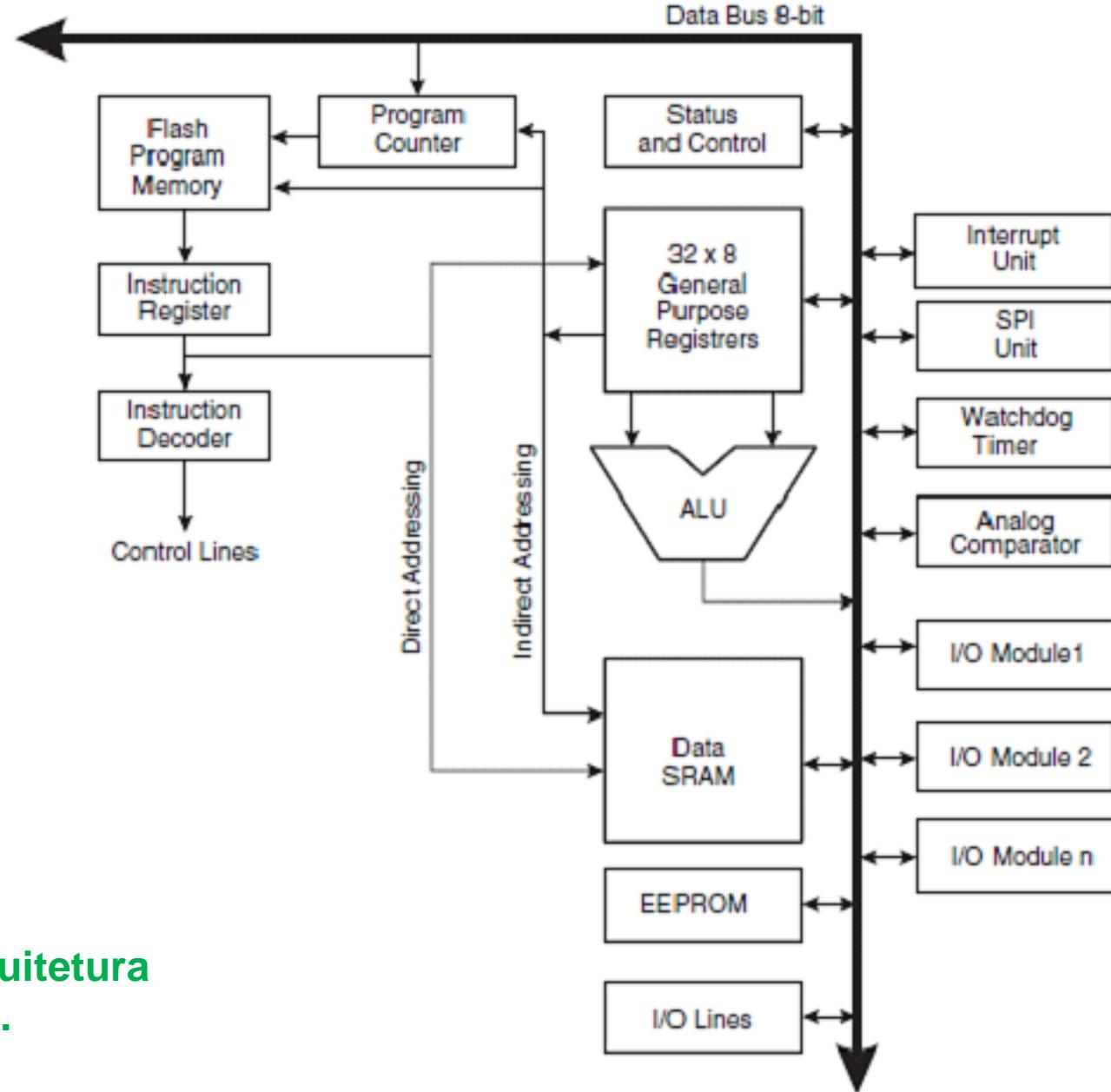


Diagrama em blocos



IMPORTANT!

Esta será nossa arquitetura
de referência.

Funcionamento básico da CPU do ATmega328:

1. Os dados são carregados em série via porta (via IDE no computador). Os mesmos são decodificados e, em seguida, as instruções são enviadas para o registro de instruções que decodificam as instruções no mesmo pulso do relógio.
2. No próximo pulso de *clock*, o próximo conjunto de instruções é carregado no registro de instruções.
3. O **registros de uso geral** são de 8 bits, mas também existem 3 registros de 16 bits.
 - a. **Registradores de 8 bits** são usados para armazenar dados para cálculos assim como os resultados.
 - b. **Registradores de 16 bits** são usados para armazenar dados do contador do Timer em 2 registros diferentes. Por exemplo: X-baixo e X-alto. São rápidos e usados para armazenar funções específicas de hardware.

- 4. EEPROM** - armazena dados permanentemente, mesmo se a energia for cortada. A programação dentro de uma EEPROM é lenta.
- 5. Interrupt Unit** - verifica se há interrupção pendente. Esta informação está presente no ISR (Interrupt Service Routine).
- 6. Serial Peripheral Interface (SPI)** - é um barramento de interface comumente usado para enviar dados entre microcontroladores e pequenos periféricos, como câmera, tela, cartões SD, etc. Ele usa linhas de relógio e dados separadas, além de uma linha de seleção para escolher o dispositivo com o qual deseja conversar.
- 7. Watchdog** - temporizador usado para detectar e se recuperar de mau funcionamento do microcontrolador.
- 8. Analog comparator** compara os valores de entrada nos pinos positivo e Negativo. Quando o valor do pino positivo é maior, a saída é definida.
- 9. Status and control** - é usado para controlar o fluxo de execução das instruções verificando outros blocos dentro da CPU em intervalos regulares.

10. ALU (Unidade Lógica e Aritmética - ULA): A ULA do AVR, de alto desempenho, opera em conexão direta com todos os 32 registros de trabalho de uso geral.

Dentro de um único ciclo de *clock*, são executadas operações aritméticas com registros de uso geral.

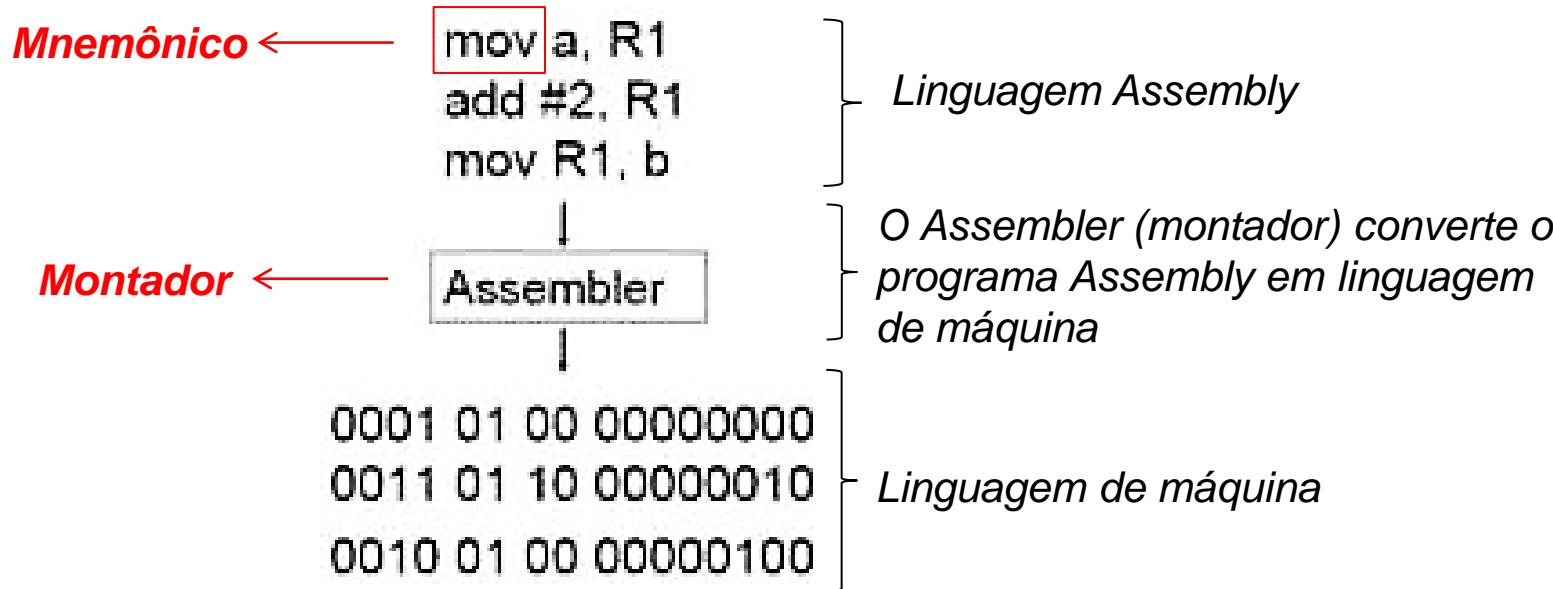
As operações da ULA são divididas em 3 categorias principais - aritmética, lógica e função de bit.

11. Pinos de E/S: As entradas e saídas digitais (E/S digital) permitem conectar os sensores, atuadores e outros CIs. Aprender a usá-los permitirá que você use o microcontrolador para fazer algumas coisas realmente úteis, como ler entradas de chave, indicadores de iluminação e controlar saídas de relé.

Linguagens de Programação

Assembly

Todo microcontrolador possui um conjunto próprio de instruções representadas por mnemônicos (assembly) que após o desenvolvimento do programa são convertidos nos zeros e uns lógicos interpretáveis pelo microprocessador.



Linguagens de Programação

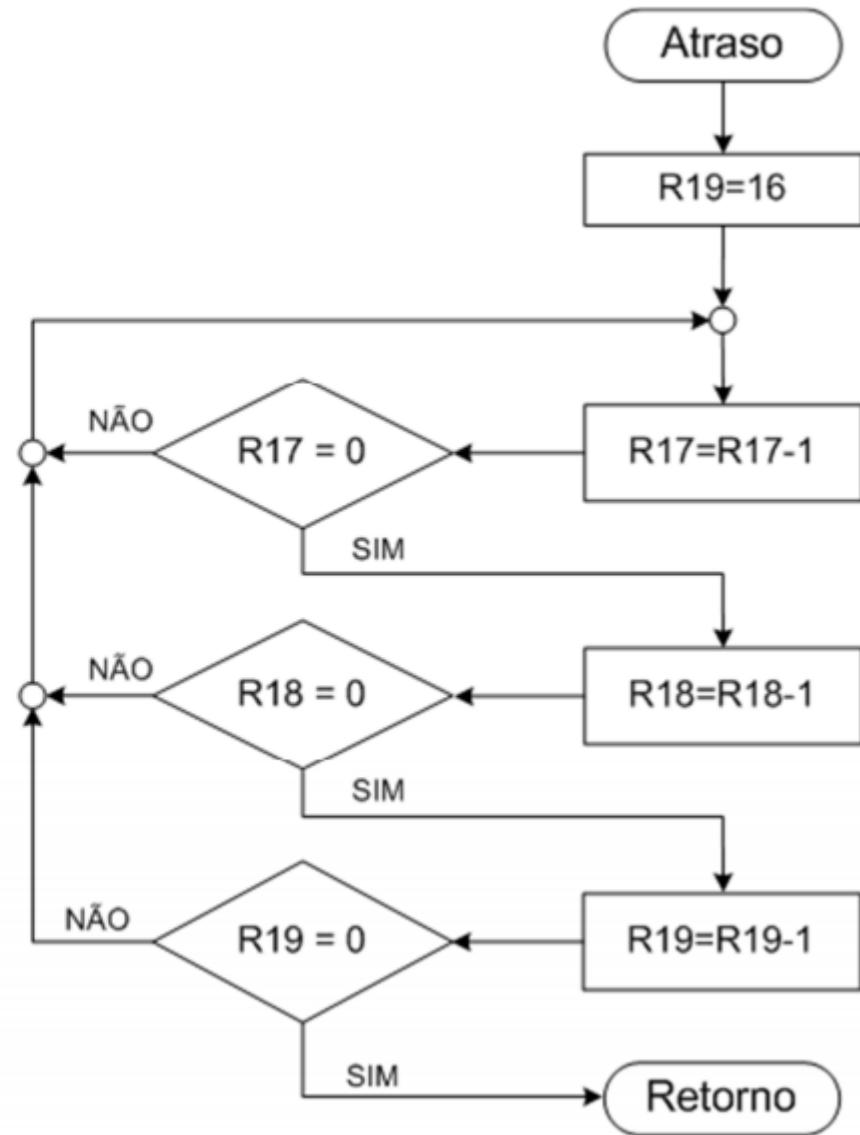
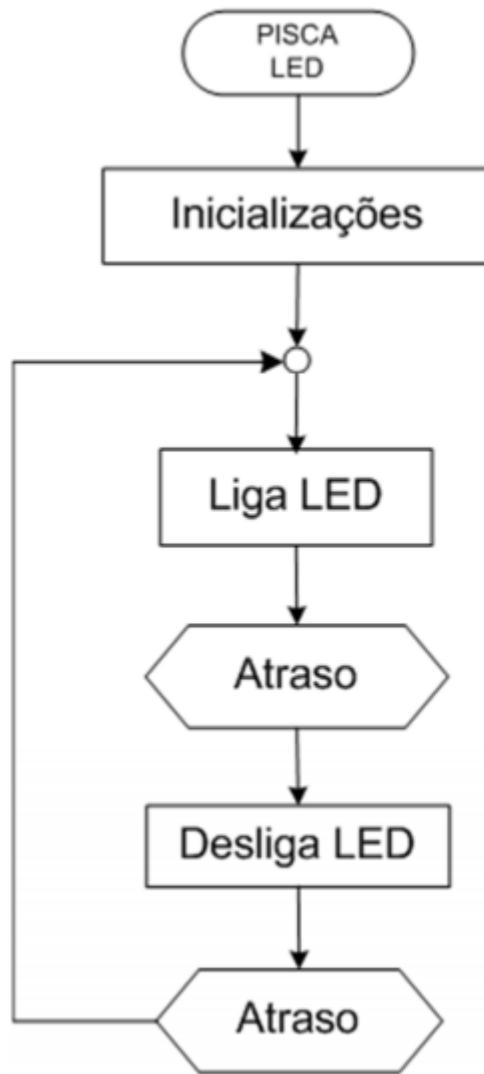
Assembly

O assembly é uma linguagem de baixo nível e permite obter o máximo desempenho de um microcontrolador, gerando o menor número de bytes de programa combinados a uma maior velocidade de processamento.

Todavia, o assembly só será eficiente se o programa estiver bem estruturado e empregar algoritmos adequados.
Programar em assembly exige muito esforço de programação.

Assembly é a linguagem da CPU do microcontrolador!

Exemplo de programação em Assembly



```

.equ LED      = PB5          //LED é o substituto de PB5 na programação
.ORG 0x000          //endereço de inicio de escrita do código

INICIO:
    LDI R16,0xFF        //carrega R16 com o valor 0xFF
    OUT DDRB,R16        //configura todos os pinos do PORTB como saída

PRINCIPAL:
    SBI PORTB, LED      //coloca o pino PB5 em 5V
    RCALL ATRASO        //chama a sub-rotina de atraso
    CBI PORTB, LED      //coloca o pino PB5 em 0V
    RCALL ATRASO        //chama a sub-rotina de atraso
    RJMP PRINCIPAL     //volta para PRINCIPAL

ATRASO:                  //atraso de aprox. 200ms (16 MHz)
    LDI R19,16

volta:
    DEC R17            //decrementa R17, começa com 0x00
    BRNE volta          //enquanto R17 > 0 fica decrementando R17
    DEC R18            //decrementa R18, começa com 0x00
    BRNE volta          //enquanto R18 > 0 volta decrementar R18
    DEC R19            //decrementa R19
    BRNE volta          //enquanto R19 > 0 vai para volta
    RET

```

**30 Bytes
15 instruções**

Linguagem C

Com a evolução tecnológica (compiladores), o Assembly foi quase que totalmente substituído pela linguagem C.

As vantagens do uso do C são numerosas:

- Redução do tempo de desenvolvimento.
- O reuso do código é facilitado.
- Facilidade de manutenção.
- Portabilidade.

- O problema de um código em C é que o mesmo pode consumir muita memória e reduzir a velocidade de processamento.
- Os compiladores tentam traduzir da melhor forma o código para o Assembly (antes de se tornarem código de máquina), mas esse processo não consegue o mesmo desempenho de um código escrito exclusivamente em Assembly.
- Como os compiladores C são eficientes para a arquitetura do AVR, a programação dos microcontroladores ATmega é feita em C.

Só existe a necessidade de se programar puramente em Assembly em casos críticos.

```
#define F_CPU 16000000UL      //define a frequência do microcontrolador 16MHz
#include <avr/io.h>          //definições do componente especificado
#include <util/delay.h>        //biblioteca para o uso das rotinas de delay

//Definições de macros
#define set_bit(Y,bit_x) (Y|=(1<<bit_x))    //ativa o bit x da variável Y
#define clr_bit(Y,bit_x) (Y&=~(1<<bit_x))   //limpa o bit x da variável Y
#define tst_bit(Y,bit_x) (Y&(1<<bit_x))     //testa o bit x da variável Y
#define cpl_bit(Y,bit_x) (Y^=(1<<bit_x))    //troca o estado do bit x da variável Y

#define LED PB5                  //LED é o substituto de PB5 na programação

//-----
int main( )
{
    DDRB = 0xFF;                //configura todos os pinos do PORTB como saídas

    while(1)                   //laço infinito
    {
        set_bit(PORTB,LED);    //liga LED
        _delay_ms(200);        //atraso de 200 ms
        clr_bit(PORTB,LED);   //desliga LED
        _delay_ms(200);        //atraso de 200 ms
    }
}
//-----
```

7,2 vezes maior



216 Bytes

```
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
    // initialize the digital pin as an output.
    pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
    digitalWrite(led, HIGH);      // turn the LED on (HIGH is the voltage level)
    delay(200);                  // wait for a second
    digitalWrite(led, LOW);       // turn the LED off by making the voltage LOW
    delay(200);                  // wait for a second
}
```



**30 bytes Assembly
216 bytes C
1084 bytes IDE Arduino**

Como programar em baixo nível utilizando linguagem C?



Quando desejamos programar um microcontrolador, é fundamental o conhecimento de suas características (arquitetura).

Um dos pontos iniciais de grande importância é a manipulação de registradores.

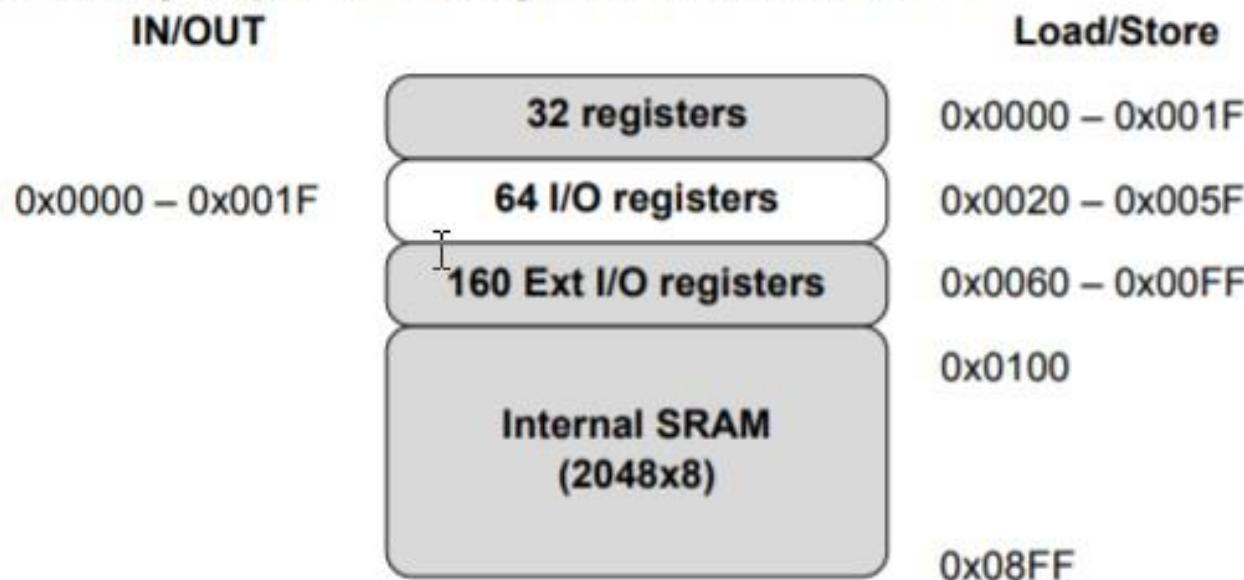
Um registrador é um tipo de memória de pequena capacidade porém muito rápida, contida na CPU, utilizado no armazenamento temporário de dados durante o processamento.

Os registradores estão no topo da hierarquia de memória, sendo desta forma o meio mais rápido e de maior custo para armazenar um dado.

Os mesmos podem ser divididos em registradores de propósito geral ou de função específica (SFR).

Os registradores (SFR – Special Function Registers) recebem nomes específicos e têm função bem definida: guardar a configuração e o estado de funcionamento atual do microcontrolador.

Data Memory Map with 2048 Byte Internal Data SRAM



Normalmente, cada bit do registrador tem uma função específica. Assim, temos um registrador para definir se as portas são de entrada ou de saída, ativar e desativar interrupções, apresentar o estado da CPU, etc.

PORTB – The Port B Data Register

Os registradores de I/O são o painel de controle dos Microcontroladores pois todas as configurações de trabalho, incluindo acesso às entradas e saídas, se encontram nessa parte da memória.

End.	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x23	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
0x24	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
0x25	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
0x26	PINC	-	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
0x27	DDRC	-	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
0x28	PORTC	-	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
0x29	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
0x2A	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
0x2B	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
-	-	-	-	-	-	-	-	-	-
0xC4	UBRR0L	Registrador da taxa de transmissão da USART, byte menor							
0xC5	UBRR0H	-	-	-	-	Registrador da taxa de transmissão da USART, byte maior			
0xC6	UDR0	Registrador I/O de dados da USART							

Total de 87 Registradores

Microcontrolador ATMEGA328p

GPIO

Os registradores e referências de bits para cada pino de porta recebe o nome de **Pxn**, a letra "x" representa o nome da porta, como PB (PORTB) ou PD (PORTD), e a letra "n" representa o número do bit, como PB[0-7] (PORTB[0-7]) ou PB[0-7] (PORTB[0-7]).

Vale ressaltar que as portas analógicas são chamadas de PC (PORTC) que passam por um conversor A/D.

Esses nomes estão indicados na pinagem do microcontrolador e são usados na programação.

PC6	1	28	PC5
PD0	2	27	PC4
PD1	3	26	PC3
PD2	4	25	PC2
PD3	5	24	PC1
PD4	6	23	PC0
VCC	7	22	GND
GND	8	21	AREF
PB6	9	20	AVCC
PB7	10	19	PB5
PD5	11	18	PB4
PD6	12	17	PB3
PD7	13	16	PB2
PB0	14	15	PB1

Para cada porta digital **três endereços de memória I/O** são alocados, sendo eles:

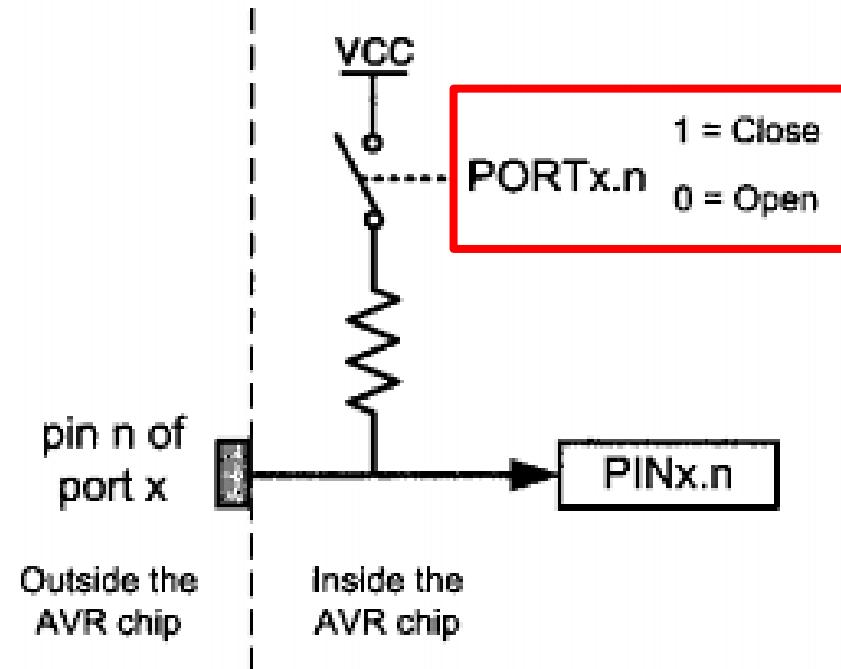
PORT_x (leitura/escrita): O registro de dados, responsável em determinar o estado do pino (HIGH/LOW).

DDR_x (leitura/escrita): A direção dos dados, responsável pela configuração de entrada ou saída do pino (OUTPUT/INPUT).

PIN_x (leitura): A entrada da porta, responsável em armazenar o estado do pino, onde a execução de uma função de escrita no PIN_x, resultará na alteração no valor do PORT_x.

Três bits de registros são setados, sendo eles:

PORTxn: Se PORTxn for setado 1/verdadeiro e o pino estiver configurado como entrada, o resistor de ***pull-up*** interno será ativado. Para desabilitar o resistor pull-up, o PORTxn deve ser escrito como 0/falso ou o pino deve ser configurado como saída. Por padrão todas as portas são inicializadas como tri-state HI-Z.



DDxn: O bit DDxn no endereço DDRx seleciona a direção desse pino. Se DDxn for escrito como 1/verdadeiro, Pxn será configurado como **saída**. Se DDxn for escrito como 0/falso, Pxn será configurado como **entrada**.

PINxn: Os bits são acessados pelo endereço PINx.

- **PORTx**: registrador de dados, usado para escrever nos pinos do PORTx.
- **DDRx**: registrador de direção, usado para definir se os pinos do PORTx são entrada ou saída.
- **PINx**: registrador de entrada, usado para ler o conteúdo dos pinos do PORTx.

Bits de controle dos pinos dos PORTs.

DDXn*	PORTXn	I/O	Pull-up	Comentário
0	0	Entrada	Não	Alta impedância (Hi-Z).
0	1	Entrada	Sim	PXn irá fornecer corrente se externamente for colocado em nível lógico 0.
1	0	Saída	Não	Saída em zero (drena corrente).
1	1	Saída	Não	Saída em nível alto (fornecce corrente).

Na IDE do Arduino

```
pinMode(3, OUTPUT);  
pinMode(5, OUTPUT);  
pinMode(7, OUTPUT);
```

Ou

```
pinMode(PIN_D3, OUTPUT);  
pinMode(PIN_D5, OUTPUT);  
pinMode(PIN_D7, OUTPUT);
```

Manipulando Registradores

DDRD = 0b10101000;

Ou

DDRD = 0xA8;

Ou

DDRD |= 1<<PD7 | 1<<PD5 | 1<<PD3;

DDRD = B1111110;

// configura portas 1 ate 7 como saídas e a porta 0 como entrada

DDRD = DDRD | B11111100;

// esta é uma forma mais segura // de configurar os pinos 2 até 7 como saída

// sem mudar as configurações dos pinos 0 e 1 que são da serial

Na IDE do Arduino

```
pinMode(0, INPUT);  
pinMode(1, INPUT);  
digitalWrite(0, HIGH);  
digitalWrite(1, HIGH);
```

Ou

```
pinMode(PIN_D0, INPUT);  
pinMode(PIN_D1, INPUT);  
digitalWrite(PIN_D0, HIGH);  
digitalWrite(PIN_D1, HIGH);
```

Manipulando Registradores

```
DDRD = 0; // all PORTD pins inputs  
PORTD = 0b00000011;  
Ou  
PORTD = 0x03;
```

Ou melhor ainda

```
DDRD &= ~(1<<PD1 | 1<<PD0);  
PORTD |= (1<<PD1 | 1<<PD0);
```

* Para alterar o conteúdo dos registradores DDR_x e $PORT_x$ é necessário realizar uma operação de escrita **de um byte completo**, mesmo que se deseje alterar apenas um dos bits.

- | OU lógico bit a bit (usado para ativar bits , colocar em 1)
- & E lógico bit a bit (usado para limpar bits, colocar em 0)
- ^ OU EXCLUSIVO bit a bit (usado para trocar o estado dos bits)
- complemento de 1 (1 vira 0, 0 vira 1)

Nr >> x O número é deslocado x bits para a direita

Nr << x O número é deslocado x bits para a esquerda

Lógica “e” bit a bit (&)

x: 10001101
y: 01010111
x & y: 00000101

Lógica “ou” bit a bit (|)

x: 10001101
y: 01010111
x | y: 11011111

Operação “não” (~)

$\sim 0 = 1$
 $\sim 1 = 0$

Operação “ou-exclusivo” (^)

$$0 \wedge 0 = 0$$

$$0 \wedge 1 = 1$$

$$1 \wedge 0 = 1$$

$$1 \wedge 1 = 0$$

Deslocamento para a esquerda

$$y = 1010$$

$$x = y << 1$$

Resulta em: $x = 0100$

Deslocamento para a direita

$$y = 1010$$

$$x = y >> 1$$

Resulta em: $x = 0101$

$$1 << 0 = 1$$

$$1 << 1 = 2$$

$$1 << 2 = 4$$

$$1 << 3 = 8$$

...

$$1 << 8 = 256$$

$$1 << 9 = 512$$

$$1 << 10 = 1024$$

- Ativação de bit, colocar em 1:

```
#define set_bit(Y,bit_x) (Y|=(1<<bit_x))
```

onde Y |= (1<<bit_x) ou Y = Y | (1<<bit_x)

Exemplo:

set_bit(PORTD,5)

PORTD = PORTD | (1<<5) ,

0bxxxxxxxx
| 0b00100000

PORTD = 0bxx1xxxxx

(PORTD, x pode ser 0 ou 1)
(1<<5 é a máscara)
(o bit 5 com certeza será 1)

- Limpeza de bit, colocar em 0:

```
#define clr_bit(Y,bit_x) (Y&=~(1<<bit_x))  
onde Y &= ~ (1<<bit_x) ou Y = Y & (~ (1<<bit_x))
```

Exemplo:

clr_bit(PORTB,2)

PORTB = PORTB & (~ (1<<2)) ,

0bXXXXXXXXXX & 0b11111011	(PORTB, x pode ser 0 ou 1) (~(1<<2) é a máscara)
PORTB = 0bXXX0xx0xx	(o bit 2 com certeza será 0)

- Troca o estado lógico de um bit, 0 para 1 ou 1 para 0:

```
#define cpl_bit(Y,bit_x) (Y^=(1<<bit_x))
```

onde $Y \wedge = (1 \ll bit_x)$ ou $Y = Y \wedge (1 \ll bit_x)$

Exemplo:

cpl_bit(PORTC,3)

PORTC = PORTC $\wedge (1 \ll 3)$,

0bxxxx1xxx
^ 0b00001000
PORTC = 0bxxxx0xxx

(PORTC, x pode ser 0 ou 1)
($1 \ll 3$ é a máscara)
(o bit 3 será 0 se o bit a ser complementado for 1 e 1 se ele for 0)

- Leitura de um bit:

```
#define tst_bit(Y,bit_x) (Y&(1<<bit_x))
```

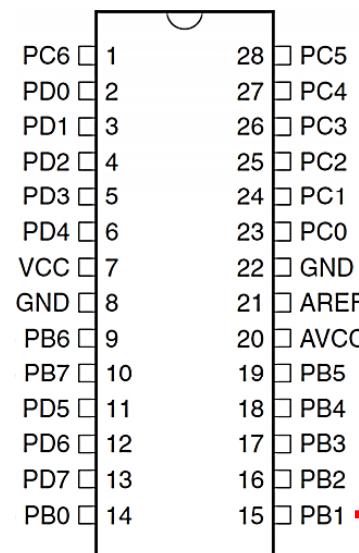
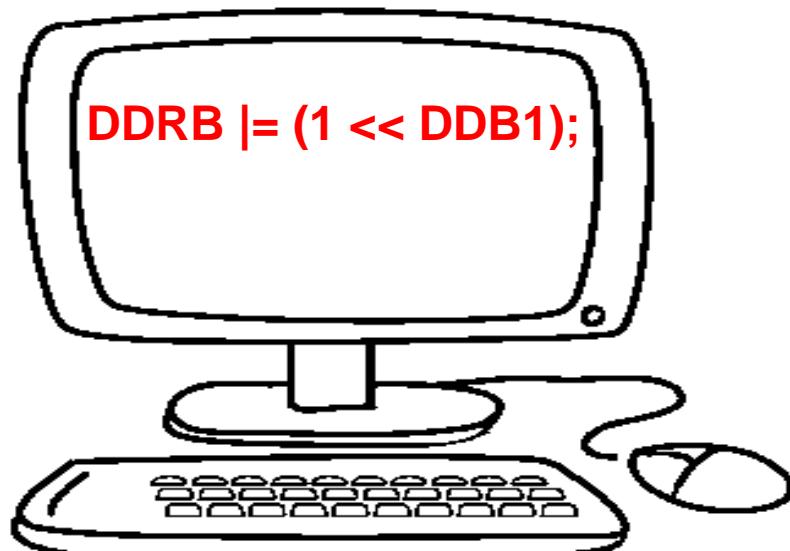
Exemplo:

tst_bit(PIND,4)

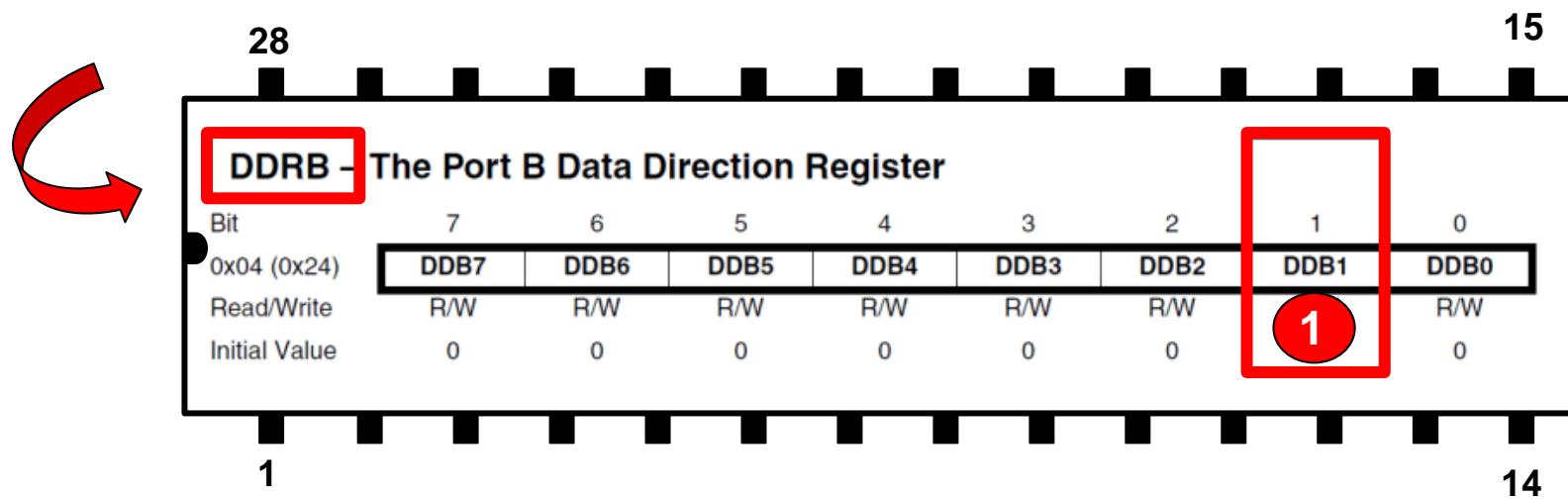
PIND & (1<<4) ,

resultado	= 0b000T0000	0bxxxxTxxxxx & 0b00010000 (PIND, x pode ser 0 ou 1) (1<<4 é a máscara) (o bit 4 terá o valor T, que será 0 ou 1)
-----------	--------------	--

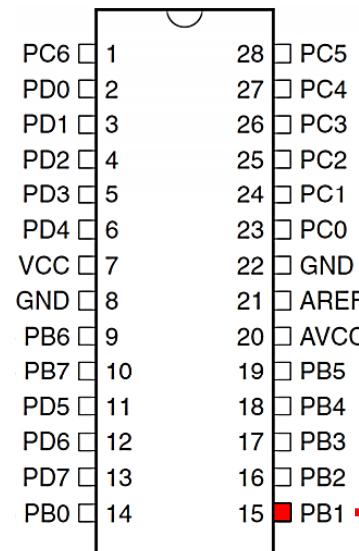
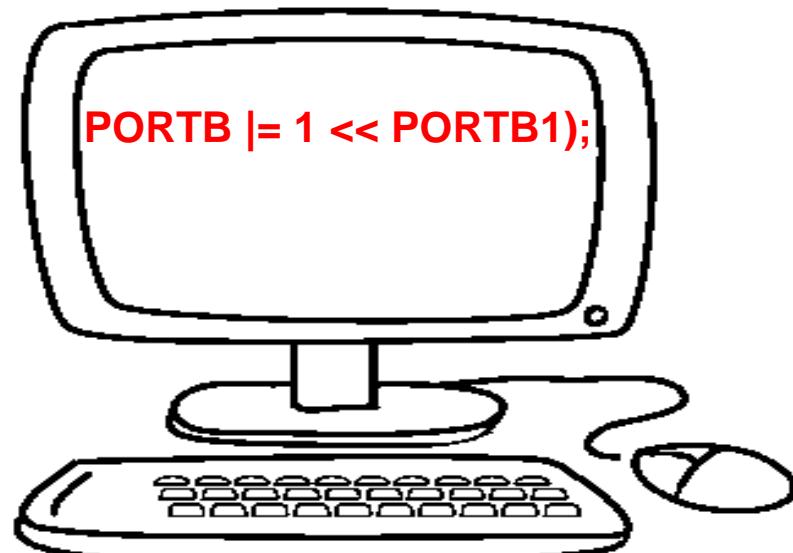
Definindo um pino com saída



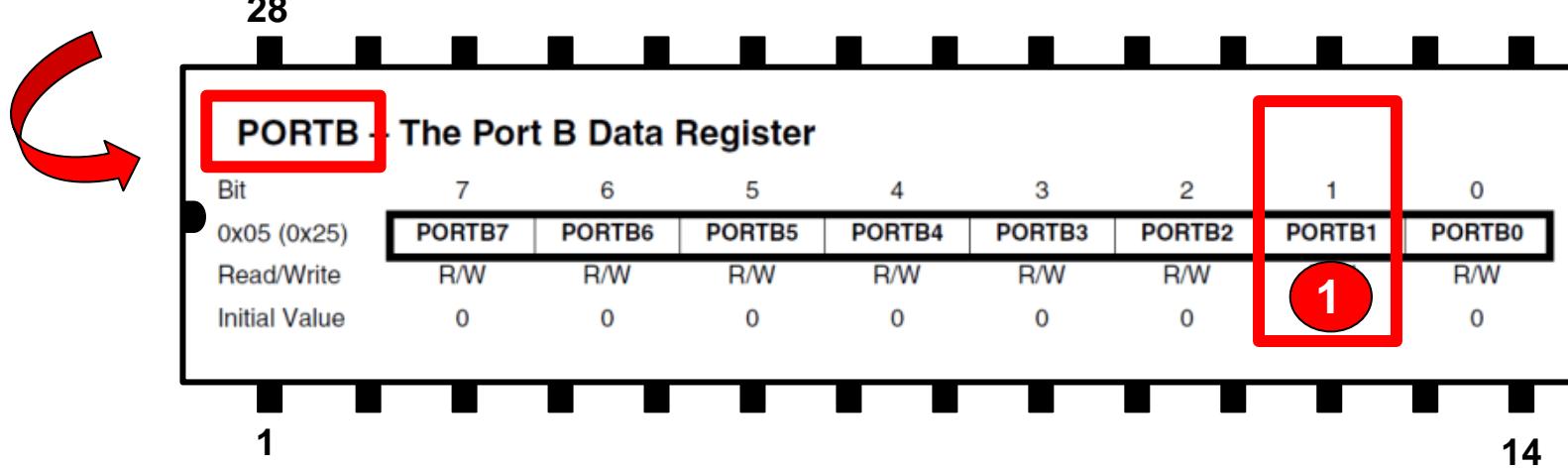
PB1 definido como saída



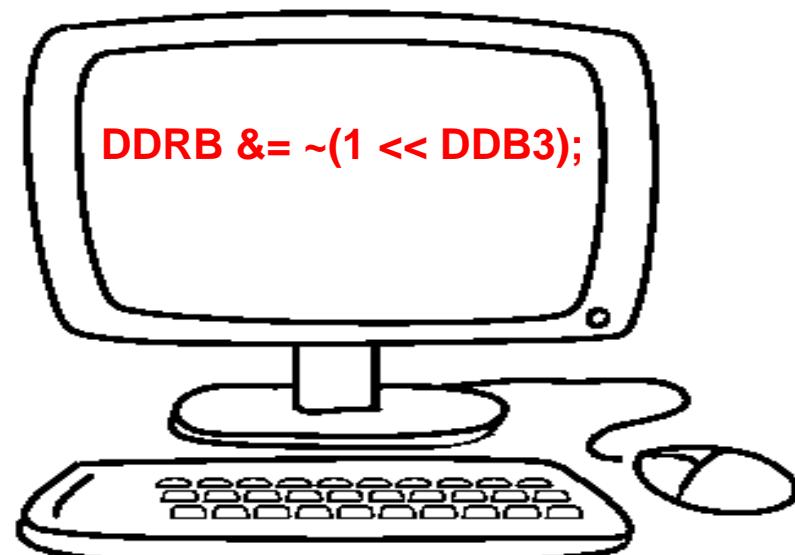
Ativando um pino de saída



O estado lógico de PB1 é definido como alto (HIGH)

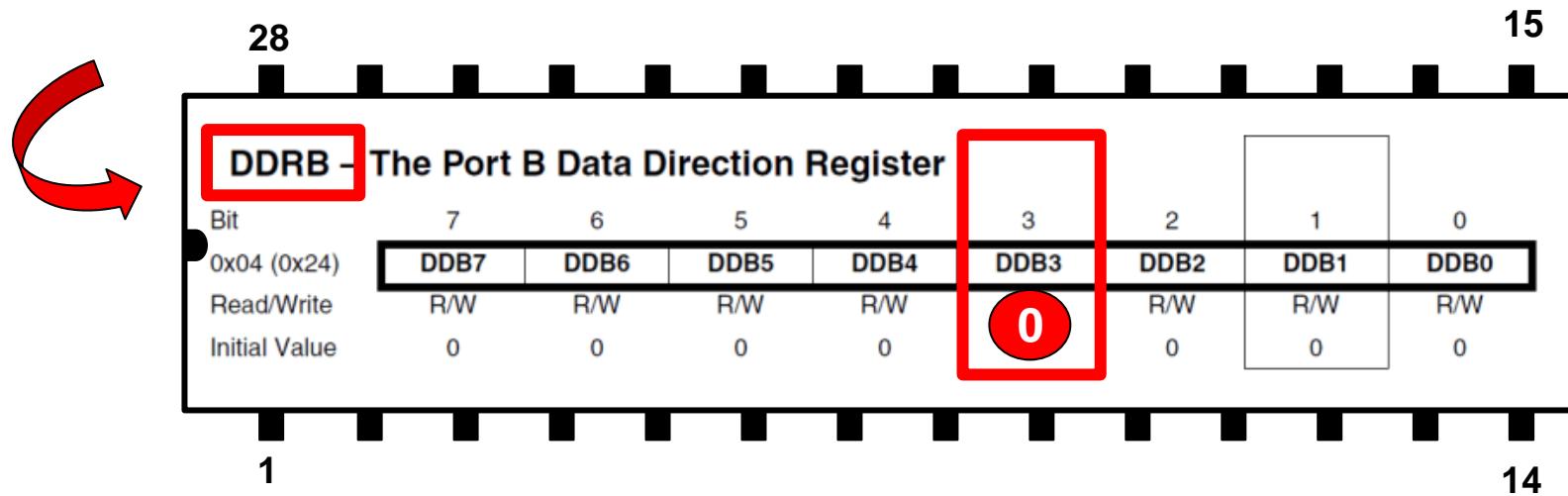


Definindo um pino com entrada

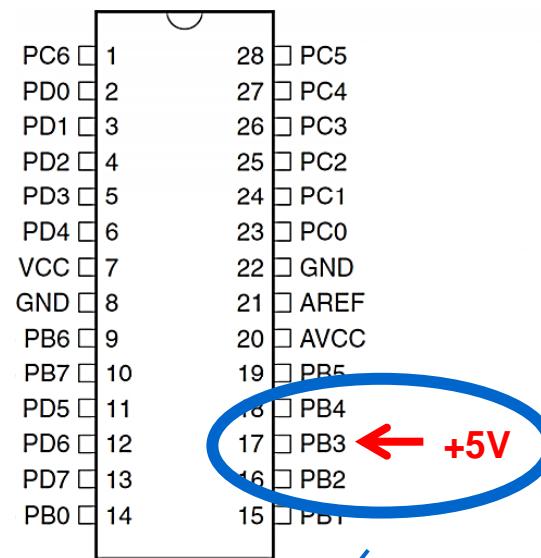
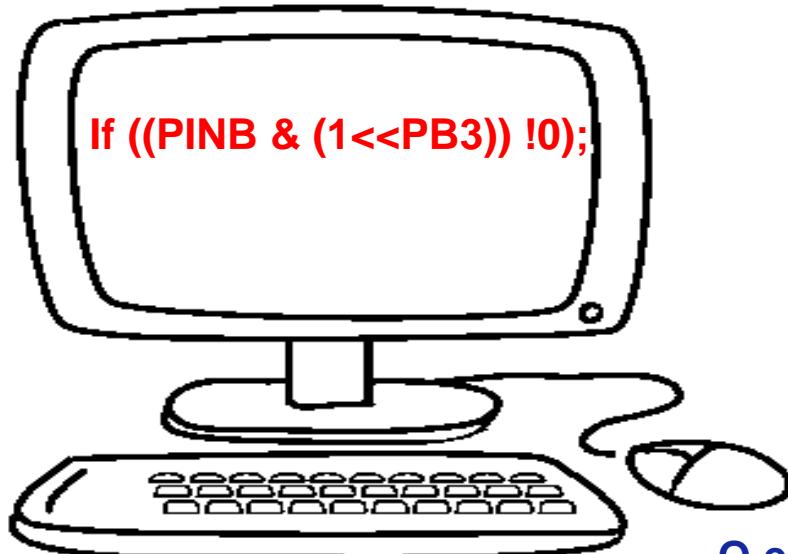


PC6	1	28	PC5
PD0	2	27	PC4
PD1	3	26	PC3
PD2	4	25	PC2
PD3	5	24	PC1
PD4	6	23	PC0
VCC	7	22	GND
GND	8	21	AREF
PB6	9	20	AVCC
PB7	10	19	PB5
PD5	11	18	PB4
PD6	12	17	PB3
PD7	13	16	PB2
PB0	14	15	PB1

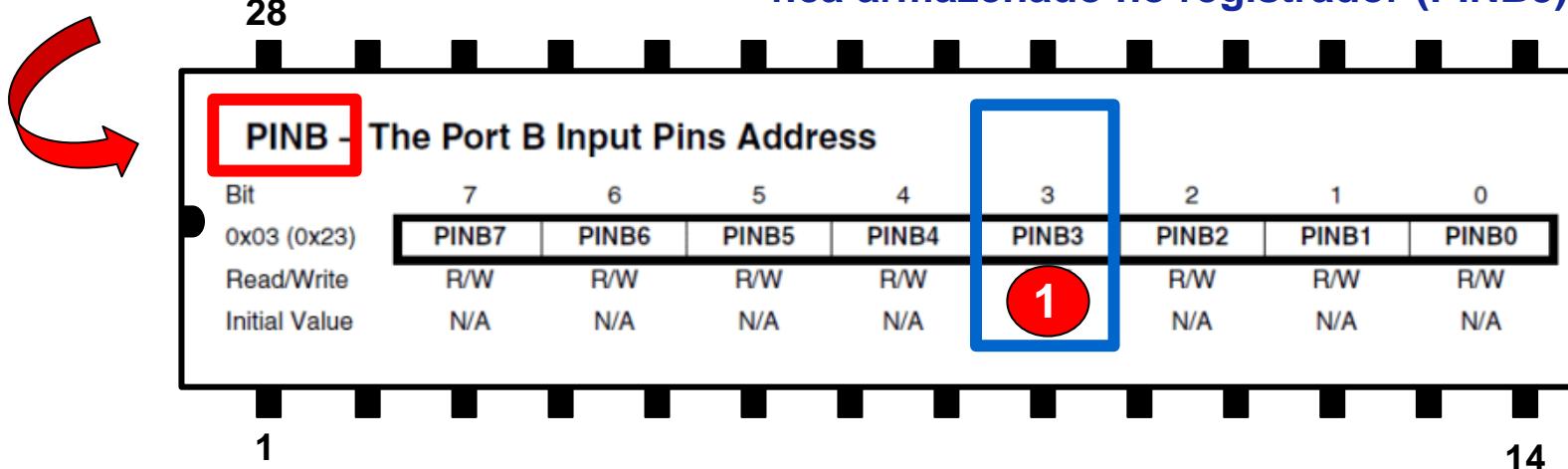
PB3 definido como entrada



Verificando o estado de um pino de entrada



O estado lógico lido em PB3
fica armazenado no registrador (PINB3)



Estudo de caso:

```
#include <avr/io.h>           // Arq. header p/ familia AVR

#define RELE_1 0 //Pino conectado ao relê 1
#define RELE_2 1 //Pino conectado ao relê 2

unsigned char i; // ou, ainda: uint8_t i;

int main(void)
{
    // Define Pinos PB0, PB1 como saídas, PB2 a PB7 entradas
    DDRB |= (1<<DDB1) | (1<<DDB0);

    // Define estado dos Pinos PB0, PB1, PB6 e PB7 em nível alto
    // e pinos PB2, PB3, PB4 e PB5 em nível baixo
    PORTB |= (1<<PB7) | (1<<PB6) | (1<<PB1) | (1<<PB0);

    // Leitura dos Pinos da Porta B
    i = PINB;

    // Leva pino PB0 para nível lógico 0
    PORTB &= ~(1<<RELE_1);

    // Leva pino PB1 para nível lógico 1
    PORTB |= (1<<RELE_2);

    return 0;
}
```

Para “ligar” um bit específico:

PORTB = PORTB | (0b00000001); Liga o bit 0

ou

PORTB = PORTB | (1 << n); Liga o bit n onde n = 0 a 7

Para “desligar” um bit específico:

PORTB = PORTB & ~(0b00000001); Desliga o bit 0

ou

PORTB = PORTB & ~(1 << n); Desliga o bit n onde n = 0 a 7

Para inverter um bit:

PORTB = PORTB ^ (0b00000001); Inverte o bit 0

ou

PORTB = PORTB ^ (1 << n); Inverte o bit n onde n = 0 a 7

Testando se o bit está “ligado”:

```
if ( PINB & (0b00000001))
```

ou

```
if ( PINB & (1 << n)) onde n é bit em teste (0 - 7)
```

Testando se o bit está “desligado”:

```
if ( !(PINB & (0b00000001)) )
```

ou

```
if ( !(PINB & (1 << n)) ) onde n é bit em teste (0 – 7)
```

EXERCÍCIOS

Microcontrolador ATMEGA328p

INTERRUPÇÕES



Interrupção é o recurso que permite ao microcontrolador interromper a execução do programa principal para atender a uma solicitação “urgente” gerada por um dispositivo externo ou por um periférico.

Por exemplo, se houve uma mudança de estado em um pino, ou se um byte de dados chegou na porta serial, ou se um temporizador transbordou, ou se a conversão foi concluída no ADC, etc.; estes são todos os casos típicos ou interrupções comuns que podem ser ativadas no microcontrolador.

Considere que o microcontrolador deve acender um LED quando um botão é pressionado.

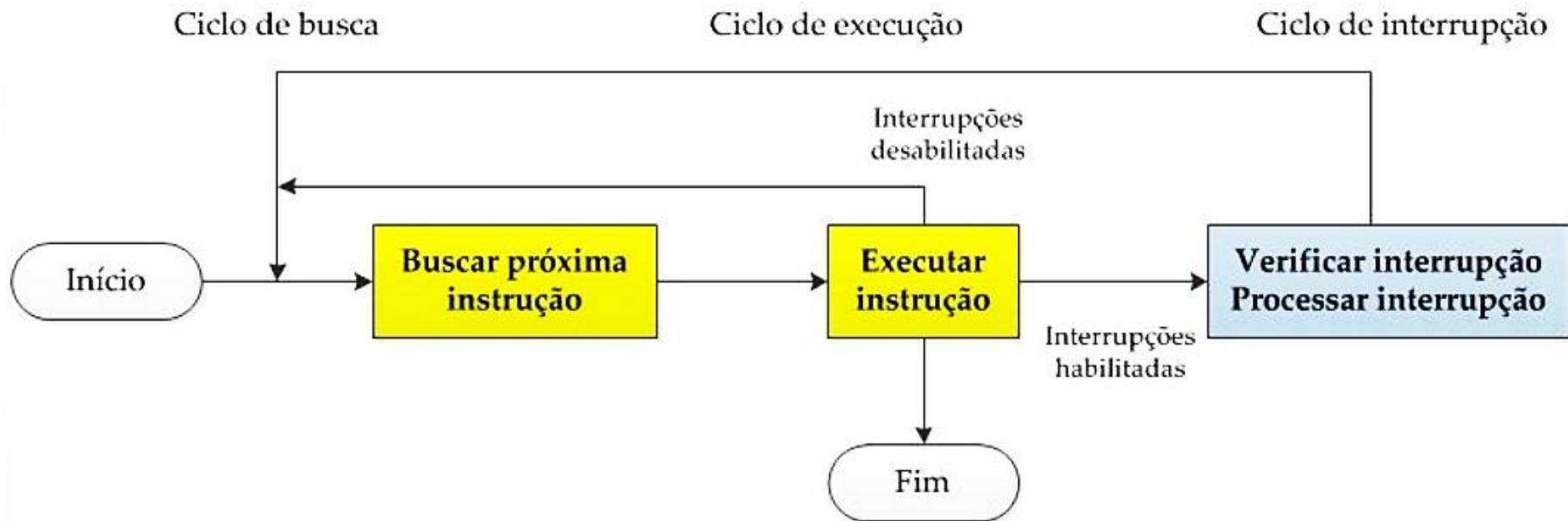
- O instante em que o botão é pressionado é absolutamente imprevisível da perspectiva do microprocessador / programa em execução.
- Há duas abordagens para se tratar um evento de natureza assíncrona:
 - ✓ Varredura
 - ✓ Interrupção

No processo de varredura, deve-se aguardar a execução da rotina ou instrução que verifica o estado da chave.

É um recurso de implementação simples, porém desperdiça ciclos de execução do processador.

Caso o recurso de interrupção esteja disponível, passa a ser a melhor alternativa.

- Neste caso, o processador suspende o programa em execução e desvia para a rotina de serviço de interrupção (*Interrupt Service Routine*, ISR).



- Cada periférico no microcontrolador pode gerar uma ou mais interrupções internas.
- Há também interrupções associadas a eventos externos.
- **Interrupções mascaráveis**: cada interrupção possui um bit de habilitação que precisa ser ativado juntamente com o Global Interrupt Enable (bit no registrador de status para que interrupções ocorram).
- **Interrupções fixas**: cada interrupção é mapeada para um endereço fixo de memória.
- As interrupções são desabilitadas automaticamente quando uma ISR está sendo executada.

Entretanto, é possível habilitá-las manualmente dentro da ISR e assim permitir que interrupções sejam atendidas durante a execução da ISR.

Pri.	Address	Interrupt Source	ISR C Function Name	Description
1	0x0000	RESET		System reset (power-on)
2	0x0002	INT0	INT0_vect	External Interrupt Request 0
3	0x0004	INT1	INT1_vect	External Interrupt Request 1
4	0x0006	PCINT0	PCINT0_vect	Pin Change Interrupt Request 0
5	0x0008	PCINT1	PCINT1_vect	Pin Change Interrupt Request 1
6	0x000A	PCINT2	PCINT2_vect	Pin Change Interrupt Request 2
7	0x000C	WDT	WDT_vect	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	TIMER2_COMPA_vect	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	TIMER2_COMPB_vect	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	TIMER2_OVF_vect	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	TIMER1_CAPT_vect	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	TIMER1_COMPA_vect	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	TIMER1_COMPB_vect	Timer/Counter1 Compare Match B
14	0x001A	TIMER1 OVF	TIMER1_OVF_vect	Timer/Counter1 Overflow

Pri.	Address	Interrupt Source	ISR C Function Name	Description
15	0x001C	TIMER0 COMPA	TIMER0_COMPA_vect	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	TIMER0_COMPB_vect	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	TIMER0_OVF_vect	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI_STC_vect	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART_RX_vect	USART Receive Complete
20	0x0026	USART, UDRE	USART_UDRE_vect	USART Data Register Empty
21	0x0028	USART, TX	USART_TX_vect	USART Transmit Complete
22	0x002A	ADC	ADC_vect	ADC Conversion Complete
23	0x002C	EE READY	EE_READY_vect	EEPROM Ready
24	0x002E	ANALOG COMP	ANALOG_COMP_vect	Analog Comparator
25	0x0030	TWI	TWI_vect	2-wire Serial Interface
26	0x0032	SPM READY	SPM_READY_vect	Store Program Memory Ready

A rotina de interrupção em C deve ser definida na forma:

```
ISR (INT0_vect)
{
}
```

INTERRUPÇÕES EXTERNAS

São interrupções geradas por dispositivos externos ao microcontrolador. Podem ser de dois tipos:

- **Interrupções geradas nos pinos INT0 e INT1**: permitem um número maior de configurações e têm maior prioridade. Podem ser ativadas nas bordas de subida, descida, em ambas ou por nível lógico baixo do sinal de interrupção.

São configuradas através dos registradores EICRA e EIMSK. Quando disparadas, ativam *flags* no registrador EIFR.

- **Pin Change Interrupts**: têm menor prioridade que as anteriores e podem ser ativadas quando há uma mudança de nível em um dos 23 pinos PCINT. São configuradas por meio dos registradores PCMSK0, PCMSK1, PCMSK2 e PCICR.

Flags no registrador PCIFR indicam quando essas interrupções são ativadas.

EICRA – External Interrupt Control Register A

Define como INT0 e INT1 são ativadas: por borda ou por nível.

Bit	7	6	5	4	3	2	1	0	
(0x69)	-	-	-	-	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 13-1. Interrupt 1 Sense Control

ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request.
0	1	Any logical change on INT1 generates an interrupt request.
1	0	The falling edge of INT1 generates an interrupt request.
1	1	The rising edge of INT1 generates an interrupt request.

Table 13-2. Interrupt 0 Sense Control

ISC01	ISC00	Description
0	0	The low level of INT0 generates an interrupt request.
0	1	Any logical change on INT0 generates an interrupt request.
1	0	The falling edge of INT0 generates an interrupt request.
1	1	The rising edge of INT0 generates an interrupt request.

EIMSK – External Interrupt Mask Register

- **Habilita INT0 e INT1.**

EIMSK – External Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0
0x1D (0x3D)	-	-	-	-	-	-	INT1	INT0
Read/Write	R	R	R	R	R	R	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

- **INT0 - External Interrupt Request 0 Enable:** quando em nível alto, habilita a geração de interrupções a partir do pino INT0.
(É necessário que a interrupção global também esteja ativa).
- **INT1 - External Interrupt Request 1 Enable:** quando em nível alto, habilita a geração de interrupções a partir do pino INT1.
(É necessário que a interrupção global também esteja ativa).

EIFR – External Interrupt Flag Register

Sinaliza quando INT0 e INT1 foram disparadas.

EIFR – External Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x1C (0x3C)	-	-	-	-	-	-	INTF1	INTF0	EIFR
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **INTF0 - External Interrupt Flag 0:** quando ocorre a condição de ativação de INT0, de acordo com as configurações em EICRA, esse bit assume nível alto.
- **INTF1 - External Interrupt Flag 1:** quando ocorre a condição de ativação de INT1, de acordo com as configurações em EICRA, esse bit assume nível alto.

PCICR – Pin Change Interrupt Control Register

Habilita as *interrupções PCI0, 1 e 2.*

Bit	7	6	5	4	3	2	1	0	
(0x68)	-	-	-	-	-	PCIE2	PCIE1	PCIE0	PCICR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **PCIE0 – Pin Change Interrupt Enable 0:** habilita interrupções geradas pelos pinos PCINT 0 a 7.
- **PCIE1 – Pin Change Interrupt Enable 1:** habilita interrupções geradas pelos pinos PCINT 8 a 14.
- **PCIE0 – Pin Change Interrupt Enable 2:** habilita interrupções geradas pelos pinos PCINT 16 a 23.

PCMSK0 – Pin Change Mask Register 0

Bit	7	6	5	4	3	2	1	0	
(0x6B)	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	PCMSK0
Read/Write	R/W								
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7:0 – PCINT[7:0]: Pin Change Enable Mask 7...0

Each PCINT[7:0] bit selects whether pin change interrupt is enabled on the corresponding I/O pin. If PCINT[7:0] is set and the PCIE0 bit in PCICR is set, pin change interrupt is enabled on the corresponding I/O pin. If PCINT[7:0] is cleared, pin change interrupt on the corresponding I/O pin is disabled.

PCMSK1 – Pin Change Mask Register 1

Bit	7	6	5	4	3	2	1	0	
(0x6C)	-	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8	PCMSK1
Read/Write	R	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 – Reserved

This bit is an unused bit in the ATmega48A/PA/88A/PA/168A/PA/328/P, and will always read as zero.

- Bit 6:0 – PCINT[14:8]: Pin Change Enable Mask 14...8

Each PCINT[14:8]-bit selects whether pin change interrupt is enabled on the corresponding I/O pin. If PCINT[14:8] is set and the PCIE1 bit in PCICR is set, pin change interrupt is enabled on the corresponding I/O pin. If PCINT[14:8] is cleared, pin change interrupt on the corresponding I/O pin is disabled.

PCMSK2 – Pin Change Mask Register 2

Bit	7	6	5	4	3	2	1	0	PCMSK2
(0x6D)	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16	
Read/Write	R/W								
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7:0 – PCINT[23:16]: Pin Change Enable Mask 23...16

Each PCINT[23:16]-bit selects whether pin change interrupt is enabled on the corresponding I/O pin. If PCINT[23:16] is set and the PCIE2 bit in PCICR is set, pin change interrupt is enabled on the corresponding I/O pin. If PCINT[23:16] is cleared, pin change interrupt on the corresponding I/O pin is disabled.

PCIFR – Pin Change Interrupt Flag Register

Sinaliza quando PCI0, PCI1 e PCI2 foram disparadas.

Bit	7	6	5	4	3	2	1	0	PCIFR
0x1B (0x3B)	-	-	-	-	-	PCIF2	PCIF1	PCIF0	
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **PCIF0 – Pin Change Interrupt Flag 0:** quando ocorre uma mudança de nível lógico em qualquer um dos pinos PCINT[7:0] dispara uma interrupção, PCIF0=1.
- **PCIF1 – Pin Change Interrupt Flag 1:** quando ocorre uma mudança de nível lógico em qualquer um dos pinos PCINT[14:8] dispara uma interrupção, PCIF1=1.
- **PCIF2 – Pin Change Interrupt Flag 2:** quando ocorre uma mudança de nível lógico em qualquer um dos pinos PCINT[23:16] dispara uma interrupção, PCIF2=1

Registradores para interrupção externa

Para configurarmos um interrupções externas nos projetos com Arduino, podemos utilizar basicamente 2 registradores:

EIMSK - Habilita INT0 e INT1 do Arduino UNO

EICRA - Define como INT0 e INT1 são ativadas (LOW, RISING, FALLING ou CHANGE)

EIMSK (External Interrupt Mask Register)

EIMSK |= (1<<INT0); // habilita a interrupção externa INT0 no pino 2 Arduino

EIMSK |= (1<<INT1); // habilita a interrupção externa INT0 no pino 3 Arduino

Atenção: Necessário que a interrupção global também esteja ativa

Funções para habilitar e desabilitar interrupções globalmente

sei(); // função que habilita as interrupções externas globalmente

cli(); // função que desabilita as interrupções externas globalmente

EICRA (External Interrupt Control Register A)

EICRA INT0 (pino 2)

```
EICRA |= (0<<ISC01) | (1<<ISC00); // configura interrupção externa int 0 - CHANGE  
EICRA |= (1<<ISC01) | (0<<ISC00); // configura interrupção externa int 0 - FALLING  
EICRA |= (1<<ISC01) | (1<<ISC00); // configura interrupção externa int 0 - RISING
```

EICRA INT1 (pino 3)

```
EICRA |= (0<<ISC11) | (1<<ISC10); // configura interrupção externa int 1 - CHANGE  
EICRA |= (1<<ISC11) | (0<<ISC10); // configura interrupção externa int 1 - FALLING  
EICRA |= (1<<ISC11) | (1<<ISC10); // configura interrupção externa int 1 - RISING
```

Rotina de Serviço de Interrupções (ISR)

```
ISR (INT0_vect) { ... } // define a rotina para quando ocorre interrupção no INT0
```

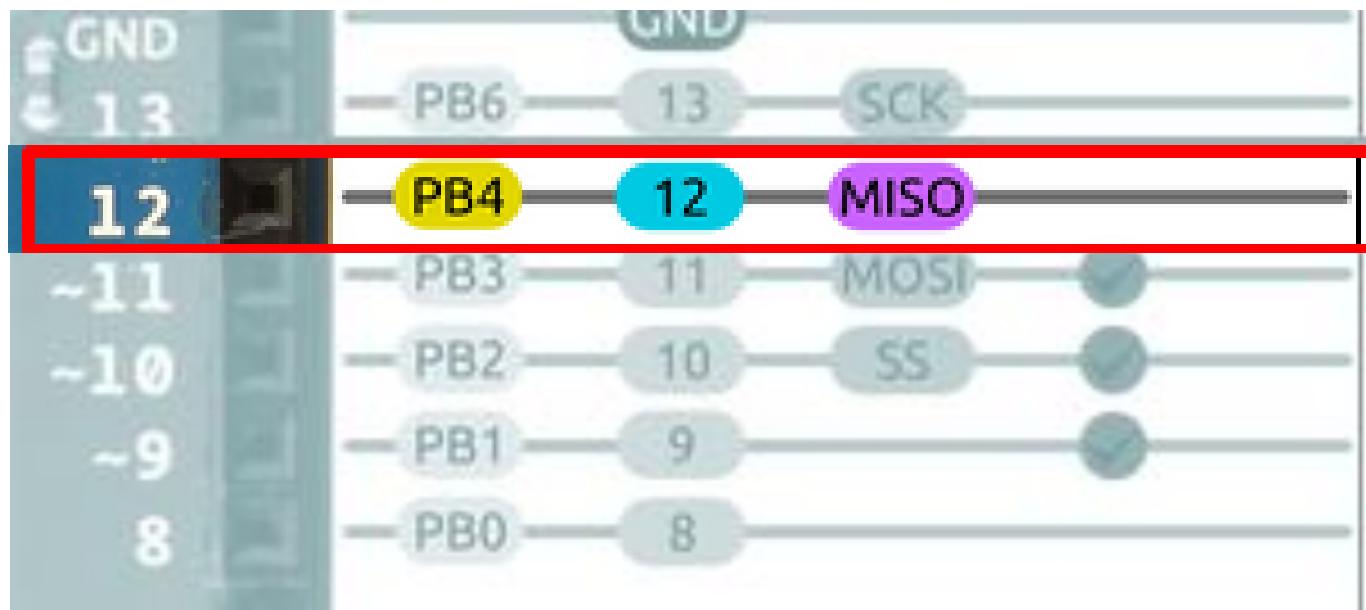
```
ISR (INT1_vect) { ... } // define a rotina para quando ocorre interrupção no INT0
```

```
1 const byte LED = (1<<4); // pino 12 (bit 4 da Porta B)
2 const byte BOT = (1<<3); // pino 03 (bit 3 da porta D)
3
4 void setup() {
5     DDRB |= LED; // configura pino 12 como saída - led
6     DDRD &= ~BOT; // configura pino 3 como entrada - botão
7     PORTD |= BOT; // habilita o pull up interno
8
9     EIMSK |= (1<<INT1); // habilita a interrupção externa int 1
10
11    EICRA |= (1<<ISC11) | (0<<ISC10); // configura interrupção externa int 1 - FALLING (pino 3)
12
13    sei(); // função que habilita as interrupções globalmente
14
15    PORTB &= ~LED; // inicia com led desligado (LOW)
16}
17
18 void loop() {
19}
20
21 // interrupção ISR's
22 ISR(INT1_vect) {
23     PORTB ^= LED;
24}
```

Primeiro as constantes LED e BOT são configuradas usando operadores *bitwise* para definir o pino de conexão.

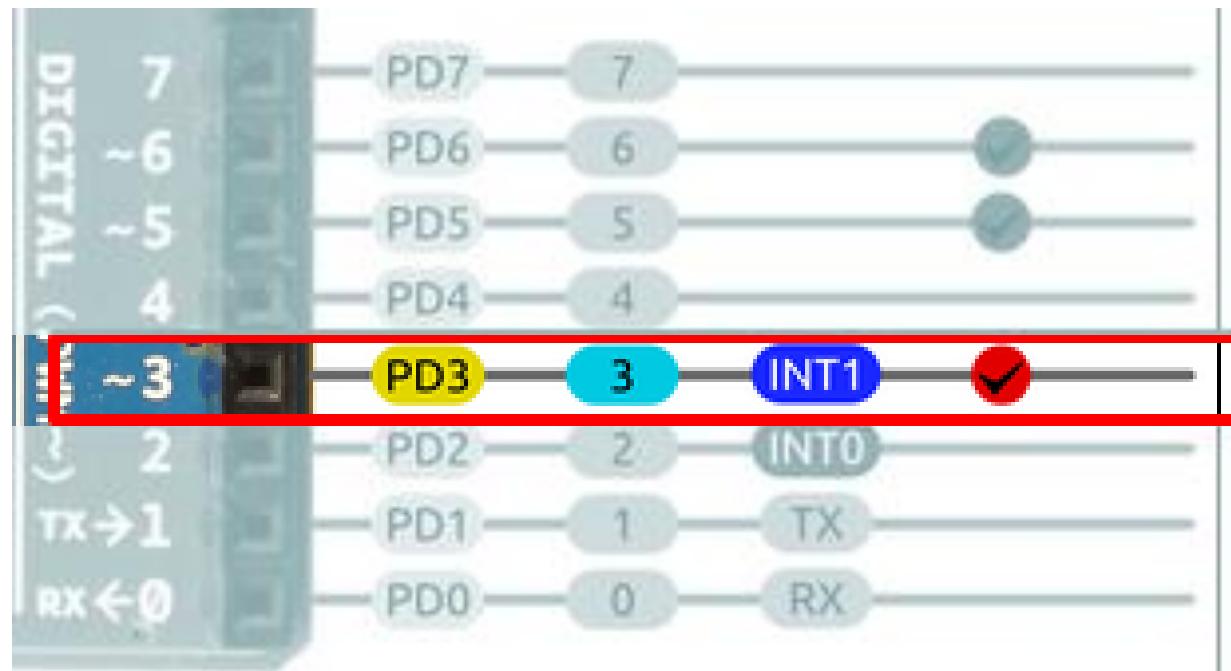
- O Pino 12 fica na Porta B4.

É necessário setar apenas o bit 4 da porta B: **const byte LED = (1<<4);**



O pino 3 fica na Porta D3.

Deve-se, então, setar então o bit 3 da porta D: **const byte BOT = (1<<3);**



Na estrutura **void setup()** teremos:

DDRB |= LED;

O pino 12 é definido como **OUTPUT**, ou seja, o **bit 4** da **PORTA B** é configurado com o valor **1**.

DDRD &= ~BOT;

O pino 3 é definido como **INPUT**, ou seja, o **bit 3** da **PORTA D** é configurado com o valor **0**.

PORTB |= BOT;

O **INPUT_PULLUP** interno do microcontrolador é habilitado. Para isto o pino 3 (bit 3) da PORTA D é colocado em “1”.

A configuração do **INPUT_PULLUP** serve para utilizarmos o resistor interno do microcontrolador e evitar flutuações indevidas de sinal.

Para habilitar a interrupção externa no pino 3 (INT1) utiliza-se o registrador **EIMSK**.

EIMSK |= (1<<INT1); // habilita a interrupção externa int 1

- Através do registrador **EICRA** configura-se o pino 3 para que ocorra uma interrupção externa na **borda de descida**, ou seja, dispara a ISR sempre que houver uma transição de HIGH para LOW (de 5V para 0V) no pino 3.

EICRA |= (1<<ISC11) | (0<<ISC10); // configura interrupção externa INT1 - FALLING (pino 3)

Obs.: Isto significa que quando o botão for pressionado, ocorrerá a interrupção externa pois o botão está configurado como **PULLUP** (gera um sinal de baixo nível - LOW - ao ser pressionado).

- A função **sei()** habilita globalmente as interrupções no circuito.
Isto é obrigatório quando se utiliza registradores EIMSK e EICRA.

PORTB &= ~LED; coloca um valor **0 (LOW)** no bit 4 da PORTA B (pino 12).

*Isto faz com que o programa se inicie com o LED apagado.
(obs.: LED = (1<<4);)*

ISR(INT1_vect) { PORTB ^= LED; } cria a função ISR que fará com que acenda ou apague o LED quando o botão for pressionado.

PORTB ^= LED; liga e desliga o LED a cada vez que o botão é pressionado.

TEMPORIZAÇÃO (TIMER)



- Os temporizadores são usados em todos os lugares.
- A gama de temporizadores varia de alguns microssegundos (como os ciclos de operação de um processador) a muitas horas (como as aulas teóricas).



O AVR é adequado para todas os casos, pois conta com temporizador com resolução de microssegundos!

Tudo neste mundo está sincronizado com o tempo.

- Você acorda às 6 horas, trabalha todos os dias por 8 horas, bebe água a cada 4 horas, etc.
- O conceito de temporização não se limita às suas rotinas diárias. Cada componente eletrônico funciona em uma base de tempo.
- Essa base de tempo ajuda a manter todo o trabalho sincronizado.
- Sem uma base de tempo, você não teria ideia de *quando* realizar determinada tarefa.

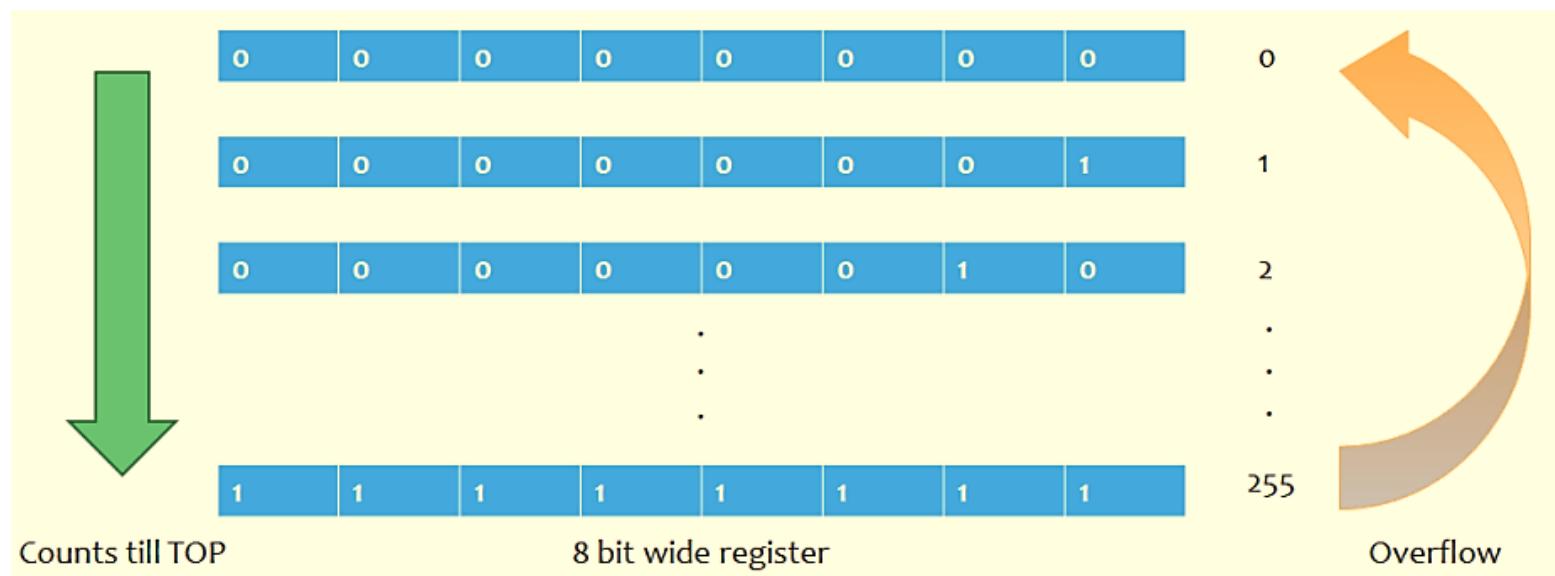
Temporizadores como registros

Basicamente, um temporizador é um registro!

Mas não é normal. O valor deste registro aumenta / diminui automaticamente.

No AVR, os temporizadores são de dois tipos: 8 e 16 bits.

O de 8 bits é capaz de contar $2^8 = 256$ etapas de 0 a 255;



Da mesma forma, um temporizador de 16 bits é capaz de contar $2^{16} = 65536$ etapas (de 0 a 65535).

Devido a esse recurso, os **temporizadores também são vistos como contadores**.

Quando atingem o seu limite máximo de contagem, retornam ao seu valor inicial de zero, ou seja, o temporizador / contador **transborda** .

No ATMEGA32, tem-se três tipos diferentes de temporizadores:

TIMER0 - temporizador de 8 bits

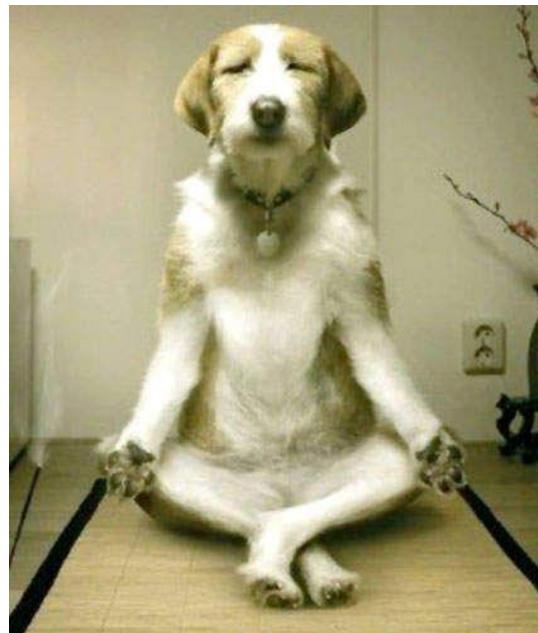
TIMER1 - temporizador de 16 bits

TIMER2 - temporizador de 8 bits

O temporizador é totalmente independente da CPU.
O mesmo opera em paralelo e não há intervenção da CPU, o que torna o cronômetro bastante preciso.

Além da operação normal, esses três temporizadores podem ser operados no modo **normal**, modo **CTC** ou modo **PWM**.

Antes, porém, vamos nos concentrar nos fundamentos de um temporizador.



Conceitos Básicos

Desde cedo, aprendemos a seguinte fórmula:

$$\text{período} = \frac{1}{\text{frequência}}$$

Considere que precisamos piscar um LED a cada 10 ms.
Sua frequência então será de $1/10 \text{ ms} = 100 \text{ Hz}$.

Se tivermos um cristal externo de 4 MHz, a frequência do *clock* da CPU será de 4 MHz.

Ao usarmos o temporizador/contador de 8 bits (que conta de 0 a 255), precisaremos de $T = 1 / 4M = 0,00025\text{s}$ para que o contador aumente uma unidade em seu estado atual.

Precisamos de um *delay* (atraso) de 10 ms.

Este pode ser um atraso muito curto, mas para o microcontrolador que tem uma resolução de 0,00025 ms, é um atraso bastante longo! 

Para ter uma ideia de quanto tempo leva, vamos calcular a contagem do temporizador a partir da seguinte fórmula:

$$\text{Contagem do temporizador} = \frac{\text{Tempo desejado}}{\text{Período de clock}} - 1$$

Substituindo o tempo desejado (10 ms) e Período de clock (0,00025 ms), obtemos uma **contagem do temporizador = 39999**.

Você consegue imaginar isso? Precisamos de 39999 pulsos de clock para um atraso de apenas 10 ms!

É possível utilizarmos o temporizador de 8 bits sabendo-se que o mesmo tem um limite superior de 255?

Definitivamente, neste caso, precisaremos de um temporizador de 16 bits (que é capaz de contar até 65535) para atingir esse atraso (*delay*).

Assumindo $f_{CPU} = 4 \text{ MHz}$ e um temporizador de 16 bits ($\text{MAX} = 65535$); substituindo na fórmula anterior, podemos obter um ***delay máximo*** de 16,384 ms.

Agora, e se precisarmos de um *delay* maior, por exemplo 20 ms? Estamos presos?!

Suponha que se diminuirmos a f_{CPU} de 4 MHz para 0,5 MHz (ou seja, 500 kHz), o período de *clock* aumentará para $1 / 500k = 0,002$ ms.

Agora, se substituirmos o *delay* necessário = 20 ms com o período de *clock* igual a 0,002 ms , teremos a **contagem do temporizador = 9999** .

Como podemos ver, isso pode ser facilmente alcançado usando um temporizador de 16 bits.

Nessa frequência, um ***delay* máximo** de 131,072 ms pode ser alcançado.

Mas quase sempre, mudar a frequência de *clock* da CPU não é uma boa ideia.

Prescaler

O prescaler (ou pré-escala) é um recurso que possibilita a utilização de subdivisões do *clock* da CPU sem alterá-lo.

Mas não pense que você pode usar o prescaler livremente.
Isso tem um custo!

Há uma compensação entre resolução e duração

Quando reduzimos a frequência de *clock* da CPU de 4MHz para 0,5MHz, o delay máximo aumentou de 16,384 ms para 131,072 ms.

A resolução também aumentou de 0,00025 ms para 0,002 ms (teoricamente, a resolução diminuiu).

Isso significa que cada período de *clock* terá 0,002 ms.

Mas qual é o problema disso?

O problema é que a precisão diminuiu!

Anteriormente podia-se gerar um *delay* de 0,1125 ms, por exemplo.

Para isto, precisaríamos que o contador do temporizador fosse configurado para 449.

$(0,1125 / 0,00025 - 1 = 449)$ -> fórmula anterior.

Com a diminuição do *clock* para 0,5 MHz, ou seja, um período igual a 0,002 ms, teremos:

$$\text{Contagem do temporizador} = \frac{\text{Tempo desejado}}{\text{Período de clock}} - 1$$

$$\text{Contagem do temporizador} = 0,1125 / 0,002 - 1 = \mathbf{55,25}.$$

Não é possível fazer com que o contador tenha módulo igual a 55,25.

O novo temporizador pode gerar um delay 0,112 ms e depois, de 0,114 ms, mas nenhum outro valor intermediário (como o 0,1125, por exemplo).

Escolhendo Prescalers

Considere que você deseje gerar um delay de 184 ms (um número aleatório) e que $f_{CPU} = 4 \text{ MHz}$.

O AVR oferece os seguintes valores de *prescaler*: 8, 64, 256 e 1024.

Um prescaler de 8 significa que a frequência de clock efetiva será $f_{CPU} / 8$.

Se substituirmos cada um desses valores na fórmula teremos os resultados a seguir:

$$\text{Required Delay} = 184 \text{ ms}$$
$$F_{\text{CPU}} = 4 \text{ MHz}$$

Prescaler	Clock Frequency	Timer Count
8	500 kHz	91999
64	62.5 kHz	11499
256	15.625 kHz	2874
1024	3906.25 Hz	717.75

Destes, o *prescaler* de 8 não pode ser usado porque o valor do contador do *timer* excede o limite de 65535.

Não é possível escolher 1024 pois o contador do *timer* tem que ser inteiro.

Somente os valores do *prescaler* de 64 e 256 são viáveis.

Qual destes dois escolhemos?

A princípio, devemos escolher o 64, pois nos fornece uma resolução maior.

No entanto, se precisarmos do temporizador para uma duração maior em outro ponto do código, devemos escolher 256.

Assim, sempre escolhemos o prescaler que fornece o valor do contador dentro do limite viável (255 ou 65535) e o valor do contador deve ser sempre um número inteiro.