# Sistemas Digitais

# Microcontroladores

Prof. Sérgio Rebelo

Email: sergio.f.rebelo@gmail.com

# UFCD 6072 – Microcontroladores

- **Carga Horária:**25 horas
- **Objetivos:**
  - Identificar a estrutura típica de um sistema microcontrolado.
  - Identificar principais características do microcontrolador em estudo.
  - Identificar os registos de usos gerais e especiais.
  - Caracterizar as memórias internas e externas.
  - Descrever o modo de funcionamento das portas de entrada e saída de dados.
  - Identificar os modos de endereço usados nas instruções do microcontrolador.
  - Descrever os diferentes grupos de instruções do microcontrolador.
  - Construir programas que utilizem as instruções de transferência e processamento de dados, assim como as de teste e salto.
  - Descrever os diferentes modos de funcionamento dos contadores/temporizadores.
  - Descrever o funcionamento das interrupções no microcontrolador.
  - Identificar e realizar fluxogramas.
  - Aplicar as principais instruções do microcontrolador em estudo.

# UFCD 6072 – Microcontroladores

- **Conteúdos**
  - Memória, microprocessador, periféricos de entrada/saída
  - Constituição de um sistema microcontrolado.
  - *Pinout* do microcontrolador.
  - Simbologia e técnicas de realização de fluxogramas.
  - Diagrama de blocos interno do microcontrolador em estudo:
    - Estrutura interna
    - Memória de programa e dados
    - A unidade lógica e aritmética
    - Registos de funções especiais
    - Modos de endereçamento
    - Tipos de instruções
    - Controlo de interrupções
    - Temporizadores
  - Conjunto de instruções do microcontrolador em estudo.
  - Utilização de software de simulação, programação e debugging.

# UFCD 6072 – Microcontroladores

## Instruções e Estruturas de Programação

# Programação

A programação é um conjunto de recursos que permite criar diversas sequências de passos lógicos com o objetivo de cumprir as necessidades dos sistemas desenvolvidos.

O processo de escrever  código requer frequentemente conhecimentos em várias áreas distintas, além do domínio da linguagem a utilizar, algoritmos específicos e lógica formal.

Inicialmente, os computadores interpretavam somente instruções em uma linguagem específica, uma linguagem de programação de baixo nível conhecida como código máquina.

Para facilitar o trabalho de programação foram criadas instruções, implementadas por palavras ou letras do inglês, codificando e criando assim uma linguagem de maior nível conhecida como Assembly.

# Programação

Algumas linguagens de Programação:

# Linguagens de programação

Linguagens de programação de alto nível são linguagens com um nível de abstração relativamente elevado, longe do código máquina e mais próximo à linguagem humana. Desse modo, as linguagens de alto nível não estão diretamente relacionadas com a arquitetura do computador.

Nas linguagens de baixo nível trata-se de linguagens de programação que compreendem as características da arquitetura do computador. Assim, utiliza-se somente instruções que necessitam de conhecimento do funcionamento e dos registos da máquina.

Nesse sentido, as linguagens de baixo nível estão diretamente relacionadas com a arquitetura do computador.

# Funções

As funções são usadas para criar pequenos blocos de código separados do programa principal.

As funções são importantes pois devolvem valores ou argumentos, ajudam a fragmentar o código em partes menores e podem ser utilizadas mais de uma vez no mesmo programa, poupando tempo de programação e inúmeras linhas de código.

A função exemplo soma dois valores e depois devolve o resultado ao programa principal.

```
int a,b,c;

int soma(int a, int b)
      { c=a+b;
      return c;
      }

      void setup () {
                  Serial.begin (9600);
                     }


      void loop() {
                        a=2;
                        b=3;
                        c= soma (a,b);
                        Serial.println(c);
                   }
```

# Funções Específicas

As duas funções principais dos programas realizadas na plataforma Arduino são a função setup() e loop().

Setup () é chamada quando um programa começa o seu funcionamento. Deve ser utilizada como função de inicialização de variáveis, os modos dos pinos, declarar o uso de bibliotecas. Esta função será executada apenas uma vez após a placa Arduino ser ligada ou quando é executado o *reset* ao microcontrolador. Setup

A a função loop() faz exatamente o que seu nome sugere, entra em looping (executa sempre o mesmo bloco de código), permitindo ao programa fazer mudanças e responder. Use esta função para controlar ativamente a placa Arduino.

# Tipos de Variáveis

Variáveis são expressões que são utilizadas nos programas para armazenar valores como a leitura de um sensor em um pino analógico. Existem por exemplo:

- Variáveis do tipo Booleana: Assim identificadas em homenagem a George Boole, podem ter apenas dois valores: verdadeiro (true) e falso (false).

  boolean running = false;

- Variáveis do tipo número inteiro: Identificadas pelo operador Int é o um tipo de dado para armazenamento numérico capaz de guardar números de 2 bytes. Isto abrange a faixa de -32.768 a 32.767 (valor mínimo de -2^15 e valor máximo de (2^15) -1).

  int ledPin = 13;

- Variáveis do tipo Caracter: Identificadas pelo operador Char é um m tipo de dado que ocupa 1 byte de memória e armazena o valor de um caractere ASCII. Caracteres literais são escritos entre aspas.

  char myChar = ‘A’;

# Operadores Lógicos

Estes operadores podem ser usados dentro da condição em uma estrutura de decisão IF.

– && (“e” lógico) Verdadeiro apenas se os dois operandos forem verdadeiros, ou seja, a primeira condição e a segunda forem verdadeiras.
Exemplo: if (digitalRead(2) == 1 && digitalRead(3) == 1)
{ // ler dois interruptores // ...
} é verdadeiro apenas se os dois interruptores estiverem fechados.

-|| (“ou” lógico) Verdadeiro se algum dos operandos for verdadeiro, ou seja, se a primeira ou a segunda condição for verdadeira.
Exemplo: if (x > 0 || y > 0)
{ // ...
} é verdadeiro apenas se x ou y forem maiores que 0.

– ! (negação) Verdadeiro apenas se o operando for falso.

Exemplo: if (!x)
{ // ...
} é verdadeiro apenas se x for falso (ou seja, se x for igual a 0).

# Estrutura de Decisão IF

A estrutura IF é usado juntamente com um operador de comparação e serve para verificar quando uma condição é satisfeita. O formato para uma verificação IF é:

```
if (algumaVariavel > 50)
    { // faça alguma coisa
    }
```

Neste exemplo o programa verifica se “algumaVariavel”) é maior que 50. Se for, o programa realiza uma ação específica. Portanto se a condição de teste for verdadeira o fluxo do programa executa o código escrito dentro dos parêntesis; Caso contrário o programa salta este bloco de código.

As chavetas podem ser omitidas após uma estrutura IF se só houver uma única linha de código que será executado em modo condicional.

# Estrutura de Decisão IF

Alguns exemplos desta estrutura são:

```
if (x > 120) digitalWrite(LEDpin, HIGH);

if (x > 120) digitalWrite(LEDpin, HIGH);

if (x > 120) {digitalWrite(LEDpin, HIGH);} //
```

A condição que  está a ser verificada necessita o uso de pelo menos um dos operadores de comparação:

x == y (x é igual a y)

x != y (x é não igual a y)

x < y (x é menor que y)

x > y (x é maior que y)

x <= y (x é menor ou igual a y)

x >= y (x é maior ou igual a y)

# Estruturas de Controlo do Fluxo do Programa

São instruções que permitem decidir e realizar diversas repetições de acordo com alguns parâmetros. Entre os mais importantes:

- Switch/case : Do mesmo modo que as estrutura if, as switch/case controlam o fluxo dos programas. Switch/case permite ao programador construir uma lista de “casos” possíveis dentro de um bloco. O programa verifica cada caso com a variável de teste e executa o código se encontrar um valor idêntico.

```
switch (var) {
case 1:
//faça alguma coisa quando var == 1
case 2:
//faça alguma coisa quando var == 2
default:
// se nenhum valor for idêntico, faça o default
// default é opcional }
```

# Estruturas de Controlo do Fluxo do Programa

A estrutura While fará com que o bloco de código entre chavetas se repita contínua e indefinidamente até que a expressão de teste se torne falsa. Alguma alteração no programa terá de provocar uma mudança no valor da variável que está sendo verificada ou o fluxo do programa ficará "preso" dentro da estrutura while.

Isto poderia ser realizado através do incremento de uma variável ou uma condição externa. Por exemplo pode escrever-se:

```
var = 0;
while(var < 200)
    { // algum código que se repete 200 vezes
    var++;
    }
```

# Estruturas de Controlo do Fluxo do Programa

A estrutura FOR é utilizada para repetir um bloco de código delimitado por chavetas.

Um contador com incremento é normalmente usado para controlar e finalizar o loop.

A estrutura FOR é útil para qualquer operação repetitiva, e é frequentemente usada com arrays para operar de forma simples em conjuntos de dados ou de pinos.

Um exemplo desta estrutura pode ser a variação do valor analógico da tensão de saída, sendo:

```
int PWMpin = 13; // um LED no pino 13
    void setup() { // nenhum setup é necessário }
    void loop() {
                for (int i=0; i <= 255; i++)
                    { analogWrite(PWMpin, i); delay(10);
                    }
            }
```

# Funções digitais específicas

As funções digitais são orientadas para a configuração e definição do estado das entradas e saídas digitais. Entre estas funções especificas para a plataforma Arduino temos:

- pinMode(pino,estado): Configura o pino especificado para que se comporte ou como uma entrada (input) ou uma saída (output). Exemplo:

  pinMode(9, OUTPUT); // determina o pino digital 9 como uma saída.

- digitalRead() Lê o valor de um pino digital especificado, devolvendo o valor HIGH ou LOW. Exemplo:

  digitalRead(pin) buttonState = digitalRead(9); // Leitura do estado do pino 9.

- digitalWrite(pino,estado) Escreve um valor HIGH ou um LOW em um pino digital.

  Exemplo:

  digitalWrite(pin, valor) digitalWrite(9, HIGH); // Coloca o pino 9 em estado HIGH.

# Funções analógicas específicas

As funções analógicas possibilitam a escrita ou a leitura de valores analógicos em determinados pinos do microcontrolador. Essas funções são:

- analogRead(): Lê o valor de um pino analógico especificado. A placa Arduino Uno contém um conversor analógico-digital de 10 bits com 6 canais.

Com é realizada uma conversão entre tensões de entrada entre 0 e 5 V para valores inteiros entre 0 e 1023. Isto permite uma resolução pré-definida entre leituras de 5/1024 unidades ou 0,0049 V (4.9 mV) por unidade.

Exemplo:

analogRead(pin) int a = analogRead (A0); // Lê o valor do pino analógico A0 e armazena este valor na variável "a".

# Funções analógicas específicas

As funções analógicas possibilitam a escrita ou a leitura de valores analógicos em determinados pinos do microcontrolador. Essas funções são:

- analogWrite() Escreve um valor analógico (através da ténica de modelação por largura de impulso PWM) em um pino específico . Pode ser usado para acender um LED variando a intensidade luminosa ou comandar um motor para este rodar com velocidade variável. Exemplo:

```
analogWrite (9,134); // Envia o valor analógico 134 para o pino 9.
```

# Bibliotecas

Quando se cria um código/programa usando uma linguagem de programação, neste caso C/C++, existe a possibilidade de usar um conjunto de funções pré-criadas por outros programadores que já resolvem determinados problemas.

A esse conjunto de funções damos o nome de bibliotecas, do inglês, library.

São exemplos destas bibliotecas:

- Servo: Esta é utilizada quando temos um servomotor acoplado ao Arduino, assim de uma forma intuitiva, consegue-se posicionar o motor numa determina posição.

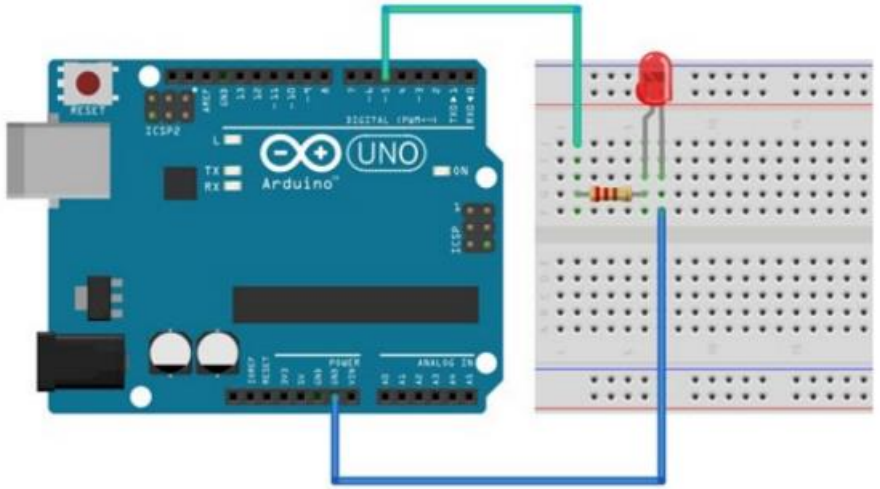
- Ultrasonic: A livraria ultrasonic é utilizada para controlo do sensor de ultrasom. Facilitando a leitura da distância lida por este.

# UFCD 6072 – Microcontroladores

## Programas e Exercícios

# Programa – Piscar LED

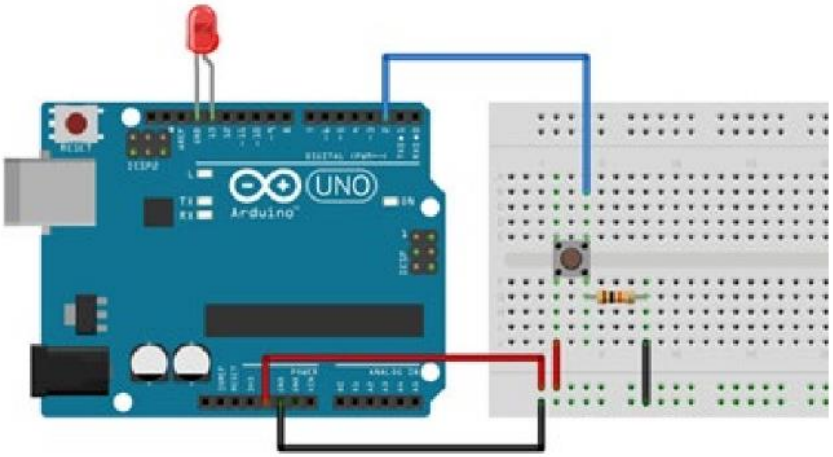
A partir do circuito representado realize um programa faça a variação do estado de uma saída (Pino 5). O LED deve estar aproximadamente 2 segundos acesso e 1 apagado.




```
// Nome para o pino 5: int led = 5;

void setup() {
// Inicializa o pino digital como saída.
pinMode(led, OUTPUT);
}

void loop() {
digitalWrite(led, HIGH); // Acende o LED
delay(2000);
// Aguarda um segundo (2s = 1000ms)
digitalWrite(led, LOW); // Apaga o LED
delay(1000);
// Aguarda um segundo (1s = 1000ms)
}
```

# Programa - Leitura de Pino de Entrada

A partir do circuito representado realize um programa que ao ser pressionado o botão (Leitura do Pino 2), faça acender o LED (Pino 13).

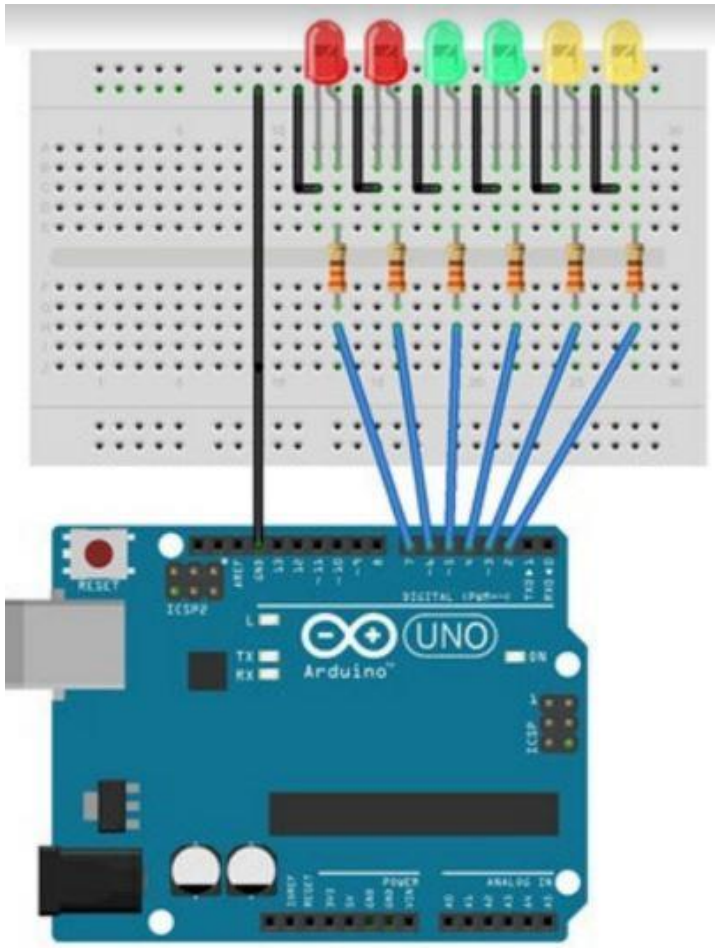



```
const int buttonPin = 2; // o numero do pino do botão
const int ledPin = 13; // o numero do pino do LED
int buttonState = 0;
// variavel para ler o estado do botao

void setup() {
// inicializa o pino do LED como saida:
pinMode(ledPin, OUTPUT);
// inicializa o pino do botao como entrada:
pinMode(buttonPin, INPUT);
}
```

```
void loop(){
// faz a leitura do valor do botao:
buttonState = digitalRead(buttonPin);
if (buttonState == HIGH) {
// liga o LED:
digitalWrite(ledPin, HIGH);
}
else {
// desliga o LED:
digitalWrite(ledPin, LOW);
}
}
```

# Programas Tipo– Ciclo FOR

Realize um programa que faça a repetição de uma ação sobre uma série de pinos. (Pinos 2 até 7). Pretende-se fazer piscar cada LED em sequência, do pino 2 até ao 7 e de seguida do pino 7 até ao 2.




Conceito

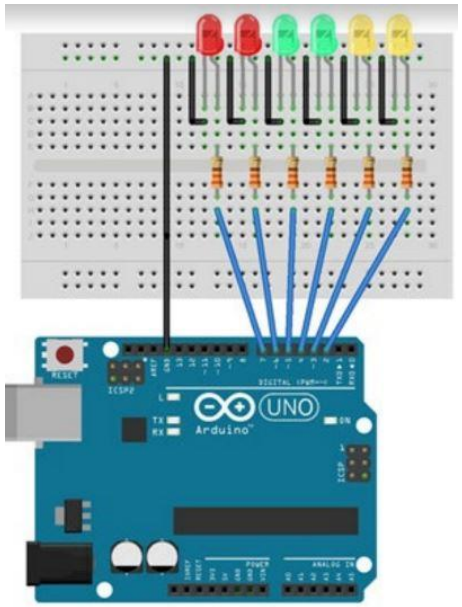
Utilização da função for() para designar os pinos digitais de 2 a 7 como saídas.

No loop principal, dois ciclos for() são usados para incrementar o ciclo, percorrendo os LED, um por um, a partir de pino 2 ao pino 7.

Uma vez que o pino 7 está aceso, o processo inverte, percorrendo por ordem inversa a ação de piscar os LED.

# Programas Tipo – Ciclo FOR

```
int timer = 100; // Quanto maior o valor,
mais lenta a sequencia de Leds.
void setup() {
// Use for loop para inicializar
cada pino como saida:
for (int thisPin = 2; thisPin < 8; thisPin++)
{
pinMode(thisPin, OUTPUT);
}
}
```



```
void loop() {
// loop desde o pino mais baixo ate o
mais alto:
for (int thisPin = 2; thisPin < 8; thisPin++)
{
// liga este pino:
digitalWrite(thisPin, HIGH);
delay(timer);
// desliga este pino:
digitalWrite(thisPin, LOW);
}
// loop desde o pino mais alto ate o mais
baixo:
for (int thisPin = 7; thisPin >= 2; thisPin--) {
// liga este pino:
digitalWrite(thisPin, HIGH);
delay(timer);
// desliga este pino:
digitalWrite(thisPin, LOW);
}
```

# Programas Tipo– Leitura Entrada Analógica

Realize um programa que faça a leitura de um pino de uma entrada analógica (Pino A0), mapeie o resultado para um intervalo de 0 a 255, e usar esse resultado para definir a modulação PWM de um pino de saída (Pino 9). Essa tensão deve ser aplicada a um LED.

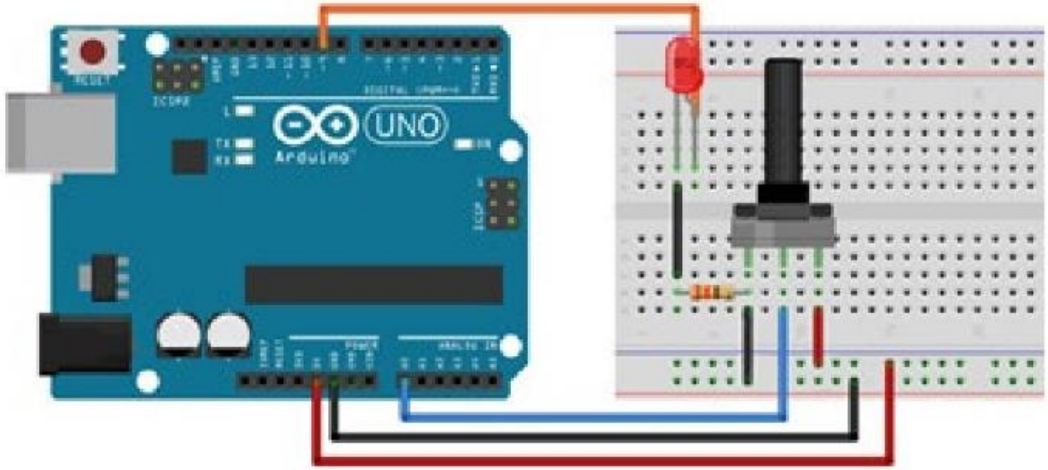
Conceitos:

Leitura de uma entrada analógica.

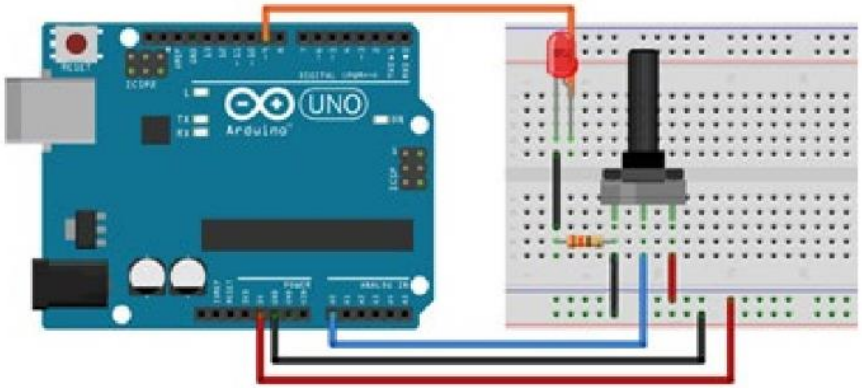
Conversão função MAP()

Visualização de mensagens (“Serial Monitor”)

Controlar uma saída analógica



# Programas Tipo – Leitura Entrada Analógica

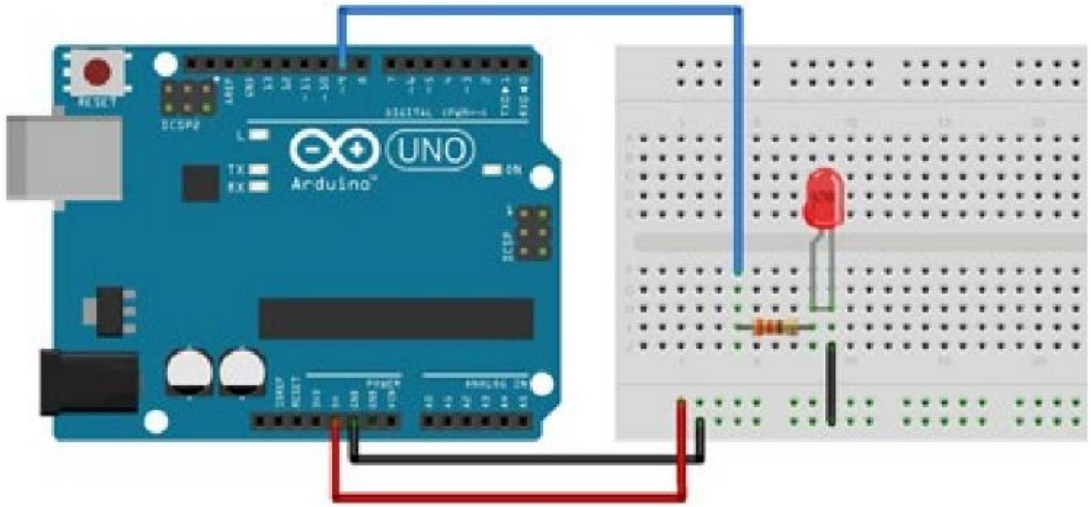



```
// constantes nao sao alteradas:
const int analogInPin = A0; // Entrada analogica
const int analogOutPin = 9; // Saida analogica
int sensorValue = 0; // leitura do potenciometro
int outputValue = 0; // leitura da saida PWM (analogica)
void setup() {
// inicializa a comunicacao serial:
Serial.begin(9600);
}
```

```
void loop()
{
// faz a leitura da entrada analógica:
sensorValue = analogRead(analogInPin);

// mapeia o resultado da entrada analógica dentro do intervalo de 0 a 255:
outputValue = map(sensorValue, 0, 1023, 0, 255);
// muda o valor da saída analógica:
analogWrite(analogOutPin, outputValue);
// imprime o resultado no monitor serial:
Serial.print("sensor = " );
Serial.print(sensorValue);
Serial.print("\t output = ");
Serial.println(outputValue);
// Aguarda 2 milissegundos antes do proximo loop:
delay(2);
}
```

# Programas Tipo – Saída Analógica

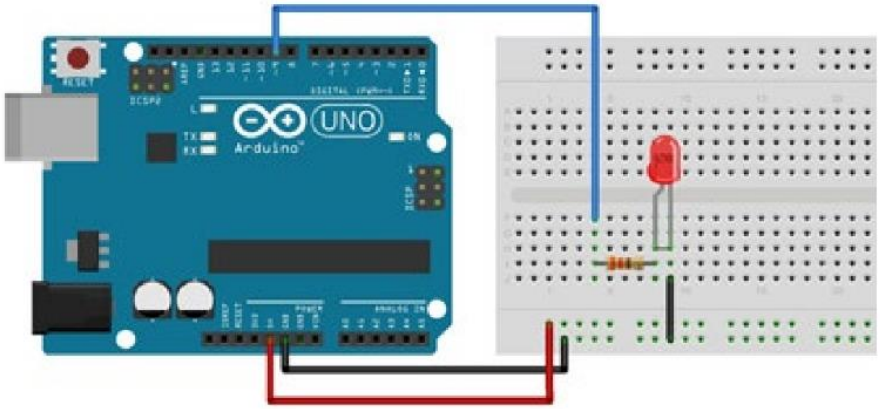
Realize um programa que faça uso da função analogWrite() para apagar um LED numa variação gradual. (PWM).




Conceito :

A função analogWrite(pin, value) que vai ser utilizada no loop principal requer dois argumentos, um deles diz respeito ao pino que deve acionar e outra indicando qual valor PWM a utilizar. O valor PWM pode variar entre 0 (totalmente desligado) a 255 (totalmente ligado)

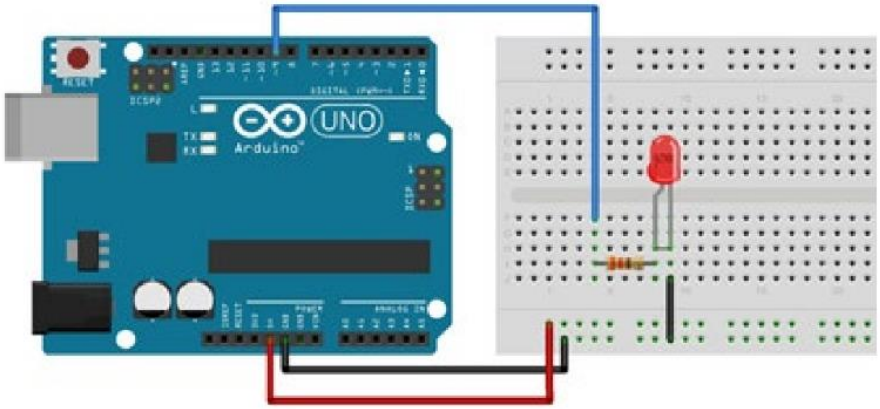
# Programas Tipo – Saída Analógica




```
int led = 9; // pino do LED
int brightness = 0; // intensidade do brilho do LED
int fadeAmount = 5; // incremento da variável
void setup() {
// define o pino 9 como saida:
pinMode(led, OUTPUT);
}
```

```
void loop() {
// define o brilho do pino 9:
analogWrite(led, brightness);
// muda o brilho para o proximo loop:
brightness = brightness + fadeAmount;
// inverte a direcao do “fade” ao final do mesmo:
if (brightness == 0 || brightness == 255) {
fadeAmount = -fadeAmount ;
}
// aguarda 30 milissegundos para ver o efeito da variação:
delay(30);
}
```

# Programas Tipo – Saída Analógica

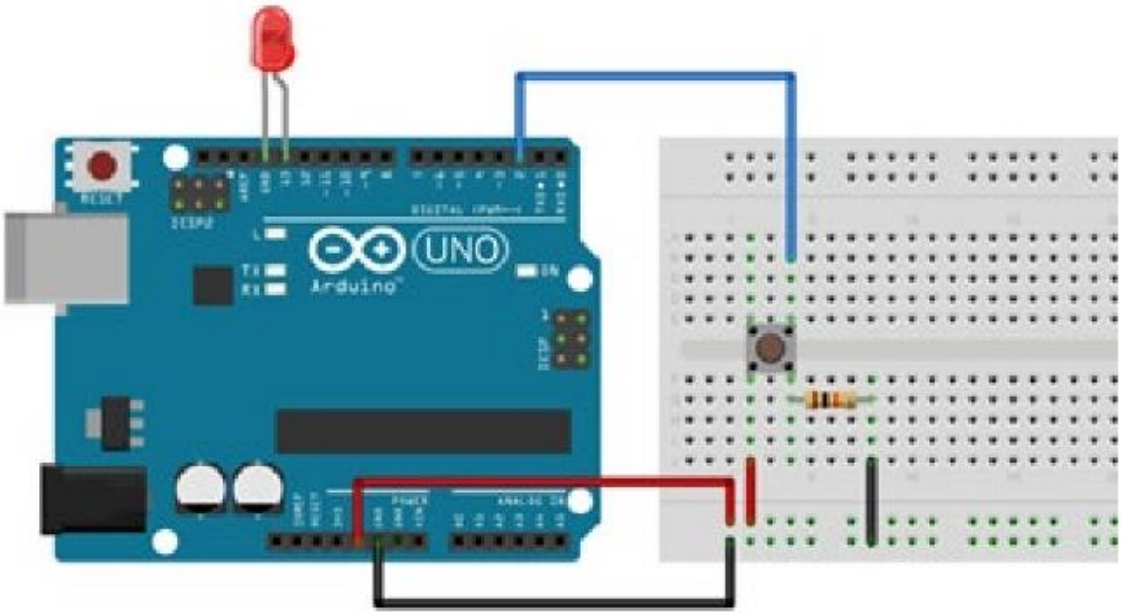



```
int led = 9; // pino do LED
int brightness = 0; // intensidade do brilho do LED
int fadeAmount = 5; // incremento da variável
void setup() {
// define o pino 9 como saida:
pinMode(led, OUTPUT);
}
```

```
void loop() {
// define o brilho do pino 9:
analogWrite(led, brightness);
// muda o brilho para o proximo loop:
brightness = brightness + fadeAmount;
// inverte a direcao do “fade” ao final do mesmo:
if (brightness == 0 || brightness == 255) {
fadeAmount = -fadeAmount ;
}
// aguarda 30 milissegundos para ver o efeito da variação:
delay(30);
}
```

# Programas Tipo

Realize um programa que ao ser pressionado o botão (Leitura do Pino 2), faça acender o LED (Pino 13) e monitorize o estado do interruptor estabelecendo a comunicação serie entre o Arduino e o computador. (Mensagens de estados “1” ou “0”)



# Programas Tipo

Conceitos:

Configuração da velocidade da comunicação serie para 9600 bits de dados por segundo;

Definir o pino digital 2, o pino que vai fazer a leitura do botão como uma entrada digital;

Criação da estrutura de controlo (IF) (Botão pressionado Mensagem “1”.)

Utilização da ferramenta “Serial Monitor” no ambiente IDE Arduino para visualização das mensagens

```
int pushButton = 2;
int ledPin = 13;.
void setup() {
Serial.begin(9600);
pinMode(pushButton, INPUT); //
define o botao como uma entrada.
pinMode(ledPin, OUTPUT); //define
o LED como uma saída.
}
```

```
void loop() {
// faz a leitura do pino de entrada:
int buttonState = digitalRead(pushButton);
if (buttonState == 1) {
digitalWrite(ledPin, HIGH);
} else {
digitalWrite(ledPin, LOW);
}
// imprime o estado do botao:
Serial.println(buttonState);
delay(1); // delay entre leituras (em milissegundos)
}
```

# UFCD 6072 – Microcontroladores

## Portos de Entrada – Registos

# Definição de registos (I/O)

- O microcontrolador ATEGA 328 têm 3 portos de I/O:

  Porto B- Usado pelo pino digital 8 ao pino digital 13; Porto C-Usado por pinos analógicos; Porto D- Usado pelo pino digital 0 ao pino digital 7.

- São utilizados registos para controlar os pinos separadamente. Esses registros são:

  DDR-Define  os pinos como entrada ou saída;

  PORT- Define torna o pino BAIXO ou ALTO;

  PIN-Utilizado para “ler” o estado dos pinos de entrada

- Porto B -DDRB, PORTB, PINB - cada um com 8 bits para os pinos 8 a 13 (os bits 6 e 7 não devem ser alterados porque são utilizados pelo circuito oscilador do microcontrolador)
- Porto C-DDRC, PORTC, PINC-cada um com 8 bits para os pinos A0 a A5 (podemos controlar apenas 6, os pinos 6 e 7 estão disponíveis em outras placas).
- Porto D - DDRD, PORTD, PIND-cada um com 8 bits para os pinos 0 a 7.

# Definição de registos (I/O)

### 14.4.2 PORTB – The Port B Data Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x05 (0x25) | PORTB7 | PORTB6 | PORTB5 | PORTB4 | PORTB3 | PORTB2 | PORTB1 | PORTB0 | PORTB |
| 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 | |

### 14.4.3 DDRB – The Port B Data Direction Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x04 (0x24) | DDB7 | DDB6 | DDB5 | DDB4 | DDB3 | DDB2 | DDB1 | DDB0 | DDRB |
| 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 | |

### 14.4.4 PINB – The Port B Input Pins Address(1)

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x03 (0x23) | PINB7 | PINB6 | PINB5 | PINB4 | PINB3 | PINB2 | PINB1 | PINB0 | PINB |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | |

- O registo DDRB dá a possibilidade de controlar o modo de funcionamento (I/O) dos pinos 8 a 13. Por exemplo, se o bit para o pino 8 tiver o valor “0”, isso significa que o pino 8 fica definido como entrada, caso contrário, se tiver valor, “1” é um pino de saída.

# Definição de registos (I/O)

- Utilização dos registos na definição dos pinos do microcontrolador:

  DDRB = B11111111; isso significa que todos os pinos são saídas

  DDRB = B11111100; isso significa que os pinos 8 e 9 são entradas e os pinos 10 a 13 são saídas.

- Outro método para declarar pode ser:

  DDRB = 0b11111111 ou DDRB = 0b11111100;

- Depois de declará-los como entradas ou saídas, os pinos podem ter o valor “0” ou “1”. Para isso temos registo PORTB que funciona de modo semelhante ao DDRB, a única diferença é que “0” significa estado BAIXO LOW e “1” significa estado ALTO HIGH. Portanto por exemplo: PORTB = B11111111; isso significa que todos os pinos vão estar no nível lógico “1” ( ALTO); PORTB = B11011111; isso significa que o pino 13 é vai estar colocado a nível lógico “0” (BAIXO).

# Programa – Utilização de Registo

- Programa 1 – Colocação de um LED (pino 13) a piscar com frequência de 1 Hz.

```
void setup () {
DDRB = B11100000; // pino 13
está em modo de saída
}
void loop () {
PORTB = B11100000; // Pino
13 alto e ligue o led
delay (1000);
PORTB = B11000000; // Pino
13 baixo e desligue o led
delay (1000);
}
```

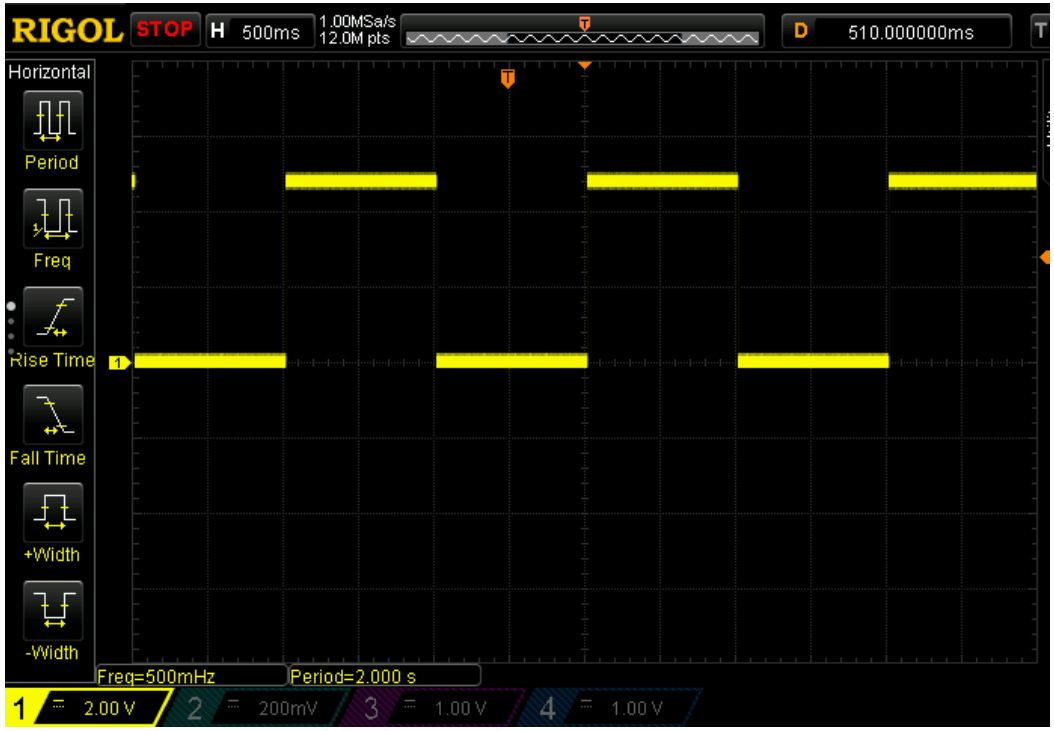



Figura 1 – Representação do nível de tensão no pino 13 (Registos).

# Programa – Utilização de funções

- Programa 2 – Colocação de um LED (pino 13) a piscar com frequência de 1 Hz. (Programa "Blink".)

```
void setup() {
  // initialize digital pin LED_BUILTIN
as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
// turn the LED on (HIGH is the
voltage level)
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
// turn the LED off by making the
voltage LOW
  delay(1000);
}
```

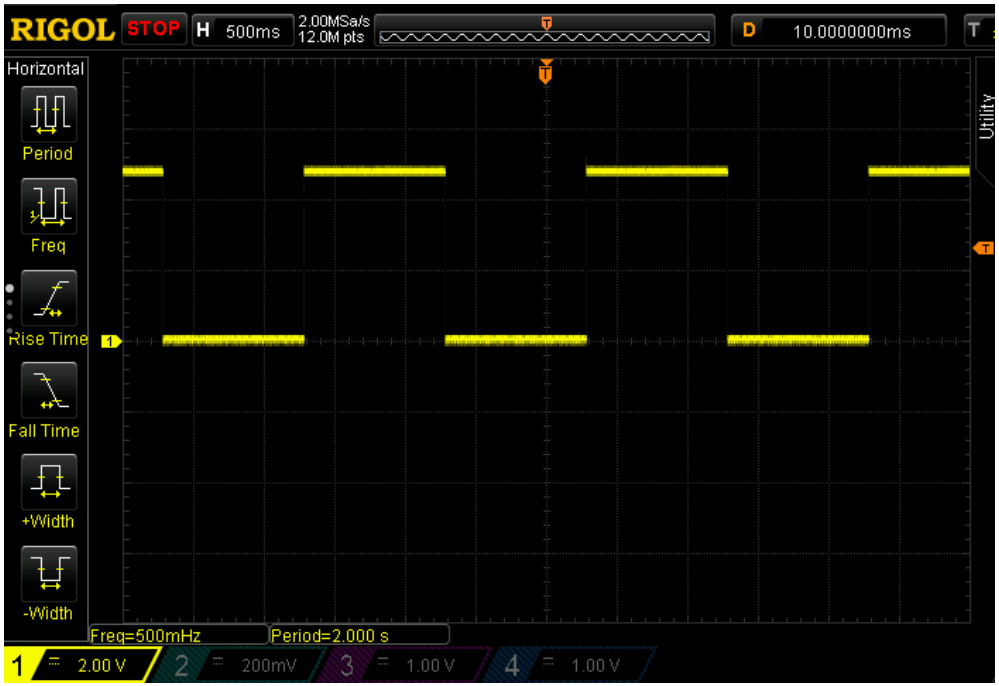



Figura 2 – Representação do nível de tensão no pino 13. (Funções pré-definidas ).

# Definição de registos (I/O)

- Para os portos C e D, a forma de definir os modo (I/O) ou o estado dos pinos (“0” ou “1”) é feito de forma análoga ao porto B. No PORTD com os pinos 0 e 1 são utilizados para a comunicação série pelo que o seu estado não deve ser alterado.

- Pode ser utilizada a função função “OU” ou o operador “|” para operações bit a bit. Porta OR (x | 0 = x ). Por exemplo para o registo DDRD:

  DDRD = DDRD | B11111100; isso significa que os pinos de 2 a 5 são saídas e os pinos 0 e 1 permanecem com o valor pré-definido sem serem modificados.

- Pode utilizar-se a função “AND” ou o operador ”&”. Se ambos os bits forem “1” o resultado é “1”, caso contrário é “0”, Porta AND (x & 1 = x).

- Por exemplo para o registo DDRD:

  DDRD = DDRD & B11111111, os pinos 0 e 1 devem ser multiplicados por “1”, caso contrário, o resultado será sempre “0”.

# Exemplo

- Programa 3 – Alterar o estado dos pinos 2 até 7 (PORTD), entre “0” e “1” à frequência de 50 KHz.

```
void setup() {
pinMode(2, OUTPUT);
 pinMode(3, OUTPUT);
pinMode(4, OUTPUT);
pinMode(5, OUTPUT);
pinMode(6, OUTPUT);
pinMode(7, OUTPUT);
}

void loop() {
  digitalWrite(2, HIGH);
  digitalWrite(3, HIGH);
  digitalWrite(4, HIGH);
  digitalWrite(5, HIGH);
  digitalWrite(6, HIGH);
  digitalWrite(7, HIGH);
  delayMicroseconds(10);
  digitalWrite(2, LOW);
  digitalWrite(3, LOW);
  digitalWrite(4, LOW);
  digitalWrite(5, LOW);
  digitalWrite(6, LOW);
  digitalWrite(7, LOW);
  delayMicroseconds(10);
}
```

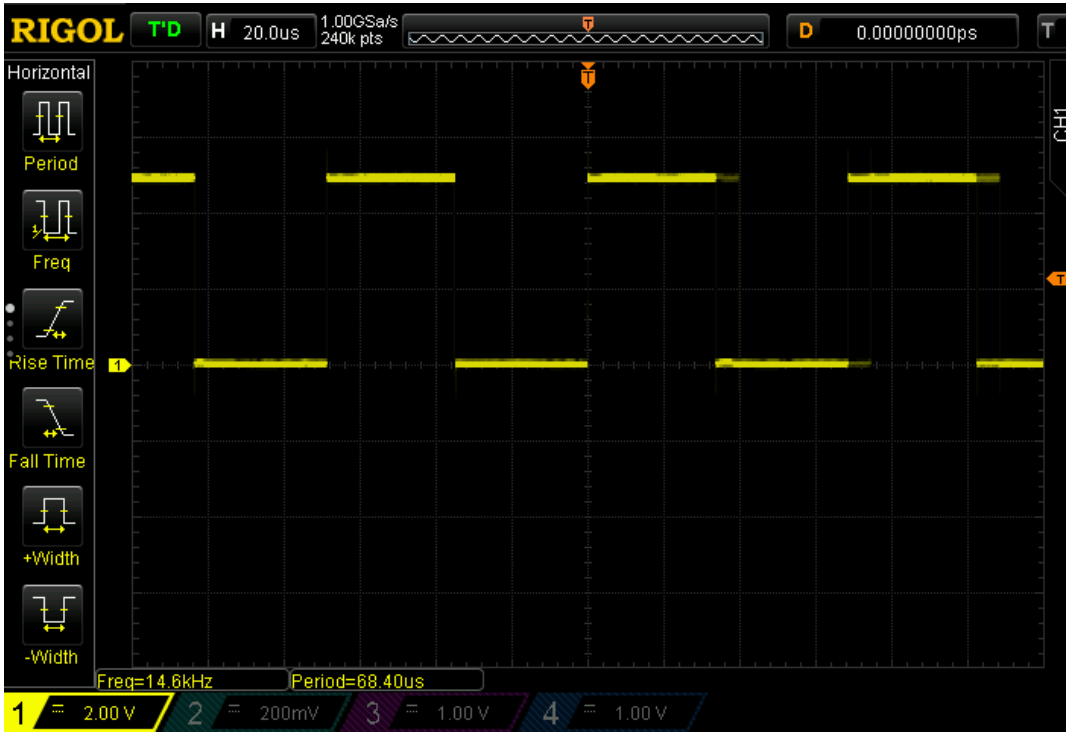



Figura 2 – Representação da evolução do nível de tensão de um dos pinos do programa. (Frequência de 14,6 kHz.)

# Exemplo

- Programa 4 – Alterar o estado dos pinos 2 até 7 (PORTD) à frequência de 50 KHz.

```
void setup() {
   for (int i=2; i<8; i++)
   {
   pinMode(i, OUTPUT);
   }
}
void loop() {
  for (int i=2; i<8; i++)
   {
   digitalWrite(i, HIGH);
   }
  delayMicroseconds(10);
  for (int i=2; i<8; i++)
   {
   digitalWrite(i, LOW);
   }
  delayMicroseconds(10);
}
```

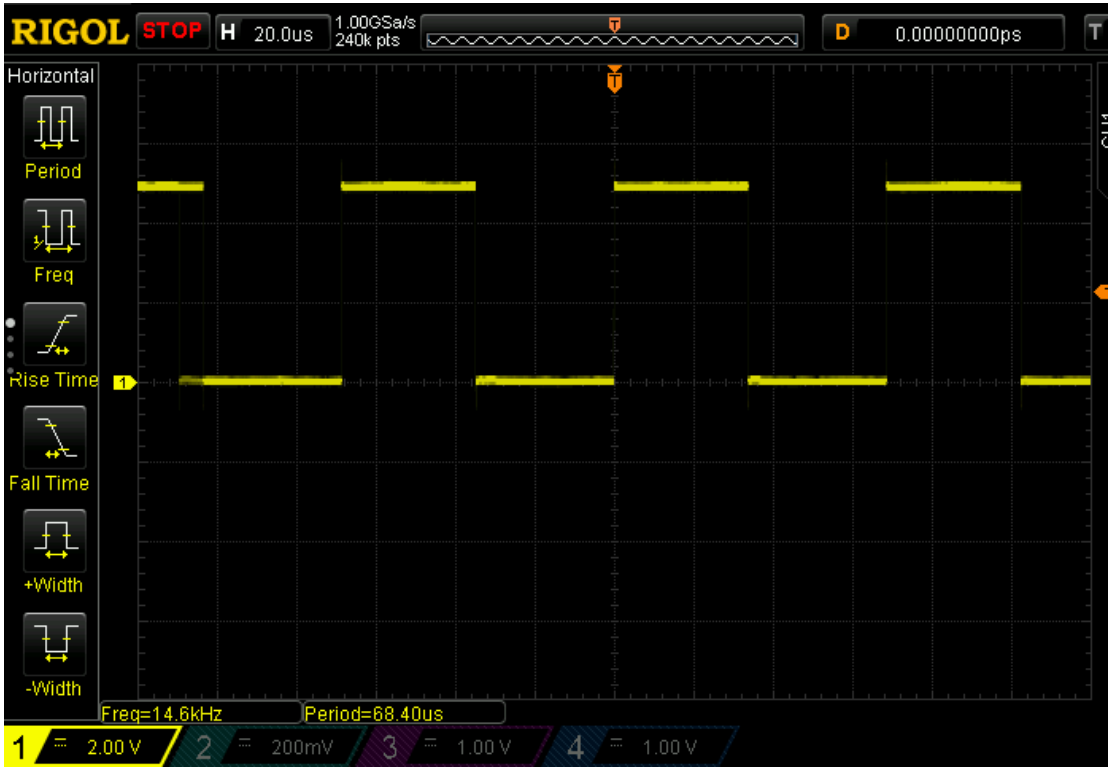



Figura 3 – Representação da evolução do nível de tensão de um dos pinos do programa. (Frequência de 14,6 kHz.)

**Conclusão: Utilizando as funções pré-definidas ( pinMode() e digitalWrite() ) não é possível atingir a frequência pedida!!**

# Exemplo

- Programa 5 – Alterar o estado dos pinos 2 até 7 (PORTD) à frequência de 50 KHz. (Utilizando os Registos)

```
void setup () {
DDRD = B11111100;
// pinos 2 até 7 como saída
}
void loop () {
PORTD = B11111100;
// pinos 2 até 7 como "1"
delayMicroseconds(10);
PORTD = B00000000;
// pinos 2 até 7 como "0"
delayMicroseconds(10);
}
```

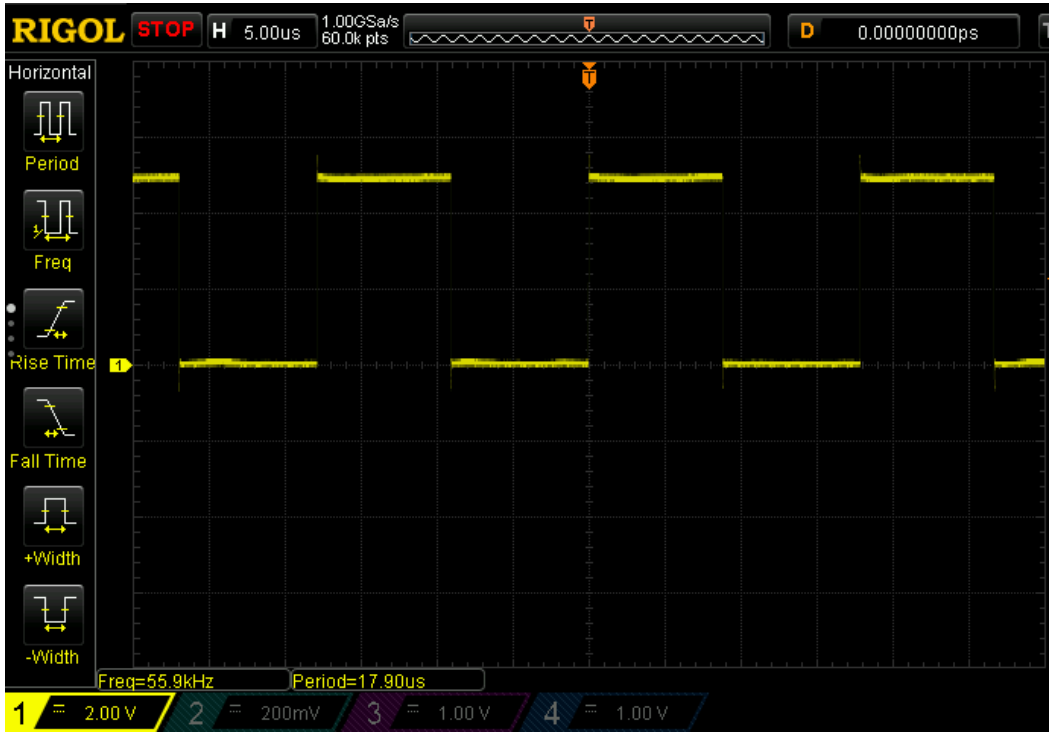



Figura 4 – Representação da evolução do nível de tensão de um dos pinos do programa. (Frequência de 50 kHz.)

**Conclusão: Alterando diretamente o valor dos registos o programa é executado mais rapidamente.**

# Exemplo

- Programa 6 – Alterar o estado dos pinos 2 até 7 (PORTD) à frequência de 50 KHz. (Utilizando os Registos e as funções “OU” e “AND” bit a bit)

```
void setup () {
DDRD = DDRD | B11111100;
// PORTA OR - pinos 2 até 7 como saída
}

void loop () {
PORTD = PORTD | B11111100;
// PORTA OR - pinos 2 até 7 como "1"
delayMicroseconds(10);
PORTD = PORTD & B00000000;
// PORTA AND - pinos 2 até 7 como "0"
delayMicroseconds(10);
}
```

# Definição de Registo bit a bit

- Pode definir-se um bit específico sem escrever todos os 8 bits do registo.
- No programa exemplo o registo DDRB para o pino 13 é colocado em modo de saída com:

  DDRB | = (1 << DDB5); (set "1"), onde DDB5 é o bit correspondente para o PIN 13.
- Para PORTB será: PORTB | = (1 << PORTB5) (set "1"),
- Para limpar um bit ou "definir para zero" no programa, usamos PORTB & = ~ (1 << PORTB5) (reset "0").

```
void setup() {
DDRB |= (1<<DDB5);
}

void loop() {
PORTB |=(1<< PORTB5);
delay(1000);
PORTB &= ~(1<<PORTB5);
delay(1000);
}
```

# Registo PIN – Leitura de pinos

- De seguida vai analisar-se  a realização da leitura de um pino  utilizando o registo PIN, em detrimento da função digitalRead(). Pretende-se realizar um programa que implica a existência de um botão no pino 8, que ao ser pressionado fará o LED do pino 13 ser desligado.
- Vai definir-se o pino 8 como entrada. Para isso DDRB & = ~ (1 << DDB0). Escrevendo todos os bits do registo, temos DDRB = B11111110; os pinos 9 a 13 são saídas e entrada do pino 8 (zero).
- Para ler se o pino 8 está no estado alto “1” ou baixo “0”, quando pressionamos o botão, o bit do registro PINB correspondente ao pino 8 (primeiro da direita) torna-se “1”.
- Realizando a operação “&” com PINB e B00000001, o resultado será diferente de zero se o botão for pressionado (com o registo PINB com o valor B00000001). Se for lido o resultado utilizando o Serial.println ((PINB & B00000001)); aparecerá “1” (2^0)

# Programa para Leitura de Pinos

- Programa 7 – Ao pressionar um botão no pino 8, fará desligar o LED do pino 13.

```
void setup()
{
//DDRB |= (1<<DDB5) ;//pino 13 como saída
//DDRB&=~(1<<DDB0) ; //pino 8 como entrada
DDRB = B11111110; //Registo de configuração
Serial.begin(9600);
}
void loop()
{
Serial.println((PINB&B00000001));
if ((PINB & B00000001)==1)
  {
 PORTB &=~ (1<<PORTB5) ;//pino 13 “0”
}
else{
    PORTB |=(1<< PORTB5) ;//pino 13 “1”
    }
}
```

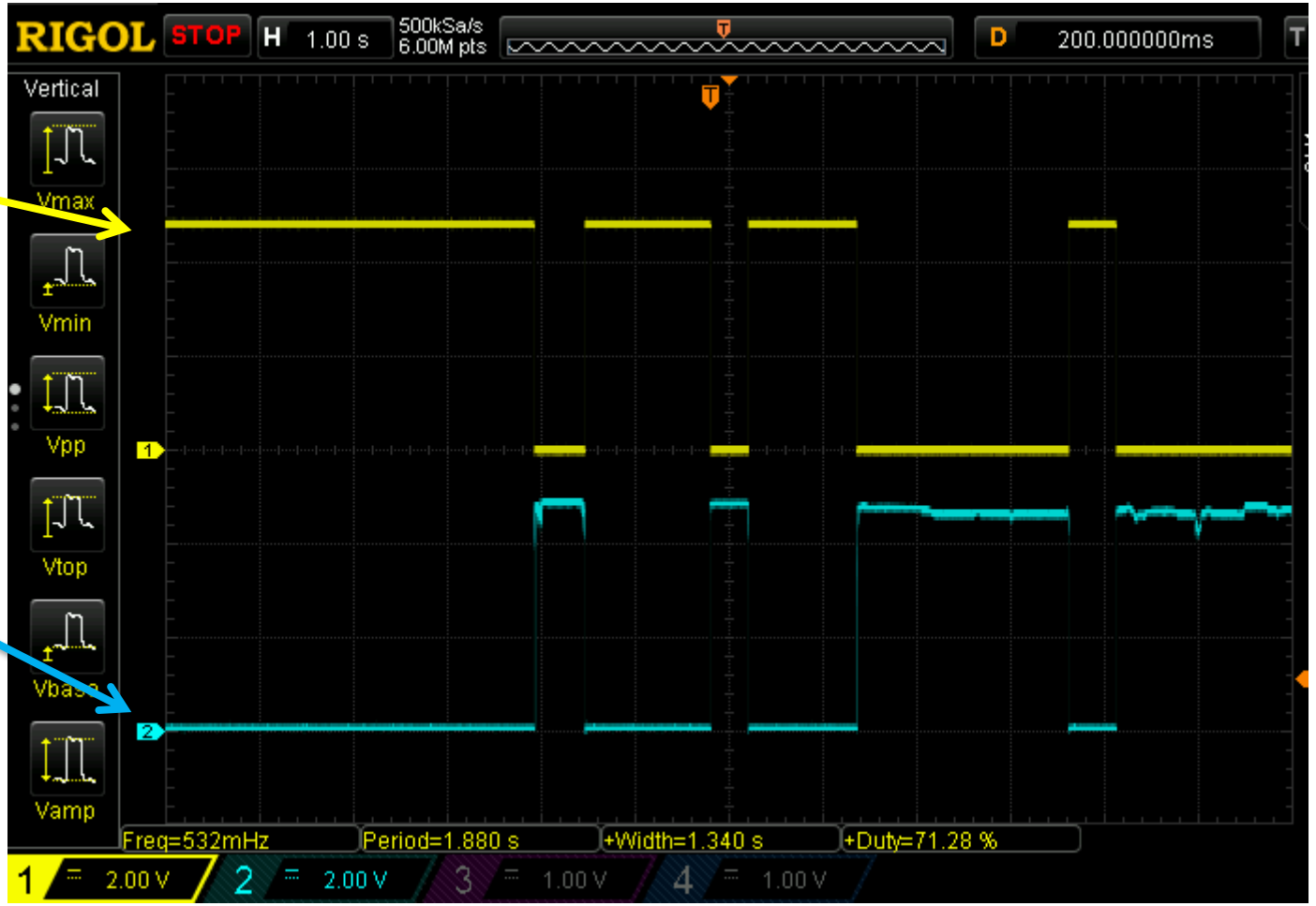
```
void setup()
{
pinMode (8, INPUT);
pinMode (13, OUTPUT);
Serial.begin(9600);
}

void loop()
{
Serial.println(digitalRead(8));
if (digitalRead(8)==1)
  {
  digitalWrite(13,LOW);
  }
else{
     digitalWrite(13,HIGH);
    }
}
```

# Programa para Leitura de Pinos

- Resultado do programa 7 – Ao pressionar um botão no pino 8, fará desligar o LED do pino 13.

**Pino 13 - Saída**

**Pino 8 - Entrada**



# Desvantagens da utilização de Registos

Aspetos negativos da utilização direta dos registos:

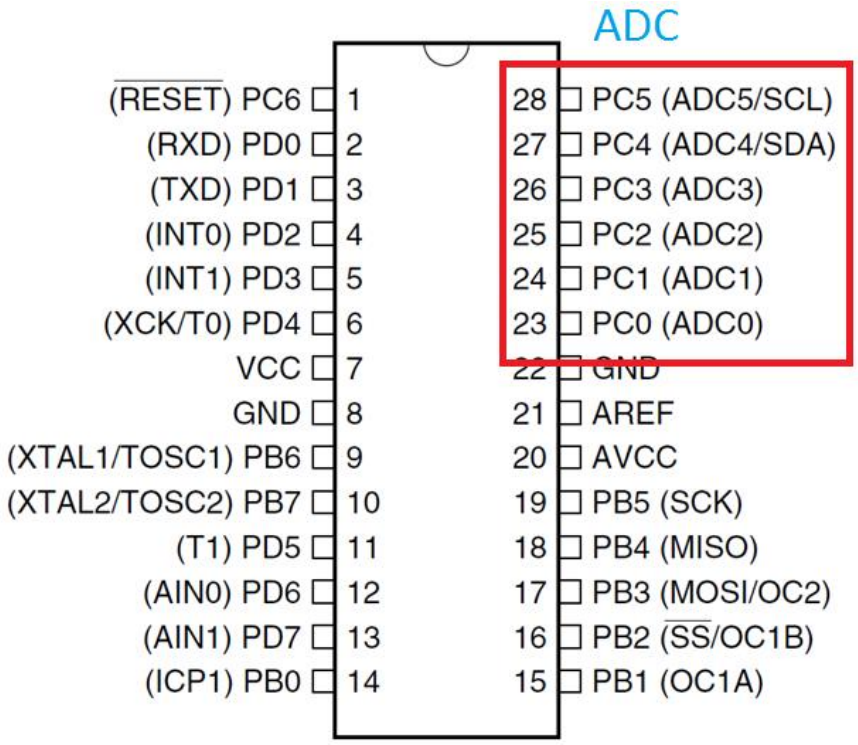
- O código é muito mais difícil para verificar e é muito mais difícil para outras pessoas o entenderem.
- O código é menos portátil. Se forem utilizadas as funções digitalRead () e digitalWrite (), é muito mais fácil escrever código que será executado em todos os microcontroladores Atmel, enquanto os registros de controle e dos portos podem ser diferentes.
- É muito mais fácil causar mau funcionamento não intencional. Observe por exemplo DDRD = B11111110; O pino 0 deve ser deixado como um pino de entrada. O pino 0 é a linha de recepção (RX) na porta série. Seria muito fácil acidentalmente fazer com que a porta série parasse de funcionar mudando o pino 0 para um pino de saída!

# UFCD 6072 – Microcontroladores

## Entradas Analógicas

# Entradas Analógicas

- Um ADC, ou conversor analógico para digital, permite converter uma tensão analógica em um valor digital que pode ser usado por um microcontrolador.
- Existem muitas fontes de sinais analógicos como sensores que medem temperatura, intensidade de luz, distância, posição e força, sendo necessário traduzir digitalmente os sinais obtidos.
- O microcontrolador ATMEGA 328 possui 6 conversores ADC de 10 bits. (Entradas A0 a A5) e utiliza 4 registros chamados ADMUX, ADCSRA, ADCL e ADCH.



# Entradas Analógicas – Registo ADMUX

- O ADMUX ("Registro de seleção do multiplexador ADC") controla a tensão de referência, a apresentação da conversão ADC (ajuste à esquerda ou ajuste à direita) e a seleção do canal analógico.

**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 | |

- Os bits REFS0 e REFS1 definem a tensão de referência da conversão. Os significados das combinações de bits são:

**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 |

Como possibilidades temos a de usar uma tensão AVCC (5V); Usar uma tensão externa a aplicar no pino AREF; Usar uma tensão gerada internamente de 1,1V.

# Entradas Analógicas – Registo ADMUX

- O estado do bit ADLAR determina a forma como o valor digital da conversão (de 10 bits) vai ser colocado no par de registos onde é guardado o resultado ADCH e ADCL (num total de 16 bits).

- Por exemplo ao definir o bit ADLAR (ADLAR="1") os 10 bits serão encostados à esquerda fazendo com que o registo ADCH guarde os 8 bits mais significativos.

- "O valor pré-definido do bit ADLAR é "0".

**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 | |

# Entradas Analógicas – Registo ADMUX

- No registo ADMUX, os bits MUX3..0 permitem escolher qual o pino que fornece a tensão a converter de acordo com a tabela:

- Note-se que no ATMEGA328P presente na placa Arduino existem apenas 6 canais externos (ADC0 a ADC5) mas outros encapsulamentos permitem ainda dois canais extra (ADC6 e ADC7).

**Table 23-4.** Input Channel Selections

| MUX3...0 | Single Ended Input |
|---|---|
| 0000 | ADC0 |
| 0001 | ADC1 |
| 0010 | ADC2 |
| ... | ... |
| 0111 | ADC7 |
| 1000 | ADC8[1] |
| 1001 | (reserved) |
| ... | ... |
| 1101 | (reserved) |
| 1110 | 1.1V ($V_{BG}$) |
| 1111 | 0V (GND) |

(1) For Temperature Sensor.

- Nos canais 14 e 15 teremos respetivamente as tensões 1,1V e 0V. O canal ADC8 é um canal interno onde existe uma tensão proporcional à temperatura, e cujos valores típicos são:

**Table 23-2.** Temperature vs. Sensor Output Voltage (Typical Case)

| Temperature / °C | -45°C | +25°C | +85°C |
|---|---|---|---|
| Voltage / mV | 242 mV | 314 mV | 380 mV |

# Entradas Analógicas – Registo ADCSRA

- No registo ADCSRA termos que ativar o conversor (bit ADEN) para dar início à conversão, e escolher nos bits ADPSx o fator de divisão do sinal de relógio do ADC ($f_{ADC}$) em relação ao sinal de relógio de 16 MHz.

### 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 | |

- Na realidade o conversor, tal como a tabela do *datasheet* mostra, não pode funcionar com uma frequência $f_{ADC}$ superior a 1MHz, pelo que o fator tem que ser de pelo menos 16.

### 28.8 ADC Characteristics

**Table 28-16.** ADC Characteristics

| Symbol | Parameter | Condition | Min. | Typ | Max | Units |
|---|---|---|---|---|---|---|
| | Resolution | | | 10 | | Bits |
| | Conversion Time | Free Running Conversion | 13 | | 260 | µs |
| | Clock Frequency | | 50 | | 1000 | kHz |

# Entradas Analógicas – Registo ADCSRA

- A tabela seguinte mostra os valores do fator de divisão do sinal de relógio do ADC, em função dos bits ADPSx.

**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 |

# Entradas Analógicas – Registo ADCSRA

Modo de realização da conversão:

- Escolher a tensão de referência REFSx;
- Definição do canal pretendido MUXx;
- Definição do factor de Divisão ADPSx;
- Ativação do bit ADSC do registo ADCSRA que inicia a conversão;
- Esperar o término da conversão (algo que pode ser verificado pela leitura da flag ADIF no mesmo registo);
- Realizar a “limpeza” da flag ADIF;
- Leitura resultado (no caso do registo ADCH ou ADCL), resultando na prática uma conversão de apenas 8 bits ou do registo ADC para a leitura dos 10 bits.

# Exemplo - Entradas Analógicas

- Programa 5 - Leitura e conversão de uma tensão aplicada em A5, utilizando os registos ADMUX, ADCSRA e ADC.

```
void setup()
 {
 Serial.begin(9600);
 }
void loop() {
 //ADC Multiplexer Selection Register
 ADMUX = 0;
 ADMUX |= (0 << REFS1);  //Tensão de referência escolhida
 ADMUX |= (1 << REFS0);  // Tensão de referência escolhida
 ADMUX |= (0 << MUX3);  // Canal escolhido A5
 ADMUX |= (1 << MUX2);  // Canal escolhido A5
 ADMUX |= (0 << MUX1);  // Canal escolhido A5
 ADMUX |= (1 << MUX0);  // Canal escolhido A5
 ADCSRA = 0;
 ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
   // habilitação da Conversão e do Prescaler para 128 111
 ADCSRA |= (1<<ADSC);         // Início da conversão
 while (!(ADCSRA & (1<<ADIF)));
 // Espera pela realização da conversão (flag ADIF = 1)
 ADCSRA |= (1<<ADIF);         // Limpar a flag ADIF
 Serial.println(ADC);       // Mostrar o valor do registo ADC
 }
```

- Programa 5 - Leitura e conversão de uma tensão aplicada em A5, utilizando a função analogRead();

```
void setup() {
  Serial.begin(9600);
}

void loop() {
 analogRead(A5);
 Serial.println(ADC);
}
```

# Entradas Analógicas

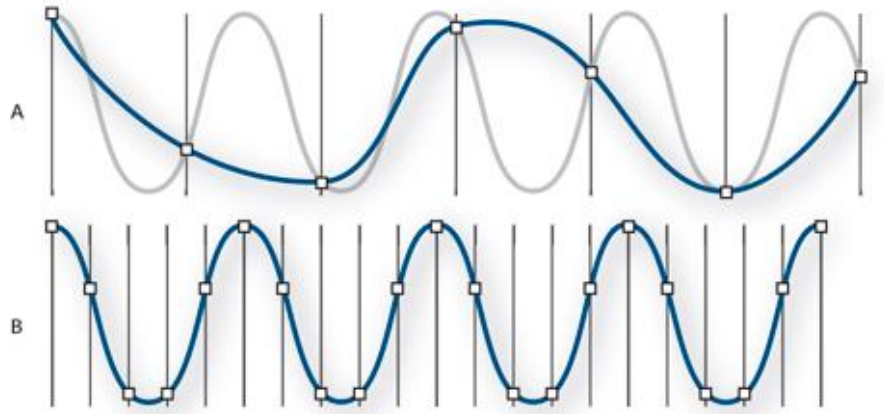
- Pode ainda recorrer-se a uma conversão com início automático. Neste modo, que estará ativo quando o bit ADATE do registo ADCSRA for igual a 1, o "disparo" (*trigger*) do processo de conversão é controlado por um evento de entre os constantes na tabela.

**Table 23-6.** 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 |

# Características - Entradas Analógicas

- Amostras são os valores de um sinal analógico medidos em um determinado instante.
- Taxa de amostragem é a quantidade de amostras de um sinal analógico recebidas num determinado tempo, para conversão em um sinal digital. (Unidade Hz).
- Quanto maior for a taxa de amostragem, mais medidas do sinal serão realizadas em um mesmo intervalo de tempo, e assim, maior será a fidelidade do sinal digital em relação ao sinal analógico.



# Características - Entradas Analógicas

- A resolução de um conversor de digital para analógico resulta diretamente do número de bits que o conversor utiliza.
- Permite perceber qual é o valor mínimo que o conversor pode representar.
- Por exemplo um conversor com 10 bits permite obter 2^10 valores, ou seja 1024 valores de saída distintos o que resulta numa resolução de aproximadamente 0, 1%. No mercado estão disponíveis conversores de digital para analógico de 6 a 24 bits.
- O conversor ADC do Atemega328 possui 10 bits de resolução, ou seja, os valores entre 0 e Vref serão convertidos entre 0 e 1023, seguindo:

$$ADC = \frac{V_{IN} \cdot 1024}{V_{REF}}$$

# UFCD 6072 – Microcontroladores

## Temporizadores e Interrupções

# Temporizadores e Interrupções

- Para implementar uma temporização de forma paralela e autónoma à execução do programa principal, que será depois interrompido quando do término da mesma é necessário recorrer ao uso de funcionalidades que na generalidade estão presente em todos os microcontroladores, são elas os temporizadores (timers) e as interrupções (interrupts)
- No que diz respeito aos temporizadores o microcontrolador ATmega328P possui 3 timers (timer 0, timer 1 e timer 2), que mais não são do que contadores que irão ser incrementados a um determinado ritmo, sendo que o timer 1 tem 16 bits, e os dois restantes são de 8 bits.

- Para efeitos de simplificação de análise será abordado em especial do timer 0, sendo que os outros timers tem funcionalidades semelhantes.

# TIMER 0 – Registos

- No caso do timer 0, o valor da contagem está presente no registo designado TCNT0, podendo ser lido e/ou escrito.

- O timer 0 pode ser colocado em um de vários modos de funcionamento, descritos na tabela.

**14.9.3 TCNT0 – Timer/Counter Register**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x26 (0x46) | TCNT0[7:0] | | | | | | | | TCNT0 |
| 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 | |

**Table 14-8.** Waveform Generation Mode Bit Description

| Mode | WGM02 | WGM01 | WGM00 | Timer/Counter Mode of Operation | TOP | Update of OCRx at | TOV Flag Set on(1)(2) |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Normal | 0xFF | Immediate | MAX |
| 1 | 0 | 0 | 1 | PWM, Phase Correct | 0xFF | TOP | BOTTOM |
| 2 | 0 | 1 | 0 | CTC | OCRA | Immediate | MAX |
| 3 | 0 | 1 | 1 | Fast PWM | 0xFF | BOTTOM | MAX |
| 4 | 1 | 0 | 0 | Reserved | – | – | – |
| 5 | 1 | 0 | 1 | PWM, Phase Correct | OCRA | TOP | BOTTOM |
| 6 | 1 | 1 | 0 | Reserved | – | – | – |
| 7 | 1 | 1 | 1 | Fast PWM | OCRA | BOTTOM | TOP |

Notes: 1. MAX = 0xFF
2. BOTTOM = 0x00

# TIMER 0 – Registos

- A escolha do modo de funcionamento faz-se atribuindo o valor adequado aos bits WGM02, WGM01 e WGM00, este dois últimos presentes no registo TCCR0A:

**14.9.1 TCCR0A – Timer/Counter Control Register A**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x24 (0x44) | COM0A1 | COM0A0 | COM0B1 | COM0B0 | – | – | WGM01 | WGM00 | TCCR0A |
| Read/Write | R/W | R/W | R/W | R/W | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- Por exemplo podem ser utilizados entre outros:
  - O modo 0 (Normal) - O contador gera um evento quando ocorre overflow (passagem entre o valor máximo 255 e o zero).
  - O modo 2 (CTC) - O contador gera um evento quando a contagem iguala o valor presente num outro registo (OCR0A). Quando isso acontece o valor de contagem é reiniciado em zero, e daí a sigla (Clear Timer on Compare).

# TIMER 0 – Registos

- Em qualquer dos modos o contador é sempre incrementado periodicamente.

**14.9.2 TCCR0B – Timer/Counter Control Register B**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x25 (0x45) | FOC0A | FOC0B | – | – | WGM02 | CS02 | CS01 | CS00 | TCCR0B |
| Read/Write | W | W | R | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- Esse período pode ser o de relógio (62,5us para a frequência de 16MHz) ou um múltiplo designado como o valor de *prescaler*. Este valor é definido pelo conjuntos dos bits CS02/CS01/CS00 presentes no registo TCCR0B.

**Table 14-9.** Clock Select Bit Description

| CS02 | CS01 | CS00 | Description |
|---|---|---|---|
| 0 | 0 | 0 | No clock source (Timer/Counter stopped) |
| 0 | 0 | 1 | $clk_{I/O}$/(No prescaling) |
| 0 | 1 | 0 | $clk_{I/O}$/8 (From prescaler) |
| 0 | 1 | 1 | $clk_{I/O}$/64 (From prescaler) |
| 1 | 0 | 0 | $clk_{I/O}$/256 (From prescaler) |
| 1 | 0 | 1 | $clk_{I/O}$/1024 (From prescaler) |
| 1 | 1 | 0 | External clock source on T0 pin. Clock on falling edge. |
| 1 | 1 | 1 | External clock source on T0 pin. Clock on rising edge. |

# TIMER 0 – Registos

- Através do registo TCCR0B é possível selecionar o valor do *prescaler*, parar por completo o timer ou fazer com que os incrementos se façam quando da descida ou subida do pino T0 do ATmega328P. (Neste caso o timer 0 irá comportar-se como um contador de eventos gerados nesse pino em detrimento de um temporizador.

- A título exemplificativo se for colocado no registo TCNT0 o valor 6 (isto é 250 contagens), e for escolhido o *prescaler* 64 será gerado um evento de *overflow* ao fim de 1ms (250x64x62,5us). Uma função de inicialização do timer 0 nestas condições será:

```
void set_timer0(void)
 {
   TCCR0A = (0<<WGM00);      // Mode 0 (Normal)
   TCCR0B = (0b011 << CS00); // Seleção de prescaler 64
   TCNT0 = 256-250;          // 1ms delay (250x64/16MHz)
}
```

# TIMER 0 – Interrupção por *overflow*

- Desta forma a temporização de 1ms é realizada de forma paralela e autónoma à execução do programa. No fim será gerado o evento de overflow que pode ser verificado lendo o bit TOV0 do registo TIFR0:

**14.9.7 TIFR0 – Timer/Counter 0 Interrupt Flag Register**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x15 (0x35) | – | – | – | – | – | OCF0B | OCF0A | TOV0 | TIFR0 |
| Read/Write | R | R | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- De forma a que não seja necessário estar constantemente a verificar esse bit o ideal seria que no momento em que o evento de *overflow* ocorrer, o programa interrompesse o seu funcionamento e uma função especial fosse chamada.
- Isto pode ser feito tirando partido do mecanismo das interrupções (*interrupts*).

# TIMER 0 – Registos

- Para tal será necessário ativar as interrupções em geral (o que é realizado pela função sei() ou interrupts() ), e também ativar o bit TOIE0 registo TIMK0 que permite que a interrupção ocorra para o evento de *overflow* do timer 0 em particular. A instrução poderia ter a forma: TIMSK0 = (1<<TOIE0);

**14.9.7 TIFR0 – Timer/Counter 0 Interrupt Flag Register**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x15 (0x35) | – | – | – | – | – | OCF0B | OCF0A | TOV0 | TIFR0 |
| Read/Write | R | R | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- Resta definir a função de interrupção que apresenta no seu cabeçalho a sigla ISR (Interrupt Service Rotine) seguido do evento que a invoca (neste caso evento de overflow do timer 0: TIMER0_OVF_vect).
- As possibilidades existentes estão descritas na biblioteca de rotinas de interrupção.

# Determinação do valor do registo TCNTx

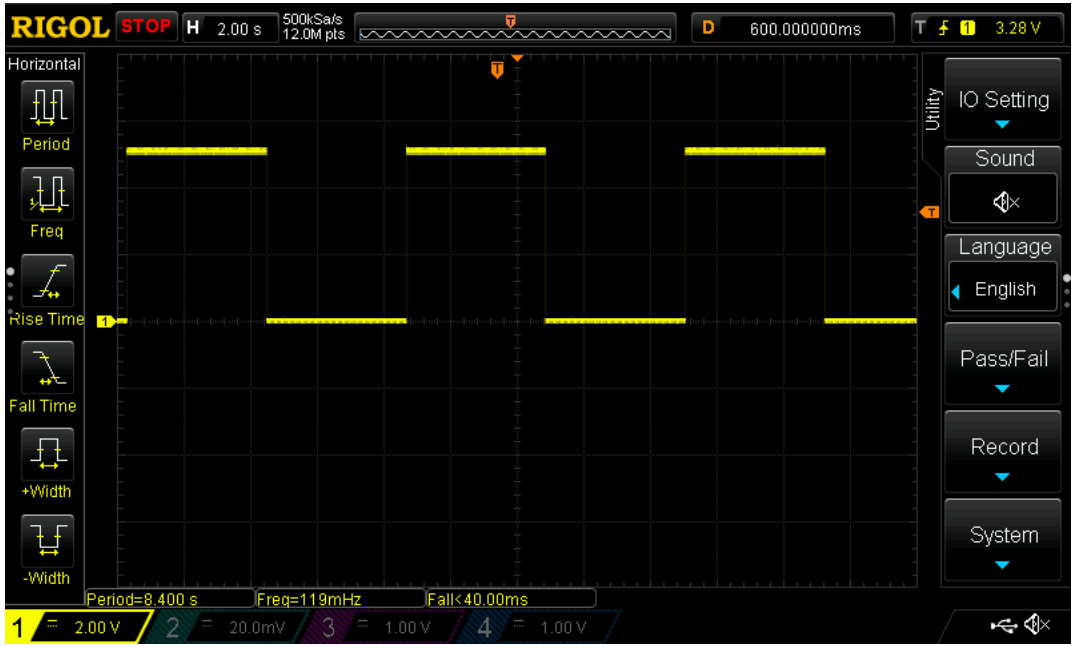
Definição do valor do inicial do TIMER (TCNTx):

- Será necessário saber a frequência da CPU. ( 16 MHz para o Arduino)
- Dividindo a frequência da CPU pelo prescaler escolhido por exemplo 1024.

  (16000000/1024= 15625 Hz). Ou seja o registo é incrementado a cada 1/15625 segundos.
- Sabendo o valor máximo do contador do temporizador (256 para 8 bits (TIMER0), 65536 para o temporizador de 16 bits (TIMER1), sabe-se o valor máximo de tempo da contagem, no exemplo dado é de aproximadamente 4,2 segundos. (65526 x 1/15625)
- Resta calcular quantos incrementos do registo TCNTx são necessários para obter a frequência pretendida.

# Resultados Obtidos

```
#define ledPin 13
void setup()
{
  pinMode(ledPin, OUTPUT);
  // inicializar timer1
  noInterrupts();          // desabilitar interrupção
  TCCR1A = 0;
  TCCR1B = 0;
  TCNT1 = 0;         // valor inicial do registo
  TCCR1B = (1<<CS12) | (0<<CS11)| (1<<CS10);
        //  prescaler 1024
  TIMSK1 |= (1 << TOIE1);
// habilitar interrupção por overflow TIMER1
  interrupts();           // habilitar interrupção
}
ISR(TIMER1_OVF_vect)
// evento que acontece com overflow TIMER1
{
  TCNT1 = 0;//;          // carregar novamente valor do registo
  digitalWrite(ledPin, !digitalRead(ledPin));
//inverter o estado do pino 13
}
void loop()
{
  // NADA!!
}
```




DADOS:

TCNT1=0;

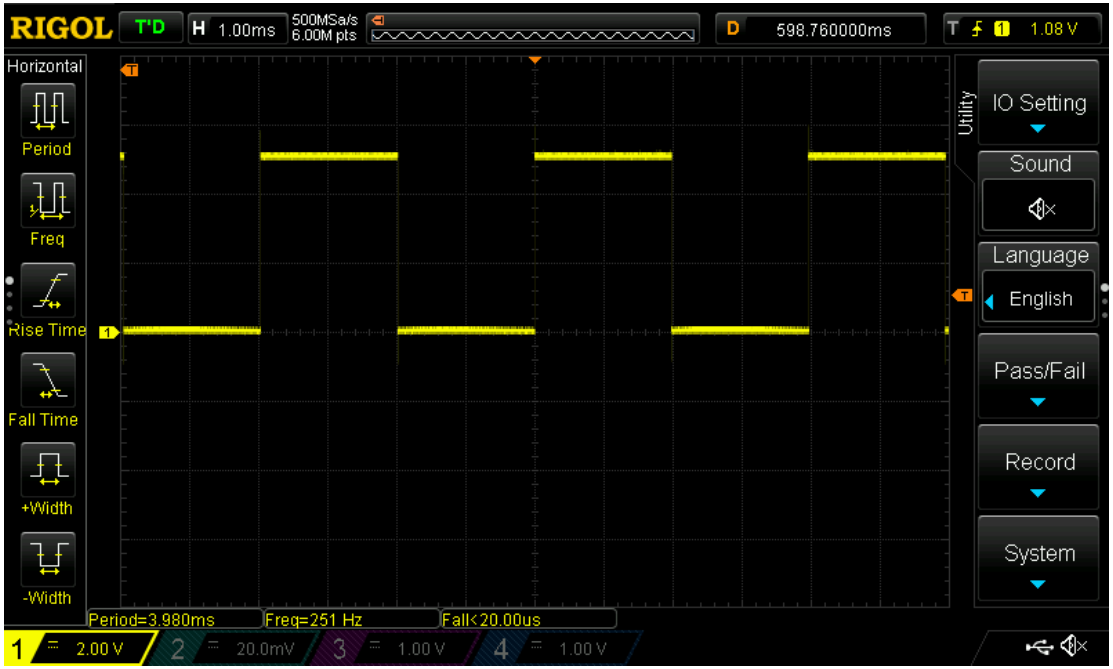
Prescaler 1024;

Interrupção por overflow;

Tempo de overflow aproximado de 4,2 s

# Resultados Obtidos

```
#define ledPin 13
void setup()
{
  pinMode(ledPin, OUTPUT);
   noInterrupts();          // desabilitar interrupção
  TCCR1A = 0;
  TCCR1B = 0;
  TCNT1 =65536-31;          // valor inicial do registo
  TCCR1B = (1<<CS12) | (0<<CS11)| (1<<CS10);
//  prescaler 1024
  TIMSK1 |= (1 << TOIE1);
 // habilitar interrupção por overflow TIMER1
  interrupts();           // habilitar interrupção
}
ISR(TIMER1_OVF_vect)
// evento que acontece com overflow TIMER1
{
  TCNT1 = 65536-31;
 // carregar novamente valor do registo
  digitalWrite(ledPin, !digitalRead(ledPin)); //inverter o estado
        do pino 13
}
void loop()
{
  // NADA!!
}
```




DADOS:

TCNT1= 65536-31; (64 us x 31 incrementos)

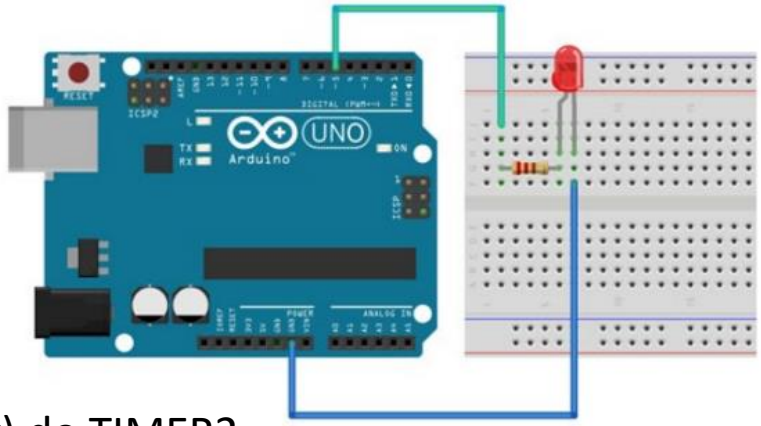
Prescaler 1024;

Interrupção por overflow;

Tempo de overflow aproximado de 1,98 ms

# Programa Exemplo

A partir dos registos descritos realize um programa que faça a variação do estado de uma saída (Pino 5), utilizando o TIMER 2. O LED deve estar piscar à frequência de 325 Hz




Conceito:

- Utilização dos registos TCCR2A e TCCR2B
- (Definição do Modo Normal e valor do Prescaler) do TIMER2;
- Inicialização da contagem do TCNT2 (No valor correspondente para a frequência de 325Hz);
- Habilitar a interrupção do TIMER2 por overflow (registo TIMSK2);
- Criar o vetor de interrupção por overflow ISR(TIMER2_OVF_vect);
- Como a interrupção por overflow voltar a colocar o valor pretendido em TCNT2;
- Realizar o *toggle* do estado do pino de saída;

# Programa Exemplo -Determinação do valor do registo TCNT2

Definição do valor do inicial do TIMER 2 (TCNT2) para a frequência de 325Hz

- Frequência da CPU de 16 MHz

- Dividindo a frequência da CPU pelo *prescaler* escolhido por exemplo 64.

  (16000000/64= 250000 Hz). Ou seja o registo é incrementado a cada 1/250000 segundos ou seja 4 us.

- Sabendo o valor máximo do contador do temporizador (256 para 8 bits (TIMER2), sabe-se o valor máximo de tempo da contagem, é de aproximadamente 1,024 ms. (256 x 4 us). Para a frequência pedida é necessário no mínimo uma contagem com duração de aproximadamente 1,53 ms (metade do período).

- Solução: Modificar o *prescaler* !!

# Programa Exemplo

Solução: Modificar o *prescaler* !!

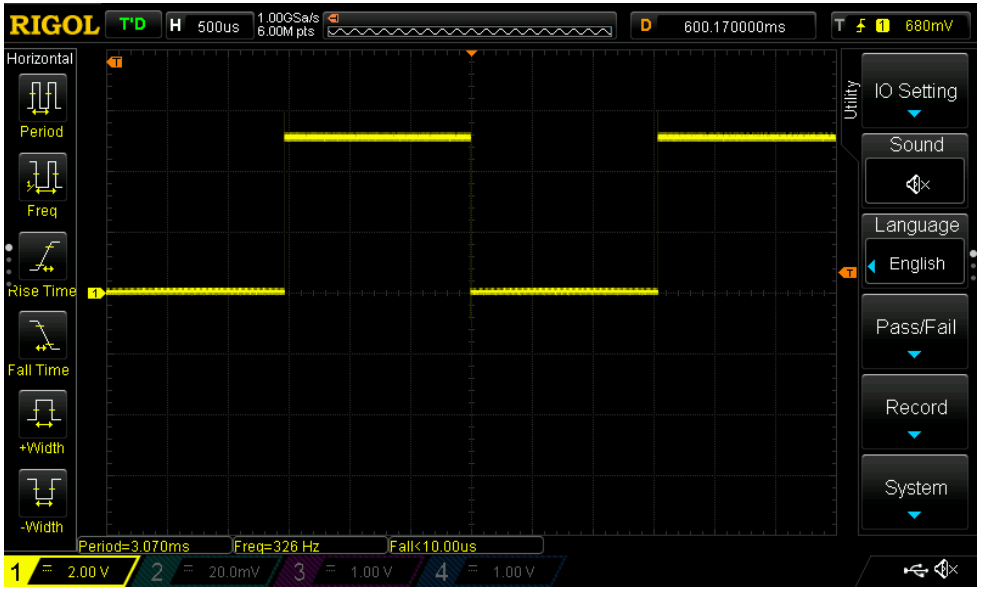
- Dividindo a frequência da CPU pelo *prescaler* escolhido por exemplo 1024.

  (16000000/1024= 15625 Hz). Ou seja o registo é incrementado a cada 1/15625 segundos ou seja 64 us.

- Sabendo o valor máximo do contador do temporizador (256 para 8 bits (TIMER2), sabe-se o valor máximo de tempo da contagem, é de aproximadamente 16384 ms. (256 x 64 us).
- Para a frequência pedida é necessário uma contagem com duração de aproximadamente 1,53 ms (metade do período) ou seja 24 incrementos (1,53/ 0,064 )
- Portanto o valor do registo será: TCNT2 =256-24;

# Programa Exemplo

```
#define ledPin 5
void setup()
{
 pinMode(ledPin, OUTPUT);
 noInterrupts();          // desabilitar interrupção
 TCCR2A = 0;
 TCCR2B = 0;
 TCNT2 =256-24;           // valor inicial do registo
 TCCR2B = (1<<CS22) | (1<<CS21)| (1<<CS20);
//  prescaler 1024
 TIMSK2 |= (1 << TOIE2);
// habilitar interrupção por overflow TIMER2
 interrupts();            // habilitar interrupção
}

ISR(TIMER2_OVF_vect)       // evento que acontece
com overflow TIMER2
{
 TCNT2 = 256-24;           // carregar novamente valor
do registo
 digitalWrite(ledPin, !digitalRead(ledPin)); //inverter o
estado do pino 5
}

void loop()
{
 // NADA!!
}
```




DADOS:

TCNT2= 256-24; (64 us x 24 incrementos)

Prescaler 1024;

Interrupção por overflow;

Tempo de overflow aproximado de 1,53 ms

# Modo de Funcionamento CTC

- Existe ainda outro modo de utilização dos temporizadores. No modo CTC o contador gera um evento quando a contagem iguala o valor presente no registo OCR0A para o TIMER 0.

- Neste modo de funcionamento não é necessário atualizar o registo TCNT0 pois este é colocado a zero automaticamente. A ativação da interrupção seria agora assim:

  TIMSK0 = (1<<OCIE0A);

- O vetor de interrupção e dado por comparação entre registos, que será escrito:

  ISR(TIMER0_COMPA_vect)

  Em detrimento da interrupção por *overflow* vista anteriormente (ISR(TIMER2_OVF_vect) )

# Modo de Funcionamento CTC

- Neste caso, em funcionamento no modo CTC, a função de inicialização do timer 0 seria:

```
void set_timer0(void)
{
   TCCR0A = (2<<WGM00);      // Mode 2 (CTC)
   TCCR0B = (0b011 << CS00); // Escolha de prescaler 64
   TCNT0 = 0;
   OCR0A = 250;              // Registo para comparação 1ms delay (250x64/16MHz)
}
```

- A ativação da interrupção seria agora: TIMSK0 = (1<<OCIE0A);

- A nova rotina de interrupção já não necessita da atualização do registo TCNT0. Uma possibilidade para a escrita da rotina do evento será:

```
ISR(TIMER0_COMPA_vect)
      {
            //Programa do evento
      {
```

# Programa Exemplo Modo CTC

A partir dos registos descritos realize um programa que faça a variação do estado de duas saídas sendo a saída 1 (Pino 13), utilizando o TIMER 1, modo CTC frequência 1kHz. A saída 2(Pino 12), utilizando o TIMER 2, modo Normal frequência 200Hz.

Conceito TIMER 1:

•Utilização dos registos TCCR1A e TCCR1B(Definição do Modo CTC e valor do Prescaler) do TIMER 1;

•Inicialização da contagem do TCNT2 em zero e o registo de comparação OCR1A no valor correspondente para a frequência pretendida;

•Habilitar a interrupção do TIMER1 por comparação (registo TIMSK1 bit a "1" OCIE1A);

•Criar o vetor de interrupção por comparação ISR(TIMER1_ COMPA _vect);

•Como a interrupção por comparação voltar realizar o *toggle* do estado do pino de saída;

Conceito TIMER 2:

•Utilização dos registos TCCR2A e TCCR2B (Definição do Modo Normal e valor do Prescaler) do TIMER 2;

•Inicialização da contagem do TCNT2 (No valor correspondente para a frequência pretendida);

•Habilitar a interrupção do TIMER2 por overflow (registo TIMSK2 bit a "1" TOIE2);

•Criar o vetor de interrupção por overflow ISR(TIMER2_OVF_vect);

•Como a interrupção por overflow voltar a colocar o valor pretendido em TCNT2;

•Realizar o *toggle* do estado do pino de saída;

# Programa Exemplo Modo CTC

Determinação do valor de OCR1A (TIMER 1)

- Dividindo a frequência da CPU pelo *prescaler* escolhido por exemplo 8.

  (16000000/8= 2000000 Hz). Ou seja o registo é incrementado a cada 1/2000000 segundos ou seja 5 us.

- Sabendo o valor máximo do contador do temporizador (65526 para 16 bits (TIMER1), sabe-se o valor máximo de tempo da contagem, é de aproximadamente 0,03s. (65526 x 5 us).
- Para a frequência pedida é necessário uma contagem com duração de aproximadamente 0,5 ms (metade do período) ou seja 1000 incrementos (0,5/ 0,005 )
- Portanto o valor do registo será: OCR1A = 1000;

# Programa Exemplo Modo CTC

Determinação do valor de TCNT2 (TIMER 2)

- Dividindo a frequência da CPU pelo *prescaler* escolhido por exemplo 1024.

  (16000000/1024= 15625 Hz). Ou seja o registo é incrementado a cada 1/15625 segundos ou seja 64 us.

- Sabendo o valor máximo do contador do temporizador (256 para 8 bits (TIMER2), sabe-se o valor máximo de tempo da contagem, é de aproximadamente 16384 ms. (256 x 64 us).
- Para a frequência pedida é necessário uma contagem com duração de aproximadamente 2,5 ms (metade do período) ou seja 39 incrementos (2,5/ 0,064 )
- Portanto o valor do registo será: TCNT2 =256-39;

# Programa Exemplo Modo CTC – Resultado

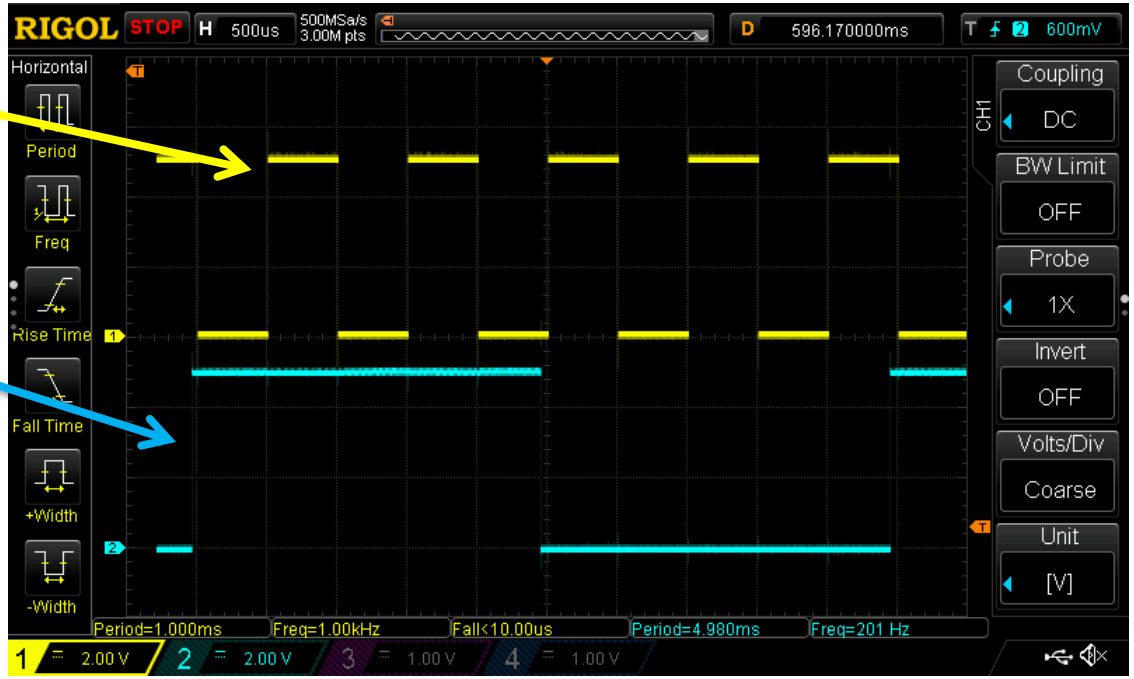
```
#define ledPin 13
#define ledPin2 12
void setup(){
  pinMode(ledPin, OUTPUT);
  pinMode(ledPin2, OUTPUT);
  noInterrupts();          // desabilitar interrupção

//inicializar timer 1
TCCR1A = 0;// set entire TCCR1A register to 0
  TCCR1B = 0;// same for TCCR1B
  TCNT1  = 0;//initialize counter value to 0;
OCR1A = 1000
TCCR1B |= (1 << WGM12);
    TCCR1B |= (1 << CS11);
    TIMSK1 |= (1 << OCIE1A);

  // inicializar timer2
  TCCR2A = 0;
  TCCR2B = 0;
  TCNT2 =256-39;          // valor inicial do registo
  TCCR2B = (1<<CS22) | (1<<CS21)| (1<<CS20);
  TIMSK2 |= (1 << TOIE2);
  interrupts();           // habilitar interrupção
}
```

```
ISR(TIMER1_COMPA_vect)// evento que acontece por
comparação OCR1A TIMER1
{
  digitalWrite(ledPin, !digitalRead(ledPin));
}

ISR(TIMER2_OVF_vect)       // evento que acontece com
overflow TIMER2
{
  TCNT2 = 256-39;            // carregar novamente valor do
registo
  digitalWrite(ledPin2, !digitalRead(ledPin2)); //inverter o
estado do pino 12
}

void loop()
{
  // NADA!!
}
```

# Programa Exemplo Modo CTC – Resultado

Pino 13 – Saída 1, utilizando o TIMER 1, modo CTC frequência 1kHz.

Pino 12 – Saída 2, utilizando o TIMER 2, modo Normal frequência 200Hz.



# UFCD 6072 – Microcontroladores

## Comunicação Série

# Comunicação Série - UART

- Uma comunicação série utiliza três tipos de dispositivos de hardware, entre elas:

  USRT - Universal Synchronous Receiver/Transmitter

  UART - Universal Asynchronous Receiver/Transmitter

  USART - Universal Synchronous/Asynchronous Receiver/Transmitter

- Numa ligação assíncrona série (UART), a comunicação faz-se através de dois fios (RX/TX) onde os bits que compõem os bytes de informação são recebidos/enviados sequencialmente.

- Os níveis de tensão que representam os “1” e os “0” são referenciados relativamente a uma terceira linha (GND). Embora possam existir mais sinais de controlo, estes são os mínimos para o estabelecimento da comunicação bidireccional.

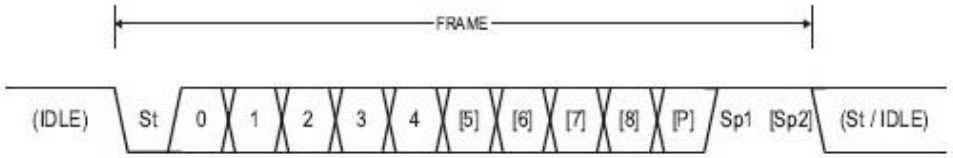
- A comunicação diz-se assíncrona pois os bits não são sincronizados com uma linha de relógio adicional, sendo contudo recebidos/enviados a uma frequência (baudrate) pré-definida.

# Comunicação Série - UART

- Na ausência de informação (idle) estão em repouso no estado lógico “1”, sendo necessário que antes do envio do primeiro bit de dados, seja enviado um "Start bit" (St) de valor lógico 0.

- Por sua vez no final da transmissão dos bits de dados (que podem variar entre 6 e 9 mas cujo valor típico é 8), é enviado um ou dois "Stop bits" (Sp1, Sp2), precedido dum eventual bit de paridade (P) para deteção de erros.

**Figure 19-4.** Frame Formats



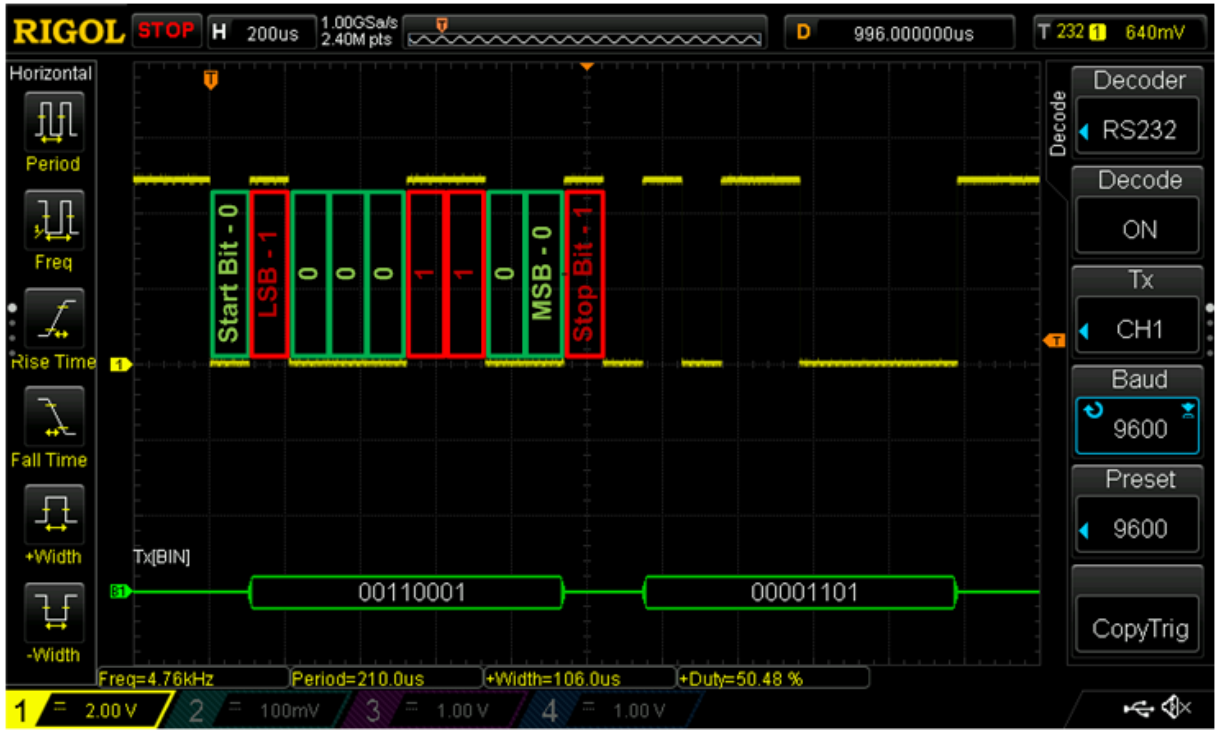

| | |
|---|---|
| **St** | Start bit, always low. |
| **(n)** | Data bits (0 to 8). |
| **P** | Parity bit. Can be odd or even. |
| **Sp** | Stop bit, always high. |
| **IDLE** | No transfers on the communication line (RxDn or TxDn). An IDLE line must be high. |

# Comunicação Série - UART

- Do ponto de vista prático o sinal obtido no pino Rx do microcontrolador pode ser representado pela figura (Baudrate = 9600 bit/s; 8 bits de dados; nº de stop bits = 1).

Programa utilizado:

```
void setup()
 {
  Serial.begin(9600);
 }

void loop() {
   char msm= Serial.read();
   Serial.println(msm);
   delay (1000);
 }
```




- A mensagem enviada corresponde a 00110001 em binário a que corresponde pela tabela ASCII ao caracter 1.

# Comunicação Série - UART

(DADOS= 01001111 Caracter O. Baudrate = 9600 bit/s; 8 bits de dados; nº de stop bits = 1).

# Comunicação Série - UART

- Como visto anteriormente a configuração da porta depende entre outros do baudrate em bit/s, do nº de bits de dados, da existência ou não de bit de paridade e a definição do número de stop bits.

- Para se definir o baudrate temos que escrever no registo UBRR um valor relacionado com a frequência de relógio (F_CPU) e o baudrate pretendido (USART_BAUD). Esse valor é dado pelas seguintes fórmulas que constam da tabela:

**Table 19-1.** Equations for Calculating Baud Rate Register Setting

| Operating Mode | Equation for Calculating Baud Rate[1] | Equation for Calculating UBRRn Value |
|---|---|---|
| Asynchronous Normal mode (U2Xn = 0) | $BAUD = \frac{f_{OSC}}{16(UBRRn+1)}$ | $UBRRn = \frac{f_{OSC}}{16BAUD} - 1$ |
| Asynchronous Double Speed mode (U2Xn = 1) | $BAUD = \frac{f_{OSC}}{8(UBRRn+1)}$ | $UBRRn = \frac{f_{OSC}}{8BAUD} - 1$ |

# Comunicação Série - UART

- Para um Baudrate de 57600 bits/s, o erro é menor caso se opte pela 2ª fórmula (onde o bit U2Xn tem que ser activado).

**Table 19-6.** Examples of UBRRn Settings f

| Baud Rate (bps) | $f_{osc}$ = 16.0000 MHz | | | |
|---|---|---|---|---|
| | U2Xn = 0 | | U2Xn = 1 | |
| | UBRRn | Error | UBRRn | Error |
| 2400 | 416 | -0.1% | 832 | 0.0% |
| 4800 | 207 | 0.2% | 416 | -0.1% |
| 9600 | 103 | 0.2% | 207 | 0.2% |
| 14.4k | 68 | 0.6% | 138 | -0.1% |
| 19.2k | 51 | 0.2% | 103 | 0.2% |
| 28.8k | 34 | -0.8% | 68 | 0.6% |
| 38.4k | 25 | 0.2% | 51 | 0.2% |
| 57.6k | 16 | 2.1% | 34 | -0.8% |
| 76.8k | 12 | 0.2% | 25 | 0.2% |
| 115.2k | 8 | -3.5% | 16 | 2.1% |

# Comunicação Série - UART

### 19.11.4 UCSRnC – USART Control and Status Register n C

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | UMSELn1 | UMSELn0 | UPMn1 | UPMn0 | USBSn | UCSZn1 | UCSZn0 | UCPOLn | UCSRnC |
| 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 | 1 | 1 | 0 | |

- Para a escolha do número de bits de dados e o número de stopbits, o bit de paridade é necessário ativar os bits apropriados no registo UCSR0C, cuja descrição detalhada pode ser vista na secção :

**Table 19-8.** UPMn Bits Settings

| UPMn1 | UPMn0 | Parity Mode |
|---|---|---|
| 0 | 0 | Disabled |
| 0 | 1 | Reserved |
| 1 | 0 | Enabled, Even Parity |
| 1 | 1 | Enabled, Odd Parity |

**Table 19-9.** USBS Bit Settings

| USBSn | Stop Bit(s) |
|---|---|
| 0 | 1-bit |
| 1 | 2-bit |

**Table 19-10.** UCSZn Bits Settings

| UCSZn2 | UCSZn1 | UCSZn0 | Character Size |
|---|---|---|---|
| 0 | 0 | 0 | 5-bit |
| 0 | 0 | 1 | 6-bit |
| 0 | 1 | 0 | 7-bit |
| 0 | 1 | 1 | 8-bit |
| 1 | 0 | 0 | Reserved |
| 1 | 0 | 1 | Reserved |
| 1 | 1 | 0 | Reserved |
| 1 | 1 | 1 | 9-bit |

# Comunicação Série - UART

- Resta a ativação dos bits que habilitam o processo de recepção/transmissão, os bits RXEN0 e TXEN0 respetivamente do registo UCSR0B, cuja descrição detalhada pode ser vista na secção:

**19.11.3 UCSRnB – USART Control and Status Register n B**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | RXCIEn | TXCIEn | UDRIEn | RXENn | TXENn | UCSZn2 | RXB8n | TXB8n | UCSRnB |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- A transmissão dum byte é feita colocando-o no registo UDR0, sendo contudo antes necessário verificar se ele está disponível, algo que pode ser constatado caso o bit UDRE0 do registo UCSR0A esteja ativo. A recepção dum byte é feita lendo o mesmo registo UDR0 quando este possui um dado válido, algo que pode ser verificado caso o bit RXC0 do registo UCSR0A esteja ativo:

**19.11.2 UCSRnA – USART Control and Status Register n A**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | RXCn | TXCn | UDREn | FEn | DORn | UPEn | U2Xn | MPCMn | UCSRnA |
| Read/Write | R | R/W | R | R | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |

# Comunicação Série - UART

- Programa utilizando as funções “Serial” pré-definidas.

```
void setup() {
  // put your setup code here, to run once:
Serial.begin(57600);
}

void loop() {
  // put your main code here, to run
repeatedly:
  if (Serial.available() > 0) {
   char inByte = Serial.read();

    Serial.println(inByte);

}
}
```

# Comunicação Série - UART

```
#define F_CPU 16000000ul
#define USART_BAUD 57600ul
#define USART_UBBR_VALUE
((F_CPU/(USART_BAUD<<3))-1)

void setup()
{
}
void init_usart(void) {
   // Set baudrate
   UBRR0H = (uint8_t)(USART_UBBR_VALUE>>8);
   UBRR0L = (uint8_t) USART_UBBR_VALUE;
   UCSR0A = (1<<U2X0);
   // Set frame format to...
   UCSR0C =
      (3<<UCSZ00) // 8 data bits
      | (1<<UPM00)  // no parity bit
      | (1<<USBS0); // 1 stop bit
   // Enable receiver and transmitter
   UCSR0B = (1<<RXEN0) | (1<<TXEN0);
}
```

```
void myputchar(char c)
{
   // Wait if UDR is not free
   while((UCSR0A & (1<<UDRE0)) == 0);

   // Transmit data
   UDR0 = c;
}

char mygetchar(void)
{
   // Wait until a byte has been received
   while((UCSR0A & (1<<RXC0)) == 0);

   // Return received data
   return UDR0;
}
```