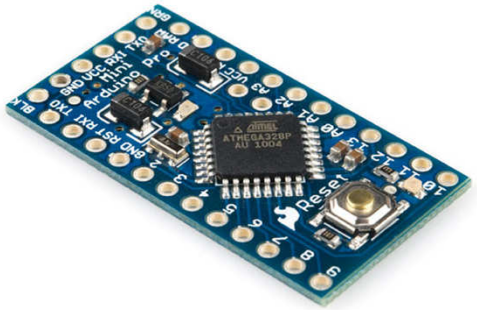
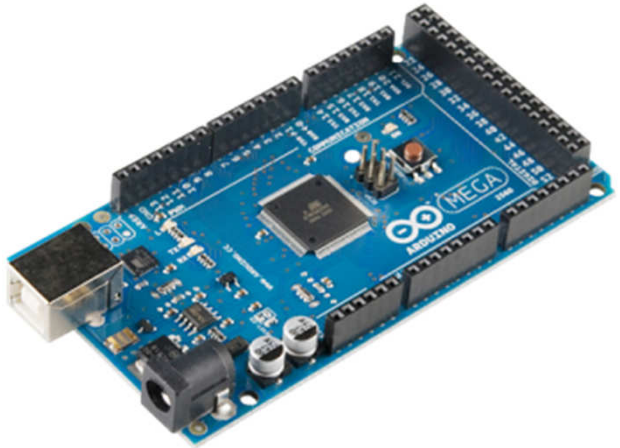
Programação C para Arduino

Prof. Charles Borges de Lima.

SUMÁRIO

INTRODUÇÃO

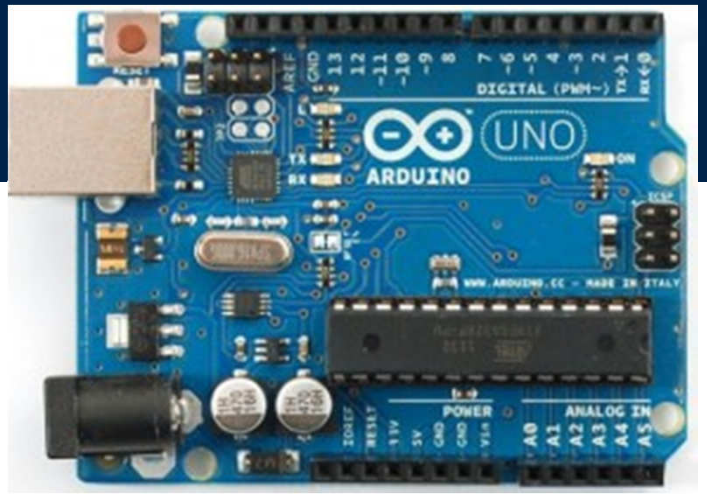
Arduino é uma plataforma eletrônica de prototipagem baseada em hardware livre e software, fáceis de usar e flexíveis.

Possui um IDE que permite a programação de forma fácil de um microcontrolador AVR. Todavia, não é eficiente e não permite o desenvolvimento profissional de projetos.

Para um projeto profissional é necessário o uso de ferramentas adequadas de programação e do conhecimento técnico do microcontrolador empregado.

A IDE do Arduino não é destinada ao desenvolvimento profissional.

O ARDUINO UNO



Como é estruturado o Hardware:

0-5 (azul) - pinos de entradas analógicas.
Entradas para o ADC, podem ser usados como I/O digital.

0-13 (verde) - pinos de I/O digitais . Pinos 0 e 1 também são utilizados para a comunicação serial.

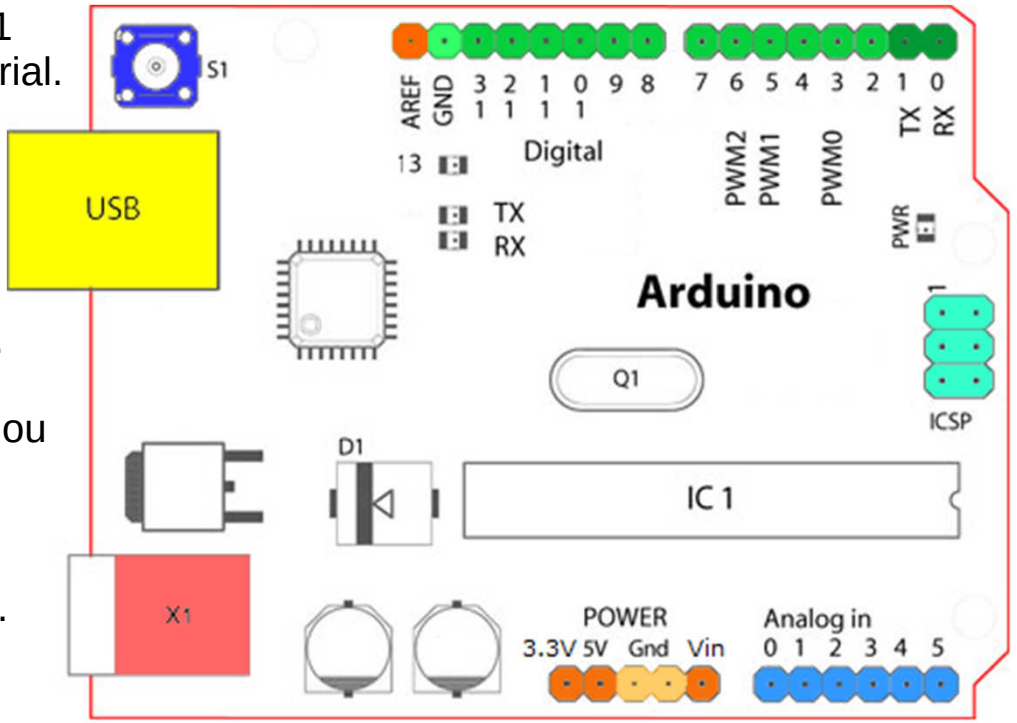
AREF(laranja) - referência analógica do ADC.

S1 (azul) - botão de inicialização.

ICSP (ciano) - conector de gravação In-Circuit.

USB (amarelo) - usado para gravar o Arduino ou energizá-lo.

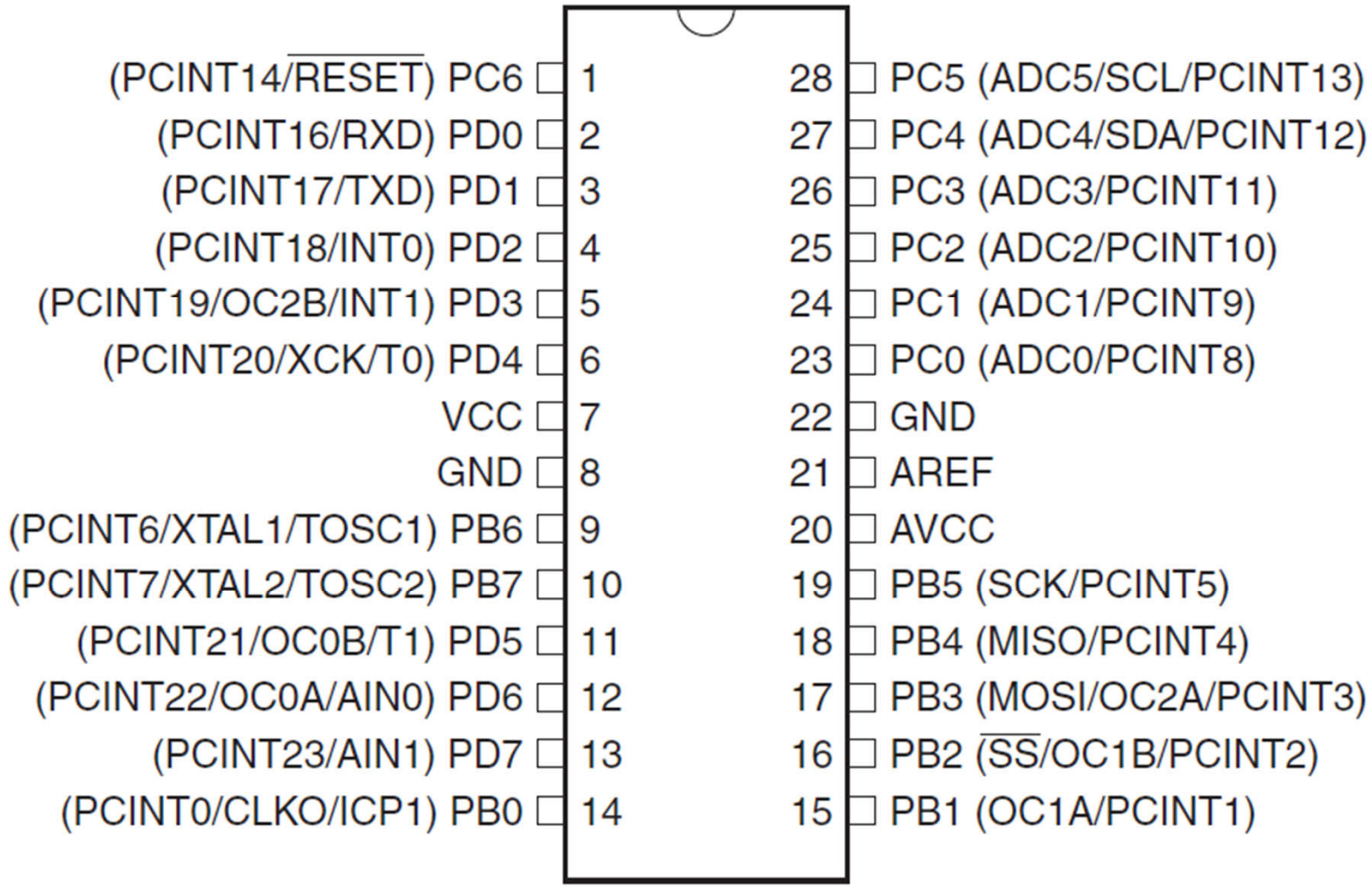
X1 (rosa) - fonte de alimentação externa (9-12VDC, após diodo estará também em Vin).



20 Pinos de I/O

Existem variações no layout da placa conforme o modelo.

O ATmega328



O microcontrolador do Arduino Uno é o ATmega328 (AVR).

PINOS DO ATmega328 NO ARDUINO

Correlação entre os pinos do Arduino e do ATmega328.

Arduino	ATmega328		Arduino	ATmega328		Arduino	Atmega328
Analog In	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			

Assembly

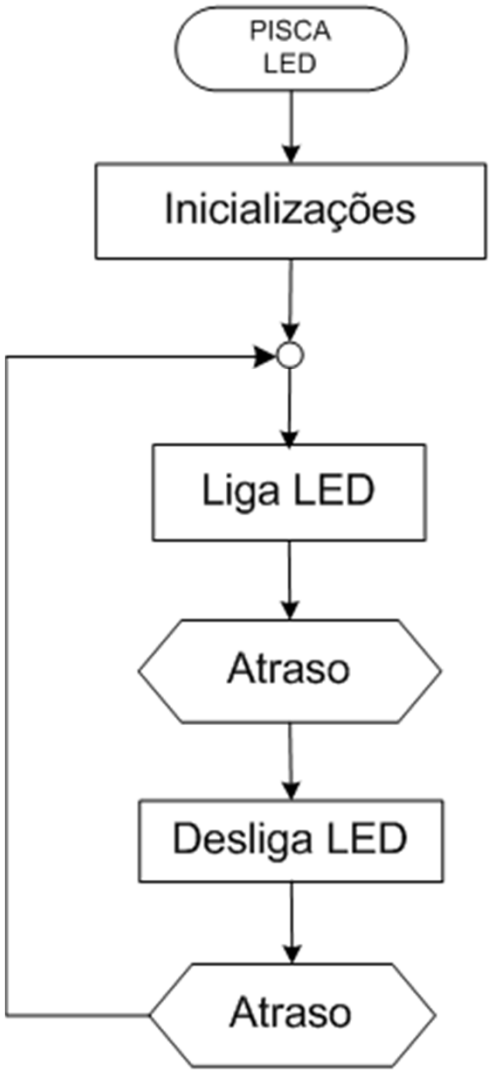
Todo microcontrolador possui um conjunto próprio de instruções representadas por mnemônicos (assembly) que após o desenvolvimento do programa são convertidos nos zeros e uns lógicos interpretáveis pelo microprocessador.

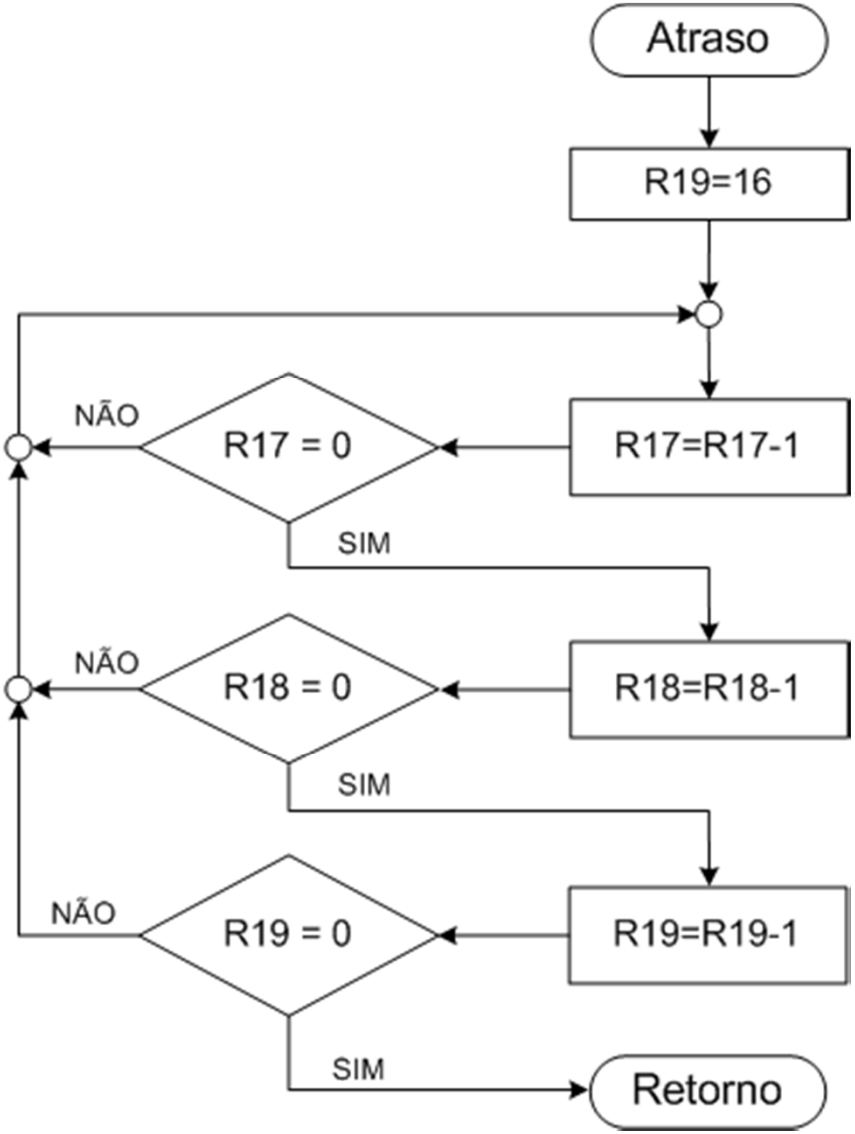
O *assembly* é uma linguagem de baixo nível e permite obter o máximo desempenho de um microcontrolador, gerando o menor número de bytes de programa combinados a uma maior velocidade de processamento.

Todavia, o *assembly* só será eficiente se o programa estiver bem estruturado e empregar algoritmos adequados.

Programar em *assembly* exige muito esforço de programação.

EXEMPLO


PISCA LED
Inicializações
Liga LED
Atraso
Desliga LED
Atraso


Atraso
R19=16
R17=R17-1
R17 = 0
NÃO
SIM
R18=R18-1
R18 = 0
NÃO
SIM
R19=R19-1
R19 = 0
NÃO
SIM
Retorno

EXEMPLO

```
.equ LED    = PB5          //LED é o substituto de PB5 na programação
.ORG 0x000                 //endereço de início 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

Piscando um LED ligado ao pino PB5 (pino 13 no Arduino).

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.

PROGRAMAÇÃO C

O problema de desenvolver o 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.

EXEMPLO

```
#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
    }
}
//-------------------------------------------------------------------------
```

216 Bytes
Otimização -Os

Usando cpl_bit(PORTB,LED) resultam 202 bytes.

IDE DO ARDUINO (Wiring)

```
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**

Linguagem própria baseada em C e C++.

REGISTRADORES DO ATmega328

Os registradores de I/O são o painel de controle do microcontrolador, 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

REGISTRADORES DOS PORTs

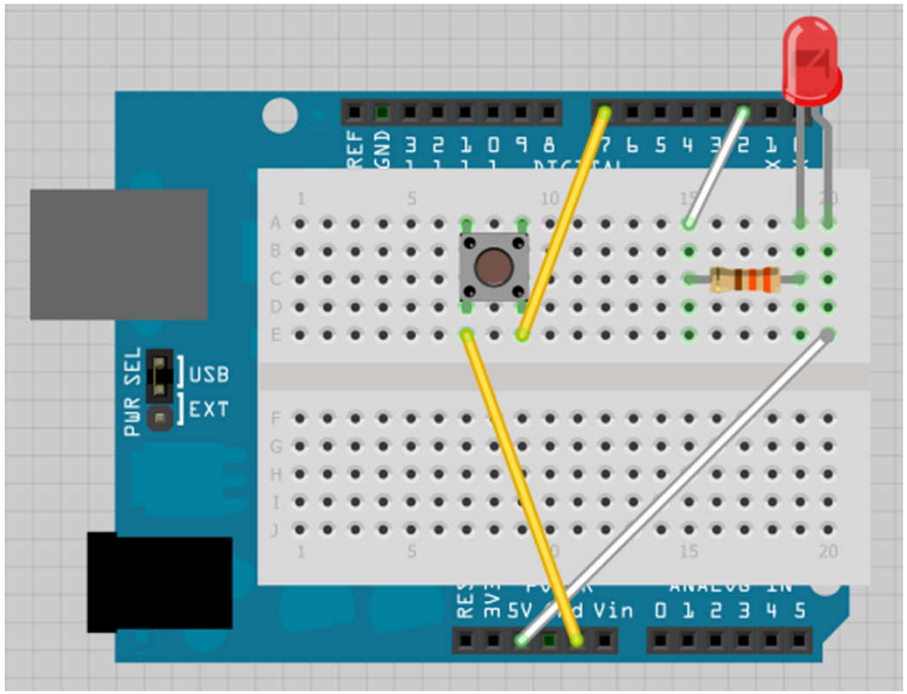
Os registradores responsáveis pelos pinos de I/O são:

- **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 (fornece corrente).

Exemplo




```
DDRD   = 0b00000100;

DDRD | = 1<< PC2;

PORTD  = 0b11111011;


set_bit(PORTD, 2);
clr_bit(PORTD, 2);

tst_bit(PIND, 7);
```

TRABALHO COM BITS

O trabalho com bits é fundamental para a programação de um microcontrolador. Assim, compreender como podem ser realizadas operações com bits é primordial para uma programação eficiente.

|	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

Exemplo:

```
ASSR   |=  1<<AS2;
TCCR2B = (1<<CS22)|(1<<CS20);
```

Ativa Bit

- **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     (PORTD, x pode ser 0 ou 1)
        | 0b00100000     (1<<5 é a máscara)
PORTD = 0bxx1xxxxx       (o bit 5 com certeza será 1)
```

Limpa Bit

- **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)) ,`

```
        0bxxxxxxxx      (PORTB, x pode ser 0 ou 1)
      & 0b11111011      (~(1<<2) é a máscara)
PORTB = 0bxxxxx0xx      (o bit 2 com certeza será 0)
```

Complementa Bit

- **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 ^ = (1<<bit_x)` ou `Y = Y ^ (1<<bit_x)`

Exemplo:

cpl_bit(PORTC,3)

PORTC = PORTC ^ (1<<3) ,

```
        0bxxxx1xxx     (PORTC, x pode ser 0 ou 1)
      ^ 0b00001000     (1<<3 é a máscara)
PORTC = 0bxxxx0xxx     (o bit 3 será 0 se o bit a ser
                        complementado for 1 e 1 se ele for 0)
```

Testa Bit

- **Leitura de um bit:**

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

Exemplo:

tst_bit(PIND,4)

PIND & (1<<4) ,

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

Exemplo

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

#define set_bit(Y,bit_x)(Y|=(1<<bit_x))
#define clr_bit(Y,bit_x)(Y&=~(1<<bit_x))
#define cpl_bit(Y,bit_x)(Y^=(1<<bit_x))
#define tst_bit(Y,bit_x)(Y&(1<<bit_x))

#define LED   PD2
#define BOTAO PD7

int main()
{
    DDRD = 0b00000100;  //configura o PORTD, PD2 saída, os demais pinos entradas
    PORTD= 0b11111111;  //habilita o pull-up para o botão e apaga o LED
    UCSR0B = 0x00;      //habilita os pinos PD0 e PD1 como I/O para uso no Arduino

    while(1)                                //laço infinito
    {
        if(!tst_bit(PIND,BOTAO))            //se o botão for pressionado executa o if
        {
            while(!tst_bit(PIND,BOTAO)); //fica preso até soltar o botão
            _delay_ms(10);               //atraso de 10 ms para eliminar o ruído do botão
            cpl_bit(PORTD,LED);          //troca o estado do LED
        }//if do botão pressionado
    }//laço infinito
}
```

PRODUZINDO UM CÓDIGO EFICIENTE

- Compile com a máxima otimização.
- Use variáveis locais sempre que possível.
- Use o menor tipo de dado possível (8 bits), **unsigned** se aplicável.
- Use **do{ } while(expressão)** se aplicável.
- Use laços com contadores decrescentes e pré-decrementados, se possível.
- Use *macros* ao invés de funções para tarefas menores que 2-3 linhas de código em *assembly*.
- Evite chamar funções dentro de interrupções.
- Se possível junte várias funções em um módulo (biblioteca), para aumentar o reuso do código.
- Todas as constantes e literais devem ser colocados na memória *flash.*

Exemplo – sem ponto flutuante

LM35 =10 mV/°C

45 °C = 0,45 V

0,45 → 450

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

V_{REF} =1,1 V

$$\boldsymbol{temp = ADC \times A}$$

$$450 = \frac{0{,}45 \times 1024}{1{,}1} \times \boldsymbol{A} \longrightarrow 450 = \frac{0{,}45 \times 1024}{1{,}1} \times \frac{1100}{1024}$$

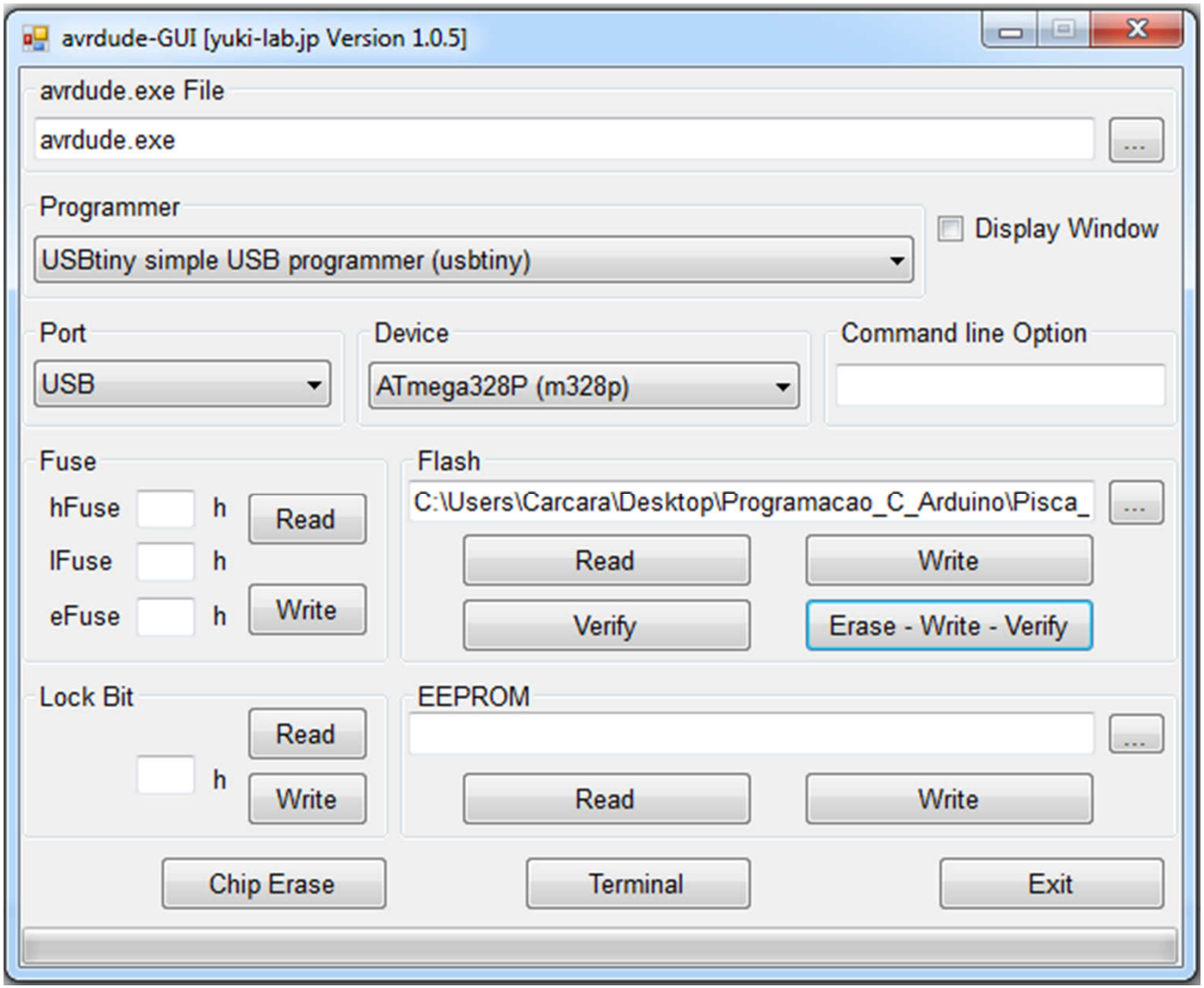
$$\boldsymbol{A} = \frac{1100}{1024} \longrightarrow \boldsymbol{A} = 1 + \frac{19}{256}$$

$$\boldsymbol{temp = ADC + \frac{(ADC \times 19)}{256}}$$

`unsigned int (16 bits)`

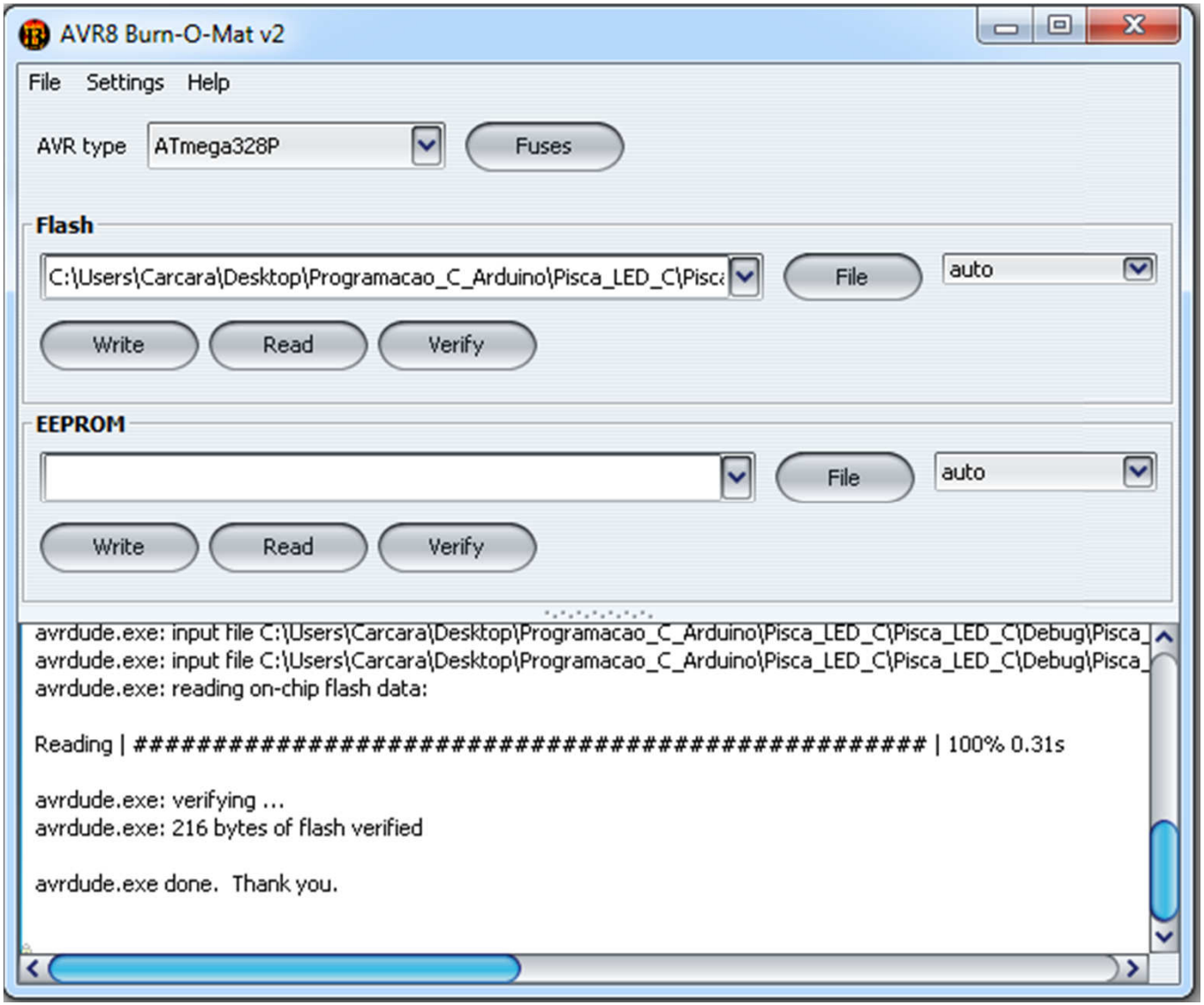
Evitar o uso de Ponto Flutuante.

GRAVAÇÃO DO FIRMWARE



Gravador USBtiny/USBasp ou diretamente pelo Arduino (conversor serial/USB com um µcontrolador com *Bootloader*).

GRAVAÇÃO DO FIRMWARE



CONCLUSÃO

O Arduino e seus Shields permitem um prototipação rápida, dado o conjunto de funções e bibliotecas disponíveis. É fácil de programar.

Todavia, a IDE do Arduino é muito limitada e inadequada ao desenvolvimento profissional. O código não é otimizado e não existem ferramentas de depuração.

O desenvolvimento profissional exige o conhecimento do microcontrolador e da programação C.

Códigos eficientes são resultantes de bons algoritmos, produzindo maior densidade de código (funcionalidade/bytes). É fundamental conhecer a arquitetura interna do microcontrolador para desenvolver os melhores programas.

REFERÊNCIAS BIBLIOGRAFICAS

LIMA, C. B.; VILLAÇA, M.V. M. **AVR e Arduino: Técnicas de Projeto**. 2ª. ed. Edição dos Autores, Florianópolis, 2012.

GANSSLE, Jack. ***The Firmware Handbook***. 1ª ed. Elsevier, United Kingdom, 2004.

ATmega48/88/168/328/A/PA/P: Microcontroladores AVR (Manual do fabricante).

Atmel AVR4027: *Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers* (*Application Note*).

http://atmel.com/

http://www.avrfreaks.net/

http://arduino.cc/

http://fritzing.org/

http://borgescorporation.blogspot.com.br/

MUITO OBRIGADO PELA ATENÇÃO!

borgescorp@gmail.com