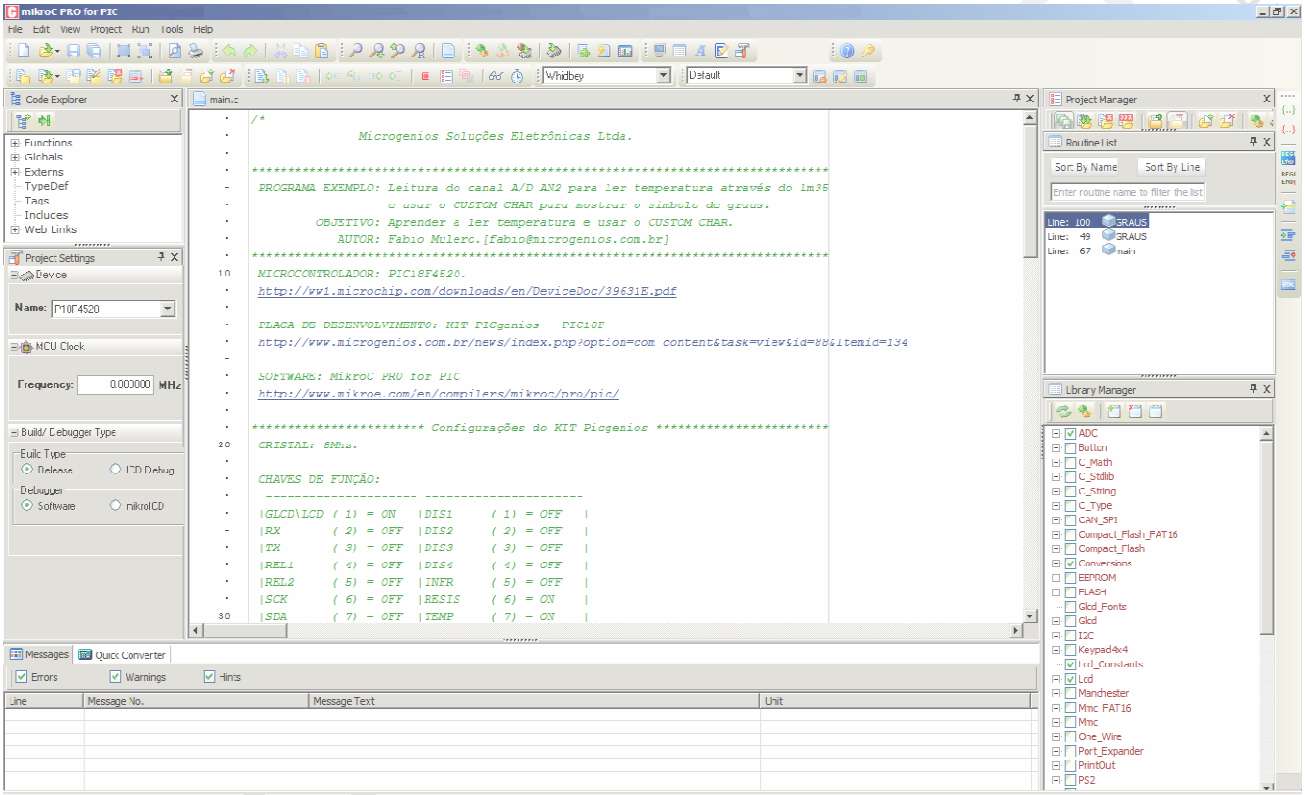
Linguagem C
Embarcada para Microcontroladores



Autor: Fernando Simplicio de Sousa
Fábio Perkowitsch Mulero
Equipe Microgenios

Livro:

Programação embarcada em Linguagem C

Eventuais erratas estarão disponíveis no site da Microgenios para download.

Dedicatória:

Dedico esse livro a Minha família e a equipe Microgenios

Advertência:

As informações e o material contido neste livro são fornecidos sem nenhuma garantia quer explícita, ou implícita, de que o uso de tais informações conduzirá sempre ao resultado desejado. Tanto o editor quanto o autor não podem ser responsabilizados por qualquer tipo de reivindicação atribuída a erros, omissões ou qualquer outra imprecisão na informação ou material fornecido neste livro, e em nenhuma hipótese podem ser incriminados direta ou indiretamente por qualquer dano, perda, lucros cessantes, etc., devido ao uso destas informações.

Prefácio

Esta obra foi concebida com o intuito de preparar os estudantes, professores e profissionais da área técnica para a criação de projetos utilizando como ferramenta uma linguagem de programação de alto nível, neste material escolhemos abordar a linguagem C, que é uma das linguagens mais poderosas e portáveis, fato este que a tornou amplamente utilizada, primeiramente para a criação de programas aplicativos para PC e mais tarde em sistemas embarcados microcontrolados.

Trabalhar com uma linguagem de alto nível, como C, para criar programas para microcontroladores, exige do profissional além de um bom conhecimento de lógica de programação e habilidade com ela, um sólido conhecimento da estrutura de hardware do microcontrolador a ser utilizado, de forma a extrair deste o máximo da funcionalidade de seus periféricos internos.

Esta obra estuda o software, propiciando um conhecimento completo ao profissional e tornando-o apto a desenvolver suas próprias aplicações, além disso, vale a pena ressaltar a preocupação, por parte do Centro de Tecnologia Microgenios, em priorizar um estudo gradual e prático, para isso usamos os kits de desenvolvimento, como base para a realização de diversas experiências que complementam e fixam o aprendizado.

Um ponto de destaque da abordagem do treinamento é o uso e detalhamento da IDE de desenvolvimento MikroC (www.mikroe.com) a qual apesar das limitações da versão de demonstração gratuita mostra-se uma excelente ferramenta de desenvolvimento e simulação.

De maneira alguma este material é apresentado como única fonte de estudo sobre o assunto, devendo aqueles que necessitarem se aprofundar nos tópicos aqui estudados buscar outras fontes de pesquisa.

Por fim a equipe Microgenios agradece a atenção de todos e deseja bons estudos e projetos.

Fernando Simplicio de Sousa
Fábio Perkowitsch Mulero
Equipe Microgenios

Capítulo 1 – Introdução a Linguagem C

Um pouco de história

A primeira versão de C foi criada por Dennis Ritchie em 1972 nos laboratórios Bell para ser incluído como um dos softwares a serem distribuídos juntamente com o sistema operacional Unix do computador PDP-11, na equipe certificada por Ken Thompson.

Pelo ponto de vista técnico, o surgimento do C iniciou-se com a linguagem ALGOL 60, definida em 1960.O ALGOL era uma linguagem de alto nível, que permitia ao programador trabalhar sem conhecimento especifico da máquina, sem se preocupar com os aspectos de como cada comando ou dado era armazenado ou processado na máquina.

O ALGOL não teve sucesso, talvez por tentar ser alto nível em uma época em que a maioria dos sistemas operacionais exigiam do usuário um grande conhecimento de hardware.

Em 1967 surgiu o CPL (Combined Programming Language) nas universidades de Londres e Cambridge com o objetivo, segundo a equipe do projeto, de simplificar o ALGOL. Da mesma forma que o ALGOL, o CPL não foi bem aceito, em especial pelos projetistas de sistemas operacionais que a consideravam difícil sua implementação.

Ainda em 1967, em Cambridge, Martin Richards criou o BCPL (Basic CPL), uma simplificação do CPL, tentando manter apenas as partes funcionais do CPL.

Em 1970, Ken Thompson, chefe da equipe que projetou o UNIX para o PDP11 do Bell Labs, implementou um compilador para uma versão mais reduzida do CPL. Que batizou de linguagem de B.

Tanto o BCPL quanto o B mostraram-se muito limitadas, resolvendo apenas para certos tipos de problemas. Isto foi sentido especialmente na primeira versão do PDP11, lançado no mercado em 1971. Um dos fatores que levou a esta constatação foi a intenção do grupo responsável pelo UNIX de reescrevê-lo todo em uma linguagem de alto nível, e para isto B era considerado lento.

Diante desses problemas a Bell Labs encarregou o projetista Dennis Ritchie a projetar uma nova linguagem, sucessora do B, que viria então, a ser chamada de C.

A linguagem C buscou manter o poder de manipulação de hardware e ainda assim dar ao programador novas condições para o desenvolvimento de programas em áreas diversas como, comercial, científica e engenharia.

Por vários anos (aproximadamente 10) a sintaxe(padronização de código) tida como padrão da linguagem C foi aquela fornecida com o UNIX versão 5.0 do Bell Labs. A principal documentação deste padrão encontra-se na publicação "The C Programming Language", de Brian Kernighan e Dennis Ritchie (K&R), tida como a "bíblia da linguagem C".

O mais interessante desta versão de C era que os programas-fonte criados para rodar em um tipo de computador podiam ser transportados e recompilados em outros sem grandes problemas. A esta característica dá-se o nome de portabilidade. Com ela, uma empresa que desenvolve um programa pode fazê-lo rodar em diferentes computadores sem ter um elevado custo a cada vez que isto for feito.

Em 1985, O ANSI (American National Standards Institute) estabeleceu um padrão oficial de C o chamado "C ANSI" que é adotado até hoje.

Conceito

C é uma linguagem estruturada, ou seja, existe um modo de organização para escrita do programa baseada no modelo “ANSI C”, a principal característica dessa linguagem é a compartilhamento do código e dos dados.

C é chamada de linguagem de médio nível, pois une as facilidades de linguagens de alto nível(BASIC, PASCAL) com manipulação de hardware somente obtida com a linguagem ASSEMBLY, mas mesmo tendo essas características é uma linguagem de fácil uso e aprendizagem.

A linguagem C é "Case Sensitive", isto é, ela faz a diferenciação das letras maiúsculas e minúsculas EX: MAin() ou mAIn() é diferente de main(), portanto utilize uma padronização para escrita de seus códigos

Compilador

A linguagem que seja os processadores, incluindo microcontroladores, entendem é chamada: "código de máquina", a partir dessa linguagem os processadores executam as tarefas programadas pelo programador. Mas como é uma tarefa muito complicada e improdutiva escrever programas em "código de máquina", foi criada uma ferramenta chamada compilador que nada mais é que um “tradutor” que pega seu código escrito em C e traduz para "código de máquina".

Capítulo 2 – Variáveis

Neste capitulo iremos aprender sobre variáveis, operadores lógicos e aritméticos alem de estruturas de controle e decisão.

Para facilitar nossos estudos iremos adotar como padrão os microcontroladores de 8 bits, mas todos os conhecimentos adquiridos nesse curso poderão ser utilizados em qualquer tipo de processador ou microcontrolador.

Para iniciarmos nossos estudos vamos falar sobre variáveis.

O que são variáveis?

Variável, do ponto de vista da área de programação, é uma região de memória previamente identificada que tem por finalidade armazenar os dados ou informações de um programa por um determinado espaço de tempo. Uma variável limita-se a armazenar apenas um valor por vez.

Tipos de dados

Existem cinco tipos primitivos de dados na linguagem C: caractere (char), inteiro (int), ponto flutuante (float e double) e sem valor (void). A tabela abaixo representa o valor e a escala de cada tipo de dado em C.

TIPO	TAMANHO EM BITS	TAMANHO EM BYTES	INTERVALO
char	8	1	0 a 255
int	16	2	0 a 65535
long	32	4	0 a 4294967295
float ou double	32	4	$-1.5 * 10^{45}$ a $+3.4 * 10^{38}$
void	0	-	nenhum valor

O tipo char pode ser utilizado tanto para manipulação de valores entre 0 e 255 quanto para representação de caracteres da tabela ASCII. Cada variável do tipo char pode representar somente um caractere ASCII.

O tipo int é utilizado para representar números inteiros de 16 bits (0 a 65535).

Esses dois primeiros tipos são os mais utilizados em linguagem C.

O tipo char e int representam números inteiros e não podem ser utilizados para representar números fracionários. Para isso, deve ser utilizado o tipo float, também chamado de ponto flutuante.

O tipo float deve ser evitado ao máximo e restrito apenas às operações que realmente necessitarem de um tipo de dados como este.

Modificadores de Tipo

Podemos utilizar comandos especiais do C para obter outros tipos de dados. Esses comandos especiais são chamados de modificadores de tipo e são os seguintes:

signed, **unsigned**, **short**, e **long**.

O modificador de tipo **signed** é utilizado para modificar um dado para que ele represente valores positivos e negativos. Este tipo é opcional, pois na linguagem C a omissão do modificador **unsigned** o próprio compilador adota como padrão o modificador **signed.**

O modificador **unsigned** indica ao compilador que a variável trabalhará somente com números positivos.

O modificador **short** é utilizado para definir uma variável com tamanho menor que o tipo modificado, ou seja, uma versão reduzida do tipo especificado.

O modificador **long** é utilizado para ampliar a magnitude de representação do tipo especificado.

TIPO	TAMANHO EM BITS	RANGE
unsigned char	8	0 à 255
(signed) char	8	-128 à +127
(signed) short (int)	8	-128 à +127
(unsigned) short (int)	8	0 à 255
signed int	16	-32768 à 32767
unsigned int	16	0 à 65535
(signed) long int	32	-2147483648 à 2147483647
unsigned long int	32	0 à 4294967295
float	32	+/- 1.17549435082E-38 à +/- 6.80564774407E38
double	32	+/- 1.17549435082E-38 à +/- 6.80564774407E38
long double	32	+/- 1.17549435082E-38 à +/- 6.80564774407E38
void	-	sem valor

Obs: No caso da nossa ferramenta de trabalho, o MikroC, os tipos de float, double e long double são considerados do mesmo tipo.

Declaração de variáveis

Definir uma variável é criá-la na memória (alocá-la), dar a ela um nome e especificar o tipo de dado que nela vai armazenar.

Sintaxe para criação de uma variável em C;

<modificador> <tipo> nome_da_variavel;

O tipo deve ser um tipo de dado válido em C tais como: char, int, float, com ou sem seus modificadores, unsigned, signed, short e long. E nome_da_variavel é o nome da variável adotada pelo programador.

As regras para a escrita de nomes de variáveis e funções em C serão melhor detalhadas mais a frente.

Em C, podemos declarar várias variáveis do mesmo tipo em um única linha de programa, bastando apenas separá-las por vírgulas, acompanhe:

```
int soma ;
unsigned char i,j,k ;
float salário;
unsigned int idade;
short int y;
long caminho, estado;
unsigned valor;
```

Outro aspecto importante da declaração das variáveis é o local onde são declaradas.

Basicamente, uma variável pode ser declarada em dois escopos distintos, o de acesso global e o de acesso local.

Variáveis globais:

Essas variáveis são declaradas fora das funções do programa, podendo ser acessadas em qualquer parte do programa.

Exemplo:

```
int contador;   //define a variável global "contador" como inteiro
char a;         //define a variável global "a" como char

void main()     //função principal do programa
{
  contador = contador + 10
}
```

Vamos apresentar mais um exemplo de programa utilizando as variáveis globais:

```
int contador;    //define a variável global "contador" como inteiro
char a;          //define a variável global "a" como char

void subrotina ()  //função de sub-rotina qualquer do programa
{
  contador = contador - 20;
  a = 100;
}

void main()     //função principal do programa
{
  contador = contador + 10;
  a = 55;
}
```

Repare no programa exemplo acima que as variáveis "contador" e "a" estão sendo manipuladas no corpo da função main() e na função subrotina(). Este tipo de manipulação de variáveis somente é possível se declararmos como sendo do tipo GLOBAIS, ou seja, necessariamente devemos defini-las no corpo do programa (fora de qualquer função do programa, inclusive a função main()).

Variáveis locais:

Ao contrário das variáveis globais, uma variável local somente existe no escopo da função em que foi declarada. Isto significa que uma variável local somente existe enquanto a função esta sendo executada. No momento que ocorre o retorno da função, as variáveis locais são descartadas;

Acompanhe o exemplo abaixo:

```
void main()       //função principal do programa
{
  int contador;   //define a variável local "contador" como int
  contador = contador + 10
}
```

Repare que a variável "contador" foi declarada (criada) dentro da função main(), as variável locais somente terá validade dentro da função que a declarou, neste caso, "contador" somente poderá ser manipulada no programa dentro da função main().

Acompanhe mais um exemplo de variável local:

```
void subrotina ()  //função de sub-rotina qualquer do programa
{
  int tempo;
  tempo = tempo * 2;
}

void main()        //função principal do programa
{
  int tempo;
  tempo = tempo / 2;
}
```

Repare no programa exemplo acima que temos duas funções, a função main() e a função subrotina(). Dentro de cada uma dessas funções foi criada uma variável chamada "tempo" do tipo int, essas variáveis são locais e somente tem validade dentro da função que a declarou, ou seja, a variável "tempo" da função subrotina() NÃO tem nenhuma ligação com a variável "tempo" da função main(), pois são variáveis locais;

Se por ventura cometermos o seguinte erro abaixo, acompanhe:

```
void subrotina ()  //função de sub-rotina qualquer do programa
{
  tempo = tempo * 2;
}
void main()     //função principal do programa
{
  int tempo;
}
```

Repare que a variável "tempo" foi declarada unicamente dentro da função main() e que seus dados estão sendo manipuladas por outra função chamada subrotina(). Ao compilarmos este programa, certamente o compilador apresentará um erro de compilação nos informando que a variável "tempo" não foi definida. Esse é um erro comum que muitos programadores inexperientes cometem na linguagem C, pois esquecem que uma variável local somente tem validade dentro da função que a declarou.

Podemos declarar variáveis como parâmetros formais de uma função, que são também tratadas como variáveis locais. Iremos estudar esse tipo de variável quando estivermos falando de funções em nosso curso.

Capítulo 3 – Tipos de Operadores

A linguagem C faz uso de diversos operadores. Podemos classificá-los em algumas categoria principais: aritméticos, relacionais, lógicos, lógicos bitwise(bit a bit) , outros.

Os Operadores Aritméticos:

Os operadores aritméticos são utilizados para efetuar operações matemáticas entre dados e são classificados em duas categorias, sendo binário ou unário. Os operadores binários atuam em operações de exponenciação, divisão, adição e subtração. Os unários atua na inversão de valores, atribuindo o sinal de positivo ou negativo.

operador	ação
+	Adição
-	Subtração ou menos unário
*	Multiplicação
/	Divisão
%	Resto da divisão inteira
++	Incremento
--	Decremento

Adição e Subtração

Os operadores de adição e subtração são muito simples de serem utilizados, acompanhe:

```
c = a + b;
d = d + b;
c = c - a;
a = d - a + c;
```

Multiplicação e Divisão

Os operadores de multiplicação e de divisão, em C, são de fácil assimilação;

```
c = a * b;
d = d * d;
c = c / a;
a = (d * a) / c;
```

O operador %

O operador % é utilizado para retornar o módulo(resto) de uma operação de divisão inteira. Vejamos um exemplo:

7 / 3 = 2, em uma divisão inteira, sendo o resto igual a 1.

Assim, o valor de 7 / 3 = 2 e o valor de 7 % 3 = 1.

O operador de Incremento ++ e Decremento --:

O operador de incremento e decremento são utilizados para somar 1 ou subtrair 1 de uma variável.

EX:

C = C + 1;

No exemplo acima a variável C foi incrementada em 1 unidade. Podemos ter o mesmo resultado da operação acima através do operador de incremento ++ que termos o mesmo resultado;

```
c++;   //equivale a c = c + 1;
c++;   //incrementa em 1 unidade o valor da variável C
d--;   //incrementa em 1 unidade o valor da variável D
```

Acompanhe o exemplo abaixo:

```
variavel_a = variavel_b++;
```

Observe que o valor da variável variavel_b é atribuído a variável variavel_a, e logo após isso, o valor de variavel_b é incrementado em 1 unidade:

Isso também é válido para o comando de decremento --.

```
variavel_a = variavel_b--;
```

Observe que o valor da variável variavel_b é atribuído a variável variavel_a, e logo após isso, o valor de variavel_b é decrementado em 1 unidade:

Devemos tomar alguns cuidados em C com relação aos operadores de incremento e decremento: vejam;

```
variavel_a = variavel_b++;
```

é diferente de

```
variavel_a = ++variavel_b;
```

Note que na primeira linha de comando, variavel_b é incrementada em 1 depois que atribuiu seu valor a variável variavel_a. Na segunda linha, o valor da variavel_b é incrementada em 1 antes de atribuir seu valor a variável variavel_a.

Vejamos um exemplo:

```
int a, b, c;
a = 0;
b = a++;
c = ++a;
```

Neste caso, após a execução dos três comandos, o valor de "a" será igual a 2, o valor da variável "b" será igual a 0 e o valor da variável "c" será igual a 2.

Operadores Relacionais;

Os operadores relacionais servem para comparar expressões. São muito utilizado para comparar condições de controle do fluxo de execução de programas.

Operador	Operação realizada
>	maior que
>=	maior ou igual a
<	menor que
<=	menor ou igual a
==	igual a
!=	diferente de

Estes operadores serão muito utilizado para construir expressões condicionais, como veremos mais adiante em nosso curso.

Operadores lógicos ou Booleanos

Os operadores lógicos ou booleanos são de grande importância para construções de testes condicionais.

Operador	Operação realizada

&&	AND (E)
||	OR (OU)
!	NOT (NÃO)

Com esses operadores podemos relacionar diversas condições diferentes em um mesmo teste lógico.

```
if (c = 10 && b = 5) c = 5;  /* condição verdadeira de teste: se a variável "c" for igual a 10 e a variável b = 5 então "c" assumirá o valor 5.*/

if (c>0 || a==0) b = a; /* condição verdadeira de teste: se a variável "c" for maior que 0 ou a for igual a 0 então "b" é igual a variável "a".*/

if (!a) b = c; /* condição verdadeira de teste: se a variável "a" for igual a 0, a variável "b" assumirá o valor da variável "c". Note que estamos utilizando o operador de negação "!" NOT, por esse motivo a variável "a" assumirá o valor verdadeiro, já que possui valor 0.*/
```

Os operadores Bitwise (Bit a Bit)

Os operadores Bit a Bit são utilizados para realizar operações lógicas entre elementos ou variáveis.

Operador	Operação realizada
&	AND (E)
|	OR (OU)
^	XOR (OU exclusiva)
~	NOT (complemento de um)
>>	deslocamento à direita
<<	deslocamento à esquerda

Operador AND (E)

O operador lógico AND realiza operação separadamente para cada bit dos operandos. Utilizamos muito o operando AND como "máscara" de um byte, para habilitar ou desabilitar somente os bits que desejamos. Esse operando funciona da seguinte maneira, verifica se nos dois operandos um respectivo bit, se nos dois forem 1 coloca no resultado 1 no bit respectivo, caso contrario coloca 0.Veja um exemplo:

```
int a, b;
a = 100;
b = 28;
a = a & b;
```

A operação AND ocorrerá da seguinte maneira:

a = 100 ----> 01100100

AND (&)

b = 28 ----> 00011100

Resultado= 00000100 ou 4 decimal

O Operador OR (|)

O operador OR é muito similar ao operador AND, sua operação também é realizada para cada bit do operando. Esse operando funciona da seguinte maneira, verifica se nos dois operandos um respectivo bit, se em qualquer dos dois for 1 coloca no resultado 1 no bit respectivo, caso contrario coloca 0. Exemplo:

```
int a, b;
a = 100;
b = 28;
a = a | b;
```

A operação OR ocorrerá da seguinte maneira:

a = 100 ----> 01100100

OR (|)

b = 28 ----> 00011100

Resultado=01111100 ou 124 decimal

O Operador XOR (^)

O operador XOR são muito utilizado em funções de comparação de valores, pois em uma operação lógica, o resultado somente será verdadeiro (nível lógico 1) se um e somente um deles for verdadeiro (nível 1). Esse operando funciona da seguinte maneira, verifica se nos dois operandos um respectivo bit, se nos dois forem diferentes coloca no resultado 1 no bit respectivo, caso contrario coloca 0. Exemplo:

```
int a, b;
a = 100;
b = 28;
a = a ^ b;
```

A operação OR ocorrerá da seguinte maneira:

a = 100 ----> 01100100

XOR (^)

b = 28 ----> 00011100

Resultado=01111000 ou 120 decimal

O Operador NOT (~)

O operador NOT inverte o estado de cada bit do operando em uma operação. Exemplo:

```
int a, b, c;
a = 1;
b = 240;
b = ~a
c = ~b
```

A operação OR ocorrerá da seguinte maneira:

a = 0B00000001; NOT de "a" ----> 0b11111110;

b = 0B11110000; NOT de "b" ----> 0b00001111;

Operador de deslocamento << >>

O operador >> desloca para a direita os bits de uma variável um determinado número de vezes. Exemplo:

```
int a, b, c;
a = 10;
b = 10;
b = a >> 1;
c = b << 5;
```

No exemplo acima, os valores dos bits da variável "a" foram deslocados 2 vezes para a direita, enquanto os bits da variável b foram deslocados 5 vezes para a esquerda.

Teremos como resultado:

```
variável a;  // ----> 00001010  - valor 10 em binário
>>
             // ---->  00000101 - valor 5 em binário
```

Observe que o valor após deslocamento passa a ser agora 5 em binário.

A descrição do operador (>>), deslocamento à direita, é análoga ao operador deslocamento à esquerda (>>), com ressalva de que os bits serão deslocados à esquerda.

Quando utilizamos (>>) – deslocamento à direita o valor do 8 bit será propagado, ou melhor, deslocado “N” vezes.

Exemplo 1:

Vamos rotacionar o valor da variável “a” 5 vezes o seguinte número para a direita e salvar em b.

```
b = a >> 5;
variável a;  // ----> 1000 0000  - valor 64 em decimal
>>
        // ----> 1111 1000  - valor 248 decimal
```

Exemplo 2:

Vamos rotacionar para a esquerda o valor da variável “a” 4 vezes, e salvar em b.

```
b = a << 4;
variável a;  // ----> 1101 1110  - valor 222 em decimal
>>
             // ----> 1110 0000  - valor 224 decimal
```

Capítulo 4 – Formas e representação numéricas e de caracteres

No compilador MikroC podemos manipular dados do tipo: decimal, binário, hexadecimais e octal.

Podemos representar um valor numérico de diversas formas. Para exemplificar, vamos supor o que desejamos carregar o valor 187 no registrador PORTB no PIC utilizando o MikroC, acompanhe:

Representação decimal:

Para representarmos um número em decimal basta colocar seu valor sem nenhuma abreviatura, conforme o linha abaixo:

```
PORTB = 187;  //representação decimal
```

NOTA:
NUNCA DEVEMOS REPRESENTAR UM NÚMERO DECIMAL INICIANDO COM 0 (ZERO), POIS O MIKROC INTERPRETARÁ O NÚMERO COMO OCTAL.

EX:

```
PORTB = 25;  // (25 é representado em decimal)
```

é diferente de

```
PORTB = 025;  // (025 é representado em octal)
```

Representação Hexadecimal:

Para representar um número em hexadecimal devemos colocar o prefixo 0x (ou 0X) antes do número hexadecimal. (0 a F)

```
PORTB = 0xBB;  //representação hexadecimal do numero 187 decimal
```

ou

```
PORTB = 0Xbb;  //representação hexadecimal do numero 187 decimal
Representação binária:
```

Para representarmos um número em binário devemos colocar o prefixo 0b (ou 0B) antes do número em binário.

```
PORTB = 0b10111011; //representação binária do número 187 decimal
```

ou

```
PORTB = 0B10111011; //representação binária do número 187 decimal
```

A representação binária de um número ocorre da seguinte forma:

0	B	1	0	1	1	1	0	1	1
-	-	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
-	-	MSB							LSB

Observe que na representação binária de um número, o bit mais significativo fica ao lado do prefixo 0b. Não confunda de forma alguma a construção e as

representações de um número binário, caso contrário seu programa não funcionará corretamente.

Representação octal:

O sistema octal não é um sistema muito difundido nos dias atuais. Apesar do compilador MikroC suportar esse tipo de representação numérica, somente devemos utilizá-la quando for realmente necessário.

Para representar um número octal é necessário colocar o prefixo 0 (zero) antes do valor numérico octal.

```
portd = 0273;  //representação octal do número 187 em decimal
```

Exercícios de fixação:

Gostaríamos de enviar aos pinos do PORTB os seguintes estados;

```
RB0 = 0; RB1 = 1; RB2 = 1; RB3 = 0; RB4 = 0; RB5 = 1; RB6 = 0; RB7 = 0
```

Qual o valor numérico que devemos colocar na instrução abaixo para representar os estados dos pinos do PORTB. Obedeça as representações numéricas solicitadas:

Representação decimal:

```
PORTB = 38;
```

Representação binária:

```
PORTB = 0b00100110;
```

Representação hexadecimal:

```
PORTB = 0x26;
```

Representação octal:

```
PORTB = 046;
```

Capítulo 5 – MikroC : Case Insensitive

Diferente do padrão ANSI C, No MikroC podemos utilizar caracteres maiúsculos e minúsculos no programa (case insensitive). Acompanhe:

contador, Contador e COnTadOR

São interpretados como sendo a mesma palavra. (esse recurso é uma particularidade do MikroC e não pertence ao padrão ANSI C Standart).

NOTA:
COMANDOS QUE FAZEM PARTE DA ESTRUTURA DO C, COMO POR EXEMPLO: IF, WHILE, FOR, SWITH , VOID MAIN, ENTRE OUTROS, OBRIGATÓRIAMENTE DEVERÃO SER DECLARADOS EM MÍNÚSCULO.

Manipulação de bit no MikroC

Podemos manipular os bits de registradores do PIC de diversas formas no compilador MikroC

Podemos manipular os bits dos registradores do PIC da seguinte maneira:

```
PORTA.F5 = 0;  // -->> faz referência ao pino RA5 do PIC.
PORTD.F7 = 1;  // -->> faz referência ao pino RD7 do PIC.
TRISB.RB0 = 1; // -->> faz referencia ao bit 0 do registrador TRISB
T0CON.PSA = 0; // -->> faz referência ao bit 3
```

OBS: Não são todos os registradores que permitem acesso direto ao seus bits.

Capítulo 6 – Como escrever programas em C

Acompanhe o exemplo abaixo:

```
//Primeiro Programa
/* Programa Pisca-Pisca (1 segundo)
Este programa tem por objetivo ascender e apagar um led conectado no pino RB0 do PIC em
intervalos de 1 segundo aproximadamente;
Curso Online: Microcontroladores PIC - Programação em C
*/
void main()
{
	TRISB = 0; //define PORTB como saída
	PORTB = 0; //coloca nível lógico 0 em todos os pinos do  PORTB

	while(1)
	{
		PORTB.F0 = 1;   // Coloca pino RB0 em 1
		Delay_ms(1000); // Aguarda 1000 ms (milisegundos)
		PORTB.F0 = 0;   // Coloca pino RB0 em 0
		Delay_1sec();   // Aguarda 1 segundo
	}
}
```

Vejamos o significado de cada linha de programa:

A primeira linha de programa:

```
//Primeiro Programa
```

É chamada de comentário. Os comentários são importantes para documentar o programa. Podemos adicionar comentários em nosso programa de duas maneiras:

Através de comentários de linhas simples: quando queremos comentar apenas uma linha de programa, iniciamos os comentários com os caracteres " // " (igual a linha do programa anterior). Esses tipo de comentário não faz parte da padronização ANSI original, mas atualmente é encontrado em vários compiladores.

Através de comentários de múltiplas linhas: podemos comentar linhas ou blocos de código de programa utilizando a seqüência de caracteres "/* " para iniciar o comentário e a seqüência "*/" para terminar o comentário.

```
/* Programa Pisca -Pisca
Este programa tem por objetivo ascender e apagar um led conectado no pino RB0 do PIC em
intervalos de 1 segundo aproximadamente;
Curso Online: Microcontroladores PIC - Programação em C
*/
```

Na próxima linha temos:

```
void main()
```

A declaração main() especifica o nome da função. A função main(), na linguagem C, é sempre a primeira a ser executada. O programa termina quando for encerrada a execução da função main().

Uma função, em C, nada mais é do que um conjunto de instruções que pode ser executada a partir de qualquer ponto do programa. Utilizamos o sinal de abertura de chave "{" para iniciar uma função e o sinal de fechamento de chave "}" para finalizar a função. Todas as instruções devem estar dentro das chaves que iniciam e terminam a função e são executadas na ordem em que as escrevemos.

No caso deste programa exemplo, ela não recebe nenhum parâmetro e também não retorna parâmetro nenhum. Isto fica explícito através da palavra-chave void escrita na frente do programa.

As funções e as suas características serão apresentadas em detalhes mais adiante em nosso curso;

Na próxima linha de programa encontramos:

```
TRISB = 0; //define PORTB como saída
PORTB = 0; //coloca nível lógico 0 em todos os pinos do PORTB
```

O comando TRISB define o sentido de acionamento do PORTB do PIC. Neste exemplo, TRISB = 0, logo o PORTB foi programado como saída.

O comando PORTB = 0 coloca nível lógico 0 em RB0 a RB7. As instruções C são sempre encerradas por um ponto-e-vírgula (;). O ponto-e-vírgula é parte da instrução e não um simples separador e devemos ao final de cada instrução colocar o acréscimo de um ponto-vírgula “;”.

Na próxima linha temos:

```
while(1)
```

Este é um comando de controle utilizado na repetição de um determinado bloco de instrução. O bloco de instrução será executado repetidamente enquanto o condição for verdadeira, ou seja, for diferente de zero. No nosso exemplo, o valor 1 utilizado no comando while garante que a condição seja sempre verdadeira. Estudaremos mais sobre o comando while mais adiante no curso;

Os comandos a seguir pertencem ao bloco da instrução while;

```
{
   PORTB.F0 = 1;   // Coloca pino RB0 em 1
   Delay_ms(1000); // Aguarda 1000 ms (milisegundos)
   PORTB.F0 = 0;   // Coloca pino RB0 em 0
   Delay_1sec();   // Aguarda 1 segundo
}
```

Como não temos nenhum comando que interrompa o laço while, os blocos de comandos apresentados serão executados indefinidamente até que o processador seja desligado ou reiniciado.

A operação PORTB.F0 = 1 faz com que o pino RB0 do PORTB seja colocado em nível lógico 1; Para se referir a um pino das portas do PIC, PORTA, PORTB, PORTC, PORTD, PORTE, devemos apresentar o nome do porta, mais o ponto ".", mais a inicial "f" e o número do pino correspondente.

Exemplo:

```
PORTB.F0 = 0; // Estamos nos referindo ao pino RB0 do PORTB
PORTD.F5 = 1; // Estamos nos referindo ao pino RD5 do PORTD
PORTE.F1 = 1; // Estamos nos referindo ao pino RE1 do PORTE
```

A linha de programa seguinte, Delay_ms(1000), é uma função interna do compilador MikroC utilizada para gerar atrasos em escala de milissegundos. No nosso exemplo, o comando irá gerar atraso de 1000 milessegundos, ou seja, 1 segundo.

A operação PORTB.F0 = 0 faz com que o pino RB0 do PORTB seja colocado em nível lógico 0;

A linha de programa seguinte, Delay_1seg(), tem a mesma função de gerar atrasos que a função estudada anteriormente, sua diferença é que o valor do atraso passa a ser de 1 segundo aproximadamente.

Desta forma, ao programarmos o PIC com o programa exemplo, o pino RB0 ficará mudando de estado lógico 1 e 0 a cada 1 segundo,

NOTA:
COLOQUE EM PRÁTICA ESTE PEQUENO PROJETO. COPIE E COMPILE ESTE PROGRAMA EXEMPLO NO COMPILADOR MIKROC, ISSO LHE AJUDARÁ NOS SEUS ESTUDOS.

Vamos estudar mais um programa:

Acompanhe:

```
//Segundo Programa
/* Programa Pisca -Pisca (100 milisegundos)
Este programa tem por objetivo ascender e apagar um led conectado no pino RB0 do PIC em
intervalos de 1 segundo aproximadamente;
Curso Online: Microcontroladores PIC - Programação em C
*/
void main()
{
   TRISB = 0; //define PORTB como saida
   PORTB = 0; //coloca nível lógico 0 em todos os pinos do PORTB

   TRISD= 0; //define  PORTB como saida
   PORTD= 0; //coloca nível lógico 0 em todos os pinos do PORTD

   while(1)
   {
      PORTB = 255;   //seta todos os pinos do port
      TRISD = 255;   //seta todos os pinos do portd
      Delay_ms(100); //aguarda 1000 ms (milisegundos)
      PORTB = 0;     //resseta todos os pinos do  PORTB
      PORTD = 0;     //resseta todos os pinos do PORTD
      Delay_ms(100); //aguarda 1000 ms (milisegundos)
   }
}
```

O programa acima tem por objetivo piscar infinitamente o PORTB e PORTD do PIC em intervalos de 100 milissegundos. Suas características são parecidas com a do programa anterior, sua única diferença está no tratamento das portas;

```
 PORTB = 255;  //seta todos os pinos do PORTB
 PORTB = 255;  //seta todos os pinos do PORTD
```

Estrutura de um programa em C

Todo programa escrito em C consiste em uma ou mais funções, tendo como particularidade deste fato a possibilidade de construir programas modulares e estruturados. O programa principal escrito em C é uma função. O C é uma linguagem extremamente estruturada e exige do programador domínio adequado de tal conceito. Veja a seguir, o menor programa possível de ser escrito em C:

```
void main() //é a primeira e principal função a ser executada
{      //inicia ou abre o corpo da função
}      //finaliza ou fecha o corpo da função
```

A função main() é a principal instrução a ser considerada em um programa escrito na linguagem C e deve estar presente em algum lugar do programa, pois é ela que marca o ponto de inicialização do processo de execução do programa.

A seguir veremos um modelo de escrita de um programa em C, com todas as definições e comentários. Acompanhe:

```
[blocos de comentários de grande importância na documentação do programa]

[<definições de pre-processamento - cabeçalhos>]

[<declaração das variáveis globais>]

[<tipo>] nome_da_funcao([<parâmetros>])
[<declaração de parâmetros>]
{
  /*
  Este trecho é reservado para o corpo da função_nome, com a declaração de suas
  variáveis locais, seus comandos e funções de trabalho. Esta função pode ser
  chamada de sub-rotina do programa.
  */
  [return ou return() ou return(valor)]
}

void main([<parâmetros>])
{
  /*
  Este trecho é reservado para o corpo da função, com a declaração de suas
  variáveis locais, seus comandos e funções de trabalho. Aqui estão os
  primeiros comandos que serão executados no programa.
  */
}
```

Toda a informação situada entre colchetes "[" e "]" indica informações que podem ou não estar presentes em um programa.

Vamos comentar cada bloco de programa:

O primeiro bloco: Os comentários gerais

```
[ blocos de comentários de grande importância na documentação do programa]
```

O programador poderá inserir ou não o comentários em seu programa. Por convenção, é importante colocarmos sempre os cabeçalhos nos programas, pois é a apresentação do programa.

O segundo bloco: O cabeçalho

```
[<definições de pré-processamento - cabeçalhos>]
[<declaração das variáveis globais>]
```

Neste trecho do programas podemos declarar todo o cabeçalho do nosso programa assim como definir variáveis globais que poderão ser utilizadas em todas as funções do nosso programa.

Exemplo de cabeçalho:

```
#include "minhas_funcoes.h"
/* incluímos em nosso programa as bibliotecas de funções que estão no arquivo
minhas_funcoes.h.
*/
 int a ;  //definimos a variável a como inteiro e como sendo global
 int b;  //definimos a variável b como inteiro e como sendo global
```

O terceiro bloco: As funções de subrotinas

```
[<tipo>] nome_da_funcao([<parâmetros>])
[<declaração de parâmentros>]
{
  /*
  Este trecho é reservado para o corpo da nome_da_funcao, com a declaração de
  suas variáveis locais, seus comandos e funções de trabalho. Esta função pode
  ser chamada de sub-rotina do programa.
 */

[return ou return() ou return(valor)]

}
```

Este último bloco trata-se de uma função que carrega o nome nome_da_funcao. Seu tipo pode ou não ser explicitado. Caso não seja, qualquer função será do tipo int por default.

Esta função não faz parte da função main() e deverá ser chamada em algum momento pelas função main() ou por outra função para seja executados seus comandos. Podemos no momento da chamada da função enviar parâmetros (estudaremos mais sobre as funções a seguir).

No final da função, encontramos o comando return, cujo objetivo é retornar a função que a chamou um valor de retorno da função.

Não é obrigatório o uso do comando return, caso não precisemos retornar nenhum valor na função.

Exemplo de função de subrotina:

```
void funcao(void) {
   PORTB = ~ PORTB;
  return;
}
```

O quarto bloco: A função main()

```
void main([<parâmetros>])
{
  /*
  Este trecho é reservado para o corpo da função, com a declaração de suas
  variáveis locais, seus comandos e funções de trabalho. Aqui estão os
  primeiros comandos que serão executados no programa.
  */
}
```

A função main() será a primeira a ser executada pelo processador. Junto a função main(), temos o modificador de tipo void que tem a função de declarar que a função main() não deve retornar nenhum valor.

A manipulação de programas em linguagem C para microcontroladores é diferente de trabalharmos com programas para PC's, pois nos computadores PC's existem o sistema operacional que receberá o retorno da função main(). Nos microcontroladores, como não possuímos um sistema operacional rodando em paralelo com o programa, não podemos retornar nenhum valor na função main(). Para este caso, devemos sempre iniciar a função main() com o modificador de tipo nulo void.

Exemplo de função main():

```
void main()
{
  PORTB = 0;
  TRISB = 0;
  PORTB = ~PORTB;
  Delay_ms(1000);
}
```

NOTA:
ANTES DE CHAMARMOS QUALQUER FUNÇÃO DO PROGRAMA, COMO POR EXEMPLO EXECUTAR UMA SUBROTINA QUALQUER, DEVEMOS GARANTIR QUE NOSSO COMPILADOR JÁ A RECONHECEU. NO MOMENTO DA COMPILAÇÃO, O PRÉ-COMPILADOR, RESPONSÁVEL PELO RECONHECIMENTO DA ESTRUTURA DO PROGRAMA, NÃO PODE ENCONTRAR UMA CHAMADA DE UMA FUNÇÃO SEM ANTES DE RECONHECE-LA. PARA EVITAR ESSE ERRO DE COMPILAÇÃO, É RECOMENDADO QUE COLOQUEMOS O PROTÓTIPO DE FUNÇÃO NO PROGRAMA A CADA INSERÇÃO DE UMA NOVA FUNÇÃO.

Vamos analisar um exemplo de programa em linguagem C disponível no próprio compilador MikroC como exemplo. Em princípio, não quero que você entenda o funcionamento do programa, mas quero que você identifique os principais blocos de funções desse programa. Vejamos:

```
/*
 * Project name:
   PWM_Test_01 (PWM library Demonstration)
 * Copyright:
   (c) MikroElektronika, 2009.
 * Test configuration:
   MCU:       P18F4520
   Dev.Board:    BIGPIC3
   Oscillator:   HS, 10.0 MHz
   Ext. Modules:  -
   SW:       MikroC PRO v2.50
 * NOTES:
   None.
*/
unsigned short j, oj;

void InitMain()
{
  PORTB = 0;
  TRISB = 0;

  ADCON1 |= 0x0F;
  PORTA = 255;
  TRISA = 255;

  PORTC = 0x00;
  TRISC = 0;
  PWM1_Init(5000);
}//~

void main()
{
  initMain();

  j   = 127;
  oj  = 0;
  PWM1_Start();

  while (1)
  {
   if (Button(&PORTA, 0,1,1))
     j++ ;
   if (Button(&PORTA, 1,1,1))
     j-- ;

   if (oj != j)
   {
     PWM1_Set_Duty(j);
     oj = j;
     PORTB = oj;
   }
   Delay_ms(200);
  }
}//~!
```

Análise da estrutura do programa:

Primeiro bloco: Os comentários

```
/*
 * Project name:
   PWM_Test_01 (PWM library Demonstration)
 * Copyright:
   (c) MikroElektronika, 2009.
 * Test configuration:
   MCU:       P18F4520
   Dev.Board:    BIGPIC3
   Oscillator:   HS, 10.0 MHz
   Ext. Modules:  -
   SW:       MikroC PRO v2.50
 * NOTES:
   None.
*/
```

Este bloco do programa são comentários e são responsável pela parte "documental do programa". No momento da compilação, esses textos serão descartados pelo compilador.

Segundo Bloco: O cabeçalho

```
unsigned short j, oj;
```

Neste trecho do programas é definido o cabeçalho do programa, e nele foi declarado as variável globais do programa.

Terceiro Bloco: as sub-rotinas

```
void InitMain()
{
  PORTB = 0;
  TRISB = 0;

  ADCON1 |= 0x0F;
  PORTA = 255;
  TRISA = 255;

  PORTC = 0x00;
  TRISC = 0;
  PWM1_Init(5000);
}//~
```

Esta função é uma sub-rotina do programa, e foi chamada pelo programador de InitMain();

Quarto Bloco: as função main()

```
void main()
{
  initMain();

  j   = 127;
  oj  = 0;
  PWM1_Start();

  while (1) {
   if (Button(&PORTA, 0,1,1))
     j++ ;
   if (Button(&PORTA, 1,1,1))
     j-- ;

   if (oj != j)
   {
     PWM1_Set_Duty(j);
     oj = j;
     PORTB = oj;
   }
   Delay_ms(200);
  }
}//~!
```

Apesar do função main() ser a última na ordem de escrita do programa, ela será a primeira função a ser executada pelo processador.

Porque devemos criar sub-rotinas ?

Para permitir o reaproveitamento de código já construído(por você ou por outros programadores);

Para evitar que um trecho de código que seja repetido várias vezes dentro de um mesmo programa;

Para permitir a alteração de um trecho de código de uma forma mais rápida. Com o uso de uma função é preciso alterar apenas dentro da função que se deseja;

Para que os blocos do programa não fiquem grandes demais e, por conseqüência, mais difíceis de entender;

Para facilitar a leitura do programa-fonte de uma forma mais fácil;

Para separar o programa em partes (blocos) que possam ser logicamente compreendidos de forma isolada.

Protótipos de Funções

Não podemos usar uma função sem declará-la previamente. Trata-se duma instrução geralmente colocada no início do programa ou do arquivo, obrigatoriamente antecedendo a definição e a chamada da função. O protótipo informa ao compilador o tipo que a função retorna, o nome da função, bem como os parâmetros que ela recebe.

Eis um exemplo:

```
void minha_rotina ();
/*protótipo de função, esta linha de programa deve ser colocada no topo do programa ,
junto com as declarações;*/

void main()
{
  // aqui esta meu programa principal
}

//***************** função de subrotina ********************
void minha_rotina()
{
  //aqui esta os comandos da sua subrotina
}
```

Conforme podemos ver no programa acima, a função de sub-rotina minha_rotina() ficou em depois da função main(), nesse caso, necessariamente, devemos colocar o protótipo de função (linha de programa com o cabeçalho da função de sub-rotina) no topo do programa, caso contrário teremos erro de compilação pois o não podemos utilizar nenhuma função antes de declará-la. No caso de colocarmos as funções de sub-rotinas antes da função main() no programa, não precisamos declarar os protótipos de funções no programa.

Os identificadores

A linguagem C define identificadores como sendo nomes usados para se fazer referência a entidades do programa (variáveis, funções, rótulos, etc.) definidas pelo programador. Em C, um identificador é composto de um ou mais caracteres, sendo que, para identificadores internos, os 31 primeiros são significativos. O primeiro caractere deve ser uma letra ou um sublinha (_) e os caracteres subseqüentes devem ser letras, números ou sublinhas. Eis aqui alguns exemplos de identificadores corretos e incorretos:

Corretos	Incorretos	Descrição
cont	1cont	Um identificador deve sempre começar com uma letra do alfabeto, nunca por um número.
valor23	alô	Um identificador não pode conter acentos (´, ^, ~,)
totalGeral	valor-total	Um identificador não pode conter simbolos (-)

Isto quer dizer que se duas variáveis têm em comum os 31 primeiros caracteres e diferem apenas a partir do trigésimo segundo, o compilador C não será capaz de distingui-Ias. Por exemplo, esses dois identificadores são iguais:

isto_e_um_exemplo_de_um_nome_longo

isto_e_um_exemplo_de_um_nome_longo_tambem

Capítulo 7 – Estruturas de Controle

As estruturas de controle são usadas nos programas em lugares em que necessitamos que a máquina faça repetidas operações ou necessite de tomadas de decisão.

O comando if (SE)

O if (SE, em português) é uma estrutura de tomada de decisão baseada no resultado lógico de um teste. Sua forma geral é:

```
if (condição) comando;
```

ou

```
if (condição) { blocos de comandos};
```

Em C, qualquer número diferente de zero é tido como verdadeiro. O que significa que uma condição em C só é falsa, quando os resultados dos operadores que aplicados for zero (0). Qualquer outro valor, mesmo negativo, é considerado como verdadeiro ou não zero.

No comando If, caso a condição seja verdadeira (diferente de zero), o comando, ou blocos de comandos serão executados, caso contrário, quando a condição for falsa (0 zero) o comando será ignorado.

Exemplo:

```
void main() {
  char a = 10;
  char b = 0 ;
  if (a) b--;
}
```

A variável "a" foi atribuída o valor 10, que é diferente de zero (0). Conseqüentemente, o comando if será executado e a condição de decremento da variável b será executado;

```
if (a) b--;
```

também é possível executar outro comando ou bloco de comandos no caso da condição ser avaliada como falsa, para isso utilizamos os recursos do comando if - else.

Acompanhe seu formato geral:

```
if (expressão)
  comando 1
else
  comando 2
```

Exemplo:

```
if (x > 5)
  z = 10;
else
  z = 20;
```

A linha de comando if (x > 5) tem por função verificar se a condição x>5 é verdadeira, caso seja verdadeira, a linha de programa z = 10 será executada. Caso a condição do comando IF seja falsa, ou seja, se "x" for menor que 5, então a linha de comando z = 20 será executada.

Todos os operadores do C podem ser incluídos no teste de uma condição, expressões válidas da linguagem C podem ser avaliadas para a geração de um resultado booleano na estrutura if. As expressões seguintes são válidas e seus resultados dependerão do valor de suas variáveis. Exemplo:

```
if (a > b) b = a;  /* se a expressão a > b for verdadeira, a variável "b" assume o valor da "a"*/
if (b < a) b = 10;  /* se a expressão a < b for verdadeira, a variável "b" assume o valor 10*/
if (a != b) b = 55;  /* se a expressão a !=b (diferente) for verdadeira, a variável "b" assume o valor 55*/
if ( (a + b) > (b + d) ) b++;  /* se a expressão (a + b) > (b + d) for verdadeira, a variável "b" será incrementada em uma unidade.*/
```

Observações importantes:

O operador de atribuição de igualdade (=) em C é diferente do operador relacional de igualdade (==). Para testar a condição de uma variável qualquer, utilizamos o operador relacional de igualdade (==). Acompanhe:

```
if (a == b) c = 10;
```

é diferente de

```
if (a = b) c = 10;
```

A condição (a == b) faz uma comparação entre as duas variáveis, caso a condição seja verdadeira, a variável "C" assumirá o valor 10;

O comando if seguinte, possui uma expressão de atribuição de igualdade (a = b). O compilador ao compilar este comando, irá primeiramente atribuir a variável "a" o valor contido em "b", e depois disso, verificará se a condição não é zero, caso seja verdadeiro o resultado (diferente de zero), a variável "c" assumirá o valor 10;

No comando if para adicionarmos blocos de programas, faz necessário o uso dos símbolos abre-chaves ({) e fecha-chaves (}).

Exemplo:

```
if ( PORTB == PORTC)
{              //blocos de comandos
  a++;
   PORTB.F0 = 1;
  d = c + e;
}
```

A estrutura if, else, if

Podemos criar arranjos (nesting) de comandos através da estrutura if, else, if. Os arranjos são criados bastando colocar estruturas if aninhadas com outras estruturas if.

Seu formato geral é:

```
if (condição1) declaração 1; else
   if (condição2) declaração 2; else
     if (condição3) declaração 3; else
       if (condição4) declaração 4;
```

Exemplo:

```
void main()
{

  int contador = 10;
  int sinal = 5;

  if (contador > sinal) sinal++; else
   if (contador < sinal) sinal--; else
     if (contador == sinal) sinal = 0;
}
```

Podemos também executar blocos de comandos utilizando os recursos das chaves ({ e }).

Exemplo:

```
void main()
{
  int contador = 10;
  int sinal = 5;

  if (contador > sinal)
  { /*utilização de blocos de comando na estrutura if-else-if*/
   sinal++;
   contador = 20;
  }
  else if (contador < sinal)
  {
   sinal--;
   contador = 15;
  }else if (contador == sinal)
   sinal = 0;
}
```

A estrutura if é uma das mais utilizadas para tomada de decisões lógicas. Iremos utilizá-la em vários programas que desenvolveremos durante nosso curso.

O comando switch

O comando switch é uma forma mais clara e elegante de tomar decisão dentro de um programa em C. Diferente do comando if, a estrutura switch não aceita expressão para a tomada de decisão, mas só aceita constante.

```
switch (variável)
{
  case constante1:
   declaração1A;
   declaração1B;
   declaração1N;
   break;
  case constante2:
   declaração2A;
   declaração2B;
   declaração2N;
   break;
  default;
   declaração_default;
}
```

O valor da variável no comando switch é comparada contra as constantes especificadas pela cláusula case. Caso a variável e a constante possuam valores iguais, os comandos seguinte a cláusula case serão executados. Caso não tenha nenhuma constante com o mesmo valor da variável, então os comandos especificados pela cláusula default serão executados.

Acompanhe o exemplo:

```
void main()
{
  int contador = 10;
  int sinal = 5;

  switch(contador)
  {
   case 2:
     sinal++;
     break;
   case 1:
     sinal = 2;
     break;
   case 10:
     contador--;
     break;
   default:
     sinal = 0;
  }
}
```

No exemplo de acima, a variável contador será comparada às constantes 2, 1 e 10. Como a variável contador possui o valor 10, conseqüentemente o comando que será executado no exemplo acima é case 10:

contador--; (decrementa a variável contador).

A cláusula break possui a função de encerrar uma seqüência de comandos de uma cláusula case.

A cláusula default é o último comando switch.

Capítulo 8 – Estruturas de Repetição

Os laços de repetição servem para repetir uma ou mais vezes determinada instrução ou blocos de instruções. Existem basicamente três tipos de estruturas de repetição na linguagem C:

- for
- while
- do - while

A estrutura for basicamente é utilizada para laços finitos de contagem, normalmente utilizando uma variável de controle da contagem.

A estrutura while basicamente é utilizado para repetição de um determinado conjunto de instrução enquanto uma condição for verdadeira.

O comando do - while é similar à estrutura while, diferenciando apenas o momento que é analisada a condição.

Vamos conhecer sobre cada estrutura de repetição, acompanhe:

O comando For

O laço for é utilizado quando necessitamos de um ciclo de repetições controlado, pois em sua declaração podemos inicializar e incrementar ou decrementar a variável de controle. Antes de serem executados os comandos do laço For-next, primeiramente é avaliada a condição do teste. Caso seja verdadeira, são executados os comandos do laço. A cada ciclo que é executado o faço for, a variável de controle será incrementada ou decrementada no valor programado no incremento.

Veja a estrutura do comando For :

```
for (inicialização ; condição ; incremento/decremento ) comando;
```

ou

```
for (inicialização ; condição ; incremento/decremento)
{
  ...
  comandoA1;
  comandoA2;
  ...
}
```

em que:

inicialização: expressão válida utilizada normalmente para inicialização da variável de controle do laço.

condição: condição para que decide pela continuidade ou não do laço de repetição, enquanto esta condição for verdadeira, o laço for permanecerá em execução.

Incremento/decremento: "valor incrementado em" a cada repetição do laço for.

Exemplo:

```
int contador;
for (contador = 0 ; contador = 10 ; contador++ )  PORTB = contador;
```

Na estrutura anterior, a variável contador inicialmente é carregada com o valor 0 (zero), e os comandos do laço for são executados. Após a execução de todos os comandos presentes no laço for, a variável contador é incrementada no passo do valor do incremento, que no nosso exemplo é 1 unidade, e novamente os comandos do laço for são executados. A estrutura de repetição for termina quando a variável contador assume o valor 10.

Exemplo:

```
int v, d = 10;
for (v = 0 ; v <= d ; v++)
{
  Comando A;
  Comando B;
}
```

Este exemplo funciona da seguinte maneira:

A variável V recebe o valor 0.
A variável V é comparada com d
O comando A e o comando B são executados.
A variável V tem seu valor incrementado em 1 unidade.
Após esta incrementação, o valor de V é comparado com o valor D, e a seqüência retorna ao item 3, para nova repetição.
Caso contrário (V ultrapassou D), o laço de repetição for é finalizado.

O grande beneficio do laço for é sua flexibilidade, pois aceita qualquer expressão válida em C, mesmo que essas expressões não tenham relacionamento com o laço de repetição diretamente.

Um exemplo simples de aplicação do laço for é a criação de pequenos tempos de atrasos (delays).

```
int atraso;
for (atraso = 0 ; atraso < 1000 ; atraso++);
```

O laço for acima faz com que o processamento fique incrementando constantemente a variável atraso em 1 até que esta variável seja igual ao valor 1000;

Observe o fragmento de programa seguinte válido dentro do loop for:

```
for (n = 0 ; (n < 100) &&  PORTB.F0 ; n++)
```

No exemplo acima a variável n será incrementada 100 vezes, desde que o pino RB0 permaneça em estado lógico alto. Se o pino RB0 em algum momento do laço cair para nível lógico baixo, o loop será imediatamente encerrado.

Programa Exemplo:

Aproveitando que estamos estudando as estruturas do laço de repetição for, vamos elaborar um programa que pisca um led conectado ao pino RD0 do PIC utilizando os recursos do laço for.

Acompanhe o esquema abaixo:

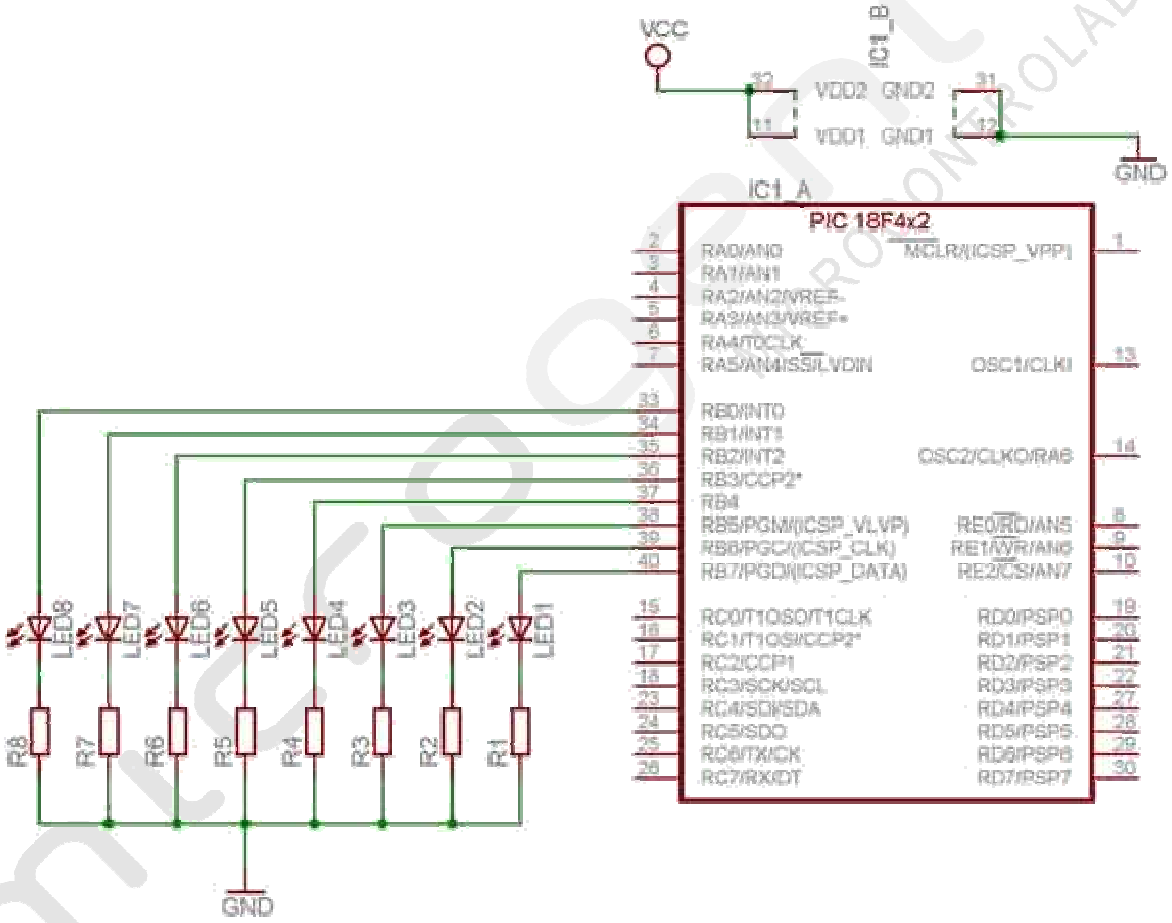


Figura 01 – Esquema de ligação dos leds no Kit PICgenios

O programa seguinte vai contar até 100.000. Para acomodar um número dessa grandeza poderíamos ter utilizado uma variável long, mas optamos pela variável int para que você entenda a construção de dois laços for encadeados. Para contar até 100,000 é necessário termos uma variável que conte até 100 e outra até 1000 (100 * 1000 = 100.000).

```
//*********************************************************
// programa de perda de tempo utilizando o laço de repetição for.
//*********************************************************

void atraso(void)  // rotina de perda de tempo
{
  int contador1, contador2;
  for (contador1 = 0 ; contador1 < 100 ; contador1++) //laço de repetição for
  for (contador2 = 0; contador2 < 1000; contador2++) ;
}

void main()
{
  TRISD = 0;
  while(1)
  {
   atraso();         //chama rotina de perda de tempo
   PORTD = ~PORTD;   // inverte os estados do PORTD do PIC
  }
}
```

O laço while

Muitos dos programadores iniciantes não sabem do que iremos comentar agora.

Os microcontroladores no geral não possuem sistemas operacionais, os programas que desenvolvemos para os microcontroladores PIC devem rodar eternamente, ou possuírem mecanismos que paralise a operação da CPU em alguns momentos.

Isso é necessário, pois, por não possuir sistema operacional, o programa do PIC não tem para onde sair, devendo ficar em operação, de preferência eterna. Esses é um dos motivos pelos quais a função principal main() dos programas para microcontroladores deve ser do tipo void (valor nulo), pois não tem quem receba os valores de retorno da função principal main().

O comando while é o comando ideal criar loops infinitos no nosso programa.

A estrutura while verifica inicialmente se a condição de teste é verdadeira. Em caso verdadeiro, todos os comandos dentro do laço while serão executados. Ao retornar para o início do laço, é verificado novamente se a condição de teste é verdadeira, se for verdadeira, executa novamente os blocos de comandos; se for falsa, o laço while é finalizado.

Acompanhe o exemplo abaixo:

```
void main()
{
  while (1) /* condição a ser testada. Neste exemplo a condição sempre será verdadeira
(1);*/
  {
   declaração 1;
   declaração 2;
   declaração n;
  }
}
```

Veja outro exemplo de programa:

```
void main()
{
  int a = 25;

  while (a = 5)
  {
    a++;
    PORTB = a;
  }
}
```

Repare que no exemplo acima o valor a condição do laço while é falsa, neste caso os comandos do laço não serão executados no programa até que a condição seja verdadeira, ou seja, até que a = 5.

O laço do - while

O comando do - while forma uma estrutura de repetição diferente dos comandos while e for estudado anteriormente. Sua diferença fundamental com relação as outras tradicionais laços de loop while e for está no fato da avaliação da condição de teste ser no final do laço de repetição, contrário dos outros laços que estudamos, que testam as condições no início de cada ciclo.

formato geral:

```
do comando while (condição);
```

ou

```
do
{
  comando 1;
  comando 2;
} while (condição de teste);
```

Na estrutura do-while a condição de teste é efetuada somente na parte final do loop. Isso faz com que as instruções contidas no interior do laço do - while sejam executadas ao menos uma vez. Caso a condição teste seja atendida, o laço de repetição é finalizado, caso contrário o bloco que está na estrutura seria novamente executado até que a condição teste seja atendida.

Exemplo:

```
void main()
{
  int a = 0;       //declara a variável a como inteiro com o valor 0
  do
  {
   a++;              //incrementa a variável a em uma unidade
   PORTD = ~PORTD;   //inverte os estados do PORTD
  }
  while(a > 100);     // enquanto a < 100 permanece no loop do-while;
}
```

O comando break e continue

Break

O comando break, na linguagem C, é utilizado para interromper (quebrar) qualquer laço de repetição instantaneamente. No momento da execução do loop, ao encontrar o comando break, o laço de repetição é finalizado.

Exemplo:

```
void main()
{
  int a = 0;
  do
  {
   a++;               //incrementa a variável A em uma unidade
   break;             //interrompe a execução do laço de repetição do-while
   PORTD = ~PORTD;    //inverte os estados dos PORTD
  }while(a < 100);    /*enquanto a variável a for menor que 100 a rotina do-while será
executada.*/
}
```

O comando break é válido para os laços do-while, while, for e case.

Continue

O comando continue reinicia novamente o teste da condição do laço de repetição. Toda a vez que no laço for encontrado o comando continue, os comandos seguintes não serão executados no laço.

Exemplo:

```
void main()
{
  int a = 0;
  do
  {
   a++;                //incrementa a variável A em uma unidade
   continue;           //retorna para o início do laço de repetição
   PORTD = ~PORTD;     //inverte os estados dos PORTD
  } while(a < 100);    /*enquanto a variável a for menor que 100 a rotina do-while será
executada*/.
 }
```

No exemplo de programa acima, os comandos abaixo de continue não serão executados, pois, toda a vez que for executado o comando continue, automaticamente o programa será redirecionado para o cabeçalho do laço de repetição independente de qualquer estado de variáveis.

O comando continue é valido para os laços do-while, while, for e case.

Capítulo 9 – Padrões de Formatação de Código

Nesse capitulo iremos sugerir algumas formas de padronização de código, essa padronização não pertence ao ANSI-C, mas facilita, em muito, a programação e leitura de seus códigos.

Declaração de variáveis:

Nas variáveis podemos declarar desta forma:

[modificador] [tipo] [letra_do_modificador][letra_do_tipo][nome_variavel];

EX:

```
unsigned char ucVariavel;
char scVariavel; // mesmo que signed char
```

Declaração de funções:

Podemos utilizar a mesma formatação para declaração de funções, excluindo a função main():

[modificador] [tipo] [letra_do_modificador][letra_do_tipo][nome_funcao] [parametros];

EX:

```
void vFuncao1(unsigned char ucContador)
{
// comandos
}

unsigned char ucFuncao2()
{
// comandos
}
```

Identação:

Um dos recursos mais importantes e necessarios para a escrita de um programa legível, ela nada mais é que o uso de recuos no seu código para identificação de blocos de código.

Em nosso curso utilizaremos o recuo de 3 espaços.

Ex:

```
unsigned char ucFuncao2()
{
   if (ucContador==1) // Recuo de 3 espaços
   {
      ucContador++;   // Recuo de 3 espaços em relação ao anterior
   }
   // comandos
}
```

Capítulo 10 – ANEXOS

TABELA ASCII

Dec	Binário	Hex	ASCII
1	0b00000001	0x01	SOH
2	0b00000010	0x02	STX
3	0b00000011	0x03	ETX
4	0b00000100	0x04	EOT
5	0b00000101	0x05	ENQ
6	0b00000110	0x06	ACK
7	0b00000111	0x07	BEL
8	0b00001000	0x08	BS
9	0b00001001	0x09	HT
10	0b00001010	0x0A	LF
11	0b00001011	0x0B	VT
12	0b00001100	0x0C	FF
13	0b00001101	0x0D	CR
14	0b00001110	0x0E	SO
15	0b00001111	0x0F	SI
16	0b00010000	0x10	DLE
17	0b00010001	0x11	D1
18	0b00010010	0x12	D2
19	0b00010011	0x13	D3
20	0b00010100	0x14	D4
21	0b00010101	0x15	NAK
22	0b00010110	0x16	SYN
23	0b00010111	0x17	ETB
24	0b00011000	0x18	CAN
25	0b00011001	0x19	EM
26	0b00011010	0x1A	SUB
27	0b00011011	0x1B	ESC
28	0b00011100	0x1C	FS
29	0b00011101	0x1D	GS
30	0b00011110	0x1E	RS
31	0b00011111	0x1F	US
32	0b00100000	0x20	SPC

Dec	Binário	Hex	ASCII
65	0b01000001	0x41	A
66	0b01000010	0x42	B
67	0b01000011	0x43	C
68	0b01000100	0x44	D
69	0b01000101	0x45	E
70	0b01000110	0x46	F
71	0b01000111	0x47	G
72	0b01001000	0x48	H
73	0b01001001	0x49	I
74	0b01001010	0x4A	J
75	0b01001011	0x4B	K
76	0b01001100	0x4C	L
77	0b01001101	0x4D	M
78	0b01001110	0x4E	N
79	0b01001111	0x4F	O
80	0b01010000	0x50	P
81	0b01010001	0x51	Q
82	0b01010010	0x52	R
83	0b01010011	0x53	S
84	0b01010100	0x54	T
85	0b01010101	0x55	U
86	0b01010110	0x56	V
87	0b01010111	0x57	W
88	0b01011000	0x58	X
89	0b01011001	0x59	Y
90	0b01011010	0x5A	Z
91	0b01011011	0x5B	[
92	0b01011100	0x5C	\
93	0b01011101	0x5D	]
94	0b01011110	0x5E	^
95	0b01011111	0x5F	_
96	0b01100000	0x60	`

Dec	Binário	Hex	ASCII
129	0b10000001	0x81	□
130	0b10000010	0x82	‚
131	0b10000011	0x83	ƒ
132	0b10000100	0x84	„
133	0b10000101	0x85	…
134	0b10000110	0x86	†
135	0b10000111	0x87	‡
136	0b10001000	0x88	ˆ
137	0b10001001	0x89	‰
138	0b10001010	0x8A	Š
139	0b10001011	0x8B	‹
140	0b10001100	0x8C	Œ
141	0b10001101	0x8D	□
142	0b10001110	0x8E	Ž
143	0b10001111	0x8F	□
144	0b10010000	0x90	□
145	0b10010001	0x91	‘
146	0b10010010	0x92	’
147	0b10010011	0x93	“
148	0b10010100	0x94	”
149	0b10010101	0x95	•
150	0b10010110	0x96	–
151	0b10010111	0x97	—
152	0b10011000	0x98	˜
153	0b10011001	0x99	™
154	0b10011010	0x9A	š
155	0b10011011	0x9B	›
156	0b10011100	0x9C	œ
157	0b10011101	0x9D	□
158	0b10011110	0x9E	ž
159	0b10011111	0x9F	Ÿ
160	0b10100000	0xA0	

Dec	Binário	Hex	ASCII
193	0b11000001	0xC1	Á
194	0b11000010	0xC2	Â
195	0b11000011	0xC3	Ã
196	0b11000100	0xC4	Ä
197	0b11000101	0xC5	Å
198	0b11000110	0xC6	Æ
199	0b11000111	0xC7	Ç
200	0b11001000	0xC8	È
201	0b11001001	0xC9	É
202	0b11001010	0xCA	Ê
203	0b11001011	0xCB	Ë
204	0b11001100	0xCC	Ì
205	0b11001101	0xCD	Í
206	0b11001110	0xCE	Î
207	0b11001111	0xCF	Ï
208	0b11010000	0xD0	Ð
209	0b11010001	0xD1	Ñ
210	0b11010010	0xD2	Ò
211	0b11010011	0xD3	Ó
212	0b11010100	0xD4	Ô
213	0b11010101	0xD5	Õ
214	0b11010110	0xD6	Ö
215	0b11010111	0xD7	×
216	0b11011000	0xD8	Ø
217	0b11011001	0xD9	Ù
218	0b11011010	0xDA	Ú
219	0b11011011	0xDB	Û
220	0b11011100	0xDC	Ü
221	0b11011101	0xDD	Ý
222	0b11011110	0xDE	Þ
223	0b11011111	0xDF	ß
224	0b11100000	0xE0	à

33	0b00100001	0x21	!
34	0b00100010	0x22	"
35	0b00100011	0x23	#
36	0b00100100	0x24	$
37	0b00100101	0x25	%
38	0b00100110	0x26	&
39	0b00100111	0x27	'
40	0b00101000	0x28	(
41	0b00101001	0x29	)
42	0b00101010	0x2A	*
43	0b00101011	0x2B	+
44	0b00101100	0x2C	,
45	0b00101101	0x2D	-
46	0b00101110	0x2E	.
47	0b00101111	0x2F	/
48	0b00110000	0x30	0
49	0b00110001	0x31	1
50	0b00110010	0x32	2
51	0b00110011	0x33	3
52	0b00110100	0x34	4
53	0b00110101	0x35	5
54	0b00110110	0x36	6
55	0b00110111	0x37	7
56	0b00111000	0x38	8
57	0b00111001	0x39	9
58	0b00111010	0x3A	:
59	0b00111011	0x3B	;
60	0b00111100	0x3C	<
61	0b00111101	0x3D	=
62	0b00111110	0x3E	>
63	0b00111111	0x3F	?
64	0b01000000	0x40	@

97	0b01100001	0x61	a
98	0b01100010	0x62	b
99	0b01100011	0x63	c
100	0b01100100	0x64	d
101	0b01100101	0x65	e
102	0b01100110	0x66	f
103	0b01100111	0x67	g
104	0b01101000	0x68	h
105	0b01101001	0x69	i
106	0b01101010	0x6A	j
107	0b01101011	0x6B	k
108	0b01101100	0x6C	l
109	0b01101101	0x6D	m
110	0b01101110	0x6E	n
111	0b01101111	0x6F	o
112	0b01110000	0x70	p
113	0b01110001	0x71	q
114	0b01110010	0x72	r
115	0b01110011	0x73	s
116	0b01110100	0x74	t
117	0b01110101	0x75	u
118	0b01110110	0x76	v
119	0b01110111	0x77	w
120	0b01111000	0x78	x
121	0b01111001	0x79	y
122	0b01111010	0x7A	z
123	0b01111011	0x7B	{
124	0b01111100	0x7C	|
125	0b01111101	0x7D	}
126	0b01111110	0x7E	~
127	0b01111111	0x7F	□
128	0b10000000	0x80	€

161	0b10100001	0xA1	¡
162	0b10100010	0xA2	¢
163	0b10100011	0xA3	£
164	0b10100100	0xA4	¤
165	0b10100101	0xA5	¥
166	0b10100110	0xA6	¦
167	0b10100111	0xA7	§
168	0b10101000	0xA8	¨
169	0b10101001	0xA9	©
170	0b10101010	0xAA	ª
171	0b10101011	0xAB	«
172	0b10101100	0xAC	¬
173	0b10101101	0xAD	
174	0b10101110	0xAE	®
175	0b10101111	0xAF	¯
176	0b10110000	0xB0	°
177	0b10110001	0xB1	±
178	0b10110010	0xB2	²
179	0b10110011	0xB3	³
180	0b10110100	0xB4	´
181	0b10110101	0xB5	µ
182	0b10110110	0xB6	¶
183	0b10110111	0xB7	·
184	0b10111000	0xB8	¸
185	0b10111001	0xB9	¹
186	0b10111010	0xBA	º
187	0b10111011	0xBB	»
188	0b10111100	0xBC	¼
189	0b10111101	0xBD	½
190	0b10111110	0xBE	¾
191	0b10111111	0xBF	¿
192	0b11000000	0xC0	À

225	0b11100001	0xE1	á
226	0b11100010	0xE2	â
227	0b11100011	0xE3	ã
228	0b11100100	0xE4	ä
229	0b11100101	0xE5	å
230	0b11100110	0xE6	æ
231	0b11100111	0xE7	ç
232	0b11101000	0xE8	è
233	0b11101001	0xE9	é
234	0b11101010	0xEA	ê
235	0b11101011	0xEB	ë
236	0b11101100	0xEC	ì
237	0b11101101	0xED	í
238	0b11101110	0xEE	î
239	0b11101111	0xEF	ï
240	0b11110000	0xF0	ð
241	0b11110001	0xF1	ñ
242	0b11110010	0xF2	ò
243	0b11110011	0xF3	ó
244	0b11110100	0xF4	ô
245	0b11110101	0xF5	õ
246	0b11110110	0xF6	ö
247	0b11110111	0xF7	÷
248	0b11111000	0xF8	ø
249	0b11111001	0xF9	ù
250	0b11111010	0xFA	ú
251	0b11111011	0xFB	û
252	0b11111100	0xFC	ü
253	0b11111101	0xFD	ý
254	0b11111110	0xFE	þ
255	0b11111111	0xFF	ÿ
0	0b00000000	0x00	NULL