Linguagem ´C´ para microcontroladores PIC

VIDAL Projetos Personalizados

Eng. Vidal Pereira da Silva Júnior

Última revisão: 25/11/1999

Objetivo:

Iniciar o projetista no uso da linguagem C para sistemas dedicados baseados na família PIC 16CXXX.

Metodologia:

O curso esta dividido em 3 partes:

- Iniciação à Linguagem C
- Implementação da CCS para a linha microchip
- Exemplos práticos de hardware e software

Parte I - Iniciação à linguagem C

A principal vantagem de se usar linguagens de alto nível (no nosso caso a linguagem C) esta na menor interação do projetista com o hardware, no que diz respeito ao controle do mesmo (ajuste de bancos de registradores, seqüências de inicialização, ...).

Desta forma o projetista dedica seu tempo basicamente à lógica do problema e não aos detalhes internos do chip.

Como exemplo vamos ver a seqüência de escrita na EEPROM do 16F84 (gravar no endereço 20H o valor 12H, pág. 27 do livro):

Em assembler:

```
bcf     STATUS,RP0
movlw   20H
movwf   EEADR
movlw   12H
movwf   EEDATA
bsf     STATUS,RP0
bcf     INTCON,GIE
bsf     EECON1,WREN
movlw   55H
movwf   EECON2
movlw   0AAH
movwf   EECON2
bsf     EECON1,WR
```

Em ‘C’

```
write_eeprom(0x20,0x12);
```

I.1 - Modelo básico de um programa em C

Quatro elementos estão presentes em um programa C:

- Comentários
- Diretivas de compilação
- Definições de dados
- Blocos com instruções e funções

Modelo básico

```
#include <....>                          Diretivas de compilação
#fuses ......


// comentário ocupando uma linha         Comentários


/* comentários entre '/ *'  e  '* /'
   podem ocupar mais de
   uma linha
*/


char i ,  j ;                            Variáveis
float Tempo;


void main()
{
   instruções do programa principal
}


void delay()
{
   instruções da função (rotina) delay
}
```

I.2 - Comentários

Comentários são informações anexadas ao programa fonte (*) que permitem ao programador ou outros entenderem o significado do que esta sendo feito.

É boa prática comentar todas as linhas, pois após um certo tempo, nem mesmo o criador do programa lembrará de tudo o que estava pensando no momento da escrita.

O compilador ignora tudo que estiver definido como comentário

(*) O programa fonte em C deve ter terminação ".C"

Exemplo: teste.c

Existem dois tipos de comentários:

- Comentários que ocupam apenas 1 linha

Este tipo de comentário é iniciado com duas barras conjuntas: ***//***

Neste caso, tudo que estiver após as duas barras será ignorado pelo compilador.

Exemplo:

```
x = x + 2;              // soma 2 à variável x
```

- Comentários com múltiplas linhas

Este tipo de comentário é iniciado com a seqüência /* e finalizado pela seqüência */.

Neste caso, tudo que estiver ENTRE estas duas seqüências será ignorado pelo compilador. Ideal para excluir temporariamente trechos de código.

Exemplo:

```
        x = x + 2;
/*      tempo++;
        a = SQRT(25);           */
        x = 0;
```

No exemplo acima, as linhas tempo++; e a=SQRT(25); serão ignoradas no momento da compilação.

I.3 - Diretivas de compilação

São instruções para o compilador, e não para o programa que será gerado.

As diretivas informam, por exemplo, o processador para o qual o código deverá ser gerado, o valor do clock que será usado pela cpu,..).

As diretivas sempre começam com ‘ # ’.

Um bom exemplo é a diretiva que inclui no processo de compilação as definições do chip.

#include <16F84.H>

A terminação .H indica um Header File da linguagem C, ou seja, um cabeçalho. Equivale ao P16F84.INC usado no assembler.

I.4 - Indicador de fim de instrução

O compilador C não é um compilador de linha, como o assembler.

O compilador C procura o sinal de que a instrução ou o bloco de instruções já acabou.

Este sinal é o “ponto e virgula” para uma instrução ou o ‘ } ‘ para o bloco (mais tarde falaremos sobre blocos de instruções).

No exemplo abaixo, as duas maneiras são corretas, pois o ‘ ; ‘ é que sinaliza o fim da instrução.

```
x = x + 25;
```

```
x =
x + 25
;
```

I.5 - Definição de variáveis, constantes e identificadores

Todas as variáveis e constantes usadas no programa devem ser devidamente definidas, com um nome e um tipo. O mesmo vale para identificadores de funções e rotinas.

Os dados básicos podem ser de 8, 16 e 32 bits de comprimento, e devido as características peculiares dos microcontroladores, variáveis de 1 bit também podem ser definidas.

Variáveis:	tamanho	faixa
• variável de 8 bits:	CHAR	(de 0 à 255)
• variável de 16 bits:	LONG INT	(de 0 à 65535 *)
• variável de 32 bits:	FLOAT	(_____)
• variável de 1 bit:	SHORT	(pode assumir 0 ou 1)

(*) Variáveis com valores negativos:

As variáveis do tipo LONG INT podem ou não ter sinal negativo. Para garantir que a variável seja sempre positiva, usamos *unsigned* antes da definição da mesma. Caso seja necessário tratar números negativos, usaremos *signed*.

```
Ex.:  unsigned long int  tempo;     // tempo irá de 0 à 65535
      signed     long int espaco;   // espaco irá de -32768 à 32767
```

Observe que o indicativo *signed* diminui o alcance da variável pela metade.

O padrão da linguagem, quando não indicado, é o tipo UNSIGNED.
CHAR é sempre UNSIGNED.
FLOAT não admite os indicativos mas pode ser positivo ou negativo.

I.5.1 - Seqüência de declaração de variáveis e constantes

Para se declarar uma variável temos a seguinte ordem das definições:

```
Tipo    Nome
char    tempo;    // a variável 'tempo' vai de 0 à 255
```

Um grupo de variáveis de mesmo tipo pode ser declarada na mesma linha.

```
char  i, j, k;    // declara que i, j e k são do tipo char.
```

Constantes:

CONST define um 'label' para valores que não serão alterados pelo programa:

Exemplo: FLOAT CONST PI = 3.14;

I.5.2 - Como escrever os nomes de variáveis, constantes e funções

Todo 'label' (nome ou identificador), seja de variável, constante ou função, deve começar por letra, ter no máximo 32 caracteres e não usar caracteres especiais ou de controle (! \ ? % ....).

Nomes de funções e rotinas internas também não podem ser utilizados.

- Exemplos de definições:

Corretas	Incorretas	
Teste	0local	*começa com número*
teste	parte!dois	
TESTE	^	
_12A	*caracter inválido* (!)	
x_2_5		

IMPORTANTE: O compilador diferencia letras minúsculas de maiúsculas.

Nos exemplos acima, Teste, teste e TESTE são 3 variáveis diferentes para o compilador.

Uma boa prática esta em diferenciar variáveis, labels e registros com tipos diferentes de letras.

Exemplo:	PORTB	registro da CPU
	TempoMs	Rotina de tempo em ms
	quantos	variável em RAM

I.5.3 - Atribuindo valores

A atribuição de valores é feita pelo sinal de igualdade,

```
tempo = 123;
```

aqui a variável tempo passa a ter o valor 123 decimal.

I.5.4 - IMPORTANTE Valores iniciais das variáveis

Devemos sempre ajustar os valores iniciais de cada variável do programa antes de usa-las, pois nada garante que estão em "0 " no reset.

I.5.5 - Variáveis locais e variáveis globais

Existem duas maneiras principais para a alocação de variáveis em memória:

Variáveis Globais:

As variáveis globais tem “alcance” para todas as rotinas e funções existentes no programa.
Isto significa que qualquer rotina ou função poderá utiliza-la.

Variáveis Locais:

As variáveis locais tem “alcance” apenas dentro das rotinas ou funções onde foram criadas.
Isto significa que apenas a rotina ou função onde a mesma foi criada poderá utiliza-la.

Vamos ver um fragmento de código como exemplo:

```
char  A,B,C;              // A, B e C são variáveis globais, pois estão fora de
                          // qualquer função ou rotina.
main ()
{
        A = 0;            // Faz A=0
        Tempo();          // chama rotina Tempo
        Espera();         // chama rotina Espera
}

Tempo()
{
char  J;                  // cria uma variável chamada J.
                          // Somente a rotina Tempo() poderá utiliza-la
        J = A;
}

Espera()
{
char  X;                  // cria uma variável chamada X.
                          // Somente a rotina Espera() poderá utiliza-la
        X = A;
        X = J;            //  Erro:  A variável J não existe para esta rotina
}
```

A principal vantagem no uso das variáveis locais esta em que a memória RAM ocupada pela mesma será compartilhada por outras variáveis locais, permitindo um “aumento” no espaço de memória.

I.6 - Funções, Procedimentos e Rotinas

Chamamos função um trecho de programa que realiza determinada operação, bem como as funções pré-definidas (SQRT, ABS, ACOS, ...).

A principal característica da função é a capacidade de retornar um valor ao ponto de chamada da mesma.

Procedimentos são trechos de programa que não retornam valor.

Podemos as vezes nos refrir as funções e aos procedimentos como 'rotinas'.

No exemplo abaixo, temos a função MAIN (falaremos dela depois) que é a rotina executada no reset do programa. A função MAIN chama uma ROTINA para ler teclas, e a rotina de ler teclas chama uma FUNÇÃO para calcular o valor absoluto da tecla.

```
main ( )
{
        ---
        LeTeclas( );
        ---
}

LeTeclas( )
{
        ---
        X = ABS(tecla_lida);
        ---
}
```

I.6.1 - Funções especialmente desenvolvidas para os PIC's

Na parte II, referente ao compilador da CCS, veremos quais as funções que existem para os PIC's e como funcionam.

I.6.2 - Retorno de funções

A idéia básica da função esta em retornar um valor para o ponto de chamada. Todas as funções não definidas como VOID (*) retornam um valor.

Para que o valor seja retornado, devemos usar a instrução **return** seguida do *valor a ser retornado*, que encerra a função e faz com que o programa volte ao ponto de origem.

Uma função pode ter vários **return**, mas somente um será executado. Podemos assim criar vários pontos de *saída* da função, um para cada situação.

Se não colocarmos o **return**, o valor retornado sera **"0"**.

(*) Veremos os tipos de função depois

Exemplo: Função para dividir dois numeros. Se o divisor for “0”, retorna “0”

```
        main( )
        {
                .....
                x = 100;                    // fax x = 100
                y = getchar( );             // faz y = ao valor recebido via serial
                                            // para o programa o valor recebido é
                                            // indetermionado
                valor = DIVID (x,y);        // chama a funcao DIVID e ao retornar atribui a
                                            // ‘valor’  o resultado da divisão
                if ( valor == 0 ) { . . .  }    //  ação caso y = 0
                .....
        }

long int DIVID ( long int A , B)
{
        if ( B == 0 ) return (0);       // se B=0 retorna 0,
        else return ( A / B );          // senao retorna a divisão
}
```

OBSERVAÇÃO:

Este programa da forma acima é apenas ilustrativo. A listagem apresentada ao final nos exemplos esta completa, com todos os detalhes para permitir a compilação.

I.6.3 - Variáveis como parâmetros

Outra facilidade da linguagem C esta na passagem de valores de uma rotina ou função para outra.

Estes valores são conhecidos por “parâmetros”

Como exemplo, temos o programa acima, onde enviamos dois números para a função que calcula a divisão.

I.7 - Expressões numéricas e de string (caracteres)

Vamos ver neste item os tipos de expressão permitidas para valores numéricos e para a manipulação de caracteres.

Números Decimais:	Não podem começar por ‘ 0 ’ (zero) Exemplos: 123; 2; 14534; 3.14; ...
Números Octais: *(Pouco utilizados)*	Devem começar por ‘ 0 ’ (zero) Exemplos: 045; 09;...
Números Binários:	Devem iniciar por ‘ 0b ‘ Exemplo: 0b10101010
Números Hexadecimais:	Devem iniciar por ‘ 0x ‘ Exemplo: 0x32; 0xA9; ...
String de 1 caractere:	Entre Apóstrofos ‘ ‘ Exemplo: ‘z’; ‘A’; ....
String de vários caracteres:	Entre aspas “ “ Exemplo: “teste de escrita”

I.8 - Operadores lógicos e aritméticos básicos da linguagem C

Principais operadores lógico-relacionais e aritméticos da linguagem C.

- \+ Adição
- ++ Incremento da variável indicada (D++ é equivalente a D = D + 1)
- \- Subtração
- -- Decremento da variável indicada (X-- é equivalente a X = X - 1)
- * Multiplicação
- / Divisão (parte inteira da divisão para variáveis inteiras)
- % Resto da divisão (se for divisão de inteiros)
- ^ Exponencial

- < Comparador lógico “menor que”
- > Comparador lógico “maior que”
- <= Comparador lógico “menor ou igual que”
- >= Comparador lógico “maior ou igual que”
- == Comparador lógico “igual a” (*****)
- != Comparador lógico “diferente de”

- && AND lógico ou relacional (todas as condições verdadeiras)
- || OR lógico ou relacional (uma das condições é verdadeira)
- ! NOT lógico ou relacional (vê se a condição é TRUE ou FALSE)

- & AND binário (bit a bit nas variáveis)
- | OR binário (bit a bit nas variáveis)
- ~ NOT binário (inverte o estado de cada bit da variável)

(*****) **IMPORTANTE:** Se numa comparação usarmos ‘ = ‘, o compilador fará uma atribuição, gerando um programa errado logicamente.

```
Exemplo correto:    SE  A == 5 .....     // Verifica se A é igual a 5

Exemplo errado:     SE  A = 5 .....      // Primeiro faz A igual a 5, o que não é o
                                         // objetivo do teste
```

O compilador não avisa deste erro. Cuidado !

Exemplo para inverter a variável X e escrever no portb:

```
portb = ~ X;          // operador NOT binário
```

Exemplo para ver se o bit T0IF ainda não chegou a 1:

```
if ( ! T0IF ) ....    // Se T0IF NÃO (not) TRUE ( true = 1),....
                      // operador NOT lógico
```

I.9 - Matrizes

Define-se como *matriz* um grupo de dados que podem ser agrupados num mesmo nome, diferenciando-se apenas pela posição no grupo.

Como exemplo temos uma rua com várias casas. A rua seria nossa matriz, por exemplo, Rua Paraná. Cada casa tem um número que indica sua posição na rua (na matriz).

Exemplo: casa 32 da Rua Paraná

Poderíamos escrever Rua Parana[32]

Em C, a definição de uma variável ou de uma constante como matriz é feita apenas pela inclusão de seu *tamanho* entre colchetes [ ].

Exemplo:

Matriz para os 20 valores de temperatura lidos, variáveis tipo char:

```
char temperatura[20];      // reserva espaço de memória para
                           // 20 bytes que indicarão a temperatura
                           // O primeiro elemento é temperatura[0]
                           // O último elemento é temperatura[19]
```

Para acessar os elementos da matriz basta escrever o nome da variável com o índice do valor desejado.

Exemplo:

```
valor = temperatura[12];   // o 13º elemento da matriz temperatura
                           // é copiado para a variável valor.
```

IMPORTANTE:

1) O índice da matriz pode ser outra variável. Ex.: H = temperatura [contador];

2) O índice de acesso a matriz vai de **0** até ***tamanho-1***.
No nosso exemplo, irá de 0 à 19.

Se voce tentar acessar um elemento de endereco maior que o existente, o compilador usara uma posição de RAM que provavelmente esta em uso por outra variavel. **O compilador não tem condições de alertar sobre este erro.**

3) Muito cuidado com o uso de matrizes, pois temos pouca memória RAM nos chips.

I.10 - Controle do programa em C

Um dos pontos mais importantes do aprendizado da linguagem C esta na sua sintaxe, isto é, como o programa deve ser escrito.

Até agora vimos apenas teorias sobre variáveis, dados, e funções.

Veremos a partir de agora como tudo isto é integrado de forma a produzir um resultado útil.

Para tal vamos estudar as estruturas de montagem de programas e ver como se controla o fluxo de “ações” que o programa deve tomar.

Para facilitar o aprendizado não vamos nos preocupar agora com o PIC e nem com as funções criadas para o mesmo, mas apenas com a forma como escrevemos o programa em C.

I.10.1 - Blocos de declarações

Sempre que mais de uma instrução tiver de ser executada para uma certa rotina, as mesmas deverão estar contidas dentro de um *BLOCO* de declarações.

Um bloco tem início com a abertura de uma chave ‘ { ‘ e é finalizado pelo fechamento da chave ‘ } ‘.

Como um bloco não termina no fim da linha, mas sim ao fechar a chave, podemos escrever o mesmo de forma mais clara, colocando seus elementos em várias linhas e permitindo melhor colocação de comentários..

Exemplo 1:

```
{  x = 1;    tempo = x * 2;   }
```

Exemplo 2:

```
{
   x = 1;                  // posso colocar um
   tempo = x * 2;          // comentário para cada
}                          // linha
```

Os dois exemplos acima realizam a mesma tarefa, mas o exemplo 2 é mais fácil de ser lido e posteriormente entendido.

I.10.2 - Bloco IF (executa se a condição for verdadeira)

Podemos entender o bloco IF como um teste simples.

Temos duas opções básicas, sendo que a condição de teste deverá estar entre parênteses:

- SE (teste == ok!) "*executa esta(s) declaração(ões)*"

- SE (teste == ok!) "*executa esta(s) declaração(ões)*"
  SENÃO "*executa esta(s) outras declaração(ões)*"

Temos vários formatos possíveis para o IF

- IF simples, com apenas uma declaração caso o teste seja verdadeiro

```
if ( A == 0 ) A = 10;          // SE a variável A estiver zerada, atribui 10
                               // à mesma.
                               // Veja teoria dos operadores lógicos para o " == "
```

- IF com mais de uma declaração caso o teste seja verdadeiro.

  Neste caso o grupo de declarações deverá estar num BLOCO, isto é, entre chaves ' { } '.

```
if ( tempo > 10 )
{
        tempo = 0;
        contador = contador + 1;
}
```

- IF com exceção (se o teste falha, executa outra declaração ou bloco).
  Pode na exceção executar uma instrução apenas ou um bloco

```
if ( teste == sim )     declaração individual ou bloco

else
                declaração individual ou bloco da exceção
```

Importante: A instrução (declaração) simples não precisa estar na mesma linha do IF ou do ELSE. (Ver item I.10.2.1, nos exemplos de IF's aninhados).

Podemos então resumir numa tabela todas as combinações dos IF's:

- if (teste desejado) *instrução para teste "OK"*

- if (teste desejado)
  {
  grupo de instruções para teste "OK"
  }

- if (teste desejado) *instrução para teste "OK"*
  else *instrução para teste "NÃO OK"*

- if (teste desejado)
  {
  grupo de instruções para teste "OK"
  }
  else *instrução para teste "NÃO OK"*

- if (teste desejado) *instrução para teste "OK"*
  else
  {
  grupo de instruções para teste "NÃO OK"
  }

- if (teste desejado)
  {
  grupo de instruções para teste "OK"
  }
  else
  {
  grupo de instruções para teste "NÃO OK"
  }

I.10.2.1 - IF's aninhados (embutidos um no outro)

Chamamos a estrutura de "IF's aninhados" quando a instrução a ser executada para um teste (seja verdadeiro ou falso) é na verdade outro IF.

Vamos ver dois exemplos que ajudarão a esclarecer o assunto.

Exemplo 1: Observe os dois trechos de programa a seguir:

```
if ( X )                    |    if ( X )
  if (Y)                    |    {
       a = a * 2;           |         if (Y)
  else                      |              a = a * 2;
       a = a * 4;           |    }
                            |    else
                            |        a = a * 4
```

No trecho da esquerda, o else refere-se ao if (Y), pois esta "mais próximo" deste. Somente se o if (Y) resultar falso é que a linha a = a * 4 será executada.
Se o if (X) resultar *falso*, nenhuma operação será realizada.

No trecho da direita, o else refere-se ao if (X), pois o if (Y) esta dentro de um bloco, não sendo visível para o else.
Se o if (X) resultar *verdadeiro* mas o if(Y) resultar *falso*, nenhuma operação será realizada.

Exemplo 2: Vários IF's seqüênciais

```
if ( posição == 1)                      // Vê se posição = 1.
       peso = 1;                        // É 1. Faz peso = 1.

else   if (posição == 2)                // Não é 1. Vê se posição = 2.
              peso = 2;                 // É 2. Faz peso = 2.

       else if (posição == 3)           // Não é 2. Vê se posição = 3.
                    peso = 4;           // É 3. Faz peso = 4.

            else if (posição == 4)      // Não é 3. Vê se posição = 4.
                         peso = 8;      // É 4. Faz peso = 8.

                 else peso = 0;         // Não é 4. Faz peso = 0.
```

I.10.3 - Bloco FOR (loop para executar por um certo número de vezes)

A idéia do bloco FOR é executar uma instrução ou um bloco de instruções repetidamente, por um número de vezes determinado pela chamado do loop.

Sua sintaxe é a seguinte:

for (*ajustes iniciais* **;** *condições de teste* **;** *ajustes dos parâmetros*)
    instrução **;**
ou
for (*ajustes iniciais* **;** *condições de teste* **;** *ajustes dos parâmetros*)
{
    (grupo de instruções)
}

Para melhor entendimento, vejamos um exemplo para escrever na variável PORTB os números de 1 a 100 e ainda somar estes números:

```
char    PORTB;                  // declarei PORTB como variável de 1 byte
                                // Não é a porta B do PIC
long int soma;                  // declarei a variável soma como 16 bits.
char   i;                       /* a variável que é usada no loop também
                                   precisa ser declarada. Neste caso, 1
                                    byte é suficiente */

soma = 0;                       // faço soma = 0 para inicializar a variável
for ( i = 1; i < 101; i++)
{
        PORTB = i;              // escreve ‘ i ‘ em PORTB
        soma = soma + i;        // a soma anterior é somada a ‘ i ‘
}
```

I.10.3.1 - Loop infinito com FOR

Podemos criar um loop infinito com a declaração

for (; ;) *instrução que será executada indefinidamente*
ou
for (; ;)
{
    Instruções que serão executadas indefinidamente
}

Lembre-se que o programa de um microcontrolador não tem fim, sempre esta rodando, geralmente dentro de um loop infinito.

I.10.4 - O condicional WHILE (enquanto)

O WHILE executa a instrução ou o bloco de instruções “enquanto” a condição de teste for verdadeira.

”Se logo no inicio do teste a condição resultar falsa, nada será executado”.

Sintaxe:

```
while ( teste )          instrução para teste verdadeiro
```

ou

```
while ( teste )
{
        ( grupo de instruções para teste verdadeiro)
}
```

Vamos proceder novamente a soma dos números de 1 a 100 como no exemplo anterior.

```
i = 0;                          // nunca esquecer de inicializar
soma = 0;                       // todas as variáveis do programa.
while ( i < 101 )               // faça enquanto I<101
{
        soma = soma + 1;
        i++;                    /* como o WHILE apenas faz o teste
                                   devo incrementar a variável */
}
```

I.10.4.1 - Loop infinito com WHILE

```
while (true)
{
        ( declarações executadas eternamente )
}
```

A condição booleana “true” sempre será verdadeira (true).

OBSERVAÇÃO:

As condições TRUE e FALSE já estão pré-definidas no compilador.

I.10.5 - O condicional DO / WHILE (faça ... enquanto)

O condicional DO / WHILE funciona de forma semelhante ao WHILE, exceto pelo fato de que

"pelo menos uma vez a instrução ou o bloco serão executados".

Sua sintaxe é:

```
do
        instrução para teste verdadeiro
while ( teste ) ;
```

ou

```
do
{
        ( grupo de instruções para teste verdadeiro)
}
while ( teste ) ;
```

I.10.5 - O condicional SWITCH

O SWITCH testa a variável e conforme seu valor pula diretamente para o grupo de instruções conforme a cláusula CASE.

Caso nenhuma das condições previstas nos vários CASE sejam satisfeitas, executa o bloco DEFAULT, se houver (não é obrigatório).

Mesmo que apenas uma instrução seja usada para um certo CASE, sempre teremos um bloco (entre chaves ‘ { } ‘) pois além da instrução a ser executada devemos incluir a instrução BREAK, que faz com que o programa vá imediatamente para o fim do SWITCH, continuando a partir dai.

Caso o BREAK não seja colocado, o programa continuará pelo CASE logo abaixo do que foi chamado (ou no DEFAULT).

Seu formato geral é:

```
switch ( variável )
{
        case constante1
        {
                ( instrução ou grupo de instruções )
                break;
        }

        case constante2
        {
                ( instrução ou grupo de instruções )
                break;
        }

        .
        .
        .

        default:
        {
                ( instrução ou grupo de instruções para falso geral )
                break;
        }
}
```

Vamos ver um exemplo que faz o mesmo que o IF aninhado que vimos anteriormente.

```
switch ( posição )
        {
                case 1:                         // CASO posição = 1 ....
                {
                        peso = 1;
                        break;                  // sai do CASE.
                }
                case 2:
                {
                        peso = 2;
                        break;
                }
                case 3:
                {
                        peso = 4;
                        break;
                }
                case 4:
                {
                        peso = 8;
                        break;
                }
                default:                        // CASO posição NÃO seja 1, 2, 3 ou 4,
                {                               // executa este bloco
                        peso = 0;
                        break;
                }
        }
```

Para ilustrar vamos escrever várias declarações em uma só linha.

```
switch ( posição )
{
                case 1: { peso = 1; break; }            // CASO peso=1, ....
                case 2: { peso = 2; break; }            // CASO peso=2, ....
                case 3: { peso = 4; break; }
                case 4: { peso = 8; break; }
                default: { peso = 0; break; }
}
```

I.11 - Abreviações úteis para instruções aritméticas

I.11.1 - Incremento e Decremento

Duas delas foram mostradas no item I.8 (Operadores lógicos e aritméticos básicos da linguagem C), a saber:

- ++ Incremento da variável indicada

O operador '++' facilita a escrita de instruções para o incremento de variáveis.
Ao invés de escrever: a = a + 1;
podemos escrever: a++; *ou* ++a; (*)

- -- Decremento da variável indicada

O operador '**--**' facilita a escrita de instruções para o decremento de variáveis.
Ao invés de escrever: a = a - 1;
podemos escrever: a**--**; *ou* **--**a; (*)

Soma ou subtração de valores maiores que '1' à uma variável

Instrução normal	*Instrução abreviada*
X = X + 10;	X + = 10;
G = G - 25;	G - = 25;

I.11.2 - Combinando abreviações

Podemos combinar numa única linha abreviações e atribuições, de forma a facilitar a escrita do programa.

Importante: Os iniciantes na linguagem devem usar inicialmente as instruções básicas para não se confundirem.

Vamos explicar com alguns exemplos:

X = X + 1;
Y = X; ---> Y = ++X;

(*) Como o ++ esta 'antes' do X, primeiro o X será incrementado para depois Y receber o valor.

Se fizermos	o equivalente será
Y = X++;	Y = X; X = X + 1;

I.12 - A variável tipo VOID

Existe um tipo de variável especial chamada VOID.

Seu significado seria mais ou menos do tipo VAZIO.

É usada principalmente na chamada de rotinas ou no retorna das mesmas, para indicar que nenhum valor foi "enviado" ou será "retornado" da função.

Para melhor entendimento, vejamos uma rotina que não precisa de parâmetros e nada retorna:

```
.....                          // trecho de programa
.....
LeTeclas( );                   // chama a  rotina sem enviar nada
.....
.....


void LeTeclas( )               // este void indica que a rotina não envia
{                              // parâmetros de volta
    .....                      // O parenteses vazio informa que a rotina não
    .....                      // recebeu parâmetros
    .....
}
```

I.13 - Protótipos de funções

No inicio do programa devemos informar ao compilador todas as rotinas que serão usadas e seus parâmetros. Isto serve para evitar que quando usarmos a rotina a mesma seja chamada de forma errônea.

Serve ainda para o compilador montar o programa ja reservando as posições de memória RAM e estimativa de pilha usada.

Esta declaração da rotina é feita da mesma forma da rotina em sí, mas sem as instruções e finalizada pelo " **;** ".

Esta declaração é conhecida por **protótipo.**

Exemplos de protótipos:

```
void LeTeclas (void);          // a rotina LeTeclas não receberá parâmetros
                               // e também não retornará valores.

float DIVID (long int A, B);   // a rotina DIVID receberá duas variáveis do tipo
                               // long int, que receberão o nome de A e B para seu
                               // uso, e retornará um valor float.
```

I.14 - A função MAIN ( )

Todo programa em C deve ter uma função chamada MAIN ( ), que é a rotina que será executada quando o programa for chamado.

No caso do PIC, o compilador vai gerar um código de inicialização e depois desviar para o MAIN propriamente dito.

Exemplo de um programa que nada faz, mas que teoricamente pode ser compilado:

```
void main ( void)              // a função MAIN não recebe ou devolve valores
{

}
```

I.15 - Exemplos de programas simples

Veremos neste item alguns programas pequenos e simples, sem função técnica específica, apenas para ilustrar os conceitos vistos até agora.

Todos os conceitos vistos são genéricos para a linguagem C, servindo inclusive para usar em outros microprocessadores e computadores pessoais.

Programa 1:

Calcula a soma dos números de 1 a 100.

```
char    i;                      // variável i usada no loop (0 a 255)

long int soma;                  // soma de 0 a 65535

void main( void)
{
        soma = 0;               // devo sempre inicializar as variáveis
        for ( i = 0;  i < 101;  i++)
                soma = soma + i;
}
```

Programa 2:

Conta indefinidamente de 1 a 10. Sempre que chegar a 10 incrementa a variável X e recomeça.

```
char    i, x;                   // declarei que i e x são variáveis do
                                // tipo char
void main ( void)
{
        i = 0;   x = 0;         // inicializei as variáveis
        while (1)               // fica eternamente neste loop
        {
                i++;            // incrementa i
                if ( i == 10 )  // vê se i = 10.
                {
                        x++;    // É. Incrementa x e faz i = 0.
                        i = 0;
                }
        }
}
```

Programa 3:

Incrementa indefinidamente a variável chamada PORTB
(na parte II veremos como executar no PIC).

```
char    PORTB;                  // declarei que PORTB  é variável do
                                // tipo char (8 bits)
void main ( void)
{
        PORTB = 0;              // inicio em 0
        while (true)            // fica eternamente neste loop
        {
                PORTB++;        // incrementa PORTB
        }
}
```

ANEXO

Temos a seguir um programa ilustrativo do que foi estudado até agora.

Este programa foi compilado para o 16F84 e não de modo genérico porque o compilador sempre precisa saber a quantidade de RAM e de ROM disponível.

Listagem do programa "REGRAS.C"

```
/*****************************************************************************
        Programa demo para o compilador C da CCS.
        Compila blocos e instruções apenas para ver a compilação.

        Aqui já temos um exemplo de comentários em varias linhas
*****************************************************************************
*/
                        // Cabeçalho inicial com diretivas de compilação

#CASE                           // COMPILADOR CASE SENSITIVE.
                                // Padrão da linguagem C

#include <16F84.H>              // Define processador
                                // Necessário para informar a
                                // quantidade de RAM e ROM disponível

// ***************************************************************************
                        // protótipos de rotinas

void _while(void);          // nao recebe e nao devolve
void _do_while(void);       // nao recebe e nao devolve parametros
void _switch(void);         // nao recebe e nao devolve parametros
void _ifs(void);            // nao recebe e nao devolve parametros

void _for(char j);          // recebe um char como parametro
                            // mas nao devolve nada

float quadrado(char q);     // recebe char e deve devolver float

// ***************************************************************************
                            // constantes e variaveis

char        a,b,c,d,e;          // variaveis de 8 bits
long int    aa,bb,cc,dd,ee;     // variaveis de 16 bits
float       VarFloat;           // variavel de 32 bits
short       Led;                // Led será um pino de I/O

float   const  PI = 3.14;       // definicao de constante tipo float

char    Temperatura[20];        // definiçao de uma matriz de 20 elementos

// ***************************************************************************
```

```
void main()                     // Programa Principal
{
        while (1)               // loop infinito
        {
                a = 65;         // atribuicao de variavel decimal 65
                b = 0b01000001; // atribuicao de variavel 65 em binario
                c = 0x41;       // atribuicao de variavel 65 em hexa
                d = 'A';        // atribuicao de variavel 65 em ascii

                _ifs();         // chamada de rotinas sem parametros
                _while();
                _do_while();
                _switch();
                _for(20);       // chama rotina com parametro de valor 20

                VarFloat = quadrado(4);
        }
}

// ********************************************************************************************
void _ifs()
{
        if (a == 0)
        {
                e = 100;
        }
        else
        {
                e = 90;
        }
}

// ********************************************************************************************

void _while()
{
        a = 0;
        while (a < 10)
        {
                b = 5;
                c = b * 10;
                a++;
        }
}
```

```
// ***********************************************************************************************
void _do_while()
{
char VarLocal;

        VarLocal = 0;
        e = 0;
        do
                e += 3;
        while ( e < 100 );
}

// ***********************************************************************************************
void _switch()
{
        switch (d)
        {
                case 1:
                {
                        a = 1;
                        break;
                }
                case 2:
                {
                        a = 2;
                        break;
                }
                default:
                {
                        a = 5;
                        break;
                }
        }
}

// ***********************************************************************************************
void _for(char j)
{
        for (e=1; e<j+1; e++)                   // 'e' varia de 1 a 'j'
                Temperatura[e] = e;
}

// ***********************************************************************************************
float quadrado(char q)
{
        return (q*q);
}
```

Parte II - O compilador C da CCS Inc.

Veremos nesta parte como escrever programa em C para os PIC's, acessando registros, portas e memória, alem de estudar as principais funções criadas pela CCS para facilitar o trabalho do desenvolvedor de sistemas.

Não vamos estudar todas as funções criadas, apenas as mais usadas de forma a permitir ao estudante aprofundar-se com facilidade posterior quando do uso do software completo e sem limitações.

Para este estudo usaremos uma versão de demonstração do compilador PCM, funcional por 30 dias, que compila *apenas* para o 16C63.

Nota importante: *para manter compatibilidade com chips e usuários mais antigos da linha PIC, em algumas funções o TIMER 0 é chamado pelo seu nome antigo, RTCC.*

II.1 - Diretivas de compilação

Ver item I.3 para teoria sobre as diretivas de compilação.

II.1.1 - #asm #endasm

Permite a inclusão de trechos em assembler dentro do programa em C.
Muito cuidado deve ser tomado com o uso desta diretiva, pois toda ação gerada em assembler é de inteira responsabilidade do usuário, e pode vir a prejudicar outras variáveis ou posições de RAM usadas pelo compilador.

Exemplo:

```
.
.
x = a;
b = x / 3;

#asm
        bsf     PORTB,3     // estas duas instruções geram um pulso
        bcf     PORTB,3     // no pino RB3.
#endasm

x = sqrt(a);
.
.
```

IMPORTANTE: Observe que embora seja um trecho em assembler o comentário esta no formato da linguagem C.

II.1.2 - #case

Indica para o compilador ser “sensitivo” a diferenças entre letras maiúsculas e minúsculas.

Embora a linguagem C de forma padrão seja sensivel ao tipo de letra, sem esta diretiva, o compilador ignora as diferenças.

Exemplo: com o #case

Teste, *teste* e *TESTE* são variáveis diferentes

Exemplo: sem o #case

Teste, *teste* e *TESTE* são a mesma variável

II.1.3 - #define *‘nome’ ‘seqüência’*

Permite a substituição, no momento da compilação, do *‘nome’* pela cadeia de caracteres (string) definida em *‘seqüência*’.

Exemplo:

```
#define     largura     4
portb = largura;            // será visto pelo compialdor como
                            // portb = 4

largura = 5;                // ERRO. Nao é uma variável.
```

Sua principal vantagem esta em não precisarmos trocar constantes em todo o programa, bastando alterar na definição, ou ainda para facilitar a escrita de funções mais complexas.

(Mais exemplos serão vistos ao final)

II.1.4 - #include *<arquivo>*

O arquivo indicado será adicionado ao fonte no momento da compilação.

Ideal para incluir trechos pré-definidos, evitando sua digitação a cada programa novo.

Exemplo típico é o arquivo PIC16F84.H que contém as definições sobre o processador.

Exemplo: #include <pic16f84.H>

II.1.5 - #fuses *'opções'*

Avisa ao compilador para incluir no arquivo .HEX informações sobre os fusíveis de configuração, facilitando na hora de se gravar o chip.
Para cada chip devemos ver em seu arquivo .H quais as opções do mesmo.
Devemos cuidar para que não haja repetição de uma mesma opção.

As principais opções são:

Oscilador:	XT (cristal até 4 MHz), LP (cristal baixa potência), HS (cristal alta velocidade), RC (oscilador RC)
WatchDog:	WDT (ligado) e NOWDT (desligado)
Código:	PROTECT (protegido) e NOPROTECT (não protegido)
Power-up:	PUT (Com power-up) e NOPUT (Sem power-up)

Exemplo: #fuses XT,NOWDT,NOPROTECT,PUT

Indica ao gravador que o modo é cristal, sem watch dog, código desprotegido e com power-up timer ligado.

II.1.6 - #ifdef #endif

Permite selecionar se o trecho entre as duas diretivas será incluido no momento da compilação ou não.

Exemplo:

```
#define     OK
.
.
#ifdef  OK
        delay_ms(200);
#endif
```

No exemplo, caso o #define OK exista, a chamada para a rotina de tempo será incluida no programa final. Caso contrario não será incluida.
Neste caso poderia servir para tirarmos o atraso durante os testes no simulador.

II.1.7 - #ifndef #endif

Funciona de forma similar, incluindo o trecho edntro da diretiva se a definição NÃO existir

II.1.8 - #inline

Informa ao compilador para **repetir** o trecho de código da rotina indicada sempre que a mesma for chamada.

Evita a criação de sub-rotinas (call's) economizando espaço da pilha e executando o programa mais rapidamente.

Em compensação ocupa mais espaço de memória de programa.

Exemplos:

1 - Sem o #inline

```
tempo()                 // definimos uma rotina
{
        for ( i=0; i<100; i++) {}
}
```

Se na seqüência de programa tivermos:

```
tempo();                // chamamos a rotina de tempo
LeTeclas();             // leitura de teclas
tempo();                // chamamos a rotina de tempo novamente
```

o compilador gerará a seguinte sub-rotina em ASM:

```
tempo:
        movlw   100     // valor apenas ilustrativo
        movwf   i
espera_i;
        decfsz  i
        goto    espera_i
        return          // retorna da rotina
```

e as chamadas serão:

```
call tempo;
call LeTeclas;
call tempo;
```

2 - Com o #inline

```
#inline
tempo()              // definimos uma rotina
{
        for ( i=0; i<100; i++) {}
}
```

Se na seqüência de programa tivermos:

```
tempo();             // chamamos a rotina de tempo
LeTeclas();          // leitura de teclas
tempo();             // chamamos a rotina de tempo novamente
```

o compilador NÃO gerará uma sub-rotina em ASM, mas fará o seguinte para o trecho exemplificado

```
tempo_A:
        movlw   100
        movwf   i
espera_i_A;
        decfsz  i
        goto    espera_i_A

        call LeTeclas;

tempo_B:
        movlw   100
        movwf   i
espera_i_B;
        decfsz  i
        goto    espera_i_B
```

Veja que ao invés de criar uma sub-rotina, o trecho que realiza a função de tempo foi repetido a cada chamada para o mesmo.

Deve-se tomar muito cuidado com este recurso !!

Embora ajude a economizar posições de pilha em casos críticos, aumenta o tamanho do programa final

II.1.9 - #INT_????? (observe o " _ " entre as palavras)

Indica para o compilador que o trecho a seguir refere-se a uma rotina de interrupção.

O compilador vai gerar um código padrão para entrar e sair da rotina, salvando o estado da máquina e zerando o flag de requisição de interrupção ao sair da mesma.

Algumas das interrupções reconhecidas são:

- INT_EXT (RB0/Int)
- INT_RTCC (Timer 0)
- INT_RB (RB4~7)
- INT_EEPROM (Eeprom interna)
- INT_AD (Conversor A/D interno)
- INT_DEFAULT (Caso entre na rotina de interrupção por engano)

Para uma lista completa do nome das diretivas de interrupção das várias dezenas de chips existentes, consultar o manual da CCS.

O nome da rotina é definido pelo programa.

Exemplo:

```
#INT_EXT          //  avisa que a rotina a seguir é a da interrup. externa
EntradaRede()
{
        .......
        .....
{
```

II.1.10 - #Priority

Permite ao programador escolher a prioridade das interrupções habilitadas. A primeira da lista é a mais prioritária.

Exemplo para o 16F84:

```
#priority      rtcc, eeprom, ext      // definimos que a interrupção do
                                      // Timer 0 é mais prioritária que a da
                                      // eeprom interna, que por sua vez é
                                      // mais prioritária que a interrupção
                                      // externa.
```

II.1.11 - #ROM

Permite a gravação de constantes na memoria de programa ou na EEPROM de dados interna (F84 e F87X).

Não influencia no programa, apenas informa ao gravador para gravar os valores indicados no endereço indicado.

Exemplo:

Determinando os 10 primeiros bytes da eeprom de dados (inicio em 2100H):

```
#ROM          0x2100 = { 0,1,2,3,4,5,6,7,8,9 }
```

II.1.12 - #use delay (clock = *'valor do clock em Hz'*)

Informa ao compilador o valor do clock da CPU em Hertz. Serve para permitir o uso de funções de tempo baseadas no clock da máquina.

Caso o watch dog seja usado, devemos informar por esta diretiva, para que o mesmo seja resetado nas rotinas de tempo.

Exemplos:

1) Sem o watch dog:

```
#use delay(clock = 4000000)       // clock de 4 MHz
```

2) Com o watch dog:

```
#use delay(clock = 10000000, Restart_Wdt)     // clock de 10 MHz
                                              // com watch dog
```

II.1.13 - #use Fast_IO(*port*) (observe o " _ " entre as palavras)

Informa ao compilador que o controle do TRIS do *port* indicado será executado pelo programa. Gera operações de entrada e saída mais rápidas.

Exemplo:

```
#use fast_io(porta)          // quem deve controlar o trisa é o programa.
```

II.1.14 - #use standard_IO(*port*) (observe o " _ " entre as palavras)

A cada acesso as portas ou pinos de I/O o compilador gerará código para o controle do tris correspondente. Permite maior flexibilidade nos casos de bits com dupla função, mas consome mais tempo e memória de programa.

II.1.15 - #use rs232 (BAUD = *taxa*, XMIT = *pinoTx*, RCV = *pinoRx*, BITS = *n*)

Ajusta os pinos indicados como **pinoRX** e pino**TX** para funcionarem como entrada e saída serial com velocidade indicada em **taxa**, e com palavra de **n** bits.

O compilador tem incorporado rotinas que permitem a transmissão e recepção no formato assíncrono típico da comunicação serial.
Para o uso destas rotinas a diretiva acima deve ser utilizada.

Embora o nome da diretiva sugira RS232, devemos lembrar que os sinais do PIC são de nível TTL.

Importante: antes do uso desta diretiva o #use delay é obrigatório.

Exemplo:

Recebo um byte pelo pino RB7 e transmito o mesmo valor pelo pino RA4, a 9600 bps e palavra de 8 bits:

```
#use delay (clock=4000000)
#use rs232 (BAUD = 9600, XMIT = pin_A4, RCV = pin_B7, BITS = 8 )

X = getchar ();          // fica esperando um byte chegar pelo serial
                         // e então escreve na variável X
putchar ( X );           // e depois envia o mesmo de volta.
```

Observação: os nomes de pinos e outras constantes serão vistos ao final e estão no arquivo .H correspondente ao chip utilizado.

II.1.16 - #byte *nome* = *endereço* E #bit *nome* = *endereço*

Permite o acesso a posições de RAM (Registros e memória geral) pela indicação de seu endereço real (lembre-se de que o gerenciamento das variáveis é realizado pelo compilador).

Exemplo: Sabemos que o PORTB esta no endereço 6, e se desejarmos acessar o PORTB como se fosse uma variável devemos usar:

```
#byte portb = 6
```

Vale o mesmo para variáveis representando bits:

```
#bit    T0IF = 0x0B.2      // T0IF agora é uma variável em C
```

Exemplo: espera o overflow do timer 0 --> while (!T0IF);

II.2 - Funções escritas para os PIC's

Neste ítem veremos algumas das funções escritas especialmente para a família PIC. Uma descrição completa de todas as funções pode ser encontrado no manual do usuário da CCS.

II.2.1 - Funções matemáticas

f = ACOS (x);	// ArcoCosseno.	*f* deve ser float e -1 < x , +1
f = ASIN (x);	// ArcoSeno.	*f* deve ser float e -1 < x , +1
f = ATAN (x);	// ArcoTangente.	*f* deve ser float e -PI/2 < x , +PI/2
f = LOG (x);	// Lob base E	*f* deve ser float e x > 0 (limitado)
f = LOG10 (x);	// Log base 10.	*f* deve ser float e x > 0 (limitado)
f = SQRT(x);	// Raiz quadrada.	*f* deve ser float e x > 0
f = FLOAT EXP(x);	// Exponencial E^x	*f* deve ser float e x > 0

II.2.2 - Funções de manipulação de bit

1) bit_clear (variável , bit) (observe o " _ " entre as palavras)

Zera o *bit* indicado na *variável* indicada. Como podemos ter variáveis de 8 ou 16 bits, o número do bit pode ir de 0~7 ou de 0~15.

Exemplo:

```
char   x;
x = 0b10001000
bit_clear(x,7);        // x = 00001000
```

2) bit_set (variável , bit) (observe o " _ " entre as palavras)

Seta o *bit* indicado na *variável* indicada. Como podemos ter variáveis de 8 ou 16 bits, o número do bit pode ir de 0~7 ou de 0~15.

Exemplo:

```
char   x;
x = 0b10001000
bit_set(x,0);   // x = 10001001
```

3) bit_test (variável , bit) (observe o " _ " entre as palavras)

Efetua um teste no *bit* indicado na *variável* indicada. Como podemos ter variáveis de 8 ou 16 bits, o número do bit pode ir de 0~7 ou de 0~15.

Se o bit estiver em '1' a função retorna um *verdadeiro (true)* e caso contrário retorna *falso (false)*.

Para o caso de desejamos testar se esta em '0' basta usarmos um **!** (NOT lógico) antes do teste.

Exemplo:

```
if  (  bit_test(x,1) ||  ! bit_test(x,0)  )      // Se o bit 1 de x for    1
{                                                //  ||          OU (||)
        ......                                   // o bit 0 de x for   0
}                                                // executa o bloco.
```

4) b = input(pino)

Lê o estado do pino indicado.

Exemplo: espera o pino RB0 ir de 0 para 1.

```
while ( ! input (pin_b0)  );          // realiza bit test . É o teste Ideal.
```

ou

```
while ( pin_b0 == 0);                 // realiza teste logico em byte
```

Observação: os nomes de pinos e outras constantes serão vistos ao final e estão no arquivo .H correspondente ao chip utilizado.

5) output_bit(pino, valor) (observe o " _ " entre as palavras)

Escreve 0 ou 1 no pino indicado.

Exemplo: gera um pulso positivo em RA2

```
output_bit ( pin_a2 , 1);          // faz RA2 = 1
output_bit ( pin_a2 , 0 );         // faz RA2 = 0
```

Observação: os nomes de pinos e outras constantes serão vistos ao final e estão no arquivo .H correspondente ao chip utilizado.

6) Output_High(pino) e Output _Low (pino) (observe o " _ " entre as palavras)

Equivalem ao **bsf** e ao **bcf** do assembler.
Servem para setar ou zerar um pino de I/O.

II.2.3 - Funções de tempo (observe o " _ " entre as palavras)

1) Delay_cycles (x); Gera um delay equivalente a 'x' *ciclos de instrução*.

```
Exemplo:        delay_cycles (1);       // equivale a 1 nop
                delay_cycles (4);       // equivale a 4 nop's
```

2) Delay_Ms (x) Gera um delay de 'x' milissegundos.

- Se 'x' for constante, pode variar de 1 à 65535.
- Se 'x' for uma variável, seu valor pode variar de 1 à 255.

O programa deve conter a diretiva #use delay vista em II.1.12 antes de se usar esta função.

Exemplo: Pulsa o pino RB2 com frequência de 100 Hz

```
while (1)
{
        output_high (pin_b2);       // RB2 = 1
        delay_ms (5);               // espera 5 ms
        output_low (pin_b2);        // RB2 = 0
        delay_ms (5);               // espera 5 ms
}
```

2) Delay_Us (x) Gera um delay de 'x' microsegundos.

- Se 'x' for constante, pode variar de 1 à 65535.
- Se 'x' for uma variável, seu valor pode variar de 1 à 255.

O programa deve conter a diretiva #use delay antes de se usar esta função.

Exemplo: Gera um sinal em RB2 com nível alto de 150 us e nível baixo de 50 us.

```
x = 150;
y = 50;
while (1)
{
        output_high (pin_b2);       // RB2 = 1
        delay_us (x);           // espera 'x' us
        output_low (pin_b2);        // RB2 = 0
        delay_us (y);           // espera 'y' us
}
```

II.2.4 - Funções para interrupções

1) enable_interrupts (*nível*) (observe o " _ " entre as palavras)

Permite habilitar interrupções individualmente ou globalmente. quando habilitando globalmente apenas o GIE será afetado, e não as habilitações individuais.

Os principais *níveis* são os seguintes:

global (GIE), int_ext (RB0), int_rtcc (timer 0),
int_rb (RB4~7), int_ad (16C7x), int_eeprom (16F84).

Exemplo: habilitando a interrupção do timer 0 e do A/D do 16C711.

```
enable_interrupts (int_rtcc);     // faz T0IE = 1
enable_interrupts (int_ad);       // faz ADIE = 1
enable_interrupts (global);       // faz GIE = 1
```

2) disable_interrupts (*nível*) (observe o " _ " entre as palavras)

Funciona de forma semelhante ao "enable_interrupts ()", desabilitando individualmente ou globalmente as interrupções.

Exemplo: desabilitando todas, desabilitando a do timer 0 e habilitando a externa e reabilitando todas.

```
disable_interrupts (global);      // faz GIE = 0
disable_interrupts (int_rtcc);    // faz T0IE = 0
enable_interrupts (int_ext);            // faz INTE = 1
enable_interrupts (global);       // faz GIE = 1
```

3) ext_int_edge *(tipo)* (observe o " _ " entre as palavras)

A variável *tipo* indica se a interrupção externa será reconhecida na descida do sinal em RB0 (tipo = H_TO_L) ou na subida do sinal (tipo = L_TO_H).

Exemplo: reconhece interrupção externa na subida.

```
ext_int_edge (L_TO_H);            // faz INTEDG = 1
enable_interrupts (int_ext);            // faz INTE = 1
```

II.2.5 - Funções para o canal A/D (apenas 16C7X e 16F87X)

1) read_adc() (observe o " _ " entre as palavras)

Lê o canal analógico do chip.

Antes de usar esta função devemos usar "setup_adc()" e "set_adc_channel()" e setup_adc_ports().

Exemplo:

```
Temperatura = read_adc();      // lê o canal já ajustado e escreve
                               // na variável Temperatura.
```

2) setup_adc (*modo*) (observe o " _ " entre as palavras)

Ajusta o conversor A/D. Os modos possíveis são:

adc_off, adc_clock_div_2, adc_clock_div_8,
adc_clock_div_32, adc_clock_div_internal,

3) set_adc_channel (*canal*) (observe o " _ " entre as palavras)

Especifica qual o canal que será lido pela função read_adc().

4) setup_adc_ports (*ajuste*) (observe o " _ " entre as palavras)

Ajusta os pinos para serem entradas analógicas ou não. Seu ajuste depende de cada chip em particular, devendo o programador cuidar para não escolher um ajuste errado para um certo modelo (consultar o arquivo .H correspondente).

Para o 16C711, temos os seguintes ajustes:

NO_ANALOGS	Pinos RA0~RA3 I/O digital
ALL_ANALOG	Pinos RA0~RA3 entradas analógicas Vref = Vdd
ANALOG_RA3_REF	Pinos RA0~RA2 entradas analógicas Vref = Tensão em RA3
RA0_RA1_ANALOG	Pinos RA0 e RA1 analogicos, RA2 e RA3 I/O digital. Vref = Vdd

Exemplo: Lendo o canal 2 do 16C711

```
setup_port_a( ALL_ANALOG );
setup_adc( ADC_CLOCK_INTERNAL );
set_adc_channel( 2 );
value = Read_ADC();
```

II.2.6 - Funções para EEPROM interna (apenas 16F8X e 16F87X)

1) Read_Eeprom (*endereço*) (observe o " _ " entre as palavras)

Lê o dado presente na posição *endereço* da eeprom interna.
O *endereço* vai de 0 à 3.

Exemplo: Posicao_0 = read_eeprom(0);

2) Write_Eeprom (*endereço* , *valor*) (observe o " _ " entre as palavras)

Escreve no *endereço* da eeprom interna o *valor* indicado.

Importante: Esta instrução leva em média 10 ms para ser finalizada.
Para cada processador ou versões mais atualizadas consultar o manual apropriado.

II.2.7 - Funções de uso geral

Veremos neste item várias funções de uso geral desenvolvidas para os pic's, como entrada e saída serial por software, ajuste de registros e outros.

1) get_timer0() (observe o " _ " entre as palavras)

Le o valor atual do registro TMR0.

Exemplos:

```
if  ( get_timer0() == 0 ) { ..... }      // executa o bloco se o registro
                                         // TMR0 = 0

Valor_do_timer = get_timer0();     // copia o registro TMR0 para a variável
```

2) set_timer0 (*valor*) (observe o " _ " entre as palavras)

Ajusta o valor do registro TMR0.
Lembre-se que sempre que escrevemos no registro TMR0 a contagem fica suspensa por dois ciclos de instrução.

Exemplo: Se o registro TMR0 for maior que 100 zera o registro.

```
if  (  get_timer0( )  >  100 )
        set_timer0( 0 );
```

3) port_b_pullups (*opção*) (observe o " _ " entre as palavras)

Controla os pull-ups da porta B (apenas para os bits ajustados entrada).

Temos como *opção* TRUE, ligando os pull-ups (RBPU\ = 0) e FALSE, desligando os pull-ups (RBPU\ = 1).

4) restart_wdt() (observe o " _ " entre as palavras)

Reseta o watch dog. Equivale ao *clrwdt* do assembler.

5) Sleep() (observe o " _ " entre as palavras)

Equivale a instrução *sleep* do assembler.

6) getc(), getch() OU getchar() (*)

São a mesma função. Entra numa rotina para esperar a chegada de um caractere pelo pino RCV definido na diretiva #use RS232.

Exemplo: espera a chegada do caractere 'A'

```
while ( getchar() != 'A' );
```

7) putc(*valor*) OU putchar(*valor*) (*)

São a mesma função. Envia "*valor*" serialmente pelo pino XMIT definido na diretiva #use RS232.

Exemplo: Enviar a palavra 'OLA' pelo serial.

```
putchar ( 'O' );
putchar ( 'L' );
putchar ( 'A' );
```

(*) Observação: Estas funções não usam o canal serial de hardware presente em alguns modelos, mas sim funções escritas para este fim. Desta forma devemos levar em conta que embora ocupem apenas uma linha de código, podem demorar vários milissegundos para serem executadas.

8) set_tris_X (*ajuste*) (observe o " _ " entre as palavras)

Ajusta o registro TRIS do port X indicado.

Deve ser usado sempre que a diretiva #fast_io (port) for utilizada, pois estas determinam nosso controle sobre os ports.

Exemplo: Ajustando todos os bits do PORTB como saída

```
set_tris_b ( 0 );
```

Ajustando os bits 0, 1 e 4 do PORTA como saída

```
set_tris_a ( 0b11101100 );
```

9) restart_cause() (observe o " _ " entre as palavras)

Esta função retorna o motivo do último reset.
Seus valores são:

wdt_from_sleep	wdt_timeout
mclr_from_sleep	normal_power_up

Exemplo:

```
switch ( restart_cause )
{
        case  wdt_from_sleep:
                {
                        "Instruções para este caso"
                        Break;                  // necessário
                }
        case  wdt_timeout:
                {
                        "Instruções para este caso"
                        Break;                  // necessário
                }
        case  mclr_from_sleep:
                {
                        "Instruções para este caso"
                        Break;                  // necessário
                }
        case  normal_power_up:
                {
                        "Instruções para este caso"
                        Break;                  // necessário
                }
}
```

11) Setup_counters (*ajuste_do_timer_0 , ajuste_do_prescaler*)

(observe o " _ " entre as palavras)

Ajusta o modo de funcionamento do timer 0 (aqui chamado de rtcc) e o direcionamento / divisão do prescaler.

Lembre-se que com o prescaler direcionado para o timer 0 o período nominal do watch dog é de 18 ms, e se o prescaler estiver direcionado para o watch dog o clock do timer não será dividido (1:1).

Constantes para ajuste do timer 0:

rtcc_internal	--> timer 0 clock interno
rtcc_ext_h_to_l	--> timer 0 clock externo, transição de 1 para 0
rtcc_ext_l_to_h	--> timer 0 clock externo, transição de 0 para 1

Constantes para ajustes do prescaler:

rtcc_div_2	--> Prescaler para o timer 0, dividindo por 2	(*)
rtcc_div_4	--> Prescaler para o timer 0, dividindo por 4	(*)
rtcc_div_8	--> Prescaler para o timer 0, dividindo por 8	(*)
rtcc_div_16	--> Prescaler para o timer 0, dividindo por 16	(*)
rtcc_div_32	--> Prescaler para o timer 0, dividindo por 32	(*)
rtcc_div_64	--> Prescaler para o timer 0, dividindo por 64	(*)
rtcc_div_128	--> Prescaler para o timer 0, dividindo por 128	(*)
rtcc_div_256	--> Prescaler para o timer 0, dividindo por 256	(*)

(*) watch dog em 18 ms

wdt_18ms	--> watch dog em 18 ms e timer 0 em 1:1
wdt_36ms	--> watch dog em 36 ms e timer 0 em 1:1
wdt_72ms	--> watch dog em 72 ms e timer 0 em 1:1
wdt_144ms	--> watch dog em 144 ms e timer 0 em 1:1
wdt_288ms	--> watch dog em 288 ms e timer 0 em 1:1
wdt_576ms	--> watch dog em 576 ms e timer 0 em 1:1
wdt_1152ms	--> watch dog em 1152 ms e timer 0 em 1:1
wdt_2304ms	--> watch dog em 2304 ms e timer 0 em 1:1

Exemplo: Ajustando o timer modo interno em 1:4, teremos watch dog com período de 18 ms.

```
setup_counters ( rtcc_internal, rtcc_div_4 );
```

II.4 - Personalizando o compilador

Vamos estudar aqui como personalizar o compilador para nossas preferencias, como por exemplo renomear os pinos de I/O do chip e ajustar funções pré-definidas para serem mais faceis de escrever.

II.4.1 - Redefinindo os nomes dos pinos de I/O

Conforme já vimos, os pinos são chamados de PIN_xx, onde 'xx' define o nome do mesmo.

Para o nosso estudo e para o pic 16C63, que é o pic da versão demo, vamos renomea-los para os nomes usuais.

```
#define RA0  40          // original PIN_A0
#define RA1  41          // original PIN_A1
#define RA2  42          // original PIN_A2
#define RA3  43          // original PIN_A3
#define RA4  44          // original PIN_A4

#define RB0  48          // original PIN_B0
#define RB1  49          // original PIN_B1
#define RB2  50          // original PIN_B2
#define RB3  51          // original PIN_B3
#define RB4  52          // original PIN_B4
#define RB5  53          // original PIN_B5
#define RB6  54          // original PIN_B6
#define RB7  55          // original PIN_B7

#define RC0  56          // original PIN_C0
#define RC1  57          // original PIN_C1
#define RC2  58          // original PIN_C2
#define RC3  59          // original PIN_C3
#define RC4  60          // original PIN_C4
#define RC5  61          // original PIN_C5
#define RC6  62          // original PIN_C6
#define RC7  63          // original PIN_C7
```

De agora em diante podemos apenas chama-los de RA3, RB2, ...

II.4.2 - Redefinindo algumas funções para nomes mais simples

Para facilitar a escrita e a memorização vamos redefinir algumas funções mais usuais (ver arquivo mudancas.inc em '\0DiscoEx'.

```
#define     TrisA(X)            set_tris_A (X)
#define     TrisB(X)            set_tris_B (X)
#define     TrisC(X)            set_tris_C (X)
#define     ClearBit(x)         output_low (x)
#define     SetBit(x)           output_high (x)
#define     ReadBit(x)          input (x)
#define     timer0(modo,presc)  setup_counters(modo,presc)
#define     Enable(X)           enable_interrupts(X)
#define     EnableAll           enable_interrupts(global)
#define     DisableAll          disable_interrupts(global)
#define     NOP                 delay_cycles(1)
#define     Ms(x)               delay_ms(x)
#define     Us(x)               delay_us(x)
#define     PullUpOn            port_b_pullups(true)
#define     PullUpOff           port_b_pullups(false)
#define     ext_0               ext_int_edge(h_to_l)
#define     ext_1               ext_int_edge(l_to_h)
```

II.4.3 - Anexo 1 - Arquivo 16F84.H

```
//////// Standard Header file for the PIC16F84 device ////////
#device PIC16F84
#nolist

////////////////////////////// I/O definitions for INPUT() and OUTPUT_xxx()
#define PIN_A0  40
#define PIN_A1  41
#define PIN_A2  42
#define PIN_A3  43
#define PIN_A4  44

#define PIN_B0  48
#define PIN_B1  49
#define PIN_B2  50
#define PIN_B3  51
#define PIN_B4  52
#define PIN_B5  53
#define PIN_B6  54
#define PIN_B7  55

////////////////////////////// Useful defines
#define FALSE 0
#define TRUE 1

#define BYTE int
#define BOOLEAN short int

#define getc getch
#define getchar getch
#define puts(s) {printf(s); putchar(13); putchar(10);}
#define putc putchar

////////////////////////////// Constants used for RESTART_CAUSE()
#define WDT_FROM_SLEEP  0
#define WDT_TIMEOUT     8
#define MCLR_FROM_SLEEP 16
#define NORMAL_POWER_UP 24

////////////////////////////// Constants used for SETUP_COUNTERS()
#define RTCC_INTERNAL   0
#define RTCC_EXT_L_TO_H 32
#define RTCC_EXT_H_TO_L 48
```

```
#define RTCC_DIV_2      0
#define RTCC_DIV_4      1
#define RTCC_DIV_8      2
#define RTCC_DIV_16     3
#define RTCC_DIV_32     4
#define RTCC_DIV_64     5
#define RTCC_DIV_128    6
#define RTCC_DIV_256    7
#define WDT_18MS        8
#define WDT_36MS        9
#define WDT_72MS        10
#define WDT_144MS       11
#define WDT_288MS       12
#define WDT_576MS       13
#define WDT_1152MS      14
#define WDT_2304MS      15
#define L_TO_H          0x40
#define H_TO_L          0

#define RTCC_ZERO         0x0B20    // Used for ENABLE/DISABLE INTERRUPTS
#define INT_RTCC          0x0B20    // Used for ENABLE/DISABLE INTERRUPTS
#define RB_CHANGE         0x0B08    // Used for ENABLE/DISABLE INTERRUPTS
#define INT_RB            0x0B08    // Used for ENABLE/DISABLE INTERRUPTS
#define EXT_INT           0x0B10    // Used for ENABLE/DISABLE INTERRUPTS
#define INT_EXT           0x0B10    // Used for ENABLE/DISABLE INTERRUPTS

#define GLOBAL            0x0B80    // Used for ENABLE/DISABLE INTERRUPTS
#undef GLOBAL
#define GLOBAL            0x0B80    // Used for ENABLE/DISABLE INTERRUPTS
#define INT_EEPROM        0x0B40    // Used for ENABLE/DISABLE INTERRUPTS

#list
```

II.4.4 - Anexo 2 - Arquivo 16C63.H

```
//////// Standard Header file for the PIC16C63 device ////////
#device PIC16C63
#nolist

////////////////////////////// I/O definitions for INPUT() and OUTPUT_xxx()
#define PIN_A0  40
#define PIN_A1  41
#define PIN_A2  42
#define PIN_A3  43
#define PIN_A4  44
#define PIN_A5  45

#define PIN_B0  48
#define PIN_B1  49
#define PIN_B2  50
#define PIN_B3  51
#define PIN_B4  52
#define PIN_B5  53
#define PIN_B6  54
#define PIN_B7  55

#define PIN_C0  56
#define PIN_C1  57
#define PIN_C2  58
#define PIN_C3  59
#define PIN_C4  60
#define PIN_C5  61
#define PIN_C6  62
#define PIN_C7  63

////////////////////////////// Useful defines
#define FALSE 0
#define TRUE 1

#define BYTE int
#define BOOLEAN short int

#define getc getch
#define getchar getch
#define puts(s) {printf(s); putchar(13); putchar(10);}
#define putc putchar
```

```
//////////////////////////////// Constants used for RESTART_CAUSE()
#define WDT_FROM_SLEEP  0
#define WDT_TIMEOUT     8
#define MCLR_FROM_SLEEP 16
#define NORMAL_POWER_UP 24

//////////////////////////////// Constants used for SETUP_COUNTERS()
#define RTCC_INTERNAL   0
#define RTCC_EXT_L_TO_H 32
#define RTCC_EXT_H_TO_L 48
#define RTCC_DIV_2      0
#define RTCC_DIV_4      1
#define RTCC_DIV_8      2
#define RTCC_DIV_16     3
#define RTCC_DIV_32     4
#define RTCC_DIV_64     5
#define RTCC_DIV_128    6
#define RTCC_DIV_256    7
#define WDT_18MS        8
#define WDT_36MS        9
#define WDT_72MS        10
#define WDT_144MS       11
#define WDT_288MS       12
#define WDT_576MS       13
#define WDT_1152MS      14
#define WDT_2304MS      15
#define L_TO_H          0x40
#define H_TO_L          0

#define RTCC_ZERO         0x0B20    // Used for ENABLE/DISABLE INTERRUPTS
#define INT_RTCC          0x0B20    // Used for ENABLE/DISABLE INTERRUPTS
#define RB_CHANGE         0x0B08    // Used for ENABLE/DISABLE INTERRUPTS
#define INT_RB            0x0B08    // Used for ENABLE/DISABLE INTERRUPTS
#define EXT_INT           0x0B10    // Used for ENABLE/DISABLE INTERRUPTS
#define INT_EXT           0x0B10    // Used for ENABLE/DISABLE INTERRUPTS

#define GLOBAL            0x0BC0    // Used for ENABLE/DISABLE INTERRUPTS

//////////////////////////////////// Constants used for Timer1 and Timer2
#define T1_DISABLED       0
#define T1_INTERNAL       5
#define T1_EXTERNAL       7
#define T1_EXTERNAL_SYNC  3
#define T1_CLK_OUT        8
#define T1_DIV_BY_1       0
#define T1_DIV_BY_2       0x10
#define T1_DIV_BY_4       0x20
#define T1_DIV_BY_8       0x30
```

```
#byte   TIMER_1_LOW=       0x0e
#byte   TIMER_1_HIGH=      0x0f
#define T2_DISABLED        0
#define T2_DIV_BY_1        4
#define T2_DIV_BY_4        5
#define T2_DIV_BY_16       6
#byte   TIMER_2=           0x11

#define INT_TIMER1         0x8C01    // Used for ENABLE/DISABLE INTERRUPTS
#define INT_TIMER2         0x8C02    // Used for ENABLE/DISABLE INTERRUPTS

//////////////////////////////////// Constants used for SETUP_CCP1()
#define CCP_OFF                    0
#define CCP_CAPTURE_FE             4
#define CCP_CAPTURE_RE             5
#define CCP_CAPTURE_DIV_4          6
#define CCP_CAPTURE_DIV_16         7
#define CCP_COMPARE_SET_ON_MATCH   8
#define CCP_COMPARE_CLR_ON_MATCH   9
#define CCP_COMPARE_INT            0xA
#define CCP_COMPARE_RESET_TIMER    0xB
#define CCP_PWM                    0xC
#define CCP_PWM_PLUS_1             0x1c
#define CCP_PWM_PLUS_2             0x2c
#define CCP_PWM_PLUS_3             0x3c
long CCP_1;
#byte   CCP_1    =                 0x15
#byte   CCP_1_LOW=                 0x15
#byte   CCP_1_HIGH=                0x16

#define INT_CCP1           0x8C04    // Used for ENABLE/DISABLE INTERRUPTS

//////////////////////////////////// Constants used for SETUP_CCP2()
long CCP_2;
#byte   CCP_2    =                 0x1B
#byte   CCP_2_LOW=                 0x1B
#byte   CCP_2_HIGH=                0x1C

#define INT_CCP2           0x8D01    // Used for ENABLE/DISABLE INTERRUPTS

//////////////////////////////////// Constants used in SETUP_SSP()
#define SPI_MASTER       0x20
#define SPI_SLAVE        0x24
#define SPI_L_TO_H       0
#define SPI_H_TO_L       0x10
#define SPI_CLK_DIV_4    0
#define SPI_CLK_DIV_16   1
#define SPI_CLK_DIV_64   2
#define SPI_CLK_T2       3
```

```
#define SPI_SS_DISABLED  1
#define SPI_SAMPLE_AT_END 0x80  // Only for some parts
#define SPI_XMIT_L_TO_H  0x40   // Only for some parts
#define INT_SSP            0x8C08    // Used for ENABLE/DISABLE INTERRUPTS



#define INT_RDA            0x8C20    // Used for ENABLE/DISABLE INTERRUPTS
#define INT_TBE            0x8C10    // Used for ENABLE/DISABLE INTERRUPTS

#list
```

Parte III - Exemplos

Nesta parte teremos vários exemplos com o objetivo de fixar os conceitos vistos até agora, inclusive a personalização do compilador.

Estes exemplos tem como objetivo apenas mostrar a facilidade de projeto usando-se a linguagem C, e não o desenvolvimento de sistemas práticos ou comerciais.

III.1 - Exemplo 1: Acessando as portas de I/O.

Vamos fazer um exemplo simples, que apenas fica incrementando o port B indefinidamente, sendo a taxa de incremento definida pelo overflow do timer 0 em seu minimo de tempo, isto é, clock interno e prescaler 1:1.

Não usamos o watch dog e nem rotinas de interrupção. O overflow do timer 0 será detectado por varredura no bit T0IF.

Em assembler:

```
        list p=16F84                    ; processador 16F84

        __config    _XT_OSC & _CP_OFF & _WDT_OFF & _PWRTE_ON

        radix   dec                     ; define padrao DECIMAL
        include     <P16F84.INC>

        org     0

        clrf        PORTB               ; portb zerado
        bsf         STATUS,RP0          ; banco 1
        clrf        TRISB               ; portb saida
        movlw       B'11011111'         ; timer interno 1:1
        movwf       OPTION_REG          ; escreve em option
        bcf         STATUS,RP0          ; banco 0
espera:
        btfss       INTCON,T0IF         ; ve se timer estourou
        goto        espera              ; nao estourou, volta
estourou:
        incf        PORTB
        bcf         INTCON,T0IF         ; zera bit de interr
        goto        espera              ; espera outro estouro
```

Em C: **(arquivo fonte: incpb.c)**

```
#include <16F84.H>                          // Define processador
#include   <c:\0ccs\mudancas.inc>           // minhas personalizações
#fuses XT,NOWDT,NOPROTECT,PUT               // Define os fusíveis
#use delay(clock=4000000)                   // Informa o clock para rotinas
                                            // de tempo.
#use FAST_IO(B)                             // portB sob meu controle

#byte          portb  =  6
#bit     T0IF = 0x0B.2

// ***************************************************************************


main()
{
        portb = 0;
        set_tris_b ( 0 );
        setup_counters ( rtcc_internal, wdt_18ms );
        while ( 1 )
        {
                while ( !T0IF );                // espera enquanto T0IF NÃO é 1
                T0IF = 0;                       // zera  o flag
                portb = ++;
        }
}
                              ==============//=============
```

III.2 - Exemplo 2: Usando a interrupção do Timer 0

Neste exemplo incrementamos o port B indefinidamente, sendo a taxa de incremento definida pela interrupção do timer 0.

Para simulação no MpLab, devemos manter a definição ESTUDOS no momento da compilação, de forma a ajustar o prescaler para o timer em 1:4.

Para execução real na placa, devemos eliminar ou comentar a definição ESTUDOS no momento da compilação, de forma a ajustar o prescaler para o timer em 1:256, tornando visível a variação do port B.

Listagem: (arquivo fonte: timer0.c)

```
// ***************************************************************************************
//
//      Programa para o estudo do compilador C da CCS
//
//      Incrementa o PORTB a cada overflow do timer 0
//      Para uso real, "comentar" a definicao ESTUDOS (Timer 1:4).
//      Para uso no MpLab, manter a definicao ESTUDOS (Timer 1:256).
//
//      Usamos FAST_IO, programando os regs TRIS conforme as necessidades,
//      gerando codigo menor e com menos uso da RAM interna do que o
//      STANDARD_IO.
//      Ajustar FUSES e DELAY conforme necessario.
//
// ***************************************************************************************

                                // cabecalho inicial
#include <16F84.H>
#include   <c:\0ccs\mudancas.inc>              // minhas personalizações
#fuses XT,NOWDT,NOPROTECT,PUT
#use delay(clock=4000000)
#use fast_io(A)
#use fast_io(B)

// ***************************************************************************************

#define        ESTUDOS            // comentar esta linha para gravar no chip
#byte  portb = 6
```

```
// ************************************************************************************

void main()
{
        portb = 0;                                      // inicia  variável em 0
        TrisB(0);

        #ifdef ESTUDOS
           timer0 (RTCC_internal,rtcc_div_4);           // timer 1:4
        #else
           timer 0(RTCC_internal,rtcc_div_256);         // timer 1:256
        #endif

        set_rtcc(0);                                    // tmr0 inicia em 0
        enable_interrupts(RTCC_ZERO);
        EnableAll;

        for(;;)
        {
                // nada faz no programa principal
        }
}

// ************************************************************************************

#int_rtcc
void TmrOvf()
{
        portb ++;
}

                              ==============//=============
```

III.3 - Exemplo 3: Usando a EEPROM interna do 16F84

Este exemplo é similar ao exemplo 5 do livro.

Ao ligar indica nos leds o endereço 32 (no meio da eeprom).

Pela tecla S0 diminuímos o endereço da eeprom até chegar ao 0.

Pela tecla S1 aumentamos o endereço da eeprom até chegar ao 63.

Pela tecla S2 gravamos no endereço indicado o valor “endereço OR 128”, fazendo então o bit 7 = 1.

Pela tecla S3 lemos o endereço indicado nos leds. Se a gravação foi efetuada com sucesso, devemos enquanto a tecla estiver pressionada ver o endereço COM o led 7 aceso.

Listagem: (arquivo fonte: eeprom.c)

```
// ************************************************************************************
//
//      Programa exemplo  para o compilador C da CCS.
//      Gravar e ler da eeprom do 16F84
//
// ************************************************************************************

//              Cabeçalho inicial com diretivas de compilação

#include <16F84.H>                          // Define processador
#include   <c:\0ccs\mudancas.inc>           // minhas personalizações
#fuses XT,NOWDT,NOPROTECT,PUT               // Define os fusiveis

#use delay(clock=4000000)                   // Informa o clock para rotinas
                                            // de tempo.

#use FAST_IO(A)                             // portA sob meu controle
#use FAST_IO(B)                             // portB sob meu controle

#byte          porta  =  5                  // declaracoes de registros de hardware
#byte          portb  =  6
#bit    S0 = porta.0
#bit    S1 = porta.1
#bit    S2 = porta.2
#bit    S3 = porta.3
```

```
// ********************************************************************************

main()
{
        valor = 32;                     // endereco inicial da eep
        leds =  valor;
        portb = leds;                   // escrevo no port b para visualizacao.
        TrisB ( 0 );                    // portb saida

        while (1)
        {
                if ( (S0 == 0) && (valor < 64) )    // se puder, incrementa
                {
                        valor++;
                        portb = valor;          // escrevo no port b para visualizacao.
                        Delay_Ms(250);          // bounce
                }

                if ( (S1 == 0) && (valor > 0) )     // se puder, decrementa
                {
                        valor--;
                        portb = valor;          // escrevo no port b para visualizacao.
                        Delay_Ms(250);          // bounce
                }

                if (S2 == 0)
                        write_eeprom(valor, valor | 128);

                if (S3 == 0)
                {
                        portb =  read_eeprom(valor);        // le e escreve nos leds
                        while (S3 == 0);                    // espera soltar S3
                        portb =  valor;                     // recupera os leds
                                                            // anteriores.
                }
        }
}
//==========       FIM       ==================================
```

III.3.4 - Exemplo 4: LCD inteligente com tabela de mensagens em ROM

Neste exemplo temos um display com controlador Hitachi 44780.
Devido ao fato do pic não permitir ponteiros para posições em ROM, cada tabela deve ser separadamente escrita.

Listagem: (arquivo fonte: lcd_rom.c)

```
// ***************************************************************************
//
//      LCD ligado a PLACA PicLab 1
//      Fica mostrando várias mensagens alternadamente
//
//      Data: 18/08/98
//
// ***************************************************************************

#include <16F84.H>                          // Define processador
#include <\0CCS\mudancas.inc>               // minha personalizacao
#fuses XT,NOWDT,NOPROTECT,PUT               // Define os fusiveis
#use delay(clock=4000000)                   // Informa o clock para rotinas
                                            // de tempo.
#use FAST_IO(A)                             // portA sob meu controle
#use FAST_IO(B)                             // portB sob meu controle

#byte   porta = 5                       // endereco do port a
#byte          display =  6                     // display no endereco do port b

#bit    ENB = porta.2                   // sinal ENABLE
#bit    RS  = porta.3                   // sinal RS

// ***************************************************************************
                        // Prototipos obrigatorios com ";" no final.

void Inicializa(void);              // nao recebem e nem devolvem valores
void Limpa(void);
void Escreve(void);
void Mensagem(void);
void Linha1(char coluna);           // recebem um char, ja indicando o nome
void Linha2(char coluna);

// ***************************************************************************
                                // Declaracao de variaveis em RAM
char    i, posicao, dado;
char    MensRam [32];               // devo copiar da ROM para a RAM porque
                                    // no pic nao posso fazer ponteiros para
                                    // a memoria ROM.
```

```
// ********************************************************************************
                // Declaracao de tabelas constantes em ROM
                // a string deve ter +1 alem do tamanho real

byte const Mensg_1 [33] = { "VIDAL   Projetos Personalizados " };
byte const Mensg_2 [33] = { "Cursos sobre PIC  com material  " };
byte const Mensg_3 [33] = { "1- Basico, ideal para iniciantes" };
byte const Mensg_4 [33] = { "2- Linguagem C     para usuarios" };
byte const Mensg_5 [33] = { "3- Especiais sob    consulta    " };
byte const Mensg_6 [33] = { "Ligue:         (011) 6451-8994 " };

// ********************************************************************************
void main()
{
        TRISA(0b11110011);                      // PortA: Bits 2 e 3 saidas.
        TRISB(0);                               // PortB: Saida dados LCD

        RS = 0;
        ENB = 0;

        Inicializa();                           // inicializa o display

        while(1)
        {
                for (i=0; i<32; i++)            // copio a mensagem da ROM
                        MensRam[i] = Mensg_1 [i];// para a RAM
                Mensagem();                     // escrevo a mensagem
                Delay_Ms(1000);                 // espero 1 seg
                Limpa();                        // limpo o display

                for (i=0; i<32; i++)
                        MensRam[i] = Mensg_2 [i];
                Mensagem();
                Delay_Ms(1000);
                Limpa();

                for (i=0; i<32; i++)
                        MensRam[i] = Mensg_3 [i];
                Mensagem();
                Delay_Ms(1000);
                Limpa();

                for (i=0; i<32; i++)
                        MensRam[i] = Mensg_4 [i];
                Mensagem();
                Delay_Ms(1000);
                Limpa();
```

```
                for (i=0; i<32; i++)
                        MensRam[i] = Mensg_5 [i];
                Mensagem();
                Delay_Ms(1000);
                Limpa();

                for (i=0; i<32; i++)
                        MensRam[i] = Mensg_6 [i];
                Mensagem();
                Delay_Ms(1000);
                Limpa();
        }
}

// ***************************************************************************
void Inicializa()               // sequencia de inicializacao do LCD
{
        RS = 0;

        display = 56;           ENB = 1;        ENB = 0;        // várias instruções numa
        Delay_Ms(5);                                            // só linha

        display = 56;           ENB = 1;        ENB = 0;
        Delay_Ms(5);

        display = 6;            ENB = 1;        ENB = 0;
        Delay_Ms(5);

        display = 14;           ENB = 1;        ENB = 0;
        Delay_Ms(5);

        display = 1;            ENB = 1;        ENB = 0;
        Delay_Ms(5);
}

// ***************************************************************************
void Mensagem()
{
        Linha1(1);                                      // vou para a linha 1 coluna 1

        for (posicao=0;posicao<32;posicao++)
        {                                               // loop de 32 vezes para
                Delay_Us(50);                           // escrever byte a byte
                display = MensRam [posicao];
                RS = 1;
                ENB = 1;
                ENB = 0;
```

```
                Delay_Ms(50);                   // 1 ms entre caracteres

                if (posicao == 15)    // deve ir para a linha 2
                {
                        Delay_Us(50);
                        Linha2(1);
                }
        }
        Delay_Us(50);
}

// ***************************************************************************
void Limpa()
{
        display = 1;            // apaga display
        RS = 0;         ENB = 1;        ENB = 0;
        Delay_Ms(2);            // 2 ms apos limpar display
}

// ***************************************************************************
void escreve()
{
        RS = 1;         ENB = 1;        ENB = 0;
        Delay_Us(50);
}

// ***************************************************************************
void Linha1(Coluna)
{
        display = 127+Coluna;                   // posiciona linha 1 col X
        RS = 0;         ENB = 1;        ENB = 0;
        Delay_Us(50);
}

// ***************************************************************************
void Linha2(Coluna)
{
        display = 191+Coluna;                   // posiciona linha 2 col X
        RS = 0;         ENB = 1;        ENB = 0;
        Delay_Us(50);
}

// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   FIM    <<<<<<<<<<<<<<<<<<<<<<<<<<<
```

III.3.5 - Exemplo 5: Teste da funcao "Putchar()"

Neste exemplo veremos a facilidade de se enviar dados na forma serial com o uso de um função especialmente desenvolvida para este fim.

Listagem: (arquivo fonte: trx_byte.c)

```
// ********************************************************************
//      Programa para enviar serialmente a 9600 bps um byte.
//      Usa funcao 'putchar' da diretiva #use rs232
//
//      A cada press de teclas, envia um byte:
//              S0: envia 55H (01010101)
//              S1: envia 66H (01100110)
//              S2: envia ccH (11001100)
//              S3: envia bbH (10111011)
//
//      18/08/97
// ********************************************************************
#include <16F84.H>                              // Define processador
#include <\0CCS\mudancas.inc>                   // minha personalizacao
#fuses XT,NOWDT,NOPROTECT,PUT                   // Define os fusiveis
#use delay(clock=4000000)                       // Informa o clock para rotinas
                                                // de tempo.
#use FAST_IO(A)                                 // portA sob meu controle
#use FAST_IO(B)                                 // portB sob meu controle

#use rs232(baud=9600,XMIT=RB7,RCV=RB6,bits=8)       // indispensável

#byte   porta = 5                               // endereco do port a
#byte           portb = 6                               // endereco do port b

#bit S3 = PORTA.3                               // Teclas S0 a S3
#bit S2 = PORTA.2
#bit S1 = PORTA.1
#bit S0 = PORTA.0

#bit Trigger = PORTB.0                          // Saida Trigger p/ scope

// ********************************************************************
                        // Prototipos obrigatorios com ";" no final.

void VeTeclas(void);

// ********************************************************************
                        // Declaracao de variaveis em RAM
char TrxReg;
```

```
// ********************************************************************
void main()
{
        TRISA(255);                             // PortA: Entrada teclas
        TRISB(0b01111110);                      // PortB: RB7 e RB0 saidas

        Trigger = 0;                            // reseta trigger para 0

        for (;;)
        {
                Delay_Ms(2);
                VeTeclas();

                if (TrxReg < 255)
                {
                        Trigger = 1;            // aciona o trigger
                        NOP;
                        Trigger = 0;

                        putchar(TrxReg);        // envia o byte
                }
        }
}

// ********************************************************************
void VeTeclas()
{
        TrxReg = 255;           // padrao para nenhuma tecla acionada

        if (S0 == 0) TrxReg = 0b10101010;
        if (S1 == 0) TrxReg = 0b01100110;
        if (S2 == 0) TrxReg = 0b11001100;
        if (S3 == 0) TrxReg = 0b10111011;
}
```

III.3.6 - Exemplo 6: Ligação pic à pic serial.

Este exemplo será dividido em dois módulos:

A) Transmissão

Ao reset a variável *valor* é inicializada com o 64. A cada toque na tecla S0 este valor é incrementado, e se chegar a 128 recomeça do 0 (varia de 0 à 127 = 7 bits).

Ao pressionar a tecla S1, a variável valor será transmitida serialmente a 9600 bps. Para permitir visualização no osciloscópio um pulso de trigger será gerado a cada byte enviado, no pino RA2.

B) Recepção

A placa receptora ficará na espera com a função GetChar( ).
O dado recebido será escrito no port B, leds 0 à 6, reproduzindo então o dado enviado pela placa transmissora.

Transmissão

Listagem: (arquivo fonte: enviap2p.c)

```
// ********************************************************************
//
//      Programa para enviar serialmente a 9600 bps um byte para outra
//      placa PicLab 2.
//
//      Funcao dos pinos:
//              RB7: nao usada
//              RB6~RB0: mostra o valor de 0 a 127
//              RA0: incrementa valor (tecla S0)
//              RA1: envia o valor (tecla S1)
//              RA2: saida trigger para o scope
//              RA3: XMIT
//              RA4: entrada RCV, nao usada
//
//      18/08/97
//
// ********************************************************************

#include <16F84.H>                          // Define processador
#include <\0CCS\mudancas.inc>               // minha personalizacao
#fuses XT,NOWDT,NOPROTECT,PUT               // Define os fusiveis
#use delay(clock=4000000)                   // Informa o clock para rotinas
                                            // de tempo.
```

```
#use FAST_IO(A)                                 // portA sob meu controle
#use FAST_IO(B)                                 // portB sob meu controle

#use rs232(baud=9600,XMIT=RA3,RCV=RA4,bits=8)       // INDISPENSÁVEL

#byte   porta = 5                               // endereco do port a
#byte  portb = 6                                // endereco do port b

#bit S1 = PORTA.1                               // teclas S0 e S1 no portA
#bit S0 = PORTA.0

#bit Trigger = PORTA.2                          // Saida Trigger p/ scope

// ***********************************************************************
char valor, leds;

void main()
{
        TRISA(0b11110011);              // PortA: 2 e 3 entrada de teclas
        TRISB(0b10000000);              // PortB: RB7 entrada, RB6~RB0 saidas

        Trigger = 0;                    // reseta trigger para 0
        valor = 64;
        leds = valor;
        portb = leds;

        for (;;)
        {
                if (S0 == 0)
                {
                        valor++;
                        if (valor == 128) valor = 0;
                        leds = valor;
                        portb = leds;
                        Delay_Ms(150);
                }

                while (S1 == 0)
                {
                        Trigger = 1;            // aciona o trigger
                        NOP;
                        Trigger = 0;
                        putchar(valor);                 // envia o byte
                        Delay_Ms(5);
                }
        }
}
```

Recepção

Listagem: (arquivo fonte: getchar.c)

```
// ********************************************************************
//
//      Programa para receber serialmente a 9600 bps um byte vindo de outra placa
//      Fica esperando chegar (funcao Getchar() do compilador)
//
//      Funcao dos pinos:
//              RB7: RCV
//              RB6~RB0: mostra o valor recebido
//              RA4: XMIT, nao usada
// ********************************************************************

#include <16F84.H>                          // Define processador
#include <\0CCS\mudancas.inc>               // minha personalizacao
#fuses XT,NOWDT,NOPROTECT,PUT               // Define os fusiveis
#use delay(clock=4000000)                   // Informa o clock para rotinas
                                            // de tempo.
#use FAST_IO(A)                             // portA sob meu controle
#use FAST_IO(B)                             // portB sob meu controle

#use rs232(baud=9600,XMIT=RA4,RCV=RB7,bits=8)       // INDISPENSÁVEL

#byte   porta = 5                           // endereco do port a
#byte  portb = 6                            // endereco do port b

// ********************************************************************
                        // Declaracao de variaveis em RAM
char valor, leds;
void main()
{
        TRISB(0b10000000);              // PortB: RB7 entrada, RB6~RB0 saida leds

        valor = 255;                    // leds todos apagados
        leds = valor;
        portb = leds;

        for (;;)
        {
                valor = getchar();
                leds = valor;
                portb = leds;
                Ms(10);
        }
}
```