Projeto de Sistemas Microcontrolados

Aula 06 - Fundamentos de Linguagem C - Parte I

Apresentação

Nesta aula, o direcionamento do aluno será voltado ao aprendizado de alguns conceitos básicos relacionados à linguagem C para microcontroladores. A utilização de uma linguagem de alto nível, como o C, fará com que o aluno interaja menos com o *hardware*, em relação ao controle do mesmo, dedicando, assim, mais tempo à lógica do problema do que aos detalhes internos construtivos do *chip*.

Serão introduzidas também algumas diretivas de dois compiladores que poderão ser utilizados durante esta e as próximas aulas, o C18 e o CCS. O aluno, mesmo com conhecimento de C, deve ficar atento a algumas características ou funções especiais desses dois compiladores, não comuns à linguagem C padrão.

Objetivos

Ao final desta aula você será capaz de:

- Definir quais as principais características da Linguagem C, sua sintaxe e tipos de dados que podem ser manipulados.
- Utilizar modificadores de tipos como forma de adequação na manipulação das variáveis presentes em um programa escrito em C.
- Descrever e utilizar as principais diretivas de compilação dos compiladores C18, da Micro*chip*, e do CCS, da Custom Computer Services Inc.

A linguagem C

A linguagem C foi desenvolvida por Dennis Ritchie, do Bell Laboratories, e implementada, originalmente, em um computador DEC PDP-11, em 1972. Essa linguagem é conhecida pelo seu equilíbrio entre estruturação e eficiência, eliminando as deficiências da linguagem *Assembly*, mas, ainda, mantendo baixo o consumo de memória.

Diferentemente de *Assembly*, a linguagem C é compilada, ou seja, o programa-fonte escrito pelo programador é processado e traduzido por outro programa, conhecido por COMPILADOR, que irá gerar um programa equivalente, em linguagem de máquina (programa-objeto) o qual, por sua vez, passa por um processo de ligação efetuado por um programa denominado de LINKER (ou ligador), gerando um código executável. Esse processo você já conhece, não? Afinal, você está vindo de uma maratona de estudos de programação no módulo básico. No entanto, fica aqui uma sugestão: reveja esses conceitos.

Como o C é uma linguagem independente de *hardware* e amplamente disponível, as suas aplicações são escritas de forma estruturada, além de serem executadas com pouca ou nenhuma modificação em uma grande variedade de sistemas computacionais.

Em nosso estudo, envolvendo os compiladores C18 e CCS, será dada especial ênfase aos itens fundamentais que compõem esses dois compiladores. Para tal, precisamos, inicialmente, entender a estrutura básica de um programa em C. Esta estrutura, mostrada no Quadro 1, é dividida em quatro partes, que são: os comentários, as diretivas de compilação, a área de definição de dados e os blocos com instruções e funções.

O uso de comentários em um programa, além de documentá-lo, irá permitir melhor compreensão do código escrito pelo programador. Considerando que os comentários são ignorados pelo compilador, seu uso, sem extremos abusivos, é uma prática extremamente recomendável ao programador.

As duas maneiras de introduzir comentários em um código C, como podem ser vistas no Quadro 1, são pelo uso de // (duas barras), para comentários de linha e pelo uso dos delimitadores /* (que inicia um comentário de múltiplas linhas) e */ que concluí o comentário.

Exemplos

// Esse é o padrão para inserção de um comentário em uma única linha.
/* Esse é o padrão para inserção de um comentário em múltiplas e consecutivas linhas de código.
Não existe limite do número de linhas para o comentário, cabendo apenas ao programador ser o mais objetivo e claro quanto possível */

Um erro comum, em se tratando de comentários, é o programador esquecer-se de encerrar um comentário com */, ou até mesmo, começar um comentário com os caracteres */ ou terminar com /*.

```
/* um comentário pode ser
colocado
em qualquer parte do código*/
```

// Comentários

```
#include<.....>
#define ......
```

// Diretivas de compilação

```
int x, y;
float temperature;
```

// Área de definição
// de dados (variáveis)

```
void main() {
instruções do programa principal
}

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

// Blocos com instruções
// e funções

Quadro 1 – Estrutura básica de um programa escrito em Linguagem C

Após a apresentação da estrutura básica de um programa escrito em C, passaremos ao estudo de itens considerados fundamentais aos compiladores tratados neste material. Essa estrutura será retomada quando formos definir, na aula 7, nossos *templates* para edição de programas em C para os compiladores CCS e C18, como fizemos para programas escritos em *Assembly*, na aula 5.

Identificadores

Identificadores são os nomes atribuídos pelo programador às constantes, variáveis, tipos, funções, programas e campos de um registro. Portanto, escolher nomes significativos para os identificadores **variáveis** e **constantes** ajuda a tornar um programa autoexplicativo, isto é, menos comentários se farão necessários. Para se definir um identificador, devemos observar o seguinte:

- Não são permitidos identificadores com o mesmo nome.
- Deve ter como primeiro caractere uma letra ou um caractere “_”.
- Após a primeira letra, só pode conter letras, dígitos ou caracteres “_”.
- Não pode conter espaços.
- Não pode ser uma palavra reservada.
- Letras maiúsculas e minúsculas são tidas como diferentes.

- Somente os 31 primeiros caracteres do identificador são válidos, no entanto, um identificador pode utilizar mais do que esse número de caracteres.

No Quadro 2, é possível verificar alguns identificadores válidos e inválidos.

Identificador válido	Identificador inválido
nome_usuario	nome usuario
_numero	@numero
telefone1	telefone...1
TotalPagamentos	1totalPagamentos

Quadro 2 – Representação de alguns identificadores válidos e inválidos

Um detalhe importante da linguagem C está relacionado à distinção que ela faz entre letras maiúsculas e minúsculas, ou seja, a linguagem C é *case sensitive*.

Um erro muito comum é o programador, ao digitar, usar uma letra maiúscula no lugar de uma letra minúscula, como por exemplo: digitar ***Nome*** em vez de ***nome.*** Para o C, trata-se de dois identificadores totalmente distintos.

Atividade 01

1. Após rever os conceitos sobre compiladores, apresente um algoritmo mostrando os passos que envolvem desde a edição do programa fonte até a geração de um programa executável, citando que programas estão envolvidos em cada passo do fluxograma.
2. Escreva dez identificadores válidos em C.

Palavras-chave

A linguagem C possui um conjunto de palavras reservadas ou comandos pré-definidos, que não podem ser utilizados como identificadores. No Quadro 3, pode-se observar algumas palavras-chave reservadas à linguagem C. São nomes de comandos ou de diretivas do próprio C, como por exemplo, short, goto, return ou nomes de estruturas de programação, como por exemplo, while, for e case.

auto	break	case	char	const	continue	default	do
double	else	enum	extern	far	float	for	goto
if	Int	long	near	overlay	ram	register	return
rom	short	signed	sizeof	static	struct	switch	typedef
union	unsigned	void	volatile	while			

Quadro 3 – Algumas palavras-chave reservadas pela Linguagem C

Tipos de dados

As linguagens de programação disponibilizam uma gama de tipos de dados, os quais definem os valores que podem ser armazenados. Os tipos de dados suportados pelos compiladores CCS e C18 são quase todos os tipos de dados disponíveis em C, padrão ANSI. Com isso, é possível construir programas de grande complexidade com relativa facilidade.

Os tipos básicos de dados para o compilador CCS estão listados no Quadro 4.

Tipo	Tamanho em *bits*	Valor
char	8	0 a 255
int	8	0 a 255
float	32	3.4E-38 a 3.4E+38
void	0	nenhum valor

Quadro 4 – Tipos de dados(compilador CCS)

O tipo **char** é utilizado para representar os caracteres ASCII de 8 *bits*.

O tipo **int** da linguagem C padrão ANSI foi definido para sempre possuir o tamanho mais eficiente, na representação de números inteiros, para a arquitetura-alvo: 8 *bits*, no caso do compilador C da CCS, e 16 *bits,* para a linguagem C do MPLab C18.

Caso esteja resolvendo uma atividade e ela necessite realizar operações com números fracionários, você deve utilizar um tipo apropriado para isso, que é o **float**.

Temos ainda o **void,** um tipo usado, normalmente, em funções do C para declarar que ela, a função, não deve retornar nenhum valor.

Modificadores de tipos

Além dos tipos de dados básicos vistos no Quadro 4, existem, ainda, os modificadores de tipos, que são empregados para modificar os tipos básicos, de forma a obter outros tipos de dados.

Os modificadores usados tanto em compiladores gerais na linguagem C quanto nos compiladores CCS e C18 são os seguintes: **signed, unsigned, short** e **long**.

O modificador **signed** é empregado para alterar um tipo base, de forma que ele possa representar tanto números positivos quanto números negativos. A representação de números negativos é realizada tomando o ***Bit* mais significativo** (de *Most Significant Bit* – MSB) da variável para representar o sinal: *bit* MSB = 1, sinal negativo, *bit* MSB = 0, sinal positivo. Alguns exemplos estão representados no Quadro 5.

Binário	com sinal	sem sinal
01111111	127	127
10000000	-0	128
10000001	-1	129

Quadro 5 – Representação binária de alguns números com e sem sinal (representação complemento de um)

Nota-se, então, que, ao se utilizar um *bit* na representação do sinal, a magnitude absoluta de representação do tipo modificado será metade da magnitude do tipo não modificado.

Assim, para o compilador CCS, uma variável do tipo **signed int** pode representar valores entre -128 e +127, em vez de 0 a 255.

O modificador **unsigned,** além de ser padrão no compilador CCS, define ainda um tipo de dado sem sinal. Dessa forma, não é necessário utilizar esse tipo de modificador, pois ele não produzirá efeito algum.

Já os modificadores **short** e **long** são usados para definir os tamanhos dos tipos de dados básicos: o primeiro modificador especifica uma variável com tamanho menor que o tipo base, e o segundo modificador é empregado para ampliar a magnitude da representação do tipo especificado.

Tipos específicos do compilador CCS

O compilador CCS dispõe de outros tipos de dados, além dos já mencionados. Esses foram criados, especificamente, para a eficiência e compatibilidade do compilador CCS junto às famílias de microcontroladores PIC.

- **int1:** especifica valores de 1 *bit*.
- **boolean:** especifica valores booleanos de *bit*.
- **int8:** especifica valores de 8 *bits*.
- **byte:** especifica valores de 8 *bits*.
- **int16:** especifica valores de 16 *bits*.
- **int32:** especifica valores de 32 *bits*.

Variáveis e constantes

Já que conhecemos os tipos de dados que são manipulados pelos compiladores, iremos agora aprender os significados de variável e de constante, e como reservar espaços de memória para elas.

Tanto uma variável quanto uma constante são espaços reservados na memória, onde um valor pode ser armazenado para ser utilizado por um programa. Uma variável se diferencia de uma constante, porque ela pode assumir novas atribuições de valor durante a execução do programa. Porém, o valor de uma constante é imutável, ou seja, o valor que ela assume no início de um programa será o mesmo ao término de sua execução.

Para reservar um espaço na memória para uma variável, basta, primeiramente, especificar o tipo e, em seguida, um nome para ela.

Sintaxe

```
1 <tipo da variável> <nome da variável>;
```

Observe a necessidade do uso do ; (ponto e virgula) após a declaração.

Exemplos

```
float media;          // Definição de uma variável chamada media do tipo float
int x;               // Definição de uma variável chamada x do tipo int
int val = 10;          // Declaração de uma variável chamada val do tipo int sendo
                      // inicialmente lhe atribuída o valor 10
char a = 'x';          // Declaração de uma variável chamada a do tipo char, com
                      // atribuição inicial do caractere ASCII x.
```

Observa-se, ainda, que uma variável pode ser **global** ou **local**. Variável **global** é aquela declarada fora do escopo das funções. Variável **local** é o tipo de variável declarada no início de um bloco e seu escopo está restrito ao bloco em que foi declarada. Nas linguagens que estudou também era assim? Se não, que outras declarações poderia haver para uma variável?

No caso das definições de constantes, o programador deve atentar para o uso do qualificador **const**, usado para evitar a modificação do valor da variável. O uso dos qualificadores será explicado com mais detalhes na próxima seção.

Sintaxe

```
<qualificador> <tipo da variável> <nome da variavel> = <valor>;
```

Exemplos

```
const int maximo = 4;
/* Definição de uma constante (qualificação const) de nome maximo do tipo int. Durante o process
todas as ocorrências da constante simbólica chamada maximo serão substituídas pelo valor que lh
na linha de comando, ou seja, pelo valor 4 */
```

Atividade 02

1. Diferencie uma variável de uma constante.
2. O que você entende por modificadores de tipo?
3. Identifique outras palavras-chave que não possam ser usadas como identificadores na Linguagem C.

Operadores

Em relação aos operadores, C é, talvez, a linguagem que possui um maior número, pois, além de apresentar todos os operadores comuns de uma linguagem de alto nível, apresenta, também, operadores usuais às linguagens de baixo nível. Os operadores que serão apresentados nesta seção estão classificados nas seguintes categorias: Atribuição; Aritmético; Bit a bit; Lógico ou Booleano; Relacional; Acesso a endereço ou de manipulação com ponteiro.

Operador de atribuição

O operador de atribuição "=" é a categoria mais utilizada em C, pois é empregado sempre que houver a necessidade de passar um dado a uma determinada variável ou para atribuir um valor a uma constante.

Sintaxe

```
<identificador>  = <expressão>;
```

Onde, **identificador** é o nome da variável ou constante e **expressão** pode ser outra variável ou constante, uma operação matemática, uma expressão ou um valor.

Exemplos:

```
x  = 10;
y  = x;
```

Operadores aritméticos

São operadores responsáveis por realizarem operações matemáticas entre variáveis e/ou constantes do mesmo tipo ou de tipos diferentes. No Quadro 6, estão representados os operadores matemáticos aceitos no C, bem como sua precedência (da maior para a menor), e no Quadro 7, são mostrados alguns exemplos.

Operador	Ação	Precedência
-- , ++	Decremento e Incremento	Maior
-	Menos unário	
* , /	Multiplicação e Divisão	
%	Módulo (resto de divisão inteira)	
- , +	Subtração e Adição	Menor

Quadro 6 – Operadores aritméticos

Sintaxe

```
1 <operando 1> <operador  aritmético> <operando 2>;
```

Expressão	Resultado
1 + 2	3
5.0 – 1	4.0
2 * 1.5	3.0
5.0 / 2.0	2.5 (divisão real)
5 / 2	2 (divisão inteira)

Expressão	Resultado
5 % 2	1 (resto da divisão inteira)

Quadro 7 – Exemplos de uso de alguns operadores aritméticos com seus respectivos resultados

Um erro comum de programação é usar o operador **%**(módulo) com operandos não inteiros, ou seja, que não são do tipo **int**.

Operadores bit a bit

Esses operadores são utilizados para manipular operações com *bits* de variáveis dos tipos inteiros **int** e **long** e do tipo **char**. As operações possíveis são as mostradas no Quadro 8.

Operador	Ação
&	AND (E)
|	OR (Ou)
^	XOR (ou exclusivo)
~	NOT (inverte o estado dos bits)
>>	Deslocamento de *bits* à direita
<<	Deslocamento de *bits* à esquerda

Quadro 8 – Operações bit a bit

Exemplos

```
unsigned char x, y, z;        //Declaração de 3 variáveis unsigned char.
unsigned int resposta;       //Declaração de uma variável int de 16 bits


x = 0b00101001; // Atribui o valor binário 0010 1001 à variável x.
y = 0b10001000; // Atribui o valor binário 1000 1000 à variável y.
z = 0b01110101; // Atribui o valor binário 0111 0101 à variável z.


resposta = x & y;                          // resposta = 0000 0000 0000 1000
resposta = y | z;                           // resposta = 0000 0000 1111 1101
resposta = ((resposta&0x00ff)&y)<<8;  // resposta = 1000 1000 0000 0000
resposta |= x;                              // resposta = 1000 1000 0010 1001
resposta = ~resposta                       // resposta = 0111 0111 1101 0110
```

Operadores relacionais

Os operadores relacionais (mostrados no Quadro 9) são empregados em testes condicionais, a fim de verificar se uma determinada condição é verdadeira (1) ou falsa (0).

Operador	Ação
==	Igual a
!=	Diferente de
>	Maior que
>=	Maior ou igual a
<	Menor que
<=	Menor ou igual a

Quadro 9 – Operadores relacionais

Sintaxe

```
1 <operando1>  <operador relacional> <operando2>
```

Exemplos

Condição	Resultado	Valor retornado
10 == 11	Falso	0
6 != 10	Verdadeiro	1
15 > 15	Falso	0
15 >= 15	Verdadeiro	1
20 < 30	Verdadeiro	1
10 <= 8	Falso	0

Um erro comum de programação é usar espaço para representar qualquer um dos operadores ==, !=, >= e <= ou fazer a inversão da ordem simbólica, como por exemplo, usar =!, => ou =< ao invés de !=, >= ou <=, respectivamente. Outro erro que ocorre com bastante frequência é confundir o operador relacional de igualdade == (igual a) com o operador de atribuição = (igual).

Operadores booleanos

Os operadores lógicos ou booleanos (mostrados no Quadro 10) são comumente usados em conjunto com os operadores relacionais. Com eles é possível conferir se a combinação de uma ou mais condições resultam em uma condição verdadeira (1) ou falsa (0).

Operador	Ação
&&	AND (E)
||	OR (OU)
!	NOT (Não)

Quadro 10 – Operadores Booleanos

Sintaxe

```
<operando  1> <operador booleano> <operando 2>
```

Exemplos:

Condição	Descrição	Resultado	Valor retornado
(11 == 15) || (5 != 7)	0 || 1 = 1	Verdadeiro	1
(11 == 15) && (5 != 7)	0 && 1 = 0	Falso	0
!(31 > 34) && (34 >= 34)	!0 && 1 = 1	Verdadeiro	1
((4>= 4) || (3 == 4)) && (14 != 3)	(1 || 0) && 1 = 1	Verdadeiro	1
((4>= 4)&& (3 == 4)) && (14 != 3)	(1 && 0) && 1 = 0	Falso	0
((4>= 4)&& (3 == 4)) || (14 != 3)	(1 && 0) || 1 = 1	Verdadeiro	1

Operadores de acesso a endereço

Esses tipos de operadores, mostrados no Quadro 11, são usados com ponteiros para acesso a endereços de memória. Como os ponteiros são variáveis que contêm endereços de memória como valores, são, normalmente, utilizados para simular a

passagem de parâmetros por referência, possibilitando que a função retorne mais de um valor.

Operador	Ação
&	Endereço do operando
*	Conteúdo do endereço apontado pelo operando

Quadro 11 – Operadores de acesso a endereços

A declaração de um ponteiro tem o seguinte formato:

Sintaxe

tipo * identificador;

Sendo:

- **tipo**: tipo de dado válido;
- **identificador:** nome da variável.

Exemplos

```
1  int vet[3] = {4, 13, 40};                    // Declara um vetor de três elementos do tipo int.
2  int *p;                                      // Declara um ponteiro para uma variável do tipo int (16 bit
3  int elemento_0, elemento_1, elemento_2;      // Declara três variáveis do tipo int.
4  p = &vet[0];                                 // O ponteiro p armazena o endereço do primeiro element
5  elemento_0 = *p;                             // O ponteiro retorna o conteúdo do endereço apontado,
6  p++;                                         // Incrementa o endereço em um.
7  elemento_1 = *p;                             // O ponteiro retorna o conteúdo do endereço, no caso 13
8  p++;                                         // Incrementa o endereço em um.
9  elemento_2 = *p;                             // O ponteiro retorna o conteúdo do endereço, no caso 40
10 p++;                                         // Incrementa o endereço em um.
```

Expressões

Expressões são combinações de variáveis, constantes e operadores. De maneira geral, as expressões seguem as mesmas regras gerais das expressões algébricas da matemática.

Exemplos

```
custo – valor;
a = b + 2;
teste * 5;
```

Também é possível a utilização de expressões condicionais, conforme exemplo a seguir:

```
int x, y;
x = 10;
y = (x > 5) * 10;
```

No trecho de código acima, constata-se que a condição avaliada é verdadeira, logo, y será igual a (1) * 10 = 10. Caso a expressão fosse falsa, o resultado da expressão seria dado por y = (0) * 10 = 0.

Isso só é possível porque em C, as expressões são avaliadas e resultam sempre em 0 (falso) ou 1 (verdadeiro).

Qualificadores

Os qualificadores, também conhecidos como modificadores, podem ser divididos em qualificadores de acesso e de armazenamento. O papel principal dos qualificadores é informar ao compilador o modo como os dados devem ser acessados ou modificados.

Qualificadores de acesso

Os compiladores CCS e C18 suportam os qualificadores padrões **volatile** e **const**. O primeiro qualificador indica que o conteúdo da variável pode ser modificado durante a execução do programa. Já o segundo qualificador mostra que o valor da variável não pode ser modificado pelo programa. Sempre que o qualificador **const** for utilizado, se faz necessária a atribuição de um valor no momento da declaração da variável.

Qualificadores de armazenamento

São elementos especiais usados para controlar a maneira como o compilador irá lidar com o armazenamento das variáveis. Os compiladores CCS e C18 fazem uso dos quatro qualificadores definidos em C, padrão ANSI: **auto**; **extern**; **static**; **registe**r. O C18 adiciona mais um qualificador: o **overlay**.

- **auto:** mostra que a variável existirá enquanto o procedimento estiver ativo. Não é necessário utilizar esse modificador, porque, por definição, as variáveis em C possuem escopo local.

- **extern:** é utilizado para especificar variáveis externas ao módulo corrente.

  Para uma melhor compreensão desse qualificador, suponha que dois arquivos (arquivo1.c e Arquivo2.c) tratem uma variável comum (resultado_mult). Caso a declaração da variável resultado_mult, localizada no **Arquivo2.c**, não seja feita com o qualificador **extern,** o compilador gerará um erro informando a ocorrência de multiplicidade de declaração.

Exemplo

```
Arquivo1.c
long resultado_mult;     // Declaração da variável
int x, y;                        //Declaração das variáveis
void main(void) {
   x=40; y=32;
   resultado_mult = multiplicacao(x,y); /*Chamada da função multiplicação que
}

Arquivo2.c
extern long resultado_mult; /*Declaração da variável resultado_mult por meio d
long multiplicacao(int a, int b)
{
   resultado_mult = a*b;
   return resultado_mult;
}
```

- **static:** declara ao compilador que a variável ocupará uma posição permanente na memória, ou seja, ela funcionará como uma variável global no sentido de que não será destruída ao termino da execução da função na qual foi declarada e, de antemão, pode funcionar como uma variável local no aspecto de que não será conhecida fora da função em que foi declarada.

- **register:** indica ao compilador que a variável declarada deve ser alocada como um registrador da CPU. Nos microcontroladores PIC, esse tipo de qualificador não faz sentido, uma vez que as variáveis já são armazenadas e tratadas como registradores.

- **Overlay:** esse qualificador pertence, exclusivamente, ao compilador C18 e tem como função principal alocar, estaticamente, variáveis locais e parâmetros. Sempre que possível, as variáveis locais serão sobrepostas com variáveis locais de outra função.

Qualificadores de armazenamento extras do C18

Para melhor tratar de dados de memória em uma arquitetura Harvard, o compilador C18 introduziu outros quatro qualificadores de memória: **ram**, **rom**, **near** e **far**. Os dois primeiros permitem distinguir se os dados estão localizados numa memória do tipo RAM ou numa memória do tipo ROM, já que essas memórias

não compartilham o mesmo barramento. A utilização dos qualificadores **ram** e **rom** deve preceder a especificação dos qualificadores **far** e **near**, como demonstrado nos exemplos a seguir:

```
ram far int variavel_1;                    // Indica ao compilador que a variável deve ser localizada n
ram near long variavel_2;                  // Indica ao compilador que a variável deve ser localizada n
rom far unsigned int variavel_1 = 4938;    /* Indica ao compilador que a variável pode ser localizada
                                              de programa. */
ram char variavel_2[ ] = {"012345"};       /* Informa ao compilador que a variável deve ser armazen
                                              endereço não pode ultrapassar 64K */
```

Diretivas de compilação

As diretivas de compilação são simples comandos de orientação ao compilador. Esses comandos são, na realidade, utilizados para especificar determinados parâmetros internos utilizados pelo compilador no momento de compilar o código-fonte. Todas as diretivas de compilação são iniciadas pelo caractere **#** e somente caracteres de espaço em branco podem aparecer antes delas em uma linha.

Os compiladores CCS e C18 possuem uma grande quantidade de diretivas, que podem ser utilizadas para incluir arquivos, definir macros e controlar diversos parâmetros. Porém, esse material não visa esgotar todo o assunto pertinente a essas diretivas de compilação, assim, veremos apenas as mais comuns aos compiladores abordados. Caso necessite, acesse o help ou seus respectivos guias de usuário. (Para o CCS, acesse http://www.ccsinfo.com/downloads/ccs_c_manual.pdf e para o C18 acesse http://ww1.microchip.com/downloads/en/devicedoc/51288f.pdf).

Diretivas #asm e #endasm

São utilizadas para inserir código em *assembly* diretamente no programa em C. Os mnemônicos são inseridos entre a diretiva **#asm** e a diretiva **#endasm**. Para o C18, essas diretivas são definidas com os símbolos **_asm** e **_endasm**.

Sintaxe

```
#asm                 // Ou, _asm para o C18
                     // código
#endasm              // Ou, _endasm para o C18
```

Exemplo

```
int teste_asm (int a, int b)
{
   #asm
   movf       a, w;         // copia o parâmetro a para w
   addwf      b, w;         // soma w com b e coloca em w
   movwf   _return_;     // armazena o resultado na variável de retorno
   #endasm
}
```

Diretiva #include

A diretiva **#include** diz ao compilador para incluir, no momento da compilação, o arquivo externo especificado.

Sintaxe

```
#include "nome_do_arquivo" ou
#include <nome_do_arquivo>
```

Exemplos

```
#include “C:\INCLUDES\COMLIB\MYRS232.C”
#include <16C54.H>
```

Obs.: A diferença entre se usar **" "** e **<>** é somente a ordem de procura do diretório do arquivo especificado. Se usado as aspas, o arquivo encontra-se no diretório corrente.

Diretivas #define e #undef

A diretiva **#define** informa ao compilador que substitua todas as ocorrências de um nome usado no programa, pela sequência de caracteres ou comandos fornecida. Já a diretiva **#undef** elimina a macro que a segue, apagando-a da tabela interna que guarda a macro.

Sintaxe

```
#define nome_da_macro sequencia_de_caracteres
#undef nome_da_macro
```

Exemplo

```
#define PI 3.1416   // Onde for encontrado o nome PI, será substituído pela
                    // sequencia de números 3.1416.
```

Diretiva #error

Essa diretiva força o compilador a gerar um código de erro (mensagem) no local onde se encontra sua designação.

Sintaxe

```
#erro mensagem // Mensagem define o texto informando o erro.
```

Exemplo

```
#error Buffer insuficiente
```

Diretivas #if, #elif, #else e #endif

A diretiva **#if** é um comando de pré-processador muito parecido com o comando **if**, pois ambos têm a função de conferir a validade de uma determinada expressão e realizar uma ação. Entretanto, a expressão contida em **#if** só é verificada durante a compilação, denotando que as expressões presentes nessas diretivas não suportam variáveis da linguagem C, somente constante, operadores-padrão e identificadores criados pela diretiva **#define**.

As diretivas **#elif** e **#else** são sempre opcionais e a diretiva **#endif** finaliza um bloco **#if**.

Sintaxe

```
#if expressão          // Expressão condicional.
   // código de programa
#elif expressão        // Expressão condicional
   // código de programa
#else
   // código de programa
#endif
```

Diretivas #ifdef e #ifndef

A diretiva **#ifdef** é uma forma abreviada da diretiva **#ifdefined (nome)**. Ela é utilizada para evitar a redefinição de operadores, constantes e identificadores, que podem, eventualmente, estar presentes nos arquivos inseridos no programa. Já a diretiva **#ifndef** é a forma negativa de **#ifdef**.

Sintaxe

```
#ifdef nome          // nome: operadores, constantes ou identificadores.
   // códigos de programa
#endif nome
```

Ou

```
#ifndef nome         // nome: operadores, constantes ou identificadores.
   // Códigos de programa
#endif nome
```

Exemplo

```
#include <stdio.h>           // Adiciona a biblioteca padrão de I/O

#define valor 590
#define valor_maximo 340

#if !defined(valor_minimo)
   #define valor_minimo 30
#endif

#ifndef valor
   #define valor 400
#endif

#ifdef valor_maximo
   #undef valor_maximo
   #define valor_maximo 500
#endif
void main(void)
{
   fprintf(std,"\nvalor=%u",valor);
   fprintf(std,"\nvalor_minimo=%u",valor_minimo);
   fprintf(std,"\nvalor_maximo=%u",valor_maximo);
}
```

Após execução do código acima exemplificado, tem-se a mensagem impressa como sendo:

```
valor = 590
valor_minimo = 30
valor_maximo = 500
```

Diretiva #pragma

Utilizaremos a diretiva **#pragma** com o compilador C18**.** Sua função será instruir o compilador a executar uma ação específica durante o processo de compilação. Existem várias ações que podem ser tratadas com a diretiva #pragma. As mais comuns são:

- code, que define uma seção como sendo de código.
- romdata, que define uma seção como sendo de dados armazenados na memória de programa.
- idata, que define uma seção como sendo de dados inicializados em uma memória de dados.
- udata, que define uma seção como sendo de dados não inicializados em uma memória de dados.
- interrupt, que define uma rotina de tratamento de interrupção como sendo de alta prioridade.
- interruptlow, que define uma rotina de tratamento de interrupção como sendo de baixa prioridade.
- config, que permite a configuração dos registradores de configuração do microcontrolador em uso.

Exemplificando este último caso, se desejar definir o oscilador como sendo a cristal e desabilitar o *power-up timer* e o *watchdog*, teremos o comando:

#pragma config OSC = XT, PWRT = OFF, WDTEN = OFF

Resumo

Nesta aula, lhe foram repassados os conceitos básicos relacionados à linguagem C para microcontroladores PIC, como identificadores, tipos de dados, modificadores de tipo, operadores e qualificadores. Você recebeu também informações sobre a estrutura básica de um programa escrito em C e quais as principais diretivas de compilação dos compiladores CCS e C18.

Autoavaliação

1. Quais as partes que compõem a estrutura básica de um programa em C?
2. Quais os tipos de dados básicos aceitáveis pelos compiladores CCS e C18?
3. Qual a funcionalidade dos modificadores de tipos de dados?
4. Por que o operador aritmético de módulo (%) só pode se utilizado em operações inteiras?
5. Explique e exemplifique o uso do qualificador de acesso const.
6. Qual a vantagem de se usar as diretivas de compilação em um programa em Linguagem C?
7. Dê outros exemplos de configuração de bits usando a diretiva #pragma config admitindo que o processador que está utilizando é o 18F45k20.

Referências

DEITEL, H. M.; DEITEL, P. J. **Como programar em C**. Rio de Janeiro: LTC, 1999.

MIYADAIRA, Alberto Noboru. **Microcontroladores PIC18**: aprenda e programe em linguagem C. São Paulo: Érica, 2009.

PEREIRA, Fábio. **Microcontroladores PIC**: técnicas Avançadas. São Paulo: Érica, 2002.

_____. **Microcontroladores PIC**: Programação em C. Fábio Pereira. São Paulo: Érica, 2005.

_____. **Microcontroladores PIC 18 Detalhado**: Hardware e Software. São Paulo: Érica, 2010.

SOUZA, David José de. **Desbravando o PIC**. São Paulo: Editora Érica, 2000.

SOUZA, David J. LAVINIA, Nicolas C. **Conectando o PIC**: Explorando recursos avançados. São Paulo: Érica, 2003.

SILVA, RENATO A. **Programando Microcontroladores PIC: Linguagem "C"**. São Paulo. Ensino Profissional. 2006.