C — Um Curso Fulminante de C

Versão 1.0 (05/04/2020)

> "*Eu e o Hélio Pellegrino temos um amigo que é o que se chama um erudito. E o pior é que se trata de um caso recente e diria mesmo de fulminante erudição. A princípio suspeitei de uma deslavada escroqueria intelectual. E aqui começa o mistério que desafia meu raciocínio e toda a minha intuição. Do dia para a noite o semianalfabeto aprendeu não sei quantos idiomas.*"
>
> Nelson Rodrigues em "O Óbvio Ululante"

As linguagens de alto nível, como o C, Pascal, Java e Python, vieram para liberar o programador da entediante tarefa que é programar em *assembly*. Essa afirmação traz uma grande verdade quando se trabalha com um sistema como o computador PC, onde existem teclado com mais de 100 teclas, monitor colorido, rede, *pen-drive* etc. Entretanto, quando se trabalha com um sistema microcontrolado que conta apenas com alguns *leds* e algumas chaves, e se precisa administrar diversas áreas de memória (geralmente pequenas), é preciso ponderar um pouco.

As linguagens de alto nível realmente facilitam a tarefa do programador, diminuem a quantidade de erros, propiciam uma documentação mais inteligível e facilitam os futuros trabalhos de manutenção. Por outro lado, há um preço a ser pago: programas maiores e mais lentos. Quando programamos em *assembly*, apesar das dificuldades, conseguimos programas menores e mais rápidos. Qual alternativa é melhor? A resposta (óbvia) é sempre programar em C, porém trechos críticos de seu programa podem ser escritos em *assembly*.

Neste apêndice apresentaremos uma introdução sobre o emprego da linguagem C para o MSP430. Não pretendemos ensinar a programar em C e nem abordar todos os detalhes de seu emprego em microcontroladores. Faremos apenas uma rápida apresentação das principais estruturas da linguagem e citaremos os pontos mais importantes. Por isso o texto está bastante denso. Entretanto, acreditamos que mesmo os leitores que não têm experiência em programação com C não terão dificuldades em ler este apêndice, entretanto a eficiência na programação só será conseguida com a prática. Boa parte do que é apresentado aqui foi inspirado no livro de ***The C Programming Language***, segunda edição, de Brian W. Kernighan e Dennis Ritchie. Recomendamos a leitura deste livro para

aqueles que quiserem se aprofundar neste assunto. Existem armadilhas em C e pode ser interessante a leitura do artigo "*C Traps and Pitfalls*" de Andrew Koenig.

Aqui se tenta estudar a linguagem C de forma genérica. É claro que se faz necessário mesclar com algumas particularidades do *Code Composer Studio* (CCS). Sempre que possível, este fato será assinalado. É claro que, por outro lado, não apresentamos todas as particularidades para o CCS. Assim, surpresas podem acontecer.

C.0. Quero Programar em C e Não Pretendo Ler Todo esse Apêndice

Em 1964, Ken Thompson participou do projeto de um sistema operacional denominado MULTICS (*Multiplexing Information and Computer Services*). No fim desses anos 60, por falta de resultados promissores, o projeto foi descontinuado e Ken Thompson voltou a trabalhar na AT&T. Junto com Dennis Ritchie, que também havia participado do MULTICS, escrevem um sistema operacional muito mais simples e, por brincadeira, o denominam de UNICS (*Uniplexing Information and Computer Services*) que viria a se transformar em UNIX. Depois, esse UNIX inspiraria a versão MINIX (Andrew Tanenbaum) que por sua vez seria a base do LINUX (Linus Torvalds).

Durante esse tempo, Ken Thompson programava usando a linguagem BCPL (*Basic Combined Programming Language*) numa versão que ele mesmo simplificou. Mais adiante, demandado pela necessidade de versar o UNIX para uma linguagem formal de alto nível, junto com Dennis Ritchie, formalizaram essa versão de BCPL com o nome C. Em 1978, Brian Kernighan e Dennis Ritchie publicam o livro "***The C Programming Language***" que, durante vários anos serviu como especificação informal da linguagem.

Se, ao atingir este ponto, o leitor ainda não consegue programar em C, então é fortemente recomendado a estudar com cuidado o restante deste apêndice.

C.1. O Básico da Programação em C

Todo programa em C composto por uma coleção de funções. Uma função pode ser entendida como uma sub-rotina, igual às que construímos quando programamos em *assembly*. As funções fornecem uma maneira conveniente de se encapsular trechos de programa. Não é preciso conhecer como está escrita uma função, basta saber o que ela faz. Normalmente, as funções recebem argumentos e retornam valores, se bem que existem funções que não recebem argumentos e tampouco retornam valores.

Os programas em C são construídos com declarações (*statements*). Toda declaração deve terminar com um "**;**" (ponto e vírgula). Usualmente as declarações realizam operações sobre variáveis. Todas as variáveis precisam ser declaradas. Em C se faz distinção entre letras maiúsculas e minúsculas. Assim, os identificadores `valor` e `Valor` são completamente distintos. Na distinção dos identificadores são considerados, pelo menos, 31 letras e dígitos. O programador pode empregar um identificador grande, entretanto,

somente os primeiros 31 caracteres serão considerados. É costume usar letras minúsculas para as variáveis comuns, sendo que as maiúsculas são reservadas para destacar as constantes ou as variáveis que são especiais para o programa.

As chaves "`{`" e "`}`" são empregadas para indicar um agrupamento de declarações dentro do programa. A não ser que se diga em contrário, as variáveis são dinâmicas, ou seja, elas só existem enquanto a função está em execução. Sendo mais preciso, as variáveis declaradas dentro de um agrupamento "`{ .... }`" só são reconhecidas dentro deste agrupamento. Apresentamos um pequeno programa ilustrativo:

```
/* Exemplo de um programa */
void main (void) {
            int i,qdr;
            i = 5;
            qdr = quadrado(i);
}

int quadrado (int x) {
            int y;              // declara variável y
            y = x*x;            // calcula o quadrado
            return y;           // retorna o quadrado
}
```

O programa inicia com uma função denominada `main`, que é o ponto de início do programa, e obrigatoriamente está presente em todo programa escrito em C. O corpo desta função é indicado pelo abre chave "`{`" e fecha chave "`}`". Logo no início são declaradas duas variáveis inteiras, a saber, `i` e `qdr`, que só são reconhecidas dentro do trecho marcado pelas chaves. No próximo item veremos os tipos de variáveis com que podemos trabalhar. Em seguida, à variável `i` é atribuída a constante `5` e, na próxima linha, é chamada a função que calcula o quadrado de um número, aqui denominada de `quadrado`.

A função `quadrado` está declarada logo abaixo. É de se notar que esta função recebe um argumento inteiro, indicado pelo campo `(int x)` e que também retorna um valor inteiro, indicado pela palavra `int` antes do nome da função. A função `quadrado`, cujo corpo está indicado pelas chaves, declara a variável inteira `y` para receber o produto de `x*x`. A função termina com um `return y`, retornando para o programa chamador o quadrado do argumento recebido.

Em C, o início de um campo de comentário é indicado por "`/*`" e seu final por "`*/`", como na primeira linha do programa. O comentário também pode ser indicado por "`//`", como foi feito na função `quadrado`. O programador escolhe como prefere indicar seus comentários.

Note que a função `void main(void)` não recebe parâmetros e tampouco retorna parâmetros, fato indicado pelo termo `void` precedendo a declaração da função `main` e pelo `(void)` logo após o nome da função. A maioria dos programas escritos para microcontroladores fica eternamente presente na memória, preso num laço infinito. Por isso, raramente usam uma função `main()` que recebe argumentos ou retorna algum valor qualquer. Diferente do que acontece com os computadores com sistemas operacionais que

chamam a função `main` passando alguns argumentos e esperam que no final da execução desta função algum valor seja retornado.

Agora que temos uma pequena ideia de como estruturar um programa em C, passemos para uma rápida descrição da linguagem, lembrando que estamos focados no compilador do CCS. Outros compiladores podem oferecer diferentes recursos.

C.2. Variáveis e Constantes

Quando se cria uma variável, na verdade, se está solicitando ao compilador que reserve uma certa quantidade de *bytes* em memória para que se possa armazenar dados (valores numéricos). Este processo é muito importante e toda linguagem o tem muito bem definido. Em C estão disponíveis quatro tipos dados: `char`, `int`, `float` e `double`, e junto com esses tipos empregam-se os qualificadores `short`, `long` e `unsigned`. Assim, é possível especificar uma variável `int`, `short int` e `long int`. Com o passar do tempo, algumas simplificações foram adotadas e, por exemplo, `long int` passou a ser usada apenas como `long`. Enfim, a tabela abaixo lista os tipos de dados aceitos pelo compilador C disponível no CCS.

Tabela C.1: Tipos de dados atendidos pelo Compilador C do CCS (slau132o, Tab 5-1, pag 89)

Tipo	*bits*	Repr.	Mínimo	Máximo
`char`	8	ASCII	0	255
`signed char`	8	C_2	-128	127
`unsigned char`	8	binário	0	255
`bool`	8	binário	0 (False)	1 (True)
`short`	16	C_2	-32 768	+32 767
`unsigned short`	16	binário	0	65 535
`int`	16	C_2	-32 768	+32 767
`unsigned int`	16	binário	0	65 535
`long`	32	C_2	-2 147 483 648	+2 147 483 647
`unsigned int`	32	binário	0	4 294 967 295
`long long`	64	C_2	9 223 372 036 854 775 808	9 223 372 036 854 775 807
`unsigned long long`	64	binário	0	18 446 744 073 709 551 615
`float`	32	IEEE 32	1,175 494 e -38	3,40 282 346 e+38
`double`	64	IEEE 64	2,22 507 385 e-308	1,79 769 313 e+308

`long double`	64	IEEE 64	2,22 507 385 e-308	1,79 769 313 e+308

bin = binário e C_2 = complemento a 2

É de se notar que para este compilador os tipos `int` e `short` são indistinguíveis. É possível aplicar os qualificadores `signed` ou `unsigned`, aos tipos `char`, `int`, `short`, `long` e `long long`. Note que o tipo `float`, `double` e `long double` sempre trabalham com sinal.
Na declaração de uma variável, se nada for dito, ela é considerada com sinal (exceção do `char`). Assim, são declarações válidas:

```
char dado;
unsigned int valor;
signed long limite = 3000;
float seno;
char cadeia[20];
int entrada [10];
```

Na lista acima, a terceira declaração já inicializa a variável `limite` com a constante inteira `3000`. A penúltima declaração cria um vetor de 20 *bytes* enquanto e que a última cria um vetor com 10 posições de 16 *bits*. A indexação dos elementos de um vetor inicia com o elemento 0. Considerando o vetor `cadeia`, seu primeiro elemento é indicado por `cadeia [0]` e seu último elemento por `cadeia [19]`.

As constantes numéricas são expressas em decimal, se nada for dito. A representação em hexadecimal deve ser precedida por 0x e em octal se precedida por 0 (zero). A representação em binário (só os fracos usam binário) deve ser precedida por 0b. Veja alguns exemplos na tabela abaixo.

Tabela C.2: Alguns exemplos de representação de números

Decimal	Hexadecimal	Octal	Binário
5	0x5	05	0b101
100	0x64	0144	0b01100100
232	0xE8	0350	0b11101000
1000	0x3E8	01750	0b001111101000
54321	0xD4E1	0152061	0b1101010000110001 (loucura!)

As constantes numéricas são empregadas intuitivamente, entretanto, é preciso tomar cuidado com a quantidade de *bits* necessários para sua representação. Por exemplo, a atribuição `char c = 1000` tenta atribuir uma constante de 16 *bits* a uma variável de 8 *bits*. No compilado do CCS o resultado é o truncamento e será atribuído o valor 0xE8, que corresponde ao decimal 232.

É boa prática de programação "dar nome" a alguns valores que são importantes. O termo correto é definir constantes. Por exemplo, com a LaunchPad, já nos acostumamos que o ACLK tem o valor de 32.768 Hz e que o SMCLK vale 1.048.576. Então, ao invés de escrevermos o valor numérico desses relógios, podemos definir um "nome" para eles, por exemplo, `ACLK_HZ` ou `SMCLK_HZ`. Isso é feito com o que se chama de constante simbólica. Ela criada com a declaração `#define`, como mostrado a seguir. O `#define` tem o mesmo papel do `.equ` que usamos em *assembly*. Note que essa declaração não tem o ";", pois não verdadeiramente um *statement*. Na lista abaixo, fazemos outras sugestões. Note que é costume empregar letras maiúsculas para tais declarações.

```
#define    ACLK_HZ      32768   //Frequência do ACLK em Hz
#define   SMCLK_HZ   1048576L   //Frequência do SMCLK em Hz
#define         CR       0x13   // ASCII do carriage return
#define         PI    3.14159   // valor de pi em float
#define    LETRA_A        ´A´   // código ASCII da letra A
```

Em C, foi previsto uma forma muito simples e útil de criar constantes com valores em sequência que é o recurso de enumeração (enum), cuja sintaxe á apresentada na próxima linha.

```
enum nome {cte_0, cte_1, ..., cte_n};
```

Por omissão (*default*), a enumeração inicia com zero (`cte_0` vale zero) e as demais seguem em sequência. É possível indicar o primeiro valor (nem todos os compiladores aceitam). Os exemplos abaixo deixam a ideia bem clara. Este recurso de enumeração foi usado no ER 7.10, para facilitar a atribuição de um número para cada nota.

```
enum boolean {NO, YES};
enum months {JAN=1,FEV,MAR,ABR,MAI,JUN,JUL,AGO,SET,OUT,NOV,DEZ};
```

Vamos abordar as Sequências de Escape, *ou Escape Sequences*, em inglês. Uma *Escape Sequence* é um conjunto de caracteres que não representa a si mesmo, mas sim, um outro caracter ou sequência de caracteres. Elas surgiram da dificuldade de se representar alguns caracteres da Tabela ASCII (Apêndice B), por exemplo, a instrução de "nova linha" ou "retorno do carro". Não tem uma "letra" ou símbolo para tais comandos, apenas um código. Em C todas as *Escape Sequences* iniciam com o *backslash* "\". A tabela abaixo apresenta uma lista das principais sequências.

Tabela C.3: Algumas Escape Sequences (as mais comuns)

Escape Sequence.	ASCII	Carcter Representado
\a	07	Alert (Beep)
\b	08	Backspace
\f	0C	Formfeed

\n	0A	Newline (Line Feed)
\r	0D	Carriage Return
\t	09	Horizontal Tab
\v	0B	Vertical Tab
\\	5C	Backslash
\'	27	Single quotation mark
\"	22	Double quotation mark

Com o CCS, quando usamos o `#include <MSP430.h>`, já carregamos a definição de uma grande quantidade de constantes, que vão facilitar nosso trabalho de programação do MSP430. Aí estão incluídos os endereços dos registradores de todos os periféricos, as posições de seus *bits*, além de algumas outras constantes auxiliares. A lista abaixo, apenas para efeito de entendimento, apresenta algumas dessas constantes.

```
#define       P1DIR   0x204   //Endereço do registrador P1DIR
#define       P2OUT   0x203   //Endereço do registrador P2OUT
#define      TA0CTL   0x340   //Endereço do registrador TA0CTL
#define     TA0CCR0   0x352   //Endereço do registrador TA0CCR0
#define       TAIFG      1   //Posição do bit TAIFG
#define    TASSEL_2   0x200   //Selecionar SMCLK no registrador TA0CTL
#define        BIT5    0x20   //Posição do bit 5
#define        BIT1       2   //Posição do bit 1
#define        BIT0       1   //Posição do bit 0
```

Precisamos agora comentar sobre cadeias de caracteres, que em inglês são chamadas de *strings*. Em C, toda *string* é indicada pelo seu endereço de início, sendo que seu final é marcado pelo *byte* 0 (zero, que para evitar confusões é também indicado por `'\0'`). Isto simplifica bastante, porque para referenciar uma *string*, usamos apenas seu endereço de início. Note que devido à presença deste zero final, o tamanho de uma *string* é aumentado de um. É possível declarar um vetor e inicializá-lo com uma *string*, como mostrado abaixo.

```
char vetor[20] = "ABCDEF";
```

Apesar da variável `vetor` ter 20 posições (o compilador reservou 20 *bytes*), empregamos apenas 6 primeiras, sendo que a sétima posição é igual a zero. As letras são representadas pelo código ASCII correspondente. Caso necessite, use a Tabela ASCII do Apêndice B. Assim, teremos:

`vetor[0] = 0x41` (ASCII de A)
`vetor[1] = 0x42` (ASCII de B)
`vetor[2] = 0x43` (ASCII de C)
`vetor[3] = 0x44` (ASCII de D)
`vetor[4] = 0x45` (ASCII de E)

`vetor[5] = 0x46` (ASCII de F) e
`vetor[6] = 0x00` ('\0', *byte* igual a zero).

Apresentamos abaixo o exemplo da string "**C fulminante**", que tem 12 letras e, como foi dito, usa 13 *bytes* pois o final é caracterizado com o zero que é adicionado. Na tabela ASCII o zero é o caracter NUL.

Sequência→	0	1	2	3	4	5	6	7	8	9	10	11	12
Letras	**C**		**f**	**u**	**l**	**m**	**i**	**n**	**a**	**n**	**t**	**e**	**0**
Hexa (ASCII)	43	20	66	75	6C	6D	69	6E	61	6E	74	65	00

Figura C.1. Organização na memória da string "C fulminante".

Abaixo está uma lista com alguns exemplos extras.

```
"Isto é uma string";     //String com 17 letras (18 bytes)
"x";                     //String com 2 bytes: 0x78 e 0x00
'x';                     //Não é uma string, seu valor é 0x78
"Linha 1\nLinha 2\n";    //String com 16 letras mais o zero final
```

A última *string* da lista acima é interessante e mostra a razão do uso das sequências de escape. A figura abaixo apresenta com é sua representada na memória e o resultado de sua impressão em um terminal ASCII. Note que cada `\n` representou a instrução de nova linha. Por isso, se mudou da linha 1 para a linha 2 e ao final da linha 2, surge uma linha em branco.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
L	**i**	**n**	**h**	**a**		**1**	**\n**	**L**	**l**	**n**	**H**	**a**		**2**	**\n**	**0**
4C	69	6E	68	61	20	31	0A	4C	69	6E	68	61	20	32	0A	00

```
Linha 1
Linha 2

```

Figura C.2. Exemplo da arrumação na memória de uma string com Escape Sequences e o resultado de sua impressão num terminal ASCII.

C.3. Operadores

A linguagem C faz uso de diversos operadores. Iniciamos com os operadores aritméticos, que são os já consagrados: `+`, `-`, `*` e `/`, listados na Tabela C.3. Existe ainda o operador módulo "`%`", ou resto da divisão, sendo que ele não é aplicável ao tipo `float`. Esses operadores podem ser empregados com qualquer tipo de variável, entretanto é preciso lembrar que o tipo impõe algumas restrições. Por exemplo, se forem empregadas variáveis

inteiras, temos: 7 / 3 = 2. Cuidado para não extrapolar a faixa de cada tipo, como mostrado a seguir.

```
char i = 100, j = 200;
i = i * j;              //20000 será truncado para um byte
```

Tabela C.3.a: Operadores Aritméticos

Operador	Operação realizada
+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo
++	Incremento
--	Decremento

Tabela C.3.b: Precedência entre operadores

++, --	Maior
- (unário)	
*, /, %	
+, -	Menor

Com a mesma precedência, avalia-se da esquerda para a direita.

O operador - (unário) é o que se usa para indicar que um número é negativo.

Quando o programador emprega esses operadores, o compilador se encarregar de ligar (com o uso do *linker*) ao programa as bibliotecas convenientes, ou seja, as bibliotecas de operações com inteiro ou com ponto flutuante. Em alguns casos é necessário adicionar a linha com o cabeçalho (*header*) para algumas funções como, por exemplo, as matemáticas `#include <math.h>`

Os operadores relacionais e lógicos servem para comparar expressões. Eles estão listados na Tabela C.4. Empregaremos esses operadores para comparar condições de controle do fluxo de execução de programas.

Tabela C.4. Operadores Relacionais e Lógicos

Operador	Operação realizada
>	Maior que
>=	Maior que ou igual a
<	Menor que
<=	Menor que ou igual a
==	Igual
!=	Diferente
&&	E
||	OU
!	NÃO

O trabalho com microcontroladores envolve a manipulação de *bits*, por isso, a próxima classe de operadores é muito importante. Veremos agora os operadores com *bits* (*bitwise*), apresentados na Tabela C.5. Eles não devem ser confundidos com os operadores relacionais ou lógicos.

Tabela C.5: Operadores de *bits*

Operador	Operação realizada
`&`	E
`|`	OU
`^`	OU-Exclusivo
`~`	NÃO ou NOT
`>>`	Deslocamento para direita
`<<`	Deslocamento para esquerda

Os operadores de deslocamento costumam gerar um pouco de confusão, por isso, a Figura C.3 deixa bem claro seu efeito. Note que, efetivamente, é um deslocamento lógico, já que não leva em conta o sinal do número. Descolar para a esquerda (`<<`) corresponde a multiplicar por 2 o número operado. Deslocar para a direita (`>>`) corresponde a dividir (divisão inteira) por 2 o número operado. É preciso tomar cuidado quando se emprega os operadores de descolamento com números negativos. Como o deslocamento é lógico, um número negativo pode ficar positivo e vice versa. É preciso pensar no número negativo (complemento a 2) na sua forma binária. De forma simplória, se costuma dizer que o resultado do deslocamento de números negativos é indefinido. Para terminar, deve-se evitar deslocamentos maiores que o tamanho da variável em *bits*.

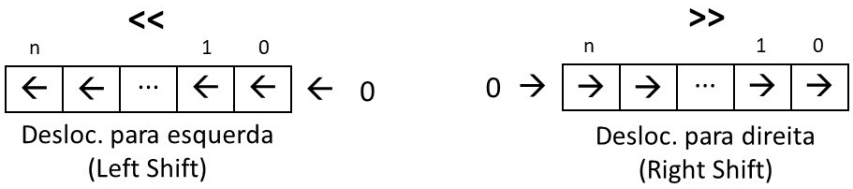

Figura C.3. Operadores de deslocamento.

Terminamos indicando os operadores de incremento `++` e decremento `--`. A operação de somar 1 ou subtrair 1 de uma variável é tão comum que provocou o surgimento de desses dois operadores. Assim, ao invés de empregarmos `x = x + 1` ou `y = y - 1`, escrevemos `x++` e `y--`, respectivamente. A atribuição `=` também tem recursos extras. Os exemplos a seguir dão uma ideia do que pode ser feito como os operadores.

```
int x, y, z;        //Declaração de três variáveis
z = x = y = 5;      //Variáveis x, y e z recebem o valor 5
z += 10;            //z = z + 10

x = (z++) + 20;     //x = z+20 e depois z = z+1 (note o pós-incremento)

x = (++z) + 20;     //z = z+1 e depois x = z+20 (note o pré-incremento)

z &= 0xff00;        //z = z & 0xff00, zera o byte menos significativo de z

y = z >> 8;         //Desloca 8 vezes para a direita o conteúdo de z
                    //e o atribui a y, corresponde a y = z/256
```

Esses operadores de *bits* são muito importantes para os programas que trabalham em baixo nível, como é nosso caso. Assim, a figura abaixo ilustra, em C, as operações mais comuns, tomando como exemplo a manipulação do quinto bit (BIT5) da variável `char z` (8 *bits*). A listagem a seguir apresenta alguns exemplos de operações que vamos necessitar na programação do MSP430.

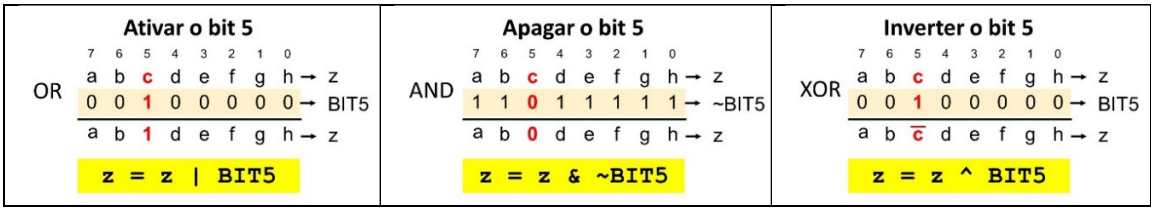

Figura C.4. Típicas operações para manipular *bits*. No caso desta figura, as operações foram ilustradas usando como exemplo a variável `char z` e seu quinto *bit*

A listagem a seguir apresenta uma série de exemplos do emprego das operações com *bits*. Lembre-se de que os *bits* são numerados da direita para a esquerda, começando com o 0, veja a numeração usada na Figura C.4. Assim, BIT0 é o primeiro *bit*, BIT1 é o segundo *bit*, e assim por diante.

```
char x,y,z;
int w;

z |= BIT3;      //Ativar o 4º bit de z (BIT3 é o quarto bit)
w &= ~BIT10;    //Apagar o 11º bit de w

y |= BIT6 | BIT2;     //Ativar o 3º e o 7º bits de y
y &= ~(BIT6 | BIT2); //Apagar o 3º e o 7º bits de y

x = 100;     //Atribui 100 à variável x
x = x >> 2; //Resulta em x = 25 (corresponde à divisão por 4)

x = 0x3F;     //Atribui o hexadecimal 3F à variável x
y = x >> 4;  //Resulta em y = 3 (separamos o nibble mais significativo)
```

```
z = x & 0xF; //Resulta me z = 0xF (separamos o nibble menos significativo)

w = 0xDF23;   //Atribui o hexadecimal DF23 à variável w
y = x >> 8;   //Resulta me y = 0xDF (separamos o byte mais significativo)
z = x & 0xFF; //Resulta me z = 0x23 (separamos o byte menos significativo)
```

Um caso muito comum de erros é, por exemplo, quando se quer alterar um determinado trecho de um registrador, sem alterar o resto. Vamos tomar como exemplo o registrador de controle do *timer* TA0, que tem o nome TA0CTL e está apresentado abaixo. Para este exemplo, não importa se o leitor já estudou, ou não o *timer*. O que é abordado é apenas a manipulação dos *bits*.

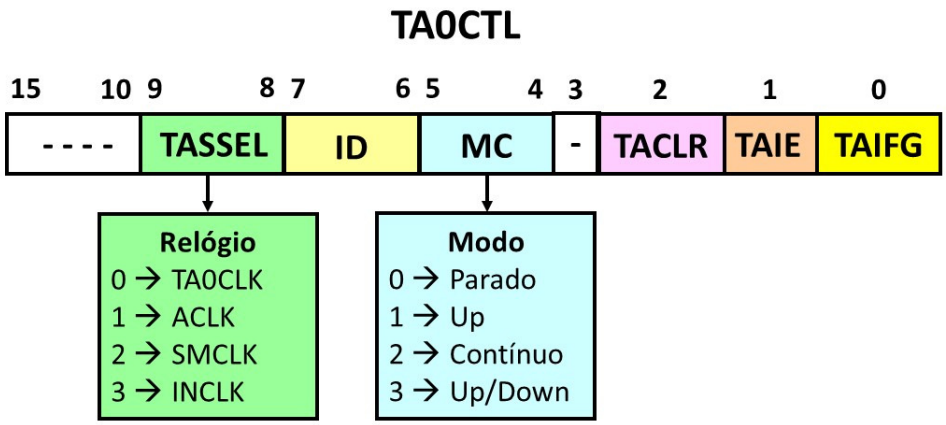

Figura C.5. Registrador TA0CTL que permite a configuração do timer TA0. Estão detalhados apenas os bits usados no exemplo a seguir.

Digamos que, usando o registrador TA0CTL (Figura C.5), o leitor programou o *timer* TA0 para operar no modo 1 (MC_1) e com o ACLK (TASSEL_1). Depois ele quer mudar para o modo 2 (MC_2). A listagem a seguir ilustra um equívoco muito comum.

```
TA0CTL = TASSEL_1 | MC_1;     //Selecionado modo 1 e o ACLK
...
// Agora o programador quer mudar para operar no modo 2
// Porém ele não quer alterar o restante do registrador

// Solução errada, pois resulta na seleção do modo 3 (MC_3)
TA0CTL |= MC_2;          //Selecionado modo 3 e não o 2 <== ERRO

// Solução correta
TA0CTL &= ~MC_3;        //Garante que os dois bits estejam em zero
TA0CTL |= MC_2;         //Programa modo 2

// Esta é também uma Solução correta
TA0CTL = TASSEL_1 | MC_2;     //Selecionado modo 2 e o ACLK
```

Vamos explicar com cuidado. Quando usamos o compilador C do CCS, já temos previamente definidas (`MSP430.h`) todas as constantes para operar os *bits* dos registradores. Para o caso do exemplo acima e para que a explicação fique clara, apresentamos como seriam feitas as definições das constantes pertinentes.

```
#define TA0CTL       0x340
#define TASSEL_0     0            //TA0CLK selecionado
#define TASSEL_1     BIT8         //ACLK   selecionado
#define TASSEL_2     BIT9         //SMCLK  selecionado
#define TASSEL_3     (BIT9|BIT8)  //INCLK  selecionado
#define MC_0         0            //Modo 0 = Parado
#define MC_1         BIT4         //Modo 1 = Ascendente (up)
#define MC_2         BIT5         //Modo 2 = Contínuo (continuous)
#define MC_3         (BIT5|BIT4)  //Modo 3 = Sobe/Desce (up/down)
```

A linha `TA0CTL = TASSEL_1 | MC_1` atribui ao registrador o valor 0x110 (*bits* de números 8 e 4 colocados em 1). Como foi uma atribuição (veja o sinal de igualdade), todo o registrador foi sobrescrito. Depois, ao tentar mudar para o modo 2, o programador fez simplesmente a operação equivocada `TA0CTL |= MC_2`. Essa operação apenas coloca em 1 o bit de número 5 do registrador. Porém, o bit de número 4 já estava em 1, assim o resultado é que os bits de números quatro e o cinco vão para 1, e essa é a especificação para o modo 3. O correto, neste caso é antes zerar o campo que se pretende alterar para só depois ativar o bit correto.

Para finalizarmos este tópico, vamos ver o que em C se chama Expressão Condicional. A sintaxe pode parecer complicada, mas o entendimento é muito simples.

`expr1 ? expr2 : expr3;`
A `expr1` é avaliada primeiro e
Se for `TRUE`, é avaliada a `expr2`
Ser for `FALSE`, é avaliada a `expr3`

```
int a, b, z;      //Três variáveis inteiras
      ...
z = (a>b) ? a : b;  //Se a é maior que b, então z = a
                    //Se a não é maior que b, então z = b
      ...
z= (a>b) ? a-b : b-a; //Se a é maior que b, então z = a-b
                      //Se a não é maior que b, então z = b-a
                      //Calculou o módulo da a-b
```

C.4. Controle do Fluxo de Programa

Veremos agora os recursos para construirmos os laços de programas e tomarmos decisões de sobre a execução de cada um deles. Cada decisão é feita com o teste de condições que podem ser verdadeiras ou falsas. Em C, representa-se o Falso por 0 (zero) e em consequência o verdadeiro por qualquer valor diferente de zero. Os recursos aqui tratados

são clássicos e serão abordados de forma resumida. Nos exemplos, para simplificar, faremos uso de variáveis supondo que elas já foram convenientemente declaradas.

É importante deixar claro o conceito de *statement* (declaração). Já dissemos que a linguagem C é composta por *statements*. Cada *statement* é finalizado com um ponto e vírgula "`;`". As chaves "`{`" e "`}`" são usadas para agrupar *statements*. O conjunto `{ ... }` tem, semanticamente, a validade de um *statement*. Veja a seguir uma lista com diferentes versões de *statements*.

```
int x, y, z;    //um statement
x = 5;          //um statement
y=6;  z=7;      //dois statements em uma linha
;               //um statement (vazio)

{  x = x +y;    //agrupamento
  z = z - x;    //de statements
  x= 30;        //sintaticamente equivale
}               //a um statement

{}              //um statement (vazio)
```

C.4.1. If - else

Esse recurso é empregado para expressar decisões. Sua sintaxe é mostrada a seguir, sendo que é permitido não fazer uso do `else`.

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

No exemplo a seguir, se o conteúdo de `x` for maior que `5`, faremos `z` igual a `10`, senão faremos `z` igual a `20`. Note que o escopo do `if` e do `else` foi marcado com chaves.

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

Quando o escopo possui apenas um único *statement*, as chaves podem ser omitidas, como é feito a seguir. Isso facilita a leitura do programa.

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

O próximo exemplo não faz uso do `else`.

```
if (x <= 10) {
   x++;
   z = 10;
}
```

C.4.2. Else - if

Essa estrutura é tão comum que é tratada em separado. Ela é uma forma de se tomar uma decisão com diversas hipóteses. As expressões são avaliadas em sequência e se uma delas for válida, a declaração a ela associada é executada e a cadeia é finalizada. O último `else` serve como "nenhum dos casos acima".

```
if (expressão)
   statement
else if (expressão)
   statement
else if (expressão)
   statement
else if (expressão)
   statement
else
   statement
```

C.4.3. Switch

Esse controle serve para uma decisão múltipla que testa se uma expressão casa com um certo número de constantes inteiras e, caso positivo, desvia para o trecho correspondente.

```
switch (expressão){
   case constante1:     statement
   case constante2:     statement
   case constante3:     statement
            ...
   default:             statement
}
```

Como exemplo apresentamos uma função que retorna o código ASCII correspondente à *nibble* recebida. Esta função não está escrita da maneira mais eficiente, a intenção foi apenas exemplificar o emprego do controle `switch`. Note o uso do `break` ao final de cada declaração, pois, caso contrário, o processamento seguiria deste ponto em diante, ou seja, o `switch` faz, na verdade, um desvio (salto) para um determinado trecho, por isso é importante que cada trecho finalize com um `break`, do contrário, os trechos adiante serão executados em sequência.

```
char nib_asc (char nib) {
```

```
   char resp;           \\ variável para receber a resposta
         nib &= 0xf;    \\ (nib=nib&0xf) garantir n0 na faixa de 0 a 15
         switch (nib) {
            case 0:  resp = '0';    break;
            case 1:  resp = '1';    break;
            case 2:  resp = '2';    break;
            case 3:  resp = '3';    break;
            case 4:  resp = '4';    break;
            case 5:  resp = '5';    break;
            case 6:  resp = '6';    break;
            case 7:  resp = '7';    break;
            case 8:  resp = '8';    break;
            case 9:  resp = '9';    break;
            case 10: resp = 'A';    break;
            case 11: resp = 'B';    break;
            case 12: resp = 'C';    break;
            case 13: resp = 'D';    break;
            case 14: resp = 'E';    break;
            default: resp = 'F';    break;
         }
       return resp;
}
```

No programa recém apresentado, o leitor deve ter notado que o código ASCII de uma letra é obtido com aspas simples. Apresentamos a seguir uma maneira mais racional de resolver o mesmo problema. Note que criamos uma tabela com os 16 símbolos (ASCII) hexadecimais. Na declaração desta tabela não foi necessário indicar seu tamanho, pois ela já é inicializada com uma *string*. Tal declaração foi feita externamente à função `nib_asc`, o que a torna uma variável global. Explicaremos este tópico de variáveis globais mais adiante.

```
char tabela[ ] = "0123456789ABCDEF";
char nib_asc (char nib) {
   nib &= 0xf;                \\Garantir número na faixa de 0 a 15
   return tabela[nib];
}
```

C.4.4. While

Este controle serve para repetir trechos de programa enquanto uma determinada condição for válida. Sua sintaxe é extremamente simples.

```
while (expressão)
  statement
```

Para exemplificar seu emprego, vamos supor que usamos o conversor ADC12 do MSP430 para ler um sensor analógico que, como todo sensor, sofre interferência de ruído. Como há ruído, não podemos confiar numa única leitura. Faremos então diversas leituras e calcularemos a média. Apresentamos abaixo a função denominada `int media_adc()`, que

não recebe argumentos, mas retorna a média de 16 leituras do ADC12. Note que a divisão por 16 foi feita com um deslocamento (>>) de 4 *bits* para a direita. O leitor não precisa se preocupar com as linhas que comandam o ADC12, o importante é entender a estrutura da função. Como o ADC12 entrega resultados de 12 bits, a soma de 16 desses valores numa variável inteira não ultrapassa seu limite de representação.

```
int media_adc(void) {                       //Considerar ADC12 inicializado
   unsigned int i=0, soma=0;
   while (i < 16){
      ADC12CTL0 |= ADC12ST;                 //Disparar conversão
      while ( (ADC12IFG&ADC12IFG0) == 0); //Aguardar conversão terminar
      soma += ADC12MEM0;
      i++;
   }
   return soma>>4;
}
```

Desafio: Ao invés de fazermos a divisão por 16, preferimos usar 4 deslocamentos para a direita o que, usualmente, é mais rápido. O desafio é explicar por que a linha abaixo melhora o resultado da divisão por 16?

```
return (soma+8)>>4;
```

Algumas vezes é preciso colocar o programa num laço infinito, isso pode ser conseguido com `while`, se a expressão de controle de repetição for sempre verdadeira. É importante comentar que com o emprego do `break` é possível sair de um laço infinito. O programa a seguir apresenta uma pequena ideia de como isso pode ser feito.

```
while (1) {       //Laço infinito
      ...
   if (condição de saída)     break;
      ...
}
```

C.4.5. For

O `for` é outro controle para repetição de determinados trechos de programa. Sua sintaxe é simples, como apresentado a seguir. O termo `expr1` faz a inicialização necessária, `expr2` é a condição de saída e `expr3` é a alteração sobre a variável de controle da repetição do laço.

```
for (expr1; expr2; expr3)
   statement
```

O `for` e o `while` são perfeitamente equivalentes, como mostrado a seguir. O programador deve escolher um deles de forma a beneficiar a clareza de seu programa.

```
expr1;
while (expr2) {
```

```
   statement
   expr3;
}
```

Reescrevemos a função `int media_adc()`, mas agora empregando o `for`. O leitor deve comparar esta função com sua equivalente que emprega o `while`.

```
int media_adc (void) {
   unsigned int i, soma=0;
   for (i = 0;  i < 16; i++){
      ADC12CTL0 |= ADC12ST;                //Disparar conversão
      while ( (ADC12IFG&ADC12IFG0) == 0); //Aguardar conversão terminar
      soma += ADC12MEM0;
   }
   return soma>>4;
}
```

É possível construir um laço infinito com o uso de `for(;;)`.

C.4.6. Do - while

O controle de repetição `while`, testa a condição de repetição logo no topo do laço, ou seja, é a primeira coisa a fazer para entrar no laço. Se essa condição for falsa, o laço não é executado. Com o `do – while`, o teste da condição de repetição do laço é feito no final do laço, ou seja, o laço é repetido pelo menos uma vez. Sua sintaxe é apresentada a seguir.

```
do
statement
while (expressão);
```

C.4.7. Break e Continue

Algumas vezes é interessante ter condições de sair de um laço de repetição sem a necessidade de executar o teste que é feito no início ou no final. A declaração `break` fornece uma maneira de se sair antecipadamente de laços tipo `for`, `while`, `do` e `switch`. O `break` faz a quebra do laço mais interno. Veja que num exemplo acima, quebramos um laço infinito com o emprego do `break`.

A declaração `continue` antecipa a repetição do laço de programa. No caso de `while` e `do`, faz com que o teste da repetição seja executado imediatamente. No caso do `for`, o controle do laço é antecipado, passando para o a fase de incremento da variável de controle da repetição.

C.4.8. Goto e Labels

Agora vamos provocar a turma de programação estruturada. Formalmente, o `goto` nunca seria necessário, e na maioria das vezes o programa fica melhor e mais inteligível quando ele não é empregado. Entretanto, existem algumas situações particulares onde o `goto` pode "quebrar o galho". Por exemplo, em caso de emergência, ele pode ser empregado para abandonar os laços `do`, `while` e `for`, e saltar imediatamente para um trecho específico do programa. O próximo exemplo é inspirado no livro de Kernighan e Ritchie e faz uso do `label` "`erro_brabo`".

```
while ( ... )
    for ( ... )
        for (...)
            if (deu_galho_serio)
                goto erro_brabo;
}
...
erro_brabo:
    comece_a_correr (agora)
```

C.5. Conceitos sobre Funções

Como já afirmamos, as funções permitem quebrar a tarefa a ser realizada em pequenos trechos de programa, de forma a facilitar o trabalho de programação e, é claro, o entendimento. Além disso, com a divisão em funções é possível que diversas pessoas trabalhem no mesmo programa, ou ainda, que se aproveite o que outros programadores fizeram no passado. Até o momento, empregamos funções de forma intuitiva, apresentamos agora sua sintaxe.

```
tipo-retornado nome_da_função (tipos de argumentos){
   declaração de variáveis
   statements
}
```

Um programa é uma coleção de declarações e de funções, sendo que a comunicação é feita através dos argumentos passados para as funções, dos valores retornados e das variáveis externas (veremos em breve o que é isso). Usualmente, toda função termina com `return`. Ao programa chamador é permitido ignorar o valor retornado. Empregamos `void` para indicar quando a função não recebe argumentos ou quando não retorna valores.

Quando o compilador encontra uma função ele precisa saber quais os argumentos a serem passados e o valor a ser retornado. O exemplo de programa que apresentamos no item C.1 traz um perigo, pois ao encontrar pela primeira vez `qdr = quadrado (i)`, como nada foi dito, o compilador supõe (e neste caso acerta) que o valor retornado seja inteiro. Para evitar isso, uma boa prática é declarar as funções logo no início do programa, como mostrado a seguir.

```
/* Exemplo de um programa */
int quadrado (int arg); //Declaração da função quadrado
void main (void) {
   int i, qdr;
   i = 5;
   qdr = quadrado(i);
}

int quadrado (int x) {
   int y;               //Declara a variável y
   y = x * x;           //Calcula o quadrado
   return y;            //Retorna o quadrado de x
}
```

No programa apresentado, as variáveis `i`, `qdr` e `y` são denominadas internas, isso porque elas foram declaradas dentro do corpo de uma função. Cada uma dessas variáveis só é "enxergada" dentro da função que a criou. Ao terminar a execução da função, as variáveis internas são descartadas. Existem, entretanto, variáveis que são externas, e que por isso mesmo são "enxergadas" a partir de qualquer ponto do programa. Tais variáveis externas ou globais podem ser empregadas para passar argumentos e receber valores. Veja o programa que calcula o quadrado, mas agora reescrito com variáveis externas.

```
/* Exemplo de um programa com variáveis externas*/
void quadrado (void);   //Declaração da função quadrado
int i, qdr;             //Variáveis externas ou globais
void main (void) {
   i = 5;
   quadrado( );
}

void quadrado (void) {  //Função para calcular o quadrado de i
   qdr = i * i;         //Calcular o quadrado e atribuir a qdr
}
```

O leitor pode se sentir tentado a empregar somente variáveis externas, mas deve resistir. As variáveis internas ou locais foram criadas para facilitar a tarefa do programador, que não precisa se lembrar de todas as variáveis que criou e o que fazia com cada uma delas. Por isso, use apenas as variáveis globais que necessitar. As variáveis externas são muito úteis quando se trabalha com interrupções, já que não é possível passar argumentos ou receber valores das funções serviço de interrupção. Outro tipo de variável externa é a função (o nome da função). Toda função é "enxergada" de qualquer ponto do programa.

O momento é propício para abordarmos as variáveis estáticas, cuja declaração é feita com a palavra `static`. Por exemplo `static int w`, cria a variável estática `w`. Toda variável estática tem existência permanente, mesmo que declarada dentro de uma função. Em outras palavras, ela não é descartada quando termina a execução da função. Seu valor é mantido e a variável (que mantém seu valor) pode ser empregada na próxima vez que a função for chamada. Elas se parecem bastante com as variáveis externas, pois nunca são descartadas, entretanto, só são "enxergadas" dentro do escopo em que foram declaradas,

ou seja, dentro do campo delimitado por "{ ... }". Ao serem declaradas, as variáveis externas e estáticas são inicializadas com zero, as demais variáveis não são inicializadas.

Para garantir uma boa organização dos programas e evitar duplicidade de declarações de variáveis e funções, é comum o emprego de arquivos cabeçalho, do inglês, *header files*. Esses arquivos cabeçalho sempre terminam com ".h". O CCS (Version: 9.3.0.00012), na instalação no Windows 64 bits, colocou os arquivos `msp430.h` e `msp430f5529.h` no diretório. Isso pode variar um pouco, de acordo com os ambientes e versões.

```
C:\ti\ccs930\ccs\ccs_base\msp430\include\msp430f5529.h
```

É interessante o usuário examinar o arquivo `msp430f5529.h`. Abaixo estão dois outros diretórios com arquivos .h. É uma grande quantidade de arquivos e não é nosso objetivo identificar para que serve cada um deles.

```
C:\ti\ccs930\ccs\tools\compiler\ti-cgt-msp430_18.12.4.LTS
C:\ti\ccs930\ccs\tools\compiler\ti-cgt-msp430_18.12.5.LTS
```

De forma bem simples, o programador inclui esses arquivos no seu programa, de acordo com os recursos de que vai necessitar. Se for trabalhar com cadeias de caracteres, inclui o `string.h`, se for trabalhar com ponto flutuante, inclui o `float.h`, se precisar de funções matemáticas (seno, log etc) inclui o `math.h`. As declarações dos recursos internos do MSP430F5529 estão todas reunidas no arquivo `msp430f5529.h`. A inclusão de arquivos cabeçalho que acompanham o compilador é feita com os símbolos "<>". Os cabeçalhos criados pelo programador são incluídos com o emprego de aspas, como mostrado a seguir.

```
#include <string.h>
#include <math.h>
#include <float.h>
#include "meu cabeçalho.h"
```

C.5.1. Funções Intrínsecas

Uma função intrínseca é uma função especial, criada junto com a confecção do compilador com a finalidade de oferecer uma solução específica. O compilador a processa de uma forma especial pois, diferentemente das demais funções, o ele tem um conhecimento íntimo dessas funções intrínsecas. O arquivo `intrinsics.h` contém um protótipo de todas essas funções. É interessante dar uma olhada na página 136 do manual slau132o, pois ali está a lista completa das funções intrínsecas do compilador C do CCS. Para dar uma melhor ideia do que fazem_ as funções intrínsecas, citamos alguns exemplos:

Função	Assembly gerado
`__bic_SR_register(mask);`	`BIC #mask,SR`
`__data16_read_addr(addr);`	`MOV.W #addr,Rx` `MOVA 0(Rx),dst`

`__data16_write_addr (addr, src);`	`MOV.W #addr, Rx` `MOVA #src, 0(Rx)`
`__delay_cycles(qtd);`	`Atraso de qtd ciclos (ver detalhes)`
`__disable_interrupt(void);`	`DINT`
`__enable_interrupt(void);`	`EINT`
`__low_power_mode_1(void);`	`BIS.W #0x58, SR`
`__no_operation(void);`	`NOP`
`__op_code(opc);`	`Insere opcode opc no programa`
`__swap_bytes(src);`	`MOV src, dst` `SWPB dst`

C.6. Ponteiros e Vetores

O emprego de ponteiros gera programas compactos e eficientes. Entretanto, quando empregados de forma descuidada podem provocar erros estranhos e gerar códigos de difícil entendimento. Assim, é importante ter bem claro os conceitos sobre ponteiros e seu emprego. Ponteiros e vetores estão fortemente relacionados, como veremos a seguir.

C.6.1. Ponteiros

Ponteiro é uma variável cujo conteúdo é um endereço. Sua declaração é apresentada a seguir, junto com um pequeno trecho de programa, onde surgem dois operadores unários: o "`*`" que é usado para declarar e operar com ponteiros e o "`&`" que retorna o endereço de uma variável. Apresentamos um pequeno trecho de programa para ilustrar o uso de ponteiros.

```
int x=10, y=20, vet[10];  //Duas variáveis e um vetor de inteiros
int *pt;                  //Declaração um ponteiro para inteiros
pt = &x;                  //Ponteiro pt recebe endereço de x
*pt = 30;                 //Equivale a x = 30
y =  *pt;                 //Equivale a y = x
pt = &vet[0];             //Ponteiro recebe endereço da primeira
                          // posição do vetor de inteiros
*pt = 50;                 //Equivale a vet[0] = 50
*(pt+1) = 100;            //Equivale a vet[1] = 100
```

Na primeira linha foram declarados duas variáveis inteiras e um vetor de 10 posições inteiras. Logo em seguida, foi declarado um ponteiro para inteiros (`int *pt`). Veja que o "`*`" é empregado para indicar que a variável é um ponteiro. Em seguida, o ponteiro foi inicializado com o endereço da variável `x` (`pt = &x`), note o uso do operador "`&`". A partir

deste ponto, passam a existir duas formas de acessar esta variável `x`: através de seu nome `x` e através do ponteiro `pt`. Ao fazermos `*pt = 30`, alteramos o valor de `x` para 30 e com `y = *pt` transferimos o conteúdo de `x` para a variável `y`. Com a linha `pt = &vet[0]` posicionamos o ponteiro para apontar para o primeiro elemento de vetor `vet`. Com o emprego do ponteiro, inicializamos a primeira e a segunda posições do vetor com 50 e 100, respectivamente.

É importante deixar claro que ponteiro está relacionado com o tipo de variável a qual ele aponta, ou seja, se aponta para `char`, `int`, `float` etc. No exemplo recém apresentado, vimos que a posição apontada pelo ponteiro (`*pt`) pode ser empregada como se fosse a própria variável. Como ponteiro é uma variável, o conteúdo de um ponteiro pode ser operado e copiado para outro ponteiro. Supondo que `pt` aponte para a variável `x`, apresentamos o exemplo:

```
pt = &x;          //Ponteiro pt aponta para a variável x
*pt = *pt+1;      //x=x+1
y = *pt+1;        //y=x+1
*pt += 1;         //x=x+1
++*pt;            //x=x+1
(*pt)++;          //x=x+1
```

C.6.2. Vetores

Os conceitos de ponteiro e vetor estão tão ligados que tudo que pode ser feito com vetor, pode ser feito com ponteiros e vice-versa. Quando declaramos o vetor, `int vet[10]`, o compilador cria o ponteiro `vet` e o inicializa de forma a apontar para uma região onde "cabem" 10 valores inteiros. Assim o nome de um vetor é, na verdade, um ponteiro para seu início. A única restrição é que o valor deste ponteiro não pode ser alterado. Quando fazemos `pt = &vet[0]`, ou `pt = vet`, copiamos para o ponteiro o endereço de início do vetor. Apresentamos alguns exemplos:

```
char vet[10];     //Declara vetor de 10 posições char (bytes)
char x=1,y=2,*pt; //Declara x e y  e o ponteiro pt
pt = vet;         //Ponteiro aponta para o início do vetor
*pt = 1;          //Equivale a vet[0] = 1
pt[1] = 3;        //Equivale a vet[1] = 3
*(pt+2) = 5;      //Equivale a vet[2] = 5
*(vet+3) = 7;     //Equivale a vet[3] = 7
pt = &vet[8];     //Ponteiro aponta para posição 8 do vetor
*pt = 9;          //Equivale a vet[8] = 9
++*pt;            //Equivale a vet[8] = vet[8] + 1
pt++;             //Agora ponteiro aponta para a posição 9
*pt = 11;         //Equivale a vet[9]=11
*(pt+1) = 13;     //Erro, pois não existe a posição vet[10]
```

Com o exemplo mostrado acima, empregamos diversos tipos de operações com ponteiros. Note que o ponteiro pôde ser incrementado. Esse ponto é importante. Ao incrementarmos um ponteiro, ele avança uma posição, segundo seu tipo. Como no exemplo o ponteiro

apontava para `char` (*bytes*), ele simplesmente avançou uma posição. Se fosse um ponteiro que apontava para `int` (2 *bytes*) ele avançaria duas posições de memória. O mesmo é válido para o decremento de ponteiros.

Um erro muito comum é levar o ponteiro para além do final do vetor com o qual se está trabalhando. Este erro não é detectado pelo compilador e é muito difícil de ser descoberto. Caso ocorra, a posição de memória logo após o final do programa é alterada (não sabemos o que havia lá). A operação `vet++` gera erro no momento da compilação, pois o ponteiro `vet` precisa marcar o início do vetor e não pode ser alterado. Finalizamos lembramos que uma `string` é na verdade um vetor de caracteres ASCII, cujo final é marcado com o *byte* zero (`'\0'`).

De acordo com o que vimos, as funções não têm como alterar o valor das variáveis do programa chamador. Entretanto, isto pode ser conseguido com o uso de ponteiros. Reescrevemos o programa que calcula o quadrado, mas agora a função `quadrado` altera o valor da variável `qdr`, pois nos seus argumentos recebeu o endereço desta variável (`&qdr`).

```
/* Exemplo de programa que passa endereço de variável */
void quadrado (int *pt, int x); //Declara a função quadrado
void main (void) {
   int i,qdr;
   i=5;
   quadrado(&qdr,i);           //Chama a função quadrado, passando
                               // o endereço de qdr e o valor de i
}

int quadrado (int *pt, int x){
   *pt = x*x;                 //Calcula o quadrado e
}                             //Atualiza qdr usando ponteiro pt
```

É importante indicar que vetor é passado para a função através de seu endereço. Este ponto merece cuidado, pois se a função recebeu o endereço do vetor, ela pode alterar o conteúdo deste vetor. Para ilustrar a passagem de endereço de um vetor, apresentamos uma função que calcula a soma dos elementos do vetor. Note que também é preciso passar o tamanho do vetor, pois a função não tem como sabê-lo. Deve causar surpresa a linha `soma += *pt++`, que primeiro acessa a posição apontada e depois incrementa o ponteiro. Esta linha também poderia ser escrita assim: `soma = soma + *(pt++)`.

```
/* Exemplo de função que trabalha com vetor */
int somat (char *pt, int tam);      //Declara a função
void main (void) {
   char vet[10];
   int i,s;
   for (i=0, i<10; i++) vet[i]=1;   //vet = [0,1,2,3, ...,7,8,9]
   s=somat(vet,10);                 //Chama a função somat
}

int somat (char *pt, int x){        //Função recebe o ponteiro
   int j,soma=0;
```

```
   for (j=0; j<10; j++) soma+=*pt++;
   return (soma);
}
```

A operação com incremento de ponteiros e o acesso para a posição apontada pode ser bastante confusa. Por isso, apresentamos a lista abaixo que ilustra os principais tipos de ocorrências.

```
int *pt;     //Declara o ponteiro para posição inteira

*++pt;       //Incrementa o ponteiro e depois acessa a posição apontada
*--pt;       //Decrementa o ponteiro e depois acessa a posição apontada
*pt++;       //Acessa a posição apontada e depois incrementa o ponteiro
*pt--;       //Acessa a posição apontada e depois decrementa o ponteiro
++(*pt);    //Incrementa a posição apontada e depois a acessa
(*pt)++;    //Acessa a posição apontada e depois a incrementa

y=a/*pt;    //Será que o CCS vai interpretar /* como início de comentário
```

C.6.3. Vetores Multidimensionais

Já aprendemos a operar vetores. Agora surge a pergunta sobre matrizes. Matrizes são bidimensionais e, em C, são representadas por vetores arrumados em sequência na memória. A notação de matriz em C é bastante intuitiva e está mostrada nos exemplos a seguir.

```
char dados [10][20];     //Matriz com 10 linhas, cada um com 20 colunas
char dados [] [];        //Erro: Compilador não tem como saber o tamanho

int mat1 [3][5] = {{4,7,9,2,1}, {1,3,5,3,6}, {9,2,1,2,7}}; //Matriz 3x5
int mat1 [][]   = {{4,7,9,2,1}, {1,3,5,3,6}, {9,2,1,2,7}}; //Matriz 3x5
```

Na lista acima, a matriz `int mat1 [3][5]` foi declarada de duas formas diferentes. Note que na segunda pudemos omitir os índices que indicam o tamanho, pois especificamos a inicialização e com isso o compilador tem como descobrir a quantidade de memória necessária. Para deixar bem clara a ideia de matriz, vamos indicar na figura abaixo sua representação na memória.

Linha 1					Linha 2					Linha 3				
0	1	2	3	4	0	1	2	3	4	0	1	2	3	4
4	7	9	2	1	1	3	5	3	6	9	2	1	2	7
n	n+2	n+4	n+6	n+8	n+10	n+12	n+14	n+16	n+18	n+20	n+22	n+24	n+26	n+28

Figura C.6. Representação de `int mat1 [3][5]` *do exemplo acima. A última linha da tabela representa os endereços de seu posicionamento na memória. Como é uma matriz de inteiros, cada posição ocupa 2 bytes.*

O exemplo acima, deixa bem claro que matriz é um vetor cujos elementos também são vetores. A exigência é que cada vetor que compõe a matriz tem o mesmo tipo e tamanho. A forma mais simples de lembrar desta representação é decorar que "*o índice mais à direita é o que conta mais rápido*".

Nada impede que este conceito seja estendido. Por exemplo, podemos criar um vetor de matrizes, ou ainda uma matriz de matrizes, ou beirando o exagero um vetor com uma matriz de matrizes e seguir assim ad nauseam. Alguns exemplos abaixo.

```
char dt [5][3][4];         //Vetor com 5 matrizes 3 x 4
char mamat [7][5][3][4];   //Matriz 5x7, cada elemento é matriz 3x4 (?)
char nauseam [3][5][7][6][8][15];   //É hora de pensar numa forma mais
                                    //inteligente de arrumar esses dados
```

Deixando os exageros de lado, vamos indicar como se passa uma matriz para uma função, ou como se cria um ponteiro para uma matriz. Novamente, o que se expõe aqui pode ser estendido para qualquer quantidade de dimensões. Vamos tomar como exemplo a declaração `int mat1 [3][5]` feita um pouco mais atrás e imaginar que precisamos chamar a função `int func ()` que faz uma operação qualquer (não importa qual) sobre essa matriz e retorna um valor inteiro.

```
int mat1 [3][5] = {{4,7,9,2,1}, {1,3,5,3,6}, {9,2,1,2,7}}; //Matriz 3x5
int mat3d [2][3][4] = { {4,7,9,2}, {1,5,3,1}, {2,9,5,3},   //Matriz
                        {5,6,2,1}, {2,4,3,9}, {6,3,8,9}};  //3 dimensões
void main(void){
      int x;
      x = func(mat1);
      ...
}

---- conferir abaixo ----

// 1) Indicando as dimensões
int func (int matriz[3][5]){ ... }

// 2) Indicando apenas a segunda dimensão (omite a primeira dimensão)
// Cabe ao programador da função saber que são 3 vetores de tamanho 5
int func (int matriz[][5]){ ... }

// 3) Indicando apenas a segunda dimensão
// Passamos ponteiro para vetores de tamanho 5
// Cabe ao programador da função saber que são 3 vetores de tamanho 5
int func (int (*matriz)[5]){ ... }

// 4) Indicando todas as dimensões
int func (int matriz3d[2][4][3]){ ... }

// 5) O3itindo a primeira dimensão
// Cabe ao programador da função saber a primeira dimensão
```

```
int func (int matriz3d[][4][3]){ ... }
```

No exemplo de número 5, foi necessário o uso de parêntesis. O operador "`[...]`" tem precedência sobre o "`*`" e se os parêntesis não fossem colocados, ficaria `int func (int *matriz[5])`. Isto indica para a função que ela está recebendo um vetor com 5 ponteiros para inteiro com o nome `matriz`. Sim, você pode ter estranhado. Podemos sim criar um vetor de ponteiros. Mas, uma matriz não era exatamente isso? Bem, quase! Vejamos as duas declarações abaixo.

- `int a [10] [20];` → aloca espaço na memória para 200 números inteiros e os acessa sob a forma de 20 x linha + coluna.
- `int *b[10];` → aloca espaço para 10 ponteiros e é responsabilidade do programa usá-los da forma correta.

No exemplo acima, podemos fazer o ponteiro `b` apontar para vetores de tamanho 20, que serão suas linhas. Veja que o programador é o responsável por criar esses vetores e inicializar o vetor de ponteiros `b`. Uma vantagem desta forma de uso é que podemos operar vetores de tamanhos diferentes. Isso nos dá versatilidade para economizar memória. Veja caso abaixo onde se precisar criar um vetor com nomes de pessoas.

```
// Matriz de 48 bytes (4 x 12)
char vnomes [][12]={"Ze","Pedro","Paulo","Epaminondas"};

// Vetor de ponteiros, os nomes que ocupam 3 + 6 + 6 + 12 = 27 bytes
char *pnomes[]= {"Ze","Pedro","Paulo","Epaminondas"};
```

No primeiro caso, colocamos os nomes nas diferentes linhas da matriz e, por ser matriz, cada linha precisa ter o mesmo tamanho. O maior nome, `Epaminondas` precisa de 12 *bytes* (11 letras mais o `'\0'` final), assim ele ditou a quantidade de colunas. Há um desperdício de espaço. No caso do nome `Ze`, 9 *bytes* nunca são usados.

No segundo caso, há um melhor aproveitamento, pois criamos um vetor de ponteiros. Durante a compilação, o compilador carrega os diversos nomes na memória (não importa onde) e coloca o endereço de início de cada nome no vetor de ponteiros.

Vamos estudar mais um assunto sobre ponteiros. As funções, na verdade, são ponteiros para um endereço na memória, que é o endereço de início da função. A única diferença é que esses ponteiros (nome da função) não podem ser alterados. Assim, é possível criar ponteiros para funções e acessá-las através desses ponteiros. Estudemos os dois exemplos abaixo. Note o uso de parêntesis. Como já foi dito, eles são importantes. Vejamos a primeira declaração que sem os parêntesis fica com a seguinte forma: `int *pfunc (int, char)`. Isto significa função que recebe um inteiro, um `char` e retorna o ponteiro (o endereço) para um inteiro.

```
//Ponteiro de função que recebe um inteiro, um char e retorna um inteiro
int (*pfunc) (int, char);

//Ponteiro de função que recebe endereço de um inteiro e retorna inteiro
int (*pfunc) (int *);
```

O uso de ponteiro de função é análogo ao uso de ponteiro de dados. Vejamos a listagem abaixo. Na primeira linha é criado um ponteiro para uma função que recebe dois inteiros e que retornar um valor inteiro. Nas duas próximas linhas criamos duas funções muito simples: uma para somar e outra para multiplicar. No programa principal, usando o mesmo ponteiro, conseguimos executar as duas funções, uma de cada vez. Os nomes em negrito ajudam a compreensão.

```
int (*pfunc) (int, int);  //Ponteiro função recebe dois int e retorna int

int soma (int a, int b) {return a+b;} //uma função simples
int mult (int a, int b) {return a*b;} //outra função simples

void main (void){
      int x=5, y=6, z;  //Criar variáveis
      pfunc = soma;     //Apontar para a função soma
      z=pfunc(x,y);     //(Z=11) Executar função com o uso do ponteiro
      pfunc = mult;     //Apontar para a função multiplicação
      z=pfunc(x,y);     //(Z=30) Executar função com o uso do ponteiro
}
```

Um último exemplo. É possível também passar para a função o ponteiro para uma outra função que se queira executar. Aproveitando ainda o exemplo anterior, mas colocando alguma modificação.

```
int soma (int a, int b) {return a+b;} //uma função simples
int mult (int a, int b) {return a*b;} //outra função simples

// Função que recebe dois inteiros mais o ponteiro para uma função e
// retorna um valor inteiro
int opera (int x, int y, int (*func)(int a, int b){
      return func(x,y);
}

void main (void){
      z=opera(3,4,soma); //(Z=7) Executar função com o uso do ponteiro
      z=opera(3,4,mult); //(Z=12) Executar função com o uso do ponteiro
}
```

Vamos finalizar com alguns exemplos interessantes, para não dizer engraçados:

- `char **ppt;` → ponteiro para um ponteiro do tipo char.
- `int (*vet)[9];` → ponteiro para um vetor de 9 posições inteiras.
- `int *vet[9];` → vetor com 9 ponteiros para inteiros.
- `void *func();` → função que retorna ponteiro `void` (vazio).
- `void (*func)();` →ponteiro para função que retorna ponteiro `void` (vazio).

Os dois próximos são intencionalmente assustadores (conferir)!

- `char (*(*func( ))[ ]( );` → `func` é função que retorna ponteiro para um vetor de ponteiros para funções que retornam `char`.

- `char (*(*vet[ 3 ])( ))[5];` → `vet` é um vetor com 3 ponteiros para funções que retornam ponteiro para um vetor com 5 `char`.

C.7. Estruturas

Estrutura é uma coleção de uma ou mais variáveis, provavelmente de tipos diferentes, agrupadas sob um único nome para facilitar sua manipulação. As estruturas servem para organizar e auxiliar as operações com dados complexos. Um bom exemplo é conseguido se imaginarmos um microcontrolador embarcado num pequeno aeromodelo. Neste caso, três informações importantes são: altitude, velocidade e direção. Empregar variável do tipo `float` para a altitude e para a velocidade, sendo que a direção será dada pelo azimute expresso em graus e, portanto, uma variável do tipo `int`. Vamos criar uma forma de agrupar essas três variáveis com o nome estado.

```
struct tres_valores{     //Indica como é composta a estrutura
   float alt;            //Altitude (membro)
   float vel;            //Velocidade (membro)
   int az;               //Azimute (membro)
};
      ...
struct tres_valores estado;   //Cria a estrutura estado
struct tres_valores *pte;     //Ponteiro p/ estrutura
```

O nome `tres_valores` serve apenas para indicar a composição da estrutura, ou seja, qual o tipo de cada um de seus membros. A penúltima linha cria realmente a estrutura estado. A última linha cria um ponteiro para estruturas do tipo `tres_valores`. A referência aos membros da estrutura é feita da seguinte forma:

```
nome_da_estrutura.membro
```

Qualquer tipo de operação é permitido com os membros de uma estrutura. A seguir apresentamos alguns exemplos com a estrutura `estado`.

```
estado.alt = 132.3;      //Altitude
estado.vel = 45 + z;     //Velocidade, z é uma variável qualquer
estado.az = 360/4;       //Azimute
```

Vamos agora aprender a trabalhar com o ponteiro `pte` que criamos na última linha do exemplo apresentado. A primeira coisa a fazer é inicializar o ponteiro, algo do tipo `pte = &estado`. Depois desta atribuição, o ponteiro `pte` passa a apontar para a estrutura `estado`. Nesse caso `*pte` é a estrutura e seus membros são: `(*pte).alt`, `(*pte).vel`, `(*pte).az`. É tão comum o emprego de ponteiro para estruturas que se criou uma alternativa mais fácil de se escrever: `ponteiro -> membro`.

```
struct tres_valores{     //Indica como é composta a estrutura
   float alt;            //Altitude (membro)
   float vel;            //Velocidade (membro)
```

```
   int az;              //Azimute (membro)
};
      ...
struct tres_valores estado;   //Cria a estrutura estado
struct tres_valores *pte;     //Ponteiro pte p/ estrutura
      ...
pte = &estado;          //Ponteiro aponta para estrutura
pte->alt = 132.3;       //Altitude
pte->vel = 45 + z;      //Velocidade, z é uma variável qualquer
pte->az = 360/4;        //Azimute
```

Quando uma estrutura é argumento passado para uma função, ela é passada por valor, ou seja, todos os elementos que compõem a estrutura são copiados. Isso pode gerar uma grande massa de dados. Esta é mais uma razão para o uso de ponteiro para estruturas. É possível também criar vetores de estruturas.

```
struct tres_valores{    //Indica como é composta a estrutura
   float alt;           //Altitude (membro)
   float vel;           //Velocidade (membro)
   int az;              //Azimute (membro)
};
      ...
struct tres_valores estado[100]; //Agora é um vetor com 100 estruturas
struct tres_valores *pte;        //Ponteiro pte p/ estrutura
      ...
pte = &estado[0];  //ponteiro para a primeira estrutura do vetor
++pte->az;         //Incrementa o membro az equivale a ++(p->az)
(++pte)->az;       //Incrementa ponteiro pte antes de acessar az
(pte++)->az;       //Acessa az e depois incrementa pte
pte++->az;         //Idem caso acima, acessa az e depois incrementa pte
```

C.8. Uniões, sizeof() e alocação Dinâmica de memória

Em C, `union` (união) é um tipo especial de variável que permite armazenar objetos de diferentes tipos e tamanhos na mesma posição de memória, mas não todos ao mesmo tempo. Fornece uma forma eficiente de usar a mesma posição de memória para diferentes finalidades. Veja, no exemplo abaixo, que a notação lembra a de estruturas. Cada novo valor destrói o anterior.

```
char c;
// uni é uma união que pode receber qualquer um dos tipos listados
// (inteiro, float ou ponteiro para char)
union u_exemp {int ival; float fval; char *sval;} uni;

uni.ival = 5;
uni.fval = 3.14;
uni.sval= &c;
```

O operador `sizeof()` é muito usado em C e retorna um número inteiro sem sinal correspondente à quantidade de *bytes* usado pelo operando.

Se o operando é um tipo de variável, retorna a quantidade de *bytes* alocada para cada tipo (vide Tabela C.1):
`sizeof (char)` → retorna 1 (1 *byte*).
`sizeof (int)` → retorna 2 (2 *bytes*).
`sizeof (long)` → retorna 4 (4 *bytes*).
`sizeof (float)` → retorna 4 (4 *bytes*).

Se o operando for uma expressão, retorna a quantidade de *bytes* alocada para representar o resultado da expressão (vide Tabela C.1). Considerando que:
`int i=1; float f=3.14;`
`int vet[10];`
`sizeof (i+f)` → retorna 4 (4 *bytes*), porque o resultado da expressão será em `float`.
`sizeof (vet)` → retorna 20 (20 *bytes*), usados para as 10 posições inteiras do vetor.

O `sizeof()` pode ser usado para descobrir o tamanho de um vetor. Veja o exemplo

```
int vet[10];
int tam;
tam = sizeof(vet)/sizeof(vet[0]);
```

Até agora, conhecíamos à priori o tamanho de todos os vetores que usamos. Porém, isso nem sempre é possível. Existem aplicações onde se necessita de vetores de diversos tamanhos, sendo que o tamanho só é descoberto durante a execução do programa. Por exemplo, de acordo com uma escolha do usuário do programa, precisamos adquirir 10, 100 ou 1.000 amostras de um sinal. A solução bruta seria criar um vetor de tamanho 1.000. Entretanto, pode ser que esse tamanho nunca seja usado e essa reserva de 1.000 posições representa um grande desperdício de memória. Para resolver situações como esta, e muitas outras que não são citadas aqui, existe a possibilidade de alocação dinâmica de memória. Em outras palavras, a possibilidade de criar vetores durante a execução de um programa. Essa função se chama `malloc()` e faz muito uso do operador `sizeof()` (ver seção 6.1.6, pag 125 de slau1320). Veja alguns exemplos na listagem abaixo. Para liberar espaço, existe a função `free()`. Acho que esta função só faz sentido quando se trabalha em ambiente cuja memória é gerenciada, por exemplo, quando se programa baixo um Sistema Operacional. Não a encontrei (`free()`) no CCS.

```
Conferir os exemplos abaixo
//Exemplo com inteiro
int *vet;
vet = malloc(10*sizeof(int)); //Equivale a int vet[10];
;
//Exemplo com estrutura
struct tres_valores{    //Indica como é composta a estrutura
   float alt;           //Altitude (membro)
   float vel;           //Velocidade (membro)
   int az;              //Azimute (membro)
};
struct tres_valores *pt_str;  //Ponteiro para estrutura
```

```
pt_str = malloc(20*sizeof(tres_valores)); //Vetor para 20 estruturas
;
//Exemplo com união
union u_exemp {int ival; float fval; char *sval;} *uni;
uni = malloc(10*sizeof(u_exemp); //Vetor com 10 unions
```

C.9. Programando com Vários Arquivos

Quando escreve programas pequenos, o leitor deve usar um único arquivo, onde hospeda todo seu código. Porém, para programas grandes e para o caso em que diversos programadores trabalham no mesmo programa, cada um desenvolvendo um certo segmento, é útil a organização em diferentes arquivos.

Apresentamos a seguir a solução do ER 7.8 "quebrado" em diferentes arquivos. Para facilitar a compreensão, esses arquivos são intencionalmente pequenos. Repetimos o pedido do exercício em questão:

ER 7.8 (repetido). Com o *Timer* B0 acionado pelo SMCLK (1.048.576 Hz), configure um PWM com período de 20 ms (50 Hz) para controlar o brilho dos dois *leds* (*led*1 = vermelho e *led*2 = verde), excursionando de forma complementar, de 0% até 100% em passos de 10%. As chaves S1 e S2 serão usadas neste controle, operando da forma listada abaixo.

- S1 → +10% para vermelho e -10% para verde
- S2 → -10% para vermelho e +10% para verde

Serão usadas as instâncias 0 e 1 dos comparadores do *Timer* B0 (TB0.1 e TB0.2), para acionar os *leds* de acordo com a conexão na figura abaixo. Reveja o trecho do Capítulo 3 (item 3.5) que trata do mapeamento da Porta P4.

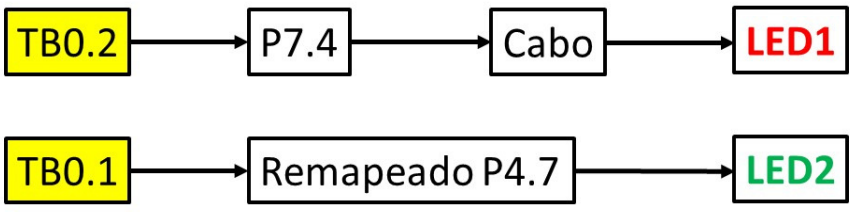

Figura C.7. Esquema usado para que o Timer B0 acione os dois leds.

Solução: Vamos operar com o TB0 no Modo 1, assim em TB0CCR0 configuramos o período do PWM.

Período de 20 ms: 0,020 x 1.048.576 = 20.971,52 → TB0CCR0 = 20.971 contagens.

O valor 20.971,5 deveria ser arredondado para 20.972, mas precisamos subtrair 1 porque o zero também é contado. Com esse valor, o erro no período de 20 ms é de 5 ηs, o que é bastante aceitável.

Para facilitar o controle da potência, vamos usar duas variáveis, `pot1` e `pot2`, que contando de 0 até 100, vão guardar a potência atual de cada *led*. De acordo com o acionamento das chaves somamos ou subtraímos 10 de cada variável. Há que se tomar cuidado para essas variáveis não irem acima de 100% ou abaixo de 0%. O monitoramento das chaves não é objetivo deste capítulo. O leitor deve consultar o Capítulo 5. A conta para calcular o valor dos comparadores é mostrada a seguir.

```
TB0CCR1 = (POT100 * (long)pot2) / 100;  //Programar PWM
TB0CCR2 = (POT100 * (long)pot1) / 100;  //Programar PWM
```

A solução deste problema é muito simples e não deveria ser quebrada em diferentes arquivos. Entretanto, isso foi feito para ilustrar esta forma de se programar. Logo a seguir é apresentada a listagem da solução deste problema, que foi quebrada em diferentes arquivos, cada um com uma finalidade:

- `main.c` → Programa principal
- `defines.h` → Definições de todas as constantes usadas
- `globais.h` → Especificação de todas as variáveis globais
- `configs.h` → Cabeçalho das funções de configuração
- `configs.c` → Funções de configuração
- `chaves.h` → Cabeçalho das funções que monitoram as chaves
- `configs.c` → Funções que monitoram as chaves

A forma de se criar arquivos fonte (.C ou .asm) e arquivos cabeçalho com o CCS está ilustrado nas Figuras C.13, C.14, C.15 e C.16.

A Figura C.8 apresenta um instantâneo do CCS com este programa “aberto”. É possível ver as diversas abas, uma para cada arquivo. É importante ressaltar que os arquivos não precisam estar abertos no editor para que sejam usados na compilação do programa. Na verdade, todos os arquivos presentes na pasta do projeto serão compilados em separado e depois unidos (*linked*) em um único arquivo executável.

Merece uma atenção especial as variáveis globais `ps1` e `ps2`. Elas foram declaradas no arquivo globais.c, mas são usadas pela rotina do arquivo chaves.c. Quando o compilador trabalha com o arquivo chaves.c, ele não tem como saber que variáveis são essas. Por isso, elas foram marcadas com `extern` no arquivo chaves.h. Variáveis marcadas como `extern` são resolvidas pelo *linker*, que faz a ligação final de todos os códigos compilados.

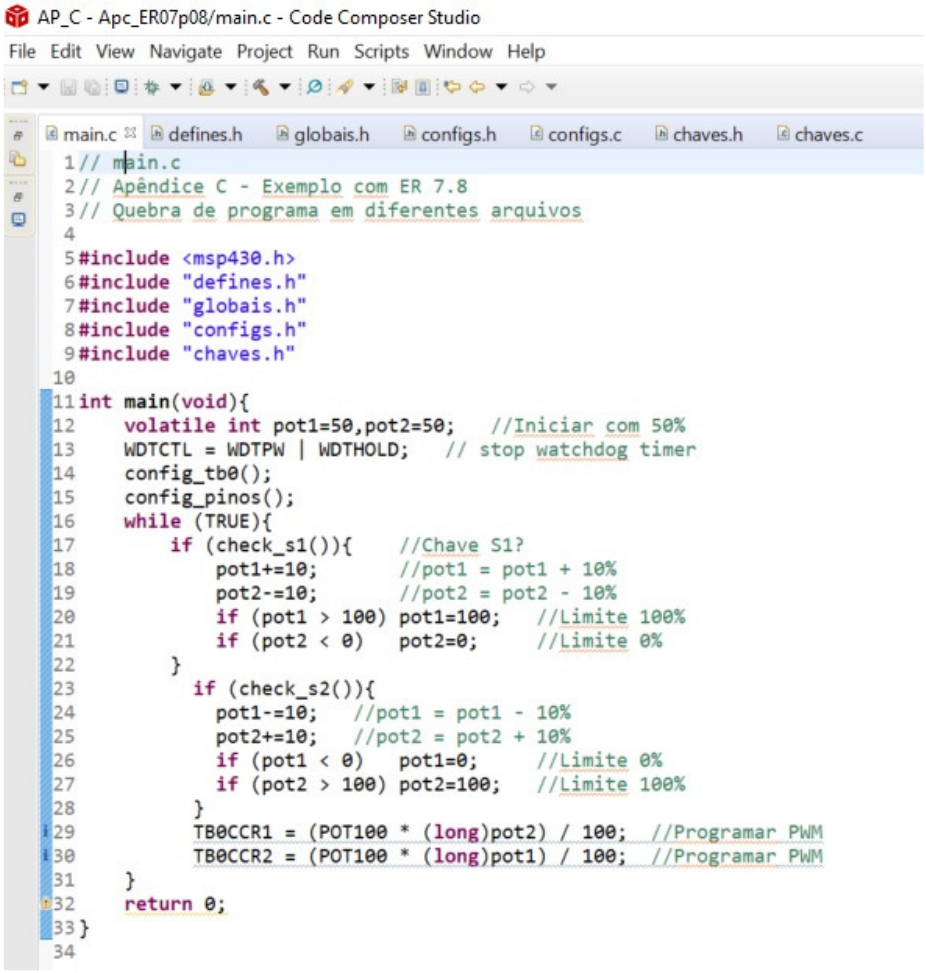

Figura C.8. Imagem do CCS com o programa solução. Notar que temos diversas abas, uma para cada arquivo.

Listagem **`main.c`**

```
// main.c
// Apêndice C - Exemplo com ER 7.8
// Quebra de programa em diferentes arquivos

#include <msp430.h>
#include "defines.h"
#include "globais.h"
#include "configs.h"
#include "chaves.h"

int main(void){
    volatile int pot1=50,pot2=50;    //Iniciar com 50%
      WDTCTL = WDTPW | WDTHOLD;      // stop watchdog timer
      config_tb0();
      config_pinos();
```

```
    while (TRUE){
        if (check_s1()){    //Chave S1?
            pot1+=10;       //pot1 = pot1 + 10%
            pot2-=10;       //pot2 = pot2 - 10%
            if (pot1 > 100) pot1=100;   //Limite 100%
            if (pot2 < 0)   pot2=0;     //Limite 0%
        }
          if (check_s2()){
            pot1-=10;   //pot1 = pot1 - 10%
            pot2+=10;   //pot2 = pot2 + 10%
            if (pot1 < 0)   pot1=0;     //Limite 0%
            if (pot2 > 100) pot2=100;   //Limite 100%
          }
          TB0CCR1 = (POT100 * (long)pot2) / 100;  //Programar PWM
          TB0CCR2 = (POT100 * (long)pot1) / 100;  //Programar PWM
    }
    return 0;
}
```

Listagem **defines.h**

```
// Defines.h
// Constantes usadas pelo programa

#ifndef DEFINES_H_
#define DEFINES_H_

#define TRUE    1
#define FALSE   0

#define ABT 1       //Constante representa Aberta
#define FEC 0       //Constante representa Fechada
#define DBC 1000    //Atraso para o debounce

#define POT100  20971 //0,02 * 1.048.576 --> 10 ms

#endif /* DEFINES_H_ */
```

Listagem **globais.h**

```
// globais.h
// Variáveis globais
// Protótipo de funções

#ifndef GLOBAIS_H_
#define GLOBAIS_H_

int ps1=ABT,ps2=ABT;        //Estado anterior das chaves

#endif /* GLOBAIS_H_ */
```

Listagem **configs.h**

```
// configs.h
// Configuração dos periféricos

#ifndef CONFIGS_H_
#define CONFIGS_H_

void config_tb0(void);
void config_pinos(void);

#endif /* CONFIGS_H_ */
```

Listagem **configs.c**

```
// Configs.c
// Configuração dos periféricos

#include <msp430.h>
#include "configs.h"
#include "defines.h"

// Configurar TB0
void config_tb0(void){
    TB0CTL = TBSSEL_2 | MC_1;   //TB0 com SMCLK e Modo Up
    TB0CCR0 = POT100;           //Período PWM
    TB0CCTL1 = OUTMOD_6;        //Saída Modo 6
    TB0CCTL2 = OUTMOD_6;        //Saída Modo 6
    TB0CCR1 = POT100/2;         //Iniciar com 50%
    TB0CCR2 = POT100/2;         //Iniciar com 50%
}

// Configurar chaves, leds e remapear
void config_pinos(void){
    P2DIR &= ~BIT1;     //S1 = entrada com pull-up
    P2REN |= BIT1;
    P2OUT |= BIT1;

    P1DIR &= ~BIT1;     //S2 = entrada com pull-up
    P1REN |= BIT1;
    P1OUT |= BIT1;

    P7DIR |= BIT4;      //TB0.2 = P7.4
    P7SEL |= BIT4;      //Ligar cabo para Led Vermelho

    P4DIR |= BIT7;
    P4SEL |= BIT7;
    PMAPKEYID = 0X02D52;    //Escrever chave
    P4MAP7 = PM_TB0CCR1A;   //TB0.1 mapeado para P4.7
}
```

Listagem **chaves.h**

```
// chaves.h
// Monitorar as chaves

#ifndef CHAVES_H_
#define CHAVES_H_

int check_s1(void);
int check_s2(void);
void debounce(void);

// Variáveis globais definidas em outro arquivo
// Serão resolvidas durante a linkagem
extern int ps1;  //Estado anterior da chave S1
extern int ps2;  //Estado anterior da chave S2

#endif /* CHAVES_H_ */
```

Listagem **chaves.c**

```
// Chaves.c
// Monitorar as chaves

#include <msp430.h>
#include "defines.h"
#include "chaves.h"

// Verificar S1
// Retorna: TRUE = A->F, FALSE = demais casos
int check_s1(void){
    if ((P2IN&BIT1) == 0){  //S1 fechada
        if (ps1==ABT){      //A --> F
            debounce();
            ps1=FEC;        //Guardar o passado
            return TRUE;
        }
        else
            return FALSE;   //F --> F
    }
    else{                   //S1 aberta
        if (ps1==ABT)
            return FALSE;   //A --> A
        else{
            debounce();
            ps1=ABT;        //Guardar o passado
            return FALSE;    //F --> A
        }
    }
}
```

```
// Verificar S2
// Retorna: TRUE = A->F, FALSE = demais casos
int check_s2(void){
    if ((P1IN&BIT1) == 0){  //S1 fechada
        if (ps2==ABT){      //A --> F
            debounce();
            ps2=FEC;        //Guardar o passado
            return TRUE;
        }
        else
            return FALSE;   //F --> F
    }
    else{                   //S1 aberta
        if (ps2==ABT)
            return FALSE;   //A --> A
        else{
            debounce();
            ps2=ABT;        //Guardar o passado
            return FALSE;    //F --> A
        }
    }
}

void debounce(void){
    volatile unsigned int i;
    for (i=DBC; i>0; i--);
}
```

C.10. Programando em C com o CCS

O CCS integra um ambiente de trabalho bastante sofisticado que inclui editor de texto, montador *assembly*, ligador (*linker*) e depurador (*debuger*). Além disso tudo, ele pode trabalhar com uma enorme variedade de microcontroladores. Por ser um ambiente tão sofisticado, algumas facilidades ficam “escondidas”.

Neste tópico iremos ver alguns recursos do CCS que muito podem ajudar ao programador que está num estágio um pouco mais avançado.

C.10.1. Gerando o Código Assembly de um Programa em C

Para o leitor que já passou pela fase de *assembly* e que finalmente está feliz programando em C, deve se perguntar: por que, carambolas, eu voltaria a trabalhar com o *assembly*? Pode parecer estranho, mas examinar o *assembly* fruto de uma compilação pode ser muito útil. Permite uma melhor compreensão de como é feita a compilação e, em muitos casos, se pode tirar proveito disso. Possíveis resultados inesperados num programa em C podem

ser compreendidos quando se examina a listagem *assembly*. Além disso, nenhum compilador está livre de *bugs*, e muitas vezes esses *bugs* só são descobertos com a análise do *assembly* correspondente. Em nosso caso particular, nos permite entender as otimizações realizadas pelo CCS (que muitas vezes remove trechos de nosso código).

Depois dessas explicações, vamos ver como habilitar o Code Composer a gerar a listagem *assembly* de um programa em C. Em primeiro lugar, é preciso configurar o compilador do CCS para gerar essa listagem do *assembly*. Na versão do CCS usada neste momento (Version: 9.3.0.00012) para Windows 64 bits, essa configuração segue o caminho mostrado abaixo. As demais versões devem ter um caminho parecido. De acordo com a Figura C.9, vamos marcar a caixa "**Keep the generated assembly language**". Assim, no diretório Debug de seu projeto irá surgir um arquivo com o nome de seu programa e com a extensão .asm. Note que esta escolha e individual para cada projeto.

```
Project -> Properties -> Build -> MSP430 Compiler -> Advanced Options ->
Assembler Options
```

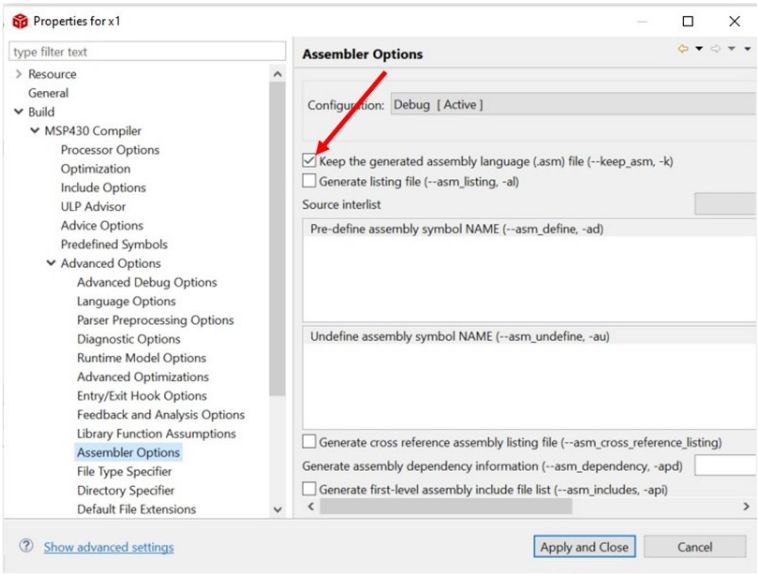

Figura C.9. Janela para instruir ao compilador para manter o arquivo assembly.

O Debug do CCS faz uso de uma depuração simbólica (*symbolic debugging*). Isto significa que ele conhece suas variáveis, sabe onde inicia e termina cada uma de suas funções, pode fazer o passo-a-passo de seu programa (entrando ou não nas suas funções etc.). Para tanto, o compilador precisa gerar uma grande quantidade de marcações para que o *debugger* se oriente. No CCS é usado o **DWARF Debugging Format** (slau131m, pag 300) e é incluída no arquivo *assembly* com o uso das diretivas listadas abaixo.

```
.dwtag,          .dwendtag,      .dwattr,      .dwpsn,     .dwcie,
.dwendentry,     .dwfde,         .dwendentry,  .dwcfi.
```

Para nós, essa marcação não tem grande utilidade. Vamos então orientar o compilador a não incluí-la no arquivo *assembly*. Conseguimos isso, desabilitando a opção de *debug* no projeto. Isto é feito com o caminho listado abaixo e selecionando a opção "**Supress all symbolic debug generation**", como mostrado na Figura C.10. Para habilitar novamente o *debug*, usamos a opção "**Full all symbolic debug**", como mostrado na Figura C.11.

```
Project -> Properties -> Build -> MSP430 Compiler -> Advanced Options ->
Advanced Debug Options
```

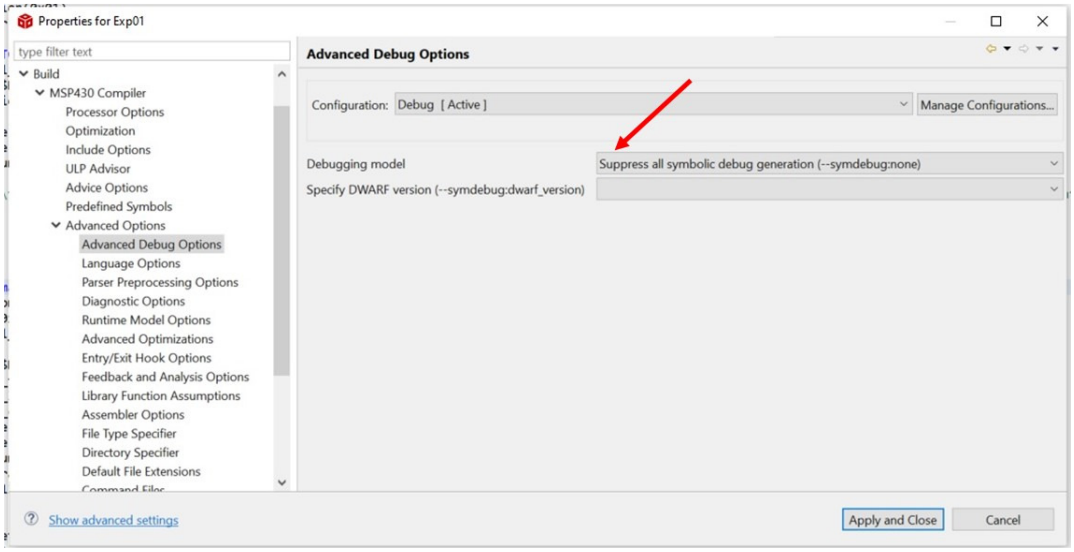

Figura C.10. Desabilitando o debug simbólico.

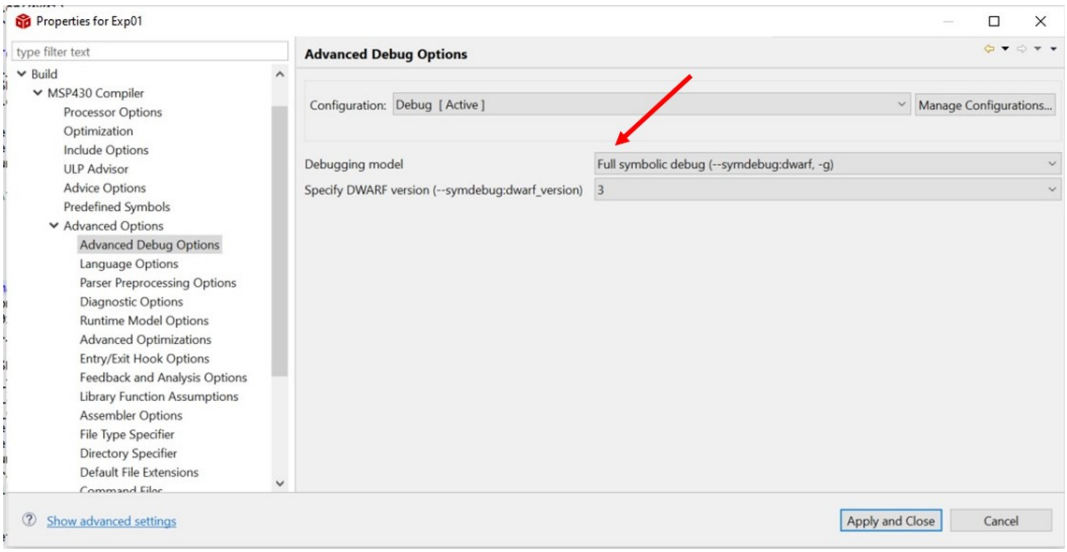

Figura C.11. Habilitando o debug simbólico.

Após desabilitar a opção de *debug* simbólico e de habilitar a opção de manter o arquivo *assembly* podemos executar o compilador, mas sem entrar no *debug*. Uma forma muito prática de fazer isso é com as teclas **CTRL+B** (*Build all*). No diretório Debug do seu projeto vai estar o arquivo com a extensão ".asm".

Apresentamos um exemplo com o programa abaixo denominado Exp1.c. Ele é bastante elementar e dispensa explicações. Vamos compilar usando o *Build all* (**CTRL+B**). Com o "Project Explorer" (aquela coluna mais à esquerda com os arquivos do projeto) do CCS vamos na pasta Debug e pedimos para abrir o arquivo main.asm. Se o leitor mudou o nome deste arquivo, deverá ter algum arquivo do tipo `nome.asm`.

Exp1.c - Listagem

```
// Exp1.c
#include <msp430.h>

int soma (int a, int b);       //Protótipo da função soma()

int main(void){
    volatile int a=2,b=3,c=0;
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    c=soma(a,b);
    while (1);
    return 0;
}

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

Ao abrirmos esse arquivo ".asm" deveremos ver algo parecido com a listagem abaixo. Note que existe uma série de linhas que pouco significado tem para nossa análise. Na listagem, marcamos em azul claro o trecho do código referente à função `soma()`, cujo início é caracterizado pela pseudo-instrução `.sect ".text:soma"` e em amarelo claro o trecho da função `main()`, indicado por `.sect ".text:main"`. Veja que os rótulos `main` e `soma` são indicados como globais. Os comentários do tipo `;[] |15|`. Indicam a linha do programa em C que eles executam. É claro que um *statement* em C deve gerar várias linhas de *assembly*. O leitor é convidado a comparar as linhas de seu programa em C com as linhas do código *assembly*. Para que incluir o [ ]? Não bastava o número. Deve ter alguma razão. Pesquisar mais.

Exp1.asm – Listagem
(A duas opções em negrito serão usadas mais adiante)

```
;******************************************************************************
;* MSP430 G3 C/C++ Codegen                                                   PC
v18.12.5.LTS *
;* Date/Time created: Mon Mar 30 08:12:56 2020                                *
```

```
;*****************************************************************************
      .compiler_opts   --abi=eabi   --diag_wrap=off   --hll_source=on   --
mem_model:code=large    --mem_model:data=large    --object_format=elf    --
silicon_errata=CPU21   --silicon_errata=CPU22   --silicon_errata=CPU23   --
silicon_errata=CPU40 --silicon_version=mspx --symdebug:none
;     C:\ti\ccs930\ccs\tools\compiler\ti-cgt-
msp430_18.12.5.LTS\bin\opt430.exe
C:\\Users\\zelen\\AppData\\Local\\Temp\\{1F65C321-3CCE-499C-8F24-
AF1DEC246547}      C:\\Users\\zelen\\AppData\\Local\\Temp\\{85BE6905-5B0E-
44B6-AC22-4CBB439BFFEB}


      .sect ".text:soma"
      .clink
      .global     soma

;*****************************************************************************
;* FUNCTION NAME: soma                                                       *
;*                                                                           *
;*   Regs Modified     : SP,SR,r12                                           *
;*   Regs Used         : SP,SR,r12,r13                                       *
;*   Local Frame Size  : 0 Args + 0 Auto + 0 Save = 0 byte                   *
;*****************************************************************************
soma:
;* ---------------------------------------------------------------------------*
        ADD.W     r13,r12               ; [] |15|
        RETA      ; []
        ; []
      .sect ".text:main"
      .clink
      .global     main

;*****************************************************************************
;* FUNCTION NAME: main                                                       *
;*                                                                           *
;*   Regs Modified     : SP,SR,r12,r13                                       *
;*   Regs Used         : SP,SR,r12,r13                                       *
;*   Local Frame Size  : 0 Args + 6 Auto + 0 Save = 6 byte                   *
;*****************************************************************************
main:
;* ---------------------------------------------------------------------------*
        SUBA      #6,SP                 ; []
        MOV.W     #2,0(SP)              ; [] |7|
        MOV.W     #3,2(SP)              ; [] |7|
        MOV.W     #7,4(SP)              ; [] |7|
        MOV.W     #23168,&WDTCTL+0      ; [] |8|
        MOV.W     0(SP),r12             ; [] |9|
        MOV.W     2(SP),r13             ; [] |9|
        CALLA     #soma                 ; [] |9|
                                        ; [] |9|
        MOV.W     r12,4(SP)             ; [] |9|
```

```
;* ---------------------------------------------------------------------*
;*   BEGIN LOOP $C$L1
;*
;*   Loop source line                 : 10
;*   Loop closing brace source line   : 10
;*   Known Minimum Trip Count         : 1
;*   Known Maximum Trip Count         : 4294967295
;*   Known Max Trip Count Factor      : 1
;* ---------------------------------------------------------------------*
$C$L1:
        JMP       $C$L1                 ; [] |10|
                                        ; [] |10|
        NOP       ; []
;* ---------------------------------------------------------------------*
;*****************************************************************************
;* UNDEFINED EXTERNAL REFERENCES                                             *
;*****************************************************************************
      .global     WDTCTL

;*****************************************************************************
;* BUILD ATTRIBUTES                                                          *
;*****************************************************************************
      .battr "TI", Tag_File, 1, Tag_LPM_INFO(1)
      .battr                "TI",                Tag_File,                1,
Tag_PORTS_INIT_INFO("012345678901ABCDEFGHIJ00000000000011110000000000000
00000000000000")
      .battr "TI", Tag_File, 1, Tag_LEA_INFO(1)
      .battr "TI", Tag_File, 1, Tag_HW_MPY32_INFO(2)
      .battr "TI", Tag_File, 1, Tag_HW_MPY_ISR_INFO(1)
      .battr "TI", Tag_File, 1, Tag_HW_MPY_INLINE_INFO(1)
      .battr "mspabi", Tag_File, 1, Tag_enum_size(3)
```

O leitor pode iniciar a sessão de *debug* (F11) com o *debug* simbólico desabilitado. Apenas será possível executar o passo-a-passo no *disassembly*. Note que o *debugger* oferece a opção de mostrar do "**Disassembly**" e o arquivo `maim.asm` (ou seu_nome.asm). O arquivo .asm deve ser o da listagem acima. Já o **Disassembly** é interessante e deve se parecer com a listagem abaixo. Nome que marcamos os pontos de início das funções `main` e `soma`.

Disassembly de Exp1.asm - Listagem

```
        main():
010000:   00B1 0006           SUBA    #0x00006,SP
010004:   43A1 0000           MOV.W   #2,0x0000(SP)
010008:   40B1 0003 0002      MOV.W   #0x0003,0x0002(SP)
01000e:   40B1 0007 0004      MOV.W   #0x0007,0x0004(SP)
010014:   40B2 5A80 015C      MOV.W   #0x5a80,&Watchdog_Timer_WDTCTL
01001a:   412C                MOV.W   @SP,R12
01001c:   411D 0002           MOV.W   0x0002(SP),R13
010020:   13B1 0036           CALLA   #soma
010024:   4C81 0004           MOV.W   R12,0x0004(SP)
```

```
        $C$L1:
010028:   3FFF                JMP     (0x0028)
01002a:   4303                NOP
        C$$EXIT(), abort():
01002c:   4303                NOP
        $C$L1:
01002e:   3FFF                JMP     (0x002e)
010030:   4303                NOP
        _system_pre_init():
010032:   431C                MOV.W   #1,R12
010034:   0110                RETA
        soma():
010036:   5D0C                ADD.W   R13,R12
010038:   0110                RETA
```

Um fato interessante que se descobre aqui é que quando se programa em C, o compilador (na verdade, o *linker*), devido ao modelo de memória **large**, grava o programa a partir do endereço 0x10000, ou seja, a partir da segunda página de 64 KB. Se a linha de endereço A16 é mantida em "1" e o programa pode imaginar que começou no endereço zero. Veja que os saltos relativos indicam apenas endereços com 16 bits, como por exemplo JMP (0x0028). O leitor pode pedir para examinar a memória a partir do endereço 0x10000 e vai descobrir que lá foram colocados os *op-codes* de seu programa. (veja na listagem de Exp1.asm as opções indicando ao compilador para usar o modelo de memória large **`mem_model:code=large --mem_model:data=large`**).

Para finalizar, convidamos o leitor a modificar a linha do programa em C, onde declara as variáveis e remover o modificador `volatile`. Compile o programa e examine a listagem *assembly*. O trecho relativo ao `main` foi reproduzida abaixo. Veja que o programa resultante nada faz. O otimizador do compilador detectou (corretamente) que a operação feita com as variáveis `a`, `b` e `c` não tinha qualquer efeito e as omitiu. Nestas condições o programa resultante apenas configura o `WDTCTL` (*Watch Dog Timer*) e fica num laço infinito. O compilador nem chegou a criar as variáveis! Sem analisar a listagem *assembly*, é difícil concluir por essa otimização. Mais uma razão para gerarmos o *assembly*: entender as misteriosas otimizações realizadas pelo CCS.

Exp1.asm – Listagem com as variáveis declaradas sem o uso do modificador `volatile`

```
      .sect ".text:main"
      .clink
      .global     main

;*****************************************************************************
;* FUNCTION NAME: main                                                       *
;*                                                                           *
;*   Regs Modified     :                                                     *
;*   Regs Used         :                                                     *
;*   Local Frame Size  : 0 Args + 0 Auto + 0 Save = 0 byte                   *
;*****************************************************************************
main:
;* ---------------------------------------------------------------------------*
```

```
        MOV.W     #23168,&WDTCTL+0      ; [] |8|
;* ------------------------------------------------------------------------*
;*   BEGIN LOOP $C$L1
;*   Loop source line                 : 10
;*   Loop closing brace source line   : 10
;*   Known Minimum Trip Count         : 1
;*   Known Maximum Trip Count         : 4294967295
;*   Known Max Trip Count Factor      : 1
;* ------------------------------------------------------------------------*
$C$L1:
        JMP       $C$L1                 ; [] |10|
        NOP       ; []
```

C.10.2. Entendendo como o CCS Gerencia a Passagem de Argumentos de Funções

O que é estudado aqui foi resumido do manual "*slau132o*". São abordados tópicos dos seguintes itens:

- 6.3 *Register Conventions* (pag 128)
- 6.4 *Function Structure and Calling Conventions* (pag 129)
- 6.4.1 *How a Function Makes a Call* (pag 130)
- 6.4.2 *How a Called Function Responds* (pag 130)
- 6.4.3 *Accessing Arguments and Local Variables*

Nossa intenção é entender um pouco como o CCS organiza os parâmetros quando se chama uma função e como a função chamada os acessa. Este tópico é uma preparação para o próximo que mistura programação em C e em *assembly*. Nos exemplos aqui estudados, sempre declaramos as variáveis como `volatile` para permitir que os programas fiquem simples e que não sejam alterados pelo otimizador. A listagem do *assembly* foi simplificada com a remoção das linhas que não eram importantes para o estudo neste momento. O leitor é convidado a seguir adiante por esse caminho, analisando diversos outros casos, ou estudando os casos específicos para suas necessidades.

Como é a primeira vez, vamos ensaiar com um programa muito simples, cuja listagem em C está logo adiante. Ele foi escrito de forma a possibilitar o entendimento de como são criadas as variáveis e chamadas as funções. Por isso, ele não faz grande coisa (na verdade, ele nada faz de útil). Este programa cria duas variáveis globais: `res1` e `res2`. A função `main` cria e inicializa 6 variáveis locais `a`, `b`, `c`, `d`, `e`, `f` e `g`, e a variável estática `dif`, também já inicializada. A inicialização é útil, pois facilita a identificação da variável na memória. Depois chama a função `soma` duas vezes e armazena os resultados em variáveis locais e em seguida as transfere para as variáveis globais `res1` e `res2`. Antes de entrar num laço infinito, guarda na variável estática `dif` uma conta com `dif`, `res1` e `res2`.

Antes de prosseguirmos, é importante citar que o CCS adota tipos de modelos de memória. Para o atual estudo, não vale a pena entrar em detalhes. De forma simples, explicamos a seguir cada um deles. A Figura C.12 indica onde eles podem ser gerenciados.

- **Small model** → (MSP430 e MSP430X) usa endereçamento de 16 bits, ou seja, tudo limitado dentro da primeira página de 64 KB.

- **Large model** → (MSP430X) endereçamento de 20 bits, neste caso o programa inicia em 0x10000 e são usadas as instruções que manipulam endereços com 20 bits, como CALLA. Ao que parece, os dados ficam limitados em 64 KB.
- **Restricted model** → para efeitos deste estudo, ele é idêntico ao *Large*.

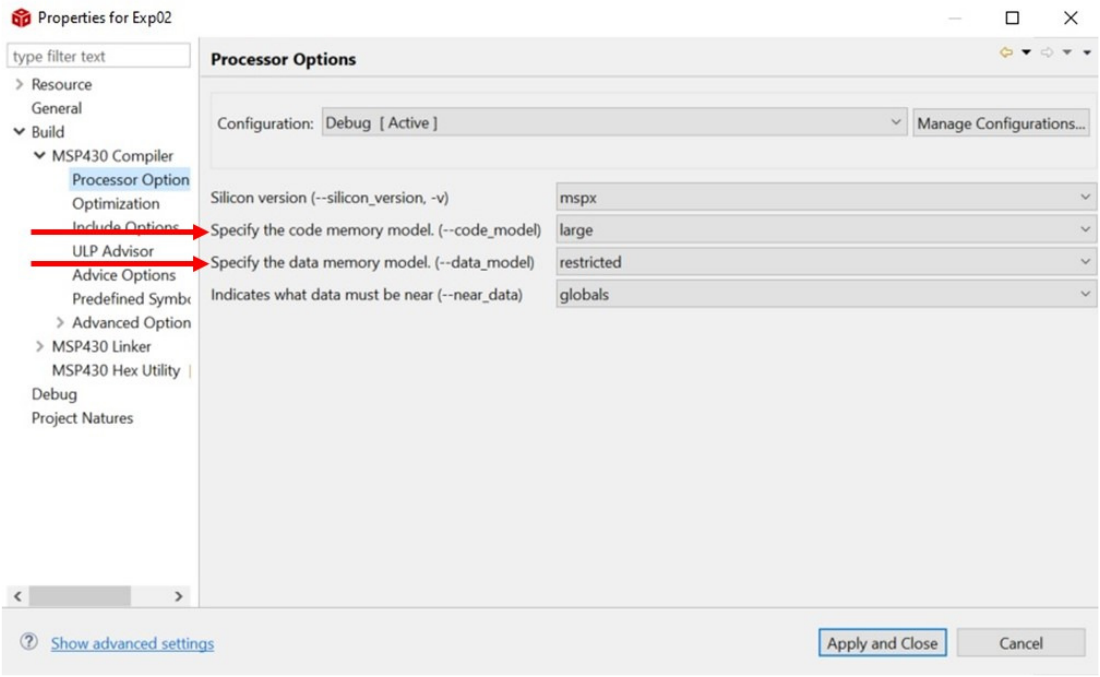

Figura C.12. Indicação de como selecionar os modelos de memória a serem usados pelo compilador.

Veja que logo no começo da listagem do arquivo.asm aparecem as indicações da linha de comando usada na compilação. Abaixo está um pequeno trecho desta linha, onde se vê em destaque que o modelo para código e dados é o `large`.

```
.compiler_opts --abi=eabi --diag_wrap=off --hll_source=on --
mem_model:code=large --mem_model:data=large ...
```

Um tópico muito importante é sobre a forma que os registradores são usados (slau132o, pag 129):

- R12, ..., R15 → passagem de argumentos das funções. R12 recebe o primeiro argumento, R13 o segundo argumento, R14 o terceiro argumento e, finalmente, R15 o quarto argumento. Se a função trabalha com mais de 4 argumentos, os demais são passados pela pilha. Eles também são usados para retornar valores.
- R11 → de uso geral, deve ser preservado pela função chamadora.
- R4, ..., R10 → são de uso geral pela função chamada, que deve preservá-los, isto significa que se a função os utiliza, deve salvá-los primeiro e restaurá-los antes de regressar.

Logo a seguir estão três listagens, com trechos marcados em amarelo e azul. Para evitar confusão e facilitar referenciamento, elas estão numeradas da seguinte forma:

- Listagem 1 → Programa original, Exp02.c
- Listagem 2 → Arquivo *assembly*, Exp02.asm
- Listagem 3 → Cópia do *Disassembly*

Listagem 1: Programa Exp02.c

```
// Exp02.c
#include <msp430.h>

int soma (int x, int y, int z, int w, int t);

volatile int result;

int main(void){
    volatile int a=1,b=2,c=3,d=4,e=5,f=6,g=7;
    volatile static int dif=9;
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    f = soma(a, b, c, d, e, 10);
    res1 = f;
    g = soma(a, b, c, 10, 20, 30);
    res2 = g;
    dif = dif + res2 - res1;
    while(1);
    return 0;
}

int soma (int x, int y, int z, int w, int t, int nr){
    volatile int aux1=15, aux2=16;
    aux1 = x + y + z;
    aux2 = w + t +nr;
    aux1 = aux1 + aux2;
    return aux1;
}
```

A listagem assembly a seguir foi simplificada. Várias linhas foram removidas, de forma a destacar o que é importante no momento. Os trechos coloridos facilitam a correspondência para com programa em C. As variáveis globais `res1` e `res2` foram criadas com as declarações abaixo. Não foi possível entender completamente o que cada declaração faz. Apresentamos uma suposição, para não dizer adivinhação.

- `.global res1` → Indica que `res1` é uma variável global.
- `.common res1,2,2` → Reserva espaço no chamado Bloco Comum (*Common Block*), que que será alocada pelo *linker*, provavelmente a partir do endereço 0x2400 (MSP430 F5529). O primeiro dígito 2 indica que o tamanho é de 2 *bytes* e o segundo dígito 2 indica que o alinhamento é de 2 em 2, ou seja, alocar em endereços pares.
- `.global res2` → Idem acima.
- `.common res2,2,2` → Idem acima, mas com o endereço 0x2402.

- `.data` → indica que será dado início ao segmento de dados
- `.align 2` → alinhamento de 2 em 2, endereços pares
- `.elfsym dif$1,SYM_SIZE(2)` → Nome da variável estática `dif`
- `dif$1: .bits 0x9,16` → 9 representado em 16 bits

Os endereços podem ser descobertos abrindo o arquivo.map, na pasta *Debug*. No caso deste exemplo, é o arquivo Exp02.map. Este arquivo oferece uma listagem de todos os símbolos globais do programa, ordenados alfabeticamente e por endereços. Faça a busca pelas palavras `main`, `soma`, `res1` e `res2`. O leitor deverá encontrar os seguintes endereços:

- `00010000 main`
- `00010106 soma`
- `00002400 res1`
- `00002402 res2`
- `00002402 dif`

Listagem 2: Arquivo Exp02.asm

```
      .global     res1
      .common     res1,2,2
      .global     res2
      .common     res2,2,2
      .data
      .align      2
      .elfsym     dif$1,SYM_SIZE(2)
dif$1:
      .bits       0x9,16

      .sect ".text:soma"
      .clink
      .global     soma
;*************************************************************************
;* FUNCTION NAME: soma                                                   *
;*   Regs Modified     : SP,SR,r10,r11,r12,r14,r15                       *
;*   Regs Used         : SP,SR,r10,r11,r12,r13,r14,r15                   *
;*   Local Frame Size  : 0 Args + 2 Auto + 4 Save = 6 byte               *
;*************************************************************************
soma:
        PUSHM.A   #1,r10              ; []
        SUBA      #4,SP               ; []
        MOVA      r15,r10             ; [] |21|
        MOV.W     14(SP),r11          ; [] |21|
        MOV.W     12(SP),r15          ; [] |21|
        MOV.W     #15,0(SP)           ; [] |22|
        MOV.W     #16,2(SP)           ; [] |22|
        ADD.W     r13,r12             ; [] |23|
        ADD.W     r12,r14             ; [] |23|
        MOV.W     r14,0(SP)           ; [] |23|
        ADD.W     r10,r15             ; [] |24|
        ADD.W     r11,r15             ; [] |24|
```

```
        MOV.W     r15,2(SP)             ; [] |24|
        ADD.W     2(SP),0(SP)           ; [] |25|
        MOV.W     0(SP),r12             ; [] |26|
        ADDA      #4,SP                 ; []
        POPM.A    #1,r10                ; []
        RETA      ; []

      .sect ".text:main"
      .clink
      .global     main
;*****************************************************************************
;* FUNCTION NAME: main                                                       *
;*   Regs Modified     : SP,SR,r11,r12,r13,r14,r15                           *
;*   Regs Used         : SP,SR,r11,r12,r13,r14,r15                           *
;*   Local Frame Size  : 4 Args + 14 Auto + 0 Save = 18 byte                 *
;*****************************************************************************
main:
        SUBA      #18,SP                ; []
        MOV.W     #1,4(SP)              ; [] |9|
        MOV.W     #2,6(SP)              ; [] |9|
        MOV.W     #3,8(SP)              ; [] |9|
        MOV.W     #4,10(SP)             ; [] |9|
        MOV.W     #5,12(SP)             ; [] |9|
        MOV.W     #6,14(SP)             ; [] |9|
        MOV.W     #7,16(SP)             ; [] |9|
        MOV.W     #23168,&WDTCTL+0      ; [] |11|
        MOV.W     12(SP),0(SP)          ; [] |12|
        MOV.W     #10,2(SP)             ; [] |12|
        MOV.W     4(SP),r12             ; [] |12|
        MOV.W     6(SP),r13             ; [] |12|
        MOV.W     8(SP),r14             ; [] |12|
        MOV.W     10(SP),r15            ; [] |12|
        CALLA     #soma                 ; [] |12|
                                        ; [] |12|
        MOV.W     r12,14(SP)            ; [] |12|
        MOV.W     14(SP),&res1+0        ; [] |13|
        MOV.W     #20,0(SP)             ; [] |14|
        MOV.W     #30,2(SP)             ; [] |14|
        MOV.W     4(SP),r12             ; [] |14|
        MOV.W     6(SP),r13             ; [] |14|
        MOV.W     8(SP),r14             ; [] |14|
        MOV.W     #10,r15               ; [] |14|
        CALLA     #soma                 ; [] |14|
                                        ; [] |14|
        MOV.W     r12,16(SP)            ; [] |14|
        MOV.W     16(SP),&res2+0        ; [] |15|
        MOV.W     &res2+0,r15           ; [] |16|
        ADD.W     &dif$1+0,r15          ; [] |16|
        SUB.W     &res1+0,r15           ; [] |16|
        MOV.W     r15,&dif$1+0          ; [] |16|
```

```
;* ----------------------------------------------------------------------*
;*   BEGIN LOOP $C$L1
;*   Loop source line                 : 17
;*   Loop closing brace source line   : 17
;*   Known Minimum Trip Count         : 1
;*   Known Maximum Trip Count         : 4294967295
;*   Known Max Trip Count Factor      : 1
;* ----------------------------------------------------------------------*
$C$L1:
        JMP       $C$L1                 ; [] |17|
                                        ; [] |17|
        NOP       ; []
;* ----------------------------------------------------------------------*
```

Listagem 3: Cópia do Disassembly

```
        main():
010000:   00B1 0012           SUBA    #0x00012,SP
010004:   4391 0004           MOV.W   #1,0x0004(SP)
010008:   43A1 0006           MOV.W   #2,0x0006(SP)
01000c:   40B1 0003 0008      MOV.W   #0x0003,0x0008(SP)
010012:   42A1 000A           MOV.W   #4,0x000a(SP)
010016:   40B1 0005 000C      MOV.W   #0x0005,0x000c(SP)
01001c:   40B1 0006 000E      MOV.W   #0x0006,0x000e(SP)
010022:   40B1 0007 0010      MOV.W   #0x0007,0x0010(SP)
010028:   40B2 5A80 015C      MOV.W   #0x5a80,&Watchdog_Timer_WDTCTL
01002e:   4191 000C 0000      MOV.W   0x000c(SP),0x0000(SP)
010034:   40B1 000A 0002      MOV.W   #0x000a,0x0002(SP)
01003a:   411C 0004           MOV.W   0x0004(SP),R12
01003e:   411D 0006           MOV.W   0x0006(SP),R13
010042:   411E 0008           MOV.W   0x0008(SP),R14
010046:   411F 000A           MOV.W   0x000a(SP),R15
01004a:   13B1 0184           CALLA   #soma
01004e:   4C81 000E           MOV.W   R12,0x000e(SP)
010052:   4192 000E 2400      MOV.W   0x000e(SP),&0x2400
010058:   40B1 0014 0000      MOV.W   #0x0014,0x0000(SP)
01005e:   40B1 001E 0002      MOV.W   #0x001e,0x0002(SP)
010064:   411C 0004           MOV.W   0x0004(SP),R12
010068:   411D 0006           MOV.W   0x0006(SP),R13
01006c:   411E 0008           MOV.W   0x0008(SP),R14
010070:   403F 000A           MOV.W   #0x000a,R15
010074:   13B1 0184           CALLA   #soma
010078:   4C81 0010           MOV.W   R12,0x0010(SP)
01007c:   4192 0010 2402      MOV.W   0x0010(SP),&0x2402
010082:   421F 2402           MOV.W   &0x2402,R15
010086:   521F 2404           ADD.W   &0x2404,R15
01008a:   821F 2400           SUB.W   &0x2400,R15
01008e:   4F82 2404           MOV.W   R15,&0x2404
        $C$L1:
```

```
010092:   3FFF                JMP      (0x0092)
010094:   4303                NOP
...
        soma():
010106:   140A                PUSHM.A #1,R10
010108:   03F1                DECDA   SP
01010a:   411B 000C           MOV.W   0x000c(SP),R11
01010e:   411A 000A           MOV.W   0x000a(SP),R10
010112:   5D0C                ADD.W   R13,R12
010114:   5C0E                ADD.W   R12,R14
010116:   5E0F                ADD.W   R14,R15
010118:   5A0F                ADD.W   R10,R15
01011a:   5B0F                ADD.W   R11,R15
01011c:   4F81 0000           MOV.W   R15,0x0000(SP)
010120:   412C                MOV.W   @SP,R12
010122:   03E1                ICDA   SP
010124:   160A                POPM.A  #1,R10
010126:   0110                RETA
```

Vamos agora entender como programa em C funciona, visto a partir do assembly. Um tópico muito importante é o uso da pilha. Vamos mostrar ao leitor como percorrer todo o programa e, ao mesmo tempo, examinar a pilha nos instantes mais importantes. Para ficar fácil a explicação, a pilha será representada na horizontal. A caixa marcada em amarelo representa a posição apontada pelo ponteiro (SP). Quando a execução do programa é iniciada, o SP aponta para o endereço de regresso da `main()`, aquele `return 0` que é colocado ao final quando o projeto é criado. Ele retorna para um endereço que prende a execução num laço infinito. Para os curiosos, a função é `void abort (void)` que só tem o *statement* `for (;;);`.

Lembrar de que temos na memória RAM as seguintes variáveis

0x2400	0x2402	0x2404
global	global	estática
res1	res2	dif
0	0	9

O trecho a seguir é bastante trabalhoso, mas pretende mostrar ao leitor como interpretar o *assembly* gerado a partir do C. Vamos ficar também como uma melhor ideia do funcionamento do C.

As variáveis locais são criadas na pilha e acessadas dentro da pilha. Quando uma função é chamada, os primeiros 4 argumentos são colocados, em sequência, em R12, R13, R14 e R15. Se existem mais que 4 argumentos, eles serão colocados na pilha. O valor retornado pela função, se inteiro, é guardado em R12, se for long, deve usar R12 e R13 (confirmar).

Em todas as tabelas abaixo que representam pilha, a primeira e a terceira linha estão com os valores em hexadecimal. O leitor pode acompanhar as ilustrações abaixo, ao mesmo tempo em que executa o programa no debug do CCS. Porém, deve ser alertado de que as tabelas que representam a pilha têm o endereço maior à esquerda, enquanto que a função *Memory Browser*, apresenta o conteúdo dos endereços na ordem contrária.

1) Estado inicial, quando o controle é passado para a função `main()`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+2	SP	-2	-4	-6	-8	-10	-12	-14	-16	-18	-20	-22	-24	-26	-28	-30	-32
0000	4416	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
RET0		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

R10 = ??, R11 = ??, R12 = ??, R13 = ??, R14 = ??, R15 = ??

2) Estado após a instrução: `SUBA #18,SP`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
RET0		-	-	-	-	-	-	-	-	-	-	-			-	-	-

R10 = ??, R11 = ??, R12 = ??, R13 = ??, R14 = ??, R15 = ??

3) Estado após a instrução: `MOV.W #1,4(SP)`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-
RET0		-	-	-	-	-	-	a	-	-	-	-				-	-

R10 = ??, R11 = ??, R12 = ??, R13 = ??, R14 = ??, R15 = ??

4) Estado após a instrução: `MOV.W #2,6(SP)`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	-	-	-	-	-	2	1	-	-	-	-	-	-	-	-	-
RET0		-	-	-	-	-	b	a	-	-	-	-					-

R10 = ??, R11 = ??, R12 = ??, R13 = ??, R14 = ??, R15 = ??

5) Assim segue até criar na pilha todas as variáveis locais. Estado após as instruções:

```
MOV.W   #3,8(SP)      MOV.W   #4,10(SP)      MOV.W   #5,12(SP)
MOV.W   #6,14(SP)     MOV.W   #7,16(SP)
```

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	7	6	5	4	3	2	1	-	-	-	-	-	-	-	-	-
RET0		g	f	e	d	c	b	a	-	-	-	-					

R10 = ??, R11 = ??, R12 = ??, R13 = ??, R14 = ??, R15 = ??

6) Colocar argumentos nos registradores. Preparação para chamar a função `f = soma(a, b, c, d, e, 10);`

Estado após as instruções:

```
MOV.W   12(SP),0(SP)      MOV.W   #10,2(SP)      MOV.W 4(SP),R12
MOV.W   6(SP),R13         MOV.W 8(SP),R14        MOV.W 10(SP),R15
```

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	7	6	5	4	3	2	1	A	5	-	-	-	-	-	-	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	-	-	-	-	-	-	-

R12 = a, R13 = b, R14 = c, R15 = d

7) Transcrevemos abaixo um trecho da listagem com o *disassembly*. O leitor deve notar que a instrução `CALLA #soma` está no endereço 0x1004A. Quando esta instrução é executada, o contador de programa (PC) está apontando para a próxima instrução (0x01004E). Este é o endereço de regresso que será colocado na pilha. Está representado do por RET1

```
01004a:   13B1 0184          CALLA   #soma
01004e:   4C81 000E          MOV.W   R12,0x000e(SP)
```

Estado após a execução da instrução `CALLA #soma`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-10	-12	-14
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	-	-	-	-	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	-	-	-	-	-

R12 = a, R13 = b, R14 = c, R15 = d

8) Ao entrar na função `soma()`, a primeira instrução é `PUSHM.A #1,r10`. Ela faz parte das instruções para a CPUX e apenas coloca R10 na pilha. Isso foi feito porque os registradores R4, ..., R10 devem ser salvos antes de serem usados. A função soma usa R10.
Estado após a execução da instrução `PUSHM.A #1,r10`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+28	+26	+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	-	-	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	-	-	-

R10 = ??, R12 = a, R13 = b, R14 = c, R15 = d

9) O SP é movido duas posições para abrir espaço para as variáveis locais `aux1` e `aux2`
Estado após a execução da instrução `SUBA #4,SP`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+32	+30	+28	+26	+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	-	-	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = ??, R12 = a, R13 = b, R14 = c, R15 = d

10) Estado após a execução da instrução `MOVA.W r15,r10`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+32	+30	+28	+26	+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	-	-	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = d, R11 = #10, R12 = a, R13 = b, R14 = c, R15 = d

11) Estado após a execução da instrução: `MOV.W 14(SP),r11`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+32	+30	+28	+26	+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	-	-	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = d, R11 = #10, R12 = a, R13 = b, R14 = c, R15 = d

12) Estado de pilha após a execução da instrução: `MOV.W 12(SP),r15`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+32	+30	+28	+26	+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	-	-	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = d, R11 = #10, R12 = a, R13 = b, R14 = c, R15 = e

13) As duas variáveis locais `aux1` e `aux2` são inicializadas
Estado após a execução das instruções: `MOV.W #15,0(SP)` `MOV.W #16,2(SP)`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+32	+30	+28	+26	+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	10	0F	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = d, R11 = #10, R12 = a, R13 = b, R14 = c, R15 = e

14) O programa agora faz a primeira série de somas.
Estado após a execução das instruções: `ADD.W r13,r12` `ADD.W r12,r14`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+32	+30	+28	+26	+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	10	0F	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = d, R11 = #10, R12 = a+b, R13 = b, R14 = a+b+c, R15 = e

15) Armazenar o resultado em `aux1` (`aux1 = x + y + z;`)
Estado após a execução da instrução: `MOV.W r14,0(SP)`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+32	+30	+28	+26	+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	10	6	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = d, R11 = #10, R12 = a+b, R13 = b, R14 = a+b+c, R15 = e

16) O programa agora faz a segunda série de somas.
Estado após a execução das instruções: `ADD.W r10,r15` `ADD.W r11,r15`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+32	+30	+28	+26	+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	10	6	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = d, R11 = #10, R12 = a+b, R13 = b, R14 = a+b+c, R15 = d+e+#10

17) Armazenar o resultado em `aux2` (`aux1 = w + t + nr;`)
Estado após a execução da instrução: `MOV.W r15,2(SP)`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+32	+30	+28	+26	+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	13	6	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = d, R11 = #10, R12 = a+b, R13 = b, R14 = a+b+c, R15 = d+e+#10

18) Calcular soma de `aux1 = aux1 + aux2;`
Estado após a execução da instrução: `ADD.W 2(SP),0(SP)`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+32	+30	+28	+26	+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	13	19	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = d, R11 = #10, R12 = a+b, R13 = b, R14 = a+b+c, R15 = d+e+#10

19) Preparar retorno, o argumento de retorno é colocado em R12 `return aux1;`
Estado após a execução da instrução: `MOV.W 0(SP),r12`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+32	+30	+28	+26	+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	13	19	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = d, R11 = #10, R12 = a+b+c+d+e+#10=0x19, R13 = b, R14 = a+b+c, R15 = d+e+#10

20) Voltar ponteiro da pilha para antes da criação das duas variáveis aux1 e aux2
Estado após a execução da instrução: `ADDA #4,SP`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+28	+26	+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	13	19	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = d, R11 = #10, R12 = a+b+c+d+e+#10=0x19, R13 = b, R14 = a+b+c, R15 = d+e+#10

21) Restaurar valor de R10. Estado após a execução da instrução: `POPM.A #1,r10`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	13	19	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = ??, R11 = #10, R12 = a+b+c+d+e+#10=0x19, R13 = b, R14 = a+b+c, R15 = d+e+#10

22) Retornar para o programa principal (PC = 0x01004E), lembrar que o resultado da soma ficou em R12.
Estado após a execução da instrução: `RETA`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10
0000	4416	7	6	5	4	3	2	1	A	5	0001	004E	?	?	13	19	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = ??, R11 = #10, R12 = a+b+c+d+e+#10=0x19, R13 = b, R14 = a+b+c, R15 = d+e+#10

23) Transferir o resultado para a variável `f = soma(a, b, c, d, e, 10);`
Estado após a execução da instrução: `MOV.W r12,14(SP)`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	7	19	5	4	3	2	1	A	5	0001	004E	?	?	13	19	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = ??, R11 = #10, R12 = a+b+c+d+e+#10=0x19, R13 = b, R14 = a+b+c, R15 = d+e+#10

24) Transferir copiar o resultado para a variável global `res1` (0x2400)
Estado após a execução da instrução: `MOV.W 14(SP),&res1+0`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	7	19	5	4	3	2	1	A	5	0001	004E	?	?	13	19	-
RET0	RET0	g	f	e	d	c	b	a	#10	e	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = ??, R11 = #10, R12 = a+b+c+d+e+#10=0x19, R13 = b, R14 = a+b+c, R15 = d+e+#10

25) Preparação dos argumentos para chamar a função `g = soma(a, b, c, 10, 20, 30);`
Estado após a execução das instruções:

```
MOV.W  #20,0(SP)     MOV.W  #30,2(SP)     MOV.W  4(SP),r12
MOV.W  6(SP),r13     MOV.W  8(SP),r14     MOV.W  #10,r15
```

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	7	19	5	4	3	2	1	1E	14	0001	004E	?	?	13	19	-
RET0	RET0	g	f	e	d	c	b	a	#30	#20	RET1	RET1	R10H	R10L	aux2	aux1	-

R10 = ??, R11 = #10, R12 = a, R13 = b, R14 = c, R15 = #10

26) Ver que o endereço de regresso para o Programa Principal é 0x10078.
Estado após a execução da instrução `CALLA #soma`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+24	+22	+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10
0000	4416	7	19	5	4	3	2	1	1E	14	0001	0078	-	-	-	-	-
RET0	RET0	g	f	e	d	c	b	a	#30	#20	RET2	RET2	-	-	-	-	-

R10 = ??, R11 = #10, R12 = a, R13 = b, R14 = c, R15 = #10

27) A função soma é executada. Tudo semelhante à partir do item 23.
Retornar para o programa principal (PC = 0x010078), lembrar que o resultado da soma ficou em R12
Estado após a execução da instrução: `RETA`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	7	19	5	4	3	2	1	1E	14	0001	0078	?	?	3C	42	-
RET0	RET0	g	f	e	d	c	B	a	#30	#20	RET2	RET2	R10H	R10L	aux2	aux1	-

R10 = ??, R11 = #20, R12 = a+b+c+#10+#20+#30=0x42, R13 = b, R14 = a+b+c, R15 = #10+#20,#30

28) Transferir o resultado para a variável `g = soma(a, b, c, 10, 20, 30);`
Estado após a execução da instrução: `MOV.W r12,16(SP)`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	42	19	5	4	3	2	1	1E	14	0001	0078	?	?	3C	42	-
RET0	RET0	g	f	e	d	c	B	a	#30	#20	RET2	RET2	R10H	R10L	aux2	aux1	-

R10 = ??, R11 = #20, R12 = a+b+c+#10+#20+#30=0x42, R13 = b, R14 = a+b+c, R15 = #10+#20,#30

29) Transferir copiar o resultado para a variável global `res2` (0x2402)
Estado após a execução da instrução: `MOV.W 16(SP),&res2+0`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	42	19	5	4	3	2	1	1E	14	0001	0078	?	?	3C	42	-
RET0	RET0	g	f	e	d	c	B	a	#30	#20	RET2	RET2	R10H	R10L	aux2	aux1	-

R10 = ??, R11 = #20, R12 = a+b+c+#10+#20+#30=0x42, R13 = b, R14 = a+b+c, R15 = #10+#20,#30

30) Estado após a execução da instrução: `MOV.W &res2+0,r15`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	42	19	5	4	3	2	1	1E	14	0001	0078	?	?	3C	42	-
RET0	RET0	g	f	e	d	c	B	A	#30	#20	RET2	RET2	R10H	R10L	aux2	aux1	-

R10 = ??, R11 = #20, R12 = a+b+c+#10+#20+#30=0x42, R13 = b, R14 = a+b+c, R15 = 0x42

31) Estado após a execução da instrução: `ADD.W &dif$1+0,r15`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	42	19	5	4	3	2	1	1E	14	0001	0078	?	?	3C	42	-
RET0	RET0	g	f	e	d	c	B	A	#30	#20	RET2	RET2	R10H	R10L	aux2	aux1	-

R10 = ??, R11 = #20, R12 = a+b+c+#10+#20+#30=0x42, R13 = b, R14 = a+b+c, R15 = 0x4B

32) Estado após a execução da instrução: `ADD.W &res1+0,r15`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	42	19	5	4	3	2	1	1E	14	0001	0078	?	?	3C	42	-
RET0	RET0	g	f	e	d	c	B	A	#30	#20	RET2	RET2	R10H	R10L	aux2	aux1	-

R10 = ??, R11 = #20, R12 = a+b+c+#10+#20+#30=0x42, R13 = b, R14 = a+b+c, R15 = 0x32

33) Estado de pilha após a execução da instrução: `MOV.W r15,&dif$1+0`

43FE	43FC	43FA	43F8	43F6	43F4	43F2	43F0	43EE	43EC	43EA	43E8	43E6	43E4	43E2	43E0	43DE	43DC
+20	+18	+16	+14	+12	+10	+8	+6	+4	+2	SP	-2	-4	-6	-8	-10	-12	-14
0000	4416	42	19	5	4	3	2	1	1E	14	0001	0078	?	?	3C	42	-
RET0	RET0	g	f	e	d	c	B	A	#30	#20	RET2	RET2	R10H	R10L	aux2	aux1	-

R10 = ??, R11 = #20, R12 = a+b+c+#10+#20+#30=0x42, R13 = b, R14 = a+b+c, R15 = 0x32

34) Programa finaliza com laço infinito `while(1)`.

```
$C$L1:  JMP     $C$L1
        NOP
```

C.11. Mesclar C com Assembly no CCS

Aqui damos início a um grande desafio: mesclar C com *assembly*, usando o CCS. É claro que não vamos conseguir abordar este tema de forma completa. Pretendemos aqui passar algumas sugestões para os leitores, que devem fazer suas adaptações de acordo com suas necessidades. Uma grande dificuldade nesta tarefa de incluir *assembly* é o otimizador do CCS. Algumas coisas não acontecem exatamente como o esperado. Por isso, espere por surpresas e faça muitos testes.

Será muito usada a declaração `__asm`, que permite inserir linhas em *assemby* dentro de um programa em C. (slau1320, pag 95, item 5.10 The __asm Statemet). O documento *slaa140a, MixingC and Assembler With MSP430 MCUs* também pode ajudar.

Proposta 1: Transformar Função em C para Assembly

Esta proposta parece ser a mais simples. Escrever primeiro o programa em C, com todas as funções. Depois de ter certeza de que ele funciona corretamente, modificar para *assembly* as funções que desejar. Assim, o compilador de encarrega de toda as tarefas e fazemos apenas uma intervenção muito bem localizada. Caso seja possível, é recomendado o uso do modelo de memória *small*.

Vamos ao trabalho. Tomando como exemplos o Exp02.c e Exp02.asm e acreditando que já conhecemos lógica do compilador, podemos escrever uma função `soma` mais enxuta. Os modelos de memória foram alterados para ***small*** para evitar as instruções com endereços de 20 bits. Nosso desafio: escrever todo programa em C e depois substituir a função `soma` pela nossa rotina em *assembly*. A listagem abaixo mostra o programa Exp03.c, que é bastante simples e que será a base para a alteração. A função `soma` cria duas variáveis, `aux1` e `aux2`, que poderiam ser dispensadas. Porém seguimos assim, como sugestão para uma nova versão que venha a necessitar de variáveis locais.

Listagem Exp03.c

```
// Exp03.c
#include <msp430.h>

int soma (int x, int y, int z, int w, int t, int nr);
volatile int res1, res2;

int main(void){
    volatile int a=1,b=2,c=3,d=4,e=5,f=6,g=7;
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    g = soma(a, b, c, d, e, f);
    res1 = g;
    while(1);
    return 0;
}

// Esta função será alterada para assembly
```

```
int soma (int p, int q, int r, int s, int t, int u){
    volatile int aux1=15, aux2=16;
    aux1 = p + q + r + s + t + u;
    return aux1;
}
```

Abaixo está a listagem assembly do trecho encarregado de executar a função `int soma (int p, int q, int r, int s, int t, int u).`

Lembramos que a passagem de argumentos é feita por registradores e pilha.
Registradores: `R12 = p,  R13 = q,  R14 = r,  R15 =s`
Pilha: `0(SP) = t,  2(SP)= u`

Listagem da parte relativa à função soma do arquivo Exp03.asm. Adicionamos alguns comentários explicativos

```
soma:  PUSHM.W   #1,r10                  ; []
        SUB.W     #4,SP                  ; []
        MOV.W     10(SP),r11             ; [] |17| R11 = u
        MOV.W     8(SP),r10              ; [] |17| R10 = t
        MOV.W     #15,0(SP)              ; [] |18|
        MOV.W     #16,2(SP)              ; [] |18|
        ADD.W     r13,r12                ; [] |19| R12=p+q
        ADD.W     r12,r14                ; [] |19| R14=p+q+r
        ADD.W     r14,r15                ; [] |19| R15=p+q+r+s
        ADD.W     r10,r15                ; [] |19| R15=p+q+r+s+t
        ADD.W     r11,r15                ; [] |19| R15=p+q+r+s+t+u
        MOV.W     r15,0(SP)              ; [] |19| aux1=p+q+r+s+t+u
        MOV.W     0(SP),r12              ; [] |20| R12 = Valor retornado
        ADD.W     #4,SP                  ; []
        POPM.W    #1,r10                 ; []
        RET                              ; []
```

Vamos agora propor uma nova função `soma` e sem as variáveis locais. Como `R12` é o registrador que deve retornar o valor inteiro, propomos que seja o destino de todas as somas. Assim, abaixo está a listagem da solução *assembly* proposta. Lembrar de que houve o `CALL`, assim, o ponteiro `SP` andou só 2 posições.

```
int soma (int p, int q, int r, int s, int t, int u){
    __asm (" ADD.W     R13,R12");     //R12=p+q
    __asm (" ADD.W     R14,R12");     //R12=p+q+r
    __asm (" ADD.W     R15,R12");     //R12=p+q+r+s
    __asm (" ADD.W     2(SP),R12");   //R12=p+q+r+s+t
    __asm (" ADD.W     4(SP),R12");   //R12=p+q+r+s+t+u
    // Não precisa incluir do retorno __asm (" RET");
}
```

Parecia tudo bem. Porém, após uma série de testes, foi detectado um problema que não conseguimos explicar. Ele só acontece quando, na chamada da função, se colocam

argumentos numéricos. Para deixar bem claro o problema, apresentamos uma tabela com diversos ensaios de argumentos.

Na tabela abaixo, para fins de comparação, temos na coluna da esquerda a preparação que o programa principal faz para chamar a função `soma` escrita em C e na da direita a preparação para chamar a função `soma` escrita em *assembly*. Note que nas duas últimas linhas, quando se usam argumentos numéricos, a preparação não é a correta e a rotina *assembly* dá erro. Note as linhas em cinza claro, que não foram colocadas e por isso os argumentos numéricos não foram passados. Pensamos que se trata de alguma ação do otimizador. Pesquisar mais.

Preparação para chamar a função `soma` escrita em C	**Preparação para chamar a função `soma` escrita em Assembly**

`g = soma(a, b, c, d, e, f);`
`MOV.W 12(SP),0(SP) ; e (pilha)`
`MOV.W 14(SP),2(SP) ; f (pilha)`
`MOV.W 4(SP),r12 ; a = R12`
`MOV.W 6(SP),r13 ; b = R13`
`MOV.W 8(SP),r14 ; c = R14`
`MOV.W 10(SP),r15 ; d = R15`
`CALL #soma ;`
	`g = soma(a, b, c, d, e, f);`
	`MOV.W 12(SP),0(SP) ; e (pilha)`
	`MOV.W 14(SP),2(SP) ; f (pilha)`
	`MOV.W 4(SP),r12 ; a = R12`
	`MOV.W 6(SP),r13 ; b = R13`
	`MOV.W 8(SP),r14 ; c = R14`
	`MOV.W 10(SP),r15 ; d = R15`
	`CALL #soma ;`

`g = soma(a, b, c, d, e, 10);`
`MOV.W 12(SP),0(SP) ; e (pilha)`
`MOV.W #10,2(SP) ; 10 (pilha)`
`MOV.W 4(SP),r12 ; a = R12`
`MOV.W 6(SP),r13 ; b = R13`
`MOV.W 8(SP),r14 ; c = R14`
`MOV.W 10(SP),r15 ; d = R15`
`CALL #soma ;`
	`g = soma(a, b, c, d, e, 10);`
	`MOV.W 12(SP),0(SP) ; e (pilha)`
	`MOV.W #10,2(SP) ; 10 (pilha)`
	`MOV.W 4(SP),r12 ; a = R12`
	`MOV.W 6(SP),r13 ; b = R13`
	`MOV.W 8(SP),r14 ; c = R14`
	`MOV.W 10(SP),r15 ; d = R15`
	`CALL #soma ;`

`g = soma(a, b, c, d, 20, 10);`
`MOV.W #20,0(SP) ; 20 (pilha)`
`MOV.W #10,2(SP) ; 10 (pilha)`
`MOV.W 4(SP),r12 ; a = R12`
`MOV.W 6(SP),r13 ; b = R13`
`MOV.W 8(SP),r14 ; c = R14`
`MOV.W 10(SP),r15 ; d = R15`
`CALL #soma ;`
	`g = soma(a, b, c, d, 20, 10);`
	`MOV.W #20,0(SP) ; 20 (pilha)`
	`MOV.W #10,2(SP) ; 10 (pilha)`
	`MOV.W 4(SP),r12 ; a = R12`
	`MOV.W 6(SP),r13 ; b = R13`
	`MOV.W 8(SP),r14 ; c = R14`
	`MOV.W 10(SP),r15 ; d = R15`
	`CALL #soma ;`

`g = soma(a, b, c, 30, 20, 10);`
`MOV.W #20,0(SP) ; 20 (pilha)`
`MOV.W #10,2(SP) ; 10 (pilha)`
`MOV.W 4(SP),r12 ; a = R12`
`MOV.W 6(SP),r13 ; c = R13`
`MOV.W 8(SP),r14 ; d = R14`
`MOV.W #30,r15 ; 30 = R15`
`CALL #soma ;`
	`g = soma(a, b, c, 30, 20, 10);`
	`MOV.W #20,0(SP) ; 20 (pilha)`
	`MOV.W #10,2(SP) ; 10 (pilha)`
	`MOV.W 4(SP),r12 ; a = R12`
	`MOV.W 6(SP),r13 ; c = R13`
	`MOV.W 8(SP),r14 ; d = R14`
	`MOV.W #30,r15 ; 30 = R15`
	`CALL #soma ;`

`g = soma(a, b, 40, 30, 20, 10);`
`MOV.W #20,0(SP) ; 20 (pilha)`
`MOV.W #10,2(SP) ; 30 (pilha)`
`MOV.W 4(SP),r12 ; a = R12`
`MOV.W 6(SP),r13 ; b = R13`
`MOV.W #40,r14 ; 40 = R14`
`MOV.W #30,r15 ; 30 = R15`
`CALL #soma ;`
	`g = soma(a, b, 40, 30, 20, 10);`
	`MOV.W #20,0(SP) ; 20 (pilha)`
	`MOV.W #10,2(SP) ; 10 (pilha)`
	`MOV.W 4(SP),r12 ; a = R12`
	`MOV.W 6(SP),r13 ; b = R13`
	`MOV.W #40,r14 ; 40 = R14`
	`MOV.W #30,r15 ; 30 = R15`
	`CALL #soma ;`

O leitor é convidado a verificar que quando se usa como argumento uma variável de tipo diferente, como mostrado no trecho abaixo, também causa problemas. Vamos então a uma série de propostas para evitar erros com o compilador.

```
int main(void){
    volatile int a=1,b=2,c=3,d=4,e=5,f=6,g=7;
    volatile char x=10;
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    g = soma(a, b, x, d, e, f);
    res1 = g;
    while(1);
    return 0;
}
```

Proposta 1a: Sempre use variáveis como argumentos

É claro que as variáveis usadas como argumentos devem ser declaradas com o modificador `volatile`. Isso ameniza bastante a interferência do compilador. Com esta regra, o programa *assembly* sugerido para a `soma` em sempre funciona. Se a função espera argumento de um tipo, não passe um tipo diferente, confiando na conversão. Por exemplo, usar como argumento da função `soma` uma variável `char`. Em alguns casos gera erros.

```
int main(void){
    volatile int a=1,b=2,c=3,d=4,e=5,f=6,g=7;
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    g = soma(a, b, c, d, e, f);
    res1 = g;
    while(1);
    return 0;
}
```

Proposta 1b: Use variáveis globais

Esta talvez seja a forma mais segura. Se a função em *assembly* sempre operar com variáveis globais, não há argumentos a serem passados. Novamente, recomendamos que essas variáveis globais sejam declaradas com o modificador `volatile`. A listagem abaixo apresenta um exemplo. A desvantagem é a perda de flexibilidade na chamada da função `soma`.

```
// Exp04.c
// Proposta 1b --> Variáveis globais
#include <msp430.h>

int soma (void);
volatile int a,b,c,d,e,f;
```

```
int main(void){
    volatile int g=7;
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    a=1; b=2; c=3; d=4; e=5; f=6; //Preparar para a soma
    g = soma();
    while(1);
    return 0;
}

// Função soma a+b+c+d+e+f (só variáveis globais)
int soma (void){
    __asm (" MOV.W     &a,R12");
    __asm (" ADD.W     &b,R12");
    __asm (" ADD.W     &c,R12");
    __asm (" ADD.W     &d,R12");
    __asm (" ADD.W     &e,R12");
    __asm (" ADD.W     &f,R12");
}
```

Proposta 1c: Encapsule a função para garantir os argumentos corretos

Nesta sugestão, usamos uma função para chamar a verdadeira função que queremos executar. Veja a listagem abaixo. O programa principal chama a função `soma` que por sua vez chama a função `soma_a` que está em *assembly*. Com essa técnica, conseguimos passar argumentos numéricos sem problemas, como na listagem. Porém, teremos problemas se misturarmos variáveis de tipos diferentes. Por exemplo colocar uma variável `char` entre os argumentos da `soma`.

O leitor é convidado a conferir a listagem *assembly* (não se esqueça de eliminar o *debug* simbólico, veja Figura C.10) e constatar que a função `soma`, na verdade não foi criada. O programa principal chama diretamente a função `soma_a`.

```
// Exp05.c
// Proposta 1c --> Encapsulamento
#include <msp430.h>

int soma (int x, int y, int z, int w, int t, int n);
int soma_a (int x, int y, int z, int w, int t, int n);

int main(void){
    volatile int a=1,b=2,c=3,d=4,e=5,f=6,g=7;
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    g = soma(6, 5, 4, 3, 2, c);
    while(1);
    return 0;
}

// Cápsula
```

```
int soma (int p, int q, int r, int s, int t, int u){
    return soma_a(p,q,r,s,t,u);
}

// Encapsulado
int soma_a (int x, int y, int z, int w, int t, int n){
    __asm (" ADD.W     R13,R12");
    __asm (" ADD.W     R14,R12");
    __asm (" ADD.W     R15,R12");
    __asm (" ADD.W     2(SP),R12");
    __asm (" ADD.W     4(SP),R12");
}
```

Proposta 1d: Encapsule a função e use ponte por variáveis globais

Esta parece ser uma sugestão bem flexível e muito segura. Vamos usar uma função para chamar a verdadeira função que queremos executar. Nesta função intermediária, vamos copiar todos os argumentos para variáveis globais que, por sua vez, são operadas pela função final. Isso garante que todos os argumentos estejam nos locais corretos e que se façam as conversões de tipo necessárias.

Na listagem abaixo, a função `soma` apenas transfere os argumentos para as variáveis globais. Então, a função `soma_a` apenas opera essas variáveis globais. Veja que agora podemos misturar variáveis com segurança. É importante indicar que a variável `char cc`, foi declarada como `signed`, do contrário ela seria usada sem sinal.

```
// Exp06.c
// Proposta 1c = Encapsulamento com ponte sobre globais
#include <msp430.h>

int soma (int x, int y, int z, int w, int t, int n);
int soma_a (void);
volatile int x1,x2,x3,x4,x5,x6;     //Globais para fazer ponte

int main(void){
    volatile int a=1,b=2,c=3,d=4,e=5,f=6,g=7;
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    signed char cc=-1;
    g = soma(-1, b, cc, d, e, f);
    while(1);
    return 0;
}

// Cápsula faz a ponte com variáveis globais
int soma (int p, int q, int r, int s, int t, int u){
    x1=p; x2=q; x3=r; x4=s; x5=t; x6=u;
    return soma_a();
}
```

```
// Encapsulado, opera somente as globais
int soma_a (void){
    __asm (" MOV.W     &x1,R12");
    __asm (" ADD.W     &x2,R12");
    __asm (" ADD.W     &x3,R12");
    __asm (" ADD.W     &x4,R12");
    __asm (" ADD.W     &x5,R12");
    __asm (" ADD.W     &x6,R12");
}
```

Proposta 2: Usar arquivo ".C" separado para Função que contém de C para Assembly

Agora a ideia é usar um arquivo separado para a função em C que vai trazer embutida as linhas de *assembly*. Como a função vai estar em um outro arquivo, o otimizador do CCS não tem como saber o que essa função faz, assim, ele arruma os argumentos na pilha da forma esperada. Essa parece ser uma excelente solução. A listagem abaixo ilustra o conceito. Apesar de não ser necessário nesse caso, fizemos uso de um arquivo cabeçalho só para indicar o protótipo da função. As listagens a seguir deixam a ideia bem clara.

Listagem do programa principal

```
// Exp07.c
// Proposta 2 = Arquivo (__asm) em separado

#include <msp430.h>
#include "soma.h"

int main(void){
    volatile int a=1,b=2,c=3,d=4,e=5,f=6,g=7;
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    signed char cc=-1;
    g = soma(-1, b, cc, 30, 20, 10);
    while(1);
    return 0;
}
```

Listagem do cabeçalho

```
//soma.h
// Exp07.c
// Proposta 2 = Arquivo (__asm) em separado

#ifndef SOMA_H_
#define SOMA_H_

int soma (int x, int y, int z, int w, int t, int n);
```

```
#endif /* SOMA_H_ */
```

Listagem da função `soma` com o código em *assembly*

```
// soma.c
// Exp07.c
// Proposta 2 = Arquivo (__asm) em separado

int soma (int x, int y, int z, int w, int t, int n){
    __asm (" ADD.W     R13,R12");
    __asm (" ADD.W     R14,R12");
    __asm (" ADD.W     R15,R12");
    __asm (" ADD.W     2(SP),R12");
    __asm (" ADD.W     4(SP),R12");
}
```

Proposta 3: Função assembly (.asm) em arquivo separado

Esta é a melhor solução de todas. Usamos um arquivo "*.asm*" para as funções escritas em *assembly*. Novamente, como a função vai estar em um outro arquivo, o otimizador do CCS não tem como saber o que essa função faz, assim, ele arruma os argumentos na pilha da forma esperada. Note que neste caso não precisamos mais usar a declaração `__asm` porque o arquivo todo está em *assembly*. O CCS se encarrega de montar o que estiver em assembly, compilar tudo o que estiver em C e depois usa o linker para criar o executável que será carregado na memória. No exemplo abaixo foi usado o modelo *small* tanto para código quanto para os dados.

O cuidado que se deve ter é indicar no arquivo ".asm" que a função é global e

- `.global soma` → indica que o `label` soma será exportado.
- `.sect ".text"` → orienta para o código ir para a área relocável.

Listagem do programa principal

```
// Exp08.c
// Proposta 3 Arquivo soma.asm em separado

#include <msp430.h>
#include "soma.h"

//extern int soma (int x, int y, int z, int w, int t, int n);

int main(void){
    volatile int a=1,b=2,c=3,d=4,e=5,f=6,g=7;
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    signed char cc=-1;
    g = soma(-1, b, cc, 30, 20, 10);
    while(1);
```

```
    return 0;
}
```

Listagem do cabeçalho

```
// soma.h
// Exp08.c
// Proposta 3 Arquivo soma.asm em separado

#ifndef SOMA_H_
#define SOMA_H_

extern int soma (int x, int y, int z, int w, int t, int n);

#endif /* SOMA_H_ */
```

Listagem da função `soma.asm` com o código em *assembly*

```
// soma.asm
// Exp08.c
// Proposta 3 Arquivo soma.asm em separado

      .global    soma      ; Símbolo a ser exportado
      .sect      ".text"   ; Código relocável

soma:
    ADD.W     R13,R12
    ADD.W     R14,R12
    ADD.W     R15,R12
    ADD.W     2(SP),R12
    ADD.W     4(SP),R12
    RET
```

As figuras a seguir indicam com criar arquivos fonte e cabeçalho dentro do CCS.

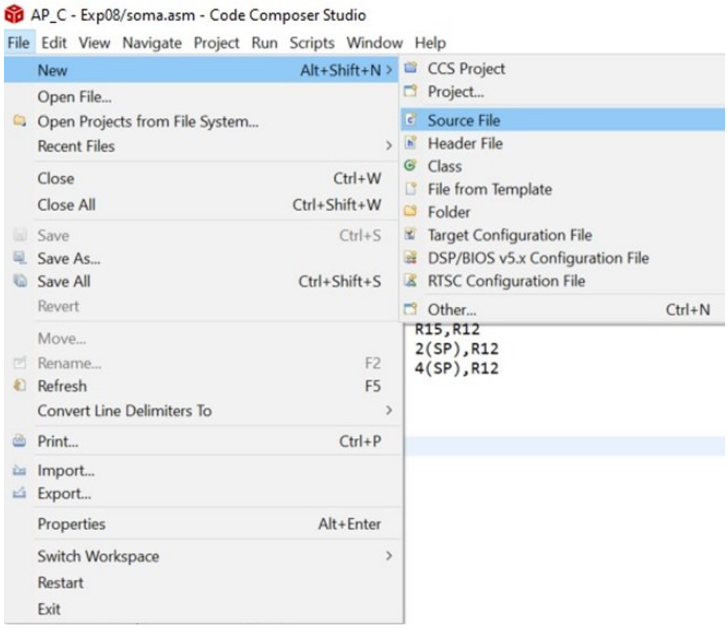

Figura C.13. Sequência para cria um arquivo fonte.

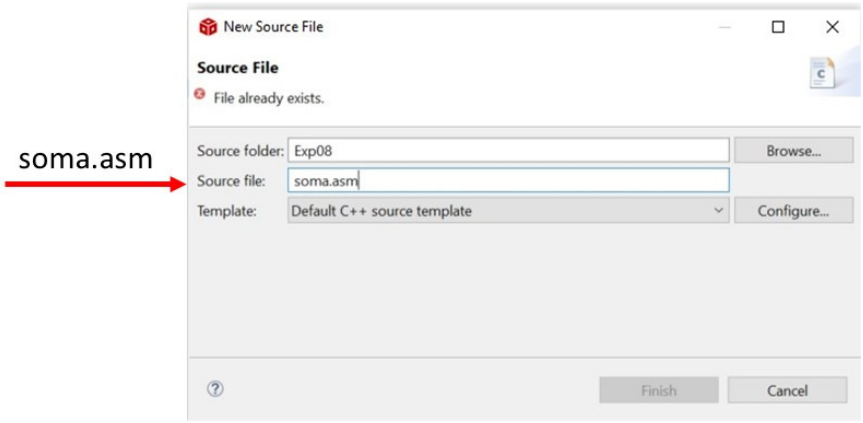

Figura C.14. Indicação do nome do arquivo fonte. É preciso incluir ".C" ou ".asm".

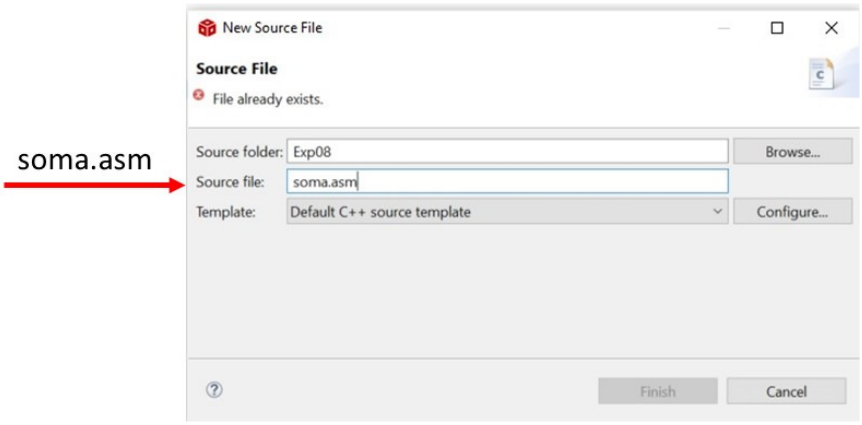

Figura C.15. Sequência para criar um arquivo cabeçalho.

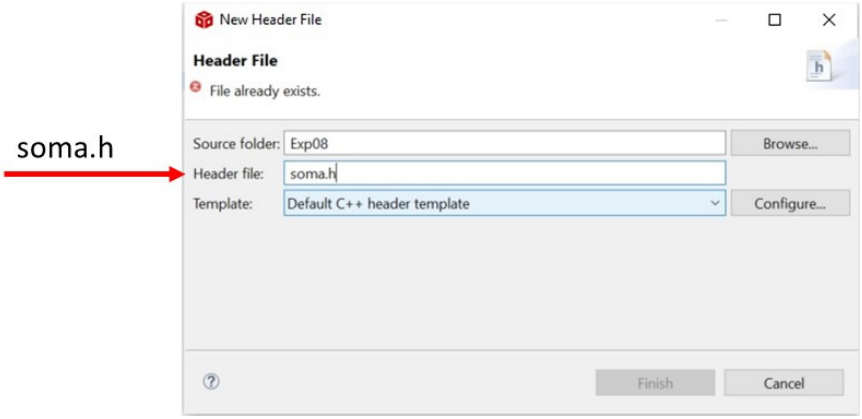

Figura C.16. Indicação do nome do arquivo cabeçalho. É preciso incluir ".h".

Pendente: Como habilitar o uso de funções de impressão, como o print
Parece que este é para salvar console durante uso do debug com printf
Project -> Properties -> Debug -> Misc/Other Optios
Default directory for File IO

C.12. Exercícios Resolvidos

ER C.1. Como primeiro exercício, vamos construir uma estrutura que é útil em diversos casos. É a fila circular. É comum o caso em que se recebem dados a uma taxa variável e muitas vezes não se consegue processar todos. Uma boa solução é construir em memória uma fila. Porém, uma fila no sentido mais comum, implicaria em um deslocamento de todos quando o primeiro elemento sair da fila. O que é proposto é andar com a marca de início

da fila, sendo que o início e o final desta fila se juntam. É o que se chama de Fila Circular. Ela já foi construída em assembly no ER 2.21. Ela será composta por um vetor de tamanho "n" e de três funções:

`char fila_poe(char dado);` → Coloca o dado na fila circular.

Retorna: TRUE se conseguiu colocar o dado na fila e
FALSE se a fila estava cheia.

`char fila_tira(char *pt);` → Retira um dado da fila circular e o coloca na posição apontada por `pt`.

Retorna: TRUE se conseguiu retirar o dado e
FALSE se a fila estava vazia.

`void fila_inic(void)` → inicializa os indexadores da fila.

Solução:

A fila circular é uma estrutura muito útil quando se trabalha com sistemas embarcados. Ela trabalha como um "amortecedor" e permite que se receba dados e se retire dados de forma não sincronizada. Um exemplo de uso é no teclado do PC. A cada ação no teclado, a BIOS coloca os códigos correspondentes às teclas acionadas numa fila circular e o Windows, à medida que tem tempo, retira esses códigos da fila. Como ilustrado na Figura C.17, a fila circular simplesmente consiste na alocação de um *buffer* em memória e no uso de dois indexadores (ponteiros):

- `pin` → ponteiro para colocar um elemento na fila e
- `pout` → ponteiro para retirar um elemento da fila.

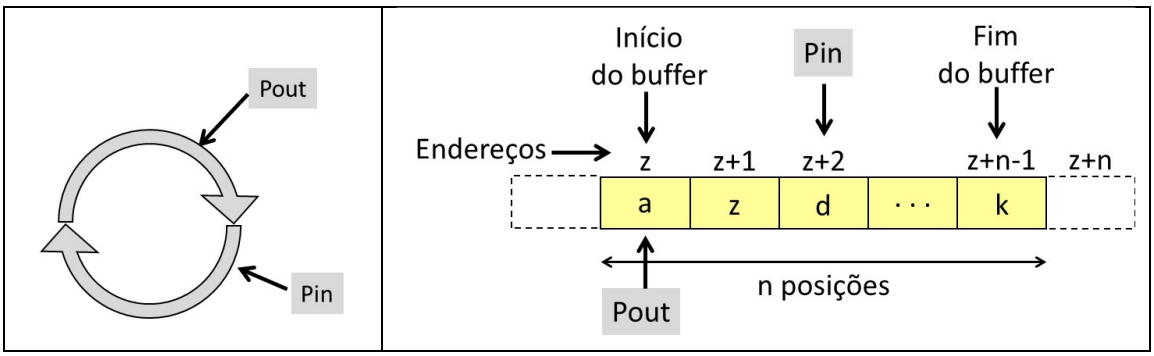

Figura C.17. Ilustração de uma Fila Circular.

A fila circular é construída com um vetor de tamanho "n" e o controle dos indexadores (`pin` e `pout`) é feito de forma a ter a ilusão de que as duas extremidades do vetor estão unidas, como ilustrado no lado esquerdo da figura acima.

É claro que um ponteiro nunca pode ultrapassar o outro. Na figura abaixo temos a caracterização de dois estados importantes, que são o de fila vazia e o de fila cheia. Isto

dispensa a necessidade de um contador para verificar se a fila está cheia. Por outro lado, se sacrifica uma posição da fila.
Fila vazia: (`pin = pout + 1`) `pin` está uma posição à direita de `pout` e
Fila cheia: (`pin = pout`) `pin` igual a `pout`.

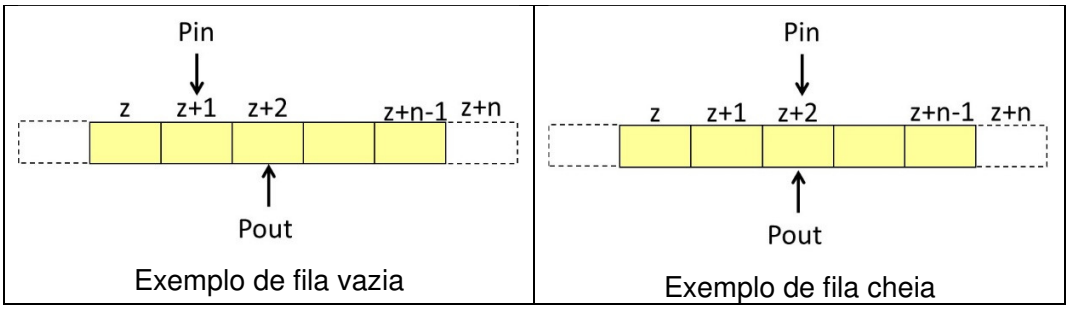

Figura C.18. Posição relativa dos ponteiros caracteriza os estados de fila vazia e fila cheia.

A regra para operação dos ponteiros e da fila é simples:

- **Põe** → se Pin ≠ Pout, então escrever o dado na posição Pin e incrementar Pin e
- **Tira** → se Pout ≠ Pin+1, então retirar o dado da fila e incrementar Pout.

A cada incremento ou comparação é preciso verificar se o ponteiro ultrapassou o final do *buffer*, situação em que ele deve voltar para o início, para assim tornar a fila circular. A Figura C.19 apresenta o fluxograma das funções `poe()` e `tira()`. Note que a cada incremento do ponteiro (`pin` ou `pout`) é necessário verificar se ele chegou ao final da fila. Caso isso aconteça, é necessário voltá-lo para o início da fila.

Logo a seguir está a listagem das funções e um programa para testar seu funcionamento. O leitor deverá fazer uso do CCS para ver a memória de dados e assim poder constatar o funcionamento dessas funções.

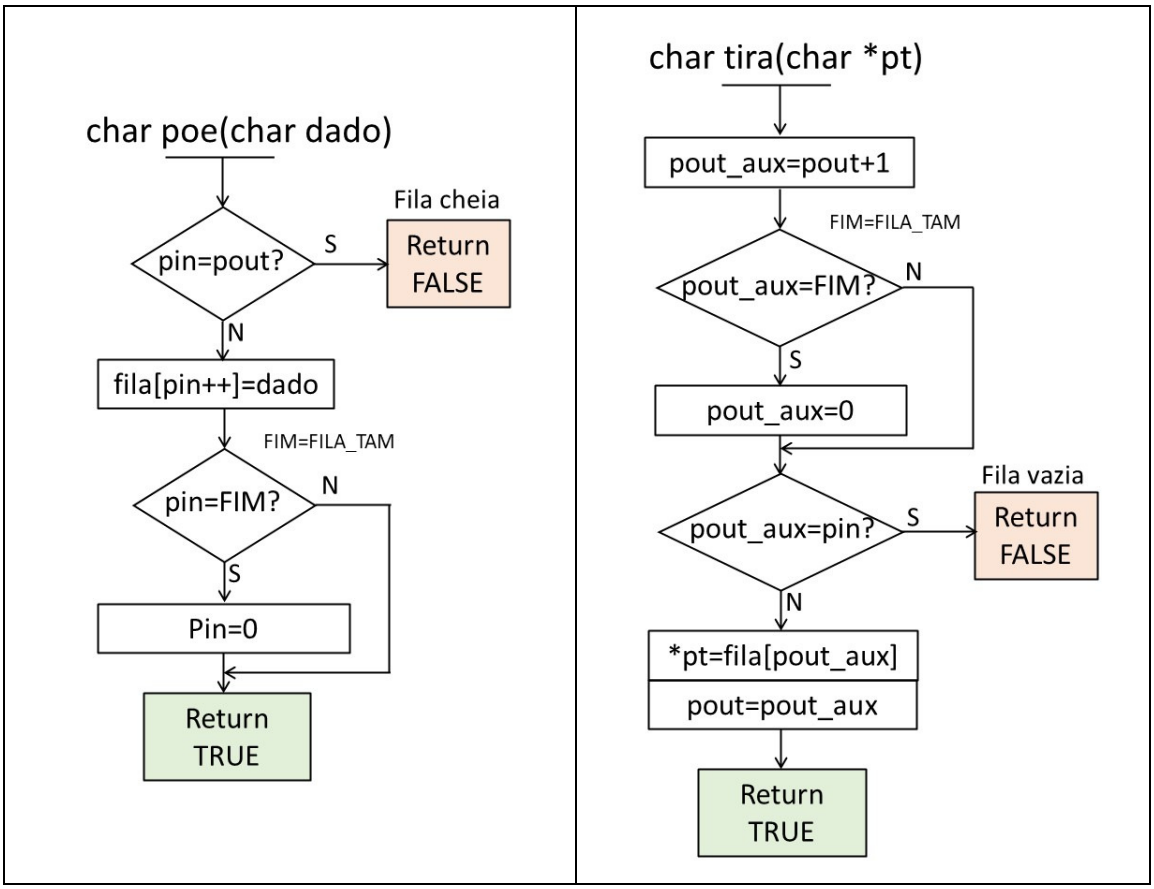

Figura C.19. Fluxograma das funções `poe()` e `tira()`.

Listagem do arquivo fila.h

```
// fila.h
// ER C.01 - Fila circular

#ifndef FILA_H_
#define FILA_H_

#define TRUE        1
#define FALSE       0
#define FILA_TAM   10    //Tamanho da fila

void fila_inic(void);
char fila_poe(char dado);
char fila_tira(char *dado);

volatile char fila[FILA_TAM];    //Vetor para fila
volatile unsigned int pin,pout;  //Indexadores da fila

#endif /* FILA_H_ */
```

Listagem do arquivo fila.c

```
// fila.c
// ER C.01 - Fila circular

#include "fila.h"

// Inicializar fila
void fila_inic(void){
    pin=1;
    pout=0;
}

// Colocar dado na fila
// Retorna TRUE se conseguiu colocar na fila
// Retorna FALSE se a fila está cheia
char fila_poe(char dado){
  if (pin == pout)  return FALSE;   //Fila cheia
  else{
      fila[pin++]=dado;             //Colar dado na fila
      if (pin==FILA_TAM)  pin=0;    //Fim da fila?
      return TRUE;
  }
}

// Retirar um dado da fila
// Retorna TRUE se conseguiu retirar *pt=dado
// Retorna FALSE se a fila está vazia
char fila_tira(char *pt){
  unsigned int pout_aux;
  pout_aux=pout+1;
  if (pout_aux==FILA_TAM)    pout_aux=0;    //Fim da fila
  if (pout_aux == pin)       return(FALSE); //Fila vazia
  else{
      *pt=fila[pout_aux];  //Retirar dado da fila
      pout=pout_aux;       //Atualizar ponteiro pout
      return TRUE;
  }
}
```

Listagem com sugestão de teste para a fila circular

```
// ER C.01
// Fila circular

#include <msp430.h>
#include "fila.h"

#define TRUE        1
```

```
#define FALSE       0

int main(void)
{
    volatile int x=0x55,y=0x55;
      WDTCTL = WDTPW | WDTHOLD;     // stop watchdog timer
      fila_inic();
      x=fila_tira(&y);
    x=fila_tira(&y);
    x=20;
      while( fila_poe(x++) == TRUE);
      x=fila_poe(1);
    x=fila_poe(2);
    while( fila_tira(&y) == TRUE);
    x=fila_tira(&y);
    x=fila_tira(&y);
      while(TRUE);
      return 0;
}
```

ER C.2. Um gerador de números aleatório deve ser capaz de gerar números dentro de uma certa faixa, sendo que cada número tem igual chance (distribuição uniforme) de "aparecer". Usar um processador para gerar números aleatórios é uma falácia, já que tudo que se faz com processadores é determinístico. Entretanto, é possível criar algoritmos que para alguns fins práticos, se aproximem de um verdadeiramente aleatório. Por isso, esses algoritmos são chamados de pseudo-aleatórios.

Sempre que se imagina um gerador aleatório, pensa-se na situação onde cada número tem igual chance de ser "sorteado". Este é o caso da distribuição de probabilidades plana, denominada de distribuição uniforme. É possível gerar números com outras distribuições de probabilidade, como a Gaussiana, com média e desvio-padrão específicos, como a distribuição de Poisson etc.

Voltemos ao exercício. Um algoritmo muito simples e eficiente para um gerador pseudo-aleatório é o "Algoritmo Congruencial Multiplicativo", listado abaixo.

1) Escolha dois parâmetros *m* e *d*;
2) Inicie com uma semente, denominado u_0;
3) Calcule u_1 = **módulo** [($m * u_0$) / d];
4) Repita passo 3 quantas vezes for necessário, e assim serão gerados os números pseudo-aleatórios u_1, u_2, u_3, ...

Observação: como o leitor já deve saber, o operador **módulo** é o resto da divisão inteira, exemplo: Módulo (20 / 7) = 6

Vamos a pedido: construa um pacote de funções, denominado “aleat”, para gerar números aleatórios. Disponibilize as seguintes funções

- `void rrand_inic(void);` → Inicializa o gerador para com os valores sugeridos no arquivo cabeçalho `aleat.h`.
- `void rrand_inic_seed(unsigned int seed);` → Inicializa o gerador para com os valores sugeridos no cabeçalho `aleat.h`., sendo que a semente é o valor indicado pelo usuário.
- `void rrand_inic_tudo(unsigned int seed, unsigned int div, unsigned int mult);` → Inicializa o gerador para com os valores indicados pelo usuário.
- `unsigned int rrand(void);` → Gera um número aleatório.
- `unsigned int rrand_fx(unsigned int faixa);` → Gera um número aleatório dentro da faixa de `0` até `faixa`.

Solução:
É preciso discutir um pouco sobre o algoritmo gerador e a escolha dos parâmetros *m* e *d*. Vamos a três observações.

1) O algoritmo funciona bem quando esses dois números *m* e *d* são primos. Para facilitar a escolha, a tabela abaixo lista os 100 primeiros números primos. Ela foi obtida com a solução do ER C.10, que constrói o Crivo de Eratóstenes. Evite usar *m = 3*.

2) Um ponto a prestar atenção é sobre o limite do cálculo realizado. Veja que as rotinas fazem uso de variáveis inteiras sem sinal, ou seja, a faixa delas é de 0 até 65.535. Por isso, o produto de *m* e *d* deve ser inferior a esse limite (*m x d < 65.535*).

3) A variável pseudo-aleatório gerada (u_n) sempre será menor que *d* (*0 < u_n < d*) e, por razões óbvias, ela nunca será igual a zero. Essa faixa pode não ser adequada para o usuário.

4) Para gerar números pseudo-aleatórios dentro de uma faixa especificada pelo usuário, por exemplo, entre 0 e LIMITE, escolhemos como valor para *d* um primo imediatamente acima desse LIMITE. Depois, adicionamos um controle muito simples, como mostrado abaixo. Veja na listagem a rotina `rrand_fx()`. Ela faz uso do recurso `do - while`.

a) u_n `= rrand();`
b) u_n `> LIMITE +1?`
b.1) Caso positivo, gerar novo número,
b.2) Caso negativo, retornar $u_n - 1$.

Tabela C.6 Os 100 primeiros números primos

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1→10	2	3	5	7	11	13	17	19	23	29
21→30	31	37	41	43	47	53	59	61	67	71
31→40	73	79	83	89	97	101	103	107	109	113
41→50	127	131	137	139	149	151	157	163	167	173

51→60	179	181	191	193	197	199	211	223	227	229
61→70	233	239	241	251	257	263	269	271	277	281
71→80	283	293	307	311	313	317	331	337	347	349
81→90	353	359	367	373	379	383	389	397	401	409
91→100	419	421	431	433	439	443	449	457	461	463

A solução é muito simples e está apresentada nas duas próximas listagens. Usamos os nomes com a letra "r" dobrada (`rrand`) para evitar confusão com o gerador `rand()`, normalmente disponível nos compiladores C. Pesquisar se o CCS oferece essa função.

Listagem com o arquivo cabeçalho do gerador de números aleatórios

```
// aleat.h
// ER C.2
// Gerador de números aleatórios

#ifndef ALEAT_H_
#define ALEAT_H_

#define ALEAT_SUGEST_M   53  //m sugerido
#define ALEAT_SUGEST_D   109 //d sugerido
#define ALEAT_SUGEST_U   13  //u sugerido (semente)

unsigned int rrand(void);
unsigned int rrand_fx(unsigned int faixa);
void rrand_inic(void);
void rrand_inic_seed (unsigned int seed);
void rrand_inic_tudo(unsigned int seed, unsigned int div, unsigned int
mult);

unsigned int rrand_m ;  //multiplicador
unsigned int rrand_d;   //divisor
unsigned int rrand_u;   //semente

#endif /* ALEAT_H_ */
```

Listagem das funções do gerador de números aleatórios

```
// aleat.c
// ER C.2
// Funções para o gerador de números aleatórios

#include "aleat.h"
```

```
// Gerar um número aleatório (zero nunca é gerado)
// Faixa do número n gerado: 0 < n < rrand_d
unsigned int rrand(void){
    rrand_u= (rrand_m * rrand_u) % rrand_d;
    return rrand_u;
}

// Gerar um número aleatório dentro de 0 --> faixa
// Faixa do número n gerado: 0 <= n <= fx
unsigned int rrand_fx(unsigned int fx){
    do rrand_u= (rrand_m * rrand_u) % rrand_d;
    while (rrand_u>fx+1);
    return rrand_u-1;
}

// Inicializar os parâmetros por omissão
void rrand_inic(void){
    rrand_u = ALEAT_SUGEST_U;  //semente
    rrand_d = ALEAT_SUGEST_D;  //divisor
    rrand_m = ALEAT_SUGEST_M;  //multiplicador
}

// Inicializar a semente
void rrand_inic_seed(unsigned int semente){
    rrand_u = semente;         //semente
    rrand_d = ALEAT_SUGEST_D;  //divisor
    rrand_m = ALEAT_SUGEST_M;  //multiplicador
}

// Inicializar todos os parâmetros do gerador
void rrand_inic_tudo(unsigned int seed,
                     unsigned int div,
                     unsigned int mult){
    rrand_u = seed;   //semente
    rrand_d = div;    //divisor
    rrand_m = mult;   //multiplicador
}
```

Para finalizar, apresentamos um pequeno programa para testar o gerador aleatório. Veja que o programa cria o vetor `teste` como variável global, assim, ele vai ser criado a partir do endereço 0x2400. Com o *Memory Browser*, podemos facilmente examiná-lo. A principal dúvida é se o gerar oferece distribuição de probabilidades uniforme. Faça vários ensaios com este programa, apenas cuidado para não ultrapassar o limite do vetor `teste`.

Listagem com uma sugestão para testar o gerador aleatório

```
// ER C.2
```

```
// Gerador de números aleatórios

#include <msp430.h>
#include "aleat.h"

volatile char teste[110];

int main(void)
{
    volatile unsigned int i,x;
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    rrand_inic_seed(60000);
      for (i=0; i<1000; i++){
          x=rrand();
        teste[x]++;
      }
      while (1);
      return 0;
}
```

ER C.3. A intenção deste exercício é forçar o uso de *strings*, de vetor de ponteiros, da função malloc e ainda empregar do gerador pseudo-aleatório preparado no exercício anterior. É pedido para construir três vetores, cada um com uma certa quantidade de palavras e um programa para sortear palavras de cada vetor e montar uma frase em memória.

Solução:
Vamos começar imaginando os três vetores de onde as palavras serão sorteadas. Será sorteada um elemento de cada vetor e uma frase será formada na memória.

Tabela C.7. Sugestão dos vetores para compor as frases sorteadas

Nº	Vetor1	Vetor2	Vetor3
1	Amanhã	levarei	seu carro
2	Segunda	roubarei	sua cama
3	De tarde	devolverei	seu cadáver
4	À meia noite	esconderei	sua alma
5	Depois de amanhã	encarnarei	seu penico
6	Um dia	queimarei	sua bicicleta
7	Ainda	desmontarei	seu código

A listagem solução está apresentada logo adiante. Vamos comentar os pontos importantes. Ao invés de usar matrizes, orientamos o compilador para colocar as palavras na memória de programa e indicar no vetor de ponteiros (`v1`, `v2` e `v3`), o início de cada uma. Note que não precisamos indicar o tamanho dos vetores, pois fornecemos usa inicialização.

Como vamos reunir palavras de cada vetor, é necessário determinar o tamanho da maior string de cada vetor. Somando esses tamanhos e ainda adicionando 3 (2 espaços e um zero final), chegamos ao tamanho máximo possível da frase. Então criamos um vetor com esse tamanho usando a alocação dinâmica de memória: `malloc`.

Após isso, ficamos num laço infinito, sorteando palavras de cada vetor e montando frases. O leitor deve usar o Memory Browser do CCS para examinar os caracters à partir de 0x2400 (provavelmente). É interessante estudar as funções `str_cpy()`, `str_len()` e `str_max()`.

Aqui é importante mostrar uma armadilha de C. Experimente substituir a função `str_cpy()` pela sugestão abaixo. Ela não vai funcionar, vai acontecer erro em uma posição. Só examinando o *assembly* gerado pelo compilador é que poderemos entender o porquê.

```
// Versão que não funciona
int str_cpy(char *ft, char *dst){
    unsigned int i=0;
    while( ft[i] != '\0'){
        dst[i]=ft[i++];
        //i++;
    }
    dst[i]=ft[i];   //Copiar o zero
    return i;
}
```

Curiosidade: o leitor consegue descobrir quem o programa abaixo homenageia?

Listagem do programa solução. O leitor deve copiar os arquivos “aleat.h” e “aleat.c” do exercício anterior.

```
//ERCp03
//
// Ilustrar o uso de strings e números aleatórios
#include <msp430.h>
#include "aleat.h"

#define TRUE    1
#define FALSE   0
#define NULL    0
#define TOTAL   7   //Quantidade de frases em cada vetor

int str_cpy(char *ft, char *dst);
```

```
int str_len(char *ft);
int str_max(char *ft[]);

char *v1[]={"Amanha", "Segunda", "De tarde", "A meia-noite",
            "Depois de amanha", "Um dia", "Ainda"};
char *v2[]={"levarei", "roubarei", "devolverei", "esconderei",
            "encarnarei", "queimarei", "desmontarei"};
char *v3[]={"seu carro", "sua cama", "seu cadaver", "sua alma",
            "seu penico", "sua bicicleta", "seu codigo"};

int main(void)
{
    char *msg;
    volatile unsigned int x,i,max,pos;
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    rrand_inic_tudo(5,13,11);   //Inicializar gerador

    //Calcular tamanho da maior frase
    max=str_max(v1);
    max+=str_max(v2);
    max+=str_max(v3);
    max+=3;     //Inserir 2 espaços mais o zero final

      msg=malloc(max);    //Alocar tamanho necessário
    if (msg == NULL)
        while(TRUE);    //Erro na alocação de memória

    while (TRUE){

        // Apagar a mensagem anterior
        for (i=0; i<max; i++)   msg[i]=' ';

        pos=0;
        x=rrand_fx(6);      //Sortear vetor 1
        pos+=str_cpy(v1[x],&msg[0]);
        msg[pos++]=' ';

        x=rrand_fx(6);      //Sortear vetor 2
        pos+=str_cpy(v2[x],&msg[pos]);
        msg[pos++]=' ';

        x=rrand_fx(6);      //Sortear vetor 3
        pos+=str_cpy(v3[x],&msg[pos]);
    }
      return 0;
}

// Copiar uma string (inclusive o zero final)
// Retorna qtd de caracteres copiados
```

```
int str_cpy(char *ft, char *dst){
    unsigned int i=0;
    while( ft[i] != '\0'){
        dst[i]=ft[i];
        i++;
    }
    dst[i]=ft[i];   //Copiar o zero
    return i;
}

// Determinar o tamanho de uma string
// Não inclui o zero final
int str_len(char *ft){
    unsigned int i=0;
    while( ft[i] != '\0')
        i++;
    return i;
}

// Maior tamanho das strings de um vetor de ponteiros
int str_max(char *ft[]){
    int max,aux,i;
    max=str_len(ft[1]);
    for (i=1; i<TOTAL; i++){
        if ( max < (aux=str_len(ft[i])) )
            max=aux;
    }
    return max;
}
```

ER C.4. ????
Solução:

ER C.5. ????
Solução:

ER C.6. ?????
Solução:

ER C.7. ?????
Solução:

ER C.8. ?????
Solução:

ER C.9. ?????
Solução:

ER C.10. Aqui neste exercício vamos tentar comparar o tempo gasto para executar uma rotina escrita em C com o tempo gasto por sua equivalente escrita em *assembly*. Vamos medir esses tempos usando o cronômetro proposto no ER 7.5. Como desculpa para o teste vamos usar o Crivo de Eratóstenes que é usado encontrar números primos. Apresentamos abaixo o algoritmo deste crivo para encontrar todos os números primos de 2 até N.
1) Crie uma lista com os números a serem testados, de 1 até N.
2) Pule o número 1, por razões óbvias.
3) O próximo número da lista é primo. Marque-o e elimine todos os seus múltiplos.
4) Repita o passo 3 até chegar ao final da lista.
5) Todos os números que sobraram na lista são primos.

Observação: em seu crivo, Eratóstenes previa que a parada poderia acontecer quando se chegasse ao número igual à $\sqrt{N}$ arredondada para baixo. Aqui não faremos isso. Temos a intenção de deixar o algoritmo mais longo e assim, evidenciar as diferenças entre C e *assembly*.

Para construir esse crivo em C, vamos criar a lista com os números na forma do vetor `nr` de tamanho N+1. Ele terá posições numeradas de 0 até N. Iniciamos escrevendo 1 em todas as posições. Quando quisermos retirar um número da lista, marcamos sua posição com 0. A figura abaixo ilustra a ideia e especifica a forma final da lista de primos, onde a posição 0 indica a quantidade de números primos encontrados e, da posição 1 em diante, listamos os primos. As demais posições devem ser marcadas com 0.

Vetor `nr` antes do crivo

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Vetor `nr` após o crivo

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0

Vetor `nr` no formato especificado como resposta

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
8	2	3	5	7	11	13	17	19	0	0	0	0	0	0	0	0	0	0	0	0

Figura C.20. Etapas intermediárias da aplicação do Crivo de Eratóstenes

Sobre Eratóstenes: Grego, nascido em 276 A.C, grande matemático, geógrafo, astrônomo etc. Nessa época ele já sabia que a Terra era redonda (já era um terrabolista, para espanto dos atuais terraplanistas) e mediu o tamanho de sua circunferência. Foi "diretor" da biblioteca de Alexandria. Seu método para encontrar números primos é muito usado para fazer *benchmark* de computadores.

Solução: A solução cuja listagem está logo abaixo é bastante simples. A função `crivo_c(nrc,N)` está escrito em C e gera sua solução no vetor `nrc`. Já a função

`crivo_a(nra,N)` está escrita em assembly em um arquivo separado e gera solução no vetor `nra`. Esta solução, por estar em assembly, é um pouco mais trabalhosa.

As variáveis `tpa` e `tpc` indicam o tempo gasto por cada função. Para o caso de N = 1.000, os valores encontrados foram: `tpc` = 57.456 e `tpa` = 49.812., que corresponde a um ganho de velocidade de 15%. Não foi tão grande quanto esperávamos. Em todo o caso, é um excelente exemplo de como mesclar *assembly* com C.

Listagem do programa solução. O leitor deve copiar os arquivos “crono.h” e “crono.c” do ER 7.5.

```
// ER C.10
//Crivo de Eratóstenes
// Usar modelo small para code e data

#include <msp430.h>
#include "crono.h"

#define N 50    //Limite para a busca < 65.535
void crivo_c(int *vet, unsigned int lim);   //C
extern void crivo_a(int *vet, unsigned int lim);   //Assembly


int nrc[N+1];    //Vetor global para a função em C
int nra[N+1];    //Vetor global para a função em assembly

int main(void)
{
    unsigned int i;
    volatile long tpc,tpa;
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    for (i=0; i<N+1; i++){
        nrc[i]=0x55;
        nra[i]=0x55;
    }
    crono_inic();
    //
    crono_zera();
    crono_start();
    crivo_c(nrc, N);    //Crivo em C
    crono_stop();
    tpc=crono_ler();
    //
    crono_zera();
    crono_start();
    crivo_a(nra, N);    //Crivo em Assembly
    crono_stop();
    tpa=crono_ler();
      while (1);
```

```
      return 0;
}

// Função para o Crivo em C
void crivo_c(int *vet, unsigned int lim){
    unsigned int i,j,passo;
    for (i=2; i<=lim; i++)  vet[i]=1;
    for (i=2; i<=lim; i++){
        if (vet[i]==1){
            passo = i;
            for (j=i+passo; j<=lim; j+=passo)
                vet[j]=0;
        }
    }
    vet[0]=0;
    i=1;
    j=2;
    for (j=2; j<=lim;j++){
        if (vet[j]==1){
            vet[i++]=j;
            vet[j]=0;
            vet[0]++;
        }
    }
}
```

Listagem com a versão em assembly do Crivo de Eratóstenes

```
; crivo.asm para o ER C.10
; crivo de Eratóstenes em assembly
; Usar modelo small para code e data

      .global   crivo_a    ; Simbolo a ser exportado
      .sect     ".text"     ; Código relocável

; Argumentos passados para a função
; R12 = endereço de início do vetor nr[N+1]
; R13 = N = tamanho do vetor
crivo_a:
            PUSH  R5                ;(Salvar R5) Ponteiro principal
            PUSH  R6                ;(Salvar R6) Último endereço do vetor
            PUSH  R7                ;(Salvar R7) Passo
            PUSH  R8                ;(Salvar R8) Ponteiro auxilizar
;
; Marcar com 1 posições nr[2] até nr[N]
            MOV         R12,R5      ;R5=&nr[0]
            MOV         R13,R6      ;R6=N
            ADD         R6,R6       ;R6=2N
```

```
            ADD         R12,R6      ;R6 = &nr[N]
            ADD         #4,R5       ;R5 = &nr[2]
LB1:        MOV         #1,0(R5)    ;@R5=1
            INCD        R5          ;Próximo endereço
            CMP         R5,R6       ;R6-R5 (comparar)
            JHS         LB1         ;Se R6>=R5, repetir (LB1)
;
; Marcar com 0 os múltiplos
            MOV         #0,0(R12)   ;Contador de primos
            MOV         R12,R5      ;R5=&nr[0]
            INCD        R5          ;R5=&nr[1]
            MOV         #1,R7       ;R7=1, PASSO
LB2:        INC         R7          ;R7 = próximo número
            INCD        R5          ;R5 avança próxima posição
            CMP         R5,R6       ;R6-R5 (comparar)
            JLO         LB4         ;Se R6<R5, sair do laço
            CMP         #1,0(R5)    ;@R5=1?
            JNZ         LB2         ;Se @R5=0, próxima posição (LB2)
            INC         0(R12)      ;Inc contador de primos
            MOV         R7,R8       ;R8=número
            ADD         R8,R8       ;R8=2*número
            ADD         R5,R8       ;@R8=primeiro múltiplo
LB3:        CMP         R8,R6       ;R6-R8 (comparar)
            JLO         LB2         ;Se R8>=R6, repetir laço
            MOV         #0,0(R8)    ;@R8=0, marcar múltiplo
            ADD         R7,R8       ;R8=R8+número
            ADD         R7,R8       ;R8=R8+2*número
            JMP         LB3         ;Repetir o laço
;
; Puxar todos os primos para a esquerda
; Agora já podemos destruir R12
LB4:        INCD        R12         ;R12=&NR[1]
            MOV         R12,R8      ;R8=&nr[1]
            MOV         #1,R7       ;R7=1, número
LB5:        INC         R7          ;R7= próximo numero
            INCD        R8          ;@R8=próxma posição
            CMP         R8,R6       ;R6-R8 (comparar)
            JLO         LB6         ;Se R6<R8, sair do laço
            CMP         #1,0(R8)    ;@R8=1?
            JNZ         LB5         ;Se @R8=0, repetir laço
            MOV         #0,0(R8)    ;Marcar @R8=0
            MOV         R7,0(R12)   ;@R12=primo
            INCD        R12         ;R12=próxima posição
            JMP         LB5         ;Repetir laço

LB6:        POP         R8              ;Restaurar R8
            POP         R7              ;Restaurar R7
            POP         R6              ;Restaurar R6
            POP         R5              ;Restaurar R5
```

```
        RET                             ;Retornar
```

---------- Vai lá para o cap 5 de GPIO e C ----------

ER 5.1. Como o leitor já percebeu, estamos toda hora acendendo e apagando os leds. Esse exercício propõe uma série de funções para controlar esses leds de uma forma mais simples. Essas funções deverão estar em um arquivo em separado. Crie as funções:

`void led_vm_config(void)` → configurar o led vermelho
`void led_vm (void)` → apaga o led vermelho
`void led_VM (void)` → acende o led vermelho
`void led_Vm (void)` → inverte o estado do led vermelho
`void led_vm_config(void)` → configurar o led vermelho
`void led_vd (void)` → apaga o led vermelho
`void led_VD (void)` → acende o led vermelho
`void led_Vd (void)` → inverte o estado do led vermelho
`void leds_num (int num)` → Cria um contador binário com os leds

Depois, proponha uma forma para comprovar o funcionamento das funções criadas.

Solução: Vamos estruturar as funções em dois arquivos, um com os códigos e outro com o cabeçalho.

criar dois arquivos paPensando numa solução para "tocar" músicas, podemos imaginar que a função básica seria uma capaz de gerar na saída a onda quadrada na frequência solicitada,