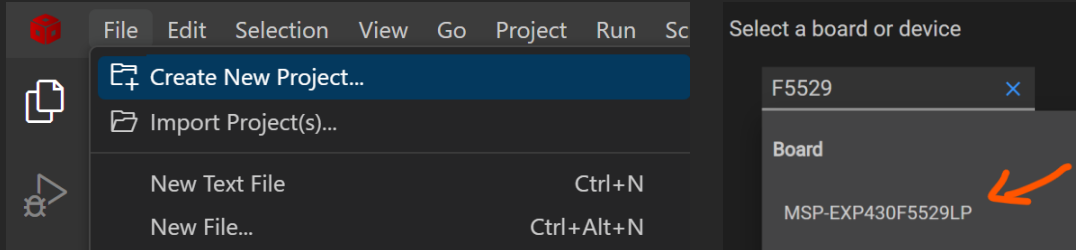


Módulo 0. Introdução ao CCS

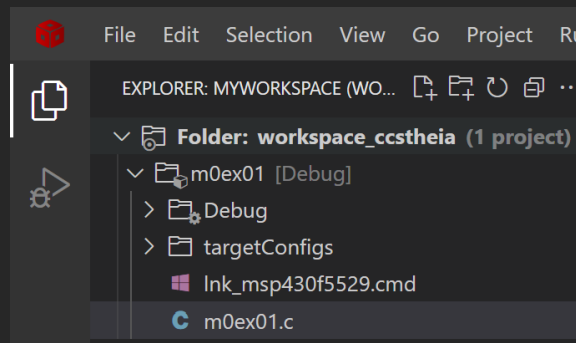
O Code Composer Studio é uma ferramenta de desenvolvimento baseada no Visual Studio Code. Ele vem com ferramentas para compilar, programar e depurar código do MSP430. Para criar um projeto, clique em *File -> Create New Project*, digite “F5529” no filtro e escolha a placa MSP-EXP430F5529LP ou o dispositivo MSP430F5529.

Figura 1 – Criar o projeto



Escolha o projeto exemplo **MSP430F55xx_1.c** que faz o LED piscar. Renomeie a pasta do projeto criado e seu arquivo fonte (apertando **F2** ou clicando com o botão direito → Renomear). Sugestão: m0ex01 e m0ex01.c, respectivamente, como na Figura 2.

Figura 2 – Estrutura do projeto



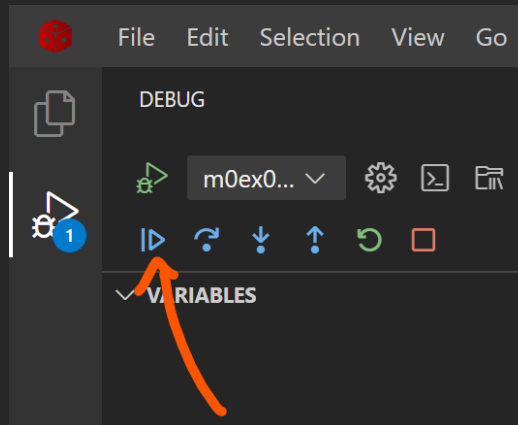
Copie o código abaixo no arquivo **m0ex01.c**. No código, a linha 5 interrompe o funcionamento do timer watchdog para que ele não reinicie o processador. A linha 6 configura o pino P1.0 como saída. Este pino está conectado ao LED vermelho. O laço da linha 8 faz o LED piscar a cada segundo.

Figura 3 – Código exemplo (copiar)

```
m0ex01.c X
1  #include <msp430.h>
2
3  void main()
4  {
5      WDTCTL = WDTPW | WDTHOLD;           // Trava o Watchdog
6      P1DIR |= BIT0;                     // P1.0 é uma saída
7
8      while(1)                           // Loop infinito
9      {
10         P1OUT ^= BIT0;                   // Alterna o estado do LED
11         __delay_cycles(1000000);         // Espera 1 segundo
12     }
13 }
```

Aperte **CTRL+B** para compilar o projeto. Um arquivo com extensão *.out será gerado na pasta “Debug”. Entre na depuração clicando em *Run* → *Debug* (**F5**). Aperte o *play* ▶, ou use o atalho **F5 novamente** e veja o programa executando (piscando o LED). Quando estiver satisfeito com o resultado, aperte *stop* □, ou use as teclas **SHIFT+F5**.

Figura 4 – Modo de depuração (F5 x 2)



Sobre o watchdog

O watchdog é um temporizador que é capaz de resetar o processador. Ele é um mecanismo de segurança para proteger o sistema contra falhas que possam travar o programa. Um programa bem projetado deve resetar o watchdog periodicamente para ele não contar até o máximo, ocasião em que gera um reset e impede nosso programa de avançar. O watchdog só será apresentado no módulo 2. Até lá, vamos sempre desativá-lo. Caso queira retirar a linha que zera o watchdog do seu programa principal, você pode mover a linha para uma função de pré inicialização, que é executada antes da main. Essa função pode inclusive ficar em outro arquivo, como boot.c ou init.c. É possível misturar arquivos em C e assembly num mesmo projeto.

Figura 5 – Função de inicialização

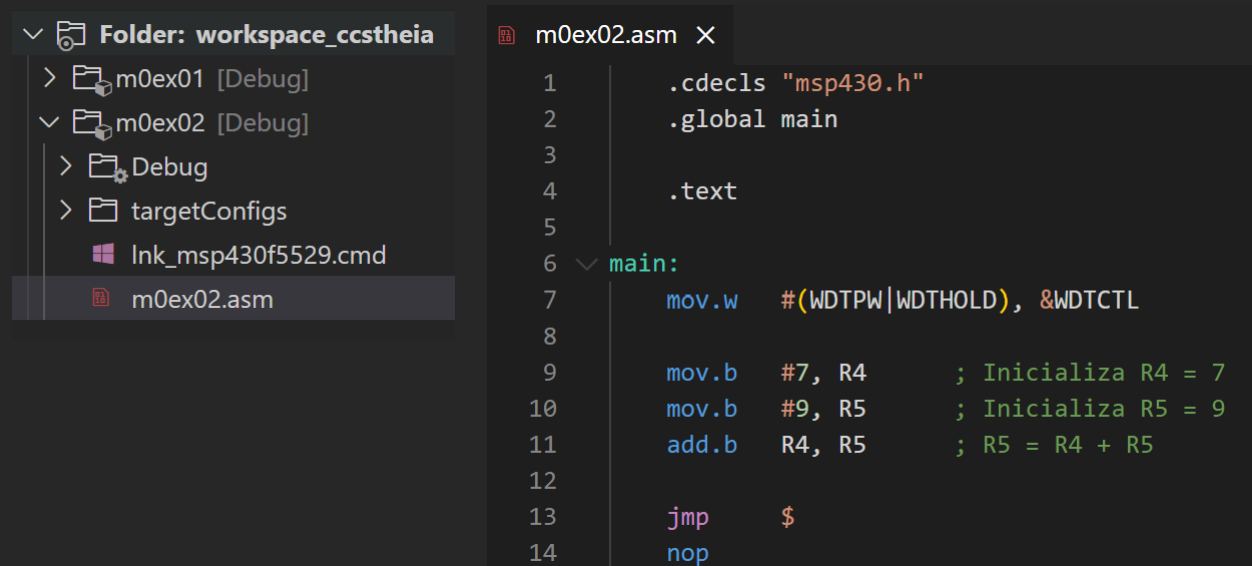
```
init.c X
1  #include <msp430.h>
2
3  void _system_pre_init()
4  {
5      WDTCTL = WDTPW | WDTHOLD;
6  }
7
```

Este é um passo opcional. O resto deste módulo considerará que essa etapa não foi realizada.

Assembly

No primeiro módulo do curso iremos trabalhar com a linguagem de programação assembly para entender os detalhes da arquitetura do processador do MSP430. Copie o projeto m0ex01 para uma nova pasta m0ex02 (usando CTRL+C, CTRL+V) e altere o nome do arquivo fonte m0ex01.c para **m0ex02.asm** (note a extensão *.asm).

Figura 6 – Código em assembly



Caso queira obter o realce de sintaxe de Assembly, instale a extensão MSP430 Assembly. As instruções estão no final deste roteiro.

Copie o código da Figura 6 no arquivo *m0ex02.asm*. Breve explicação:

- A linha 1 importa os símbolos do cabeçalho msp430.h (como P1DIR e WDTPW).
- A linha 2 exporta o rótulo “main” para o ligador saber onde começar o programa.
- A linha 4 indica que o código abaixo vai para a memória flash e não para a RAM.
- A linha 7 interrompe o funcionamento do watchdog para ele não resetar o processador.
- As linhas 9, 10 e 11, realizam a inicialização dos registros R4 e R5, somando-os ao final.
- As linhas 13 e 14 travam o programa num laço infinito.

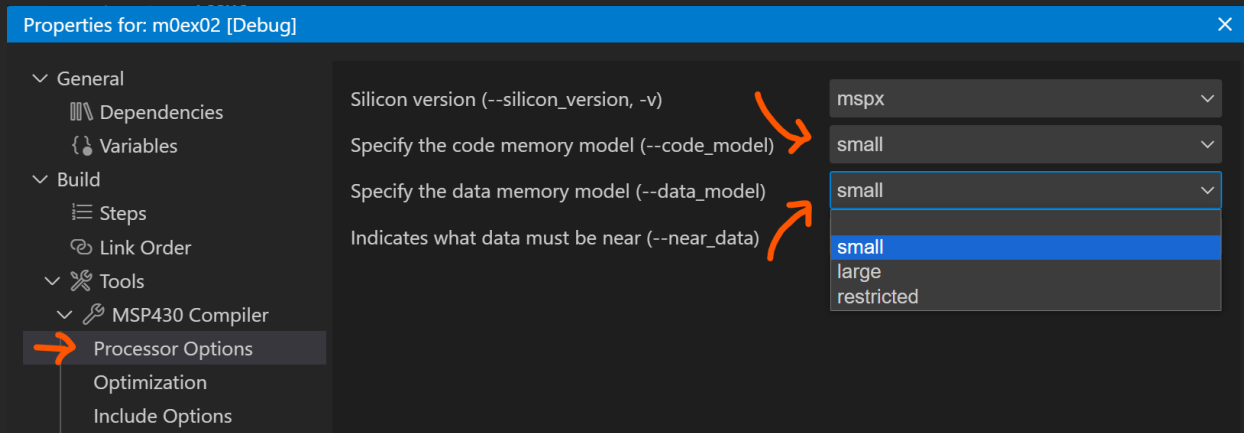
É importante notar que **as instruções são indentadas**, ou sejam, precisam de um espaçamento no início da linha. Todo símbolo no início da linha é reconhecido com um rótulo (como main, por exemplo). O símbolo \$ significa “a linha atual” e a instrução nop (no operation) não faz nada de útil, apenas protege o programa contra um bug de hardware (CPU40).

Os comentários iniciam com ponto-e-vírgula. Mantenha o seu código organizado, alinhando as instruções, operandos e comentários.

Em assembly, as instruções são escritas pelos seus mnemônicos (abreviação cujo som lembra a palavra, como “mov” para move, “jmp” para jump). Note que cada instrução tem um modificador “.b” (de byte) ou “.w” (de word). Ele indica se a instrução opera em 8 bits (byte) ou 16 (word). Na ausência desses modificadores, o padrão é uma operação em words de 16 bits.

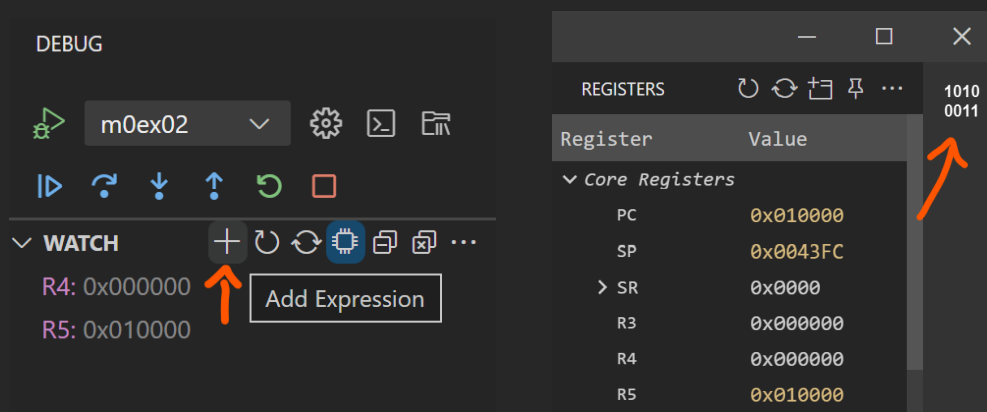
No nosso curso, iremos limitar o alcance da memória para cobrir apenas os primeiros 16 bits endereçáveis. Para isso, abra as configurações do projeto, clicando com o botão direito na pasta de projeto e escolha “Properties” ou aperte ALT+ENTER com a pasta de projeto selecionada. Uma outra forma é usar o menu superior Project → Properties. Na nova janela, navegue até a configuração Build → Tools → MSP430 Compiler → Processor Options e selecione o modelo “small” para o código e para os dados como na Figura 7.

Figura 7 – Modelo de memória






Com o projeto configurado, gere o binário com CTRL+B e entre na depuração apertando F5. Você irá perceber que a primeira linha da main ficou marcada. Ela é a próxima instrução que será executada. Para visualizar o valor dos registradores “R4” e “R5”, use a aba “Watch” na vista de depuração, como na Figura 8 à esquerda. Acrescente os registros usando o botão “+” (Add Expression).

Figura 8 – Observando variáveis



Através da aba Debug -> Watch ou pela vista dos registradores

Uma outra forma de ver o conteúdo dos registradores é através da vista dos registradores como indicado na Figura 8 à direita. Se precisar reiniciar o seu programa use o botão  ou a combinação de teclas CTRL+SHIFT+F5. Se a aba de registradores não estiver visível, selecione as opções “View” → “Registers”.

Você pode executar o seu programa passo a passo apertando o botão step over , ou usando o atalho F10. Acompanhe as modificações dos registradores. O valor dos registradores só será mostrado se o programa estiver pausado. Se o programa estiver em execução, aperte pause  para visualizar o resultado dos registradores

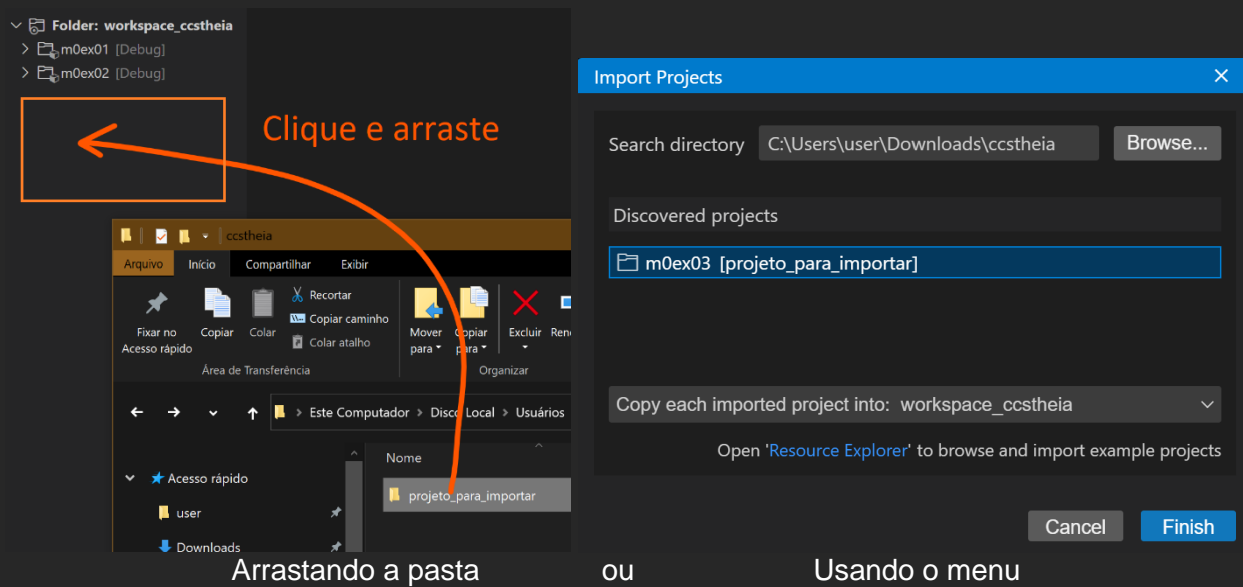
Para refletir: Porque o resultado em R5 deu 0x10? Você pode mudar o formato do número clicando nos 3 pontinhos no canto superior direito da vista dos registradores. É recomendado trabalhar e pensar em hexadecimal.

Neste ponto você deve estar se perguntando porque usamos os registros R[4,5,6] e não R[1,2,3]? O MSP430 possui 16 registradores numerados de R0 a R15. Os 4 primeiros (R0 a R3) são especiais. Só podemos usar a partir de R4. Veremos mais detalhes nas próximas semanas (módulo 1).

Importar um projeto

Importe o projeto que está na pasta “Downloads”. Basta **arrastar a pasta para a lista de projetos**. Uma outra forma é usar o menu “File” -> “Import” e selecionar a pasta raiz onde se encontra a pasta de projeto que você quer importar, neste caso, a pasta Downloads. Não é possível importar usando pastas compactadas (zip).

Figura 8 – Importar um projeto



Execute o programa importado. Ele é semelhante ao anterior, mas usa instruções de 16 bits. *Para testar: Modifique a soma para operar em 8 bits. Qual é o resultado?*

Subrotinas e ponto de parada (Breakpoint)

Em programação, é comum reaproveitar parte do código em funções, ou, como chamamos em assembly, subrotinas. Crie um projeto para o programa m0ex04 e copie o código da Figura 9.

Figura 9 – Subrotinas em assembly

```
m0ex04.asm X
1      .cdecls "msp430.h"
2      .global main
3
4      .text
5
6      main:
7          mov.w    #(<WDTPW|WDTHOLD), &WDTCTL
8
9          mov     #3, R4      ; Inicializa
10         mov     #4, R5      ; R4 = 3, R5 = 4
11         clr     R6          ; R6 = 0
12
13     main_loop:
14         call    #acumula    ; Chama acumula
15         dec     R4          ; R4 vezes
16         jnz     main_loop
17
18         jmp     $           ; Resultado:
19         nop     ; R6 = R4 x R5
20
21     ; -----
22     ; Subrotina que acumula R5 em R6
23     acumula:
24         add     R5, R6      ; R6 = R6 + R5
25         ret
26
```

Nas linhas 9 a 11, o programa inicializa os registradores R4 = 3, R5 = 4, R6 = 0.



Na linha 14, a subrotina “acumula” é chamada. Nesse ponto, a *main* é interrompida e a subrotina “acumula” executa na linha 24. Toda vez que ela é chamada, ela soma 4 (valor de R5) ao registro R6.

A subrotina retorna para a main na instrução da linha 25, ou seja: “ret” (de retorno). O retorno lembra de onde a subrotina foi chamada e o programa volta aonde parou: na linha 15.

R4 é então decrementado. O salto “jnz” (jump if not zero) avalia o resultado da linha anterior e vai para o rótulo “main_loop” se R4 não for zero.

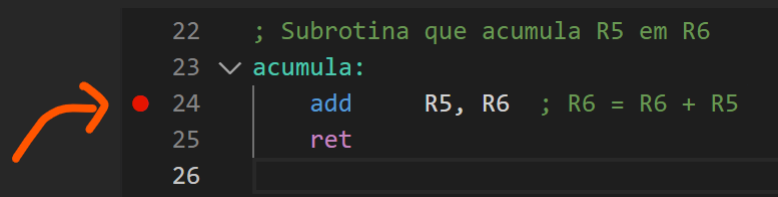
Quando R4 chega em 0, o salto não é realizado e o programa termina.

Se você não configurou o projeto para uso de memória curta (small memory model), faça-o agora, caso contrário as instruções call e ret não funcionarão (mas podem ser substituídas por calla e reta).

Tente executar esse programa passo-a-passo usando o botão *step-into* , ou o atalho F11 ao invés do *step-over*  (F10) usado anteriormente. Explore a diferença das duas formas ao executar a linha 14, onde chamamos a subrotina através da instrução “call”. O passo *step-over* executa, mas não entra na subrotina, já o passo *step-into* mergulha na subrotina e permite avaliar o que acontece dentro dela. As duas formas executam a subrotina.

Vamos agora introduzir o conceito de **ponto de parada (breakpoint)**. Quando a execução atinge um ponto de parada, o software é interrompido. Esse recurso é útil para verificar partes específicas de códigos grandes. Coloque um ponto de parada na linha 24, ou seja, na instrução **add R5, R6**, dentro da subrotina. Você pode fazer isso usando o atalho F9 com o cursor na linha desejada ou clicando com o mouse do lado esquerdo do número da linha.

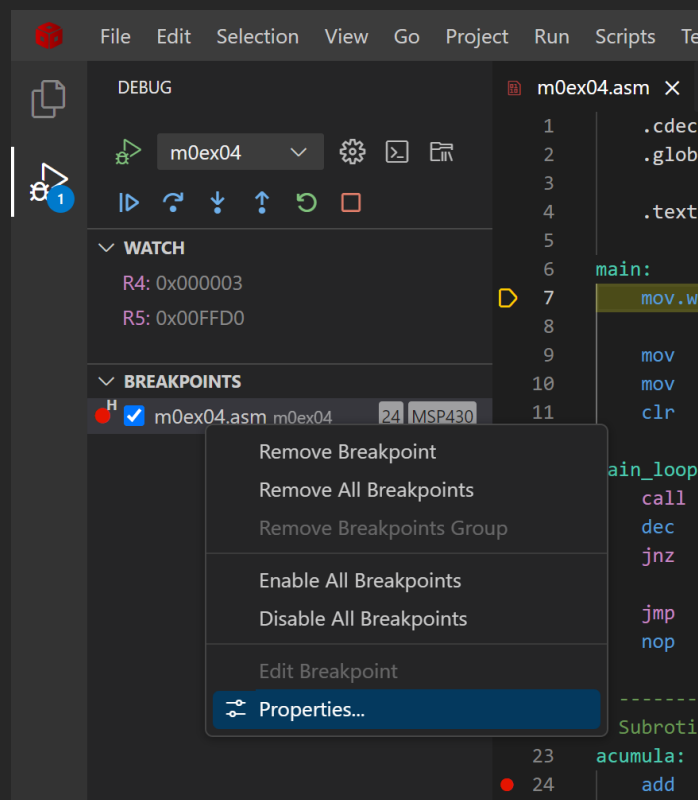
Figura 10 – Atribuição de um breakpoint



Rode o programa usando F5 para ver o efeito.

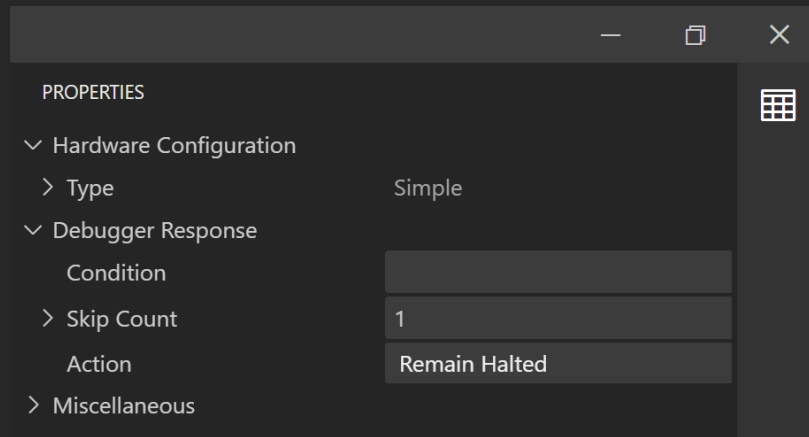
Agora imagine o seguinte: Você quer encontrar um bug no seu software que só acontece na segunda chamada de uma subrotina. Você pode programar um breakpoint para fazer exatamente isso. Depois de criado, o breakpoint aparece na janela de depuração, clique com o botão direito e selecione “Properties”.

Figura 11 – Breakpoint na vista de depuração



Reinicie o programa com CTRL+SHIFT+F5. Na guia de propriedades, com o breakpoint selecionado, altere a contagem “Skip Count” para 1, como na Figura 12. Rode o programa e note que no estado que ele parou já houve uma execução daquela subrotina antes, ou seja, o valor de R6 = 4.

Figura 12 – Propriedade do breakpoint



Memória RAM

Crie o projeto m0ex05. Nele vamos demonstrar o uso da pseudo instrução “.data”. Tudo que for escrito depois de .data vai ser colocado na memória RAM. Os dados que são colocados na RAM não perduram depois de um reset. Essa seção é usada apenas para testes durante a etapa de depuração e geralmente é escrita no final do arquivo. No código exemplo, a RAM é preenchida de 4 formas diferentes.

Figura 13 – Declaração de elementos na RAM

```
25      .data ; Tudo abaixo vai p/ a RAM
26      v1: .byte    1 , 2 , 3 , 4
27      v2: .word    1 , 2 , 3 , 4
28      v3: .byte    '1','2','3','4'
29      v4: .byte    "1 , 2 , 3 , 4"
```

No exemplo, são criados 4 vetores. Cada vetor é declarado de uma forma diferente, resultando num vetor diferente. O vetor v1 é composto por 4 bytes com os valores de 1 a 4. O segundo vetor v2, também é composto pelos mesmos elementos, porém cada elemento ocupa dois bytes, ou seja, uma word de 16 bits. O terceiro vetor v3, é declarado com os caracteres da tabela ASCII que representam os números de 1 a 4. Os números nessa tabela iniciam no byte 0x30 (zero) e vão até 0x39 (que representa o número nove). Note que cada caractere foi declarado usando aspas simples. O último vetor foi declarado usando aspas duplas, que converte todos os caracteres para bytes da tabela ASCII, incluindo espaço em branco e vírgula. Para avaliar cada vetor, propomos o seguinte programa:

Figura 14 –Exemplo de uso da RAM

```

6  main:
7      mov.w    #(WDTW|WDTHOLD), &WDTCTL
8
9      mov.w    #v1,    R4        ; R4 = 0x2400
10     mov.b    1(R4), R5        ; --> 2
11
12     mov.w    #v2,    R4        ; R4 = 0x2404
13     mov.b    1(R4), R6        ; --> 0 (MSByte de 0x0001)
14
15     mov.w    #v3,    R4        ; R4 = 0x240C
16     mov.b    1(R4), R7        ; --> 0x32 (char '2')
17
18     mov.w    #v4,    R4        ; R4 = 0x2410
19     mov.b    1(R4), R8        ; --> 0x20 (char ' ')
20     |         |         |         |         |         |
21     |         |         |         |         |         |
22     jmp      $
23     nop
24

```

Rode o programa para avaliar o que acontece com os registros de R5 a R8. Os operandos #vn se referem ao endereço do vetor na RAM, já os operandos 1(R4) representam os objetos apontados por R4 + 1, ou seja, o segundo byte do vetor, já que o primeiro índice é o zero.

Para visualizar a memória RAM, clique em “View” → “Memory” (não é memory allocation nem memory map). Digite o endereço 0x2400 no campo “Location” e tecla ENTER. Veja o conteúdo dos 4 vetores como na Figura 15.

Figura 15 – Valores dos bytes na memória.

Memory (1) X

Location0x2400Format8-Bit Hex - TI Style

0x02400	01	02	03	04	01	00	02	00	03	00	04	00	31	32	33
0x0240F	34	31	20	2C	20	32	20	2C	20	33	20	2C	20	34	F7
0x0241E	E0	37	DF	FB	38	E9	7F	7B	9D	F5	FF	EA	FF	FB	F7
0x0242D	D7	F7	DD	FE	EF	D0	A7	C7	8D	FF	55	FF	AC	3B	B8
0x0243C	E9	FD	7F	7F	F7	A0	DF	7C	FD	0D	57	C0	39	65	15

Explore os modos de visualização [8/16]-bit Hex – TI Style e o modo de caracteres. O que você notou de esquisito na ordem dos bytes no segundo vetor?

Conclusão

Com isso terminamos o primeiro tutorial. O objetivo deste roteiro é apenas familiarizar-nos com a ferramenta Code Composer Studio Theia baseada no Visual Studio Code. Não se preocupe com a linguagem de programação (C ou Assembly) ou a arquitetura do MSP430 neste primeiro instante. Aprenderemos tudo isso e muito mais nas próximas aulas.

Abaixo, segue a lista dos atalhos vistos neste roteiro.

Tabela 1 – Resumo dos atalhos

Função	Atalho	Descrição
Build	CTRL+B	Compila: Gera o binário
Flash sem Debug	CTRL+F5	Grava o binário na Flash
Debug	F5	Grava e entra na depuração
Run	F5	Roda o programa
Pause	F6	Pausa o programa
Breakpoint	F9	Insere um breakpoint na linha
Step-over	F10	Executa um passo (incluindo funções)
Step-into	F11	Executa um passo (entra nas funções)
Step-out	SHIFT + F11	Executa até o retorno
Restart	CTRL+SHIFT+F5	Reinicia a depuração
Stop	SHIFT+F5	Termina a depuração

Extras

Para obter o realce de sintaxe para a linguagem Assembly do MSP430, é necessário instalar a extensão “MSP430 Assembly”. Baixe o arquivo no link a seguir:

https://zacharypenn.gallery.vsassets.io/_apis/public/gallery/publisher/zacharypenn/extension/msp430-assembly/1.0.0/assetbyname/Microsoft.VisualStudio.Services.VSIXPackage

e renomeie a extensão para "vsix". Em seguida instale manualmente clicando no botão de reticências "...", na aba extensões e selecione “Install from VSIX”, conforme a Figura 16:

Figura 16 – Instalando a extensão de realce de sintaxe para o assembly do MSP430

