Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica
e de Computação

EA871 - Laboratório de Programação Básica de Sistemas Digitais

Roteiro 2 - IDE CodeWarrior

Aluno: Fernando Teodoro de Cillo

RA: 197029

Campinas
Março de 2022

1 Introdução

O intuito deste experimento é expor e utilizar os conceitos básicos de programação em C para o microcontrolador MKL25Z128 no IDE CodeWarrior. São apresentadas algumas das ferramentas mais importantes do CodeWarrior, além de aplicação de conceitos e programação de *firmwares* e alocação de memória.

2 Experimento

2.1

Quando qualificamos variáveis como *global* ou *static* elas são armazenaas como variáveis não inicializadas, o que consequentemente aumenta o tamanho do arquivo *.bss*. Já a definição do tipo `uint8_t` é um caso curioso, pois aumenta o tamanho do arquivo *.txt*, apesar de as variáveis individualmente consumirem um espaço menor. Isso pode acontecer porque, em algum estágio da *toolchain*, uma instrução adicional foi gerada para aceitar esse tipo de variável, e o espaço ocupado por essa instrução em linguagem de máquina é maior do que o espaço economizado pelas variáveis. A tabela 1 mostra este comportamento interessante.

Tabela 1: alocação de memória

	(a)		(b)		(c)		(d)	
	End. inicial	bytes	End. inicial	bytes	End. inicial	bytes	End. inicial	bytes
a	20002FF4	4	1FFFF020	4	20002FF7	1	1FFFF020	1
b	20002FF0	4	1FFFF004	4	20002FF6	1	1FFFF004	1
c	20002FEC	4	20002FF4	4	20002FF5	1	20002FF7	1
d	20002FE8	4	20002FF0	4	20002FF4	1	20002FF6	1
`.txt`	-	1128	-	1140	-	1132	-	1144
`.data`	-	24	-	24	-	24	-	24
`.bss`	-	2076	-	2084	-	2076	-	2084

2.2

O espaço alocado para uma variável deve ser coerente com o número de bits necessários para representar o seu valor. O valor 720, por exemplo, precisa de 10 bits para ser representado, e por isso é mostrado incorretamente em uma representação de 8 bits (ocorre o *overflow* e o valor retornado é 208). Outro problema que encontramos em variáveis *signed* é que o primeiro bit é sempre interpretado como bit de sinal, e isso faz com que qualquer valor que tenha o bit 1 no bit mais significativo seja entendido como um valor negativo, e é daí que vem a diferença entre os valores obtidos para `uint16_t` e `int16_t` na tabela 2, por exemplo.

Um fato interessante mostrado na tabela é que o valor de 13! está errado para todas as representações, inclusive `uint32_t` e `int32_t`, porque o valor correto precisaria de 33 bits para

Tabela 2: impacto da tipagem para a representação correta de valores

	uint8_t	uint16_t	uint32_t	int8_t	int16_t	int32_t
n=5	120	120	120	120	120	120
n=6	208	720	720	-48	720	720
n=7	176	5040	5040	-80	5040	5040
n=8	128	40320	40320	-128	-25216	40320
n=9	128	35200	362880	-128	-30336	362880
n=10	0	24320	3628800	0	24320	3628800
n=11	0	5376	39916800	0	5376	399168800
n=12	0	64512	479001600	0	-1024	479001600
n=13	0	52224	1932053504	0	-13312	1932053504

ser representado. Os valores obtidos são iguais porque o processador corta os bits mais significativos (neste caso, um bit de valor 1 que seria entendido como sinal negativo na representação `int32_t`, ou seja, essa tipagem exigiria no mínimo 34 bits para mostrar o valor correto).

2.3

Tabela 3: Caption

	a	b	c	d	Justificativa
Original	15	60	0	15	A divisão 15/60, que dá 0.25, é truncada para 0 porque estamos operando com inteiros.
(a)	15	60	0.0	15.0	A divisão de inteiros é truncada, mesmo salvando o resultado da operação em *float*.
(b)	15.0	60.0	0.0	15.0	A divisão de inteiros é truncada, mesmo salvando as variáveis em *float*.
(c)	15.0	60.0	0.25	15.0	A linha alterada faz com que a divisão ocorra corretamente.
(d)	erro				Resto de *float* por *float* não está definido em C.
(e)	15	60	0	15	O valor de a não é alterado na memória.

2.4

A ordenação adotada pelo microcontrolador é *little endian*, ou seja, ele armazena os *bytes* menos significativos em endereços menores. Podemos ver isso pela figura **??**, em que vemos vários valores 00 nos *bits* mais significativos das variáveis (colunas E e F), o que suger que esses sejam os *bits* mais significativos de variáveis que não utilizam todos os *bits* possíveis da representação (e por isso os *bits* mais significativos estão zerados)

0x20002ff4 : 0x20002FF4 <Hex Integer> | New Renderings...

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20002F90	9A	E4	49	9F	7F	83	E5	50	10	E6	75	DC	F8	8F	31	1B
20002FA0	66	2E	99	44	6A	90	B6	3C	9D	5C	6B	93	94	B0	BB	5B
20002FB0	91	D4	97	C4	10	E9	1F	68	5D	42	7C	14	45	25	E1	73
20002FC0	FF	FC	96	FA	FF	FC	96	FA	1C	F0	FF	1F	85	09	00	00
20002FD0	E8	2F	00	20	EC	2F	00	20	1C	F0	FF	1F	FF	FC	96	FA
20002FE0	E8	2F	00	20	00	F0	FF	1F	00	00	02	40	55	0B	00	00
20002FF0	1C	F0	FF	1F	2D	03	61	47	2D	03	61	47	35	09	00	00
20003000	??	??	??	??	??	??	??	??	??	??	??	??	??	??	??	??

Figura 1: endianess

2.5

Inserindo um *breakpoint* na linha 18 podemos parar a execução do programa em um momento em que as variáveis "frase"e "palavra"já foram chamadas, mas não buscadas. Assim, sobrescrevendo sobre os valores armazenados nessas variáveis, é possível fazer o programa continuar com valores editados pela perspectiva Debug do IDE CodeWarrior. Na aba "Variables"é possível editar diretamente o valor das variáveis apenas escrevendo sobre os valores antigos (a figura 2 mostra, em amarelo, os valores editados desde a execução da última linha, e podemos notar que é essencial colocar as letras entre aspas simples para serem entendidos como carcteres). Outra maneira de inserir uma frase ou palavra nessas variáveis é pela aba "Memory". Neste caso, é preciso escrever na célula de memória a correspondência do caractere em ASCII e as células editadas aparecem com o valor destacado em vermelho, como é possível ver pela figura 3.

Variables | Breakpoints | Registers | Memory | Modules

Name	Value	Location
i	536866844	0x20002ff0
j	2933	0x20002fec
numOcor	0	0x20002fe8
frase	0x20002f90	0x20002f90
[0]	'f'	0x20002f90
[1]	'e'	0x20002f91
[2]	'r'	0x20002f92
[3]	'n'	0x20002f93
[4]	'a'	0x20002f94
[5]	'n'	0x20002f95
[6]	'd'	0x20002f96
[7]	'o'	0x20002f97
[8]	' '	0x20002f98
[9]	' '	0x20002f99
[10]	' '	0x20002f9a
[11]	' '	0x20002f9b
[12]	' '	0x20002f9c
[13]	' '	0x20002f9d
[14]	' '	0x20002f9e

Figura 2: aba Variables

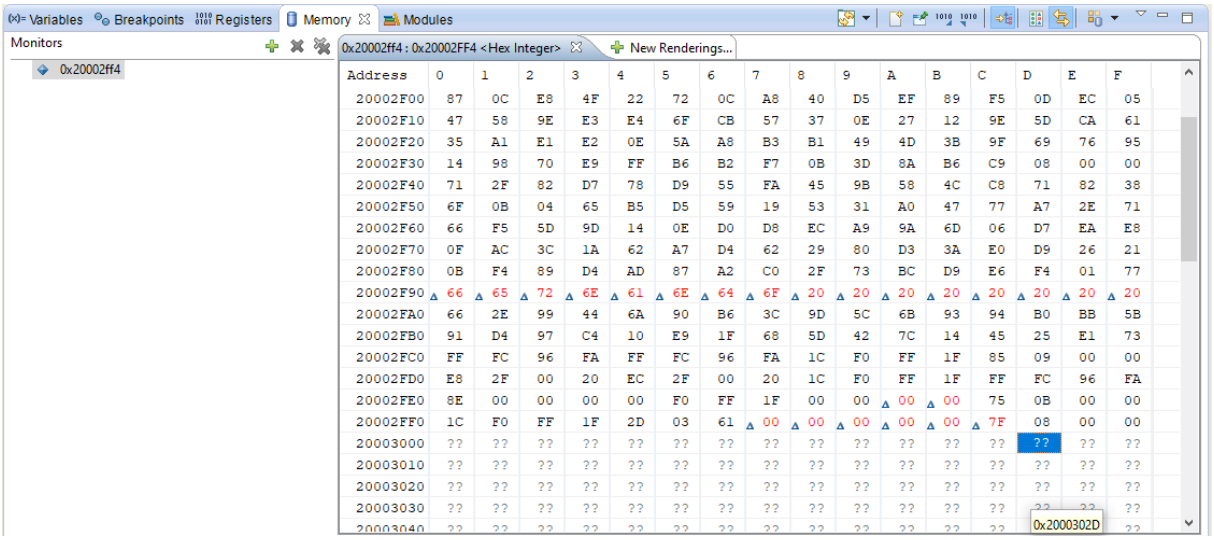

Figura 3: aba Memory

2.6

Tabela 4: endereços

		0x00000800 0x1FFFF000 0x20003000	0x0000_1000 0x1fff_f500 0x2000_2f60	Tamanho do espaço de memória ocupado
programa	Início fim	0x00000898 0x000009b6	0x00001098 0x000011b6	296
ocorre	Início fim	0x20002ff7 0x20002ff8	0x20002f57 0x20002f58	1
m	Início fim	0x20002fe4 0x20002fe8	0x20002f44 0x20002f48	4
n	Início fim	0x20002fe0 0x20002fe4	0x20002f40 0x20002f44	4
i	Início fim	0x20002ff0 0x20002ff4	0x20002f50 0x20002f54	4
j	Início fim	0x20002fec 0x20002ff0	0x20002f4c 0x20002f50	4
numOcor	Início fim	0x20002fe8 0x20002fec	0x20002f48 0x20002f4c	4
frase	Início fim	0x20002f90 0x20002fe0	0x20002ef0 0x20002f40	80
palavra	Início fim	0x20002f40 0x20002f90	0x20002ea0 0x20002ef0	80

As alterações feitas no *script* de ligação alteram a posição de origem dos diversos segmentos da memória. Como modificamos o segmento de instruções, o segmento de memória e o topo da pilha, as posições em que as instruções e as nossas variáveis foram gravadas são diferentes, mas as posições relativas e o tamanho do espaço de memória ocupado não.