DISCIPLINA EA701
Introdução aos Sistemas Embarcados

ROTEIRO 4: Organização de Memória e Tipos de Dados em C

TECNOLOGIAS DE MEMÓRIA, ALINHAMENTO E ORDENAÇÃO DE *BYTES*, MAPA E SEGMENTAÇÃO DE MEMÓRIA, MEMÓRIA CACHE E MEMÓRIA FORTEMENTE ACOPLADA (TCM), SISTEMA DE MEMÓRIA NO STM32H7A3ZIT6-Q, TIPOS DE DADOS EM C.

Profs. Antonio A. F. Quevedo e Wu Shin-Ting

FEEC / UNICAMP

Revisado em agosto de 2024

INTRODUÇÃO 2
UM PROJETO-EXEMPLO PARA EXPLORAR MEMÓRIA 3
TECNOLOGIAS DE MEMÓRIA: PROM, EPROM, EEPROM, FLASH NAND, NOR, SRAM e DRAM 12
MAPA DE MEMÓRIA 16
ALINHAMENTO DE DADOS 19
ORDENAÇÃO DOS BYTES 20
SEGMENTAÇÃO DE MEMÓRIA PRINCIPAL 21
MEMÓRIA CACHE 24
MEMÓRIA FORTEMENTE ACOPLADA (TCM) 28
SISTEMA DE MEMÓRIA NO STM32H7A3 28
TIPOS DE DADOS EM C 30

INTRODUÇÃO

A memória é um componente fundamental em todos os sistemas computacionais. Nela são armazenadas as instruções a serem executadas (código), bem como os valores de variáveis usadas (dados) pelo programa. Os microcontroladores costumam ter a memória incluída no *chip*, e alguns deles ainda permitem configurar parte de seus pinos para acesso a componentes de memória externos aos mesmos.

Os dois tipos de memória mais usados em microcontroladores são a tecnologia FLASH e a SRAM. A memória FLASH é uma memória não-volátil (preserva o conteúdo mesmo sem energia) que pode ser apagada e gravada em blocos, e é normalmente usada para armazenar de forma semi-permanente o código a ser executado, ao mesmo tempo permitindo que o código seja modificado e regravado a partir de fontes externas, um processo que acontece rotineiramente em atualizações de *firmware* nos equipamentos. A memória SRAM, ou RAM estática, é uma memória volátil (perde os dados ao ser retirada a energia) com um custo e tamanho maiores que os da memória DRAM (RAM dinâmica), porém de maior velocidade e sem depender de um controlador específico. Para quantidades de memória RAM não muito grandes, como é comum em microcontroladores, a SRAM é ideal. Esta memória é normalmente usada para armazenar os valores de variáveis, na forma de alocação estática, *heap* ou *stack*, dependendo do tipo de informação armazenada.

Um fator que determina o uso de certos tipos de memória é sua velocidade. A memória SRAM é mais rápida que a FLASH para a leitura. Sistemas computacionais de maior porte costumam carregar uma cópia da parte do código em uma memória SRAM dedicada para

acesso mais rápido, sendo esta memória denominada **memória *cache***. Apesar de não ser tão comum em sistemas microcontrolados, alguns microcontroladores, como o STM32H7A3ZIT6-Q, possuem *cache* para otimizar a velocidade de execução. Entretanto, o gerenciamento desta memória é algo bastante complexo. A **TCM** (do inglês *Tightly Coupled Memory*) é mais comumente encontrada em microcontroladores e sistemas embarcados, onde a proximidade e a alta velocidade de acesso são essenciais para o desempenho de tarefas específicas. Esta memória especial é projetada para operar extremamente próxima ao processador, oferecendo latências muito baixas e alta largura de banda.

Neste roteiro, veremos a organização dos diferentes tipos de memória, bem como seu uso pelos diversos tipos de dados.

UM PROJETO-EXEMPLO PARA EXPLORAR MEMÓRIA

1. Crie um projeto denominado “Memoria”, da mesma forma que foi feito nos roteiros anteriores. Apague o código-fonte do arquivo “main.c” e o preencha com o seguinte código:

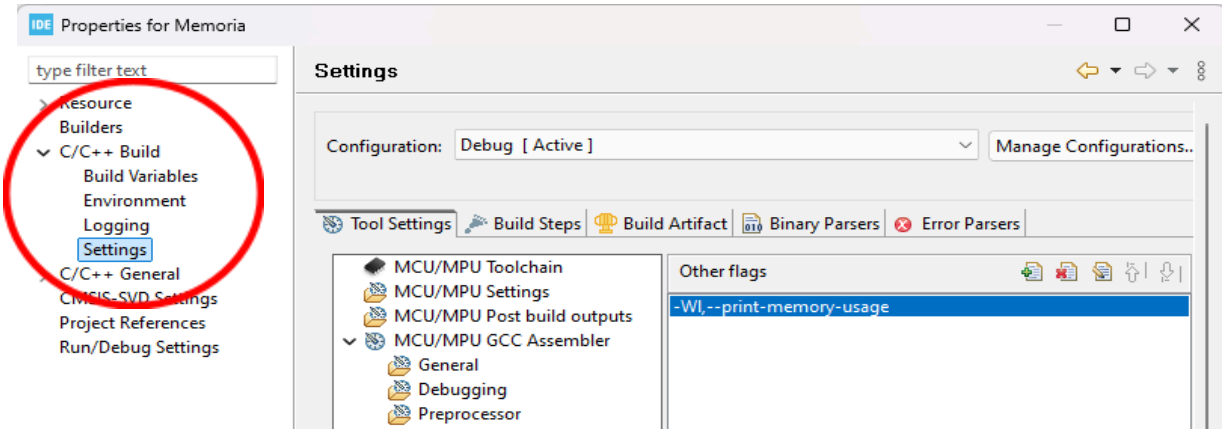
```
#include <stdint.h>
#include <malloc.h>
#include <string.h>
// Variaveis Globais inicializadas, armazenadas no segmento "data"
uint8_t global_var1 = 42;
// Variaveis Globais Não-inicializadas, armazenadas no segmento "BSS"
uint32_t uninitialized_var1;
// Variaveis constantes, armazenadas no segmento Rodata (FLASH)
const uint32_t const_var1 = 100;
// Variavel para demonstrar alocacao no "heap" e strings
char *heap_var;
// Outras variaveis a serem usadas na demonstracao
uint8_t *usig;
int16_t *sig;
int16_t k;
// Funcao para demonstrar o uso de qualificador register e static
uint32_t reg_demo(uint32_t v) {
        //Var local
        uint16_t reg_local;
  // Var local com qualificador register
        register uint32_t reg_var;
        // Variaveis estaticas, armazenadas no segmento "data"
        static uint8_t reg_static_var1 = 10;
        reg_local = const_var1;
        reg_var = v + 1;
        reg_static_var1++;
        if (reg_static_var1 == 0xff) reg_static_var1 = 0;
        return reg_var;
```

```
}
volatile uint32_t *RCC_AHB4ENR = ((uint32_t *)0x58024540);
volatile uint32_t *GPIOB_MODER = ((uint32_t *)0x58020400);
volatile uint32_t *GPIOB_OTYPER = ((uint32_t *)0x58020404);
volatile uint32_t *GPIOB_ODR = ((uint32_t *)0x58020414);
#define ITERACOES 5000
int main(void)
{
	struct mallinfo mi;
	uint32_t i;
	float *fp;
	float x;
	//Aloca 100 bytes para o vetor heap_var com 100 elementos do tipo
	//char. Observe a conversao explicita de tipo de retorno "void"
	//da funcao malloc para "char *"
	heap_var = (char *)malloc(sizeof(char) * 100);
	//Atribuir 200 a heap_var[0] que é equivalente a *heap_var
	*heap_var = 200;
	//Ler a informacao sobre alocacao de memoria
	mi = mallinfo();
	global_var1++;
	uninitialized_var1 = 5;
	//const_var1++;
	global_var1 = reg_demo(10);
	// A funcao copia a segunda string na primeira
	// Por padrao, as variaveis processadas pelas funcoes da biblioteca
	// "string" sao do tipo "char"
	strcpy(heap_var, "This is a NULL-terminated String");
	// Muda o espaço alocado
	// Observe a conversao explicita do tipo de variavel heap_var
	heap_var = (char *)realloc(heap_var, sizeof(uint8_t) * 200);
	mi = mallinfo();
	usig = (uint8_t *)0xAABBCCDD;
	sig = &k;
	k = -10;
	fp = &x;
	*fp = 3.375;
	*RCC_AHB4ENR =  *RCC_AHB4ENR | 0x00000002;
	*GPIOB_MODER = ((*GPIOB_MODER | 0x00000001) & 0xFFFFFFFD);
	*GPIOB_OTYPER = *GPIOB_OTYPER & 0xFFFFFFFE;
	for(;;) {
		*GPIOB_ODR = *GPIOB_ODR & 0xFFFFFFFE; //PB0 = 0;
		for (i=0; i<ITERACOES; i++); // Espera
		*GPIOB_ODR |= *GPIOB_ODR | 0x00000001; //PB0 = 1;
		for (i=0; i<ITERACOES; i++); // Espera
	}
}
```

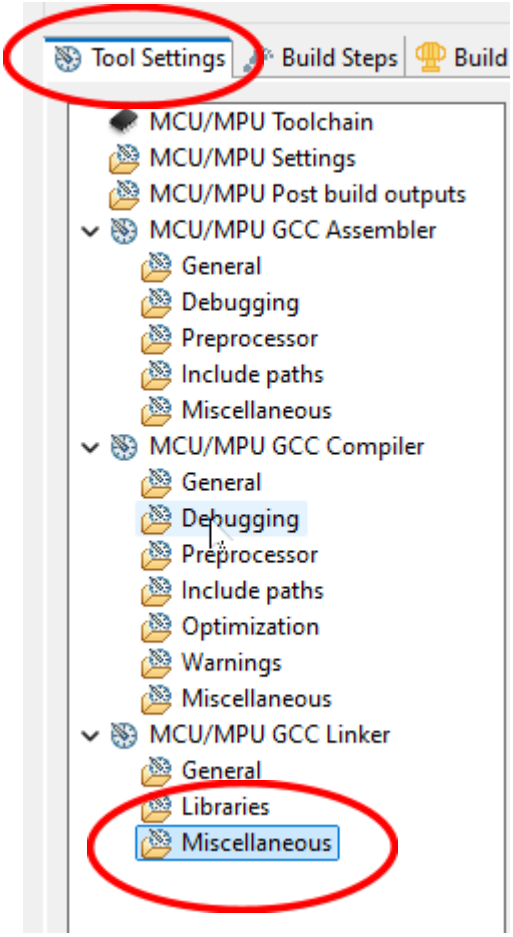
Observe o uso do tipo de dado “struct mallinfo”. Este tipo de dado é fornecido por algumas implementações da biblioteca padrão de alocação dinâmica de memória, como a GNU C Library (glibc). Ele é utilizado para fornecer informações sobre a alocação de memória em um programa que usa a função “malloc”, por meio da chamada da função “mallinfo()”. Para

que o compilador possa utilizar o tipo “struct mallinfo” e a função “mallinfo()”, é necessário incluir o arquivo de cabeçalho “malloc.h” no código com a diretiva “#include <malloc.h>”.

2. Para podermos analisar o uso da memória pelo programa em diferentes estágios de ligação, vamos ativar o ligador para que ele exiba estatísticas sobre o uso de memória. Com o projeto selecionado no “Project Explorer”, use o menu principal “Project > Properties”. Na janela que se abre, no painel à esquerda, expanda a opção “C/C++ Build” e selecione a opção “Settings”.

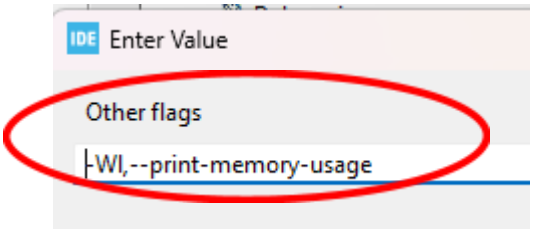


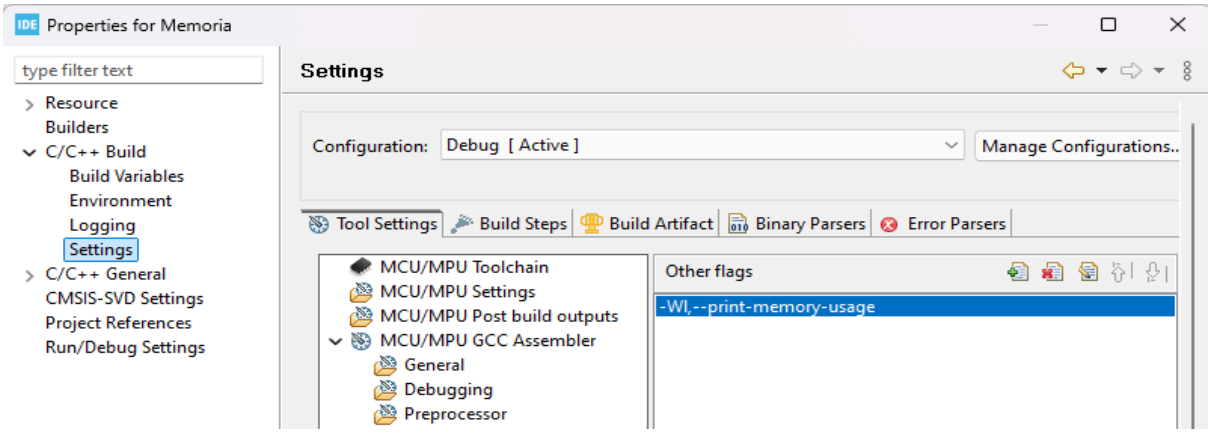
3. Mantendo ativa a aba “Tool Settings”, no conjunto de opções encontre o conjunto “MCU GCC Linker” e clique em “Miscellaneous” (expanda o conjunto se necessário).



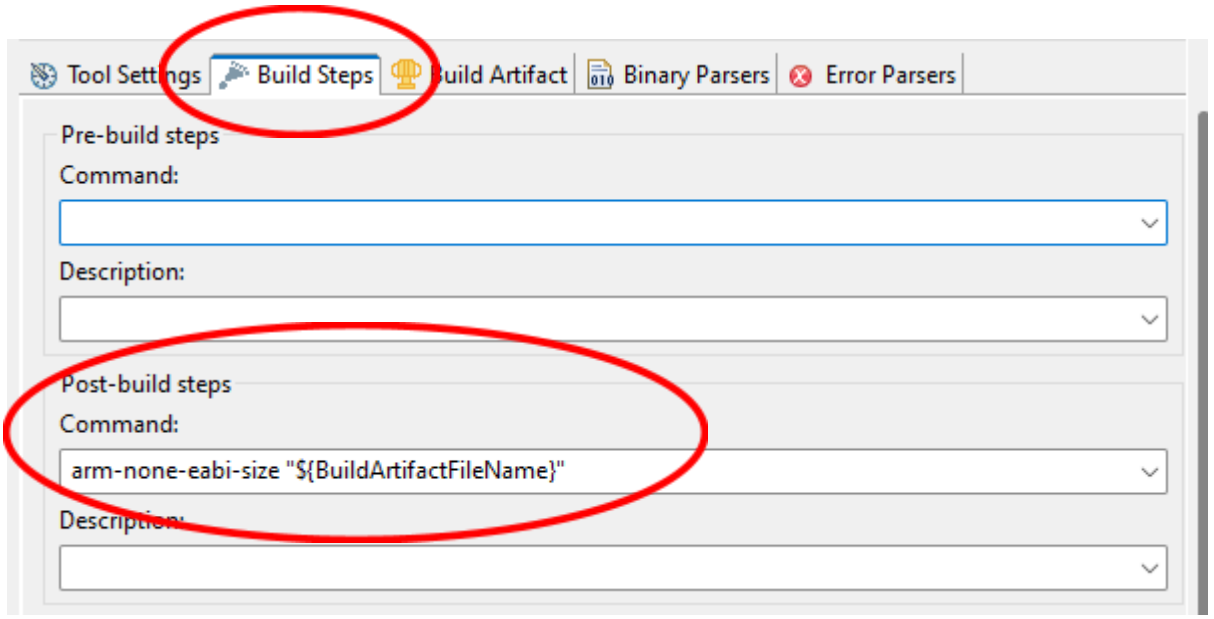
4. No campo “Other flags” clique no botão com o símbolo de “+” em verde.

5. Na janela que se abre, digite “-Wl,--print-memory-usage” e clique no botão “OK”. O “-W1”, informa ao GCC que as opções a seguir são para o ligador e não para o próprio compilador. O “--print-memory-usage” é a opção específica para o linker que faz com que ele exiba estatísticas sobre o uso de memória.





6. Adicionamos o comando “arm-none-eabi-size "${BuildArtifactFileName}”” para exibir, na etapa de pós-construção, o tamanho das seções do arquivo binário "${BuildArtifactFileName}” gerado pela compilação e ligação. Clique na aba “Build Steps”. Na seção “Post-build steps” adicionar o comando: arm-none-eabi-size "${BuildArtifactFileName}”



Vale destacar que este comando é incluído no comando anterior. Optamos por esta redundância para mostrar o comando que exibe o tamanho dos segmentos de dados e instruções em *bytes*.

7. Finalmente, clique no botão “Apply and Close”. Com estas configurações no projeto, ao se realizar um “Build”, ao final do processo serão mostradas as seguintes informações, incluindo a ocupação das unidades de memória físicas e o tamanho dos segmentos de instruções (text), de dados inicializados (data) e dados não inicializados (bss). Esses tamanhos serão apresentados tanto em decimal (dec) quanto em hexadecimal (hex), medidos em *bytes*.

```
Memory region         Used Size  Region Size  %age Used
         ITCMRAM:           0 GB        64 KB      0.00%
           FLASH:        2456 B         2 MB      0.12%
        DTCMRAM1:           0 GB        64 KB      0.00%
        DTCMRAM2:           0 GB        64 KB      0.00%
             RAM:        2064 B         1 MB      0.20%
          RAM_CD:           0 GB       128 KB      0.00%
         RAM_SRD:           0 GB        32 KB      0.00%
Finished building target: Mwmoria.elf

arm-none-eabi-size  Mwmoria.elf
arm-none-eabi-objdump -h -S Mwmoria.elf  > "Mwmoria.list"
   text    data     bss     dec     hex filename
   2352     104    1960    4416    1140 Mwmoria.elf
Finished building: default.size.stdout

Finished building: Mwmoria.list
```

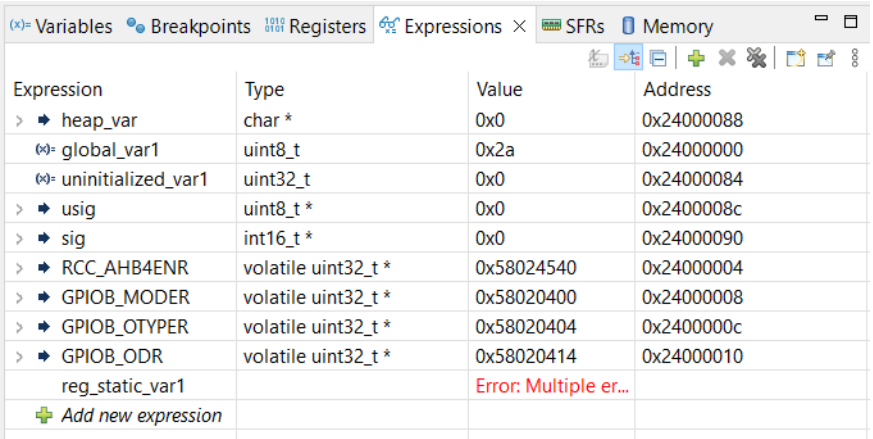
Há duas tecnologias de RAM, SRAM (RAM estática) e DRAM (RAM dinâmica). A qual tipo de RAM se refere na lista de informações?

8. Anote os valores apresentados, em termos de *bytes* e percentagem, do uso das unidades de memória físicas. Identifique os tipos de memória usados para o segmento de instruções e de dados inicializados e não-inicializados.

9. Clique no "Debug" para carregar o programa no microcontrolador. A perspectiva é chaveada para "Debug" e o fluxo de execução deve estar pausado (pequena seta azul à esquerda da linha) na linha "heap_var = (char *)malloc(sizeof(char));"

```
79    //Aloca 100 bytes para o vetor heap_var com 100 elementos do tipo
80    //char. Observe a conversao explicita de tipo de retorno "void"
81    //da funcao malloc para "char *"
82    heap_var = (char *)malloc(sizeof(char) * 100);
```

10. Na perspectiva de "Debug", abra a aba "Expressions" e adicione as expressões conforme a imagem abaixo. Com base nos endereços alocados para essas variáveis, identifique em qual unidade de memória física elas estão armazenadas e determine se elas estão localizadas em endereços mais baixos ou mais altos dentro dessa unidade de memória.



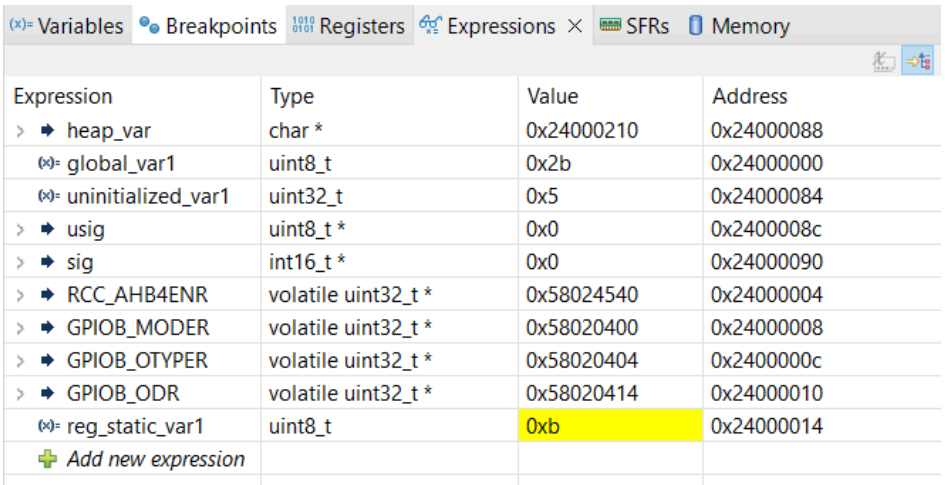
Expression	Type	Value	Address
heap_var	char *	0x0	0x24000088
global_var1	uint8_t	0x2a	0x24000000
uninitialized_var1	uint32_t	0x0	0x24000084
usig	uint8_t *	0x0	0x2400008c
sig	int16_t *	0x0	0x24000090
RCC_AHB4ENR	volatile uint32_t *	0x58024540	0x24000004
GPIOB_MODER	volatile uint32_t *	0x58020400	0x24000008
GPIOB_OTYPER	volatile uint32_t *	0x58020404	0x2400000c
GPIOB_ODR	volatile uint32_t *	0x58020414	0x24000010
reg_static_var1		Error: Multiple er...	
Add new expression			

Note que todos os ponteiros são indicados por uma seta azul-escuro. A coluna “Value” mostra o valor do ponteiro em si e não o valor para o qual ele aponta. À esquerda da seta pode-se expandir a variável para mostrar o valor para o qual ela aponta clicando em “>”. Note que os valores dos ponteiros não inicializados é zero.

11. Use o botão “Step Over” ou a tecla F6 para avançar uma linha de código. Execute as duas próximas linhas e veja o que acontece com os valores das expressões. Olhe também a aba “Variables”, que mostra “mi”. Esta variável contém as informações sobre a alocação de “heap_var”, especialmente o elemento “uordblks”, que indica o número de *bytes* alocados dinamicamente pelo usuário. Quais outras variáveis são mostradas nesta aba? Qual é a função em execução? Podemos dizer que a aba “Variables” apenas mostra variáveis locais da função em execução? Veja os endereços das variáveis que estão nesta aba. identifique em qual unidade de memória física elas estão armazenadas e determine se elas estão localizadas em endereços mais baixos ou mais altos dentro dessa unidade de memória.

12. Execute as duas linhas seguintes, que modificam valores em variáveis. Veja o que acontece com os seus valores na aba “Expressions”.

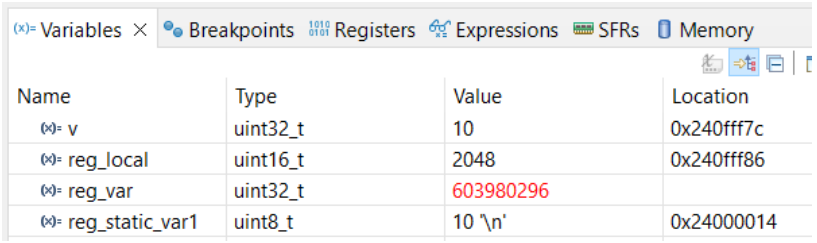
13. Ao chegar na linha “global_var1 = reg_demo(10);”, ao invés de usar “Step over” (Tecla F6), use o botão “Step Into” (Tecla F5) para entrar na função. Veja o que acontenceu com a última linha da aba “Expressions”. Por que foi preenchida esta linha depois da chamada da função?



Expression	Type	Value	Address
heap_var	char *	0x24000210	0x24000088
global_var1	uint8_t	0x2b	0x24000000
uninitialized_var1	uint32_t	0x5	0x24000084
usig	uint8_t *	0x0	0x2400008c
sig	int16_t *	0x0	0x24000090
RCC_AHB4ENR	volatile uint32_t *	0x58024540	0x24000004
GPIOB_MODER	volatile uint32_t *	0x58020400	0x24000008
GPIOB_OTYPER	volatile uint32_t *	0x58020404	0x2400000c
GPIOB_ODR	volatile uint32_t *	0x58020414	0x24000010
reg_static_var1	uint8_t	0xb	0x24000014
Add new expression			

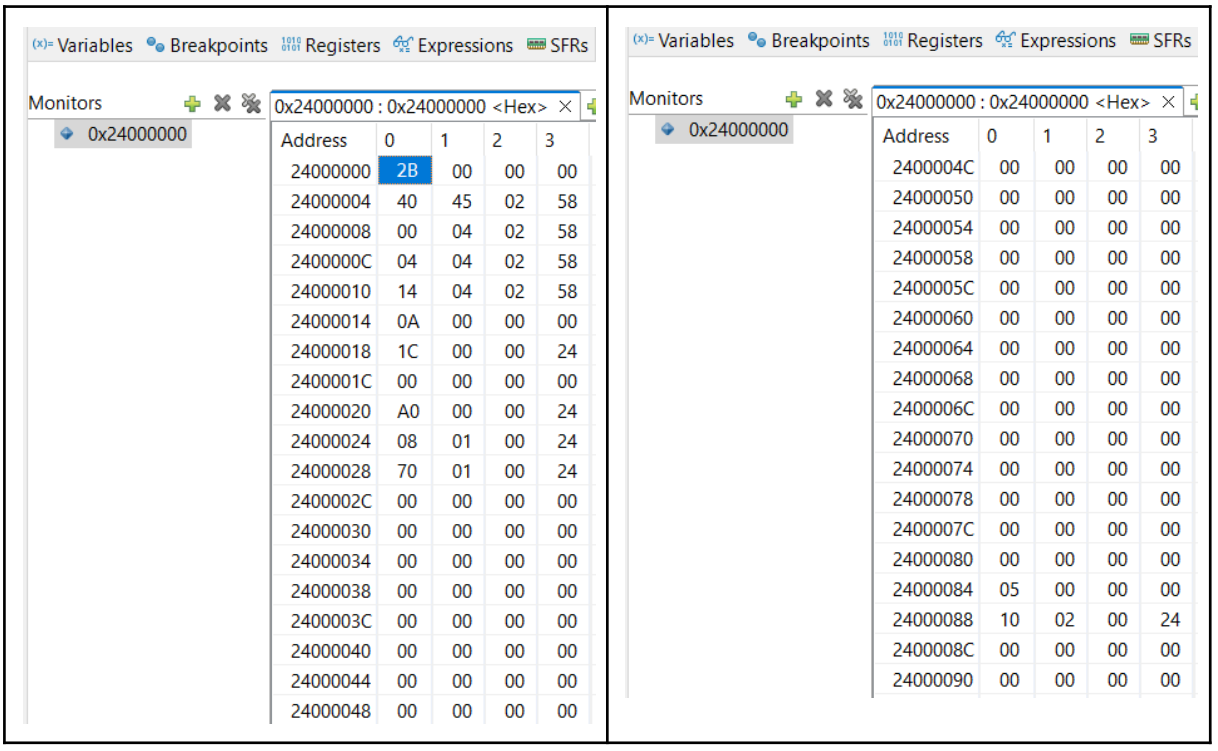
14. Vá para a aba “Variables” e veja as variáveis mostradas. Em qual função são utilizadas estas variáveis? Observe também que o campo “Location” da variável “reg_var” está vazio. Mude para a aba “Registers” e veja os registradores de uso geral. Um deles deve ter o valor “10” (ou “0xa”) passado para a função. Volte para a aba “Variables”. Execute, passo a passo, as instruções e veja o que acontece depois da execução da linha “reg_var = v + 1;” Volte para a aba “Registers” e veja se há algum registrador de uso geral com o mesmo valor no campo “Value” da variável “reg_var”. Qual foi o efeito do qualificador “register” na declaração da

variável “reg_var” dentro da função “reg_demo()”? Você poderia classificar esta variável como local ou global?



Name	Type	Value	Location
v	uint32_t	10	0x240fff7c
reg_local	uint16_t	2048	0x240fff86
reg_var	uint32_t	603980296	
reg_static_var1	uint8_t	10 '\n'	0x24000014

15. Vá para a aba “Memory” e analise a ordenação de *bits* das variáveis. São ordenados em *little-endian* ou em *big-endian*? Verifique ainda se as variáveis estão alocadas em espaços contíguos de memória, ou seja, se elas ficam adjacentes uma à outra. Caso contrário, identifique quais variáveis têm "buracos" entre elas e explique por que o compilador e o ligador podem não otimizar a alocação de memória para manter as variáveis consecutivas.

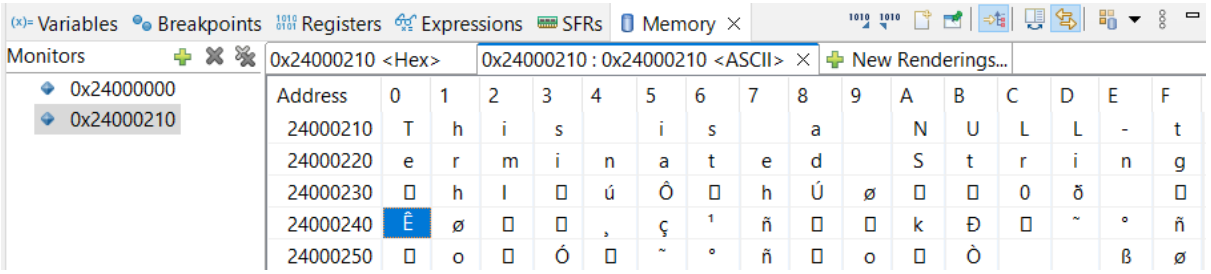


Address	0	1	2	3
24000000	2B	00	00	00
24000004	40	45	02	58
24000008	00	04	02	58
2400000C	04	04	02	58
24000010	14	04	02	58
24000014	0A	00	00	00
24000018	1C	00	00	24
2400001C	00	00	00	00
24000020	A0	00	00	24
24000024	08	01	00	24
24000028	70	01	00	24
2400002C	00	00	00	00
24000030	00	00	00	00
24000034	00	00	00	00
24000038	00	00	00	00
2400003C	00	00	00	00
24000040	00	00	00	00
24000044	00	00	00	00
24000048	00	00	00	00

Variables
Breakpoints
Registers
Expressions
SFRs
Monitors
0x24000000
0x24000000 : 0x24000000 <Hex>

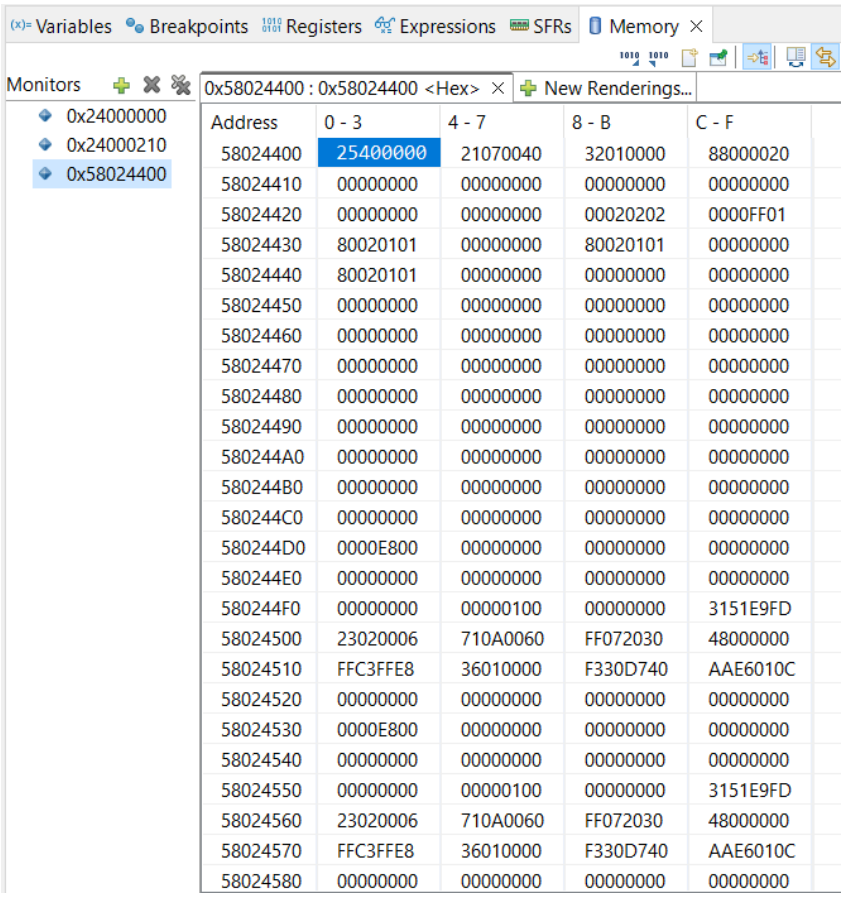
Address	0	1	2	3
2400004C	00	00	00	00
24000050	00	00	00	00
24000054	00	00	00	00
24000058	00	00	00	00
2400005C	00	00	00	00
24000060	00	00	00	00
24000064	00	00	00	00
24000068	00	00	00	00
2400006C	00	00	00	00
24000070	00	00	00	00
24000074	00	00	00	00
24000078	00	00	00	00
2400007C	00	00	00	00
24000080	00	00	00	00
24000084	05	00	00	00
24000088	10	02	00	24
2400008C	00	00	00	00
24000090	00	00	00	00

16. Ao sair da função, execute a instrução seguinte. Veja o valor do ponteiro “heap_var” e, na aba “Memory”, crie um monitor para o endereço definido no ponteiro. Clique com o botão DIREITO em qualquer célula e selecione “Format”. Na janela que se abre, mude o valor de “Column Size” para 1. Além disso, adicione uma nova renderização clicando em “+” e selecione “ASCII” como novo formato de renderização. Compare os valores hexadecimais dos bytes com os códigos ASCII. Note que a linguagem C usa um valor para indicar que a string terminou. Qual é este valor?



16. Execute as duas instruções a seguir e observe o efeito sobre a variável “mi”. Você consegue explicar o que a função “realloc” faz?

17. Execute as instruções até a linha “for(;;);”. Em seguida, vá para a aba "Monitor" e examine o conteúdo da memória a partir do endereço “0x58024400”. Compare os valores exibidos nas células de memória com os valores dos registradores do periférico RCC na aba "SFRs". Por que há tanta semelhança entre esses dois conjuntos de valores? O tipo de dado “struct” em C permite mapear um bloco de memória, como o intervalo de “0x58024400” a “0x5802457F”, associando grupos de células de memória com nomes de registradores mapeados. Isso facilita a organização e a leitura dos dados na memória, rotulando os grupos de células conforme os registradores correspondentes.



18. Onde está localizada a constante “const_var1”? Por quê?

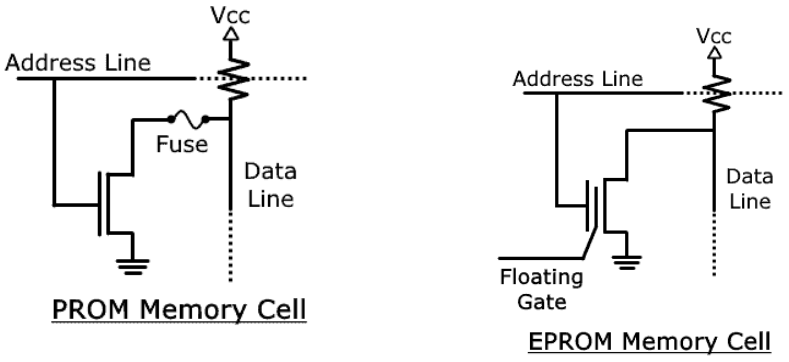
19. Termine a execução do programa (CTRL+F2). Volte para o editor da perspectiva "C/C++". Apague as barras de comentário ("//") da linha "const_var1++;" e tente realizar um "Build". O que acontece? **Dica:** Sempre leia atentamente as mensagens de erros e avisos.

20. Com base na sua análise, organize os segmentos Code, Data, BSS, *Stack* e *Heap* de acordo com seus endereços de memória. Verifique se o *layout* desses segmentos segue a ordenação típica, do endereço mais baixo para o mais alto: Code, Data, BSS, *Heap* e *Stack*. Com base na sua análise, organize os segmentos Code, Data, BSS, *Stack* e *Heap* conforme os endereços da memória. Certifique se o *layout* desses segmentos seguem a ordenação, de endereço mais baixo para o mais alto, Code, Data, BSS, *Heap* e *Stack*.

TECNOLOGIAS DE MEMÓRIA: PROM, EPROM, EEPROM, FLASH NAND, NOR, SRAM e DRAM

Entre as principais tecnologias de memória estão PROM, EPROM, EEPROM, SRAM, DRAM, FLASH NAND e FLASH NOR, cada uma com suas particularidades e aplicações específicas.

OTP (do inglês *One-Time Programmable*) refere-se a um tipo de memória que pode ser programada apenas uma vez. Após a gravação inicial, os dados armazenados na memória OTP não podem ser alterados. Nos microcontroladores, a memória OTP é aplicada no armazenamento de informações permanentes, identificadores únicos, códigos de configuração e dados de calibração. Além disso, é utilizada para armazenar dados de segurança, como chaves criptográficas. Existem várias tecnologias utilizadas para implementar a memória OTP em microcontroladores. Uma das tecnologias mais comuns é a **PROM** (do inglês *Programmable Read-Only Memory*). A PROM é uma **memória não-volátil**, programada uma única vez através de um processo que envolve a aplicação de uma alta tensão para fundir fusíveis internos, alterando permanentemente a configuração da memória.



Outra tecnologia relacionada é a EPROM (do inglês *Erasable Programmable Read-Only Memory*), que pode ser apagada com luz ultravioleta e reprogramada. Embora a EPROM seja reprogramável, em alguns casos ela é utilizada como uma memória OTP quando a capacidade

de apagamento não é necessária. A memória EPROM utiliza células baseadas em transistores de porta flutuante. Durante a programação, uma alta tensão é aplicada, resultando em uma descarga de avalanche de elétrons que se acumula no eletrodo da porta. Essa carga é isolada pela porta flutuante, impedindo seu vazamento e garantindo que os dados sejam armazenados de forma permanente. Para apagar os dados, é necessário expor a EPROM à luz ultravioleta, que dissipa a carga acumulada. Apesar de sua capacidade de ser reprogramada, a EPROM é considerada uma **memória não-volátil** porque mantém seu conteúdo mesmo quando o *chip* é desenergizado.

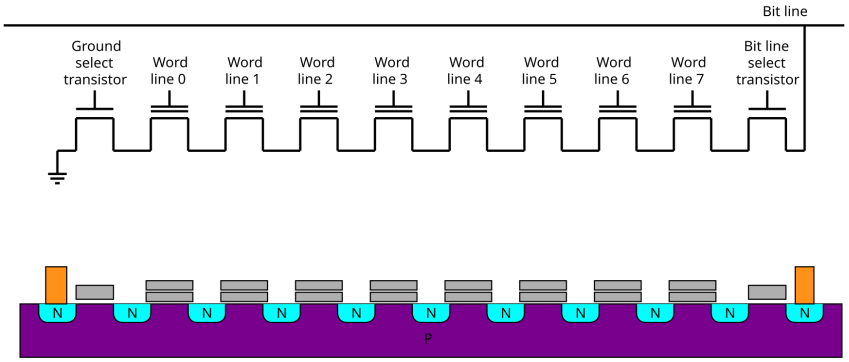
Desenvolvida para substituir o processo de apagamento por luz ultravioleta da EPROM, os *bytes* individuais da **EEPROM** (do inglês *Electrically Erasable Programmable Read-Only Memory)* podem ser apagados e reprogramados diretamente dentro do circuito usando um circuito de programação especial. Embora a EEPROM permita a reprogramação *byte* a *byte*, o que pode resultar em uma operação relativamente mais lenta, a memória FLASH, que é um tipo de EEPROM, foi projetada para oferecer alta velocidade. A memória FLASH possui um número limitado de ciclos de reprogramação, cerca de 10 mil, enquanto os modelos mais recentes de EEPROM podem suportar até 1 milhão de ciclos. Apesar de a EEPROM ser mais cara, a memória FLASH enfrenta a desvantagem de exigir a exclusão de grandes blocos de memória, em vez de *bytes* individuais

A memória FLASH é baseada em transistores de efeito de campo (FET) com uma camada adicional de porta flutuante. O nome FLASH deriva da ideia de *flash* em inglês, que transmite a ideia de algo que ocorre rapidamente, refletindo a capacidade da memória de apagar e gravar dados rapidamente. Em dispositivos semicondutores, o **tunelamento** é um fenômeno quântico que permite que elétrons atravessem barreiras de potencial que, segundo a física clássica, seriam intransponíveis. No caso da memória FLASH, o fenômeno específico conhecido como **tunelamento Fowler-Nordheim** permite que uma carga elétrica seja transferida para a porta flutuante do transistor, representando dados na memória. Quando um transistor é "programado" ou "escrito", uma carga é colocada na porta flutuante através do tunelamento Fowler-Nordheim. A presença ou ausência dessa carga define o valor dos *bits*, com a carga representando um *bit* "0" e a ausência de carga representando um *bit* "1". No estado não programado, as células de memória FLASH geralmente correspondem ao valor lógico "1" (ou nível alto).

FLASH NAND é um tipo de **memória não-volátil** que armazena dados em células dispostas em série dentro de cada linha de *bit*. Essa configuração faz com que a lógica da leitura dessas células se assemelha à lógica de uma porta NAND. Assim como uma porta NAND retorna uma saída baixa apenas se todas as entradas são altas, a leitura de uma linha de células em série depende do estado de todas as células na linha. Se qualquer célula na série estiver em um estado que impede a passagem de corrente, a leitura geral da linha é afetada. As células conectadas em série são, por sua vez, agrupadas em páginas, e essas páginas são organizadas em blocos. Devido à estrutura em série, o apagamento deve ser realizado em blocos inteiros, o que é eficiente para limpar grandes áreas de memória, mas impede o apagamento e a gravação de células individuais. A gravação, por outro lado, é realizada por páginas, o que permite a escrita de dados em uma página específica dentro de um bloco.

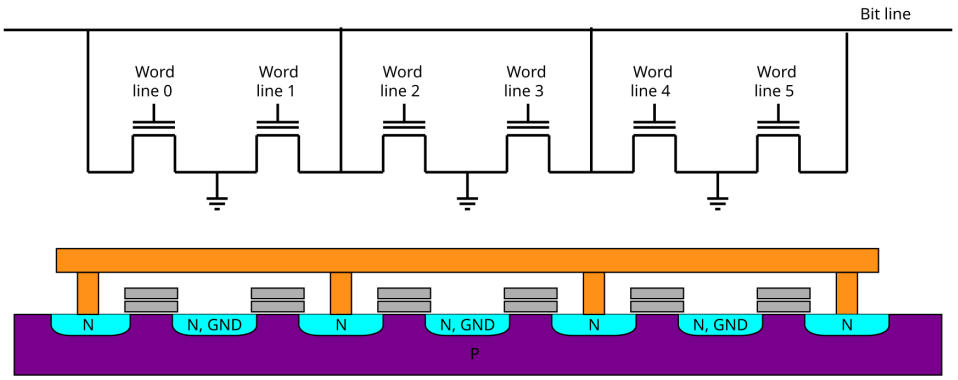
Essa arquitetura é amplamente utilizada em dispositivos de armazenamento devido à sua alta densidade e custo relativamente baixo. A memória FLASH NAND é ideal para aplicações que requerem grandes capacidades de armazenamento, como cartões de memória e unidades SSD (do inglês *Solid State Drives*). No entanto, a principal desvantagem é que a escrita e o apagamento devem ser feitos em blocos inteiros, o que pode reduzir a eficiência em

operações de leitura e escrita aleatória. Por essas razões, a FLASH NAND é menos comum em microcontroladores, que frequentemente demandam acesso rápido e flexível a pequenas porções de dados. A complexidade de controle e as limitações associadas à escrita e apagamento em grandes blocos não se adequam bem às necessidades típicas de desempenho e flexibilidade para operações de código e dados em microcontroladores.

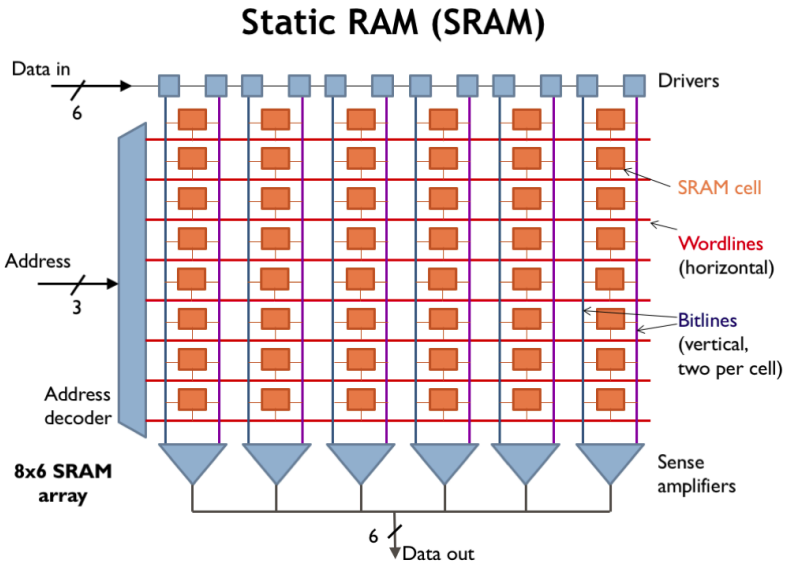


FLASH NOR é outra forma de **memória não-volátil** que armazena dados em células conectadas diretamente a uma linha de *bit* e a uma linha de palavra. Esta configuração em paralelo faz com que a leitura da memória FLASH NOR se assemelhar à lógica de uma porta NOR, no sentido de que a detecção do estado da célula reflete um tipo de "comparação" semelhante à verificação de todas as entradas em uma porta NOR. Se a célula está programada ("0") ou apagada ("1"), isso afeta a corrente medida e, por conseguinte, o resultado da leitura, refletindo uma lógica baseada no estado da célula. Ao contrário da FLASH NAND, que organiza as células em série e requer o acesso a blocos ou páginas inteiras, a FLASH NOR permite a leitura e gravação de dados em células individuais ou em unidades menores. Isso significa que um processador pode acessar diretamente qualquer endereço da memória sem precisar ler blocos inteiros.

Essa característica faz da FLASH NOR uma escolha ideal para armazenar *firmware* e código de inicialização que precisa ser acessado rapidamente. Sua arquitetura proporciona acesso aleatório direto, facilitando operações de leitura. No entanto, é geralmente mais lenta para operações de escrita e tem uma capacidade de armazenamento menor em comparação com a FLASH NAND. Por essas razões, a FLASH NOR é frequentemente preferida em aplicações que exigem execução direta de código da memória, como em sistemas embarcados e *firmware* de dispositivos.



SRAM (do inglês *Static Random-Access Memory*) é uma tecnologia de **memória volátil** que se destaca pela sua alta velocidade e simplicidade de operação. Ao contrário da DRAM (do inglês *Dynamic RAM*), que precisa ser periodicamente atualizada, a SRAM retém seus dados enquanto a energia estiver sendo fornecida. Isso se deve ao seu componente interno, que utiliza *flip-flops* para armazenar cada *bit* de informação. Essa estrutura permite acesso quase instantâneo aos dados, o que a torna ideal para aplicações que requerem altas taxas de transferência e baixa latência. No entanto, a SRAM é mais cara e consome mais energia por *bit* armazenado comparada à DRAM e à memória FLASH, o que limita seu uso a tamanhos relativamente menores.



DRAM (do inglês *Dynamic Random Access Memory*) é uma forma de **memória volátil** amplamente utilizada para armazenar dados temporários em computadores e outros dispositivos eletrônicos. A tecnologia da DRAM baseia-se em células de memória compostas por capacitores e transistores. Cada célula de DRAM armazena um *bit* de dado através de um capacitor que retém uma carga elétrica, representando um valor binário de 0 ou 1, enquanto um transistor controla o acesso à carga para ler ou escrever dados na célula. Quando a DRAM precisa ler um dado, o transistor é ativado para permitir a detecção da carga no capacitor, que é então amplificada e interpretada. Para escrever dados, a carga do capacitor é ajustada através do transistor para refletir o novo valor. Um aspecto importante da DRAM é

que a carga no capacitor pode vazar ao longo do tempo, o que exige que o conteúdo da célula seja periodicamente refrescado para manter a integridade dos dados.

Apesar de suas vantagens em termos de densidade de armazenamento, a DRAM não é comumente utilizada em microcontroladores. Isso se deve principalmente à necessidade constante de **refresco** (em inglês *refreshing*) **dos dados**, o que adiciona complexidade ao sistema e aumenta o consumo de energia, tornando-a menos eficiente para aplicações onde a simplicidade e a eficiência energética são prioritárias. Além disso, a DRAM é mais cara e complexa de implementar em comparação com outras tecnologias, como SRAM, que é preferida em microcontroladores devido à sua simplicidade, menor custo e desempenho mais rápido.

MAPA DE MEMÓRIA

A organização geral da memória de um sistema computacional pode ser vista no **mapa de memória**. Este mapa divide a memória em blocos, usando como critérios o tipo de memória e a utilização da mesma. A maioria dos microcontroladores ARM, por usarem registradores de 32 *bits*, podem acessar um espaço de memória total de 4GB. Note que, apesar de a CPU poder acessar todo este espaço, muitas vezes apenas parte dele está ocupado com elementos físicos de memória. A conversão de um endereço no espaço de endereçamento do processador para um endereço específico em uma unidade de memória física é realizada pelo **decodificador de endereços**. Normalmente, o endereço é dividido em dois campos: [CS | *Offset*]. O campo de *bits* mais significativos é usado para gerar o sinal de Seleção de *Chip* (CS), que identifica qual unidade de memória física deve ser ativada. Já o campo de *bits* menos significativos é utilizado para endereçar os *bytes* dentro da unidade de memória selecionada. Nesse contexto, o valor *Offset* = 0 corresponde ao **endereço-base**, ou seja, o endereço inicial do bloco contíguo em que a unidade de memória física está mapeada.

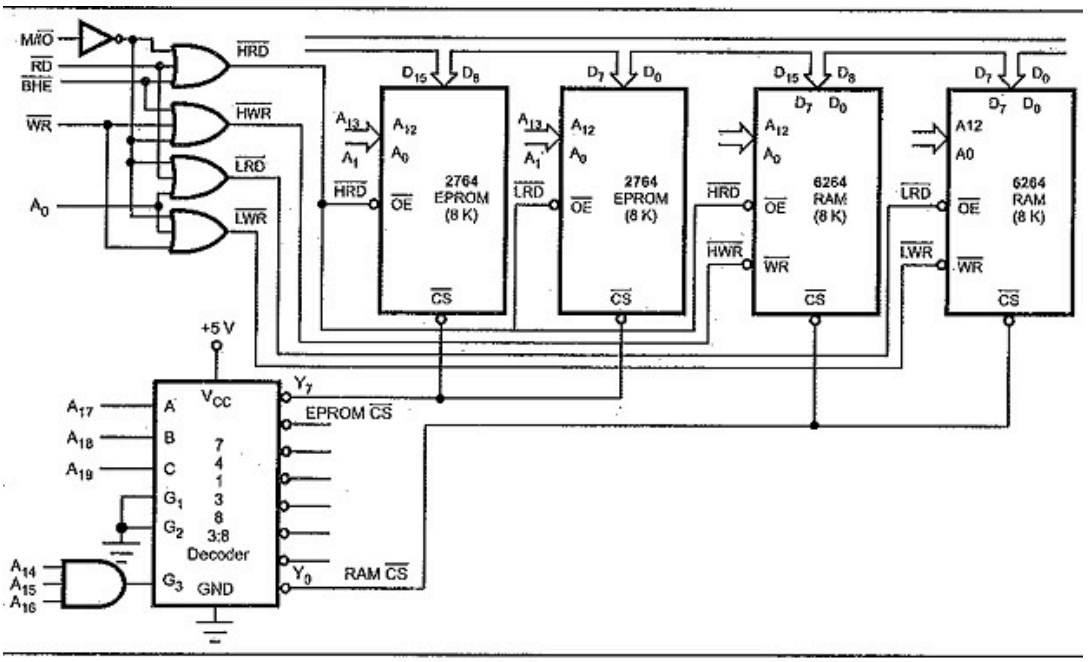
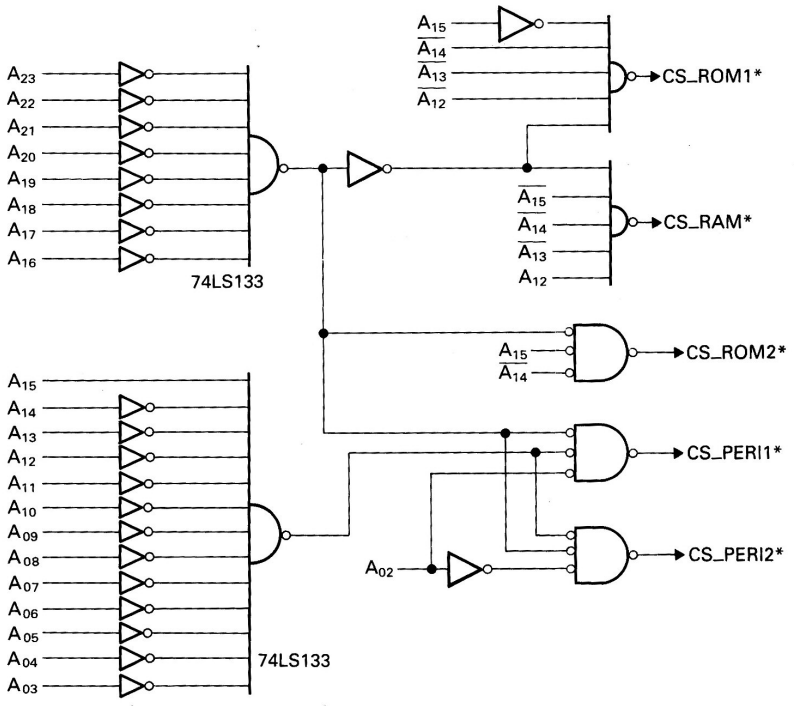


Fig. 10.14 Block decoding

Segue-se um exemplo de circuito de decodificação dos endereços do processador de 24 bits *em* sinais de Seleção de *Chip* de ROM1, ROM2, RAM, PERI1 e PERI2, sendo PER1 e PER2 dois periféricos mapeados no espaço de endereçamento do processador.



O Manual de Referência apresenta a organização de memória do microcontrolador STM32HA3ZIT-Q, começando com a informação de que memórias de programa, dados, registradores e portas de entrada/saída compartilham o mesmo espaço linear de 4 GB. A Tabela 6 da seção 2.3.2 apresenta o mapa de memória, incluindo blocos reservados para memórias externas e internas. Observe que os endereços de valores mais baixos estão no final

da tabela, enquanto os limites de cada bloco de memória são endereços alinhados em múltiplos de 4 *bytes*. Internamente, os seguintes blocos estão ocupados por elementos físicos de memória:

- **Periféricos (0x4000 0000 a 0x5FFF FFFF):** Espaço reservado para o acesso aos registradores mapeados em memória dos periféricos. É importante notar que esses elementos de memória não são do tipo SRAM ou FLASH; em vez disso, o acesso a esses endereços é direcionado aos registradores dos diferentes periféricos do microcontrolador.

Peripherals	0x4000 0000 - 0x5FFF FFFF	Peripherals (refer to *Table 7: Register boundary addresses*)	Device	-	Yes

- **RAM (0x2000 0000 a 0x3FFF FFFF):** Espaço da SRAM, dividido em vários blocos, sendo alguns reservados (sem elementos físicos) e outros com blocos de SRAM ligados a vários barramentos internos, Backup, SDR (do inglês *Single Data Rate*), AHB (do inglês *Advanced High-Performance*) e AXI (do inglês *Advanced eXtensible Interface*). Esses barramentos internos servem para interconectar a SRAM com diferentes partes do sistema, cada um com suas características e finalidades específicas.
- **Code (0x0000 0000 a 0x1FFF FFFF):** Área que inclui as memórias não-voláteis, como memória de sistema (registradores que guardam configurações gerais do microcontrolador), FLASH de código, e uma área OTP (do inglês *One-Time Programming*, que não pode ser regravada, usada para guardar dados permanentes).

Table 6. Memory map and default device memory area attributes (continued)

Region	Boundary address	Arm® Cortex®-M7	Type	Attributes	Execute never
RAM	0x3880 1000 - 0x3FFF FFFF	Reserved	Normal	Write-back, write allocate cache attribute	No
	0x3880 0000 - 0x3880 0FFF	Backup SRAM			
	0x3801 0000 - 0x387F FFFF	Reserved			
	0x3800 0000 - 0x3800 7FFF	SDR SRAM			
	0x3002 0000 - 0x37FF 7FFF	Reserved			
	0x3001 0000 - 0x3001 FFFF	AHB SRAM2			
	0x3000 0000 - 0x3000 FFFF	AHB SRAM1			
	0x2600 0000 - 0x2FFF FFFF	Reserved			
	0x2500 0000 - 0x25FF FFFF	GFXMMU			
	0x2410 0000 - 0x24FF FFFF	Reserved			
	0x240A 0000 - 0x240F FFFF	AXI SRAM3			
	0x2404 0000 - 0x2409 FFFF	AXI SRAM2			
	0x2400 0000 - 0x2403 FFFF	AXI SRAM1			
	0x2002 0000 - 0x23FF FFFF	Reserved			
	0x2000 0000 - 0x2001 FFFF	DTCM			
Code	0x1FF2 0000 - 0x1FFF FFFF	Reserved	Normal	Write-through cache attribute	No
	0x1FF0 0000 - 0x1FF1 FFFF	System Memory			
	0x08FF F400 - 0x1FEF FFFF	Reserved			
	0x08FF F000 - 0x08FF F3FF	OTP area	Normal	-	No
	0x0820 0000 - 0x08FF EFFF	Reserved	Normal	Write-through cache attribute	No
	0x0810 0000 - 0x081F FFFF	Flash memory bank 2			
	0x0800 0000 - 0x080F FFFF	Flash memory bank 1			
	0x0001 0000 - 0x07FF FFFF	Reserved			
	0x0000 0000 - 0x0000 FFFF	ITCM RAM			

Note que a SRAM e os periféricos podem ser lidos e escritos diretamente pela CPU usando instruções padrão, porém a região de *Code*, onde as instruções são armazenadas, permite apenas leitura direta pela CPU, mas as escritas devem ser feitas através de um periférico interno específico para gravação da FLASH.

ALINHAMENTO DE DADOS

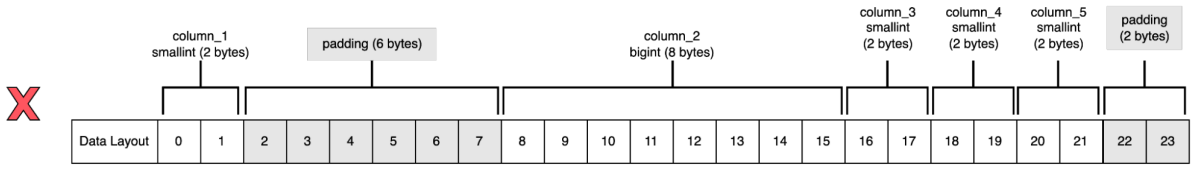
A menor unidade de acesso a todas as memórias é o *byte*. O **alinhamento de dados** se refere à organização dos dados na memória de forma que os endereços estejam alinhados conforme a arquitetura do processador. Processadores modernos frequentemente exigem que os dados sejam armazenados em endereços que são múltiplos de dois. Além disso, o barramento de memória, que conecta o processador à memória principal, é projetado para transferir dados em blocos do mesmo tamanho que a palavra de memória do processador. Quando os dados estão alinhados com o limite de palavra (em inglês *word*), o processador pode acessar uma palavra inteira em uma única operação, utilizando toda a largura do barramento. Isso permite operações de leitura ou escrita em uma única transação, ao contrário dos dados desalinhados,

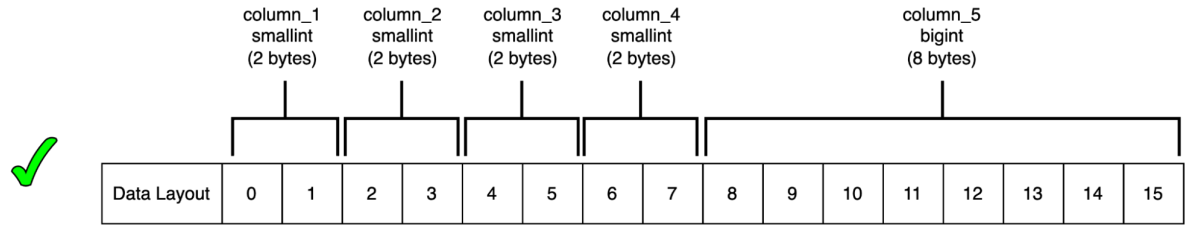
que exigem múltiplas transações. Assim, o alinhamento adequado maximiza a eficiência do acesso à memória e melhora o desempenho geral do sistema.

Quando os dados não estão alinhados corretamente, o sistema precisa adicionar ***padding*** (espaço não utilizado) para garantir o alinhamento adequado. Os *paddings* são *bytes* extras inseridos entre os dados para ajustar o alinhamento. Isso pode causar um aumento significativo no tamanho das estruturas de dados. Por exemplo, se uma estrutura contém um "char" seguido de um "int", o compilador pode adicionar *bytes* de padding entre eles para garantir que o *int* esteja alinhado em um múltiplo de 4 bytes. No entanto, o uso de *padding* pode resultar em desperdício de memória, especialmente em estruturas grandes ou em sistemas com recursos limitados.

Para otimizar o alinhamento e evitar *padding* desnecessário, é fundamental organizar os dados e estruturas de forma a minimizar o desperdício de memória. Isso pode ser feito posicionando os membros maiores das estruturas antes dos menores, alinhando dados de acordo com o tamanho da palavra do processador e utilizando diretivas de alinhamento fornecidas pelo compilador. Além disso, a análise e o ajuste do *layout* das estruturas para garantir que os dados estejam alinhados com os múltiplos adequados pode reduzir a quantidade de *padding* necessário. Técnicas como a reorganização de membros em estruturas e o uso das ferramentas de compilação ajudam a melhorar o uso da memória e a eficiência do sistema.

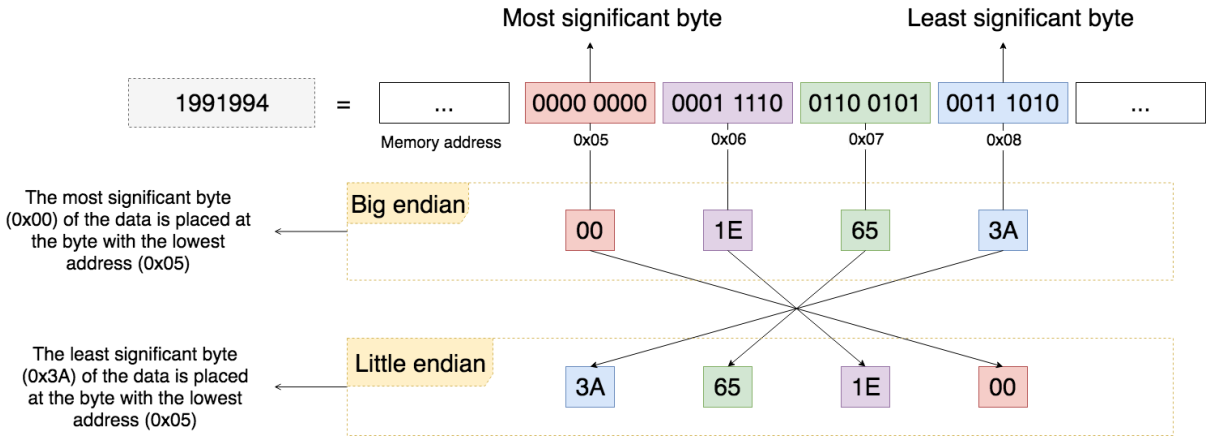
Por exemplo, considere uma tabela contendo quatro colunas do tipo "smallint" e uma coluna do tipo "bigint", se uma coluna "smallint" (2 *bytes*) for seguida por uma coluna "bigint" (8 *bytes*), o sistema adiciona 6 *bytes* de preenchimento após a coluna "smallint" para garantir que a coluna "bigint" esteja alinhada a um limite de 8 *bytes*. No entanto, ao reorganizar as colunas "smallint" para que todas elas precedam a coluna "bigint", é possível otimizar o *layout* da tabela e reduzir a necessidade de preenchimento. Por exemplo, ao posicionar todas as quatro colunas "smallint" antes da coluna "bigint", pode-se minimizar o espaço de preenchimento necessário e usar a memória de forma mais eficiente.





ORDENAÇÃO DOS *BYTES*

Como uma palavra de memória do processador pode conter mais de um *byte*, existem duas estratégias principais para a organização desses *bytes* nos endereços de memória, conhecidas como **ordenação de bytes**. Os termos ***little-endian*** e ***big-endian*** descrevem esses métodos distintos. Em um sistema *little-endian*, o *byte* menos significativo é armazenado no menor endereço de memória, enquanto o *byte* mais significativo ocupa o endereço mais alto. Por exemplo, para o valor 1991994 = 0x001E53A, a organização seria 0x3A no menor endereço e 0x00 no maior. Em contraste, em um sistema *big-endian*, a organização é inversa: o *byte* mais significativo é armazenado no menor endereço e o menos significativo no maior. Assim, para o mesmo valor, 0x00 estaria no menor endereço e 0x3A no maior. A escolha entre esses esquemas é crucial para a interoperabilidade entre diferentes sistemas, pois garante que os dados sejam interpretados corretamente pelo processador ao serem trocados entre sistemas com diferentes ordenações. Compreender e aplicar a ordenação de *bytes* adequada maximiza a eficiência e a compatibilidade no processamento de dados.



SEGMENTAÇÃO DE MEMÓRIA PRINCIPAL

Para organizar e gerenciar eficientemente os recursos de memória em sistemas embarcados, garantindo a integridade e segurança dos dados e instruções em execução, é comum adotar uma abordagem de segmentação na memória responsável por armazenar dados e instruções que o processador precisa acessar rapidamente durante a execução de um programa. A memória utilizada para esse propósito é conhecida como **memória principal**. Em linguagens como C, essa segmentação reflete-se na forma como diferentes tipos de dados e código são alocados e acessados. Cada segmento de memória possui regras e comportamentos específicos para o acesso e manipulação, alinhados com o tipo de dado que ele contém.

A **memória de instruções**, também conhecida como **memória de programa** ou **segmento de texto** (do inglês, ***text***), desempenha o papel crucial de armazenar as instruções do programa em execução. Essas instruções são acessadas sequencialmente pelo processador e interpretadas/executadas para gerar os sinais de controle necessários. Contendo o código do programa, a memória de instruções delineia a sequência de operações a serem executadas pelo processador. Geralmente, essa memória é designada como somente leitura, uma vez que

o código do programa não devem sofrer modificações durante a execução. Em sistemas embarcados, este segmento é geralmente armazenado na memória FLASH, uma vez que a FLASH é uma memória não volátil que retém dados mesmo quando o sistema está desligado.

A **memória de dados** tem a finalidade de armazenar os dados utilizados pelo programa em execução, abrangendo variáveis, vetores, estruturas de dados e outros elementos essenciais para a execução dos programas. Esses dados são passíveis de leitura, escrita e manipulação pelo processador conforme necessário durante a execução do programa, o que torna necessário o mapeamento dessa memória em unidades físicas de memória regraváveis, como a SRAM.

Para uma organização eficiente e gerenciamento otimizado dos diversos tipos de dados armazenados em um programa, são tipicamente delineados segmentos específicos na memória de dados. Entre eles, destacam-se o **bss** (do inglês *Block Started by Symbol*), **rodata** (do inglês *Read-Only Data*), **data**, **pilhas** e ***heap***. Essa distinção facilita a alocação, manipulação e controle de diferentes categorias de dados. Muitas linguagens de programação, especialmente a C, associam diferentes tipos de dados a diferentes regiões da memória de dados.

O **segmento BSS** é projetado para variáveis com uma vida útil que se estende por toda a execução do programa, mas que não são inicializadas explicitamente no código. Essas variáveis são reservadas na memória SRAM sem valores iniciais específicos; em vez disso, um código de inicialização pode ser incluído no próprio código principal para garantir que este segmento seja zerado antes do início da execução do programa. A estratégia aqui é alocar um bloco de dados para todas as variáveis não inicializadas em vez de alocar células de memória individuais para cada uma delas. Essa abordagem permite economizar tempo de alocação e inicialização, uma vez que não é necessário armazenar os valores iniciais dessas variáveis. Tipicamente, este segmento fica armazenado na memória SRAM.

O **segmento Rodata** é usado para armazenar dados somente leitura, como constantes e literais. Esses dados são apenas lidos e não podem ser modificados durante a execução do programa. Ao separar os dados somente leitura em um segmento específico, é possível otimizar o acesso a esses dados e economizar espaço, pois eles não precisam ser duplicados em diferentes partes do programa, podendo ficar no espaço do arquivo executável. Em sistemas embarcados, esses dados são armazenados na memória FLASH, pois eles não mudam durante a execução do programa.

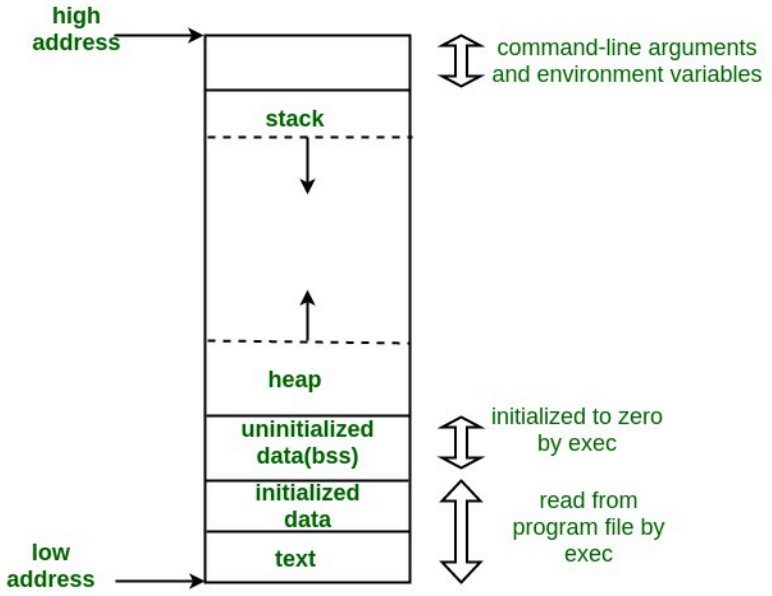
O **segmento Data** é usado para armazenar variáveis que permanecem ativas durante toda a execução do programa. Essas variáveis são inicializadas uma vez e podem ser modificadas conforme o programa avança. O espaço de cada variável é alocado e inicializado individualmente. Tipicamente, esse segmento reside entre o segmento text e o segmento BSS.

As **pilhas** (do inglês ***stacks***) são amplamente usadas em sistemas computacionais para armazenar dados temporários, necessários dentro de um escopo específico de fluxo de

controle, como vimos no Roteiro 3. Tendo suporte em *hardware* na maioria dos processadores, é apenas necessário organizar um bloco contíguo de células na memória RAM e armazenar o endereço do topo do bloco no registrador de ponteiro da pilha SP. Tipicamente, os endereços armazenados no ponteiro de pilha são alinhados em 8 *bits*.

A **memória *heap*** corresponde a um bloco de memória RAM reservado para alocação dinâmica de posições de memória, isto é, alocação de posições durante a execução de um programa. Isso possibilita o compartilhamento de memória entre diferentes partes de um programa, podendo ser a única opção para microcontoladores com grandes restrições de memória. A alocação/desalocação dinâmica de memória não é ainda implementada por circuitos específicos. Ela é implementada por *software* de um sistema operacional, como por exemplo FreeRTOS, ou de uma biblioteca de gerenciamento de memória dinâmica, como em C *Bare Metal*. Por determinismo e por automatização, recomenda-se minimizar o uso de memória *heap* em projetos de sistemas embarcados a nível de *bare metal*. O gerenciamento dinâmico da memória *heap* pode introduzir complexidade imprevisível e afetar a previsibilidade do sistema, o que é crítico para aplicações de tempo real e sistemas com requisitos rigorosos de desempenho.

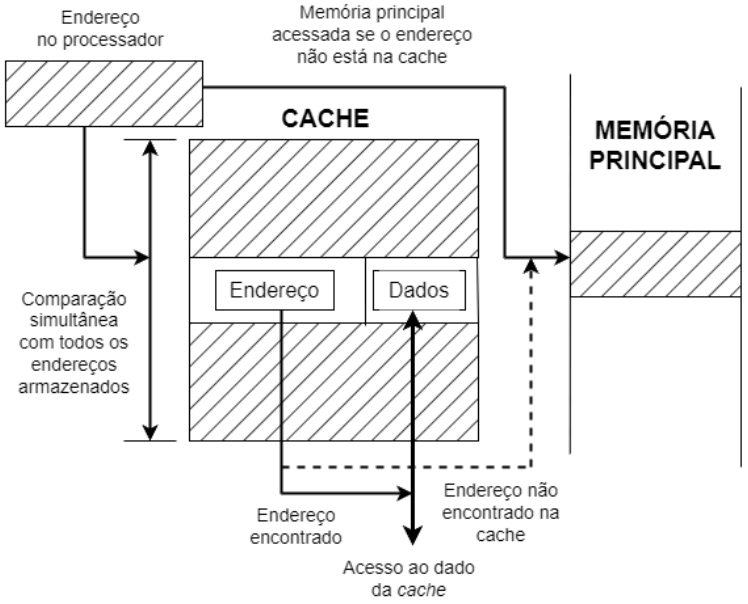
A figura a seguir ilustra a organização dos diferentes segmentos de insrruções e dados na memória, incluindo áreas distintas para código (text), dados estáticos, *heap* e pilha, demonstrando como cada segmento é alocado e gerenciado durante a execução de um programa.



MEMÓRIA *CACHE*

Embora muitos microcontroladores não incluam *cache* devido a considerações de custo e simplicidade, a introdução de *cache* em modelos mais avançados pode trazer significativos ganhos de desempenho e eficiência. O *cache* é particularmente vantajoso em aplicações mais complexas ou exigentes, onde a redução da latência e a melhoria da eficiência energética são importantes. Em comparação com sistemas de alto desempenho, como CPUs que possuem múltiplos níveis de *cache*, os microcontroladores geralmente utilizam uma arquitetura de *cache* mais simplificada. Tipicamente, esses dispositivos incorporam um único nível de *cache* diretamente no núcleo do processador, o que é adequado para atender às suas necessidades mais específicas e restritas.

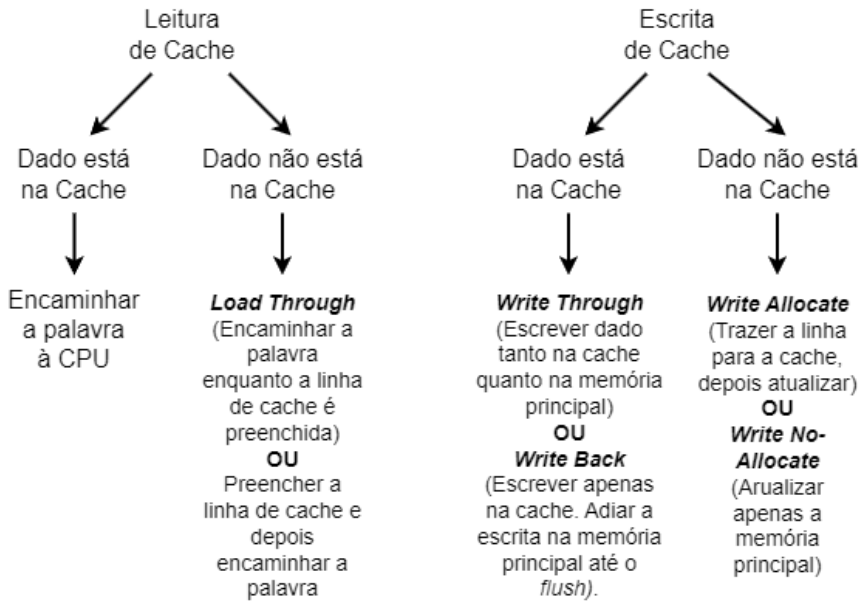
A **memória *cache*** funciona como um "*buffer*" intermediário entre o processador e a memória principal. Baseando-se em estimativas e suposições sobre o programa em execução, realiza-se uma cópia das instruções e dados necessários no *cache*. Assim, quando o processador acessa uma instrução ou dado, espera-se que ocorra um ***cache hit***, ou seja, que a instrução, ou o dado, já esteja presente no *cache*. Se a maioria dos dados requisitados estiver no *cache*, a latência média de acesso à memória de instruções ou dados será reduzida em comparação com um sistema sem *cache*. No entanto, se o dado solicitado não estiver no *cache*, ocorre um ***cache miss***. Nesse caso, o processador deve buscar as instruções ou os dados na memória principal, o que demanda mais tempo num ciclo de acesso.



A transferência entre a memória principal e o *cache* geralmente ocorre em blocos de instruções e dados, conhecidos como **linhas de *cache***. A expectativa de que ocorra um *cache hit* nos acessos subsequentes é baseada nos princípios de localidade temporal e localidade espacial. O *cache* é projetado para aproveitar essas propriedades, mantendo dados e instruções frequentemente acessados ou adjacentes no *cache* para reduzir a latência de acesso e melhorar a eficiência do sistema. A **localidade temporal** garante que dados ou instruções

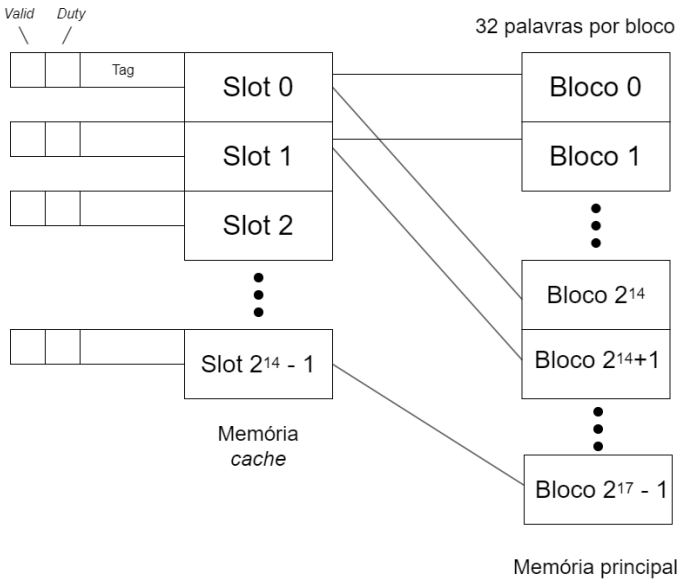
recentemente acessados tenham uma alta probabilidade de serem reutilizados, enquanto a **localidade espacial** garante que dados próximos ao recentemente acessado também sejam carregados e disponíveis para acessos futuros. Com isso, o processador passa menos tempo esperando para acessar a memória principal e pode manter uma taxa de execução mais alta.

Observe que, quando o dado é modificado no *cache*, a memória principal deve ser consistentemente atualizada. Usa-se o *bit* "*Dirty"* para mostrar o estado de modificação de cada linha de *cache*. A figura sintetiza as diferentes políticas que podem ser adotadas na leitura e escrita de dados na memória principal com a presença do cache.



Essencialmente, há três tipos principais de mapeamento de blocos de memória principal para o *cache*, conhecidos como métodos de mapeamento de *cache*: *cache* direto (em inglês, *direct-mapped cache*), *cache* associativo por conjunto (em inglês, *set-associative cache*) e *cache* totalmente associativo (em inglês, *fully associative cache*).

O ***cache* direto** é o método mais simples e direto de mapeamento. Neste esquema, cada bloco de memória principal é mapeado para um único local específico no *cache*. Ou seja, cada linha i de *cache*, Slot i, pode armazenar dados de apenas uma linha, Bloco j, de memória principal. A decisão de onde armazenar um bloco de memória principal é feita através de um cálculo direto baseado no endereço da memória, como i = j%m onde m é a quantidade total de linhas no *cache*. Isso facilita a implementação e reduz a complexidade. No entanto, isso pode levar a conflitos se vários blocos de memória principal mapearem para o mesmo local no *cache*, resultando em *cache misses* frequentes.



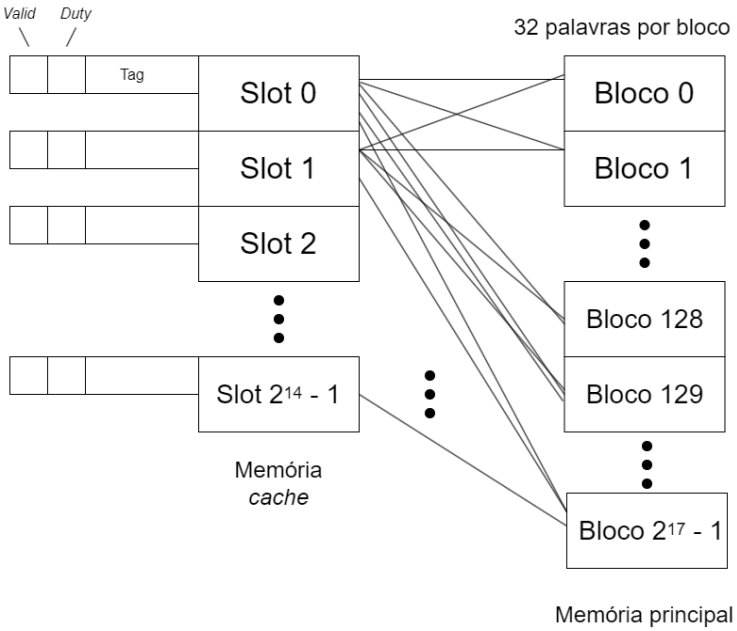
No mapeamento de um endereço de memória em um *cache* direto, divide-se o endereço em três campos: [*Tag*|*Slot*|*Offset*]:

Tag: identifica o bloco de memória principal específico.

Slot: Determina a linha de *cache* onde o bloco deve ser armazenado.

Offset: Indica a posição do *byte* dentro do bloco de dados.

O ***cache* associativo por conjunto** oferece um equilíbrio entre complexidade e flexibilidade. Neste método, o *cache*, composto de m linhas, é dividido em vários conjuntos, cada um contendo k linhas. Cada bloco de memória principal, Bloco j, pode ser mapeado em qualquer uma das k linhas dentro de um conjunto específico. O conjunto é determinado por (j%(m/k)), onde m/k representa o número total de conjuntos no *cache*. O número de linhas por conjunto, k, é conhecido como **associatividade** do *cache*. Por exemplo, um *cache* associativo de 2 vias (em inglês, 2-*way* associative) permite que cada bloco de memória principal, Bloco j, seja armazenado em qualquer uma de duas linhas dentro do conjunto identificado, como ilustra a figura. Esta abordagem reduz os conflitos que podem ocorrer no *cache* direto, pois um bloco de memória principal tem mais opções de armazenamento, o que melhora a taxa de *cache hits*.



O endereço de memória principal é dividido em três campos: [*Tag* | Conjunto | *Offset*]. Nesse esquema:

- ***Tag***: Identifica de forma única o bloco de dados específico na memória principal.
- **Conjunto**: Localiza o conjunto de *cache* onde o bloco pode estar armazenado.
- ***Offset***: Especifica a posição do *byte* dentro do bloco de dados carregado na cache.

A localização de um bloco de memória principal dentro do conjunto de *cache* é feita através da comparação da *Tag* de cada linha de *cache* do conjunto de *cache* com os *bits* mais significativos do endereço de memória.

O **cache totalmente associativo** é o método mais flexível e complexo, permitindo que qualquer bloco de memória principal seja armazenado em qualquer linha do *cache*. Não há restrições quanto ao local onde um bloco pode ser colocado, o que minimiza os conflitos e maximiza a taxa de *cache hits*. No entanto, essa flexibilidade vem com um custo maior em termos de *hardware* e complexidade, pois o sistema precisa verificar todas as linhas do *cache* para encontrar um bloco específico. Isso torna a implementação e o gerenciamento do *cache* mais complexos e dispendiosos.

Devido às suas restrições de custo e complexidade, *caches* totalmente associativos são mais comuns em sistemas de alto desempenho e processadores de servidores, onde o desempenho e a flexibilidade são críticos e o custo de implementação mais complexo pode ser justificado. Em microcontroladores, são mais comuns *caches* direto ou associativo por conjunto, que oferecem um bom equilíbrio entre desempenho e custo, adequados para as necessidades específicas e restrições desses dispositivos.

MEMÓRIA FORTEMENTE ACOPLADA (TCM)

A memória TCM (do inglês *Tightly Coupled Memory*) é uma forma de memória projetada para operar extremamente próxima ao processador, oferecendo baixa latência e alta largura de banda. Essa tecnologia é comumente implementada usando SRAM e se comunica com a CPU de forma direta e paralela, sem intermediários como controladores de memória que poderiam adicionar latência.

A principal motivação para a inclusão da TCM em microcontroladores é a necessidade de acesso rápido e eficiente a dados e instruções críticos. Em aplicações que demandam alto desempenho e baixa latência, como sistemas embarcados em tempo real, a TCM é utilizada para armazenar partes do código e dados frequentemente acessados. A proximidade da TCM ao núcleo do processador reduz significativamente o tempo de acesso à memória, melhorando o desempenho geral do sistema. Isso é alcançado ao evitar a latência associada ao acesso a memórias mais lentas e distantes, como a memória principal ou a memória externa. O diagrama de blocos a seguir ilustra a conexão direta entre o processador e as memórias SRAM e FLASH, que apresentam latências, via um controlador de TCM na arquitetura Cortex-M7.

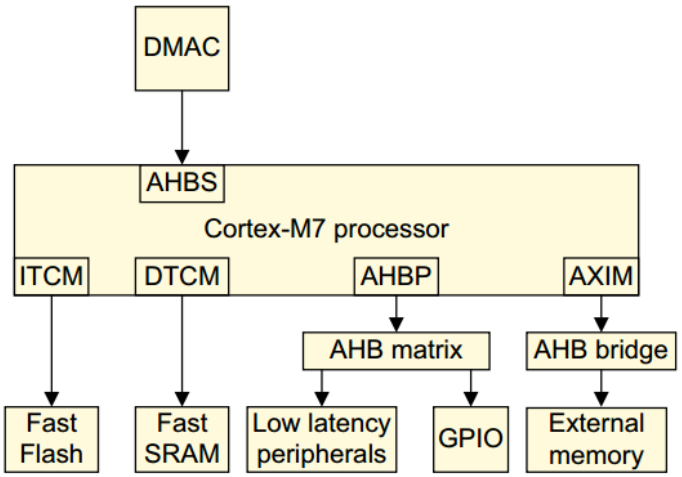


Figure 1-1 Example Cortex-M7 system

SISTEMA DE MEMÓRIA NO STM32H7A3

O seguinte texto extraído do *Datasheet* sumariza as unidades de memória internas e externas suportadas pelo microcontrolador STM32H7A3.

Memories

- Up to 2 Mbytes of flash memory with read while write support, plus 1 Kbyte of OTP memory
- ~1.4 Mbytes of RAM: 192 Kbytes of TCM RAM (inc. 64 Kbytes of ITCM RAM + 128 Kbytes of DTCM RAM for time critical routines), 1.18 Mbytes of user SRAM, and 4 Kbytes of SRAM in Backup domain
- 2x Octo-SPI memory interfaces, I/O multiplexing and support for serial PSRAM/ NOR, Hyper RAM/flash frame formats, running up to 140 MHz in SRD mode and up to 110 MHz in DTR mode
- Flexible external memory controller with up to 32-bit data bus:
  - SRAM, PSRAM, NOR flash memory clocked up to 125 MHz in Synchronous mode
  - SDRAM/LPSDR SDRAM
  - 8/16-bit NAND flash memories
- CRC calculation unit

O microcontrolador STM32H7A3, que utiliza o processador Cortex-M7 da ARM, apresenta um sistema de memória altamente otimizado para maximizar o desempenho em aplicações complexas. A arquitetura do Cortex-M7, lançada pela ARM em 2014, é a primeira da série Cortex-M a incluir *cache* de memória de um nível, marcando uma evolução significativa em relação aos modelos anteriores. Esta arquitetura adota uma variante da arquitetura Harvard, conhecida como Harvard modificada, que incorpora *cache* separado para instruções e dados. Esse desenho de projeto melhora o desempenho ao reduzir conflitos e otimizar o acesso às informações armazenadas, com cada tipo de *cache* especializado para o tipo de dado que armazena.

O Manual de Cortex-M7 mostra que o *cache* de dados é configurado como um *cache* associativo por conjunto de quatro vias, enquanto o *cache* de instruções é um *cache* associativo por conjunto bidirecional. Ambos os *caches* utilizam linhas de 32 *bytes*, o que permite que cada bloco de memória principal seja carregado em uma linha de *cache*. Quando todas as linhas de *cache* em um conjunto estão ocupadas e um novo bloco precisa ser carregado, o controlador de *cache* deve substituir uma linha existente para abrir espaço para o novo bloco de dados ou instruções. Os dados e instruções armazenados no *cache* são recuperados da memória externa através da interface AXI Master (AXIM), e os controladores de *cache* utilizam SRAMs integradas ao processador para armazenar temporariamente essas informações durante a operação.

O STM32H7A3 inclui a memória TCM. De acordo com o Manual do Cortex-M7, a arquitetura ARMv7E-M suporta acesso direto a essa memória através da interface AHBS (do inglês *Advanced High-performance Bus Slave*). Isso permite acesso quase instantâneo a dados e instruções críticos. Para garantir a integridade dos dados no sistema de memória, ajudando a detectar e prevenir erros de dados e aumentando a confiabilidade do sistema, o microcontrolador é provido de uma unidade CRC (do inglês *Cyclic Redundancy Check*) que é responsável por calcular e verificar o código CRC dos dados armazenados na memória.

No ambiente integrado de desenvolvimento STM32CubeIDE, o *layout* da memória e a alocação de espaço para diferentes segmentos, como instruções e dados, são definidos por meio de um *script* de *link*, como vimos no Roteiro 2. Esse *script*, com a extensão ".ld", determina como a memória é organizada e onde cada tipo de dado e código será armazenado. No entanto, a pré-carga da memória TCM, bem como a configuração e ajuste dos *caches* de

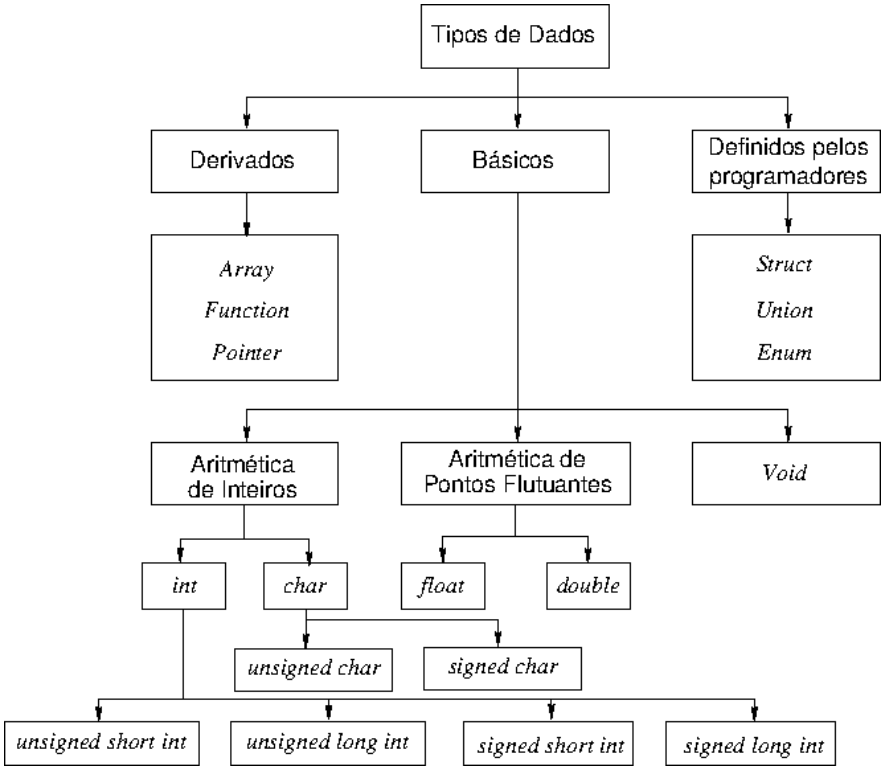
instruções e dados, são realizadas no código-fonte do *firmware* do desenvolvedor. Isso envolve a configuração dos registradores específicos do processador para ajustar os parâmetros do sistema de memória conforme necessário.

O microcontrolador também oferece pinos para acesso a memórias externas, com modos de operação que podem ser ajustados para otimizar o desempenho conforme as necessidades específicas da aplicação. Entre esses modos, destacam-se o SRD (do inglês *Single Read Data*) e o DTR (do inglês *Dual Transfer Rate*). No modo SRD, é possível transferir um dado por ciclo de *clock*, adequado para operações básicas, enquanto o modo DTR permite a transferência de dois dados por ciclo de relógio, oferecendo uma largura de banda significativamente maior e melhorando a eficiência do sistema.

TIPOS DE DADOS EM C

No contexto da programação, a memória do computador é um grande bloco contíguo de *bytes*, e esses *bytes* representam os dados que o *software* manipula. No entanto, a memória bruta, composta apenas por *bytes*, não é diretamente compreensível ou útil em termos práticos. Os tipos de dados fornecidos pelas linguagens de programação transformam esses blocos de *bytes* em formas mais legíveis e úteis. Cada tipo de dado é uma maneira específica de interpretar e organizar esses *bytes*, facilitando o acesso e a manipulação das informações.

Do ponto de vista computacional, os **tipos básicos** de dados processados pelo processador são classificados em quatro categorias principais: números inteiros sem sinal, números inteiros com sinal, números de ponto flutuante e caracteres alfanuméricos. Os caracteres alfanuméricos incluem letras do alfabeto (tanto maiúsculas quanto minúsculas), os 10 dígitos arábicos e uma variedade de caracteres especiais, como hífen, travessão, ponto e vírgula. Existem várias abordagens para realizar essa conversão, e vamos explorar as mais comuns e amplamente utilizadas nos sistemas computacionais modernos.



Números Naturais ou Números Inteiros sem Sinal: O sistema de numeração decimal, amplamente conhecido, utiliza dez dígitos (0 a 9) e adota a notação posicional, onde o dígito mais à direita tem o menor valor e cada posição vale 10 vezes mais do que a posição imediatamente à direita. No entanto, este sistema não se adapta à natureza binária dos computadores. Em contraste, o sistema binário, que usa apenas dois dígitos (0 e 1), é mais adequado para a representação em computadores. Nesse sistema, cada dígito (ou *bit*) tem um valor que é o dobro do dígito à sua direita, seguindo a notação posicional.

O sistema binário é empregado para codificar números naturais ou inteiros sem sinal. Com n *bits*, é possível representar até 2^n valores distintos, variando de 0 até 2^n - 1. Dado que a menor unidade de armazenamento na maioria dos computadores é um *byte* (8 *bits*), a representação de números inteiros sem sinal geralmente é expressa em termos de *bytes*, podendo ser 1 *byte*, 2 *bytes,* 4 *bytes* ou 8 *bytes*, conforme a necessidade.

Para facilitar a manipulação e a leitura de grandes sequências binárias, os *bits* são frequentemente agrupados em blocos que correspondem a potências de 2, como 4 *bits* (nibble), 8 *bits* (*byte*), e 16 *bits*. Entre esses sistemas, a notação hexadecimal é particularmente útil em sistemas embarcados. O sistema hexadecimal, com base 16, utiliza os dígitos de 0 a 9 e as letras de A a F, que correspondem aos valores decimais de 10 a 15. Cada dígito hexadecimal representa um *nibble* (4 *bits*), facilitando a conversão entre binário e hexadecimal e tornando a interpretação dos dados mais eficiente.

Números Inteiros com Sinal: Números inteiros com sinal incluem tanto os números positivos quanto os negativos. Para representar esses números em binário destacam-se quatro diferentes formas.

A primeira delas é denominada representação de **sinal e magnitude**, em que um *bit*, usualmente o *bit* mais significativo, é reservado para representar o sinal do valor. Por convenção, 0 representa o sinal "+", correspondendo a um número positivo, e 1 representa "-", o que corresponde a um número negativo. Esse *bit* é chamado de **bit de sinal**. Cabe ressaltar que o restante da sequência de *bits*, tanto para um número positivo quanto para um negativo, é usada para o código binário do módulo do valor.

A segunda representação é conhecida como **complemento de um**, em que todos os números negativos são gerados a partir da sua contrapartida positiva, complementando *bit* a *bit* todos os seus *bits*. Ou seja, troca-se os 0's por 1's, e vice-versa. Esta representação tem a desvantagem de que há dois códigos binários associados ao número 0. Por exemplo, os dois códigos binários 0000 e 1111 representam o valor zero para um conjunto de números representado por 4 *bits*.

A terceira alternativa é a representação por **complemento de 2**, que resolve o problema da representação em complemento de 1 e que também facilita a realização de operações aritméticas sobre os números inteiros. A notação de complemento de 2 de um número negativo é obtida ao somar 1 à sua representação em complemento de 1. Com n *bits* nesta notação, pode-se representar os valores de -2^{n-1} até $2^{n-1} - 1$. O número 0 possui representação única e o *bit* mais significativo é o *bit* de sinal. É adotada a mesma convenção

das outras duas representações para interpretar o *bit* de sinal: se 0, então o número é positivo, senão o número é negativo. O interessante é que se quiser converter um número negativo em complemento de 2 para a sua contrapartida positiva, também em complemento de 2, basta complementar bit a bit o número e somar 1 ao resultado. Um problema observado nesta representação é a ordenação dos números pelos seus códigos binários, pois uma simples comparação de *bits* pode levar a conclusões erradas se não forem considerados o *bit* de sinal e os valores absolutos dos números.

A quarta representação é a representação de **excesso-N**. Esta representação usa o valor N como um valor de deslocamento (polarização), de forma que o código binário de N corresponde ao número 0 e o código binário com todos os *bits* zerados corresponde a −N. Desta forma, pode-se ordenar os números negativos e positivos comparando diretamente os seus códigos binários. Por exemplo, para o valor em decimal 35, a sua representação em excesso-128 é 35+128 = 163.

Pontos flutuantes: É importante notar que a quantidade distinta de valores que um conjunto de n *bits* pode representar de forma única é limitada a 2^n. No entanto, para representar uma gama infinita de valores dentro de um intervalo real, utilizamos a **representação em ponto flutuante**. De acordo com o padrão IEEE 754, um número em ponto flutuante é descrito em notação científica por três componentes principais: **sinal**, **expoente** e **mantissa**.

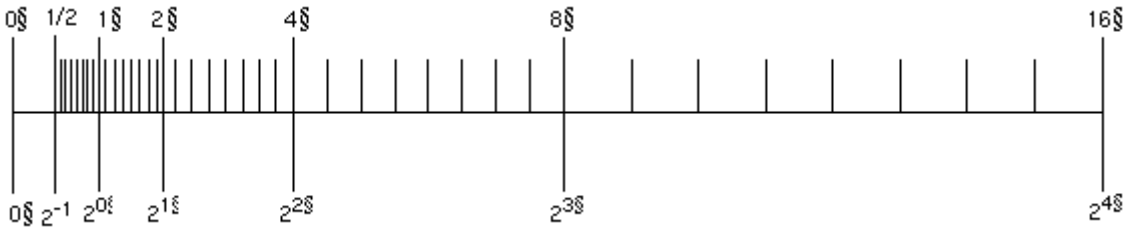
S - sinal	E - expoente	F(ração) - mantissa

O campo de sinal ocupa um *bit* e determina se o número é positivo ou negativo, similar às representações de números inteiros com sinal. Os campos de expoente e mantissa variam conforme a precisão da representação. O padrão IEEE 754 define dois formatos principais: precisão simples (32 *bits* ou 4 *bytes*) e precisão dupla (64 *bits* ou 8 *bytes*). Na precisão simples, o expoente é representado por 8 *bits* e a mantissa por 23 *bits*; na precisão dupla, o expoente é representado por 11 *bits* e a mantissa por 52 *bits*.

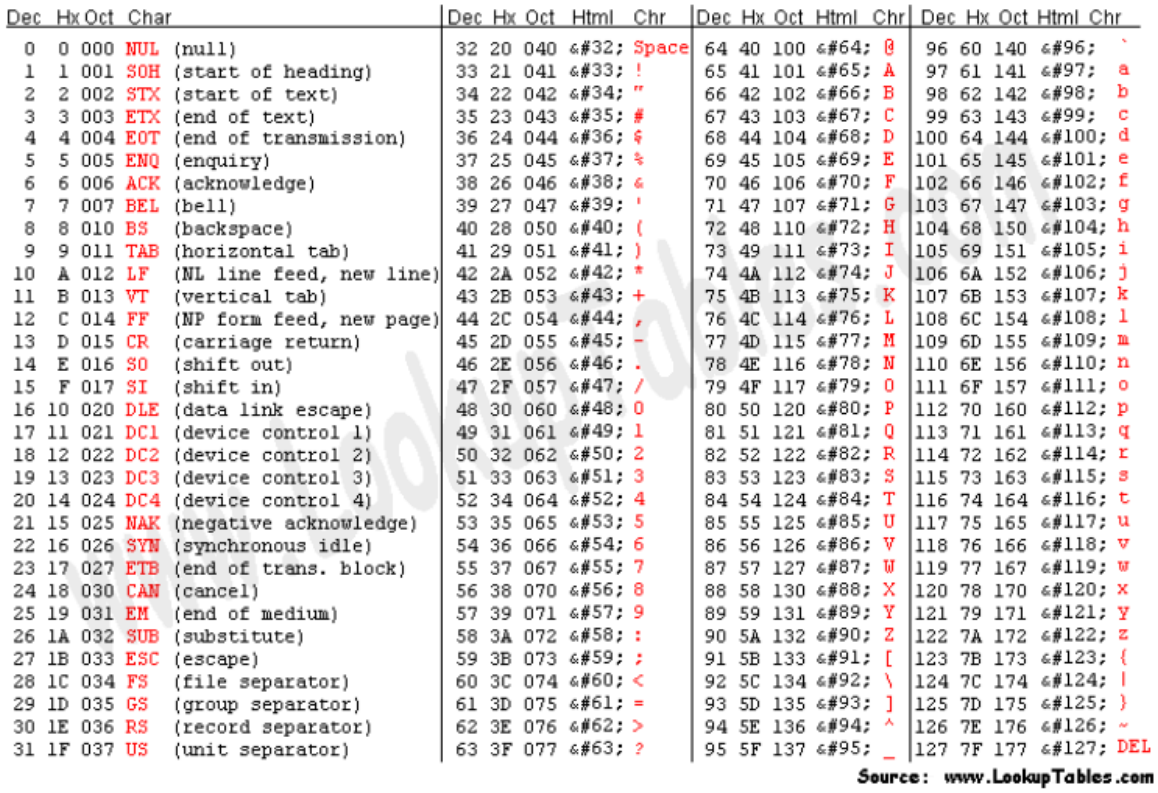
Na representação IEEE 754, os valores são organizados em intervalos que se estendem ao longo da reta numérica real. A quantidade de *bits* alocados para a mantissa determina a precisão dentro de cada intervalo, ou seja, quantos valores distintos podem ser representados dentro de um intervalo específico. Por outro lado, a quantidade de *bits* destinados ao expoente define quantos desses intervalos estão disponíveis ao longo da reta real, com os valores do expoente determinando a escala e a separação entre os intervalos. À medida que o valor do expoente aumenta, a distância entre os intervalos também aumenta exponencialmente, mantendo simetria em relação ao zero na reta real.

O termo “ponto flutuante” se refere à capacidade de ajustar a posição da vírgula decimal, através do expoente, permitindo a acomodação de números em diferentes ordens de magnitude. Assim, a vírgula “flutua” para se ajustar ao tamanho do número, facilitando a representação de uma ampla gama de valores com diferentes magnitudes. Essa flexibilidade na posição do ponto decimal facilita a representação precisa de números muito grandes e

muito pequenos, como ilustrado na figura com os números representáveis ao longo da reta real.



A representação de caracteres alfanuméricos se refere à maneira como letras do alfabeto (maiúsculas e minúsculas), números e alguns caracteres especiais são codificados em sistemas computacionais. Normalmente, isso é feito atribuindo a cada caractere um código numérico único, que é armazenado e processado em forma binária (sequência de 0s e 1s). Os sistemas de codificação mais comuns para caracteres alfanuméricos incluem o código ASCII (*American Standard Code for Information Interchange*), o ASCII estendido ou EBCDIC (sigla de *Extended Binary Coded Decimal Interchange Code*) e o Unicode (sigla de *Universal Coded Character Set*). No código ASCII, por exemplo, cada caractere é representado por um número inteiro de 7 ou 8 *bits*, permitindo a representação de 128 ou 256 caracteres diferentes, respectivamente. A tabela abaixo sintetiza os 128 códigos ASCII dos caracteres de controle e caracteres alfanuméricos representados em 7 *bits*. O código ASCII estendido expande essa codificação, reservando valores adicionais dentro do intervalo de 8 *bits* (geralmente de 0 a 255) para representar caracteres adicionais, símbolos especiais, caracteres acentuados, letras acentuadas usadas em várias línguas e outros caracteres que não estão presentes no conjunto ASCII original de 128 caracteres. Entre as diversas variantes do código ASCII estendido está o código o Latin-1 (ISO 8859-1), amplamente usado para línguas européias. O código Unicode é mais abrangente e permite a representação de um conjunto muito maior de caracteres de várias línguas e símbolos do mundo todo, usando codificações de 8, 16 ou 32 *bits*.



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

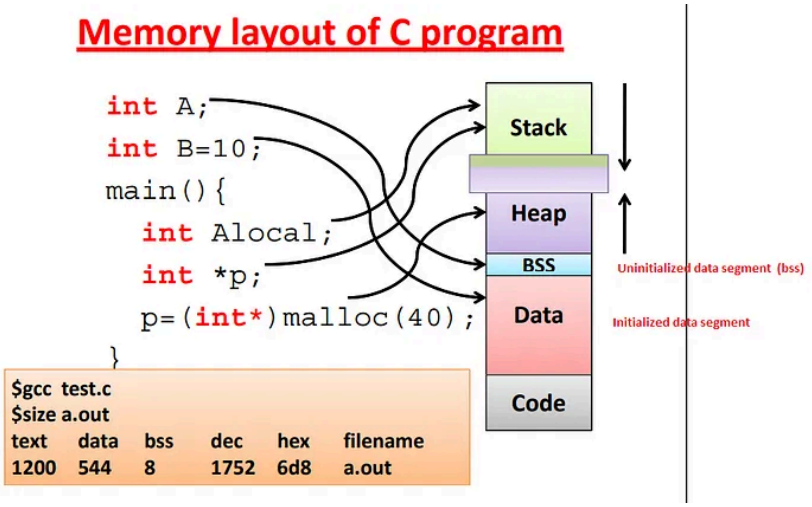
Em C, a representação de valores numéricos e caracteres alfa-numéricos está diretamente vinculada aos **tipos de dados**, influenciando o tamanho da memória necessária, enquanto os qualificadores de tipo determinam o intervalo de valores e a representação dos dados. E a alocação da memória e a duração da sua alocação são mais diretamente afetadas pelo escopo da variável (local, global, estática) e pelo método de alocação (pilha, *heap*, alocação estática).

Tipos de dados numéricos como “int”, com seus qualificadores “signed”, “unsigned”, “short” e “long”, assim como “float” e “double”, são utilizados para armazenar valores inteiros e de ponto flutuante. Cada tipo possui uma representação binária específica que define seu tamanho e precisão na memória, sendo a representação de números inteiros com sinal geralmente feita por meio do complemento de dois. Por outro lado, caracteres alfa-numéricos são representados pelo tipo “char”, que armazena caracteres individuais usando códigos ASCII ou Unicode, convertendo letras e números em valores inteiros manipuláveis. A conversão entre números e caracteres é facilitada por funções de biblioteca e operadores, permitindo a transformação de dados entre diferentes formatos. A escolha do tipo de dado em C não só determina o formato de armazenamento e manipulação dos valores, mas também impacta como os dados são interpretados e utilizados pelo programa.

Além disso, a relação entre variáveis e memória em C é determinada **pelo escopo e pela vida útil** das variáveis. **Variáveis locais** são alocadas na pilha (*stack*) e têm escopo e vida útil restritos ao bloco de código da função em que são declaradas, como discutido no Roteiro 3. **Variáveis globais e estáticas**, por sua vez, são armazenadas no segmento de dados (“data”), se inicializadas, ou no segmento de não inicializados (“bss”), se não inicializadas,

proporcionando um escopo global e uma vida útil que se estende por toda a execução do programa. Já as **variáveis alocadas dinamicamente**, criadas com "malloc()" e "calloc()", são armazenadas na memória *heap* e permanecem alocadas até serem explicitamente liberadas com "free()", independentemente do escopo das funções que as criaram. Em contraste, as **variáveis de alocação estática** têm sua alocação definida no tempo de compilação. Além disso, **constantes e literais** em C são armazenados no segmento de dados somente leitura ("Rodata"), que é geralmente alocado junto com as instruções do programa. Esse segmento é projetado para conter dados que não são modificados durante a execução, garantindo a integridade e a proteção dessas informações.

A figura abaixo sumariza a relação entre os tipos de dados declarados em C e a sua alocação nos diferentes segmentos de memória.



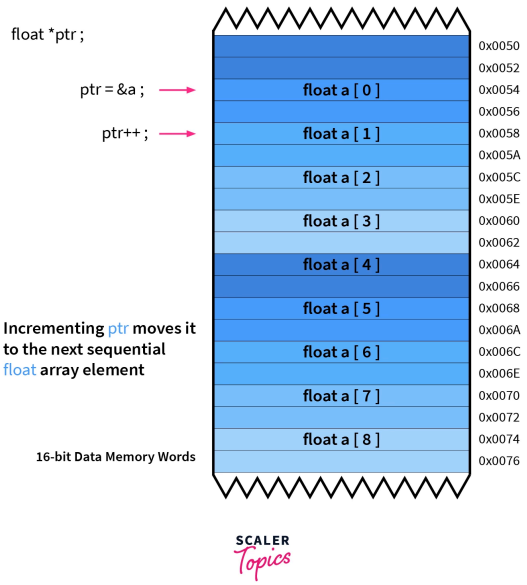
Por fim, é uma estratégia comum em programação de baixo nível, especialmente em desenvolvimento de sistemas embarcados, mapear nomes mais legíveis aos endereços dos registradores usando o tipo de dado "struct":

1. **Definição de Estrutura**: Define-se uma "struct" que mapeia cada registrador para um membro da estrutura. Cada membro da "struct" representa um registrador específico e é associado a um endereço de memória fixo que o fabricante definiu para esse registrador.
2. **Respeito aos Intervalos de Endereços**: O fabricante do microcontrolador ou processador especifica os endereços de memória que cada registrador ocupa. Esses endereços são geralmente documentados no *datasheet* do componente. Na definição da "struct", esses endereços são usados para inicializar os ponteiros ou *offsets* dos campos da "struct", garantindo que cada membro acesse o registrador correto.
3. **Acesso aos Registradores**: Com a "struct" definida e mapeada no endereço inicial do bloco de registradores de um periférico/módulo, os registradores podem ser acessados usando nomes de membros legíveis em vez de endereços de memória brutos. Isso facilita a leitura e a escrita dos registradores, além de tornar o código mais legível e manutenível.

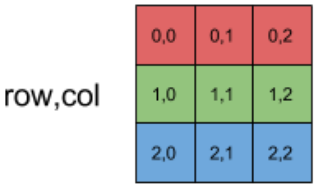
Os tipos básicos de dados são diretamente mapeados para a memória como blocos de *bytes*. Por exemplo, um tipo "int" pode ocupar 4 *bytes*, enquanto um "char" pode ocupar 1 *byte*. Os

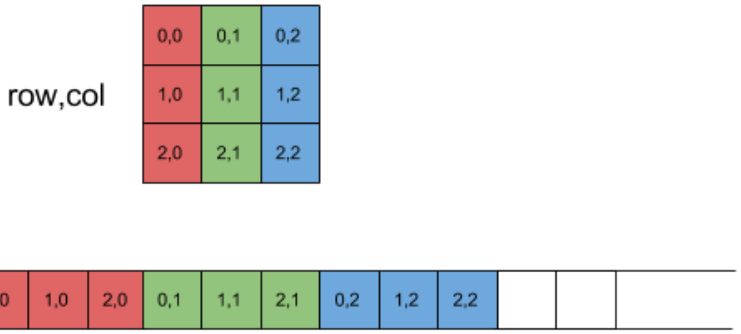
tipos derivados de dados são, por sua vez, construídos a partir dos tipos básicos e permitem representar dados de formas mais complexas e estruturadas. Estes incluem:

- **Vetor (Array)**: Um vetor é uma coleção de elementos do mesmo tipo armazenados sequencialmente em blocos contíguos de memória. Por exemplo, um vetor de tipo float mapeia um bloco contíguoo de memória onde cada float ocupa um espaço fixo. Quando acessamos um elemento do vetor, estamos acessando uma parte específica desse bloco de *bytes*, calculando o deslocamento (*offset*) com relação ao endereço-base do vetor, ou seja o endereço do elemento de índice 0.



- **Matriz**: Uma matriz é uma extensão do vetor para mais de uma dimensão. Quando mapeada na memória, uma matriz é armazenada como um bloco contíguo de *bytes*, mas é interpretada de forma a representar linhas e colunas. Cada elemento da matriz é acessado com base em uma fórmula que considera a sua posição em ambas as dimensões. Isso permite que o mesmo bloco de memória seja visualizado e acessado como uma estrutura bidimensional, ao invés de apenas uma sequência linear, como ilustra a figura.

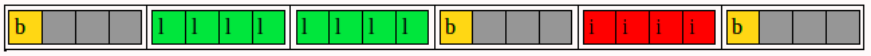




- **Estrutura (“struct”)**: Uma estrutura é uma combinação de diferentes tipos básicos e derivados, agrupados em uma única unidade. Por exemplo, uma estrutura pode conter um “int”, um “char”, e um “float”. Na memória, os campos da estrutura são organizados sequencialmente e podem haver inserções de *padding* para garantir alinhamento, como ilustra a figura.

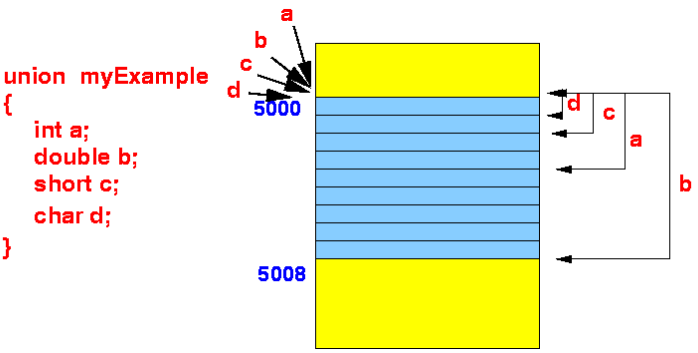
```
struct SRandomOrder
{
    uint8_t     m_byte1;
    uint64_t    m_long;
    uint8_t     m_byte2;
    uint32_t    m_int;
    uint8_t     m_byte3;
};
```

Since the members cannot be re-order by compiler, to achieve proper data alignment this struct will be placed in memory (x86 architecture) like below — padding is presented as gray fields. The `sizeof(SRandomOrder)` is equal to 24 bytes. Note there is only 15 bytes of data and 9 bytes of padding!



Structure with ineffective member placement

- **União (“union”)**: Uma união é um tipo de dado que pode armazenar diferentes tipos de dados na mesma área de memória, mas apenas um tipo de cada vez. Isso significa que, embora uma união possa conter vários membros de diferentes tipos, todos compartilham o mesmo bloco de memória. O tamanho total de uma união é determinado pelo tamanho do maior membro, garantindo que haja espaço suficiente para armazenar qualquer um dos tipos que ela pode conter, como mostra a figura. Assim, a união oferece a flexibilidade de acessar o mesmo bloco de memória de diferentes formas, dependendo do tipo de dado em uso no momento.



A habilidade de mapear um bloco de *bytes* em diferentes tipos de dados nos permite adaptar a nossa representação e processamento dos dados conforme as necessidades específicas do nosso programa. Isso não só facilita a programação, mas também melhora a eficiência ao utilizar a memória de maneira mais eficaz. Por exemplo, em um contexto onde precisamos processar dados sequenciais, um vetor pode ser a melhor escolha. Em outro contexto, onde os dados são melhor representados em uma forma tabular ou matricial, uma matriz pode ser mais adequada. Em ambos os casos, estamos acessando o mesmo bloco de *bytes* na memória, mas o tipo de dado utilizado altera a maneira como interpretamos e manipulamos esses *bytes*.

A regra básica de *padding* adotada pelos compiladores C visa garantir o alinhamento adequado dos diferentes tipos de dados, como ilustra a alocação do *byte*s alocados para os membros de uma "struct" na figura a seguir. Para o tipo "char", os endereços são alinhados por *byte*. Já para o tipo "unit16_t" ("short" nos processadores de 32 *bits*), os endereços são alinhados em endereços pares. No caso dos tipos "uint32_t" ("int" nos processadores de 32 *bits*) e "float", o alinhamento ocorre em endereços múltiplos de 4 *bytes*. Por fim, para o tipo "uint64_t" ("double" nos processadores de 32 *bits*), os endereços são alinhados em múltiplos de 8 *bytes*. O tamanho total do bloco de memória alocado deve ser sempre múltiplo de 4 em um processador de 32 *bits*.