

DISCIPLINA EA701
Introdução aos Sistemas Embarcados

ROTEIRO 2: Linguagem para programação de microcontroladores

**INTRODUÇÃO À PROGRAMAÇÃO DO STM32H7A3 NO STM32CubeIDE
USANDO LINGUAGEM C**

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

FEEC / UNICAMP

Revisado em julho de 2024



This work is licensed under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International. To view a copy of this license, visit

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

INTRODUÇÃO	2
PROJETOS-EXEMPLO	3
Criando um novo projeto em C bare metal	4
Criando um novo projeto em C bare-metal usando CMSIS	15
PYTHON E C	22
DETALHES DA LINGUAGEM C	25
PROGRAMAÇÃO C EM MICROCONTROLADORES	30
TOOLCHAINS	35

INTRODUÇÃO

No primeiro roteiro, vimos como um microcontrolador pode comandar seus periféricos, e como o microcontrolador pode ser programado para executar automaticamente uma sequência de tarefas através de linguagem assembly. Procuramos evidenciar que programar microcontroladores consiste essencialmente em configurar sequencialmente os *bits* dos registradores desses microcontroladores, permitindo que eles executem as funções desejadas. É uma habilidade fundamental para engenheiros e entusiastas da eletrônica, pois possibilita criar sistemas embarcados capazes de realizar uma vasta gama de tarefas.

A programação de microcontroladores pode variar em termos de nível de abstração. A manipulação direta dos *bits*, frequentemente chamada de “escovar os *bits*”, envolve a configuração manual dos *bits* nos registradores do microcontrolador. É a forma mais básica e de baixo nível de executar sequencialmente um procedimento, proporcionando total controle sobre o *hardware*. Embora seja eficiente, pode ser propensa a erros e difícil de manter. A programação em *assembly* é um nível de abstração ligeiramente superior à manipulação direta dos *bits*. Vimos no Roteiro 1 que *assembly* permite escrever instruções que correspondem diretamente às operações humanas, oferecendo um controle preciso sobre o *hardware*. Ela automatiza operações humanas, mas também pode ser complexa e difícil de ler e manter, similar à manipulação direta dos *bits*.

Agora, vamos introduzir uma **linguagem de programação compilada**, conhecida como **C**. Esta linguagem pode ser considerada de nível médio, e é muito adequada para a programação de microcontroladores, pois tem uma sintaxe mais legível e estruturada do que *assembly*, sem perder a capacidade de controlar diretamente os registradores. Além disso, C é amplamente suportada por compiladores e ferramentas de desenvolvimento para microcontroladores. Nos últimos anos, houve esforços significativos para trazer linguagens de alto nível, como Python, para o domínio dos sistemas embarcados. *Frameworks*, como MicroPython e CircuitPython, permitem programar

microcontroladores usando Python, oferecendo uma sintaxe amigável e rápida prototipagem. No entanto, esses esforços enfrentam desafios no estado da arte. Python é uma **linguagem de programação interpretada** e geralmente menos eficiente em termos de tempo de execução e uso de memória em comparação com C. Isso pode ser um impedimento em sistemas embarcados críticos, que exigem alto desempenho e comportamento determinístico. Além disso, o suporte para diferentes microcontroladores e periféricos em Python é mais limitado em comparação com C. Muitos *drivers* e bibliotecas ainda estão em desenvolvimento ou não estão disponíveis.

A relação entre C e Python é mais estreita do que se pode imaginar. As sintaxes das duas linguagens apresentam muitas semelhanças, exceto nos operadores relacionados ao uso da memória. Muitos módulos de Python são implementados em C ou C++. Por ser considerada uma camada que expõe a linguagem C/C++ de forma mais acessível, muitos educadores, como Carl Burch, acreditam que uma boa maneira de introduzir um programador a C é começar com Python. Este roteiro representa uma transição de Python para C.

É importante mencionar que escolher a linguagem de programação certa para microcontroladores depende das necessidades específicas do projeto, do nível de controle requerido e das preferências do desenvolvedor. Enquanto a programação em *assembly* proporciona total controle sobre o *hardware*, a linguagem C oferece um equilíbrio robusto entre controle e facilidade de uso. Por outro lado, iniciativas para utilizar Python estão em andamento e prometem simplificar ainda mais o desenvolvimento, embora ainda enfrentem desafios significativos. Independentemente da linguagem escolhida, o objetivo final permanece o mesmo: configurar os *bits* dos registradores do microcontrolador para que ele realize as tarefas desejadas de maneira eficiente e confiável.

Neste roteiro, exploraremos **como utilizar a linguagem C para programar os *bits* dos microcontroladores e como combinar código em *assembly* com C**. Abordaremos também **como aproveitar os recursos da linguagem para acessar registradores através de mnemônicos e nomes mais legíveis**, em vez de usar diretamente endereços e *bits* dos registradores.

PROJETOS-EXEMPLO

No Roteiro 1, destacamos que, além da programação em linguagem *assembly*, no nível de máquina, é possível utilizar a linguagem C, um nível de programação intermediário, para programar a maioria dos microcontroladores. Neste roteiro, exploraremos como realizar a programação nesse nível e analisaremos o suporte de depuração oferecido por diversas ferramentas profissionais.

Criando um novo projeto em C *bare metal*

Neste projeto, configuraremos o microcontrolador para realizar a mesma tarefa de piscar o LED verde abordada no Roteiro 1, mas utilizando a linguagem C para a programação. Para desenvolver o programa, siga os passos 1 a 7.

1. Deve-se iniciar um novo projeto, exatamente como foi feito no módulo anterior, selecionando a placa NUCLEO e criando o projeto com o nome “PiscaBare”, lembrando de selecionar a opção “Empty” no campo “Targeted Project Type”.

2. A seguir, o IDE entra na perspectiva de Programação. À esquerda, temos o painel “Project Explorer”, no qual podemos ver a estrutura de arquivos do projeto criado. Expanda todas as subpastas clicando nos ícones de seta (‘>’). A subpasta “Includes” geralmente contém arquivos de cabeçalho que são incluídos globalmente em todo o projeto. Por padrão, o IDE inclui arquivos de cabeçalho da biblioteca padrão C e ARM nesta pasta, que fornecem declarações de funções, variáveis globais, macros e tipos de dados.

A subpasta “Inc” é frequentemente utilizada para armazenar arquivos de cabeçalho específicos do projeto, como definições e interfaces relacionadas ao código desenvolvido. A subpasta “Src” contém os arquivos de código-fonte que são compilados durante a construção do projeto. A subpasta “Startup” inclui código-fonte em *assembly*, “startup_stm32h7a3zitxq.s”, responsável pela inicialização básica do microcontrolador, garantindo seu funcionamento correto. A subpasta “Debug” armazena todos os arquivos intermediários relacionados à construção do projeto. Além disso, dois *scripts*, STM32H7A3ZITXQ_FLASH.ld e STM32H7A3ZITXQ_RAM.ld, especificam a organização dos códigos nas memórias Flash e RAM, ou apenas em RAMs, do microcontrolador quando são transferidos para ele.



Dê um duplo-clique no arquivo “main.c” na sub-pasta “Src” para abri-lo no editor de texto interno do IDE.

3. Agora vamos criar um programa em C que faz o LED verde da placa piscar, semelhante ao programa em *assembly* descrito no Roteiro 1. Em vez de usar diretivas de *assembly*, usaremos comandos em linguagem C para configurar os mesmos *bits* nos mesmos registradores. Para isso,

declararemos variáveis para os endereços dos registradores, permitindo que possamos aplicar operações sobre eles. Em vez da diretiva “.word”, declaramos quatro variáveis: RCC_AHB4ENR, GPIOB_MODER, GPIOB_OTYPER e GPIOB_ODR, todas do tipo “uint32_t *” (ponteiro para valores inteiros de 32 *bits*). Essas variáveis são globais, porque são declaradas fora do escopo da função “main”.

Definimos uma macro ITERACOES para representar a constante 500000, que é substituída literalmente no código antes da compilação. Durante a compilação, o compilador C alocará automaticamente um espaço de memória para armazenar o valor da constante 500000. Não é necessário alocá-lo explicitamente como em *assembly*. Note que usamos uma conversão explícita dos valores numéricos dos endereços dos registradores com (uint32_t *), como em (uint32_t *)58024540, para que sejam tratados como endereços de memória. Além disso, o tipo de ponteiro “uint32_t *” é qualificado com “volatile” para indicar que o conteúdo desses registradores pode ser alterado por eventos externos ao fluxo de controle do processador.

Assembly	C
<pre> 104 105 .align 4 106 RCC_AHB4ENR: 107 .word 0x58024540 108 GPIOB_MODER: 109 .word 0x58020400 110 GPIOB_OTYPER: 111 .word 0x58020404 112 GPIOB_ODR: 113 .word 0x58020414 114 ITERACOES: 115 .word 500000 116 .end -- </pre>	<pre> 18 19 #include <stdint.h> 20 21 #if !defined(__SOFT_FP__) && defined(__ARM_FP) 22 #warning "FPU is not initialized, but the project is compiling for an FPU." 23 #endif 24 25 volatile uint32_t *RCC_AHB4ENR = ((uint32_t *)0x58024540); 26 volatile uint32_t *GPIOB_MODER = ((uint32_t *)0x58020400); 27 volatile uint32_t *GPIOB_OTYPER = ((uint32_t *)0x58020404); 28 volatile uint32_t *GPIOB_ODR = ((uint32_t *)0x58020414); 29 30 #define ITERACOES 500000 31 32 int main(void) 33 { 34 /* Loop forever */ 35 for(;;); 36 } 37 </pre>

4. Em termos de operações lógicas, devemos aplicar as seguinte operações sobre o conteúdo dos registradores

RCC_AHB4ENR = RCC_AHB4ENR OR 0x00000002;

GPIOB_MODER = ((GPIOB_MODER OR 0x00000001) AND 0xFFFFFFF0);

GPIOB_OTYPER = GPIOB_OTYPER AND 0xFFFFFFF0;

LED apagado: GPIOB_ODR = GPIOB_ODR AND 0xFFFFFFF0;

LED aceso: GPIOB_ODR |= GPIOB_ODR OR 0x00000001;

Isso pode ser implementado diretamente em C usando as máscaras OR e/ou AND, tornando o código muito mais simples e legível em comparação com a versão em *assembly*:

```
*RCC_AHB4ENR = *RCC_AHB4ENR | 0x00000002;
```

```
*GPIOB_MODER = ((*GPIOB_MODER | 0x00000001) & 0xFFFFFFF0);
```

```
*GPIOB_OTYPER = *GPIOB_OTYPER & 0xFFFFFFF0;
```

```
LED apagado: *GPIOB_ODR = *GPIOB_ODR & 0xFFFFFFF0;
```

```
LED aceso: *GPIOB_ODR |= *GPIOB_ODR | 0x00000001;
```

```
32 int main(void)
33 {
34     *RCC_AHB4ENR = *RCC_AHB4ENR | 0x00000002;
35     *GPIOB_MODER = ((*GPIOB_MODER | 0x00000001) & 0xFFFFFFF0);
36     *GPIOB_OTYPER = *GPIOB_OTYPER & 0xFFFFFFF0;
37
38     /* Loop forever */
39     for(;;) {
40         *GPIOB_ODR = *GPIOB_ODR & 0xFFFFFFF0;
41         *GPIOB_ODR |= *GPIOB_ODR | 0x00000001;
42     }
43 }
```

5. Vamos incluir o bloco de instruções para aumentar o intervalo de tempo entre os dois estados do LED verde usando o comando de laço “for” suportado por C. Este comando precisa de um iterador que deve ser declarado antes do uso. Declaramos uma variável “i” do tipo uint32_t.

```
32 int main(void)
33 {
34     *RCC_AHB4ENR = *RCC_AHB4ENR | 0x00000002;
35     *GPIOB_MODER = ((*GPIOB_MODER | 0x00000001) & 0xFFFFFFF0);
36     *GPIOB_OTYPER = *GPIOB_OTYPER & 0xFFFFFFF0;
37
38     /* Loop forever */
39     uint32_t i;
40     for(;;) {
41         *GPIOB_ODR = *GPIOB_ODR & 0xFFFFFFF0;
42         for (i=0; i<500000; i++); // Gera delay
43         *GPIOB_ODR |= *GPIOB_ODR | 0x00000001;
44         for (i=0; i<500000; i++); // Gera delay
45     }
46 }
```

6. Não se esqueça de salvar as modificações do arquivo (Ctrl-S ou o ícone de salvar). Depois use o ícone de “martelo” para fazer o “Build”, ou seja, compilar os arquivos do projeto, realizar as ligações entre eles e gerar o executável “PiscaBare.elf” usando o *script* “STM32H7A3ZITXQ_FLASH.ld” de configuração do *layout* das instruções e dados na memória. Os passos de construção são mostrados na janela “CDT Build Console.” São 4 comandos de compilação e 1 comando de linkagem.

```

22:46:25 **** Build of configuration Debug for project PiscaBare ****
make -j4 all
arm-none-eabi-gcc -mcpu=cortex-m7 -g3 -DDEBUG -c -x assembler-with-cpp -MMD -MP -MF"Startup/startup_stm32h7a3zitxq.d" -MT"Startup/startup_stm32h7a3zitxq.o" --
arm-none-eabi-gcc "../Src/main.c" -mcpu=cortex-m7 -std=gnu11 -g3 -DDEBUG -DSTM32 -DSTM32H7SINGLE -DSTM32H7 -DSTM32H7A3ZITXQ -DNUCLEO_H7A3ZI_Q -c -I../Inc -O0
arm-none-eabi-gcc "../Src/syscalls.c" -mcpu=cortex-m7 -std=gnu11 -g3 -DDEBUG -DSTM32 -DSTM32H7SINGLE -DSTM32H7 -DSTM32H7A3ZITXQ -DNUCLEO_H7A3ZI_Q -c -I../Inc -O0
arm-none-eabi-gcc "../Src/systemem.c" -mcpu=cortex-m7 -std=gnu11 -g3 -DDEBUG -DSTM32 -DSTM32H7SINGLE -DSTM32H7 -DSTM32H7A3ZITXQ -DNUCLEO_H7A3ZI_Q -c -I../Inc -O0
../Src/main.c:22:4: warning: #warning "FPU is not initialized, but the project is compiling for an FPU. Please initialize the FPU before use." [-Wcpp]
22 | #warning "FPU is not initialized, but the project is compiling for an FPU. Please initialize the FPU before use."
    |
    | ~~~~~
arm-none-eabi-gcc -o "PiscaBare.elf" "objects.list" -mcpu=cortex-m7 -T"E:\Ting\Projects\STM32Cube\EA701\2s24\PiscaBare\STM32H7A3ZITXQ_FLASH.ld" -specs=nos
Finished building target: PiscaBare.elf

arm-none-eabi-size PiscaBare.elf
arm-none-eabi-objdump -h -S PiscaBare.elf > "PiscaBare.list"
   text    data    bss     dec     hex filename
   1096     16    1568    2680    a78 PiscaBare.elf
Finished building: default.size.stdout

Finished building: PiscaBare.list

22:46:33 Build Finished. 0 errors, 1 warnings. (took 7s.620ms)

```

7. Após o “Build”, estamos prontos para carregar o programa na placa seguindo o *layout* definido no *script* STM32H7A3ZITXQ_FLASH.ld (*default*) e executá-lo.

Podemos, no entanto, fazer alguma análise antes da execução. No *script* STM32H7A3ZITXQ_FLASH.ld, a função “Reset_Handler” é definida como o ponto de entrada do programa, ou seja, a primeira instrução a ser executada após um *reset*. O topo da pilha, *_estack*, fica no endereço $\text{ORIGIN}(\text{RAM}) + \text{LENGTH}(\text{RAM}) = 0x24000000 + 2^{10} * 1024 = 0x24000000 + 0x100000 = 0x24100000$. O tamanho mínimo recomendado para a pilha, que armazena variáveis locais, e para o *heap*, que armazena dados alocados dinamicamente durante a execução, são, respectivamente, 0x400 bytes e 0x200 bytes.

```

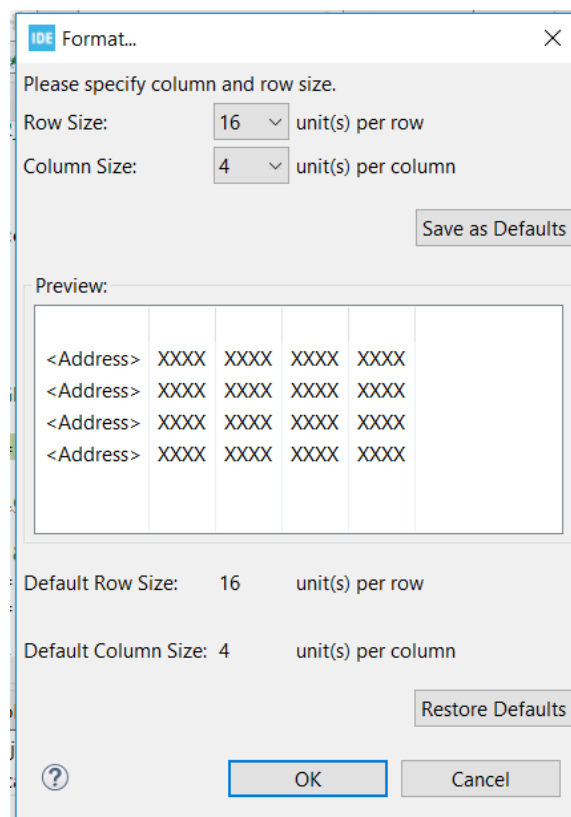
33
34 /* Entry Point */
35 ENTRY(Reset_Handler)
36
37 /* Highest address of the user mode stack */
38 _estack = ORIGIN(RAM) + LENGTH(RAM); /* end of RAM */
39 /* Generate a link error if heap and stack don't fit into RAM */
40 _Min_Heap_Size = 0x200; /* required amount of heap */
41 _Min_Stack_Size = 0x400; /* required amount of stack */
42
43 /* Specify the memory areas */
44 MEMORY
45 {
46     ITCMRAM (xrw) : ORIGIN = 0x00000000, LENGTH = 64K
47     FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 2048K
48     DTCMRAM1 (xrw) : ORIGIN = 0x20000000, LENGTH = 64K
49     DTCMRAM2 (xrw) : ORIGIN = 0x20010000, LENGTH = 64K
50     RAM (xrw) : ORIGIN = 0x24000000, LENGTH = 1024K
51     RAM_CD (xrw) : ORIGIN = 0x30000000, LENGTH = 128K
52     RAM_SRD (xrw) : ORIGIN = 0x38000000, LENGTH = 32K
53 }
54

```

Os endereços do topo da pilha e da primeira instrução são carregados, respectivamente, nos endereços 0x80000000 e 0x80000004 ao carregarmos o código executável para o microcontrolador. Podemos constatar isso se habilitarmos a aba “Memory” (“Window” > “Show View” > “Memory”).

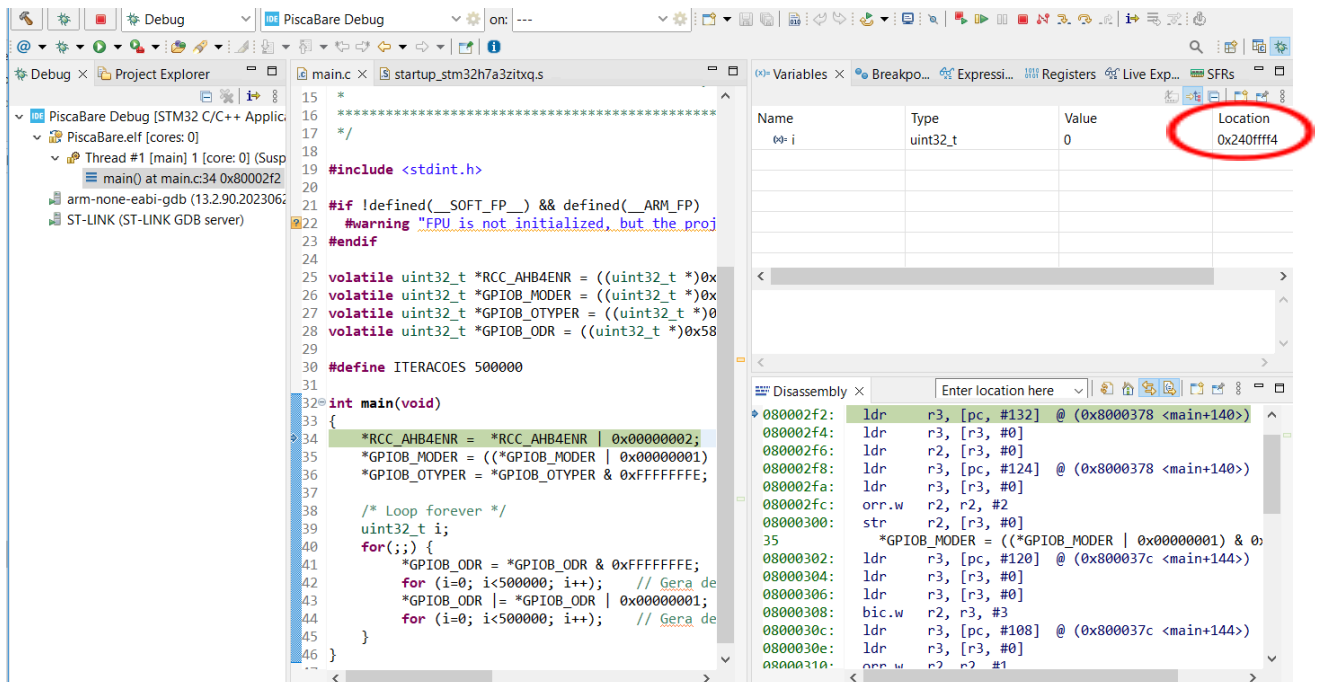
Address	0 - 3	4 - 7	8 - B	C - F
08000000	00001024	A9030008	F9030008	F9030008
08000010	F9030008	F9030008	F9030008	00000000
08000020	00000000	00000000	00000000	F9030008
08000030	F9030008	00000000	F9030008	F9030008
08000040	F9030008	F9030008	F9030008	F9030008
08000050	F9030008	F9030008	F9030008	F9030008
08000060	F9030008	F9030008	F9030008	F9030008
08000070	F9030008	F9030008	F9030008	F9030008
08000080	F9030008	F9030008	F9030008	F9030008
08000090	F9030008	F9030008	F9030008	F9030008
080000A0	F9030008	F9030008	F9030008	F9030008
080000B0	F9030008	F9030008	F9030008	F9030008
080000C0	F9030008	F9030008	F9030008	F9030008
080000D0	F9030008	F9030008	F9030008	F9030008
080000E0	F9030008	F9030008	00000000	F9030008

É possível alterar o formato de apresentação dos *bytes* organizados na memória. Ao clicar qualquer dado renderizado, aparecerá um *pop-up* menu na aba “Memory”. Selecione “Format ...” e surgirá a seguinte janela para configurar a quantidade de *bytes* por linha e a quantidade de *bytes* por coluna. Dê “OK” após a configuração. O *layout* padrão exibe 16 *bytes* por linha, com os *bytes* agrupados em blocos de 4. Para verificar a ordenação dos *bytes* na memória com maior precisão, utilize o agrupamento de 1 *byte* por vez.

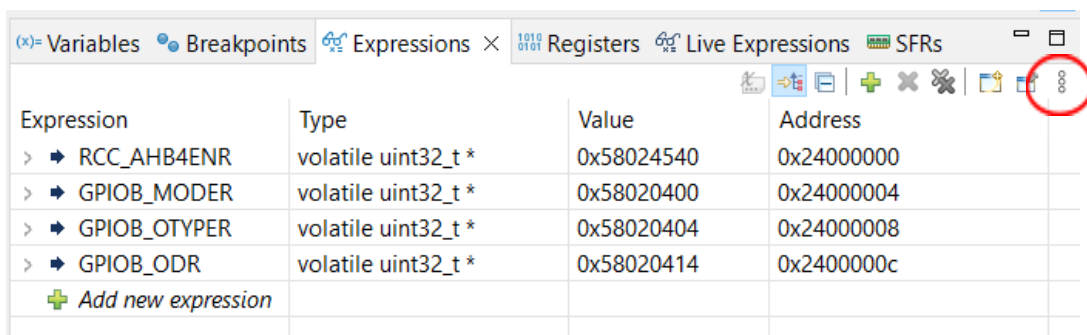


8. Vamos verificar se os endereços das instruções e dados transferidos para o microcontrolador estão condizentes com o *layout* especificado. Para isso, vamos adicionar a janela “Disassembly” (“Window” > “Show View” > “Disassembly”). Nesta janela, observe os endereços das instruções na

primeira coluna, destacados em verde. Estão as instruções armazenadas na memória Flash? E a variável “i”, que é um dado, está armazenada em qual memória? Também observe uma tarja verde tanto na janela do editor quanto na janela “Disassembly”. Essa tarja verde indica a instrução atual em execução, correspondente a um comando em C e sua respectiva instrução em *assembly*.



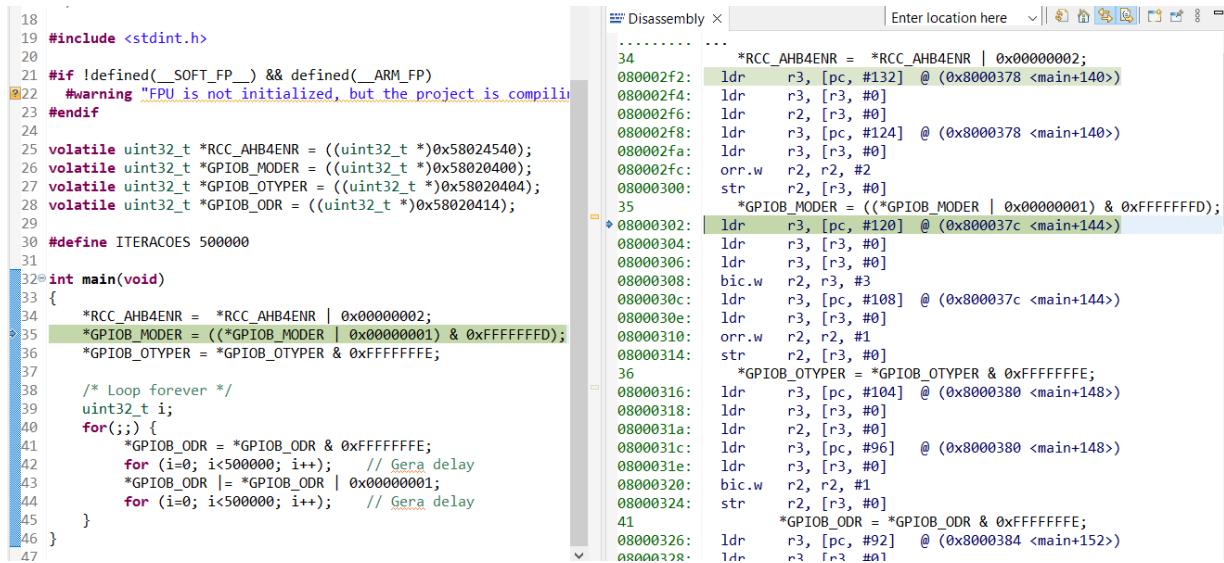
Para monitorar variáveis globais, habilite a aba “Expressions” (“Window” > “Show View” > “Expressions”). Na janela que aparece, há apenas um símbolo “+” em verde, para que novas expressões sejam adicionadas. Assim, pode-se adicionar expressões elaboradas para visualização. Como o projeto foi criado da forma básica, não há suporte para visualização direta dos registradores dos periféricos, e por isso vamos adicionar como expressões os ponteiros criados para acesso aos registradores. Clicando no símbolo de “+”, aparece um espaço para digitar a expressão. Neste espaço, digite “*RCC_AHB4ENR” e dê “Enter”. O ponteiro com seu valor aparecerá na linha. Repita o processo para os outros três ponteiros criados. Para visualizar os endereços dessas variáveis, clique no ícone de três pontos no canto superior direito da aba “Expressions” e habilite “Address” no *pop-up* menu que aparece ao seguir o caminho “Layout” > “Select Columns”. Ao final do processo, deve-se ter a aba de Expressões conforme a figura abaixo.



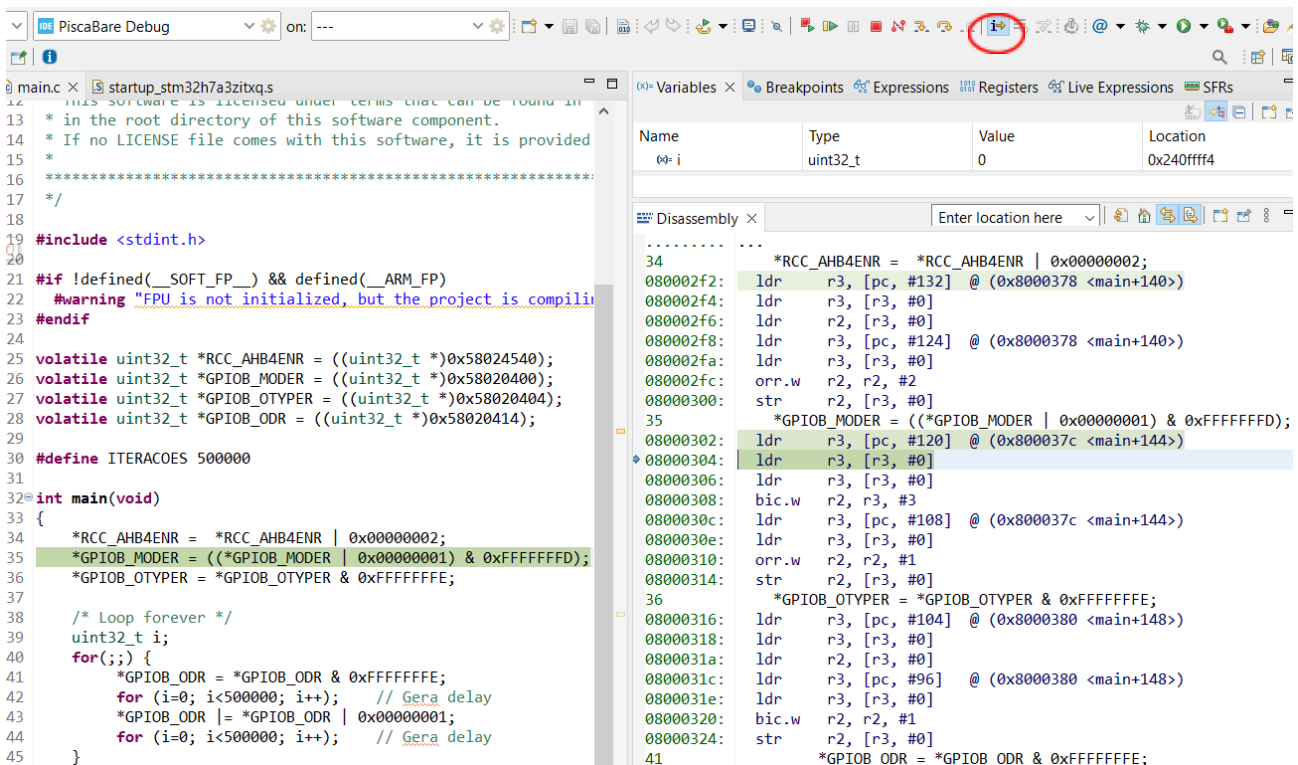
Onde estão armazenadas essas variáveis globais?

9. Vamos explorar dois métodos para monitorar o fluxo de execução. Ao clicar no ícone “Step Over” (ou pressionar F6), observamos o avanço de uma linha de instrução na janela do editor de C.

O bloco de instruções em *assembly* entre duas linhas de código em C, que estão comentadas no código *assembly*, corresponde à tradução da instrução “*RCC_AHB4ENR = *RCC_AHB4ENR | 0x00000002;” em instruções *assembly* pelo compilador C.

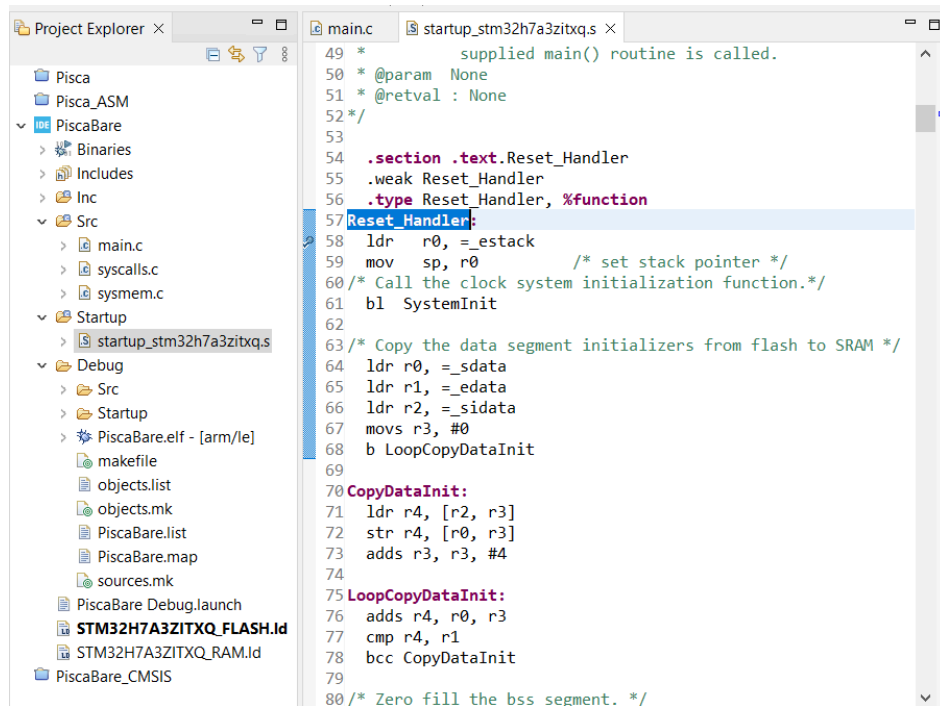


Habilite o modo “Instruction Stepping Mode”, clicando no ícone “i com uma seta amarela”, e avance um passo para frente com um clique no ícone “Step Over”. Em qual janela, “Editor” or “Dissassembly”, foi avançada uma linha de comando?



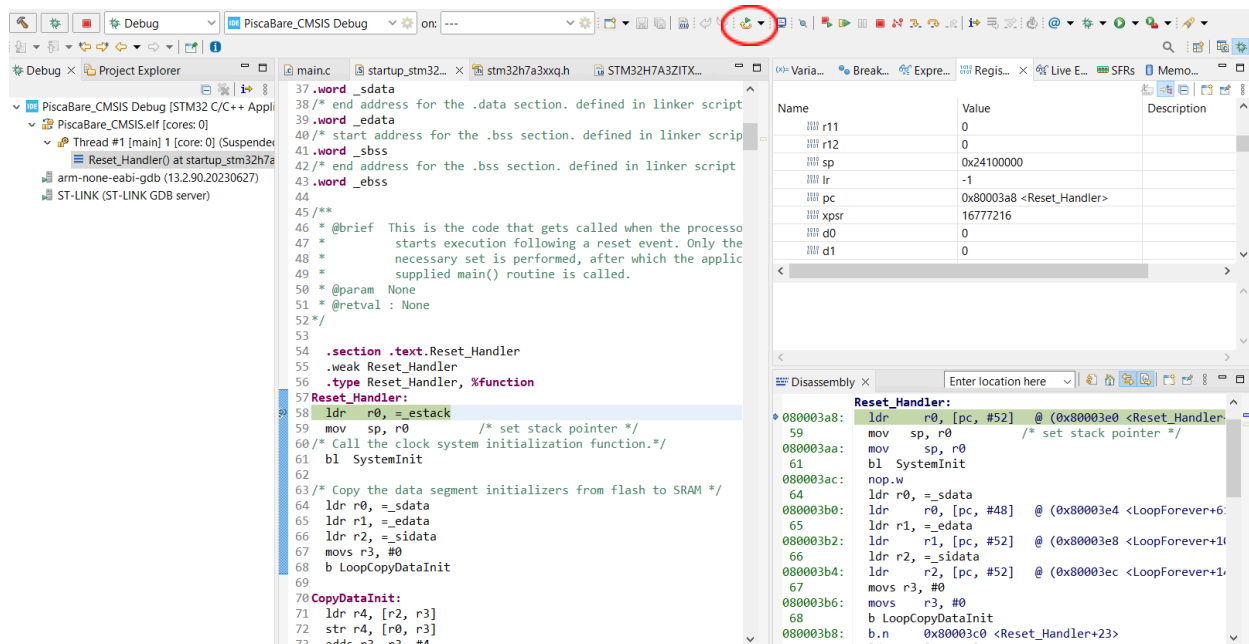
10. Para verificar se os valores dos registradores SP e PC estão de acordo com o *layout* especificado, abra a aba “Registers” e expanda o item “General Registers” clicando em “>”. Como a função “Reset_Handler” está definida no arquivo “startup_stm32h7a3zitxq.s”, retorne à perspectiva de Programação e abra esse arquivo com um duplo clique na aba “Project Explorer”. Na janela de

editor aberta, localize a função “Reset_Handler” em *assembly* e insira um ponto de interrupção na sua primeira instrução.



```
49 *      supplied main() routine is called.
50 * @param None
51 * @retval : None
52 */
53
54 .section .text.Reset_Handler
55 .weak Reset_Handler
56 .type Reset_Handler, %function
57 Reset_Handler:
58     ldr r0, =_estack
59     mov sp, r0      /* set stack pointer */
60 /* Call the clock system initialization function.*/
61     bl SystemInit
62
63 /* Copy the data segment initializers from flash to SRAM */
64     ldr r0, =_sdata
65     ldr r1, =_edata
66     ldr r2, =_sidata
67     movs r3, #0
68     b LoopCopyDataInit
69
70 CopyDataInit:
71     ldr r4, [r2, r3]
72     str r4, [r0, r3]
73     adds r3, r3, #4
74
75 LoopCopyDataInit:
76     adds r4, r0, r3
77     cmp r4, r1
78     bcc CopyDataInit
79
80 /* Zero fill the bss segment. */
```

Volte para a Perspectiva de Depuração e pressione o ícone “Reset the chip and restart the debug session”. Leia o conteúdo do SP e PC na aba “Registers”. Observe que o *bit* 0 do valor do PC, 0x800003A8, é ‘0’, enquanto o *bit* 0 do valor carregado no endereço 0x08000004, 0x800003A9, é ‘1’. Este *bit* determina o modo de processamento de instrução usado para interpretar as instruções buscadas pelo PC em cada etapa do seu progresso: se o *bit* 0 é ‘0’, o processador está em modo ARM (instruções de 32 *bits*); se o *bit* 0 é ‘1’, o processador está em modo Thumb (instruções de 16 *bits*).



Name	Value	Description
r11	0	
r12	0	
sp	0x24100000	
lr	-1	
pc	0x800003a8 <Reset_Handler>	
xpsr	16777216	
d0	0	
d1	0	

Address	Disassembly
080003a8	ldr r0, [pc, #52] @ (0x800003e0 <Reset_Handler>)
080003aa	mov sp, r0 /* set stack pointer */
080003ac	bl SystemInit
080003ad	nop.w
080003b0	ldr r0, =_sdata @ (0x800003e4 <LoopForever+6>)
080003b2	ldr r1, =_edata @ (0x800003e8 <LoopForever+10>)
080003b4	ldr r2, =_sidata @ (0x800003ec <LoopForever+14>)
080003b6	movs r3, #0
080003b8	b LoopCopyDataInit

11. Clique no ícone “Terminate and Relaunch”. Em seguida, “Resume”. O que aconteceu com o LED verde?

12. Vamos agora adicionar novas funções ao arquivo 'main.c' para praticar chamadas de funções. Substituímos o laço “for” por uma chamada de função “void espera(uint32_t valor)”, que passa o parâmetro “valor” por valor. Esta função, por sua vez, invoca a função “void multiplo_iteracoes(uint32_t valor, uint32_t *j)” para calcular o número de iterações que o comando “while” na função “espera” deve executar.

Note que a função “multiplo_iteracoes” possui dois parâmetros: o primeiro é passado por valor, enquanto o segundo recebe o valor de endereço da variável “i”, declarada na função “espera”. Observe também como esses parâmetros são acessados dentro da função. O valor do parâmetro passado por valor de endereço, “j”, é acessado através de “*j”.

Dentro da função “espera” em que a variável “i” é declarada, o valor de endereço desta variável é passada pelo comando “&i”.

```
29
30 #define ITERACOES 5000
31
32 void multiplo_iteracoes (uint32_t valor, uint32_t *j)
33 {
34     *j = valor * ITERACOES;
35     return;
36 }
37
38 void espera (uint32_t valor)
39 {
40     uint32_t i;
41
42     multiplo_iteracoes (valor, &i);
43     while (i) i--;
44 }
45
46 int main(void)
47 {
48     *RCC_AHB4ENR = *RCC_AHB4ENR | 0x00000002;
49     *GPIOB_MODER = ((*GPIOB_MODER | 0x00000001) & 0xFFFFFFF0);
50     *GPIOB_OTYPER = *GPIOB_OTYPER & 0xFFFFFFF0;
51
52     /* Loop forever */
53     for(;;) {
54         *GPIOB_ODR = *GPIOB_ODR & 0xFFFFFFF0;
55         espera (1000);
56         *GPIOB_ODR |= *GPIOB_ODR | 0x00000001;
57         espera (1000);
58     }
59 }
```

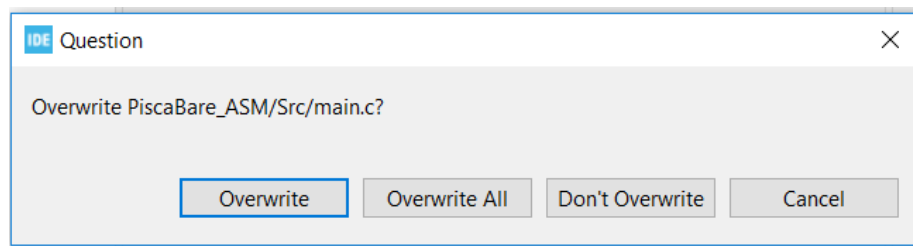
13. Faça “Build” da nova versão e reexecute o projeto no modo “Debug”. Certifique na aba “Disassembly” a integração das duas novas funções “espera” e “multiplo_iteracoes” através dos endereços das duas instruções.

14. Agora altere o programa para que pisque o LED amarelo (ligado em PE1) em vez do LED verde. Deve-se alterar o *bit* do registrador do RCC para ativar GPIOE em vez de GPIOB. Além disso, os *bits* a serem modificados nos registradores do GPIOE se referem ao pino 1 e não mais ao pino 0.

CRIANDO UM NOVO PROJETO EM C *BARE-METAL* E ASSEMBLY

Nem todas as funções de *hardware* têm um comando equivalente direto na linguagem C, como é o caso do deslocamento aritmético ASR. Para superar essas limitações, a linguagem C oferece a diretiva “asm”, que permite a integração de código em *assembly*. Isso possibilita aproveitar as vantagens de ambas as linguagens de forma complementar. Este projeto demonstra essa integração ao realizar a mesma tarefa de piscar o LED verde.

1. Vamos criar um novo projeto, selecionando a placa NUCLEO e criando o projeto com o nome “PiscaBare_ASM”, lembrando de selecionar a opção “Empty” no campo “Targeted Project Type”.
2. Vamos reusar o código “main.c” do projeto “PiscaBare”, sobrescrevendo o arquivo “main.c” com o “main.c” do projeto “PiscaBare”. Para isso, localize o arquivo “main.c” do projeto “PiscaBare” com uso de um explorador de arquivos. Copie o arquivo e vá para a aba “Project Explorer”. Clique o botão direito na sub-pasta “Src” e selecione “Paste” para colar o arquivo copiado. O STM32CubeIDE perguntará se você quer sobrescrever o arquivo existente. Confirme “Overwrite”.



3. Abra o arquivo “main.c” no “Editor” com duplo-clique. Vamos substituir as instruções em C por assembly na função “multiplo_interacoes” usando a diretiva “asm”, ou “__asm” ou “__asm__”, dependendo do compilador. Para não reinventar a roda, vamos analisar como o compilador traduziu essas instruções em *assembly* na aba “Disassembly” da perspectiva de Depuração. Isso requer que construamos e executemos o projeto. Vale ressaltar que o compilador C insere automaticamente instruções para o chaveamento de contexto ao traduzir uma chamada de função. Essas instruções, destacadas pela linha vermelha, incluem o salvamento dos registradores de trabalho (R0-R7, SP, LR e PC), a alocação de espaço na pilha para empilhar variáveis locais na entrada da função, e a recuperação dos valores dos registradores e desempilhamento das variáveis locais na saída da função. Abordaremos esse processo em detalhes mais adiante. O que nos interessa são as 4 instruções no formato de instrução ARM, “mov.w”, “mul.w”, “ldr” e “str”.

```

Disassembly x
Enter location here

080002e6: movs    r4, #0
080002e8: lsls    r4, r3, #17
080002ea: lsrs    r0, r0, #32
multiplo_iteracoes:
080002ec: push    {r7}
080002ee: sub     sp, #12
080002f0: add     r7, sp, #0
080002f2: str     r0, [r7, #4]
080002f4: str     r1, [r7, #0]
080002f6: ldr     r3, [r7, #4]
080002f8: movw    r2, #5000 @ 0x1388
080002fc: mul.w   r2, r3, r2
08000300: ldr     r3, [r7, #0]
08000302: str     r2, [r3, #0]
08000304: nop
08000306: adds    r7, #12
08000308: mov     sp, r7
0800030a: ldr.w   r7, [sp], #4
0800030c: bx      lr
espera:
08000310: push    {r7, lr}
08000312: sub     sp, #16
08000314: add     r7, sp, #0

```

Como nosso microcontrolador suporta apenas o formato Thumb, precisamos ajustar o formato das instruções para que sejam compatíveis com esse conjunto de instruções. Além disso, como não temos acesso direto ao endereço que o compilador alocou para a constante “ITERACOES”, devemos criar uma variável local chamada “i” para armazenar o valor da constante e, em seguida, passar o endereço dessa variável para as instruções em *assembly*.

```

30 #define ITERACOES 5000
31
32 void multiplo_iteracoes (uint32_t valor, uint32_t *j)
33 {
34     /*j = valor * ITERACOES;
35     uint16_t i = ITERACOES;
36
37     asm ("mov r2, %2 \n\t"
38         "mov r3, %1 \n\t"
39         "muls r3, r2 \n\t"
40         "mov %0, r3 \n\t"
41         : "=r"(*j)
42         : "r"(valor), "r"(i)
43         : "r2", "r3"
44         );
45
46     return;
47 }
48

```

4. Faça “Build” e execute o novo projeto.

5. Redefina as funções conforme descrito abaixo e traduza as instruções da nova função “multiplo_iteracoes” para *assembly* embutido em C.

```

multiplo_iteracoes (uint32_t *valor) {
    *valor = *valor * ITERACOES;
}

void espera (uint32_t valor) {
    uint32_t i = valor;
    multiplo_iteracoes (&i);
}

```

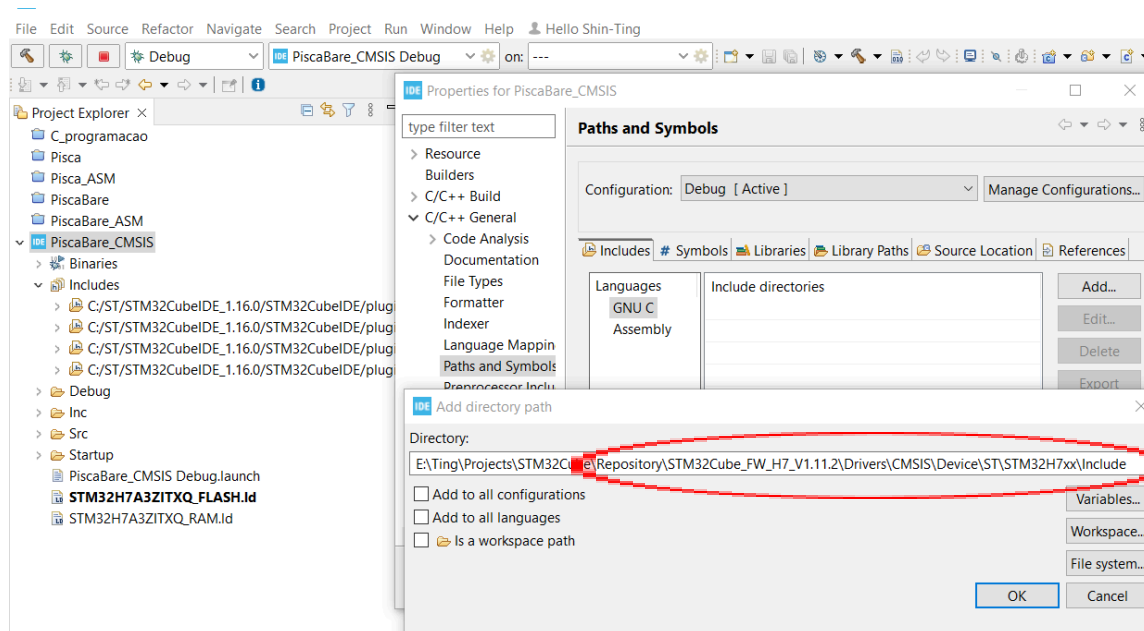
```
while (i) i--;  
}
```

Mude a cor do LED de verde para amarelo. Faça “Build” e execute o projeto. Depois exporte o projeto em um arquivo ZIP, após filtrar com “Clean ...”, para ser incluído no Moodle.

Criando um novo projeto em C bare-metal usando CMSIS

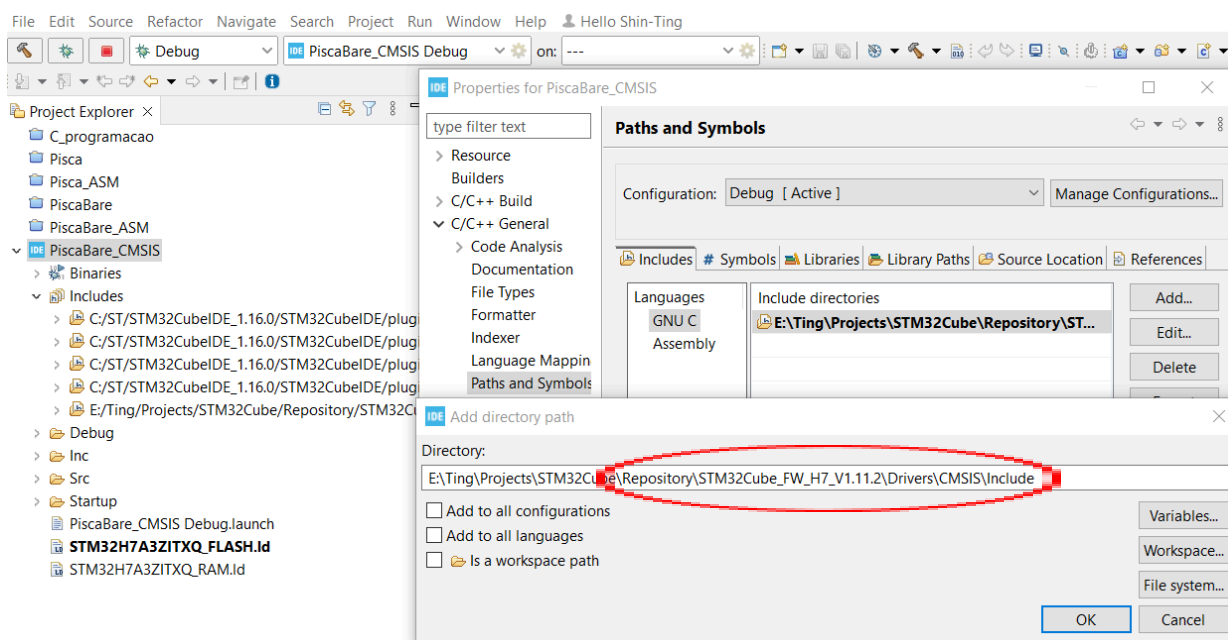
A CMSIS (do inglês *Cortex Microcontroller Software Interface Standard*) desempenha um papel crucial no desenvolvimento de códigos para microcontroladores, oferecendo uma série de vantagens significativas em termos de portabilidade, acesso eficiente ao *hardware* e disponibilidade de recursos pré-desenvolvidos. Esta padronização, promovida por fabricantes de microcontroladores como ARM, busca unificar e simplificar a criação de *software* para diferentes dispositivos, beneficiando tanto desenvolvedores quanto fabricantes. A padronização reflete em diversos níveis, desde o nível do processador até o sistema completo de uma placa de desenvolvimento. Vamos apresentar um projeto em que explora a padronização a nível do microcontrolador STM32H7A3ZIT6-Q.

1. Inicia-se um novo projeto, exatamente como foi feito no módulo anterior, selecionando a placa NUCLEO e criando o projeto com o nome “PiscaBare_CMSIS”, lembrando de selecionar a opção “Empty” no campo “Targeted Project Type”.
2. A seguir, o IDE entra na perspectiva de Programação. Antes de iniciar a edição de um programa em C, vamos adicionar os arquivos de cabeçalho que oferecem mnemônicos padronizados para acessar os registradores e seus campos de *bits*. Vamos para “Project” > “Properties”. Em seguida, selecionamos “C/C++ General” > “Paths and Symbols”. Na aba “Includes”, selecione (“Add”) para incluir o caminho da pasta que contém o arquivo “stm327a3xxq.h”. Geralmente é o caminho para as bibliotecas CMSIS e HAL no repositório fornecido pela STMicroelectronics.



Dê “OK” e “Apply”. É feita, então, a construção automática do índice (C/C++ *index*) de todos os arquivos, definições, declarações, e símbolos presentes no projeto, mantidos pelo STM32CubeIDE, para que todos os arquivos de cabeçalho na pasta passam a ser disponibilizados ao projeto.

3. Quando concluída a reconstrução, aparecerá o caminho inserido na lista da pasta “Includes” na vista “Project Explorer”. Precisamos ainda incluir com “Add” o caminho da pasta que contém o arquivo “core_cm7.h”. Este arquivo contém as definições, declarações e símbolos relacionados ao núcleo Cortex-M7.



Dê “OK”, “Apply and Close” e confirme “Rebuild index”.

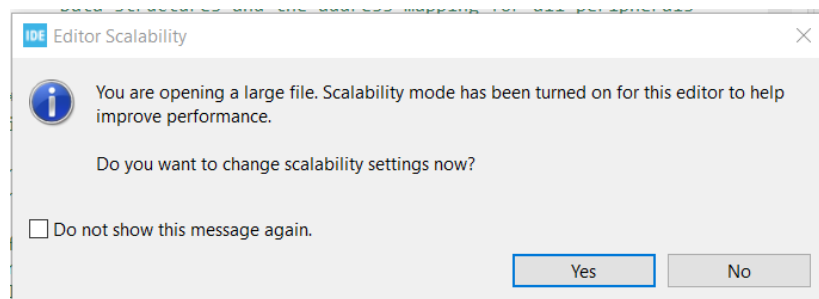
4. Ao final do procedimento, temos duas novas pastas incluídas na pasta “Includes” que nos permitem usar mnemônicos padrão para manipular os registradores do microcontrolador.

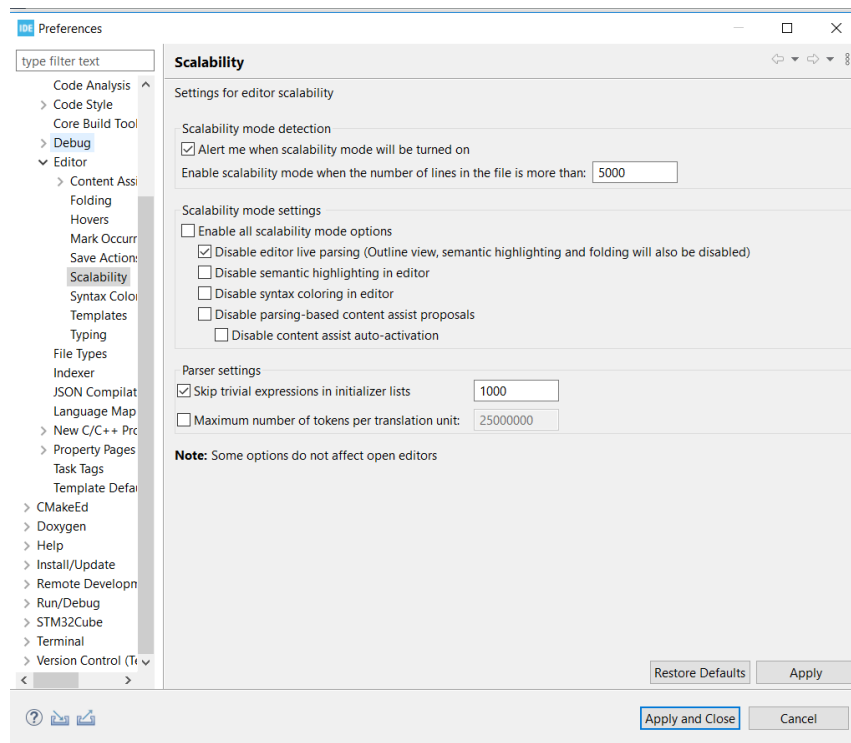


5. Vamos incluir o arquivo de cabeçalho “stm32h7a3xxq.h” no “main.c”. Adicione o comando “#include <stm32h7a3xxq.h>”, conforme a figura abaixo. Depois, clique com o botão direito sobre o nome “stm32h7a3xxq.h” para habilitar um *pop-up* menu. Selecione “Open Declaration” no menu. Alternativamente, pode-se dar um clique simples com o botão esquerdo para colocar o cursor sobre o nome do arquivo e pressionar F3.

```
19 #include <stdint.h>
20
21 #include <stm32h7a3xxq.h>
22
23 #if !defined(__SOFT_FP__) && defined(__ARM_FP)
24 #warning "FPU is not initialized, but the project is compiling for an FPU. Please i
25 #endif
26
27 #define ITERACOES 5000
28
```

Ao tentar abrir o arquivo, aparecerão dois avisos sobre o tamanho do arquivo. Dê “OK” e, em seguida, desmarque a caixa “Disable editor live parsing (...)” e clique em “Apply and Close”.





5. Observando o arquivo “stm32h7a3xxq.h”, podemos ver que todos os módulos X integrados no microcontrolador STM32H7A3ZIT6-Q são abstraídos em estruturas (“structs”), onde cada elemento representa um registrador específico. Essas estruturas são então redefinidas para novos tipos X_DefType utilizando a diretiva “#typedef”. Por exemplo, a estrutura formada pelos registradores do RCC é redefinida como RCC_DefType, enquanto a estrutura dos registradores GPIOx é redefinida como GPIO_DefType

```

1134 uint32_t      RESERVED12;          /*!< Reserved,
1135 __IO uint32_t  AHB3LPENR;          /*!< RCC AHB3 peripheral sleep clock register,
1136 __IO uint32_t  AHB1LPENR;          /*!< RCC AHB1 peripheral sleep clock register,
1137 __IO uint32_t  AHB2LPENR;          /*!< RCC AHB2 peripheral sleep clock register,
1138 __IO uint32_t  AHB4LPENR;          /*!< RCC AHB4 peripheral sleep clock register,
1139 __IO uint32_t  APB3LPENR;          /*!< RCC APB3 peripheral sleep clock register,
1140 __IO uint32_t  APB1LLPENR;         /*!< RCC APB1 peripheral sleep clock Low Word register,
1141 __IO uint32_t  APB1HLPENR;         /*!< RCC APB1 peripheral sleep clock High Word register,
1142 __IO uint32_t  APB2LPENR;          /*!< RCC APB2 peripheral sleep clock register,
1143 __IO uint32_t  APB4LPENR;          /*!< RCC APB4 peripheral sleep clock register,
1144 uint32_t      RESERVED13[4];       /*!< Reserved, 0x120-0x12C
1145
1146 } RCC_TypeDef;

```

```

894 typedef struct
895 {
896     __IO uint32_t  MODER;           /*!< GPIO port mode register, Address offset: 0x00 */
897     __IO uint32_t  OTYPER;          /*!< GPIO port output type register, Address offset: 0x04 */
898     __IO uint32_t  OSPEEDR;         /*!< GPIO port output speed register, Address offset: 0x08 */
899     __IO uint32_t  PUPDR;           /*!< GPIO port pull-up/pull-down register, Address offset: 0x0C */
900     __IO uint32_t  IDR;             /*!< GPIO port input data register, Address offset: 0x10 */
901     __IO uint32_t  ODR;             /*!< GPIO port output data register, Address offset: 0x14 */
902     __IO uint32_t  BSRR;            /*!< GPIO port bit set/reset, Address offset: 0x18 */
903     __IO uint32_t  LCKR;            /*!< GPIO port configuration lock register, Address offset: 0x1C */
904     __IO uint32_t  AFR[2];          /*!< GPIO alternate function registers, Address offset: 0x20-0x24 */
905 } GPIO_TypeDef;

```

Redefinindo os intervalos de endereços de memória para os tipos das estruturas, podemos acessar diretamente os registradores mapeados na memória usando os endereços base dos blocos de

memória como ponteiros. Esses endereços são definidos por macros, como ((RCC_TypeDef *) RCC_BASE) e ((GPIO_TypeDef *) GPIOB_BASE), em “stm32h7a3xxq.h” e são referenciados pelas macros RCC e GPIOB, respectivamente. Isso evita o uso direto dos endereços-base no código.

```
2409 #define PSSI ((PSSI_TypeDef *) PSSI_BASE)
2410 #define RCC ((RCC_TypeDef *) RCC_BASE)
2411 #define FLASH ((FLASH_TypeDef *) FLASH_R_BASE)
2412 #define CRC ((CRC_TypeDef *) CRC_BASE)
2413
2414 #define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
2415 #define GPIOB ((GPIO_TypeDef *) GPIOB_BASE)
2416 #define GPIOC ((GPIO_TypeDef *) GPIOC_BASE)
2417 #define GPIOD ((GPIO_TypeDef *) GPIOD_BASE)
```

Os endereços-base, GPIOB_BASE e RCC_BASE, são macros definidas no mesmo arquivo.

```
2052 /*!< SRD_AHB4PERIPH peripherals */
2053 #define GPIOA_BASE (SRD_AHB4PERIPH_BASE + 0x0000UL)
2054 #define GPIOB_BASE (SRD_AHB4PERIPH_BASE + 0x0400UL)
2055 #define GPIOC_BASE (SRD_AHB4PERIPH_BASE + 0x0800UL)
2056 #define GPIOD_BASE (SRD_AHB4PERIPH_BASE + 0x0C00UL)
2057 #define GPIOE_BASE (SRD_AHB4PERIPH_BASE + 0x1000UL)
2058 #define GPIOF_BASE (SRD_AHB4PERIPH_BASE + 0x1400UL)
2059 #define GPIOG_BASE (SRD_AHB4PERIPH_BASE + 0x1800UL)
2060 #define GPIOH_BASE (SRD_AHB4PERIPH_BASE + 0x1C00UL)
2061 #define GPIOI_BASE (SRD_AHB4PERIPH_BASE + 0x2000UL)
2062 #define GPIOJ_BASE (SRD_AHB4PERIPH_BASE + 0x2400UL)
2063 #define GPIOK_BASE (SRD_AHB4PERIPH_BASE + 0x2800UL)
2064 #define RCC_BASE (SRD_AHB4PERIPH_BASE + 0x4400UL)
2065 #define PWR_BASE (SRD_AHB4PERIPH_BASE + 0x4800UL)
2066 #define BDMA2_BASE (SRD_AHB4PERIPH_BASE + 0x5400UL)
2067 #define DMAMUX2_BASE (SRD_AHB4PERIPH_BASE + 0x5800UL)
2068
```

A definição de GPIOB_BASE e RCC_BASE envolve a macro SRD_AHB4PERIPH_BASE, que é também definida no mesmo arquivo.

```
1993 #define SRD_APB4PERIPH_BASE (PERIPH_BASE + 0x18000000UL) /*!< D3_APB1PERIPH_BASE (PERIPH_BASE + 0x18000000UL)
1994 #define SRD_AHB4PERIPH_BASE (PERIPH_BASE + 0x18020000UL) /*!< D3_AHB1PERIPH_BASE (PERIPH_BASE + 0x18020000UL)
```

A definição da macro SRD_AHB4PERIPH_BASE depende, por sua vez da macro PERIPH_BASE.

```

1948 * @{
1949 */
1950 #define CD_ITCMRAM_BASE (0x00000000UL) /*!< Base address of : 64KB RAM reserved for CPU execution/instruction accessible over ITCM */
1951 #define CD_DTCMRAM_BASE (0x20000000UL) /*!< Base address of : 128KB (2x64KB) system data RAM accessible over DTCM */
1952 #define CD_AXIFLASH_BASE (0x08000000UL) /*!< Base address of : (up to 2 MB) embedded FLASH memory accessible over AXI */
1953
1954 #define CD_AXISRAM1_BASE (0x24000000UL) /*!< Base address of : (up to 256KB) system data RAM1 accessible over over AXI */
1955 #define CD_AXISRAM2_BASE (0x24040000UL) /*!< Base address of : (up to 384KB) system data RAM2 accessible over over AXI */
1956 #define CD_AXISRAM3_BASE (0x240A0000UL) /*!< Base address of : (up to 384KB) system data RAM3 accessible over over AXI */
1957 #define CD_AHBSRAM1_BASE (0x30000000UL) /*!< Base address of : (up to 64KB) system data RAM1 accessible over over AXI->AHB Bridge */
1958 #define CD_AHBSRAM2_BASE (0x30010000UL) /*!< Base address of : (up to 64KB) system data RAM2 accessible over over AXI->AHB Bridge */
1959
1960 #define SRD_BKPSRAM_BASE (0x38000000UL) /*!< Base address of : Backup SRAM(4 KB) over AXI->AHB Bridge */
1961 #define SRD_SRAM_BASE (0x38000000UL) /*!< Base address of : Backup SRAM(32 KB) over AXI->AHB Bridge */
1962
1963 #define OCTOSPI1_BASE (0x90000000UL) /*!< Base address of : OCTOSPI1 memories accessible over AXI */
1964 #define OCTOSPI2_BASE (0x70000000UL) /*!< Base address of : OCTOSPI2 memories accessible over AXI */
1965
1966 #define FLASH_BANK1_BASE (0x08000000UL) /*!< Base address of : (up to 1 MB) Flash Bank1 accessible over AXI */
1967 #define FLASH_BANK2_BASE (0x08100000UL) /*!< Base address of : (up to 1 MB) Flash Bank2 accessible over AXI */
1968 #define FLASH_END (0x081FFFFFFUL) /*!< FLASH end address */
1969
1970 /* Legacy define */
1971 #define FLASH_BASE FLASH_BANK1_BASE
1972 #define D1_AXISRAM_BASE CD_AXISRAM1_BASE
1973
1974 #define FLASH_OTP_BASE (0x08FFF000UL) /*!< Base address of : (up to 1KB) embedded FLASH Bank1 OTP Area */
1975 #define FLASH_OTP_END (0x08FFF3FFUL) /*!< End address of : (up to 1KB) embedded FLASH Bank1 OTP Area */
1976
1977 /*!< Device electronic signature memory map */
1978 #define UID_BASE (0x08FFF800UL) /*!< Unique device ID register base address */
1979 #define FLASHSIZE_BASE (0x08FFF80CUL) /*!< FLASH Size register base address */
1980 #define PACKAGE_BASE (0x08FFF80EUL) /*!< Package Data register base address */
1981
1982 #define PERIPH_BASE (0x40000000UL) /*!< Base address of : AHB/ABP Peripherals */
1983
1984 /*!< Peripheral memory map */

```

A partir do endereço 0x40000000 da macro PERIPH_BASE, é possível retroceder pelas macros que passamos para encontrar os endereços-base das macros RCC_BASE e GPIOB_BASE.

5. Com essa abordagem, podemos escrever o nosso código sem a necessidade de criar variáveis adicionais para acessar os registradores; podemos utilizar diretamente os mnemônicos definidos no arquivo de cabeçalho “stm32h7a3xxq.h”.

```

27 #define ITERACOES 5000
28
29 void multiplo_iteracoes (uint32_t valor, uint32_t *j)
30 {
31     *j = valor * ITERACOES;
32     return;
33 }
34
35 void espera (uint32_t valor)
36 {
37     uint32_t i;
38
39     multiplo_iteracoes (valor, &i);
40     while (i) i--;
41 }
42
43 int main(void)
44 {
45     // Inicializa GPIOB, pino 0 (PB0)
46     RCC->AHB4ENR |= 0x00000002; // GPIOB clock enable
47
48     // PB0 como saída digital
49     GPIOB->MODER |= 0x00000001;
50     GPIOB->MODER &= 0xFFFFFFF0;
51
52     GPIOB->OTYPER &= 0xFFFFFFF0; // PB0 como push-pull
53
54     /* Loop forever */
55     for(;;) {
56         GPIOB->ODR &= 0xFFFFFFF0; // PB0 = 0
57         espera(1000); // Gera delay
58         GPIOB->ODR |= 0x00000001; // PB0 = 1
59         espera(1000); // Gera delay
60     }
61 }

```

6. Além dos endereços-base de cada módulo, o arquivo “stm32h7a3xxq.h” inclui diversas macros para a configuração individual dos *bits*, o que torna o código mais legível. A figura a seguir ilustra as macros associadas à diferentes máscaras de *bits* do registrador MODER de um módulo GPIOx. Por exemplo, a macro GPIO_MODER_MODE0_Msk define a máscara 0x00000003, enquanto a macro GPIO_MODER_MODE4_Msk define uma máscara 0x00000300.

```

9627 /***** Bits definition for GPIO_MODER register *****/
9628 #define GPIO_MODER_MODE0_Pos      (0U)
9629 #define GPIO_MODER_MODE0_Msk      (0x3UL << GPIO_MODER_MODE0_Pos) /*!< 0x00000003 */
9630 #define GPIO_MODER_MODE0_Pos      (0U)
9631 #define GPIO_MODER_MODE0_Msk      (0x3UL << GPIO_MODER_MODE0_Pos) /*!< 0x00000003 */
9632 #define GPIO_MODER_MODE0_Pos      (0U)
9633 #define GPIO_MODER_MODE0_Msk      (0x3UL << GPIO_MODER_MODE0_Pos) /*!< 0x00000003 */
9634 #define GPIO_MODER_MODE1_Pos      (2U)
9635 #define GPIO_MODER_MODE1_Msk      (0x3UL << GPIO_MODER_MODE1_Pos) /*!< 0x0000000C */
9636 #define GPIO_MODER_MODE1_Pos      (2U)
9637 #define GPIO_MODER_MODE1_Msk      (0x3UL << GPIO_MODER_MODE1_Pos) /*!< 0x0000000C */
9638 #define GPIO_MODER_MODE1_Pos      (2U)
9639 #define GPIO_MODER_MODE1_Msk      (0x3UL << GPIO_MODER_MODE1_Pos) /*!< 0x0000000C */
9640 #define GPIO_MODER_MODE2_Pos      (4U)
9641 #define GPIO_MODER_MODE2_Msk      (0x3UL << GPIO_MODER_MODE2_Pos) /*!< 0x00000030 */
9642 #define GPIO_MODER_MODE2_Pos      (4U)
9643 #define GPIO_MODER_MODE2_Msk      (0x3UL << GPIO_MODER_MODE2_Pos) /*!< 0x00000030 */
9644 #define GPIO_MODER_MODE2_Pos      (4U)
9645 #define GPIO_MODER_MODE2_Msk      (0x3UL << GPIO_MODER_MODE2_Pos) /*!< 0x00000030 */
9646 #define GPIO_MODER_MODE3_Pos      (6U)
9647 #define GPIO_MODER_MODE3_Msk      (0x3UL << GPIO_MODER_MODE3_Pos) /*!< 0x000000C0 */
9648 #define GPIO_MODER_MODE3_Pos      (6U)
9649 #define GPIO_MODER_MODE3_Msk      (0x3UL << GPIO_MODER_MODE3_Pos) /*!< 0x000000C0 */
9650 #define GPIO_MODER_MODE3_Pos      (6U)
9651 #define GPIO_MODER_MODE3_Msk      (0x3UL << GPIO_MODER_MODE3_Pos) /*!< 0x000000C0 */
9652 #define GPIO_MODER_MODE4_Pos      (8U)
9653 #define GPIO_MODER_MODE4_Msk      (0x3UL << GPIO_MODER_MODE4_Pos) /*!< 0x00000300 */
9654 #define GPIO_MODER_MODE4_Pos      (8U)
9655 #define GPIO_MODER_MODE4_Msk      (0x3UL << GPIO_MODER_MODE4_Pos) /*!< 0x00000300 */
9656 #define GPIO_MODER_MODE4_Pos      (8U)
9657 #define GPIO_MODER_MODE4_Msk      (0x3UL << GPIO_MODER_MODE4_Pos) /*!< 0x00000300 */

```

Vamos substituir as máscaras usadas nos comandos de configuração dos registradores pelas máscaras definidas na CMSIS.

```

43 int main(void)
44 {
45     // Inicializa GPIOB, pino 0 (PB0)
46     //RCC->AHB4ENR |= 0x00000002; // GPIOB clock enable
47     RCC->AHB4ENR |= RCC_AHB4ENR_GPIOBEN_Msk;
48
49     // PB0 como saída digital
50     //GPIOB->MODER |= 0x00000001;
51     //GPIOB->MODER &= 0xFFFFFFF0;
52     GPIOB->MODER &= ~(GPIO_MODER_MODE0_Msk);
53     GPIOB->MODER |= GPIO_MODER_MODE0_Pos;
54
55     //GPIOB->OTYPER &= 0xFFFFFFF0; // PB0 como push-pull
56     GPIOB->OTYPER &= ~GPIO_OTYPER_OT0_Msk;
57
58     /* Loop forever */
59     for(;;) {
60         // GPIOB->ODR &= 0xFFFFFFF0; // PB0 = 0
61         GPIOB->ODR &= ~GPIO_ODR_OD0_Msk;
62         espera(1000); // Gera delay
63         //GPIOB->ODR |= 0x00000001; // PB0 = 1
64         GPIOB->ODR |= GPIO_ODR_OD0_Msk; // PB0 = 1
65         espera(1000); // Gera delay
66     }
67 }

```

7. Faça “Build” e execute o projeto para certificar se o LED verde pisca.

8. Agora altere o programa que pisque o LED amarelo para uma versão que usa a interface CMSIS. Depois exporte o projeto em um arquivo ZIP para ser incluído no relatório.

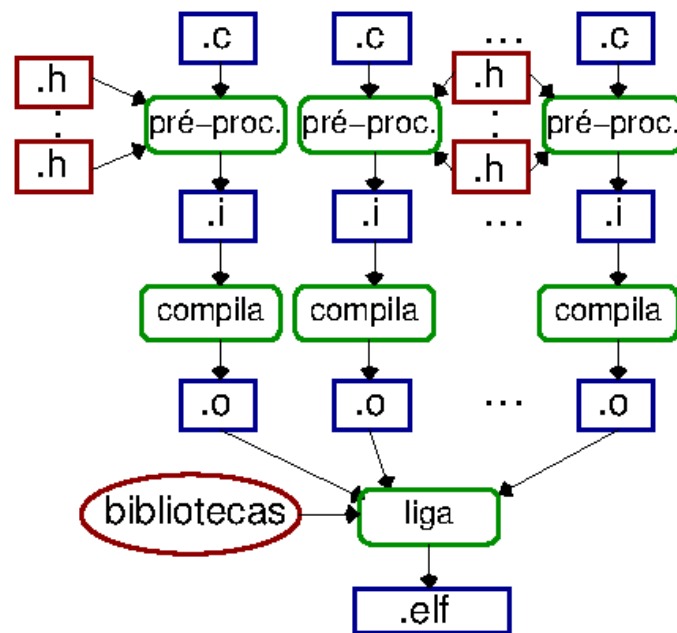
DICAS: Além de mudar os nomes dos registradores para RCC->##### ou GPIOE->#####, as máscaras numéricas devem ser substituídas pelas máscaras da CMSIS.

PYTHON E C

Python é conhecido por sua sintaxe clara e legível, o que facilita a escrita e manutenção do código. Em contraste, a sintaxe de C é mais complexa e detalhada, exigindo mais atenção aos detalhes. **Python é uma linguagem interpretada** de alto nível para propósito geral. Ele foi desenvolvido pelo Guido van Rossum em 1989 com o objetivo de ter uma linguagem que apresenta uma sintaxe intuitiva, similar à linguagem natural inglês, sem precisar se preocupar com a tipagem e o armazenamento de dados na memória como a linguagem C. A linguagem **C é uma linguagem compilada de médio nível**, também para propósito geral. Foi inventada pelo Dennis Ritchie em *Bell Laboratories* entre 1972–73 para programar sistemas operacionais que antes eram implementados com uma linguagem de baixo nível, o assembly. O sistema operacional Unix dos minicomputadores como DEC DPD7 foi integralmente implementado com linguagem C.

Uma linguagem compilada é aquela que requer a conversão dos códigos de um programa em código binário específico da máquina antes de sua execução. Por outro lado, uma linguagem interpretada é aquela cujas instruções são traduzidas em tempo de execução, referenciando funções pré-implementadas (*built-in functions*) e valores de seus argumentos. Devido à ausência dessa etapa de interpretação durante a execução, o tempo de execução de um programa compilado é menor do que o de um programa interpretado, resultando em um melhor desempenho temporal.

Tipicamente, uma cadeia de ferramentas, conhecida como *toolchain*, é utilizada para converter códigos em linguagem C, armazenados em arquivos com extensão “.c”, em um arquivo executável com extensão “.elf” (*executable and linkable format*) numa arquitetura ARM. Esse processo envolve várias etapas: um pré-processador traduz as diretivas de C em arquivos com extensão “.i”, que contêm apenas instruções puras de C; um compilador converte essas instruções puras de C em arquivos-objeto com extensão “.o”, contendo instruções de máquina do processador-alvo; e um ligador (*linker*) junta as instruções de diferentes arquivos para construir um arquivo executável com extensão “.elf”. Em cada estágio, diferentes tipos de erros podem ser gerados, e é necessário corrigi-los antes de avançar para a próxima etapa. Em contraste, os erros em Python são detectados durante a interpretação das instruções no momento da execução, já que todas as funções pré-implementadas foram previamente testadas.



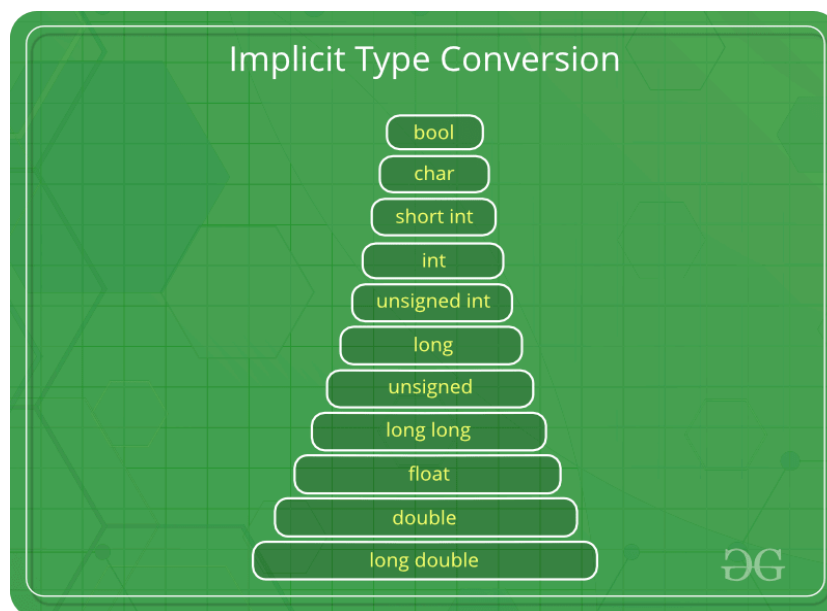
Ambas as linguagens compartilham estruturas de controle semelhantes, como comandos de controle de fluxo de iterações (“for”, “while”), comandos de quebra de fluxo (“break”, “continue”), e comandos condicionais (“if”, e “if-elif-else” em Python ou “if-else if-else” em C). Somente em Python 3.10 foi introduzido o comando de alternativas “switch-case” de C. Tanto Python quanto C permitem a definição e chamada de funções, promovendo a modularidade do código. Ambas as linguagens incluem operadores aritméticos (+, -, *, /), operadores lógicos (&&, ||, ! em C; “and”, “or”, “not” em Python), operadores relacionais (<, <=, >, >=, ==, !=), operadores lógicos *bit-a-bit* (&, |, ~), deslocamentos de *bits* (<<, >>) e atribuição (= em C; := a partir de Python 3.8).

Python usa tipagem dinâmica, o que significa que os tipos de dados são verificados em tempo de execução. Já em C, a tipagem é estática, e os tipos de dados são verificados em tempo de compilação, exigindo declaração explícita de variáveis. Uma atenção especial deve ser dada aos tipos de dados dos resultados de uma operação em C. Os tipos de dados dos resultados, sejam inteiros ou em ponto flutuante, dependem dos tipos dos operandos. Por exemplo, uma operação entre dois operandos inteiros sempre resulta em um valor inteiro, mesmo que o resultado matemático seja uma fração. Assim, a operação 1/2 resulta em 0, em vez de 0.5. Isso ocorre porque a divisão inteira trunca a parte fracionária do resultado para gerar um tipo de dado consistente com os operandos. Em contraste, Python usa tipagem dinâmica, o que significa que os tipos de dados dos resultados se ajustam dinamicamente conforme necessário. Em Python, a operação 1/2 resulta em 0.5, pois a divisão entre inteiros retorna um número de ponto flutuante. Python ajusta automaticamente os tipos de dados dos operandos e resultados durante a execução, permitindo maior flexibilidade e evitando a necessidade de conversões explícitas.

Tanto C quanto Python têm sistemas de escopo que determinam a visibilidade de variáveis em diferentes partes do código. Em C, as variáveis possuem escopo estático, o que significa que podem

ser globais ou locais a uma função específica. Variáveis globais, declaradas fora de todas as funções dentro de um arquivo, são acessíveis em todo o programa, enquanto variáveis locais existem apenas dentro da função onde são definidas. Em contraste, Python utiliza um sistema de escopo baseado em indentação e na estruturação de blocos por meio de espaços em branco, como tabulações ou espaços. As variáveis em Python são dinamicamente tipadas e têm escopo local quando definidas dentro de uma função. Variáveis globais em Python são declaradas fora de todas as funções do programa ou explicitamente marcadas como globais dentro de uma função para serem acessadas globalmente.

C possui o conceito de conversão implícita de tipos, que é aplicado em operações envolvendo operandos de tipos de dados distintos. Neste caso, os operandos são promovidos a um tipo de dados comum, mais abrangente, antes da execução da operação, conforme [a hierarquia de “abrangência”](#) mostrada na figura. Por exemplo, se um inteiro e um ponto flutuante são usados em uma operação, o inteiro é automaticamente promovido a ponto flutuante para garantir que a operação seja realizada corretamente. No entanto, C não consegue expandir o espaço de memória previamente alocado pelo compilador quando uma operação requer mais *bits*. Por exemplo, ao fazer um deslocamento à esquerda de um *bit* no valor 0b11000000 de uma variável do tipo de dados de 8 *bits* (“char” ou “bool”), o resultado é 0b10000000, pois o valor excedente é descartado. Em contraste, Python, sem uma tipagem previamente fixa, consegue realocar o espaço de memória dinamicamente, permitindo representar 0b11000000 sem perda de dados.



Em termos de gerenciamento de memória, Python cuida disso automaticamente através de um coletor de lixo, enquanto C requer que os programadores gerenciem a alocação e liberação de memória manualmente, utilizando funções como “malloc” e “free”. Python possui uma vasta biblioteca padrão e suporte a muitos módulos externos, facilitando o desenvolvimento rápido de

aplicações. C também tem bibliotecas, mas o desenvolvimento pode exigir mais código manual e integração.

Apesar dessas semelhanças, as diferenças em gestão de memória, tipagem, tipos de erros e execução influenciam significativamente a escolha entre Python e C, dependendo das necessidades do projeto e do nível de controle requerido sobre o *hardware*. Devido à facilidade no gerenciamento de memória, no tratamento de erros, na geração de código executável e na tipagem dinâmica, Python tem se tornado popular para prototipagem rápida e provas de conceito no desenvolvimento de projetos em geral. No entanto, essas facilidades comprometem o desempenho e o determinismo do sistema. Em aplicações onde esses fatores são essenciais, como em sistemas embarcados, a linguagem C continua sendo a preferida.

DETALHES DA LINGUAGEM C

A linguagem C oferece uma proximidade significativa com a linguagem *assembly* através de operadores entre operandos, manipulação direta de memória e declaração explícita de variáveis. Sua capacidade de organizar registradores dos módulos usando estruturas (“struct”), como a disponibilidade de ponteiros, a tornam especialmente adequada para programação de sistemas embarcados, onde o controle detalhado sobre o *hardware* é essencial. Ao mesmo tempo, C fornece um nível de abstração suficiente para facilitar a criação de código modular e reutilizável, equilibrando eficiência e legibilidade.

Em C, os **operadores** são muito semelhantes aos usados em *assembly*, permitindo manipulações diretas de *bits* e *bytes*. A seguinte tabela mostra a correspondência direta entre os operadores lógico-aritméticos e relacionais da linguagem C e algumas instruções do repertório ARM Thumb-2 do Cortex-M7.

C	Descrição	ARM Thumb-2 (Cortex-M7)
+	Adição	ADD, VADD
-	Subtração	SUB, VSUB
*	Multiplicação	MUL, VMUL
/	Divisão	SDIV, UDIV, VDIV
%	Resto da divisão	-
++	Incremento	ADD (imediato)
--	Decremento	SUB (imediato)
&	E lógico <i>bit a bit</i>	AND

	Ou lógico <i>bit a bit</i>	ORR
^	Ou exclusivo bit a bit	EOR
~	Complemento de <i>bits</i>	MVN
<<	Deslocamento para esquerda	LSL
>>	Deslocamento para direita	LSR
&&	E lógico	CMP e <i>bits</i> de condição
	Ou lógico	CMP e <i>bits</i> de condição
!	Negação lógica	VNEG
==	Igual a	CMP e <i>bits</i> de condição
!=	Não é igual a	CMP e <i>bits</i> de condição
>	Maior que	CMP e <i>bits</i> de condição
<	Menor que	CMP e <i>bits</i> de condição
>=	Maior ou igual a	CMP e <i>bits</i> de condição
<=	Menor ou igual a	CMP e <i>bits</i> de condição

Vale destacar que a linguagem C suporta versões contraídas de operações lógico-aritméticas, chamadas de operadores compostos. Estes operadores permitem realizar operações aritméticas e lógicas de forma mais compacta e eficiente, combinando a operação e a atribuição em uma única expressão. Por exemplo, ao invés de escrever “`x = x + 5;`”, vale também “`x += 5;`”.

A declaração de variáveis em C é explícita e requer que o programador especifique o tipo de dados e o nome da variável. Isso é semelhante à reserva de espaço de memória em *assembly*, onde cada espaço é definido por diretivas como “.byte” (1 *byte*) e “.word” (4 *bytes* em ARM), que determinam o tamanho do espaço a ser alocado e, muitas vezes, inicializam o espaço com um valor específico.

Em C, os **tipos de dados** não apenas definem como os dados são armazenados na memória, mas também como são interpretados pelo compilador. Eles determinam quais operações matemáticas e lógicas podem ser aplicadas aos dados. Todas as variáveis utilizadas devem ser previamente declaradas com um tipo de dados para que o compilador possa reservar o espaço necessário na memória e garantir que as operações apropriadas possam ser realizadas sobre elas.

Os tipos de dados podem ser divididos em tipos básicos e tipos derivados. Os tipos de dados básicos incluem:

int: Representa números inteiros.

char: Armazena caracteres individuais, como letras e símbolos. Internamente, é tratado como um número inteiro de 8 *bits*..

float: Representa números de ponto flutuante de precisão simples. É usado para valores com casas decimais com até 8 *bits* de expoente.

double: Representa números de ponto flutuante de dupla precisão com até 11 *bits* de expoente.

void: Representa um tipo de dado que não possui valor ou tipo específico. É utilizado para criar funções que não precisam retornar um valor ou para trabalhar com ponteiros de tipos indefinidos.

Para todas as variáveis de tipos de dados que possuem valor, é possível acessar seu endereço de memória usando o operador '&'. Por exemplo, para uma variável 'a' declarada como do tipo "char" com o comando "char a;", podemos obter o endereço de 'a' utilizando a sintaxe "&a".

Os tipos derivados permitem estruturar dados de maneira mais complexa e são construídos a partir dos tipos básicos. Entre os tipos de dados derivados estão:

Arranjos: Coleções de elementos do mesmo tipo, acessados por um índice. São tipos derivados porque são construídos a partir de tipos básicos.

Structs: Estruturas que permitem combinar diferentes tipos de dados sob um único nome. Cada elemento dentro de uma estrutura pode ter um tipo diferente. Para acessar esses elementos, utiliza-se o operador '->' quando a variável é um ponteiro do tipo struct, e o operador '.' quando é um valor direto do tipo struct.

Union: Similar a uma struct, mas todos os elementos compartilham o mesmo espaço de memória. A união é útil quando se deseja economizar espaço, mas só se precisa armazenar um tipo de dado por vez.

Enumerações (enum): Define um novo tipo de dados que pode assumir um número limitado de valores, que são enumerados explicitamente.

Ponteiros: Variáveis que armazenam endereços de memória alocados para um tipo de dado específico. Eles permitem a manipulação direta da memória e são essenciais para a alocação dinâmica de memória, bem como para a passagem de parâmetros por endereço. Para declarar um ponteiro para um tipo de dado X, usamos a sintaxe "X *". Por exemplo, para declarar ponteiros para os tipos de dados "int" e "float", utilizamos "int *" e "float *", respectivamente. Pode-se também acessar o conteúdo de um ponteiro, usando o operador '*'. Por exemplo, para um ponteiro 'b' declarado como do tipo "char *" com o comando "char *b;", podemos obter o conteúdo de 'b' utilizando a sintaxe "*b".

C suporta a palavra-chave **typedef**, interpretada diretamente pelo compilador como parte da sintaxe da linguagem. Esta palavra é usada para criar novos nomes para tipos de dados existentes. Isso pode ajudar a tornar o código mais legível e portátil.

Em C, os **qualificadores de tipos de dados** são palavras-chave que modificam os comportamentos padrão dos tipos de dados básicos. Eles são usados para especificar certas propriedades ou restrições adicionais sobre como os dados devem ser tratados ou armazenados pelo compilador. Seguem-se os principais qualificadores:

const: é um modificador de tipo que indica que o valor de uma variável não pode ser alterado após sua inicialização. Isso significa que uma vez atribuído um valor, esse valor não pode ser modificado através dessa variável.

volatile: é um modificador de acesso, indicando que o valor de uma variável pode ser alterado inesperadamente por processos externos ao programa. Isso previne otimizações de compilador que poderiam, de outra forma, assumir que o valor não muda. É frequentemente usado para variáveis acessadas por múltiplas *threads* ou por dispositivos de *hardware*.

signed e unsigned: são modificadores que determinam como o *bit* mais significativo dos tipos de dados numéricos inteiros (`int`, `char`, `short`, `long`) deve ser interpretado: como um *bit* de sinal (“signed”) ou um *bit* numérico (“unsigned”). Por padrão, os tipos inteiros em C são “signed” (com sinal). Portanto, “unsigned” (sem sinal) deve ser utilizado explicitamente quando se deseja trabalhar apenas com números não negativos.

static: é um modificador do escopo e a duração de vida de uma variável. Em vez de ser criada e destruída a cada chamada de função, uma variável `static` persiste durante a execução do programa, mas apenas acessível na função em que ela é declarada.

Em C, valores inteiros podem ser representados em bases binária, decimal, octal e hexadecimal. A decimal é a base padrão e usa os dígitos de 0 a 9. A binária, que só usa os dígitos 0 e 1, é prefixada por “0b”. A octal utiliza os dígitos de 0 a 7 e é prefixada com “0”. A hexadecimal usa os dígitos de 0 a 9 e as letras de ‘A’ a ‘F’ (ou ‘a’ a ‘f’), sendo prefixada com “0x” ou “0X”. Por exemplo, 10, 0b1010, 012 e 0xA são representações do mesmo valor numérico em diferentes bases em C. Um número em ponto flutuante pode ser representado em notação decimal (por exemplo, 4.15) ou em notação científica (por exemplo, 0.415e1, que equivale a 4.15). Um caractere, tipicamente um valor em ASCII, é escrito entre aspas simples, como ‘A’ e ‘a’. Uma *string* em C é um arranjo de caracteres terminado com um caractere nulo (‘\0’). Pode ser escrita entre aspas duplas. Por exemplo, a *string* “Bom dia!” é representada como um arranjo de caracteres através da declaração: “char str[9]= “Hello”,”.

O uso de funções em linguagens de programação não só melhora a estrutura e organização do código, mas também promove a modularidade, a reutilização de código, a abstração de complexidade e facilita a manutenção do software. A estrutura básica de uma função em C compreende a sua declaração e definição. Na declaração, especificamos o tipo de dado que a função retorna, o nome da função (único dentro do escopo onde é definida), e a lista de parâmetros que são variáveis que a função recebe como entrada e podem ser modificadas. Os parâmetros são opcionais e podem ser de qualquer tipo válido na linguagem. Mesmo sem parâmetros, os parênteses vazios ainda são necessários. A definição da função, por sua vez, consiste em três partes: declaração de variáveis locais, execução de comandos que operam sobre essas variáveis, e retorno de um valor

```
<tipo_de_dado_retorno> <nome_da_funcao> (<lista_de_parâmetros>) {  
// Declaração de variáveis locais  
// Execução de comandos  
// Retorno de valor (se houver)  
}
```

Em C, o mecanismo de chamada de funções definidas segue os seguintes passos básicos:

chamada da função: para chamar uma função, utiliza-se seu nome seguido por parênteses contendo os argumentos necessários.

passagem de parâmetros: os parâmetros podem ser passados para a função de duas maneiras: por valor do conteúdo da memória (dado) e por valor de endereço da memória (endereço). Quando se passa por valor, o valor real do argumento é copiado para o parâmetro da função. Modificações no parâmetro dentro da função não afetam o argumento original. E quando se passa por endereço, ou ponteiro, o endereço do argumento é passado como parâmetro para a função. Isso permite à função acessar e modificar diretamente o valor do argumento original na memória.

execução da função: dentro da função, os parâmetros podem ser manipulados usando diretamente a variável se a passagem é por valor, ou a variável precedido de ‘*’ se é por valor de endereço.

C é equipada com diretivas de pré-processador precedidas por ‘#’, ou mecanismos que ajudam a definir constantes, incluir arquivos de cabeçalho, e criar novos tipos de dados:

#define: é usada para definir macros, que são substituições de texto realizadas pelo pré-processador antes da compilação. Pode ser usada para criar constantes ou macros mais complexas com argumentos.

#include: é usada para incluir o conteúdo de um arquivo de cabeçalho no código fonte. Isso permite a reutilização de código e a separação de declarações e definições em diferentes arquivos.

#ifdef, #ifndef, #if, #else, #elif, #endif: essas diretivas são usadas para incluir ou excluir partes do código com base em condições, permitindo a compilação condicional. Isso é particularmente útil para configurar compilação específica para diferentes ambientes ou configurações de *hardware*.

Em C, o código é tipicamente dividido em arquivos de código-fonte (de extensão .c) e arquivos de cabeçalho (de extensão .h). Arquivos de código-fonte em C são arquivos que contêm a implementação de um procedimento, incluindo definições de funções, variáveis, e a lógica do programa. Arquivos de cabeçalho geralmente contêm declarações de funções, variáveis globais, macros, e tipos de dados (structs, enums, typedefs) que podem ser usados em múltiplos arquivos de código-fonte. Isso permite que diferentes arquivos de código-fonte compartilhem essas declarações sem duplicação. Essa divisão serve para organizar e modularizar o código, facilitando a reutilização e a manutenção.

À medida que avançamos ao longo deste curso, exploraremos o uso dessas funcionalidades de C para otimizar e configurar nossos projetos de acordo com os requisitos específicos de *hardware* e *software*.

PROGRAMAÇÃO C EM MICROCONTROLADORES

Em programação de microcontroladores, é comum usar os tipos de dados `uint*_t` e `int*_t` (onde `*` representa o tamanho em *bits*, como `uint8_t`, `int16_t`, etc.) em vez do tipo de dados nativo `int`. Esses tipos são definidos no cabeçalho `<stdint.h>` da biblioteca padrão do C. A portabilidade é um dos principais motivos. Usar esses tipos garante que o tamanho do tipo de dados será o mesmo em qualquer plataforma. Isso é crucial na programação de microcontroladores, onde o tamanho dos tipos de dados pode variar entre diferentes arquiteturas. O programador pode prever exatamente quantos *bits* serão usados para armazenar uma variável, o que é essencial para manipulação dos registradores do microcontrolador e protocolos de comunicação. O controle e a precisão são também importantes. Em sistemas embarcados, a memória é um recurso escasso. Usar tipos de dados com tamanho específico ajuda a gerenciar a memória de maneira eficiente, garantindo que não se use mais memória do que o necessário e evitando problemas de *overflow*. Operações *bit a bit* requerem precisão no tamanho dos dados para configurações a nível de *bits*.

A conversão explícita em C, também conhecida como *casting* ou *typecast*, permite ao programador informar ao compilador como interpretar um determinado valor. Isso é particularmente útil quando é necessário diferenciar entre valores inteiros e endereços de memória, duas entidades que, em sistemas embarcados, frequentemente compartilham o mesmo espaço de *bits*, mas têm significados distintos. Em um sistema de microcontrolador, essa distinção é crucial para a manipulação direta dos registradores de microcontrolador, cujos endereços são representados por valores inteiros. Por

exemplo, para explicitar que `0x58024540` é um endereço de memória, podemos usar o comando `“(uint32_t *)0x58024540”` para converter o valor `0x58024540`, que por padrão é do tipo `“uint32_t”`, para um tipo `“uint32_t *”`, indicando que se trata de um endereço de memória cujo conteúdo ocupa 4 *bytes*.

Para minimizar a quantidade de registradores nos periféricos, vários *bits* funcionalmente independentes são compactados em um único registrador, de modo que as operações devam ser realizadas *bit a bit*. Podemos construir diferentes máscaras de *bits* que nos permitem manipular alguns *bits* específicos de um registrador sem afetar os outros. Denominamos essa operação de **mascaramento**. Três das máscaras de *bits* mais utilizadas na programação de microcontroladores são:

máscara de OU (OR lógico, *bit a bit*) ou máscara de 1: seta um ou mais *bits* em 1 sem afetar os demais. A máscara deve ter valor ‘1’ nos *bits* correspondentes aos *bits* que se deseja setar.

máscara de AND (AND lógico, *bit a bit*) ou máscara de 0: reseta um ou mais *bits* em 0 sem afetar os demais. A máscara deve ter valor ‘1’ nos *bits* correspondentes aos *bits* que não se deseja alterar. A máscara deve ter valor ‘0’ nos *bits* correspondentes aos *bits* que se deseja resetar.

máscara de OU exclusivo (XOR lógico, *bit a bit*): inverte um ou mais *bits* sem afetar os demais. A máscara deve ter valor ‘1’ nos *bits* correspondentes aos *bits* que se deseja inverter.

Por exemplo, para configurar o pino PB0 como um pino de saída, é necessário definir os *bits* [1:0] do registrador GPIOB_MODER como “01”. Isso é feito através de duas operações lógicas *bit a bit*: primeiro, aplicando uma máscara AND com 0xFFFFF0FC para limpar os dois *bits* alvo; em seguida, utilizando uma máscara OR com 0x00000001 para definir o *bit* “0” como “1”:

```
GPIOB_MODER &= 0xFFFFF0FC;
```

```
GPIOB_MODER |= 0x00000001;
```

Em vez de realizar operações *bit a bit* separadas, podemos também criar uma palavra com todos os *bits* de controle configurados e fazer uma única atribuição dos *bits* quando o acesso de escrita simultâneo desses *bits* é imprescindível:

```
uint32_t tmp;;
```

```
tmp = GPIOB_MODER;
```

```
tmp &= 0xFFFFF0FC;
```

```
tmp |= 0x00000001;
```

```
GPIOB_MODER = tmp;
```

Em programação de microcontroladores, uma prática comum é a abstração dos registradores de um módulo, que são mapeados em um espaço contíguo de endereços de memória, utilizando a estrutura de dados “struct” em C. Essa técnica organiza de forma clara e acessível os registradores, facilitando o desenvolvimento e a manutenção do código. Vamos considerar a definição de uma “struct” no arquivo de cabeçalho <stm32ha3xxq.h> do padrão de interface CMSIS (do inglês *Cortex Microcontroller Software Interface Standard*). Aqui está um exemplo de definição de uma “struct” para o módulo GPIO:

```
typedef struct {  
    __IO uint32_t MODER; /*!< GPIO port mode register, Address offset: 0x00 */
```

```

__IO uint32_t OTYPER; /*!< GPIO port output type register, Address offset: 0x04 */
__IO uint32_t OSPEEDR; /*!< GPIO port output speed register, Address offset: 0x08 */
__IO uint32_t PUPDR; /*!< GPIO port pull-up/pull-down register, Address offset: 0x0C */
__IO uint32_t IDR; /*!< GPIO port input data register, Address offset: 0x10 */
__IO uint32_t ODR; /*!< GPIO port output data register, Address offset: 0x14 */
__IO uint32_t BSRR; /*!< GPIO port bit set/reset, Address offset: 0x18 */
__IO uint32_t LCKR; /*!< GPIO port configuration lock register, Address offset: 0x1C */
__IO uint32_t AFR[2]; /*!< GPIO alternate function registers, Address offset: 0x20-0x24 */
} GPIO_TypeDef;

```

Esta “struct”, nomeada como `GPIO_TypeDef`, agrupa todos os registradores do módulo GPIO, que são mapeados em um espaço contíguo de memória. Cada elemento da “struct” representa um registrador específico do módulo GPIO. Alguns deles já vimos no Roteiro 1:

__IO uint32_t MODER: Registrador que controla se cada pino do módulo é configurado como entrada, saída, função alternativa ou analógica. O *offset* de endereço é 0x00.

__IO uint32_t OTYPER: Registrador que define se as saídas são *push-pull* ou *open-drain*. O *offset* de endereço é 0x04.

__IO uint32_t OSPEEDR: Registrador que configura as velocidades de comutação dos pinos de saída. O *offset* de endereço é 0x08.

__IO uint32_t PUPDR: Registrador que configura os resistores de *pull-up* e *pull-down* para os pinos de entrada. O *offset* de endereço é 0x0C.

__IO uint32_t IDR: Registrador de dados de entrada do módulo GPIO, utilizado para ler o estado dos pinos configurados como entradas. O *offset* de endereço é 0x10.

__IO uint32_t ODR: Registrador de dados de saída do módulo GPIO, utilizado para escrever os valores nos pinos configurados como saídas. O *offset* de endereço é 0x14.

__IO uint32_t BSRR: Registrador que permite a manipulação atômica dos pinos, configurando ou resetando individualmente os pinos do módulo. O *offset* de endereço é 0x18.

__IO uint32_t LCKR: Registrador de bloqueio de configuração do módulo GPIO, utilizado para bloquear a configuração dos pinos, prevenindo modificações acidentais. O *offset* de endereço é 0x1C.

__IO uint32_t AFR[2] (0x58020420): Registradores de função alternativa do módulo GPIO, que permitem configurar funções alternativas para os pinos, como UART, SPI, etc. Existem dois registradores para acomodar até 16 pinos. O *offset* de endereço é 0x20-0x24.

Para usar a struct “`GPIO_TypeDef`” e acessar os registradores do módulo GPIO, geralmente um ponteiro para a struct é definido, apontando para o endereço base do módulo GPIO. Por exemplo, para o GPIOB, [cujo intervalo de endereço é 0x58020400 - 0x580207FF](#),

(x)= Variables Breakpoi... Expressio... Registers Live Expr... SFRs x			
RD x16 x			
All registers			
type filter text			
Register	Address	Value	
GPIOB			
> MODER	0x58020400	0xfffffebd	
> OTYPER	0x58020404	0x0	
> OSPEEDR	0x58020408	0xc0	
> PUPDR	0x5802040c	0x100	
> IDR	0x58020410	0x10	
> ODR	0x58020414	0x0	
> BSRR	0x58020418		
> LCKR	0x5802041c	0x0	
> AFRL	0x58020420	0x0	
> AFRH	0x58020424	0x0	

podemos converter explicitamente o valor 0x58020400 para um ponteiro de struct “GPIO_TypeDef” da seguinte maneira

```
#define GPIOB ((volatile GPIO_TypeDef *) 0x58020400)
```

Com essa definição, podemos acessar os seus elementos, que são registradores do GPIOB, usando mnemônicos como GPIOB->MODER, GPIOB->OTYPER ou GPIOB->ODR. Este é o padrão adotado pelo CMSIS para criar os mnemônicos dos registradores de um microcontrolador: <MÓDULO>-><Registrador>. Os arquivos de cabeçalho são amplamente utilizados em programas de microcontroladores. Um exemplo notável são os arquivos do padrão de interface CMSIS, que padroniza uma série de mnemônicos e facilita a programação dos microcontroladores.

Embora a programação em C seja muitas vezes suficiente e mais legível, é comum precisar de um controle mais granular sobre o *hardware* e a execução do código em programação de microcontroladores. Esse nível de controle pode ser obtido através da inserção de código *assembly* diretamente em um programa C, utilizando o recurso de *inline assembly*, que é uma extensão oferecida pelos compiladores. A sintaxe básica para o uso do [inline assembly no GCC](#) é

```
asm [qualificadores] (instruções_asm
    : Operandos_de_saída
    [: Operandos_de_entrada
    [: Clobbers]])
```

O qualificador mais utilizado é o “volatile”, que indica ao compilador que o código *assembly* deve ser tratado de forma especial, sem otimizações que possam alterar o comportamento pretendido. As “instruções_asm” são fornecidas como *strings* literais e são passadas diretamente para o montador, sem modificações pelo compilador. Para formatar o código *assembly* de maneira legível, frequentemente utilizam-se caracteres de controle como ‘\n’ (nova linha) e ‘\t’ (tabulação), que

ajudam a separar e organizar as instruções no formato tradicional de *assembly*, garantindo que o montador receba e interprete as instruções corretamente. O termo “Clobbers” se refere a uma lista de registradores ou variáveis que o código *assembly* pode modificar durante sua execução. Em *inline assembly*, isso é importante porque o compilador precisa saber quais recursos são alterados pelo código *assembly* para gerar o código apropriado e evitar otimizações que poderiam resultar em comportamento incorreto.

O formato de um operando de saída em Operandos_de_saída, associado ao nome <variávelC> em C, é:

[[<nomeSimbólico>] “<restriçõesS>” (<variávelC>)]

Para referenciar um operando usando o nome simbólico dentro das instruções em *assembly*, devemos usar a notação %[nomeSimbólico]. Se um nome simbólico não for especificado explicitamente, os operandos são referenciados por seus valores posicionais em Operandos_de_saída, com %0 para o primeiro operando, %1 para o segundo, e assim por diante. A lista <restrições> geralmente começa com =(para sobreescrever o valor existente) e é seguida por restrições como r (registrador), m (endereço de memória) ou rm (preferência por registrador ou memória). Essas restrições indicam ao montador onde alocar preferencialmente o operando durante o processamento das instruções.

O formato de um operando de entrada em Operandos_de_entrada, associado ao nome <variávelC> em C, é:

[[<nomeSimbólico>] “<restriçõesE>” (<variávelC>)]

Assim como em operando de saída, podemos atribuir um nome simbólico a um operando em Operandos_de_entrada ou usar valores posicionais a partir do último valor posicional atribuído aos operandos de saída. A lista <restriçõesE> especifica os locais onde o operando pode ser armazenado durante o processamento das instruções em *assembly*. Ela pode também conter o valor posicional de um operando de saída, o que significa que o operando de entrada e o operando de saída compartilham o mesmo local de armazenamento.

O formato de um operando de Clobbers é:

“<recursos>”

Aqui, um recurso é tipicamente um registrador (por exemplo, r0, r1 etc.), um espaço de memória (memory) ou o registrador de status (cc) que é utilizado pelas instruções de *assembly*.

Abaixo está um exemplo de *inline assembly* que implementa a operação “counter = counter + 1;” em C. Neste código, “counter” é utilizado tanto como operando de entrada quanto de saída. No *inline assembly*, isso é feito com o “counter” localizado no lado direito (entrada) e no lado esquerdo (saída) da atribuição. Os parâmetros de entrada e saída são designados de forma distinta com base na posição e na função do operando na lista de entrada e saída. Portanto,

%0 para se referir ao operando de saída (counter na primeira linha iniciada com ‘:’).

%1 para se referir ao operando de entrada que é a mesma variável (counter na segunda linha iniciada com ‘:’).

Para garantir que o montador use o mesmo registrador para ambos os operandos, é através das restrições especificamos o compartilhamento de registradores por esses operandos. A restrição do operando “%0” é “=r”, indicando que é um operando que pode ser sobreescrito e deve ser armazenado preferencialmente em um registrador, sem especificar qual registrador. E a restrição do operando “%1” é “0” indicando que usa o mesmo registrador usado pelo operando “%0”.

```
int counter=0;
for (;;) {
    asm ("mov r1, %1 \n\t"
        "add r1, #1 \n\t"
        "mov %0, r1 \n\t"
        : "=r" (counter)
        : "0" (counter)
        : "r1"
    );
}
```

Ressalta-se aqui a ausência de funções de entrada e saída, como aquelas definidas no arquivo de cabeçalho “stdio.h” da biblioteca padrão de C, em muitos ambientes de programação de microcontroladores. Isso se justifica pela necessidade de uma interação mais direta e personalizada com os periféricos do *hardware*. Diferentemente de sistemas operacionais de alto nível que abstraem a comunicação com dispositivos através de funções genéricas, a programação de microcontroladores exige que o desenvolvedor configure e gerencie diretamente os registradores dos módulos de comunicação com os periféricos integrados nos microcontroladores. Essa abordagem permite um controle mais preciso e eficiente sobre como os dados são lidos e escritos, adaptando a comunicação às especificidades de cada dispositivo, como portas seriais, ADCs, ou *timers*. Ao longo da disciplina, exploraremos como essas configurações diretas são essenciais para o funcionamento adequado dos sistemas embarcados, mostrando a importância de entender a comunicação em um nível mais detalhado do que o fornecido pelas abstrações de bibliotecas padrão.

TOOLCHAINS

O STM32CubeIDE oferece um conjunto completo de ferramentas (*toolchain*) para geração de códigos executáveis no microcontrolador STM32H7A3ZIT6-Q a partir de duas linguagens de médio nível, C e C++. O [*GNU Arm Embedded Toolchain*](#), desenvolvido e mantido pela FSF (do inglês *Free Software Foundation*), é a ferramenta fundamental por trás desse processo. O STM32CubeIDE inclui um editor de código-fonte baseado no Eclipse, um compilador cruzado (*cross compiler*) GCC para ARM (“arm-none-eabi-gcc”), um montador cruzado (*cross assembler*)

GCC para ARM (“arm-none-eabi-as”), um ligador cruzado (*cross linker*) GCC para ARM (“arm-none-eabi-ld”) e um depurador cruzado (*cross debugger*) GDB (“arm-none-eabi-gdb”). O termo “cruzado” (cross) é utilizado para indicar que a compilação, montagem e ligação dos programas são realizadas em um computador, comumente denominado *host*, diferente do microcontrolador-alvo, onde as instruções são efetivamente executadas. Da mesma forma, a depuração do fluxo de controle executado no microcontrolador-alvo é realizada a partir de um *host*.

A figura a seguir ilustra as etapas de transformação dos endereços das instruções para três funções diferentes de um mesmo programa ao longo de um *toolchain*. Cada função é compilada separadamente, atribuindo a cada instrução um endereço de memória correspondente ao seu deslocamento em relação ao endereço inicial (0) da função. O ligador então combina essas funções, ajustando os endereços das instruções para garantir que não haja sobreposição e estabelecendo um endereço inicial comum (0) para todas as funções. Além disso, o ligador inclui o código de inicialização (*Reset*) recomendado pelo fabricante. Finalmente, com base na definição do *layout* da memória, a ferramenta de carregamento transfere o código executável e os dados para as memórias Flash e RAM do microcontrolador, preparando-o para execução.

