EA871 – LAB. DE PROGRAMAÇÃO BÁSICA DE SISTEMAS DIGITAIS

EXPERIMENTO 2 – Ferramentas de Desenvolvimento

Profa. Wu Shin-Ting

OBJETIVO: Apresentação do ambiente de desenvolvimento de programas para MKL25Z128

ASSUNTOS: Construção de um projeto no ambiente *IDE CodeWarrior 10*: linguagem de programação C, documentação e depuração de programas

O que você deve ser capaz ao final deste experimento?

Ter uma noção da placa de desenvolvimento (*kit*) FRDM-KL25.

Ter uma noção da placa auxiliar (*shield*) FEEC871.

Ter uma visão da organização das ferramentas disponíveis no IDE CodeWarrior.

Saber criar e importar um projeto no IDE CodeWarrior.

Ter uma noção de como um *firmware* (um código executável) é construído.

Saber acessar os registradores do microcontrolador em C.

Saber aplicar o mascaramento de *bits* para modificar seletivamente campos de *bits* de um registrador em C.

Saber distinguir em C os endereços de memória dos valores numéricos.

Saber usar o qualificador `volatile` para evitar otimizações indevidas do compilador C.

Saber documentar os códigos-fonte em C

Ter uma visão da relocação dos endereços de um *firmware*.

Depurar um código executado no microcontrolador em tempo real.

INTRODUÇÃO

Nesta disciplina, os conceitos abordados em EA869 serão revisados e expandidos de uma maneira prática. Para alcançar esse objetivo, faremos uso de uma placa de desenvolvimento conhecida como FRDM-KL25. Desenvolvido pela NXP/*Freescale Semiconductors*, esse *kit* oferece diversas conexões com circuitos digitais, apresentando uma variedade de padrões de interface, além de contar com periféricos analógicos.

Placa FRDM-KL25Z

O "comandante" da placa FRDM é o microcontrolador MKL25Z128VLK4, um dispositivo de 32 *bits* pertencente à série Kinetis L [1]. Esse microcontrolador integraum núcleo de processamento Cortex-M0+ de arquitetura *ARM*® juntamente com vários módulos, como temporizadores, interfaces de comunicação, conversores DA/AD, controlador de interrupção e 3 unidades de memória. Sua combinação de baixo custo, baixo consumo de energia, tamanho reduzido e flexibilidade no desenho de novos projetos o torna uma alternativa ideal para o desenvolvimento de aplicativos portáteis e de alto desempenho.

São integrados na placa FRDM-KL25 um circuito de alimentação e de *clock* (8MHz), um *touchpad* capacitivo, um acelerômetro, e um LED RGB [2]. Além disso, há na placa um adaptador serial e de depuração, *OpenSDA*, que permite que sejam transferidos programas

executáveis (*firmware*) desenvolvidos num computador-hospedeiro para a memória interna do microcontrolador-alvo (MKL25Z) e que os mesmos sejam depuráveis através de um aplicativo instalado no computador-hospedeiro. O adaptador viabiliza ainda a comunicação serial entre o computador-hospedeiro e o microcontrolador-alvo por meio de uma porta mini-USB. Vale notar que o circuito do *OpenSDA* é baseado num outro microcontrolador da família Kinetis K20, o K20DX128VFM5, da *NXP*/*Freescale*.

Shield **FEEC8712**

Em cima da placa FRDM-KL25 está conectada uma placa auxiliar (*shield* FEEC871) com alguns periféricos adicionais, mais especificamente um LCD (*Liquid Cristal Display*), 8 LEDs vermelhos (ligados a um *latch*), 3 botoeiras, e um circuito de chaveamento de média potência baseado em transistor Darlington. Há ainda pinos de acesso direto aos pinos PTB0, PTB1, PTE20, PTE21, PTE22, PTE23 do MKL25Z [3]. Isso proporciona mais alternativas para a prática de programação do microcontrolador-alvo.

Arquitetura RISC

O núcleo integrado no microcontrolador MKL25Z é o **ARM Cortex-M0+**, um processador de **arquitetura RISC** (do inglês *Reduced Instruction Set Computing*). Essa arquitetura se destaca por apresentar um repertório de instruções mais compacto e eficiente em comparação a processadores CISC (do inglês *Complex Instruction Set Computing*). Contudo, ela demanda uma quantidade maior de instruções para executar determinadas tarefas, o que resulta em um consumo maior de espaço de memória. Dado que a gestão eficiente de memória é crítica para microcontroladores, foi introduzido o repertório de instruções **Thumb**, composto por instruções de 16 *bits*, como uma alternativa às instruções de 32 *bits* da arquitetura ARM.

Embora o repertório Thumb seja eficiente, algumas instruções específicas da arquitetura ARM não podem ser codificadas em 16 *bits*. Em resposta a isso, foram acrescentadas ao repertório Thumb algumas instruções de 32 *bits*, denominadas Thumb-2. O núcleo Cortex M0+ oferece suporte tanto ao repertório Thumb quanto a algumas instruções codificadas em 32 *bits* [1]. Vale ressaltar que alguns processadores ARM têm a capacidade de alternar entre os dois estados, ARM (32 *bits*) e Thumb (16 *bits*), utilizando o *bit* 0 do endereço acessado pela instrução BX (0 para o estado ARM e 1 para o estado Thumb). No entanto, é importante observar que o núcleo Cortex-M0+ opera exclusivamente no estado Thumb.

Arquitetura de Von Neumann

Na arquitetura de Von Neumann um sistema computacional digital é constituído de 3 unidades básicas: unidade central de processamento (UCP), unidade de memória e unidade de entrada/saída. Essas unidades são conectadas por barramentos que se distinguem em barramento de dados/instruções, barramento de endereços e barramento de controle. Diferente da sua variante, **a arquitetura de Harvard**, as instruções e os dados na **arquitetura de Von Neumann** são armazenados no mesmo espaço de memória e compartilham os mesmos barramentos de dados e de endereços. A arquitetura do microcontrolador MKL25Z é de Von Neumann. São integradas nele 3 unidades de memória, uma memória Flash de $128x2^{10}x2^{10}$*bytes* (=128Mb), uma memória RAM de $16x2^{10}$*bytes*(=16Kb) e uma memória ROM de $4x2^{10}$*bytes* (=4Kb) (Seção 4.2/página 105 em [1]). As instruções são executadas sequencialmente de acordo com o conteúdo

do contador de programa (PC). Em cada ciclo de execução a instrução contida no endereço do PC é buscada, decodificada em sinais de controle e executada pelo processador, e o conteúdo do PC é atualizado para o endereço da próxima instrução.

Arquitetura de E/S mapeada na mamória

O processador, ou **núcleo ARM Cortex-M0+**, estabelece comunicação com os periféricos por meio da **arquitetura de E/S mapeada na memória**, onde as instruções de acesso aos periféricos e à memória são idênticas. Do ponto de vista da programação dos periféricos e módulos conectados ao processador, os projetistas podem abstrair todo o *hardware* de um periférico através de um conjunto de registradores com endereços pré-fixados pelo fabricante.

O manual de referência dos microcontroladores da subfamília KL25 [1] fornece um mapeamento completo dos espaços de endereços por módulos na Seção 4.2/página 105. Além disso, cada capítulo dedicado a um módulo específico inclui uma seção detalhada que descreve minuciosamente o mapa de registradores dentro desse espaço de endereços. Esses endereços, sendo valores inteiros sem sinal, são interpretados pelo compilador como valores inteiros, a menos que uma conversão explícita seja aplicada. Por exemplo, o registrador de 32 *bits* SIM_SCGC5 (Seção 12.2.9/página 206 em [1]), pertencente ao módulo *System Integration Module*, está mapeado no endereço 0x40048038. Para que este valor seja reconhecido como endereço de um tipo `uint32_t` pelo compilador C, aplicamos a seguinte conversão:

```
(uint32_t *) 0x40048038
```

e para acessos ao conteúdo do registrador `SIM_SCGC5`pelo endereço 0x40048038, basta aplicarmos o operador “valor-de” *:

```
*(uint32_t *) 0x40048038.
```

Figura 1 ilustra como as 3 unidades de memória, FLASH, RAM e ROM, integradas no microcontrolador MKL25Z são mapeadas no espaço de endereçamento do núcleo ARM Cortex-M0+.

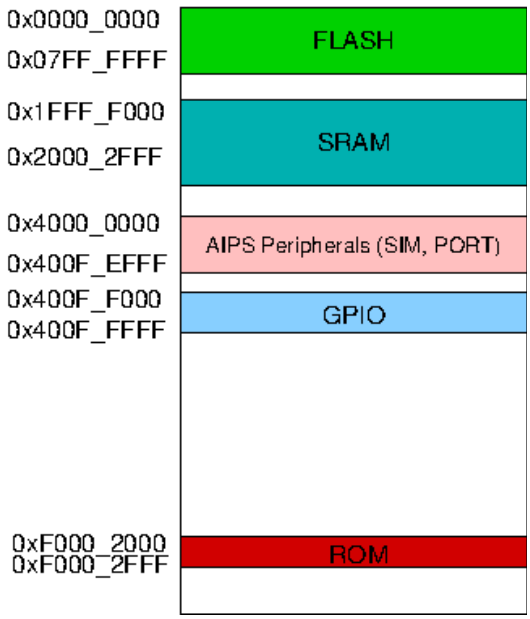

Figura 1: Faixas de endereços de memória em que FLASH, RAM e ROM são mapeadas.

IDE CodeWarrior: Criação de um Projeto

Ao adotarmos o procedimento padrão de criação de um novo projeto no `IDE CodeWarrior`, é gerada automaticamente um projeto contendo arquivos minimamene necessários para gerar um **executável *bare-metal***, de extensão `.elf`. Esses arquivos estão organizados em 3 pastas: `Project_Headers` (arquivos-cabeçalho de extensão `.h`), `Project_Settings` (arquivos de configuração do projeto) e `Sources` (arquivos-fonte de extensão `.c`).

Os arquivos de configuração são caracterizados pela sua baixa volatilidade, sendo organizados em três subpastas dentro do diretório `Project_Settings`: `Debugger`, `Linker_Files` e `Startup_Code`. Na subpasta `Debugger`, encontram-se arquivos (*scripts*) de suporte à depuração dos programas. Já a subpasta `Linker_Files` contém um *script* que fornece dados essenciais, incluindo a definição do nome simbólico `__thumb_startup` para o endereço inicial de execução. Esses dados permitem que o ligador organize e vincule diferentes módulos ou unidades de código para formar o programa executável, além de carregá-lo nos espaços de endereços de memória adequados, conforme mostrados na Figura 1. Por fim, a subpasta `Startup_Code` contém códigos-fonte de instruções comuns de inicialização.

As instruções de inicialização desempenham um papel crucial, abrangendo desde a construção da tabela de vetores de interrupção até a preparação do ambiente de execução, antes de transferir o controle para o programa específico do usuário. Geralmente, tais instruções são recomendadas pelos fabricantes de microcontroladores, sendo implementadas no IDE CodeWarrior como uma adaptação das diretrizes apresentadas na Seção 1.1.4/página 12 em [9]. Adicionalmente, os códigos gerados pelo IDE incluem a chamada da função `main`, localizada no arquivo `main.c` dentro da pasta `Sources`. Com esse ambiente de inicialização, o desenvolvedor é liberado das complexidades iniciais, necessitando apenas focar na personalização da função `main` para atender às especificidades do seu aplicativo. Isso implica que, para reproduzir o executável de um projeto, não é suficiente ter apenas o arquivo `main.c`; todos os outros arquivos dependentes são igualmente essenciais. Uma abordagem segura para exportar o conjunto completo de arquivos dependentes de um projeto é utilizar a função de exportação disponível no ambiente IDE (Seção 2.7/página 39 em [5]).

Programação em C: Mascaramento de *Bits*

Para minimizar a quantidade de registradores endereçáveis nas interfaces de E/S, vários *bits* de controle funcionalmente independentes são compactados num mesmo registrador de maneira que as operações devam ser por *bits*. Podemos construir diferentes **máscaras de *bits*** que nos permitem manipular alguns *bits* específicos de um registrador sem afetar os outros. Denominamos **mascaramento** a operação efetuada com as máscaras de *bits*. Três máscaras de *bits* mais aplicadas 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.

Programação em C: Modificador de Acesso *volatile*

Os periféricos/módulos integrados num microcontrolador têm circuitos de controle dedicados. Esses circuitos tem ações diretas sobre os registradores dos módulos, sem intervenção do núcleo do microcontrolador. A fim de evitar otimizações indevidas dos compiladores sobre instruções que envolvem endereços de registradores cujos conteúdos não são necessariamente modificáveis pelo núcleo, usa-se o **qualificador/modificador de acesso** `volatile`. Recomenda-se aplicar esse qualificador em todos os registradores dos periféricos mapeados na memória [8]. Por exemplo, para assegurar que as instruções envolvendo o registrador SIM_SCGC5 não sejam otimizadas de forma indevida, devemos adicionar o qualificador `volatile` na conversão do valor 0x40048038:

```
(uint32_t volatile *) 0x40048038
```

e para acessos ao conteúdo do registrador `SIM_SCGC5`pelo endereço,

```
*(uint32_t volatile *) 0x40048038.
```

Documentação dos Códigos-fonte

A documentação do código facilita a comunicação entre os desenvolvedores envolvidos em um projeto. Ela não apenas auxilia na identificação e correção de erros, mas também se mostra muito útil para a manutenção do código por terceiros. Além disso, documentação bem elaborada possibilita uma retrospectiva eficiente das decisões tomadas durante o desenvolvimento. Nesta disciplina, optaremos pelo uso do gerador de documentação a partir de códigos-fonte em C, o Doxygen [11]. . Uma breve introdução aos comandos básicos pode ser encontrada na Seção 2.9/página 48 em [5], proporcionando uma visão inicial sobre como utilizar essa ferramenta para criar uma documentação abrangente e compreensível da interface das funções de um projeto.

IDE CodeWarrior: Geração de Código Executável

No roteiro 1 foi apresentado um procedimento para gerar um código executável de extensão ***elf*** com o aplicativo *CodeWarrior* 10, um *IDE* baseado na plataforma *Eclipse* [4,5]. Esse código é então transferido para o microcontrolador-alvo, onde é efetivamente executado. Vale ressaltar que os programas passam por etapas de pré-processamento, compilação e ligação em um processador diferente daquele em que serão executados [7].

No *IDE CodeWarrior*, estão disponíveis **cadeias de ferramentas** (*toolchains*) para geração de códigos executáveis no microcontrolador MKL25Z a partir de duas linguagens de médio nível, C e C++. Esse ambiente integra um conjunto de **ferramentas cruzadas** (*cross tools*) que possibilitam escrever, em uma linguagem de programação, a sequência de ações definidas no

pseudocódigo em um código-fonte (**editor**, Seção 2.2.1/página 12 em [5]). Posteriormente, essas ferramentas são usadas para gerar códigos-objeto com extensão **.o** (linguagem de máquina) a partir dos códigos-fonte (**compilador cruzado**) e, por fim, para construir um código executável no microcontrolador-alvo a partir dos códigos-objeto compilados separadamente (**ligador cruzado,** Seção 2.3/página 18 em [5]). Nesta disciplina, optamos por utilizar a linguagem C para o desenvolvimento dos programas.

Ao traduzir as instruções de um arquivo em linguagem C para os códigos de máquina, o compilador atribui ficticiamente a cada código um endereço relativo ao primeiro código do arquivo. Esse processo pode ser visualizado por meio do comando *Disassemble* do IDE, ano qual os códigos em C são traduzidos para as instruções de máquina (Seção 2.2.5/página 15 em [5]). Cabe ao ligador alocar um espaço de endereços disjunto para cada bloco de códigos compilados, resolver as referências entres eles usando os endereços alocados e gerar um código executável.

IDE CodeWarrior: Implantação de Código Executável

Somente após a transferência desse código para o microcontrolador (Seção 2.4/página 24 em [5]), as instruções e os dados são efetivamente relocados para os endereços especificados no *script* `Project_Settings/LinkerFiles/MKL25Z128_flash.ld`. Figura 2 ilustra os endereços atribuídos aos 3 blocos de instruções (funções) e ao bloco de inicialização típica recomendada pelo fabricante ao longo de um *toolchain* de geração de executáveis.

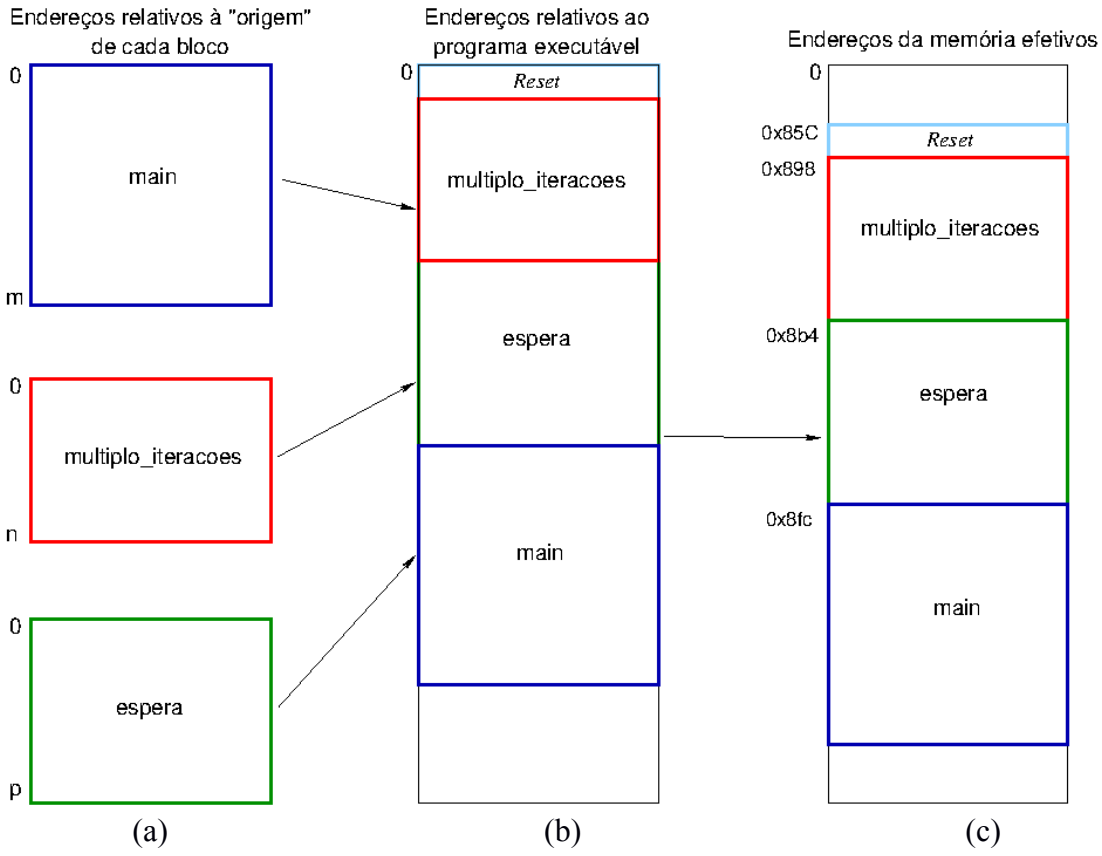

Figura 2: Endereços das instruções *Thumb* (a) após a compilação, (b) após a ligação e (c) após a transferência para o microcontrolador.

Essencialmente, há três tipos de erros que o IDE pode nos ajudar a detectar a fonte: erros sintáticos, erros de dependência, e erros lógicos. Os **erros sintáticos** são tipicamente identificados pelo compilador e são mostrados como erros ou avisos (*warnings*) em *Problem view* e *Console view*. **Não despezem os avisos**. Em muitos casos, a correção dos avisos é suficiente para corrigir alguns erros aparentemente não explicáveis. Os **erros de dependência** entre as funções são identificados pelo ligador. Os **erros lógicos**, que incluem lógica incorreta nos algoritmos, geralmente requerem uma análise mais profunda do código e podem ser descobertos durante a execução do programa por meio de testes e depuração.

Os **erros lógicos**, frequentemente intrincados e contrários à intuição, apresentam desafios significativos na detecção. Para mitigar esses erros, é recomendável expressar a lógica da solução de um problema de maneira gráfica, como em diagramas de blocos, fluxogramas, diagramas de estados, ou por meio de pseudocódigos [9]. Essa abordagem facilita a análise global da lógica do programa. Ao identificar erros, utilize um depurador para acompanhar o fluxo de controle implementado pelo compilador, rotina por rotina, bloco por bloco, ou até linha por linha. Esse processo detalhado de acompanhamento auxilia na detecção e correção eficaz de erros lógicos no código-fonte.

OpenSDA

O adaptador OpenSDA serve como a interface entre o processador-hospedeiro e o microcontrolador-alvo MKL25Z. Por meio dele, realiza-se a transferência do código executável no formato *elf*, um executável capaz de rodar diretamente na circuitaria do microcontrolador sem nenhuma camada adicional de códigos, para a memória FLASH do microcontrolador. Caso ocorra algum erro durante a transferência, é recomendável verificar se o projeto está ativado em *CodeWarrior Projects view* (em negrito e inserido no campo Name), e se a configuração *Debug Settings* (`Settings > Debug Settings`) está habilitada para OpenSDA (Seção 2.4/página 24 em [5]).

Após a transferência do *firmware* do computador-hospedeiro para o microcontrolador no modo de depuração (`Debug`), o IDE executa automaticamente as instruções de inicialização e pausa na primeira instrução da função `main`, aguardando os comandos do desenvolvedor para a execução subsequente das instruções.

IDE CodeWarrior: Endianidade

A menor unidade de uma memória é um *byte*. Quando um dado é definido por mais de um *byte*, há duas formas de ordenar os seus *bytes* na memória. A primeira é alinhar o *byte* menos significativo com o *byte* de endereço de menor valor, conhecida por terminação ***little endian***. E a segunda é alinhar o *byte* menos significativo com o *byte* de endereço de maior valor, denominada por terminação ***big endian***. A forma de ordenação dos *bytes* na memória é configurável no microcontrolador MKL25Z. Para certificarmos a endianidade configurada, podemos verificar essa ordenação através da aba *Memory* na perspectiva Debug (Seção 2.5.6/página 36 em [5]).

IDE CodeWarrior: Depuração

O IDE oferece a capacidade de sincronizar a execução de uma sequência de instruções no microcontrolador com o fluxo de monitoramento no computador hospedeiro por meio da

interface SWD [6]. O depurador integrado ao IDE apresenta uma série de ferramentas organizadas na barra de ferramentas da perspectiva `Debug` (Seção 2.5/página 26 em [5]) e da aba `Debug` (Seção 2.5.2/página 31 em [5]). Essas ferramentas permitem que um desenvolvedor, por meio do OpenSDA, controle o fluxo de execução do programa, possibilitando paradas e passos de execução configuráveis.

Vale atentar ao fato de que nem todos os códigos disponíveis nas bibliotecas do IDE tem seus códigos-fonte acessíveis, como uma operação em pontos flutuantes. As operações em ponto flutuante são emuladas. Somente os códigos de máquina dessas operações são visíveis na aba `Disassembly` (Seção 2.5.5/página 34 em [5]). Quando se tenta acessar as instruções em códigos-fonte aparece na tela uma mensagem de aviso como ilustra na lado esquerdo da Figura 3. Para "saltar" as execuções dessas operações, passo a passo, em códigos de máquina mostrados na aba `Disassembly`, adicione um ponto de parada (*breakpoint*) numa próxima instrução em C que não seja uma operação em ponto flutuante. É importante sempre lembrar que o IDE suporta até 2 *breakpoints* ativos simultaneamente.

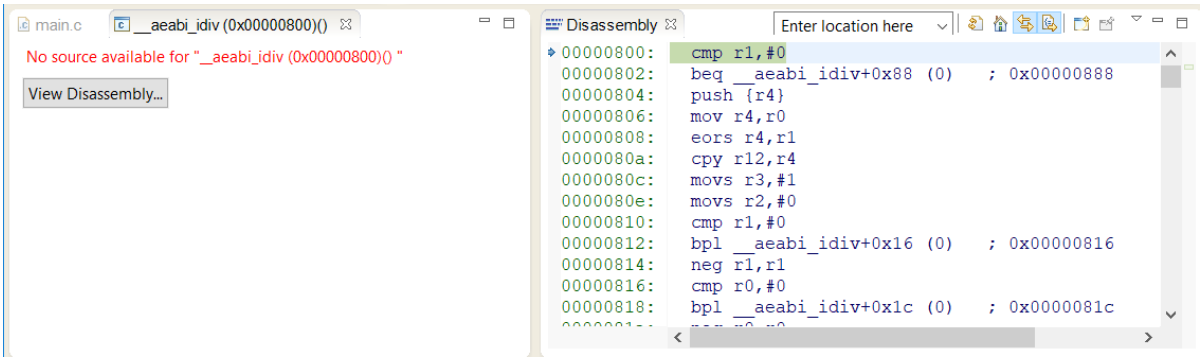

Figura 3: Mensagem de aviso pela falta de códigos-fonte.

A Figura 4 mostra os *views* disponíveis na perspectiva `Debug` do IDE CodeWarrior, através dos quais podemos monitorar os valores das variáveis, inclusive a adição das globais pelo ícone destacado com a seta vermelha (na aba `Variables`, Seção 2.5.3/página 31 em [5]), os pontos de parada setados (`Breakpoints`, Seção 2.5.1/página 28 em [5]), o conteúdo de todos os registradores dos módulos-periférico do microcontrolador mapeados no espaço de memória (`Registers`, Seção 2.5.4/página 33 em [5]), o conteúdo de um espaço de memória (`Memory`, Seção 2.5.6/página 35 em [5]) e os módulos carregados no microcontrolador (`Modules`, Seção 2.5.7/página 38 em [5]).

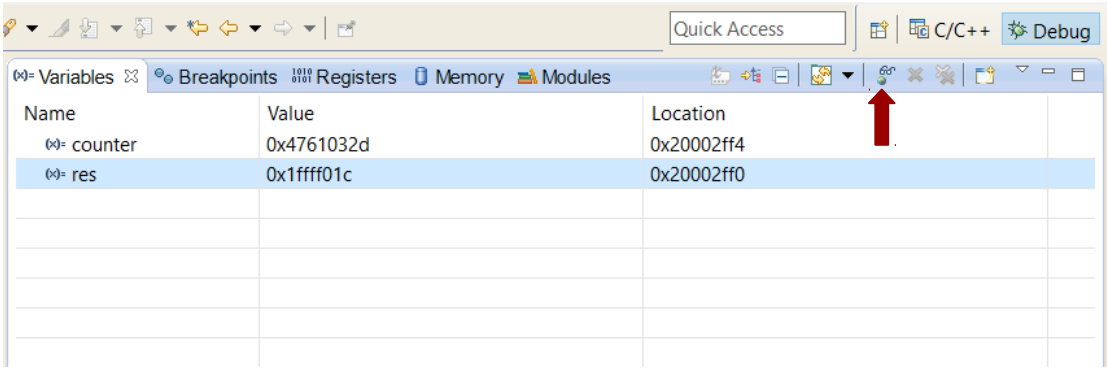

Figura 4: Diferentes *views* na perspectiva de *Debug* padrão: `Variables`, `Breakpoints`, `Registers`, `Memory` e `Modules`.

IDE CodeWarrior: Demanda de Memória

Dentro das ferramentas adicionais do IDE, destaca-se a ferramenta `Print Size`. Figura 5 mostra como se habilita a ferramenta `Print Size`.Esta ferramenta oferece uma visão do tamanho do espaço de memória ocupado pelo *firmware* (Seção 2.3/página 18 em [5]), abrangendo instruções e variáveis declaradas como constantes (`.text`), dados com inicialização de endereços fixos (`.data`), dados sem inicialização de endereços fixos (*block started by symbol* ou `.bss`) [10]. O segmento `.text` é armazenado na memória FLASH, enquanto os segmentos `.data` e `.bss` ocupam os endereços mais baixos da memória RAM. O restante das variáveis, declaradas localmente em cada função, é armazenado na **pilha**, que é alocada para os endereços mais altos da memória RAM. O IDE CodeWarrior reseta, por padrão, o topo da pilha para o endereço 0x20003000 (Seção 2.2/página 10 em [5]).

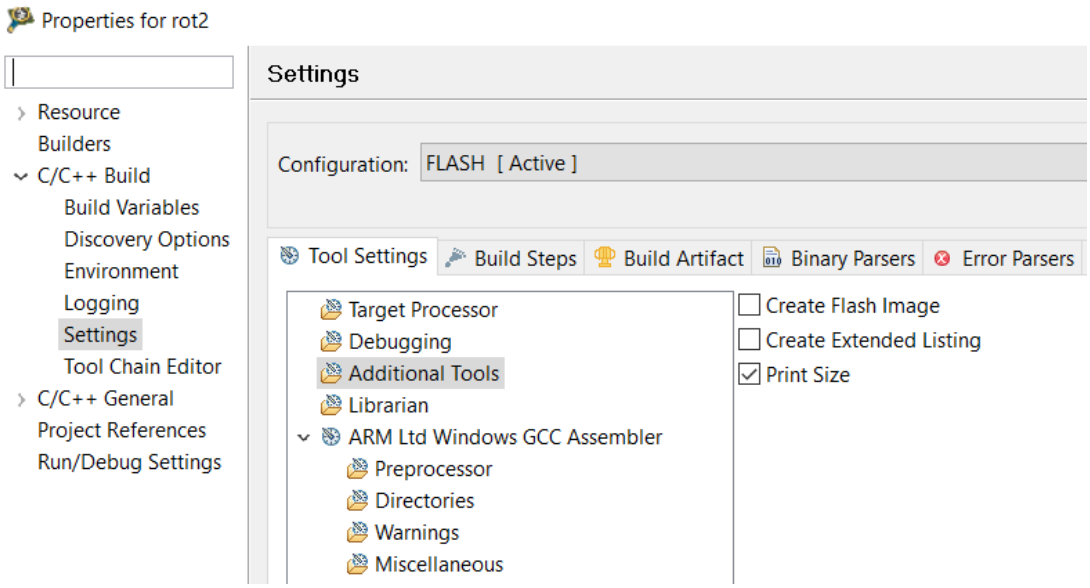

Figura 5: Habilitação da ferramenta `Print Size`.

Se habilitarmos `Print Size`, veremos no final da execução do ligador a quantidade de *bytes* ocupados pelas instruções (`text`), pelos dados inicializados (`data`), pelos dados não inicializados (`bss`, *block starting symbol*) e o total na base decimal (`dec`). Figura 6 apresenta a saída de `Print Size` de um projeto

```
'Invoking: ARM Ltd Windows GNU Print Size'
"C:/Freescale/CW MCU v10.6/Cross_Tools/arm-none-eabi-gcc-4_7_3/bin/arm-none-eabi-size"  --format=berkeley
rot2.elf
   text    data     bss     dec     hex filename
    936      24    2076    3036     bdc rot2.elf
'Finished building: rot2.siz'
' '
```

Figura 6: Espaços de memória ocupados pelo projeto rot2.

IDE CodeWarrior: Importação e Exportação de um Projeto

Um projeto completo desenvolvido no IDE contém diversos arquivos organizados em diferentes pastas. Todos esses arquivos são necessários para gerar um executável. Exportar e importar um projeto para o IDE equivale a exportar e importar todas as pastas relacionadas com o projeto,

respectivamente. A interface do IDE dispõe de comandos tanto para importar como para exportar um projeto. Consulte os procedimentos na Seção 2.7/página 40 em [5].

EXPERIMENTO

No experimento 1, mostramos como se cria um novo projeto no ambiente IDE CodeWarrior [7]. Neste experimento, vamos explorar o projeto `rot2_aula` [12] usando as diversas funcionalidades do ambiente IDE CodeWarrior.

1 **Perspectiva C/C++:** Importe o projeto `rot2_aula` no IDE e abra o projeto clicando o nome do projeto na aba `CodeWarrior Projects`.

1.a **Criação de um Projeto - Inicialização padrão do microcontrolador:** A Seção 1.1.4/página 12 em [9] apresenta uma sequência padrão de instruções para inicializar o microcontrolador MKL25Z antes de transferir o controle para o programa específico. IDE implementou essa sequência na rotina/função `__thumb_startup` dentro do arquivo `__arm_start.c`. Expanda as pastas/subpastas do projeto para encontrar o arquivo e abra-o clicando sobre o nome do arquivo. Compare as instruções implementadas com as sugeridas pelo fabricante. Destaque os passos que foram suprimidos na inicialização-padrão de projetos criados no IDE CodeWarrior.

1.b **Endereço da pilha:** A pilha é uma estrutura de dados essencial na execução de programas nos processadores atuais, oferecendo uma abordagem organizada para gerenciar dados temporários e controlar o fluxo de execução, especialmente durante a chamada das sub-rotinas. O endereço da pilha usada num projeto desenvolvido no IDE CodeWarrior é configurado no *script* `MKL25Z128_flash.ld`. Localize o arquivo e procure nele o endereço da pilha configurado. Além disso, por meio da Figura 1, determine a unidade de memória física em que a pilha é implementada.

1.c **Implantação de Código Executável - Programa específico:** Localize o arquivo `main.c`, que engloba a rotina `main` contendo as instruções específicas relacionadas a uma tarefa de interesse. Deve-se executar primeiro as intruções em `__thumb_startup` antes de transferir o controle para `main`. Qual instrução é responsável pela chamada automática de `main`, de forma que ao carregar apenas o endereço da primeira instrução da rotina `__thumb_startup` no contador de programa (PC) todas as instruções programadas serão executadas automaticamente em sequência?

1.d **Implantação de Código Executável - Tamanho do programa específico:** No IDE, os tamanhos, em *bytes*, dos códigos de máquina correspondentes às rotinas programadas em C, podem ser verificados antes de gerar o executável. Use o comando `Disassemble` (Seção 2.2.5/página 15 em [5]) para verificar os endereços `m`, `n` e `p` atribuídos aos blocos de instruções em *assembly* na Figura 2, e o tamanho, em *bytes*, de cada uma das 3 funções. Certifique-se de incluir na contagem as instruções "adicionais" de "chaveamento de contexto" como parte de instruções traduzidas, de C para *assembly*, para o comando `return`.

1.e **Arquitetura RISC:** O relatório gerado pelo comando `Disassemble` mostra para cada linha de comando em C as instruções em *assmebly* correspondentes. Com base no tamanho das instruções em *assembly*, qual arquitetura de instruções, Thumb ou ARM, é adotada pelo núcleo ARM Cortex-M0+?

1.f **Demanda de Memória do Projeto:** O tamanho do código executável excede a soma dos tamanhos das 3 funções do programa específico. Além do código correspondente ao programa específico, o executável também engloba os códigos de inicialização e depuração. Crie um executável com extensão *elf* (Seção 2.3/página 19 em [5]). Registre os tamanhos de memória, em *bytes*, ocupados pelos segmentos `.text`, `.data` e `.bss`, e o tamanho total do código executável exibidos na aba `Console`.

1.g **Arquitetura de Von Neumann:** A prática de dividir um programa em segmentos de instruções e dados é uma técnica que oferece benefícios em termos de organização, modularidade e gerenciamento eficiente de memória. No entanto, para efetuar essa divisão, é necessário mapear esses segmentos em espaços físicos de memória. Devido à arquitetura de Von Neumann, esses segmentos compartilham o mesmo espaço de memória. No IDE CodeWarrior, o mapeamento é configurado por meio do *script* `MKL25Z128_flash.ld`. Procure no arquivo os espaços de endereços configurados para os três segmentos, sabendo que `m_text` ↔ `.text` e `m_data` ↔ `.data` + `.bss`. Além disso, por meio da Figura 1, determine a unidade de memória física em que cada segmento é implementado.

2 **Perspectiva Debug:** Transfira o código executável para o microcontrolador no modo `Debug`. Automaticamente, a perspectiva do IDE é chaveada para o modo `Debug`.

2.a **Aba Modules:** Na aba `Modules` são listados os *firmwares* carregados no microcontrolador (Seção 2.5.7/página 37 em [5]). Após a transferência, qual *firmware* específico está registrado na aba `Modules`?

2.b **Aba Dissassembly**: A aba `Dissassembly` apresenta as informações dos códigos de máquina a serem executados, incluindo os endereços efetivos de cada instrução. Em quais endereços efetivos são realocadas a função `__thumb_startup` e as 3 funções do arquivo `main.c`? Registre também os endereços efetivos de `m`, `n` e `p` mostrados na Figura 2 e certifique se os endereços das instruções estão dentro do espaço de endereços atribuído para `m_text` identificado no item 1.g.

2.c **Aba Memory/Endianidade:** A aba `Memory` é um mapa do conteúdo do espaço de endereços processável pelo núcleo ARM Cortex-M0+ (Seção 2.5.6/página 34 em [5]). Os endereços da pilha e da primeira instrução do código executável precisam ser predefinidos, pois é a partir deles que o processador inicia sua tarefa de execução. Esses endereços são carregados, respectivamente, nos endereços 0x00000000 e 0x00000004, durante a transferência de um código executável para o microcontrolador. Registre os valores armazenados nesses dois endereços, consultando a aba `Memory`? Qual é a ordenação dos 4 *bytes* que representam o valor de um endereço em relação aos valores dos endereços que os 4 *bytes* ocupam?

2.d **Aba Registers:** A aba `Registers` exibe o conteúdo de todos os registradores do microcontrolador, incluindo os registradores de trabalho no núcleo ARM Cortex-M0+ (Seção 2.5.4/página 32 em [5]). Compare o conteúdo do registrador PC mostrado na aba `Registers` com o valor armazenado no endereço 0x00000004 do espaço de memória e o endereço da primeira instrução da rotina `__thumb_startup` mostrado na aba `Disassembly`. Explique a diferença entre esses valores.

2.e **Arquitetura de E/S mapeada na memória:** Todos os registradores dos módulos integrados no microcontrolador são mapeados no espaço de memória com endereços pré-

fixados, e seus conteúdos podem ser consultados na aba `Registers`. Para identificar os registradores dos módulos `PORT`, `SIM` e `GPIO` que foram usados em `main.c`, consulte as seções 11.5 (página 177), 12.2 (página 192), e 41.2 (página 773) em [1], ou a aba `Registers`. Vale ressaltar que em [1], por questões de legibilidade, os endereços em que cada registrador é mapeado estão em hexadecimal e separados por "_" (*underscore*) em grupo de 4 dígitos. A quais registradores correspondem os endereços usados na função `main`?

2.f **Execução:** Execute o programa carregado no microcontrolador, conforme a explicação dada na Seção 2.5/página 25 em [5]. Descreva a tarefa executada pelo programa.

3 **Depuração:** No modo `Debug` do IDE CodeWarrior, é possível controlar de forma precisa o fluxo de execução do código carregado no microcontrolador. Esse controle inclui a execução das instruções passo a passo, por linha em C ou por instrução em *assembly*. Além disso, é possível a execução por bloco de instruções entre dois pontos de parada predefinidos dinamicamente. Também é viável reiniciar a execução de um programa ao resetar o valor do PC com o endereço da primeira instrução da rotina `___thumb_startup`.

3.a **Aba Breakpoints:** A aba `Breakpoints` permite gerenciar os pontos de parada setados. Siga as instruções descritas na Seção 2.5.1/página 27 em [5], para inserir um *breakpoint* na primeira instrução da rotina `espera` e outro na primeira instrução na rotina `multiplo_iteracoes`. Após configurar esses dois pontos, registre o que é listado na aba `Breakpoints` e reinicie a execução do programa (Seção 2.5.2/página 30 em [5]) para certificar as paradas configuradas. Além disso, justifique se é possível estabelecer um ponto de parada adicional na primeira instrução da rotina `main`.

3.b **Aba Variables**: A aba `Variables` gerencia as variáveis locais usadas na rotina em execução, fornecendo informações como valores e endereços, como vimos no roteiro 1 **[7]**. Reinicie a execução do programa, pulando a inicialização-padrão, com os dois pontos de parada setados. Anote os endereços **que aparecem na aba `Variables`** ao reiniciar a execução a partir da primeira instrução de `main` e nos dois pontos de parada, `espera` e `multiplo_iteracoes`, e associe esses endereços às unidades de memória físicas mostradas na Figura 1.

	Variáveis	Endereço	Flash/RAM (pilha)/RAM (m_data)/ROM
main	teste		
	0x40048038		
	0x4004a04c		
	0x400ff044		
	0x400ff048		
	0x400ff054		
espera	valor		
	contador		
	contador_offset		
	i		
multiplo_iteracoes	valor		

	i		

3.c **Endereços predefinidos pelos fabricantes:** Os endereços predefinidos pelos fabricantes são tratados como constantes em C, e esses valores são armazenados juntamente com as instruções no segmento `m_text`. Geralmente, esses endereços estão localizados no final do bloco de instruções da rotina. Utilize a aba `Memory` para localizar os endereços que contêm os valores das constantes predefinidas, ou seja endereços dos registradores, e insira esses endereços, como também a unidade de memória física, na tabela do item 3.b.

3.d **Variáveis globais:** A existência das variáveis locais está restrita à execução da função em que são declaradas, enquanto as variáveis globais persistem durante a execução do programa. As variáveis globais são exibidas na aba `Variables` somente se forem explicitamente adicionadas (Figura 4). Adicione a variável `teste` na aba `Variables` , conforme as instruções dadas na Seção 2.5.3/página 31 em [5]) e insira o seu endereço, como também a unidade de memória física, na tabela do item 3.b.

3.e **Variáveis estáticas:** As variáveis estáticas, `contador` e `contador_offset`, preservam seu valor entre chamadas de uma função e mantém seu escopo durante a execução do programa, sendo declaradas com o qualificador `static`. O que há em comum entre os endereços das variáveis globais e estáticas? Qual é a diferença entre essas duas classes de variáveis e as variáveis locais em termos da localidade na pilha?

3.f **Alteração do valor de uma variável no tempo de execução**: O que ocorrerá, se dobrarmos o valor da variável `teste` na aba `Variables` enquanto a execução do programa está pasada na linha "`(*(uint32_t volatile *) 0x400FF044u) = (1<<19);` ", e então retomarmos o fluxo do programa (*resume*)? Justifique.

3.g **Alteração do valor de um registrador mapeado na memória no tempo de execução:** Se pausarmos numa linha após a linha "`(*(uint32_t volatile *) 0x400FF044u) = (1<<19);`", qual é o estado do LED verde? O que ocorrerá, se atribuirmos 0 manualmente na aba `Registers` somente o *bit* 19 do registrador `GPIOB_PDOR`? Justifique.

4 **Documentação dos Códigos-fonte:** São incluídos blocos de comentários seguindo as regras de sintaxe do Doxygen (`Seção` 2.9/página 47 em [5]) nas três funções implementadas no arquivo `main.c`.

4.a **Geração de *script* para documentação**: Crie na pasta do projeto rot2_aula um *script* chamado `rot2_aula.doxyfile` para gerar exclusivamente documentação no formato HTML, seguindo o procedimento detalhado na `Seção` 2.9.4/página 50 em [5]. Certifique de que que a documentação inclua os gráficos de dependência entre as três funções contidas em `main.c`.

4.b **Limpeza de Projeto:** Para reduzir o tamanho do projeto salvo, remova-se o código executável e todos os arquivos intermediários entre os códigos-fonte e o executável. Siga o procedimento descrito na Seção 2.3.2/página 22 em [5]. Além disso, remova as pastas html, pdf, etc gerados pelo Doxygen, conforme as instruções na Seção 2.2.3/página 14 em [5].

4.c **Exportação de um Projeto:** Exporte o projeto, contendo o *script* `rot2_aula.doxyfile`, num arquivo comprimido (Seção 2./página 39 em [5]).

RELATÓRIO

O relatório deve ser devidamente identificado, contendo a identificação do instituto e da disciplina, o experimento realizado, o nome e RA do aluno. Para este experimento, responda os itens 1-3 do roteiro num arquivo de formato pdf. Suba-o com o arquivo exportado do item 4, em **arquivos separados**, no sistema *Moodle*.

REFERÊNCIAS

[1] Freescale. KL25 Sub-Family Reference Manual
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/KL25P80M48SF0RM.pdf
[2] FRDM-KL25Z User's Manual – Freescale Semiconductors, Setembro 2012
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/FRDMKL25Z.pdf
[3] Esquemático do *shield* FEEC871.
https://www.dca.fee.unicamp.br/cursos/EA871/references/complementos/Esquematico_EA871-Rev3.pdf
[4] Freescale. *CodeWarrior Development Suite: Eclipse Quick Reference Windows*.
https://www.dca.fee.unicamp.br/cursos/EA871/references/complementos/CWMCUQRCARD.pdf
[5] Wu, S.T. Ambiente de Desenvolvimento – *Software*
https://www.dca.fee.unicamp.br/cursos/EA871/references/apostila_C/AmbienteDesenvolvimentoSoftware_V1.pdf
[6] 2-Pin Debug Port
https://developer.arm.com/architectures/cpu-architecture/debug-visibility-and-trace/coresight-architecture/serial-wire-debug
[7] Roteiro 1
https://www.dca.fee.unicamp.br/cursos/EA871/1s2024/roteiros/roteiro1.pdf
[8] Modificadores de acesso na Linguagem C
https://embarcados.com.br/modificadores-de-acesso-na-linguagem-c/
[9] Freescale. Kinetis L Peripoeral Module Quick Reference
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/KLQRUG.pdf
[10] text, data and bss: Code and Data Size Explained
https://mcuoneclipse.com/2013/04/14/text-data-and-bss-code-and-data-size-explained/
[11] Doxygen
https://www.doxygen.nl/index.html
[12] rot2_aula.zip
https://www.dca.fee.unicamp.br/cursos/EA871/1s2024/codes/rot2_aula.zip

Revisado em Janeiro/2024
Revisado em Janeiro/2023
Revisado em Agosto/2022
Revisado em Fevereiro/2022
Revisado em Julho/2021
Revisado em Março/2021