UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Relatório do Lab 5

Disciplina de Sistemas Embarcados – Prof. Douglas Renaux

Autores: João Felipe Sarggin Machado e Waine Barbosa de Oliveira Junior

Versão: 03-Out-2022

1 Introdução

Um dos tipos de softwares mais comuns e utilizados por todo tipo de dispositivo são os chamados sistemas operacionais (SO). Tais programas oferecem diversas funcionalidades que permitem que o programador tenha acesso a recursos e abstrações que, caso o SO não estivesse ali, teriam que ser escritos na mão e, provavelmente, alterados de plataforma para plataforma. Tendo em vista essas características, é natural que sistemas operacionais estejam presentes na maioria das plataformas (PC, celular, *videogames*, microcontroladores, etc.). Isso porque todas elas têm em comum problemas que são resolvidos pelo SO.

Na área de microcontroladores os sistemas operacionais possuem especificidades o bastante para que sejam chamados por outro nome: *Real Time Operating System* (RTOS ou sistema operacional de tempo real). Isso tem relação com os requisitos de tempo real que são muito comuns em aplicações embarcadas, como o tempo de resposta de uma função ou a necessidade de atendimento e tratamento de um sinal em no máximo *N* microssegundos. O desenvolvimento do sistema operacional deve levar em conta tais requisitos, portanto o RTOS difere consideravelmente de um SO tradicional.

Há vários RTOSs disponíveis para as mais diversas plataformas e microcontroladores. Um deles é o ThreadX, sistema desenvolvido pela *Express Logic* que foi comprada em 2019 pela *Microsoft*. Alguns dos recursos disponíveis são: chaveamento de contexto; comunicação entre *threads;* sincronização entre *threads*; notificação de eventos; gerenciador de arquivos embarcados (FileX), dentre outros.

Neste laboratório são feitos estudos de aplicações utilizando ThreadX, com os objetivos de compreensão das funcionalidades presentes no RTOS e sua operacionalização. Além disso, a correta configuração do projeto no IAR é fundamental para o correto funcionamento do projeto.

A primeira aplicação é a demo "*sample_threadx.c*" fornecida com o código fonte do ThreadX, que utiliza vários recursos do RTOS. Já a segunda é um LED pisca-pisca que permite checar se a temporização do sistema está configurada de maneira correta. Por fim é descrito o processo de inicialização do RTOS, desde o reset do sistema até a inicialização de uma *thread*.

2 Estudo *sample_threadx.c*

Configuração

Foram utilizados os *projetos* já configurados na pasta "*ThreadX/ports/cortex_m4/iar/example_build*". Após a adição de tais projetos no workspace, as seguintes configurações foram aplicadas:

- Certificar que o projeto está configurado para Cortex-M4
- Adicionar o coprocessador numérico
- Na aba de pré-processamento definir o símbolo *TX_ENABLE_IAR_LIBRARY_SUPPORT*
- Na aba *General Options → Library Configuration* selecionar "*Enable thread support in library*"

Tabelas

Thread Name	***Entry function***	***Stack size***	***Priority***	***Auto start***	***Time slicing***
thread 0	thread_0_entry	1024	1	sim	não
thread 1	thread_1_entry	1024	16	sim	sim (4)
thread 2	thread_2_entry	1024	16	sim	sim (4)
thread 3	thread_3_and_4_entry	1024	8	sim	não

thread 4	thread_3_and_4_entry	1024	8	sim	não
thread 5	thread_5_entry	1024	4	sim	não
thread 6	thread_6_and_7_entry	1024	8	sim	não
thread 7	thread_6_and_7_entry	1024	8	sim	não

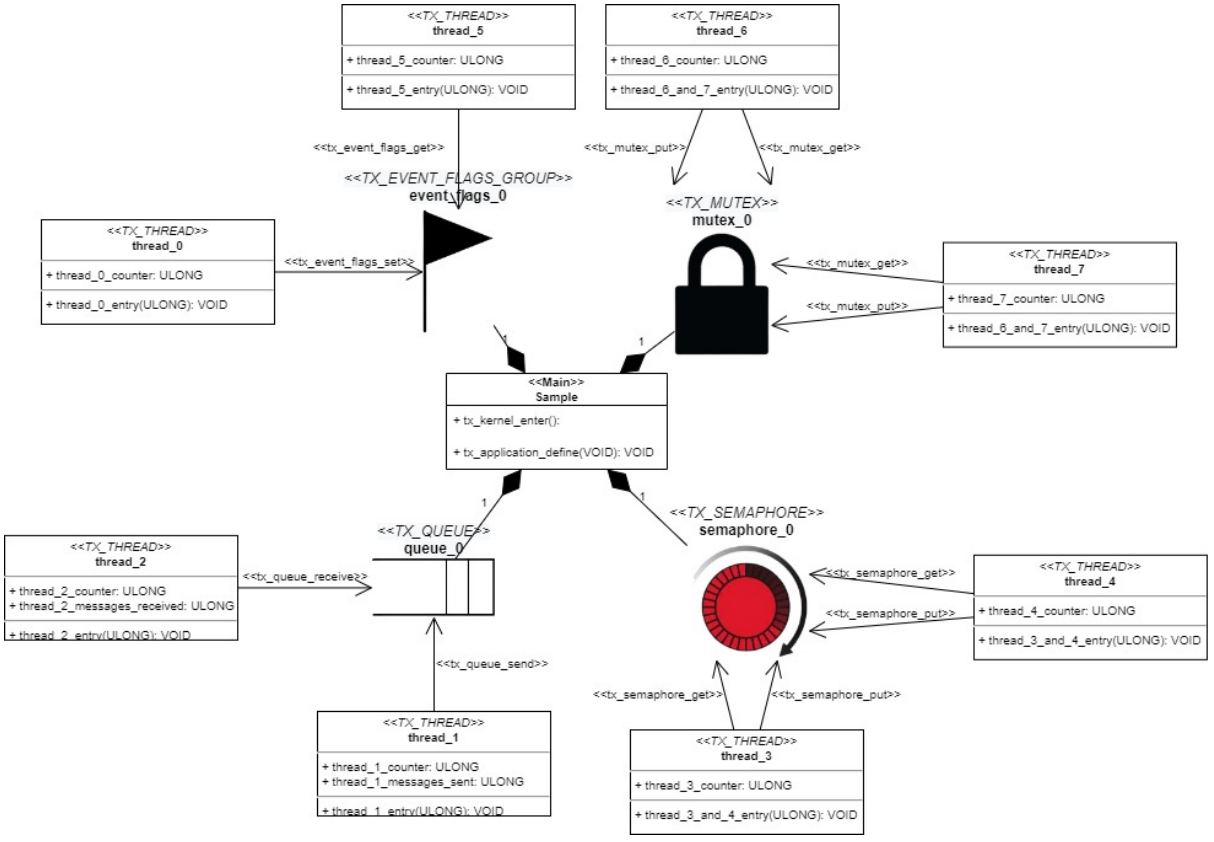
Threads no *sample_threadx.c*

Nome	***Control structure***	***Size***	***Location***
byte pool 0	byte_pool_0	9120	byte_pool_memory
queue 0	queue_0	400	byte_pool_memory
semaphore 0	semaphore_0	32 (tamanho da estrutura)	global memory
event flags 0	event_flags_0	40 (tamanho da estrutura)	global memory
mutex 0	mutex_0	52 (tamanho da estrutura)	global memory
block pool 0	block_pool_0	100	byte_pool_memory

Objetos no *sample_threadx.c*

Diagrama do código

O diagrama do código segue abaixo



O sample representa a main, a qual centraliza todos os recursos, enquanto as threads se comunicam entre si a partir deles.

3 Teste de temporização

Configuração

Após a criação do projeto Lab5 (.ewp), foram feitas as seguintes configurações:

1. Refeito os passos da configuração do estudo *sample_threadx.c*
2. Adicionadas as pastas de *include* do ThreadX
3. Copiados para a pasta *Lab5/src* os seguintes arquivos
    a. *sample_threads.c*
    b. *startup_ewarm.c* (enviado pelo professor)
    c. *tx_initialize_low_levels.s*
4. Inclusão das bibliotecas
    a. *tx.a* a partir de *ThreadX\ports\cortex_m4\iar\example_build\Debug\Exe*
    b. *driverlib.a* a partir de *TivaWare_C_Series-2.2.0.295\driverlib\ewarm\Exe*
5. Configurar o *linker* para utilizar o arquivo *Tiva.icf* (disponibilizado pelo professor)
6. Na aba *Debugger → Plugin* habilitar *ThreadX*.
7. No arquivo *tx_initialize_low_levels.s*, o *define SYSTEM_CLOCK* foi alterado para 120 MHz.
8. Na função main do arquivo *sample_threads.c* foi chamada a função *SysCtlClockFreqSet* do *TivaWare* para ajustar a frequência para 120 MHz, e as funções *SysCtlPeripheralEnable*, *SysCtlPeripheralReady* e *GPIOPinTypeGPIOOutput* para ativar o led de usuário 1.
9. A função *tx_application_define* foi modificada para apenas criar um *byte* pool, alocar desse *byte pool* a quantidade mínima para uma *stack* de *thread* e por fim foi criada uma *thread* com prioridade 1, sem fatia de tempo e com início automático.
10. Por fim, a função *thread_0_entry foi ajustada para alternar o estado do led e dormir por 1 segundo. Fazendo isso indefinidamente dentro de um laço.*

Teste

O teste da corretude do programa foi feito gravando um vídeo da placa e checando que o LED, de fato, piscava de um em um segundo.

4 Rotina de Inicialização

Após o reset ser acionado, uma série de registradores e variáveis serão definidas para um valor padrão (normalmente zero, como é o caso das exceções ativas, do ***PRIMASK***, etc), entre elas, será carregado o endereço do ***Main Stack Pointer (SP)*** a partir do primeiro endereço do vetor de exceções e o endereço da rotina de reset será carregada no ***Program Counter (PC)*** ao final desse primeiro estágio do reset.

Após saltar para a rotina de reset (que nesse exemplo é representada pelo label ***__iar_program_start***), a primeira configuração a ser feita é a inicialização do unidade de ponto flutuante por meio da função ***__iar_init_vfp***. Em seguida ocorre a inicialização das variáveis que possuem valores iniciais (.data) e a inicialização das que possuem valor inicial igual a zero (.bss) por meio das rotinas ***__iar_data_init3***, que chama as rotinas ***__iar_zero_init3*** (para limpar a região .bss com o valor zero) e ***__iar_copy_init3*** (Para copiar os valores das constantes presentes na memória de código para a adequada região da memória RAM .data).

Em seguida a função ***main*** é chamada, onde para muitas aplicações ela simplesmente chama a função ***tx_kernel_enter***, porém as aplicações podem utilizar esse momento para algum

processamento preliminar, como configuração e inicialização de hardware antes de entrar no ***ThreadX***. No caso da simples aplicação do laboratório 5, esse momento foi utilizado para ajustar a frequência do sistema para ***120 MHz*** e para preparar o uso do led de usuário 1, por meio da inicialização do ***GPIO Port N*** e da configuração do pino **N1** como saída ***GPIO***. Ambas as configurações foram feitas por meio das funções do ***TivaWare***.

Quando o fluxo de execução entrar no ***ThreadX***, várias inicializações são realizadas, entre elas a inicialização de baixo nível, que são específicas para cada ***port***. Essa inicialização ocorre na função cujo ***label*** é ***_tx_initialize_low_level*** e está implementada no arquivo .s de mesmo nome em linguagem ***ARM assembly***. Dentre as configurações que essa função realiza estão: Definir o início da memória livre (memória não utilizada que será utilizada pela função ***tx_application_define*** mais tarde), carregar no registrador ***VTOR*** onde se encontra o endereço inicial do vetor de exceções, pegar o valor do endereço do ***stack pointer*** do sistema (a partir do primeiro endereço do vetor de exceções) e armazená lo em uma variável interna do ***ThreadX*** (***tx_thread_system_stack_ptr***), configurar o ***SysTick*** (Definindo seu valor de recarga para um centésimo do valor do ***clock*** do sistema e o configurando para gerar interrupções no ***NVIC***, utilizar o ***clock*** do sistema e o ativando para operar de uma maneira ***multi-shot***) e por fim, configurar as prioridades dos ***handlers de sistema Usage Fault, Bus Fault e Memory Management Faul***t para zero, do ***SVCall*** e do ***PendSV*** para 7 e o do ***SysTick Exception*** para 2.

Após encerradas as inicializações de baixo nível, serão realizadas as inicializações de alto nível, por meio da função cujo ***label*** é ***_tx_initialize_high_level***. Essa função é responsável por inicializar os módulos de controle para cada um dos serviços do ***ThreadX*** (*thread*, *timer*, *semaphore*, *queue*, *event flags*, *block pool*, *byte pool* e *mutex*).

Depois de algumas pós inicializações específicas para o ***port***, a função ***tx_application_define*** finalmente é chamada, onde todos os recursos iniciais da aplicação são definidos. Para o exemplo do laboratório 5, apenas foi necessário criar um ***byte pool***, alocar desse ***byte pool*** uma área para que será utilizada como ***stack*** para a ***thread***, e por fim criar essa ***thread***, que nesse exemplo foi criada com prioridade 1, sem fatia de tempo e com início automático.

Em seguida, o ***ThreadX*** muda seu estado de sistema para "Inicialização Finalizada", algumas configurações de pré escalonamento específicas para o ***port*** são realizadas e o escalonador é chamado, decidindo qual ***thread*** será a próxima a executar, realizando o chaveamento de contexto e por fim saltando para a função de entrada da ***thread***.