

Roteiro 03 - Preparação

Ambiente Keil uVision + CMSIS:RTOS + RedPill

Este tutorial foi traduzido e adaptado a partir do trabalho disponível em [“The Designers Guide to the Cortex-M Processor Family” de Trevor Martin](#).

1) Introdução

Neste tutorial, veremos como usar um RTOS de pequeno porte em execução em um microcontrolador baseado em Cortex-M. Especificamente, vamos usar um RTOS que atenda à especificação RTOS '*Cortex Microcontroller Interface Standard*' (CMSIS).

Esta especificação define uma API RTOS padrão para uso com microcontroladores baseados em Cortex-M. A API CMSIS-RTOS fornece todos os recursos de que precisamos para desenvolver com um RTOS em uma ampla variedade de dispositivos. Também é uma interface padrão para quem deseja desenvolver componentes de software reutilizáveis.

Para executar os exemplos deste tutorial, primeiro é necessário instalar a cadeia de ferramentas MDK-ARM.

O tutorial para instalação foi apresentado no roteiro anterior. Se o instalador do pacote tiver problemas para acessar o pacote remoto, você pode baixá-lo através do [link](#). Selecione novamente o pacote STM32F1xx e salve-o em seu disco rígido. O arquivo pode ser salvo como um arquivo .zip dependendo do navegador que você está usando. Se for salvo como .zip, altere a extensão .zip para .pack, você poderá instalá-lo localmente clicando duas vezes no arquivo STM32F1xx.pack.

2) Acessando a API CMSIS-RTOS

Para acessar qualquer um dos recursos do CMSIS-RTOS em nosso código de aplicativo, é necessário incluir o seguinte arquivo de cabeçalho:

```
#include <cmsis_os.h>
```

Este arquivo de cabeçalho é mantido pela ARM como parte do padrão CMSIS-RTOS.

Para o CMSIS-RTOS Keil RTX, esta é a API padrão. Outros RTOS terão sua própria API proprietária, mas podem fornecer uma camada *wrapper* para implementar a API CMSIS-RTOS para que possam ser usados onde a compatibilidade com o padrão CMSIS for necessária.

Laboratório de Sistemas Operacionais Embarcados (ECOS13)

Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

3) *Threads*

Os blocos de construção de um programa 'C' típico são funções que chamamos para executar um procedimento específico e que então retornam à função de chamada. No CMSIS-RTOS a unidade básica de execução é uma "Thread".

Para construir um programa RTOS simples, declaramos cada *thread* como uma função 'C' padrão e também declaramos uma variável de ID de thread para cada função:

```
void thread1 (void);  
void thread2 (void);  
osThreadId thrdID1, thrdID2;
```

Por padrão, o *scheduler* CMSIS-RTOS estará em execução quando `main()` for inserido e a função `main()` se tornar a primeira *thread* ativa. Uma vez na `main()`, podemos interromper a troca de tarefas do *scheduler* chamando `osKernelInitialize()`.

Enquanto o RTOS está parado, podemos criar mais *threads* e outros objetos. Assim que o sistema estiver em um estado definido, podemos reiniciar o *scheduler* RTOS com `osKernelStart()`.

Você pode executar qualquer código de inicialização que desejar antes de iniciar o RTOS.

```
void main (void){  
    osKernelInitialize ();  
    IODIR1 = 0x00FF0000; // Do any C code you want  
    Init_Thread(); //Create a Thread  
    osKernelStart(); //Start the RTOS  
}
```

Quando as *threads* são criadas, elas também recebem uma prioridade. Se houver um número de *threads* prontas para execução e todas tiverem a mesma prioridade, eles receberão tempo de execução em um modo *round-robin*.

No entanto, se um encadeamento com prioridade mais alta ficar pronto para execução, o *scheduler* RTOS cancelará o encadeamento em execução no momento e iniciará a execução do encadeamento de alta prioridade. Isso é chamado de agendamento baseado em prioridade preemptiva. Ao atribuir prioridades, você deve ter cuidado porque a *thread* de alta prioridade continuará em execução até que entre em um estado de espera ou até que uma *thread* de prioridade igual ou superior esteja pronta para ser executada.

Laboratório de Sistemas Operacionais Embarcados (ECOS13)

Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

4) Criação do primeiro projeto CMSIS-RTOS

Este projeto o guiará pelas etapas necessárias para criar e depurar um projeto baseado em CMSIS-RTOS:

- a) Inicie o µVision e selecione: **Project >> New uVision Project** (Figura 01).

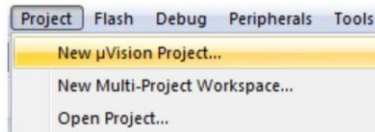


Figura 01 - Project >> New uVision Project

- b) Na caixa de diálogo do novo projeto, insira um nome de projeto e diretório adequados e clique em Salvar.

- c) Em seguida, o banco de dados do dispositivo será aberto. Navegue até o **STMicroelectronics::STM32F103:STM32F103C8** (semelhante à Figura 02).

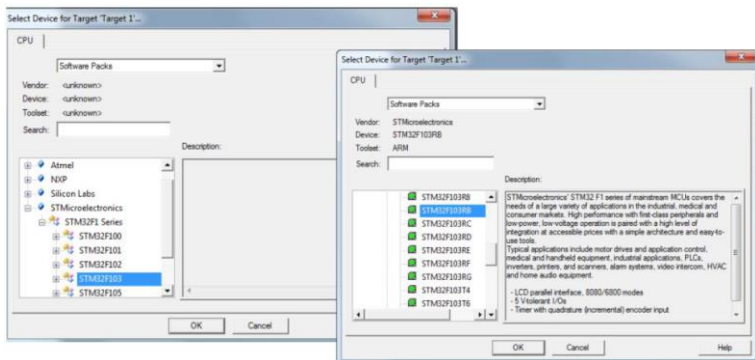


Figura 02 - STMicroelectronics::STM32F103:STM32F103RB

- d) Depois de selecionar este dispositivo, clique em ok (Figura 02).

- e) Assim que a variante do microcontrolador for selecionada, o **Run Time Environment Manager (RTE)** será aberto (Figura 03). Isso permite configurar a plataforma de componentes de software que você usará em um determinado projeto. Além de exibir os componentes disponíveis, o RTE entende suas dependências de outros componentes.

Laboratório de Sistemas Operacionais Embarcados (ECOS13)

Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

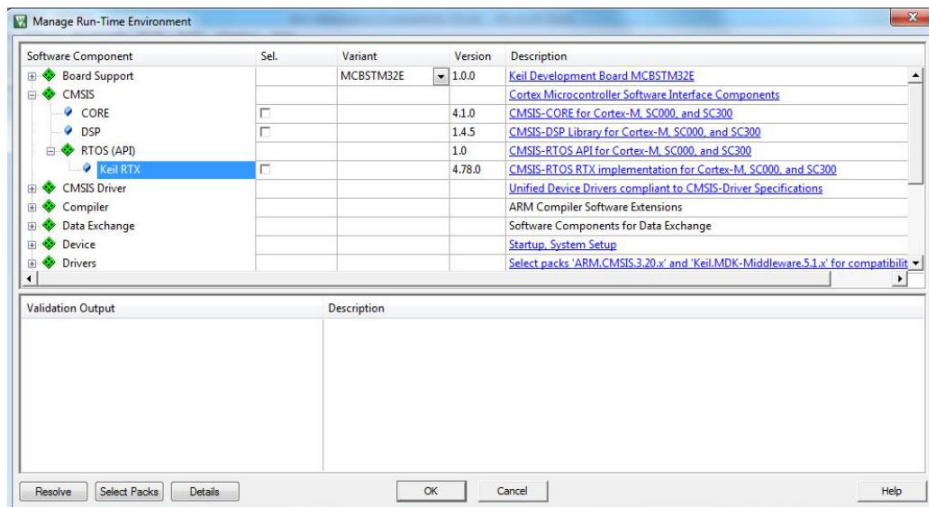


Figura 03 - CMSIS::RTOS (API):Keil RTX

f) Para configurar o projeto para uso com o CMSIS-RTOS Keil RTX, basta marcar a caixa **CMSIS::RTOS (API):Keil RTX** (Figura 03). Após a seleção é necessário resolver suas dependências, que são exibidas na guia *Validation Output*.

g) Agora pressione o botão OK e todos os componentes selecionados serão adicionados ao novo projeto. Os componentes do CMSIS são adicionados às pastas exibidas como um losango verde ◆ (Figura 04).

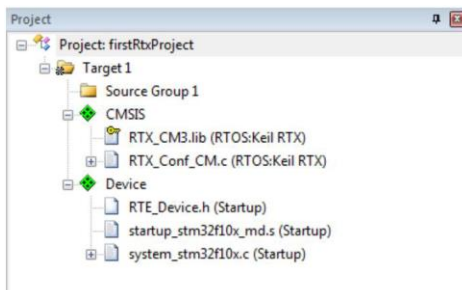


Figura 04 – Componentes do CMSIS

Laboratório de Sistemas Operacionais Embarcados (ECOS13)

Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

5) Ajuste do primeiro projeto CMSIS-RTOS

Existem dois tipos de arquivo na hierarquia do projeto:

- O primeiro tipo é um arquivo de biblioteca que é mantido dentro da cadeia de ferramentas e não é editável. Este arquivo é mostrado com uma chave amarela para mostrar que está 'bloqueado' (somente leitura).
- O segundo tipo de arquivo é um arquivo de configuração.

Cada um desses arquivos pode ser exibido como um arquivo de texto, mas também é possível visualizar as opções de configuração como um conjunto de listas de seleção e menus suspensos.

Para ver isso, abra o arquivo **RTX_Conf_CM.c** e, na parte inferior da janela do editor, selecione a guia 'Assistente de configuração' (Figura 05).

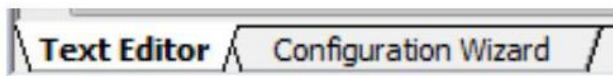


Figura 05 - 'Assistente de configuração'

Clique em Expandir tudo para ver todas as opções de configuração como uma lista de seleção gráfica apresentada na Figura 06.

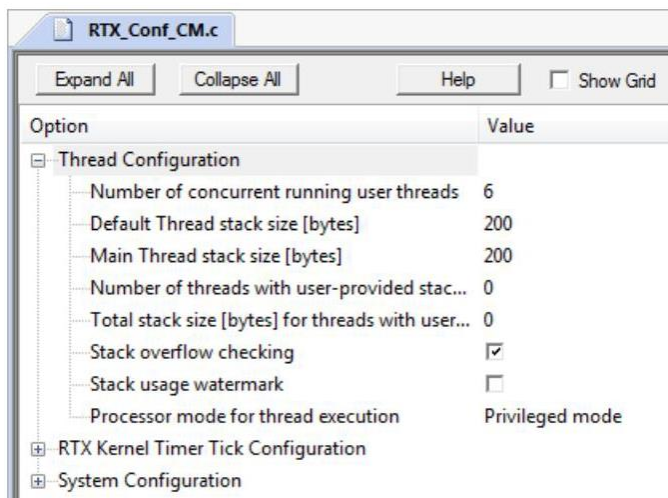


Figura 06 -Lista de seleção Gráfica

Laboratório de Sistemas Operacionais Embarcados (ECOS13)

Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

Por enquanto não é necessário fazer nenhuma alteração aqui e essas opções serão examinadas no futuro.

Agora que temos a plataforma básica para o nosso projeto, podemos adicionar algum código-fonte do usuário que iniciará o RTOS e criará um *thread* em execução.

6) Adicionando arquivos de código-fonte

Para adicionar arquivos de código-fonte:

a) Clique com o botão direito do mouse na pasta 'Source Group 1' e selecione 'Add new item to Source Group 1' (Figura 07):

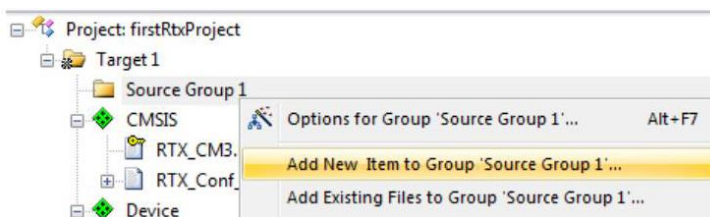


Figura 07 – Adição de arquivos

b) Na caixa de diálogo Adicionar novo item, selecione o ícone 'Modelo de código de usuário' (User Code Template) e na seção CMSIS, selecione a *main function* do CMSIS-RTOS e clique em Adicionar (Figura 08):

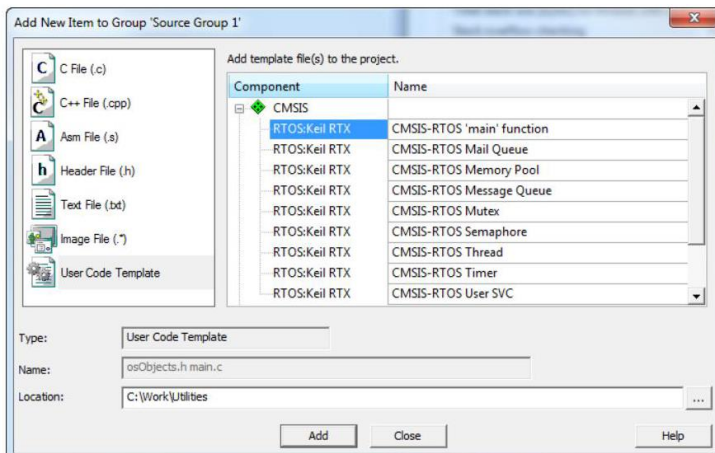


Figura 08 – Modelo de código de usuário (templates)

Laboratório de Sistemas Operacionais Embarcados (ECOS13)

Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

c) Repita os itens **a** e **b**, mas desta vez selecione '**CMSIS-RTOS Thread**'.

d) Estes passos adicionarão dois arquivos de origem ao nosso projeto main.c e Thread.c (Figura 09):

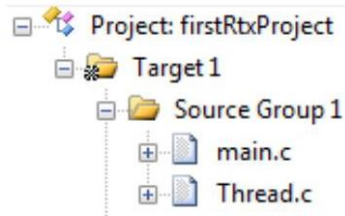


Figura 09 – Arquivos adicionados

7) Analisando o código-fonte

a) Abra o arquivo *Thread.c* no editor. Por enquanto, este arquivo contém duas funções *Init_Thread()* que são usadas para iniciar a execução do encadeamento e a função do encadeamento real.

b) Em *main.c*, adicione o protótipo *Init_Thread* como uma declaração externa e, em seguida, chame-a após a função *osKernelInitialize* conforme mostrado a seguir:

```
#define osObjectsPublic
#include "osObjects.h"
extern int Init_Thread (void); //Add this line

int main (void) {
    osKernelInitialize ();
    Init_Thread (); //Add this line
    osKernelStart ();
}
```

c) Inicialmente, podemos utilizar o simulador de depurador para rodar o código sem a necessidade de hardware externo.

d) Construa o projeto (**Build - F7**)

e) Selecione a pasta 'Target 1' e abra suas opções (Figura 10):

Laboratório de Sistemas Operacionais Embarcados (ECOS13)

Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

- Acesse a guia Debug
- Verifique se 'Dialog DLL' está definido como 'DARMSTM.DLL'
- Verifique se o parâmetro está definido como '-pSTM32F103RB'

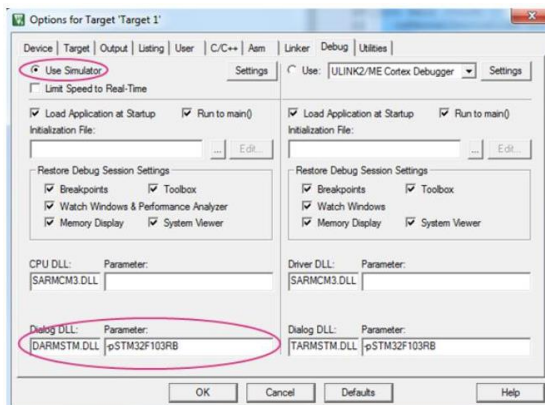


Figura 10 – Configuração de debug

- f) Clique em ok para fechar as opções do menu de destino (Figura 10).

8) Utilizando o debugger

- Inicie o debugger (Ctrl+F5)
- Inicie a execução do código (F5)
- Abra o **Debug >> OS Support >> System and Thread Viewer** (Figura 11).

System and Thread Viewer							
Property	Value						
	Item	Value					
	Tick Timer:	1.000 mSec					
	Round Robin Timeout:	5.000 mSec					
	Default Thread Stack Size:	200					
	Thread Stack Overflow Check:	Yes					
	Thread Usage:	Available: 7, Used: 3 + os_idle_demon					
ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				32%
2	main	Normal	Running				0%
255	os_idle_demon	None	Ready				

Figura 11 – System and Thread Viewer

Laboratório de Sistemas Operacionais Embarcados (ECOS13)

Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

d) Esta visualização de depuração mostra todos os threads em execução e seu estado atual. No momento, temos três threads principais, os `_idle_demon` e `osTimerThread`.

e) Inicie a execução do código (F5).

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				32%
3	Thread	Normal	Running				16%
255	_idle_demon	None	Ready				32%

Figura 12 – System and Thread Viewer (2)

f) Agora a *thread* do usuário é criado e o main são encerrados (Figura 12).

g) Sair do *debugger*

h) Embora este projeto não faça nada, ele demonstra as etapas necessárias para começar a usar o CMSIS-RTOS.

9) Criando Threads

Depois que o RTOS estiver em execução, várias chamadas de sistema são usadas para gerenciar e controlar os encadeamentos ativos.

Por padrão, a função `main()` é criada automaticamente como a primeira *thread* em execução. No primeiro exemplo, nós o usamos para criar uma *thread* adicional e deixá-la. No entanto, se quisermos, podemos continuar a usar `main` como um *thread* por si só. Se quisermos controlar *main* como uma *thread*, devemos obter seu ID de thread.

A primeira função RTOS que devemos, portanto, chamar é `osThreadGetId()`, que retorna o número de ID da *thread* em execução no momento. Isso é armazenado em seu identificador de ID. Quando quisermos nos referir a essa *thread* em futuras chamadas do sistema operacional, usaremos esse identificador em vez do nome da função da *thread*.

```
osThreadId main_id; //create the thread handle
```

```
void main (void){
```

```
    /* Read the Thread-ID of the main thread */
```

```
    main_id = osThreadGetId ();
```

```
    while(1) {
```

```
        //.....
```

```
    }
```

```
}
```

Laboratório de Sistemas Operacionais Embarcados (ECOS13)

Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

Agora que temos um identificador de ID para *main*, podemos criar as *threads* do aplicativo e, em seguida, chamar *osTerminate(main_id)* para finalizar a *thread* principal. Esta é a melhor maneira de terminar este ciclo. Alternativamente podemos adicionar um *loop while(1)* como mostrado acima e continuar a usar *main* em nosso aplicativo.

Como vimos no primeiro exemplo, a *thread* principal é usada como uma *thread* iniciadora para criar as *threads* do aplicativo. Isso é feito em duas etapas. Primeiro, uma estrutura de *thread* é definida; isso nos permite definir os parâmetros de funcionamento:

```
osThreadId thread1_id; //thread handle  
void thread1 (void const *argument); //function prototype for thread1  
osThreadDef(thread1, osPriorityNormal, 1, 0); //thread definition structure
```

A estrutura da *thread* exige que definamos o nome da função da *thread*, sua prioridade, o número de instâncias que serão criadas e seu tamanho de pilha.

Uma vez que a estrutura foi definida, a *thread* pode ser criada usando o *osThreadCreate()* podemos adicionar um *loop while(1)* como mostrado acima e continuar a usar *main* em nosso aplicativo.

Em seguida, a *thread* é criada a partir do código do aplicativo, geralmente dentro da *thread* principal, mas pode estar em qualquer ponto do código.

```
thread1_id = osThreadCreate(osThread(thread1), NULL);
```

Isso cria a *thread* e a inicia em execução. Também é possível passar um parâmetro para a *thread* quando ela inicia:

```
uint32_t startupParameter = 0x23;  
thread1_id = osThreadCreate(osThread(thread1), startupParameter);
```

Quando cada *thread* é criada, ela também recebe sua própria pilha para armazenar dados durante a troca de contexto. Isso não deve ser confundido com a pilha nativa do processador Cortex; é realmente um bloco de memória alocado para a *thread*. Se necessário, um *thread* pode receber recursos de memória adicionais definindo um tamanho de pilha maior na estrutura do *thread*.

```
osThreadDef(thread1, osPriorityNormal, 1, 0); //assign default stack size to this thread  
osThreadDef(thread2, osPriorityNormal, 1, 1024); //assign 1KB of stack to this thread
```


Laboratório de Sistemas Operacionais Embarcados (ECOS13)

Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

No entanto, se você alocar um tamanho de pilha maior para um encadeamento, a memória adicional deverá ser alocada no arquivo de configuração do RTOS.

10) Criando e gerenciando threads

Neste projeto iremos criar e gerenciar algumas *threads* adicionais. Cada uma das *threads* criadas alternará um pino GPIO na porta **GPIO B** para simular o piscar de um LED. Podemos então visualizar esta atividade no simulador.

Acesse o seguinte botão no menu 

Para acessar os projetos de exercícios, abra o instalador do pacote no µVision (Figura 13).

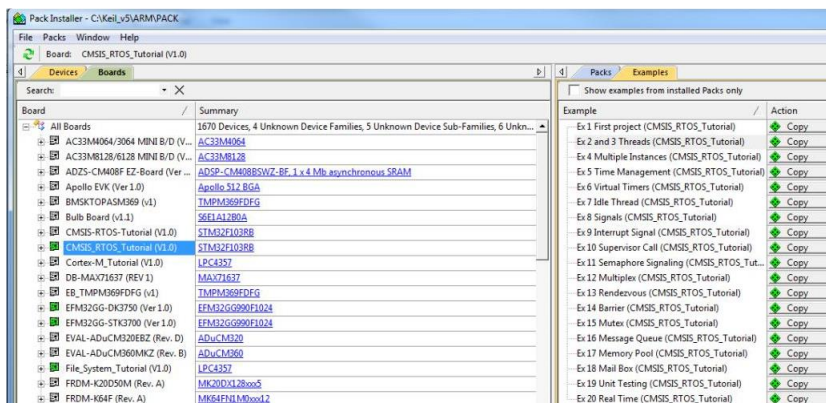


Figura 13 – Packet installer

a) Selecione a guia de placas (*boards*) e selecione o *CMSIS-RTOS_Tutorial*.

b) Selecione a guia exemplos e todos os projetos de exemplo para este tutorial serão mostrados.

- Para exibir em ordem, clique no cabeçalho da coluna cinza 'Exemplo' (*Examples*).
- Uma cópia de referência do primeiro exercício apresentado neste tutorial está incluída como Exercício 1.
- Selecione “*Ex 2 and 3 Threads*” e pressione o botão copiar. Isso instalará o projeto em um diretório de sua escolha e abrirá o projeto no µVision.

c) Abra o Gerenciador de Ambiente de Tempo de Execução (*Manage Run-Time Environment*).

Laboratório de Sistemas Operacionais Embarcados (ECOS13)

Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

d) Na seção de suporte da placa, a caixa **Board Support:MCBSTM32E:LED** está marcada. Isso adiciona funções de suporte para controlar o estado de um banco de LEDs na porta GPIO B do microcontrolador.

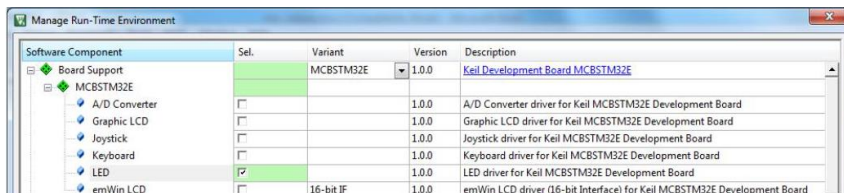


Figura 14 – Gerenciador RTE

e) Quando o RTOS inicia, `main()` é executado como uma *thread* e, além disso, criaremos duas *threads* adicionais. Primeiro criamos *handles* para cada uma das *threads* e depois definimos os parâmetros de cada *thread*. Isso inclui a prioridade em que a *thread* será executada, o número de instâncias de cada *thread* que criaremos e seu tamanho de pilha (a quantidade de memória alocada para ele) zero indica que ela terá o tamanho de pilha padrão.

```
osThreadId main_ID, led_ID1, led_ID2;  
osThreadDef(led_thread2, osPriorityNormal, 1, 0);  
osThreadDef(led_thread1, osPriorityNormal, 1, 0);
```

f) Então, na função `main()`, as duas *threads* são criadas:

```
led_ID2 = osThreadCreate(osThread(led_thread2), NULL);  
led_ID1 = osThreadCreate(osThread(led_thread1), NULL);
```

g) Quando a *thread* é criada, podemos passar um parâmetro no lugar da definição `NULL`.

h) Compile o projeto e inicie o depurador

i) Inicie a execução do código e abra **Debug >> OS Support >> System and Thread Viewer** (Figura 15).

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				32%
3	led_thread2	Normal	Running				0%
4	led_thread1	Normal	Ready				32%
255	os_idle_demon	None	Ready				

Figura 15 – System and Thread Viewer (3)

j) Temos quatro *threads* ativas com uma rodando e as outras prontas.

Laboratório de Sistemas Operacionais Embarcados (ECOS13)

Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

k) Abra o **Debug >> OS Support >> Event Viewer** (Figura 16).

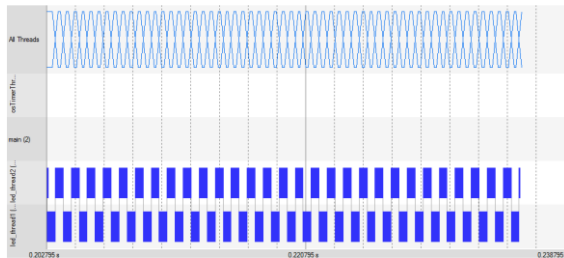


Figura 16 – Event Viewer

l) O visualizador de eventos mostra a execução de cada thread como um rastreamento em relação ao tempo. Isso permite que você visualize a atividade de cada *thread* e tenha uma ideia do tempo de CPU consumido por cada *thread*.

m) Abra **Peripherals >> General Purpose IO >> GPIOB window** (Figura 17).

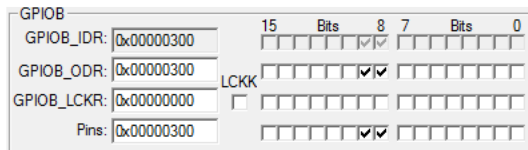


Figura 17 – GPIOB window

n) Nossos dois threads de led estão cada um alternando um pino de porta GPIO. Deixe o código em execução e observe os pinos alternarem por alguns segundos. Se você não vir a atualização das janelas de depuração, verifique se a opção de atualização da janela *view\periodic* está marcada.

```
void led_thread2 (void const *argument) {  
    for (;;) {  
        LED_On(1);  
        delay(500);  
        LED_Off(1);  
        delay(500);  
    }  
}
```

Cada *thread* chama funções para ligar e desligar um LED e usa uma função de atraso entre cada ligar e desligar. Várias coisas importantes estão acontecendo aqui.

Laboratório de Sistemas Operacionais Embarcados (ECOS13)

Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

Primeiro, a função de atraso pode ser chamada com segurança por cada *thread*. Cada *thread* mantém variáveis locais em sua pilha para que não possam ser corrompidas por nenhuma outra *thread*.

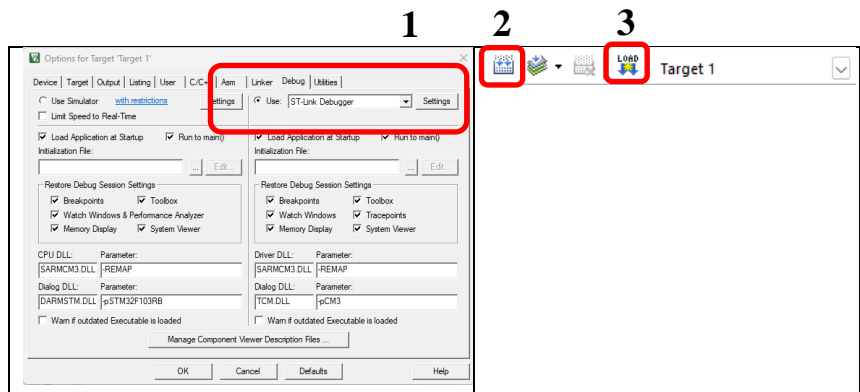
Em segundo lugar, nenhum dos *threads* entra em um estado de espera desprogramado, o que significa que cada um executa sua fatia de tempo alocada antes de mudar para o próximo *thread*.

Como este é um *thread* simples, a maior parte de seu tempo de execução será gasto no loop de atraso, efetivamente desperdiçando ciclos. Por fim, não há sincronização entre as *threads*. Eles estão sendo executados como 'programas' separados na CPU e, como podemos ver na janela de depuração do GPIO, os pinos alternados aparecem aleatoriamente.

11) Entendendo o código-fonte

Após realizar o estudo, a análise e a correta execução dos itens anteriores, faça:

- Compare e analise as diferenças entre o arquivo *main* utilizado no item 10 com o arquivo disponível [neste link](#). Anote as mudanças e alterações percebidas.
- Substitua o código-fonte utilizado no item 10 pelo novo código apresentado na subtópico anterior (disponível [neste link](#)) e execute-o na placa *RedPill*. Para isso é necessário alterar a configuração do ambiente de desenvolvimento para o modo de programação que utiliza o **ST-Link Debugger**, conforme figura a seguir:



- Altere os valores das variáveis *val01_delay* e *val02_delay*, recompile, carregue o código e verifique o funcionamento da placa *RedPill*.