Esse artigo foi escrito pelo engenheiro Ismael Lopes da Silva, exclusivamente para o site "www.embarcados.com.br". O link para o artigo é "https://www.embarcados.com.br/blue-pill-stm32f103c8t6-introducao-ao-freertos/".

No oitavo artigo da série "A Blue Pill", daremos continuidade no estudo. A dica é acompanhar a série desde o primeiro artigo porque é um caminho estruturado, onde a sequência traz benefícios no aprendizado (linha de raciocínio). Nesse artigo finalizaremos a análise do programa miniblink, e daremos início ao Sistema Operacional em Tempo Real de código aberto – FreeRTOS. Criaremos nosso primeiro programa usando FreeRTOS.

Atraso no programa miniblink

No programa anterior, denominado miniblink, ler o sexto artigo da série, podemos ver o seguinte trecho que nos interessa, então, examinaremos o aspecto do tempo de atraso.

```
For (;;)
{
        gpio_clear(GPIOC,GPIO13);          /*LED ascende */
        for (i = 0; i < 1500000; i++)      /*atraso */
        __asm__("nop");
        gpio_set(GPIOC,GPIO13);            /*LED apaga */
        for (i = 0; i < 500000; i++)       /*atraso */
        __asm__("nop");
}
```

A primeira coisa a observar nesse trecho de programa, é que as contagens de loop diferem: 1.500.000 no primeiro e 500.000 no segundo. Isso faz com que o LED permaneça aceso 75% do tempo e apague 25% do tempo.

A instrução __asm __ ("nop") é uma instrução "nop" do assembler do ARM, e foi inserida em ambos os loops. Por que isso é necessário? O problema com um loop vazio é que o compilador pode otimizá-lo. A otimização do compilador está sempre sendo aprimorada e esse tipo de construção pode ser visto pelo compilador como redundante. Esse recurso também é sensível às opções de otimização usadas na compilação. Esse truque __asm__ é uma maneira de forçar o compilador a sempre produzir o código do loop e executar a instrução nop (no operation), e também essa instrução consome ciclo de processamento, adicionando atraso.

O problema com atraso programado

Uma vantagem dos atrasos programados é que eles são fáceis de codificar. Mas, existem vários problemas, que são:

- ✔ Quantas iterações são necessárias para um determinado atraso programado?
- ✔ Portabilidade do código fonte é ruim, porque:
  - ✗ O atraso variará para diferentes plataformas;
  - ✗ O atraso varia de acordo com a taxa de clock da CPU;
  - ✗ O atraso varia de acordo com os diferentes contextos de execução.
- ✔ Desperdiça tempo de CPU, que pode ser usada por outras tarefas em um ambiente multitarefa;

O primeiro problema é a dificuldade de calcular o número de iterações necessárias para obter um atraso. Essa contagem de loop depende de vários fatores, como a seguir:

- ✔ A taxa de clock da CPU;
- ✔ Os tempos do ciclo de instruções utilizados, incluindo a instrução "nop";
- ✔ Ambiente único ou multitarefa.

No programa miniblink, não havia uma velocidade de clock da CPU estabelecida. Conseqüentemente, esse código está à mercê do que foi usado. Se executarmos os mesmos loops da SRAM em vez da flash, os atrasos serão menores. Isso ocorre porque não há ciclos de espera necessários para buscar as palavras de instrução na SRAM. A busca por instruções na flash, por outro lado, pode envolver ciclos de espera, dependendo da velocidade de clock da CPU. Em um ambiente multitarefa, como o FreeRTOS, os atrasos programados são uma péssima escolha. Um dos motivos é que você não sabe quanto tempo é consumido pelas outras tarefas.

Os atrasos programados não são portáveis para outras plataformas. Talvez o código fonte seja reutilizado em um dispositivo STM32F4, onde a eficiência da execução é diferente. O código precisará de intervenção manual para corrigir a deficiência de tempo. Por todos esses motivos pelos quais o FreeRTOS fornece uma API para tempo e atraso. Isso veremos quando aplicarmos o FreeRTOS em nossos programas.

FreeRTOS

Veremos o FreeRTOS, que é um Sistema Operacional em Tempo Real de código aberto, disponível gratuitamente. O código-fonte do FreeRTOS é licenciado sob a licença GPL 2. Faremos uma transição para o uso do FreeRTOS, porque isso oferece várias vantagens, principalmente a programação se torna muito mais simples e com maior confiabilidade. Nem todas as plataformas são capazes de suportar Sistema Operacional em Tempo Real.

Num sistema multitarefa cada tarefa exige que seja alocado algum espaço de pilha (stack) onde são armazenadas variáveis e endereços de retorno de chamadas de função. O MCU STM32F103C8T6, contido na Blue Pill, possui 20K de SRAM disponíveis para dividir entre um conjunto razoável de tarefas.

Recursos do FreeRTOS

O que torna um RTOS desejável? O que um RTOS tem a oferecer? Examinaremos algumas dos principais recursos encontrados no FreeRTOS:

- ✔ Tarefas múltiplas e agendamento;
- ✔ Fila de mensagens;
- ✔ Semáforos e mutexes(1);
- ✔ Temporizadores;
- ✔ Grupos de eventos.

(1) Um objeto de exclusão mútua (mutex) é um objeto de programa que permite que vários threads de programa compartilhem o mesmo recurso, mas, não simultaneamente.

Tarefas

MCUs com aplicações que não utilizam RTOS, tudo é executado como uma única tarefa, com uma única pilha para variáveis e endereços de retorno. Esse estilo de programação exige que você execute um loop varrendo cada evento a ser atendido. A cada iteração, pode ser necessário pesquisar o sensor de temperatura e, em seguida, chamar outra rotina como parte do loop para transmitir esse resultado.

Com um RTOS, em nosso casso FreeRTOS, as funções lógicas são colocadas em tarefas separadas que são executadas independentemente. Uma tarefa pode ser responsável pela leitura e cálculo da temperatura atual. Outra tarefa pode ser responsável pela transmissão da última temperatura calculada. De fato, torna-se um par de programas em execução ao mesmo tempo.

Para aplicações muito simples, essa sobrecarga de agendamento de tarefas pode ser vista como um exagero, mas, à medida que a complexidade aumenta, as vantagens de particionar o problema em tarefas tornam-se muito mais vantajoso.

O FreeRTOS é muito flexível. Ele fornece dois tipos de agendamento de tarefas:

- Multitarefa preemptiva(2);
- Multitarefa cooperativa.

(2) - Preemptividade, algumas vezes preempção, é o ato de interromper temporariamente uma tarefa sendo executada por um sistema operacional, sem exigir sua cooperação, e com a intenção de retomar à tarefa posteriormente.

Com a multitarefa preemptiva, uma tarefa é executada até ficar sem tempo, ou torna-se bloqueada ou gera controle explicitamente. O agendador de tarefas gerencia qual tarefa será executada a seguir, levando em consideração as prioridades. Esse é o tipo de multitarefa que usaremos.

Outra forma de multitarefa são as cooperativas. A diferença é que a tarefa atual é executada até abrir mão do controle. Não há intervalo de tempo ou tempo limite. Se nenhuma chamada de função ocorrer, uma função de rendimento deve ser chamada para entregar o controle para outra tarefa. O agendador de tarefas decide qual tarefa passar o controle para a próxima. Essa forma de agendamento é desejável para aplicativos críticos de segurança que precisam de controle rigoroso do tempo da CPU.

Fila de Mensagens

Assim que você adota a multitarefa, você herda um problema de comunicação. Usando o exemplo de leitura de temperatura, como a tarefa de leitura de temperatura comunica com segurança o valor à tarefa de transmissão de temperatura? Se a temperatura é armazenada como quatro bytes, como você passa esse valor sem interrupção? A multitarefa preemptiva significa que a cópia de quatro bytes de dados para outro local pode ser interrompida parcialmente.

Uma maneira grosseira de resolver isso seria inibir interrupções ao copiar sua temperatura para um local usado pela tarefa de transmissão. Mas essa abordagem pode ser intolerável se houver interrupções frequentes. O problema piora quando os objetos a serem copiados aumentam de tamanho.

O recurso de fila de mensagens no FreeRTOS fornece uma maneira segura de tarefas de comunicar uma mensagem completa. A fila de mensagens garante que apenas as mensagens completas serão recebidas. Além disso, limita o comprimento da fila para que uma tarefa de envio não consuma toda a memória. Usando um comprimento de fila predeterminado, a tarefa de adicionar mensagens fica bloqueada até que o espaço esteja disponível. Quando uma tarefa é bloqueada, o agendador de tarefas alterna automaticamente para outra tarefa pronta para execução, que pode remover mensagens da mesma fila. O comprimento fixo fornece à fila de mensagens uma forma de controle de fluxo.

Semáfaros e Mutexes

Na implementação de uma fila, há uma operação mutex em funcionamento. O processo de adição de uma mensagem pode exigir várias instruções para ser concluído. No entanto, em um sistema multitarefa preemptivo, é possível que uma mensagem seja parcialmente adicionada antes de ser interrompida para executar outra tarefa.

No FreeRTOS, a fila é projetada para incluir mensagens de maneira atômica. Para fazer isso, algum tipo de dispositivo mutex é usado nos bastidores. O mutex é um dispositivo de tudo ou nada. Semelhante aos mutexes, existem semáforos. Em algumas situações em que você pode limitar um certo número de solicitações simultâneas, por exemplo, um semáforo pode gerenciar isso de maneira atômica. Pode permitir um valor máximo de três, por exemplo. Então, até três solicitações de "aceitação" serão bem-sucedidas. Solicitações adicionais de "recebimento" serão bloqueadas até que um ou mais pedidos de "entrega" sejam feitos para devolver o recurso.

Temporizadores

Os temporizadores são importantes para muitas aplicações, incluindo as que piscam LEDs. Quando você tem várias tarefas que consomem tempo da CPU, uma rotina de atraso não é apenas não-confiável, mas, também rouba o tempo de CPU de outras tarefas que poderiam ter sido usadas de maneira mais produtiva.

Dentro de um sistema RTOS, geralmente há uma interrupção "systick" que ajuda no gerenciamento de tempo. Essa interrupção systick não apenas rastreia o número atual de "ticks" emitidos até o momento, mas, também é usada pelo agendador de tarefas para alternar tarefas.

No FreeRTOS, você pode optar por atrasar a execução por um número especificado de ticks. Isso funciona observando o "tick time" atual e cedendo a outra tarefa até que o "tick time" necessário chegue. Dessa maneira, a precisão do atraso é limitada apenas ao intervalo de ticks configurado. Também permite que outras tarefas realizem um trabalho real até chegar o momento certo.

O FreeRTOS também possui a facilidade de temporizadores de software que podem ser criados. Somente quando o cronômetro expirar é executado o retorno de chamada da função. Essa abordagem é econômica para a memória, porque todos os temporizadores farão uso da mesma pilha.

Grupos de eventos

Um problema que ocorre com frequência é que uma tarefa pode precisar monitorar várias filas ao mesmo tempo. Por exemplo, uma tarefa pode precisar bloquear até que uma mensagem chegue de uma das duas filas diferentes. O FreeRTOS fornece a criação de "conjuntos de filas". Isso permite que uma tarefa seja bloqueada até que uma mensagem de qualquer uma das filas do conjunto tenha uma mensagem.

E os eventos definidos pelo usuário? Grupos de eventos podem ser criados para permitir que bits binários representem um evento. Uma vez estabelecida, a API do FreeRTOS permite que uma tarefa aguarde até que ocorra uma combinação específica de eventos. Os eventos podem ser acionados a partir do código de tarefa normal ou de um Rotina de Serviço de Interrupção (ISR).

O Programa blinky2 usando FreeRTOS

Mude para o diretório blinky2, conforme mostrado a seguir:

```
~$ cd stm32f103c8t6/rtos/blinky2
~/stm32f103c8t6/rtos/blinky2$
```

Nesse subdiretório, você deve encontrar o arquivo-fonte do programa main.c.

```
/* Simples pisca LED, usando atrasos temporizados
*
* O LED na porta PC13 alterna na task1
*
*/
#include "FreeRTOS.h"
#include "task.h"
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>

extern void vApplicationStackOverflowHook (xTaskHandle *pxTask, signed portCHAR *pcTaskName);

void vApplicationStackOverflowHook (xTaskHandle *pxTask __attribute((unused)),
        signed portCHAR *pcTaskName __attribute((unused)))
{
        for(;;);
}

static void task1 (void *args __attribute((unused)))
{
        for (;;)
        {
                gpio_toggle (GPIOC,GPIO13);
                vTaskDelay (pdMS_TO_TICKS(500));
        }
}

int main (void)
{
```

```
	rcc_clock_setup_in_hse_8mhz_out_72mhz ();
	rcc_periph_clock_enable (RCC_GPIOC);
	gpio_set_mode (
		GPIOC,
		GPIO_MODE_OUTPUT_2_MHZ,
		GPIO_CNF_OUTPUT_PUSHPULL,
		GPIO13);

	xTaskCreate (task1,"LED",100,NULL,configMAX_PRIORITIES-1,NULL);
	vTaskStartScheduler ();

	for (;;);
	return 0;
}
```

O programa blinky2 usa a API do FreeRTOS para implementar um pisca-pisca em um ambiente RTOS. A parte superior do arquivo de código-fonte main.c, contém os arquivos de inclusão, que incluem:

- FreeRTOS.h;
- task.h;
- Libopencm3/stm32/rcc.h;
- Libopencm3/stm32/gpio.h.

Vimos os arquivos de cabeçalho da biblioteca libopencm3 na aplicação anterior. O arquivo de cabeçalho task.h define macros e funções relacionadas à criação de tarefas, e o arquivo FreeRTOS.h, que todo projeto precisa para personalizar e configurar o FreeRTOS.

Após as inclusões programa main.c define o protótipo de função para a função denominada vApplicationStackOverflowHook(). O FreeRTOS não fornece um protótipo de função para ele, portanto, devemos fornecê-lo aqui para evitar que o compilador se queixe.

Depois do protótipo de função, temos a definição da função opcional vApplicationStackOverflowHook (). Esta função pode ter sido deixada de fora do programa sem causar problemas. É fornecido aqui para ilustrar como você a definiria, se quisesse. Se a função estiver definida, o FreeRTOS a chamará quando detectar que ultrapassou um limite de pilha. Isso permite que o programador do aplicativo decida o que deve ser feito sobre isso. Você pode, por exemplo, querer piscar um LED vermelho especial para indicar falha no programa.

Na sequência temos a tarefa faz o LED piscar. Essa função, chamada task1, aceita um argumento, que não usamos neste exemplo. O __attribute ((unused)) é um atributo do compilador gcc, para indicar ao compilador que o argumento args não está sendo utilizado e evita avisos sobre ele. O corpo da função task1() é muito simples. Dentro do loop infinito temos a função que alterna o estado da GPIO PC13, e a função de atraso de 500 ms. A função vTaskDelay() requer que o número de ticks de atrasado. Geralmente, é mais conveniente especificar milissegundos. A macro pdMS_TO_TICKS() converte milissegundos em ticks de acordo com a sua configuração do FreeRTOS.

Depois, para finalizar, temos o corpo principal do programa. A função main() é definida como retornando um int, mesmo que essa função nunca deva retornar neste contexto do MCU. Isso simplesmente satisfaz o compilador que está em conformidade com os padrões POSIX (Portable Operating System Inferface). A declaração de retorno, penúltima linha, nunca é executada.

A primeira função dentro de main ilustra algo novo - o estabelecimento da velocidade do clock da CPU. Para o seu dispositivo Blue Pill, normalmente você deseja chamar essa função para obter o melhor desempenho. Ela configura os clocks para que o HSE, que é o oscilador externo de alta velocidade, use o cristal de 8 MHz, multiplicado por 9 (implícito), por um PLL (Phase-Locked Loop), para atingir uma velocidade de clock de 72 MHz. Sem essa chamada, teríamos que confiar no clock RC.

Na segunda função dentro de main, o clock GPIO para a porta C é habilitado. Este é o primeiro passo para o alinhamento de configuração que vimos no artigo anterior, para termos sucesso na configuração. A próxima função define o restante desse alinhamento de configuração da porta que utilizaremos, portanto, a porta PC13 é configurada como saída, com frequência máxima de 2 MHz, e do tipo push-pull.

Dando sequência, uma nova tarefa é criada, usando a função xTaskCreate(), e nomeando a tarefa como “task1”. Atribuímos à tarefa um nome simbólico de "LED", que pode ser o nome de sua escolha. O terceiro argumento especifica quantas palavras de pilha são necessárias para o espaço de pilha. Observe a ênfase nas "words". Para a plataforma STM32, uma “word” (palavra) são quatro bytes. A estimativa do espaço da pilha geralmente é complicada e há maneiras de medi-la. Por enquanto, aceite que 400 bytes, isto é, 100 “words” será suficiente. O quarto argumento, um ponteiro que aponta para todos os dados que você deseja passar para sua tarefa, e como não precisamos aqui, então, especificamos NULL. Esse ponteiro é passado para o argumento args na task1(). O quinto argumento especifica a prioridade da tarefa. Temos apenas uma tarefa neste exemplo, além da tarefa main, então, simplesmente damos uma alta prioridade. O último argumento permite que um manuseador da tarefa seja retornado se fornecermos um ponteiro. Não precisamos que o manuseador seja retornado, portanto NULL é fornecido.

Criar uma tarefa sozinha não é suficiente para iniciá-la. Você pode criar várias tarefas antes de iniciar o agendador de tarefas. Depois de chamar a função FreeRTOS vTaskStartScheduler(), as tarefas serão iniciadas a partir do endereço da função que você nomeou no primeiro argumento. Tenha cuidado ao escolher as funções a serem chamadas antes do início do agendador de tarefas. Algumas das funções mais avançadas podem ser chamadas somente após a execução do agendador, mas, existem outras que podem ser chamadas apenas antes do início do agendador. Uma vez que o agendador de tarefas esteja em execução, nesse caso, ele nunca retornará e executará a próxima instrução, a menos que o agendador esteja parado. Caso retorne, é habitual colocar um loop infinito, como em nosso, após a chamada do agendador, para impedir que encerre a função main, com a instrução return 0.

Construir e testar o programa blinkly2

Com o programador ST-Link conectado ao dispositivo, faça o seguinte:

```
~/stm32f103c8t6/rtos/blinky2$ make clobber
~/stm32f103c8t6/rtos/blinky2$ make
~/stm32f103c8t6/rtos/blinky2# make flash
```

O comando ‘make clobber’ exclui todos os componentes previamente construídos (build), de modo que o comando ‘make’ construa (build) novamente o projeto. O comando ‘make flash’ invocará o utilitário st-flash para gravar o novo programa na Blue Pill. Se necessário, pressione o botão Reset, mas, o programa normalmente iniciar por conta própria.

O código mostra que o LED conectado ao pino PC13 deve mudar de estado a cada 500 ms. Se você tem um osciloscópio, poderá confirmar que isso realmente acontece. Isto não apenas confirma que o programa funciona conforme o planejado, mas, também confirma que nosso arquivo FreeRTOS.h foi configurado corretamente.

Execução do programa

O programa blinkly2 é bastante simples, mas, vamos ver o que está acontecendo:

• A função task1() é executada, alternando o funcionamento do LED conectado ao pino PC13. Ela é cronometrada pelo recurso de timer através do uso da função vTaskDelay().

• A função main() chamou a função vTaskStartScheduler(). Isso dá controle ao agendador do FreeRTOS, que inicia e alterna entre várias tarefas. A função main() continuará sendo executado pelo FreeRTOS dentro do agendador, a menos que uma tarefa pare o agendador.

A tarefa task1() possui uma pilha alocada do heap para executar, fornecemos 100 palavras. Se essa tarefa fosse excluída, esse armazenamento seria retornado ao heap. Atualmente, a tarefa principal está em execução no agendador do FreeRTOS, usando a pilha que foi fornecida.

Embora esses itens possam parecer pontos elementares, é importante saber onde os recursos estão alocados. Aplicativos maiores precisam alocar cuidadosamente a memória e a CPU, para que nenhuma tarefa se torne inativa. Isso também descreve a estrutura geral de controle que está em operação.

O uso de multitarefa preemptiva requer nova responsabilidade. O compartilhamento de dados entre tarefas requer a utilização de disciplinas seguras para encadeamento. Este exemplo simples contorna o problema porque há apenas uma tarefa. Projetos maiores exigirão comunicação entre tarefas, e veremos isso nesta série.

Conforme a proposta inicial estamos evoluindo no aprendizado, portanto, aqui concluo o oitavo artigo da série.