No décimo segundo 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 continuaremos utilizando o periférico USART, que é um Transmissor e Receptor Universal Síncrono e Assíncrono, rodando no FreeRTOS. Trabalharemos em uma nova aplicação, digamos melhorada.

O Projeto UART2

O arquivo-fonte desse projeto possui algumas melhorias, com relação ao projeto anterior. Primeiro, existe um novo arquivo de cabeçalho chamado “fila.h”, fornecido pelo pacote FreeRTOS. Isso permite que manuseador de fila de mensagens, a função **static QueueHandle_t uart_txq**, seja declarada.

Mude para o seguinte diretório:

```
~$ cd ~/stm32f103c8t6/rtos/uart2
```

A seguir temos o programa melhorado, que envia dados através da porta serial. O arquivo-fonte pode ver encontrado em “/stm32f103c8t6/rtos/uart2/uart.c”:

```
// Cabeçalho do programa uart.c
#include <FreeRTOS.h>
#include <task.h>
#include <queue.h>
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/usart.h>

// Instancia um manuseador de fila
static QueueHandle_t uart_txq;

// Configuração do periférico UART
static void uart_setup (void)
{
	rcc_periph_clock_enable (RCC_GPIOA);
	rcc_periph_clock_enable (RCC_USART1);

	// UART TX pino PA9 (GPIO_USART1_TX)
	gpio_set_mode (
		GPIOA,
		GPIO_MODE_OUTPUT_50_MHZ,
		GPIO_CNF_OUTPUT_ALTFN_PUSHPULL,
		GPIO_USART1_TX);

	usart_set_baudrate (USART1,38400);
	usart_set_databits (USART1,8);
	usart_set_stopbits (USART1,USART_STOPBITS_1);
	usart_set_mode (USART1,USART_MODE_TX);
	usart_set_parity (USART1,USART_PARITY_NONE);
	usart_set_flow_control (USART1,USART_FLOWCONTROL_NONE);
```

```
	usart_enable (USART1);

	// Cria uma fila para os dados a serem transmitos da UART
	uart_txq = xQueueCreate (256,sizeof(char));
}

// Tarefa USART
static void uart_task (void *args __attribute__((unused)))
{
	char ch;
	for (;;)
	{
		// Recebe o carácter para ser TX
		if ( xQueueReceive (uart_txq,&ch,500) == pdPASS)
		{
			while (!usart_get_flag (USART1,USART_SR_TXE))
				// Solicita outra tarefa até estar pronto
				taskYIELD ();
				// Envia carácter pela USART
				usart_send (USART1,ch);
		}
		gpio_toggle (GPIOC,GPIO13); // Pisca o LED PC13
	}
}

// Coloca na fila o carácter para ser TX
static void uart_puts (const char *s)
{
	for ( ; *s; ++s )
	{
		xQueueSend (uart_txq,s,portMAX_DELAY);
	}
}

// Tarefa de demosntração
static void demo_task (void *args __attribute__((unused)))
{
	for (;;)
	{
		uart_puts ("Now this is a message..\n\r");
		uart_puts ("  sent via FreeRTOS queues.\n\n\r");
		vTaskDelay (pdMS_TO_TICKS(1000));
	}
}

// Programa Main
int main (void)
{
	rcc_clock_setup_in_hse_8mhz_out_72mhz ();
```

```
	// Configura LED PC13
	rcc_periph_clock_enable(RCC_GPIOC);
	gpio_set_mode (
		GPIOC,
		GPIO_MODE_OUTPUT_2_MHZ,
		GPIO_CNF_OUTPUT_PUSHPULL,
		GPIO13);

	// Configura o periférico USART1
	uart_setup ();

	// Cria duas tarefas para o agendador FreeRTOS gerenciar
	xTaskCreate (uart_task,"UART",100,NULL,configMAX_PRIORITIES-1,NULL);
	xTaskCreate (demo_task,"DEMO",100,NULL,configMAX_PRIORITIES-1,NULL);
	// Ativa o agendador de tarefa do FreeRTOS
	vTaskStartScheduler ();

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

Analise do programa

A função **static void uart_setup** (void), que configura o periférico USART, praticamente permanece a mesma, exceto pela criação da fila de mensagens, usando a função **xQueueCreate** (256,sizeof(char)). Essa função cria uma fila de mensagens que conterá no máximo 256 mensagens, cada uma com um comprimento de 1 byte. A variável **uart_txq** é instaciada e recebe um manuseador de fila válido.

A função para escrever caracteres agora é executada a partir da função **uart_task** (), que está agendada como uma tarefa, na função **main** (). A função **uart_task** () opera dentro de um loop for. Dentro desse loop a função FreeRTOS **xQueueReceive** (uart_txq,&ch,500) é chamada para obter uma mensagem. O último argumento 500 dessa função, indica que esta chamada deve atingir o tempo limite após 500 ticks. Ao atingir o tempo limite, o LED PC13 alterna para indicar que o programa ainda está ativo. A função **xQueueReceive** () retorna pdFAIL quando atinge o tempo limite.

Quando **xQueueReceive** () retorna pdPASS, no entanto, a tarefa recebeu uma mensagem. A mensagem é recebida como um único caractere através da variável “ch”. Recebermos um carácter da fila, e precisamos enviá-lo para o UART.

Dentro do loop for, na função **uart_task** (), temos um loop while. Isso chama a função FreeRTOS **taskYIELD** (), até que o UART possa aceitar outro caractere. A biblioteca libopencm3 fornece a função **usart_get_flag** () para permitir o teste de vários flags do Registrador de Status da USART. Dessa maneira, o bit TXE, do Registro de Status da USART, transmissão vazia, é testado. Enquanto esse flag indique que "não está vazio", a função **taskYIELD** () solicita o agendador executar outra tarefa, portanto, passa o controle para o agendador.

Se não der controle, simplesmente ficaria aguardando o UART ficar pronto. Se o UART não estivesse pronto a tempo, essa situação consumiria o tempo da CPU até que o intervalo de tempo da tarefa acabasse. Essa espera ainda parece funcionar bem para esse aplicativo, mas, desperdiçaria tempo de CPU que poderia ser usado de forma otimizada. Como o flag TXE indica que o UART está pronta para transmitir, então, podemos usar a função **usart_send** (), para enviar dados.

As funções **uart_task** () e o **demo_task** () são tarefas do agendador do FreeRTOS, que são executadas concorrentemente. A função **demo_task** (), enfileira pares de linhas a serem enviadas. Essas duas tarefas são criadas na função **main**(), conforme mostradas a seguir. A ordem de criação não é importante, porque o agendador trabalha com prioridades.

xTaskCreate (**uart_task**,"UART",100,NULL,configMAX_PRIORITIES-1,NULL)
xTaskCreate (**demo_task**,"DEMO",100,NULL,configMAX_PRIORITIES-2,NULL)

O função **demo_task** () chama uma nova função, que é a função **uart_puts** (), simplesmente para colocar cada caractere de uma string na UART. Um ponto importante é que a chamada da função **xQueueSend** (uart_txq,s,portMAX_DELAY) será bloqueada se a fila ficar cheia. O terceiro argumento especifica portMAX_DELAY para que fique bloqueiado até que seja bem-sucedido. Como essa é uma chamada do FreeRTOS, o controle é dado para outra tarefa quando a fila estiver cheia.

Este é um resumo geral de como o programa funciona:

- ✔ A tarefa **demo_task** () chama a função **uart_puts** () para enviar seqüências de texto para o UART, com um segundo de diferença.
- ✔ A função **uart_puts** () enfileira os caracteres em uma fila de mensagens referenciada pelo manuseador de fila **uart_txq**. E se a fila está cheia, o controle é passado para **demo_task** ().
- ✔ A função **uart_task** () remove da fila os caracteres recebidos da fila, referenciados pelo manuseador de fila **uart_txq**.
- ✔ Cada caractere recebido é entregue para a UART, para ser enviado, desde que esteja pronto. Quando o UART está ocupado, o controle é entregue para outra tarefa.

Embora este tenha sido um exemplo simples, podemos ver a elegância do FreeRTOS em ação. Uma tarefa produz enquanto outra consome. Os loops de controle para ambos são triviais. Ao particionar um aplicativo em tarefas, dividimos o problema em componentes gerenciáveis. Vemos que a comunicação entre tarefas pode ser realizada com segurança por meio de uma fila de mensagens do FreeRTOS.

Build e Flash o programa

Verifique se o adaptador serial USB foi desconectado antes de conectar o programador. Com o programador pronto, apague compilação anterior (clobber), build o projeto e escreva na Blue Pill (flash) da seguinte maneira:

```
~/stm32f103c8t6/rtos/uart$ make clobber
~/stm32f103c8t6/rtos/uart$ make
~/stm32f103c8t6/rtos/uart$ make flash
```

Depois que o programa for escrito na Blue Pill, e começar a ser executado, o LED PC13 piscará e produzirá texto pela USART. Para vê-lo, você precisará usar um emulador de terminal no seu Desktop/Notebook. Desconecte o programador ST-Link V2 e conecte o adaptador serial USB, conforme detalhamos no artigo anterior.

Com a energia aplicada no adaptador serial, a Blue Pill será alimentada, então, o LED de PWR acende e o LED PC13 pisca. Usei o programa terminal minicom, conforme detalhamos no artigo anterior. Não detalharei novamente esses passos, porque no artigo anterior isso já foi realizado, portanto, basta consultá-lo se tiver dúvida. ATENÇÃO! Jamais alimente a Blue Pill com mais do que uma fonte.

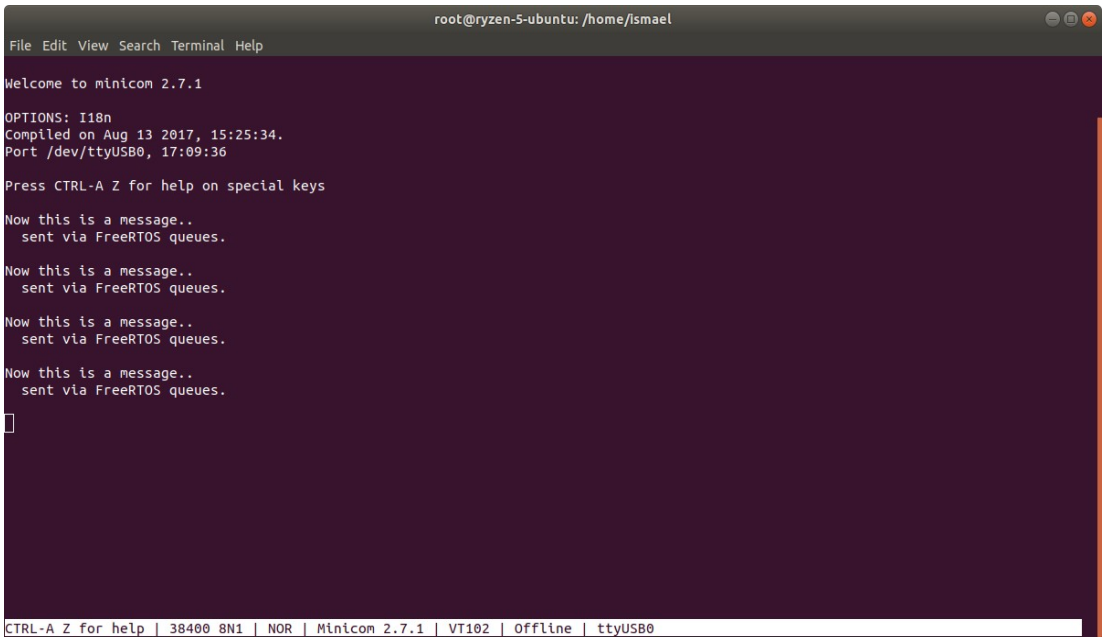

Figura 1 – Mensagens sendo recebidas pelo minicom

Para encerrar o programa minicom pressione CTRL+A, e depois pressione a tecla "Q", e escolha a opção "Yes", para encerrar a execução do minicom.

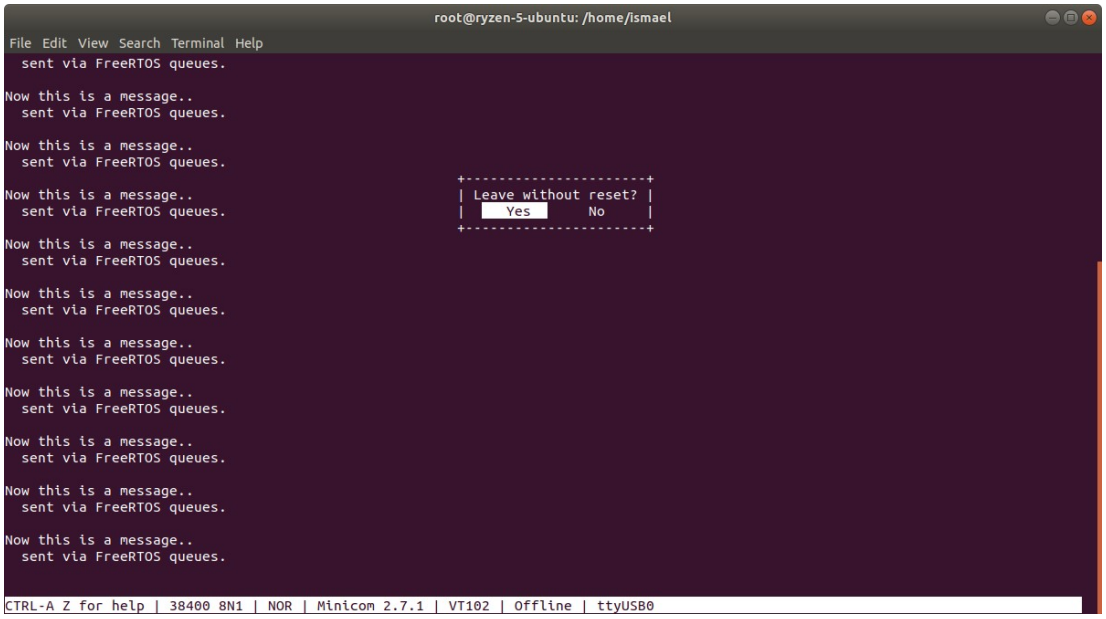

Figura 2 – Saindo do minicom

Depois que o minicom é fechado, o driver USB é encerrado, então, é seguro desconectar o adaptador serial.

Análise de sinal com o Analisador Lógico Saleae 24MHz 8 Canais

Antes de concluir esse artigo, eu coletei dados da porta LED PC13 e PA9 TX. Para minha surpresa eu observei um comportamento que eu não esperava. Observei que o sinal do LED alterna a cada byte transmitido.

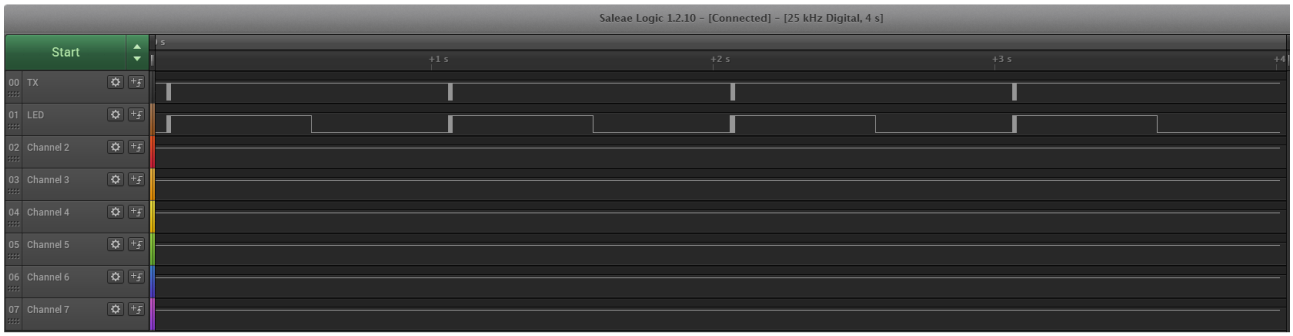

Figura 3 – Sinais da porta PC13 e PA9

Na próxima ilustração eu destaquei a transmissão de pacote de mensagens. Observe que o sinal do LED oscila numa frequência não perceptível ao olho humano.

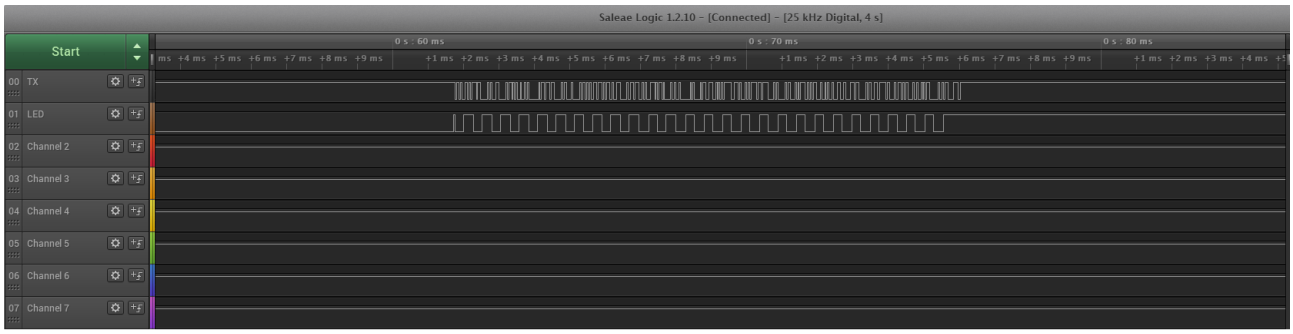

Figura 4 – Sinais da porta PC13 e PA9,

Como destaquei no primeiro artigo dessa série, os artigos são com base no livro do Warren Gay – Beginning STM32, então, analisando o comportamento do sinal do LED, obviamente é explicado porque a instrução **gpio_toggle** (GPIOC,GPIO13) foi inserida dentro da função que envia as mensagens, mas, ao meu ver, conceitualmente o autor não foi feliz.

Eu enviei um e-mail para o Warren Gay, reportando esse detalhe, e ele prontamente respondeu explicando porque o LED oscila. Não bem isso que eu queria saber, mas, ele foi muito atencioso, e eu não quis insistir no assunto. No meu pensamento, o conceito de multitarefa, nesse caso usando FreeRTOS, para cada tarefa independente deveria ser realizada sem interferir na outra. Em nosso caso uma tarefa é piscar e LED e a outra é enviar as mensagens via USART.

Obviamente que o Warren Gay é um mestre no assunto, e facilmente corrigiria esse comportamento. Apenas teríamos uma outra tarefa exclusiva para piscar o LED. Para essa aplicação isso não afeta, então, sem problemas. Apenas é importante ficar atento porque em outra aplicação poderia ser um problema. Outro aprendizado é que quando possível analisar os dados reais, porque tem coisas que não são percebidas a olho nu. Conforme a proposta inicial estamos evoluindo no aprendizado, portanto, aqui concluo mais um artigo da série.