

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-usart/>".

No décimo primeiro 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 usaremos o periférico USART, que é um Transmissor e Receptor Universal Síncrono e Assíncrono, rodando no FreeRTOS.

O Periférico USART/UART

O Transmissor e Receptor Universal Síncrono e Assíncrono, conhecido como USART, é um dos primeiros periféricos a estudarmos, devido sua relevância. Veremos que os termos USART e UART são usados de forma intercambiável. A diferença entre os dois é que USART é a abreviação de Transmissor e Receptor Universal Síncrono e Assíncrono, e UART apenas descarta a função síncrona.

Os periféricos USART/UART enviam dados serialmente através de um fio. Um fio é usado para transmitir dados (TX) e outro para receber dados (RX). Está implícita uma conexão comum de terra entre os dois pontos de extremidade. A comunicação síncrona às vezes requer uma extremidade para atuar como mestre, e fornecer um sinal de clock. A comunicação assíncrona não usa um sinal de clock separado, mas, exige que ambas as extremidades concordem com precisão em uma taxa de clock - conhecida como taxa de transmissão. A comunicação assíncrona começa com um bit de início e termina com um bit de parada para cada caractere transmissor ou recebido.

Os periféricos USART fornecidos pelo STM32F103 são bastante flexíveis. Eles podem realmente funcionar como USART ou UART, dependendo da configuração. Iniciaremos com o modo assíncrono para simplificar o primeiro contato com esse periférico.

Dado Assíncrono

Para exemplificar vamos transmitir um byte (8 bits) assíncrono, contendo o valor 0x65. Este sinal começa com a linha inativa, em nível alto. O início da transmissão é marcado por um bit em nível baixo, conhecido como bit de início (start bit). Isso alerta o receptor que os bits de dados serão transmitidos serialmente. Os bits menos significativos são transmitidos primeiro. Após transmitir os oito bits, o final da transmissão é marcado por um bit de parada (stop bit). Se o bit de parada não for detectado pelo receptor, um erro ocorrerá.



Figura 1 – Transmissão assíncrona de um byte 0x65

Os valores enviados podem ser configurados para ter 8 ou 9 bits de comprimento. Quando ativado, o último bit é o bit de paridade. O(s) bit(s) de parada finalizam a transmissão de um caractere, e podem ser configurados como 0.5, 1, 1.5 ou 2 bits de comprimento.

Adaptadores Serial USB

Um adaptador serial USB TTL é uma coisa extremamente útil quando se trabalha com microcontroladores. Com pouca conexão, você pode usar um programa de terminal, no seu Desktop/Notebook, para se comunicar com o STM32. Isso elimina a necessidade de uma tela LCD e teclado.

Se você ainda não adquiriu um, aqui estão algumas dicas:

- ✓ Ele deve ser um adaptador "TTL" (sinais em +5 volts ou +3,3 volts);
- ✓ O dispositivo USB deve ser suportado pelo seu sistema operacional;
- ✓ Deve suportar controle de fluxo de hardware (RTS e CTS).

Os adaptadores seriais TTL, tem seus sinais entre zero e +5 volts e podem ser usados com qualquer uma das entradas tolerantes a 5 volts. Felizmente, a ST Microelectronics providenciou que a linha do Receptor (RX), para UART 1 e 3, possuísse entradas tolerantes a 5 volts. Quando o STM32 envia sinal de 3,3 volts, funciona bem, porque o valor desse sinal está acima do limite necessário para o Receptor do adaptador.

Certifique-se de trabalhar com um adaptador que suporte controle de fluxo de hardware. Isso incluirá conexões RTS e CTS. Sem controle de fluxo de hardware, não suportará taxas de comunicação mais altas, como 115.200 bauds, sem perder dados.

Conexões

Trabalharemos com um adaptador serial USB TTL conectado ao seu Desktop/Notebook, então, a seguinte conexões serão necessárias:

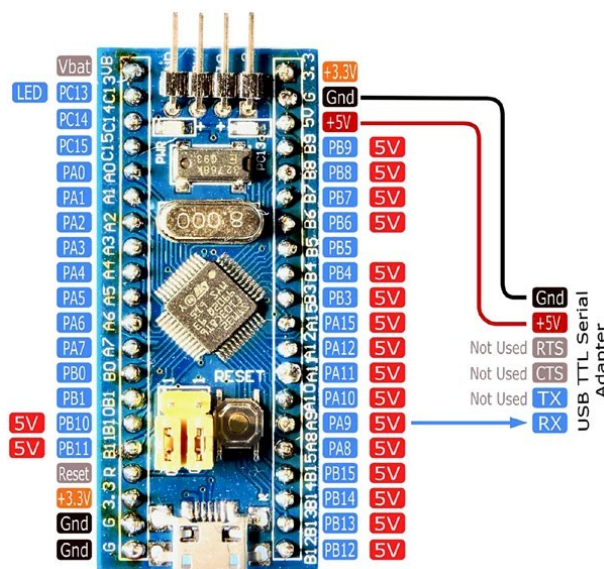


Figura 2 – Conexão do Adaptador com a Blue Pill

A taxa de transmissão que usaremos é 38.400 bits por segundo, que é uma velocidade relativamente baixa. Isso nos permite evitar o controle de fluxo, para simplicidade nossa inicialização.

A Blue Pill será alimentada a partir da linha de +5 volts do adaptador serial, conforme ilustrado na figura 2. Ao receber +5 volts do adaptador o regulador Blue Pill forneça 3,3 volts ao MCU, portanto, não requer nenhuma outra alimentação.

Se isso não for possível, alimente a Blue Pill separadamente. Certifique-se de fazer uma conexão comum de terra, entre a fonte de alimentação, o MCU e o adaptador serial. Observe que apenas uma conexão de dados é necessária. Isso porque nosso programa apenas transmite e não recebe dados do seu Desktop/Notebook.

O Projeto UART

Esse projeto tem uma parte parecida como o programa blinkly2, que já vimos. Essa parte é a que o LED PC13, construído internamente na Blue Pill, pisca, usando FreeRTOS. Aqui o LED piscará a cada 200 milissegundos. Esse pisca-pisca não tem nada a ver com a UART, mas, representa que a aplicação estará ativa. Com relação a UART, a aplicação enviará dados seriais para um programa no Desktop/Notebook. Esse programa receberá os dados seriais e mostrará no monitor do seu Desktop/Notebook.

Mude para o seguinte diretório:

```
~$ cd ~/stm32f103c8t6/rtos/uart
```

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. A figura 2 ilustra as conexões.

Com a energia aplicada no adaptador serial, a Blue Pill será alimentada, então, o LED de PWR acende e o LED PC13 pisca. Eu usarei o programa terminal minicom, mas, outro bom programa é o putty. Use o gerenciador de pacotes do seu sistema para instalar um deles, se necessário.

```
~$ sudo apt install minicom
```

Para configurar o seu adaptador serial, você precisará conhecer o nome do caminho do dispositivo específico do sistema operacional ou a porta COM. Para Linux, você poderá descobri-lo apenas

olhando para o diretório `/dev`". Quando você o desconecta, o nome desse dispositivo desaparece, então, poderia identificá-lo observando as mudanças.

Usando o `minicom`, primeiramente vamos configurar a porta de comunicação, fornecendo a opção `"-s"`, conforme mostrado na figura 3.

```
~$ minicom -s
```

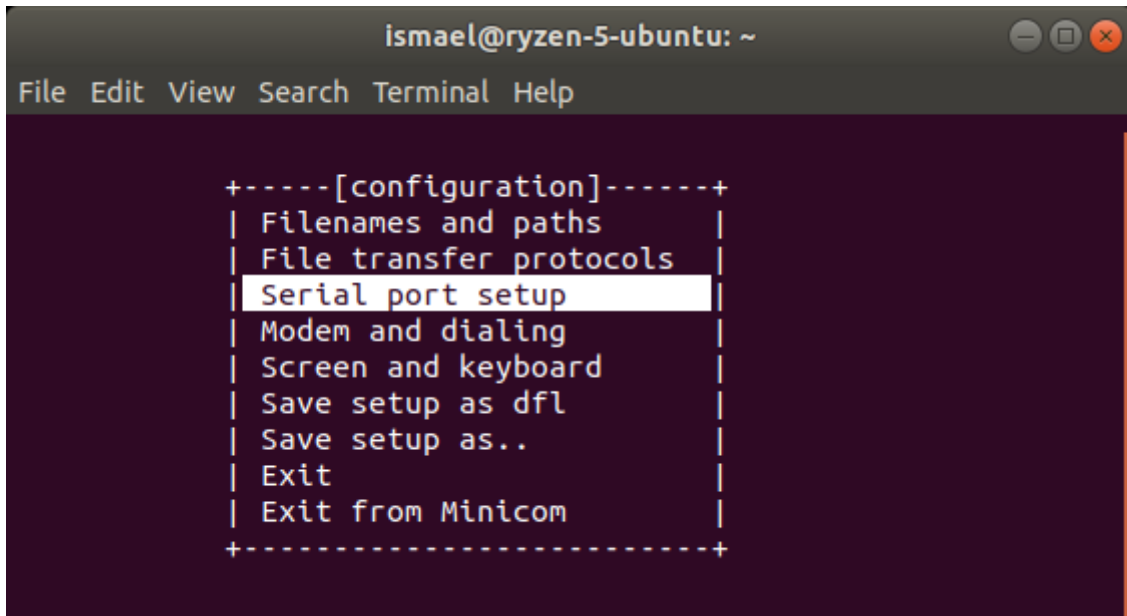


Figura 3 – Configurando a porta serial do minicom

Selecione a opção "Serial port Setup" e pressione Enter

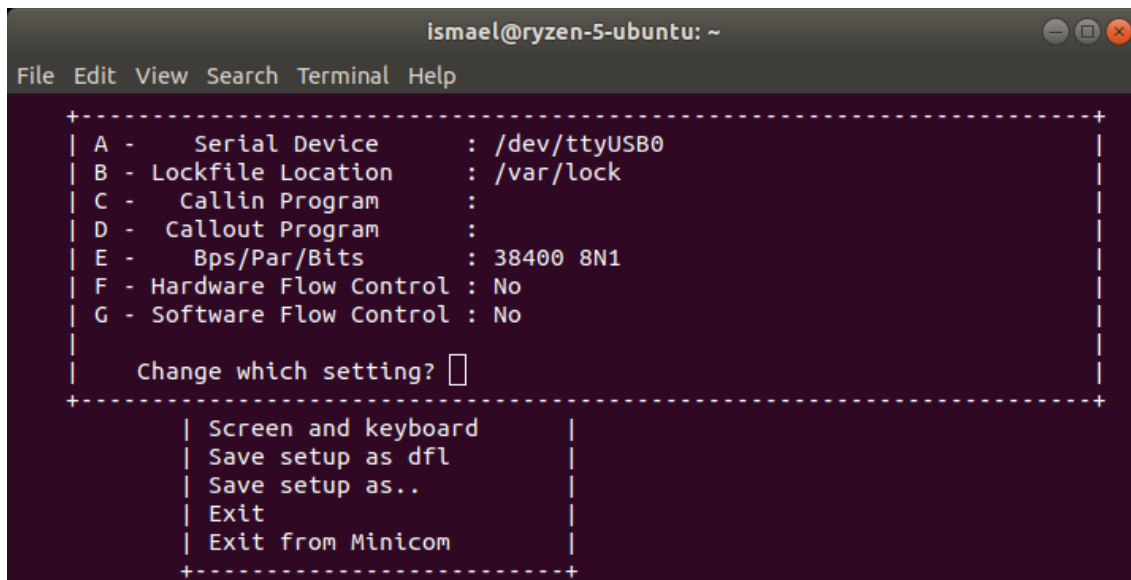


Figura 4 – Ajustando os parâmetros da porta serial do minicom

O nome do meu adaptador no Linux é `"ttyUSB0"`. Edite para que a configuração atual fique `"Current: 38400 8N1"`, conforme a figura 4.

Depois, no menu principal, escolha `"Salve setup as ..."` para salvar suas configurações. Usaremos `"bluep"` para o nome do arquivo de configuração, e pressione `"Enter"`. Se você tiver dificuldade

para salvar, é provável que o minicom tenha definido o diretório em um local em que você não tem permissão. Nesse caso, eu usei o usuário root para facilitar as coisas.

No meu caso o arquivo de configuração "minicom.bluep", foi salvo na pasta "/etc/minicom". Usei o comando cat para ver o conteúdo desse arquivo, e confirmei que a configuração está correta.

```
# cat /etc/minicom/minirc.bluep
```

```
# Machine-generated file - use "minicom -s" to change parameters.
```

```
pu port      /dev/ttyUSB0
pu baudrate  38400
pu bits      8
pu parity    N
pu stopbits  1
pu rtscts    No
```

O programa escreverá lentamente linhas repetidas de caracteres, com um determinado tempo entre cada caractere. Ser lento assim evita a necessidade de controle de fluxo. Para iniciar o minicom, agora poderá usar as configurações salvas da seguinte maneira:

```
# minicom bluep
```

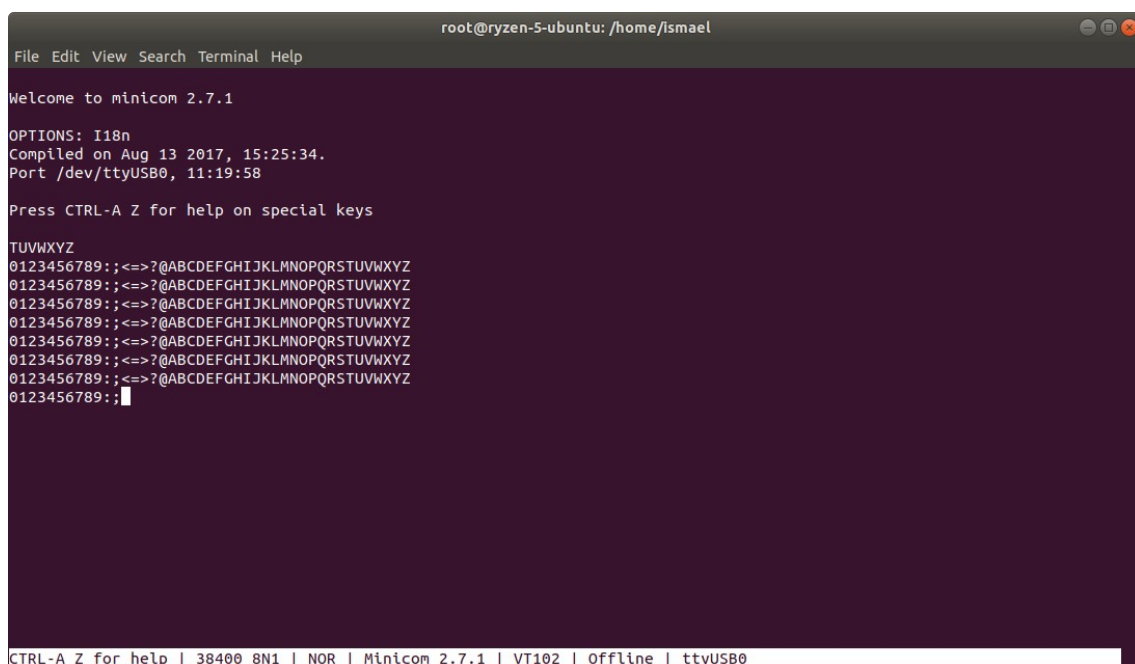


Figura 5 – Programa minicom recebendo os dados da Blue Pill

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

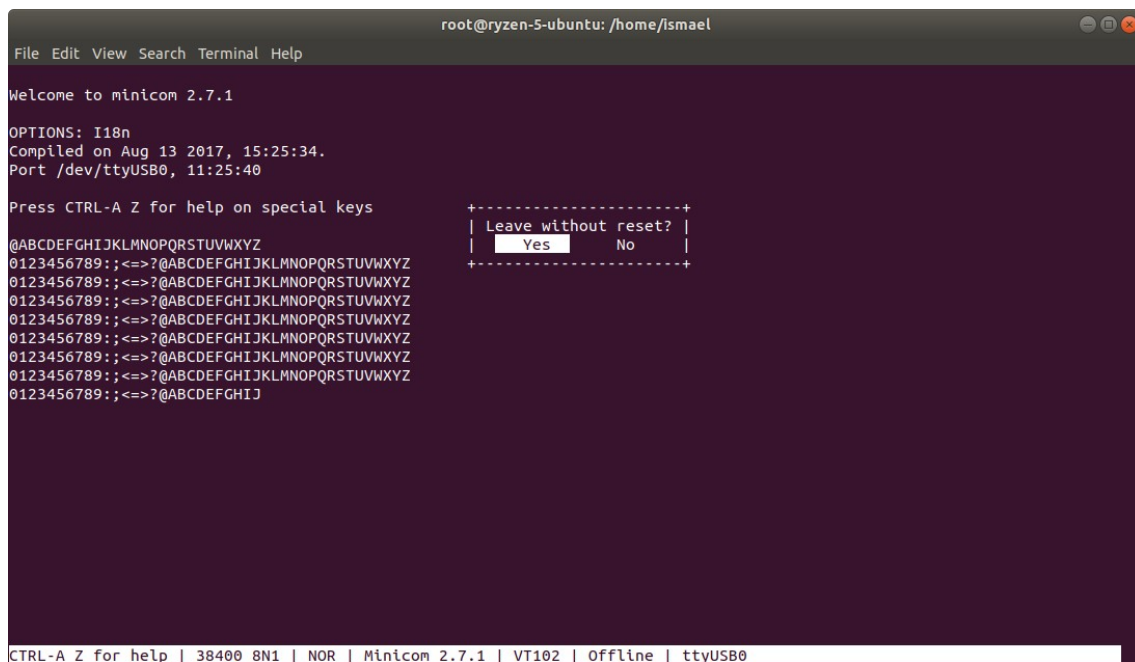


Figura 6 – Saindo do programa minicom

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

A seguir temos o programa que envia dados através da porta serial. O arquivo-fonte pode ver encontrado em `"/stm32f103c8t6/rtos/uart/uart.c"`:

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

// 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);
}

// Enviar um caracter para a UART
static inline void uart_putc (char ch)
{
    usart_send_blocking (USART1,ch);
}

// Enviar caracteres lentamente para a UART
static void task1 (void *args __attribute__((unused)))
{
    int c = '0' - 1;

    for (;;)
    {
        // Pisca o LED PC13
        gpio_toggle (GPIOC,GPIO13);
        // Tarefa controlada pelo agendador do FreeRTOS
        vTaskDelay (pdMS_TO_TICKS(200));
        // Controle de fluxo do programa
        if ( ++c >= 'Z' )
            // Envia o último carácter e os caracteres de controle pela USART
            {
                uart_putc (c);
                uart_putc ('\r');
                uart_putc ('\n');
                c = '0' - 1;
            }
        eles
        // Envia carácter pela USART
        {
            uart_putc (c);
        }
    }
}

// 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 ();
    // Criar a única tarefa para o FreeRTOS
    xTaskCreate (task1,"task1",100,NULL,configMAX_PRIORITIES-1,NULL);
    // Ativa o agendador de tarefa do FreeRTOS
    vTaskStartScheduler ();

    for (;;)
    return 0;
}

```

No início do programa temos comentários e as declarações dos arquivos de cabeçalhos, essenciais para que todos os recursos dessa aplicação FreeRTOS funcione corretamente.

A função static void uart_setup (void)

Essa função configura a UART1. Observe que dois sistemas de clock estão ativados. A saída TX do UART1 por padrão utiliza saída PA9, portanto, a porta GPIOA precisa ter seu clock ativado. O periférico USART1 também precisa de um clock.

Usamos a função **gpio_set_mode()** para configurar essa função alternativa de saída. A opção de maior velocidade foi escolhida, **GPIO_MODE_OUTPUT_50_MHZ**, para permitir um sinal mais nítido. Observe especialmente que a macro **GPIO_CNF_OUTPUT_ALTFN_PUSHPULL** especifica que é uma função alternativa (ALTFN) e que a saída deveria usar uma configuração push/pull. O aspecto de função alternativa é supercrítico aqui. Um erro comum é nesse momento da configuração escolher o formato padrão para GPIO, e não como uma função alternativa.

A macro **GPIO_USART1_TX**, da biblioteca **libopenm3**, na plataforma **STM32F103** equivale ao pino PA9. O uso da macro **GPIO9** teria sido igualmente válido, embora o código seja mais portátil, conforme apresentado.

Para configurar a taxa de transmissão da USART1, usamos a função **usart_set_baudrate** (USART1,38400). Essa função calcula um divisor necessário para chegar ao valor aproximado da taxa de transmissão. As taxas de transmissão de valor ímpar podem não ter a precisão que as taxas de transmissão padrão possuem.

Para configurar quantos bits cada dado conterá, usamos a função **usart_set_databits** (USART1,8). As opções válidas são 8 ou 9. Com a paridade ativada, isso implica 7 ou 8 bits de dados, respectivamente.

Para configurar um bit de parada, usamos a função **usart_set_stopbits** (USART1,USART_STOPBITS_1). Para o periférico somente transmitir, usamos a função **usart_set_mode** (USART1,USART_MODE_TX). Para que nenhum bit de paridade seja enviado,

usamos a função **usart_set_parity** (USART1,USART_PARITY_NONE), e para indicar que nenhum controle de fluxo de hardware será usado, usamos a função **usart_set_flow_control** (USART1,USART_FLOWCONTROL_NONE). Finalmente, para habilitar o periférico USART1, usamos a função **usart_enable** (USART1).

Como pode ser visto, existem vários detalhes na configuração da UART, mas, essas configurações devem corresponder ao programa de terminal no seu Desktop/Notebook, portanto, também requer configuração, como já mostramos.

A função static inline void uart_putc (char ch)

A função **uart_putc()**, simplesmente chama a uma função da biblioteca libopenm3 denominada **usart_send_blocking()**, que enviará um carácter pela USART. Conforme implícito no nome da função, o controle não retorna até que o USART esteja pronto para aceitar mais dados.

A função int main (void)

Como já comentei o programa "uart.c" também tem o pisca-pisca, então, o que é novo na função **main()** é a chamada para a função **uart_setup()**. Não tem muito que explicar, porque já foi explicado no projeto anterior (ver o nono artigo da série).

A função static void task1 (void *args __attribute__((unused)))

A função **task1()** foi projetada para piscar o LED PC13 e colocar linhas de texto no seguinte formato:

```
0123456789; <=>? @ ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Como parte do loop, dentro da função **task1()**, o LED PC13 alterna seu estado, piscando a cada 200ms. Isso mostra que o programa está sendo executado. Também a **task1()** envia um caractere a cada 200ms. Isso evita que tenhamos que lidar com o controle de fluxo por enquanto.

A tarefa **vTaskDelay** (pdMS_TO_TICKS(200)), é a única tarefa controlada pelo agendador do FreeRTOS, então, gera um atraso preciso para que a aplicação funcione conforme desejamos.

Cada carácter é transmitido chamando a função **uart_putc()**. O argumento dessa função é o carácter a ser enviado pela USART. A função **uart_putc()** está dentro do um controle de fluxo de programa, através de uma instrução "if", então, inicia com o carácter "0", incrementa e envia o carácter até chegar no carácter "Z". Quando atingir o carácter "Z", a instrução "if" desvia o fluxo do programa, envia o carácter "Z", envia um carácter de controle de retorno do cursor e envia um carácter de controle de nova linha. Depois inicializa novamente para o carácter "0" e fica nesse ciclo.

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