

Esse artigo foi escrito pelo engenheiro Ismael Lopes da Silva, exclusivamente para o site "[www.embarcados.com.br](http://www.embarcados.com.br)". O link para o artigo é "<https://www.embarcados.com.br/blue-pill-api-usart/>".

No décimo terceiro 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 veremos a API USART.

## API USART

Para referência, vamos listar a API (Interface de Programação de Aplicação - Application Programming Interface) usada nessa série e documentar seus argumentos. Além disso, funções avançadas da API serão incluídas para manter a referência agrupada.

Para as funções que serão listadas, alguns argumentos precisam de valores especiais, fornecidos pelas definições de macros na biblioteca libopenm32. A tabela a seguir lista as diferentes USARTs disponíveis para o MCU STM32F103C8T6. Os pinos padrões são listados para cada função.

USART	MACRO	5V	TX	RX	CTS	RTS
1	USART1	Yes	PA9	PA10	PA11	PA12
2	USART2	No	PA2	PA3	PA0	PA1
3	USART3	Yes	PB10	PB11	PB14	PB12

**Tabela 1** – USARTs disponíveis no STM32F103C8T6 (Argumento USART)

MACRO	DESCRIÇÃO
USART_PARITY_NONE	Sem paridade
USART_PARITY_EVEN	Paridade par
USART_PARITY_ODD	Paridade ímpar
USART_PARITY_MASK	Máscara

**Tabela 2** – Macros de paridade da USART (Argumento Paridade)

MACRO	DESCRIÇÃO
USART_MODE_RX	Somente recebe
USART_MODE_TX	Somente transmite
USART_MODE_TX_RX	Transmite e recebe
USART_MODE_MASK	Máscara

**Tabela 3** – Macros de modos de operação da USART (Argumento Modo)

MACRO	DESCRIÇÃO
USART_STOPBITS_0_5	0.5 bit de parada

USART_STOPBITS_1	1 bit de parada
USART_STOPBITS_1_5	1.5 bits de parada
USART_STOPBITS_2	2 bits de parada

**Tabela 4** – Macros de Bits de Parada da USART (Argumento Stopbits)

MACRO	DESCRIÇÃO
USART_FLOWCONTROL_NONE	Sem controle de fluxo por hardware
USART_FLOWCONTROL_RTS	Controle de fluxo por hardware RTS
USART_FLOWCONTROL_CTS	Controle de fluxo por hardware CTS
USART_FLOWCONTROL_RTS_CTS	Controle de fluxo por hardware RTS e CTS
USART_FLOWCONTROL_MASK	Máscara

**Tabela 5** – Macros de Controle de Fluxo da USART

VALOR	BITS DADOS (Sem paridade)	BITS DADOS (Com paridade)
8	8	7
9	9	8

**Tabela 6** – Bits de Dados para USART (Argumentos Bits)

MACRO	DESCRIÇÃO DO FLAG
USART_SR_CTS	Flag Limpa para enviar
USART_SR_LBD	Flag de LIN break-detection
USART_SR_TXE	Buffer de transmissão de dados está vazio
USART_SR_TC	Transmissão completada
USART_SR_RXNE	Registro de leitura de dados não está vazio
USART_SR_IDLE	Linha em IDLE foi detectada
USART_SR_ORE	Erro de Overrun
USART_SR_NE	Flag erro de ruído
USART_SR_FE	Erro de framing
USART_SR_PE	Erro de paridade

**Tabela 6** – Macros dos Bits de Flag de Status da USART (Argumento Flag)

## Arquivos de Cabeçalho

```
#include <libopencm3/stm32/rcc.h>
```

```
#include <libopencm3/stm32/usart.h>
```

## Clocks

```
rcc_periph_clock_enable (RCC_GPIOx);
```

```
rcc_periph_clock_enable (RCC_USARTn);
```

### **Configuração**

```
void usart_set_mode (uint32_t usart, uint32_t mode);  
void usart_set_baudrate (uint32_t usart, uint32_t baud);  
void usart_set_databits (uint32_t usart, uint32_t bits);  
void usart_set_stopbits (uint32_t usart, uint32_t stopbits);  
void usart_set_parity (uint32_t usart, uint32_t parity);  
void usart_set_flow_control (uint32_t usart, uint32_t flowcontrol);  
void usart_enable (uint32_t usart);  
void usart_disable (uint32_t usart);
```

### **DMA**

```
void usart_enable_rx_dma (uint32_t usart);  
void usart_disable_rx_dma (uint32_t usart);  
void usart_enable_tx_dma (uint32_t usart);  
void usart_disable_tx_dma (uint32_t usart);
```

### **Interrupções**

```
void usart_enable_rx_interrupt (uint32_t usart);  
void usart_disable_rx_interrupt (uint32_t usart);  
void usart_enable_tx_interrupt (uint32_t usart);  
void usart_disable_tx_interrupt (uint32_t usart);  
void usart_enable_error_interrupt (uint32_t usart);  
void usart_disable_error_interrupt (uint32_t usart);
```

### **Entrada/Saída/Status**

```
uint16_t usart_recv (uint32_t usart)  
void usart_send (uint32_t usart, uint16_t data)  
bool usart_get_flag (uint32_t usart, uint32_t flag)
```

### **Alinhamento de configuração para USART**

Com exceção das interrupções e do DMA, a seguir é apresentado um resumo dos alinhamentos que devem ser realizados para tornar o seu periférico UART funcional:

- 1) Habilite o clock da GPIO apropriada: **rcc\_periph\_clock\_enable** (RCC\_GPIOx).

- 2) Habilite o clock para o periférico UART selecionado: **rcc\_periph\_clock\_enable** (RCC\_USARTn).
- 3) Configure o modo dos seus pinos de E/S com **gpio\_set\_mode** ().
  - a) Para pinos de saída, no terceiro argumento escolha GPIO\_CNF\_OUTPUT\_ALTFN\_PUSHPULL.
  - b) Para entradas, escolha GPIO\_CNF\_INPUT\_PULL\_UPDOWN ou GPIO\_CNF\_INPUT\_FLOAT.
- 4) `uart_set_baudrate()`
- 5) `uart_set_databits()`
- 6) `uart_set_stopbits()`
- 7) `uart_set_mode()`
- 8) `uart_set_parity()`
- 9) `uart_set_flow_control()`
- 10) `uart_enable()`

## FreeRTOS

Até aqui utilizamos algumas funções da API do FreeRTOS, então, vamos resumi-las por conveniência. A seguir, são apresentadas as funções do FreeRTOS relacionadas a tarefas que usamos para criar tarefas, iniciar o planejador e atrasar execução, respectivamente:

### **BaseType\_t xTaskCreate** (

```

    TaskFunction_t pvTaskCode,    // function ptr
    const char * const pcName,    // string name
    unsigned short usStackDepth,  // stack size in words
    void *pvParameters,          // Pointer to argument
    uBaseType_t uxPriority,        // Task priority
    TaskHandle_t *pxCreatedTask   // NULL or pointer to task handle
);                                // Returns: pdPass or
                                // errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY

```

**void vTaskStartScheduler** (void); // Start the task scheduler

**void vTaskDelay** (TickType\_t xTicksToDelay);

**void taskYIELD** ();

O valor do ponteiro `pvTaskCode` é simplesmente um ponteiro para uma função da seguinte forma:

**void minha\_task** (void \* args)

O valor fornecido para "args" vem de pvParameters, na chamada da função **xTaskCreate()**. Se não for necessário, esse valor pode ser fornecido com NULL. A profundidade do stack está em words (4 bytes cada).

Cada tarefa tem uma prioridade associada, e é fornecida pelo argumento uxPriority. Se você estiver executando todas as tarefas com a mesma prioridade, forneça o valor configMAX\_PRIORITIES-1 ou use apenas o valor 1. A menos que você precise de prioridades diferentes, configure-as para o mesmo valor. Quando necessário, você pode criar mais tarefas após a chamada do **vTaskStartScheduler()**.

Uma macro que é muito comum usar para **vTaskDelay()** é mostrada a seguir. Isso converte um milissegundo de tempo em uma contagem de ticks para conveniência da programação.

**pdMS\_TO\_TICKS** (ms)      // Macro: converte ms em ticks

## Queues

As funções API sobre filas (queue) usadas foram:

### **QueueHandle\_t xQueueCreate**

```
(
    UBaseType_t uxQueueLength,    // Max # of items
    UBaseType_t uxItemSize        // Item size (bytes)
);                                // Returns: handle else NULL
```

### **BaseType\_t xQueueSend**

```
(
    QueueHandle_t xQueue,          // Queue handle
    const void *pvItemToQueue,    // pointer to item
    TickType_t xTicksToWait       // 0, ticks or portMAX_DELAY
);                                // Returns: pdPASS or errQUEUE_FULL
```

### **BaseType\_t xQueueReceive**

```
(
    QueueHandle_t xQueue,          // Queue handle
```

```
void *pvBuffer,           // Pointer to receiving item buffer
TickType_t xTicksToWait   // 0, ticks or portMAX_DELAY
);                          // Returns: pdPASS or errQUEUE_EMPTY
```

A função **xQueueCreate()** aloca armazenamento para a fila criada, e seu manuseador é retornado. O argumento `uxQueueLength` indica o número máximo de itens que podem ser mantidos na fila. O valor `uxItemSize` especifica o tamanho de cada item.

A função **xQueueSend()** adiciona um item à fila. O item apontado por `pvlItemToQueue` é copiado no armazenamento da fila. Por outro lado, **xQueueReceive()** pega um item da fila e o copia para o armazenamento do chamador no endereço `pvBuffer`. Esse buffer deve ter, pelo menos, o tamanho fornecido pelo `uxItemSize` ou ocorrerá corrupção de memória. Se não houver nenhum item a ser recebido, a chamada será bloqueada de acordo com `xTicksToWait`.

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