Curso de Programação para Microcontroladores STM32

PREDEBON

BECKER

Sumário

Lista de Figuras

Lista de Acrônimos e Abreviaturas

ADC	*Analog to Digital Converter*
ARM	~~Braço~~ Advanced RISC Machine (anteriormente Acorn RISC Machine)
ARR	*Auto Reload Register* (registrador que guarda o período de *timer*)
BSRR	*Bit Set/Reset Register* (registrador usado para *setar* e *resetar* portas GPIO)
BRR	*Bit Reset Register* (registrador usado para *resetar* portas)
CDC	*Communication Device Class*
COM	*Virtual Serial Communication Port*
CC	*Capture/Compare* (designador comum para funcionalidades de *input capture* ou *output compare* de um *timer*)
DAC	*Digital to Analog Converter*
DFU	*Device Firmware Update* (parte do protocolo USB, possibilita a atualização de *firmwares* de dispositivos diretamente pelo USB)
DMA	*Direct Memory Access*
DR	*Data Register*
EXTI	*Extended Interrupts and Events Controller*
GUI	*Graphical User Interface*
GPIO	*General-purpose Input/Output* (entrada/saída de uso genérico)
HAL	*Hardware Abstraction Layer* (camada de abstração de hardware)
HID	*Human Interface Device* (parte do protocolo USB que estabelece um *framework* para mouses, teclados, etc.)
JRE	*Java Runtime Environment*
IDE	*Integrated Developtment Environment*
IDR	*Input Data Register* (registrador que guarda o estado das portas GPIO em modo entrada)
IRQ	*Interrupt Request*
ISR	*Interrupt Service Routine*
LED	*Light-emitting Diode*
LL	*Low Level* (baixo nível)
MCU	~~*Marvel Cinematic Universe*~~ *Microcontroller Unit*
NVIC	*Nested, Vectored Interrupts Controller* (gerenciador de interrupções da arquitetura ARM)
OCxPE	*Output Compare Preload Enable* do canal 'x'. Registrador de *timer*
ODR	*Output Data Register* (registrador que guarda o estado das portas GPIO em modo saída)
PLL	*Phase-Locked Loop* (multiplicador de frequência)
PWM	*Pulse Width Modulation*
UART	*Universal Asynchronous Receiver/Transmitter*
USART	*Universal Synchronous/Asynchronous Receiver/Transmitter*
USB	*Universal Serial Bus* (barramento serial universal)
STM	ST Microelectronics

1 Introdução

Bem-vindo ao nosso curso de programação ~~braço~~ ARM! Estamos muito felizes de ter-lo conosco, sinceramente. O processo de preparação deste curso foi um caminho cheio de surpresas (boas e ruins), onde nós trabalhamos e aprendemos bastante. O objetivo do curso é disseminar o uso das novidades (não tão novas assim) no mundo de microcontroladores, apresentando o que há de melhor. A ST vem ganhando mais e mais espaço no mercado devido à qualidade e completude notável de seus microcontroladores, principalmente os de 32 bits que utilizam *core* ARM.

Este é um curso introdutório, onde cobriremos como trabalhar com os principais periféricos, mas que não nos impede de ir um pouco além e mostrar do que o MCU é capaz usando funcionalidades mais complexas. Com já diziam os romanos, "longo é o caminho por preceitos, breve e eficaz por exemplos" (*Longum iter est per praecepta, breve et efficax per exempla*). Assim, buscou-se um *approach* mais *hands on*, explicando os conceitos com exemplos de código.

1.1 Afinal, o que é um ARM?!

ARM Ltd. é uma empresa que produz propriedade intelectual de núcleos de processadores. Ela começou como um projeto da Acorn Computers, com seus engenheiros infelizes com os processadores existentes no mercado e se perguntando por que os fabricantes de *chips* não faziam circuitos integrados digitais de um jeito que parecia mais sensato para eles - com a menor contagem de portas lógicas possível, o que faria os circuitos mais rápidos, mais energeticamente eficientes, e com menor área de silício ocupada. Eles perseguiram essa filosofia de *design*, sempre procurando por um "*gotcha!*", mas nunca encontrando nenhum. Assim nasceram os processadores ARM. Após alguns anos, nasceu a ARM Ltd e as arquiteturas que conhecemos hoje.

Hoje em dia, fala-se em perfis de núcleos Cortex, que são famílias de processadores ARM projetados para diferentes finalidades. Eles são:

- **A**pplication - Usado em dispositivos como celulares e tabblets. São processadores projetados para performance, são vendidos combinados com periféricos de alto nível, como Ethernet, decodificador MPEG, etc. Possui *set* de instruções grande, com instruções de DSP. Geralmente rodam algum OS, geralmente Linux.
- **R**eal-time - Usados em aplicações críticas e de tempo real, como controles industriais ou automotivos.
- **M**icrocontroller - Processadores mais *mainstream*, que "dão um pau" em MCUs convencionais. São feitos pensando em baixa latência de interrupções (NVIC), baixo consumo, e fácil integração em FPGAs.

Nosso STM32F042 é um Cortex-M0, da série *Mainstream* da STM. Um pouco mais sobre ele a seguir.

1.2 STM32F042F6

O microcontrolador de escolha é da linha USB da STM. Ele possui oscilador interno de precisão de 48 MHz, casamento de impedância e *pull-ups* internos nos pinos do USB, tudo o necessário para usar USB em *full speed* sem necessidade de componentes externos. Possui um ADC de 1 Msps de 12 bits (configurável para 10, 8 e 6 bits), multiplexado para 10 canais externos, mais 3 internos (sensor de temperatura integrado, medidor de tensão de bateria, e tensão interna de referência). Possui um módulo de comunicação I^2C, duas USART, duas SPI, e, claro, um USB (com unidade de cálculo CRC). Também possui vasta opção de osciladores internos, incluindo multiplicador PLL.

É bastante coisa para um microcontrolador do seu tamanho. Isso é conseguido graças ao pequeno *footprint* do núcleo Cortex-M0 e à ausência de EEPROM, que é substituída por flash, e um emulador de EEPROM, gerenciador de memória parecido com os utilizados em *mass storage devices* como cartões SD e *pen drives*. A escolha de periféricos, seu tamanho e custo, o fazem ideal para aplicações de instrumentação, sensoreamento, comunicação, dispositivo USB, ou mesmo conversão de USB para outros protocolos (RS232, por exemplo), etc. Mouses, teclados (inclusive de toque capacitivo) e amplificadores USB veem à mente.

Ele também possui *bootloader* DFU, que usaremos durante o curso. DFU é parte do protocolo USB e permite a atualização do *firmware* de dispositivos diretamente, sem necessidade de conversão para outros protocolos como RS232 ou JTAG, eliminando conversores intermediários.

2 Instalação

2.1 Windows

Para podermos programar nosso STM32, precisamos instalar alguns softwares. Para o *download*, será necessário preencher um cadastro no site da ST. Não tem muito segredo, é só fazer o que eles pedem.

- JRE: O programador roda em java.

  `http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html`

- STM32CubeIDE - IDE integrada da STM, ela junta o CubeMX com o TrueStudio, assim fazendo toda a cadeia de geração de código até debug.

  `https://www.st.com/en/development-tools/stm32cubeide.html`

- Gravador STM32CubeProgrammer, utilizaremos ele para gravar o microcontrolador, ja que estamos usando *bootloader*.

  `https://www.st.com/en/development-tools/stm32cubeprog.html`

  A instalação também não tem segredos, e é só ir clicando em *Next*. Só não esqueça de instalar o Java antes dos Cubes.

  Para programas que usem Porta Serial Virtual (Porta COM), talvez seja necessário instalar um *driver* para o *MCU*:

- VCP V1.4.0 - *Driver* de porta COM

  `http://www.st.com/en/development-tools/stsw-stm32102.html#getsoftware-scroll`

Pronto agora você tem as ferramentas que precisamos para nosso curso. A seção 3 mostra como utilizá-las para compor ~~seu primeiro *single*~~ seu primeiro programa.

2.2 GNU/Linux

O processo de instalação das ferramentas é semelhante ao do Windows.

Para instalar o Java é simples, apenas rodar o comando via apt:

- *sudo apt-get install default-jre*

Os softwares podem ser baixados nos mesmos link da seção do Windows.

O Linux ja interpreta de maneira correta o *driver* da porta ACM, então não é necessário o download de *drivers* adicionais.

3 Construindo seu Primeiro Programa

Vamos criar um programa bem simples, piscar os LEDs da placa de desenvolvimento utilizando as ferramentas instaladas.

3.1 Gerando o Código Base

Para começar, abra o CubeIDE e crie um novo projeto de STM32. No campo de busca informe o microcontrolador STM32F042F6.

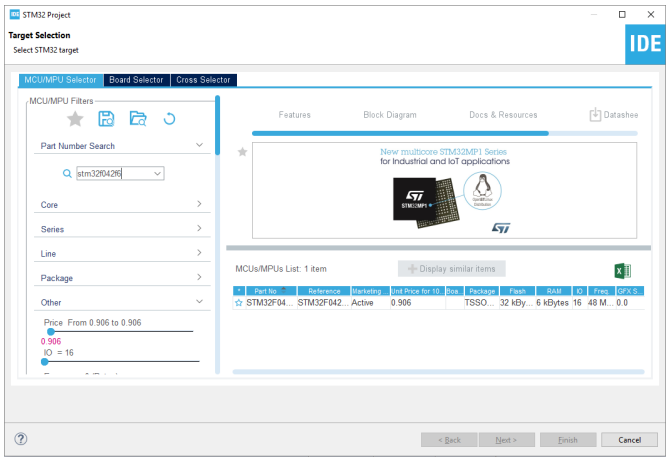

Figura 1: Página de busca do Cube

Assim que você escolher o microcontrolador, selecione que a linguagem será em C e crie o projeto.

Uma janela com a pinagem do microcontrolador deve aparecer, clique no PA4 e PA5 e os configure para GPIO_Output, assim como PA6 e PB8 como GPIO_Input. Com a exceção do USB, essa é a configuração cobre todas as funções do *kit* de desenvolvimento.

Vale ressaltar que configurar PB8 como entrada é só uma proteção adicional que estamos dando ao microcontrolador, este pino é usado durante o *boot* para entrar em modo de *DFU* ou iniciar normalmente o programa.

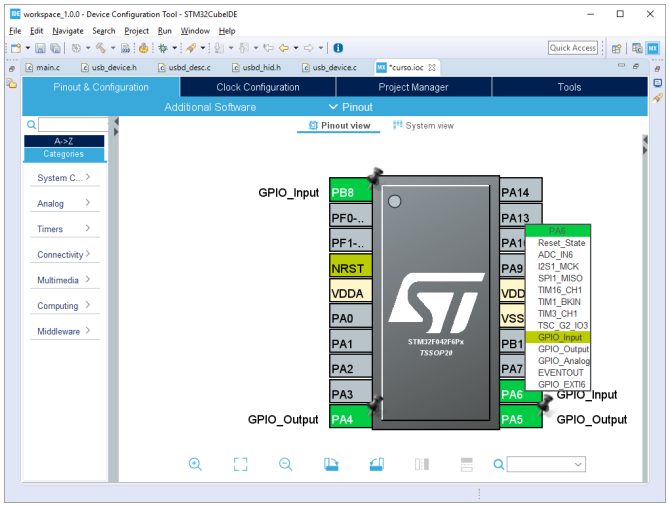

Figura 2: Pinos selecionados no Cube

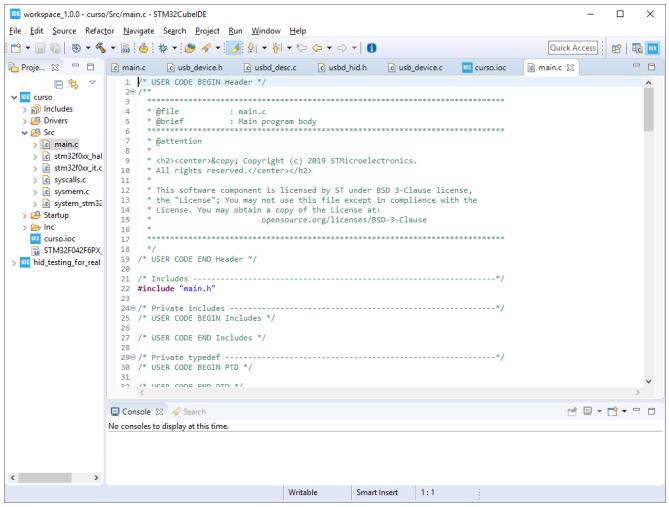

Figura 4: Main aberta

A partir desse ponto você já tem tudo necessário para gerar o programa base para a nossa aplicação, mas por padrão o Cube não vai levar em conta o PLL do clock, iniciando com a frequência do RC calibrado HSI, neste caso é igual a 8MHz. Para aumentar a frequência, devemos selecionar o PLLCLK no *Mux* do System Clk, na aba *Clock Configuration*, e aumentar o valor da multiplicação do PLL dentro do retângulo azul, a frequência máxima para microcontroladores STM32F0 costuma ser de 48MHz.

Como o STM32F042F6 é da linha de microcontroladores USB-F0, temos a opção de selecionar o HSI48 RC (um oscilador interno calibrado em 48 MHz que poderá ser utilizado com o periférico de USB).

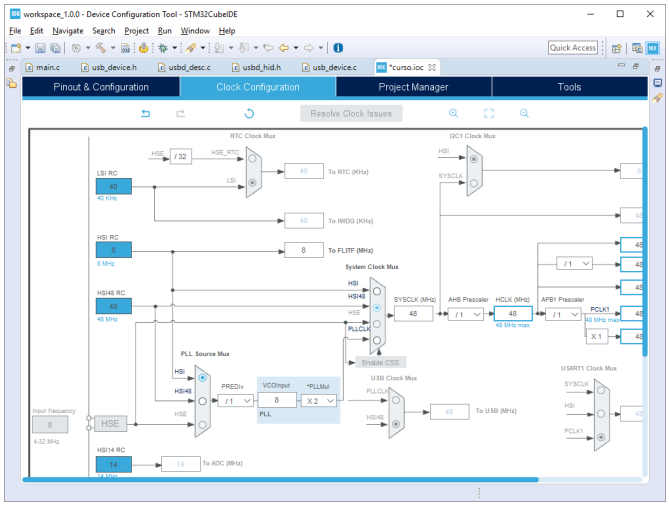

Figura 3: Arvore de Clock

Salvando o projeto, a IDE vai te perguntar se você quer gerar o código, aceite.

Na barra à esquerda, temos a navegação do projeto, vamos abrir a main.c.

3.2 Edição Simples do Código

Vale alertar que tudo que estiver escrito entre USER CODE BEGIN # e USER CODE END #, será mantido se houver a re-geração do código no *Cube*, uma boa prática seria manter tudo o que for gerado pelo usuário dentro dessas marcações.

No *User Code 3*, vamos alternar o estado da *GPIO* A4 a cada um segundo utilizando a *HAL*:

```
/* USER CODE BEGIN 3 */

  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4, 0);
  HAL_Delay(1000);
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4, 1);
  HAL_Delay(1000);
}
/* USER CODE END 3 */
```

Compile o código (ctrl-B).

3.3 Ritual de Programação do Microcontrolador

Muito bem, agora vamos gravar nosso primeiro programa. Para permitir que o microcontrolador seja gravado, precisamos entrar em DFU (Direct Firmware Update), uma espécie de bootloader USB. Para isso, devemos colocar a chave do canto superior esquerdo da placa para cima (conectando o B8 e 3V3), em seguida resetar o microcontrolador pressionando o botão *rst*, na parte inferior esquerda da placa.

Abra o *CubeProgrammer* e na seleção da forma de gravação escolha USB (também é possível utilizar o software com *bootloader UART* ou gravador *SWDIO*), conecte ao microcontrolador. Assim que conectado o mapa

de memória na região da memória de programa será mostrado, caso o microcontrolador nunca tenha sido gravado, todos os endereços estarão preenchidos por *0xFF*.

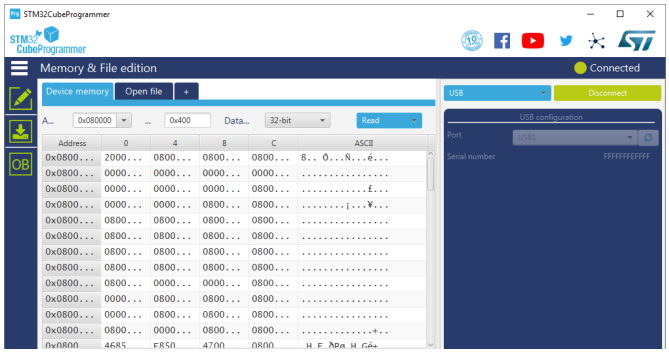

Figura 5: Janela do STM32CubeProg

Na esquerda temos 3 opções neste software, ver o mapa de memória, gravar um hexadecimal ou alterar os *option bytes* (OB), para gravar o hexadecimal, utilizaremos a opção intermediária.

No botão *Browse*, devemos buscar pelo hexadecimal gerado (por padrão, a IDE gera apenas o .elf, pode ser este arquivo ou o .bin), assim que encontrado podemos pressionar *Start Programming*, caso a operação seja bem sucedida, teremos uma mensagem pop-up contendo "File Download Complete". Para testar o programa, é necessário retornar a chave à posição inferior e resetar a placa (através do botão esquerdo).

3.4 Substituindo HAL por LL

Seguindo a função GPIO_WritePin() (Go to Definition), podemos ver que os registradores que a HAL usa para impor um valor lógico nessa IO são BSRR e BRR. Seguindo a definição assim como fizemos para a função de *write*, podemos ler as descrições dos registradores do ponteiro de estrutura *GPIOx*.

Podemos ver que o ODR é responsável pelos dados de saída da GPIO, então retomando o código podemos reduzir o número de instruções executadas na função *write* a substituindo-a por um ponteiro para o registrador.

Outra indeterminação é o diabos seria GPIO_PIN_4? Se o seguirmos podemos ver que ele está definido como um inteiro de 16 bits sem sinal com valor de 10 Hexadecimal. Nota-se que isso é equivalente a (1«4), assim percebemos que a GPIO trata seus bits como qualquer microcontrolador.

```
/* USER CODE BEGIN 3 */

  GPIOA->ODR^=(1<<4);
  HAL_Delay(1000);
}
/* USER CODE END 3 */
```

Para o outro LED, na porta PA5, podemos fazer o mesmo:

```
/* USER CODE BEGIN 3 */

  GPIOA->ODR^=0x30;
  HAL_Delay(1000);
}
/* USER CODE END 3 */
```

3.5 Adicionando Funcionalidades: Entrada

Usar GPIO como entrada pode ser feita de maneira similar, com o uso da função ReadPin. Iremos adicionar o botão da placa para acionar ou não os LEDs:

```
/* USER CODE BEGIN 3 */

  if(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_6)){
    GPIOA->ODR^=0x30;
    HAL_Delay(1000);
  }
  else
    GPIOA->ODR&=~0x30;
}
/* USER CODE END 3 */
```

Assim, com o botão pressionado, os LEDs devem permanecer desligados. De maneira similar ao procedimento anterior, podemos seguir a função ReadPin e achar o registrador IDR. Ficando assim:

```
/* USER CODE BEGIN 3 */

if(GPIOA->IDR & 0x40){
  GPIOA->ODR^=0x30;
  HAL_Delay(1000);
}
else
  GPIOA->ODR&=~0x30;

/* USER CODE END 3 */
```

E assim você tem um simples programa para piscar LEDs feito em LL e HAL.

4 *GPIO*

4.1 *Overview*

*GPIO*s (*General Purpose Input/Output*) são o coração de um microcontrolador. Um módulo *GPIO* completo possui 16 portas, mas usualmente, nem todas são roteadas para os pinos do *package*. Nosso STM32F042F6 possui três desses módulos: *GPIOA*, *GPIOB* e *GPIOF*.

O módulo *GPIOA* possui 14 portas (de PA0 a PA7, e de PA9 a PA14). Dois pinos são sobrecarregados com duas portas *GPIO*: pino 17 com as portas PA9 e PA11, e pino 18 com PA10 e PA12. Elas podem ser permutadas por *firmware* para disponibilizar no *pinout* diferentes funcionalidades. Por exemplo, precisamos das portas PA11 e 12 para USB, mas das PA9 e 10 para USART naqueles pinos.

O módulo *GPIOB* tem duas portas disponibilizadas, PB1 e PB8. PB8 divide o pino 1 com a funcionalidade BOOT0, usada para *bootar* (entrar em modo de gravação) por *hardware*. Portanto, pede-se cuidado ao utilizar a porta em projetos, pois pode-se impossibilitar a gravação do dispositivo com uso indevido.

O módulo *GPIOF* tem duas portas disponíveis, PF0 e PF1, nos pinos 2 e 3. Elas são as entradas para um oscilador externo opcional, e estão lá principalmente por causa disso. Também podem ser usadas como pinos I^2C, estando convenientemente em pinos adjacentes.

4.2 *Input/Output*

Começaremos a explorar a *GPIO* com um exemplo simples: acender os *LED*s da placa de desenvolvimento ao pressionarmos o botão.

Abra o inicializador de códigos e siga os passos para criação de projetos da seção 3, criando o projeto com um nome de preferência (nós escolhemos "button_leds") e com exatamente o mesmo pinout daquele projeto (isto é, PA4 e 5 como *output* e PA6 como *input*). Usaremos os dois *LED*s da placa, mais o botão, conectado à porta PA6. Este botão aterra o pino quando pressionado e um *pull-up* não está presente na placa pois há resistores de *pull-up/down* à disposição nos módulos *GPIO*. Vamos adicionar um *pull-up*:
Vá em System -> GPIO. Nesta janela, selecione PA6 na tabela de portas e, na área *PA6 Configuration* que aparece abaixo, selecione *Pull-up* na *drop down box* com rótulo *GPIO Pull-up/Pull-down*.
Feito isso, vá até *Project->Settings* na barra de menus, mude a *Toolchain/IDE* para MDK-ARM, e mande gerar o código, abrindo-o no μVision quando estiver pronto.

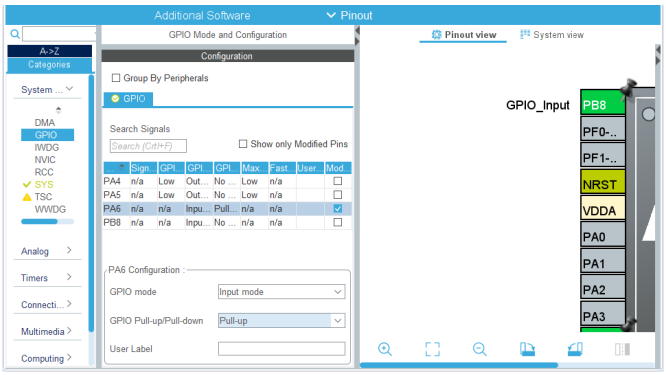

Figura 6: Alterando a configuração da *GPIOA*.

Voltando à *main*, podemos utilizar a função HAL de leitura da porta.

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
  // Caso o botao seja pressionado:
  if(!HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_6))
  {
    // Liga os LEDs
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4|GPIO_PIN_5,
          GPIO_PIN_SET);
  }
  else
  {
    // Desliga os LEDs
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4|GPIO_PIN_5,
          GPIO_PIN_RESET);
  }
/* USER CODE END WHILE */

/* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

Como mencionado anteriormente, é muito importante manter nosso código entre "/* USER CODE BEGIN WHILE */"e "/* USER CODE BEGIN WHILE */"pois assim ele não será sobrescrito caso geremos novamente configurações com o *Cube*. Baixe o código no microcontrolador como explicado na Seção 3 e dê *reset*. Agora, ao pressionar o botão da placa, os dois LEDs devem acender, e ficarem acesos até que você solte o botão.

Ainda não usamos a função HAL_GPIO_TogglePin, porém. Vamos fazer um breve teste. Para testarmos a função, substitua o código do *loop* pelo código abaixo:

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
```

```
3  while (1)
4  {
5    // Espera ateh que o botao ser pressionado:
6    while(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_6));
7
8    // Inverte os LEDs
9    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_4|GPIO_PIN_5);
10
11   // Delay para debounce
12   HAL_Delay(300);
13
14 /* USER CODE END WHILE */
15
16 /* USER CODE BEGIN 3 */
17 }
18 /* USER CODE END 3 */
```

Você deve observar, que cada vez que pressionar o botão, os *LED*s inverterão seu estado, acendendo e apagando. Assim, concluímos nosso exemplo.

4.2.1 Entendento a Configuração

Olhando o arquivo main.c, vemos que a função MX_GPIO_Init é chamada pouco antes do *loop* do programa. Essa é a função que inicializa a *GPIO* como a configuramos no *Cube*. Vá até a a definição da função. Sua implementação deve ser parecida com isso:

```
1  /** Configure pins as
2      * Analog
3      * Input
4      * Output
5      * EVENT_OUT
6      * EXTI
7  */
8  static void MX_GPIO_Init(void)
9  {
10
11   GPIO_InitTypeDef GPIO_InitStruct;
12
13   /* GPIO Ports Clock Enable */
14   __HAL_RCC_GPIOA_CLK_ENABLE();
15
16   /*Configure GPIO pin Output Level */
17   HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4|GPIO_PIN_5,
         GPIO_PIN_RESET);
18
19   /*Configure GPIO pins : PA4 PA5 */
20   GPIO_InitStruct.Pin = GPIO_PIN_4|GPIO_PIN_5;
21   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
22   GPIO_InitStruct.Pull = GPIO_NOPULL;
23   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
24   HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
25
26   /*Configure GPIO pin : PA6 */
27   GPIO_InitStruct.Pin = GPIO_PIN_6;
28   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
29   GPIO_InitStruct.Pull = GPIO_PULLUP;
30   HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
31
32 }
```

Seguindo o código acima linha a linha, vemos que na linha 8 é declarada uma instância da *struct* GPIO_InitTypeDef. Essa é uma das estruturas de configuração que a *HAL* usa para configurar periféricos. Cada periférico tem sua estrutura de inicialização, que é montada como desejado e depois passada para uma função de inicialização, que escreve as configurações especificadas nos registradores do periférico. No código acima, isso é feito nas linhas 24 e 30. Repare que a mesma *struct* é utilizada, mas que esta foi completamente reescrita.
Na linha 14, um *define* da *HAL* é usado para ativar o *clock* da *GPIOA* e, basicamente, tirá-la do sono.
Na lina 17, como já vimos, as portas de saída são desligadas, para garantir que eles iniciem em 0.
A partir da lina 20 começa, de fato, a montagem da estrutura de configuração da *GPIO*.
O campo *Pin* é, obviamente, quais pinos estamos configurando. Os pinos são passados em formato binário com uma variável de 16 bits, cada número dpo bit correspondendo ao número da porta (bit 0 -> PA0, bit 1 -> PA1, etc.).
O campo *Mode* seleciona o modo da porta. Neste caso, GPIO_MODE_OUTPUT_PP significa *push/pull*. Outros modos podem ser, por exemplo, *open drain*, modo analógico (entrada do *ADC*), etc. (falaremos sobre eles mais tarde, mas você pode ver os *#defines* dos modos indo até a definição de GPIO_MODE_OUTPUT_PP).
Pull seleciona entre *pull-up* (GPIO_PULLUP), *pull-down* (GPIO_PULLDOWN) e nenhum *pull* (GPIO_NOPULL). O campo *Speed* seleciona a velocidade de chaveamento da porta e pode ter seguintes valores:

- GPIO_SPEED_FREQ_LOW (até 2 MHz)
- GPIO_SPEED_FREQ_MEDIUM (de 4 a 10 MHz)
- GPIO_SPEED_FREQ_HIGH (de 10 a 50 MHz)

Na linha 24, configuramos efetivamente as portas PA4 e PA5, passando para a função HAL_GPIO_Init o endereço base do periférico desejado (textitGPIOA) e, por referência, a estrutura de inicialização que montamos. Das linhas 27 a 30, montamos a estrutura de configuração para a porta PA6, nossa entrada. Note que *Mode* agora é GPIO_MODE_INPUT e *Pull* agora é GPIO_PULLUP, que está de acordo com o que configuramos no *Cube*. Estas configurações podem ser mudadas a contento, mas caso novas configurações sejam geradas com o *Cube*, elas serão sobreescritas. Então, cuidado.

4.3 Conhecendo a *HAL* Mais a Fundo

No código anterior, usamos duas das três funções básicas de operação de portas da *HAL*, cujos protótipo são:

```
GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx,
     uint16_t GPIO_Pin);

void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx,
    uint16_t GPIO_Pin, GPIO_PinState PinState);

void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx,
    uint16_t GPIO_Pin);
```

A primeira função, HAL_GPIO_ReadPin, lê um pino e retorna seu estado, que tem tipo GPIO_PinState. A segunda função, HAL_GPIO_WritePin, escreve em um pino com o nível desejado. O pino é passado no segundo argumento no formato GPIO_PIN_x, se do x o número do pino, e o nível é passado pelo terceiro argumento, sendo GPIO_PIN_SET para nível alto e GPIO_PIN_RESET para nível baixo.
A terceira função, HAL_GPIO_TogglePin, inverte o estado do pino especificado. Todas as funções têm como primeiro argumento o endereço base do módulo GPIO desejado. Isso não é muito aparente quando se usa a função, mas fica claro ao olharmos o protótipo.
Se analisarmos a definição de GPIOA (sempre clicando com o botão direito e selecionando "*Go To Definition Of '...'*"), somos levados até o arquivo stm32f042x6.h, onde encontramos:

```
#define GPIOA   ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB   ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC   ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOF   ((GPIO_TypeDef *) GPIOF_BASE)
```

Ou seja, GPIOA é um *casting* para ponteiro de GPIO_TypeDef... Mas vamos mais a fundo, vendo a definição de GPIOA_BASE:

```
/*!< AHB2 peripherals */
#define GPIOA_BASE            (AHB2PERIPH_BASE + 0
    x00000000)
#define GPIOB_BASE            (AHB2PERIPH_BASE + 0
    x00000400)
#define GPIOC_BASE            (AHB2PERIPH_BASE + 0
    x00000800)
#define GPIOF_BASE            (AHB2PERIPH_BASE + 0
    x00001400)
```

Interessante, parece que todas as *GPIO* estão em um mesmo bloco de memória e seus endereços-base são, por sua vez, *offsets* sobre o endereço-base deste bloco. Se formos até a página 40 do *datasheet* do STM32F042F6, veremos o seguinte:

Isso é um excerto do mapa de memória do microcontrolador, mostrando o endereço dos registradores dos

Table 17. STM32F042x4/x6 peripheral register boundary addresses

Bus	Boundary address	Size	Peripheral
	0x4800 1800 - 0x5FFF FFFF	~384 MB	Reserved
AHB2	0x4800 1400 - 0x4800 17FF	1 KB	GPIOF
	0x4800 0C00 - 0x4800 13FF	2 KB	Reserved
	0x4800 0800 - 0x4800 0BFF	1 KB	GPIOC
	0x4800 0400 - 0x4800 07FF	1 KB	GPIOB
	0x4800 0000 - 0x4800 03FF	1 KB	GPIOA
	0x4002 4400 - 0x47FF FFFF	~128 MB	Reserved
AHB1	0x4002 4000 - 0x4002 43FF	1 KB	TSC
	0x4002 3400 - 0x4002 3FFF	3 KB	Reserved
	0x4002 3000 - 0x4002 33FF	1 KB	CRC
	0x4002 2400 - 0x4002 2FFF	3 KB	Reserved
	0x4002 2000 - 0x4002 23FF	1 KB	Flash memory interface
	0x4002 1400 - 0x4002 1FFF	3 KB	Reserved
	0x4002 1000 - 0x4002 13FF	1 KB	RCC
	0x4002 0400 - 0x4002 0FFF	3 KB	Reserved
	0x4002 0000 - 0x4002 03FF	1 KB	DMA
	0x4001 8000 - 0x4001 FFFF	32 KB	Reserved
APB	0x4001 5C00 - 0x4001 7FFF	9 KB	Reserved
	0x4001 5800 - 0x4001 5BFF	1 KB	DBGMCU
	0x4001 4C00 - 0x4001 57FF	3 KB	Reserved
	0x4001 4800 - 0x4001 4BFF	1 KB	TIM17
	0x4001 4400 - 0x4001 47FF	1 KB	TIM16
	0x4001 3C00 - 0x4001 43FF	2 KB	Reserved
	0x4001 3800 - 0x4001 3BFF	1 KB	USART1
	0x4001 3400 - 0x4001 37FF	1 KB	Reserved
	0x4001 3000 - 0x4001 33FF	1 KB	SPI1/I2S1
	0x4001 2C00 - 0x4001 2FFF	1 KB	TIM1
	0x4001 2800 - 0x4001 2BFF	1 KB	Reserved
	0x4001 2400 - 0x4001 27FF	1 KB	ADC
	0x4001 0800 - 0x4001 23FF	7 KB	Reserved
	0x4001 0400 - 0x4001 07FF	1 KB	EXTI
	0x4001 0000 - 0x4001 03FF	1 KB	SYSCFG
	0x4000 8000 - 0x4000 FFFF	32 KB	Reserved

Figura 7: Endereços-base das *GPIOA*.

periféricos. Na parte superior da tabela, podemos ver, para os módulos *GPIO* correspondentes, os mesmos *offsets* que no bloco de *#defines* acima. Indo ainda mais a fundo, até a definição de AHB2PERIPH_BASE, chegamos aqui:

```
/** @addtogroup Peripheral_memory_map
  * @{
  */

#define FLASH_BASE      ((uint32_t)0x08000000U) /*
    !< FLASH base address in the alias region */
#define FLASH_BANK1_END ((uint32_t)0x08007FFFU) /*
    !< FLASH END address of bank1 */
#define SRAM_BASE       ((uint32_t)0x20000000U) /*
    !< SRAM base address in the alias region */
#define PERIPH_BASE     ((uint32_t)0x40000000U) /*
    !< Peripheral base address in the alias region
    */

/*!< Peripheral memory map */
#define APBPERIPH_BASE  PERIPH_BASE
#define AHBPERIPH_BASE  (PERIPH_BASE + 0x00020000)
#define AHB2PERIPH_BASE (PERIPH_BASE + 0x08000000)
```

Onde fica claro por simples análise que AHB2PERIPH_BASE = 0x48000000U, como vimos na tabela de registradores.
Mas afinal, o que é mesmo aquele GPIO_TypeDef?

Bom, como o nome diz, ele é uma definição de tipo de *GPIO*... Vamos até sua definição, também no arquivo stm32f042x6.h:

```
/**
  * @brief General Purpose I/O
  */

typedef struct
{
  __IO uint32_t MODER;  /*!< GPIO port mode
      register,   Address offset: 0x00*/
  __IO uint32_t OTYPER; /*!< GPIO port output type
      register,  Address offset: 0x04*/
  __IO uint32_t OSPEEDR;/*!< GPIO port output speed
       register, Address offset: 0x08*/
  __IO uint32_t PUPDR;  /*!< GPIO port pull-up/pull
      -down register,  Address offset: 0x0C*/
  __IO uint32_t IDR;    /*!< GPIO port input data
      register,   Address offset: 0x10*/
  __IO uint32_t ODR;    /*!< GPIO port output data
      register,  Address offset: 0x14*/
  __IO uint32_t BSRR;   /*!< GPIO port bit set/
      reset register,Address offset: 0x1A */
  __IO uint32_t LCKR;   /*!< GPIO port
      configuration lock register, Address offset:
      0x1C*/
  __IO uint32_t AFR[2]; /*!< GPIO alternate
      function low register,  Address offset: 0x20
      -0x24 */
  __IO uint32_t BRR;    /*!< GPIO bit reset
      register,   Address offset: 0x28*/
} GPIO_TypeDef;
```

Aqui nós vemos que o tipo é uma organização em estrutura de todos os registradores de um módulo *GPIO*, construido em cima de endereços relativos (*offset*) a uma base. Quando configuramos uma *GPIO*, estamos alterando estes registradores. Quando ligamos e desligamos uma porta, estamos alterando os campos BSRR e BRR da estrutura, respectivamente. Estes registradores dão um comando para escrever ou limpar os respectivos bits do registrador ODR daquela *GPIO*. Eles podem ser apenas escritos e não lidos, isso porque seu valor é imediatamente apagado pelo microcontrolador. Este ODR (*Output Data Register*) pode ser lido ou escrito pelo usuário e contém o estado das portas de saída do microcontrolador. O estado dos pinos de entrada são dados pelo registrador IDR, que pode ser apenas lido.

Voltando aos argumentos das funções básicas, o segundo argumento é o pino que desejamos alterar. Seu tipo é apenas um uint16_t. Se formos até a definição de, por exemplo, GPIO_PIN_6, vemos o seguinte:

```
/** @defgroup GPIO_pins GPIO pins
  * @{
  */
#define GPIO_PIN_0  ((uint16_t)0x0001U)  /* Pin 0
    selected    */
#define GPIO_PIN_1  ((uint16_t)0x0002U)  /* Pin 1
    selected    */
#define GPIO_PIN_2  ((uint16_t)0x0004U)  /* Pin 2
    selected    */
#define GPIO_PIN_3  ((uint16_t)0x0008U)  /* Pin 3
    selected    */
#define GPIO_PIN_4  ((uint16_t)0x0010U)  /* Pin 4
    selected    */
#define GPIO_PIN_5  ((uint16_t)0x0020U)  /* Pin 5
    selected    */
#define GPIO_PIN_6  ((uint16_t)0x0040U)  /* Pin 6
    selected    */
#define GPIO_PIN_7  ((uint16_t)0x0080U)  /* Pin 7
    selected    */
#define GPIO_PIN_8  ((uint16_t)0x0100U)  /* Pin 8
    selected    */
#define GPIO_PIN_9  ((uint16_t)0x0200U)  /* Pin 9
    selected    */
#define GPIO_PIN_10 ((uint16_t)0x0400U)  /* Pin 10
    selected    */
#define GPIO_PIN_11 ((uint16_t)0x0800U)  /* Pin 11
    selected    */
#define GPIO_PIN_12 ((uint16_t)0x1000U)  /* Pin 12
    selected    */
#define GPIO_PIN_13 ((uint16_t)0x2000U)  /* Pin 13
    selected    */
#define GPIO_PIN_14 ((uint16_t)0x4000U)  /* Pin 14
    selected    */
#define GPIO_PIN_15 ((uint16_t)0x8000U)  /* Pin 15
    selected    */
#define GPIO_PIN_All((uint16_t)0xFFFFU)  /* All
    pins selected */

#define GPIO_PIN_MASK  (0x0000FFFFU) /* PIN mask
    for assert test */
```

Estes *#defines* especificam cada porta da *GPIO* como um bit de um uint16_t, que será escrito no BSRR ou BRR, dependendo do que se quer fazer, se *setar* ou *resetar* estas portas.

O terceiro argumento da função HAL_GPIO_WritePin é o estado desejado do pino. Se nós analisarmos a definição do tipo do argumento, GPIO_PinState (sempre clicando com o botão direito e selecionando "*Go To Definition Of '...'*"), veremos que ela é uma enumeração (código abaixo) com os elementos GPIO_PIN_RESET, igual a 0 *unsigned*, e GPIO_PIN_SET, literalmente, qualquer outra coisa que não 0.

```
/**
  * @brief  GPIO Bit SET and Bit RESET enumeration
  */
typedef enum
{
  GPIO_PIN_RESET = 0U,
  GPIO_PIN_SET
}GPIO_PinState;
```

Assim, o tipo GPIO_PinState implementa efetivamente um tipo booleano. Poderíamos ter escrito 0 e 1 no lugar de GPIO_PIN_RESET e GPIO_PIN_SET que o

efeito seria o mesmo. Esses contornos parecem despendiosos e sem muito propósito, mas são otimizados fora pelo compilador e melhoram a compreensão de código por se aproximarem da linguagem natural.

4.4 Interrupções Externas

Interrupções externas são gerenciadas pelo módulo *EXTI* (*Extended Interrupts and Events Controller*). Este módulo possui três *interrupt requests* para as portas *GPIO*. Chamando-as pelo prefixo do nome da função que trata seus *requests*, elas são:

- EXTI0_1_IRQ - gera os *requests* para as portas Px0 e Px1 (linhas 0 e 1 do *EXTI*);
- EXTI2_3_IRQ - gera os *requests* para as portas Px2 e Px3 (linhas 2 e 3 do *EXTI*);
- EXTI4_15_IRQ - gera os *requests* para as portas restantes, de Px4 a Px15 (linhas de 4 a 15 do *EXTI*);

Notamos por "Px"os pinos de qualquer *GPIO*, isto é, Px0 pode ser PA0, PB0, PC0, etc. Cada linha pode servir uma porta por vez, ou seja, não é possível, por exemplo, ter interrupções nas portas PA0 e PF0 ao mesmo tempo, pois elas compartilham a mesma linha do *EXTI*.
Vamos a um exemplo. Nós controlaremos novamente os *LED*s da placa, desta vez usando interrupções.
Crie um novo projeto para o nosso microcontrolador. As configurações são as mesmas que no projeto anterior, exceto para a porta PA6, como vemos na Figura 8. Para a porta PA6, escolhemos GPIO_EXTI6, que seleciona o modo *EXTI* da porta.

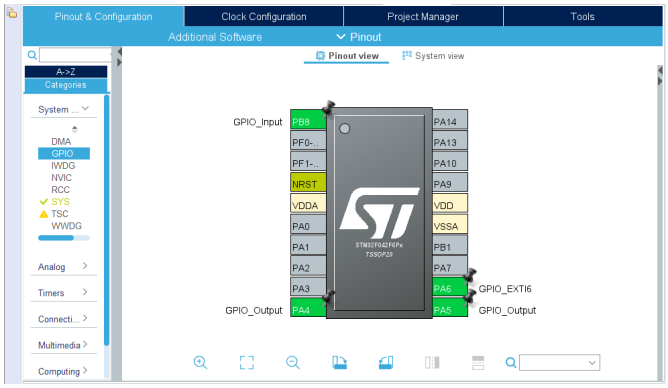

Figura 8: *Pinout* para o exemplo de uso de interrupção externa.

Abra as configurações de *GPIO* e selecione a porta PA6 da tabela. Para ela, coloque um *pull-up* e escolha interrupção na borda subida, selecionando *External Interrupt Mode with Rising edge trigger detection* na *dropdown box* de nome *GPIO mode* (como na Figura 9). Ainda na mesma aba, abra as configurações do NVIC (*Nested Vectored Interrupt Controller*, gerenciador de interrupções da arquitetura ARM). Lá, habilite as interrupções das linhas de 4 a 15 do *EXTI*, marcando a *checkbox* do lado de *EXTI line 4 to 15 interrupts* na tabela de interrupções, como na Figura 10.

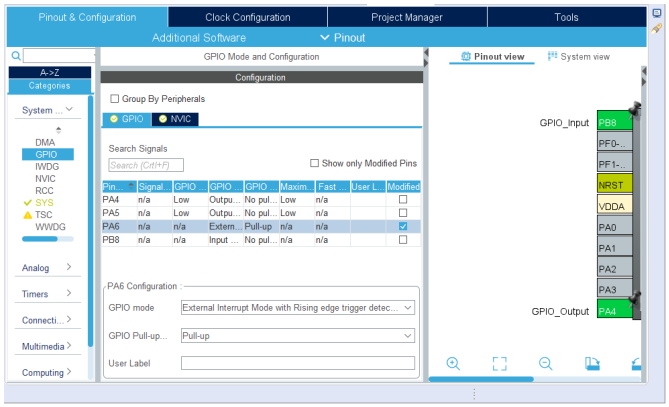

Figura 9: Configuração da porta PA6 para o exemplo de interrupção externa.

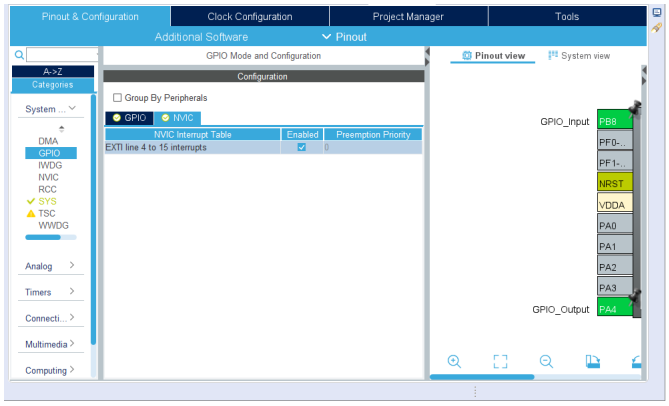

Figura 10: Configuração do *NVIC* para o exemplo de interrupção externa.

Abra o arquivo stm32f0xx_it.c. Encontre a implementação da função EXTI4_15_IRQHandler, que deve estar no final do arquivo. Ela deve se parecer assim:

```
1  void EXTI4_15_IRQHandler(void)
2  {
3    /* USER CODE BEGIN EXTI4_15_IRQn 0 */
4
5    /* USER CODE END EXTI4_15_IRQn 0 */
6    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_6);
7    /* USER CODE BEGIN EXTI4_15_IRQn 1 */
8
9    /* USER CODE END EXTI4_15_IRQn 1 */
10 }
```

Esta função é chamada quando uma interrupção ocorre nas linhas de 4 a 15 do *EXTI*. Dentro dela

são então chamadas funções de serviço de interrupção para cada linha, que então verificam se há interrupções pendentes naquela linha, posteriormente chamando uma função *Callback*. Vá até a definição de HAL_GPIO_EXTI_IRQHandler. Ela deve se parecer com isso:

```
/**
  * @brief  Handle EXTI interrupt request.
  * @param  GPIO_Pin Specifies the port pin
      connected to corresponding EXTI line.
  * @retval None
  */
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
{
  /* EXTI line interrupt detected */
  if(__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != RESET)
  {
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
    HAL_GPIO_EXTI_Callback(GPIO_Pin);
  }
}
```

O que é feito no código acima, basicamente, é verificar a interrupção no pino está pendente com a macro __HAL_GPIO_EXTI_GET_IT(), limpar o *flag* de interrupção daquele pino com a macro __HAL_GPIO_EXTI_CLEAR_IT(), e finalmente chamar a função HAL_GPIO_EXTI_Callback, onde será implementado de fato o que nós queremos fazer dentro da interrupção da porta. Vá até a definição de HAL_GPIO_EXTI_Callback. Você deve ver algo assim:

```
/**
  * @brief  EXTI line detection callback.
  * @param  GPIO_Pin Specifies the port pin
      connected to corresponding EXTI line.
  * @retval None
  */
__weak void HAL_GPIO_EXTI_Callback(uint16_t
    GPIO_Pin)
{
  /* Prevent unused argument(s) compilation warning
       */
  UNUSED(GPIO_Pin);

  /* NOTE: This function should not be modified,
      when the callback is needed,
            the HAL_GPIO_EXTI_Callback could be
                 implemented in the user file
   */
}
```

Isso é um *placeholder* da função de *callback* que deve ser implementada pelo usuário. A função é implementada com o modificador __weak, o que faz com que esta implementação seja ignorada pelo compilador sem lançar um erro caso haja alguma implementação outra mais forte, sem o __weak, como a que vamos implementar agora. No arquivo main.c, insira esta implementação na seção de código do usuário 4 (USER CODE BEGIN 4 e USER CODE END 4), lá pelo final do arquivo:

```
/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if(GPIO_Pin == GPIO_PIN_6)
  {
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_4|GPIO_PIN_5
        );

    // Debounce:
    HAL_NVIC_DisableIRQ(EXTI4_15_IRQn); // desliga
        interrupt
    has_interrupted = 1;  // flag para o delay no
        main
  }
}
/* USER CODE END 4 */
```

Repare que não precisamos (nem devemos!) declarar o protótipo da função lá em cima, antes da função main(), pois seu protótipo já foi declarado pela *HAL* no header stm32f0xx_hal_gpio.h. Agora, na função main(), sobrescreva o *loop* infinito com o seguinte código:

```
/* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    if(has_interrupted)
    {
      has_interrupted = 0;

      HAL_Delay(300);

      __HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_6);
      HAL_NVIC_EnableIRQ(EXTI4_15_IRQn);
    }
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
```

E lá no começo do arquivo, declare a variável has_interrupted nas variáveis privadas:

```
/* USER CODE BEGIN PV */
/* Private variables -------*/
int has_interrupted = 0;
/* USER CODE END PV */
```

Compile e baixe o programa no microcontolador. Você deve observar que, para cada vez que você solta o botão, os *LED*s invertem seu estado (sem *bouncing*!), que era nosso objetivo.

O que fizemos foi o seguinte: na função HAL_GPIO_EXTI_Callback(), na linha 4, testamos qual pino (qual linha do *EXTI*) gerou aquele *interrupt request*. Isso é feito pois as linhas de 4 a 15, como visto, disparam o mesmo vetor de interrupção e acabam chamando a mesma função de *callback*. Assim, se

tivermos mais de um pino com interrupção ativada, precisamos verificar qual está interrompendo. Após isso, chaveamos os *LED*s. Depois, implementamos um *debounce à la* Arduino, desligando as interrupções do *EXTI* e executando um *blocking delay* (dentro da função main()). Na função main(), dentro do laço *while*, nós testamos se a interrupção ocorreu na linha 5. Caso sim, contamos 300 ms, limpamos o flag de interrupção da linha 6 do *EXTI* (porta PA6), e ligamos novamente as interrupções. Note que é muito importante limparmos os *flags* de interrupção logo antes de ligá-las novamente. Isso porque nós desligamos apenas a geração de *interrupt requests* por parte do módulo *NVIC*, mas não a geração de *flags* de interrupção por parte do módulo *EXTI*. Portanto, nada garante que não haja uma *flag* de interrupção pendente, que será tratada assim que ligarmos as interrupções, e por isso devemos sempre limpá-las.

5 *Timers*

5.1 *Overview*

Timers são cruciais para várias tarefas, desde uma rotina de *debounce*, até o chaveamento de um inversor, passando por *dimmers* e motores servo.

Nosso STM32F0 possui 6 *timers*, de três tipos diferentes. Os *timers* são listados abaixo, juntamente com uma breve descrição das *features* de cada tipo de *timer*:

- *Advanced-Control Timer* - TIM1
  - *Prescaler* e ARR de 16 bits;
  - Modos de contagem *up*, *down*, e *center aligned* (*up/down*) 1, 2, e 3;
  - 4 canais (para *input capture*, *input compare*, geração de *PWM*, e modo de pulso);
  - Modo *slave* (para concatenar *timers*);
  - Contador de repetição, para acionar o evento de *update* após *N over/underflows*;
  - Saída de *PWM* complementar geração de *dead time*;
  - Entrada de sinal de *break* (para resetar todo o *timer* de uma vez).
- *General-Purpose Timer* - TIM2 e TIM3
  - *Prescaler* e ARR de 32 (TIM2) e 16 (TIM3) bits;
  - Modos de contagem *up*, *down*, e *center aligned* (*up/down*) 1, 2, e 3;
  - Modo *slave* (para concatenar *timers*);
  - 4 canais (para *input capture*, *input compare*, geração de *PWM*, e modo de pulso);
- *General Purpose Timer* (mais para *Basic Timer* na verdade...) - TIM14, TIM16, e TIM17
  - *Prescaler* e ARR de 16 bits;
  - Modo de contagem *up* (e só);
  - 1 canal (para *input capture*, *input compare*, geração de *PWM*, e modo de pulso);

A finalidade dos *timer* TIM1 é o uso com aplicações mais complexas, que requerem cuidados especiais. Por exemplo, TIM1 seria ideal para um inversor de frequência: a *feature* de *PWM*s complementares serviria para controlar dois MOSFETs conectados aos dois braços e ao mesmo nó, quando apenas um pode estar ligado de cada vez; *deadtime* (período de interstício após o chaveamento onde os dois PWMs complementares ficam desligados) evitaria um quase-curto entre braços durante o chaveamento dos mesmos; o *break* para o funcionamento de todo o módulo de uma só vez e o coloca em estado conhecido; etc. Também cobre as funcionalidades dos *General Purpose Timers*.

Os *General Purpose Timers* são destinados a aplicações mais pedestres, como PWMs sincronizados, *trigger* de outros módulos, como ADC, captura de largura de pulso, etc.

Os *General Purpose Timers* mais básicos são bem feijão-com-arroz, fazendo apenas o mínimo que se espera de um *timer* e sendo usados para aplicações básicas, basicamente para não gastar um *timer* com mais opções.

Falando um pouco de terminologia, *updtate* é como se chama, genericamente, os eventos de *overflow* (quando em contagem crescente de contagem) ou *underflow* (quando em direção decrescente). *Capture/Compare* é como se chama, genericamente, as funcionalidades de *Input Capture*, onde um intervalo de tempo é capturado em um CCR (*Capture/Compare Register*), e textitOutput Compare, onde o contador do timer é comparado com um CCR para, quando atingir o valor, chavear uma saída ou disparar uma interrupção. *Center Aligned Mode* é o modo de contagem *up/down*, contando de modo crescente até o valor do período (ARR, ou *Auto-Reload*

Register), e depois de modo decrescente até zero. Este modo é subdividido em *Center Aligned Mode* 1, 2, e 3. A diferença é quando as *interrupt flags* dos *Output Compares* são geradas, respectivamente, na contagem crescente, decrescente, e em ambas as direções de contagem.

A seguir, veremos alguns exemplos de como usar os *timers*.

5.2 *PWM*

mais um *blinky*, affs... (-.-)'

Vamos brincar um pouco com *PWM*s. Primeiramente, vamos criar um *blinky* de 0,5 Hz, para enxergamos o *PWM* funcionando, depois faremos um *dimmer*, mudando apenas o *prescaler*.

Crie um novo projeto. No *Pinout* do projeto, configure a porta PA5 como canal 1 do Timer 2 clicando sobre ela e selecionando TIM2_CH1 (Figura 11). Na árvore de periféricos à esquerda, expanda TIM2 e ligue-o selecionando *Clock Source* como *Internal Clock*. Também acione a saída PWM na porta PA5 selecionando *PWM Generation CH1* na *dropdown box Channel 1*.

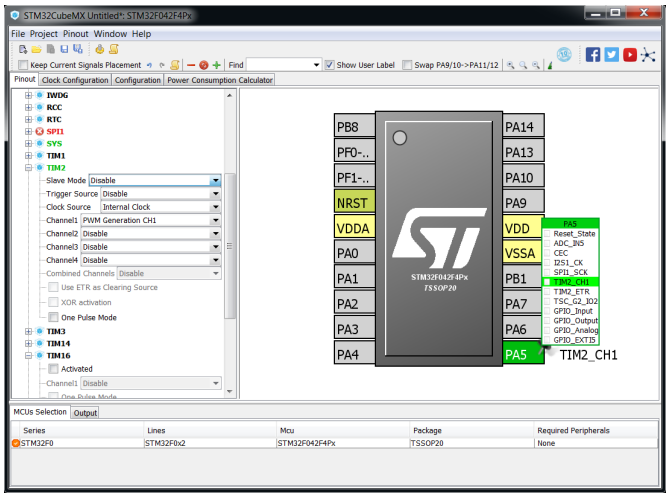

Figura 11: Configuração do TIM2 para nosso primeiro exemplo de *PWM*.

Abra as configurações do TIM2 na aba *Configuration*, clicando em TIM2 na coluna *Control* (Figura 12). Nosso *PWM* terá 0,5 Hz inicialmente, e uma resolução de 1000 pontos. Como nosso *clock*, por *default*, é 8 MHz, precisamos dividí-lo por 16.000.000, ou por 16.000 e depois por 1.000. Para isso é que existe *prescaler*. Note antes, porém, que *prescaler* é um contador, e se quisermos dividir um *clock* por X, precisamos carregá-lo com X-1, já que ele começa a contar do 0. A mesma observação vale para os outros valores. Assim, entre com 15.999 para o *prescaler* e 999 para o *Counter Period*.

Repare que não usamos a *Internal Clock Division (CKD)* porque esta *não tem nada a ver* com divisão de *clock* para gerar o *clock* do *timer*. Ela se destina a outros módulos. O divisor legítimo de *clock* do *timer* é o *prescaler*.

Vamos escolher um *Duty Cycle* inicial de 25%, escolhendo um *Pulse* de 249 sob *PWM Generation Channel 1*. Ative o *auto-reload preload* e selecione *PWM mode 1* no modo do *PWM*. Já podemos gerar o código base.

O modo 1 do *PWM* seta a saída até que o contador atinja o valor de *Pulse*, quando ela é resetada. O modo 2 faz o contrário, deixando a saída resetada até que o contador atinja o valor de *Pulse*, quando ela é setada. O *auto-reload preload* melhora a alteração do registrador ARR (*Auto Reload Register*), que define nosso período. Com ele ativo, quando mudamos o perído, o registrador de *preload* é carregado, passando o valor para ARR no momento oportuno, logo após a contagem acabar. Assim, o período no qual tentamos mudar o ARR continuará até o final do jeito que estava e assumirá a nova configuração ao final da contagem (*overflow* ou *underflow*).

Caso isso não fosse feito, nossa contagem seria imediatamente resetada se o novo valor de período fosse menor que a contagem presente (criando um pulso anômalo). Isso na verdade, também deve ser feito para os CCRs (*Capture Compare Register*), setando o bit CCxPE no registrador TIMy_CCMRx ('y' representando on número do *Timer* e 'x' o número do registrador de capturacomparação), mas isso já é feito automaticamente pela *HAL* quando ligamos o *PWM*, de tão imprevisível que o comportamento seria caso não o fizéssemos. Isso é um grande problema em outros microcontroladores, caso o usuário não tome muito cuidado com a atualização do *Duty Cycle*, por exemplo, alterando-o diretamente no *loop* do programa.

Abra o main.c. Quando usamos a *HAL* para gerar nossas configurações, ela inicializa nossos periféricos (configura), mas não os *inicia*, ou "*starta*". Precisamos fazer isso nós mesmos, no ponto onde desejamos começar a usar o periférico. Digite o seguinte código no USER CODE 2:

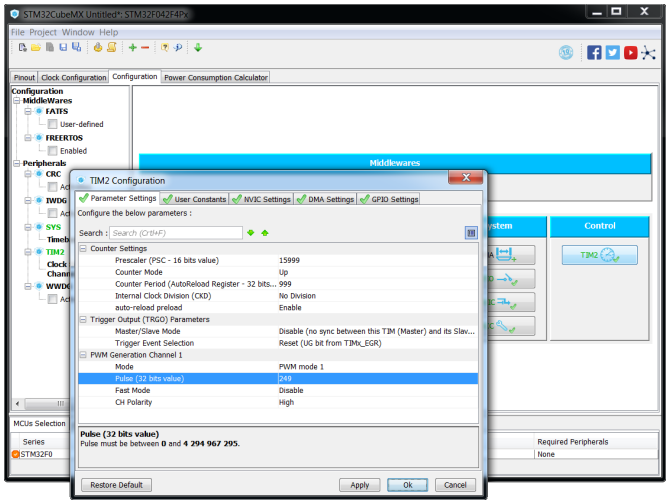

Figura 12: *Pinout* para nosso exemplo de *PWM*.

```
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start(&htim2);
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
/* USER CODE END 2 */
```

Baixe o código no *MCU*. Você deve observar um pulso curto, que se repete a cada dois segundos. O que acabamos de fazer é um *PWM* muito lento, de 0,5 Hz. Mais abaixo aumentaremos a frequência para 32 kHz e teremos um *PWM* mais reconhecível.

Das funções acima, HAL_TIM_Base_Start() inicia o funcionamento do *Timer*, começando a contar. HAL_TIM_PWM_Start() habilita a operação dos canais, neste caso, direcionado para a saída na porta PA5. O primeiro argumento das duas funções é uma referência para um tal de htim2. Isso é um *handler* de *timer*, conceito que ainda não tínhamos visto nas *GPIO*. O *handler* guarda diferentes aspectos do periférico e como ele se relaciona com outros (como *DMA*, por exemplo), além de servir de "alça"(*handler*) para manusear-lo. Se subirmos algumas linhas no código, veremos isso:

```
/* Private variables ---------*/
TIM_HandleTypeDef htim2;
```

E se seguirmos a definição do tipo TIM_HandleTypeDef, veremos a estrutura do *handler*:

```
/**
  * @brief  TIM Time Base Handle Structure definition
  */
typedef struct
{
  TIM_TypeDef *Instance; /*!< Register base address */
  TIM_Base_InitTypeDef  Init;  /*!< TIM Time Base required parameters */
  HAL_TIM_ActiveChannel Channel;  /*!< Active channel */
  DMA_HandleTypeDef *hdma[7];  /*!< DMA Handlers array.This array is accessed by a @ref TIM_DMA_Handle_index */
  HAL_LockTypeDef Lock;  /*!< Locking object */
  __IO HAL_TIM_StateTypeDef  State;  /*!< TIM operation state */
}TIM_HandleTypeDef;
```

Aqui vemos alguns conhecidos: "Instance"é um ponteiro para o *type* de *timer*, que é uma organização em estrutura dos seus registradores; e "Init"é a estrutura de inicialização do periférico. "Channel"é usado para interrupções e nos diz qual canal está "ativo", ou seja, com interrupção pendente (usaremos isso mais adiante). Em seguida são definidos alguns *handlers* de *DMA*. "Lock"é uma proteção contra escrita dos registradores. Por fim, "State"é o estado do periférico, que pode ser, a título de conhecimento, HAL_TIM_STATE_RESET, HAL_TIM_STATE_READY, HAL_TIM_STATE_BUSY, etc. É sempre muito elucidativo adentrar as inúmeras definições de estruturas, tipos e enumerações da *HAL*, mas o faremos menos e menos para os periféricos mais complexos, simplesmente por ser impraticável entrar por todos esses buracos de coelho como esse. Mas recomendamos sempre que você faça isso na usa *IDE*.

PWM de verdade – *duty cycle variável*

Agora que entendemos um pouco melhor o que está acontecendo, vamos elaborar. Faremos um *PWM* de 8 kHz, variando continuamente seu *duty cycle* de 0 a 100%, o que será feito com outro *Timer*. Volte ao projeto do *Cube*. No *Pinout*, ative o TIM3, expandindo-o e marcando a *checkbox Internal Timer*. Não usaremos nenhum canal, apenas o contador. Na aba *Configuration*, abra as configurações do TIM2 e zere o *Prescaler*. Assim, não dividiremos o *clock* além de pelo período, e então ficamos com um *PWM* de 8 kHz (8 MHz/(999+1)).

Nas configurações do TIM3, use um *Prescaler* de 7.999 (dividindo por 8.000) e um período de 999 (como na figura 13). É muito importante este período ser idêntico ao do TIM2, pois atribuiremos diretamente o valor do contador do TIM3 ao CCR1 (*Capture/Compare Register*, vulgo *duty cycle*) do TIM2. Também ative a contagem crescente *e* descrescente, escolhendo *Center Aligned Mode* em *Counter Mode*. A diferença entre *Center Aligned Mode* 1, 2, e 3, é quando as *interrupt flags* dos *Output Compares* são geradas, respectivamente, na contagem para cima, para baixo, e em ambas as direções de contagem.

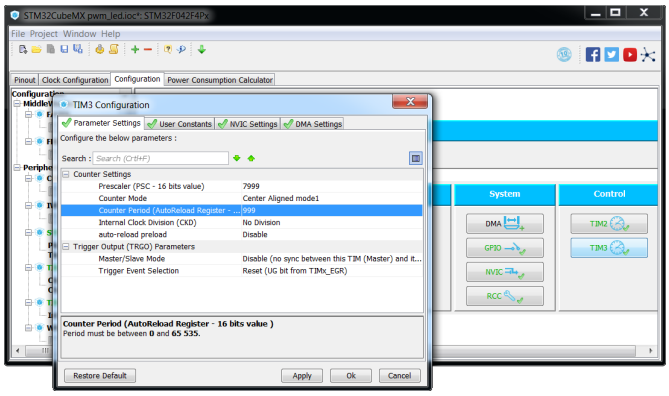

Figura 13: Configuração do TIM3 para nosso segundo exemplo de *PWM*.

Podemos gerar novamente o código-base. No main.c, ative o TIM3 logo após o TIM2, antes do *loop* do programa:

```
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start(&htim2);
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);

HAL_TIM_Base_Start(&htim3);
/* USER CODE END 2 */
```

Dentro do *loop*, atualizaremos o CCR1 usando uma macro da *HAL*, atribuindo-o diretamente o valor do contador do TIM3 (CNT):

```
/* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
  __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1,
      htim3.Instance->CNT);
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
```

Compile e baixe o programa. Você verá o *LED* da porta PA5 variar constantemente seu brilho, de máximo a mínimo, com um período de 2 segundos. Repare que o jeito como isso foi feito, atualizando o *duty cycle* a cada iteração do *loop*, só pôde funcionar por causa dos registradores de *preload* (OCxPE, ou *Output Compare Preload Enable* do canal 'x'), com os quais não precisamos nos preocupar pois a *HAL* nem nos dá a opção de usar *PWM*s sem ligá-los. Sem eles, as atualizações do CCR1 se dariam em momentos imprevisíveis, fazendo a onda "pular"e criar pulsos anômalos. Assim, podemos, por exemplo, atualizar a largura de pulso do sinal de um motor servo, sem nos preocuparmos com o momento em que isso ocorre (o que normalmente usaria uma interrupção do CCRx).

5.3 Interrupções

~~Interrompemos a nossa programação para transmitir em rede nacional o pronunciamento do Excelentíssimo Senhor Presidente da República Tom Marvolo Riddle. Dentro de 20 minutos, continue assistindo Domingão do Faustão.~~

Os *timers* dos STM32 possuem vetores de interrupção (ou IRQ) compartilhados, gerando a mesma requisição de interrupção ao *NVIC* para vários eventos, como *overflow* e *underflow* (o que a STM chama conjuntamente de *update*), chegada ao valor dos CCRx, etc. Esse vetor de interrupção deve então ser analisado dentro da rotina de serviço de interrupção (ou *callback function*, na nossa nomenclatura) para determinar de qual evento se trata e tomar as ações correspondentes. Esse é o tal do vetor de interrupção "mascarável"(*maskable interrupt vector*), bem comum em microcontroladores.

O *timer* TIM1 possui dois vetores de interrupção. Um para os CCRx e um para os eventos de *break*, *update*, *trigger* e *commutation*.

Os *timers* restantes (TIM2, TIM3, etc.) dos STM32 possuem um único vetor de interrupção, compartilhado com todos os eventos e chamado de *global interrupt*.

Vamos a um exemplo de como usá-las com o TIM2.

Usando Interrupções de Update e CC

Vamos fazer um *PWM* (do jeito errado, para fins ilustrativos) usando interrupções de *update* e CC. Ligaremos o *LED* no *overflow* (*update*) e desligaremos quando o *timer* atingir o CCR1. Crie um projeto no *Cube*, configure as portas PA4 e PA5 como saída, e ligue o *timer* TIM2 (como na Figura 14). Na aba *Configuration* abra o TIM2 e ligue o vetor de interrupção global (*global interrupt*), como na Figura 15, sob a abar *NVIC*. Como na Figura 16, configure um período de 4.000 ciclos (ARR = 3.999). Assim, como nosso *clock* é o *default* de 8 MHz, obtemos uma frequência do *timer* de 2 kHz com um período de 2.000 ciclos do *clock* do *timer*. Gere o código base e abra-o.

Como com qualquer interrupção que utilizemos com a *HAL*, usaremos *callback functions* para tratar as interrupções. Na árvore do projeto, abra o arquivo stm32f0xx_it.c sob a pasta *Application/User*. Lá, procure pela função TIM2_IRQHandler. Ela é a função chamada quando uma IRQ é acionada e deve se parecer com isso:

```
/* @brief This function handles TIM2 global
    interrupt.
*/
void TIM2_IRQHandler(void)
```

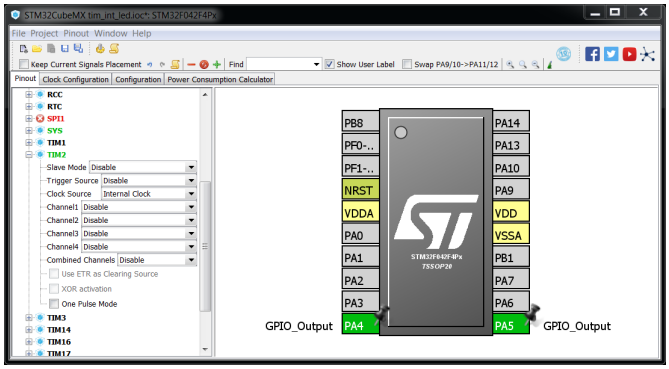

Figura 14: Configuração do *timer*

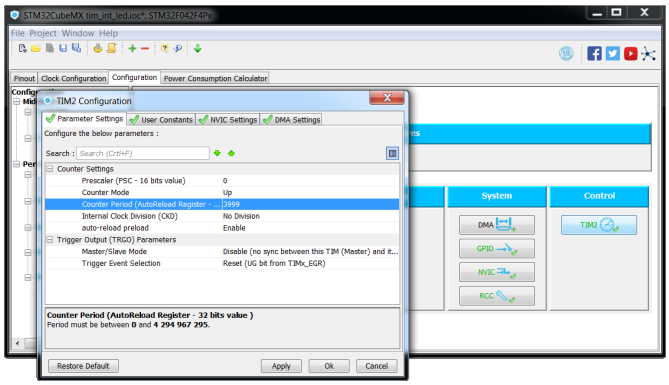

Figura 15: Configuração do *timer*

```
4  {
5    /* USER CODE BEGIN TIM2_IRQn 0 */
6
7    /* USER CODE END TIM2_IRQn 0 */
8    HAL_TIM_IRQHandler(&htim2);
9    /* USER CODE BEGIN TIM2_IRQn 1 */
10
11   /* USER CODE END TIM2_IRQn 1 */
12 }
```

Como você pode ver, a única coisa que a função faz é chamar HAL_TIM_IRQHandler, passando para ela o *handler* to TIM2. Caso houvesse mais *timers* ativados, haveria mais chamadas da função, uma para cada *timer*.

Vá até a definição da função HAL_TIM_IRQHandler. Ela é um tanto extensa, portanto mostramos apenas a parte que interessa no código abaixo. Na função, a linha 9 verifica se a *flag* de interrupção vem do CC1. Caso sim, na linha 11 é verificado se as interrupções para esse canal estão mesmo ligadas. Caso sim, na linha 14 a *flag* é limpa. Portanto, não precisamos limpar a *flag* nós mesmos, como de costume em *maskable interrupts*.
Em seguida, o campo Channel do *handler* do *timer* é escrito com o canal com interrupção pendente. Isso é usado dentro da função de *callback* para determinar qual dos canais gerou a interrupção e deve ser tratado, já que o mesmo vetor de interrupção é usado para todos. Após isso, verifica-se o modo de

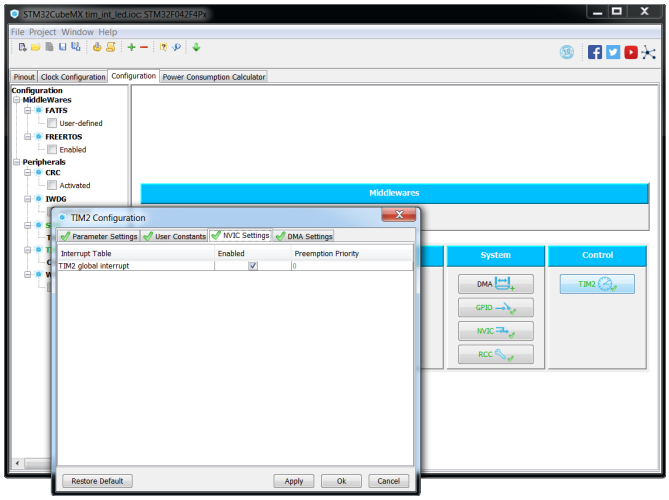

Figura 16: Configuração das interrupções

funcionamento do canal, se *input capture* ou *output compare*. O nosso caso é o último, portanto, a função HAL_TIM_OC_DelayElapsedCallback é chamada, recebendo como argumento o *timer* cuja interrupção está sendo tratada.

A função HAL_TIM_OC_DelayElapsedCallback é declarada no mesmo arquivo como o modificador __weak, ou seja, é um *placeholder* para uma substituição do usuário. Caso a *flag* de interrupção não venha do CC1, os outros canais são testados (CC2, CC3 e CC4), e, após isso, o *flag* de *update*. Após verificar se a interrupção está mesmo ligada e limpar o *flag*, a função HAL_TIM_PeriodElapsedCallback é chamada. Esta função não é compartilhada com outro evento e não exige verificação da fonte dentro dela. Também é definida com __weak e também será substituída.

```
1  /**
2    * @brief  This function handles TIM interrupts
         requests.
3    * @param  htim TIM  handle
4    * @retval None
5    */
6  void HAL_TIM_IRQHandler(TIM_HandleTypeDef *htim)
7  {
8    /* Capture compare 1 event */
9    if(__HAL_TIM_GET_FLAG(htim, TIM_FLAG_CC1) !=
         RESET)
10   {
11     if(__HAL_TIM_GET_IT_SOURCE(htim, TIM_IT_CC1) !=
           RESET)
12     {
13       {
14         __HAL_TIM_CLEAR_IT(htim, TIM_IT_CC1);
15         htim->Channel = HAL_TIM_ACTIVE_CHANNEL_1;
16
17         /* Input capture event */
18         if((htim->Instance->CCMR1 & TIM_CCMR1_CC1S)
               != 0x00U)
```

```
19          {
20            HAL_TIM_IC_CaptureCallback(htim);
21          }
22          /* Output compare event */
23          else
24          {
25            HAL_TIM_OC_DelayElapsedCallback(htim);
26            HAL_TIM_PWM_PulseFinishedCallback(htim);
27          }
28          htim->Channel =
                HAL_TIM_ACTIVE_CHANNEL_CLEARED;
29        }
30      }
31    }
32    // ... A mesma coisa para CC2, CC3, e CC4 ...
33
34    /* TIM Update event */
35    if(__HAL_TIM_GET_FLAG(htim, TIM_FLAG_UPDATE) !=
          RESET)
36    {
37      if(__HAL_TIM_GET_IT_SOURCE(htim, TIM_IT_UPDATE)
            !=RESET)
38      {
39        __HAL_TIM_CLEAR_IT(htim, TIM_IT_UPDATE);
40        HAL_TIM_PeriodElapsedCallback(htim);
41      }
42    }
43
44    // ... a mesma coisa para break e trigger
45  }
```

Vamos redefinir as funções de *callback*. No main.c, no *User Code 4*, quase no final do arquivo, digite o seguinte código:

```
1  /* USER CODE BEGIN 4 */
2  void HAL_TIM_PeriodElapsedCallback(
       TIM_HandleTypeDef *htim)
3  {
4    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET
         );
5    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4,
         GPIO_PIN_RESET);
6  }
7
8  void HAL_TIM_OC_DelayElapsedCallback(
       TIM_HandleTypeDef *htim)
9  {
10   if( htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1 )
11   {
12     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5,
           GPIO_PIN_RESET);
13     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4,
           GPIO_PIN_SET);
14
15     // muda a largura de pulso do PWM
16     uint16_t ccr1_temp = htim2.Instance->CCR1;
17     ccr1_temp = (ccr1_temp>=3999) ? 0:ccr1_temp+1;
18     __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1,
           ccr1_temp);
19   }
20 }
21 /* USER CODE END 4 */
```

Nosso *PWM* será da seguinte forma: a largura de pulso da PA5 aumentará, enquanto o da PA4 diminuirá. A função HAL_TIM_PeriodElapsedCallback não tem segredo. Na função HAL_TIM_OC_DelayElapsedCallback, como explicamos anteriormente, verificamos qual canal gerou a interrupção. Teríamos um teste para cada canal ativo. Escrevemos as portas *GPIO* de acordo, e após isso, aumentamos do o valor do CCR1 de uma unidade. Quando atingimos o mesmo valor do período do *timer*, voltamos a zero (criando uma onda dente de serra para o *duty cycle* do *PWM*), o que é feito com o operador ternário da linha 17. A variável ccr1_temp deve ser otimizada fora pelo compilador.

Agora, precisamos dar *start* na base do *timer* e no canal 1. Isso é feito no *User Code 2* da seguinte forma:

```
1 /* USER CODE BEGIN 2 */
2 HAL_TIM_Base_Start_IT(&htim2);
3 HAL_TIM_OC_Start_IT(&htim2, TIM_CHANNEL_1);
4 /* USER CODE END 2 */
```

Inicializamos os dois com interrupções, significado do "_IT"no final do nome da função da *HAL*. Como de costume, passamos o *handler* do *timer* por referência.

Agora, temos um programa funcional. Compile e baixe o código (não esqueça de configurar a geração de hexadecimal, bem como a ferramenta de *download* externa). Você deve observar que o brilho do *LED* da porta PA5 aumenta do mínimo ao máximo, enquanto o da PA4 faz o contrário, e isso se repete a cada 2 segundos. Isso conclui nosso exemplo.

6 ADC

6.1 *Overview*

Para um microcontrolador com mais de um ADC, existem dois modos de operação para a captura de dados:

- Modo Independente;
- Modo Duplo/Triplo;

O modo triplo começa a aparecer em microcontroladores mais complexos, enquanto o duplo pode ser encontrado nos microcontroladores da família F1. Estes modos múltiplos permitem a execução dos ADC de forma paralela ou sequencial, efetivamente multiplicando a taxa de amostragem pelo número de ADC instanciados (12MSps no STM32H7, que contém ADCs de 16 bits).

Como os STM32F0x2 têm apenas um ADC, podemos dar uma olhada melhor nos modos de operação individuais.

O ADC pode operar fazendo conversões de um ou vários canais nas seguintes configurações:

- Um canal, conversão simples;
- Um canal, conversão contínua;
- Varredura, conversão simples;
- Varredura, conversão contínua;
- Conversão injetada.

Os dados obtidos pelo ADC são armazenados no registrador do periférico e podem ser obtidos por software ou DMA, seu disparo também é configurável, podendo ser disparado por software ou pelo sinal de interrupção gerado por um timer.

Já que você ja é o bixão do *Timer*, podemos pegar um dos seus códigos de pwm e adicionar um ADC, controlando o brilho de um LED utilizando potenciômetro, por exemplo.

6.2 *ADC* Simples

Comece instanciando o ADC no CUBE, os canais cujos pinos já estão em utilização ficarão vermelhos, impossibilitando-os de serem escolhidos. Escolha então um dos canais em que o pino está sobrando.

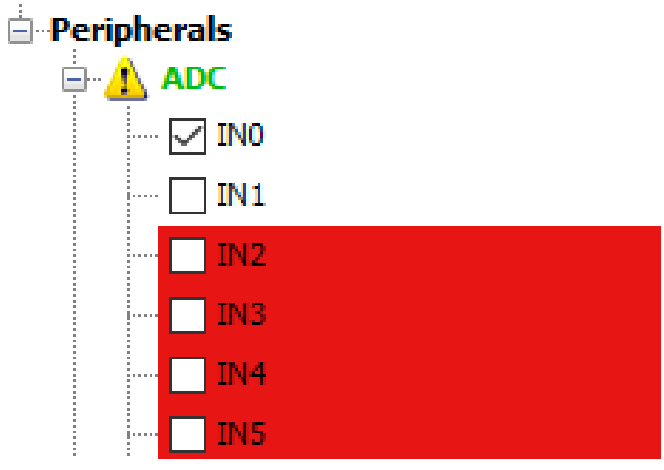

Figura 17: Canais do ADC escolhidos para os exemplos

Na aba de configuração do ADC nos periféricos analógicos, utilize o modo mais simples possível: conversão contínua disparada por software e sem requisição de DMA.

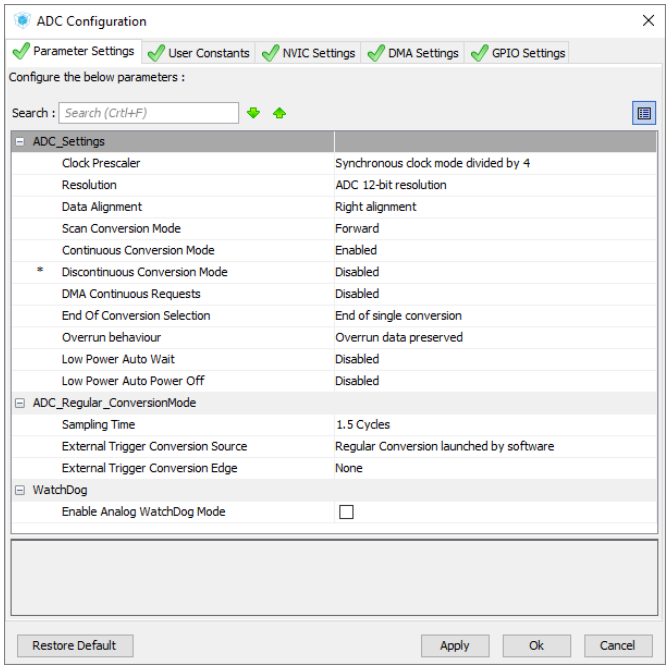

Figura 18: Configuração simples do ADC

Ao gerar o código, adicione tudo necessário para testar seu ADC (no meu caso, o TIM2 operando como PWM no canal 1, contando até 4095). Estamos prontos para fazer uma operação de poll com o ADC.

A operação de poll funciona de tal maneira que você fornece um ADC e um tempo limite para ela e então, a CPU checa continuamente até o tempo acabar se a flag de conversão do periférico foi setada. A operação retorna erro se a conversão for mal sucedida.

Este é o modo mais utilizado em outros microcontroladores, devido a sua simplicidade. É considerado um modo deprecado do uso deste periférico, porém pode vir a calhar em uma aplicação de baixa potência ou se todos os *timers* já tiverem sido instanciados.

Para utilizá-lo, siga o exemplo:

```
/* USER CODE BEGIN 2 */
// --PWM--
HAL_TIM_Base_Start(&htim2);
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);

// --ADC--
HAL_ADC_Start(&hadc);
uint16_t val=0;
/* USER CODE END 2 */

...

/* USER CODE BEGIN 3 */
  if(HAL_ADC_PollForConversion(&hadc, 100)==HAL_OK)
  {
    val= HAL_ADC_GetValue(&hadc);
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1 ,
        val);
```

```
  }
  HAL_Delay(10);
}
/* USER CODE END 3 */
```

A função *HAL_ADC_PollForConversion()* recebe um handler de ADC e um timeout dado em milissegundos, tendo como retorno o código HAL_OK em caso de sucesso. *HAL_ADC_GetValue()* apenas busca o valor no registrador DR (data register) da instância do conversor.

6.3 *ADC* Disparado por *Timer*

Veremos agora como dispará-lo por um timer. Voltamos para o Cube, adicionamos um timer qualquer (indicando o clock interno como a fonte de clock), como contador simples, para isso devemos mantê-lo em contagem ascendente até o número que queremos contar e com o Update Event na seleção de evento do trigger, assim nosso timer vai se comportar apenas como um temporizador.

Estarei utilizando o timer 3 contando até 4799, para obter uma frequência de amostragem de 48kHz.

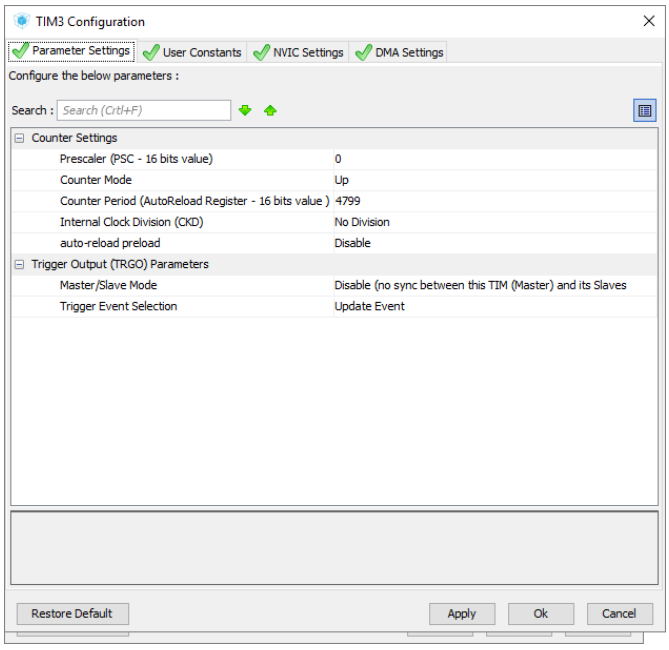

Figura 19: Configuração do timer como temporizador de 48kHz

Para o ADC, é necessária a conversão descontínua disparada pelo *trigger out* do timer 3.

O princípio de funcionamento é o mesmo, mas não gasta processamento para a espera da conversão. Assim que o timer sobe sua flag de trigger out, o periférico entende que uma conversão deve ser iniciada, quando a operação é concluída e o resultado é mantido no DR.

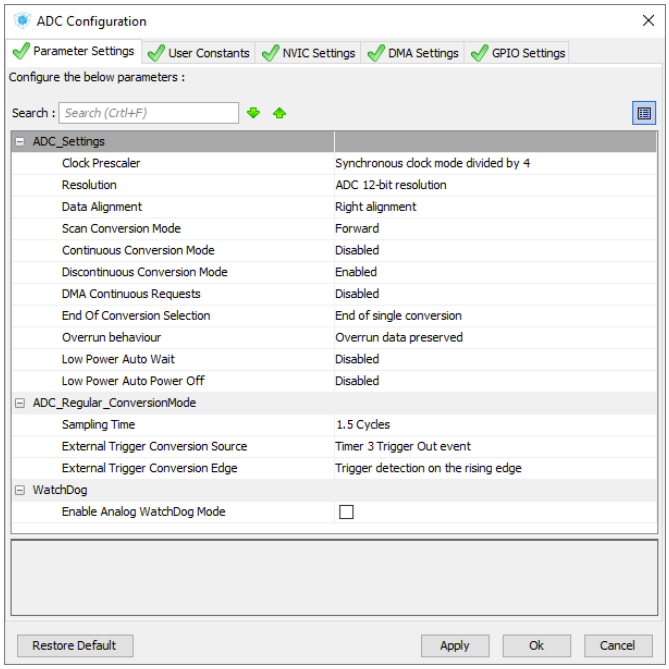

Figura 20: Configuração do ADC para disparar com o timer

O código fica:

```
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start(&htim2);
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
HAL_ADC_Start(&hadc);
HAL_TIM_Base_Start(&htim3);
uint16_t val=0;
/* USER CODE END 2 */

...

/* USER CODE BEGIN 3 */
  val= HAL_ADC_GetValue(&hadc);
  __HAL_TIM_SET_COMPARE(&htim2,TIM_CHANNEL_1,val);
  HAL_Delay(10);
}
/* USER CODE END 3 */
```

Obviamente não estamos fazendo proveito dos 48kSps, isso é devido ao delay que foi colocado dentro do laço de *while*, para tirar proveito da taxa de amostragem que setamos, devemos ser criativos. Nesse modo, soluções podem envolver rotear o tigger para o CCR e temporizar o laço de while usando o update, ou até mesmo recuperar o valor do DR dentro de uma interrupção de update.

6.4 *ADC, Timer* e *DMA*

Outra maneira que podemos explorar, nos dando maior flexibilidade é utilizando o DMA. O acesso direto a memória, quando utilizado com o ADC, tem o comportamento de guardar as n últimas amostras para um vetor

de n posições. O DMA guarda os dados por instância de ADC, ou seja, caso esteja sendo utilizado um dos métodos de multiplexação de canais, os dados retidos terão a ordem que foram obtidos.

Este método se torna útil em aplicações como data-logging e osciloscópios, ao se utilizar um grande vetor de dados que contém as últimas conversões. Para aquisições simultâneas, é recomendado que o tamanho do vetor seja um múltiplo inteiro do número de canais alocados [n], assim cada canal terá posições fixas dentro do vetor, se repetindo a cada n índices.

Então vamos lá, fazer a operação da mudança do duty cycle com apenas uma instrução de máquina. Para começar, habilite a opção 'DMA Continuous Requests' na janela do ADC e então,habilite o DMA no item de DMA ou na aba de DMA dentro do ADC (elas levam para o mesmo lugar essencialmente).

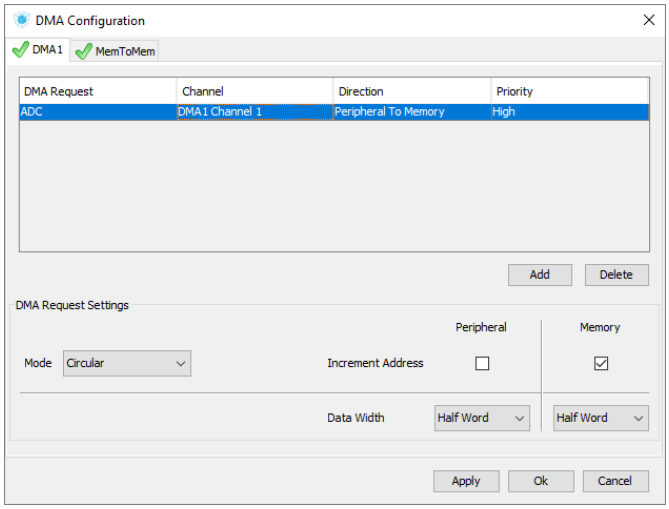

Figura 21: Configuração do DMA para operar com o ADC

Nosso DMA estará operando no modo periférico -> memória, microcontroladores com dois DMAs terão um para cada sentido dessa rota. Como o ADC é de 12bits, precisamos alocar variáveis de 16bits apenas, então a largura dos dados deve ser setada para meia word na memória e no periférico, vamos incrementar apenas o endereço do vetor da memória, e o modo deve ser circular, garantindo que ao chegar no fim dos índices do vetor, o próximo índice é 0, assim nunca parando de contar.

No código fazemos as seguintes modificações:

```
/* USER CODE BEGIN PV */
/* Private variables ----------------------*/
uint16_t adVal[1]={0};
/* USER CODE END PV */

...

/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start(&htim2);
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
HAL_ADC_Start_DMA(&hadc, (uint32_t*)adVal, 1);
HAL_TIM_Base_Start(&htim3);
/* USER CODE END 2 */

...

/* USER CODE BEGIN 3 */
  __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1 ,
      adVal[0]);
  HAL_Delay(1);
}
/* USER CODE END 3 */
```

Ao bater o olho nesse código, podemos fazer as seguintes observações:

- Como não estamos em um laço temporizado, temos que dar um jeito de não deixar o *set compare* sozinho, se isso acontecer, o timer não tem tempo de fazer suas instruções até a próxima operação;
- Declaramos um vetor de uma posição pois não temos interesse nos valores passados do brilho do LED, ao menos que façamos uma média com eles;
- Declarar um vetor de uma posição nesse caso, pode ser substituído por uma declaração simples de variável, contanto que no *ADC Start DMA* seja fornecido o endereço dessa variável.

7 UART

7.1 *Overview*

O nosso microcontrolador tem dois periféricos independentes de USART. O módulo USART pode ser setado como síncrono ou assíncrono, no caso do modo assíncrono, os padrões de comunicação mais utilizados são o RS232 e RS485.

7.2 Caso Mais Utilizado

Como exemplo, podemos usar o modo padrão de RS232, tendo um *stop bit*, sem paridade e *handshake* e uma taxa de 115200 baud. O periférico utilizado é o USART2, pois o primeiro entra em conflito com a pinagem do USB, impossibilitando sua utilização se desejada.

Acessando a aba de configuração e selecionando a USART2, podemos alterar as configurações do periférico.

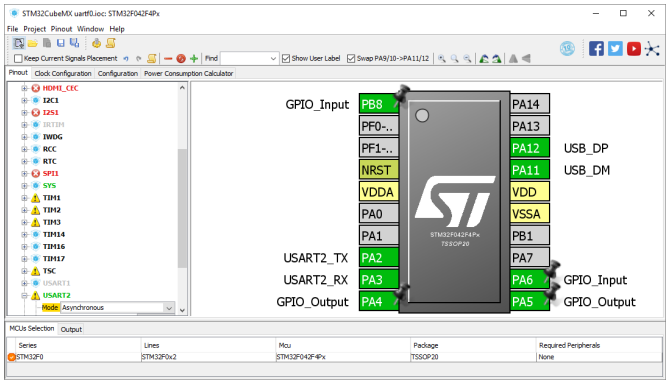

Figura 22: Pinagem da USART2 como UART

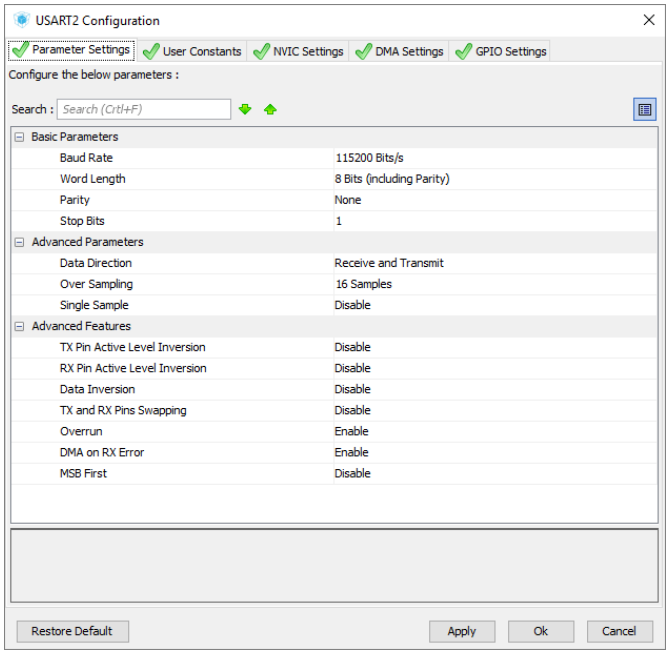

Figura 23: Configuração padrão da UART

Gerando o código, podemos criar uma *string* de teste:

```
/* USER CODE BEGIN PV */
/* Private variables ---------------------*/
char buf[] = "isso eh um teste\n";
/* USER CODE END PV */
  ...
/* USER CODE BEGIN 3 */
  HAL_UART_Transmit(&huart2, buf, sizeof(buf), 100)
      ;
  HAL_Delay(500);
}
/* USER CODE END 3 */
```

A função *HAL_UART_Transmit* recebe como entrada o *handler* da UART, o ponteiro para o início do buffer que contém a mensagem, o tamanho da mensagem em bytes e o *timeout* da operação.

Para realizar o teste, é possível usar um conversor USB-Serial como o FT232 ou o CH340G, ou até um Arduino devidamente configurado.

Para receber dados, devemos usar a função de *Receive* da UART, usando os mesmos parâmetros.

```
/* USER CODE BEGIN PV */
/* Private variables ---------------------*/
char buf[20];
/* USER CODE END PV */
  ...
/* USER CODE BEGIN 3 */
  HAL_UART_Receive(&huart2, buf, sizeof(buf), 100);
  if (buf[0]>0)
    HAL_UART_Transmit(&huart2, buf,bsizeof(buf)
        ,100);
  for (int i=0; i<sizeof(buf); i++)
    buffer[i]=0;
}
/* USER CODE END 3 */
```

Nesse caso, estamos repetindo o que nos é mandado na UART, nota-se que é importante apagar o buffer inteiro, desta maneira não é preciso se preocupar com o aparecimento de caracteres 'fantasma'.

8 USB

8.1 *Overview*

O STM32F042, por ser da linha USB da STM32F0, já possui o periférico de USB, podendo assumir várias classes (o Cube gera COM, HID, Audio e DFU) nativamente. Seu padrão é o USB Full-Speed de 48MHz, obedecendo as normas do padrão USB 2.0.

8.2 Configurações Obrigatórias no Cube

Primeiramente, temos que setar a flag Swap PA9/PA10 -> PA11/PA12 no Cube, então podemos ativar o USB na aba de periféricos (Quando ligado, na pinagem deve aparecer PA11 e PA12 como USB_DM e USB_DP respectivamente, como na figura 22). O clock também é muito importante que seja setado corretamente, no caso do F0x2 deve-se usar o HSI48.

Para selecionar a classe de USB, selecionamos na aba USB_DEVICE.

8.3 COM

A classe de comunicação, mais conhecida como porta COM, é a classe responsável pela emulação de uma porta RS-232 no computador.

Essa é a classe mais simples de todas, tendo a necessidade de alterar poucos parâmetros para ser utilizada.

Provavelmente você não pode gerar o código com os valores padrões do Cube, isto é, um vetor de envio e recebimento de dados de 2kB. Isso é causado pela pouca memória do dispositivo, como ele só tem 6kB no total, não sobra muita coisa. Como exemplo, setei os tamanhos do buffer de RX e TX para 512B, nada impede de se usar tamanhos diferentes.

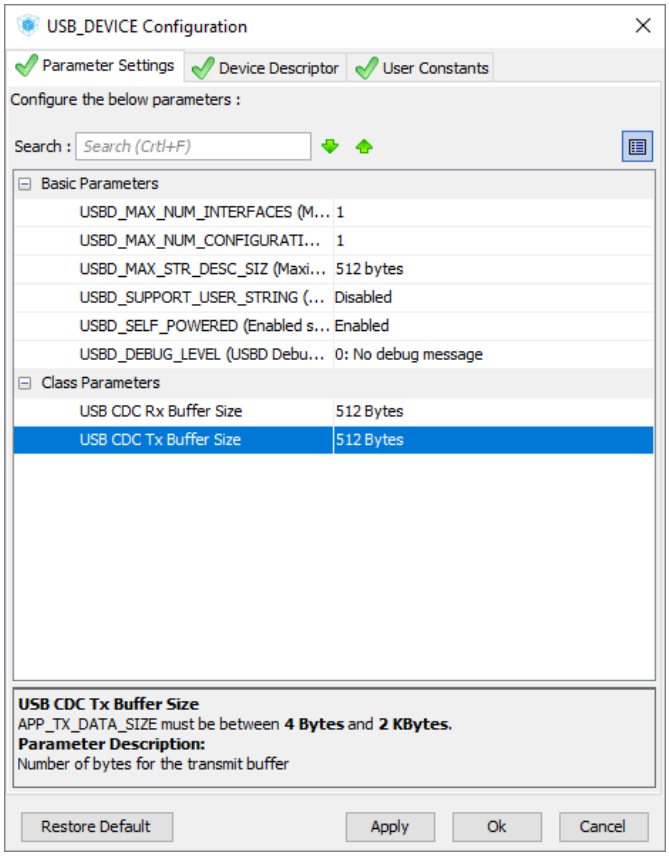

Figura 24: Janela de parâmetros da porta COM

Os dados como nome do dispositivo são por padrão setados como STM32 VirtualComPort. Para alterar esses valores, clique no USB_DEVICE dentro da seção de Middlewares na aba da configurações.

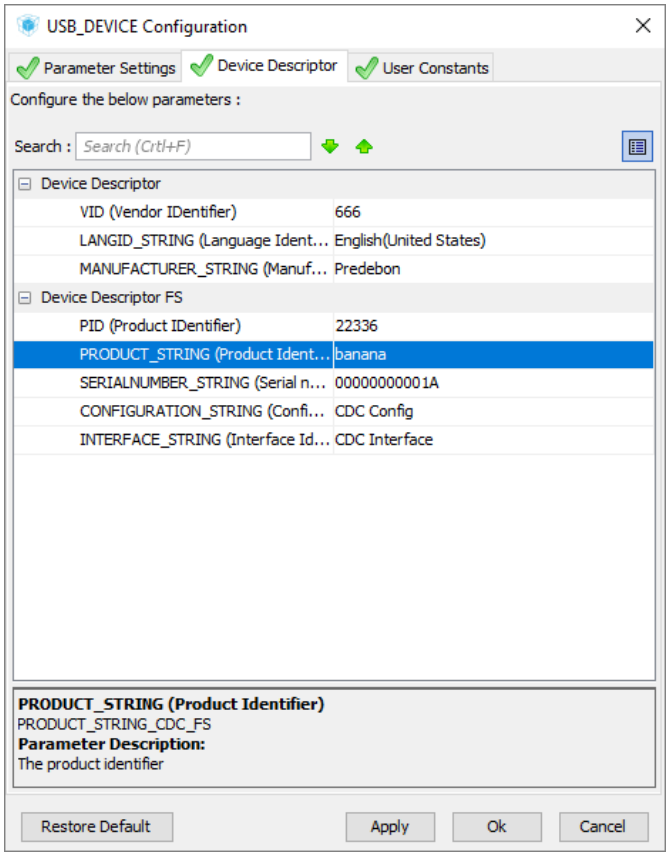

Figura 25: Janela de descrição da porta COM

Indo para o editor, abrimos o main.c e logo de cara ja temos que fazer a seguinte edição:

```
/* USER CODE BEGIN Includes */
#include "usbd_cdc_if.h"
/* USER CODE END Includes */
```

Isso é necessário pois o Cube não gera a dependência dos middlewares da USB no código principal.

Assim como a UART, não iremos entrar em detalhe de como é feito a low level esse protocolo. Para enviar dados é só utilizar a função CDC_Transmit_FS(), ela recebe o ponteiro de início e o tamanho do buffer de dados a serem enviados.

Para enviar uma string de teste, fazemos o seguinte:

```
/* USER CODE BEGIN PV */
/* Private variables ----------------------*/
uint8_t buf[]="atireiopaunogato\n";
/* USER CODE END PV */
 ...
/* USER CODE BEGIN 3 */
  CDC_Transmit_FS(buf, sizeof(buf));
  HAL_Delay(1000);
}
/* USER CODE END 3 */
```

Caso você tenha feito tudo corretamente, uma mensagem informando que seu dispositivo está sendo confi-

gurado deverá aparecer (assumindo que você esteja no Windows). Caso o dispositivo não tenha sido reconhecido, algo está errado.

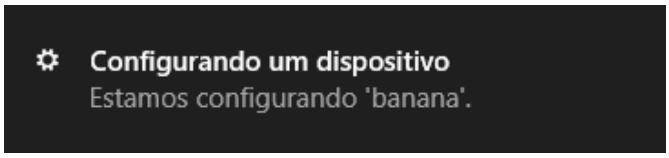

Figura 26: Aviso padrão do Windows

Para efetuar o teste, pode-se utilizar qualquer terminal serial. Não se preocupe com a configuração do pacote de dados do terminal, o protocolo de Virtual COM já cuida destes detalhes.

Agora que você domina a arte de enviar caracteres ASCII pela USB, você deve aprender a recebê-los.

Se seguirmos a função *CDC_Transmit_FS()* podemos ver que existe de fato uma *CDC_Receive_FS()*, porém não pode ser utilizada pelo usuário pois é chamada por outras funções. Ao procurar todas as instâncias desta função, podemos observar que é utilizada pela estrutura *USBD_Interface_fops_FS* que é a interface responsável pela inicialização, de-inicialização, controle e recebimento dos dados da COM.

Assim sendo, ao receber um dado pela COM, o microcontrolador já o manda para o vetor *UserRxBufferFS*. Como o vetor de RX faz parte do usbd_cdc_if.c, temos que o importar e redefinir o tamanho do buffer para o main().

```
/* USER CODE BEGIN PV */
/* Private variables ---------------------------*/
#define APP_RX_DATA_SIZE 512
extern uint8_t UserRxBufferFS[APP_RX_DATA_SIZE];
char buf[]="atireiopaunogato\n";
/* USER CODE END PV */
```

Para utilizar o *buffer* fica a seu critério. Como exemplo vou repetir tudo que me é mandado na COM:

```
/* USER CODE BEGIN 3 */
  if (UserRxBufferFS[0]){
    int size=strlen((const char*)UserRxBufferFS);
    CDC_Transmit_FS(UserRxBufferFS, size);
    for (int i=0; i!=size; i++)
      UserRxBufferFS[i]=0;
    HAL_Delay(1000);
  }
}
/* USER CODE END 3 */
```

Primeiro verificamos se a primeira posição do vetor não é nula, isso garante que há pelo menos um caractere querendo ser enviado, o *size*, mede o tamanho da string para que possamos transmitir só o que recebemos e o laço *for* zerando cada posição, garante que não haverá lixo no *buffer* para a próxima transmissão.

8.4 HID

O *Human Interface Device* é um protocolo responsável pela comunicação do computador com diversos dispositivos, sendo teclados, mouses e gamepads exemplos de suas aplicações.

A STM foi muito legal e deixou um exemplo de mouse para brincarmos, vamos explorá-lo!

Começaremos gerando um código base para HID. Assim como na seção anterior, gere a configuração padrão para o USB:

- HSI48 como *clk*;
- Troca da PA9/PA10 pela PA11/PA12;
- Ativado o USB como *Device*.

No campo *USB_DEVICE*, você vai escolher *Human Interface Device*. Selecionando essa opção, já são gerados as tabelas de descrição do dispositivo (mouse).

Caso selecionado o *Custom Human Interface Device*, essas tabelas não serão geradas, assim é necessária uma ferramenta para gerar a tabela.

Gerando o código, temos que incluir no main() a *usbd* como na COM.

```
/* USER CODE BEGIN Includes */
#include "usbd_hid.h"
/* USER CODE END Includes */
```

Chegamos a um ponto que o protocolo limita nossas possibilidades. A maneira mais simples de operação do HID é por *reports*. Ao mandar um report, o *Host* USB processa o quadro de maneira definida nas tabelas de configuração.

Para as tabelas geradas pelo código padrão, temos que respeitar a seguinte características:

- Enviar um pacote de tamanho constante contendo informação sobre botões, eito x e y do cursor e a roda do mouse;
- Tamanho de 32b;
- Modelo incremental do ponteiro e roda do mouse.

Para implementar isso, definimos uma estrutura que contenha as seguintes características:

```
struct HID_t {
  uint8_t buttons;
  int8_t x;
  int8_t y;
  int8_t wheel;
  }mouse;
```

A primeira variável contém a informação dos botões pressionados, os bits mais importantes desta são os três menos significativos, sendo eles os botões esquerdo, direito e central.

As variáveis x e y e wheel são valores sensitivos a sinal cuja amplitude determina o quanto será alterado desses valores para o mouse. Por exemplo, se for enviado um *report* com x=10, o ponteiro do mouse deve mexer em 10 pixel no sentido positivo do eixo x, se for enviado y=-25, o ponteiro retrocede 25 pixel no eixo y, assim acontece com a direção de rolagem da tela, controlada pelo wheel.

```
/* USER CODE BEGIN 2 */
struct HID_t {
  uint8_t buttons;
  int8_t x;
  int8_t y;
  int8_t wheel;
}mouse;

mouse.buttons = 0;
mouse.x = 10;
mouse.y = 0;
mouse.wheel = 0;
/* USER CODE END 2 */

...

 /* USER CODE BEGIN 3 */
  USBD_HID_SendReport(&hUsbDeviceFS, (uint8_t*)&
      mouse, sizeof(mouse));
  HAL_Delay(100);
}
/* USER CODE END 3 */
```

Ao compilar e testar, você verá que o mouse se arrastará para direita 10 pixel a cada 100ms.

Vamos só adicionar um botão:

```
/* USER CODE BEGIN 3 */
  int bot=HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_6);
  mouse.buttons =bot;
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4, bot);
  USBD_HID_SendReport(&hUsbDeviceFS, (uint8_t*)&
      mouse, sizeof(mouse));
  HAL_Delay(10);
}
/* USER CODE END 3 */
```

Lembrando que deve se adicionar uma porta com *pull-up* para ser usada como botão. Expandindo o conceito para utilizar interrupções e ADCs, podemos chegar a um código bem mais útil.

Parabéns, você já possui o conhecimento necessário para fazer um super mouse gamer com muitos leds e botões que ninguém sabe pra que serve.

Para explorar novas possibilidades, dê uma olhada nos nossos repositórios e na pasta do curso no Google Drive.

9 Links

9.1 Repositórios Git

- Predebon:
  - `github.com/predebon` (não tem muita coisa, mas eventualmente eu coloco algo lá)
- Wesley:
  - `github.com/AvatarBecker` (idem)

9.2 Google Drive

- Pasta do curso:
  - Drive de 2018: `https://drive.google.com/drive/folders/1GFLthlYzQOJv8MzuE3Df1NK-lfDDhVEj?usp=sharing`
    - Contém documentação, programas de exemplo, instaladores de *software* e uma surpresa agradável...

10 Pinout

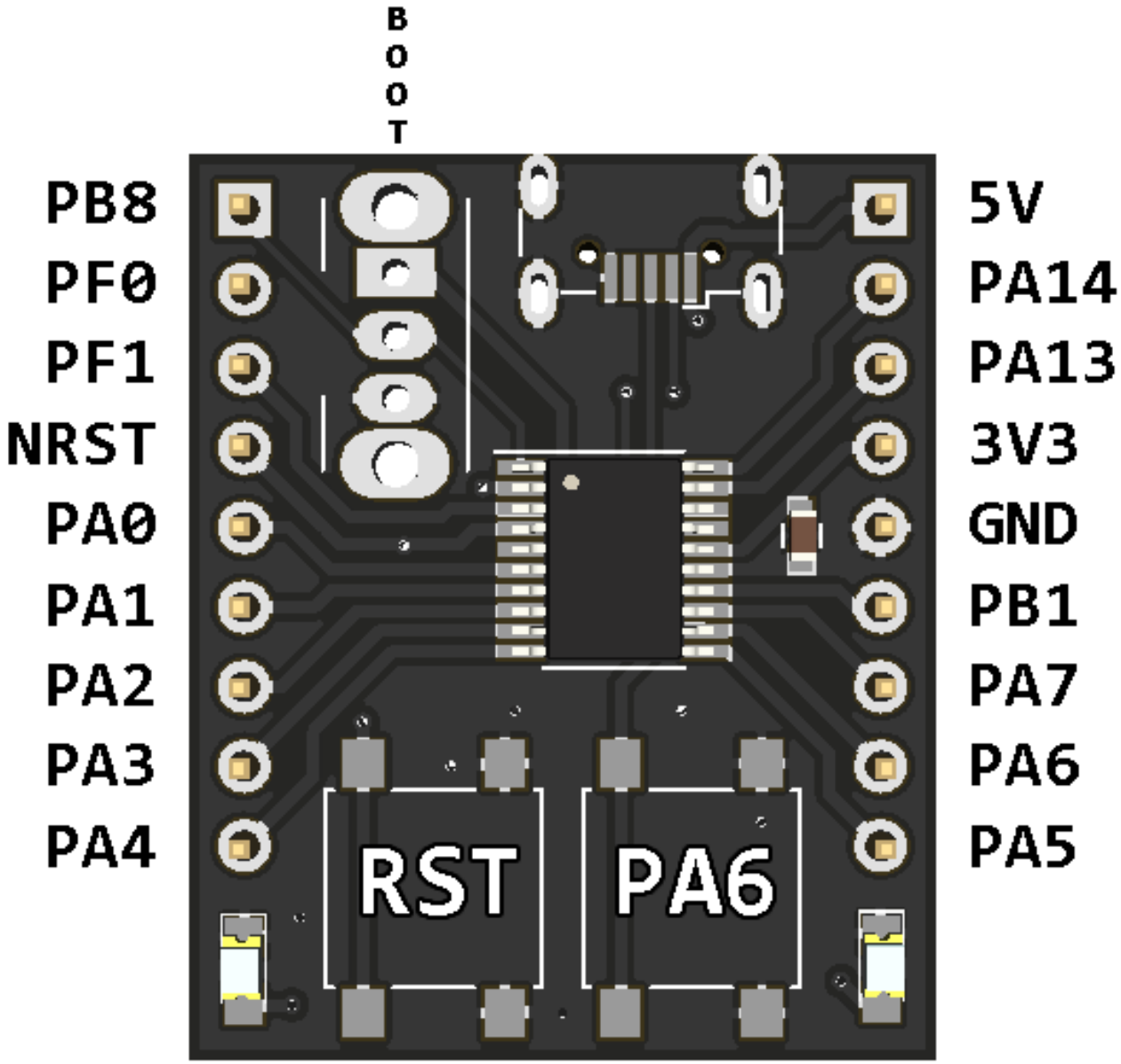

Figura 27: Pinagem para o módulo capivara de 2019