UNIVERSIDADE FEDERAL DE SERGIPE
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
GRUPO DE PESQUISA EM INSTRUMENTAÇÃO

MANUAL DE SOFTWARE
PLACA DE CONTROLE BASEADA EM STM32F103C8

Ruan Robert Bispo dos Santos

Abril, 2020.

RESUMO

Neste manual são listados os primeiros passos com o microcontrolador stm32f103c8, levando em consideração a plataforma STM32CubeIDE, a fim de simplificar a utilização inicial da placa desenvolvida. Também são listados alguns exemplos de implementação de código base, os quais encontram-se disponíveis no GitHub em formato Open Source no *link* abaixo. Através da utilização deste manual, em conjunto com os códigos e comentários disponibilizados, é possível ter uma visão mais completa sobre o escopo da plataforma.

https://github.com/RuanBispo/STM32F103C8_CUBEIDE_PLATAFORMA

Sumário

FIGURAS

1 CONFIGURAÇÃO DO PROJETO

A partir da configuração básica na plataforma é possível desenvolver diversas aplicações, sejam elas com foco na placa desenvolvida neste trabalho ou apenas com o uso da plataforma stm32f103c8. Sendo assim, o primeiro passo é fazer o download da STM32CubeIDE no próprio site da fabricante, entrando no link a seguir:

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

Após a instalação do programa é necessário criar um *workspace* padrão, que pode ser modificado depois. É importante direcionar o local para uma pasta que possa ser acessada facilmente, pois o programa salvará novos projetos em subdiretórios criados no *workspace* por padrão.

Figura 1: Workspace

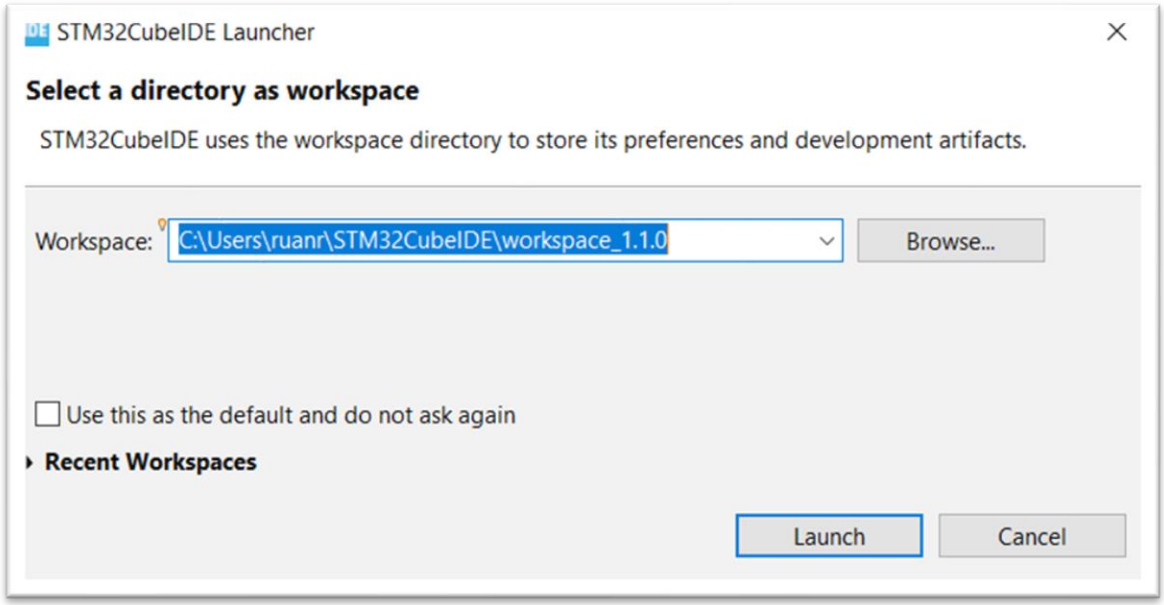

Figura 1: Workspace

O segundo passo que deverá ser executado sempre que forem criados novos projetos é a seleção da placa utilizada, que nesse caso é a stm32f103c8. Ao clicar em novo projeto (New->STMProject), é possível ver todas as famílias de microcontroladores da fabricante, sendo necessário fazer a seleção da plataforma base. Assim, basta digitar o nome do componente na barra de busca *Part Number Search*, no lado esquerdo da janela de inicialização do programa, que será filtrada a família desejada. Como é possível observar na Figura 2, são mostradas algumas informações da placa como o *datasheet*, diagrama de blocos, faixa de preço, dentre outras informações.

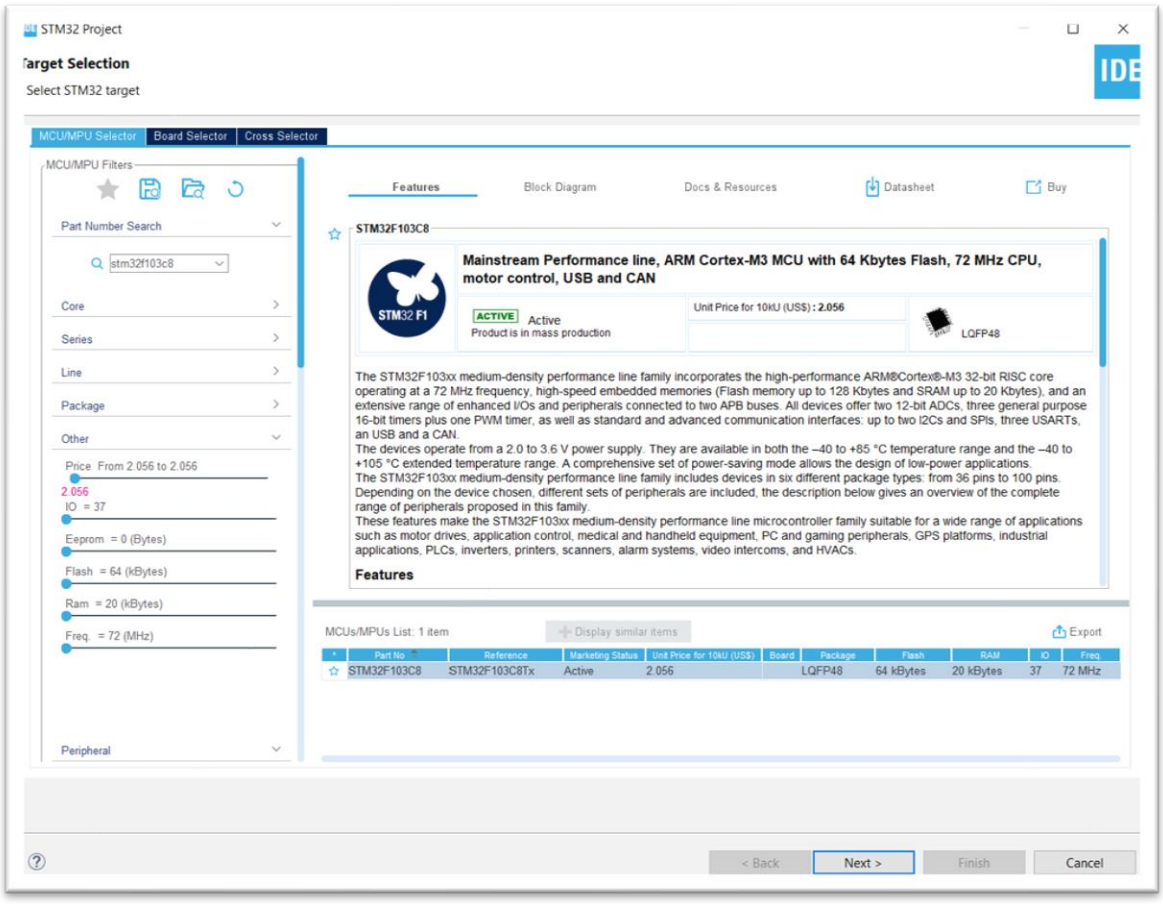

Figura 2: Menu de seleção de microcontrolador

Uma vez realizada a escolha da placa, selecione o botão *Next* no lado direito inferior na janela para prosseguir para as configurações relacionadas à linguagem utilizada e informações iniciais relativas à arvore do projeto. Vale salientar que toda vez que um novo projeto é criado é necessário passar por esta etapa, o que é essencial para que todo o restante da programação funcione bem. A próxima etapa de configuração pode ser ilustrada pela Figura 3.

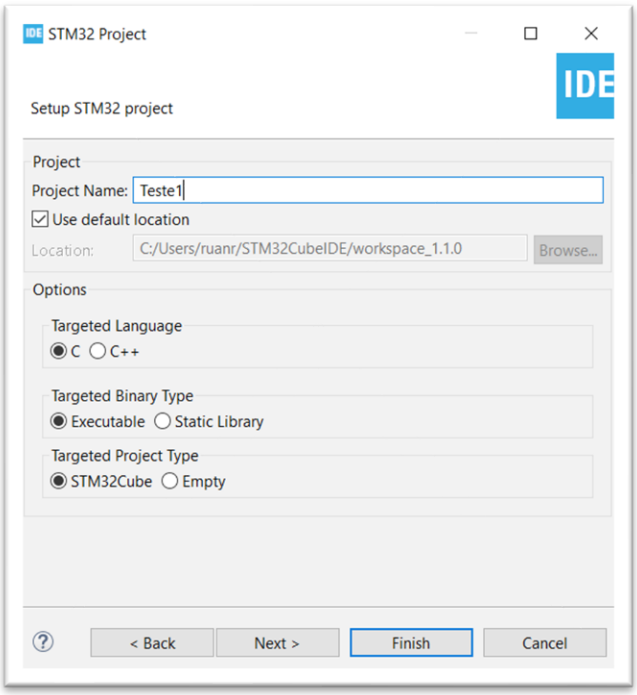

Figura 3: Configuração de projeto

Após realizar as configurações como descrito na Figura 3 e selecionar um nome para o projeto clique em *Finish* para que o programa inicie a etapa de configuração do ambiente de trabalho que será utilizado para programar a plataforma.

Finalizada essa etapa, a IDE abrirá uma janela com as configuração do projeto, as quais são realizadas através do arquivo .ioc, o que permite uma configuração de forma mais visual e simplificada do processo, auxiliando no aprendizado.

Como primeiro passo para todo projeto, é necessário configurar a frequência do *clock*, que para todos os exemplos ilustrados neste manual será utilizada a frequência máxima da plataforma, equivalente a 72MHz. Para isso, na aba *System Core*, selecione a opção de RCC e logo após configure como *High Speed Clock* para *BYPASS Clock Source.* A Figura 4 ilustra como a configuração deve ser realizada.

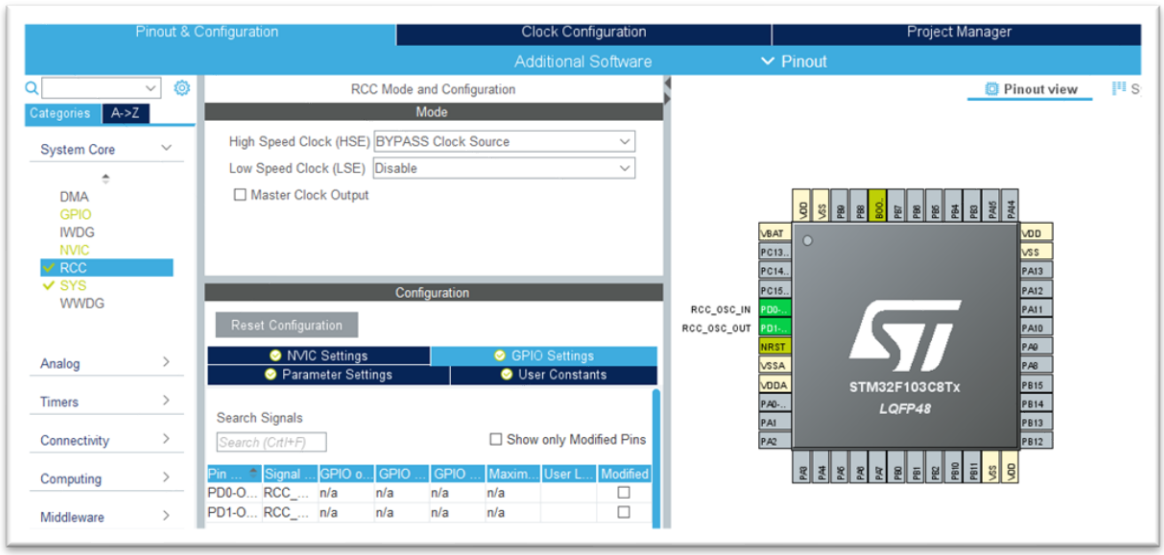

Figura 4: Configuração de clock

Em sequência, selecione a opção SYS, dentro da mesma aba System Core, para definir a forma de Debug. Até o momento a única forma de gravar no microcontrolador é executando um debug com o código desenvolvido e, para isso, basta seguir as instruções Figura 5, selecionando o modo *Debug* para *Serial Wire*. Por ser uma plataforma relativamente nova, algumas funções da IDE não estão completamente fechadas.

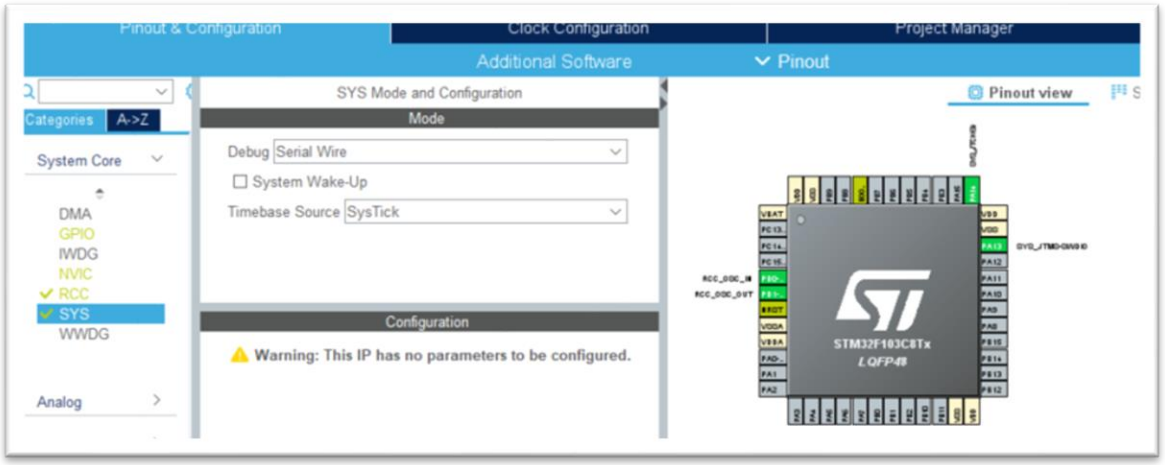

Figura 5: Configuração do Debug

Realizadas essas etapas, é necessário apenas configurar a frequência do *clock*. Utilizando como base essa configuração, é possível desenvolver quaisquer outros projetos descritos neste manual. É importante saber, também, que é possível alterar as configuração realizadas através de um duplo clique no arquivo .ioc criado na pasta raiz do projeto.

Como etapa final para a configuração básica, é necessário selecionar a aba *clock configuration* na parte superior da janela de configurações. Deste modo será aberto um diagrama que ilustra de forma simples o funcionamento do *clock* do STM, sendo possível alterar as configurações de forma visual.

Para os exemplos base deste manual será utilizada a configuração exemplificada na Figura 6, em que é preciso apenas mudar o valor da caixa HCLK para 72 e pressionar *enter*. Assim, o próprio programa encontrará configurações entre os comparadores para que a frequência se ajuste ao desejado.

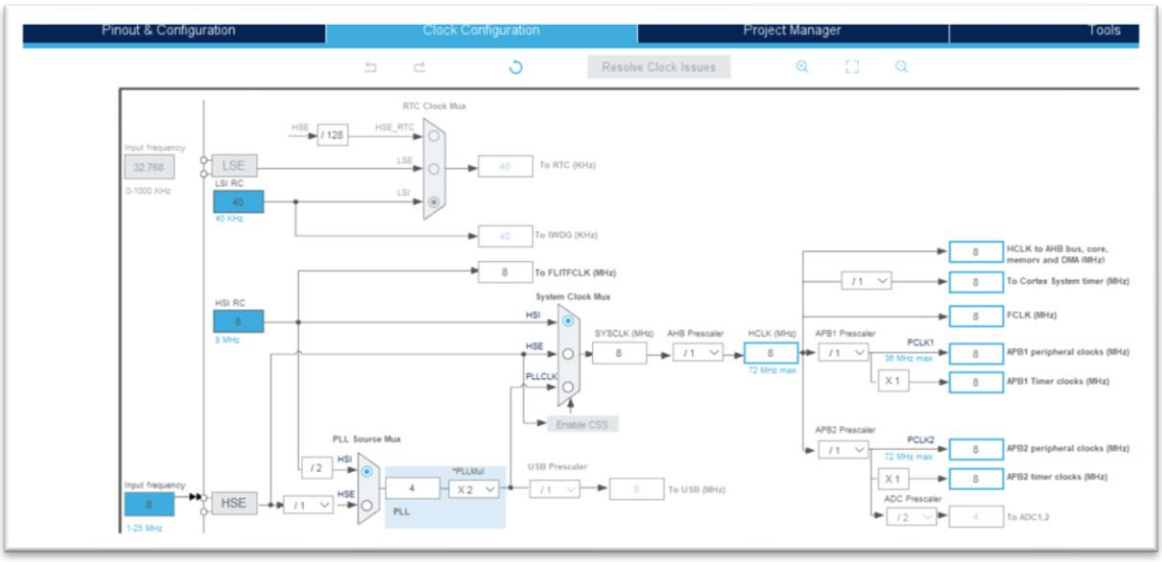

Figura 6: Diagrama de comparadores

Realizados todos os procedimentos, selecione o menu *File->Save All*, ou salve todos os arquivos através do atalho Ctrl+Shift+S. Por fim, será configurado o projeto e gerada a main.c que poderá ser utilizada para programação dos códigos. Como organização do código, é utilizado o padrão de salvamento em árvore que utiliza como repositório dos arquivos de extensão .c a pasta Src e para arquivos .h a pasta Inc.

É muito importante ressaltar que **sempre** que o arquivo .ioc é modificado e salvo, a *main* é atualizada e são gerados os códigos para configuração. Assim, se algum código desenvolvido pelo usuário não estiver nas seções especificadas pelo programa como permitidas para o programador (código conta com áreas comentadas para o usuário) o código é apagado e perdido.

Em todos os exemplos ilustrados neste manual foi utilizada a linguagem C para a programação, no entanto, é possível fazer a codificação em C++ também na mesma IDE.

2 COMUNICAÇÃO USB

É importante ressaltar que o stm32 simula uma porta COM, mas não temos controle sobre a velocidade de atualização dos dados como em uma porta serial real, então as configurações de taxa de transmissão são iguais para qualquer configuração. Isso significar que uma configuração de 9600 e 115200 bits por segundo reagem à mesma velocidade.

Outro ponto importante é que, sem a solicitação do computador para recebimento dos dados do microcontrolador, é necessário reduzir o comprimentos do buffer de recebimento, pois, como a taxa de envio do microcontrolador é mais alta que a taxa de leitura dos dados recebidos (para este exemplo é utilizado o MATLAB 2019a), os dados enchem o *buffer* e isso resulta em uma impressão com atraso na visualização dos dados. As subseções a seguir são dedicadas a explicar como configurar a comunicação USB no stm32f103c8.

Inicialmente, será mostrado como configurar o stm32 utilizando a abordagem de solicitação de pacote, em que é enviado um bit de dados do computador para o microcontrolador para ser utilizado como solicitação de dados do microcontrolador. Assim, quando o microcontrolador recebe a solicitação, é ativada uma interrupção que se encarrega de enviar os dados de interesse de volta para o computador através de uma COM.

2.1 CONFIGURAÇÕES

Como primeiro passo do processo de configuração da comunicação USB, desde que já tenham sido feitas as configurações de *clock* iniciais, é realizada a configuração das portas do microcontrolador através da aba *Pinout Configuration* da plataforma STM32CubeIDE, presente no menu *Categories*. Assim, através da sub-aba *Connectivity*, como ilustrado na Figura 7, abra o item *USB* e marque a opção de *Device(FS)*.

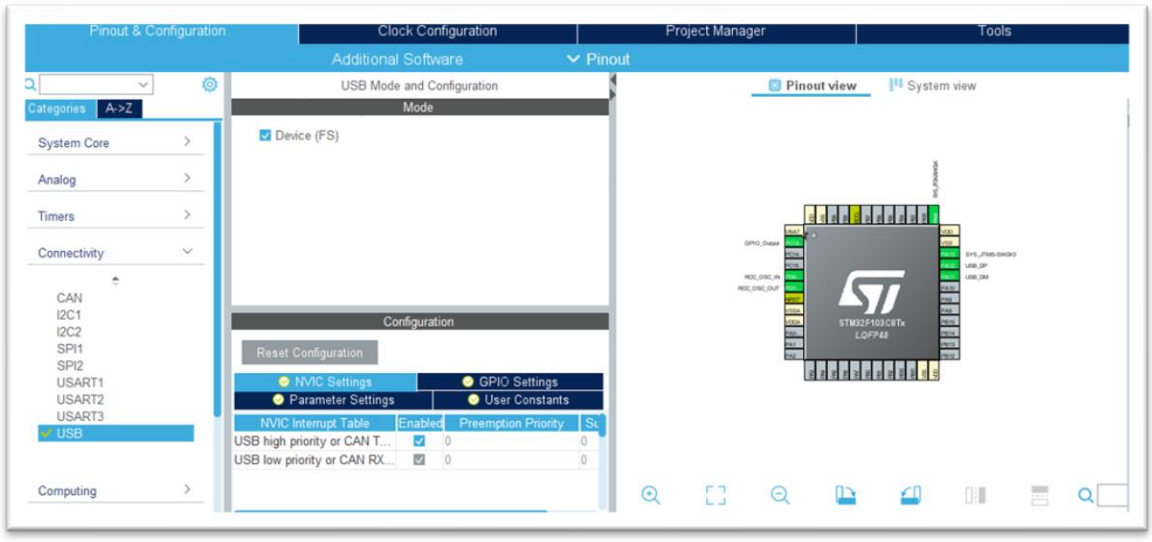

Figura 7: Configuração USB

Em seguida, volte para o menu *Categories* e abra a sub-aba *Middleware*, clique na opção *USB-DEVICE* e ative o modo CDC(*Communication Device Class*) para a opção de *Class For FS IP*. Por fim, abra a aba *Clock Configuration* e modifique a caixa de parâmetro de velocidade no USB para 48 MHz, que é a taxa máxima de transmissão USB, e dê *enter*, de forma similar ao exemplo de configurações básicas de *clock*.

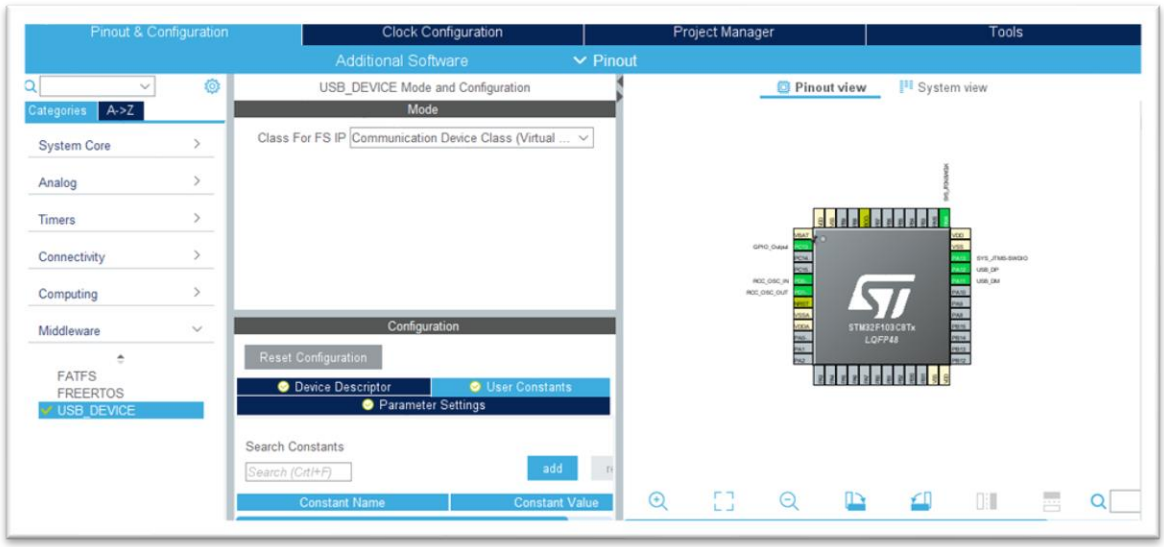

Figura 8: Configuração de Modo USB

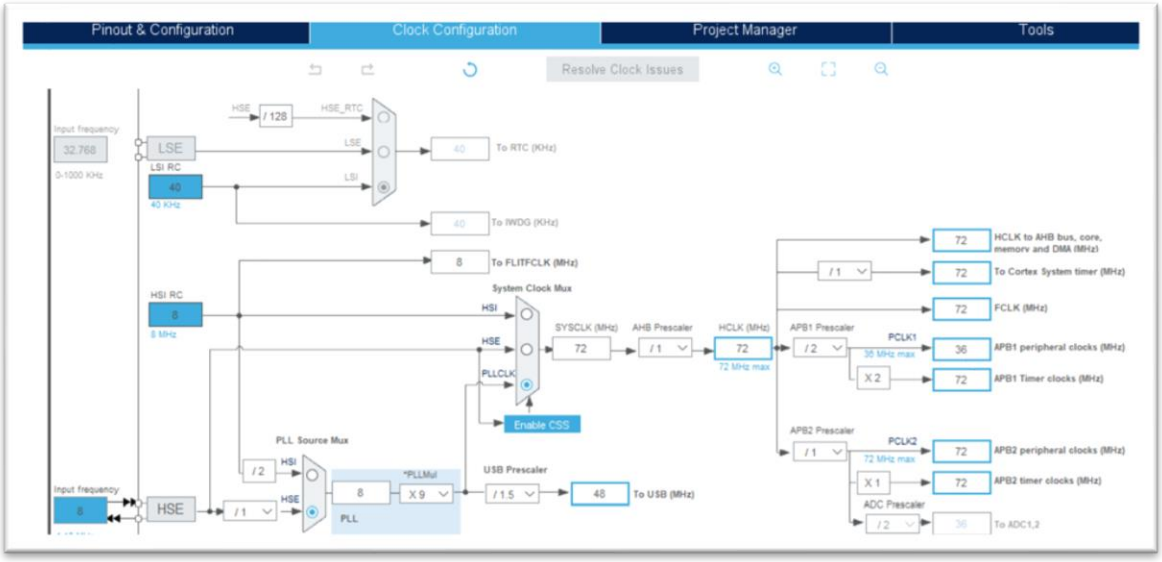

Figura 9: Definição de clock

A plataforma reajustará os comparadores para que a velocidade seja permitida. É importante citar que algumas vezes o sistema não encontra uma solução de configurações para a velocidade citada e, nesse caso, é preciso fazer o ajuste de forma manual. Feito isso, basta salvar o arquivo e começar a codificar a partir das funções criadas na pasta /USB-DEVICE contida no novo projeto.

2.2 CÓDIGO MICROCONTROLADOR

Realizada a etapa de configuração será gerado um arquivo ”usbd-cdc-if.c” na pasta USB-DEVICE/App, no qual está contida a função base ”CDC-Receive-FS()”, que é acionada toda vez que ocorre o recebimento de um pacotes pelo microcontrolador. Logo, a ação desejada deve ser implementada diretamente no arquivo descrito. É importante ressaltar que essa função é diferente da função ”CDC-Transmit-FS()”, que pode ser chamada na ”main.c” para transmitir os dados.

Assim, para que os dados sejam transmitidos toda vez que o microcontrolador receber o pacote de solicitação do computador, uma rotina é implementada dentro da função ”CDC-Receive-FS()”. Em casos gerais, adicionar a função ”CDC-Transmit-FS()” dentro da função ”CDC-Receive-FS()” já cumpre com a tarefa de mandar os dados de volta para o computador solicitante.

Os dados podem ser enviados na forma de *string* ou inteiro, sendo lidos, respectivamente, pelas funções ”fscanf” e ”fread” do MATLAB. Para o primeiro caso, configurando no formato *string*, o MATLAB identifica o final de um pacote pelo *terminator* "\n" para quebra de linha e "\0" para indicar a finalização da *string*, que devem ser adicionados ao pacote antes de enviar pelo microcontrolador. O código da Figura 110 ilustra um exemplo feito com um contador.

```
static int8_t CDC_Receive_FS(uint8_t* Buf, uint32_t *Len)
{
  /* USER CODE BEGIN 6 */
  USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]); //Padrão da função de recebimento
  USBD_CDC_ReceivePacket(&hUsbDeviceFS);        //Padrão da função de recebimento


////////////////////////////////////////////////////////////////////////////////////////////
// VARIÁVEIS UTILIZADAS
////////////////////////////////////////////////////////////////////////////////////////////
  static uint8_t cont = 0;      //Contador de 8 bits para testar comunicação
                                //UserTxBufferFS[1000] é a variável padrão do buffer

////////////////////////////////////////////////////////////////////////////////////////////
// EX1: ENVIA DADOS DO CONTADOR ASSIM QUE RECEBE SINAL DO PC (ECO)
////////////////////////////////////////////////////////////////////////////////////////////

          cont = cont + 1;                        //Incrementa contador
          itoa(cont,UserTxBufferFS,10);           //Converte de inteiro para
          uint16_t tam = strlen(UserTxBufferFS);  //Salva o tamanho da string
          UserTxBufferFS[tam] = '\n';             //Adiciona quebra de linha
          UserTxBufferFS[tam+1] = '\0';           //Adiciona o Terminator da string
          CDC_Transmit_FS(UserTxBufferFS, tam+1); //Transmite dados de volta ao computador
////////////////////////////////////////////////////////////////////////////

  return (USBD_OK);
  /* USER CODE END 6 */
}
```

Figura 10: Código do USB

Vale salientar que, como a função é implementada em um arquivo diferente da ”main.c”, para fazer o envio de variáveis declaradas na ”main.c” é preciso declarar novamente as variáveis no arquivo ”usbd-cdc-if.c”, mas como ”*extern*”. A Figura 11 ilustra um exemplo dessa declaração, para um *buffer* que armazena 10 *bytes*.

```
/* USER CODE BEGIN PV */
/* Private variables ----------
extern char UserTxBuffer[10];
extern char UserRxBuffer[10];
```

Figura 11: Declaração de variáveis extern

A função ”itoa”, mostrada na função ”CDCReceiveFS”, na Figura 12, tem o objetivo de converter as variáveis utilizadas de inteiro para *string*, inserindo no *buffer* que será transmitido, no qual são concatenados os caracteres terminais. Observe que o buffer foi declarado como *char*, como mostra a Figura 11, pois os dados serão enviados como *string*.

Também é possível enviar mais de uma variável pela USB, alterando a função ”CDCReceiveFS” e inserindo um símbolo de referência que será utilizado pelo MATLAB para identificação e separação, como mostra a Figura 12.

```
static int8_t CDC_Receive_FS(uint8_t* Buf, uint32_t *Len)
{
  /* USER CODE BEGIN 6 */
  USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]); //Padrão da função de recebimento
  USBD_CDC_ReceivePacket(&hUsbDeviceFS);        //Padrão da função de recebimento


  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
  // VARIÁVEIS UTILIZADAS
  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
      static uint8_t cont = 0;                //Contador de 8 bits para testar comunicação (static garante que só é inicializado uma vez)
      float var_1 = -41;                      //Ex2: variável 1 de teste, atribuido qualquer valor
      int var_2 = 902;                        //Ex2: variável 2 de teste, atribuido qualquer valor
      char BufferAux[32];                     //Buffer auxiliar para concatenar
  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
  // EX2: CONCATENA E ENVIA DADOS DE DUAS VARIÁVEIS ASSIM QUE RECEBE SINAL DO PC - SEPARADO POR SIMBOLO "+"
  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
      /* Segure "Ctrl" e aperte "/" no código abaixo para descomentar o exemplo 2 (comente o exemplo 1) */

      itoa(var_1,UserTxBufferFS,10);          //Converte de inteiro para string - ex:589 = ["5","8","9"]
      uint16_t tam = strlen(UserTxBufferFS);  //Salva o tamanho da string

      UserTxBufferFS[tam] = '+';              //Adiciona um simbolo base para servir como referencia para separar no PC/Código do MATLAB
      UserTxBufferFS[tam+1] = '\0';           //Adiciona o Terminator da string informando que a palavra acabou
      itoa(var_2,BufferAux,10);               //Converte de inteiro para string - ex:589 = ["5","8","9"]
      concatenar(UserTxBufferFS,BufferAux);   //Concatena os dados de BufferAux no final de UserTxBufferFS (função implementada)

      tam = strlen(UserTxBufferFS);           //Salva o novo tamanho da string
      UserTxBufferFS[tam] = '\n';             //Adiciona quebra de linha
      UserTxBufferFS[tam+1] = '\0';           //Adiciona o Terminator da string informando que a palavra acabou
      CDC_Transmit_FS(UserTxBufferFS, tam+1); //Transmite dados de volta ao computador


  return (USBD_OK);
  /* USER CODE END 6 */
}
```

Figura 12: Código USB concatenado

Por outro lado, para enviar os dados como inteiro, a variável do *buffer* deve ser inicializada como inteiro de 8 *bits*, por exemplo: uint8t UserTxBuffer[4], para 4 bytes. No código, basta inserir em cada posição do buffer a variável que deve ser enviada. No exemplo mostrado na Figura 13 deseja-se enviar duas variáveis de 16 bits: contagem_encoder e contagem_interrupção.

```
// CÓDIGO PARA ENVIO NO USB

/* INSERINDO AS VARI?VEIS NO VETOR DE BUFFER*/
UserTxBuffer[0] = contagem_encoder;                 // encoder low
UserTxBuffer[1] = contagem_encoder>>8;              // encoder high
UserTxBuffer[2] = contagem_interrupcao;             // timer low
UserTxBuffer[3] = contagem_interrupcao>>8;          // timer high

/* TRANSMITINDO OS DADOS */
CDC_Transmit_FS(UserTxBuffer, 4);  // inserir o número de pacotes em bytes que é enviado
```

Figura 13: Código USB para vários bytes

Para isso, são alocados na posição [0] do buffer os 8 bits menos significativos da primeira variável, em seguida os 8 bits mais significativos, que são alocados na posição [1]. O mesmo procedimento é feito para a segunda variável. É importante ressaltar que o deslocamento é feito pois só são enviados 8 bits por vez no *buffer*.

2.3 CÓDIGO MATLAB

Qualquer programa que tenha acesso à COM do computador pode ser utilizado como interface de comunicação com o microcontrolador. Por praticidade, será ilustrado um exemplo de implementação no *software* MATLAB 2019a. Inicialmente, é preciso criar um objeto do tipo serial para a porta COM, pelo *script* do MATLAB. Para isso, a configuração pode ser feita como na função ”Config_USB.m”, como ilustrado pela Figura 14.

```
function [ s ] = Config_USB(  )

% Função que configura toda a porta serial utilizada pelo uC e corrige os
% erros que são gerados por remoção da porta e não fechamento da COM.
% É necessário alterar o número da COMx toda vez que alterar o computador,
% para conferir a porta utilizada, ir em gerenciador de dispositivos

%%% CONFIGURAÇÃO DA PORTA SERIAL %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Caso não acesse é porque nao foi fechado
if isempty(instrfind) ~= 1 %ou seja, se não estiver vazio
    fclose(instrfind);     %fecha
    delete(instrfindall);  %deleta
end

% Nome da porta Serial a ser Conectada
s = serial('COM17');       %varia de acordo com o computador
set(s,'BaudRate',57600);   %pode ser qualquer valor padrão
set(s,'DataBits',8);
set(s,'Parity','None');
set(s,'stopBits',1);
set(s,'FlowControl','none');
set(s,'InputBufferSize',100);
set(s,'OutputBufferSize',100);
set(s,'Timeout',1);
set(s,'Terminator','LF');

%Abrindo a porta serial
fopen(s);
flushinput(s);
flushoutput(s);


end
```

Figura 14: Código Matlab função Config_USB

Para fins de visualização são inicializadas as variáveis (y,u,t e tamos), representando o sinal controlado, o esforço de controle, o tempo e o vetor de período de amostragem calculado a cada *loop*. Assim, pode ser possível construir um sistema de controle em *loop* e visualizar as variáveis utilizadas. O tamanho desse vetor de variáveis representa, também, o tamanho da janela de visualização.

Em seguida, dentro do *loop* principal, são atualizados os dados dos vetores de memória e plotados os dados. Isso é realizado pela função "Update-data()", que será detalhada mais à frente neste texto. Por fim, é realizado o envio de um pacote de solicitação de recebimento de dados, do computador para o microcontrolador, que pode ser uma informação de esforço de controle, ou apenas uma confirmação. Todo o procedimento é ilustrado pela Figura 15.

```
clear
close all
clc

s = Config_USB();              % configuração do USB

%% INICIALIZAÇÃO DE VARIÁVEIS (y,u,t,tamos)
amost = 200;                   % amostras salvas no vetor de visualização
y = zeros(1,amost);            % recebida (saida do sistema)
t = zeros(1,amost); t(end)=0;  % tempo (300 é o tamanho da janela)
u = zeros(1,amost);            % enviada (variavel controlada)
tamos = zeros(1,amost);        % período de amostragem (teste)
tic                            % inicio da contagem de tempo
parada=1;                      % flag de parada do while
flag_plot = 1;                 % se for 1, os dados são plotados

%% LAÇO PRINCIPAL
if(flag_plot), uicontrol('String','Parar','Callback','parada=0;'),end

while(parada)

    % atualização de variaveis e plots
    [tamos,y,u,t] = Update_data(tamos,y,u,t,flag_plot);

    % Comunicação USB [envia o u(t) e recebe o y(t)]
    fwrite(s,u(end),'uint8');
    y(end) = fscanf(s,'%f');

    % Interface com o usuário: Command Window
    CD_tamos_y_u = [tamos(end) y(end) u(end)];

    % Critério de parada - 1000 loops
    if(~flag_plot),parada=parada+1; if(parada==1000) ,parada=0;end,end

end
```

Figura 15: Código Matlab main

Caso seja necessário enviar mais de uma variável, pode ser utilizado um caractere como separador, que nesse caso foi o "+". Assim, a leitura no *script* do MATLAB é feita com o "fscanf" e a separação feita com a função "split", conforme mostra a Figura 16. Vale salientar que foi, ainda, adicionado um critério de parada acionado por botão no código exemplo e outro critério por iterações, que pode ser modificado a partir da variável *flagplot*.

```
while(parada)
    % Comunicação USB [envia o u(t) e recebe o y(t)]
    fwrite(s,1,'uint8');

    % Recebimento de dados da IMU
    aux1 = fscanf(s,'%c');
    mpu = split(aux1,'+');
    imu(:,end) = str2double(mpu(1:2));

    %%%%% Atualização de variáveis (tamos,y,u,t) %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    t(end) = toc;            % atualização do tempo em segundos
    tamos(end) = t(end) - t(end-1);

    tamos(1:end-1) = tamos(2:end);
    t(1:end-1) = t(2:end);   % Vetores responsáveis pela visualização em
    imu(1,1:end-1) = imu(1,2:end);   % janela, assim a última amostra é deslocada
    imu(2,1:end-1) = imu(2,2:end);   % janela, assim a última amostra é deslocada

    %%%%% Plots %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
     segundos = 0.1;                    %plota a cada x segundos
     if (mod(t(end),segundos) < 0.01)    %limita a quantidade de plots
         plot(t,imu(1,:),'r');                %plot saída do sistema
         hold on
         plot(t,imu(2,:),'b');
         grid on                        %adicionar grade a visualização
         axis([min(t) max(t) -185 185])  %configurar eixos de visualização
         drawnow
     end

    % Interface com o usuário: Command Window
%     CD_tamos_y_u = [tamos(end) y(end) u(end)];

    % Critério de parada - 1000 loops
    if(~flag_plot),parada=parada+1; if(parada==5000) ,parada=0;end,end

end
```

Figura 16: Código Matlab múltiplas variáveis

Tal implementação ocorreu devido ao período de amostragem, que cresce consideravelmente com *plot* de figuras. Por fim, a função “Update_data()”, que não tinha sido detalhada, pode ser ilustrada pela Figura 17, em que é realizada apenas a atualização dos vetores de dados e os gráficos são plotados.

É possível configurar a frequência de *plot* a partir de uma condição de contagem, expressa no exemplo. O código utilizado como exemplo foi implementado com base no problema de realizar um *loop* de controle entre o microcontrolador, o qual realizará a interface de comunicação com os atuadores e variáveis controladas, e o computador, que pode ser utilizado para implementar controladores diferentes de forma simplificada.

No entanto, tal sistema possui a falha de gerar um período de amostragem muito grande para alguns processos, sendo uma alternativa utilizar o computador apenas como um visualizador.

```
function [ tamos,y,u,t ] = Update_data( tamos,y,u,t,flag_plot )

% Função que atualiza os dados de 'tamos', 'y', 'u' e 't'; atualiza a janela
% de observação para o plot (gráfica); plota os dados de y e u; e fixa o
% periodo de amostragem do loop, o deixando constante.

%%%%% Atualização de variáveis (tamos,y,u,t) %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    t(end) = toc;            % atualização do tempo em segundos
    tamos(end) = t(end) - t(end-1);
    tamos(1:end-1) = tamos(2:end);

    t(1:end-1) = t(2:end);   % Vetores responsáveis pela visualização em
    y(1:end-1) = y(2:end);   % janela, assim a última amostra é deslocada
    u(1:end-1) = u(2:end);   % para a esquerda dando impressão de movimento


%%%%% Plots %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    if(flag_plot)
        segundos = 0.1;                  %plota a cada x segundos
        if (mod(t(end),segundos) < 0.01)   %limita a quantidade de plots
            plot(t,y,'r');               %plot saída do sistema
            grid on                      %adicionar grade a visualização
            axis([min(t) max(t) -0.1 255])  %configurar eixos de visualização
            drawnow
        end
    end

%%%%% Fixação do período de amostragem %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%     p_amos = 0.01;                   % periodo de amostragem em segundos
%     while(toc < (t(end-1) + p_amos))  % fixa o periodo de amostragem
%     end

end
```

Figura 17: Código Matlab função Update_data

Para o caso em que as variáveis são enviadas como inteiro, o *script* principal do MATLAB deve ser alterado para o modelo mostrado na Figura 18. O dado recebido do MATLAB é lido na variável pacote, utilizando a função ”fread”, na qual é especificada a quantidade de *bytes* recebidos, que para o exemplo são 4.

Em seguida, os dados separados são convertidos para as variáveis de interesse utilizando a função ”typecast”. O restante do código é semelhante, sendo apenas adaptado para atualizar as novas variáveis nos vetores ”VETORENCODER” e ”VETORTEMPOTIMER”

```
while(parada)
    %comunicacao usb envia u e recebe y
    fwrite(s,u(end),'uint8');

    %atualizacao de variaveis do plot
    [tamos, y, u, VETOR_ENCODER, VETOR_TEMPO_TIMER, t] = UpD(tamos, y, u, VETOR_ENCODER, VETOR_TEMPO_TIMER, t, flag_plot);

    % leitura do pacote de 4 bytes, enviado pelo stm
    pacote = fread(s,4);

    % convertendo as variáveis em 16 bits novamente
    encoder = typecast(uint8(pacote(1:2)), 'uint16');
    tempo = typecast(uint8(pacote(3:4)), 'uint16');
    y(end) = encoder;

    % atualização do último valor
    VETOR_TEMPO_TIMER(end) = tempo;
    VETOR_ENCODER(end)     = encoder;

    %interface com o usuário Comand Window
    CD_tamos_y_u = [tamos(end) y(end) u(end)];

    %critério de parada ->1000 loops
    if(~flag_plot), parada = parada + 1; if(parada==1000), parada = 0; end, end
end
```

Figura 18: Código Matlab typecast

Com base nessas implementações é possível modificar o programa, caso o usuário deseje utilizar outro *software* diferente do MATLAB. Para isso, basta abrir a porta serial COM no *software* destino. Outro ponto importante é observar qual COM está sendo utilizada pelo dispositivo no momento e alterar as funções aqui apresentadas com esse direcionamento. Para encontrar qual COM está sendo utilizada, basta abrir o painel de dispositivos do Windows, entrar no gerenciador de dispositivos e buscar pela porta COM em uso. A Figura 19 ilustra um exemplo de porta utilizada, no caso a COM 17.

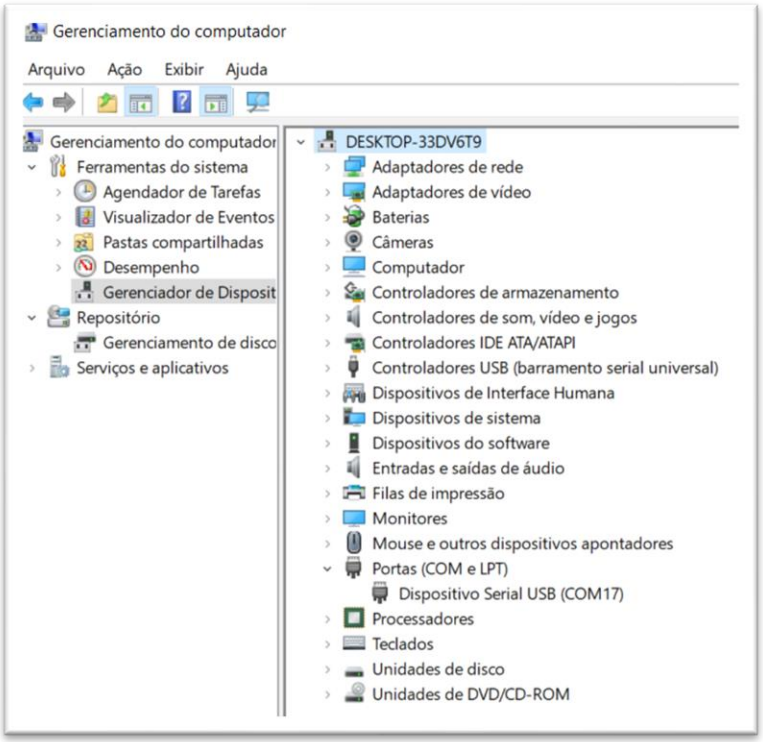

Figura 19: Porta COM utilizada

3 TIMERS

O STM32 possui 3 categorias de TIMERS, cada um com um conjunto de funcionalidades, são eles: básico, de propósito geral e avançado [1]. A quantidade depende da série do microcontrolador, sendo que o stm32F103C8, utilizado neste guia, possui 4. A lista completa é mostrada na Figura 20.

Nucleo P/N	Basic Timers	General Purpose Timers	Advanced Timers	High-resolution Timers	Low-power Timers	MAX clock speed
NUCLEO-F446RE	TIM6-7	TIM2-5 TIM9-14	TIM1 TIM8	-	-	90/180MHz
NUCLEO-F411RE	-	TIM2-5 TIM9-11	TIM1	-	-	100MHz
NUCLEO-F410RB	TIM6	TIM5 TIM9 TIM11	TIM1	-	LPTIM1	100MHz
NUCLEO-F401RE	-	TIM2-5 TIM9-11	TIM1	-	-	84MHz
NUCLEO-F334R8	TIM6-7	TIM2-3 TIM15-17	TIM1	HRTIM1	-	72/144MHz
NUCLEO-F303RE	TIM6-7	TIM2-4 TIM15-17	TIM1 TIM8 TIM20	-	-	72/144MHz
NUCLEO-F302R8	TIM6	TIM2 TIM15-17	TIM1	-	-	72/144MHz
NUCLEO-F103RB	-	TIM3-4	TIM1	-	-	64/72MHz
NUCLEO-F091RC	TIM6-7	TIM2-3 TIM14-17	TIM1	-	-	48MHz
NUCLEO-F072RB	TIM6-7	TIM2-3 TIM14-17	TIM1	-	-	48MHz
NUCLEO-F070RB	TIM6-7	TIM3 TIM14-17	TIM1	-	-	48MHz
NUCLEO-F030R8	TIM6	TIM3 TIM14-17	TIM1	-	-	48MHz
NUCLEO-L476RG	TIM6-7	TIM2-5 TIM15-17	TIM1 TIM8	-	LPTIM1 LPTIM2	80MHz
NUCLEO-L152RE	TIM6-7	TIM2-5 TIM9-11	-	-	-	32MHz
NUCLEO-L073RZ	TIM6-7	TIM2-3 TIM21-22	-	-	LPTIM1	32MHz
NUCLEO-L053R8	TIM6	TIM2-3 TIM21-22	-	-	LPTIM1	32MHz

Figura 20: Lista de TIMERs para cada modelo de STM32, disponível na página 311 de [1]

3.1 INTERRUPÇÃO POR ESTOURO DO TIMER

Para configurar a interrupção do *Timer* é preciso ir em *Pinout Configuration*. Na seção *Mode* escolha o *Timer* de sua preferência, alterando a opção *Clock Source* para *Clock* interno. Em seguida, na seção *Configuration*, selecione as opções *break interrupt* e *update interrupt* em *NVIC Settings*. Esse passo a passo é mostrado na Figura 21.

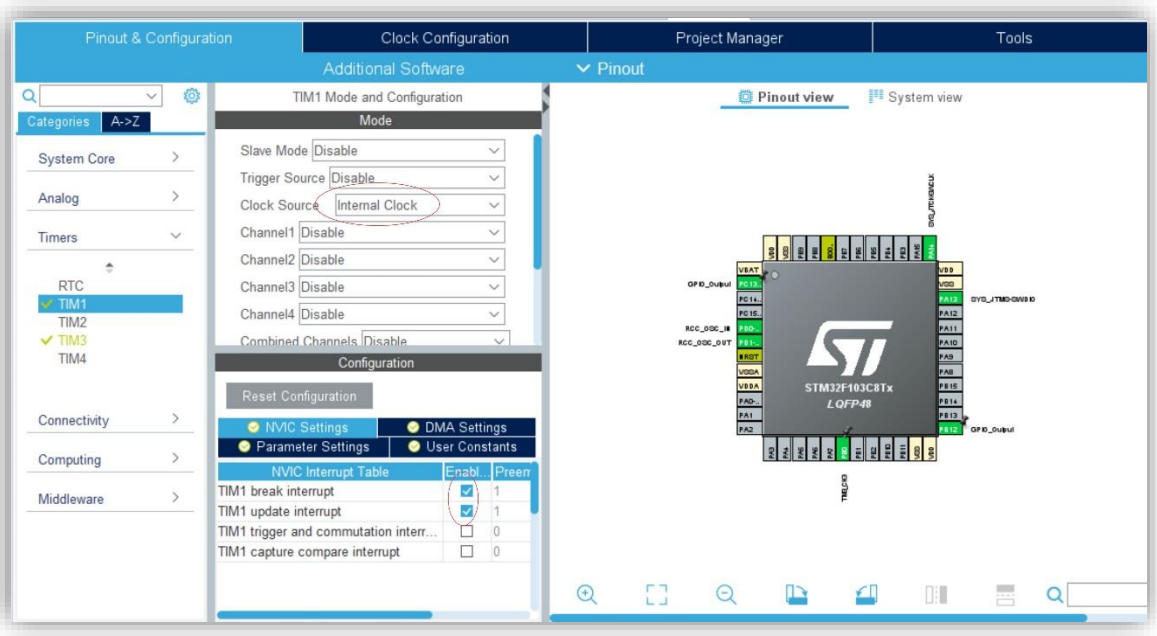

Figura 21: Configuração básica do Timer

Após isso, deve-se configurar a frequência do *Timer* de acordo com a Equação 1, que é válida para os de propósito avançado[1]:

$$f = \frac{clk}{(cp+1)(pr+1)(rp+1)} \quad (1)$$

Em que, *clk* é o *clock* do STM; *cp* é o ARR (do inglês, *Auto Reload Register*, registrador de 16 bits no qual é pré carregado o valor de contagem, cujo máximo é 65535; *pr* é o *prescaler*, o fator de divisão de frequência, cujo valor máximo também é de 65535; e *rp* é o RCR (do inglês, *Repetition Counter*) é um registrador responsável por aumentar ainda mais o período entre dois eventos e possui 8 *bits*, mas está disponível apenas nos *Timers* de propósito avançado.

Existe, ainda, o *counter mode*, registrador que determina o sentido de contagem, em que *up* determina a contagem de 0 ao valor predefinido (ARR) e *down* no sentido contrário. Se quisermos, por exemplo, uma interrupção de *Timer* configurada para a frequência de 10KHz e utilizando *clock* máximo de 72MHz, podemos escolher $pr = 71$ e $cp = 99$, como mostra a Figura 22.

[1] Para o cálculo de frequência de um *Timer* de propósito geral, basta desconsiderar o termo relacionado ao registrador RCR do denominador.

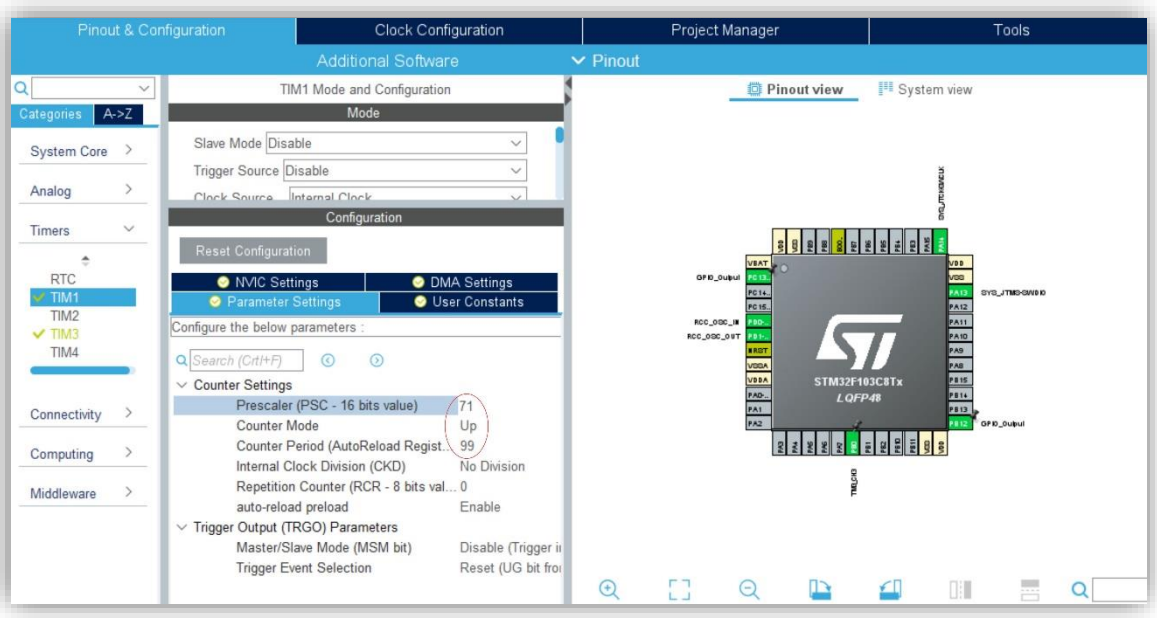

Figura 22: Configuração do prescaler e ARR

No código referente ao arquivo main.c é necessário inicializar o TIMER1 utilizando a biblioteca HAL, como ilustra a Figura 23.

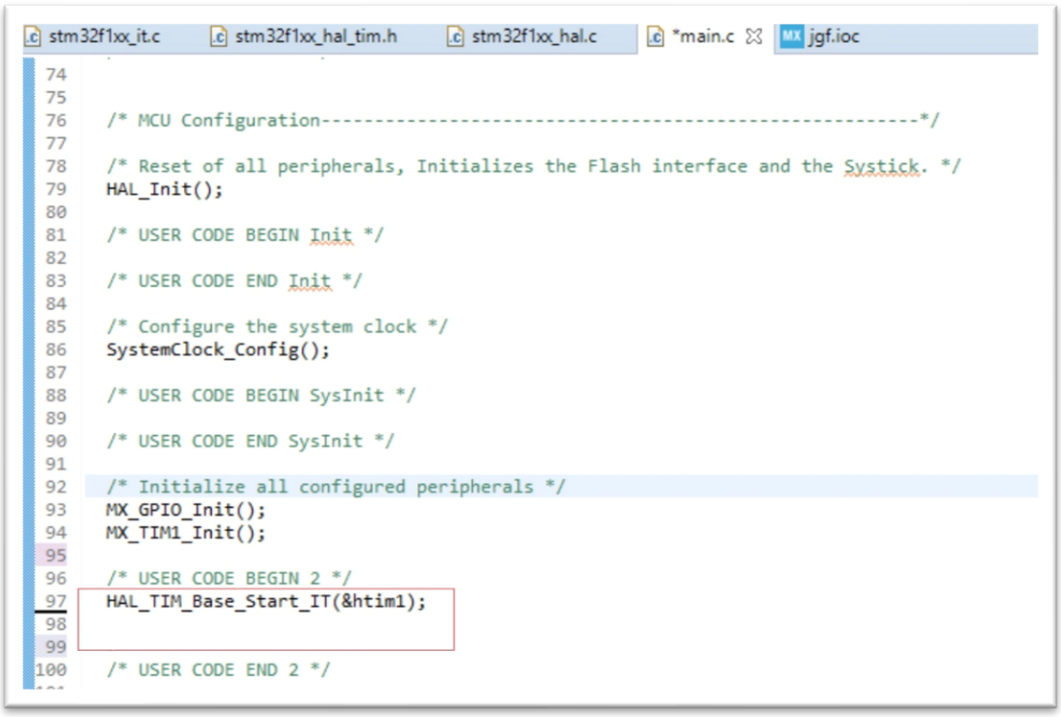

Figura 23: Inicialização do Timer para estouro

Por fim, caso deseje visualizar um LED piscando na frequência configurada para a interrupção do *Timer*, insira uma linha de código no arquivo stm32f1xx_it.c, referente à biblioteca HAL, como mostra a Figura 24. Lembre-se de ativar o pino no qual irá conectar o LED como *GPIO_output*, podendo ser o que já está presente na placa, que neste caso é o Pin13.

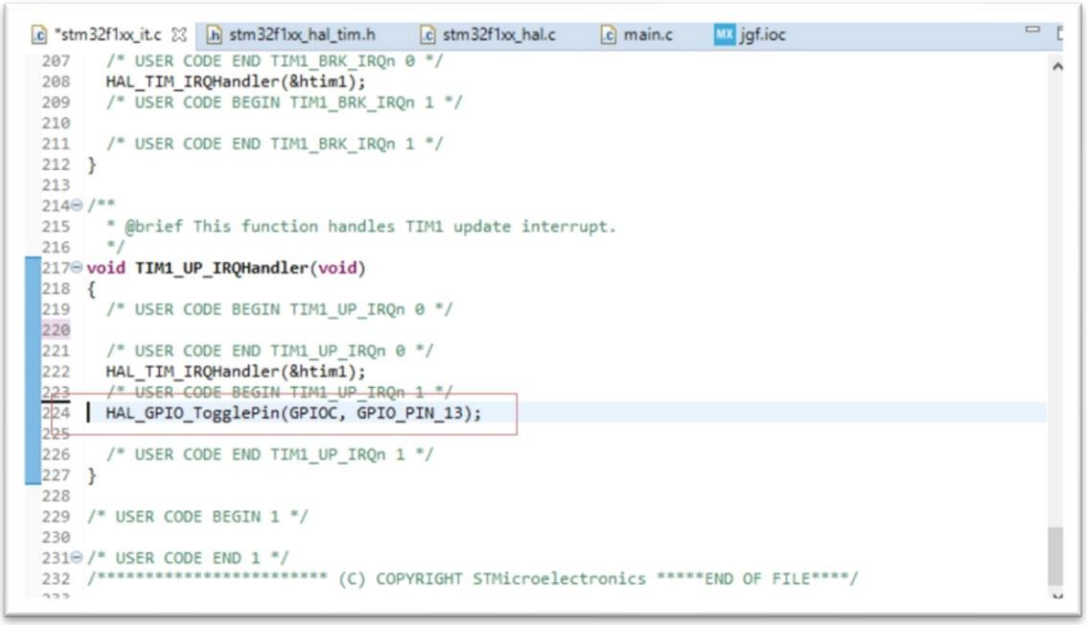

Figura 24: Exemplo de código de estouro do Timer

Observação: Se testado no osciloscópio, será possível observar que a frequência é metade do que foi projetado. Isso acontece se estiver utilizando a função *TogglePin*, porque a cada estouro o nível é trocado, ou seja, a cada dois estouros temos um período completo.

3.2 MÓDULO PWM

Para gerar um PWM, o Timer deve ser inicializado como mostrado na subseção anterior, seguido da seleção do canal, como mostra a Figura 25. Feito isso, o pino referente ao Timer e canal escolhidos ficará verde.

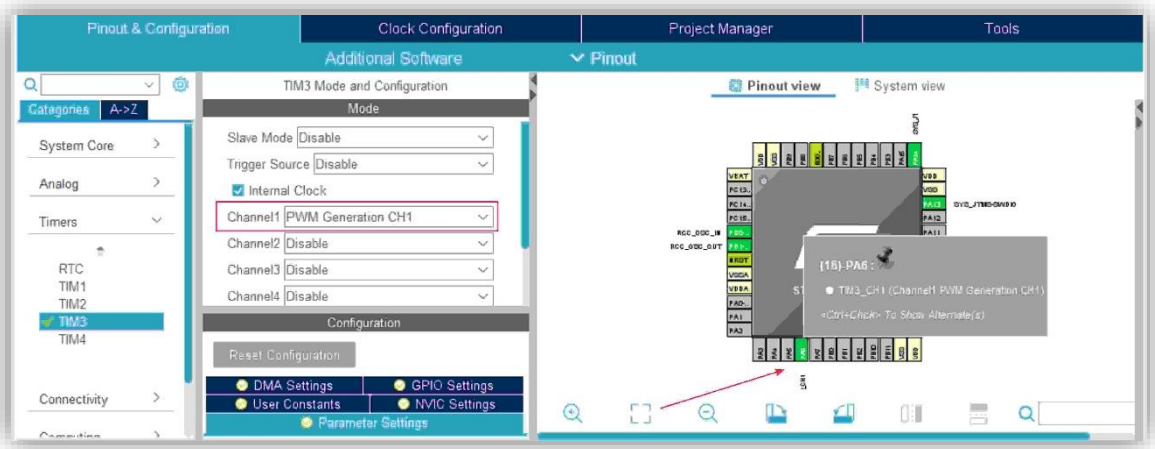

Figura 25: Configuração do módulo PWM

Em seguida, na barra *Parameter Settings*, devem ser definidos o valor do *Prescaler* e do *Auto Reload Register*, a partir dos quais definimos a frequência, calculada a partir da Equação 1. O ARR funciona como um contador de pulsos e, a partir do registrador CCR (do inglês *Capture/Compare Register*), é possível configurar o *duty cycle*. O cálculo do CCR para a definição do *duty cycle*, a partir do valor de ARR selecionado, é mostrado na Equação 2.

$$CCR = \frac{Duty\_Cycle * ARR}{100} \quad (2)$$

Se o ARR for de 100, é possível observar que o valor do CCR será o próprio *duty_cycle* desejado, sendo isso apenas um fator de conversão, visto que o ARR pode variar de 0 a 65535. O funcionamento dessa contagem pode ser verificado na Figura 26.

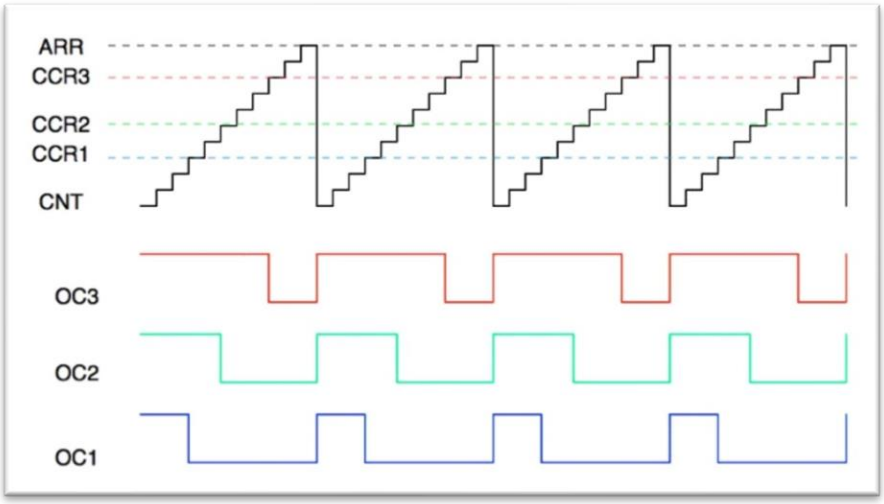

Figura 26: Funcionamento do contador

Por fim, para definir a frequência, basta escolher o *Prescaler*. Após serem salvas as configurações será criado automaticamente o arquivo main.c. Neste arquivo, antes do comando *while*, o PWM deve ser inicializado através da função HAL_TIM_PWM_Start(), conforme mostra a Figura 27. É possível observar que foi criada uma variável do tipo inteiro, na qual é definido o valor *duty cycle*. No loop *while* é feita a conversão utilizando a Equação 2.

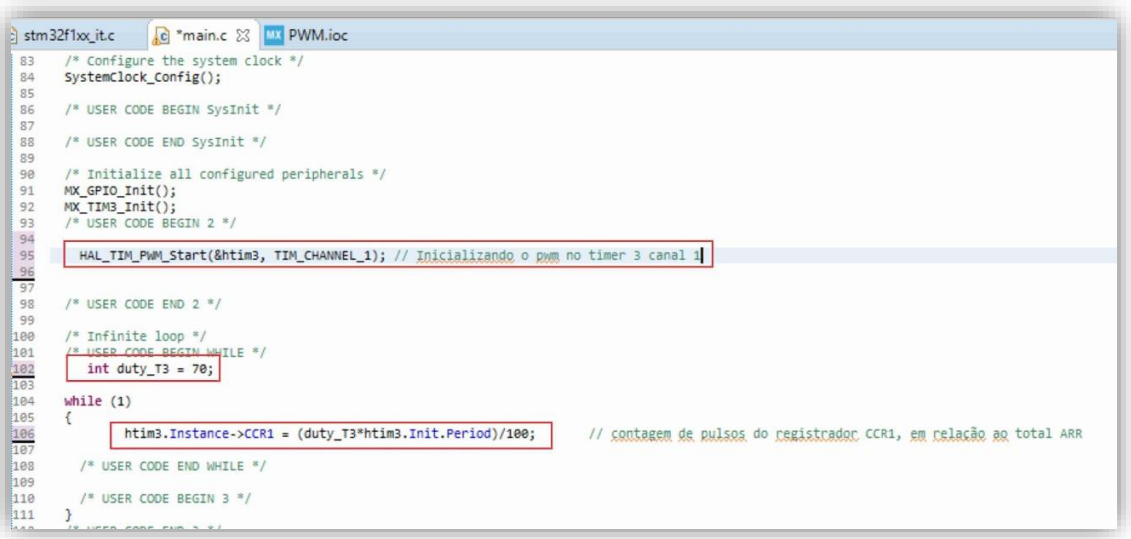

Figura 27: Código exemplo PWM

3.3 MÓDULO ENCODER DE QUADRATURA

Para a utilização do módulo contador que fornecerá a informação de posição relativo ao *encoder* de quadratura, necessária para a obtenção da velocidade, é preciso configurar o microcontrolador através da interface .ioc. É possível ver essa configuração através da Figura 28.

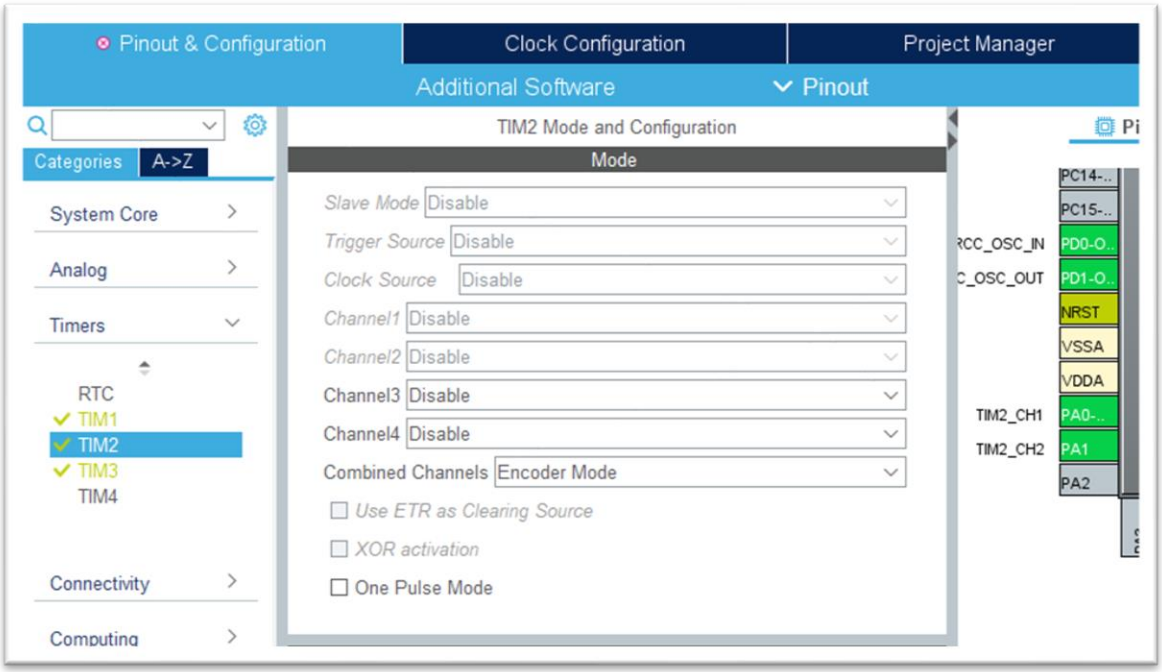

Figura 28: Configuração módulo encoder

O *Timer2* é utilizado na configuração *Encoder Mode*, sendo feita a combinação de dois canais (canal 1 e canal 2) através dos pinos PA0 e PA1, como pode ser visto no lado direito da Figura 29. Assim que o modo *encoder* é selecionado é preciso configurar o menu *Parameter Settings*, definindo o *prescaler* para 0. Nesta parte é importante ressaltar que, como o *prescaler* é um divisor de frequência, deve ser deixado em zero para essa aplicação, assim o registrador ARR pode ficar livre para contar os pulsos enviados pelo *encoder* na frequência correta de operação.

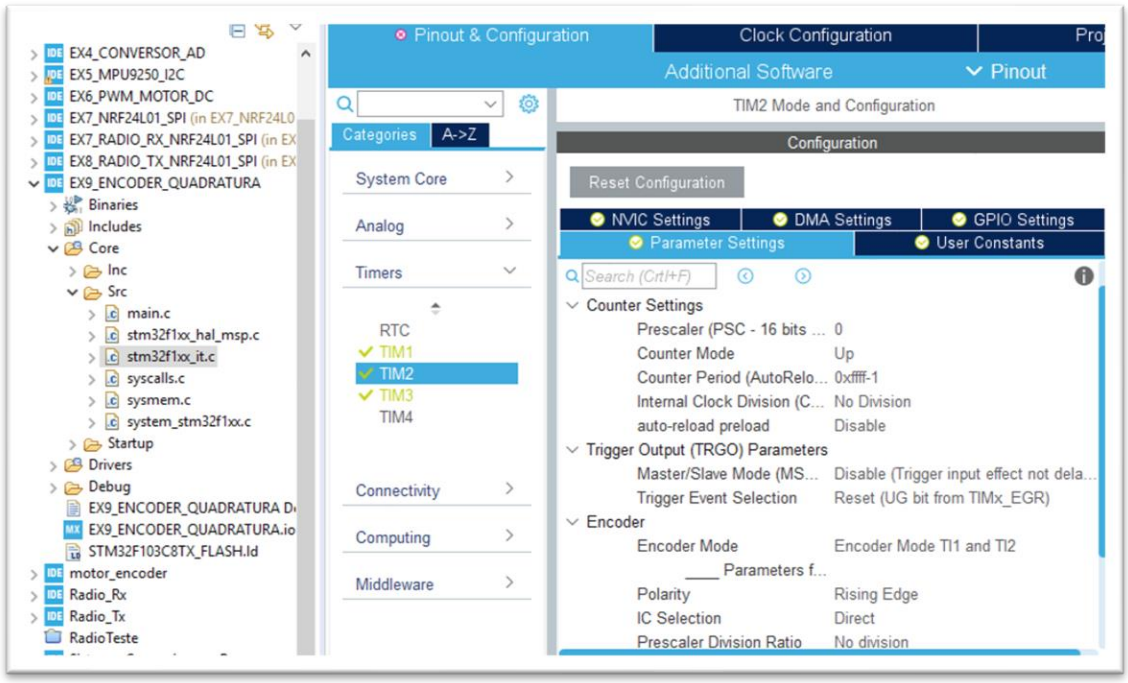

Figura 29: Configuração Encoder de quadratura

O registrador TIMx->CNT ("x" deve ser substituído pelo número do timer utilizado, que neste caso é o *Timer 2*), é o responsável pela contagem de eventos, que pode ser feita por borda de subida, descida ou ambos, esta última selecionada para a aplicação do *encoder*. Entretanto, até a confecção deste manual, a interface STM32CubeIDE não possui a opção de ativação no modo de borda de subida e descida, pois o menu disponibiliza apenas as opções *Rising Edge* (borda de subida) ou *Falling Edge* (borda de descida). Deste modo, foi realizada uma opção de configuração manual diretamente no código da main.c, como descrito na Figura 30.

```
static void MX_TIM2_Init(void)
{
////////////////////////////////////////////////////////////////////////
////////////// AJUSTE DE INTERRUPÇÃO DO ENCODER DE QUADRATURA /////////////
////////////////////////////////////////////////////////////////////////

  sConfig.IC1Polarity = TIM_ICPOLARITY_BOTHEDGE;
  sConfig.IC2Polarity = TIM_ICPOLARITY_BOTHEDGE;
  ////////////////////////////////////////////////////////////////////////
}
```

Figura 30: Ajuste de polaridade de borda

Realizada a etapa de configuração, é preciso apenas coletar os dados do *encoder* e utilizar as informações para encontrar a posição e velocidade, se desejado. Para verificar a posição, basta verificar o contador e relacioná-lo à resolução do *encoder*. Para isso, pode-se modificar o registrador ARR para contar até o valor equivalente ao número de pulsos por revolução do *encoder* utilizado, pois, sempre que o contador reiniciar, é porque houve uma volta completa.

O *encoder* utilizado para esse exemplo possui uma resolução de 1440 pulsos por revolução e, caso deseje visualizar a contagem em graus, basta realizar a conversão conforme ilustra o código na Figura 31.

```
////////////////////////////////////////////////////////////////////////////////
//////////////////////////// REGISTRO DE POSIÇÃO DO ENCODER ////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

POSICAO_ENCODER1 = TIM1->CNT; // variavel que recebe o registrador de posição do encoder
POSICAO_ENCODER2 = TIM2->CNT; // variavel que recebe o registrador de posição do encoder

POSICAO_ENCODER1_GRAUS = (360/1440)*POSICAO_ENCODER1; //converte saida para graus
POSICAO_ENCODER2_GRAUS = (360/1440)*POSICAO_ENCODER2; //converte saida para graus

////////////////////////////////////////////////////////////////////////////////
```

Figura 31: Código de posição do encoder

A medição de velocidade pode ser realizada utilizando pelo método de janelamento, em que é definida uma janela fixa de tempo, como 1 segundo, e a partir disso são contados quantos pulsos houveram no intervalo, o que é melhor para frequências altas. Por outro lado, pode ser feita a contagem de tempo entre pulsos, em que é medido o tempo entre cada borda de subida e descida, sendo valorizada a precisão para frequências mais baixas. É possível realizar ambas as implementações com os conceitos já apresentados neste guia.

Para o caso do janelamento, basta configurar um *timer* para estourar em um determinado intervalo de tempo, capturando o valor do contador do *encoder* quando a interrupção deste *timer* for chamada (ou seja, quando houver o estouro). Armazenando em variáveis o valor anterior e o atual e conhecendo o intervalo de tempo, definido pelo usuário, é calculada a velocidade.

Para contar o tempo entre pulsos é possível deixar um *timer* configurado para contar até o valor máximo, ou seja, ARR em 65535. Inicializando o *Timer* configurado em modo *encoder* como interrupção, basta chamar uma função *callback* sempre que houver detecção de borda de subida ou descida, na qual é feita a captura do contador do *Timer* que está realizando a contagem. De forma semelhante ao caso anterior, basta armazenar os tempos atual e anterior, efetuar a diferença, e calcular o intervalo.

4 CONVERSOR ANALÓGICO/DIGITAL

O STM32 possui um módulo conversor analógico/digital de 12 *bits*, o que resulta em uma precisão de 4096 subpartes de 3.3V, sendo possível detectar variações com o limite mínimo de até 0.8mV para entradas analógicas. Para configurar o conversor, inicialmente é necessário, definir qual canal INx deseja-se utilizar no menu principal do arquivo .ioc. É possível verificar que, ao selecionar uma das opções, o pino referente à entrada ficará verde no menu de visualização do *chip*. A Figura 32 ilustra este processo.

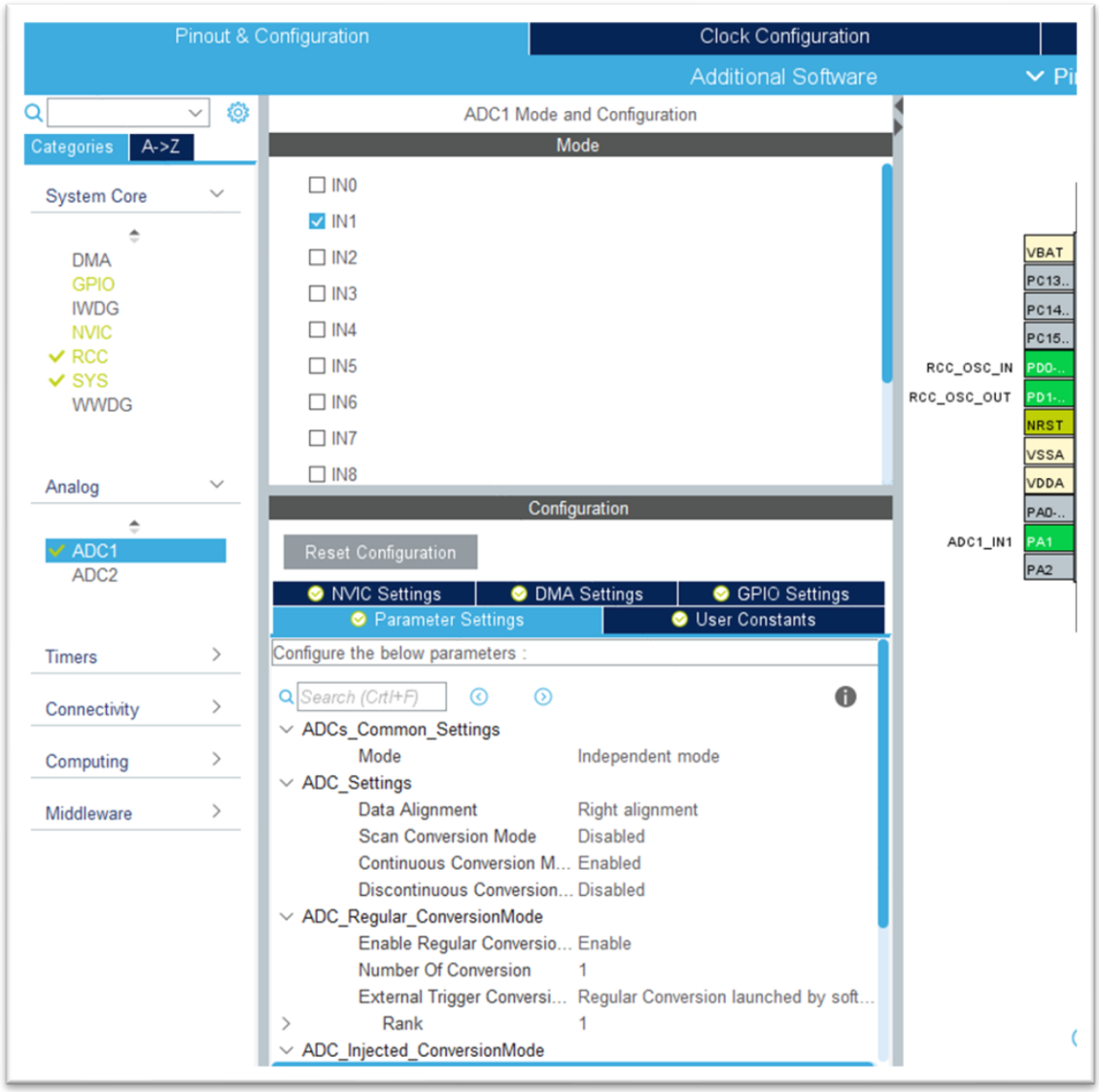

Figura 32: Configuração conversor A/D

Após esta etapa de configuração, é necessário salvar o arquivo para que o programa gere os arquivos editáveis. Desse modo, é possível configurar o momento de aquisição do sinal na main.c, bem como seu tempo de conversão, e atribuir tal valor a uma variável que poderá ser utilizada posteriormente.

Através da main.c, dentro do laço principal do programa, é iniciado o conversor A/D, coletados os dados e armazenados em uma variável do tipo uint16_t, para que depois esse dado convertido possa ser utilizado nas aplicações desejadas.

```
while (1)
{
////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////// CONVERSOR A/D ///////////////////////////////////

            HAL_ADC_Start(&hadc1);                      // Inicia o conversor A/D
            HAL_ADC_PollForConversion(&hadc1, 100);     // (100 é referente ao Timeout)
            valor_ad1 = HAL_ADC_GetValue(&hadc1);       // Atribui valor à variável utilizada
            HAL_ADC_Stop(&hadc1);                       // Para conversor

////////////////////////////////////////////////////////////////////////////////////////
}
```

Figura 33: Código conversor A/D

```
/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc1;

/* USER CODE BEGIN PV */

uint16_t valor_ad1 = 0; // Declara variável que será usada para armazenar os dados
                        // do conversor AD1 exemplificado nesse código.

/* USER CODE END PV */
```

Figura 34: Declaração de variáveis do conversor A/D

É importante salientar que a aquisição pode ser feita de outras formas, como interrupção ou mesmo aquisição ininterrupta, a depender da aplicação desejada. Para exemplificar, foi implementada apenas uma destas formas.

5 MPU9250 (COMUNICAÇÃO I2C)

Esta seção apresenta a comunicação com o dispositivo de sensoriamento inercial MPU9250, que utiliza como base o chip MPU6050 acrescido de um sensor magnetômetro AK8963, logo, os dados aqui apresentados são válidos para ambos os sensores. Sua comunicação com o microcontrolador é realizada via I2C, sendo escolhido por se tratar de um dos sensores mais utilizados para confecção de plataformas de controle. Desse modo, como o objetivo deste manual é simplificar a confecção de plataformas utilizando o microcontrolador stm32f103c8, optou-se por implementar um código teste para i2c com base neste sensor.

Para a etapa de comunicação, foi observado que o acesso aos dados de magnetômetro não são obtidos de forma direta através do I2C. No entanto, a interface auxiliar I2C do MPU9250 possui um multiplexador de desvio de interface, que permite ao processador do sistema acessar os registradores do sensor AK8963. Como não foi encontrada uma biblioteca que fizesse essa comunicação entre os sensores e o microcontrolador escolhido, foi elaborada e disponibilizada, na plataforma GitHub, uma biblioteca a nível de registradores utilizando a IDE STM32CUBE, utilizando como base a biblioteca desenvolvida por Sina Darvishi (2016), para que os dados pudessem ser acessados pelo microcontrolador e utilizados no projeto.

Para configurar a comunicação I2C no STM32CubeIDE basta marcar a opção de I2C no menu inicial do projeto, ou seja, o .ioc. através da aba *Pinout* & *Configuration*, no sub-menu *Mode*. Assim que o procedimento for realizado os pinos utilizados aparecerão na aba de visualização do *chip*.

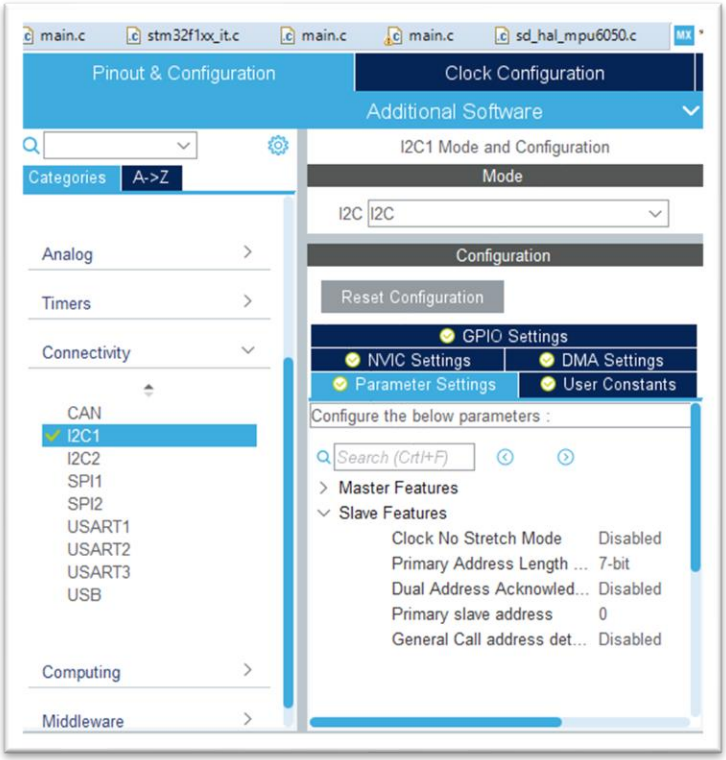

Figura 35: Configuração I2C

Após esta etapa inicial de configuração, é necessário copiar os arquivos sd_hal_mpu9250.c e sd_hal_mpu9250.h da biblioteca utilizada para as pastas de src e include, respectivamente, e inserir um include do arquivo sd_hal_mpu9250.h na main.c, como ilustrado na Figura 36 e Figura 37.

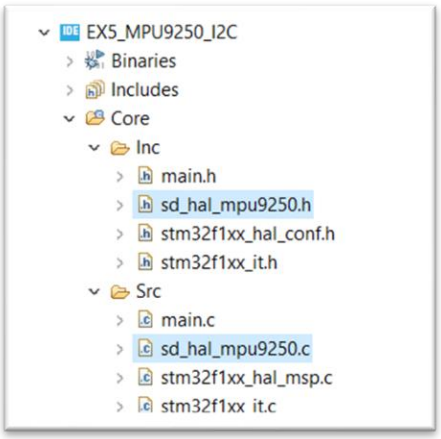

Figura 36: Arquivos da biblioteca da IMU

```
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "sd_hal_mpu9250.h"

/* USER CODE END Includes */
```

Figura 37: Include da biblioteca da IMU

Realizada a etapa de configuração inicial é possível utilizar as funções implementadas na biblioteca. Antes de coletar qualquer dado devem ser utilizadas as funções "SD_MPU6050_Init" e "initAK8963", que inicializam os módulos MPU6050 e AK8963, respectivamente, e realizam as etapas de configuração inicial, configurando valores de resolução e estabelecendo a comunicação com o sensor.

```
// inicialização do MPU9250: MPU6050(Accel & Gyro) e AK8963(Magnetometer)
SD_MPU6050_Init(&hi2c1,&mpu1,SD_MPU6050_Device_0,SD_MPU6050_Accel_2G,SD_MPU6050_Gyro_250s );

initAK8963(&hi2c1,magCalibrate);
```

Figura 38: Inicialização de funções da IMU

Uma vez efetuada a inicialização dos dispositivos, o sensor pode ser lido no laço principal do programa, como no exemplo, ou pode ser efetuada a leitura em sub-funções implementadas pelo usuário. Ambas as opções utilizam as funções de leitura “SD_MPU6050_ReadAll()“, que lê os dados referentes ao giroscópio, acelerômetro e temperatura, e a função “SD_MPU9250_ReadMagnetometer()”, que lê os dados do magnetômetro. As duas funções salvam os dados no *struct* de dados brutos, cujos valores podem ser atribuídos a variáveis convertidas que contém os valores de grandezas físicas referentes à velocidade angular, aceleração linear, campo magnético e temperatura.

```
typedef struct  {

uint8_t Address;         /*!< I2C address of device. */
float Gyro_Mult;         /*!< Gyroscope corrector from raw data to "degrees/s".*/
float Acce_Mult;         /*!< Accelerometer corrector from raw data to "g". */

int16_t Accelerometer_X; /*!< Accelerometer value X axis */
int16_t Accelerometer_Y; /*!< Accelerometer value Y axis */
int16_t Accelerometer_Z; /*!< Accelerometer value Z axis */

int16_t Gyroscope_X;     /*!< Gyroscope value X axis */
int16_t Gyroscope_Y;     /*!< Gyroscope value Y axis */
int16_t Gyroscope_Z;     /*!< Gyroscope value Z axis */

int16_t Magnetometer_X;     /*!< Magnetometer value X axis */
int16_t Magnetometer_Y;     /*!< Magnetometer value Y axis */
int16_t Magnetometer_Z;     /*!< Magnetometer value Z axis */

float   Temperature;       /*!< Temperature in degrees */
//I2C_HandleTypeDef* I2Cx;
} SD_MPU6050;
```

Figura 39: Struct do sensor MPU9250

```
while (1)
{
          ////////////////////////////////////////////////////////
          // MPU9250 ****************************************//
          ////////////////////////////////////////////////////////

          SD_MPU6050_ReadAll(&hi2c1,&mpu1);
          SD_MPU9250_ReadMagnetometer(&hi2c1,&mpu1);
          g_x = gRes*mpu1.Gyroscope_X - gyroBias[0];
          g_y = gRes*mpu1.Gyroscope_Y - gyroBias[1];
          g_z = gRes*mpu1.Gyroscope_Z - gyroBias[2];
          a_x = aRes*mpu1.Accelerometer_X - accelBias[0];
          a_y = aRes*mpu1.Accelerometer_Y - accelBias[1];
          a_z = aRes*mpu1.Accelerometer_Z - accelBias[2];
          m_x = mRes*mpu1.Magnetometer_X*magCalibrate[0] - magBias[0];
          m_y = mRes*mpu1.Magnetometer_Y*magCalibrate[1] - magBias[1];
          m_z = mRes*mpu1.Magnetometer_Z*magCalibrate[2] - magBias[2];

          ////////////////////////////////////////////////////////
```

Figura 40: Código base de coleta de dado do sensor

Para cada escala de resolução configurada é necessário um novo fator de conversão. Para auxiliar no auxílio desse cálculo, estão disponibilizados os *datasheets* com todas as informações referentes ao sensor e os registradores que controlam a resolução do dispositivo. Assim, para um melhor uso do dispositivo, é necessário verificar a documentação.

Observação: para realizar a comunicação com o sensor não há a necessidade de um aprofundamento maior no estudo do protocolo i2c, mas, para um melhor entendimento da biblioteca, é recomendado que o usuário entenda como a comunicação ocorre. Essas informações podem ser encontradas no *datasheet* do sensor anexado ao *link* do GitHub disponibilizado no início deste trabalho.

6 NRF24L01+ (COMUNICAÇÃO SPI)

Para a transmissão de dados do sistema embarcado da placa desenvolvida foi utilizado o módulo transceptor de rádio frequência NRF24L01+, por possuir um longo alcance e baixo ruído, dificultando a perda de pacotes por distância e por possuir uma documentação grande disponível construída pela comunidade acadêmica e entusiastas. Desse modo, estão disponibilizadas as implementações do sistema receptor e transmissor neste projeto.

Para realizar a comunicação com o sensor é utilizado o protocolo SPI, que conta com cinco pinos de comunicação: CE (do inglês, *Chip Enable*), utilizado para selecionar se o rádio transmitirá ou receberá dados; CSN (do inglês, *Chip Select Now*), chave seletora para permitir o dispositivo captar os dados da SPI; SCK (do inglês, *Serial Clock*), para sincronizar a comunicação com pulsos de clock; MOSI (do inglês, *Master Out Slave In*), é uma entrada SPI para o rádio; e MISO (do inglês, *Master In Slave Out*), é uma saída do rádio. A presença desse protocolo possibilita uma comunicação mais rápida para o dispositivo.

Inicialmente deve ser configurada a comunicação SPI através do arquivo de configuração .ioc, de forma semelhante às demais configurações aqui já exemplificadas. A opção de comunicação SPI com o *Mode Full-Duplex Master* deve ser habilitada na aba de *Connectivity*, para esse exemplo, e definido o *prescaler* para 32 se for utilizado um *clock* de 72MHz. Assim, a velocidade é definida para a máxima aceita pelo protocolo. Essa configuração pode ser ilustrada pela Figura 42.

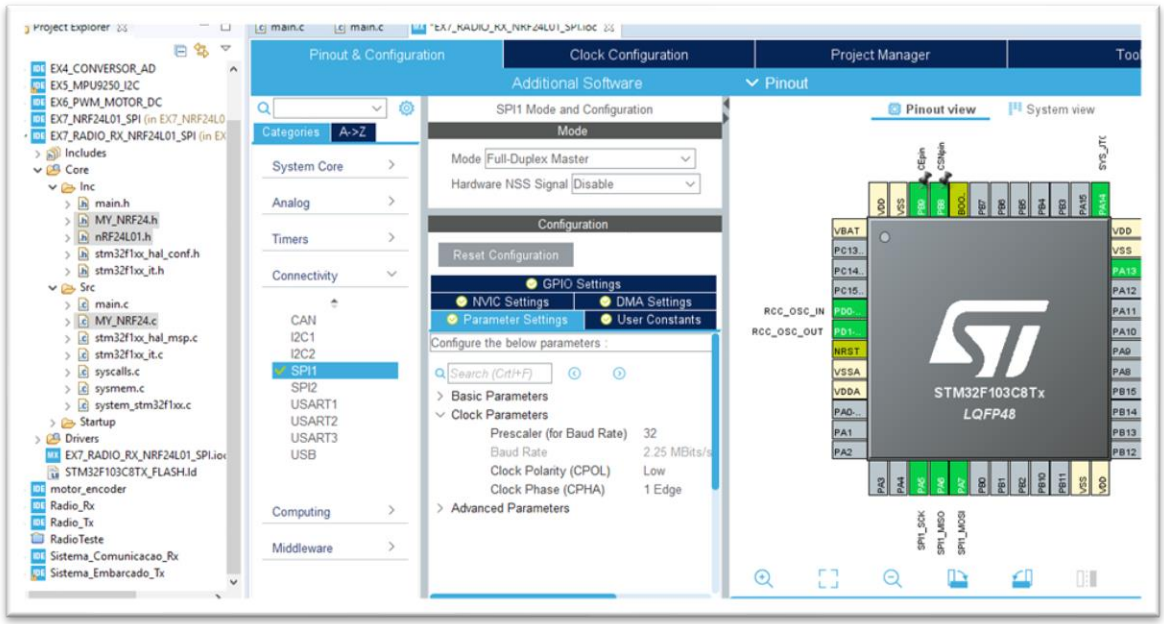

Figura 41: Configuração comunicação SPI

Realizada a etapa de configuração, é necessário, como no capítulo anterior, adicionar os arquivos da biblioteca utilizada. Sendo alocados o (MY_NRF24.h), (MY_NRF24.c) e o (nRF24L01.h) nas pastas Core/Src (para arquivos .c) e Core/Inc (para arquivos .h) e, por fim, adicionar o include do arquivo "MY_NRF24.h" na *main*. Vale salientar que não é necessário incluir o outro arquivo "nRF24L01.h' na *main* também, uma vez que este já foi adicionado dentro de "MY_NRF24.h".

```
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "MY_NRF24.h"
/* USER CODE END Includes */
```

Figura 42: Include biblioteca do rádio

Após isso, devem ser declaradas as variáveis do código do receptor do rádio e do código do transmissor, lembrando que devem ser programados dois códigos diferentes. Ambos os códigos devem conter o mesmo endereço de rádio e mesmo canal para que a comunicação ocorra com sucesso. A seguir é exemplificada a etapa de declaração de variáveis nos dois casos, sendo que as variáveis utilizadas para checagem serão explicadas posteriormente.

```
/* USER CODE BEGIN PV */
//////////////////// VARIAVEIS RECEPTOR ///////////////////////////////

uint64_t RxpipeAddrs = 0x11223344AA; // endereço do radio
char myRxData[32];                   // buffer receptor radio
char myAckPayload[32] = "Ack ok";    // palavra usada para reconhecimento

////////////////////////////////////////////////////////////////////////
```

Figura 43: Definição de variáveis do receptor

```
/* USER CODE BEGIN PV */
//////////////////// VARIAVEIS TRANSMISSOR ////////////////////////////

uint64_t TxpipeAddrs = 0x11223344AA; // endereço do radio
char myTxData[32] = "OLA MUNDO! \n"; // buffer transmissor radio
char AckPayload[32] = "vazio \n";    // palavra usada para reconhecimento

////////////////////////////////////////////////////////////////////////
```

Figura 44: Definição de variáveis do transmissor

A próxima etapa na confecção do código de comunicação a rádio é a inicialização dos dispositivos. Essa inicialização pode ser realizada de forma que os dados sejam enviados e recebidos sem nenhum tipo de checagem de pacote, o que torna mais simples e mais rápido o processo, mas reduz a robustez da comunicação.

Pode, também, ser definida de tal forma que inclua uma palavra de reconhecimento, que irá se encarregar de fazer o processo de checagem se o pacote chegou ao destino ou não. Assim, se o receptor receber a mensagem enviada pelo rádio transmissor, deve ser emitido um sinal de confirmação, pelo receptor, para informar que a mensagem foi recebida inteira. Isso pode ser ilustrado pelos dois códigos a seguir, os quais possuem exemplos de transmissor e receptor.

```
  ////////////////////**** TRANSMISSOR ****////////////////////////////////////

        NRF24_begin(CEpin_GPIO_Port,CSNpin_Pin,CEpin_Pin,hspi1);
    printRadioSettings();
    NRF24_stopListening();
    NRF24_openWritingPipe(TxpipeAddrs);
    NRF24_setAutoAck(false);      //Ativar para o caso de reconhecimento
    NRF24_setChannel(52);         //Deve ser o mesmo do receptor
    NRF24_setPayloadSize(32);
//      NRF24_enableDynamicPayloads();//(descomentar no caso de reconhecimento)
//      NRF24_enableAckPayload();     //(descomentar no caso de reconhecimento)

  ////////////////////////////////////////////////////////////////////////////
  while (1)
  {
          /////////////////////////// RADIO ///////////////////////////////////
          if(NRF24_write(myTxData, 32)) //envia dados para receptor
                {
//                      NRF24_read(AckPayload, 32); //recebe confirmação de envio
                }
          ////////////////////////////////////////////////////////////////////
   }
```

Figura 45: Código do rádio transmissor

```
  ////////////////////**** RECEPTOR ****////////////////////////////////////

        NRF24_begin(CEpin_GPIO_Port,CSNpin_Pin,CEpin_Pin,hspi1);
        printRadioSettings();
        NRF24_setAutoAck(false); //Ativar para o caso de reconhecimento
        NRF24_setChannel(52);    //Deve ser o mesmo do transmissor
        NRF24_setPayloadSize(32);
        NRF24_openReadingPipe(1, RxpipeAddrs);
        //NRF24_enableDynamicPayloads(); //Descomentar para reconhecimento
        //NRF24_enableAckPayload();      //Descomentar para reconhecimento
        NRF24_startListening();

  ////////////////////////////////////////////////////////////////////////////
  while (1)
  {
          /////////////////////////// RADIO ///////////////////////////////////
          if(NRF24_available()) //verifica se há dados disponíveis
                {
                        NRF24_read(myRxData, 32); // Lê dados recebido
                        //NRF24_writeAckPayload(1, myAckPayload, 32);//Transmite de volta
                }
          ////////////////////////////////////////////////////////////////////
   }
```

Figura 46: Código do rádio receptor

Possuindo 125 canais disponíveis para esse chip, pode-se modificar o canal utilizado se, no ambiente de teste, houverem ruídos de comunicação que gerem interferências no sinal. A única restrição deve-se ao fato de que, sempre, os canais do receptor e transmissor devem ser os mesmos, senão a comunicação não ocorre com sucesso.

Observação: Ao implementar o sistema de comunicação com o rádio, depois de diversos testes, foi encontrado um problema relacionado ao acesso de dados pelo SPI. Analisando o *datasheet* do dispositivo, foi verificado que o mesmo não possuía os capacitores de filtragem necessários para o seu funcionamento. A solução deste problema foi a soldagem dos componentes, que pode ser visualizado no manual de hardware deste mesmo projeto. De acordo com pesquisa realizada, nem todos os dispositivos apresentam este problema, varia de acordo com o fabricante do módulo.

REFERÊNCIAS

[1] Noviello, Carmine. "Dominando o STM32." *Leadpub. Obtido em http: // www2. keil. com / mdk5 / uvision* (2017).