
# AN-1
# Protocolo de comunicação RF

**Trabalho da disciplina de Programação de Sistemas Embarcados da UFMG**


## Aplicação de um STM32F401RE como transmissor e receptor usando verificador de integridade de mensagens
## por E. Villani e G. Gomes


## Introdução

Essa *Application Note* mostrará o funcionamento de um protocolo de de transmissão por meio da placa STM32F401RE e a verificação da integridade da mensagem por meio de um algoritmo CRC (*Cyclic redundancy check*), isto é, um algoritmo detector de falhas em protocolos de transmissão (dado que essas estão sujeitas a interferências que podem alterar os valores de bits). Para os códigos desenvolvidos, esses podem ser acessados em no repositório do GitHub AN_RFMod_CRC (clique aqui).

## Conteúdo

## Tecnologias e Conceitos

### ASK

Em telecomunicações há várias formas de se transmitir mensagens. No dia-a-dia, estamos acostumados em usar rádios *FM* e *AM*, os quais utilizam métodos de transmissão que trabalham em cima da variação da frequência e na variação da modulação. O *ASK* (*Amplitude-shift keying*, ou Modulação em Amplitude por chaveamento) é uma forma de transmissão AM para sinais digitais (o nosso caso com o uso do núcleo STM32F401RE).

Simplificadamente, o *ASK* funciona no sistema aberto e fechado, onde aberto é o nível de sinal 1 e onde fechado é o nível de sinal 0. Uma onda portadora qualquer pode ser utilizada. O resultado modulado é: o valor da onda portadora onde o valor da modulação é 1 e 0 onde o valor da modulação é zero. Esquematicamente, isto pode ser representado pela figura 1.

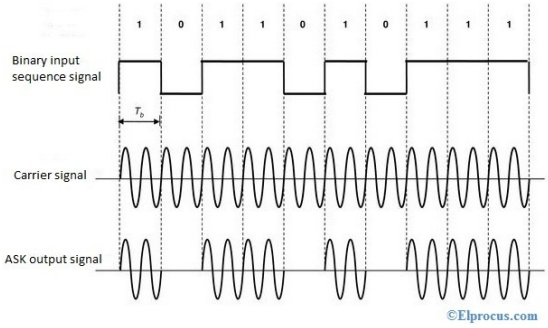


Figura. 1: Exemplo esquemático da modulação *ASK*.

### PLL

Por PLL entende-se Phase-locked loop, ou Malha de Captura de Fase. Ele é um sistema de controle que relaciona a fase de entrada e a fase de saída de sinais com o objetivo de mantê-las em sincronia. O bloco de construção deste sistema é mostrado na figura 2.

O PLL consegue detectar se o sinal atual está atrasado do sinal recebido ou adiantando. De acordo com a situação atual, o sinal de saída é corrigido diminuindo o tempo em que está ativo ou aumentando. A figura 3 representa uma situação dessa.

Uma das aplicação do PLL é justamente para detecção e demodulação de sinais AM (importando para o nosso caso, pois utilizamos um sinal modulado desta forma).

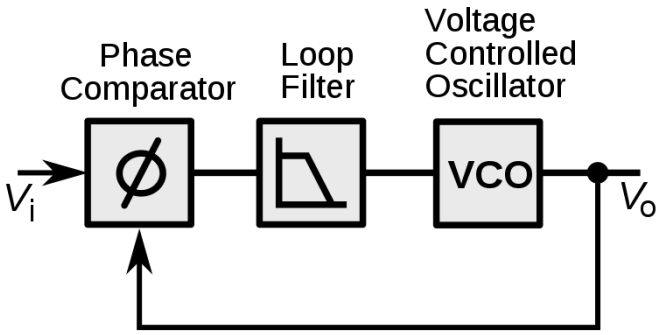


Figura. 2: Diagrama do bloco de um PLL genérico.

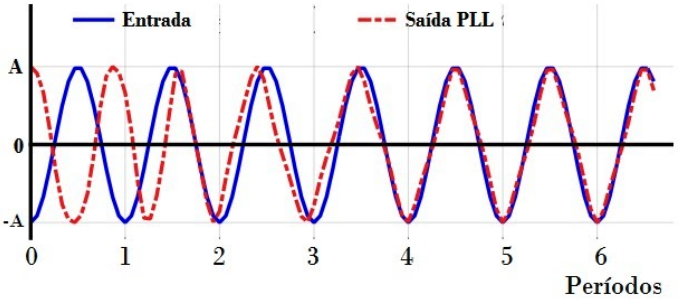


Figura. 3: Exemplo de funcionamento de um PLL.

### Receptor e Circuito Fatiador

Para o processamento do sinal, temos um receptor. Na figura abaixo podemos ver os blocos principais do circuito de recepção. O circuito possui por um filtro passa-baixas e um capacitor de desacoplamento, além de um circuito fatiador de dados (*data slicer circuit*) que é, resumidamente, um comparador tendo como limiar (*threshold*) a metade da tensão da fonte. O conjunto filtro passa-baixas mais capacitor de desacoplamento faz com que o receptor só consiga receber sinais numa certa faixa de frequência, ou seja, a largura de cada pulso possui valores mínimos e máximos aceitáveis. Isso, por sua vez, faz com que um sinal com muitos 1's ou 0's seguidos não seja reconhecido pelo receptor. Essa limitação é tratada com um preâmbulo antes das mensagens e uma codificação com equilíbrio DC, conceitos explicados em seguida.

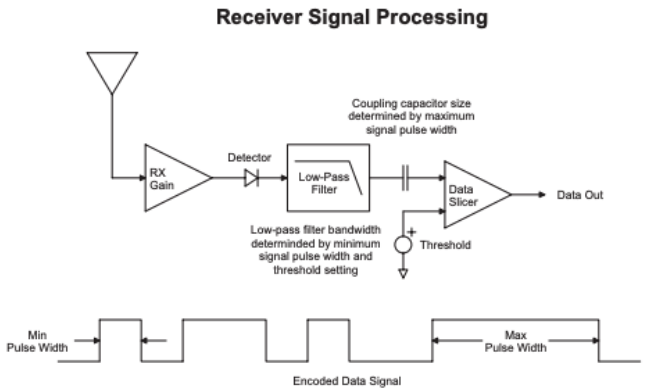


Figura. 4: Receptor e circuito fatiador.

## Pacote

### Formato da Pacote

Independente do tipo de comunicação que se queira fazer (seja comunicação entre pessoas ou entre protocolos), há-se uma padronização do formato da mensagem. No caso da língua portuguesa, numa comunicação, há as regras gramaticais e vocabulários previamente estabelecidos entre aqueles que se comunicam. Para os protocolos é a mesma coisa. Seja um protocolo *TCP/IP*, *UDP/IP* ou *Ethernet*, teremos um padrão pré-estabelecido entre os pacotes de mensagens para garantir que a mensagem seja devidamente identificada. Claramente quando se trata de mensagens e pactos de protocolos, a padronização trata de informações referentes a características e tipos do pacote e da mensagem em si.

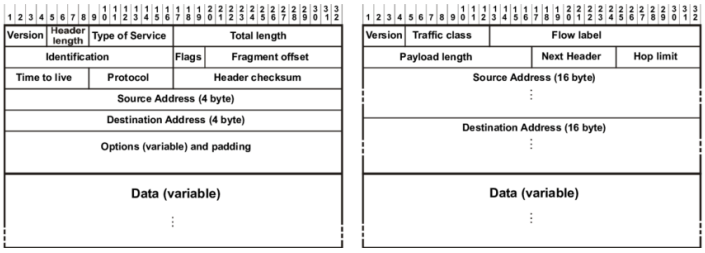


Figura. 5: Exemplo de pacotes IPv4 (esquerda) e IPv6 (direita). Além de seguirem um padrão entre si, o IPv6 foi feito pensando em garantir a compatibilidade com o IPv4, dado que este ainda é bastante comum em muitos sistemas que utilizam o protocolo IP e ainda demorará para desaparecer. É uma ótima exemplificação da necessidade de ser ter um padrão para conseguir comunicar.

Para o caso desta *Application Note*, foi usado um pacote da figura 7.

Esse formato de pacote foi escolhido para ser o mais simples possível, uma vez que a proposta de uso - comunicação peer-to-peer sem garantia de recebimento - não é muito exigente.

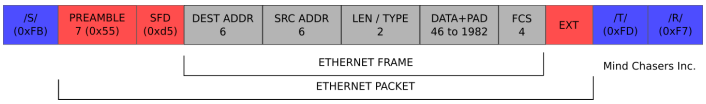


Figura. 6: Exemplo de um pacote *Ethernet*.

| Preâmbulo | Palavra de Início | Tamanho da Mensagem | Payload | CRC |
|---|---|---|---|---|
| 36 bits | 12 bits 0xb38 (111000111000) | 8 bits | Variavel | 32 bits |

Figura. 7: Pacote usado nessa AN.

### Preâmbulo da mensagem

O protocolo adotado nesta aplicação prevê um preâmbulo de 16 bits, sendo 1's e 0's alternados. Tal preâmbulo tem como objetivo normalizar a tensão de entrada do circuito fatiador e preparar o software para o recebimento da mensagem.

Quando o receptor fica parado por muito tempo, o nível médio de tensão na entrada do circuito fatiador é deslocado para longe do limiar de comparação, e a saída se fixa em 1 em em 0. À medida que a sequência de bits alternados chega, o nível médio retorna para o valor ideal para a recepção e o sinal pode ser recebido. A imagem abaixo mostra como o preâmbulo faz esse ajuste. Note como a entrada do comparador (*comparator input*) se encontra deslocada para cima ao início da recepção.

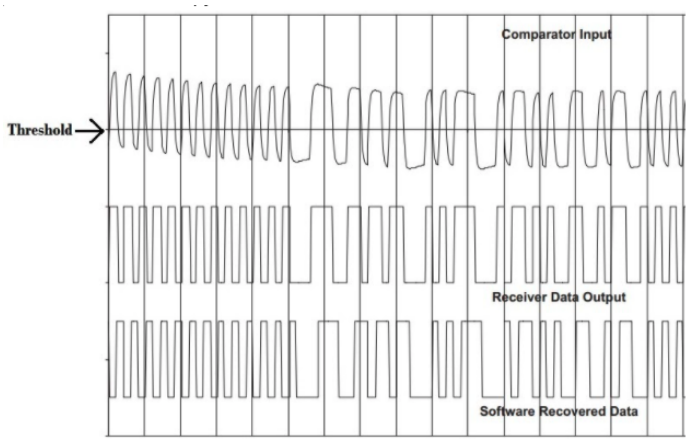


Figura. 8: Preâmbulo ajustando o sinal.

### Codificação do pacote

O pacote da figura 7 foi codificado de forma a permitir que ela seja transmitida pelo ar mantendo um equilíbrio DC, algo necessário para a correta recepção dos dados. Cada nibble (4 bits) da mensagem é codificado num símbolo de 6 bits que contém três 1's e três 0's. Isso mantém a entrada do circuito fatiador próxima do limiar e possibilita a recuperação do sinal.

## CRC

### Definição

O *CRC*, ou *Cyclic Redundancy Check* é um código de detecção de erro e nota-se que ele está presente tanto no pacote da mensagem *Ethernet*, IPv4, IPv6 e na nossa (figuras 6, 5 com o campo *checksum* e figura e 7 com o campo CRC). É um código extremamente importante, pois sendo mensagem de dispositivos ondas eletromagnéticas, essas estão sujeitas a interferências externas que eventualmente podem fazer com que os bites tenham seus valores trocados.

O CRC são representados por polinomiais de n-bits. Um CRC 1-bits é representado por x + 1 (polinômios de primeiro grau), por exemplo, o que indica uma sequência de bits 1. Ou então um CRC de 4 bits representado pelo polinômio $x^4+x+1$ (polinômio de quarto grau) pela sequência de bits 0011. Existem aplicações e aplicações para cada CRC de um polinômio específico. O CRC funciona dividindo um valor de entrada pelo polinômio específico do CRC. Essa divisão gerará um resto. Esse resto é somado ao dividendo. Para averiguar a integridade da mensagem, o divisor somado ao resto deve, então resultar um resto zero, isto é, uma divisão exata.

### Hardware

O STM32F401RE possui uma unidade de cálculo de CRC que utiliza o algoritmo de 32 bits *Ethernet* (polinômio gerador 0x4C11DB7) . Aqui será feita uma breve descrição do hardware e de como chamar as funções da HAL para usar a unidade de CRC no exemplo proposto.

O periférico do CRC possui alguns registradores para controle e para armazenar os dados, possibilitando mais usos do que o mostrado aqui - o mínimo para se usar o hardware. Confira o *Reference Manual* do microcontrolador e a *AN* relativa ao CRC citados neste documento.

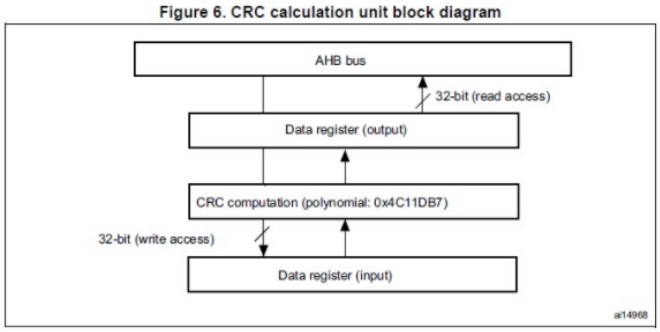


Figura. 9: Diagrama do bloco de calculo do CRC do núcleo F410RE.

Aqui temos um diagrama de blocos mostrando como é feito o cálculo do CRC. O periférico possui apenas um registrador de dados, de 32 bits de largura, que é usado tanto para escrita pelo usuário quanto para leitura do resultado. O importante aqui é observar que só se pode fazer o cálculo do CRC em blocos de 32 bits. Para o cálculo com dados de menos bits é necessário "encaixar" tais dados num espaço de 32 bits e saber interpretar o resultado do CRC.

Quanto ao *driver* da unidade de CRC, serão usadas duas funções: HAL_CRC_Calculate e HAL_CRC_Accumulate.

Antes de usar essas funções é preciso ativar a unidade de CRC no *STMCubeMX* e localizar o *handle* do CRC. A configuração no STMCubeMX já inicia o periférico sem precisar de intervenção do usuário, então basta encontrar o *handle* criado pelo Cube para poder chamar as funções. Ele provavelmente estará neste formato:

```
CRC_HandleTypeDef hcrc;
```

Agora às funções: *Calculate*

```
uint32_t HAL_CRC_Calculate(CRC_HandleTypeDef
    *hcrc, uint32_t pBuffer, uint32_t
    BufferLength)

    /*
Parametros:
CRC_HandleTypeDef * hcrc: ponteiro para o
    handle do CRC
uint32_t pBuffer: ponteiro para o buffer com
    os dados
uint32_t BufferLength: tamanho do buffer em
    palavras de 32 bits

Retorna:
```

```
uint32_t CRC: resultado do calculo

Calcula o CRC considerando o valor inicial padrao 0xffffffff. Faz-se XOR bit-a-bit do valor inicial com o dado (eh como uma soma para o CRC) e divide o resultado pelo polinomio gerador. O valor retornado eh o resto da divisao, mas esse resto continua guardado no registrador para uso posterior.
*/
```

*Accumulate*

```
uint32_t HAL_CRC_Accumulate(CRC_HandleTypeDef *hcrc, uint32_t pBuffer, uint32_t BufferLength)

  /*
Parametros:
CRC_HandleTypeDef * hcrc: ponteiro para o handle do CRC
uint32_t pBuffer: ponteiro para o buffer com os dados
uint32_t BufferLength: tamanho do buffer em palavras de 32 bits

Retorna:
uint32_t CRC: resultado do calculo

O calculo do CRC eh feito tomando o valor anteriormente guardado no registrador como o valor inicial. Eh feito XOR bit-a-bit do valor inicial com o dado e realiza-se a divisao pelo polinomio gerador. O valor retornado eh o resto da divisao, mas esse resto continua guardado no registrador para uso posterior.
*/
```

# Descrição do Firmware

## Implementação do PLL

O PLL consiste num contador de 0 a 159 que estoura a cada período de bit, seguido de um integrador que soma as amostras obtidas dentro daquele período. A cada amostra da entrada (são 8 por período), o PLL deve decidir como incrementar o contador. São três as opções possíveis:

1. não houve mudança na amostra em relação a anterior então o contador é incrementado normalmente com 20;
2. houve mudança na amostra e o contador está abaixo de 80, então o PLL atrasa a contagem e incrementa 11 ao contador;
3. houve mudança na amostra e o contador está acima ou igual a 80 e, nesse caso, o PLL adianta a contagem e incrementa 29 ao contador.

A figura 10 baixo exemplifica a atuação do PLL.

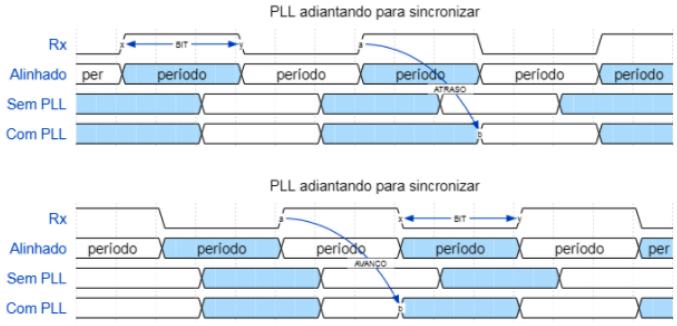


Figura. 10: Exemplo da atuação de um PLL.

Esse procedimento faz com que, em poucos períodos, o receptor esteja alinhado com o transmissor e com que todas as amostras de um período correspondam ao mesmo bit. Os bits de preâmbulo da mensagem são usados para o PLL se alinhar. Ao fim de cada período, o integrador tira a média das amostras, dividindo a soma de 1's e 0's obtidos por 8. Se a média for maior ou igual a 5, o bit é lido como '1', caso contrário, como '0'. Implementação do PLL com integrador em C:

```
/*
   * TRANSICAO_RAMPA = 80
   * INCR_RAMPA_ATRASO = 11
   * INCR_RAMPA_AVANCO = 29
   * INCR_RAMPA = 20
   * COMP_RAMPA = 160
   */
bool rxAmostra;
rxAmostra = HAL_GPIO_ReadPin(Rx_RF_GPIO_Port, Rx_RF_Pin); //amostra do pino Rx, 8 por periodo

if (rxAmostra)
    _rxIntegrador++;
if (rxAmostra != _rxUltAmostra)
{
    //quando o sinal mudou de estado, tem que reavaliar a sincronia
    if (_rxRampa < TRANSICAO_RAMPA)
        _rxRampa += INCR_RAMPA_ATRASO;
    else
        _rxRampa += INCR_RAMPA_AVANCO;
    _rxUltAmostra = rxAmostra;
}
```

```
22 else
23     _rxRampa += INCR_RAMPA; //incremento
       normal
24
25 if (_rxRampa >= COMP_RAMPA) //terminou o
       periodo de um bit
26 {
27     _rxBits >>= 1;
28     // se a media for mais proxima de 1, o
       bit lido eh 1 e colocado em _rxBits
29     if (_rxIntegrador >= 5)
30         _rxBits |= 0x800;
31
32     _rxRampa -= COMP_RAMPA; //reinicia o
       contador
33     _rxIntegrador = 0;
```

## Implementação da codificação 4B6B

Antes de explicar a implementação, a tabela com o *nibble* e seu símbolo correspondente, usada no código desta aplicação.

| Hex | Bin | Símbolo |
|---|---|---|
| 0 | 0000 | 001101 |
| 1 | 0001 | 001110 |
| 2 | 0010 | 010011 |
| 3 | 0011 | 010101 |
| 4 | 0100 | 010110 |
| 5 | 0101 | 011001 |
| 6 | 0110 | 011010 |
| 7 | 0111 | 011100 |
| 8 | 1000 | 100011 |
| 9 | 1001 | 100101 |
| A | 1010 | 100110 |
| B | 1011 | 101001 |
| C | 1100 | 101010 |
| D | 1101 | 101100 |
| E | 1110 | 110010 |
| F | 1111 | 110100 |
| Palavra de Início | | |
| Hex (já codificado) | Símbolo | |
| 0xb38 | 111000111000 | |

Tabela. 1: Tabela com o *nibble* e seu simbolo correspondente.

A codificação monta os bits exatos que serão enviados pelo transmissor ou recebido pelo receptor. Antes de se codificar os dados da mensagem, é preciso montar um cabeçalho com o preâmbulo e a palavra de início. Como explicado anteriormente, o preâmbulo é um conjunto de 1's e de 0's alternados para o alinhamento do *PLL*. Já a palavra de início é um conjunto de dois símbolos que nunca aparecerá em outra parte da mensagem. Só depois de escrito o cabeçalho que se pode começar a escrever a mensagem.

A conversão em símbolos é bastante simples. Como o tamanho de cada espaço de memória é de 8 bits, cada símbolo fica guardado em um byte. Para converter um *nibble*, foi criado um *array* cuja posição i corresponde ao símbolo relativo ao *nibble* i:

```
1  static uint8_t simbolos[] =
2      {
3          0xd,
4          0x3,
5          0x13,
6          0x15,
7          0x16,
8          0x19,
9          0x1a,
10         0x1c,
11         0x23,
12         0x25,
13         0x26,
14         0x29,
15         0x2a,
16         0x2c,
17         0x32,
18         0x34
19     }
```

Para converter, basta ler o *array* usando o *nibble* como índice.

```
1  simbolo =  simbolos[nibble]
```

## Implementação do CRC

Como a mensagem do protocolo é montada com um byte de cada vez, o CRC foi implementado calculando-se a divisão com um byte de cada vez. O cálculo do CRC do pacote segue dois passos:

1. uma chamada da função HAL_CRC_Calculate para iniciar o cálculo com o valor padrão 0xFFFFFFFF.

2. várias chamadas da função HAL_CRC_Accumulate, lendo um byte da mensagem de cada vez e atualizando o CRC a cada chamada. O efeito de chamar a função uma próxima vez é como emendar mais um byte ao final do pacote e calcular o novo CRC do pacote estendido.

O valor obtido na última chamada é o CRC que deve ser anexado ao final do pacote, respeitando a *endianness* do microcontrolador.

Para conferir a integridade da mensagem recebida, basta seguir os dois passos descritos acima, mas agora tratando os últimos 32 bits de CRC adicionados como parte da mensagem: os últimos 4 bytes são reunidos num dado de 32 bits (atenção à endianness) e é feito o CRC final com esse dado. Se o pacote estiver sem erros, o resultado final da função será zero.

## Código com a implementação do CRC

Função para calcular o CRC com um byte de cada vez

```
/*Funcaoo que desloca o byte e calcula o CRC */
uint32_t atualiza_crc_ethernet(uint8_t dados)
{
    uint32_t crc;
    uint32_t dados32[1] = {(uint32_t)dados << 24};

    crc = HAL_CRC_Accumulate(&hcrc, dados32, 1);

    return crc;
}
```

Trecho de código-exemplo calculando o CRC de uma mensagem e anexando ele à mesma.

```
/* Calculando o CRC e anexando no pacote */
uint8_t dados[6] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05}; //dados aguardando o CRC ser anexado
uint8_t bufferEnvio[10];
                //buffer que receber mensagem + CRC
uint32_t crc, aux[1];
uint8_t i;

//primeira chamada com Calculate, iniciando o registrador do CRC com 0xFFFFFFFF
//coloca os 8 bits de dados como MSBits e o restante da palavra como 0
aux[0] = (uint32_t)dados[0] << 24;
crc = HAL_CRC_Calculate(&hcrc, aux, 1);

//varias chamadas da Accumulate, reduzida na funcao atualiza_crc_ethernet
for (i = 1; i < 6; i++)
{
    crc = atualiza_crc_ethernet(dados[i]);
}

//preenche o buffer de envio e anexa CRC
for (i = 0; i < 6; i++)
{
    bufferEnvio[i] = dados[i];
}
for (i = 0; i < 4; i++)
{
    bufferEnvio[6 + i] = (uint8_t)((crc >> i * 8) & 0xff); //processador little endian
}
```

Trecho de código-exemplo para conferir o CRC recebido de uma mensagem (a mesma enviada no trecho acima)

```
/* conferindo o CRC */
//primeiro, atualiza o CRC com os dados
aux[0] = (uint32_t)bufferRecebido[0] << 24;
crc = HAL_CRC_Calculate(&hcrc, aux, 1);
for (i = 1; i < 6; i++)
{
    crc = atualiza_crc_ethernet(bufferRecebido[i]);
}
//Por fim, junta os ultimos 32 bits e chama a funcao Accumulate
crc = HAL_CRC_Accumulate(&hcrc, (uint32_t *)bufferRecebido + 6, 1); //aqui o CRC deve ser zero
```

## Controle de Mensagens

O controle de mensagens depende de vários indicadores (*flags*) e variáveis auxiliares. Esta seção será dividida em controle de envio e em controle de recebimento. Como os trechos de código referentes ao controle são muitos e estão espalhados no programa, não serão mostrados.

### Controle de envio

Como a mensagem a ser enviada não corre risco de interferência por ruído como as que são recebidas, além de outros fatores, o controle de envio é o mais simples. O programa confere se o tamanho da mensagem a ser enviada é válido, ou seja, maior que zero e menor que o máximo do protocolo. Além disso, a função de envio aguarda algum envio anterior que ainda esteja em andamento. Uma vez feito o controle, a mensagem é montada normalmente com a codificação, a contagem de bytes e o CRC.

### Controle de recebimento

O recebimento usa os bits obtidos diretamente do *PLL*, e está sujeito a erros devido à transmissão, por isso, o controle de recebimento é mais complexo. A primeira camada de controle é a verificação da palavra de início antes de se começar a guardar os bits recebidos. Uma vez recebida a palavra de início, é feita uma conferência do tamanho da mensagem - primeiro byte da mensagem contém seu tamanho - para decidir se a mensagem será guardada ou não. Por fim, é feita a verificação com CRC para descobrir se a mensagem está inteira.

## Descrição do Hardware

### Itens

1. A placa utilizada é a Núcleo-F401RE;
2. O cabo utilizado é um cabo USB simples com conector duPont fêmea;
3. Módulos TX/RX - RF 433MHz utilizado para envio e recepção do sinal, ambos alimentados em 5V.

### Setup

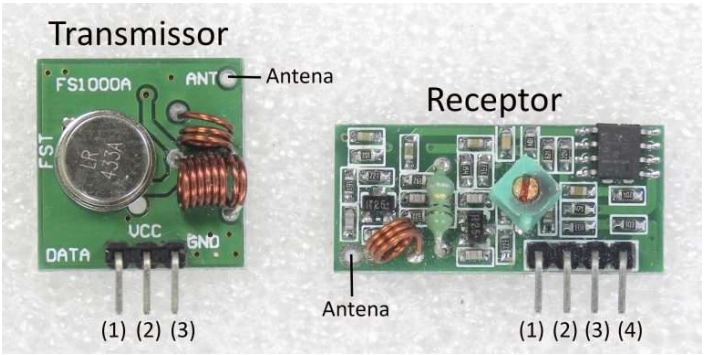


Figura. 11: Pinagem do modulo RF.

Para o transmissor, temos a seguinte pinagem:

1. Data
2. VCC
3. GND

Para o receptor, temos a seguinte pinagem:

1. VCC
2. Data
3. Data (em curto com o pino 2)
4. GND

Para o funcionamento, basta utilizar um resistor de 10kΩ como pull-up conectado a saída do pino transmissor do núcleo; os demais, basta conectar o VCC em 5V, um dos pinos de dado para o pino receptor do núcleo e todos os GND em um GND em comum.

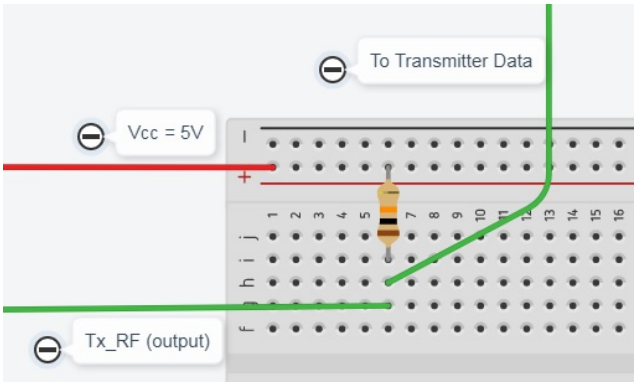


Figura. 12: Circuito do Resitor Pull-Up.

## Exemplo prático

### Guia para montar

Para o uso deste exemplo, necessita-se o software STMCubeMX, o workbench SW4STM32 e do RealTerm (disponível somente para Windows). Como hardware, necessita-se da placa Núcleo STM32F401RE e os módulos e o resistor e fios e protoboard.

### STMCubeMX

Crie um novo projeto selecionando a placa STM32F401RE Nucleo64 no STM32CubeMX. Quando perguntado se deseja inicializar os periférico no modo default, selecione *"Yes"*

Vá para a aba *Clock Configuration* e coloque os valores 64 na caixa *HCLK (MHz)*, aperte Enter para que o STMCubeMX re-configure os valores. Em seguida, coloque o valor 16 na caixa *APB2 Timer Clocks (MHz)*.

Para esse projeto, somente será usado os pinos PC8 e PC9, respectivamente como *GPIO_Input* e *GPIO_Output*.

Em *Pinout and Configuration*, selecione a categoria *Timers* e o *timer TIM1*. No seletor *Clock Source*, escolha *Internal Clock*. Ainda nesse *timer*, em baixo em *Paramters Settings*, coloque *Prescaler (PSC - 16 bits value)* como 0 e *Counter Period (AutoReload Register - 16 bits value)* como 999. Na aba *NVIC Settings*, selecione *TIM1 Update interrupt and TIM10 global interrupt* como *Enabled*.

Agora, iremos selecionar *GPIP* em *Pinout and Configuration*. Selecione o pino PC8 e coloque seu nome como *Rx_RF*. Selecione o pino PC9 e coloque seu nome como *Tx_RF*. Selecione seu *GPIO mode* como *Output Open Drain*.

Vá para a categoria Connectivity, selecione USART2 e verifique se Mode Baud Rate, Word Lenght, Parity e Stop Bits estão respectivamente com os valores Assynchronous, 115200 Bits/s, 8 bits (including parity), None e 1.

Vá para a categoria Computing e ative o CRC clicando na caixa de seleção.

Vá para Project Manager, coloque um nome para o projeto, escolha como IDE o SW4STM32.

### SW4STM32

Abra o projeto gerado na última seção. Obtenha os arquivos *RFModSTM32f4x.c* e *RFModSTM32f4x.h* os arquivos a partir do repositório do trabalho. Os arquivos encontram-se na pasta RFModSTM32f4x. Coloque o .c na pasta Src e o .h na pasta Inc.

Após fazê-lo, clique com o botão direito em cima do nome do projeto e selecione a opção *Refresh* no menu aberto. Insira o .h recentemente incluído nos arquivos *main.c* e *stm32f4xx_it.c* que se encontram na pasta Src. por meio de um include.

No arquivo *stm32f4xx_it.c*, vá para a função *TIM1_UP_TIM10_IRQHandler* e adicione a função *TimerRotinaInterrupcao()* na seção *USER CODE BEGIN TIM1_UP_TIM10_IRQHandler 0*. Finalmente chame as funções *HAL_TIM_Base_Start_IT()* e *Inicia_Mod_RF()* no *main.c* na seção *Initialize all configured peripherals*.

## Referências

Datasheet STM32F401xD/E (https://www.st.com). Acesso em 08/03/2021

RM0368 Reference manual STM32F401xB/C and STM32F401xD/E (https://www.st.com). Acesso em 08/03/2021

UM1725 Description of STM32F4 HAL and low-layer drivers (https://www.st.com). Acesso em 08/03/2021

AN 4187 Using the CRC peripheral in the STM32 family (https://www.st.com). Acesso em 08/03/2021

A Painless Guide to CRC Error Detection Algorithms (http://ross.net/crc/crcpaper.html). Acesso em 06/03/2021

ASH Transceiver Software Designer's Guide
É o documento que descreve o protocolo usado. Disponível em (https://wireless.murata.com). Acesso em 08/03/2021

Amplitude-shift keying (ASK) (https://tinyurl.com/2tua2hu9). Acesso em 08/03/2021

Tabela. 2: Revision history.

| Revisão | Autores | Mudanças |
|---|---|---|
| AN-1 | E. Villani e G. Gomes | Primeira versão |