DISCIPLINA EA701
Introdução aos Sistemas Embarcados

ROTEIRO 11: REDE DE CONTROLE
Topologia de Rede, Barramento CAN, Protocolo CAN, *Transceiver* MCP2551

Profs. Antonio A. F. Quevedo e Wu Shin-Ting

FEEC / UNICAMP

Revisado em outubro de 2024

INTRODUÇÃO

Nos Roteiros 7 e 10, exploramos três protocolos de comunicação em sistemas embarcados, com foco em distâncias relativamente curtas e em ambientes com menor suscetibilidade a interferências. No entanto, à medida que avançamos, percebemos a crescente necessidade de interligar componentes a longas distâncias, garantindo altas taxas de transmissão e operando eficazmente em ambientes repletos de ruído elétrico. Um exemplo significativo é a comunicação entre subsistemas digitais em veículos, além da conexão entre elementos de iluminação e efeitos em cenários cênicos, onde a robustez e a confiabilidade da comunicação são fundamentais.

Para atender a essas demandas complexas, frequentemente utilizam-se **redes** de elementos, onde múltiplos nós—unidades de um sistema digital—compartilham dados através de um único meio físico. Nesses ambientes, é essencial que haja uma coordenação eficiente da troca

de mensagens entre os nós, mediada por **protocolos de comunicação** que garantam a integridade e a rapidez das informações transmitidas.

Neste Roteiro, daremos um passo adiante ao aprofundar nosso conhecimento sobre o **protocolo CAN** (do inglês, *Controller Area Network*). Este protocolo é especificamente projetado para a transmissão de dados em distâncias médias e utiliza **linhas de transmissão diferencial**, o que o torna ideal para cenários críticos e com alta demanda de comunicação. Ao longo do roteiro, apresentamos três projetos-exemplo, que incluem a implementação da função de *loopback*, a filtragem de identificadores e a montagem de uma rede CAN. Veremos, através da prática, como esses conceitos se aplicam em situações do mundo real.

PROJETOS-EXEMPLO

Nesta seção, exploraremos o barramento CAN com três projetos que vão progressivamente introduzir conceitos importantes sobre o mesmo.

Projeto de comunicação CAN usando *loopback*

Você já pensou em como testar e diagnosticar a operação de uma comunicação sem depender de ligações de *transceivers* por um par diferencial, como em uma comunicação CAN? Uma abordagem eficaz para verificar a comunicação em um barramento é o uso do recurso de ***loopback***. Esse recurso consiste em estabelecer uma conexão direta entre os sinais Tx e Rx do controlador, que envia uma mensagem para si mesmo, permitindo uma verificação interna da comunicação. Esse procedimento permite que, durante a transmissão, o controlador mestre monitore se os níveis lógicos presentes na linha coincidem com os dados enviados. Sem a conexão entre Tx e Rx, o controlador pode interpretar a falta de resposta como uma colisão no barramento, levando ao abortamento da transmissão. Assim, o *loopback* não só facilita testes sem necessidade de equipamentos adicionais, mas também assegura que o controlador funcione corretamente, detectando eventuais falhas na comunicação.

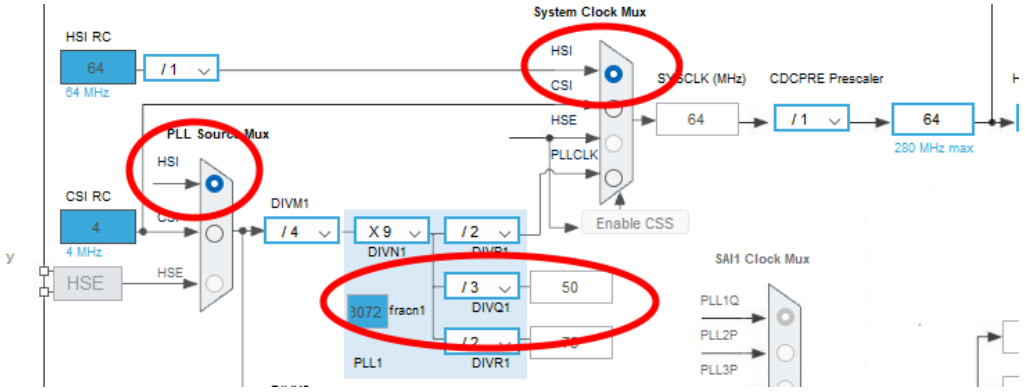
Para ilustrar essa funcionalidade, vamos desenvolver um projeto utilizando um módulo CAN do STM32H7A3. O STM32H7A3 possui 2 controladores CAN do tipo *Flexible-Datarate*, mas vamos usar apenas um deles, em modo padrão. Os controladores podem ser configurados para o modo *loopback* interno ou externo, ou seja, eles não precisam de intervenção física para realizar a conexão entre seus sinais. No modo **interno,** CAN_Tx e CAN_Rx são ligados um ao outro, e ambos são desligados de seus pinos no encapsulamento do microcontrolador. No modo **externo**, o pino CAN_Tx permanece ligado a seu pino correspondente. Vamos usar o *loopback* externo para que se possa visualizar os sinais do controlador, conectando-se o analisador lógico ao pino CAN_Tx..

Devido à complexidade do controlador CAN, utilizaremos as facilidades do CubeMX e as funções HAL para a programação. Paralelamente, fornecemos instruções análogas utilizando as macros CMSIS, destacadas em azul, para que seja possível compreender as nuances da programação desse controlador.

1. Crie um projeto “CAN_Base”, **sem inicializar os periféricos**. Na aba *Pinout & Configuration*, ative o *Debug*. Dentro do grupo “Connectivity”, selecione “FDCAN2” e no painel “Mode”, selecione a caixa “Activated”. Agora temos o FDCAN2 ativado. É necessário

ativar o FDCAN2 **antes** de configurar o PLL1, pois neste projeto o controlador é o único módulo que usa o *clock* gerado pelo PLL1 em seu *kernel*. Sem ter pelo menos um módulo ativado usando um determinado sinal de *clock*, o *CubeMX* não permite ajustar os parâmetros que determinam aquele sinal.

2. Vá para a aba *Clock Configuration*. Aqui vamos ajustar o *clock* do sistema para 64MHz e o do CAN para 50MHz, usando o gerador de *clock interno*. Na figura abaixo, vemos a seleção do HSI tanto para o *clock* geral como para a entrada dos PLL:



Note ainda dentro do bloco do PLL1 que temos o valor de “fracn1” ajustado para 3072, e o valor de “DIVQ1” ajustado para 3, ficando a saída de DIVQ1 com o valor de 50MHz. Ou seja, DIVM1, DIVN1, DIVQ1 e fracn1 devem ser tais que pll1_q_ck seja igual a 50MHz. O “Device Configuration Tool Code Generation” gera um código análogo ao seguinte se FDCAN2 estiver ativado.

```
{
        //Configura o sinal de relogio pll1_q_ck para fdcan_ker_clk
        //A fonte HSI ja eh habilitada e configurada pelo MxCube
        //Desabilita PLL1
        RCC->CR &= ~RCC_CR_PLL1ON;
        while (RCC->CR & RCC_CR_PLL1RDY); //aguarda a desabilitacao
        //Configura pre-scaler e DIVM1
        RCC->PLLCKSELR &= ~(RCC_PLLCKSELR_PLLSRC |
                RCC_PLLCKSELR_DIVM1);
        RCC->PLLCKSELR |= RCC_PLLCKSELR_DIVM1_2; // 0x4
        //Configura os fatores de divisoes/multiplicacoes
        RCC->PLL1DIVR &= ~(RCC_PLL1DIVR_N1 |
                RCC_PLL1DIVR_Q1);
        RCC->PLL1DIVR |= (((12U-1U) << RCC_PLL1DIVR_N1_Pos) |
                ((2U-1U) << RCC_PLL1DIVR_Q1_Pos));
        //Configura o fator frcionario
        RCC->PLL1FRACR = (4096U << RCC_PLL1FRACR_FRACN1_Pos);
        //Faixa de frequencias de operacao de PLL1 ( 8 a 16 MHz)
        RCC->PLLCFGR |= RCC_PLLCFGR_PLL1RGE;
        //Faixa de frequencia de VCO usado por PLL1 (128 a 560 MHz)
        RCC->PLLCFGR &= ~RCC_PLLCFGR_PLL1VCOSEL;
        //Habilita as configuracoes dos divisores
        RCC->PLLCFGR |= (RCC_PLLCFGR_PLL1FRACEN |
                        RCC_PLLCFGR_DIVQ1EN );
        // Reativar PLL1
        RCC->CR |= RCC_CR_PLL1ON;
        while (!(RCC->CR & RCC_CR_PLL1RDY)); //aguarda a ativacao
}
```

Observe que as configurações referentes ao PLL1 só são permitidas quando o relógio PLL1 está desabilitado. Primeiro, o *bit* `RCC_PLLCKSELR_PLLSRC` é usado para selecionar a fonte de *clock* HSI para o PLL. Em seguida, são definidos os fatores de divisão e o fator fracionário para o *clock* de entrada do PLL. É importante escolher a faixa de frequências de operação do PLL1, que deve estar entre 8 e 16 MHz, e a faixa de frequência do VCO (do inglês, *Voltage Controlled Oscillator*) utilizado pelo PLL1, normalmente entre 128 e 560 MHz. Por fim, é necessário habilitar as configurações dos divisores antes de reativar o PLL1.

2. Ainda na aba *Clock Configuration* use as barras de deslizamento para levar a figura a seu canto inferior direito. Ali temos o controle do *clock* dos FDCAN. Confirme que foi selecionada a opção do meio (PLL1Q), com a frequência de 50MHz para os módulos.

3. Voltando novamente à aba *Pinout & Configuration,* vamos configurar o FDCAN2. Olhando o mapa dos pinos, podemos encontrar o pino PB13 e ver que ele foi configurado como "FDCAN2_TX", o que está correto para a placa auxiliar. Entretanto, na placa auxiliar o pino reservado para "FDCAN2_RX" é PB5, e pode-se ver que PB12 foi alocado para esta função. Vá até o pino PB5, clique-o com o botão esquerdo e selecione a opção "FDCAN2_RX". Assim, os sinais do FDCAN2 estão ligados aos pinos correspondentes no *header* H7 (CAN2).

O código equivalente ao gerado pelo "Device Configuration Tool Code Generation" segue-se

```
{
	//Ativar clock gating de FDCAN e dos perifericos necessarios a sua operacao
	// Seleciona ker_ck para FDCAN (pll1_q_ck)
	RCC->PLLCFGR |= RCC_PLLCFGR_DIVQ1EN_Msk;   // habilita pll1_q_ck
	RCC->CDCCIP1R &= RCC_CDCCIP1R_FDCANSEL_Msk;  // seleciona a fonte para FDCAN
	RCC->CDCCIP1R |= RCC_CDCCIP1R_FDCANSEL_0;
	// Habilita o clock do FDCAN2
	RCC->APB1HENR |= RCC_APB1HENR_FDCANEN;
	/** Configura os pinos FDCAN2 GPIO
	   PB13     ------> FDCAN2_TX
	   PB5      ------> FDCAN2_RX
	 */
	RCC->AHB4ENR |= (RCC_AHB4ENR_GPIOBEN);
	// Configura o pino FDCAN2_TX
	GPIOB->MODER &= ~(GPIO_MODER_MODE13_Msk); // Limpar modos
	GPIOB->MODER |= GPIO_MODER_MODE13_1; // Modo alternativo
	// Seleciona a função alternativa FDCAN2_TX (AF9) para o pino
	GPIOB->AFR[1] &= ~(GPIO_AFRH_AFSEL13_Msk); // AF9 para PB13
	GPIOB->AFR[1] |= (GPIO_AFRH_AFSEL13_0); // AF9 para PB13
	GPIOB->AFR[1] |= (GPIO_AFRH_AFSEL13_3); // AF9 para PB13
	GPIOB->OSPEEDR &= ~(GPIO_OSPEEDR_OSPEED13_Msk); //Baixa velocidade
	// Configurar o pino FDCAN2_RX
	GPIOB->MODER &= ~(GPIO_MODER_MODE5_Msk); // Limpar modos
	GPIOB->MODER |= GPIO_MODER_MODE5_1; // Modo alternativo
	// Selecionar a função alternativa FDCAN2_RX (AF9) para o pino
	GPIOB->AFR[0] &= ~(GPIO_AFRL_AFSEL5_Msk); // AF9 para PB5
	GPIOB->AFR[0] |= (GPIO_AFRL_AFSEL5_0); // AF9 para PB5
	GPIOB->AFR[0] |= (GPIO_AFRL_AFSEL5_3); // AF9 para PB5
	GPIOB->OSPEEDR &= ~(GPIO_OSPEEDR_OSPEED5_Msk); // Baixa velocidade
}
```

Além de habilitar as fontes de *clock* do PLL1 e selecionar a fonte de *clock* do FDCAN, também é ativada a fonte de *clock* para o FDCAN e para o GPIOB, que controla os pinos PB5 e PB13. Esses pinos são configurados para a função alternativa AF9.

4. No painel “Configuration”, vamos ajustar os parâmetros do FDCAN2.

Para o FDCAN, isso significa que ele foi colocado em modo de inicialização e que a configuração foi habilitada, conforme ilustrado pelo seguinte bloco de código análogo ao gerado pela ferramenta “Device Configuration Tool Code Generation”:

```
{
        //Chaveia para o modo de inicializacao
        //Sair do modo Sleep
        FDCAN2->CCCR &= ~FDCAN_CCCR_CSR;
        /* Check Sleep mode acknowledge */
        while ((FDCAN2->CCCR & FDCAN_CCCR_CSA) == FDCAN_CCCR_CSA);
        // Coloca o FDCAN em modo de inicialização
        FDCAN2->CCCR |= FDCAN_CCCR_INIT; // Ativa o modo de inicialização
        // Aguarda até que o modo de inicialização seja efetivo
        while (!(FDCAN2->CCCR & FDCAN_CCCR_INIT));
        //Configuracao dos registradores de FDCAN propriamente dito
        FDCAN2->CCCR |= FDCAN_CCCR_CCE_Msk;  // Habilita a configuração
}
```

Observe o laço de espera para cada configuração, que aguarda a conclusão da ação antes de prosseguir para a próxima.

Siga as figuras abaixo para os valores configurados:

Basic Parameters	
Frame Format	Classic mode
Mode	External LoopBack mode
Auto Retransmission	Disable
Transmit Pause	Disable
Protocol Exception	Disable
Nominal Sync Jump Width	13
Data Prescaler	25
Data Sync Jump Width	1
Data Time Seg1	2
Data Time Seg2	1

Ajustamos o FDCAN2 para usar quadros de formato padrão (“Classic”), e ativamos o modo *external loopback*. A auto retransmissão em caso de erros está desabilitada. Os parâmetros iniciados com “Data” são temporizações, que dependem inclusive do atraso gerado por *transcievers*. Um código análogo ao gerado pelo “Device Configuration Tool Code Generation” corresponde a:

```
{
        //Configuracao dos registradores de FDCAN propriamente dito
        FDCAN2->CCCR |= FDCAN_CCCR_CCE_Msk;  // Habilita a configuração
        // Desabilita retransmissao automatica
        FDCAN2->CCCR |= FDCAN_CCCR_DAR_Msk;
        // Desabilita o recurso de pausa de transmissao
        FDCAN2->CCCR &= ~FDCAN_CCCR_TXP_Msk;
        // Desabilita protocolo de processamento de excecao
        FDCAN2->CCCR |= FDCAN_CCCR_PXHD_Msk;
        // Defina o formato de quadro/campo de dados (Quadro Classico)
        FDCAN2->CCCR &= ~(FDCAN_CCCR_FDOE |
                          FDCAN_CCCR_BRSE);
        // Defina o modo de operacao (Loopback externo)
```

```
        FDCAN2->CCCR &= ~(FDCAN_CCCR_TEST |
                          FDCAN_CCCR_MON |
                          FDCAN_CCCR_ASM);
        FDCAN2->TEST &= ~FDCAN_TEST_LBCK;
        FDCAN2->CCCR |= FDCAN_CCCR_TEST;
        FDCAN2->TEST |= FDCAN_TEST_LBCK;
}
```

Message Ram Offset	0
Std Filters Nbr	1
Ext Filters Nbr	0
Rx Fifo0 Elmts Nbr	1
Rx Fifo0 Elmt Size	8 bytes data field
Rx Fifo1 Elmts Nbr	0
Rx Fifo1 Elmt Size	8 bytes data field
Rx Buffers Nbr	0
Rx Buffer Size	8 bytes data field
Tx Events Nbr	0
Tx Buffers Nbr	0
Tx Fifo Queue Elmts Nbr	1
Tx Fifo Queue Mode	FIFO mode
Tx Elmt Size	8 bytes data field
⊟ Clock Calibration Unit	
Clock Calibration	Disable
⊟ Bit Timings Parameters	
Nominal Prescaler	1
Nominal Time Quantum	20.0 ns
Nominal Time Seg1	86
Nominal Time Seg2	13
Nominal Time for one Bit	2000 ns
Nominal Baud Rate	500000 bit/s

O parâmetro “Message Ram Offset” determina qual o endereço inicial da RAM dedicada que será usado para as mensagens deste módulo. Este endereço, na verdade, representa um *offset* em relação ao endereço base da memória do módulo FDCAN, conhecido como *CAN Message RAM*, que está mapeada no intervalo de 0x4000AC00 a 0x40003DFF do espaço de endereçamento da CPU. Temos ainda a definição de 1 “filtro” padrão, 1 elemento na FIFO0 com tamanho de 8 *bytes*, que é um valor padrão. O valor de “Tx Fifo Queue Elmts Nbr” define o número de elementos da fila do *buffer* de Tx, com um elemento de 8 *bytes*. Por fim, os “*Bit Timings Parameters*” concluem a configuração da temporização nominal de *bits*, e pode-se ver na última linha que a *baud rate* nominal estabelecida é de 500.000 *bits* por segundo.

Estes valores de configuração são atribuídos aos registradores pelos códigos gerados por “Device Configuration Tool Code Generation”, que são análogos ao seguinte bloco de instruções:

```
{
        // Configura tamanho do campo/quaudro/frame de dados para TX e RX
        // Define o uso do buffer TxFIFO para transmissao
        FDCAN2->TXBC &= ~FDCAN_TXBC_TFQM_Msk;
        // Configurar o tamanho de cada campo/quadro/frame de dados de uma mensagem TX (8 bytes)
        FDCAN2->TXESC &= ~FDCAN_TXESC_TBDS_Msk;
        // Configura tamanho de cada campo de dados de uma mensagem RX (8 bytes)
        FDCAN2->RXESC &= ~FDCAN_RXESC_F0DS_Msk;
}
{
        //Configura enderecos iniciais de cada segmento em SRAMCAN
```

```
        //Cada unidade do segmento tem 32 bits (4U bytes)
        //RX FIFO 0: 1 mensagem de 8 bytes de dados + 8 bytes de cabecalho
        //TX FIFO: 1 mensagem de 8 bytes de dados + 8 bytes de cabecalho
        uint32_t Origem = 0;
        /* Endereco inicial da lista de filtros padrao  */
        MODIFY_REG(FDCAN2->SIDFC, FDCAN_SIDFC_FLSSA, (Origem << FDCAN_SIDFC_FLSSA_Pos));
        /* Quantidade de elementos (32 bits) na lista */
        MODIFY_REG(FDCAN2->SIDFC, FDCAN_SIDFC_LSS, (1U << FDCAN_SIDFC_LSS_Pos));
        /* Endereco inicial de Rx FIFO 0 */
        Origem += (1U);
        MODIFY_REG(FDCAN2->RXF0C, FDCAN_RXF0C_F0SA, (Origem << FDCAN_RXF0C_F0SA_Pos));
        /* Quantidade de elementos ((2+ n dados em bytes) * 32 bits) Rx FIFO 0 */
        MODIFY_REG(FDCAN2->RXF0C, FDCAN_RXF0C_F0S, (1U << FDCAN_RXF0C_F0S_Pos));
        /* Endereco inicial de Tx buffer  */
        Origem += (4U);
        MODIFY_REG(FDCAN2->TXBC, FDCAN_TXBC_TBSA, (Origem << FDCAN_TXBC_TBSA_Pos));
        /* Quantidade de elementos Tx FIFO/queue */
        MODIFY_REG(FDCAN2->TXBC, FDCAN_TXBC_TFQS, (1U << FDCAN_TXBC_TFQS_Pos));
        /* Zera o conteudo da area reservada na memoria de mensagens CAN */
        for (uint32_t RAMcounter = SRAMCAN_BASE; RAMcounter < SRAMCAN_BASE+(1U+4U+4U)*4U; RAMcounter += 4U)
        {
                *(uint32_t *)(RAMcounter) = 0x00000000;
        }
}
{
        // Configura o tempo de amostragem e prescalers para o tempo de bit
        // nominal (Formato de quadro classico)
        // Configuracao de tempo de bit de dados apenas no modo FD + BRS.
        FDCAN2->NBTP = ((((uint32_t)13 - 1U) << FDCAN_NBTP_NSJW_Pos) |
                (((uint32_t)86U - 1U) << FDCAN_NBTP_NTSEG1_Pos)   | // Nominal Time Segment 1
                (((uint32_t)13U - 1U) << FDCAN_NBTP_NTSEG2_Pos)   | // Nominal Time Segment 2
                (((uint32_t)1U - 1U) << FDCAN_NBTP_NBRP_Pos));    // Nominal Bit Rate Prescaler
}
```

Além de configurar o tamanho dos elementos nos *buffers* de recepção e transmissão, são alocados espaços na memória de mensagens CAN para os diferentes *buffers* suportados pelo módulo FDCAN e a lista de filtros de identificadores. O endereço inicial e o tamanho de cada segmento de memória são configurados nos registradores. Por fim, é necessário definir os tempos de *bit* nominal e de dados nos registradores `FDCAN_NBTP` e `FDCAN_DBTP`, respectivamente. Esses tempos são relacionados com a taxa de transmissão do FDCAN.

5. Entre na configuração do NVIC e ative a interrupção “FDCAN2 interrupt 0”, ajustando a prioridade para 1. As instruções subjacentes correspondem ao seguinte bloco de instruções

```
{
        /* Habilita IRQ correspondente a FDCAN2 interrupt Init (Linha de interrupção 0) */
        NVIC_SetPriority(FDCAN2_IT0_IRQn, 0);
        NVIC_EnableIRQ(FDCAN2_IT0_IRQn);
}
```

Se quisermos que todas as instruções de inicialização sejam geradas pelo “Device Configuration Tool Code Generation”, basta salvar e gerar o código com o **FDCAN ativado**. O *MxCube* criará a função de inicialização `MX_FDCAN2_Init` e inserirá sua chamada na

função `main`. Caso prefiramos implementar nossa própria versão, podemos criar no escopo /* USER CODE BEGIN 4 */ uma função `FDCAN2_Init()` cujo corpo será formado pelas instruções descritas nos itens 2 a 5, e inserir a sua chamada no escopo /* USER CODE BEGIN 2 */.

6. No arquivo "main.c", vamos precisar de algumas variáveis globais. Na seção `/* USER CODE BEGIN PV */`, declare as variáveis:

```
FDCAN_TxHeaderTypeDef   TxHeader;
FDCAN_RxHeaderTypeDef   RxHeader;
uint8_t               TxData[8];
uint8_t               RxData[8];
int indx = 0;
```

Para a robustez e legibilidade do código, a HAL se comunica utilizando tipos de dados `struct` redefinidos com o nome **`Header`**. Usaremos neste projeto duas `structs` definidas na HAL, `FDCAN_TxHeaderTypeDef` e `FDCAN_RxHeaderTyedef` para definir duas variáveis que armazenam as informações dos quadros de transmissão e de recepção. Além disso, definimos dois vetores para armazenar os 8 *bytes* do campo de dados do quadro, na transmissão e na recepção e uma variável para ajudar a montar as sequências de *bytes* a serem enviadas.

No caso de um projeto com interface CMSIS, a comunicação é realizada diretamente por meio das variáveis. Assim, apenas as três últimas variáveis, destacadas em azul, devem ser adicionadas.

7. Precisamos agora iniciar o FDCAN2, ativar as notificações de mensagens recebidas, e configurar o *header* de transmissão com as macros definidas para o *driver* do *firmware* de FDCAN na HAL. Este será configurado uma única vez, pois os parâmetros do quadro de transmissão serão sempre os mesmos. Na seção `/* USER CODE BEGIN 2 */`, escreva o código:

```
if(HAL_FDCAN_Start(&hfdcan2)!= HAL_OK) {
      Error_Handler();
}
if (HAL_FDCAN_ActivateNotification(&hfdcan2, FDCAN_IT_RX_FIFO0_NEW_MESSAGE,
0) != HAL_OK) {
      /* Notification Error */
      Error_Handler();
}

TxHeader.Identifier = 0x11;
TxHeader.IdType = FDCAN_STANDARD_ID;
TxHeader.TxFrameType = FDCAN_DATA_FRAME;
TxHeader.DataLength = FDCAN_DLC_BYTES_8;
TxHeader.ErrorStateIndicator = FDCAN_ESI_ACTIVE;
TxHeader.BitRateSwitch = FDCAN_BRS_OFF;
TxHeader.FDFormat = FDCAN_CLASSIC_CAN;
TxHeader.TxEventFifoControl = FDCAN_NO_TX_EVENTS;
TxHeader.MessageMarker = 0;
```

Note que no *header* de transmissão temos definido o ID da transmissão. Cada mensagem CAN tem um identificador, geralmente associado ao nó que a enviou. Assim cada nó que recebe a mensagem sabe sua origem. Este ID é do tipo padrão de 11 *bits* (`FDCAN_STANDARD_ID`) e o quadro é de dados com 8 *bytes* (`FDCAN_DATA_FRAME`, `FDCAN_DLC_BYTES_8`). O tipo de quadro é padrão clássico, sem o uso de *Bitrate Stitch*, característico no modo *Fast Datarate* (`FDCAN_CLASSIC_CAN`, `FDCAN_BRS_OFF`).

É importante notar que as funções HAL escrevem valores nos campos de *bits* dos registradores, como fizemos em projetos anteriores manualmente. As funções calculam os valores e posições de todos os grupos de *bits* dos registradores e escreve nos mesmos. Algumas funções também leem valores de registradores para retornar valores ao programador. Os campos das *structs* usadas nas funções contêm os parâmetros a serem configurados nos registradores de periféricos. As instruções correspondentes às duas funções `HAL_FDCAN_Start` e `HAL_FDCAN_ActivateNotification` são, respectivamente:

```
{
	// Sai do modo de inicialização
	FDCAN2->CCCR &= ~FDCAN_CCCR_INIT; // Desativa o modo de inicialização
	// Aguarda até que o modo de inicialização seja efetivo
	while ((FDCAN2->CCCR & FDCAN_CCCR_INIT));
}
{
	// Habilita a linha de interrupcao 0
	FDCAN2->ILE |= FDCAN_ILE_EINT0;  // habilita a linha de interrupcao
	FDCAN2->IE |= FDCAN_IE_RF0NE;    // habilita evento para novas mensagens na Rx FIFO 0
	FDCAN2->ILS &= ~FDCAN_ILS_RF0NL;    //associa a linha de interrupcao 0
}
```

Essas funções devem ser inseridas após a chamada da nossa função privada implementada `FDCAN2_Init`.

8. Estando configurado o CAN, vamos entrar em um *loop* que a cada segundo envia um quadro de 8 *bytes* em sequência. No primeiro quadro, os valores dos *bytes* vão de 0 a 7; no segundo quadro, de 8 a 15, e assim em diante. Abaixo da linha `/* USER CODE BEGIN 3 */`, escreva o código:

```
for (int i=0; i<8; i++) {
	TxData[i] = indx++;
}

if (HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan2, &TxHeader, TxData)!= HAL_OK) {
	Error_Handler();
}

HAL_Delay (1000);
```

O *loop* monta o vetor de *bytes* e usa a função `HAL_FDCAN_AddMessageToTxFifoQ` para colocar a mensagem na FIFO de transmissão. Depois, espera 1 segundo antes de preparar o próximo vetor de transmissão.

Ao invés de usar a função `HAL_FDCAN_AddMessageToTxFifoQ` com os argumentos dos tipos de dados definidos na HAL, podemos definir, implementar nossa própria função,

utilizando as macros da CMSIS no escopo /* USER CODE BEGIN 4 */, e substituir a chamada de `HAL_FDCAN_AddMessageToTxFifoQ` por ela.

```
void FDCAN2_AddMessageToTxFifoQ (uint32_t id) {
	uint32_t PutIndex;
	uint32_t TxElementW1;
	uint32_t TxElementW2;
	uint32_t *TxAddress;
	uint32_t ByteCounter;
	if ((FDCAN2->TXBC & FDCAN_TXBC_TFQS) && !(FDCAN2->TXFQS & FDCAN_TXFQS_TFQF)) {
		// Le o indice do buffer TxFIFO/Queue
		PutIndex = (FDCAN2->TXFQS & FDCAN_TXFQS_TFQPI) >> FDCAN_TXFQS_TFQPI_Pos;
		// Monta a primeira palavra do cabecalho
		TxElementW1 = ((0x0 << 31) | //FDCAN_ESI_ACTIVE
				(0b0 << 30) | // FDCAN_STANDARD_ID
				(0b0 << 29) | // FDCAN_DATA_FRAME
				(id << 18U));  //Identifier padrao nos bits [28:18]
		// Monta a segunda palavra do cabecalho
		TxElementW2 = ((0x00 << 24U) |      // Message Marker
				(0b0 << 23U) | // FDCAN_NO_TX_EVENTS
				(0b0 << 21U) | // FDCAN_CLASSIC_CAN
				(0b0 << 20U) | // FDCAN_BRS_OFF
				(0b1000 << 16U)); // DataLength (8-bytes)
		/* Calcula o endereco do buffer TxFIFO/Queue */
		TxAddress = (uint32_t *)(SRAMCAN_BASE+(1U+4U)*4U +
				(PutIndex*((FDCAN2->TXBC & FDCAN_TXBC_TFQS) >> FDCAN_TXBC_TFQS_Pos)*4U));
		/* Escreve 2 palavras de cabecalho na RAM de mensagens CAN */
		*TxAddress = TxElementW1;
		TxAddress++;
		*TxAddress = TxElementW2;
		TxAddress++;
		/* Escreve dados da mensagem */
		ByteCounter = Code2ByteCounter
				((FDCAN2->TXESC & FDCAN_TXESC_TBDS) >> FDCAN_TXESC_TBDS_Pos);
		for (uint8_t i = 0; i < ByteCounter; i += 4)
		{
		  *TxAddress = (((uint32_t)TxData[i + 3U] << 24U) |
				((uint32_t)TxData[i + 2U] << 16U) |
				((uint32_t)TxData[i + 1U] << 8U)  |
				(uint32_t)TxData[i]);
			TxAddress++;
		}
		/* Ativa requisicao de transmissao do elemento Index no TxFIFO/Queue */
	    FDCAN2->TXBAR = ((uint32_t)1 << PutIndex);
	}
}
```

Essencialmente, a função monta mensagens com identificador `id` no buffer TX a partir dos dados do vetor TxData, de acordo com a estrutura definida no Manual. Em seguida, armazena essas mensagens no endereço reservado na memória de mensagens CAN previamente alocado, antes de solicitar a a transmissão ao enviar o índice `PutIndex` da mensagem. Para reuso, as instruções de conversão do código adotado pelo fabricante para o tamanho de dados em valor decimal computável são separadas na seguinte função que deve ser incluída no escopo /* USER CODE BEGIN 4 */.

```
uint8_t Code2ByteCounter(uint8_t code) {
	uint8_t ByteCounter = 0;
	switch (code) {	// Do codigo binario para ...
	case 0:
		ByteCounter = 8;
		break;
	case 1:
		ByteCounter = 12;
		break;
	case 2:
		ByteCounter = 16;
		break;
	case 3:
		ByteCounter = 20;
		break;
	case 4:
		ByteCounter = 24;
		break;
	case 5:
		ByteCounter = 32;
		break;
	case 6:
		ByteCounter = 48;
		break;
	case 7:
		ByteCounter = 64;
		break;
	}
  return (ByteCounter);
}
```

9. Falta ainda tratar as mensagens recebidas. Como a interrupção de FDCAN foi habilitada, o *CubeMX* implementa o código da IRQ correspondente, que limpa o *flag* de interrupção e realiza uma chamada para uma função de *callback* definida com o nome "HAL_FDCAN_RxFifo0Callback". Na seção `/* USER CODE BEGIN 4 */`, implemente o código para a função de *callback*:

```
void HAL_FDCAN_RxFifo0Callback(FDCAN_HandleTypeDef *hfdcan, uint32_t
RxFifo0ITs) {
	if((RxFifo0ITs & FDCAN_IT_RX_FIFO0_NEW_MESSAGE) != RESET) {
		/* Retreive Rx messages from RX FIFO0 */
		if (HAL_FDCAN_GetRxMessage(hfdcan, FDCAN_RX_FIFO0, &RxHeader,
RxData) != HAL_OK) {
			/* Reception Error */
			Error_Handler();
		}
	}

}
```

Inicialmente, a função verifica a causa da interrupção. Se for de nova mensagem recebida na FIFO (`FDCAN_IT_RX_FIFO0_NEW_MESSAGE`), ela chama a função `HAL_FDCAN_GetRxMessage` para receber a mensagem e guardar os dados do cabeçalho em `RxHeader` e os dados recebidos em `RxData`.

Isso é equivalente a inserir no escopo /* USER CODE BEGIN 1 */ do arquivo `stm32h7xx_it.c` a seguinte rotina de serviço em um projeto desenvolvido com a interface CMSIS:

```
void FDCAN2_IT0_IRQHandler(void)
{
	if (FDCAN2->IR & FDCAN_IR_RF0N) { // Recepcao de nova mensagem
		FDCAN2_GetMessageFromRxFifo0 ();
		FDCAN2->IR |= FDCAN_IR_RF0N_Msk;
	}
}
```

A macro correspondente ao evento de uma nova mensagem no CMSIS é `FDCAN_IR_RFON`. Utilizamos uma função própria que foi implementada no escopo `/* USER CODE BEGIN 4 */` do `main.c`:

```
void FDCAN2_GetMessageFromRxFifo0 (void) {
	uint32_t GetIndex = 0;
	uint32_t *RxAddress;
	uint32_t ByteCounter;
	if ((FDCAN2->RXF0C & FDCAN_RXF0C_F0S) && (FDCAN2->RXF0S & FDCAN_RXF0S_F0FL)) {
		//Le mensagens quando FIFO 0 cheia
		if (((FDCAN2->RXF0S & FDCAN_RXF0S_F0F) >> FDCAN_RXF0S_F0F_Pos) == 1U)
		{
			/* Calcula o indice do elemento no Rx FIFO 0  */
			GetIndex += ((FDCAN2->RXF0S & FDCAN_RXF0S_F0GI) >> FDCAN_RXF0S_F0GI_Pos);
			/* Calcula o endereco efetivo da mensagem */
			RxAddress = (uint32_t *)(SRAMCAN_BASE+(1U)*4 +
					(GetIndex * ((FDCAN2->RXF0C & FDCAN_RXF0C_F0S) >> FDCAN_RXF0C_F0S_Pos) * 4U));
		}
		/* Incrementa RxAddress para acessar a segunda palavra do cabecalho */
		RxAddress++;
		/* Incrementa RxAddress para acessar os campos de dados */
		RxAddress++;
		//Dados a serem lidos
		ByteCounter = Code2ByteCounter
				((FDCAN2->RXESC & FDCAN_RXESC_F1DS) >> FDCAN_RXESC_F1DS_Pos);
		/* Retrieve Rx payload */
		for (uint8_t i = 0; i < ByteCounter; i++)
		{
			RxData[i] = ((uint8_t *)RxAddress)[i];
		}
		// Acknowlege a recepcao, incrementando Get indx
		FDCAN2->RXF0A = GetIndex;
	}
}
```

Trata-se de uma função inversa à `FDCAN2_AddMessageToTxFifoQ`, que extrai os dados das mensagens recebidas no espaço de memória CAN previamente alocado.

Por fim, deve-se incluir os protótipos das próprias funções no escopo /* USER CODE BEGIN PFP */ do `main.c` de um projeto que usa a interface CMSIS

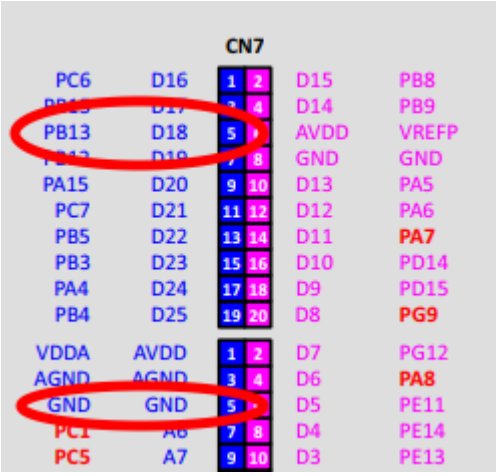
```
/* USER CODE BEGIN PFP */
void FDCAN2_Init(void);
void FDCAN2_AddMessageToTxFifoQ (uint32_t id);
void FDCAN2_GetMessageFromRxFifo0 (void);
uint8_t Code2ByteCounter(uint8_t code);
/* USER CODE END PFP */
```

E no arquivo `stm32h7xx_it.c` deste projeto, deve-se inserir

```
/* USER CODE BEGIN PFP */
void FDCAN2_GetMessageFromRxFifo0 (void);
/* USER CODE END PFP */
```

10. Realize o *build* e inicie um *debug*. Quando o programa estiver carregado, vá à aba de *Live Expressions* e adicione as expressões "TxData" e RxData". Depois inicie a execução. Veja os valores de TxData sendo atualizados a cada segundo, e os valores de RxData atualizados de acordo.

11. Conecte o canal 0 do analisador lógico ao pino PB13. Isto pode ser feito no pino 3 do *header* H7 (CAN2) ou no pino 5 de CN7 da placa NUCLEO. Lembre-se de conectar o GND do analisador a um GND da placa (pino 5 do *header* H7 ou um dos GND disponíveis nos conectores CN7 a CN10). Ajuste a aquisição para iniciar na borda de descida do canal e faça uma aquisição de 1ms. Ative o analisador CAN (deve-se usar o sinal "+" no campo de analisadores) configurado com a *Bit Rate* em 500.000. Note os vários elementos do quadro e compare com a referência, lembrando que este quadro é de formato padrão.



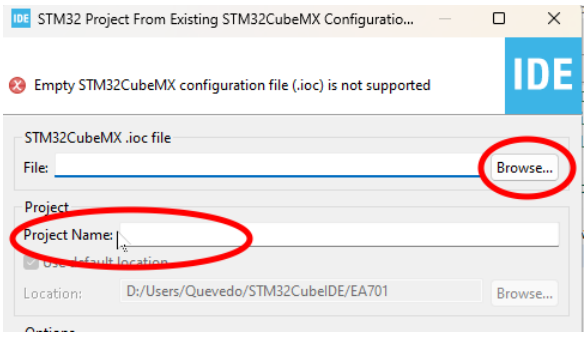
11. Mude o valor do elemento `TxHeader.Identifier` para qualquer valor que caiba em 11 *bits*, exceto o zero (1 a 2047). Execute o programa. O que acontece? As mensagens ainda são recebidas?

Note que aqui todas as mensagens que forem colocadas no barramento irão gerar a interrupção de recepção. Um programa mais completo precisa verificar o ID da mensagem, através do valor do campo “RxHeader.Identifier” para decidir o que vai fazer com ela.

Projeto de comunicação CAN usando *loopback* e filtros

Você já parou para pensar em como as informações que você recebe na internet são filtradas? Muitas vezes, você se depara apenas com conteúdos que realmente interessam a você, como se houvesse um assistente pessoal organizando tudo. Mas como isso acontece? É fascinante saber que existem programas que analisam o conteúdo e enviam apenas o que se alinha aos seus interesses. Imagine se isso pudesse ser feito diretamente pelo *hardware*, bloqueando dados irrelevantes antes mesmo de chegarem ao seu processador! Essa otimização na comunicação não é apenas teórica. No nosso projeto anterior, discutimos como as mensagens CAN operam. Cada mensagem que passa pelo barramento aciona a interrupção de recepção do módulo, e é o *software* que decide se essa mensagem é relevante ou não. Em um ambiente onde muitas mensagens estão trafegando, essa tarefa pode sobrecarregar a CPU. É aqui que entram os **filtros de Identificador**. Esses filtros analisam as mensagens assim que chegam, e só geram interrupções se atenderem a critérios específicos. Isso não só alivia a carga de trabalho do processador, mas também torna a comunicação muito mais eficiente. Vamos explorar juntos como essas tecnologias podem ser aplicadas na prática e como vocês podem desenvolver soluções inovadoras para otimizar ainda mais a troca de informações!

1. Crie um projeto “CAN_Filtros”, **sem inicializar os periféricos**. Este projeto tem a mesma configuração do projeto anterior. Assim, para facilitar o trabalho, use a opção do menu *File – New – STM32 Project from an Existing STM32CubeMX Configuration File (.ioc)*. Na janela que se abre, clique no botão “Browse” e navegue em seu *workspace* até a pasta com o nome do projeto anterior (CAN_Base). Nesta pasta, selecione o arquivo “CAN_Base.ioc”. Voltando à janela anterior, note que o campo do nome do projeto foi preenchido com o nome do arquivo selecionado. Mude o nome do projeto para “CAN_Filtros” e clique no botão “Finish”. Será criado um projeto com o nome novo selecionado, usando uma cópia do arquivo de configuração do outro projeto. Assim, o projeto já inicia configurado como o anterior.



2. Não será necessário alterar nem adicionar nenhuma configuração em relação ao projeto anterior, seja para a interface HAL ou para CMSIS. Na verdade, é necessário definir o número de filtros, mas isto também foi feito no outro projeto. Gere o código e abra o arquivo “main.c”.

3. Boa parte do código do projeto anterior, tanto para HAL quanto para CMSIS, também será aproveitada. Refaça os itens 6, 7 e 9 do projeto anterior, estabelecendo as variáveis globais, a configuração inicial e a função de *callback*. No projeto com a interface CMSIS, siga o procedimento do item 8 na implementação das funções insira as funções `Code2ByteCounter` `FDCAN2_AddMessageToTxFifoQ` no arquivo main.c de acordo com as explicações no item 8.

4. Agora vamos estabelecer os filtros. Dentro da função `MX_FDCAN2_Init`, gerada pelo CubeMX, existem áreas para implementação de código adicional pelo programador. Na seção `/* USER CODE BEGIN FDCAN2_Init 2 */`, faça a configuração do filtro:

```
FDCAN_FilterTypeDef sFilterConfig;

sFilterConfig.IdType = FDCAN_STANDARD_ID;
sFilterConfig.FilterIndex = 0;
sFilterConfig.FilterType = FDCAN_FILTER_RANGE;
sFilterConfig.FilterConfig = FDCAN_FILTER_TO_RXFIFO0;
sFilterConfig.FilterID1 = 0x11;
sFilterConfig.FilterID2 = 0x15;
sFilterConfig.RxBufferIndex = 0;
if (HAL_FDCAN_ConfigFilter(&hfdcan2, &sFilterConfig) != HAL_OK) {
      /* Filter configuration Error */
      Error_Handler();
}
/* Configure global filter to reject all non-matching frames */
HAL_FDCAN_ConfigGlobalFilter(&hfdcan2, FDCAN_REJECT, FDCAN_REJECT,
FDCAN_REJECT_REMOTE, FDCAN_REJECT_REMOTE);
```

Cada CAN pode ter mais de um filtro, mas aqui utilizamos apenas um. O filtro tem vários modos possíveis, e aqui escolhemos o modo por faixa (`FDCAN_FILTER_RANGE`). Existem dois elementos na *struct* de configuração do filtro (`FilterID1` e `FilterID2`) que definem os valores de identificadores que podem passar pelo filtro. Como estamos usando o modo por faixa, eles especificam o valor inicial e o valor final da faixa de passagem do filtro. No caso, todas as mensagens com identificadores entre 0x11 e 0x15, inclusive, irão gerar a interrupção.

No projeto baseado em CMSIS, precisamos apenas incluir o seguinte bloco de instruções no final da função `FDCAN2_Init`

```
{
        // Configura filtros de Identificadores para armazenamento no RxFIFO0
        uint32_t *FilterAddress;
        uint32_t FilterElementW1;
        FilterElementW1 = ((TIPO_FILTRO << 30U)  |  //0b00; 0b01 (dual); 0b10 e 0b11 (desabilita)
                (0b001 << 27U)   |                  //FDCAN_FILTER_TO_RXFIFO0
                (SFID1 << 16U)   |      // FilterID1
                (SFID2));                           // FilterID2
        /* Calcule o endereco do filtro na RaM de mensagens */
        FilterAddress = (uint32_t *)(SRAMCAN_BASE + (0 * 4U)); // offset = FilterIndex
        /* Escreve elemento de filtro na SRAMCAN  */
        *FilterAddress = FilterElementW1;
        /* Configura filtragem global */
```

```
        FDCAN2->GFC = ((0b10 << FDCAN_GFC_ANFS_Pos) | // NonMatchingStd - Reject
                (0b10 << FDCAN_GFC_ANFE_Pos) |  // NonMatchingExt - Reject
                (0b1 << FDCAN_GFC_RRFS_Pos) |   // RejectRemoteStd
                (0b1 << FDCAN_GFC_RRFE_Pos));   // RejectRemoteExt
}
```

e definir as seguintes macros no escopo `/* USER CODE BEGIN PD */` de `main.c`

```
#define SFID1 0x11
#define SFID2 0x15
#define TIPO_FILTRO 0b00  // 0b00: Intervalo; 0b10: Mascaramento
```

5. Vamos agora estabelecer o código do *loop*. Para distinguir as mensagens, para um identificador iremos enviar as sequências de *bytes* para valores entre 0 e 127. Para o outro identificador, a sequência vai de 128 até 256. Após a linha `/* USER CODE BEGIN 3 */`, escreva o código:

```
TxHeader.Identifier = 0x11;
for (int i=0; i<8; i++) {
      TxData[i] = indx++;
}

if(indx > 127) {
      indx = 0;
}
if (HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan2, &TxHeader, TxData)!= HAL_OK) {
      Error_Handler();
}

HAL_Delay (1000);

TxHeader.Identifier = 0x14;
for (int i=0; i<8; i++) {
      TxData[i] += 128;
}

if (HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan2, &TxHeader, TxData)!= HAL_OK) {
      Error_Handler();
}

HAL_Delay (1000);
```

Isso equivale a implementar o seguinte trecho de código em um projeto baseado em CMSIS usando a função própria definida no item 8.

```
        for (uint8_t i=0; i<8; i++) {
                TxData[i] = indx++;
        }
        if (indx > 127) indx = 0;
        FDCAN2_AddMessageToTxFifoQ (0x11);
        HAL_Delay (1000);
        for (uint8_t i=0; i<8; i++) {
                TxData[i] += 128;
        }
        FDCAN2_AddMessageToTxFifoQ (0x14);
```

HAL_Delay (1000);

Agora enviamos alternadamente os dois conjuntos de mensagens com os IDs 0x11 e 0x14, ambos na faixa de passagem do filtro.

6. Realize o *build* e o *debug*. Veja novamente as variáveis TxData e RxData na aba *Live Expressions*.

7. Vamos mudar a faixa do filtro. Mude o valor na linha `sFilterConfig.FilterID1 = 0x11;` para 0x12 (ou modificar o valor de SID1 para 0x12 em um projeto CMSIS). Como a faixa agora inicia em 0x12, as mensagens com *bytes* entre 0 e 127 (ID = 0x11) não irão chamar a função de *callback*, sendo totalmente ignoradas. Realize o *build* e o *debug*, executando o programa e vendo o que acontece.

8. Vamos mudar o tipo de filtro. Na linha `sFilterConfig.FilterType = FDCAN_FILTER_RANGE (0b00);` mude o parâmetro para `FDCAN_FILTER_MASK` (0b10) (no projeto CMSIS, substitua 0b00 por 0b10 a definição da macro TIPO_FILTRO). Agora o elemento `FilterID1` contém o valor base do filtro e o elemento `FilterID2` contém a máscara. A partir do valor base, apenas os *bits* correspondentes aos *bits* em "1" na máscara serão comparados no filtro. Mude o elemento `FilterID1` (no projeto CMSIS, `SFID1`) para 0x11 e o elemento `FilterID2` (no projeto CMSIS, `SFID2`) para 0xFA. Ao combinar o valor base com a máscara, os IDs que irão passar pelo filtro são 0b00010x0x, ou seja, tanto 0x11 quanto 0x14 irão passar. Realize o *build* e o *debug* e veja o resultado.

9. Mude apenas o elemento `FilterID2` (no projeto CMSIS, `SFID2`) para 0xFB. Agora os IDs que passam são 0b00010x01. Apenas as mensagens com ID 0x11 irão passar. Compile e execute novamente o programa e confirme esta afirmação.

Projeto de comunicação CAN usando múltiplos nós

Imagine a possibilidade de criar uma rede de controle que conecta dispositivos de forma ágil e eficiente, utilizando apenas um par de fios trançados. Com o entendimento do protocolo CAN bem estabelecido, você pode visualizar como podemos transmitir tanto sinais de dados quanto comandos de controle, integrando a serialização dos sinais e um sinal de *clock* para a recuperação desses dados no receptor. Neste projeto, vamos explorar a implementação de uma rede de controle CAN composta por múltiplos nós, onde cada bancada funcionará como um nó, enviando e recebendo mensagens que acionam dispositivos físicos e respondem a eventos em tempo real. Essa configuração não só demonstra a eficácia da comunicação entre sensores e atuadores, mas também proporciona uma experiência prática de colaboração. Sugerimos que **quatro duplas** colaborem, com cada grupo contribuindo como um nó de uma rede interconectada. Dessa forma, **cada fileira de quatro bancadas implementará sua própria rede**, proporcionando uma oportunidade única de transformar teoria em prática. Este projeto

não apenas estimula a aplicação dos conceitos aprendidos, mas também incentiva a criatividade no *design* de sistemas de controle, permitindo que vocês desenvolvam soluções em um ambiente colaborativo.

Cada nó irá ler um valor de 16 *bits* em um canal do ADC, no qual estará ligado um potenciômetro ou um eixo do *joystick* a uma taxa de 5 leituras por segundo. O valor lido será guardado em uma variável e transmitido pelo barramento CAN com um identificador próprio. Para o valor de 16 *bits* do ADC, serão usados os dois primeiros *bytes*, com o *byte* mais significativo primeiro. Cada nó irá também receber as mensagens dos demais nós com suas leituras do seus ADCs e guardar os valores em variáveis. A cada meio segundo, o *display* Nokia do nó será atualizado, apresentando em cada linha o valor atual recebido de cada nó, além do valor adquirido de seu próprio ADC.

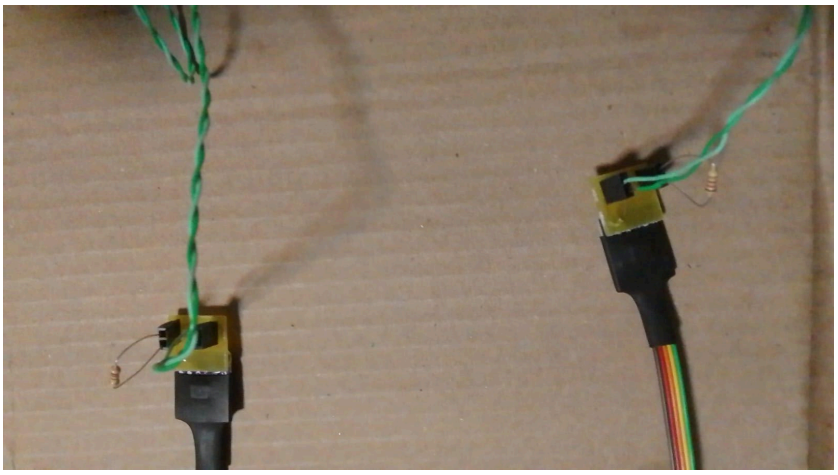
Para que a rede seja implementada corretamente, será necessário implementar a **camada física** da mesma. Esta camada é implementada com o auxílio de ***transcievers*** MCP2551, que são circuitos que convertem os sinais de transmissão do controlador CAN (que faz parte do microcontrolador) em níveis lógicos padrão do microcontrolador, em sinais **diferenciais** entre duas saídas, ligadas ao par trançado de fios (**CANH** e **CANL**). Os *transcievers* ainda convertem os sinais presentes no par trançado do barramento em níveis lógicos padrão para recepção pelo controlador CAN.

Note que como os *bits* são definidos pela **diferença entre** os pinos CANH e CANL, não importa o potencial entre cada um destes pinos e o GND. Assim, **não é necessário** conectar os GNDs de nós distintos. Porém, como o par trançado se comporta como uma linha de transmissão, é importante que em suas extremidades haja um resistor (entre CANH e CANL) com o mesmo valor da impedância característica do par trançado, no caso 120Ω, para garantir uma comunicação clara e eficaz entre os nós da rede. Sem terminações adequadas, os sinais enviados podem **refletir** nas extremidades do cabo, gerando distorções, falhas na comunicação e comprometendo a integridade das mensagens.

1. Os alunos deverão retirar no almoxarifado, para cada fileira de bancadas, 4 *transcievers*, 3 trechos de par trançado de fios, e 2 resistores de 120Ω. Os *transcievers* possuem cabos multi-vias com um conector na ponta, que deve ser ligado no conector H7 (CAN2) da placa auxiliar. Este conector disponibiliza os sinais de Tx e Rx do controlador CAN, bem como a alimentação de 5V para o *transciever*.

2. Na placa de cada *transciever*, foi adaptado um par de *headers* de 2 pinos, sendo cada *header* ligado aos sinais de CANH e CANL do *transciever*. Para localizar cada nó da rede, vamos usar um número entre 0 e 3. O nó 0 é o mais à esquerda na fileira de bancadas (parede), sendo que o nó 1 é o imediatamente a seu lado e assim em diante. Em um *header* do adaptador no nó 0, introduza os terminais de um resistor de 120Ω. No outro *header*, introduza as pontas desencapadas de uma das extremidades de um dos trechos de par trançado. Note bem qual fio fica mais perto da placa do *transciever* (CANL) e qual fica mais perto da borda da placa adaptadora (CANH). Em um *header* do nó 1, introduza as pontas desencapadas da

outra extremidade do mesmo trecho de par trançado, mantendo a mesma posição usada no nó 0. No outro *header*, introduza as pontas de uma extremidade de outro par trançado, sendo que a outra extremidade deste par deve ser ligada a um dos *headers* do nó 2, **mantendo a polaridade**. Da mesma forma, o último trecho de par trançado deve ser ligado entre um *header* do nó 2 e um *header* do nó 3, sendo que neste último o *header* sobrante deve ter o outro resistor de 120Ω conectado. A figura a seguir ilustra a configuração de uma rede com dois nós. Em ambas as placas, os resistores de terminação estão conectados a um *header*, enquanto uma extremidade do par trançado se liga ao *header* do outro nó.



Além das ligações do *transciever*, é necessário ligar um potenciômetro ou um canal do *joystick*. Conecte os extremos do potenciômetro ou os pinos “GND” e “+5V” do *joystick* nos pinos 2 (+3.3V) e 7 (GND) do conector H9 (ADC). O terminal central do potenciômetro ou um dos canais do joystick (Vx ou Vy) deve ser ligado ao pino 3 do mesmo conector (PC4). As ligações são as mesmas feitas no roteiro de DAC e ADC.

3. Vamos agora implementar a programação do controlador CAN e da CPU. Devido à sua complexidade relativa maior que projetos anteriores, o projeto todo foi implementado com a interface HAL e disponibilizado no arquivo CAN_Network.zip para importação no STM32CubeIDE. Importe o projeto em todos os computadores da fileira de bancadas. Nos próximos itens, vamos explorar os detalhes de configuração do microcontrolador e do funcionamento do código.

4. Inicialmente, abra o arquivo “CAN_Network.ioc” para ver a configuração do *CubeMX* em modo gráfico. Aqui vários periféricos foram configurados. O *clock* foi configurado como nos projetos anteriores deste roteiro, porém foi adicionada a configuração para o PLL2, para que este sirva de *clock* para o ADC1.O valor de PLL2P foi definido para 43MHz, pois o valor máximo deste *clock* é abaixo dos 50MHz usados no FDCAN. Assim, não podemos usar a mesma fonte de *clock* para ambos.

5. O FDCAN2 foi configurado de maneira semelhante aos projetos anteriores, mas com algumas adaptações importantes. Primeiramente, o parâmetro “Auto Retransmission” foi

alterado para “Enable”, permitindo que o controlador tente reenviar mensagens em caso de colisão de barramento, dado que estamos lidando com quatro nós transmitindo informações simultaneamente. Em segundo lugar, o parâmetro “Rx Fifo0 Elmts Nbr” foi aumentado de 1 para 4, possibilitando o armazenamento de até quatro mensagens na FIFO0 de recepção. Os demais parâmetros permanecem inalterados, incluindo a interrupção do FDCAN, que continua habilitada com prioridade 1.

6. O ADC foi configurado usando o *CubeMX*, ativando o pino PC4 como o canal 4, e parâmetros no padrão inicial, exceto pelo “Oversampling Ratio” igual a 32, o “Oversampling Right Shift” igual a “5 bit shift” e o “Sampling Time” igual a 32.5 ciclos. O tempo de amostragem é alongado para que a carga do capacitor de *sample and hold* possa acontecer integralmente mesmo com uma fonte de resistência relativamente alta. O *oversampling* permite usar um *trigger* para amostrar o mesmo canal várias vezes em sequência e somar os resultados das conversões, sendo que ao final a soma sofre um deslocamento para a direita. No caso, somamos 32 conversões e deslocamos 5 *bits*, dividindo a soma por 32. Assim, realizamos uma média de conversões, reduzindo o ruído elétrico na entrada do ADC. Por fim, o parâmetro “External Trigger Conversion Source” foi mudado para “Timer 6 Trigger Event”, para disparar o ADC na atualização do TIM6. A interrupção de ADC foi habilitada com prioridade 2.

7. O TIM6 foi ajustado com *prescaler* de 64000 e *AutoReload* em 200, gerando assim uma frequência de repetição de *updates* de 5Hz. O parâmetro *Trigger Event Selection* foi modificado para *Update Event*. Assim, o ADC é disparado 5 vezes por segundo via eventos de interconexões internas dos módulos. Não é habilitada a interrupção de *timer*.

8. O TIM7 foi ajustado com *prescaler* de 64000 e *AutoReload* em 500, gerando assim uma frequência de repetição de *updates* de 2Hz. A interrupção foi habilitada com prioridade 3. Essa interrupção sinaliza o momento de atualizar o *display*. Ao concluir a configuração de todos os componentes necessários, é gerado o código-fonte de inicialização em C através de “*Device Configuration Tool Code Generation*”. O CubeMX cria uma estrutura de dados de inicialização para cada componente, armazenando os valores configurados. Em seguida, ele insere no arquivo `main.c` chamadas às funções HAL correspondentes a cada componente, seguindo o padrão de nomenclatura HAL_x_Init e passando as respectivas estruturas de dados como argumento. Essa abordagem garante uma inicialização organizada e consistente dos componentes, além de facilitar a manutenção e melhorar a legibilidade do código.

9. O programa principal inclui a biblioteca “Nokia_5110” usada em roteiro anterior para apresentar as linhas de texto. Ele ainda define as macros ***ID***, ***IDBASE*** e ***IDMASK*****,** sendo que a primeira deve definir um valor diferente para cada nó (**número do nó na rede**), entre 0 e 3, e as outras duas são as mesmas em todos os nós, para definir o filtro de identificadores.

10. O código usa as mesmas variáveis para os *headers* e dados de transmissão e de recepção via CAN que foram usadas nos projetos anteriores. Ele ainda define um vetor de 4 elementos

de 16 *bits* sem sinal para guardar os 4 valores dos ADCs: índice 0 para o nó 0, índice 1 para nó 1, e assim em diante.

11. O programa inicialmente configura o *display* e escreve "CAN DEMO" na primeira linha, esperando 5 segundos e apagando o *display*. Depois, inicia o CAN com parâmetros idênticos aos demais projetos, sendo que o identificador da mensagem é IDBASE mais o número do nó. Assim, os identificadores dos 4 nós serão 0xA0. 0xA1, 0xA2 e 0xA3. O filtro de identificador é definido com base 0xA0 (0b0000 1010 0000) e máscara 0x7FC (0b111 1111 1100), o que habilita identificadores de 11 *bits* com o valor binário igual a 0b000 1010 00XX. Os dois primeiros *bytes* de dados serão o valor do ADC (*Byte* mais significativo primeiro), enquanto que os demais serão sempre 0xAA. Por fim, o programa habilita o ADC e os *timers*.

12. A função *callback* de conversão de ADC concluída coloca o *flag* ***adcnovo*** em 1. A função *callback* de *update* de TIM7 coloca o *flag* ***update*** em 1. A primeira *flag* indica que o valor novo do ADC deve ser escrito no vetor de valores (no índice que corresponde a seu número de nó) e a segunda indica que o *display* deve ser atualizado.

13. A função *callback* de evento da FIFO de recepção do controlador CAN recupera a mensagem recebida, extrai o número do nó transmissor a partir do identificador, e guarda o valor recebido nos dois primeiros *bytes* na posição correspondente do vetor de valores.

14. No *loop* principal do programa, inicialmente é verificado se a *flag* de valor novo no ADC foi setada, e em caso positivo é montado o quadro de dados da mensagem CAN, e a mesma é transmitida. Além disso, o valor é guardado na posição correspondente ao número do nó do vetor de valores. Depois, verifica-se se a *flag* de atualização do *display* foi setada, e em caso positivo o *display* é atualizado.

15. Na atualização do *display*, é usada uma função ***BuildString*** para montar a *string* de cada linha a partir do número da linha (linha 0 para valor do nó o e assim em diante) e do valor de 16 *bits* a ser escrito. A função é chamada para os 4 nós.

16. Compile e execute o programa nas 4 placas do barramento (lembre-se de compilar o projeto usando um ID diferente para cada placa). Observe o *display* e atue sobre os potenciômetros ou *joysticks* em cada placa, vendo as mudanças nos valores.

17. Conecte o analisador lógico no pino PB13 e veja o tráfego de informações transmitidas pelo barramento.

REDE DE COMUNICAÇÃO[1]

Uma **rede** é composta por nós—os pontos de conexão em uma rede—e os ***links*** que os conectam. Por exemplo, em uma rede local (LAN, do inglês *Local Area Network*), cada computador é um nó. Um **roteador** é um dispositivo que atua como um nó ao conectar seu computador à *internet*. Uma **ponte de rede** é um tipo de nó que conecta dois segmentos de rede entre si, permitindo que os dados fluam entre eles. Um **repetidor** recebe informações, remove ruídos e, em seguida, retransmite o sinal para o próximo nó na rede. Um **switch** conecta diferentes dispositivos dentro de uma LAN, direcionando as informações para o dispositivo correto e gerenciam o tráfego de dados entre eles.

Os *links* são os meios de transmissão usados para enviar informações entre os nós da sua rede. O tipo mais comum de *link* é um cabo, embora o tipo de cabo utilizado dependa da rede que está sendo criada. Por exemplo, **cabos coaxiais** são comumente usados em redes LAN; **cabos de par trançado** são amplamente utilizados para linhas telefônicas e em redes de telecomunicações; **cabos de fibra óptica** transmitem pulsos de luz que comunicam dados e são frequentemente usados para *internet* de alta velocidade e cabos de comunicação submarina.

Topologias de rede

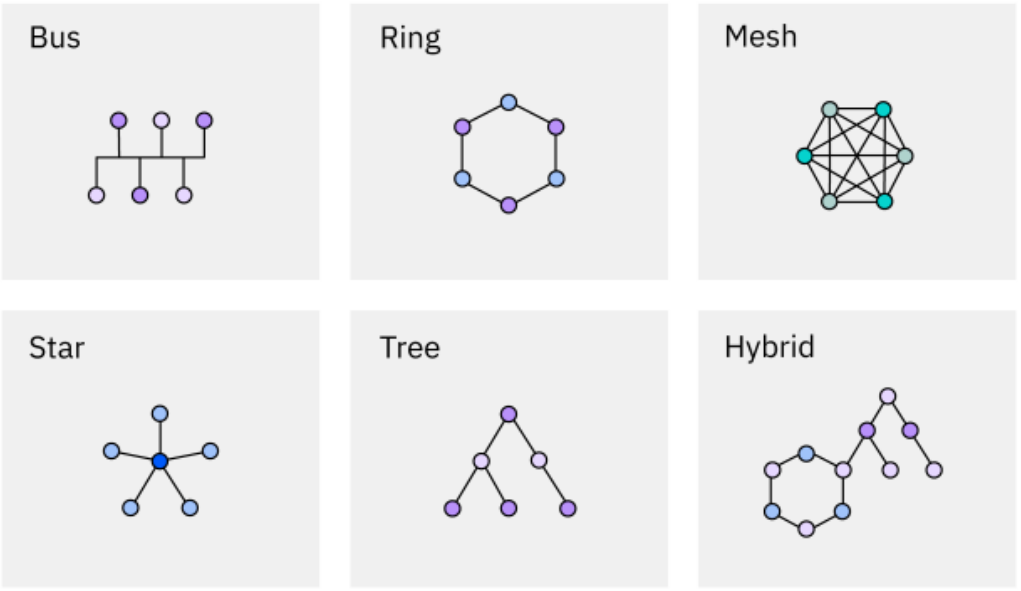
Topologia de rede refere-se à maneira como os **nós** e as **conexões** estão organizados, tanto fisicamente quanto logicamente, em uma rede. As redes são formadas por uma série de **links** e **nós**. Os nós incluem dispositivos como **roteadores**, **switches**, **repetidores** e **computadores**. A topologia de rede descreve como esses componentes estão dispostos em relação uns aos outros e como os dados circulam pela rede.

A topologia de rede afeta muitos aspectos da funcionalidade da rede, incluindo **velocidades de transferência de dados**, **eficiência da rede** e **segurança da rede**. Existem vários tipos diferentes de topologia, cada um com suas vantagens e desvantagens. É importante considerar essas características ao decidir qual topologia é mais adequada para uma rede. Ela descreve dois aspectos diferentes de uma rede de comunicações: a **topologia física** e a **topologia lógica**. A **topologia física** de uma rede descreve a disposição de cada componente na rede e como eles estão fisicamente conectados. Um **mapa de topologia de rede** pode ajudar os administradores de rede a visualizar como os dispositivos estão conectados entre si e como organizar melhor os *links* e nós. Por outro lado, a **topologia lógica** descreve como os dispositivos de rede parecem estar conectados uns aos outros e como os dados fluem pela rede. Os dados não necessariamente fluem em todas as direções em cada rede, e a topologia lógica da rede pode ilustrar como os dados devem ser transferidos e o número de *links* e nós pelos quais os dados passam antes de chegarem ao seu destino.

Existem vários tipos de topologias de rede, incluindo: topologia ponto a ponto, topologia em barramento, topologia em anel, topologia em estrela, topologia em árvore, topologia em malha e topologia híbrida.

---

[1] O conteúdo desta seção é uma versão estendida do material disponível pelo IBM.



Ponto-a-ponto

Uma **rede ponto a ponto**, ou **topologia ponto**, é a rede mais fácil de entender e o tipo mais básico de topologia de rede. Consiste simplesmente em dois nós conectados por um único link. Os dados trafegam de um ponto para outro entre esses dois extremos. Embora este seja o tipo de rede mais fácil de configurar, sua simplicidade é também sua desvantagem. Uma topologia ponto a ponto não é aplicável para a maioria dos casos de uso modernos.

Barramento

Na topologia de barramento, todos os nós estão conectados a um único cabo, conhecido como barramento ou *backbone*, semelhante a paradas de ônibus que se ramificam de uma rota principal. Os dados viajam em ambas as direções ao longo do cabo, permitindo que todos os dispositivos se comuniquem entre si. Essa configuração é econômica e fácil de implementar, tornando-a uma escolha popular em diversas aplicações. No entanto, possui algumas limitações. Um dos principais problemas é o ponto único de falha: se o *backbone* falhar, toda a rede fica inoperante. Além disso, as redes de barramento são menos seguras, uma vez que compartilham o mesmo cabo. À medida que mais nós se conectam ao cabo central, o risco de colisões de dados aumenta, o que pode reduzir a eficiência da rede e causar lentidões.

Topologia em anel

Na topologia em anel, os nós estão conectados de forma circular, com cada nó tendo exatamente dois vizinhos. Os dados fluem em uma única direção ao redor do anel, embora sistemas de anel duplo possam enviar dados em ambas as direções. Essas redes costumam ser baratas para instalar e expandir, e os dados se movem rapidamente dentro da rede. A principal vulnerabilidade das redes em anel é que a falha de um único nó pode derrubar toda a rede. Para proteger contra esse tipo de falha, são utilizadas redes de anel duplo. Uma rede de anel duplo possui dois anéis concêntricos em vez de um. Os anéis enviam dados em direções opostas. O segundo anel é ativado quando há uma falha no primeiro. Essa redundância

minimiza o tempo de inatividade e garante que os dados possam continuar a fluir mesmo se um dos anéis falhar.

Topologia em estrela

Na topologia em estrela, todos os nós estão conectados a um *hub* central, que amplifica e retransmite os sinais para os dispositivos. Os nós estão posicionados ao redor desse *hub* central, formando uma estrutura que se assemelha a uma estrela. Se um único nó falhar, o restante da rede não é afetado, desde que o *hub* central esteja funcionando. A topologia em estrela é geralmente fácil de diagnosticar e gerenciar, o que a torna uma escolha popular para redes locais (LANs, do inglês *Local Area Networks*). Sua estrutura centralizada também facilita a adição ou remoção de dispositivos, contribuindo para sua escalabilidade. No entanto, o desempenho de toda a rede depende do *hub* central e das conexões com ele. Se o *hub* central falhar, toda a rede ficará inoperante.

Topologia em árvore

A topologia em árvore combina elementos das redes em barramento e em estrela, criando uma estrutura hierárquica. Nessa configuração, um *hub* central serve como o nó raiz, conectando-se a várias redes em estrela em vez de nós individuais. Essa arquitetura permite um maior número de dispositivos conectados a um centro de dados central, melhorando a eficiência do fluxo de dados. Assim como nas redes em estrela, as topologias em árvore facilitam a identificação e a resolução de problemas em nós individuais. Nos casos de topologias em árvore, os nós da rede dependem de um *hub* central, criando dependências que podem afetar o desempenho da rede. Além disso, as topologias em árvore herdam vulnerabilidades tanto das redes em barramento quanto das redes em estrela. O ponto único de falha no *hub* central pode interromper toda a rede.

Topologia em malha

A topologia em malha é uma estrutura de rede altamente interconectada, onde cada nó está diretamente ligado a múltiplos outros nós. Em uma configuração de malha completa, todos os nós se conectam entre si, criando caminhos redundantes para a transmissão de dados. Isso significa que, se uma conexão falhar, os dados podem ser redirecionados por outros caminhos, aumentando a resiliência e a tolerância a falhas da rede. Nas topologias de **malha parcial**, apenas alguns nós estão diretamente conectados a todos os outros, oferecendo um equilíbrio entre a robustez da malha completa e a economia de topologias mais simples. Essa estrutura descentralizada reduz a dependência de um único ponto de falha, o que melhora tanto a segurança quanto a eficiência.

As redes em malha oferecem várias vantagens, como maior velocidade na transmissão de dados e escalabilidade, tornando-as adequadas para aplicações críticas, redes sem fio e cenários que exigem alta confiabilidade e desempenho. No entanto, esses benefícios vêm com

uma complexidade maior no *design* e na gestão da rede. O grande número de conexões pode resultar em custos mais altos de implementação e manutenção, especialmente em configurações de malha completa em redes grandes. Apesar desses desafios, as topologias em malha são amplamente utilizadas em diversas áreas, devido à sua capacidade de garantir um funcionamento eficaz e resiliente.

Topologia híbrida

A topologia híbrida combina elementos de diferentes tipos de topologias para atender a necessidades específicas. Por exemplo, uma rede pode utilizar configurações em estrela e em malha para equilibrar escalabilidade e confiabilidade. Uma rede em árvore, que junta uma rede em estrela com uma rede em barramento, também é um exemplo de topologia híbrida. Cada topologia de rede híbrida pode ser personalizada para construir uma arquitetura de rede eficiente, com base em casos de uso e necessidades empresariais específicas. No entanto, criar uma arquitetura de rede personalizada pode ser desafiador e exigir mais cabos e dispositivos de rede, o que pode aumentar os custos de manutenção.

Protocolos de comunicação

Os protocolos de comunicação constituem um conjunto de regras que orientam a transmissão e o recebimento de informações, permitindo que dispositivos distintos se comuniquem, independentemente de suas diferenças internas e estruturais. Para entender essa dinâmica, é importante considerar modelos como o *Open Systems Interconnection* (OSI), que ilustra como os sistemas de computadores interagem em uma rede. O modelo OSI é composto por sete camadas, e cada uma delas opera com diferentes protocolos que regulam a comunicação em suas respectivas funções. Por exemplo, o Protocolo de Internet (em inglês, *Internet Protocol* – IP), que atua na camada de rede, é responsável por roteirizar dados, controlando informações essenciais como os endereços de origem e destino dos pacotes.

Além disso, os protocolos de comunicação são relevantes nos sistemas de controle na área de automação, definindo a maneira como a troca de dados entre dispositivos ocorre de forma eficiente e confiável. Compreender o funcionamento e a inter-relação desses protocolos é essencial para garantir a eficácia e a integração das redes de controle na automação moderna, assegurando que os sistemas operem de maneira harmoniosa e coordenada. Quanto à forma como os dispositivos se comunicam entre si e como a autoridade de controle, denominada **mestre**, é distribuída na rede, podemos distinguir as seguintes classes de protocolos:

- **Mestre-Escravo**: Neste modelo, um dispositivo (mestre) controla a comunicação, solicitando dados e enviando comandos, enquanto os dispositivos escravos respondem às suas solicitações. Protocolos como I2C são típicos desse modelo.
- **Multi-Mestre**: Semelhante ao mestre-escravo, este modelo permite que vários mestres compartilhem a mesma rede. Qualquer mestre pode iniciar a comunicação, utilizando arbitragem para evitar colisões. O protocolo CAN é um exemplo desse tipo.

- **Multi-Escravo**: Este modelo permite que um único mestre controle múltiplos escravos, mas não suporta a presença de vários mestres na mesma rede. O mestre se comunica individualmente com cada escravo, como ocorre no SPI.
- **Ponto-a-Ponto**: Este protocolo estabelece uma conexão direta entre dois dispositivos, permitindo a troca de dados. Embora seja simples e eficaz, limita a comunicação a apenas dois dispositivos. O UART é um exemplo clássico deste tipo de protocolo.
- **Ponto-a-Multiponto**: Nesse modelo, um dispositivo (geralmente um controlador) se comunica simultaneamente com múltiplos dispositivos, permitindo que um único transmissor envie dados para vários receptores. O protocolo DMX representa bem essa abordagem.

REDES DE CONTROLE

Uma rede de controle é um sistema de comunicação projetado para gerenciar e monitorar dispositivos e processos em ambientes industriais, automação predial, transporte e outras aplicações. O objetivo principal de uma rede de controle é garantir que os dispositivos conectados possam se comunicar de maneira eficaz, permitindo o controle em tempo real e a coleta de dados sobre o funcionamento dos sistemas. Essas redes são compostas por diversos dispositivos, como sensores, atuadores, controladores e interfaces de usuário, todos interligados por **protocolos de comunicação** que definem como as informações são trocadas.

Protocolos como UART/USART, I2C, SPI, CAN e DMX são amplamente utilizados em redes de controle com fio, cada um oferecendo características específicas que atendem a diversas necessidades. O UART e o USART, apresentados no Roteiro 7, são protocolos de comunicação serial que permitem a troca de dados entre dispositivos de maneira simples e eficiente, sendo ideais para conexões de longa distância e com taxas de transmissão ajustáveis. O I2C é ideal para comunicação entre microcontroladores e periféricos em curtas distâncias, enquanto o SPI se destaca pela alta velocidade de transmissão, permitindo transferências rápidas de dados. Detalhes técnicos sobre esses dois protocolos foram abordados no Roteiro 10. O protocolo CAN, que será discutido neste Roteiro, é projetado para ambientes industriais e automotivos, garantindo robustez e confiabilidade em sistemas críticos. Já o DMX, também explorado neste Roteiro, é amplamente utilizado no controle de iluminação e efeitos especiais, permitindo uma comunicação eficiente entre dispositivos em sistemas de entretenimento. As redes de controle podem ser organizadas em diferentes topologias, como estrela, malha ou barramento, dependendo das exigências específicas de cada aplicação.

As principais funções de uma rede de controle incluem o monitoramento, que permite coletar dados em tempo real sobre o estado dos dispositivos e processos, proporcionando a visualização e análise do desempenho. Além disso, o controle, que envolve o envio de comandos e ajustes para dispositivos, como motores e válvulas, garante que os processos operem conforme esperado. A automação é outra função importante, pois implementa lógicas de controle que permitem a operação automática de sistemas, reduzindo a necessidade de

intervenção manual. Por fim, a comunicação facilita a troca de informações entre dispositivos e sistemas, promovendo a integração e a interoperabilidade em ambientes complexos.

Barramento DMX

O protocolo DMX (*Digital Multiplex*) é um padrão de comunicação digital amplamente utilizado para controlar sistemas de iluminação em teatros, *shows*, eventos, e em alguns casos, *displays* LED e outras soluções de controle de iluminação. Ele permite o controle preciso de dispositivos de iluminação, como projetores de luzes, *strobes*, e outros equipamentos cenográficos (por exemplo, máquinas de neblina).

O protocolo DMX surgiu na década de 1980, desenvolvido pela USITT (*United States Institute for Theatre Technology*) como uma solução padronizada para substituir sistemas analógicos, que eram propensos a ruídos e limitações no controle simultâneo de múltiplos dispositivos. Em 1990, o DMX foi formalizado como o padrão internacional DMX512, que especifica como os dados são transmitidos entre controladores e dispositivos de iluminação. Atualmente, o DMX é usado principalmente para controlar dispositivos de iluminação, permitindo que um controlador ou mesa de luz envie comandos para diferentes dispositivos (também chamados de *fixtures*), alterando parâmetros como cor, intensidade, posição e efeitos. Por ser confiável e amplamente compatível, tornou-se o padrão para eventos de iluminação, e com a introdução de LEDs, sua aplicabilidade cresceu ainda mais. Entretanto, por não possuir métodos de detecção e correção de erros, não é recomendado para aplicações críticas.

O DMX usa a camada física RS-485 (também conhecida como EIA-485), um padrão robusto e resistente a interferências, ideal para transmissões de dados em ambientes com ruído elétrico. O RS-485 permite transmissões a distâncias de até 1200 metros com taxas de dados de até 250 kbps, que são adequadas para o DMX. A conexão física geralmente é feita com conectores XLR de 5 pinos, onde dois pinos transmitem os dados e o terceiro é um fio de terra, enquanto os pinos restantes podem ser usados para extensões futuras. Em algumas aplicações, conectores XLR de 3 pinos são usados, embora não seja oficialmente recomendado.

O DMX512 é um protocolo de transmissão unidirecional (do controlador para os dispositivos) que usa uma taxa de transmissão fixa de 250 kbps, com cada mensagem composta por 512 canais. Cada canal DMX carrega valores entre 0 e 255, correspondendo a um *byte* de dados, e cada valor representa um parâmetro específico, como intensidade de luz ou cor. No início de cada transmissão de dados, o DMX envia um “break” de pelo menos 88 microssegundos, que sinaliza aos dispositivos que uma nova sequência de dados está sendo enviada. Após o break, os dados são enviados em série, de forma contínua, permitindo que cada dispositivo conectado interprete os canais que lhe foram designados, na ordem em que são transmitidos.

As principais especificações técnicas do DMX512 são:

- Camada Física RS-485 (conectores XLR de 3 ou 5 pinos).
- Taxa de Transmissão de 250 kbps.
- Suporte para até 512 canais, onde cada canal transporta um *byte* de dados.
- Cada dispositivo é configurado para escutar canais específicos, permitindo controle independente.
- Sinal unidirecional, com possibilidade de ser complementado com tecnologias como o RDM (*Remote Device Management*) para comunicação bidirecional.
- O barramento DMX permite encadeamento de até 32 dispositivos em uma linha (ou **universo**) e pode ser expandido com o uso de amplificadores ou *splitters* para maiores quantidades.

O protocolo DMX é considerado confiável, escalável e versátil para o controle de iluminação em eventos, e sua simplicidade o tornou uma solução duradoura e amplamente adotada no mercado.

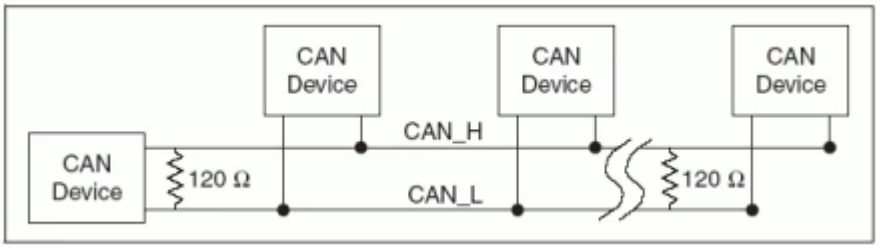
Barramento CAN

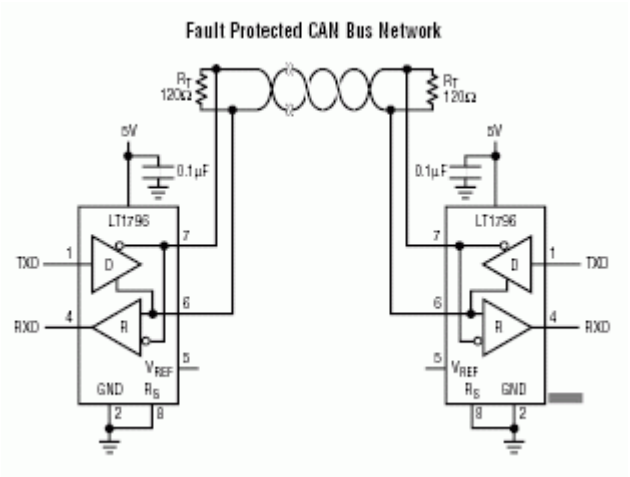
O barramento CAN (*Controller Area Network*) é um protocolo de comunicação em rede amplamente utilizado em sistemas embarcados, particularmente em automóveis, mas também em áreas como automação industrial, robótica e dispositivos médicos. Foi desenvolvido pela empresa alemã Bosch na década de 1980 com o objetivo de simplificar a comunicação entre os vários componentes eletrônicos de um veículo, substituindo o complexo sistema de fios ponto a ponto utilizado até então.

A primeira versão do protocolo foi lançada em 1986, com a padronização ocorrendo em 1993 como ISO 11898. A tecnologia CAN rapidamente se tornou uma escolha padrão para a indústria automotiva, sendo adotada por fabricantes para controlar sistemas como motores, transmissões, *airbags* e sistemas de assistência ao motorista. O barramento CAN permite a comunicação eficiente, robusta e de alta confiabilidade entre diversos dispositivos sem a necessidade de um controlador central, facilitando o compartilhamento de dados em tempo real.

O CAN é um protocolo ***multimaster*** e **assíncrono**. A sincronização entre os nós durante a comunicação ocorre com os dispositivos na rede ajustando seus próprios *clocks* com base na transição dos bits enviados no barramento. Essa característica permite que dispositivos com pequenas variações de *clock* possam se comunicar de forma eficiente. O nó que assume o papel de *master* sempre transmite um **quadro** (ou ***frame***), ou seja, cada nó transmite seus dados para todos os outros nós, e para um nó receber dados ele deve esperar aquele nó realizar sua transmissão. Existe uma forma de um nó requisitar que o outro nó transmita seus dados, que é denominado **quadro remoto**. Para que a informação seja devidamente identificada, o quadro possui um campo de **Identificador** (11 *bits* no modo padrão e 29 *bits* no modo estendido). Cada nó decide o que fazer com a informação de cada quadro recebido, em função de seu identificador.

Fisicamente, o barramento CAN geralmente é implementado utilizando dois fios: **CAN_H** (*CAN High*) e **CAN_L** (*CAN Low*), que formam uma linha **diferencial**. A comunicação diferencial oferece vantagens como imunidade a ruídos, possibilidade de não-conexão entre "terras" de nós distintos, e comunicação em distâncias longas. Porém, exige uma **camada física** específica para que os dados trafeguem adequadamente. Ou seja, existe um circuito especial que controla os níveis de tensão entre os fios da linha diferencial de acordo com o que deve ser transmitido, e outro circuito para ler a tensão diferencial e determinar o *bit* correspondente. Normalmente, os circuitos controladores de CAN apresentam duas conexões, **CAN_Tx** e **CAN_Rx**. Um circuito integrado denominado ***transciever*** converte os *bits* em CAN_Tx em sinais aplicados na linha diferencial, e converte os sinais da linha em *bits* em CAN_Rx. Como o par diferencial pode apresentar longas distâncias, deve ser tratado como uma **linha de transmissão**, exigindo que suas extremidades tenham **resistores de terminação**, de acordo com a impedância característica da linha diferencial, no caso 120Ω.





Assim como no protocolo I2C, quando mais de um dispositivo coloca um nível lógico no barramento, um dos níveis predomina. Este é denominado ***bit* dominante**, enquanto o outro é denominado ***bit* recessivo.**

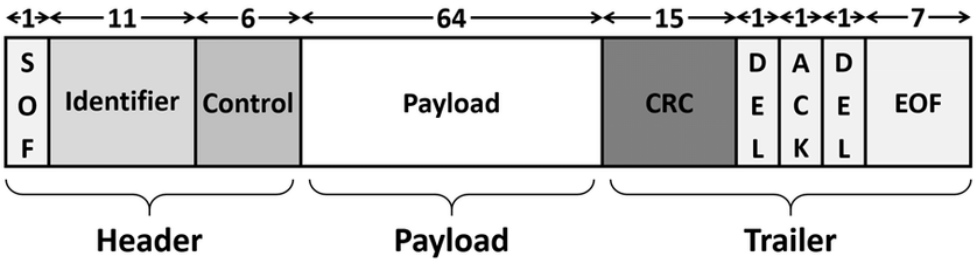
***Bit* dominante (0 lógico):** O CAN_H é elevado a cerca de 3,5V, enquanto o CAN_L cai para aproximadamente 1,5V, resultando em uma diferença de 2V entre os fios.
***Bit* recessivo (1 lógico):** Tanto o CAN_H quanto o CAN_L ficam próximos a 2,5V, resultando em uma diferença mínima entre eles (próxima de 0V).

No protocolo I2C, os resistores *pull-up* são usados para garantir que o barramento fique em um estado conhecido (alto) quando nenhum dispositivo está transmitindo. No caso do CAN, a tensão passiva entre CAN_H e CAN_L no *bit* recessivo já é definida por meio do *driver* diferencial, e a lógica de detecção de bits é baseada na diferença de tensão entre as duas linhas, não na presença de uma tensão de *pull-up* para definir o nível "alto" ou "baixo".
Isso elimina a necessidade de resistores *pull-up* no barramento CAN, pois o estado recessivo é determinado pela ausência de diferença significativa entre CAN_H e CAN_L, e o estado dominante é definido pela criação de uma diferença de 2V entre CAN_H e CAN_L.

O protocolo CAN utiliza uma técnica de arbitragem não destrutiva baseada em prioridade para resolver colisões. Isso acontece durante a transmissão dos bits de identificação de cada mensagem. Quando dois ou mais dispositivos tentam transmitir ao mesmo tempo, cada um coloca seu identificador de mensagem no barramento. Cada nó que está transmitindo também monitora o estado do barramento enquanto transmite seus *bits*. Se um nó transmite um *bit* recessivo (1), mas lê um *bit* dominante (0) no barramento, significa que um outro dispositivo está transmitindo uma mensagem de maior prioridade. O nó que detecta que sua mensagem tem menor prioridade (por perceber que um *bit* dominante foi transmitido por outro nó quando ele queria transmitir um *bit* recessivo) interrompe sua transmissão e espera até o barramento estar livre novamente. O dispositivo com a mensagem de maior prioridade continua sua transmissão sem interrupções. Como a arbitragem ocorre nos *bits* de prioridade, essa técnica garante que não há perda de tempo ou dados, mesmo quando há colisões.

O protocolo CAN suporta diferentes tipos de quadros (frames), os quais contêm as informações a serem transmitidas:

- Quadro de dados (*Data Frame*): O mais comum, contém os dados a serem enviados. Pode ter até 8 bytes de dados.
- Quadro remoto (*Remote Frame*): Usado para solicitar dados de outros dispositivos sem enviar dados em si.
- Quadro de erro (*Error Frame*): Transmitido quando um dispositivo detecta um erro no barramento.
- Quadro de sobrecarga (*Overload Frame*): Usado para solicitar mais tempo para processamento entre quadros de dados.



Formato de quadro CAN

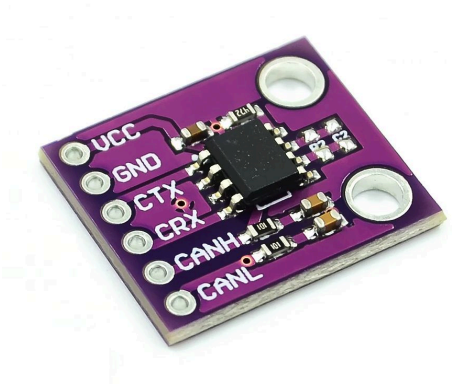
O CAN suporta taxas de transmissão variáveis, dependendo da versão do protocolo e da rede utilizada. No CAN clássico (ISO 11898-1), a taxa de dados pode chegar a até 1 Mbps. Com o CAN FD (*Flexible Data-Rate*), a taxa de bits pode ser aumentada durante a fase de transmissão dos dados, permitindo velocidades de até 5 Mbps, mantendo a compatibilidade com dispositivos CAN clássicos durante a fase de arbitragem.

O CAN tem robustos mecanismos de detecção de erros, incluindo:

- Verificação de *bit*: Cada nó verifica se o valor do *bit* no barramento corresponde ao que foi transmitido.
- Verificação de campo CRC (*Cyclic Redundancy Check*): Um código de verificação é transmitido junto com os dados para garantir a integridade da mensagem.
- Verificação de sobrecarga de tempo: Garante que o barramento não fique ocioso por muito tempo entre mensagens.

Com suas características de comunicação assíncrona, uso de sinais diferenciais, arbitragem por prioridade e robustos mecanismos de detecção de erros, o barramento CAN é ideal para sistemas distribuídos em tempo real que exigem alta confiabilidade, como redes automotivas e de automação industrial.

Transceivers MCP2551



Em sistemas embarcados que utilizam o protocolo CAN, os transceivers desempenham um papel crucial na comunicação entre os dispositivos. O protocolo CAN é amplamente adotado em aplicações industriais e automotivas devido à sua robustez e capacidade de operar em ambientes ruidosos. Nesse contexto, o *transceiver* MCP2551 se destaca como uma solução eficaz para a interface entre o microcontrolador e a rede CAN. Embora suas funcionalidades sejam robustas, incluindo comunicação diferencial que converte sinais TTL em sinais diferenciais, proteção contra curtos-circuitos e transientes elétricos, controle de inclinação para reduzir emissões de RFI (do inglês, *Radio Frequency Interference*), e modos de espera que diminuem o consumo de corrente, é importante observar que o MCP2551 não é mais recomendado para novos *designs*, com o MCP2561 sendo a alternativa sugerida.

O MCP2551 é um *transceiver* CAN projetado para servir como a interface entre um controlador de protocolo CAN e o barramento físico, operando em sistemas de 12V e 24V com suporte a velocidades de até 1 Mb/s, conforme o padrão ISO-11898. Ele converte os sinais digitais do microcontrolador em sinais diferenciais, necessários para a comunicação na rede CAN, e vice-versa. O MCP2551 permite a conexão de até 112 nós em um barramento CAN, operando com uma carga mínima de 45Ω e uma resistência de entrada diferencial mínima de 20 kΩ, além de incluir uma detecção de condição de falha que desabilita os *drivers* de saída para evitar a corrupção dos dados no barramento.

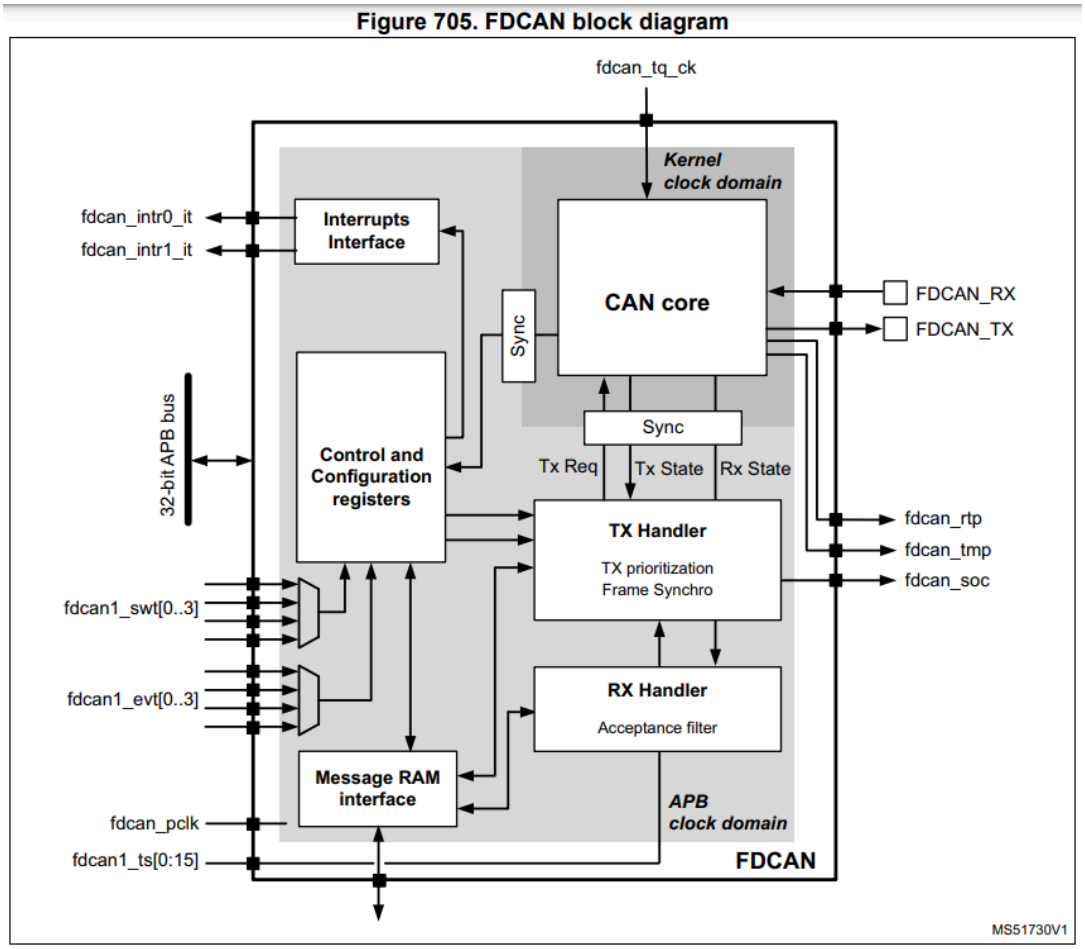
Integrar o MCP2551 em um sistema embarcado envolve conectá-lo ao microcontrolador e à rede CAN, onde ele atua como o elo entre a lógica digital do microcontrolador e os requisitos elétricos da rede. Com essa configuração, o sistema embarcado é capaz de se comunicar efetivamente em uma rede CAN, facilitando a troca de informações em tempo real entre múltiplos dispositivos.

STM32H7A3: MÓDULO FDCAN

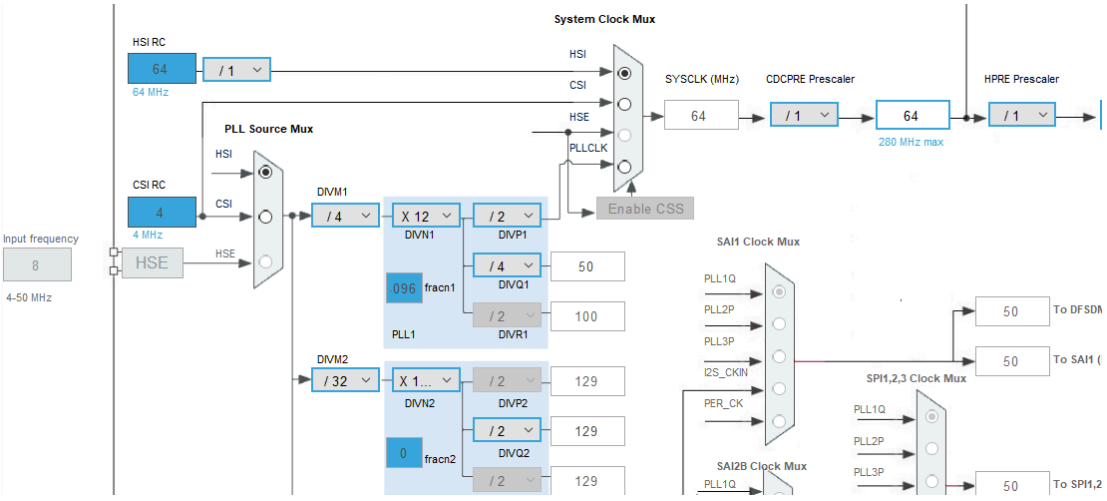
O subsistema de rede de controlador (CAN) integrado no microcontrolador STM32H73A consiste em dois módulos CAN, uma RAM de mensagens compartilhada e uma unidade de calibração de *clock*. Os módulos FDCAN são compatíveis com a norma ISO 11898-1:2015 e a especificação do protocolo CAN FD (do inglês, *Controller Area Network with **F**lexible **D**ata Rate*) versão 1.0. O primeiro módulo, FDCAN1, suporta CAN acionado por tempo (TTCAN), conforme a ISO 11898-4, incluindo comunicação sincronizada por eventos, tempo global do sistema e compensação de deriva do *clock*. O FDCAN1 possui registros adicionais para essa funcionalidade. A opção CAN FD pode ser utilizada junto com a comunicação acionada por eventos e por tempo.

O CAN FD representa uma evolução em relação ao protocolo CAN clássico, oferecendo maior flexibilidade e velocidades de transmissão de dados. Ele é compatível com os padrões CAN versão 2.0 e CAN FD 1.0, o que garante uma transição suave para sistemas já existentes. Uma das características mais notáveis do CAN FD é a capacidade de suportar quadros com até 64 *bytes* de dados, em comparação com o limite de 8 *bytes* do CAN clássico, tornando-o ideal para aplicações que requerem a transmissão de grandes volumes de informações. O protocolo introduz o **modo de quadro longo**, que permite campos de dados superiores a 8 *bytes*, e o **modo de quadro rápido**, que possibilita a transmissão de campos com taxas de *bits* mais altas durante a fase de dados. Isso é particularmente vantajoso, pois a comutação da taxa de *bits* pode ser realizada dentro do quadro, otimizando a eficiência da comunicação. Além disso, a codificação do *Data Length Code* (DLC) foi ajustada para suportar tamanhos de campo de dados que variam de 9 a 64 *bytes*, refletindo a maior capacidade de dados do CAN FD. A habilitação da operação CAN FD é feita através do registro de controle, onde o *bit* `FDCAN_CCCR_FDOE` é programado. O tratamento de exceções de protocolo também é uma melhoria importante, permitindo que eventos de erro sejam gerenciados de forma eficaz.

O seguinte diagrama de blocos de FDCAN proporciona uma visão geral do módulo.



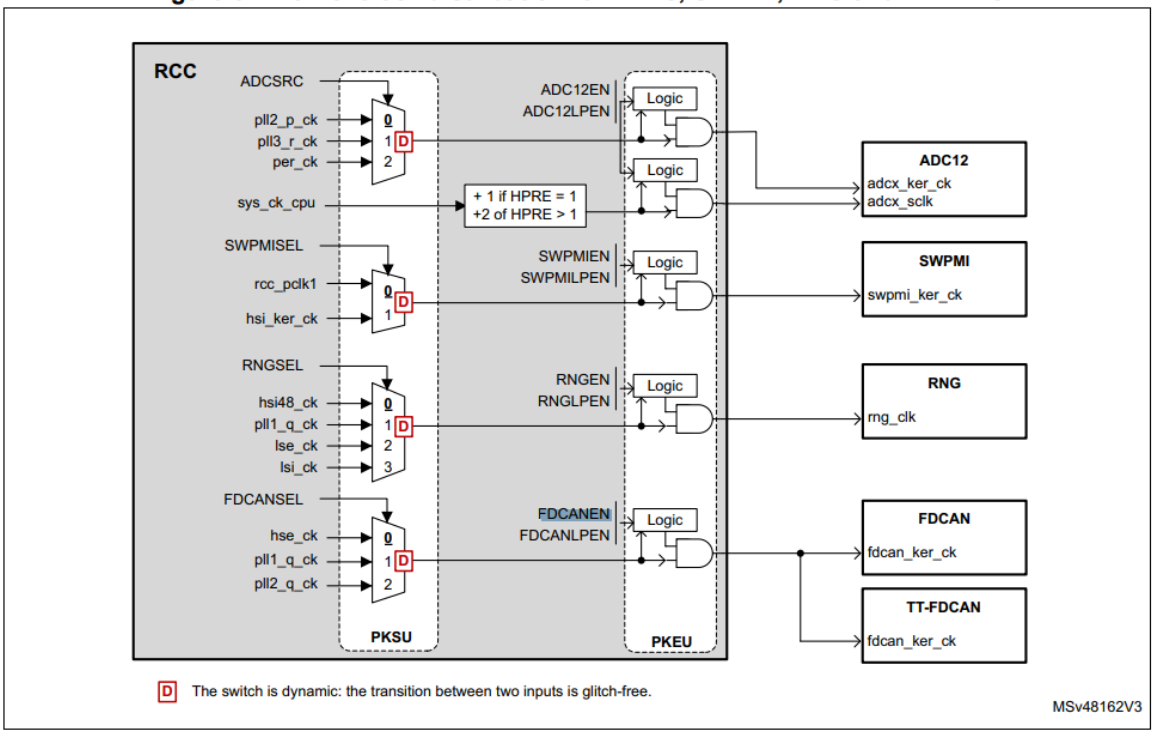
Os módulos FDCAN, assim como outros módulos de comunicação serial como I2C, SPI e UART/USART, utilizam dois domínios de sinais de relógio. Um é o relógio do núcleo (em inglês, *kernel*) `fdcan_ker_ck` selecionado pelo campo RCC_CDCCIP1R_FDCANSEL. As três possíveis fontes de `fdcan_ker_ck` são `hse_ck`, `pll1_q_ck` e `pll2_q_ck`. É necessário habilitar cada uma delas por meio de *software* ao serem selecionadas.



Por exemplo, se `pll1_q_ck` for utilizado, é necessário habilitá-lo por meio do *bit* `RCC_PLLCFGR_DIVQ1EN`. A frequência de `pll1_q_ck` pode ser configurada no editor gráfico do STM32CubeMx, onde é possível ajustar interativamente os parâmetros DIVM1, DIVN1, DIVQ1 e FRACN1. Contudo, se não houver a ativação de um periférico que utilize esses sinais, o código correspondente aos valores configurados não será gerado automaticamente. Nessa situação, é necessário configurar manualmente os registradores `RCC_PLLCKSEL`, `RCC_PLL1DIVR`, `RCC_PLL1FRACR` e `RCC_PLLCFGR`, garantindo que os *bits* `RCC_CR_PLL1ON` e `RCC_CR_PLL1RDY` estejam em "0". Após a configuração, deve-se ativar o PLL1 definindo `RCC_CR_PLL1ON` como "1" e aguardar até que `RCC_CR_PLL1RDY` indique "1".

$$pll1\_q\_ck = \frac{Fonte}{DIVM1} \times \frac{\left(DIVN1 + \frac{fracn1}{2^{13}}\right)}{DIVQ1}$$

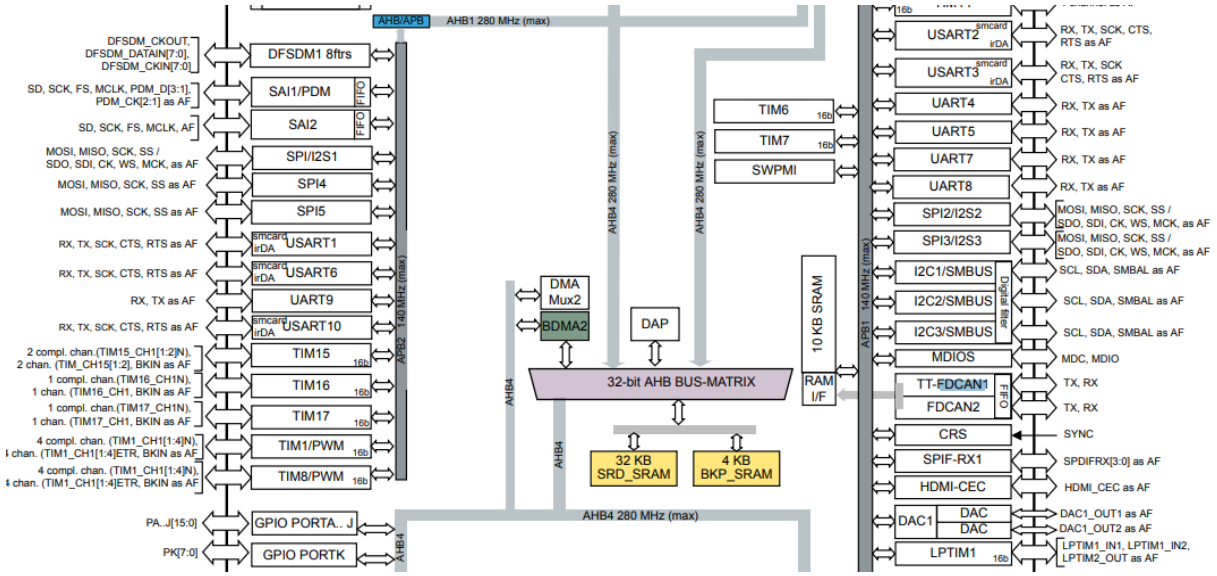
Figure 62. Kernel clock distribution for ADCs, SWPMI, RNG and FDCANs



1. **X** represents the selected mux input after a system reset.
2. This figure does not show the connection of the bus interface clock to the peripheral. For details on each enable cell, refer to *Section 8.5.11: Peripheral clock gating control*.

O outro relógio é o relógio da interface de barramento `fdcan_pclk` para os periféricos (em inglês, *peripheral*), como mostra o excerto do diagrama de blocos do STM32H7A3. O *clock gating* para o FDCAN é habilitado pelo *bit* `RCC_APB1HENR_FDCANEN`. A unidade de calibração de *clock* comum é opcional e pode gerar um *clock* calibrado para cada FDCAN a partir do oscilador RC interno HSI e do PLL, avaliando mensagens CAN recebidas pelo FDCAN1. Essa separação permite que a comunicação com o processador seja independente das pulsações dos contadores que controlam as taxas de transferência de dados com

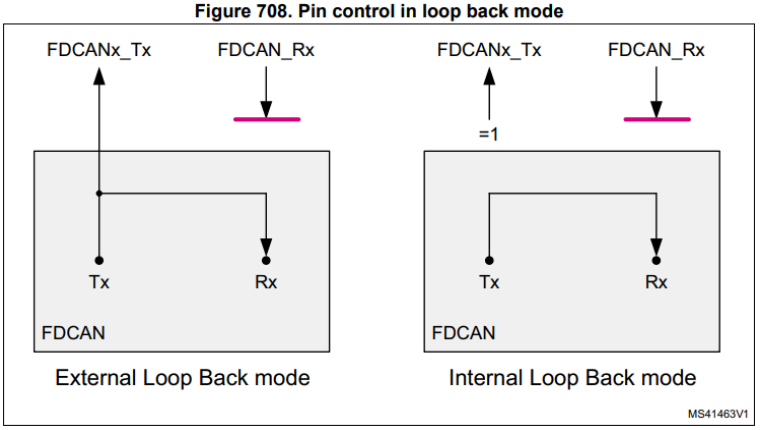
dispositivos externos. Isso proporciona maior flexibilidade e eficiência no gerenciamento de tempo e taxa de transmissão de dados.



Os diferentes modos de operação do FDCAN, juntamente com seus respectivos *bits* de configuração, estão resumidos na tabela a seguir:

	Normal	Operação Restritiva	Monitoramento do barramento	*Loopback* Interno	*Loopback* Externo
FDCAN_CCCR_TEST	0	0	0	1	1
FDCAN_CCCR_MON	0	0	1	1	0
FDCAN_TEST_LBCK	0	0	0	1	1
FDCAN_CCCR_ASM	0	1	0	0	0

Figure 708. Pin control in loop back mode



O FDCAN possui pinos dedicados para transmissão e recepção. O **pino de transmissão (**FDCAN_TX**)** é utilizado para enviar dados do FDCAN para a rede. Quando o controlador deseja transmitir uma mensagem, ele a envia por meio desse pino. O **pino de recepção** (FDCAN_RX) é usado para receber dados da rede. Mensagens enviadas por outros

dispositivos na rede chegam a este pino. Além desses pinos, a comunicação em rede CAN requer resistores de terminação de 120Ω em cada extremidade do barramento para evitar reflexões de sinal, garantindo a integridade da comunicação. Esses pinos precisam ser configurados para a função alternativa utilizando os registradores GPIOx_MODER, GPIOx_AFRL e GPIOx_AFRH. as funções alternativas `FDCAN2_RX` e `FDCAN2_TX`, associadas aos pinos PB5 e PB13, são designadas pela alternativa AF9.

O FDCAN utiliza uma memória RAM estática para armazenar mensagens a serem transmitidas e recebidas, além de filtros de aceitação que permitem ao FDCAN selecionar quais mensagens recebidas serão armazenadas com base em seus identificadores. Essa memória é mapeada no espaço de endereçamento da CPU, que vai de 0x4000AC00 a 0x4000D3FF.

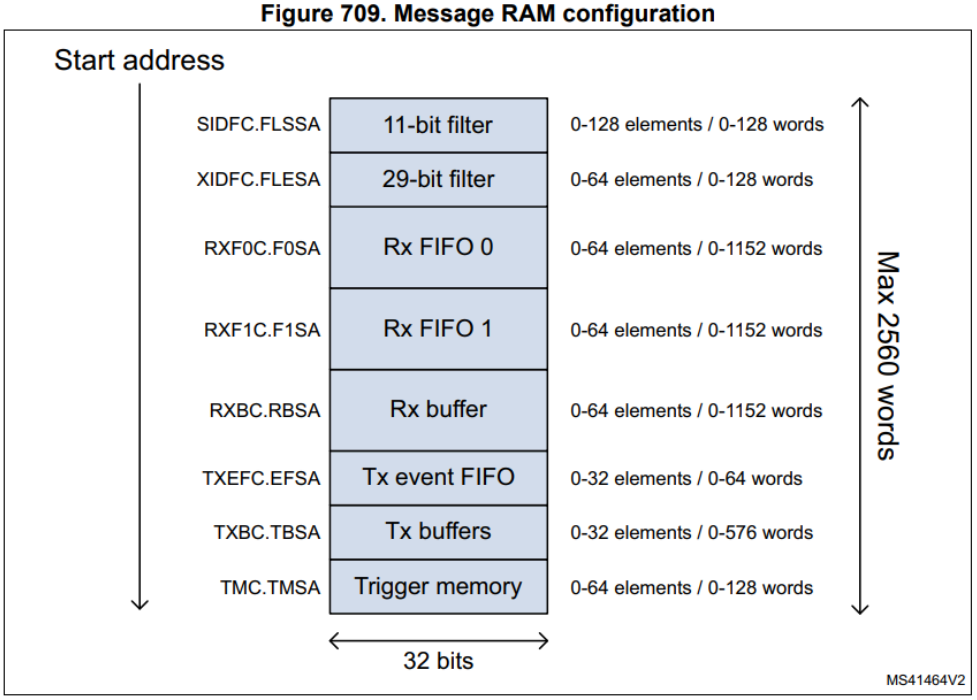
Boundary address	Peripheral
0x4000AC00 - 0x4000D3FF	CAN Message RAM
0x4000A800 - 0x4000ABFF	CAN CCU
0x4000A400 - 0x4000A7FF	FDCAN
0x4000A000 - 0x4000A3FF	TT-FDCAN

Essa memória, com largura de 32 *bits,* permite a alocação de até 2560 palavras, cada uma com 32 *bits*. Ela é organizada em segmentos dedicados a diferentes funções, como filtros e *buffers*, conforme ilustrado na figura a seguir. É importante destacar que não é necessário configurar cada segmento individualmente, nem há restrições quanto à ordem em que elas são alocadas. Para configurar o endereço inicial e o tamanho de cada segmento de memória, o FDCAN emprega diversos registradores. O `FDCAN_SIDFC_FLSSA` define o endereço inicial da lista de filtros para identificadores de mensagem padrão, enquanto o `FDCAN_XIDFC_FLESA` faz o mesmo para identificadores de mensagem estendidos. Esses filtros determinam quais mensagens recebidas pelo FDCAN serão processadas e quais serão descartadas, otimizando o uso dos recursos do sistema e garantindo que apenas as mensagens relevantes sejam tratadas pela aplicação. Os registradores `FDCAN_RXF0C_F0SA` e `FDCAN_RXF1C_F1SA` definem os endereços iniciais das FIFOs de recepção 0 e 1, respectivamente. O registrador `FDCAN_TXEFC_EFSA` é utilizado para a FIFO de eventos de transmissão, e o `FDCAN_RXBC_RBSA` especifica o segmento de *buffers* de recepção dedicados, que também pode ser usada para referenciar mensagens de depuração. Adicionalmente, o `FDCAN_TMC_TMSA` define o endereço inicial da memória de *trigger*, enquanto o `FDCAN_TXBC_TBSA` designa o início do segmento de *buffers* de transmissão, que pode ser dividida em *buffers* dedicados ou uma fila de transmissão, ou uma FIFO de transmissão, ou mesmo uma combinação destes.

O FDCAN utiliza duas FIFOs de recepção (Rx FIFO 0 e Rx FIFO 1) para armazenar mensagens recebidas após a filtragem de aceitação. Cada FIFO pode armazenar até 64 mensagen e o tamanho do campo de dados de cada mensagem em cada FIFO é definido no

registrador FDCAN_RXESC. A ordem de armazenamento nas FIFOs segue o princípio "primeiro a entrar, primeiro a sair", garantindo que as mensagens sejam processadas na ordem em que são recebidas. O FDCAN oferece flexibilidade na configuração dos *buffers* de transmissão, permitindo a implementação de uma Tx Queue (fila de transmissão) ou de uma Tx FIFO (FIFO de transmissão) através do campo FDCAN_TXBC_TFQM, mas não de ambas simultaneamente. A Tx Queue possibilita a priorização de mensagens com base nos IDs, transmitindo primeiro a mensagem com o menor ID. Por outro lado, na Tx FIFO, as mensagens são transmitidas na ordem em que foram escritas, seguindo o princípio "primeiro a entrar, primeiro a sair". O tamanho do campo de dados de cada mensagem no *buffer* de transmissão é especificado pelos 3 *bits* FDCAN_TXESC_TBDS.

Figure 709. Message RAM configuration



Uma mensagem representa a unidade de dados transmitida e recebida pelos nós na rede CAN. A estrutura básica de uma mensagem FDCAN é composta por vários campos, incluindo um identificador da mensagem (ID), dados a serem transmitidos (DBi) e quantidade de *bytes* na mensagem (DLC). O número de mensagens pode variar de 0 a 64 e o tamanho total, em palavras de 32 *bits*, numa FIFO de recepção pode variar de 0 a 1152 palavras. A Tabela 489 do Manual de Referência apresenta a estrutura dos campos de informações em uma mensagem no canal RX. A seção também detalha o conteúdo de cada um desses campos.

Table 489. Rx buffer and FIFO element

Bit	31			24	23				16	15	8	7	0
R0	ESI	XTD	RTR		ID[28:0]								
R1	ANMF	FIDX[6:0]			Res.	FDF	BRS	DLC[3:0]		RXTS[15:0]			
R2	DB3[7:0]				DB2[7:0]					DB1[7:0]		DB0[7:0]	
R3	DB7[7:0]				DB6[7:0]					DB5[7:0]		DB4[7:0]	
⋮	⋮				⋮					⋮			
Rn	DBm[7:0]				DBm-1[7:0]					DBm-2[7:0]		DBm-3[7:0]	

E a Tabela 491 do Manual de Referência mostra a estrutura dos campos em uma mensagem no canal TX, com os dados precedidos por duas palavras de cabeçalho de metadados, assim como no canal RX.

Table 491. Tx buffer and FIFO element

Bit	31			24	23				16	15	8	7	0
T0	ESI	XTD	RTR		ID[28:0]								
T1	MM[7:0]				EFC	Res.	FDF	BPS	DLC[3:0]	Reserved			
T2	DB3[7:0]				DB2[7:0]					DB1[7:0]		DB0[7:0]	
T3	DB7[7:0]				DB6[7:0]					DB5[7:0]		DB4[7:0]	
⋮	⋮				⋮					⋮			
Tn	DBm[7:0]				DBm-1[7:0]					DBm-2[7:0]		DBm-3[7:0]	

A alocação do endereço inicial do bloco de memória é de responsabilidade do *software*. Ele deve configurar os registradores mencionados com os endereços adequados para cada segmento da memória RAM, assegurando que não haja sobreposição entre os segmentos e que o total alocado não ultrapasse o limite de 2560 palavras. Para configurar o endereço inicial de cada segmento, o *software* escreve o valor desejado nos *bits* correspondentes do registrador. Por exemplo, para definir o endereço inicial da FIFO de recepção 0, deve-se escrever o valor nos *bits* `FDCAN_RXF0C_F0SA`. O FDCAN **não verifica se a configuração da Message RAM está correta**. É responsabilidade do usuário garantir que os endereços de início das diferentes segmentos e o número de elementos de cada segmento sejam configurados corretamente para evitar a falsificação ou perda de dados.

É importante observar que os endereços configurados são de palavra de 32 *bits*, e os dois *bits* menos significativos são ignorados pelo FDCAN. Além disso, o acesso de escrita aos *bits* de configuração do endereço inicial é protegido; a escrita só é permitida quando os bits `FDCAN_CCCR_CCE` e `FDCAN_CCCR_INIT` estão configurados como “1”. Dessa forma, a configuração correta do espaço de memória é crucial para o funcionamento eficiente do FDCAN.

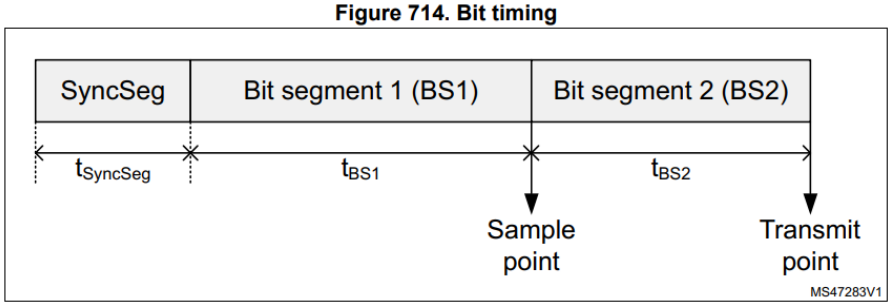
Para transferência de dados, distinguem-se três formatos de quadro, configuráveis pelos *bits* `FDCAN_CCCR_BRSE` e `FDCAN_CCR_FDOE`:

1. **Quadro CAN Clássico (quadro curto)**: Formato padrão definido na especificação CAN 2.0 A e B.
2. **Quadro CAN FD (quadro longo) sem Comutação de Taxa de *Bits***: Quadro CAN FD com taxa de *bits* fixa, definida no registrador FDCAN_DBTP.
3. **Quadro CAN FD (quadro longo) com Comutação de Taxa de *Bits***: Quadro CAN FD que permite alterar a taxa de *bits* durante a fase de dados. A taxa de *bits* nominal (para fase de arbitragem) é definida no registrador FDCAN_NBTP, enquanto a taxa de *bits* rápida (para fase de dados) é definida no registrador FDCAN_DBTP.

No contexto de redes de comunicação, como a CAN (Controller Area Network), o tempo de bit é um parâmetro crucial que determina como os dados são transmitidos ao longo da rede. O **tempo de *bit* nominal** é utilizado durante a fase de arbitragem, onde os dispositivos sincronizam sua comunicação, enquanto o **tempo de *bit* de dados** se refere à transmissão efetiva dos dados. Esses tempos são segmentados para garantir a precisão e a sincronização na comunicação. O circuito de temporização de *bits* do FDCAN monitora a linha de barramento serial e realiza a amostragem e o ajuste do ponto de amostra, sincronizando-se na borda do *bit* de início e se ressincronizando nas bordas seguintes.

A figura a seguir esquematiza a temporização de um *bit* no FDCAN, em conformidade com o padrão CAN. O tempo de um *bit* é dividido em 3 segmentos:

Figure 714. Bit timing



- **Segmento de Sincronização (SYNC_SEG)**: neste segmento, espera-se que ocorra uma mudança de *bit*. Esse tempo tem uma duração fixa de um quantum de tempo (1 x tq).
- **Segmento de Bit 1 (BS1)**: define a localização do ponto de amostra. Inclui o **PROP_SEG** e o **PHASE_SEG1** do padrão CAN. Sua duração é programável entre 1 e 16 quanta de tempo, mas pode ser automaticamente estendida para compensar desvios positivos de fase, que ocorrem devido a diferenças nas frequências dos vários nós da rede.
- **Segmento de Bit 2 (BS2)**: define a localização do ponto de transmissão. Representa o **PHASE_SEG2** do padrão CAN. Sua duração é programável entre um e oito quanta de tempo, mas também pode ser automaticamente encurtada para compensar desvios negativos de fase.

A taxa de transmissão, ou *baud rate*, é o inverso do tempo de *bit*, que, por sua vez, é a soma de três componentes. $t_{SyncSeg}+t_{BS1}+t_{BS2}$, onde:

- Para o tempo de *bit* nominal, utilizamos os seguintes componentes:
  - tq (quantum de tempo): $tq=(\texttt{FDCAN\_NBTP\_NBRP}[8:0]+1)\times t_{fdcan\_tq\_ck}$
  - $t_{SyncSeg}$ (segmento de sincronização): $t_{SyncSeg}=1$ tq
  - t_{BS1} (segmento de *bit* 1): $t_{BS1}=tq\times(\texttt{FDCAN\_NBTP\_NTSEG1}[7:0]+1)$
  - t_{BS2} (segmento de *bit* 2): $t_{BS2}=tq\times(\texttt{FDCAN\_NBTP\_NTSEG2}[6:0]+1)$
- Para o tempo de *bit* de dados, o cálculo é semalhante
  - tq (quantum de tempo): $tq=(\texttt{FDCAN\_DBTP\_DBRP}[4:0]+1)\times t_{fdcan\_tq\_ck}$
  - $t_{SyncSeg}$ (segmento de sincronização): $t_{SyncSeg}=1$ tq
  - t_{BS1} (segmento de *bit* 1): $t_{BS1}=tq\times(\texttt{FDCAN\_DBTP\_DTSEG1}[4:0]+1)$
  - t_{BS2} (segmento de *bit* 2): $t_{BS2}=tq\times(\texttt{FDCAN\_DBTP\_DTSEG2}[3:0]+1)$

Os registradores `FDCAN_NBTP` e `FDCAN_DBTP` são responsáveis pelo *pre-scaler* do quantum de tempo, que, em geral, corresponde ao *clock* de núcleo, `fdcan_ker_ck`. Esses registradores definem o tempo de *bit* nominal e o tempo de *bit* de dados desejado.

A calibração do *clock* é crucial, pois afeta o *clock* de quantum de tempo (`fdcan_tq_ck`) usado nos cálculos. O FDCAN possui uma unidade de calibração de clock (CCU) que gera um *clock* calibrado entre 0,5 e 25 MHz. A precisão da calibração depende de fatores como a tolerância do *clock* de entrada e erros de medição. Note que, quando a calibração de *clock* está ativa, os *pre-scalers* de *baud rate* devem estar inativos. E a configuração do tempo de *bit* só pode ser realizada quando o dispositivo está em modo de espera (*Standby*). No modo de inicialização, o *bit* `FDCAN_CCCR_CCE` precisa ser ativado para permitir alterações na configuração.

O FDCAN gerencia de forma independente os índices de mensagens para transmissão e recepção, utilizando FIFOs (First-In, First-Out) e *buffers* dedicados. Quando uma mensagem é recebida, o FDCAN a submete a um processo de filtragem de aceitação, configurado por meio dos registradores `FDCAN_GFC`, `FDCAN_SIDFC` (para IDs de 11 *bits*) e `FDCAN_XIDFC` (para IDs de 29 *bits*). Se a mensagem passar na filtragem, ela é armazenada em um *buffer* de recepção dedicado ou em uma das duas FIFOs de recepção (FDCAN_RXF0 e FDCAN_RXF1), conforme definido pelo elemento de filtro correspondente. Cada FIFO possui dois índices: o `Put Index` (no campo `FDCAN_RXFnS_FnPI`), que indica a próxima posição livre para armazenamento de uma nova mensagem, e o **Get Index** (no campo `FDCAN_RXFnS_FnGI`), que indica a próxima mensagem a ser lida. O `Put Index` é automaticamente incrementado pelo *hardware* sempre que uma nova mensagem é adicionada, enquanto o `Get Index` é atualizado pelo *software* através do registrador `FDCAN_RXFnA_FnA` (`Acknowledge Index`). O *software* deve escrever o índice do último elemento lido no registrador `FDCAN_RXFnA_FnA`. Após a leitura de uma mensagem, o índice do último elemento lido deve ser escrito nesse registrador, permitindo que o `Get Index` avance para o próximo elemento. Se a FIFO estiver cheia (quando o `Put Index` é igual ao `Get Index`), novas mensagens serão descartadas, e essa condição será sinalizada.

Em modo de sobrescrever (configurado por FDCAN_RXFnC_FnOM = 1), uma nova mensagem aceita pode substituir a mensagem mais antiga quando a FIFO está cheia.

Para a transmissão, o FDCAN possui até 32 *buffers* de transmissão dedicados, configurados pelo registrador FDCAN_TXBC. Além disso, pode utilizar uma FIFO ou fila de transmissão, também configurada por FDCAN_TXBC. Assim como nas FIFOs de recepção, a FIFO de transmissão tem índices de Put (FDCAN_TXFQS_TFQPI) e Get (FDCAN_TXFQS_TFGI). O Put Index é incrementado a cada solicitação de adição de mensagem à fila, através do registrador FDCAN_TXBAR, enquanto o Get Index é incrementado automaticamente pelo *hardware* após cada transmissão. A ordem de transmissão das mensagens depende da configuração da FIFO: se for uma FIFO, as mensagens são transmitidas na ordem em que foram adicionadas. Se for uma fila, a transmissão ocorre com base na prioridade, determinada pelo ID da mensagem.

O FDCAN suporta dois conjuntos de filtros de aceitação: um para identificadores padrão (11 *bits*) e outro para identificadores estendidos (29 *bits*). Cada conjunto pode incluir múltiplos elementos de filtro, que podem ser configurados para corresponder a IDs específicos de mensagens, intervalos de IDs ou máscaras de *bits*. A configuração dos filtros de aceitação requer a programação de diversos registradores do FDCAN, como o FDCAN_GFC, que controla o comportamento geral da filtragem, especificando como tratar mensagens não correspondentes e se os quadros que não correspondem a nenhum filtro devem ser aceitos ou rejeitados. Os registradores FDCAN_SIDFC e FDCAN_XIDFC são utilizados para configurar as listas de filtros para IDs de 11 e 29 *bits*, respectivamente, enquanto o FDCAN_XIDAM fornece uma máscara adicional para IDs estendidos, permitindo uma filtragem mais flexível. A verificação dos filtros configurados em uma lista é realizada de forma sequencial, interrompendo o processamento assim que uma correspondência é encontrada, o que aciona a ação designada. Observe que o FDCAN não verifica se há configurações inválidas na RAM.

Cada elemento de filtro, seja *standard* ou *extended*, é configurado individualmente na memória RAM, nos endereços especificados nos registradores FDCAN_SIDFC ou FDCAN_XIDFC. Os tipos de filtro disponíveis incluem o **filtro de intervalo**, que define um intervalo de IDs aceitos; o **filtro de ID duplo**, que permite a correspondência com dois IDs específicos; e o **filtro de máscara de *bits* clássica**, que utiliza uma máscara para determinar quais *bits* do ID devem corresponder exatamente e quais podem ser ignorados. Para especificá-los, o elemento de filtro contém 4 campos: SFT especifica o tipo de filtro (00: filtro de intervalo de SFID1 a SFID2; 01: filtro dual para aceitar apenas mensagens com SFID1 ou SFID2; e 10: filtro clássico SFID1 cujos bits de filtragem são definidos pela máscara SFID2) e SFEC define o comportamento do FDCAN quando o filtro correspondente é ativado (001: armazena em Rx FIFO 0 se o filtro corresponder; 010: armazena em Rx FIFO se o filtro correponder; 011: rejeita o campo de dados se o filtro corresponder; 100: define a *flag* de interrupção de alta prioridade se o filtro corresponder; 101: define a *flag* de interrupção de alta prioridade e armazena o campo de dados no Rx FIFO 0 se o filtro corresponder; 110: define a *flag* de interrupção de alta prioridade e armazena o campo de dados no Rx FIFO 1 se

o filtro corresponder; 111: armazena o campo de dados em um *buffer* Rx dedicado ou como uma mensagem de depuração).

Table 495. Standard message ID filter element

Bit	31		24	23	16	15	8	7	0
S0	SFT[1:0]	SFEC[2:0]		SFID1[10:0]		Res.		SFID2[10:0]	

Essa configuração deve ser feita no **modo de inicialização**, assim como outras alterações nas configurações do FDCAN. Para colocar um FDCAN no modo de inicialização, é fundamental desativar o modo *Sleep*, resetando o *bit* FDCAN_CCCR_CSR. Esse passo é essencial para retomar a comunicação CAN após o periférico ter sido colocado em modo de baixo consumo de energia. A sequência completa para sair do modo *Sleep* inclui a reativação dos *clocks*, a limpeza do *bit* `FDCAN_CCCR_CSR` e a atribuição de "1" ao *bit* `FDCAN_CCCR_INIT`. O FDCAN confirma a saída do modo Sleep ao resetar o *bit* `FDCAN_CCCR_CSA`. O acesso aos registradores de configuração do FDCAN só é permitido quando o *bit* `FDCAN_CCCR_INIT` está definido como 1. Portanto, é importante aguardar que este *bit* seja definido como "1" para garantir que o código de configuração não tente acessar os registradores antes que o modo de inicialização esteja realmente ativo, prevenindo assim erros e comportamentos inesperados.

Além disso, é necessário habilitar a mudança de configuração no FDCAN, permitindo que a CPU escreva nos registradores de configuração protegidos, definindo o *bit* `FDCAN_CCCR_CCE` como "1". Após concluir as configurações nos registradores do FDCAN, para iniciar a comunicação CAN, é preciso resetar o *bit* `FDCAN_CCCR_INIT`. Isso fará com que o FDCAN se sincronize com a transferência de dados no barramento CAN, aguardando a detecção de uma sequência de 11 *bits* recessivos consecutivos (indicando um estado de barramento ocioso - *Bus_Idle*). Após essa sincronização, o FDCAN estará apto a participar das atividades do barramento e iniciar a transferência de mensagens.

O FDCAN possui duas linhas de interrupção, `fdcan_intr0_it` e `fdcan_intr1_it`, para gerenciar uma variedade de eventos, cada um com propósitos distintos. Essas interrupções são habilitadas pelos respectivos bits no registrador FDCAN_IE. No entanto, esse registrador não define qual linha de interrupção será utilizada para sinalizar o evento. A associação entre o evento e a linha de interrupção é realizada pelo registrador FDCAN_ILS. Através desse registrador, podemos configurar qual das duas linhas irá atender ao evento habilitado: se desejamos associar o evento à linha 0, devemos resetar o valor para 0; se preferimos associá-lo à linha 1, devemos definir o valor como 1.

ttfdcan_intr0_it	26	19	FDCAN1_IT0	TTCAN Interrupt 0	0x0000 008C
fdcan_intr0_it	27	20	FDCAN2_IT0	FDCAN Interrupt 0	0x0000 0090
ttfdcan_intr1_it	28	21	FDCAN1_IT1	TTCAN Interrupt 1	0x0000 0094
fdcan_intr1_it	29	22	FDCAN2_IT1	FDCAN Interrupt 1	0x0000 0098

Além disso, é necessário habilitar a linha de interrupção selecionada utilizando o registrador FDCAN_ILE. Portanto, a configuração das interrupções no FDCAN exige uma sequência de passos, desde a habilitação da linha de interrupção até a ativação de interrupções específicas para eventos. Esse processo assegura que a CPU esteja adequadamente informada e possa responder de maneira eficiente a esses eventos. Observe que o *bit* de estado do evento de interrupção deve ser limpo pelo *software*, escrevendo “1” no *bit* correspondente no registrador FDCAN_IR.