DISCIPLINA EA701
Introdução aos Sistemas Embarcados

ROTEIRO 11: REDES INDUSTRIAIS

Topologias de Rede, Modelos de Comunicação, Arbitragem de Barramento, Mecanismos de Sincronização e Robustez, Filtragem de Mensagens, Rede CAN (Protocolo CAN, *Transceiver* MCP2551), Rede DMX

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

FEEC / UNICAMP

Revisado e modificado em maio de 2025 por Ting com auxílio do Chatgpt
Revisado em outubro de 2024

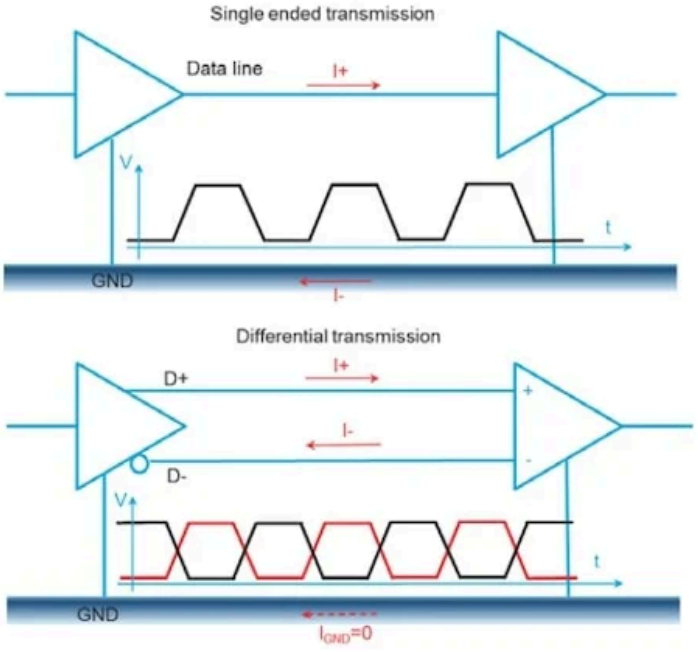
INTRODUÇÃO

Nos sistemas embarcados modernos, sensores e atuadores são comumente integrados ao microcontrolador utilizando interfaces seriais como SPI (do inglês *Serial Peripheral Interface*) e I2C (do inglês *Inter-Integrated Circuit*). Essas interfaces são populares por sua simplicidade, baixo custo e eficiência em curtas distâncias. No Roteiro 10, vimos que o I2C é ideal para conectar múltiplos dispositivos com apenas duas linhas de comunicação (SDA e SCL), tornando-o apropriado para sistemas com restrição de espaço e número limitado de fios. Já o SPI, com suas quatro linhas (MISO, MOSI, SCLK e SS), oferece maior velocidade de comunicação e é preferido em aplicações que exigem maior taxa de transferência de dados.

Historicamente, antes do advento dessas interfaces otimizadas para comunicação interna em placas, o protocolo RS-232 foi um dos primeiros e mais difundidos padrões para comunicação serial entre dispositivos, como computadores e periféricos. Embora ainda seja encontrado em algumas aplicações legadas e se comunicam com os periféricos UART/USART dos microcontroladores através de transceptores (em inglês, *transceivers*), como um MAX232, suas limitações de velocidade, distância e sensibilidade a ruído o tornaram menos adequado para as exigências de sistemas embarcados modernos e ambientes industriais.

Contudo, tanto o SPI quanto o I2C (e o RS-232, em maior grau) apresentam limitações significativas em ambientes industriais e automotivos. Esses são frequentemente classificados como ambientes hostis devido à presença de ruído eletromagnético, variações extremas de temperatura, umidade e vibração mecânica. Em tais ambientes, onde falhas de comunicação podem comprometer a segurança e a funcionalidade do sistema como um todo, a evolução natural é a adoção de redes de comunicação mais resilientes, como **a rede CAN** (do inglês *Controller Area Network*). Desenvolvida originalmente para a indústria automotiva pela Bosch, a rede CAN tornou-se um padrão em sistemas embarcados críticos por diversos motivos, destacando-se:

- **alta imunidade a ruídos**: Diferente dos sinais unipolares (em inglês *single-ended*) utilizados nas interfaces UART, SPI e I2C, que transmitem um sinal referenciado ao terra, o **sinal diferencial** utilizado na rede CAN transmite a informação pela diferença de potencial entre duas linhas (CAN_H e CAN_L), tornando-o muito mais resistente a interferências comuns no ambiente industrial e automotivo.



Fonte: All About Circuits.

- **detecção e correção de erros**: O protocolo implementa mecanismos sofisticados de verificação de integridade dos dados, como CRC (do inglês *Cyclic Redundancy Check*), detecção de *bits* dominantes e recessivos, além de retransmissão automática em caso de falha.

- **topologia distribuída e flexível**: Um aspecto crucial da rede CAN é a sua capacidade de gerenciar a comunicação de múltiplos nós de forma descentralizada e sem colisões, através de um sofisticado mecanismo de **arbitragem de barramento**, onde as mensagens mais prioritárias (determinadas pelo identificador) têm precedência na transmissão, garantindo previsibilidade e determinismo — requisitos fundamentais em sistemas embarcados críticos.
- **escalabilidade**: Enquanto no SPI o número de dispositivos é diretamente limitado pela necessidade de uma linha de seleção dedicada (\CS) para cada escravo, e no I2C pelo espaço de endereçamento (normalmente 7 *bits*, limitando a 127 dispositivos teóricos), a rede CAN permite a coexistência de múltiplos nós no mesmo barramento, com a seleção lógica baseada no conteúdo da mensagem (ID), o que oferece uma escalabilidade muito superior, além de simplificar o cabeamento.
- **alcance maior dos sinais**: Enquanto I2C e SPI são geralmente limitados a poucos metros de cabo, a rede CAN pode operar eficientemente em distâncias superiores a 40 metros em velocidades mais altas (até 1 Mbps) e até 1 km em velocidades reduzidas.
- **padrões e interoperabilidade**: A padronização da rede CAN (como CAN 2.0A/B e CAN FD) garante compatibilidade entre dispositivos de diferentes fabricantes, o que facilita a manutenção e o desenvolvimento modular de sistemas.

Além da evolução física dos meios e da robustez dos sinais, essa transição representa também uma mudança na granularidade da informação transmitida, onde passamos de protocolos que basicamente trocam bytes e bits de forma sequencial (como UART, SPI e I2C) para redes que operam no nível de **mensagens estruturadas**, cada uma identificada e tratada conforme sua prioridade ou endereço lógico no barramento. Esse conceito é essencial para compreender como redes como a CAN conseguem oferecer comunicação eficiente, escalável e determinística.

A adoção da rede CAN em sistemas embarcados marca não apenas uma evolução em termos físicos e de confiabilidade, robustez e eficiência do meio de transmissão, mas também a entrada em uma nova abordagem conceitual baseada em redes de controle distribuídas. Nesse novo paradigma, dispositivos não são apenas sensores ou atuadores isolados conectados a um microcontrolador, mas **nós inteligentes** que compartilham informações em um barramento comum e cooperam para o funcionamento de todo o sistema. Para compreender plenamente essa transição e os benefícios da rede CAN, é essencial introduzir alguns conceitos fundamentais de **redes de controle**, que irão sustentar a arquitetura e os protocolos utilizados.

Neste Roteiro, aprofundaremos nosso estudo das redes de controle. Começaremos com uma abordagem prática, explorando o periférico FDCAN (do inglês *Flexible Data-Rate Controller Area Network*) integrado no STM32H7A através de três projetos-exemplo: a implementação da função de loopback, a filtragem de identificadores e a montagem de uma rede CAN completa. Em seguida, passaremos para os fundamentos teóricos. Abordaremos as **topologias de rede** e os diversos tipos de comunicação que nelas ocorrem, os quais exigem um conceito esencial: a **arbitragem de barramento**. Introduziremos o **sinal diferencial**, contrastando-o com o sinal unipolar que abordamos anteriormente. Para garantir a confiabilidade e robustez da comunicação, apresentaremos **mecanismos modernos de detecção e correção de erros** integrados em protocolos como o CAN. Além da checagem de *bit* de paridade (detecção de erros em um *bit*) e *handshaking* (processo de negociação que pode revelar a ocorrência de um erro) já vistos em roteiros anteriores, exploraremos conceitos como verificação cíclica de redundância (em inglês, *Cyclic Redundancy Check* – CRC), *bit stuffing* e a confirmação da integridade da mensagem.

Com esses conceitos estabelecidos, mostraremos duas redes de controle práticas: a rede CAN e a rede DMX. O protocolo CAN, projetado para transmissão de dados em distâncias médias, utiliza linhas de transmissão diferencial, o que o torna ideal para cenários críticos e de alta demanda. Já o protocolo DMX é amplamente empregado no controle de iluminação e efeitos especiais, utilizando um barramento serial unidirecional para a transmissão de dados.

PROJETOS-EXEMPLO

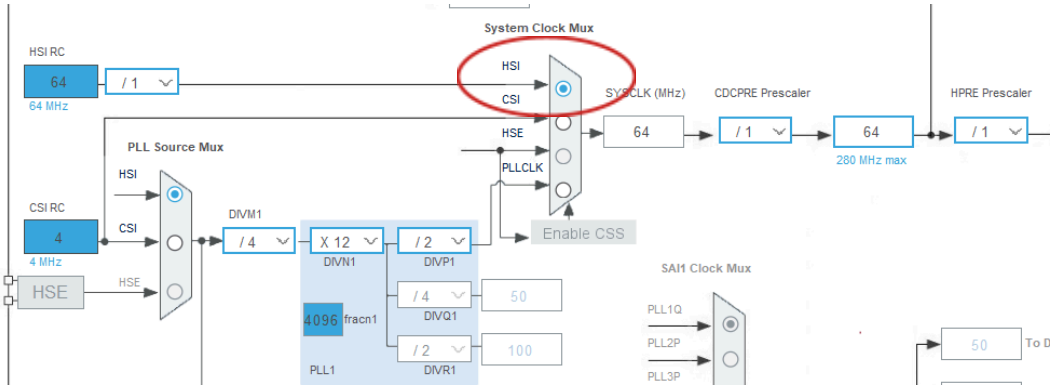
Nesta seção, exploraremos a implementação de um barramento CAN utilizando o periférico FDCAN (do inglês *Flexible Data-Rate Controller Area Network*) do microcontrolador STM32H7A. Apresentaremos três projetos práticos, organizados de forma progressiva, para introduzir os conceitos e as configurações necessárias para o barramento. O **FDCAN** é uma evolução do clássico protocolo CAN, cuja característica mais distintiva, e o motivo do "FD" no nome, é permitir que a fase de dados de um quadro CAN seja transmitida a uma taxa de *bits* mais alta do que a fase de arbitragem (onde o ID da mensagem é transmitido). Isso significa que, após a arbitragem bem-sucedida, a comunicação pode acelerar significativamente (podendo atingir 5 Mbps ou mais), otimizando a largura de banda. A fase de arbitragem continua na velocidade nominal para garantir a robustez. O STM32H7A3 possui 2 controladores FDCAN do tipo *Flexible-Datarate*, mas vamos usar apenas um deles, em modo padrão.

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 de um transmissor com o Rx de um receptor, 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.

Os controladores FDCAN do STM32H7 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.

1. Crie um projeto chamado “CAN_Base”, sem inicializar os periféricos. Ative o *Debug*. Selecione HSI como fonte do *Clock* do Sistema e mantenha o *clock* geral em 64MHz (configurações padrão). Gere o código.



2. Abra o arquivo “main.c”. Vamos criar variáveis globais para os *buffers* de dados de transmissão e recepção e uma variável geradora de dados. Na seção /* USER CODE BEGIN PV */

```
uint8_t                 TxData[8];
uint8_t                 RxData[8];
int indx = 0;
```

Insira no escopo /* USER CODE BEGIN PFP */ os protótipos das funções que implementaremos

```
void FDCAN2_Init(void);
void FDCAN2_AddMessageToTxFifoQ (uint32_t id);
void FDCAN2_GetMessageFromRxFifo0 (void);
void FDCAN2_AlocaSRamCANSegmentos (void);
uint8_t Code2ByteCounter(uint8_t code);
```

3. Vamos implementar as 5 funções no escopo /* USER CODE BEGIN 4 */. Para a primeira função de inicialização do FDCAN, insira o seguinte código:

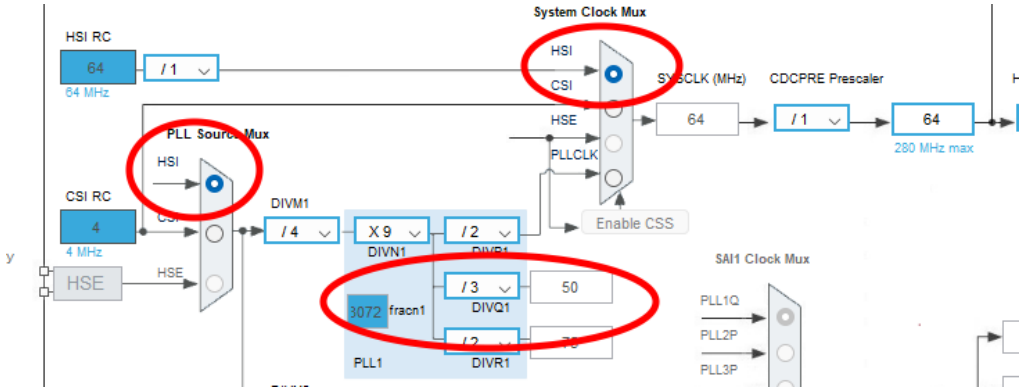
```
void FDCAN2_Init () {
   {
      //Configura o sinal de relogio pll1_q_ck para fdcan_ker_clk
      //A fonte HSI ja eh habilitada e configurada pelo MxCube
      //Desabilita PLL1
      RCC->CR &= ~RCC_CR_PLL1ON;
      while (RCC->CR & RCC_CR_PLL1RDY); //aguarda a desabilitacao
      //Configura 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
    }
    {
        //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; // 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
        /* Habilita IRQ correspondente a FDCAN2 interrupt Init (Linha de
interrupção 0) */
        NVIC_SetPriority(FDCAN2_IT0_IRQn, 1);
        NVIC_EnableIRQ(FDCAN2_IT0_IRQn);
    }
    {
        //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
        // 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;
        // 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)|//Nom.Time Segm. 1
          (((uint32_t)13U - 1U) << FDCAN_NBTP_NTSEG2_Pos)|//Nom.Time Segm. 2
          (((uint32_t)1U - 1U) << FDCAN_NBTP_NBRP_Pos));//Nom.BitRate PScaler
    }
    {
        // 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
        FDCAN2_AlocaSRamCANSegmentos ();
    }
    {
        // 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
    }
}
```

Note que, por didática, as instruções foram agrupadas em 8 blocos delimitados pelas chaves.

O primeiro bloco configura o circuito PLL1 (do inglês *Phase-Locked Loop 1*) do microcontrolador STM32H7A para gerar um sinal de *clock* de 50MHz para o periférico FDCAN. O objetivo é ajustar a "velocidade" e a "precisão" do sinal de relógio que o FDCAN usará para operar em 50MHz. Durante a configuração, o PLL1 é temporariamente desabilitado. A fonte de *clock* HSI (*clock* padrão) é usada, e um *prescaler* de 4 é aplicado antes de entrar no PLL1. Em seguida, os fatores de divisão e multiplicação, incluindo o fator fracionário do PLL1, são configurados para que o sinal de saída final seja de 50MHz. Por fim, as configurações dos divisores são habilitados antes de reabilitar PLL1. A figura a seguir sintetiza os campos de *bits* configurados, resultando em um sinal de 50MHz para o FDCAN. É importante assegurar que a faixa de frequências de operação do PLL1 estar entre 8 e 16 MHz e a faixa de frequência do VCO (do inglês, *Voltage Controlled Oscillator*), utilizado pelo PLL1, entre 128 e 560 MHz.

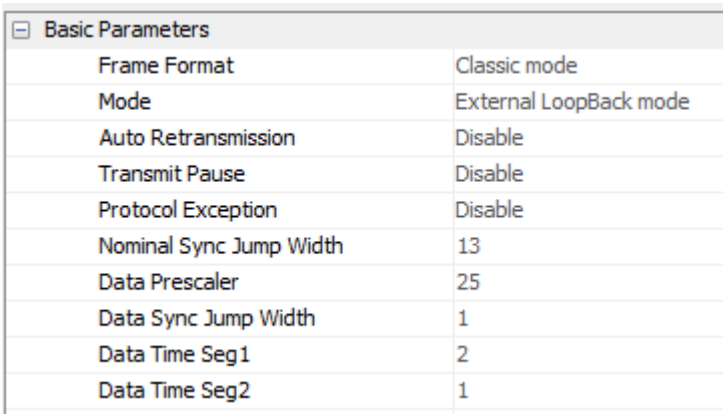


O segundo bloco realiza a habilitação e configuração inicial do periférico FDCAN2 e seus pinos GPIO associados no microcontrolador STM32H7A, além de configurar a interrupção. Ele prepara o *hardware* para que o FDCAN2 possa se comunicar. O código primeiramente habilita o *clock gating* para o FDCAN2 e seus componentes auxiliares, garantindo que recebam energia e sinais de *clock*. Ele seleciona `pll1_q_ck` (o *clock* de 50MHz configurado anteriormente) como a fonte de *clock* principal para o FDCAN. Em seguida, as linhas GPIO PB13 (para FDCAN2_TX) e PB5 (para FDCAN2_RX) são configuradas em modo de função alternativa (AF9), conectando-as internamente ao periférico FDCAN2. Por fim, a interrupção `FDCAN2_IT0_IRQn` é habilitada e configurada com prioridade 0, permitindo que o microcontrolador responda a eventos específicos do FDCAN2.

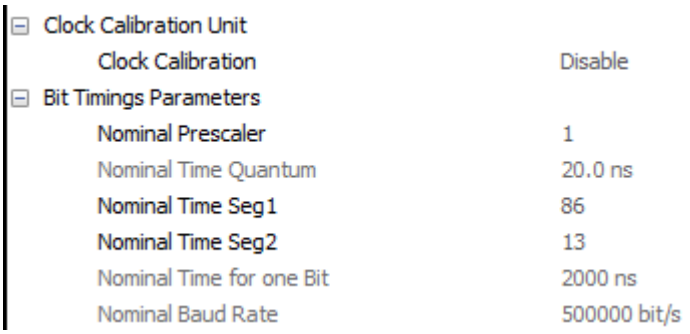
O terceiro bloco de instruções coloca o periférico FDCAN2 no modo de inicialização (em inglês *Initialization mode*). O código garante que o FDCAN2 saia do modo *Sleep*, um estado de baixa energia, para então ativá-lo no modo de inicialização. Este modo é essencial, pois permite a configuração dos parâmetros operacionais do FDCAN, como as taxas de *baud*, filtros, e outros ajustes antes que o periférico possa começar a enviar ou receber mensagens na rede CAN. O código também inclui laços de *while* para aguardar a confirmação de que o FDCAN2 realmente saiu do *Sleep* e entrou no modo de inicialização, garantindo que as próximas configurações sejam aplicadas corretamente.

O quarto bloco de instruções realiza a configuração detalhada dos parâmetros operacionais do periférico FDCAN2, incluindo o modo de comunicação, regras de retransmissão e a

temporização de *bits*. O código habilita o modo de configuração do FDCAN2 (`CCE`) e, em seguida, desabilita a retransmissão automática (`DAR`) e o recurso de pausa de transmissão (`TXP`). Ele também desativa o processamento de exceção de protocolo (`PXHD`) e define o formato do quadro como CAN Clássico, desabilitando as funcionalidades *Flexible Data-Rate* (`FDOE`) e *Bit Rate Switch* (`BRSE`). Além disso, configura o FDCAN2 para **operar em modo *Loopback* Externo**. Por fim, o código ajusta os parâmetros de temporização nominal do *bit* (`NBTP`), incluindo o *prescaler* de taxa de *bit*, os segmentos de tempo (`NTSEG1`, `NTSEG2`) e a largura do *jump* de sincronização (`NSJW`). Esses tempos são relacionados com a taxa de transmissão do FDCAN e são cruciais para garantir a correta sincronização e comunicação na rede CAN. As seguintes figuras sumarizam os valores configurados neste bloco. Note na última linha que a *baud rate* nominal estabelecida é de 500.000 *bits* por segundo.



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



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 quinto bloco realiza a configuração do tamanho dos campos de dados (*payload*) para as mensagens de transmissão (TX) e recepção (RX) do periférico FDCAN2, além de definir o modo de operação da fila de transmissão. O código primeiro define que o FDCAN2 usará uma fila de transmissão (TxFIFO) em vez de *buffers* dedicados, limpando o *bit* `TFQM`. Em seguida, ele configura o tamanho da carga útil (*payload*) para as mensagens de transmissão (`TXESC`) e recepção (`RXESC`) para 8 *bytes* cada. Essa configuração é essencial para o FDCAN2, especialmente quando operando em modo CAN Clássico (onde o *payload* máximo é 8 *bytes*, como configurado no bloco anterior), garantindo que o controlador saiba o volume de dados que espera enviar e receber por mensagem.

O sexto bloco configura as áreas de memória RAM dedicadas ao periférico FDCAN2 para armazenar filtros, *buffers* de recepção e transmissão. Essas instruções são agrupadas na função `FDCAN2_AlocaSRamCANSegmentos` que implementaremos e detalharemos no próximo item.

O sétimo bloco tira o periférico FDCAN2 do modo de inicialização, tornando-o operacional para a transmissão e recepção de dados na rede CAN. O código desativa o *bit* `INIT` no registrador `CCCR` do FDCAN2, o que o remove do modo de inicialização. Em seguida, um laço `while` é utilizado para aguardar a confirmação de que o periférico realmente saiu desse modo. Após a execução bem-sucedida deste bloco, o FDCAN2 estará pronto para participar ativamente da comunicação no barramento CAN, seja enviando ou recebendo mensagens, de acordo com as configurações previamente estabelecidas.

Por último, o oitavo bloco habilita a linha de interrupção 0 do FDCAN2 e a configura para disparar quando uma nova mensagem é recebida na *Rx FIFO 0*. O código habilita a linha de interrupção 0 (`EINT0`) no registrador `ILE` do FDCAN2, que é o ponto de entrada para o sistema de interrupções. Em seguida, ele ativa o evento específico de interrupção para novas mensagens na *Rx FIFO 0* (`RF0NE`) através do registrador `IE`. Por fim, o código associa explicitamente este evento à linha de interrupção 0, garantindo que o microcontrolador será notificado via interrupção sempre que uma nova mensagem CAN for recebida e armazenada na *Rx FIFO 0* do FDCAN2.

4. Vamos implementar a função `FDCAN2_AlocaSRamCANSegmentos`, adicionando o seguinte código depois da função `FSCAN2_Init`.

```
void FDCAN2_AlocaSRamCANSegmentos (void) {
 uint32_t Origem = 0;
 /* Endereco inicial da lista de filtros padrao (Standard ID Filter List)
 */
 MODIFY_REG(FDCAN2->SIDFC, FDCAN_SIDFC_FLSSA, (Origem <<
 FDCAN_SIDFC_FLSSA_Pos));
 /* Quantidade de elementos na lista de filtros padrao (cada elemento = 1
 palavra de 32 bits) */
 MODIFY_REG(FDCAN2->SIDFC, FDCAN_SIDFC_LSS, (1U << FDCAN_SIDFC_LSS_Pos));
 /* Endereco inicial de Rx FIFO 0 */
 Origem += (1U);   // Avança 1 palavra apos os filtros
 MODIFY_REG(FDCAN2->RXF0C, FDCAN_RXF0C_F0SA, (Origem <<
 FDCAN_RXF0C_F0SA_Pos));
 /* Quantidade de elementos na Rx FIFO 0 (cada elemento tem o tamanho fixo
configurado do payload maximo. Para playload de ate 8 bytes são 16 bytes -
4 palavras de 32 bits (2 de header + 2 de carga util). */
 MODIFY_REG(FDCAN2->RXF0C, FDCAN_RXF0C_F0S, (1U << FDCAN_RXF0C_F0S_Pos));
 /* Endereco inicial de Tx buffer  */
 Origem += (4U); // Avança 4 palavras (tamanho fixo de 1 elemento RX FIFO0)
 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
  (soma das palavras dos filtros, RX FIFO e TX FIFO)*/
 for (uint32_t RAMcounter = SRAMCAN_BASE; RAMcounter <
 SRAMCAN_BASE+(1U+4U+4U)*4U; RAMcounter += 4U)
 {
      *(uint32_t *)(RAMcounter) = 0x00000000;
 }
}
```

Os periféricos FDCAN exigem uma região de memória interna dedicada para processar diversas estruturas de dados, como filtros, *buffers* de recepção e transmissão. Essa área é chamada de *CAN Message RAM*, ou `SRAM CAN`, e deve ser alocada manualmente em uma região apropriada da SRAM do microcontrolador. O endereço base dessa RAM é configurado por meio da macro `SRAMCAN_BASE`, que informa aos periféricos FDCAN onde começam seus dados dentro da memória RAM. A macro está definida no arquivo `stm32h7a3xxq.h`. É importante notar que todos os endereços utilizados pelos registradores internos dos periféricos FDCAN são, na verdade, *offsets* relativos a esse endereço base da `SRAM CAN`, e não endereços absolutos da CPU.

A função `FDCAN2_AlocaSRamCANSegmentos` é responsável por gerenciar o *layout* da `SRAM CAN`, definindo os endereços iniciais e os tamanhos das estruturas de dados. Ela configura o endereço de início e o número de elementos para a lista de filtros padrão (SIDFC). Em seguida, estabelece o endereço de início e a capacidade da *Rx FIFO 0 (RXF0C)*. Por fim, define o endereço de início e o tamanho da fila/*buffer* de transmissão (TXBC). A variável `Origem` é utilizada para calcular esses endereços de forma incremental dentro da `SRAM CAN`. Após a configuração das posições, o código limpa toda a área da memória que será utilizada, preenchendo-a com zeros para assegurar um estado inicial limpo antes das operações do FDCAN2. Segue-se um resumo do *layout* configurado com `FDCAN2_AlocaSRamCANSegmentos` para a CAN Message RAM do FDCAN2.

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

A macro `MODIFY_REG` definida no arquivo-cabeçalho stm32h7a3xxq.h

```
#define MODIFY_REG(REG, CLEARMASK, SETMASK)  \
  ((REG) = (((REG) & (~(CLEARMASK))) | (SETMASK)))
```

realiza dois passos atômicos em um único comando: aplica a máscara CLEARMASK para zerar os *bits* que se quer alterar e a máscara SETMASK para setar os novos valores desses *bits*. Os *bits* que não estão envolvidos nas máscaras **permanecem inalterados**.

5. Diferentemente de protocolos como UART, SPI e I2C, que operam com a transmissão sequencial de *bytes* (ou *bits*), o protocolo CAN é orientado à mensagem. Isso significa que, no CAN, o que circula pela rede não são *bytes* isolados enviados de um transmissor para um receptor, mas sim mensagens estruturadas e autocontidas, compostas por campos bem definidos como identificador, controle, dados, verificação e delimitação. Vamos adicionar as

funções `FDCAN2_AddMessageToTxFifoQ` **e** `Code2ByteCounter` para envio de mensagens em /* 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);
}

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 elem. Index no TxFIFO/Queue */
      FDCAN2->TXBAR = ((uint32_t)1 << PutIndex);
  }
}
```

A função `Code2ByteCounter` é uma função auxiliar que converte um código numérico, que especifica o tamanho do *payload*, para o número real de *bytes* de dados. Por exemplo, code 0 retorna 8 *bytes* (2 palavras de 4 *bytes*), code 7 retorna 64 *bytes* (16 palavras de 4 *bytes*).

A função `FDCAN2_AddMessageToTxFifoQ` tem como objetivo preparar e agendar uma mensagem CAN para transmissão usando a *Tx FIFO/Queue* do FDCAN2. Ela verifica se a fila de transmissão está habilitada e não está cheia. Se houver espaço, a função calcula o próximo índice disponível na fila. Em seguida, ela monta as duas palavras de cabeçalho da mensagem CAN (que incluem o ID da mensagem, padrão do ID (11 *bits*), tipo de quadro (Clássico), e comprimento dos dados (8 *bits*)), e as escreve na `SRAM CAN` no local correto da fila. Após isso, os dados da mensagem (lidos de um *array*/vetor `TxData` global) são copiados para a mesma área de memória. Finalmente, a função solicita a transmissão da mensagem recém-adicionada ao FDCAN2, ativando o *bit* correspondente ao índice da mensagem no registrador `TXBAR`, o que permite que o *hardware* inicie o processo de envio.

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.

6. Agora, vamos implementar a função `FDCAN2_GetMessageFromRxFifo0` para recepção de mensagens, que também será adicionada ao escopo /* USER CODE BEGIN 4 */

```
void FDCAN2_GetMessageFromRxFifo0 (void) {
   uint32_t GetIndex = 0;
   uint32_t *RxAddress;
   uint32_t ByteCounter;

   //Bytes a serem lidos
   ByteCounter = Code2ByteCounter
      ((FDCAN2->RXESC & FDCAN_RXESC_F1DS) >> FDCAN_RXESC_F1DS_Pos);
```

```
   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);
          /* Obtem o tamanho de dados de RX FIFO0 na unidade de 32 bits */
          uint8_t data_size = ByteCounter/4;
          /* Calcula o endereco efetivo da mensagem
           * Cada elemento de RXFIFO0:
           * dados (data_size) + header (2 palavras de 32 bits) */
          RxAddress = (uint32_t *)(SRAMCAN_BASE+(1U)*4 +
                                   (GetIndex * (2U + data_size) * 4U));
      }
      /* Incrementa RxAddress para acessar a seq. palavra do cabecalho */
      RxAddress++;
      /* Incrementa RxAddress para acessar os campos de dados */
      RxAddress++;
      /* 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;
   }
}
```

Para acessar um quadro na fila `RX FIFO0`, utilizamos o índice `GetIndex`, que é atualizado automaticamente pelo *hardware* a cada acesso. O tamanho de cada quadro, expresso em unidades de 32 *bits*, é determinado pelo *payload* configurado em `FDCAN_RXESC_F1DS`, acrescido de duas palavras de 32 *bits* referentes ao cabeçalho de uma mensagem.

7. Estando implementadas as funções básicas de inicialização, transmissão e recepção de mensagens do protocolo CAN, voltamos à função `main` para implementar a tarefa do projeto. Adicione no escopo /* USER CODE BEGIN 2 */

```
FDCAN2_Init();
```

e no escopo /* USER CODE BEGIN 3 */

```
for (int i=0; i<8; i++) {
    TxData[i] = indx++;
}
FDCAN2_AddMessageToTxFifoQ (0x11);
HAL_Delay (1000);
```

A tarefa implementada configura o periférico FDCAN2 no microcontrolador STM32H7A para operar em modo *loopback*, e então, a cada segundo (1000 mili-seg), ele gera uma nova mensagem CAN com um ID fixo (0x11) e um *payload* de dados sequencialmente crescente (no primeiro quadro, os valores dos *bytes* vão de 0 a 7; no segundo quadro, de 8 a 15, e assim em diante.), adicionando-a à fila de transmissão. Dada a configuração em modo *loopback*, essas mensagens transmitidas serão imediatamente recebidas de volta pelo próprio FDCAN2,

permitindo testar a funcionalidade de transmissão e recepção sem a necessidade de um barramento CAN externo.

Essa estrutura é um excelente ponto de partida para testar e depurar a sua configuração do FDCAN2, especialmente no modo *loopback*.

8. Falta ainda implementar a rotina de serviço `FDCAN2_IT0_IRQHandler` para tratar o evento `RF0NE` que corresponde ao evento de recepção de uma nova mensagem na `Rx FIFO 0`. A interrupção deste evento foi habilitada em `FDCAN2_Init`. Abra o arquivo `stm32h7xx_it.c` e insira no escopo /* USER CODE BEGIN PFP */

```
void FDCAN2_GetMessageFromRxFifo0 (void);
```

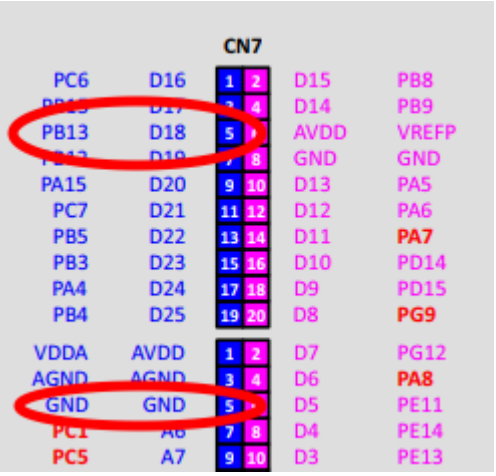
e em /* USER CODE BEGIN 1 */

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

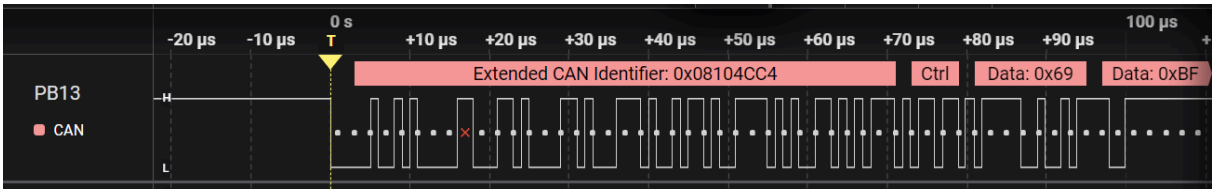
Quando o FDCAN2 recebe uma nova mensagem na *Rx FIFO 0*, ele gera uma interrupção, e esta função `FDCAN2_IT0_IRQHandler` é executada, se a interrupção `FDCAN2_IE_RFONE` e a linha `FDCAN2_IT0_IRQ` estiverem habilitadas. Dentro da ISR, o código verifica se a *flag* `RF0N` (Recepção de Nova Mensagem na FIFO 0) está ativa no registrador de interrupções (`FDCA2_IR`). Se estiver, isso indica que uma mensagem foi de fato recebida, e a função `FDCAN2_GetMessageFromRxFifo0()` é então chamada para processar e ler essa mensagem. Após o processamento, a *flag* `RF0N` é limpa (`FDCAN2->IR |= FDCAN_IR_RF0N_Msk;`), indicando que a interrupção foi tratada e preparando o sistema para futuras novas mensagens.

9. Realize o *build* e transfira o código executável para o STM32H7A no modo *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 modo CAN do analisador (abra mais opções clicando no sinal "+" no campo superior direito, ao lado de "Analyzer"). Configure-o com a *Bit Rate* em 500.000. Note os vários elementos do quadro no sinal capturado e compare com a referência, lembrando que este quadro é de formato padrão.



11. Mude o valor do parâmetro de entrada 0x11 na função `FDCAN2_AddMessageToTxFifoQ(0x11)` 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 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 "ID" do cabeçalho para decidir o que vai fazer com ela.

12. As unidades de transmissão no protocolo CAN são os *quadros de mensagens* estruturado (*frames*), e eles são efetivamente construídos antes da transmissão, enviados quadro por quadro. Com base nesse entendimento, você consegue explicar a demanda por um espaço de memória `SRAM CAN` que deve ser configurado manualmente para complementar a operação do *hardware* de um periférico FDCAN?

13. Comparado aos protocolos RS-232, I2C e SPI, quais **informações adicionais** são inseridas no cabeçalho de uma mensagem CAN? Há alguma vantagem prática para esse *overhead* de dados? Não se preocupe se ainda não tiver a resposta; abordaremos isso mais adiante.

14. O que são os segmentos de tempo em *bits* configuráveis em uma rede CAN, e qual é o papel deles na sincronização entre os nós e na garantia da integridade dos dados? De que

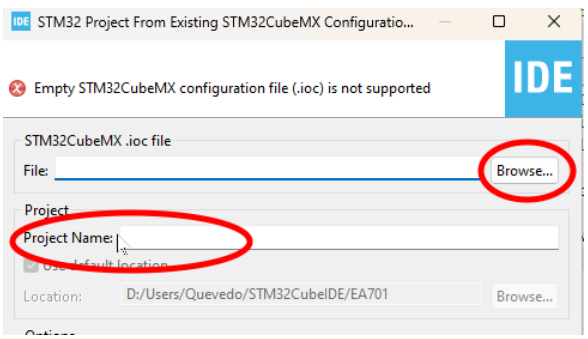
maneira esses segmentos contribuem para esses objetivos? Caso você ainda não saiba a resposta, não se preocupe. Vamos explorar esse conceito adiante.

15. Imagine um cenário automotivo onde sensores de freio e o sistema de infoentretenimento precisam enviar dados simultaneamente pela mesma rede CAN. Sendo uma rede multi-mestre, na qual diversos dispositivos podem iniciar transmissões a qualquer instante, como o protocolo CAN decide qual mensagem "vence" o acesso ao barramento e será transmitida sem interrupções? Por quê uma simples configuração com *pull-ups* e *open-drain, como no I2C,* não seria suficiente para gerenciar essa arbitragem complexa? E qual o papel crucial dos identificadores de mensagem nesse processo?

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. Gere o código e sobrescreva os arquivos main.c e stm32h7xx_it.c com os arquivos do projeto “CAN_Base”. Boa parte do código do projeto anterior será aproveitada.

3. Abra `main.c`. Inclua as seguintes definições de macros no escopo /* USER CODE BEGIN PD */

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

Inclua o seguinte bloco de códigos entre o bloco que contém a chamada de `FDCAN2_AlocaSRamCANSegmentos` e o sétimo bloco 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
}
```

4. 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:

```
	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 transfira o código executável para o microcontrolador no modo *debug*. Veja novamente as variáveis TxData e RxData na aba *Live Expressions*.

7. Vamos mudar a faixa do filtro. Mude o valor da macro `SFID1` para 0x12. Como a faixa agora inicia em 0x12, as mensagens com *bytes* entre 0 e 127 (ID = 0x11) não irão gerar o evento `RFONE`, sendo totalmente ignoradas. Realize o *build* e transfira o código executável para o microcontrolador no modo *debug*. Execute o programa e veja o que acontece.

8. Vamos mudar o tipo de filtro, modificando a definição da macro `TIPO_FILTRO` de 0b00 (`FDCAN_FILTER_RANGE`) para 0b10 (`FDCAN_FILTER_MASK`). Agora o elemento `SFID1` contém o valor base do filtro e o elemento `SFID2` 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 `SFID1` para 0x11 e o elemento `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*, transfira o executável para o microcontrolador no modo *debug* e veja o resultado.

9. Mude apenas o elemento 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.

10. Com base no que você já explorou, em que nível operam os filtros de mensagens? No nível do *hardware* ou de *software* do controlador CAN? Eles realmente impedem a recepção física das mensagens no barramento, ou apenas evitam que essas mensagens, uma vez recebidas, cheguem ao processador? Se você ainda não tem a resposta, não se preocupe: isso será abordado em detalhes no material que você está prestes a ler.

11. O código apresentado no item 3 ilustra a configuração básica da filtragem de mensagens pelo periférico FDCAN. Você consegue identificar a função primária de cada linha de instrução? Se teve dificuldades para decifrá-las, não se preocupe. Voltaremos a essa configuração em detalhes mais adiante.

12. Em um sistema crítico com alta taxa de mensagens, como um barramento automotivo que conecta dezenas de módulos (controle de motor, freios, *airbags*, sistema multimídia, sensores de estacionamento, entre outros), por que a correta configuração dos filtros de *hardware* do protocolo CAN é decisiva para garantir o funcionamento seguro e eficiente do sistema? Ao filtrar mensagens, os filtros reduzem a sobrecarga no barramento, no processador, ou em ambos? Se você ainda não tiver essa resposta, não se preocupe. Vamos explorar em breve essa característica específica do protocolo CAN, que não está presente nos protocolos que vimos anteriormente.

Projeto de comunicação CAN usando 4 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.

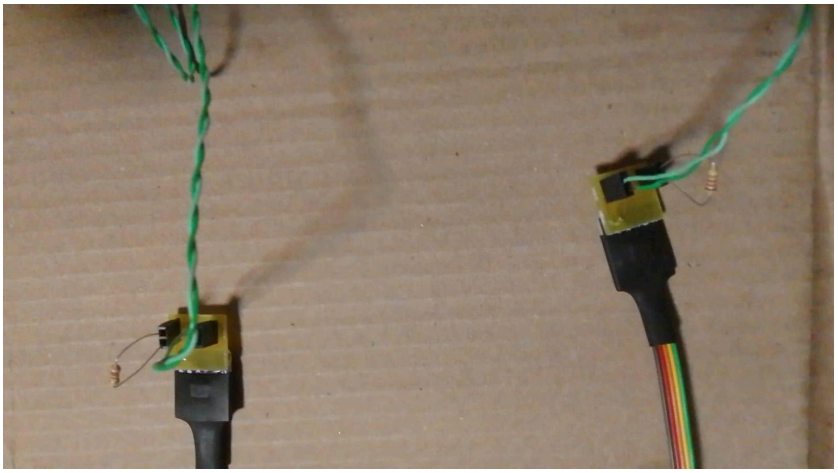
Cada nó irá ler um valor de 16 *bits* em um canal do ADC, no qual estará ligado um potenciômetro ou um eixo do *joystick* a uma taxa de 5 leituras por segundo. O valor lido será guardado em uma variável e transmitido pelo barramento CAN com um identificador próprio. Para o valor de 16 *bits* do ADC, 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 de um controlador CAN (que faz parte do microcontrolador) em níveis lógicos padrão do microcontrolador par 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 de um 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 *transceivers* possuem cabos multi-vias com um conector na ponta, que deve ser ligado no conector H7 (CAN2) da placa de expansão. Este conector disponibiliza os sinais de Tx e Rx do controlador CAN, bem como a alimentação de 5V para o *transceiver*.

2. Na placa de cada *transceiver*, foi adaptado um par de *headers* de 2 pinos, sendo cada *header* ligado aos sinais de CANH e CANL do *transceiver*. 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 *transceiver* (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.



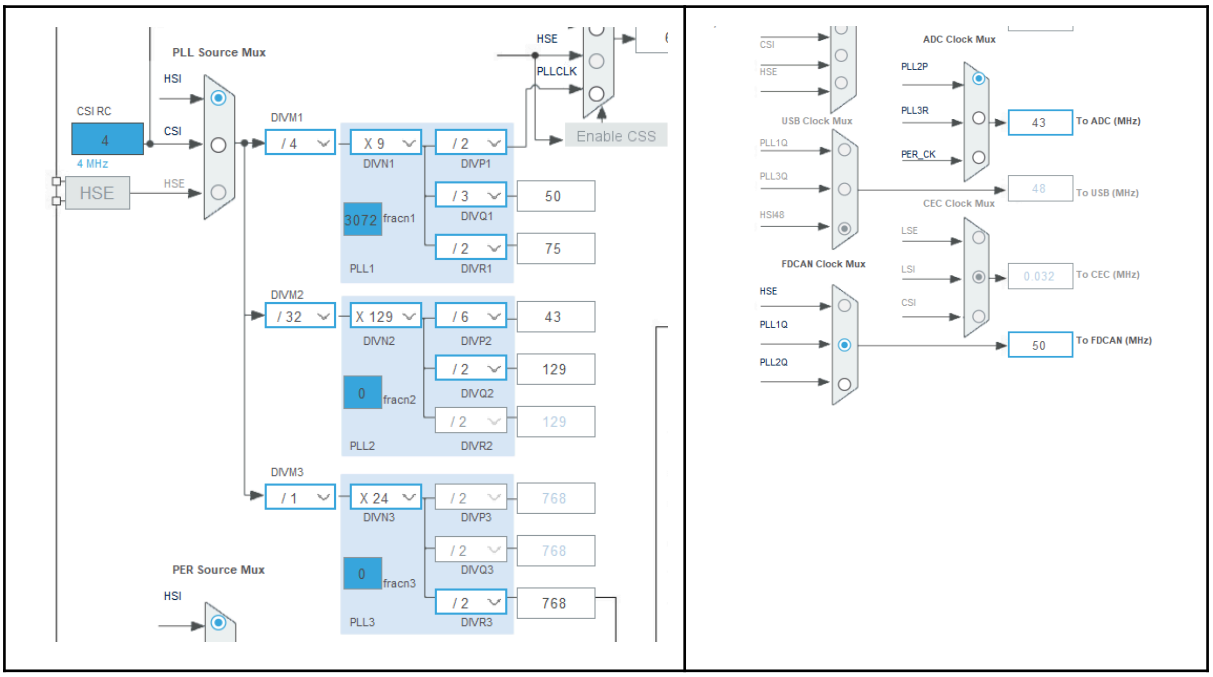
Além das ligações do *transciever*, é necessário ligar um potenciômetro ou um canal do *joystick*. Conecte os extremos do potenciômetro ou os pinos “GND” e “+5V” do *joystick* nos pinos 2 (+3.3V) e 7 (GND) do conector H9 (ADC). O terminal central do potenciômetro ou um dos canais do joystick (Vx ou Vy) 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.

Se optarmos por uma implementação via CMSIS como nos dois projetos anteriores, podemos criar um projeto "CAN_Network_CMSIS" a partir do arquivo CAN_Base.ioc ou CAN_Filtros.ioc, **sem inicializar os periféricos**. Use a opção do menu *File – New – STM32 Project from an Existing STM32CubeMX Configuration File (.ioc)*. O código completo dessa versão do projeto está no arquivo CAN_Network_CMSIS.zip.

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.



A configuração da fonte de *clock* PPL2 para o periférico ADC1 é similar à que fizemos para o FDCAN2, especificamente antes de habilitar os sinais de clock. As seguintes instruções podem ser inseridas na inicialização do ADC1, antes do *clock gating*.

```
{
        //Configurar a frequência de PLL2 em 43MHz
        // Desabilita PLL2
        RCC->CR &= ~RCC_CR_PLL2ON;
        while (RCC->CR & RCC_CR_PLL2RDY); // Aguarda a desabilitação
        // Configura fonte (mantém a existente) e DIVM2
        RCC->PLLCKSELR &= ~RCC_PLLCKSELR_DIVM2;
```

```
	RCC->PLLCKSELR |= (32U << RCC_PLLCKSELR_DIVM2_Pos);
	// Configura os divisores e multiplicadores de PLL2
	RCC->PLL2DIVR =
		((129U - 1U) << RCC_PLL2DIVR_N2_Pos) |  // DIVN2
		((6U   - 1U) << RCC_PLL2DIVR_P2_Pos) |  // DIVP2 → Saída P
		((2U   - 1U) << RCC_PLL2DIVR_Q2_Pos) |  // DIVQ2 → Se quiser
habilitar Q
		((2U   - 1U) << RCC_PLL2DIVR_R2_Pos);   // DIVR2 → Se quiser
habilitar R
	// Configura o fator fracionário (não utilizado → zero)
	RCC->PLL2FRACR = (0U << RCC_PLL2FRACR_FRACN2_Pos);
	// Configura faixa de entrada e faixa de VCO
	// RGE: Faixa de frequência de entrada → Selecionar adequada
	// Suponha faixa 8-16 MHz → RCC_PLLCFGR_PLL2RGE_1 (consultar manual)
	RCC->PLLCFGR &= ~RCC_PLLCFGR_PLL2RGE;
	RCC->PLLCFGR |= RCC_PLLCFGR_PLL2RGE_1;  // Faixa 8-16 MHz (exemplo)
	// VCO: Faixa de frequência do VCO
	// 0 = Medium VCO (150-420 MHz)
	// 1 = Wide VCO (192-836 MHz)
	RCC->PLLCFGR &= ~RCC_PLLCFGR_PLL2VCOSEL; // Medium VCO
	// Habilita os divisores desejados (PLL2P no caso)
	RCC->PLLCFGR |= RCC_PLLCFGR_DIVP2EN;
	RCC->PLLCFGR &= ~(RCC_PLLCFGR_DIVQ2EN | RCC_PLLCFGR_DIVR2EN); //
Desabilita Q e R se não usados
	// Desabilita o modo fracionário (não usado)
	RCC->PLLCFGR &= ~RCC_PLLCFGR_PLL2FRACEN;
	// Habilita PLL2
	RCC->CR |= RCC_CR_PLL2ON;
	while (!(RCC->CR & RCC_CR_PLL2RDY)); // Aguarda PLL2 pronto
}
{
	//Ativar clock gating de ADC e perifericos associados
	// Seleciona ker_ck para FDCAN (pll1_q_ck)
	RCC->PLLCFGR |= RCC_PLLCFGR_DIVQ2EN_Msk;   // habilita pll2_q_ck
	RCC->SRDCCIPR &= ~RCC_SRDCCIPR_ADCSEL_Msk;  // seleciona a fonte para
ADC
	// Habilitar o clock do ADC1
	RCC->AHB1ENR |= RCC_AHB1ENR_ADC12EN;
	//Configurar PC4 como analog
	RCC->AHB4ENR |= RCC_AHB4ENR_GPIOCEN;
	GPIOC->MODER |= GPIO_MODER_MODE4;
	NVIC_SetPriority(ADC_IRQn, 2); // Configura NVIC para interrupcoes do
ADC, prioridade 2
	NVIC_EnableIRQ(ADC_IRQn);
}
```

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. Além disso, o *loopback* foi desabilitado.

```
      // Defina o modo de operacao (Loopback externo)
      FDCAN2->CCCR &= ~(FDCAN_CCCR_TEST |
            FDCAN_CCCR_MON |
            FDCAN_CCCR_ASM);
      FDCAN2->TEST &= ~FDCAN_TEST_LBCK;
      // Operacao normal (alterado)
//    FDCAN2->CCCR |= FDCAN_CCCR_TEST;
//    FDCAN2->TEST |= FDCAN_TEST_LBCK;
```

Basic Parameters

Frame Format	Classic mode
Mode	Normal mode
Auto Retransmission	Enable
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
Message Ram Offset	0
Std Filters Nbr	1
Ext Filters Nbr	0

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.

Rx Fifo0 Elmts Nbr	4
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

Modificamos `FDCAN2_AlocaSRamCANSegmentos` para incluir 4 elementos em vez de 1, visando atender a este quesito. A alteração está detalhada no trecho de código.

```
//MODIFY_REG(FDCAN2->RXF0C, FDCAN_RXF0C_F0S, (1U << FDCAN_RXF0C_F0S_Pos));
MODIFY_REG(FDCAN2->RXF0C, FDCAN_RXF0C_F0S, (4U << FDCAN_RXF0C_F0S_Pos));
/* Endereco inicial de Tx buffer (alterado) */
//Origem += (4U);  // Avança 4 palavras (tamanho fixo de 1 elemento RX
FIFO0)
```

```
Origem += (4U * 4U);  // Avança 16 palavras (tamanho fixo de 4 elementos RX FIFO0)
MODIFY_REG(FDCAN2->TXBC, FDCAN_TXBC_TBSA, (Origem << FDCAN_TXBC_TBSA_Pos));
```

Essa alteração implica ainda que os acessos de escrita na fila TX FIFO precisam ser deslocados para acomodar os 4 quadros de mensagens na fila RX FIFO0. Em `FDCAN2_AddMessageToTxFifoQ`, foi feita a seguinte substituição

```
/* Calcula o endereco do buffer TxFIFO/Queue  */
//RxFIFO passou de 1 elemento para 4 elementos (alterado)
//TxAddress = (uint32_t *)(SRAMCAN_BASE+(1U+4U)*4U +
//(PutIndex*((FDCAN2->TXBC & FDCAN_TXBC_TFQS) >> FDCAN_TXBC_TFQS_Pos)*4U));
TxAddress = (uint32_t *)(SRAMCAN_BASE+(1U+4U*4U)*4U +
(PutIndex*((FDCAN2->TXBC & FDCAN_TXBC_TFQS) >> FDCAN_TXBC_TFQS_Pos)*4U));
```

Os demais parâmetros permanecem inalterados, incluindo a interrupção do FDCAN2, que continua habilitada com prioridade 1.

Em terceiro lugar, a distinção das mensagens geradas pelos 4 nós é essencial. Nos dois projetos-exemplo anteriores, a configuração *loopback* implicava que o canal TX estava conectado internamente ao canal RX. Isso resultava no empilhamento exclusivo de quadros de mensagem com o mesmo identificador (ID) na fila RX FIFO0. No entanto, este projeto, ao estar aberto para receber mensagens de uma rede com 4 nós, permite que a fila RX FIFO0 de cada nó empilhe mensagens com IDs variados. Para processar essas mensagens distintamente, é necessário extrair o ID do campo de cabeçalho de cada mensagem. Isso levou a uma modificação mais radical nos códigos:

1. Para garantir o correto armazenamento dos dados enviados pelos 4 nós, a ISR `FDCAN2_IT0_IRQHandler` precisa recuperar o ID e o *payload* de cada nova mensagem recebida. Dessa forma, o processador pode alocar os dados recuperados na posição apropriada do vetor `valor`, que dedica uma posição distinta para cada nó. Assim, foi feita a seguinte alteração em `FDCAN2_IT0_IRQHandler`

   ```
   //Armazenar o payload na posição correpondente do valor (alterado)
   //FDCAN2_GetMessageFromRxFifo0 ();
   msg_id = FDCAN2_GetMessageFromRxFifo0 ();
   PutValueInVector(msg_id);
   ```

   A definição da função `PutValueInVector` é adicionada no escopo /* USER CODE BEGIN 0 */ do arquivo `main.c`

   ```
   void PutValueInVector (uint32_t msg_ID) {
   uint8_t p;
   p = ((uint8_t)msg_ID) & 0x03;
   valor[p] = (RxData[0] * 256) + RxData[1];
   }
   ```

2. Para recuperar o ID da mensagem recebida, a função `FDCAN2_GetMessageFromRxFifo0` necessita de uma atualização. Visando minimizar o impacto no restante do código, decidimos que o ID seja retornado por

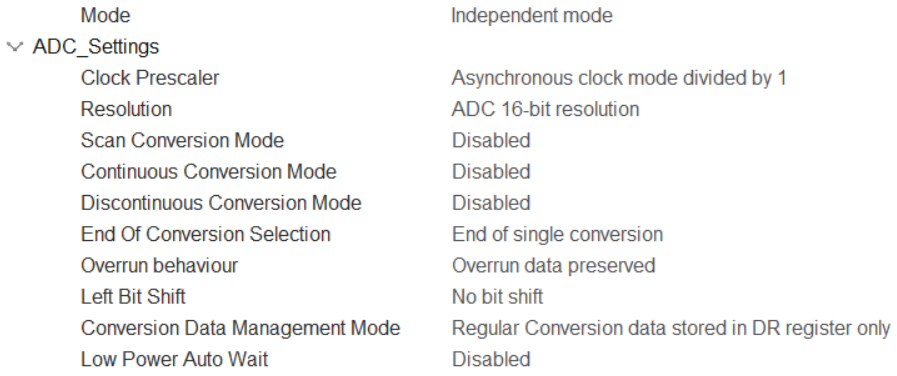
essa função e, em seguida, passado para `PutValueInVector`. É em `PutValueInVector` que o *payload*, já extraído para `RxData`, é armazenado na posição do vetor `valor` que corresponde ao ID.

```
//Retorna o ID da mensagem (alterado)
//void FDCAN2_GetMessageFromRxFifo0 (void) {
uint32_t FDCAN2_GetMessageFromRxFifo0 (void) {
  uint32_t GetIndex = 0;
  uint32_t *RxAddress;
  uint32_t ByteCounter;
  uint32_t IdType, Identifier = 0;
  //Obter o indice (posicao) da mensagem na RX FIFO0
  //e correspondente endereco na SRAM CAN (alterado)
  //    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));
  //          }
  //    }
  /* Rx FIFO 0 está vazio */
  if ((FDCAN2->RXF0S & FDCAN_RXF0S_F0FL) == 0U)
  {
    /* RX FIFO esta vazio */
    return 0;
  }
  else
  {
    //Bytes a serem lidos
    ByteCounter = Code2ByteCounter
        ((FDCAN2->RXESC & FDCAN_RXESC_F1DS) >> FDCAN_RXESC_F1DS_Pos);
    /* Verifica se Rx FIFO 0 esta cheio e se o modo de sobreescreve
esta ativado */
    if (((FDCAN2->RXF0S & FDCAN_RXF0S_F0F) >> FDCAN_RXF0S_F0F_Pos) ==
1U)
    {
        if (((FDCAN2->RXF0C & FDCAN_RXF0C_F0OM) >>
FDCAN_RXF0C_F0OM_Pos) == ((uint32_t)0x00000001U))
        {
        /* When overwrite status is on discard first message in FIFO */
              GetIndex = 1U;
```

```
            }
        }
        /* Calcula o indice do elemento no Rx FIFO 0  */
        GetIndex += ((FDCAN2->RXF0S & FDCAN_RXF0S_F0GI) >>
FDCAN_RXF0S_F0GI_Pos);
        /* Obtem o tamanho de dados de RX FIFO0 na unidade de 32 bits */
        uint8_t data_size = ByteCounter/4;
        /* Calcula o endereco efetivo da mensagem
         * Cada elemento de RXFIFO0:
         * dados (data_size) + header (2 palavras de 32 bits) */
        RxAddress = (uint32_t *)(SRAMCAN_BASE+(1U)*4 +
                        (GetIndex * (2U + data_size) * 4U));
      }
      {
        /* Recuperar o identificador (alterado)*/
        IdType = *RxAddress & ((uint32_t)0x40000000U);
        /* Recuperar o identificador */
        if (IdType == ((uint32_t)0x00000000U)) /* Standard ID element */
        {
            Identifier = ((*RxAddress & ((uint32_t)0x1FFC0000U)) >> 18U);
        }
        else /* Extended ID element */
        {
            Identifier = (*RxAddress & ((uint32_t)0x1FFFFFFFU));
        }
      }
      /* Incrementa RxAddress para acessar a segunda palavra do cabecalho
*/
      RxAddress++;
      /* Incrementa RxAddress para acessar os campos de dados */
      RxAddress++;
      /* 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;
      return Identifier;
}
```

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

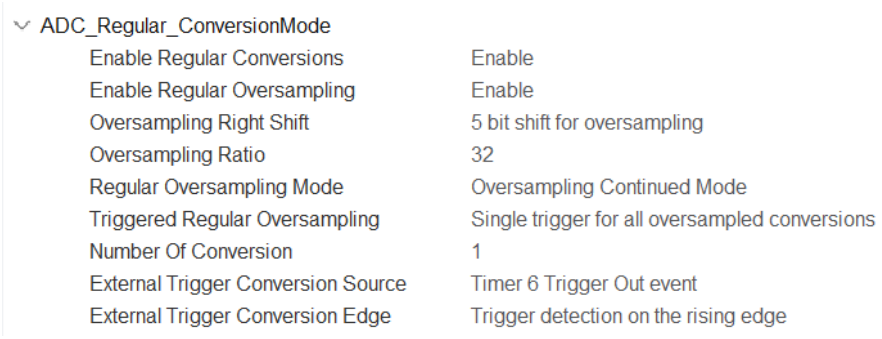


Mode	Independent mode
ADC_Settings	
Clock Prescaler	Asynchronous clock mode divided by 1
Resolution	ADC 16-bit resolution
Scan Conversion Mode	Disabled
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled
End Of Conversion Selection	End of single conversion
Overrun behaviour	Overrun data preserved
Left Bit Shift	No bit shift
Conversion Data Management Mode	Regular Conversion data stored in DR register only
Low Power Auto Wait	Disabled

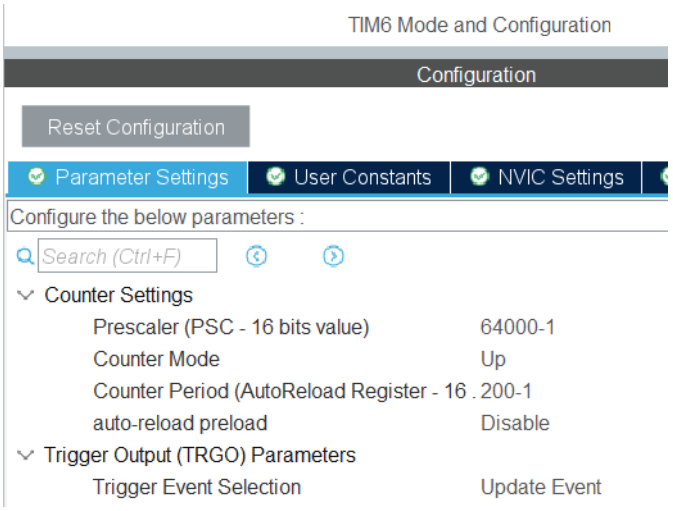
prioridade 2.

A configuração direta das funções do periférico ADC1 via registradores segue o mesmo procedimento que temos utilizado.

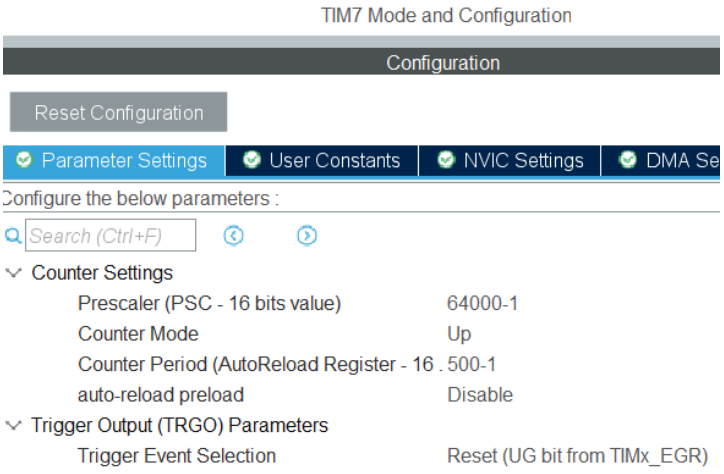
7. O TIM6 foi ajustado com *prescaler* de 64000 e *AutoReload* em 200, gerando assim uma frequência de repetição de *updates* de 5Hz. O parâmetro *Trigger Event Selection* foi modificado para *Update Event*. Assim, o ADC é disparado 5 vezes por segundo via eventos de interconexões internas dos módulos.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



ADC_Regular_ConversionMode	
Enable Regular Conversions	Enable
Enable Regular Oversampling	Enable
Oversampling Right Shift	5 bit shift for oversampling
Oversampling Ratio	32
Regular Oversampling Mode	Oversampling Continued Mode
Triggered Regular Oversampling	Single trigger for all oversampled conversions
Number Of Conversion	1
External Trigger Conversion Source	Timer 6 Trigger Out event
External Trigger Conversion Edge	Trigger detection on the rising edge



8. O TIM7 foi ajustado com *prescaler* de 64000 e *AutoReload* em 500, gerando assim uma frequência de repetição 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. **O código completo está disponível no Moodle.**

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*[1] de conversão de ADC concluída, equivalente à ISR `ADC_IRQHandler`, coloca o *flag* ***adcnovo*** em 1. A função *callback* de *update* de TIM7, equivalente à ISR `TIM7_IRQHandler`, coloca a *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, equivalente à rotina de serviço `FDCAN2_IT0_IRQHandler`, 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.

Para demonstrar que o nó transmissor de uma mensagem não a recebe de volta em uma operação normal, realizamos uma pequena modificação no fluxo principal. Além das adaptações habituais do ambiente HAL (em inglês, *Hardware Abstraction Layer*) e CMSIS (em inglês, *Cortex Microcontroller Software Interface Standard*), optamos por armazenar o

[1] A ISR atua como um "gatilho" rápido para a ocorrência de um evento, e o *callback* implementado no nível de abstração HAL é a função que "reage" a esse gatilho, processando as informações e realizando as ações necessárias em um ambiente mais flexível e seguro, evitando que a ISR fique muito longa e bloqueie outras interrupções ou o fluxo principal do programa.

valor lido em uma variável local, em vez de guardá-lo na posição do vetor "`valor`" de valores correspondente ao número do nó.

O código foi inserido no escopo /* USER CODE BEGIN 3 */

```
if(leADCFlag()) {
	Stop_Conv();
	//	valor[ID] = HAL_ADC_GetValue(&hadc1);
	uint16_t data = leADCAmostra();
	Start_Conv();
	//	TxData[0] = data / 256;
	//	TxData[1] = data % 256;
	TxData[0] = data >> 8; // computo alternativo
	TxData[1] = data & 0x00FF;
	FDCAN2_AddMessageToTxFifoQ (IDBASE + ID);
}
```

Para garantir a compatibilidade com o formato de mensagem do protocolo CAN, um dado de 16 *bits* (`data`) foi dividido em dois *bytes* (`TxData[0]` e `TxData[1]`). Avaliamos duas abordagens para essa operação: uma baseada em divisões e outra em deslocamentos de *bits*. Priorizamos a alternativa de deslocamentos de *bits* por ser comprovadamente mais eficiente.

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 montar as mensagens referentes aos 4 nós. O seguinte bloco de instruções foi inserido após o bloco apresentado no item 14.

```
if (leDisplayFlag()) {
	resetaDisplayFlag();
	clearDisplay(WHITE);
	for( i = 0; i < 4U; i++) { // Corre pelos 4 valores e os imprime no
display
		BuildString(i, valor[i], buf);
		setStr((char *)buf, 0, i * 8, BLACK);
	}
	updateDisplay();
}
```

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.**

18. Quando você instala a rede CAN com o projeto CAN_Network_CMSIS, qual valor é lido na posição do vetor "`valor`" correspondente ao nó com o *display* Nokia? E que valor você lê

ao instalar o projeto CAN_Network? Se houver alguma dúvida sobre o que está acontecendo, a resposta pode ser encontrada no item 14.

19. No código, identifique a instrução responsável por gravar uma mensagem no *buffer* de transmissão de SRAM CAN e aquela que lê uma nova mensagem do *buffer* de recepção de SRAM CAN. Qual instrução, especificamente, envia a mensagem pela rede e qual incrementa o índice de elementos no *buffer* de RX FIFO quando se recebe um novo quadro de mensagem? Para auxiliar sua compreensão do fluxo de controle, execute o código passo a passo dentro das funções `FDCAN2_AddMessageToTxFifoQ` e `FDCAN2_GetMessageFromRxFifo0`.

Projeto de comunicação CAN usando múltiplos nós parametrizáveis

A quantidade máxima de nós que podem ser conectados em uma depende de vários fatores, mas a limitação teórica mais comum é de 112 nós. Essa limitação é baseada principalmente nas características elétricas do barramento, como impedância de linha, capacitância dos *transceivers*, capacitância do cabo e capacitância de corrente dos *drivers* CAN. No entanto, na prática, esse número é geralmente menor (por volta de 30 a 50 nós), dependendo dos *transceivers* usados (alguns permitem maior carga no barramento), comprimento total do barramento (quanto mais longo, menos nós suportados), taxa de transmissão (velocidades mais altas requerem menos nós e cabos mais curtos), e qualidade de cabos e terminações.

Este projeto representa uma evolução da implementação do projeto-exemplo CAN_Network_CMSIS.zip para quatro nós. O objetivo principal é desenvolver um **código parametrizável** onde o **tamanho da rede, ou o número de nós, é definido por meio de uma macro**. Consequentemente, todos os elementos do código que dependem dessa quantidade de nós se ajustarão automaticamente, proporcionando maior flexibilidade e escalabilidade ao sistema. A capacidade de definir o número de nós por uma macro elimina a necessidade de modificações manuais extensas no código ao adicionar ou remover nós, reduzindo erros e acelerando o desenvolvimento. Isso é essencial para aplicações industriais e automotivas, onde a configuração da rede pode variar consideravelmente.

Para que esta solução parametrizável seja totalmente eficaz, dois desafios importantes precisam ser endereçados:

1. Ajuste de macros dependentes do tamanho da rede: Deve-se desenvolver uma metodologia para ajustar automaticamente outras macros que são intrinsecamente ligadas ao número de nós da rede. Um exemplo crítico é a definição do campo de ID (Identificador) das mensagens CAN. A quantidade de *bits* necessária para o ID é diretamente proporcional ao número de nós na rede (por exemplo, para N nós, precisamos de no mínimo $\log_2 N$ *bits* para IDs únicos). O projeto deve implementar uma lógica que calcule dinamicamente o tamanho do campo de ID com base na macro do número de nós, garantindo que cada nó possua um ID único e que a comunicação

seja eficiente.

2. Acomodação de IDs em *display* Nokia 5110: Apresentar o estado de todos os 32 nós em um *display* Nokia 5110, que possui uma resolução limitada de 84x48 *pixels*, é um desafio de interface do usuário. Utilizando o tamanho de caractere padrão de 5x8 *pixels* (comum em bibliotecas para este *display*), exibir 32 IDs de forma legível exige um arranjo otimizado. Precisamos avaliar e escolher a melhor estratégia entre opções populares, como compressão de IDs, rolagem de tela e organização espacial dos IDs.

A resolução desses pontos garantirá que o projeto não apenas ofereça uma rede CAN flexível e escalável em termos de código, mas também uma interface de usuário prática e informativa para monitoramento, mesmo em displays com recursos limitados.

1. Vamos criar o projeto `CAN_NetworkN_CMSIS` utilizando o arquivo de configuração STM32CubeMX `CAN_Network_CMSIS.ioc`. A versão comprimida deste projeto é CAN_NetworkN_CMSIS.zip.

2. Substitua os arquivos `main.c` e `stm32h7xx_it.c` na pasta `Core/Src` pelos seus equivalentes do projeto `CAN_Network_CMSIS`.

3. Vamos agora substituir a quantidade fixa de nós, 4U, pela macro `RXFIFO_ELMTS_NBR` no arquivo `main.c`. Para isso, inclua a seguinte definição no escopo `/* USER CODE BEGIN PM */`.

```
#define RXFIFO_ELMTS_NBR 15
```

e modifique as seguintes linhas no arquivo `main.c`

```
/* Endereco inicial de Tx buffer (alterado) */
//    Origem += (4U);  // Avança 4 palavras (tamanho fixo de 1 elemento RX
FIFO0)
Origem += (4U * RXFIFO_ELMTS_NBR);  // Avança 16 palavras (tamanho fixo de
4 elementos RX FIFO0)

//A quantidade total da area reservada aumentou (alterado)
/* Zera o conteudo da area reservada (soma das palavras dos filtros, RX
FIFO e TX FIFO)*/
/*    for (uint32_t RAMcounter = SRAMCAN_BASE; RAMcounter <
SRAMCAN_BASE+(1U+4U+4U)*4U; RAMcounter += 4U)  */
for (uint32_t RAMcounter = SRAMCAN_BASE; RAMcounter <
SRAMCAN_BASE+(1U+(4U*RXFIFO_ELMTS_NBR)+4U)*4U; RAMcounter += 4U)
    {
       *(uint32_t *)(RAMcounter) = 0x00000000;
    }

 /* Calcula o endereco do buffer TxFIFO/Queue  */
 //RxFIFO passou de 1 elemento para 4 elementos (alterado)
 // TxAddress = (uint32_t *)(SRAMCAN_BASE+(1U+4U)*4U +
 // (PutIndex*((FDCAN2->TXBC & FDCAN_TXBC_TFQS) >>
FDCAN_TXBC_TFQS_Pos)*4U));
```

```
 TxAddress = (uint32_t *)(SRAMCAN_BASE+(1U+4U*RXFIFO_ELMTS_NBR)*4U +
   (PutIndex*((FDCAN2->TXBC & FDCAN_TXBC_TFQS) >>
FDCAN_TXBC_TFQS_Pos)*4U));

//uint16_t valor[4];
//uint16_t valor[RXFIFO_ELMTS_NBR];
uint8_t valor[RXFIFO_ELMTS_NBR];
```

A segunda modificação da declaração do vetor `valor` será explicada logo adiante.

4. Dando continuidade, faremos os ajustes das macros que dependem do tamanho da rede. Para isso, o primeiro passo é transformá-las em variáveis, possibilitando que seus conteúdos sejam definidos dinamicamente durante a execução. Abaixo, apresentamos a nova versão das macros e variáveis.

```
/* USER CODE BEGIN PM */
#define ID 8
#define CONTRASTE 55
#define RXFIFO_ELMTS_NBR 15
#define IDBASE 0x0A0 // Base para os identificadores
#define SFID1 IDBASE
//As seguintes macros terao valores variados
//conforme RXFIFO_ELMTS_NBR que nao pode passar de 31 a qtde de nos
//#define IDMASK 0x7FC // Mascara desconsiderando os 2 bits menos
significativos
//#define SFID2 IDMASK
#define TIPO_FILTRO 0b10  // 0b00: Intervalo; 0b10: Mascaramento

/* USER CODE END PM */

/* USER CODE BEGIN PV */
uint8_t           TxData[8];
uint8_t           RxData[8];
//uint16_t valor[4];
//uint16_t valor[RXFIFO_ELMTS_NBR];
uint8_t valor[RXFIFO_ELMTS_NBR];
uint16_t IDMASK;
uint8_t MSGIDMASK;
uint16_t SFID2;
int indx = 0;
/* USER CODE END PV */
```

e implementamos a função `setarMascaras` para ajustar essas novas variáveis dinamicamente, utilizando o número de nós na rede (`N = RXFIFO_ELMTS_NBR`) como base.

```
void setarMascaras() {
    uint8_t i, N;
    N = RXFIFO_ELMTS_NBR;
    i = 0;
    while (!(N & 0x80)) {
       i++;
       N = N << 1;
    }
    N = 8 - i;
    MSGIDMASK = (pow(2,N)-1);
```

```
    IDMASK = 0x7FF & ~MSGIDMASK;
    SFID2 = IDMASK;
}
```

Em seguida substituímos a única máscara numérica na função `PutValueInVector` pela variável `MSGIDMASK`

```
//    p = ((uint8_t)msg_ID) & 0x03;
    p = ((uint8_t)msg_ID) & MSGIDMASK;
```

5. Para completar, vamos otimizar a disposição dos valores de cada nó no *display* Nokia. Buscando preservar ao máximo a estrutura existente, reduzimos a resolução do conversor ADC de 16 para 8 *bits*, alterando a resolução de amostragem do ADC1 na função `ADC1_Init`

```
// Configurar a resolucao (8 bits)
//    ADC1->CFGR &= ~ADC_CFGR_RES;
    ADC1->CFGR |= ADC_CFGR_RES;
```

e a declaração da variável `adc1` no arquivo `stm32h7xx_it.c`

```
//uint16_t adc1;
uint8_t adc1;
```

Além disso, vamos remover o prefixo “IDx:” de cada nó na função *BuildString*

```
void BuildString(uint8_t id, uint16_t val, uint8_t *b) {
    char buffer[6]; // uint16_t tem no máximo 5 dígitos
    uint8_t i = 0;
    uint8_t j;

    // Inicio da string
//    *b = 'I';
//    *(b + 1) = 'D';
//    *(b + 2) = 0x30 + id;
//    *(b + 3) = ':';
//    *(b + 4) = ' ';

    // Conversao do valor em string a partir da posicao 5 do buffer
    // Caso especial do zero
    if(val == 0) {
//      *(b + 5) = '0';
//      *(b + 6) = '\0';
       *(b) = '0';
       *(b + 1) = '\0';
       return;
    }

    // Extrai cada dígito, do menos significativo ao mais significativo
    while (val > 0) {
       buffer[i++] = (val % 10) + '0';
       val /= 10;
    }

    // Inverte a ordem dos dígitos para a string final
    for (j = 0; j < i; j++) {
```

```
//      *(b + 5 + j) = buffer[i - j - 1];
        *(b + j) = buffer[i - j - 1];
    }
    // Adiciona o terminador nulo no final da string
//    *(b + 5 + i) = '\0';
    *(b + i) = '\0';
}
```

Com essa otimização, reduzimos a necessidade de 9 caracteres por nó (que ocupariam 54 *pixels* na linha) para apenas 3 caracteres (18 *pixels*). Para garantir a legibilidade, reservaremos 4 caracteres por nó, totalizando 24 *pixels* por entrada. Considerando um display de 84 *pixels* de largura por 48 *pixels* de altura, podemos organizar os dados em uma matriz de 3.5 colunas (84/24) por 6 linhas (48/8). Para transformar o índice `i` de um vetor (unidimensional) para as coordenadas `(j, k)` dessa matriz (bidimensional), utilizaremos o seguinte algoritmo:

- linha (j): j = i % 6;
- coluna (k): k = i / 6;

Em termos de códigos, foram feitas as seguintes alterações na função `main`:

```
// Quantidade paramentrizada (alterado)
//           for( i = 0; i < 4; i++) { // Corre pelos RXFIFO_ELMTS_NBR
valores e os
for( i = 0; i < RXFIFO_ELMTS_NBR; i++) { // Corre pelos RXFIFO_ELMTS_NBR
valores e os imprime no display
      BuildString(i, valor[i], buf);
//                              Nova disposicao dos elementos de valor
(alterado)
//                 setStr((char *)buf, 0, i * 8, BLACK);
      row = i%6;
      col = i/6;
      setStr((char *)buf, col * 24, row * 8, BLACK);
}
```

6. Configure a rede CAN com `N` nós, utilizando potenciômetros como atuadores, de forma semelhante ao projeto-exemplo `CAN_Network_CMSIS`. Em seguida, instale o programa em cada um desses nós. Para cada nó, configure a macro `RXFIFO_ELMTS_NBR` com o valor de `N` e atribua um `ID` diferente, variando de `0` a `N`. Após essa configuração, compile e execute o programa em todos os nós. Com o programa em execução, observe o *display* de cada nó e interaja com os potenciômetros ou *joysticks* de cada placa para visualizar as mudanças nos valores.

FUNDAMENTOS TEÓRICOS

No cenário atual de automação e controle, as **redes industriais** são a espinha dorsal de qualquer processo produtivo, garantindo que máquinas, sensores e atuadores se comuniquem de forma eficiente e confiável. Compreender os fundamentos teóricos por trás dessas redes é essencial para projetar, implementar e manter sistemas robustos e seguros.

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.

Abordaremos os pilares teóricos que sustentam a operação das redes industriais, desde a sua estrutura física até os sofisticados mecanismos que garantem a integridade e a confiabilidade dos dados em ambientes hostis.

TOPOLOGIAS DE REDE

A **topologia de rede** descreve a estrutura e o arranjo dos componentes de uma rede, tanto em sua disposição física quanto na lógica de comunicação entre eles. Uma rede é essencialmente uma coleção de nós, como roteadores, switches, computadores, e *links*, ou seja as conexões entre eles. A topologia, portanto, detalha como esses elementos se conectam e como os dados fluem por essa estrutura.

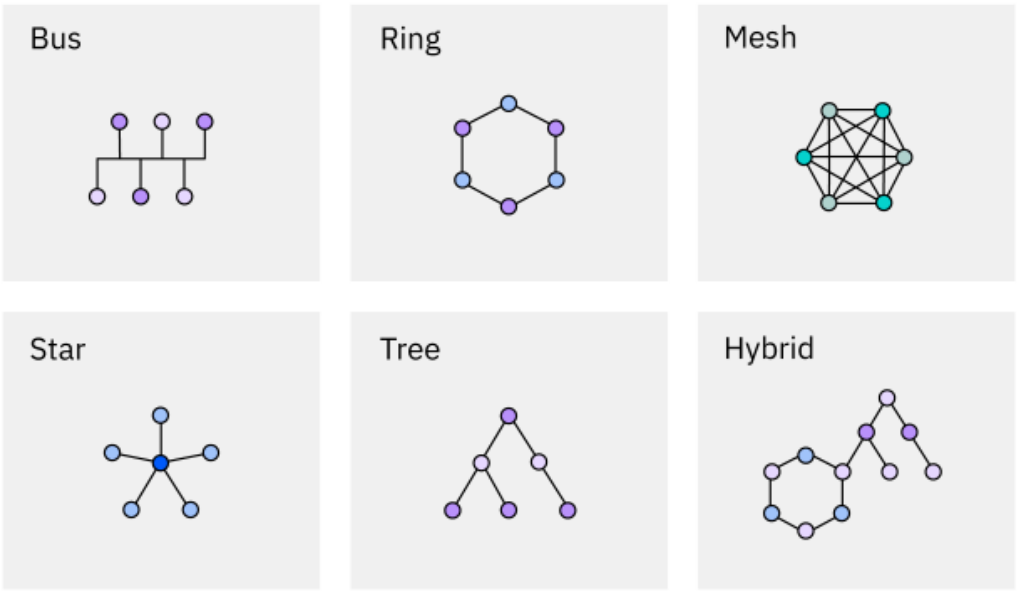
A topologia de rede abrange três aspectos cruciais:

- **Topologia física:** Refere-se à **disposição real** dos componentes da rede e à forma como estão conectados por cabos ou links sem fio. Um mapa de topologia física é uma ferramenta visual indispensável para administradores, ajudando a organizar dispositivos e conexões para uma gestão eficiente.
- **Topologia lógica:** Descreve **como os dados percorrem a rede**, independentemente da conexão física. Ela ilustra o caminho que os dados seguem e o número de nós pelos quais passam para chegar ao destino. É importante notar que a topologia física e a lógica nem sempre são idênticas. Por exemplo, uma rede fisicamente organizada em barramento pode operar logicamente como uma estrela, dependendo do protocolo de comunicação utilizado.
- **Topologia de controle**: Diz respeito à **organização e ao gerenciamento do fluxo de comunicação e do controle** dentro da rede. Esse aspecto é particularmente relevante em **sistemas embarcados**, onde a previsibilidade, robustez e tempo de resposta são essenciais. A topologia de controle define como os dispositivos coordenam suas ações, como os dados são priorizados e como a rede reage a eventos e falhas, garantindo a performance e a segurança necessárias para aplicações críticas.

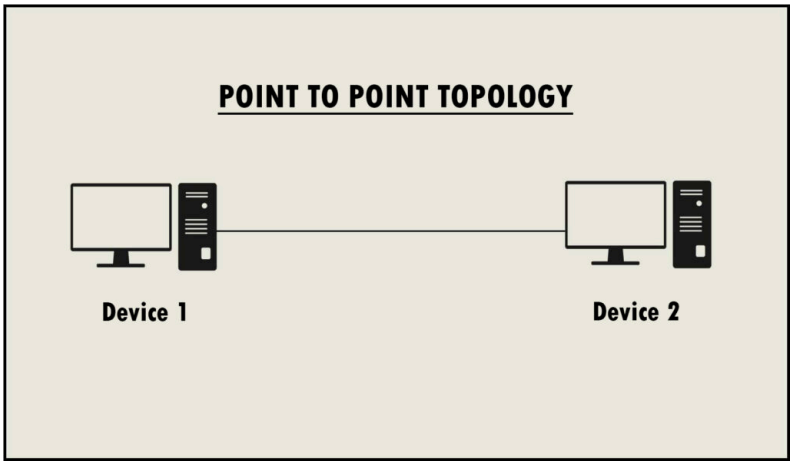
A escolha da topologia de rede tem um impacto significativo em vários aspectos da funcionalidade da rede, incluindo **velocidades de transferência de dados** (como a informação se move), eficiência (capacidade de gerenciar o tráfego de forma otimizada) e

segurança da rede (resiliência contra falhas e ataques). Existem diversos tipos de topologias de rede, cada uma com suas vantagens e desvantagens específicas. Ao planejar ou expandir uma rede, é fundamental considerar as características de cada rede para selecionar a topologia mais adequada às necessidades e aos objetivos desejados.

A seguir, são descritos os principais **tipos de topologia**, tanto física quanto lógica, frequentemente encontrados em sistemas embarcados e redes de dados.

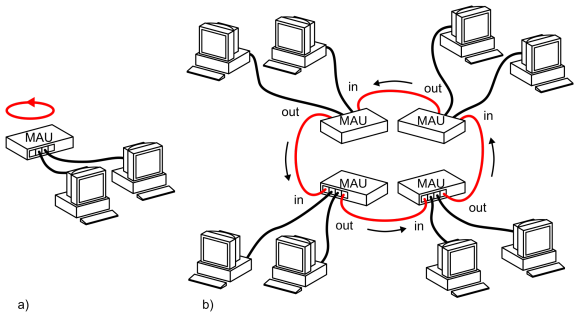


- **Ponto-a-ponto:** A topologia ponto-a-ponto representa a conexão mais direta e fundamental em redes, ligando dois dispositivos exclusivamente por um único *link* de comunicação. Essa simplicidade se traduz em instalação fácil, custo reduzido para pares de nós, alta velocidade e baixa latência, já que a comunicação é dedicada e livre de disputas. Adicionalmente, oferece maior segurança intrínseca e facilidade na resolução de problemas. No entanto, sua principal limitação é a escalabilidade extremamente restrita, tornando-a impraticável para redes maiores, pois a necessidade de múltiplos *links* diretos eleva exponencialmente o custo e a complexidade, além de apresentar um ponto único de falha no *link* e ser ineficiente para comunicação em *broadcast* (transmissão a todos os nós simultaneamente). Protocolo intrinsecamente ponto-a-ponto é a UART.

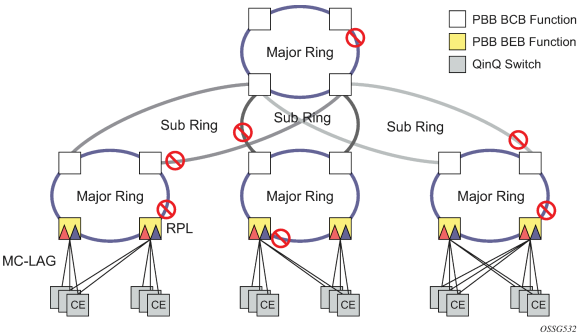


Fonte: ITRelease

- **Barramento:** Na topologia de barramento, ou **topologia multiponto**, 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. As mensagens são transmitidas no barramento e todos os nós as recebem. Cada nó verifica se a mensagem lhe pertence. 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. Exemplos de redes de topologia de barramento são barramentos CAN (controlador de área) usados na indústria automotiva e RS-485 multiponto e a rede de sensores I2C em que todos os dispositivos compartilham as duas linhas SCL e SDA.
- **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. Exemplos de redes que adotam esta topologia são rede *token ring* introduzida pela IBM em 1984 e redes industriais baseadas em anel *Ethernet* (com redundância).



Fonte: Wikipedia

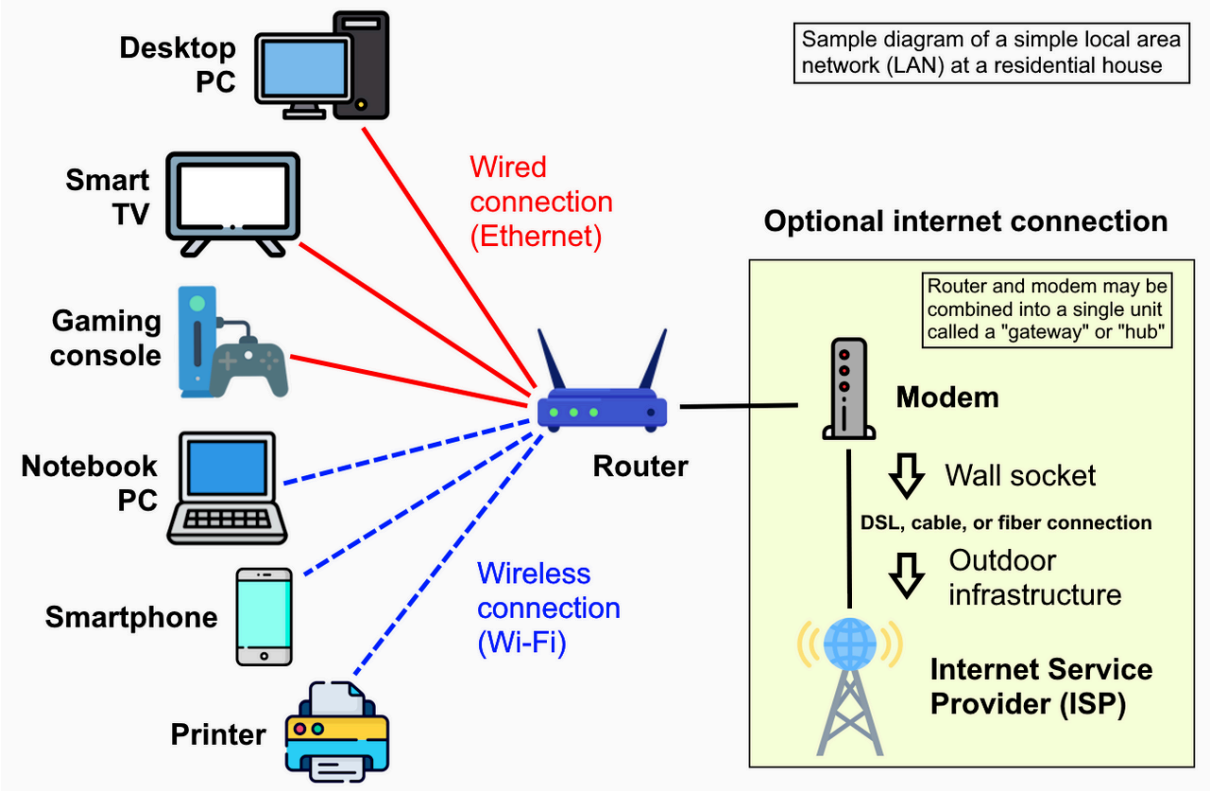


Fonte: Nokia

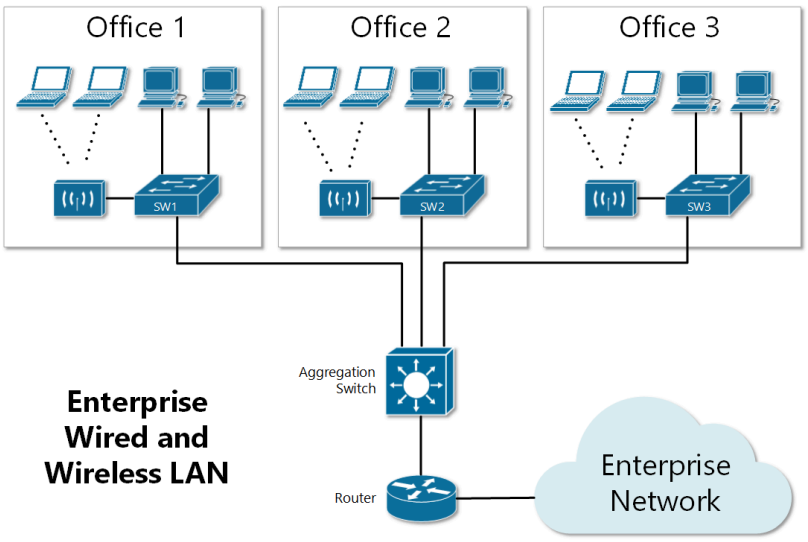
- **Estrela**: Na topologia em estrela, todos os nós estão conectados a um nó central (*hub*, *switch* ou controlador), que amplifica e retransmite os sinais para os dispositivos. Os nós estão posicionados ao redor desse nó central, formando uma estrutura que se assemelha a uma estrela. O nó central gerencia a comunicação. Todos os dados passam por ele. Essa estrutura centralizada facilita a adição ou remoção de dispositivos, contribuindo para sua escalabilidade. No entanto, o desempenho de toda a rede depende do nó central e das conexões com ele. Se o nó central falhar, toda a rede ficará inoperante. Porém, se um único nó falhar, o restante da rede não é afetado,

desde que o nó 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*).

No caso da rede SPI, há um mestre e múltiplos escravos. O mestre tem uma linha de clock (SCLK), uma linha de saída de dados (MOSI) e uma linha de entrada de dados (MISO), que são compartilhadas entre todos os escravos, definindo uma topologia física de barramento. No entanto, a linha \CS (*Chip Select* ou *Slave Select*) é dedicada para cada escravo, estabelecendo uma topologia física em estrela. No entanto, embora as linhas de dados sejam compartilhadas fisicamente, a linha \CS garante que o mestre se comunica com apenas um escravo por vez de forma lógica. Quando o \CS de um escravo está ativo, ele e o mestre formam uma conexão lógica ponto-a-ponto.

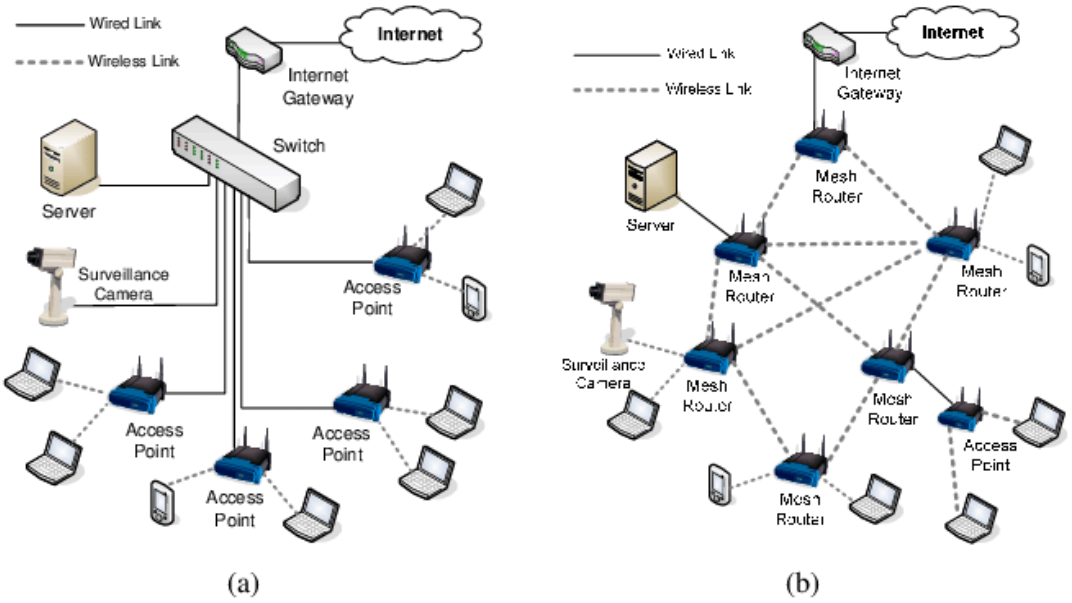


- **Árvore**: A topologia em árvore combina elementos das redes em barramento e em estrela, criando uma estrutura hierárquica. Nessa configuração, um nó central serve como o nó raiz, conectando-se a várias redes em estrela em vez de nós individuais. Essa arquitetura permite um maior número de dispositivos 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 nó 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 nó-raíz central pode interromper toda a rede. Ethernet corporativa em edifícios e algumas arquiteturas de sistemas embarcados modulares são exemplos de rede de topologis em árvore.

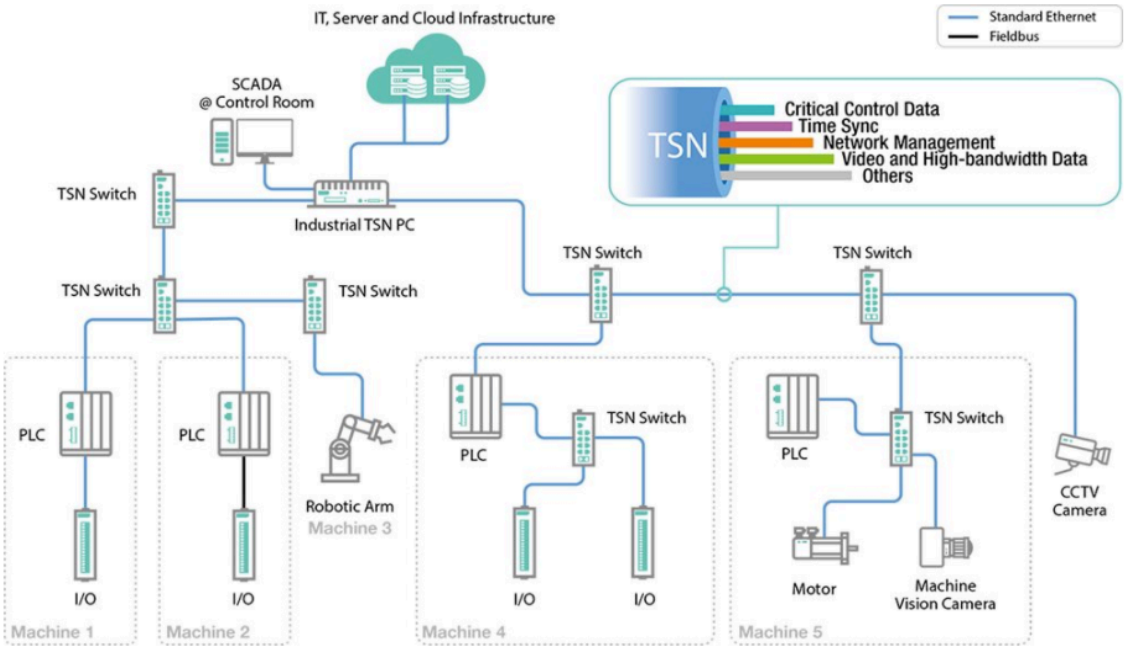


Fonte: Network Academy

- **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, como redes *mesh* sem fio em IoT e automação residencial e redes industriais com protocolo TSN (do inglês *Time-Sensitive Networking*), devido à sua capacidade de garantir um funcionamento eficaz e resiliente. A figura à esquerda ilustra a diferença entre uma topologia WLAN tradicional e uma rede *mesh* sem fio.



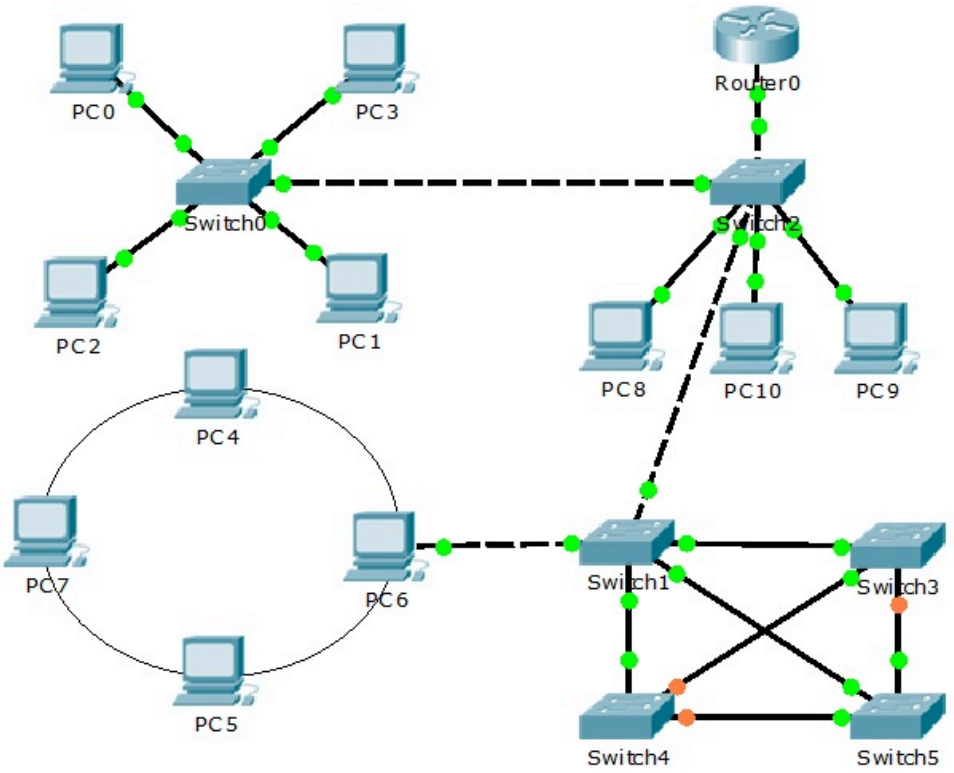
Fonte: Research Gate



Fonte: SPHINX

- **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. Exemplos de redes híbridas são redes industriais modernas que combinam Ethernet, CAN e redes sem fio e sistemas de automação predial com segmentos RS-485 (barramento) e *gateways* Ethernet (estrela ou árvore).



Fonte: hiTechMV

MODELOS DE COMUNICAÇÃO

Um **modelo de comunicação** descreve de forma estruturada como a informação é transmitida entre emissores e receptores. Ele organiza essa comunicação em camadas, onde cada camada é responsável por um conjunto específico de funções, garantindo que os dados possam viajar de forma eficiente e confiável entre diferentes dispositivos. O modelo inclui desde a descrição dos sinais físicos até protocolos de alto nível com os quais a maioria dos desenvolvedores de aplicativos de rede interagem.

Tecnologicamente, existem **modelos de comunicação de propósito genérico**, como o modelo OSI e o modelo TCP/IP, que foram desenvolvidos para atender a uma ampla gama de aplicações, especialmente em redes de computadores e na *internet*, oferecendo flexibilidade, escalabilidade e interoperabilidade entre sistemas diversos. Por outro lado, há **modelos de comunicação voltados para propósitos específicos**, como os utilizados em sistemas embarcados e industriais, que possuem requisitos mais restritivos, como baixa latência, alta confiabilidade, determinismo e tolerância a falhas. Esses modelos, presentes tanto em interfaces seriais de baixo nível, como SPI, I2C e UART, quanto em protocolos de rede mais

estruturados, como CAN, LIN, FlexRay e Ethernet Industrial, costumam ter menos camadas, funções mais integradas e são fortemente orientados à eficiência, simplicidade de implementação, baixa latência e alta robustez, atendendo aos requisitos específicos de sistemas embarcados e industriais.

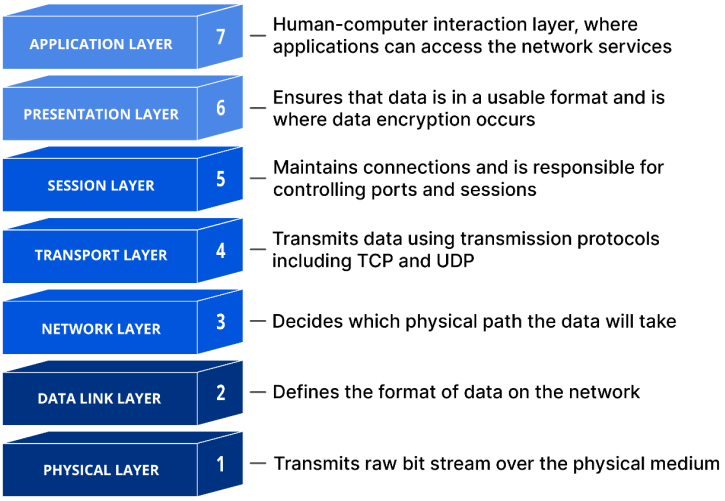
Modelos de comunicação de propósito genérico

Os modelos de comunicação são *frameworks* conceituais que dividem a complexidade da comunicação digital em camadas lógicas e bem definidas. Eles são de "propósito genérico" porque não são específicos de uma aplicação (como um *software* de e-mail), mas sim fornecem uma estrutura universal para a transmissão de dados em redes, independentemente da tecnologia subjacente do *hardware* (fibra óptica, Wi-Fi, Ethernet, etc.) ou do tipo de dado sendo transmitido.

Os dois exemplos mais proeminentes de modelos de comunicação de propósito genérico são os modelos OSI (do inglês *Open Systems Interconnection*) e TCP/IP (do inglês *Transmission Control Protocol/Internet Protocol*). O primeiro é um padrão de referência teórico (de direito), enquanto o segundo é o padrão implementado e usaso "de fato". Ambos os modelos são importantes. O OSI fornece uma estrutura conceitual mais granular, enquanto o TCP/IP é a base funcional das comunicações de rede no mundo real.

O modelo **OSI** serve como um guia fundamental para entender o processo de comunicação em redes, abrangendo desde a transmissão física dos dados na camada 1 até a interação com os aplicativos do usuário na camada 7. Embora não seja completamente implementado na prática como um conjunto de protocolos para a *internet*, ele é largamente empregado como uma ferramenta didática e de diagnóstico, oferecendo uma linguagem comum que auxilia profissionais de rede na análise e resolução de problemas. Como mostra a figura que segue, as sete camadas que compõem o Modelo OSI, dividindo a complexidade da comunicação, são:

- **Física** – transmissão elétrica, óptica ou rádio.
- **Enlace** (em inglês, *Data Link Layer*) – formatação de quadros de *bits*, incluindo detecção e/ou correção de erros.
- **Rede** – roteamento de pacotes.
- **Transporte** – controle de entrega, erros e fluxo.
- **Sessão** – gerenciamento de sessões de comunicação.
- **Apresentação** – codificação, compressão, criptografia.
- **Aplicação** – interface com o usuário (protocolos de alto nível).

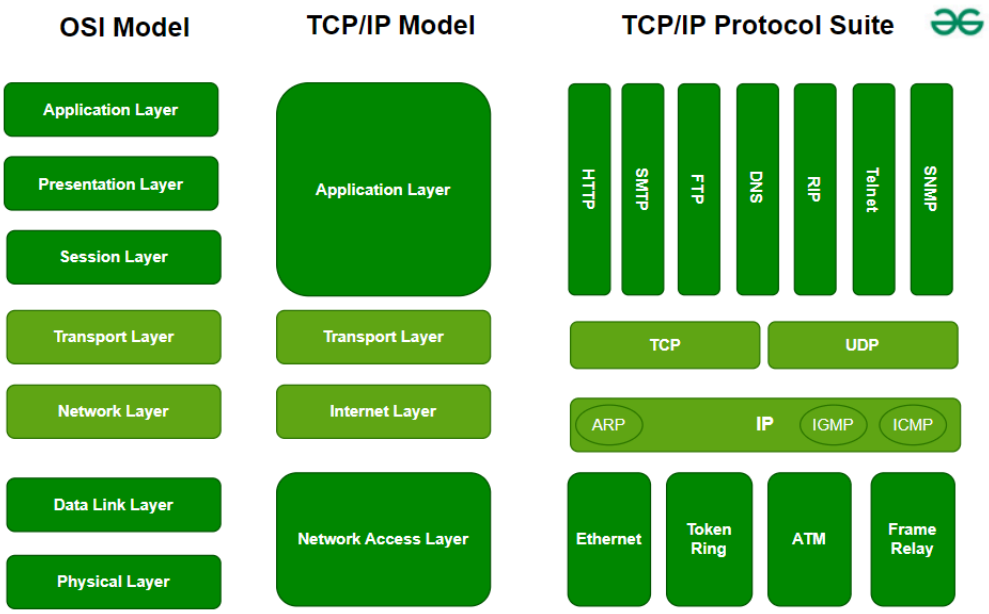


Fonte: Cloudflare

O modelo **TCP/IP** é a base tecnológica da *internet* e da maioria das redes que usamos todos os dias. Ele organiza as complexas tarefas de comunicação em quatro ou cinco camadas de abstração. A versão de quatro camadas, que é a mais comum e simplificada, inclui:

- **Camada de Acesso à Rede (ou Interface de rede/*link*) –** integra as funcionalidades da camada Física e de Enlace de dados do modelo OSI. Ela lida com os detalhes físicos da transmissão de dados através de uma mídia específica (como cabos Ethernet, Wi-Fi), incluindo o acesso ao meio físico e o endereçamento MAC (do inglês *Media Access Control*). Exemplos de protocolos nessa camada são Ethernet e Wi-Fi (IEEE 802.11).
- **Camada de *Internet* (ou Rede) –** corresponde à camada de Rede do modelo OSI. Lida com o roteamento de pacotes através de redes interconectadas. Ela é responsável por endereçar os pacotes e determinar o melhor caminho para que cheguem ao destino final, mesmo que passem por várias redes diferentes. Exemplo de protocolo amplamente usado é o IP (do inglês *Internet Protocol*).
- **Camada de Transporte** – corresponde à camada de Transporte do modelo OSI. É responsável por fornecer comunicação de ponta a ponta entre processos em diferentes *hosts*. Garante a entrega confiável e ordenada de dados (com TCP) ou oferece uma entrega mais rápida, mas não garantida (com UDP). Gerencia o fluxo de dados e a detecção/correção de erros. Exemplos de protocolos nessa camada são TCP (do inglês *Transmission Control Protocol*) e UDP (do inglês *User Datagram Protocol*).
- **Camada de Aplicação** – integra as funcionalidades das camadas de Sessão, Apresentação e Aplicação. É a camada mais próxima do usuário. Ela lida com os aplicativos e os serviços de rede que os usuários finais utilizam. Define como os aplicativos interagem com a rede e entre si. Como exemplos de protocolos nessa camada temos HTTP (navegação *web*), FTP (transferência de arquivos), SMTP (e-mail) e DNS (resolução de nomes de domínio).

A figura a seguir ilustra a correspondência entre as camadas do modelo de referência OSI e as camadas da tecnologia TCP/IP.



Fonte: Geeksforgeeks

Embora o modelo TCP/IP seja frequentemente descrito com quatro camadas, uma versão mais detalhada, que distingue entre as camadas Física e de Enlace, pode ser representada com cinco camadas. Ambas as representações são válidas e amplamente utilizadas. Quando essas camadas são implementadas na prática por meio de *software* e *hardware*, o conjunto resultante é chamado de **pilha de comunicação**, como a pilha TCP/IP utilizada na maioria dos sistemas conectados.

Modelos de comunicação para propósitos específicos

Embora os modelos OSI e TCP/IP forneçam uma base conceitual e prática amplamente utilizada em redes de computadores, eles não atendem plenamente às exigências específicas de ambientes industriais e embarcados. Esses sistemas demandam características rigorosas, como operação em tempo real, alta confiabilidade, tolerância a condições ambientais adversas, eficiência no uso de recursos e interoperabilidade com tecnologias legadas, que vão além das capacidades do modelo TCP/IP tradicional. Como resultado, foram desenvolvidos protocolos e modelos de comunicação especializados, capazes de oferecer desempenho superior em aplicações de missão crítica. Muitos desses protocolos industriais surgiram antes da ampla adoção da Ethernet convencional e, por isso, utilizam pilhas de comunicação próprias, frequentemente otimizadas para operação nas camadas Física e de Enlace, com foco em **comunicação determinística**. Atualmente, diversos padrões modernos também se baseiam em variações da Ethernet industrial, combinando alta velocidade com requisitos de tempo real.

Entre os modelos e padrões específicos para redes industriais e embarcadas com interfaces seriais de baixo nível, destacam-se:

- **I2C** (do inglês *Inter-Integrated Circuit*): Utilizado para comunicação de curta distância entre componentes na mesma placa. A camada física é composta por duas linhas de comunicação (SDA e SCL) com resistores *pull-up*. A camada de enlace trata do endereçamento dos dispositivos e do enquadramento dos dados em *bytes*. Esses aspectos foram abordados no Roteiro 10.
- **SPI** (do inglês *Serial Peripheral Interface*): Protocolo de alta velocidade também voltado para comunicação entre dispositivos na mesma placa. A camada física pode conter até quatro linhas (MISO, MOSI, SCLK, e SS/CS), enquanto a camada de enlace lida com o enquadramento dos dados. Detalhes foram discutidos no Roteiro 10.
- **UART** (do inglês *Universal Asynchronous Receiver/Transmitter*): Usado para comunicação serial ponto a ponto entre dois dispositivos. Sua camada física envolve linhas TX e RX (eventualmente GND), e a camada de enlace trata da sincronização e enquadramento de *bytes*. Esse protocolo foi abordado no Roteiro 7.

Os modelos, como CAN, LIN e DMX, possuem estruturas mais completas e padronizadas em comparação com protocolos como SPI, I2C e UART, que são voltados principalmente para comunicação ponto a ponto em curtas distâncias. Essa diferença se reflete na presença de camadas mais bem definidas e funções integradas, como controle de acesso ao meio, detecção de erros, e suporte a múltiplos nós em um barramento compartilhado. A seguir, são detalhadas as camadas de cada um desses protocolos — evidenciando como sua arquitetura oferece maior escalabilidade, robustez e capacidade de operação em ambientes com múltiplos dispositivos interconectados.

- **CAN** (do inglês *Controller Area Network*): Desenvolvido originalmente para sistemas automotivos, é amplamente adotado em automação e aplicações embarcadas devido à sua confiabilidade, tolerância a falhas e capacidade de operação em tempo real.
  - **Física**: Esta camada define as características elétricas e mecânicas do barramento, como os níveis de tensão dos sinais (diferencial, tipicamente CAN_H e CAN_L), as taxas de transmissão de *bits* (de 10 kbps a 1 Mbps), a impedância de terminação da rede (geralmente 120 ohms) e os tipos de conectores. A comunicação é realizada por meio de um par de fios trançados, com um sistema de arbitragem não-destrutiva baseado em *bits* para gerenciar o acesso ao meio e resolver colisões.
  - **Enlace**: É a camada central do CAN. Ela é responsável pela formatação e transmissão de mensagens (também chamadas de "*frames*"), pelo controle de acesso ao barramento. Além disso, implementa robustos mecanismos de detecção de erros (CRC, *bit stuffing*, etc.) e sinalização de falhas, garantindo a alta confiabilidade dos dados mesmo em ambientes ruidosos.
  - **Aplicação** (Application Layer): Ao contrário de redes mais complexas, o padrão CAN original não define uma camada de aplicação específica. Em vez disso, essa camada é implementada por protocolos de alto nível que constroem

sobre a camada de enlace do CAN para fornecer funcionalidades específicas para diferentes domínios de aplicação. Esses protocolos definem a interpretação dos dados dentro das mensagens CAN, a estrutura dos objetos de dados e as operações de rede. Exemplos de protocolo CAN nessa camada são CANopen (usado em automação industrial, controle de movimento, equipamentos médicos), J1939 (amplamente utilizado em veículos pesados, como caminhões, ônibus e máquinas agrícolas para diagnóstico e comunicação de componentes), e outros protocolos proprietários ou específicos de fabricantes.

- **LIN** (do inglês *Local Interconnect Network*): Uma alternativa mais simples e de menor custo ao CAN, projetada para aplicações de baixa velocidade, como controle de janelas e espelhos em veículos.
    - **Física**: Esta camada define a comunicação em fio único, o que contribui significativamente para seu baixo custo e simplicidade. Baseia-se no padrão ISO 9141 (linha K) e opera em níveis de tensão que geralmente se referem ao terra (0V - dominante) e à tensão da bateria do veículo (recessivo, tipicamente 12V ou 24V). As taxas de transmissão são relativamente baixas, variando de 1 kbps a 20 kbps, com um comprimento máximo de barramento de 40 metros. O barramento requer resistores de terminação específicos no mestre e nos escravos.
    - **Enlace**: A camada de enlace do LIN implementa um modelo mestre-escravo, onde um único nó mestre controla toda a comunicação no barramento. A transmissão de dados ocorre em quadros agendados pelo mestre, que envia um *header* (cabeçalho) contendo um identificador de mensagem. Os escravos respondem a esses *headers* preenchendo o quadro com dados (geralmente de 2, 4 ou 8 bytes). O LIN não possui arbitragem de barramento como o CAN; em vez disso, o mestre garante o controle do acesso ao meio através do agendamento. Mecanismos de detecção de erros (como *checksums* e *parity bits*) são incluídos para garantir a integridade dos dados, mas não há retransmissão automática de mensagens como no CAN.
    - **Aplicação**: Assim como o CAN, o LIN não define um protocolo de aplicação padrão em sua especificação original. A interpretação dos dados dentro das mensagens LIN e a lógica de comunicação de alto nível são definidas pelos desenvolvedores e pelos padrões específicos de cada aplicação. Essa camada é frequentemente auxiliada por um Arquivo de Descrição LIN (em inglês *LIN Description File* – LDF), que descreve as mensagens, sinais e agendamento da rede, permitindo a configuração e interoperabilidade entre os nós. O LIN é tipicamente usado para funções não-críticas e de baixa velocidade em veículos, como controle de janelas, travas de portas, espelhos, sistemas de conforto do assento, e iluminação interna.
- **DMX** (do inglês *Digital Multiplex*): Protocolo padrão na indústria de entretenimento, utilizado para o controle de equipamentos de iluminação profissional, efeitos de palco

e sistemas audiovisuais. Opera com comunicação unidirecional e simples, baseada em transmissão contínua de pacotes.

- **Física:** A camada física do DMX é baseada no padrão RS-485, utilizando um sinal diferencial para alta imunidade a ruído e distâncias de transmissão. Tipicamente, emprega cabos de par trançado blindado de 110 Ohms, com conectores XLR de 5 pinos (embora 3 pinos sejam, por vezes, erroneamente usados, com risco de confusão com áudio). O sinal é digital e unidirecional, transmitido a 250 kbaud. Uma única "linha" ou "universo" DMX pode suportar até 32 dispositivos e estender-se por distâncias consideráveis (centenas de metros), necessitando de terminações no final da linha.
- **Enlace:** A camada de enlace do DMX define a estrutura do fluxo de dados que é transmitido do mestre (console ou controlador DMX) para os escravos (luminárias, dimmers, máquinas de fumaça, etc.). A comunicação é iniciada por um "*Break*" (pulso de baixo nível) seguido por um "*Mark After Break*" (MAB), que sinaliza o início de um novo quadro de dados. Após o MAB, um "*Start Code*" (geralmente 0x00 para dados de dimmer/controle) é enviado, seguido por até 512 *bytes* de dados (cada um de 0 a 255). Cada *byte* de dados corresponde a um "canal" DMX. Os dispositivos escravos "escutam" o barramento e interpretam os bytes a partir do endereço DMX que lhes foi configurado. Não há detecção de erros ou retransmissão no próprio protocolo, e a comunicação é puramente unidirecional (mestre para escravo).
- **Aplicação:** No DMX, a camada de aplicação é onde a "personalidade" de cada dispositivo é definida e controlada. Cada um dos 512 canais em um universo DMX pode ser mapeado para uma função específica de uma luminária (por exemplo, brilho, cor, *pan*, *tilt*, estrobo, gobo, foco). O desenvolvedor do *fixture* define como os 256 níveis de um canal DMX (0-255) controlam suas funções. Por exemplo, no canal 1 de uma luminária, 0 pode ser "desligado" e 255 "brilho máximo". Para uma cabeça móvel, um canal pode controlar a cor, com diferentes valores correspondendo a diferentes cores predefinidas. É nesta camada que os programadores de iluminação definem a "linguagem" para controlar os efeitos visuais desejados.

Apesar de protocolos como CAN, LIN e DMX oferecerem soluções eficientes e consolidadas para redes embarcadas e industriais específicas, há uma tendência crescente de convergência e integração das redes em direção a arquiteturas baseadas em Ethernet. Essa movimentação busca simplificar a infraestrutura, aumentar a interoperabilidade entre dispositivos e facilitar a conectividade com sistemas maiores e distribuídos, como redes corporativas e ambientes de nuvem. No entanto, essa transição não é trivial: a implementação do modelo TCP/IP em sistemas embarcados impõe desafios significativos, como o consumo adicional de recursos computacionais, maior complexidade de *software* e dificuldades de integração com protocolos legados. Ainda assim, a demanda por padronização, escalabilidade e integração horizontal tem impulsionado a adaptação gradual de tecnologias baseadas em Ethernet, inclusive no contexto de aplicações de tempo real e controle crítico.

Nesse cenário, surge a **Ethernet Industrial** como uma solução intermediária, que combina a infraestrutura amplamente difundida da Ethernet convencional com os requisitos técnicos exigidos por aplicações industriais e de missão crítica. Diferentemente da Ethernet tradicional, que foi projetada para ambientes de escritório com tráfego de dados não

determinístico, a Ethernet Industrial incorpora mecanismos de tempo real, redundância, robustez contra interferências e tolerância a falhas, principalmente nas camadas Física e de Enlace. Esses aprimoramentos permitem sua operação confiável em ambientes hostis, como linhas de produção, usinas, veículos automotivos e sistemas embarcados complexos. Entre os exemplos mais utilizados estão:

- **EtherCAT** (do inglês *Ethernet for Control Automation Technology*): muito usado em sistemas de automação com requisitos de tempo real rigorosos, como controle de movimento e robótica industrial.
- **Profinet** (do inglês *Process Field Network*): amplamente empregado na indústria de manufatura e processos, oferecendo integração com dispositivos de campo e suporte a diagnósticos avançados.
- **Modbus TCP**: uma versão do protocolo Modbus, criado em 1979 pela Modicon (hoje parte da *Schneider Electric*), adaptada para redes Ethernet, frequentemente utilizada para supervisão, monitoramento e integração de sistemas legados com redes modernas.

Esses protocolos demonstram como a Ethernet Industrial permite a interligação de sensores, atuadores, controladores e sistemas de supervisão dentro de uma mesma rede, promovendo maior padronização, flexibilidade e conectividade, ao mesmo tempo em que mantém a confiabilidade exigida em aplicações industriais.

ARBITRAGEM DE BARRAMENTO

Após a apresentação do modelo de comunicação em camadas, é importante compreender como os dispositivos em uma rede compartilham o meio físico de transmissão, especialmente em barramentos onde múltiplos nós podem tentar se comunicar ao mesmo tempo. Esse processo é regido por **mecanismos de arbitragem**, que definem qual dispositivo pode transmitir em determinado momento, evitando colisões e garantindo a integridade da comunicação.

A **arbitragem de barramento** está tipicamente associada à **camada de Enlace**, pois é nessa camada que se realizam o controle de acesso ao meio, o enquadramento dos dados e a detecção de erros. Em redes industriais, esse controle é ainda mais relevante devido à presença de múltiplos dispositivos interligados em um mesmo barramento, muitos dos quais operam sob restrições de tempo real e precisão. Nesse contexto, surgem dois conceitos fundamentais: o de mestre e escravo. Em protocolos com uma arquitetura mestre-escravo, um dispositivo central (**o mestre**) controla o fluxo de comunicação, determinando quais **escravos** podem transmitir e quando. Já em redes mais flexíveis, sem um mestre fixo, a arbitragem é descentralizada, exigindo mecanismos mais sofisticados para resolver conflitos de transmissão de forma eficiente e sem perda de dados.

Diferentes protocolos de comunicação empregam distintos tipos de **controle de arbitragem** para gerenciar o acesso ao meio de transmissão. Essas abordagens variam desde métodos simples e centralizados até soluções robustas e descentralizadas, que permitem que múltiplos nós transmitam dados sem colisões, um conceito essencial para redes determinísticas e

altamente confiáveis. Cada tipo de controle de arbitragem possui suas próprias vantagens, limitações e é mais adequado para contextos específicos. A seguir, exploraremos os quatro principais tipos.

Sem mecanismo de arbitragem

Nesse modelo, o barramento **não possui um mecanismo interno de arbitragem**. A transmissão de dados é permitida para apenas um dispositivo por vez, e essa exclusividade é determinada por fatores externos ao próprio protocolo, como o design do sistema, a configuração física ou um acordo prévio entre os componentes.

Sua principal vantagem reside na simplicidade e no baixo custo de implementação, já que não há sobrecarga de controle de acesso. No entanto, a desvantagem é significativa: o sistema se torna totalmente dependente desse controle externo. Se dois dispositivos tentarem transmitir simultaneamente, ocorrerão colisões físicas no barramento, sem qualquer meio interno para resolvê-las.

Exemplos típicos incluem uma UART unidirecional simples, onde a transmissão de dados ocorre de ponto a ponto sem disputa, ou o uso de GPIOs paralelos para linhas de dados básicas, que também não possuem controle de acesso embutido. Além disso, sistemas com chaveamento manual ou outros métodos de controle físico se enquadram nessa categoria, como barramentos industriais simples que não utilizam multiplexação.

Controle centralizado

O **controle centralizado**, também conhecido como modelo **mestre-escravo**, é uma abordagem direta para gerenciar a comunicação em uma rede. Nele, um único dispositivo, designado como **mestre**, assume total responsabilidade por iniciar todas as comunicações e ditar qual dos outros dispositivos, chamados **escravos**, tem permissão para transmitir dados. Essa metodologia elimina colisões de forma eficaz, já que os escravos permanecem passivos, respondendo apenas quando solicitados pelo mestre.

A principal vantagem desse modelo reside na sua simplicidade e previsibilidade, o que facilita bastante a implementação. No entanto, apresenta limitações significativas. A flexibilidade é baixa, e a dependência de um único mestre cria um ponto único de falha, tornando-o ineficiente em redes com um grande número de nós ativos.

Dois exemplos práticos ilustram bem esse controle. No SPI, o mestre utiliza um sinal de seleção (CS/SS) para escolher o escravo com o qual deseja se comunicar e, em seguida, inicia a troca de dados; os escravos, por sua vez, jamais começam uma transmissão por conta própria. Similarmente, no LIN, um nó mestre envia cabeçalhos específicos para iniciar a comunicação, e os escravos só respondem quando são explicitamente solicitados.

Arbitragem simples descentralizada

No modelo de **arbitragem de colisões**, também conhecido como **multi-mestre**, diversos dispositivos têm a capacidade de tentar transmitir dados ao mesmo tempo. Contudo, ele incorpora um mecanismo inteligente que detecta e resolve esses conflitos de forma ordenada enquanto a transmissão está em andamento.

A principal vantagem dessa abordagem é que ela efetivamente evita colisões, permitindo a comunicação sem a necessidade de um mestre fixo. Isso a torna particularmente adequada

para sistemas que possuem múltiplos emissores ativos. Por outro lado, as limitações incluem a complexidade inerente à sua implementação, exigindo um sincronismo preciso e a aplicação de regras de prioridade estritas para garantir seu funcionamento.

Um exemplo clássico dessa dinâmica é o protocolo **I2C**. Nele, se dois mestres iniciam a transmissão simultaneamente, ambos monitoram o barramento *bit* a *bit*. No momento em que um nó envia um *bit* alto, mas detecta um *bit* baixo no barramento, ele automaticamente compreende que perdeu a arbitragem e cessa sua transmissão, permitindo que o outro dispositivo prossiga sem interrupções.

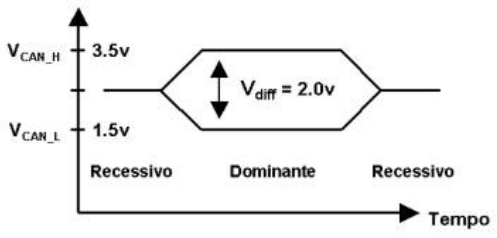
Arbitragem com priorização de identificadores

A **arbitragem por Identificador de mensagem**, também classificada como um sistema **multi-mestre**, representa um método de controle de acesso ao meio que opera de forma descentralizada, eficiente e determinística. Nesse sistema, a prioridade de transmissão é estabelecida pelos identificadores únicos associados a cada mensagem. O dispositivo que possui o identificador de menor valor binário, e, portanto, a maior prioridade, vence a arbitragem, garantindo que sua transmissão ocorra sem colisões.

As principais vantagens desse mecanismo incluem seu caráter determinístico, essencial para aplicações de tempo real, além de uma comunicação livre de colisões e perdas, mesmo quando múltiplos nós tentam acessar o barramento simultaneamente. Por outro lado, uma limitação relevante é a necessidade de uma definição rigorosa das prioridades das mensagens na fase de projeto, o que pode reduzir a flexibilidade frente a protocolos que oferecem QoS (do inglês *Quality of Service*) dinâmico, capazes de ajustar as prioridades conforme as condições operacionais em tempo real.

Um excelente exemplo desse método é encontrado na rede CAN. Durante a transmissão, a arbitragem ocorre em tempo real, comparando cada *bit* do identificador das mensagens diretamente no barramento físico. O funcionamento do barramento CAN é baseado em uma lógica "*wired-AND*", como I2C, onde a interação elétrica dos sinais garante a robustez da arbitragem:

- ***Bit* dominante (Lógica 0)**: Corresponde fisicamente a um estado onde o *transceiver* força uma diferença de potencial entre as linhas CAN_H e CAN_L, geralmente, CAN_H ≈ 3,5V e CAN_L ≈ 1,5V. Isso cria uma diferença de aproximadamente 2V, caracterizando um nível dominante no barramento.
- ***Bit* recessivo (Lógica 1)**: Nesse estado, os *transceivers* liberam o barramento, que é mantido passivamente pelo circuito de terminação. Ambas as linhas CAN_H e CAN_L ficam aproximadamente em 2,5V, ou seja, uma diferença muito pequena (próxima de 0V), indicando o nível recessivo.



Fonte: Alexag

O princípio elétrico fundamental da arbitragem é que um *bit* dominante (lógico '0') sempre sobrepõe um *bit* recessivo (lógico '1') no barramento. Isso significa que, se qualquer dispositivo transmitir um '0' (dominante), todos os outros dispositivos na rede verão esse '0', mesmo que estejam tentando transmitir um '1' (recessivo) ao mesmo tempo. Durante a fase de arbitragem, que ocorre na transmissão do identificador da mensagem, cada dispositivo age de forma simultânea: ele transmite um *bit* e monitora o barramento ao mesmo tempo, *bit* a *bit*, do mais significativo para o menos significativo.

Se um dispositivo tenta enviar um *bit* recessivo ('1'), mas detecta no barramento um *bit* dominante ('0'), ele imediatamente reconhece que perdeu a arbitragem. Isso indica que outro dispositivo, com um identificador de maior prioridade (ou seja, de menor valor binário), está transmitindo. O nó que perde a arbitragem cessa sua tentativa de transmissão na hora, passando para o modo de escuta (recepção), sem gerar qualquer tipo de colisão.

Esse processo garante que, independentemente de quantos dispositivos tentem transmitir ao mesmo tempo, apenas aquele com a mensagem de maior prioridade continuará a transmissão. É por essa razão que o mecanismo de arbitragem é chamado de **não-destrutivo**: o *bit* dominante enviado pelo nó vencedor se mantém no barramento, enquanto o nó perdedor simplesmente se retira, permitindo que a mensagem vencedora prossiga sem interrupções ou corrupção de dados.

É importante notar que o CAN transmite o *bit* mais significativo (em inglês, *Most Significant Bit* - MSB) primeiro para o ID da mensagem durante a fase de arbitragem e também para os *bits* de dados. Isso é fundamental para a arbitragem, pois o nó com o ID de menor valor (maior prioridade) será determinado rapidamente, *bit* a *bit*, do MSB para o LSB.

GRANULARIDADE DE COMUNICAÇÃO

Além dos modelos de comunicação e dos mecanismos de acesso ao meio, os protocolos também se diferenciam na forma como estruturam os dados que trafegam pelo barramento. Esse aspecto é conhecido como **granularidade de comunicação** e se refere à menor unidade de informação útil que pode ser transmitida de forma autônoma. Nos protocolos mais simples, como UART, SPI e I2C, essa granularidade ocorre no nível de *bytes*, transmitidos sequencialmente e com pouca ou nenhuma estruturação definida pela camada física. Já em protocolos mais sofisticados, como CAN e Ethernet, os dados são organizados em quadros (em inglês, *frames*[2]) que encapsulam não apenas a carga útil (em inglês, *playload*), mas também informações de controle, como identificadores, verificação de erros, sinalização de início e fim, e eventualmente mecanismos de priorização.

O tamanho desses quadros varia de acordo com o protocolo e influencia diretamente o comportamento da comunicação. No CAN, por exemplo, cada quadro transporta até 8 *bytes* de carga útil, refletindo sua otimização para a troca rápida de variáveis de controle em

[2] Embora o termo "*frame*" tenha sido usado na UART para o encapsulamento de um *byte*, é importante observar que essa é uma unidade de transmissão muito mais primitiva e de responsabilidades limitadas em comparação com os "frames" robustos e multifuncionais de protocolos de rede estruturados, como CAN e Ethernet.

sistemas embarcados e aplicações de tempo real. Por outro lado, no Ethernet, cada quadro pode carregar até 1500 *bytes* de dados úteis em sua configuração padrão, ou até mais quando se utilizam *jumbo frames*, tornando-o mais adequado para a transmissão de grandes volumes de dados, como arquivos, imagens ou vídeos.

Essa diferença na granularidade impacta diretamente aspectos como desempenho, latência, eficiência e robustez. Uma granularidade mais fina, como a do CAN, favorece a baixa latência e o alto determinismo, pois quadros pequenos são transmitidos rapidamente e liberam o barramento em intervalos curtos, permitindo que múltiplos eventos ou mensagens urgentes sejam processados quase em tempo real. Contudo, essa abordagem tem um custo: a proporção de *overhead*, ou seja, dos dados de controle presentes em cada quadro, se torna mais alta em relação à carga útil, reduzindo a eficiência da largura de banda quando há necessidade de transmitir grandes volumes de dados.

Por outro lado, uma granularidade mais grossa, como na Ethernet, torna a comunicação mais eficiente do ponto de vista da utilização da banda, já que o *overhead* é diluído em um maior volume de dados úteis. Isso reduz o número de quadros necessários para grandes transferências, aliviando o processamento e o uso do barramento. No entanto, essa mesma característica pode comprometer o tempo de resposta e o determinismo, já que um quadro grande ocupa o meio físico por mais tempo, atrasando outras transmissões, especialmente as de maior prioridade, o que é crítico em aplicações de controle ou tempo real.

O impacto da granularidade também se manifesta na recuperação de erros. Protocolos baseados em quadros pequenos conseguem detectar e retransmitir rapidamente apenas a unidade afetada, minimizando o impacto de falhas de comunicação. Já protocolos com quadros maiores, embora sejam mais eficientes em termos de proporção de dados úteis, enfrentam uma penalidade maior quando há erros, já que todo o quadro precisa ser retransmitido, aumentando o tempo de ocupação do barramento e a latência da comunicação.

Por fim, a granularidade influencia diretamente o dimensionamento de *buffers*, o processamento nos nós da rede e a complexidade do sistema. Protocolos com quadros pequenos demandam menos espaço de memória para cada unidade, facilitando o gerenciamento em dispositivos embarcados, mas exigem mais transações para completar grandes volumes de dados. Protocolos com quadros grandes, por outro lado, simplificam a movimentação de grandes blocos de dados, porém requerem mais memória para *buffers* e podem enfrentar desafios no gerenciamento de latência.

Assim, a escolha da granularidade não é uma questão puramente técnica, mas um equilíbrio entre os requisitos de cada aplicação. Redes de controle e automação industrial, como CAN e Ethernet industrial (EtherCAT ou PROFINET IRT), priorizam granularidade fina para garantir resposta rápida, robustez e determinismo. Redes orientadas à transmissão de grandes volumes de dados, como Ethernet convencional ou Fibre Channel, adotam granularidade mais grossa para maximizar a eficiência da largura de banda e reduzir o custo por *byte* transmitido.

MECANISMOS DE SINCRONIZAÇÃO

Após compreender os métodos de arbitragem utilizados para controlar o acesso ao meio compartilhado, é essencial entender como os dispositivos mantêm a sincronização durante a

transmissão dos dados. Esse tema responde a uma pergunta fundamental em qualquer sistema de comunicação digital:

Como o receptor sabe onde começa e onde termina cada *bit* ou cada mensagem?

A transmissão de dados digitais exige que o receptor consiga interpretar corretamente os sinais recebidos, identificando os instantes exatos em que cada *bit* deve ser lido. Essa sincronização é garantida por mecanismos específicos, implementados nas **camadas físicas e de Enlace**, que asseguram tanto o alinhamento temporal quanto estrutural dos dados. Além de permitir que os dados sejam corretamente interpretados, esses mecanismos aumentam a imunidade a ruídos, preservam a integridade da informação e garantem taxas de transmissão confiáveis, mesmo em ambientes eletricamente ruidosos.

Os métodos de sincronização variam de acordo com o tipo de protocolo. Vimos no Roteiro 7 que **protocolos assíncronos UART,** não utilizam um sinal de *clock* compartilhado. Nesse caso, transmissor e receptor devem operar na mesma taxa de transmissão, definida previamente. A cada *byte* transmitido, utiliza-se uma estrutura composta por um *start bit*, seguido pelos *bits* de dados, um possível *bit* de paridade e um ou mais *stop bits*. O *start bit* (nível lógico baixo) indica o início da transmissão e permite que o receptor alinhe seu temporizador interno para realizar a amostragem dos *bits* subsequentes. O *stop bit* (nível lógico alto) marca o fim do *byte* e assegura que o receptor finalize corretamente a recepção antes de aguardar o próximo caractere. Este processo de realinhamento a cada *byte* reduz os **efeitos de *drift*** (desvio temporal acumulado), comum em sistemas assíncronos.

Por outro lado, o Roteiro 10 nos mostra que **protocolos síncronos como SPI e I²C** utilizam uma linha de *clock* compartilhado, gerada tipicamente pelo dispositivo mestre. Cada transição do sinal de clock define o instante exato em que os bits de dados devem ser lidos ou gravados. Dessa forma, não há necessidade de start ou stop bits para alinhamento dos bits individuais, pois o clock externo garante a sincronização precisa durante toda a comunicação.

Em redes industriais mais sofisticadas, como CAN e Ethernet, são utilizados mecanismos avançados de sincronização. Junto com eles, são introduzidos conceitos essenciais de segmentação de tempo em *bits* e quadros (em inglês *frames*), *bit stuffing* e sincronização de mensagens.

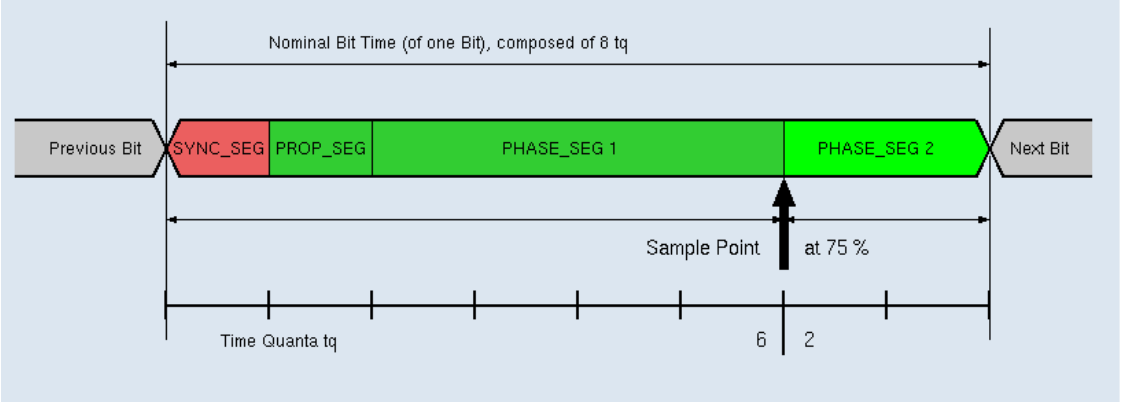
Segmentação de tempo em *bits*

A **segmentação de tempo em *bits*** (em inglês, *Bit Time Segmentation*) é a divisão precisa do período de duração de um único *bit* (em inglês *bit time*) em segmentos menores e bem definidos. Essa técnica é crucial para sincronizar os nós em um barramento de comunicação e para tolerar atrasos de propagação e erros de fase no sinal. Ela garante que todos os nós na rede concordem sobre o estado do barramento a cada instante, mesmo com variações e atrasos, o que permite uma comunicação robusta e determinística.

No barramento CAN, um exemplo clássico onde essa segmentação é fundamental, cada tempo de *bit* nominal (em inglês *Nominal Bit Time* – NBT) é dividido em quatro segmentos:

- **Segmento de Sincronização** (em inglês, *Sync Segment* - SyncSeg): É o primeiro segmento do *bit time* e tem um tamanho fixo (geralmente 1 ***Time Quantum*** - TQ). Ele é usado para sincronizar os nós na rede, marcando o ponto de transição esperado do sinal.

- **Segmento de Propagação** (em inglês, *Propagation Segment* - PropSeg): Compensa os atrasos físicos que o sinal leva para viajar pelos cabos e *transceivers* entre os nós. Seu tamanho é programável e ajustado para a topologia e comprimento da rede.
- **Segmento de Fase 1** (em inglês, *Phase Segment 1* - PS1): Usado para compensar erros de fase das bordas do sinal no barramento. Pode ser estendido durante a ressincronização para corrigir desvios de temporização.
- **Segmento de Fase 2** (em inglês, *Phase Segment 2* - PS2): Também compensa erros de fase e pode ser encurtado durante a ressincronização. É onde o "ponto de amostragem" (em inglês, *sample point*) geralmente ocorre, que é o momento em que o receptor lê o valor do *bit* no barramento.



Fonte: CAN-Wiki

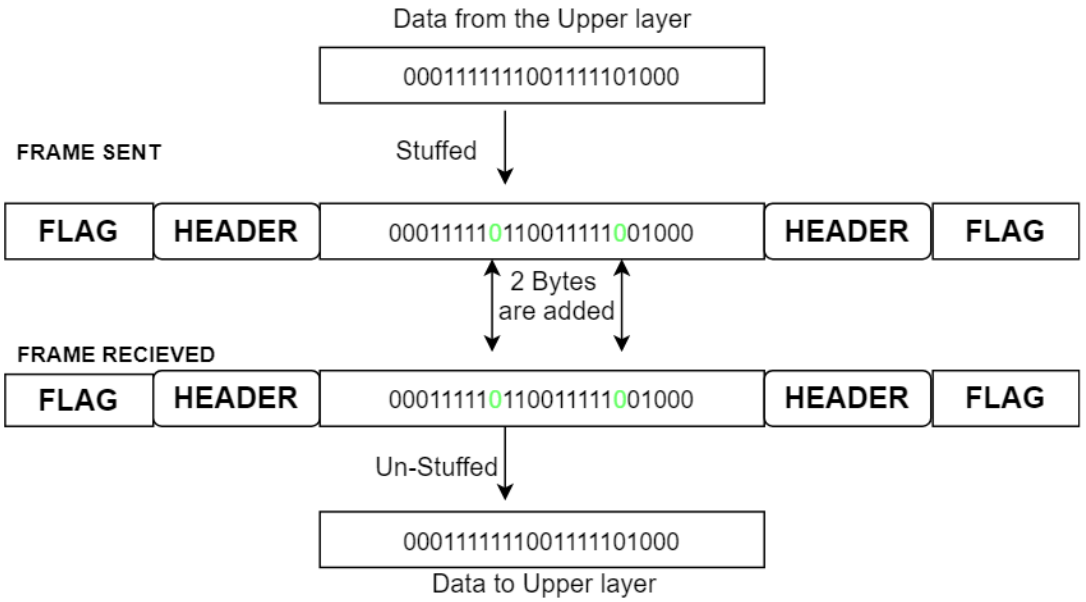
O *bit time* nominal no CAN é fixo para uma dada taxa de transmissão (baud rate). No entanto, a divisão em segmentos programáveis como *Propagation Segment*, *Phase Segment 1* (PS1) e *Phase Segment 2* (PS2) permite que o controlador CAN ajuste o *bit time* real em resposta a desvios de sincronização. A principal ferramenta para essa adaptação é o ***Resynchronization Jump Width*** (SJW) que define, por *software*, o **valor máximo** pelo qual PS1 ou PS2 podem ser encurtados ou alongados em uma única ressincronização. Isso evita ajustes excessivos que poderiam desestabilizar a rede.

Bit stuffing

O ***bit stuffing*** é uma técnica empregada para garantir a sincronização de *bits* e a integridade dos dados em transmissões digitais. Seu propósito principal é evitar que sequências específicas de *bits*, que servem como delimitadores ou sinais de controle (como *flags* de início/fim de quadro), apareçam acidentalmente dentro da carga útil (em inglês, *playload*) da mensagem.

Em sistemas que codificam dados por transições de sinal (por exemplo, bordas de subida ou descida), sequências muito longas de *bits* idênticos poderiam causar a perda de sincronização, pois a ausência de transições dificulta o realinhamento dos nós. O *bit stuffing* resolve isso forçando transições regulares: após a transmissão de cinco *bits* consecutivos de mesmo valor (todos '1's ou todos '0's), o transmissor insere automaticamente um *bit* de valor oposto. O receptor, ciente dessa regra, remove esse *bit* extra durante a recepção, restaurando a mensagem original. Essas transições forçadas permitem que os nós possam se ressincronizar continuamente, auxiliando na **recuperação do *clock* interno** (em inglês, ***bit timing recovery***), processo que utiliza a segmentação de tempo em *bits*. Além de preservar o sincronismo, o *bit*

stuffing aumenta a imunidade a ruído, já que as transições frequentes permitem uma detecção mais rápida de distúrbios, e ajuda a manter o sincronismo mesmo em ambientes industriais sujeitos a interferências eletromagnéticas.



Fonte: Codespeedy

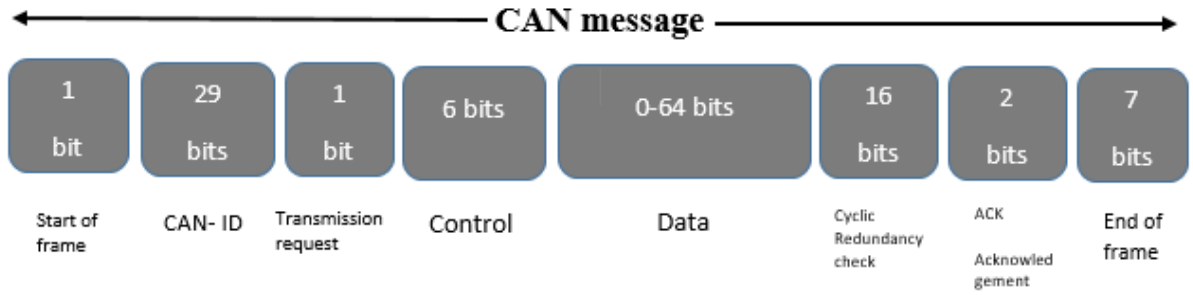
O *bit stuffing* é uma característica intrínseca e essencial do protocolo CAN para garantir a integridade dos dados e a sincronização do *clock* entre os nós na rede. Para a Ethernet, o *bit stuffing* no mesmo sentido do barramento CAN não é diretamente utilizado. A Ethernet emprega outras técnicas de codificação de linha (como 8B/10B para *Fast Ethernet* e *Gigabit Ethernet*, ou 64B/66B para velocidades ainda maiores) que garantem um número mínimo de transições para a recuperação do *clock* e evitam longas sequências de *bits* idênticos, além de utilizarem preâmbulos e delimitadores específicos para o início e fim dos quadros, que não são confundidos com os dados.

Segmentação de quadros

Segmentação de quadros (em inglês, *frame segmentation*) é o processo de dividir uma unidade de dados maior (como uma "mensagem" ou "pacote" de uma camada superior) em partes menores, discretas e de tamanho limitado, chamadas quadros (em inglês, *frames*), antes de enviá-las pela rede. Essa divisão otimiza a eficiência e a confiabilidade do transporte de dados, transformando informações extensas em pedaços gerenciáveis. É um mecanismo essencial, presente em praticamente todos os protocolos. Na prática, consiste em definir claramente os limites de início e fim de cada mensagem transmitida. Essa organização permite que múltiplas mensagens sejam transmitidas no mesmo meio de forma ordenada e sem ambiguidades.

Em redes industriais, por exemplo, a segmentação pode ser crucial para assegurar que mensagens críticas em tempo real não sejam atrasadas por mensagens longas e de baixa prioridade. Um exemplo disso é a Ethernet Sensível ao Tempo (em inglês, *Time-Sensitive Networking* – TSN), onde técnicas como o *Frame Preemption* permitem que um quadro de alta prioridade interrompa a transmissão de um quadro de menor prioridade, enviando-o em várias partes.

No contexto do CAN Bus, suas mensagens já possuem um tamanho máximo de 8 *bytes* de carga útil, o que significa que o protocolo define um "quadro" naturalmente pequeno. Caso uma aplicação precise enviar mais do que 8 *bytes*, ela terá que dividir essa informação em múltiplas mensagens CAN, caracterizando uma segmentação no nível da aplicação ou em uma camada de protocolo superior ao CAN físico. Já na Ethernet, a Unidade de Transmissão Máxima (em inglês, *Maximum Transmission Unit - MTU*) é de até 1500 bytes de carga útil. Se um pacote gerado na camada de aplicação exceder esse limite, ele será fragmentado (segmentado) na própria camada de aplicação ou na camada de rede em pacotes menores, que então se encaixam nos quadros Ethernet. Ambos os protocolos utilizam campos específicos, como delimitadores de início e fim de quadro, para demarcar claramente o início e o fim dos dados e garantir que as mensagens sejam interpretadas corretamente.



Fonte: ResearchGate

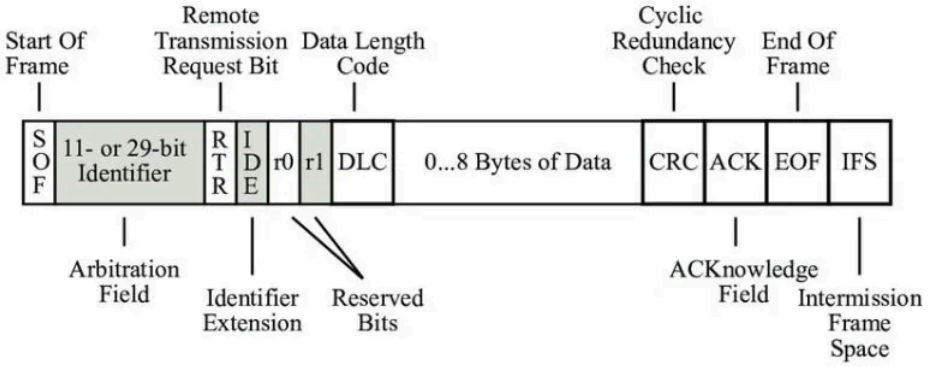
Sincronização de mensagens

Sincronização de mensagens (em inglês, *Time Synchronization*) garante que os relógios de tempo em diferentes dispositivos de uma rede estejam coordenados e alinhados com alta precisão. Diferente da sincronização de *bits*, que foca no alinhamento de cada *bit* individual, a sincronização de mensagens é crucial em sistemas distribuídos, como em aplicações de automação e redes sensíveis ao tempo, onde eventos devem ocorrer de forma coordenada. Sincronização assegura a sequência exata de eventos entre dispositivos fisicamente separados, o que é vital para que os *timestamps* em logs, diagnósticos, análises forenses e de segurança reflitam a hora real dos acontecimentos. Além disso, a sincronização perfeita de comandos é essencial para o controle coordenado em aplicações de movimento com múltiplos eixos. Por fim, em setores como finanças e energia, a precisão da sincronização não é apenas uma conveniência, mas um requisito de conformidade regulatória.

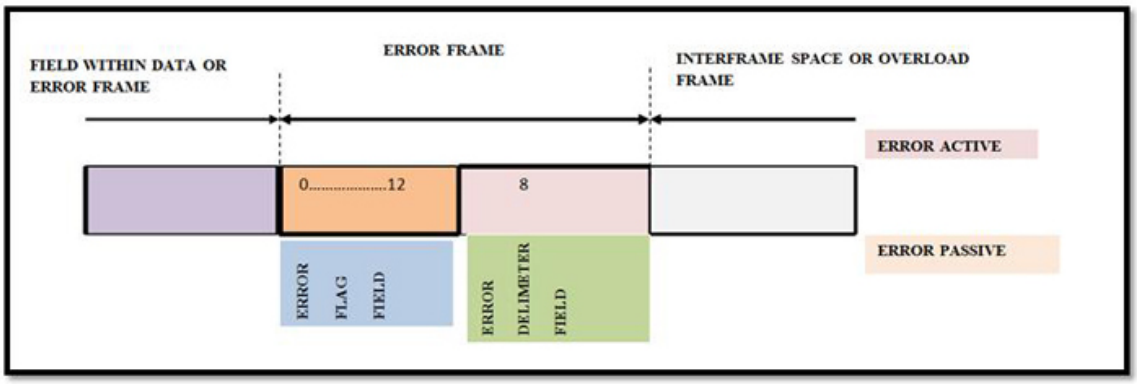
Essa sincronização é geralmente realizada por protocolos de rede específicos:

- NTP (do inglês *Network Time Protocol*): É o protocolo mais comum para sincronização de tempo na *internet* e em muitas redes corporativas. Ele usa algoritmos para compensar latências e *drifts* de relógio, mas oferece precisão em milissegundos.
- PTP (do inglês *Precision Time Protocol* - IEEE 1588): É um protocolo de sincronização de tempo de alta precisão (em microssegundos ou até nanossegundos), projetado para aplicações que exigem tempo real rigoroso, como redes industriais (Ex: PROFINET IRT, EtherCAT, Ethernet/IP com PTP). Ele usa mensagens de tempo trocadas entre mestres e escravos para calcular e compensar atrasos.
- Mecanismos Internos do Protocolo: Alguns protocolos industriais (como EtherCAT) têm mecanismos de sincronização distribuída incorporados diretamente em sua operação de camada de enlace para alinhar os relógios dos nós conectados.

O barramento CAN não possui um protocolo de sincronização de tempo integrado em seu nível mais baixo. No entanto, ele, assim como muitos outros protocolos, utiliza delimitadores de quadro (como *start of frame* - SOF, *end of frame* - EOF) e delimitadores de erro representados por 8 *bits* recessivos para garantir que o início e o fim da mensagem completa sejam interpretados corretamente. A Ethernet padrão, por sua vez, também não garante a sincronização de tempo entre os dispositivos por si só. Contudo, com a evolução para a TSN (do inglês, *Time-Sensitive Networking*) e a incorporação do protocolo PTP (IEEE 1588), a Ethernet industrial agora consegue alcançar níveis muito altos de sincronização de tempo, o que a torna adequada para aplicações que exigem tempo real rigoroso.



Fonte: Medium



Fonte: compraco

MECANISMOS DE ROBUSTEZ

Após entendermos como os dispositivos compartilham o meio de comunicação através de mecanismos de arbitragem e como alinham os dados no tempo por meio da sincronização, surge uma questão igualmente crítica: como garantir que esses dados trafeguem de forma íntegra, segura e sem erros, mesmo em ambientes repletos de ruídos, interferências e falhas ocasionais?

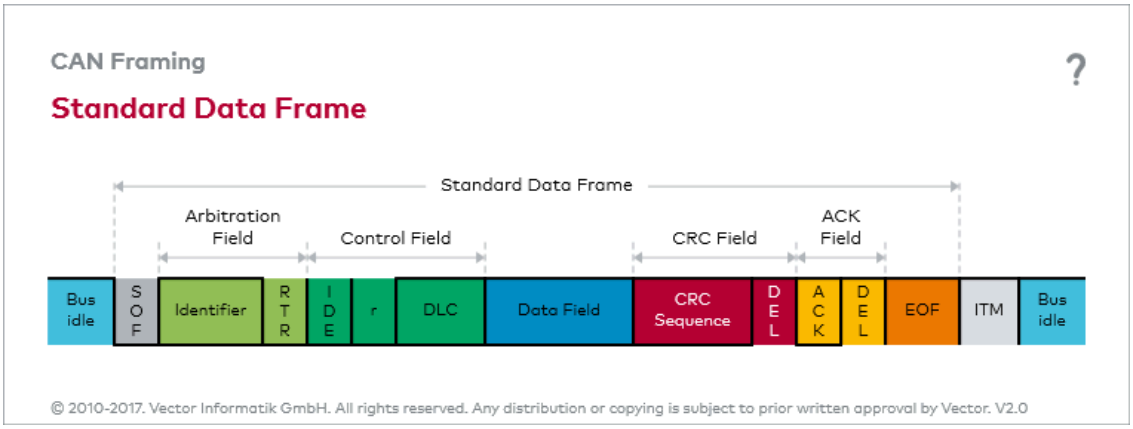
Diferente de redes de propósito geral, como Wi-Fi ou Ethernet doméstica, onde a perda ocasional de um pacote é tolerável, redes industriais, automotivas e embarcadas operam com dados críticos, em tempo real e sob rigorosos requisitos de segurança. Nesses ambientes, uma

falha na transmissão, seja de um comando de frenagem, no controle de um motor ou na leitura de um sensor, pode acarretar consequências severas, comprometendo a segurança de pessoas, a integridade física de equipamentos e a confiabilidade de processos, podendo, inclusive, causar desde paralisações na produção até acidentes graves. A confiabilidade é, portanto, um requisito essencial para essas redes. Não basta que os dispositivos saibam "quando transmitir" (arbitragem) e "como alinhar os *bits* no tempo" (sincronização); eles também precisam detectar erros, evitar a propagação de dados corrompidos e, sempre que possível, corrigir ou solicitar a retransmissão de informações com falha.

Mesmo em interfaces seriais de baixo nível, já encontramos alguns mecanismos de robustez, embora simples. Protocolos assíncronos como a UART possuem um sistema extremamente básico: a verificação de erro é opcional e limitada ao *bit* de paridade, que detecta erros de um único *bit*, e à detecção da falta do *stop bit*, que sinaliza erros grosseiros no alinhamento do *byte*. Nesses casos, não há mecanismos nativos de *ACK* (confirmação) ou de retransmissão automática; qualquer verificação adicional depende do protocolo implementado na camada de aplicação. Já os protocolos síncronos SPI e I2C oferecem integridade no nível dos *bits* graças ao *clock* compartilhado. No entanto, eles praticamente não possuem mecanismos nativos de detecção de erros, além do controle básico de *ACK/NACK* no I2C, que apenas confirma a recepção do *byte* sem validar seu conteúdo. A checagem de integridade e eventuais retransmissões, se necessárias, devem ser implementadas pela camada de aplicação.

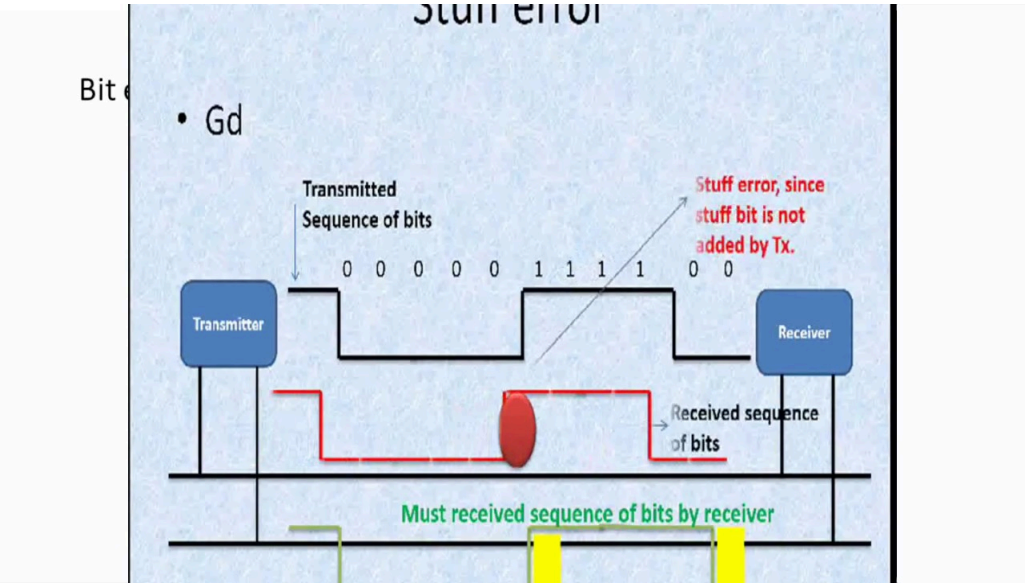
Para atender ao elevado nível de exigência dos ambientes hostis, os protocolos de comunicação concebidos para redes automotivas e industriais implementam uma série de mecanismos que trabalham em conjunto:

- **Checagem Cíclica de Redundância** (em inglês, *Cyckic Redundancy Check* - CRC): O CRC é um algoritmo essencial para a detecção de erros em dados transmitidos. Baseado em divisões polinomiais aplicadas à mensagem, o CRC gera um código de verificação que permite ao receptor identificar alterações acidentais nos dados, como inversões de *bits* causadas por ruído ou interferências. Se o CRC calculado pelo receptor não corresponder ao código enviado pelo transmissor, a mensagem é considerada inválida. A figura a seguir ilustra um quadro padrão do CAN, que inclui um campo de *bits* dedicado ao armazenamento do código CRC gerado pelo transmissor.
- ***Acknowledgment*** (ACK) **de quadro recebido**: Alguns protocolos, como o CAN, implementam um mecanismo de ACK diretamente no nível físico. Quando um receptor valida que um quadro foi recebido corretamente, ou seja, sem erros de CRC, *bit stuffing* ou formatação, ele gera um sinal de ACK no barramento. Esse sinal permite que o transmissor confirme que sua mensagem foi entregue com sucesso. A figura a seguir demonstra o campo de *bits* reservado para o sinal de ACK.



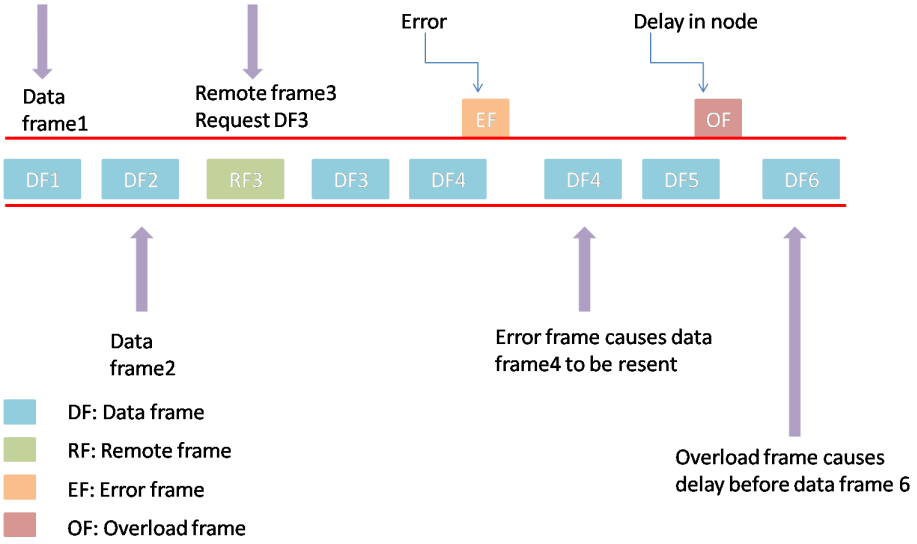
Fonte: Vector

- **Detecção de erros de *bit stuffing***: Nos protocolos que utilizam *bit stuffing*, como o CAN, a própria violação da regra de inserção de *bits* funciona como um detector de erro. Se o receptor identifica uma sequência de *bits* que desobedece à regra do *stuffing*. por exemplo, ao receber seis *bits* consecutivos de mesmo valor, ele imediatamente reconhece um erro de transmissão.



Fonte: Vector

- **Retransmissão Automática**: Quando um erro é detectado, seja por falha no CRC, erro de *bit stuffing* ou ausência de *ACK*, o protocolo pode automaticamente retransmitir o quadro, sem a necessidade de intervenção do *software* da aplicação, como ilustrado na figura a seguir. No CAN, por exemplo, essa retransmissão é feita de forma nativa, o que o torna um protocolo extremamente robusto e tolerante a falhas momentâneas.



Fonte: Embien

Enquanto protocolos simples como UART, SPI e I2C dependem principalmente da camada de aplicação para controle de erros, o protocolo CAN foi projetado desde sua origem para operar em ambientes críticos, como automóveis, veículos pesados, sistemas industriais e robótica. Sua robustez contra erros de transmissão, ruído e falhas temporárias é um dos principais fatores que explicam sua ampla adoção nesses setores.

O CAN incorpora mecanismos de robustez diretamente nas camadas **Física** e de **Enlace**, sem depender de camadas superiores. Entre esses mecanismos estão CRC obrigatório, que garante a integridade dos dados transmitidos, detecção automática de erros de *bit stuffing*, assegurando a correta interpretação dos dados no barramento, mecanismo de ACK embutido no próprio barramento, onde todos os nós confirmam se a mensagem foi recebida corretamente, e retransmissão automática, acionada sempre que qualquer nó detecta erro na mensagem. Além disso, o CAN implementa um sofisticado sistema de controle de erros, capaz até de desconectar temporariamente um nó defeituoso, preservando a integridade e estabilidade da rede como um todo.

FILTRAGENS DE MENSAGENS

Em redes de comunicação orientadas a transmissão a todos os nós (em inglês, *broadcast*), como o protocolo CAN, todas as mensagens transmitidas no barramento são visíveis para cada dispositivo conectado. Contudo, nem todas essas mensagens são relevantes para todos os nós da rede. Para gerenciar essa situação de forma eficiente, esses protocolos incorporam um mecanismo chamado **filtragem de mensagens**. Essa filtragem permite que cada dispositivo selecione automaticamente quais mensagens devem ser processadas e quais podem ser descartadas, realizando essa triagem já na camada de enlace ou até na interface física. É importante notar que protocolos mais simples, como UART, I2C e SPI, não utilizam esse tipo de filtragem. Eles operam de maneiras mais diretas: UART em comunicação ponto a ponto, o I2C por endereçamento explícito e o SPI por seleção de dispositivo, onde o mestre controla diretamente a comunicação.

O principal objetivo da filtragem é reduzir a carga de processamento dos dispositivos, evitando que eles precisem analisar e descartar manualmente informações que não são de seu interesse. Além disso, ela contribui diretamente para a organização lógica da comunicação e para a obtenção de respostas rápidas e determinísticas, algo essencial em sistemas embarcados, automotivos e industriais, onde atrasos no processamento de dados podem comprometer a operação do sistema. Portanto, a filtragem de mensagens viabiliza a comunicação eficiente em barramentos compartilhados, permitindo que diferentes fluxos de informação coexistam sem sobrecarregar desnecessariamente os dispositivos. Além de melhorar a escalabilidade da rede, esse mecanismo contribui diretamente para o determinismo e a robustez do sistema, características indispensáveis em aplicações críticas.

O funcionamento da filtragem se baseia no campo de identificador presente nas mensagens. Esse identificador, em vez de representar diretamente um endereço de origem ou destino, descreve o conteúdo ou a função da mensagem. Cada dispositivo da rede possui registradores internos específicos para configurar filtros e máscaras de aceitação dos identificadores (IDs). Quando uma mensagem chega, o controlador compara o ID da mensagem com os critérios definidos nos filtros. Se houver uma correspondência, a mensagem é entregue ao microcontrolador; caso contrário, ela é descartada de forma automática, sem ocupar tempo de processamento.

Existem diferentes formas de implementar a filtragem. A mais simples é a **filtragem por identificação exata**, na qual o dispositivo só aceita mensagens cujo ID corresponde exatamente ao valor programado no filtro. Uma abordagem mais flexível é a **filtragem por máscara e padrão**, que permite selecionar grupos de mensagens com IDs semelhantes. Nesse método, uma máscara binária define quais *bits* do ID são considerados na comparação e quais podem ser ignorados. Sempre que um *bit* da máscara está ajustado como 1, significa que aquele *bit* do ID será comparado; se está em 0, aquele *bit* é irrelevante para o filtro. Dessa forma, é possível configurar, por exemplo, um nó que aceite qualquer mensagem cujo ID comece com uma sequência específica, independentemente dos *bits* restantes. A seguinte figura ilustra o mecanismo de filtragem via uma máscara.

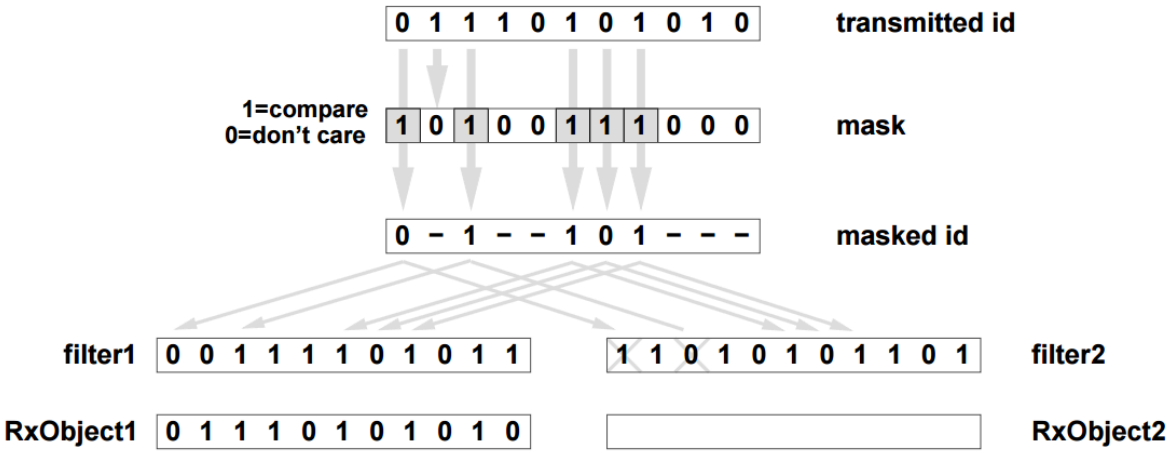


Figure 2.11: Masks and filters for message reception.

Fonte: Cnblogs

Em protocolos como o CAN, essa filtragem é realizada diretamente pelo *hardware* do controlador de barramento, o que garante extrema eficiência. A capacidade de configurar múltiplos filtros e máscaras permite que um único dispositivo participe de múltiplos grupos funcionais, ouvindo diferentes tipos de mensagens simultaneamente. Já em protocolos como Ethernet, a filtragem normalmente ocorre baseada nos endereços MAC, permitindo que pacotes *unicast*, *multicast* ou *broadcast* sejam tratados de acordo com a configuração da placa de rede.

Para entender melhor como essa filtragem funciona na prática, consideremos um exemplo: Imagine uma rede CAN em um veículo, onde vários módulos eletrônicos trocam informações continuamente. Suponha que existam mensagens diferentes, como dados do motor, *status* dos freios e informações do sistema de iluminação, cada uma identificada por um identificador (ID) único. Um módulo responsável apenas pelo controle da iluminação deve receber somente mensagens relacionadas a esse sistema. Se a rede transmite IDs como:

- 0x100 a 0x10F: Dados do motor,
- 0x200 a 0x20F: *Status* dos freios,
- 0x300 a 0x30F: Sistema de iluminação,

o módulo da iluminação pode configurar um filtro com máscara para aceitar somente mensagens cujo ID esteja na faixa 0x300 a 0x30F. Para isso, ele usa uma máscara binária que seleciona os *bits* relevantes do identificador e descarta os demais. Dessa forma, quando uma mensagem com ID 0x305 chega, o controlador CAN do módulo da iluminação compara o ID com o filtro configurado e aceita a mensagem para processamento. Se chegar uma mensagem com ID 0x105, ela será descartada automaticamente pelo *hardware*, sem ocupar recursos do microcontrolador. Este mecanismo evita que o módulo da iluminação precise analisar todas as mensagens da rede, permitindo uma resposta rápida e eficiente apenas às informações que são realmente importantes para sua função.

PROTOCOLOS DE REDES INDUSTRIAIS ESTRUTURADOS

Agora que já exploramos os conceitos fundamentais que sustentam o funcionamento das redes de comunicação, como modelos de comunicação, arbitragem de acesso ao meio, granularidade, sincronização, mecanismos de robustez e filtragem de mensagens, é possível compreender como esses princípios se materializam na prática por meio de protocolos concretos. Nesta seção, vamos apresentar dois exemplos de protocolos estruturados bastante representativos, mas aplicados a contextos muito diferentes: DMX512 e CAN. Uma visão mais detalhada desses dois protocolos permite consolidar os conceitos teóricos previamente discutidos.

PROTOCOLO 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.

PROTOCOLO CAN

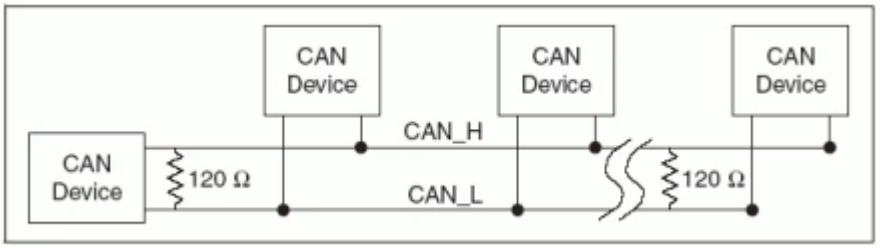
O CAN (do inglês *Controller Area Network*) é um protocolo de comunicação em rede amplamente utilizado em sistemas embarcados, particularmente em automóveis, mas também em áreas como automação industrial, robótica e dispositivos médicos. Foi desenvolvido pela empresa alemã Bosch na década de 1980 com 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.

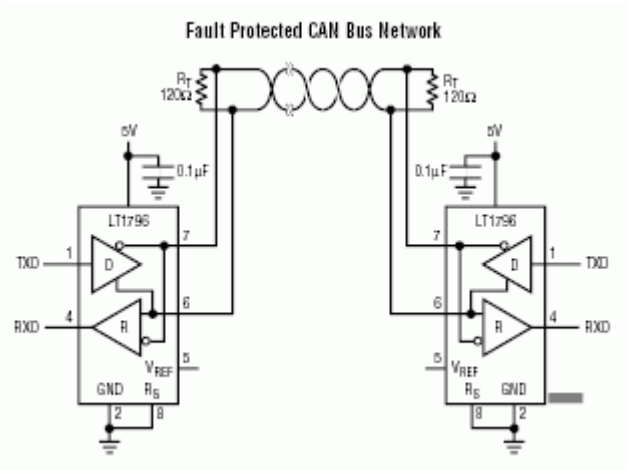
A primeira versão do protocolo foi lançada em 1986, com a padronização ocorrendo em 1993 como ISO 11898. A tecnologia CAN rapidamente se tornou uma escolha padrão para a indústria automotiva, sendo adotada por fabricantes para controlar sistemas como motores, transmissões, *airbags* e sistemas de assistência 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 no nível de mensagens**. 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.





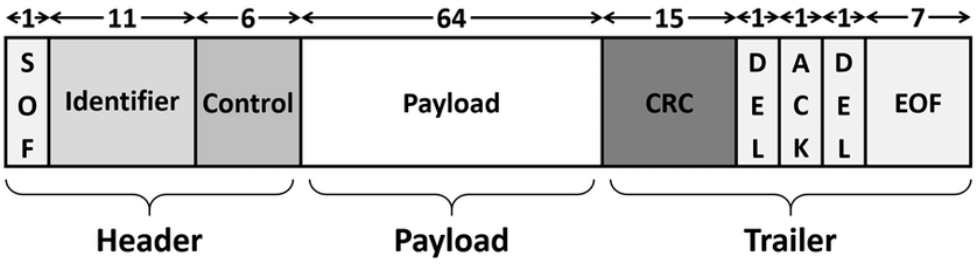
Assim como no protocolo I2C, quando mais de um dispositivo coloca um nível lógico no barramento, um dos níveis predomina. Este é denominado ***bit* dominante** (0 lógico), enquanto o outro é denominado ***bit* recessivo** (1 lógico)**.** No protocolo I2C, os resistores *pull-up* são

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**

O protocolo CAN é amplamente utilizado em aplicações automotivas e industriais devido à sua robustez, confiabilidade e capacidade de operar em ambientes eletricamente ruidosos. Para que um microcontrolador se comunique fisicamente com o barramento CAN, é indispensável o uso de *transceivers*, que fazem a conversão entre os sinais digitais TTL e os sinais diferenciais utilizados no barramento. Nesse contexto, o *transceiver* MCP2551 foi, por muitos anos, uma das soluções mais populares, oferecendo não apenas a conversão de sinais, mas também recursos como proteção contra curtos e transientes elétricos, controle de inclinação para reduzir emissão de ruído (RFI) e modos de baixo consumo. No entanto, atualmente não é mais recomendado para novos projetos, sendo o MCP2561 o substituto indicado, com melhor desempenho e conformidade com padrões mais recentes.

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.

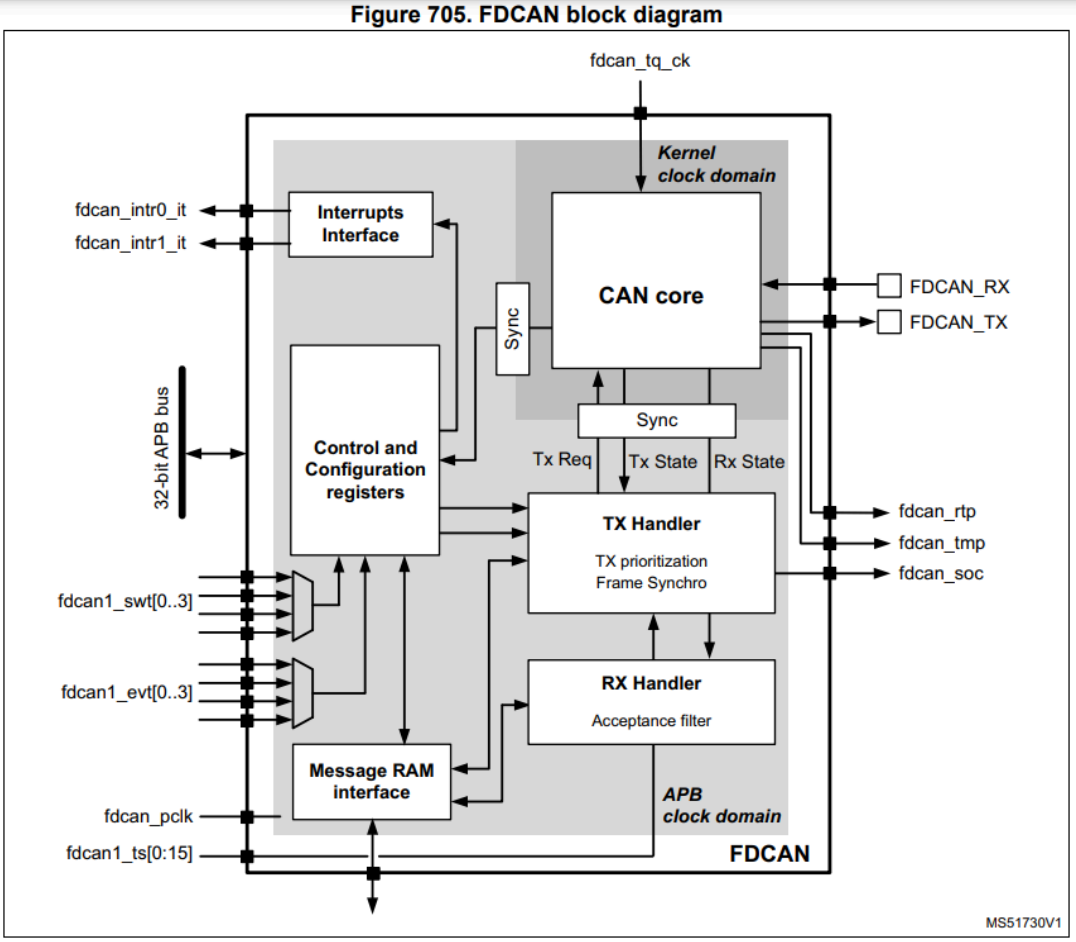
STM32H7A3: MÓDULO FDCAN

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.

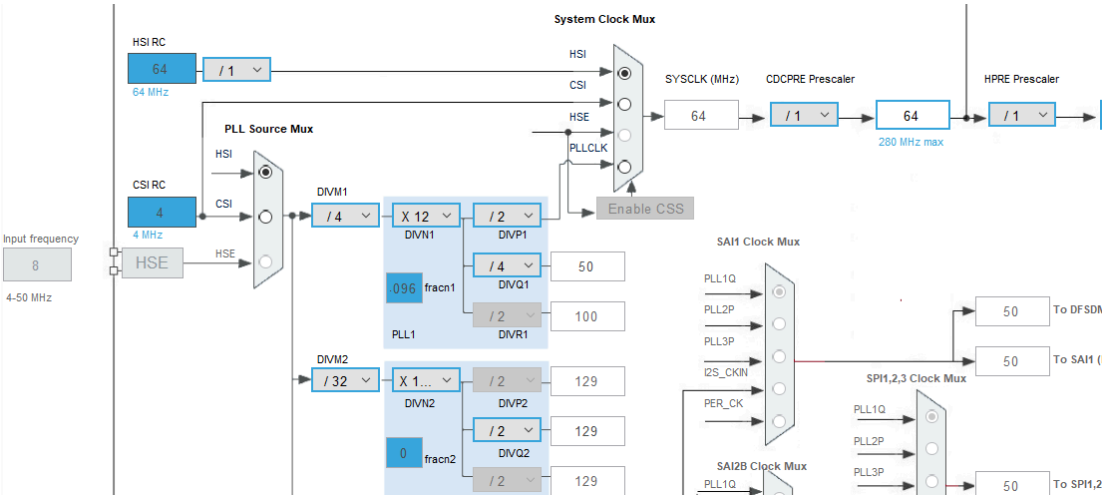
O CAN FD representa uma evolução em relação ao protocolo CAN clássico, oferecendo maior flexibilidade e velocidades de transmissão de dados. Ele é compatível com os padrões CAN versão 2.0 e CAN FD 1.0, o que garante uma transição suave para sistemas já existentes. Uma das características mais notáveis 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.

Figure 705. FDCAN block diagram



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.

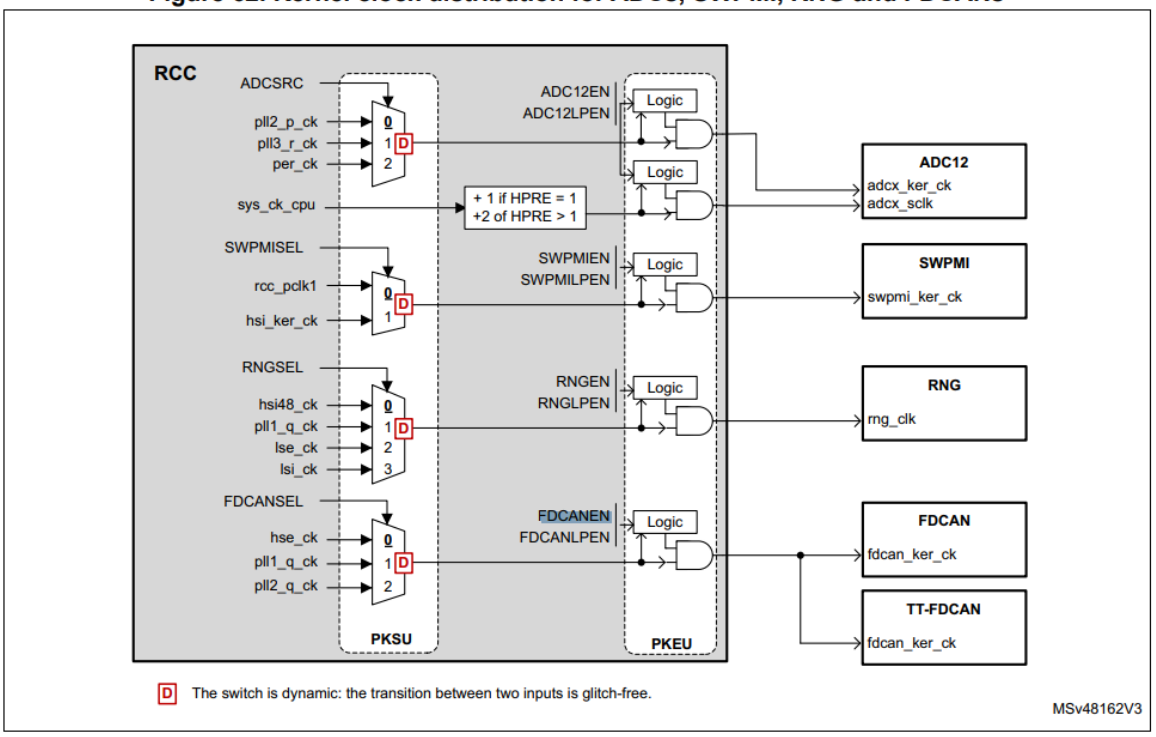


Por exemplo, se `pll1_q_ck` for utilizado, é necessário habilitá-lo por meio do *bit* RCC_PLLCFGR_DIVQ1EN. A frequência de `pll1_q_ck` pode ser configurada no editor

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".

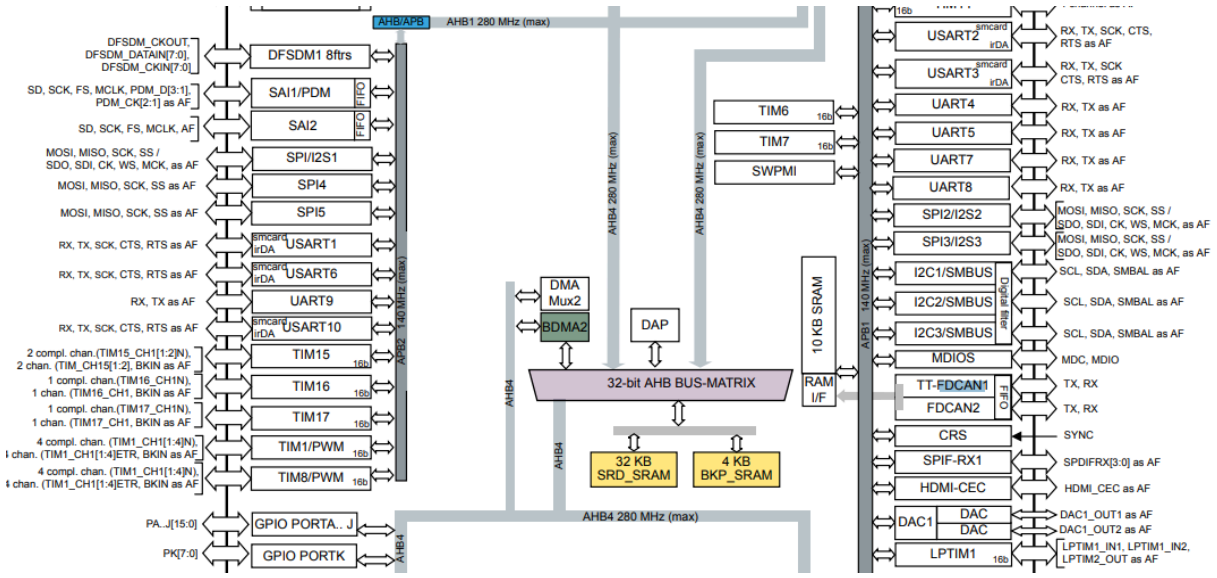
$$pll1\_q\_ck = \frac{Fonte}{DIVM1} \times \frac{\left(DIVN1 + \frac{fracn1}{2^{13}}\right)}{DIVQ1}$$

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*.

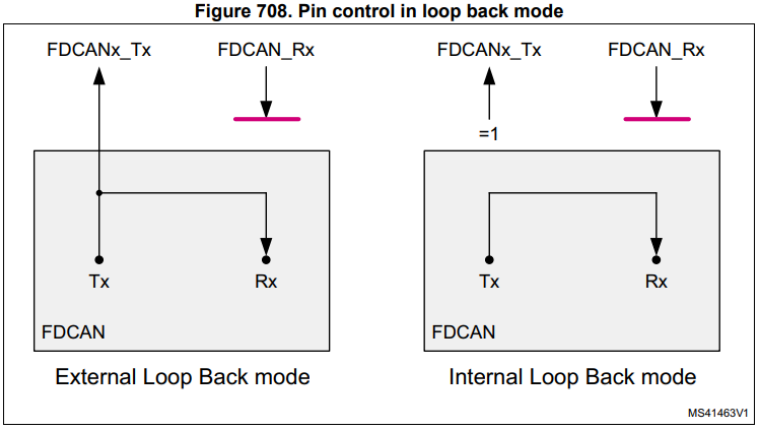
O outro relógio é o relógio da interface de barramento fdcan_pclk para os periféricos (em inglês, *peripheral*), como mostra o excerto do diagrama de blocos do STM32H7A3. O *clock gating* para o FDCAN é habilitado pelo *bit* RCC_APB1HENR_FDCANEN. A unidade de calibração de *clock* comum é opcional e 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.

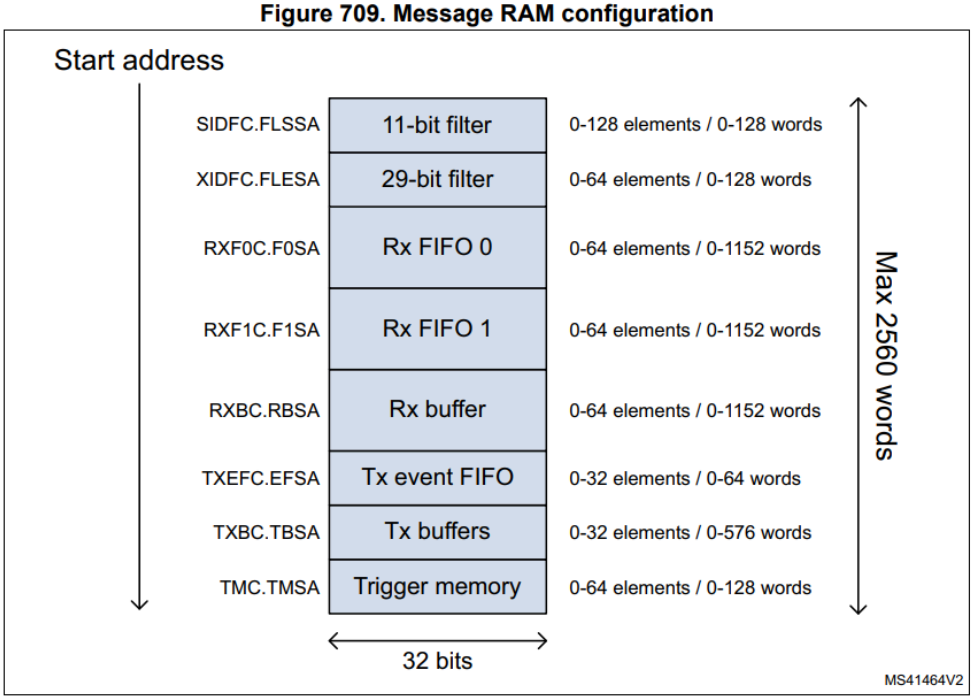
Boundary address	Peripheral
0x4000AC00 - 0x4000D3FF	CAN Message RAM
0x4000A800 - 0x4000ABFF	CAN CCU
0x4000A400 - 0x4000A7FF	FDCAN
0x4000A000 - 0x4000A3FF	TT-FDCAN

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.

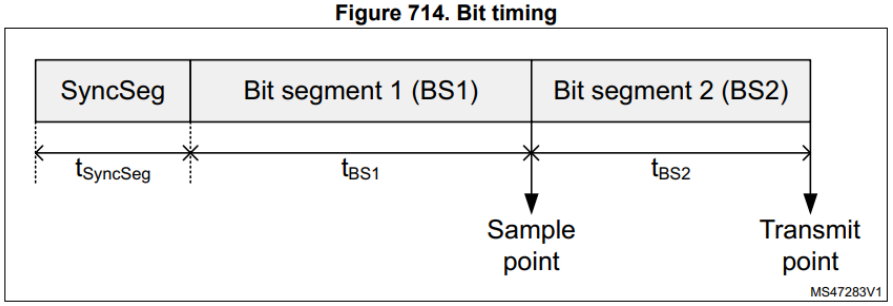
Para transferência de dados, distinguem-se três formatos de quadro, configuráveis pelos *bits* `FDCAN_CCCR_BRSE` e `FDCAN_CCR_FDOE`:

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.