JEFFERSON WILHELM MEYER SOARES

ATV 10 — Da UART para a rede CAN

Da UART para a rede CAN: ESP32, DHT11 e o transceiver WCMCU-230

Nesta prática, dois ESP32 trocam a leitura de um DHT11, primeiro por um fio de UART e depois por uma rede CAN. A informação é a mesma nos dois casos, temperatura e umidade; o que muda é o caminho e as regras do caminho. Em vez de ver a leitura pronta na tela, vocês vão ver os bytes que chegam e descobrir, a partir deles, o que foi enviado.

Os dois ESP32 têm papéis fixos. O **transmissor** é o que tem o DHT11 ligado: lê a temperatura e a umidade e envia as leituras. O **receptor** mostra no monitor serial exatamente o que chega, byte a byte, em hexadecimal. O **sensor** é só o DHT11.

Cada ESP32 pode ficar ligado por USB ao seu próprio computador, ou os dois ao mesmo. Com dois computadores, o terra das placas não passa mais pelo USB, e o fio de GND entre os dois ESP32 é o que dá a referência comum para os sinais. Ele aparece em todas as montagens e nunca sai.

Material:

- 2 ESP32 DevKit V1
- 2 módulos transceiver CAN WCMCU-230 (SN65HVD230)
- 1 DHT11
- fios, protoboard e multímetro
- o firmware, no anexo `atv10-firmware.zip` deste material: a pasta `10 - firmware`, um só código com quatro modos de gravação, `transmissor-uart`, `receptor-uart`, `transmissor-can` e `receptor-can`, que abre no Arduino IDE ou no PlatformIO

O firmware vem pronto. Ninguém programa nesta prática: nos modos CAN, o arquivo `protocolo.h` guarda só a configuração da rede (identificadores, posição dos bytes, taxa e filtro), e o que se faz nele é trocar valores. O monitor serial trabalha a 115200 bps nos dois ESP32.

Com os dois ESP32 no mesmo computador, cada um aparece numa porta, `/dev/ttyUSB0` e `/dev/ttyUSB1` no Linux ou `COM3`, `COM4` no Windows. A ordem depende de qual USB foi ligado primeiro e muda quando se religa uma placa. Antes de cada gravação, confira qual porta é a do ESP32 do DHT11.

Gravar pelo Arduino IDE. Instale, uma vez, o pacote de placas **esp32** da Espressif (Ferramentas > Placa > Gerenciador de placas) e as bibliotecas **DHT sensor library** e **Adafruit Unified Sensor**, da Adafruit (Gerenciador de bibliotecas). Abra `10 - firmware/atv10/atv10.ino`; o `protocolo.h` abre junto, numa aba. Escolha a placa **ESP32 Dev Module** e a porta do ESP32 que vai gravar. O modo de gravação é escolhido no topo do `atv10.ino`: deixe ativa só a linha do modo, por exemplo `#define TRANSMISSOR_UART`, com as outras três começando por `//`, e clique em Carregar. Para gravar o outro ESP32, troque a linha ativa, troque a porta e carregue de novo. O monitor serial fica em Ferramentas > Monitor serial, a 115200.

Gravar pelo PlatformIO. Abra a pasta `10 - firmware`. Cada modo é um ambiente, e a linha do topo do `atv10.ino` não vale: `pio run -e transmissor-uart -t upload --upload-port /dev/ttyUSB0` grava e `pio device monitor -p /dev/ttyUSB0` abre o monitor serial.

Nos dois casos, "grave `receptor-can`" quer dizer gravar com o modo `RECEPTOR_CAN`, e o `protocolo.h` é um só para os dois ESP32: depois de mudar um valor nele, grave de novo cada ESP32 que o passo pede.

A leitura por UART

Na UART, o transmissor escreve a leitura como uma linha de texto e manda essa linha caractere por caractere por um fio, do pino TX2 de um ESP32 ao RX2 do outro. Cada caractere viaja como um byte, e o valor de cada byte vem da tabela ASCII: a letra `A` é o byte 41, o algarismo `0` é o byte 30, tudo em hexadecimal. A linha termina com um caractere de quebra de linha, e é por ele que o receptor sabe onde uma leitura acaba e a outra começa.

É uma ligação ponto a ponto: um fio de ida, um transmissor e um receptor. Não há endereço nem confirmação de recebimento.

A tabela ASCII abaixo traz os caracteres que podem aparecer nesta prática, cada um com o seu byte em hexadecimal.

Byte	Caractere	Byte	Caractere	Byte	Caractere	Byte	Caractere
0A	quebra de linha (\n)	34	4	44	D	4F	O
20	espaço	35	5	45	E	50	P
25	%	36	6	46	F	51	Q

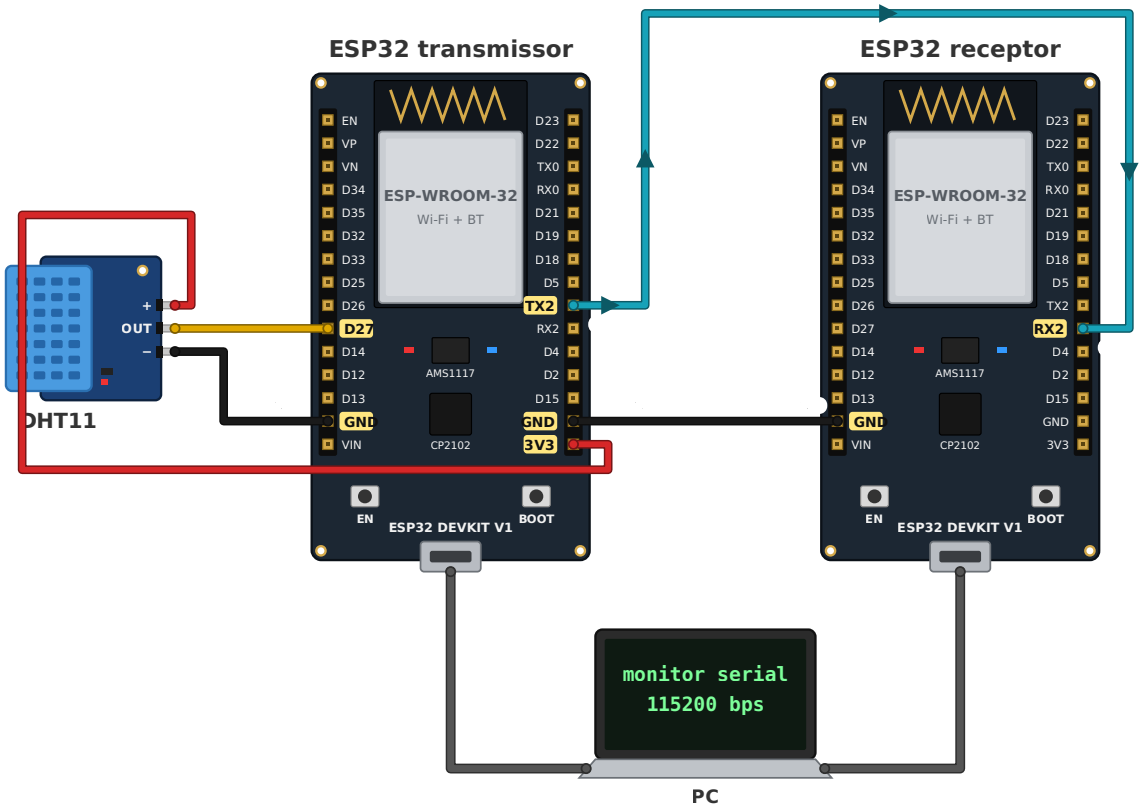
Byte	Caractere	Byte	Caractere	Byte	Caractere	Byte	Caractere
2C	,	37	7	47	G	52	R
2D	-	38	8	48	H	53	S
2E	.	39	9	49	I	54	T
30	0	3A	:	4A	J	55	U
31	1	3B	;	4B	K	56	V
32	2	3D	=	4C	L	57	W
33	3	41	A	4D	M	58	X
.	.	42	B	4E	N	59	Y
.	.	43	C	.	.	5A	Z

Question 1.

Resolvemos juntos. Um receptor de UART mostrou esta linha:

```
Recebido: 54 3D 31 39 3B 55 3D 36 35 0A
```

- Texto da linha, sem a quebra de linha: _____
- Temperatura (°C): _____
- Umidade (%): _____
- Bytes na linha, contando a quebra de linha: _____



1. Monte a ligação da figura.
2. Grave `transmissor-uart` no ESP32 do DHT11 e `receptor-uart` no outro.
3. Abra o monitor serial do receptor. A cada 2 s chega uma linha de bytes. Anote três linhas e, com a tabela ASCII, descubra o texto, a temperatura e a umidade de cada uma.

Question 2.

Três linhas no monitor serial do receptor.

	Bytes que o receptor mostrou (hex)	Quantidade de bytes	Texto da linha	Temperatura (°C)	Umidade (%)
Linha 1					
Linha 2					
Linha 3					

1. Desconecte o fio de dados do DHT11 e decodifique a linha que passa a chegar. Depois, reconecte o fio.

Question 3.

Com o fio de dados do DHT11 desconectado.

Bytes que o receptor mostrou (hex)	Quantidade de bytes	Texto da linha

1. Abra o monitor serial do transmissor. Ele mostra só `Leitura enviada` a cada envio. Tire o USB do receptor e veja o que muda no transmissor. Depois, religue o receptor.

Question 4.

Com o USB do receptor desligado, o transmissor da UART percebe que ninguém recebeu a linha.

( ) True

( ) False

Question 5.

Agora é com você. Num dia frio, um receptor de UART mostrou:

```
Recebido: 54 3D 2D 34 3B 55 3D 39 0A
```

- Temperatura (°C): _____
- Umidade (%): _____
- Bytes na linha, contando a quebra de linha: _____

Do fio ao barramento CAN

No CAN, os ESP32 não se ligam um ao outro: cada um se liga a um **barramento**, um par de fios trançados chamado CANH e CANL, e vários nós podem ficar pendurados no mesmo par.

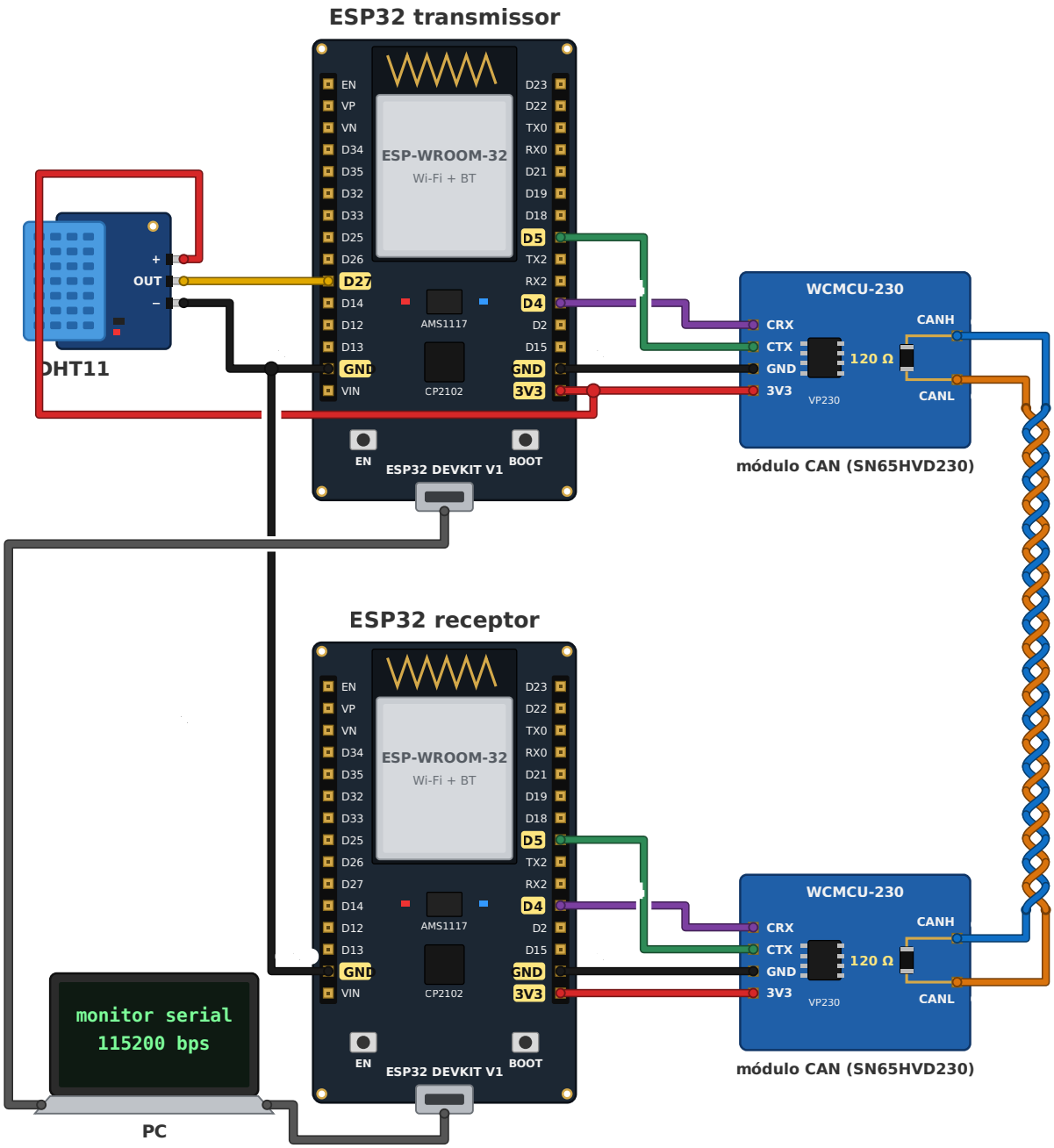
Quem conversa com o par é o **transceiver**, aqui o WCMCU-230. Dentro do ESP32 há um controlador CAN, que monta e confere os quadros; ele fala com o transceiver por dois pinos, CTX (o que o ESP32 quer pôr no barramento) e CRX (o que está no barramento). O transceiver transforma isso em tensão diferencial no par:

- **bit recessivo (1)**: o transceiver solta o par, e CANH e CANL ficam na mesma tensão, por volta de 2,3 V;
- **bit dominante (0)**: o transceiver puxa CANH para cima e CANL para baixo, com uns 2 V de diferença entre eles.

Quem lê olha só a diferença entre os dois fios. Um ruído que entra igual nos dois some na subtração, e é por isso que o par é trançado.

Se dois nós transmitem ao mesmo tempo, o dominante vence: basta um nó puxar o par para ele ficar dominante. Todo nó lê o barramento enquanto transmite e percebe quando o que lê é diferente do que mandou.

Nas duas pontas do par vai um resistor de **120 Ω** entre CANH e CANL, a terminação. Ele evita que o sinal reflita na ponta do cabo e é ele também que traz o par de volta ao recessivo quando ninguém puxa. O WCMCU-230 já traz esse resistor na placa.



O WCMCU-230 é alimentado pelo **3V3** do ESP32, o GPIO 5 vai no **CTX** e o GPIO 4 no **CRX**. Com CTX e CRX trocados, nada chega ao barramento e o controlador só vê erro.

1. Retire o fio da UART e monte a ligação da figura.
2. Com os dois ESP32 desligados, meça a resistência entre CANH e CANL.

Question 6.

Resistência entre CANH e CANL, com os dois ESP32 desligados.

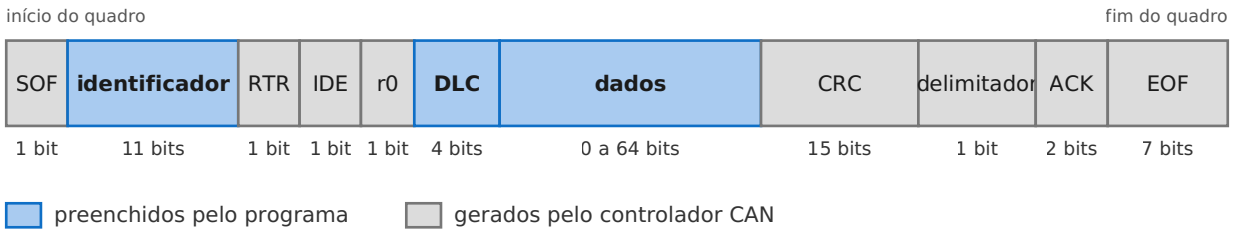
	Resistência (Ω)
Com os dois módulos ligados ao par	
De um módulo sozinho	

O quadro CAN

No CAN, a informação viaja em **quadros**. Cada quadro leva um **identificador** de 11 bits, que diz o que é aquela mensagem, um **DLC**, que diz quantos bytes de dados vêm, de 0 a 8, e os **dados**. Não há endereço de destino: o quadro vai para o barramento todo, e cada nó decide pelo identificador se aquela mensagem interessa.

O identificador também é a prioridade. Na disputa pelo barramento, os nós transmitem o identificador bit a bit; quem manda um recessivo e lê um dominante perde e para. Por isso, quanto menor o identificador, maior a prioridade.

O resto do quadro o controlador CAN monta sozinho: o início (SOF), o CRC para conferir os bits, o campo de ACK e o fim (EOF). A configuração só escolhe o identificador e onde vai cada informação nos dados.



O campo de **ACK** é a confirmação: o transmissor deixa esse bit recessivo, e qualquer nó que recebeu o quadro sem erro o puxa para dominante enquanto ele passa. Se ninguém puxa, o transmissor sabe que ninguém recebeu.

O gráfico mostra quanto tempo um quadro ocupa o barramento, pela taxa e pelo número de bytes de dados.

```
import matplotlib.pyplot as plt


def br(valor, casas=0):
    return f"{valor:.{casas}f}".replace(".", ",")
```

```
campos = [("SOF", 1), ("identificador", 11), ("RTR, IDE, r0", 3), ("DLC", 4),
("dados", 8 * int(dlc)), ("CRC", 16), ("ACK", 2), ("EOF", 7), ("intervalo",
3)]
programa = {"identificador", "DLC", "dados"}
bit_us = 1000 / taxa
fig, eixo = plt.subplots(figsize=(11, 2.6))
inicio = 0
for nome, bits in campos:
    if bits == 0:
        continue
    eixo.barh(0, bits * bit_us, left=inicio * bit_us, color="C0" if nome in
programa else "C7", edgecolor="white")
    if bits >= 4:
        eixo.text((inicio + bits / 2) * bit_us, 0, nome, ha="center",
va="center", color="white", fontsize=9)
    inicio += bits
eixo.set_yticks([])
eixo.set_xlabel("tempo (µs)")
eixo.set_title(f"um quadro com {int(dlc)} byte(s) de dados a {int(taxa)}
kbit/s; em azul, o que o programa preenche")
fig.tight_layout()

print(f"Duração de um bit: {br(bit_us, 2)} µs")
print(f"Quadro: {inicio} bits, {br(inicio * bit_us)} µs, sem contar os bits
de enchimento")
print(f"Dados: {8 * int(dlc)} de {inicio} bits ({br(100 * 8 * int(dlc) /
inicio)} %)")
```

Interactive code: run it in the online version of the material.

O arquivo `protocolo.h` é a configuração da rede:

```
const uint32_t ID_LEITURA = 0x000;  // identificador da mensagem com a
leitura do DHT11
const uint32_t ID_FALHA = 0x000;    // identificador da mensagem de falha do
DHT11
const int BYTE_TEMPERATURA = 0;     // posição da temperatura nos dados do
quadro, de 0 a 7
const int BYTE_UMIDADE = 1;         // posição da umidade nos dados do
quadro, de 0 a 7
const int TAXA_KBPS = 500;          // 125, 250, 500 ou 1000
const uint32_t FILTRO_ID = 0;       // 0 aceita todos os identificadores
```

Transmissor e receptor usam o mesmo arquivo. O transmissor mostra no monitor serial cada quadro que envia, e o receptor, cada quadro que recebe, os dois no mesmo formato: `id=0x100 dlc=2 dados=1B 58`.

Question 7.

Resolvemos juntos. Um receptor CAN mostrou `Recebido: id=0x2A0 dlc=3 dados=05 1F C8`.

	Valor
Identificador, em hexadecimal	
Bytes de dados	
Último byte de dados, em hexadecimal	
Bits só de dados	

1. No `protocolo.h`, dê um identificador à leitura, por exemplo `ID_LEITURA = 0x100`.
2. Grave `transmissor-can` no ESP32 do DHT11 e `receptor-can` no outro. O DHT11 precisa estar ligado: se a leitura dele falha, o transmissor manda outro quadro, o de falha, que fica para a próxima parte.
3. Abra o monitor serial dos dois e anote um quadro de cada lado.

Logo depois de gravar, o `erros_tx` do transmissor pode aparecer alto e ir caindo de 1 em 1. Enquanto um ESP32 está sendo gravado, ninguém confirma os quadros do outro, e cada quadro sem confirmação conta como erro. A última parte explica esse contador.

Question 8.

Um quadro nos dois monitores seriais.

	Identificador (hex)	DLC	Dados (hex)
Enviado pelo transmissor			
Recebido pelo receptor			

Question 9.

Agora é com você. O transmissor manda um quadro com `id=0x100`. Para quem vai esse quadro?

( ) Só para o receptor cujo endereço é 0x100.

( ) Para o primeiro nó que confirmar o quadro com o ACK.

( ) Para todos os nós do barramento, e cada um decide pelo identificador se aquela mensagem interessa.

( ) Para o nó de menor identificador do barramento.

A leitura do DHT11 dentro do quadro

Na UART, a leitura ia como texto, um caractere por byte. No CAN, cada byte de dados é um número de 0 a 255, e a temperatura e a umidade vão como números, cada uma num byte. Quem define em que byte vai cada informação é a configuração: `BYTE_TEMPERATURA` e `BYTE_UMIDADE`. Transmissor e receptor precisam usar a mesma definição, senão o receptor lê a umidade no lugar da temperatura.

O DLC é o número de bytes até a última posição usada: com a temperatura no byte 0 e a umidade no byte 1, o quadro leva 2 bytes. Uma posição que fica no meio sem uso viaja mesmo assim, com o valor 00.

A temperatura pode ser negativa, e por isso vai num byte com sinal: de 00 a 7F são os números de 0 a 127, e de 80 a FF são os negativos, contados a partir de 256. O byte FF é −1, o FE é −2, e assim por diante. A umidade vai de 0 a 100 e não precisa de sinal.

Question 10.

Resolvemos juntos. Com a temperatura no byte 0 e a umidade no byte 1, um receptor mostrou `id=0x100 dlc=2 dados=17 41`.

- Temperatura (°C): _____
- Umidade (%): _____

1. Com a configuração que vem no arquivo, temperatura no byte 0 e umidade no byte 1, anote três quadros recebidos e converta os dados de hexadecimal para decimal.

Question 11.

Três quadros recebidos.

	Dados (hex)	Temperatura (°C)	Umidade (%)
Quadro 1			
Quadro 2			
Quadro 3			

Question 12.

Bytes que uma leitura de temperatura e umidade ocupa.

	Bytes por leitura
Na UART, com a quebra de linha	
No CAN, só os dados	

1. No `protocolo.h`, ponha a umidade no byte 0 e a temperatura no byte 3: `BYTE_UMIDADE = 0` e `BYTE_TEMPERATURA = 3`. Grave o `transmissor-can` e anote um quadro no receptor.

Question 13.

Um quadro com a umidade no byte 0 e a temperatura no byte 3.

DLC	Byte 0	Byte 1	Byte 2	Byte 3

1. Volte a configuração para `BYTE_TEMPERATURA = 0` e `BYTE_UMIDADE = 1` e grave o `transmissor-can`.

Question 14.

Agora é com você. Com a temperatura no byte 0 e a umidade no byte 1, um receptor mostrou `dados=FB 5A`.

- Temperatura (°C): _____
- Umidade (%): _____

Outra rede usa a temperatura no byte 0 e a umidade no byte 2. Uma leitura de 21 °C e 60 % sai nela com:

- DLC: _____
- Dados, em hexadecimal, separados por espaço: _____

A mensagem de falha e o filtro

Cada mensagem diferente ganha seu próprio identificador. Quando a leitura do DHT11 falha, o transmissor manda um quadro com o identificador `ID_FALHA` e nenhum byte de dados, DLC 0: o identificador sozinho já diz o que aconteceu.

Como o identificador é a prioridade, a escolha do número tem consequência. Se a falha e a leitura de outro nó disputarem o barramento no mesmo instante, passa primeiro a de menor identificador, e a outra espera e tenta de novo logo em seguida.

O controlador CAN do receptor pode ter um **filtro de aceitação**: ele compara o identificador de cada quadro com o filtro e descarta, no próprio hardware, o que não interessa, sem o programa ver. Com `FILTRO_ID = 0`, o receptor aceita tudo; com um identificador em `FILTRO_ID`, só aceita aquele.

Question 15.

Resolvemos juntos. Dois nós começam a transmitir no mesmo instante, um com o identificador 0x080 e outro com 0x100. Em binário, de 11 bits, eles são 000 1000 0000 e 001 0000 0000. Qual quadro passa?

( ) O 0x100, porque é o número maior.

( ) O 0x080, porque no primeiro bit em que os dois diferem ele manda dominante (0) e o outro manda recessivo (1) e desiste.

( ) Nenhum, porque os dois quadros se destroem e os dois nós repetem.

( ) O que começou a transmitir alguns microssegundos antes.

1. No `protocolo.h`, dê um identificador à falha em `ID_FALHA`, decidindo se ele fica maior ou menor que o da leitura.
2. Grave `transmissor-can` e `receptor-can` e desconecte o fio de dados do DHT11.
3. Reconecte o DHT11. No `protocolo.h`, faça `FILTRO_ID = ID_LEITURA` e grave só o `receptor-can`.
4. Desconecte de novo o fio de dados do DHT11 e veja o que o receptor mostra.

Question 16.

O que o monitor serial do receptor mostra.

	Receptor sem filtro	Receptor com filtro
DHT11 ligado		
DHT11 desconectado		

Question 17.

Qual identificador vocês escolheram para a falha, e por que ele faz sentido em relação ao da leitura? Por que, com o filtro, os quadros de falha não aparecem no receptor?

Question 18.

Agora é com você. O receptor está com `FILTRO_ID = ID_FALHA`, a falha em 0x080 e a leitura em 0x100, e o DHT11 está ligado e lendo bem. O que o monitor serial do receptor mostra?

( ) As leituras, normalmente, a cada 2 s.

( ) As leituras e, de vez em quando, um quadro de falha.

( ) Nada, porque só passam quadros 0x080 e, com o sensor bem, o transmissor só manda quadros 0x100.

( ) Uma mensagem de erro do filtro a cada quadro descartado.

Erros no barramento

Todo nó CAN confere cada quadro, e quem encontra um erro avisa o barramento todo na hora, com seis bits dominantes seguidos que destroem o quadro. O transmissor percebe e repete o quadro sozinho, sem o programa pedir.

Para um nó com defeito não travar a rede repetindo erros para sempre, cada nó conta seus erros. O **contador de erros de transmissão** sobe 8 a cada quadro que falha e desce 1 a cada quadro que dá certo, e pelo valor do contador o nó muda de estado:

- **ativo**, abaixo de 128: o nó funciona normalmente e avisa os erros que vê;
- **passivo**, de 128 a 255: o nó continua transmitindo, mas não destrói mais os quadros dos outros;
- **bus-off**, acima de 255: o nó se desliga do barramento e para de transmitir.

Um quadro que fica sem ACK também conta como erro, mas a norma não deixa esse caso, sozinho, levar o nó do passivo ao bus-off: um nó que não tem com quem falar fica esperando, sem sair da rede.

Neste firmware, o transmissor sai do bus-off sozinho: espera o barramento ficar livre por um tempo, passa por **recuperando** e volta a **ativo** com o contador zerado. Os quadros que ainda não saíram esperam numa **fila** de envio, que tem lugar para seis.

O gráfico simula o contador de um nó que transmite muitos quadros, uma parte deles com erro.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
gerador = np.random.default_rng(10)
contador, historico, saiu = 0, [], None
for k in range(int(quadros)):
    if gerador.random() < erro / 100:
        contador += 8
    else:
        contador = max(0, contador - 1)
    historico.append(contador)
    if contador > 255:
        saiu = k + 1
        break
fig, eixo = plt.subplots(figsize=(11, 4))
eixo.axhspan(0, 128, color="C2", alpha=0.12, label="ativo")
eixo.axhspan(128, 256, color="C1", alpha=0.15, label="passivo")
eixo.axhspan(256, 300, color="C3", alpha=0.15, label="bus-off")
eixo.plot(range(1, len(historico) + 1), historico, color="C0", linewidth=1.5)
eixo.set_xlim(0, quadros)
eixo.set_ylim(0, 300)
eixo.set_xlabel("quadros transmitidos")
eixo.set_ylabel("contador de erros de transmissão")
eixo.legend(loc="upper left", fontsize=9)
eixo.grid(True)
fig.tight_layout()

saldo = 8 * erro / 100 - (1 - erro / 100)
print(f"Saldo médio por quadro: {saldo:+.2f}".replace(".", ","))
print(f"Bus-off no quadro {saiu}" if saiu else f"Contador no fim:
{historico[-1]}")
```

Interactive code: run it in the online version of the material.

O monitor serial do transmissor mostra, a cada envio, o retorno de `twai_transmit` (`transmit`), os quadros na fila (`na_fila`, que conta também o quadro que acabou de entrar), o contador de erros de transmissão (`erros_tx`) e o estado do controlador (`estado`).

1. No `protocolo.h`, volte `FILTRO_ID = 0` e grave o `receptor-can`.
2. Preencha a tabela pelo monitor serial do transmissor. Para desligar o receptor, tire o USB dele.
3. Para a linha dos 250 kbit/s, faça `TAXA_KBPS = 250` e grave só o `receptor-can`. Para a última linha, volte `TAXA_KBPS = 500` e grave só o `receptor-can` de novo.

Question 19.

O monitor serial do transmissor em cada situação.

	transmit	na_fila	erros_tx	estado
Receptor ligado				
Receptor desligado, primeiros envios				
Receptor desligado há 30 s				
Receptor religado há 10 s				
Receptor a 250 kbit/s				
Receptor de volta a 500 kbit/s há 10 s				

Question 20.

Sem o receptor, o contador de erros do transmissor parou em 128 e ele não saiu da rede. Com o receptor a 250 kbit/s, o transmissor foi a bus-off. Explique a diferença entre os dois casos e compare com o que o transmissor da UART percebeu quando o receptor foi desligado.