Protocolos de comunicação

Protocolo de comunicação é um conjunto de regras que permite que dois ou mais dispositivos comuniquem entre si, para tal os dispositivos têm de conhecer as regras do protocolo de comunicação. O protocolo de comunicação define quantos dispositivos podem comunicar, se necessitam de identificadores únicos ou não, se são síncronos ou assíncronos, se são unidirecionais ou bidirecionais e se funcionam com um dispositivo que "manda", *master*, nos outros dispositivos, funcionando os últimos como *slaves*. Exemplos de protocolos de comunicação são UART, I2C, SPI e CAN, sendo estes os que vamos dar mais relevância na disciplina de Sistemas Embebidos.

Comunicação síncrona vs assíncrona

Comunicação síncrona em regra geral é uma comunicação mais rápida, necessita de algum tipo de sincronismo (normalmente usa um *clock* para garantir que todos os dispositivos estão sincronizados).

Comunicação assíncrona todos os dispositivos podem comunicar quando necessitarem, não necessitam de um *clock* para comunicarem.

Comunicação série vs paralela

Comunicação série permite a trocas de dados de uma forma sequencial, ou seja, a informação flui *bit* a *bit* não necessitando de uma linha específica por *bit*. Ao contrário da **comunicação paralela** que necessita de uma linha específica para transferir a informação por *bit.*

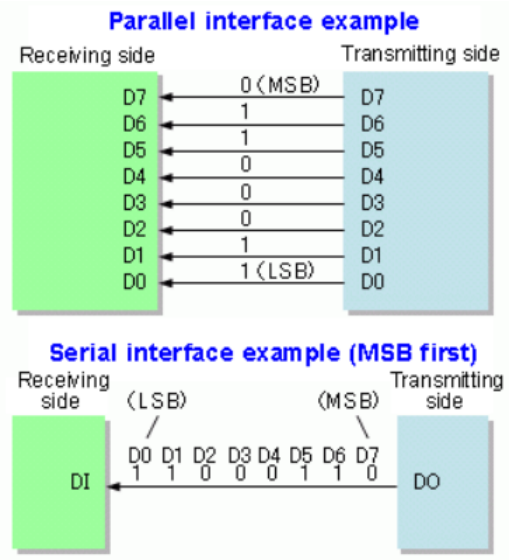


Figura 1

UART

Universal Synchronous Asynchronous Receiver Transmitter, UART, é um protocolo de comunicação do tipo série, assíncrono. A informação flui por duas linhas: TX, transferência de informação, e RX, receção de informação, permitindo assim a informação fluir nos dois sentidos em simultâneo, *full-duplex.* A velocidade de comunicação dos dispositivos é acordada entre ambos e é denominada por *baudrate.*

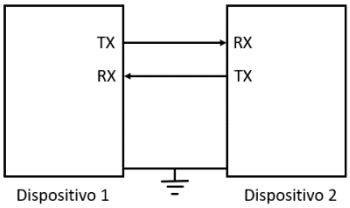


Figura 2

A trama de comunicação é constituída por: bit de começo, *start bit,* informação e *stop* bit.

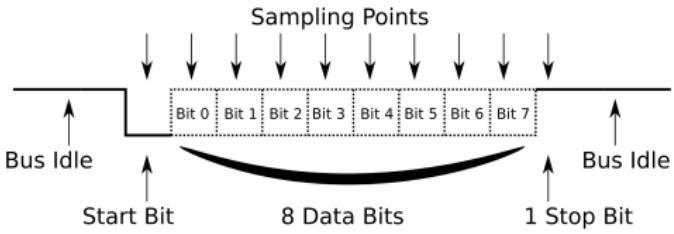


Figura 3

SPI

Serial Peripheral Interface, SPI, é um protocolo de comunicação síncrono, ou seja, necessita de um sinal relógio para efetuar a comunicação. Possui um dispositivo que funcionará com *master* que é responsável por geral o *clock* que comandará os restantes dispositivos, *slaves*. Para a comunicação os dispositivos necessitam de quatro linhas: MOSI, *Master Output Slave Input*, MISO, *Master Input Slave Output*, o SCLK, *clock*, e o CS, *Chip Select,* que permite selecionar o dispositivo que o *master* pretende comunicar. Uma vez que usa duas linhas de comunicação, uma para transmitir e outra para receber informação, o SPI também é *full-duplex.*

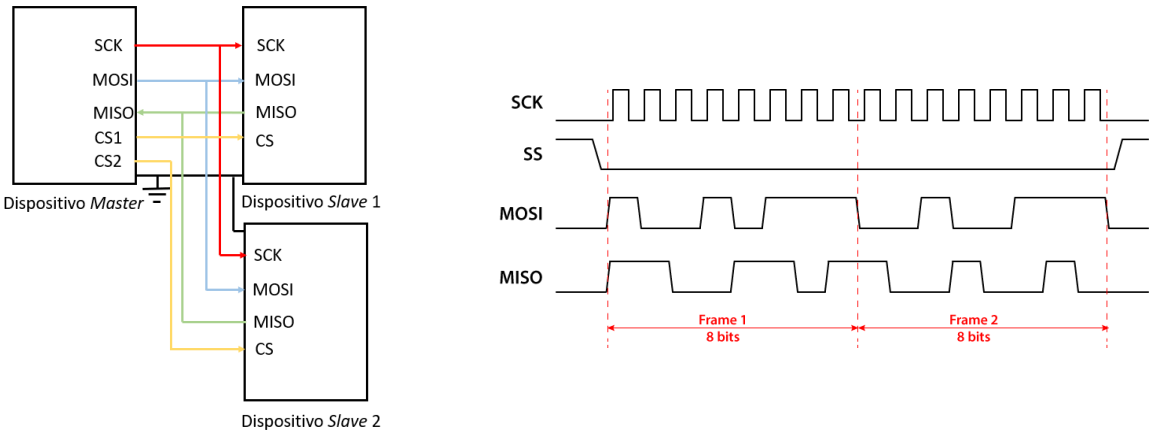


Figura 4

I²C

Inter-Integrated Circuit, I²C, é um protocolo de comunicação síncrono que apenas necessita de duas linhas para o seu funcionamento, o SDA, *Serial Data,* por onde flui a informação e o SCL, *Serial Clock,* que permite sincronizar os diversos dispositivos. Como no SPI o responsável pela geração do sinal de relógio é o dispositivo *master*, no entanto os dispositivos *slaves* podem forçar o relógio ao nível lógico de zero para parem a comunicação por breves instantes, estando definida esta operação no protocolo como *clock stretching*. O I²C tens vantagens sobre o UART e o SPI, em comparação com o UART para a comunicação funcionar ambos os dispositivos necessitam de estar configurados previamente para uma determina frequência de comunicação, *baudrate,* o que faz com que ambos os dispositivos disponham de *clocks* próprios e não os partilhem entre si. Em comparação com o SPI o I²C permite que no barramento existam mais *masters* ao contrário do SPI que apenas permite um. O I²C permite ainda conectar aos dispositivos por um endereço, ou seja, cada dispositivo *slave* possui um ID único de 7bits evitando assim a ligação física de um pino específico a cada dispositivo, como no caso do SPI.

Ao contrário do SPI e UART as linhas SDA e SCL têm resistência de *pull-up* o que faz com que os dispositivos apenas consigam "puxar" a linha ao sinal lógico de zero, permitindo assim evitar dissipação de energia desnecessária e dando visibilidade aos outros dispositivos no barramento que está a decorrer uma comunicação.

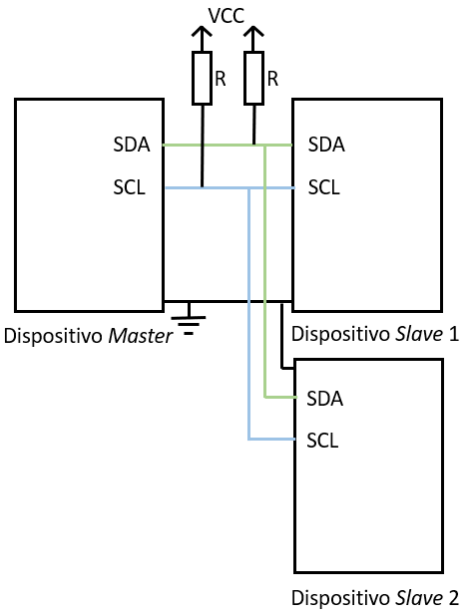


Figura 5

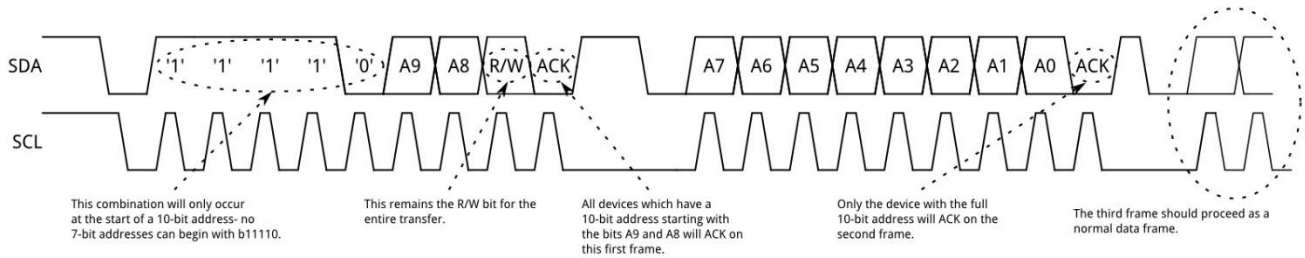


Figura 6

CAN

CAN é um protocolo de comunicação que foi criado para a indústria automóvel em 1986 por Robert Bosch, sendo utilizado hoje em dia em diferentes indústrias. O principal objetivo deste protocolo era o de simplificar as complexas cablagens que permitiam a comunicação entre os diversos micro-controladores (nós) existentes nos componentes dos veículos, permitindo uma monitorização e controlo dos mesmos. Ao longo dos anos tem vindo a verificar-se uma crescente utilização destes micro-controladores na indústria automóvel, sendo o protocolo CAN amplamente utilizado e assunto de standards internacionais aprovados pela ISO (International Standard Organization).

No protocolo de comunicação CAN existem dois formatos distintos usados para a estrutura das mensagens: Standard CAN (version 2.0A) e Extended CAN (version 2.0B). Estes dois apenas diferem num dos campos da estrutura das tramas: o identificador (11 bits no **Standard CAN** e 29 bits no **Extended CAN**). Independentemente do formato usado, apenas existem 4 tipos de mensagens diferentes: *Data Frame, Remote Frame, Error Frame* e *Overload Frame*. Como cada uma destas mensagens tem uma estrutura única, é necessária uma análise detalhada de cada campo. No entanto, neste módulo vamos apenas dar enfâse às *Data frames* e a imagem que se segue esquematiza os diferentes campos de uma típica trama de dados.

Nota: Sugere-se a pesquisa da estrutura e utilidade dos outros tipos de tramas.

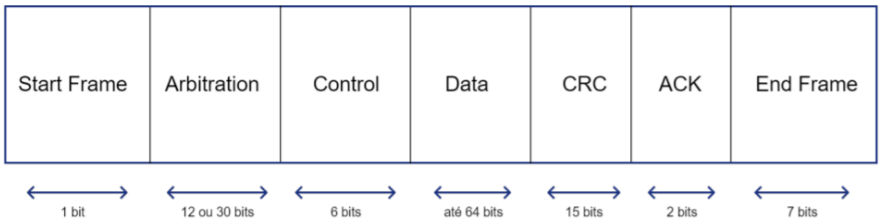


Figura 7

A implementação do protocolo CAN engloba ainda processos de codificação (*Bit Stuffing*), sincronização e mecanismos complexos de deteção e controlo de erros. Por essa razão surgiram standards do protocolo CAN que se focam em níveis superiores do protocolo, não precisando o programador de lidar com este tipo de tarefas executadas num nível inferior do modelo *Open System Interconnection*. Alguns exemplos de standards CAN direcionados para a indústria automóvel são: CANopen, CANfd, Device Net, FTT-CAN, entre outros. Contudo, cada um destes protocolos poderá ser estendido e tornando-se específico de um fabricante, como é o caso do standard CAN GMLAN, específico do fabricante General Motors.

Independentemente standard utilizado, a comunicação CAN é feita através de dois fios (CANH e CAN-L) que são conectados a todos os dispositivos. Os sinais têm a mesma sequência de dados, mas amplitude oposta. Desta maneira o circuito fica mais imune a interferências eletromagnéticas e a probabilidade de se corromper dados é menor. Como a Figura representa, se um pulso for enviado pelo fio CAN-H, este apresentará uma voltagem entre 2.5V e 3.75 V. Caso seja enviado pelo CAN-L, este apresenta uma amplitude entre 2.5V e 1.25V. Para efeitos de nomenclatura, um bit com tensão igual 2.5V, é um bit 0 (*dominant bit*) e um bit com tensão igual a 0 é bit 1 (*recessive bit*).

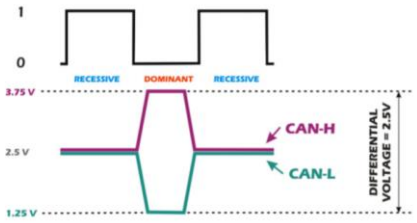


Figura 8

Relativamente à velocidade e cablagem existem quatro standards, sendo que só iremos abordar **High SpeedCAN (ISO 11898-2).** Este permite baud rates compreendidas entre 40 Kbit/s e 1 Mbit/s, dependendo do comprimento do fio. Cada uma das extremidades da rede possuí uma resistência de 120 ohm. É o standard mais utilizado devido à facilidade de cablagem. Esta topologia está representada na Figura

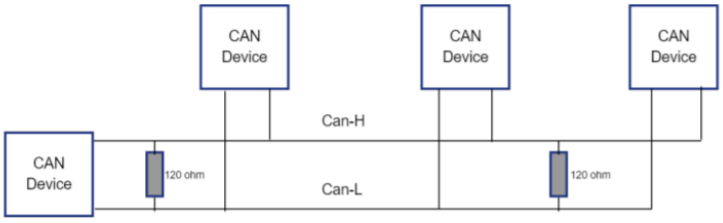


Figura 9

De modo a existir interoperabilidade, cada nó tem uma estrutura semelhante. Por essa razão, um micro-controlador e um transceiver são componentes obrigatórios. O transceiver é conectado à rede enviando os dados para o micro-controlador (e vice-versa), através de um componente do do controlador). A Figura 2.13 mostra o esquemático de um nó.

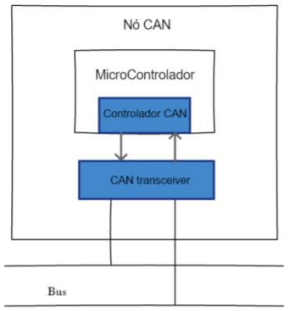


Figura 10

Exercício proposto: Desenvolva um programa que envie duas mensagens periódicas CAN (uma a cada 100 ms e outra a cada 500ms) e que reenvie para o barramento qualquer mensagem recebida. Utilize o transceiver Peak CAN para servir de segundo nó. Divida este problema em pequenos problemas de forma a tornar mais eficiente e organizado:

1. Inicialize uma estrutura para contabilizar o tempo através de uma interrupção periódica de 1ms.

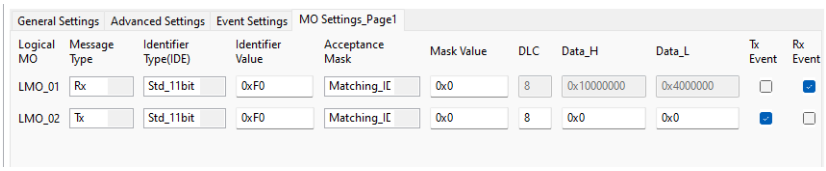
Sugestão: Utilize uma estrutura como a apresentada na imagem a baixo.

```
struct Timer_struct {
    uint16_t miliseconds; // milliseconds
    uint8_t seconds; // seconds
    uint8_t minutes; // minutes
    uint8_t hours; // hours
    uint8_t days; // days
};
struct Timer_struct Timer_struct;
```

Figura 11

2. Construa uma função **update_timer()** que será utilizada para incrementar a estrutura **Timer_struct** a cada ms.
3. Crie uma APP CAN_NODE (4.1.24) e configure-a com dois **LMOS**: Um para escrita e outro para recepção.

Sugestão: Crie um filtro que aceite todos os IDs tanto para transmissão como para recepção. Analise o seguinte artigo para uma explicação detalhada https://www.cnblogs.com/shangdawei/p/4716860.html. Para que serve esta filtragem por *hardware?* Não se esqueça de atribuir os pinos CAN da placa *platform2go* à *app*.



4. Construa agora função chamada **can_routine()** que será chamada no ciclo da função *main* a cada ms. Esta função terá como objetivo verificar o estado do timer e deve chamar uma função que transmita uma mensagem CAN. Exemplo: **void can_transmit(uint16_t id, uint8_t *data, uint8_t length);**

5. Desenvolva agora a função exemplificada acima que deverá ser invocada com os 3 elementos básicos de uma mensagem CAN: identificador da mensagem, *array* de dados e tamanho. Baseie-se nos seguintes métodos descritos no menu *App help* da *app* CAN_NODE *(4.1.24)* para construção desta função

CAN_NODE_STATUS_t CAN_NODE_MO_Transmit (const CAN_NODE_LMO_t * *lmo_ptr*)

void CAN_NODE_MO_Init (const CAN_NODE_LMO_t * *lmo_ptr*)

6. Conecte a placa *platform2go* ao *peak transceiver* e analise os resultados com o *software pcan-view.*
7. Crie agora uma interrupção e conecte-a ao pino RX da app CAN. Sempre que receber uma mensagem CAN de outro nó, esta interrupção deverá ser chamada e deverá enviar para o barramento a mensagem recebida. Baseie-se no método seguinte presente no *App help* para tratamento das mensagens recebidas CAN.

CAN_NODE_STATUS_t CAN_NODE_MO_Receive (CAN_NODE_LMO_t * *lmo_ptr*)