

## Aula 19

- *Device drivers*
- Princípios gerais
- Caso de estudo: *device driver* para uma UART
- Princípio de operação e estruturas de dados
- Funções de interface com a aplicação
- Funções de serviço de interrupções e interface com o hardware

José Luís Azevedo, Bernardo Cunha, Tomás O. Silva, P. Bartolomeu

# Introdução

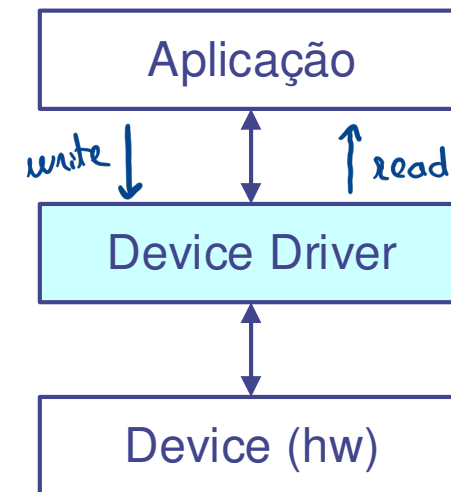
- O **número** de **periféricos** existentes é muito **vasto**:
  - Teclado, rato, placas (gráfica, rede, som, etc.), disco duro, *pen drive*, scanner, câmara de vídeo, etc.
- Estes periféricos apresentam características distintas:
  - **Operações suportadas**: leitura, escrita, leitura e escrita
  - **Modo de acesso**: carater, bloco, etc.
  - **Representação da informação**: ASCII, UNICODE, *Little/Big Endian*, etc.
  - **Largura de banda**: desde alguns bytes/s até MB/s
  - **Recursos utilizados**: portos (I/O, *memory mapped*), interrupções, DMA, etc.
  - **Implementação**: diferentes dispositivos de uma dada classe podem ser baseados em implementações distintas (e.g. diferentes fabricantes, diferentes modelos) com reflexos profundos na sua operação interna

# Introdução

- As aplicações/Sistemas Operativos (SO) **não podem conhecer todos os tipos de dispositivos** passados, atuais e futuros com um nível de detalhe suficiente para realizar o seu controlo a baixo nível!
- **Solução:** Criar uma camada de abstração que permita o acesso ao dispositivo de forma independente da sua implementação
- **Device driver**
  - Um **programa** que permite a outro programa (aplicação, SO) interagir com um dado **dispositivo de hardware**
  - Implementa a camada de abstração e lida com as particularidades do dispositivo controlado
  - Como o *Device Driver* tem de lidar com os aspetos específicos da implementação hardware, o seu fornecimento é normalmente assegurado pelo fabricante
- Aspetos-chave:
  - **Abstração:** esconde as complexidades do hardware, permitindo um **acesso simplificado**
  - **Uniformização de acesso:** **diferentes dispositivos** da mesma classe podem ser **accedidos de forma semelhante**
  - **Independência entre aplicações/SO e o hardware:** facilita a **compatibilidade** e a **portabilidade do software**

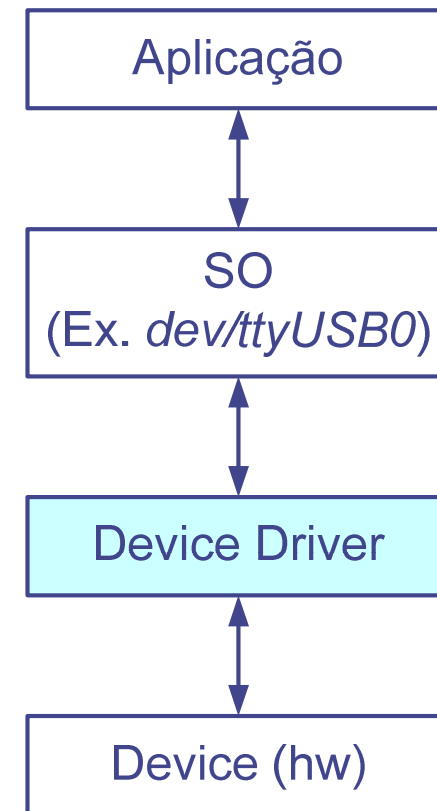
# Princípios gerais

- O acesso, por parte das aplicações, a um *device driver* é diferente num sistema embutido e num sistema computacional de uso geral (com um Sistema Operativo típico, e.g. Linux, Windows, Mac OS)
- Sistemas embutidos: as aplicações acedem diretamente aos *device drivers*
  - Estes sistemas normalmente não têm *kernel* (núcleo de um SO) ou têm um *kernel* mínimo, permitindo que o software de aplicação interaja diretamente com o hardware
  - A ausência de uma camada de abstração complexa permite respostas rápidas e baixo *overhead*, fatores críticos para sistemas de tempo real



# Princípios gerais

- Nos sistemas operativos típicos (e.g., Linux, Windows, macOS), as aplicações não interagem diretamente com os *device drivers*: as aplicações fazem chamadas ao sistema (*system calls*) quando precisam de interagir com o hardware
  - O kernel do SO recebe estas chamadas e faz o acesso ao hardware, utilizando os *device drivers* adequados
  - Os *device drivers* executam as operações específicas de acesso ao hardware

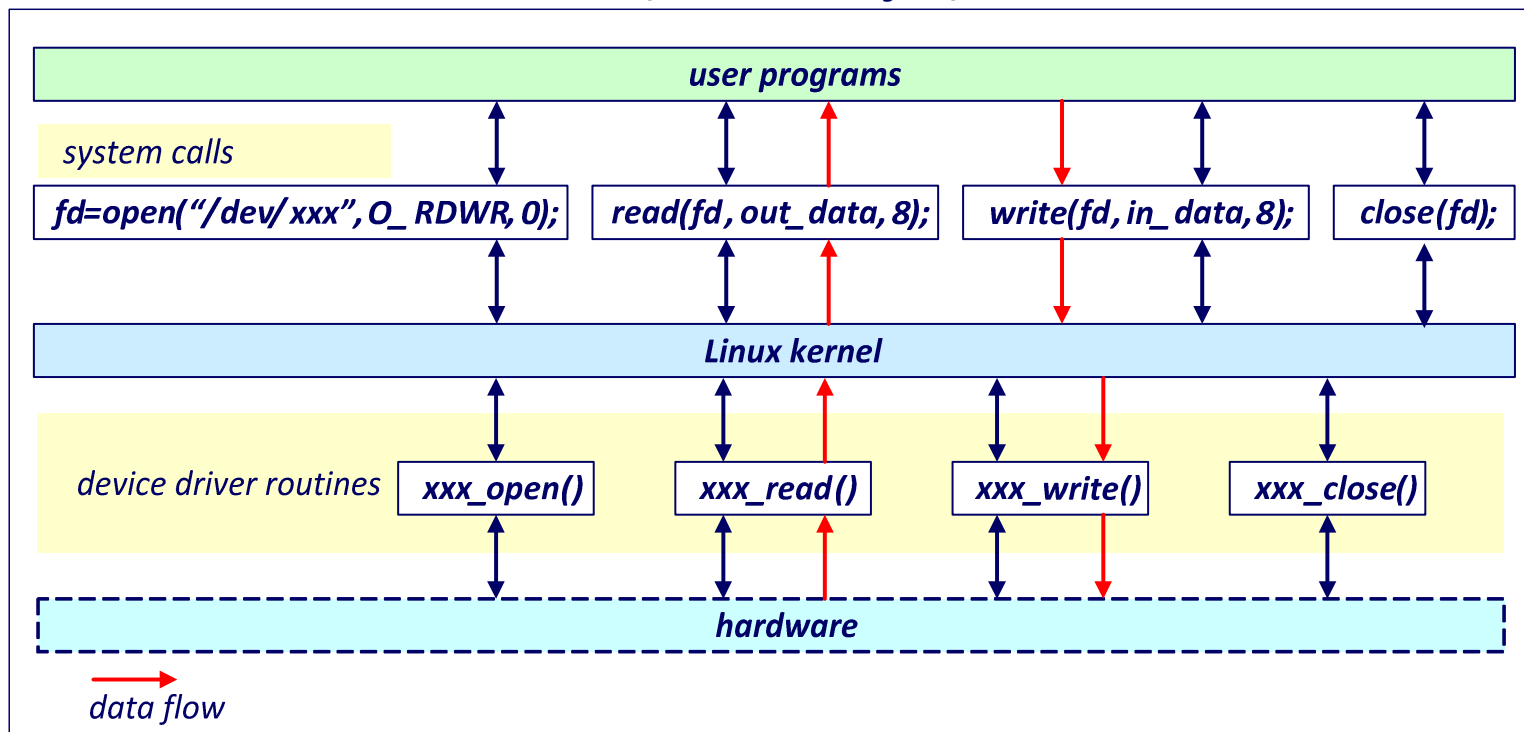


# Princípios gerais

- A separação em camadas é feita através de dois ambientes de execução distintos: "user space" e "kernel space"
  - "Espaço de utilizador" (User Space): onde as aplicações comuns são executadas, sem acesso direto ao hardware, o que evita que erros nas aplicações afetem o funcionamento do sistema operativo
  - "Espaço de Kernel" (Kernel Space): onde o sistema operativo e os *device drivers* operam, com acesso privilegiado ao hardware, permitindo ao sistema controlar os recursos físicos da máquina de forma segura
- Vantagens: segurança, estabilidade
  - Aplicações maliciosas ou com erros não podem aceder diretamente ao hardware ou interferir com o sistema operativo
  - Erros em aplicações no espaço de utilizador não comprometem o funcionamento do sistema como um todo (o *kernel* continua a funcionar normalmente)

# Princípios gerais

- O Sistema Operativo especifica classes de dispositivos e, para cada classe, uma interface que estabelece como é realizado o acesso a esses dispositivos
  - A função do *device driver* é traduzir as chamadas realizadas pela aplicação/SO em ações específicas do dispositivo
  - Exemplos de classes de dispositivos: interface com o utilizador, armazenamento de massa, comunicação, ...



# Exemplo de um *device driver*: comunicação série

- Admitindo que é fornecida uma biblioteca que apresenta a seguinte interface:

- `void comDrv_init(int baudrate, char dataBits,  
char parity, char stopBits);`
- `char comDrv_getc(void); // read a character`
- `void comDrv_putc(char ch); // write a character`
- `void comDrv_close(void);`

- **Do ponto de vista da aplicação:**

- Do ponto de vista funcional é relevante qual o modelo/fabricante do dispositivo de comunicação série?
- Se o dispositivo de comunicação for substituído por outro com arquitetura interna distinta, sendo fornecida uma biblioteca com interface compatível, é necessário alterar a aplicação?



# Caso de estudo

- Aspectos-chave da implementação de um *device driver* para uma UART RS232 (*Universal Asynchronous Receiver Transmitter*) para executar num sistema com microcontrolador (i.e., sem sistema operativo)
- Princípio de operação
  - Desacoplamento da transferência de dados entre a UART e a aplicação, realizada por meio de FIFOs (um FIFO de transmissão e um de receção). Do ponto de vista da aplicação:
    - A **transmissão** consiste em copiar os dados a enviar para o FIFO de transmissão do *device driver*
    - A **receção** consiste em ler os dados recebidos que residem no FIFO de receção do *device driver*
  - A transferência de dados entre os FIFOs e a UART é realizada por interrupção, i.e., sem intervenção explícita da aplicação
  - Um FIFO pode ser implementado através de um *buffer* circular

# Princípio de operação – recepção

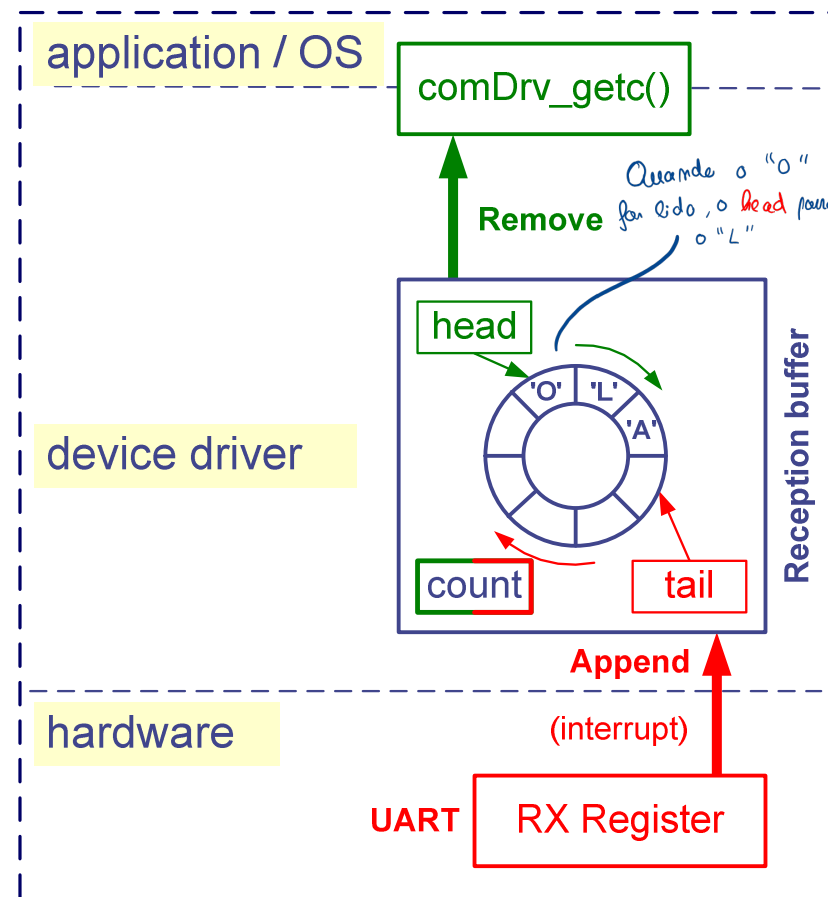
- **"tail"** – posição do buffer circular onde a **rotina de serviço à interrupção escreve** o próximo carácter lido da UART
- **"head"** – posição do buffer circular de onde a função **comDrv\_getc()** lê o próximo carácter
- **"count"** – número de caracteres residentes no buffer circular (ainda não lidos pela aplicação)
- **O acesso à variável "count" tem que ser feito numa secção crítica do código. Porquê?**

proxima posição de escrita do proximo byte

indice do array que vai ser lido da memoria

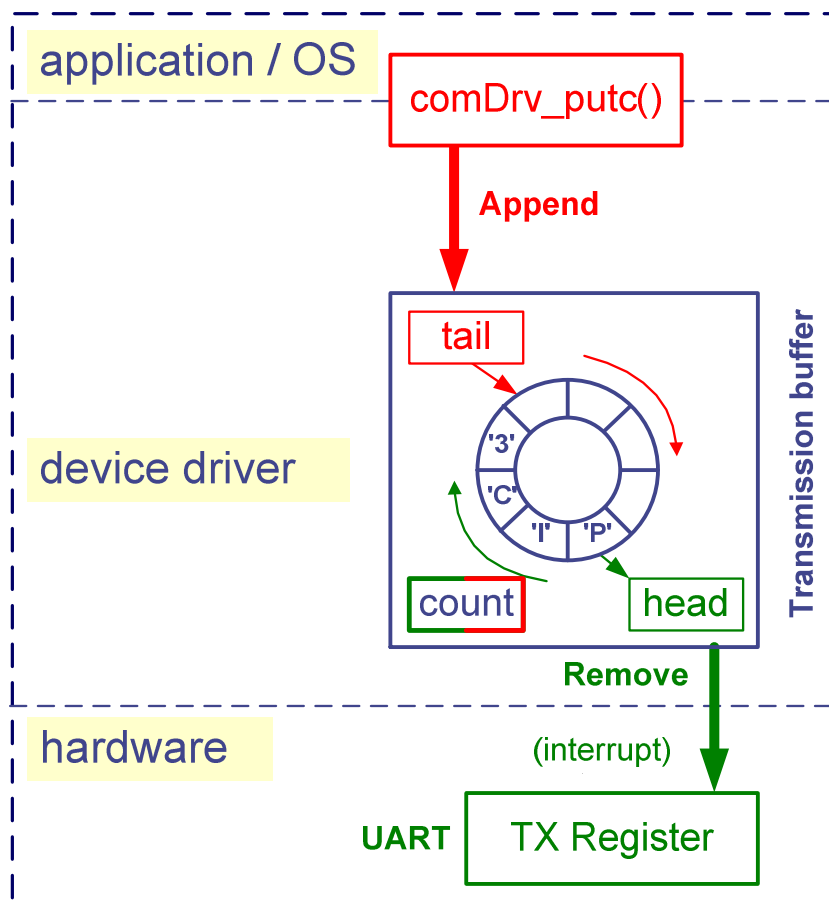
Nº de bytes que estão no FIFO

Sem interrupções



# Princípio de operação – transmissão

- **"tail"** – posição do buffer circular onde a função **comDrv\_putc()** escreve o próximo caracter
- **"head"** – posição do buffer circular de onde a **rotina de serviço à interrupção** lê o próximo caracter a enviar para a UART
- **"count"** – número de caracteres residentes no buffer circular (ainda não enviados para a UART)
- **O acesso à variável "count" tem que ser feito numa secção crítica do código. Porquê?**



# Implementação – FIFO

- FIFO - Buffer circular implementado através de um *array* linear:

```
#define BUF_SIZE 32

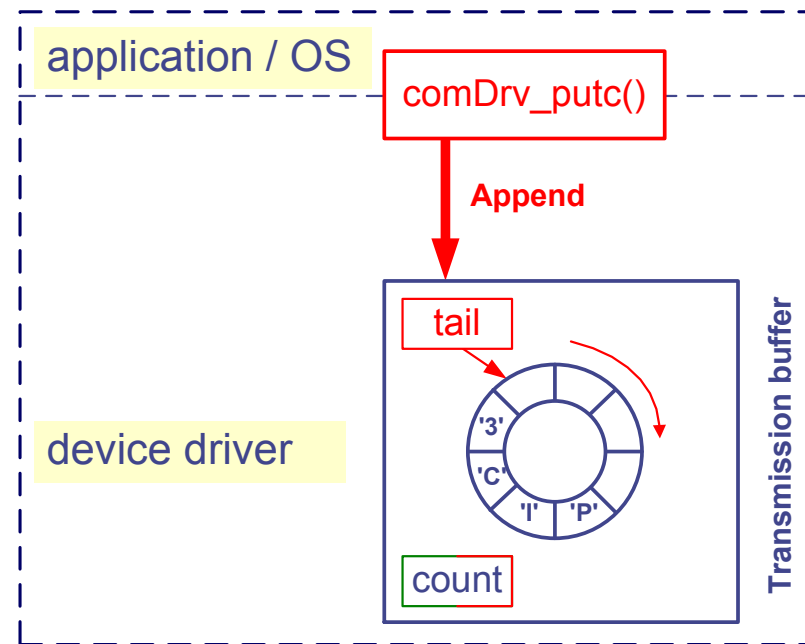
typedef struct
{
    unsigned char data[BUF_SIZE];
    unsigned int head;
    unsigned int tail;
    unsigned int count;
} circularBuffer;

circularBuffer txb; // Transmission buffer
circularBuffer rxb; // Reception buffer
```

- A constante "BUF\_SIZE" deve ser definida em função das necessidades previsíveis de pico de tráfego.
- Se "BUF\_SIZE" for uma potência de 2 simplifica a atualização dos índices do buffer circular (podem ser encarados como contadores módulo  $2^N$  e podem ser geridos com uma simples máscara)

# Implementação – Função de transmissão

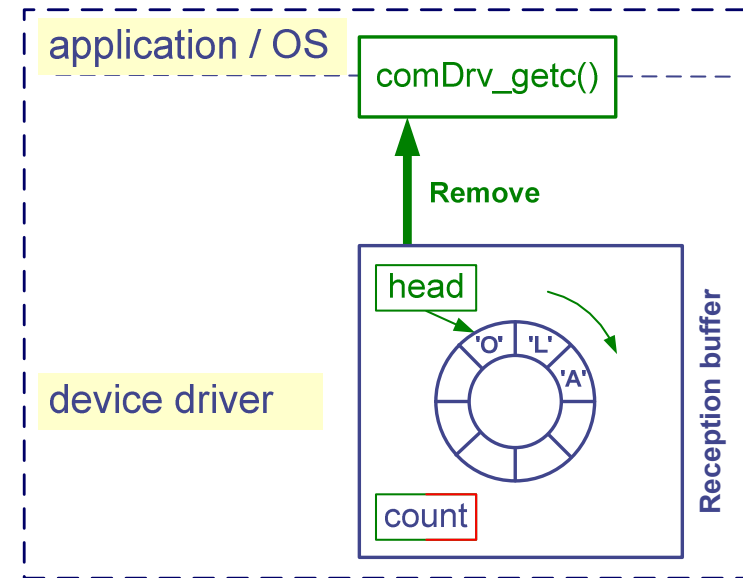
- A função de transmissão, evocada pela aplicação, copia o carácter para o **buffer de transmissão** (posição "**tail**"), incrementa o índice "**tail**" e o contador



```
void comDrv_putc(char ch)
{
    Wait while buffer is full (txb.count==BUF_SIZE)
    Copy "ch" to the buffer ("tail" position)
    Increment "tail" index (mod BUF_SIZE)
    Increment "count" variable
}
```

# Implementação – Função de receção

- A função de receção, evocada pela aplicação, verifica se há caracteres no **buffer de receção** para serem lidos e, caso haja, retorna o caracter presente na posição "**head**", incrementa o índice "**head**" e decrementa o contador

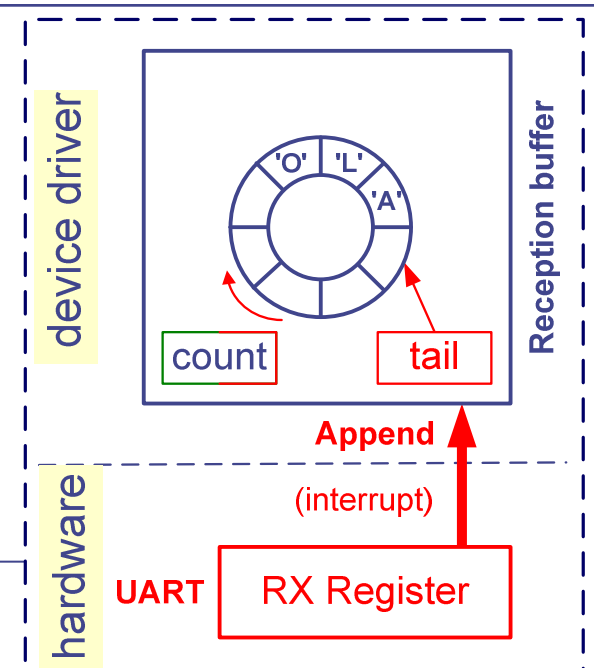


```
int comDrv_getc(char *pchar)
{
    If "count" variable is 0 then return false
    Copy character at position "head" to *pchar
    Increment "head" index (mod BUF_SIZE)
    Decrement "count" variable
    return true;
}
```

# Implementação – RSI de receção

- A rotina de serviço à interrupção da receção é executada sempre que a UART recebe um novo caracter
- O caracter recebido pela UART deve então ser copiado para o **buffer de receção**, na posição "**tail**"; a variável "**count**" deve ser incrementada e o índice "**tail**" deve ser igualmente incrementado

```
void interrupt isr_rx(void)
{
    Copy received character from UART to RX
    buffer ("tail" position)
    Increment "tail" index (mod BUF_SIZE)
    Increment "count" variable
}
```

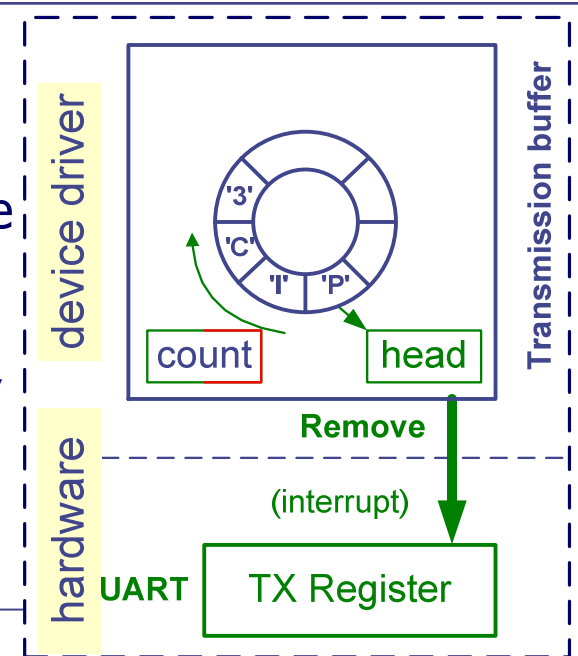


O que acontece no caso em que o *buffer* circular está cheio e a UART recebe um novo caracter? Como resolver esse problema?

# Implementação – RSI de transmissão

- A UART gera, normalmente, uma interrupção de transmissão quando tiver disponibilidade para transmitir um novo caracter
- As tarefas a implementar na respetiva rotina de serviço à interrupção são:
  - Se o número de caracteres no **buffer de transmissão** for maior que 0 ("**count**" > 0), copiar o conteúdo do buffer na posição "**head**" para a UART
  - Decrementar a variável "**count**" e incrementar o índice "**head**"

```
void interrupt isr_tx(void)
{
    If "count" > 0 then {
        Copy character from TX buffer ("head") to UART
        Increment "head" index (mod BUF_SIZE)
        Decrement "count" variable
    }
    If "count" == 0 then disable UART TX interrupts
}
```





# Atualização do TX "count" - Secção crítica

```
void comDrv_putc(char ch)
{
    Wait while buffer is full (count==BUF_SIZE)
    Copy "ch" to the transmission buffer ("tail")
    Increment "tail" index (mod BUF_SIZE)
    Disable UART TX interrupts
    Increment "count" variable
    Enable UART TX interrupts
}

void interrupt isr_tx(void)
{
    if "count" > 0 then
    {
        Copy character from TX buffer ("head") to UART
        Increment "head" index (mod BUF_SIZE)
        Decrement "count" variable
    }
    if "count" == 0 then disable UART TX interrupts
}
```

**Secção crítica**  
("count" é um recurso partilhado)

Se a UART estiver disponível para transmitir, desencadeia a imediata geração da interrupção de transmissão

# Atualização do RX "count" - Secção crítica

```
int comDrv_getc(char *pchar)
```

```
{
```

```
    If "count" variable is 0 then return false  
    Copy character at position "head" to *pchar  
    Increment "head" index (mod BUF_SIZE)
```

```
    Disable UART RX interrupts
```

```
    Decrement "count" variable
```

```
    Enable UART RX interrupts
```

```
    return true;
```

```
}
```

Secção crítica  
("count" é um  
recurso partilhado)

```
void interrupt isr_rx(void)
```

```
{
```

```
    Copy received character from UART to RX buffer  
    ("tail" position)
```

```
    Increment "tail" index (mod BUF_SIZE)
```

```
    Increment "count" variable
```

```
}
```