Universidade do Porto
Faculdade de Engenharia
FEUP

Protocolo de Ligação de Dados

Redes de Computadores

Tomás Alexandre Torres Pereira
up202108845@edu.fe.up.pt

Yves François Joseph Bonneau
up202205062@edu.fe.up.pt

Resumo—

Este relatório científico apresenta uma adaptação do mecanismo Stop and Wait, projetada especificamente para a transmissão de ficheiros entre dois computadores conectados por uma porta série RS-232. Além de garantir a integridade do ficheiro desde o início até ao fim da transmissão, o protocolo usado é robusto, o que assegura uma recuperação de erros eficiente durante a transmissão. Adicionalmente, tivemos em conta a independência entre diferentes camadas do protocolo, de forma a que estas operem de forma isolada, sem conhecerem a implementação de camadas superiores ou inferiores. O principal objetivo do relatório é descobrir a eficiência do Stop and Wait, sob a utilização de diferentes parâmetros, que fazem variar a qualidade da transmissão.

Em suma, a nossa implementação funciona como descrito neste relatório, e garante uma eficiência aproximada aos valores teóricos do Stop and Wait.

I. Introdução

A. *Motivação e descrição do problema*

Este relatório demonstra a implementação dum protocolo robusto e eficiente que permita a transmissão de um ficheiro (em formato binário) entre dois computadores através de uma porta série RS-232. Consideramos importante dividir os objetivos do trabalho em dois tópicos:

*1) **Implementação do protocolo**:* A primeira parte da realização deste trabalho consiste na implementação dum mecanismo de transmissão de dados. Dentro da implementação consideramos os seguintes objetivos de grande importância:

- **Integridade do ficheiro** - Tanto o tamanho como o conteúdo da imagem devem ser verificados no final da transmissão, de forma a garantir uma transferência completa e correta.
- **Verificação e controlo de erros** - Durante a transmissão, na eventualidade de um erro ocorrer, o protocolo deve ser capaz de recuperar, caso contrário fecha a transmissão, no caso de partes do ficheiro terem sido corrompidas.
- **Independência e interface entre camadas** - É importante garantir que as camadas sejam isoladas entre si. Esta independência permite que a implementação seja mais eficiente, simples e modular. Este conceito também é usado nos protocolos mais usados internacionalmente como TCP/IP e OSI. A aproximação da nossa implementação aos padrões internacionais garante uma maior fiabilidade do protocolo desenvolvido.

*2) **Eficiência e estudo do protocolo**:* Depois do protocolo estar implementado, é necessário verificar a eficiência do mesmo quando testado sob a variação de diferentes parâmetros, como por exemplo, o número de retransmissões, a taxa de transmissão, o tempo de espera de resposta de uma transmissão de um pacote com dados, atrasos na propagação, e erros durante a transmissão.

B. *Descrição sucinta da estrutura do relatório*

O relatório está estruturado da seguinte forma:

- **Arquitetura** - A arquitetura contém os blocos funcionais e interfaces utilizadas.
- **Estrutura do código** - Esta seção exibe as API's e principais funções e estruturas de dados usadas.
- **Principais casos de uso** - Identificação de casos de uso para diferentes partes do protocolo, e sequência de chamadas de principais funções.
- **Protocolo na camada de ligação lógica** - Implementação e estratégias usadas para resolver os diferentes problemas na camada de ligação lógica.
- **Protocolo na camada de aplicação** - Implementação e estratégias usadas para resolver os diferentes problemas na camada de aplicação.
- **Validação do protocolo** - Diferentes testes usados para testar a eficiência do protocolo e resultados.
- **Eficiência do protocolo** - Caracterização estatística do protocolo e comparação com os valores teóricos estabelecidos para o mecanismo Stop and Wait.
- **Conclusão** - Síntese dos resultados e reflexão sobre o trabalho desenvolvido.

II. Arquitetura

O protocolo foi estruturado em três camadas: duas delas desenvolvidas por nós (camada de aplicação e camada de ligação lógica) e uma fornecida *a priori* (camada física). O protocolo implementado utiliza um serviço orientado à conexão entre o emissor e o recetor. O diagrama a seguir ilustra a arquitetura implementada:

- **Camada de aplicação** - Camada responsável por construir os pacotes enviados para a camada inferior e trabalhar com a lógica da transmissão de um ficheiro através de uma porta série.

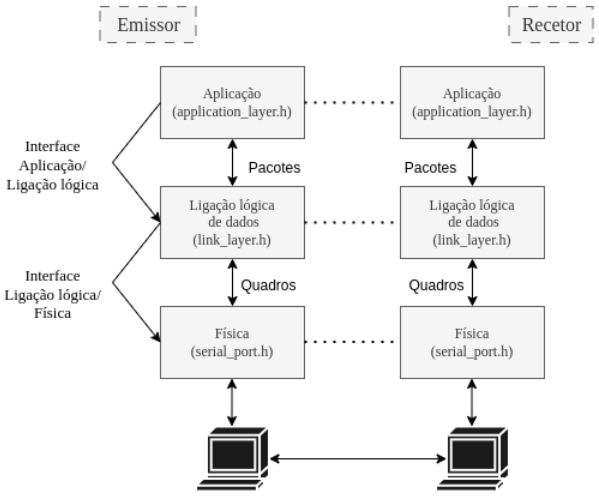

Figura 1. Arquitetura do protocolo

- **Camada de ligação lógica de dados** - Camada responsável pela deteção de erros de transmissão, retransmissão e construção dos quadros.
- **Camada física (RS232)** - Camada responsável por transmissão de bits por um canal de comunicação.
- **Pacotes** - Unidade de dados utilizada na camada de aplicação.
- **Quadros** - Unidade de dados utilizada na camada de ligação lógica de dados.

III. Estrutura do código

O código desenvlovido encontra-se divido em 3 ficheiros diferentes que representam as 3 camadas do protocolo:

- **link_layer.c** - Contém toda a implementação associada com a camada de ligação lógica dos dados.
- **application_layer.c** - Contém toda a implementação associada com a camada de aplicação.
- **serial_port.c** - Constitui um API para a camada de ligação lógica de dados.

A. *API's*

*1) **Ligação lógica de dados/Aplicação:*** A camada de ligação lógica de dados implementa uma API que pode ser utilizada como serviços

pela camada superior. A camada de aplicação pode utilizar os seguintes métodos da camada inferior:

- **llopen** - Esta função é responsável pela conexão entre emissor e recetor.
- **llwrite** - Esta função é responsável por enviar os dados do ficheiro desde o emissor até ao recetor.
- **llread** - Esta função é responsável por receber os dados, e envia-los de volta para a camada de aplicação.
- **llclose** - Esta função é responsável pelo término da conexão entre o emissor e o recetor.

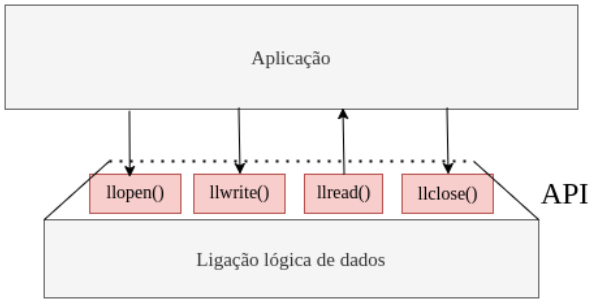

Figura 2. API fornecida pela ligação lógica de dados para a camada de aplicação

*2) **Porta série/Ligação lógica de dados:*** A camada de porta série implementa uma API que proporciona à camada superior o acesso às funcionalidades do sistema operativo relacionadas com o uso da porta série. Esta camada é uma abstração da verdadeira camada física. A camada de ligação lógica de dados pode utilizar os seguintes métodos, da camada inferior:

- **openSerialPort** - Esta função abre e configura a porta série.
- **closeSerialPort** - Esta função restaura as configurações iniciais da porta série e fecha-a.
- **writeBytesSerialPort** - Esta função é responsável por enviar um número exato de bytes pela porta série.
- **readByteSerialPort** - Esta função lê o último byte recebido na porta série.

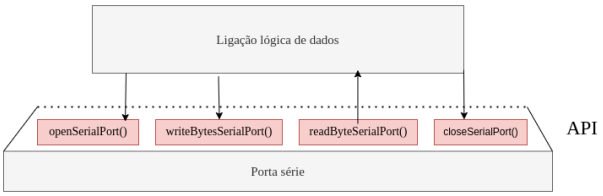

Figura 3. API fornecida pela porta série para a camada de ligação lógica de dados

B. Principais funções e estruturas de dados

*1) **Aplicação:***

- **void applicationLayer(const char *serialPort, const char *role, int baudRate, int nTries, int timeout, const char *filename);** - Esta função é responsável pela lógica principal da aplicação. Recebe como argumentos um apontador para a porta série, o número de retransmissões máximos para um quadro, o tempo de espera por uma resposta de confirmação para um quadro, e o nome do ficheiro. O argumento *role* indica se a transmissão ocorre do lado do emissor ou recetor ('tx' - emissor , 'rx' - recetor).
- **long getFileSize(FILE* fptr);** - Esta função é responsável por extrair o tamanho de um ficheiro.
- **unsigned char* buildControlPacket(long fileSize, const char* filename, int* sizePacket, unsigned char control);** - Esta função é responsável por construir um pacote de controlo, que pode ser de dois tipos: o pacote START, indicando o início da transmissão de dados, e o pacote END, sinalizando o término da transmissão de dados.
  O propósito destes pacotes é sinalizar ao recetor quando a transmissão de dados começa e termina. Na nossa implementação apenas usamos dois campos de controlo (tamanho e nome do ficheiro), mas é possível adicionar outros campos.

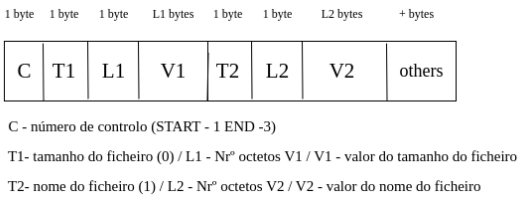

Figura 4. Estrutura de um pacote de controlo

- **unsigned char* buildDataPacket(FILE* fptr, int payload, int s, int* sizeDataPacket);** - Esta função é responsável por construir um pacote de dados. Ela recebe, como argumento, um payload que indica o número de bytes a serem extraídos do ficheiro (referenciado pelo apontador fptr). Além disso, recebe um número de sequência, que permite verificar se os pacotes são recebidos na ordem correta no lado do recetor.

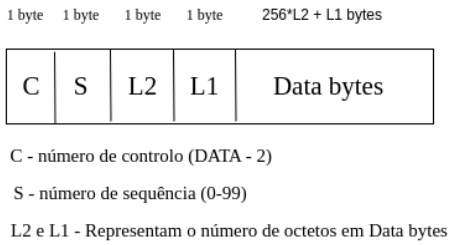

Figura 5. Estrutura de um pacote de dados

2) Ligação lógica de dados :

- **int llopen(LinkLayer connectionParameters);** - Esta função é responsável por estabelecer a conexão entre o emissor e o recetor. A estrutura *LinkLayer* armazena os parâmetros essenciais para a configuração e gestão da conexão. Em caso de sucesso, a função retorna o *file descriptor*; em caso de erro, retorna -1.
- **int llwrite(const unsigned char *buf, int bufSize);** - Esta função é responsável por enviar dados armazenados em *buf*, com um tamanho de *bufSize*, do emissor para o recetor, em forma de quadros. Em caso de sucesso, retorna o número de bytes escritos; em caso de erro, retorna -1.
- **int llread(unsigned char *packet);** - Esta função é responsável por receber os quadros enviados pelo llwrite, e verificar se os mesmos estão sem erros. Em caso de sucesso, retorna o número de bytes guardados no *packet*; em caso de erro, retorna -1.
- **int llclose(int showStatistics);** - Esta função é responsável pelo término da conexão entre o emissor e o recetor. O argumento *showStatistics*, se for igual a '1' mostra as seguintes estatísticas: Tempo gasto durante a transmissão do ficheiro, e alarmes foram ativos,interrompidos, e fechados. Em caso de sucesso, retorna 1; em caso de erro, retorna -1.
- **void buildFrameSupervision(unsigned char* frame, const unsigned char address, const unsigned char control);** - Esta função é responsável por construir a estrutura de um quadro de supervisão e não numerados.

Figura 6. Estrutura de um quadro de supervisão ou não numerado

- **void buildFrameInformation(unsigned char* frame, const unsigned char* data, const unsigned char control, const int bytes);** - Esta função é responsável por construir a estrutura de um quadro de informação.

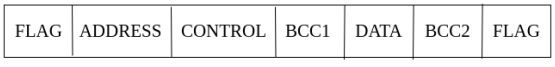

Figura 7. Estrutura de um quadro de informação

Uma possível melhoria seria usar um CRC (cyclic redundancy check), isto porque o BCC1 e BCC2 não verificam se os bytes estão fora de ordem.

- **Linklayer e LinkLayerRole** - Estas duas estruturas de dados guardam informações sobre a conexão na ligação lógica de dados.
- **SupervisionState e InformationState** - Estas duas enumerações guardam os diferentes estados usados para uma máquina de estados que recebe quadros de supervisão e não numerados, ou para uma máquina de estados que recebe quadros de informação.

IV. Casos de uso Principais

A. *Interface do programa*

A transmissão do ficheiro, pode ser feita através de uma porta série virtual, ou feito na sala I322 da FEUP. Siga as seguintes instruções:

1) Ligar o terminal e abrir a pasta do projeto onde se encontra a Makefile.
2) Dentro da Makefile, faça as seguintes alterações:
   - TX_SERIAL_PORT = /dev/ttyS10 (virtual) /dev/ttyS0
   - RX_SERIAL_PORT = /dev/ttyS11 (virtual) /dev/ttyS0 (FEUP)
   - TX_FILE = 'nome do ficheiro no emissor'
   - RX_FILE = 'nome do ficheiro no recetor'
   - BAUD_RATE = 'valor da taxa de transmissão, valor predefinido = 9600'
3) Para correr o programa, abra 3 janelas na mesma pasta e siga as seguintes instruções:
   - Na primeira janela, corra o seguinte comando -

     `sudo make run_cable`

     (Apenas necessário em modo virtual).
   - Na segunda janela, corra o seguinte comando -

     `sudo make run_rx`
   - Na terceira janela, corra o seguinte comando -

     `sudo make run_tx`

B. *Sequência de chamadas de funções*

A transmissão de dados dá-se da seguinte forma:

1) Primeiramente abre-se a conexão entre emissor e recetor, sendo que ambos chamam o método llopen() através da camada de aplicação. Se a troca de quadros entre os dois utilizadores estiver incorreta o programa termina sem abrir a conexão.
2) Após a abertura da conexão, o emissor divide o ficheiro em partes mais pequenas, formando pacotes. Através do método llwrite() vai enviando estes pacotes, até esgotar todos os bytes do ficheiro.
3) No lado do recetor, este fica a espera de receber todos os pacotes através do uso do llread() e guarda os dados num novo ficheiro.
4) No final ambos terminam a transmissão com o llclose().

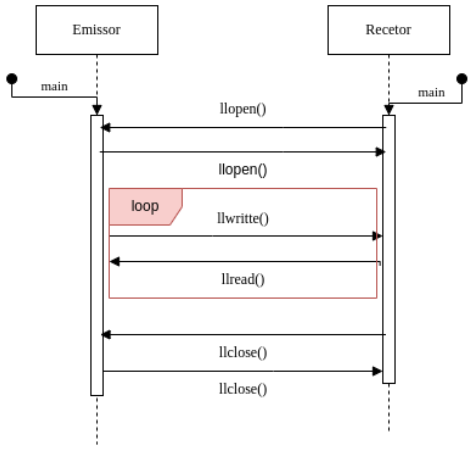

Figura 8. Diagrama da sequência geral da chamada de funções

V. Protocolo de ligação lógica

A implementação deste protocolo é baseada no mecanismo Stop and Wait. Todos os quadros recebidos sofrem verificação de erros a partir do BCC (Block Check Character). Existindo diferença no BCC calculado e recebido, é pedido uma retransmissão com REJ pelo

recetor. Os quadros de dados são também acompanhados de um número de sequência, que permite detetar quadros duplicados (i.e.,no caso de ocorrer retransmissão enquanto um RR está a ser escrito na porta série). Para além disso, se o recetor não responder durante algum tempo, é retransmitido o mesmo quadro, até um número limite, momento esse em que é terminada a conexão.

A. *llopen*

Inicialmente, é estabelecida a ligação entre o transmissor e recetor através do llopen(), que abre a porta série e a configura para leitura em modo "Blocking Read", o que faz com que a leitura da porta não retorne nada enquanto não tiver lido um byte, no caso. Depois disto, o transmissor procede ao envio de um quadro SET. Após a receção pelo recetor, este responde com um quadro de supervisão. O cabo de transmissão têm ruídos, o que dificulta a passagem correta destes quadros. Existem 4 casos principais em que é necessário retransmitir o quadro SET:

- Quadro SET não chega ao recetor
- Quadro SET chega com erros ao recetor
- Quadro UA chega com erros ao emissor
- Quadro UA não chega ao emissor

Todos estes 4 casos de retransmissão são tidos em conta na nossa implementação.

B. *llwrite*

A partir desse momento pode ser iniciada a transmissão de dados por parte do emissor. O emissor recebe um pacote com dados da camada superior. A informação deste pacote de dados não têm nenhum significado na camada de ligação lógica, o que ajuda na independência entre camadas. De seguida, constrói-se um quadro de informação com os dados recebidos. Contudo, o quadro ainda não pode ser enviado imediatamente, pois os dados podem conter bytes semelhantes às flags dos quadros, o que exige um tratamento adicional. Para resolver este problema desenvolvemos um mecanismo de *byte stuffing*. Após o *stuffing*, devido a possíveis realocações dos dados em memória, o quadro fica com um tamanho possivelmente diferente. Após ter o quadro pronto para envio, utilizamos uma versão ligeiramente adaptada do Stop and Wait original. O emissor envia um quadro I(Ns) (Ns - número do quadro), e fica a espera de receber um quadro de confirmação do recetor. No caso do emissor enviar um I(Ns) e receber um RR(1-Ns) dentro do estipulado para cada transmissão, significa que o emissor pode enviar I(1-Ns). No caso do emissor receber um REJ(0) ou REJ(1), o emissor precisa voltar a retransmitir o quadro se ainda for possível. No nosso mecanismo de Stop and Wait, quando o emissor recebe um REJ(Ns), o número de tentativas restantes para retransmitir o quadro é diminuído. Esta é uma modificação em relação ao mecanismo original, que consideramos adequada, pois, no contexto da nossa experiência, estamos a utilizar um ficheiro de pequeno tamanho e a enfrentar poucos erros por quadro numa transmissão com pouco ruído. Outra razão foi a possibilidade do recetor enviar quadros REJ de maneira infinita, nunca existindo um término por parte do transmissor. Os outros dois casos que são suscetíveis a retransmissão são a perca de quadros durante a transmissão.

C. *llread*

Do lado do recetor, a função llread() permite a leitura de quadros de dados e envio de alguns quadros de supervisão. Ao receber quadros de informação, para cada byte, se este corresponder a um escape character, inicia um processo nomeado byte destuffing. Após a receção do quadro (devidamente "unstuffed") verifica se o BCC e número de sequência são os esperados. Se tudo estiver de acordo, é enviado um quadro RR(n) (Ready to receive frame with sequence number n), e espera a receção do próximo quadro. Se não estiver de

acordo, pode ser ou enviado um REJ no caso de BCC errado não duplicado, ou um RR se for um quadro duplicado, independemente do valor esperado do BCC.

D. *llclose*

No momento do término da comunicação entre os dois constituintes da comunicação, é preciso uma maneira simples de ambos se desconectarem sem erros. No lado do emissor enviamos um quadro DISC para o recetor de desconexão. Quando o recetor recebe o quadro DISC, envia o seu próprio quadro DISC. O envio de ambos os quadros é protegido pelo sistema de alarme que desenvolvemos. O alarme do lado do emissor funciona de maneira idêntica ao do llopen, mas neste caso a máquina de estados espera receber um quadro DISC. No lado do recetor funciona da mesma forma, e espera receber um UA do emissor.

Nota 1: O UA que o transmissor envia para terminar a conexão é inútil e poderia ser removido. Isto porque o transmissor termina a sua própria execução depois de ter recebido um DISC e enviado o quadro UA. A perda deste quadro pode resultar no recetor ficar "à espera"infinitamente, mesmo com retransmissão, pois o transmissor já não estará ativo para enviar a confirmação. A solução optada é a de ter um alarme do lado do recetor, que ao fim de 3 tentativas falhadas de retransmissão termina o programa com sucesso. Este é um problema conhecido (Two Generals' Problem) sem solução, e como nesta altura já ouve uma troca de quadros DISC entre o transmissor e recetor, o ficheiro já terá sido transmitido por completo com sucesso, pelo que se pode apenas terminar a execução de ambos os processos.

Nota 2: Para fins de teste nos computadores da FEUP, é necessário adicionar um

```
sleep(1)
```

logo após o envio do quadro UA. Isto ocorre porque o envio deste quadro precede o fecho da porta série, e pode acontecer que o mesmo ainda não tenha sido totalmente enviado antes do encerramento da porta série.

VI. Protocolo de aplicação

A camada de aplicação é responsável por dividir um ficheiro em pacotes e utilizar a API da camada inferior para enviar esses pacotes. No lado recetor, a camada de aplicação reagrupa os pacotes recebidos para reconstruir o ficheiro transmitido.

A. *Emissor*

No lado do emissor, o objetivo principal é dividir o ficheiro em pacotes e enviá-los para a camada de ligação lógica de dados. Inicialmente, é necessário estabelecer a conexão entre o emissor e o recetor. Uma vez estabelecida, o emissor abre o ficheiro e obtém o seu tamanho em bytes. A etapa seguinte consiste em construir e transmitir os pacotes. O primeiro e o último pacotes são pacotes de controlo, designados por START e END, respetivamente, que indicam ao recetor quando iniciar e concluir a receção dos pacotes de dados. Entre esses dois pacotes de controlo, o emissor envia pacotes de dados, com um tamanho máximo de MAX_PAYLOAD_SIZE, até que todos os bytes do ficheiro tenham sido transmitidos. Por fim, basta fechar o ficheiro e encerrar a conexão.

B. *Recetor*

O emissor recebe os pacotes e escreve esses dados para um ficheiro de output. Depois de estabelecida a conexão, espera receber um pacote de controlo START. A partir daí espera receber pacotes de dados, escrevendo-os para o ficheiro. Ao receber o pacote STOP, a conexão é parada, sendo chamado a função llclose().

VII. Validação

Para a validação do programa, foram efetuados uma série de testes, tanto usando o ficheiro

disponibilizado (cable.c), como os computadores disponibilizados na sala I322. Os testes realizados foram alvo de estudo, tendo sido efetuadas medições para efeitos de realização de gráficos e tabelas durante os mesmos, tais como: o tempo de execução, a eficiência, o bitrate, e o número de alarmes (interrompidos e não). Todas as medições foram efetuadas 10 vezes com os mesmos parâmetros, sendo as medições registadas a média destes. Os valores de bitrate apresentados são calculados dividindo o tamanho do ficheiro em bits pelo tempo de execução, e os valores eficiência dividindo o bitrate pelo baudrate. Os testes efetuados foram os seguintes:

- Desconexão temporária da conexão do cabo.
- Introdução de ruído e variação do mesmo.
- Variação da "payload size"das tramas.
- Variação do tamanho do ficheiro transmitido..
- Variação do baudrate.

Em todos os casos é conseguido a receção do ficheiro original.

VIII. Eficiência do protocolo de ligação de dados

A eficiência do protocolo foi testada com a seguinte fórmula de eficiência:

$$\frac{\text{Tamanho do Ficheiro}}{\text{Tempo de Execução}} \times \frac{100}{\text{Baud Rate}} \quad (1)$$

A. *Variação do baudrate*

Para este teste, temos as seguintes constantes:

- **Tamanho do ficheiro**: 10968 bytes = 87744 bits
- **Bit Error Ratio (BER)**: 0.000050 ≅ 4.877% FER
- **Timeout**: 5.0s
- **Limite de Retransmissão**: 10

Foi observado que com o aumento do baudrate o tempo de execução diminuí, tal como a eficiência. Uma razão para a diminuição da eficiência é que ao aumentar o baudrate, os bytes irão ser transferidos mais rapidamente, de tal modo que o programa pode não ter tempo de os processar rápido o suficiente, e estes ficarem na fila de espera no buffer mais tempo, diminuindo a eficiência.

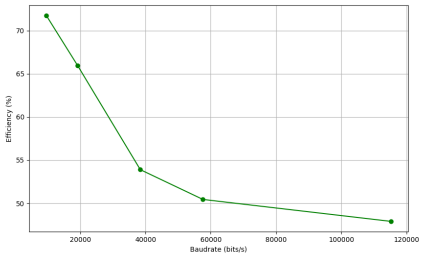

Figura 9. Eficiência sobre baudrate

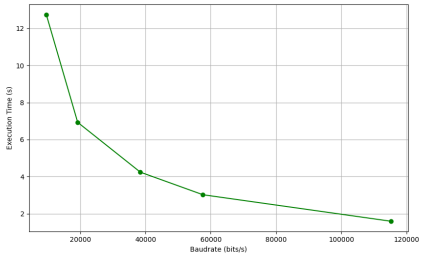

Figura 10. Tempo de execução sobre baudrate

Baud rate (baud/s)	Tempo de execução (s)	Bitrate (bit/s)	Eficiência (%)
9600	12.74	6887	71.74
19200	6.93	12661	65.94
38400	4.24	20694	53.89
57600	3.02	29054	50.44
115200	1.59	55184	47.90

Tabela I
Dados para diferentes valores de baud rate

B. *Variação do BER*

Para este teste, temos as seguintes constantes:

- **Tamanho do ficheiro**: 10968 bytes = 87744 bits
- **Baud rate**: 9600
- **Timeout**: 5.0s
- **Limite de Retransmissão**: 10

É possível observar que o tempo de execução e eficiência ambos crescem com o aumento do BER. Isto é um resultado esperado, pois com o aumenta de taxa de erros, é necessário acontecer um maior número de retransmissões, aumentando o tempo de execução. Como o baudrate é constante, isto resulta numa diminuição da eficiência.

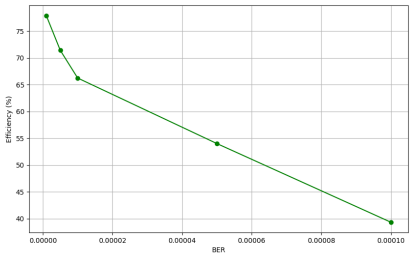

Figura 11. Eficiência sobre BER

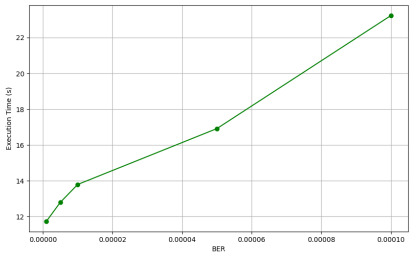

Figura 12. Tempo de execução sobre BER

BER (Bit error ratio)	Tempo de execução (s)	Bitrate (bit/s)	Eficiência (%)
0.000001	11.74	7473	77.85
0.000005	12.8	6855	71.40
0.00001	13.8	6358	66.23
0.00005	16.92	5185	54.01
0.0001	23.23	3777	39.34

Tabela II
DADOS PARA DIFERENTES VALORES DE BER

IX. CONCLUSÕES

Este projeto apresentou-nos o protocolo Stop And Wait, dando-nos a oportunidade de implementar diversos mecanismos, como verificação de erros com BCC, retransmissões, e byte stuffing. Foram implementadas 2 camadas, a camada de ligação de dados, para criação de quadros, e a camada de aplicação, que permite a criação de pacotes. A nossa implementação consegue efetuar a transmissão e receção de diferentes ficheiros de diversos tamanhos, e é resistente a ruído e interrupções, mantendo sempre uma eficiência razoável (entre 55% a 90%) em função dos parâmetros definidos (77% para tamanho de trama de 1000 bytes, baudrate de 9600 bits/s, e BER de 0.000001).

BIBLIOGRAFIA

[1] Andrew Tanenbaum, David Wetherall, *Computer Networks*, 5/E, Prentice Hall, 2011
[2] Apontamentos disponibilizados pelos docentes no moodle

ANEXO I - CÓDIGO FONTE

A. *Aplicação*

- application_layer.h -

```c
// Application layer protocol header.
// NOTE: This file must not be changed.

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

// Application layer main function.
// Arguments:
//   serialPort: Serial port name (e.g., /dev/ttyS0).
//   role: Application role {"tx", "rx"}.
//   baudrate: Baudrate of the serial port.
//   nTries: Maximum number of frame retries.
//   timeout: Frame timeout.
//   filename: Name of the file to send / receive.
void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename);

// Get the size of a file
// Arguments:
//  fptr: File pointer
long getFileSize(FILE* fptr);

// Function that returns a control packet
// Arguments:
//  fileSize: File size
//  filename: Name of the file
//  sizePacket: Number of bytes of the control packet
//  control: Number that specifies the control packet
unsigned char* buildControlPacket(long fileSize, const char* filename, int* sizePacket, unsigned char control);

// Function that returns a data packet
// Arguments:
//  fptr: File pointer
//  payload: number of bytes of data
//  s: sequence number of the packet
//  sizeDataPacket: Number of bytes of the packet (number of packet + sequence number + number of bytes payload + data)
unsigned char* buildDataPacket(FILE* fptr, int payload, int s, int* sizeDataPacket);


#endif // _APPLICATION_LAYER_H_
```

- application_layer.c -

```c
// Application layer protocol implementation

#include "application_layer.h"
#include "link_layer.h"

// Definitions
// Booleans
#define FALSE 0
#define TRUE 1
// Control Field
#define START 0x01
#define DATA 0x02
#define END 0x03
// T value
#define FILE_SIZE 0x00
#define FILE_NAME 0x01


long getFileSize(FILE* fptr){


    if ((fseek(fptr, 0, SEEK_END)) == -1) {
        perror("ERROR: Unable to seek to end of file\n");
        exit(-1);
    }

    long fileSize = ftell(fptr);
    if (fileSize == -1){
        perror("ERROR: Unable to obtain file size\n");
        exit(-1);
    }
    rewind(fptr);
    return fileSize;
}

unsigned char* buildControlPacket(long fileSize, const char* filename, int* sizePacket, unsigned char control){


    int l1 = (fileSize > 0) ? (int)(log2(fileSize) / log2(256)) + 1 : 1;
    int l2 = strlen(filename) + 1;
    (*sizePacket) = 5 + l1 + l2;

    unsigned char* controlPacket = (unsigned char *)malloc((*sizePacket) * sizeof(unsigned char));
    if (controlPacket == NULL){
        perror("ERROR: Unable to allocate memory\n");
        exit(-1);
    }

    controlPacket[0] = control; // C
    controlPacket[1] = FILE_SIZE; // T1
    controlPacket[2] = (unsigned char) l1; // L1

    //V1
    for (int i = 0 ; i < l1; i++){
        controlPacket[3 + i] = (fileSize >> (8 * i)) & 0xFF;
    }

    controlPacket[3 + l1] = FILE_NAME; //T2
    controlPacket[4 + l1] = (unsigned char) l2; //L2

    //V2
    for (int i = 0; i < l2; i++){
        controlPacket[5 + l1 + i] = filename[i];
    }
```

```
    return controlPacket;
}

unsigned char* buildDataPacket(FILE* fptr, int payload, int s, int* sizeDataPacket){


    (*sizeDataPacket) = payload + 4;
    unsigned char* dataPacket = (unsigned char*) malloc((*sizeDataPacket) * sizeof(unsigned char));

    if (dataPacket == NULL){
        perror("ERROR: Unable to allocate memory\n");
        exit(-1);
    }

    dataPacket[0] = DATA;
    dataPacket[1] = (unsigned char) s;
    dataPacket[2] = (unsigned char) (payload >> 8) & 0xFF;
    dataPacket[3] = (unsigned char) payload & 0xFF;

    for (int i = 0; i < payload; i++){
        int byte = fgetc(fptr);

        if (byte == EOF){
            perror("ERROR: Unexpected end of file\n");
            exit(-1);
        }

        dataPacket[4 + i] = (unsigned char) byte;
    }
    return dataPacket;
}


void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{


    int showStats;
    printf("0 - hide statistics 1 - show statistics: ");
    scanf("%d", &showStats);


    LinkLayer openConnection;
    strcpy(openConnection.serialPort, serialPort);
    openConnection.baudRate = baudRate;
    openConnection.nRetransmissions = nTries;
    openConnection.timeout = timeout;
    int fd;


    if (strcmp(role, "tx") == 0){

        // Open connection
        openConnection.role = LlTx;
        fd = llopen(openConnection);
        if (fd == -1){
            perror("ERROR: Error opening connection\n");
            exit(-1);
        }

        FILE* fPtr = fopen(filename, "rb");
        if (fPtr == NULL){
            perror("ERROR: Error opening file\n");
            exit(-1);
        }

        long fileSize = getFileSize(fPtr);

        // Start packet
        int sizeStartPacket = 0;
        unsigned char* startPacket = buildControlPacket(fileSize, filename, &sizeStartPacket,START);
        if (llwrite(startPacket, sizeStartPacket) == -1){
            perror("ERROR: Error sending start packet\n");
            exit(-1);
        }
        free(startPacket);

        // Data packets
        int s = 0; // Sequence number
        int usePayload;
        int sendPayload = fileSize;
        int sizeDataPacket;

        while (sendPayload > 0) {
            if (sendPayload > MAX_PAYLOAD_SIZE){
                usePayload = MAX_PAYLOAD_SIZE;
            }
            else{
                usePayload = sendPayload;
            }

            unsigned char* dataPacket = buildDataPacket(fPtr, usePayload, s, &sizeDataPacket);


            if (llwrite(dataPacket, sizeDataPacket) == -1){
                perror("ERROR: Error sending data packet\n");
                exit(-1);
            }

            free(dataPacket);

            s++;
            if (s > 99){
                s = 0;
            }
            sendPayload -= usePayload;
        }

        // End packet
        int sizeEndPacket = 0;
        unsigned char* endPacket = buildControlPacket(fileSize, filename, &sizeEndPacket,END);
        if (llwrite(endPacket, sizeEndPacket) == -1){
            perror("ERROR: Error sending start packet\n");
            exit(-1);
```

```
        }

        free(endPacket);

        if (fclose(fPtr) == -1){
            perror("ERROR: Error closing file\n")
                ;
            exit(-1);
        }

        int closeConnection = llclose(showStats);
        if (closeConnection == -1){
            perror("ERROR: Error closing
                connection\n");
            exit(-1);
        }

    }
    else if (strcmp(role, "rx") == 0) {

        openConnection.role = LlRx;
        fd = llopen(openConnection);

        if (fd == -1) {
            perror("Error: Error opening
                connection");
            exit(-1);
        }

        unsigned char* packet = (unsigned char*)
            malloc((MAX_PAYLOAD_SIZE + 4)*sizeof(
            unsigned char));
        if (packet == NULL) {
            perror("ERROR: Unable to allocate
                memory");
            exit(-1);
        }

        while (TRUE){
            int size = llread(packet);

            if (size > 0){
                if (packet[0] != START){
                    perror("Unexpected START
                        packet");
                    free(packet);
                    exit(-1);
                }
                break;
            }
        }

        FILE* file = fopen(filename, "wb");

        if (file == NULL){
            perror("ERROR: Error opening file\n")
                ;
            free(packet);
            exit(-1);
        }

        int expectedSequenceNumber = 0;
        while (TRUE) {
            int size = llread(packet);

            if (size > 0){
                if (packet[0] == DATA){
                    if (packet[1] ==
                        expectedSequenceNumber){
                        expectedSequenceNumber =
                            (
                            expectedSequenceNumber
                            + 1) % 100;
                        fwrite(packet+4, sizeof(
                            unsigned char), size
                            -4, file);
                    }
                    else{
                        perror("Unexpected DATA
                            packet");
                        free(packet);
                        exit(-1);
                    }
                }
                else if (packet[0] == END){
                    break;
                }
            }
        }

        free(packet);

        if (fclose(file) == -1){
            perror("ERROR: Error closing file\n")
                ;
            exit(-1);
        }

        fd = llclose(showStats);
        if (fd == -1){
            perror("Error: Error clsosing
                connection");
            exit(-1);
        }

    }
}
```

B. *Ligação lógica de dados*

- link_layer.h -

```
// Link layer header.
// NOTE: This file must not be changed.

#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <sys/time.h>

typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;

typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
```

```
} LinkLayer;

// SIZE of maximum acceptable payload.
// Maximum number of bytes that application layer
      should send to link layer
#define MAX_PAYLOAD_SIZE 1000

// MISC
#define FALSE 0
#define TRUE 1

// Build a supervision frame
// Arguments:
//  frame - pointer to frame
//  address - address (receiver/sender)
//  control - message control
void buildFrameSupervision(unsigned char* frame,
     const unsigned char address, const unsigned
     char control);

// Build a information frame
// Arguments:
//  frame - pointer to frame
//  data - information data
//  control - I(0) / I(1) control
//  bytes - number of data bytes
void buildFrameInformation(unsigned char* frame,
     const unsigned char* data, const unsigned
     char control, const int bytes);

// Open a connection using the "port" parameters
     defined in struct linkLayer.
// Return "1" on success or "-1" on error.
int llopen(LinkLayer connectionParameters);

// Send data in buf with size bufSize.
// Return number of chars written, or "-1" on
     error.
int llwrite(const unsigned char *buf, int bufSize
     );

// Receive data in packet.
// Return number of chars read, or "-1" on error.
int llread(unsigned char *packet);

// Close previously opened connection.
// if showStatistics == TRUE, link layer should
     print statistics in the console on close.
// Return "1" on success or "-1" on error.
int llclose(int showStatistics);

#endif // _LINK_LAYER_H_
```

- link_layer.c -

```
        #define _POSIX_SOURCE 1 // POSIX
             compliant source
// Link layer protocol implementation
#include "link_layer.h"
#include "serial_port.h"


// MISC

LinkLayerRole role;
int timeout;
int nRetransmissions;
int alarmRinging = FALSE;
int alarmCount = 0;
int TotalAlarmCount = 0;
int alarmInterrupted = 0;
int baudRate;

// Definitions
// Booleans
#define FALSE 0
#define TRUE 1
// Sizes
#define FRAME_SIZE_S 5
// Flag
#define FLAG 0x7E

// Address
#define A0 0x03 // Sender
#define A1 0x01 // Receiver

#define C_IF(Nr) (Nr << 7)

// Control
#define SET 0x03
#define UA 0x07
#define RR(Nr) (0xAA | Nr)
#define REJ(Nr) (0x54 | Nr)
#define DISC 0x0B

// Byte stuffing
#define ESC 0x7D

// Statistics
struct timeval  tv1, tv2;

typedef enum SupervisionState
{

    S_WAITING_FLAG,
    S_WAITING_ADDR,
    S_WAITING_CTRL,
    S_WAITING_BCC1,
    S_WAITING_FLAG2,
    S_STOP_STATE
} SupervisionState;
typedef enum InformationState
{

    I_WAITING_FLAG,
    I_WAITING_ADDR,
    I_WAITING_CTRL,
    I_WAITING_BCC1,
    I_WAITING_DATA,
    I_WAITING_BCC2,
    I_ESC_OCT_RCV,
    I_STOP_STATE
} InformationState;

unsigned char Ns; // expected number for frame

void alarmHandler(int signal)
{
    // Alarm enabled means that the alarm got
         triggered (time set expired)
    if (signal == SIGALRM) {
        alarmRinging = FALSE;
        alarmCount++;
        printf("Alarm␣#%d␣expired\n", alarmCount)
             ;
    }
    else{
        printf("Unexpected␣signal␣%d\n", signal);
    }
}
```

```c
void buildFrameSupervision(unsigned char* frame,
    const unsigned char address, const unsigned
    char control){

    frame[0] = FLAG;
    frame[1] = address;
    frame[2] = control;
    frame[3] = address ^ control;
    frame[4] = FLAG;
}

void buildFrameInformation(unsigned char* frame,
    const unsigned char* data, const unsigned
    char control, const int bytes){

    frame[0] = FLAG;
    frame[1] = A0;
    frame[2] = control;
    frame[3] = A0 ^ control;

    unsigned char bcc2 = 0;
    for (int i = 0; i < bytes; i++){
        frame[4 + i] = data[i];
        bcc2 ^= data[i];
    }
    frame[4 + bytes] = bcc2;
    frame[5 + bytes] = FLAG;
}


////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{

    int fd = openSerialPort(connectionParameters.
        serialPort, connectionParameters.baudRate
        );
    if (fd == -1)
        return fd;
    printf("Serial port opened\n");

    // Define parameters
    role = connectionParameters.role;
    timeout = connectionParameters.timeout;
    nRetransmissions = connectionParameters.
        nRetransmissions;
    baudRate = connectionParameters.baudRate;


    SupervisionState state = S_WAITING_FLAG;
    unsigned char byte_read = 0;

    struct sigaction sa;
    sa.sa_handler = alarmHandler;
    sa.sa_flags = 0;

    if (sigaction(SIGALRM, &sa, 0) == -1){
        perror("ERROR: Setting signal handler\n")
            ;
        return -1;
    }

    if (role == LlTx)
    {
        int lastAlarmCount = 0;

        while (state != S_STOP_STATE &&
            alarmCount < nRetransmissions)
        {
            if (alarmRinging == FALSE)
            {
                lastAlarmCount = alarmCount;
                alarm(timeout);
                alarmRinging = TRUE;
                state = S_WAITING_FLAG;

                // Send the SET message
                unsigned char buf[FRAME_SIZE_S] =
                    {FLAG, A0, SET, A0 ^ SET,
                    FLAG};
                if (writeBytesSerialPort(buf,
                    FRAME_SIZE_S) != FRAME_SIZE_S
                    )
                {
                    perror("Error writing SET
                        frame to serial port\n");
                    alarm(0);
                    if (alarmCount ==
                        lastAlarmCount){
                        alarmRinging = FALSE;
                        alarmCount++;
                        alarmInterrupted++;
                        printf("Alarm %d
                            interrupted\n",
                            alarmCount);
                    }
                    continue;
                }
            }

            if (readByteSerialPort(&byte_read) >
                0)
            {
                switch (state)
                {
                case S_WAITING_FLAG:
                    if (byte_read == FLAG)
                    {
                        state = S_WAITING_ADDR;
                    }
                    break;
                case S_WAITING_ADDR:
                    if (byte_read == A0)
                        state = S_WAITING_CTRL;
                    else if (byte_read != FLAG)
                        state = S_WAITING_FLAG;
                    break;
                case S_WAITING_CTRL:
                    if (byte_read == UA)
                        state = S_WAITING_BCC1;
                    else if (byte_read == FLAG)
                        state = S_WAITING_ADDR;
                    else
                        state = S_WAITING_FLAG;
                    break;
                case S_WAITING_BCC1:
                    if (byte_read == (A0 ^ UA))
                        state = S_WAITING_FLAG2;
                    else if (byte_read == FLAG)
                        state = S_WAITING_ADDR;
                    else
                    {
                        state = S_WAITING_FLAG;
                    }
                    break;
                case S_WAITING_FLAG2:
                    if (byte_read == FLAG){
                        state = S_STOP_STATE;
```

```
                        alarm(0);
                        if (lastAlarmCount ==
                             alarmCount){
                            alarmCount++;
                            alarmInterrupted++;
                            printf("Alarm␣%d␣
                                 interrupted\n",
                                 alarmCount);
                        }
                        printf("Connection␣
                             established\n");
                        gettimeofday(&tv1, NULL);
                        printf("Starting␣counting
                             ␣now\n");
                    }
                    else {
                        state = S_WAITING_FLAG;
                        break;
                    }
                    case S_STOP_STATE:
                        break;
                }
            }
        }

        TotalAlarmCount += alarmCount;
        if (state != S_STOP_STATE){
            perror("Failed␣to␣establish␣
                 connection\n");
            return -1;
        }

    }
    else if (connectionParameters.role == LlRx)
    {
        while (state != S_STOP_STATE)
        {
            if (readByteSerialPort(&byte_read) >
                 0)
            {
                switch (state)
                {
                case S_WAITING_FLAG:
                    if (byte_read == FLAG)
                    {
                        state = S_WAITING_ADDR;
                    }
                    break;
                case S_WAITING_ADDR:
                    if (byte_read == A0)
                        state = S_WAITING_CTRL;
                    else if (byte_read != FLAG)
                        state = S_WAITING_FLAG;
                    break;
                case S_WAITING_CTRL:
                    if (byte_read == SET)
                        state = S_WAITING_BCC1;
                    else if (byte_read == FLAG)
                        state = S_WAITING_ADDR;
                    else
                        state = S_WAITING_FLAG;
                    break;
                case S_WAITING_BCC1:
                    if (byte_read == (A0 ^ SET))
                        state = S_WAITING_FLAG2;
                    else if (byte_read == FLAG)
                        state = S_WAITING_ADDR;
                    else
                    {
                        state = S_WAITING_FLAG;
                    }
                    break;
                case S_WAITING_FLAG2:
                    if (byte_read == FLAG)
                    {
                        unsigned char buf[
                             FRAME_SIZE_S] = {FLAG
                             , A0, UA, A0 ^ UA,
                             FLAG};
                        if (writeBytesSerialPort(
                             buf, FRAME_SIZE_S) !=
                              FRAME_SIZE_S)
                        {
                            perror("Failed␣to␣
                                 write␣UA␣frame\n"
                                 );
                            state =
                                 S_WAITING_FLAG;
                            break;
                        }
                        else
                        {
                            state = S_STOP_STATE;
                        }
                    }
                    else {
                        state = S_WAITING_FLAG;
                        break;
                    }
                    case S_STOP_STATE:
                        break;
                }
            }
        }
    }

    return fd;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize
     ) {


    struct sigaction sa;
    sa.sa_handler = alarmHandler;
    sa.sa_flags = 0;
    if (sigaction(SIGALRM, &sa, 0) == -1) {
        perror("ERROR:␣Setting␣signal␣handler\n")
             ;
        return -1;
    }

    if (buf == NULL) {
        perror("ERROR:␣buffer␣is␣null\n");
        return -1;
    }

    int frameBytes = bufSize + 6;

    unsigned char* frameBufferSend = (unsigned
         char*)malloc(frameBytes * sizeof(unsigned
          char));
    if (frameBufferSend == NULL) {
        perror("ERROR:␣Allocating␣memory␣for␣
             frameBufferSend\n");
        return -1;
    }
```

```
    buildFrameInformation(frameBufferSend, buf,
         C_IF(Ns), bufSize);

    int extraBytes = 0;

    // Byte stuffing (+1 for bcc2 stuffing)
    for (int i = 0; i < bufSize + 1; i++) {
        if (frameBufferSend[4 + i] == FLAG ||
             frameBufferSend[4 + i] == ESC){
            extraBytes++;
        }
    }


    if (extraBytes > 0) {

        frameBytes += extraBytes;
        frameBufferSend = realloc(frameBufferSend
             , frameBytes);
        if (frameBufferSend == NULL) {
            perror("ERROR: Reallocating memory
                 for frameBufferSend\n");
            free(frameBufferSend);
            return -1;
        }


        for (int i = 0; i < bufSize + extraBytes
             + 1; i++) {
            if (frameBufferSend[4 + i] == FLAG ||
                  frameBufferSend[4 + i] == ESC) {

                memmove(&frameBufferSend[4 + i +
                     2], &frameBufferSend[4 + i +
                     1], frameBytes - (4 + i + 2))
                     ;

                if (frameBufferSend[4 + i] ==
                     FLAG) {
                    frameBufferSend[4 + i] = ESC;
                    frameBufferSend[4 + i + 1] =
                         FLAG^0x20;
                } else if (frameBufferSend[4 + i]
                      == ESC) {
                    frameBufferSend[4 + i] = ESC;
                    frameBufferSend[4 + i + 1] =
                         ESC^0x20;
                }
                i++;
            }
        }

    }

    unsigned char* frameBufferReceive = (unsigned
          char*)malloc(FRAME_SIZE_S * sizeof(
         unsigned char));
    if (frameBufferReceive == NULL) {
        perror("ERROR: Allocating memory for
             frameBufferReceive\n");
        free(frameBufferSend);
        return -1;
    }

    SupervisionState state = S_WAITING_FLAG;
    int byte;
    unsigned char buffer_read = 0;
    alarmCount = 0;
    int lastAlarmCount = 0;
    int rejected_frame = FALSE;
    alarmRinging = FALSE;

    while (alarmCount < nRetransmissions) {

        if (alarmRinging == FALSE ||
             rejected_frame == TRUE) {

            lastAlarmCount = alarmCount;
            state = S_WAITING_FLAG;
            alarmRinging = TRUE;
            rejected_frame = FALSE;
            alarm(timeout);

            byte = writeBytesSerialPort(
                 frameBufferSend, frameBytes);
            if (byte != frameBytes) {
                perror("ERROR: writing bytes to
                     serial port\n");
                alarm(0);
                if (alarmCount == lastAlarmCount)
                     {
                    alarmRinging = FALSE;
                    alarmCount++;
                    alarmInterrupted++;
                    printf("Alarm %d interrupted\
                         n", alarmCount);
                }
                continue;
            } else {
                printf("I(%d) sent, bytes written
                     = %d \n",Ns,byte);
            }
        }


        byte = readByteSerialPort(&buffer_read);

        if (byte > 0) {
            switch (state) {
                case S_WAITING_FLAG:
                    if (buffer_read == FLAG) {
                        frameBufferReceive[0] =
                             buffer_read;
                        state = S_WAITING_ADDR;
                    }
                    break;
                case S_WAITING_ADDR:
                    if (buffer_read == A0) {
                        frameBufferReceive[1] =
                             buffer_read;
                        state = S_WAITING_CTRL;
                    } else if (buffer_read !=
                         FLAG) {
                        state = S_WAITING_FLAG;
                    }
                    break;
                case S_WAITING_CTRL:
                    if (buffer_read == REJ(0) ||
                         buffer_read == REJ(1) ||
                         buffer_read == RR(0) ||
                         buffer_read == RR(1)) {
                        frameBufferReceive[2] =
                             buffer_read;
                        state = S_WAITING_BCC1;
                    } else if (buffer_read ==
                         FLAG) {
                        state = S_WAITING_ADDR;
                    } else {
                        state = S_WAITING_FLAG;
                    }
```

```
                break;
            case S_WAITING_BCC1:
                if (buffer_read == (
                     frameBufferReceive[1] ^
                     frameBufferReceive[2])) {
                    frameBufferReceive[3] =
                         buffer_read;
                    state = S_WAITING_FLAG2;
                } else if (buffer_read ==
                     FLAG) {
                    state = S_WAITING_ADDR;
                } else {
                    state = S_WAITING_FLAG;
                }
                break;
            case S_WAITING_FLAG2:
                if (buffer_read == FLAG) {
                    frameBufferReceive[4] =
                         buffer_read;
                    state = S_STOP_STATE;
                    alarm(0);
                    if (alarmCount ==
                         lastAlarmCount){
                        alarmCount++;
                        alarmInterrupted++;
                        printf("Alarm␣%d␣
                             interrupted\n",
                             alarmCount);
                    }
                } else {
                    state = S_WAITING_FLAG;
                }
                break;
            default:
                state = S_WAITING_FLAG;
        }
    }

    if (state == S_STOP_STATE && byte > 0) {
        if (frameBufferReceive[2] == REJ(0)
             || frameBufferReceive[2] == REJ
             (1)) {
            printf("REJ(%d)␣received\n",
                 frameBufferReceive[2] == REJ
                 (0) ? 0 : 1);
            printf("Frames␣sent␣with␣errors\n
                 ");
            rejected_frame = TRUE;
        }
        else if ((frameBufferReceive[2] == RR
             (0) && Ns == 1) || (
             frameBufferReceive[2] == RR(1) &&
              Ns == 0)) {
            Ns ^= 1;
            printf("RR(%d)␣received\n",
                 frameBufferReceive[2] == RR
                 (0) ? 0 : 1);
            printf("Frames␣sent␣and␣received␣
                 successfully\n");
            free(frameBufferSend);
            free(frameBufferReceive);
            TotalAlarmCount += alarmCount;
            return frameBytes;
        }
    }
}

free(frameBufferSend);
free(frameBufferReceive);
return -1;
}



////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////
int llread(unsigned char *packet)
{


    InformationState state = I_WAITING_FLAG;
    unsigned char byte_read = 0;
    unsigned char received_IF = 0;
    int byte_nr = 0;
    while (TRUE) {
        if (readByteSerialPort(&byte_read) > 0) {
            switch (state) {
                case I_WAITING_FLAG:
                    if (byte_read == FLAG) {
                        state = I_WAITING_ADDR;
                    }
                    break;
                case I_WAITING_ADDR:
                    if (byte_read == A0) {
                        state = I_WAITING_CTRL;
                    } else if (byte_read == FLAG)
                         {
                        state = I_WAITING_ADDR;
                    } else {
                        state = I_WAITING_FLAG;
                    }
                    break;
                case I_WAITING_CTRL:

                    if (byte_read == C_IF(0) ||
                         byte_read == C_IF(1)) {
                        received_IF = byte_read;
                        state = I_WAITING_BCC1;
                    } else if (byte_read == FLAG)
                         {
                        state = I_WAITING_ADDR;
                    } else {
                        state = I_WAITING_FLAG;
                    }
                    break;
                case I_WAITING_BCC1:
                    if (byte_read == (A0 ^ C_IF
                         (0)) || byte_read == (A0
                         ^ C_IF(1))) {
                        state = I_WAITING_DATA;
                    } else if (byte_read == FLAG)
                         {
                        state = I_WAITING_ADDR;
                    } else {
                        state = I_WAITING_FLAG;
                    }
                    break;
                case I_WAITING_DATA:
                    if (byte_read == ESC) {
                        state = I_ESC_OCT_RCV;
                    }
                    else if (byte_read == FLAG) {


                        unsigned char bcc2_rcv =
                             packet[byte_nr - 1];
                        byte_nr--;
                        unsigned char bcc2_actual
                             = 0;
```

```
                for (size_t i = 0; i <
                     byte_nr; i++)
                {
                    bcc2_actual =
                         bcc2_actual ^
                         packet[i];
                }

                if (bcc2_actual ==
                     bcc2_rcv)
                {
                    // Frame received,
                         ready to receive
                         next frame
                    if (C_IF(Ns) ==
                         received_IF) {
                        Ns ^= 1;

                        unsigned char*
                             frame = (
                             unsigned char
                             *) malloc (
                             FRAME_SIZE_S*
                             sizeof(
                             unsigned char
                             ));
                        buildFrameSupervision
                             (frame, A0,
                             RR(Ns));
                        if (
                             writeBytesSerialPort
                             (frame,
                             FRAME_SIZE_S)
                              !=
                             FRAME_SIZE_S)
                        {
                            perror("
                                 Failed to
                                 write RR
                                 frame");
                            return -1;
                        }
                        printf("Writing
                             RR(%d) frame\
                             n", Ns);
                        return byte_nr;

                    }

                    // Duplicate frame,
                         discarding
                    else
                    {

                        unsigned char*
                             frame = (
                             unsigned char
                             *) malloc (
                             FRAME_SIZE_S*
                             sizeof(
                             unsigned char
                             ));
                        buildFrameSupervision
                             (frame, RR(Ns
                             ),A0);
                        if (
                             writeBytesSerialPort
                             (frame,
                             FRAME_SIZE_S)
                              !=
                             FRAME_SIZE_S)
                        {
                            perror("
                                 Failed to
                                 write RR
                                 packet")
                                 ;
                            return -1;
                        }

                        printf("Duplicate
                             frame,
                             writing RR(%d
                             ) frame\n",
                             Ns);
                        // ignore packet
                             received till
                              now
                        return 0;
                    }
                }
                // BCC2 error, rejecting
                     frame
                else
                {
                    if (C_IF(Ns) ==
                         received_IF)
                    {
                        unsigned char*
                             frame = (
                             unsigned char
                             *) malloc (
                             FRAME_SIZE_S*
                             sizeof(
                             unsigned char
                             ));
                        buildFrameSupervision
                             (frame, A0,
                             REJ(
                             received_IF
                             >> 7));
                        if (
                             writeBytesSerialPort
                             (frame,
                             FRAME_SIZE_S)
                              !=
                             FRAME_SIZE_S)
                        {
                            perror("
                                 Failed to
                                 write
                                 REJ frame
                                 \n");
                            return -1;
                        }

                        printf("Writing
                             REJ(%d) frame
                             \n",
                             received_IF
                             >> 7);
                        return -1;

                    }
                    else
                    {
                        unsigned char*
                             frame = (
```

```
                                    unsigned char
                                    *) malloc (
                                    FRAME_SIZE_S*
                                    sizeof(
                                    unsigned char
                                    ));
                                buildFrameSupervision
                                    (frame, RR(Ns
                                    ),A0);
                                if (
                                    writeBytesSerialPort
                                    (frame,
                                    FRAME_SIZE_S)
                                     !=
                                    FRAME_SIZE_S)
                                {
                                    perror("
                                        Failed to
                                        write RR
                                        frame\n"
                                        );
                                    return -1;
                                }
                                // ignore packet
                                     received till
                                     now
                                return 0;
                            }
                        }
                    }

                    // Reading data
                    else {
                        packet[byte_nr] =
                            byte_read;
                        byte_nr++;
                    }
                    break;
                case I_ESC_OCT_RCV:
                    packet[byte_nr] = byte_read^0
                        x20;
                    byte_nr++;
                    state = I_WAITING_DATA;
                    break;
                default:
                    state = I_WAITING_FLAG;
            }
        }
    }
}

int llclose(int showStatistics) {

    struct sigaction sa;
    sa.sa_handler = alarmHandler;
    sa.sa_flags = 0;
    if (sigaction(SIGALRM, &sa, 0) == -1) {
        perror("ERROR: Setting signal handler\n")
            ;
        return -1;
    }

    SupervisionState state;
    alarmCount = 0;
    alarmRinging = FALSE;
    int lastAlarmCount = 0;

    unsigned char* frameBufferSend = malloc(
        FRAME_SIZE_S * sizeof(unsigned char));
    if (frameBufferSend == NULL) {
        perror("ERROR: Allocating memory for
            frameBufferSend\n");
        return -1;
    }
    unsigned char* frameBufferReceive = malloc(
        FRAME_SIZE_S * sizeof(unsigned char));
    if (frameBufferReceive == NULL) {
        perror("ERROR: Allocating memory for
            frameBufferReceive\n");
        return -1;
    }

    if (role == LlTx) {
        printf("LLCLOSE: Lltx\n");
        buildFrameSupervision(frameBufferSend, A0
            , DISC);

        unsigned char buffer_read = 0;
        int byte;
        state = S_WAITING_FLAG;

        while (state != S_STOP_STATE &&
            alarmCount < nRetransmissions) {
            // Sending DISC frame
            if (!alarmRinging) {
                lastAlarmCount = alarmCount;
                alarmRinging = TRUE;
                alarm(timeout);
                state = S_WAITING_FLAG;

                byte = writeBytesSerialPort(
                    frameBufferSend, FRAME_SIZE_S
                    );
                if (byte != FRAME_SIZE_S) {
                    perror("Error writing DISC
                        frame to serial port\n");
                    alarm(0);
                    if (alarmCount ==
                        lastAlarmCount){
                        alarmRinging = FALSE;
                        alarmCount++;
                        alarmInterrupted++;
                        printf("Alarm %d
                            interrupted\n",
                            alarmCount);
                    }
                    continue;
                } else {
                    printf("DISC frame sent,
                        bytes written = %d\n",
                        byte);
                }
            }

            // Reading DISC frame
            byte = readByteSerialPort(&
                buffer_read);

            if (byte > 0) {
                switch (state) {
                    case S_WAITING_FLAG:
                        if (buffer_read == FLAG)
                            {
                            frameBufferReceive[0]
                                = buffer_read;
                            state =
                                S_WAITING_ADDR;
                        }
                        break;
                    case S_WAITING_ADDR:
```

```
                    if (buffer_read == A1) {
                        frameBufferReceive[1]
                             = buffer_read;
                        state =
                             S_WAITING_CTRL;
                    } else if (buffer_read !=
                         FLAG) {
                        state =
                             S_WAITING_FLAG;
                    }
                    break;
                case S_WAITING_CTRL:
                    if (buffer_read == DISC)
                         {
                        frameBufferReceive[2]
                             = buffer_read;
                        state =
                             S_WAITING_BCC1;
                    } else if (buffer_read ==
                         FLAG) {
                        state =
                             S_WAITING_ADDR;
                    } else {
                        state =
                             S_WAITING_FLAG;
                    }
                    break;
                case S_WAITING_BCC1:
                    if (buffer_read == (
                         frameBufferReceive[1]
                          ^ frameBufferReceive
                         [2])) {
                        frameBufferReceive[3]
                             = buffer_read;
                        state =
                             S_WAITING_FLAG2;
                    } else if (buffer_read ==
                         FLAG) {
                        state =
                             S_WAITING_ADDR;
                    } else {
                        state =
                             S_WAITING_FLAG;
                    }
                    break;
                case S_WAITING_FLAG2:
                    if (buffer_read == FLAG)
                         {
                        frameBufferReceive[4]
                             = buffer_read;
                        state = S_STOP_STATE;
                        alarm(0);
                        if (alarmCount ==
                             lastAlarmCount){
                            alarmCount++;
                            alarmInterrupted
                                 ++;
                            printf("Alarm␣%d␣
                                 interrupted\n
                                 ", alarmCount
                                 );
                        }
                    } else {
                        state =
                             S_WAITING_FLAG;
                        break;
                    }
                case S_STOP_STATE:
                    break;
            }
            }
        }

        TotalAlarmCount += alarmCount;

        if (state != S_STOP_STATE) {
            perror("ERROR:␣Timeout␣during␣
                 receiving␣DISC\n");
            free(frameBufferSend);
            free(frameBufferReceive);
            return -1;
        }
        else{
            printf("DISC␣frame␣received\n");
        }


        // Create and send UA frame
        buildFrameSupervision(frameBufferSend, A1
             , UA);

        byte = writeBytesSerialPort(
             frameBufferSend, FRAME_SIZE_S);

        if (byte != FRAME_SIZE_S) {
            perror("Error␣writing␣bytes␣to␣serial
                 ␣port\n");
            free(frameBufferSend);
            free(frameBufferReceive);
            return -1;
        } else {
            printf("UA␣frame␣sent,␣bytes␣written␣
                 =␣%d\n",byte);
        }

        gettimeofday(&tv2, NULL);
        printf("Stopping␣counting␣now\n");

    } else if (role == LlRx) {
        printf("LLCLOSE:␣LlRx\n");
        unsigned char buffer_read = 0;
        int byte;
        state = S_WAITING_FLAG;

        while (state != S_STOP_STATE) {
            byte = readByteSerialPort(&
                 buffer_read);
            if (byte > 0) {
                switch (state) {
                    case S_WAITING_FLAG:
                        if (buffer_read == FLAG)
                             {
                            frameBufferReceive[0]
                                 = buffer_read;
                            state =
                                 S_WAITING_ADDR;
                        }
                        break;
                    case S_WAITING_ADDR:
                        if (buffer_read == A0) {
                            frameBufferReceive[1]
                                 = buffer_read;
                            state =
                                 S_WAITING_CTRL;
                        } else if (buffer_read !=
                             FLAG) {
                            state =
                                 S_WAITING_FLAG;
                        }
                        break;
```

```
                    case S_WAITING_CTRL:
                        if (buffer_read == DISC)
                             {
                            frameBufferReceive[2]
                                 = buffer_read;
                            state =
                                 S_WAITING_BCC1;
                        } else if (buffer_read ==
                             FLAG) {
                            state =
                                 S_WAITING_ADDR;
                        } else {
                            state =
                                 S_WAITING_FLAG;
                        }
                        break;
                    case S_WAITING_BCC1:
                        if (buffer_read == (
                             frameBufferReceive[1]
                              ^ frameBufferReceive
                             [2])) {
                            frameBufferReceive[3]
                                 = buffer_read;
                            state =
                                 S_WAITING_FLAG2;
                        } else if (buffer_read ==
                             FLAG) {
                            state =
                                 S_WAITING_ADDR;
                        } else {
                            state =
                                 S_WAITING_FLAG;
                        }
                        break;
                    case S_WAITING_FLAG2:
                        if (buffer_read == FLAG)
                             {
                            frameBufferReceive[4]
                                 = buffer_read;
                            state = S_STOP_STATE;
                        } else {
                            state =
                                 S_WAITING_FLAG;
                            break;
                        }

                    case S_STOP_STATE:
                        break;
                }
            }
        }

        // Create DISC frame
        buildFrameSupervision(frameBufferSend, A1
             , DISC);
        state = S_WAITING_FLAG;
        alarmCount = 0;

        while (state != S_STOP_STATE &&
             alarmCount < nRetransmissions) {

            // Sending DISC frame
            if (!alarmRinging) {
                lastAlarmCount = alarmCount;
                alarmRinging = TRUE;
                alarm(timeout);
                state = S_WAITING_FLAG;

                byte = writeBytesSerialPort(
                     frameBufferSend, FRAME_SIZE_S
                     );
                if (byte != FRAME_SIZE_S) {
                    perror("Error writing DISC
                         frame to serial port\n");
                    alarm(0);
                    if (alarmCount ==
                         lastAlarmCount){
                        alarmRinging = FALSE;
                        alarmCount++;
                        alarmInterrupted++;
                        printf("Alarm %d
                             interrupted\n",
                             alarmCount);
                    }
                    continue;
                } else {
                    printf("DISC frame sent,
                         bytes written = %d\n",
                         byte);
                }
            }

            // Reading UA frame
            byte = readByteSerialPort(&
                 buffer_read);

            if (byte > 0) {
                switch (state) {
                    case S_WAITING_FLAG:
                        if (buffer_read == FLAG)
                             {
                            frameBufferReceive[0]
                                 = buffer_read;
                            state =
                                 S_WAITING_ADDR;
                        }
                        break;
                    case S_WAITING_ADDR:
                        if (buffer_read == A1) {
                            frameBufferReceive[1]
                                 = buffer_read;
                            state =
                                 S_WAITING_CTRL;
                        } else if (buffer_read !=
                             FLAG) {
                            state =
                                 S_WAITING_FLAG;
                        }
                        break;
                    case S_WAITING_CTRL:
                        if (buffer_read == UA) {
                            frameBufferReceive[2]
                                 = buffer_read;
                            state =
                                 S_WAITING_BCC1;
                        } else if (buffer_read ==
                             FLAG) {
                            state =
                                 S_WAITING_ADDR;
                        } else {
                            state =
                                 S_WAITING_FLAG;
                        }
                        break;
                    case S_WAITING_BCC1:
                        if (buffer_read == (
                             frameBufferReceive[1]
                              ^ frameBufferReceive
                             [2])) {
                            frameBufferReceive[3]
```

```
                                = buffer_read;
                            state =
                                S_WAITING_FLAG2;
                        } else if (buffer_read ==
                             FLAG) {
                            state =
                                S_WAITING_ADDR;
                        } else {
                            state =
                                S_WAITING_FLAG;
                        }
                        break;
                    case S_WAITING_FLAG2:
                        if (buffer_read == FLAG)
                             {
                            frameBufferReceive[4]
                                 = buffer_read;
                            state = S_STOP_STATE;
                            alarm(0);
                            if (alarmCount ==
                                 lastAlarmCount){
                                alarmCount++;
                                alarmInterrupted
                                     ++;
                                printf("Alarm %d
                                     interrupted\n
                                     ", alarmCount
                                     );
                            }
                        } else {
                            state =
                                S_WAITING_FLAG;
                            break;
                        }

                    case S_STOP_STATE:
                        break;
                }
            }
        }
        TotalAlarmCount += alarmCount;
        if (state != S_STOP_STATE) {
            perror("ERROR: Timeout during
                 receiving UA frame\n");
            free(frameBufferSend);
            free(frameBufferReceive);
            return -1;
        }
        else {
            printf("UA frame received\n");
        }
    }

    free(frameBufferSend);
    free(frameBufferReceive);

    int clstat = closeSerialPort();

    if (clstat < 0) {
        perror("Error closing serial port\n");
        return clstat;
    }

    if (showStatistics == TRUE){
        if (role == LlTx){
            printf("Stats - Tx:\n");
        }
        else{
            printf("Stats - Rx:\n");
        }

        printf("Total alarms triggered: %d\n",
             TotalAlarmCount);
        printf("Alarms interrupted: %d\n",
             alarmInterrupted);
        printf("Alarms completed without
             interruption: %d\n", TotalAlarmCount
             - alarmInterrupted);

        printf ("Total time = %f seconds\n",
        (double) (tv2.tv_usec - tv1.tv_usec) /
             1000000 +
        (double) (tv2.tv_sec - tv1.tv_sec));
    }

    return clstat;
}
```

C. *Porta série*

- serial_port.h -

```
        // Serial port header.
// NOTE: This file must not be changed.

#ifndef _SERIAL_PORT_H_
#define _SERIAL_PORT_H_

// Open and configure the serial port.
// Returns -1 on error.
int openSerialPort(const char *serialPort, int
     baudRate);

// Restore original port settings and close the
     serial port.
// Returns -1 on error.
int closeSerialPort();

// Wait up to 0.1 second (VTIME) for a byte
     received from the serial port (must
// check whether a byte was actually received
     from the return value).
// Returns -1 on error, 0 if no byte was received
     , 1 if a byte was received.
int readByteSerialPort(unsigned char *byte);

// Write up to numBytes to the serial port (must
     check how many were actually
// written in the return value).
// Returns -1 on error, otherwise the number of
     bytes written.
int writeBytesSerialPort(const unsigned char *
     bytes, int numBytes);

#endif // _SERIAL_PORT_H_
```

- serial_port.c -

```
        // Serial port interface implementation
// DO NOT CHANGE THIS FILE

#include "serial_port.h"
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>
```

```c
// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source

int fd = -1;           // File descriptor for open serial port
struct termios oldtio; // Serial port settings to restore on closing

// Open and configure the serial port.
// Returns -1 on error.
int openSerialPort(const char *serialPort, int baudRate)
{
    // Open with O_NONBLOCK to avoid hanging when CLOCAL
    // is not yet set on the serial port (changed later)
    int oflags = O_RDWR | O_NOCTTY | O_NONBLOCK;
    fd = open(serialPort, oflags);
    if (fd < 0)
    {
        perror(serialPort);
        return -1;
    }

    // Save current port settings
    if (tcgetattr(fd, &oldtio) == -1)
    {
        perror("tcgetattr");
        return -1;
    }

    // Convert baud rate to appropriate flag
    tcflag_t br;
    switch (baudRate)
    {
    case 1200:
        br = B1200;
        break;
    case 1800:
        br = B1800;
        break;
    case 2400:
        br = B2400;
        break;
    case 4800:
        br = B4800;
        break;
    case 9600:
        br = B9600;
        break;
    case 19200:
        br = B19200;
        break;
    case 38400:
        br = B38400;
        break;
    case 57600:
        br = B57600;
        break;
    case 115200:
        br = B115200;
        break;
    default:
        fprintf(stderr, "Unsupported baud rate (must be one of 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200)\n");
        return -1;
    }

    // New port settings
    struct termios newtio;
    memset(&newtio, 0, sizeof(newtio));

    newtio.c_cflag = br | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;

    // Set input mode (non-canonical, no echo,...)
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 0; // Block reading
    newtio.c_cc[VMIN] = 1;  // Byte by byte

    tcflush(fd, TCIOFLUSH);

    // Set new port settings
    if (tcsetattr(fd, TCSANOW, &newtio) == -1)
    {
        perror("tcsetattr");
        close(fd);
        return -1;
    }

    // Clear O_NONBLOCK flag to ensure blocking reads
    oflags ^= O_NONBLOCK;
    if (fcntl(fd, F_SETFL, oflags) == -1)
    {
        perror("fcntl");
        close(fd);
        return -1;
    }

    // Done
    return fd;
}

// Restore original port settings and close the serial port.
// Returns -1 on error.
int closeSerialPort()
{
    // Restore the old port settings
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        perror("tcsetattr");
        return -1;
    }

    return close(fd);
}

// Wait up to 0.1 second (VTIME) for a byte received from the serial port (must
// check whether a byte was actually received from the return value).
// Returns -1 on error, 0 if no byte was received, 1 if a byte was received.
int readByteSerialPort(unsigned char *byte) {
    return read(fd, byte, 1);
}

// Write up to numBytes to the serial port (must check how many were actually
// written in the return value).
// Returns -1 on error, otherwise the number of bytes written.
int writeBytesSerialPort(const unsigned char *
```

```
     bytes, int numBytes)
{
    return write(fd, bytes, numBytes);
}
```