Protocolo de Ligação de Dados

(1.º Trabalho Laboratorial)

Redes de Computadores

LEIC - 10/2023

Turma 2 - Grupo 4

André dos Santos Faria Relva - up202108695@fe.up.pt

Rui Pedro Almeida da Silveira - up202108878@fe.up.pt

23 de outubro de 2023

1. Sumário

Este trabalho foi desenvolvido no âmbito da disciplina de Redes de Computadores e envolve a implementação de um protocolo de transferência de ficheiros através da porta série.

Com isto, foi-nos possível aprender sobre os principais dilemas que se apresentam na implementação de um protocolo de transferência de dados, assim como os potenciais *trade-off*s que podem ser feitos na sua especificação para maximizar a eficiência após devida experimentação.

2. Introdução

Neste trabalho foi desenvolvida uma aplicação de Linux que estabelece um protocolo de transferência de ficheiros através da porta série.

Posto isto, o presente relatório explorará os conceitos e observações relevantes a este protocolo ao longo das seguintes oito secções.

3. Arquitetura

Componentes Funcionais

O protocolo implementado neste trabalho está dividido em duas componentes funcionais: a *link layer* e a *application layer*.

Cada um destes componentes apresenta um nível distinto de abstração, de modo a compartimentalizar os algoritmos e funcionalidades em interfaces de baixo (*link layer*) e alto (*application layer*) nível.

No entanto, cada camada está intrinsecamente ligada: a camada superior depende necessariamente da camada inferior: através da implementação de processos de baixo nível numa camada especializada, é-nos possível abstrair o desenvolvimento de algoritmos que geram os dados enviados e recebidos da camada inferior—semelhante a uma *black box*—assegurando, deste modo, maior exatidão na semântica do programa, assim como maior facilidade não só na estruturação como também na reestruturação do programa, devido à sua modularidade.

Enquanto que a *application layer* está mais próxima ao utilizador, tratando do processamento de dados, a *link layer* é a que está mais próxima à máquina (*bare metal*) e implementa a interface de comunicação através da porta série.

Interface

```
Usage: ./main <SerialPort> <mode> <filename>
Example: ./main /dev/ttyS1 sender penguin.gif
Available modes:
       sender
       receiver
```

Acima está explícita a interface do programa, sendo necessário especificar a porta série a usar, o modo, e, caso o programa seja corrido em modo transmissor, o nome do ficheiro, uma vez que o protocolo implementado envia informações como o tamanho e nome original do ficheiro para além do seu conteúdo, não sendo necessário especificar um nome arbitrário para o ficheiro recebido quando corrido em modo recetor.

4. Estrutura do Código

O código do programa, tal como previamente dito, está dividido em duas camadas: a *application layer* e a *link layer*. No entanto, para uma melhor organização do projeto, as funcionalidades foram distribuídas entre múltiplos ficheiros.

De seguida será descrita a estruturação destas camadas.

Application Layer

```c
// sender.h / sender.c
int handle_sender(const char *serialPort, int role, int baudRate,
                  int nTries, int timeout, const char *filename)

// receiver.h / receiver.c
int handle_receiver(const char *serialPort, int role, int baudRate,
                    int nTries, int timeout, const char *filename);
```

Estas funções tratam da funcionalidade de transmissão e de receção de ficheiros, respetivamente.

Como podemos ver, ambas têm a mesma *function signature*, uma vez que estes serão os parâmetros posteriormente passados para a *link layer* para estabelecer a ligação.

Estes correspondem à seguinte *struct* definida para facilitar a encapsulação destes dados:

```c
// serialio.h
typedef struct {
   char serialPort[50];
   LinkLayerRole role;
   int baudRate;
   int nRetransmissions;
   int timeout;
} LinkLayer;
```

- `serialPort` - *C-string* contendo o caminho da porta série a ser utilizada para a ligação;
- `role` - *enum* pré-definido para enumerar ambos os modos do programa:

```c
typedef enum {
   LlTx, // Transmissor
```

```
    LlRx, // Recetor
} LinkLayerRole;
```

- `baudRate` - o *baud rate* definido para a transmissão, para assegurar a sincronização das operações *I/O*, correspondente a símbolos ou pulsos por segundo;
- `nRetransmissions` - o número máximo de tentativas de reconexão ao não receber resposta após a transmissão de um *frame*, por qualquer razão que seja;
- `timeout` - o tempo, em segundos, entre cada tentativa de retransmissão/reconexão acima descrita.

Como é possível notar, apenas `filename` não é utilizada na estrutura acima descrita, uma vez que apenas é relevante à *application layer*, que tratará de ler e de escrever para o ficheiro a informação recebida. Os restantes argumentos, tal como foi dito, serão passados para a *link layer*.

Olhando agora para o processamento de dados, foi empregado num ficheiro separado a funcionalidade de criação de pacotes, que serão posteriormente passados para a *link layer* para serem subsequentemente *framed* e enviados.

```
// packet.h / packet.c
size_t build_packet(PacketType type, unsigned char *buffer, const char
*filename, long filesize);
```

Esta função auxiliar facilita a construção de ambos os tipos de pacotes estabelecidos na especificação do projeto (ou `CONTROL` ou `DATA`), passados através do argumento `type`, tratando-se de um *enum* contendo um de esses mesmos tipos.

Aqui, `buffer` cumpre a função de *output*, devolvendo o *array* devidamente preenchido com os *bytes* do pacote e a quantidade de *bytes* escritos como valor de retorno, essencialmente correspondendo ao tamanho do *buffer* relevante.

É importante notar que `filename` e `filesize` só são relevantes em pacotes de controlo, podendo passar `NULL` quando se tratar de pacotes de dados.

Adicionalmente, existe uma variável global `file_content_buffer` que permite a esta função ler os conteúdos que foram lidos do ficheiro e subsequentemente escritos neste *buffer* previamente a ser chamada. O número total de *bytes* lidos estará disponível através de uma outra variável global `packet_data_size.`

Finalmente, foi desenvolvido um simples ficheiro com funções que imprimem para a consola informações de erro consoante o resultado das funções chamadas ao longo do programa. A principal razão da existência deste módulo deve-se à necessidade de existência de mensagens de erro para o utilizador poder ter consciência do estado do programa a qualquer altura, mas também para evitar a repetição de código, uma vez que estas funções são chamadas múltiplas vezes. Como se tratam de funções extremamente simples, apenas estarão visíveis no anexo do código.

Link Layer

No que diz respeito à *link layer*, temos várias componentes em movimento, cada uma indispensável ao correto funcionamento do programa.

Começaremos pelo protocolo de ligação, descrito a seguir:

```
// serialio.h / serialio.c
int llopen(LinkLayer *connParam);
int llwrite(const unsigned char *buffer, int bufSize);
int llread(unsigned char *packet);
int llclose(void);
```

Estas quatro funções são responsáveis por estabelecer, manter e fechar a ligação usada para a transmissão do ficheiro.

Tal como o nome sugere, `llopen` trata de abrir a ligação entre as duas máquinas usando os parâmetros contidos no *struct* já acima discutido.

As duas seguintes funções, `llwrite` e `llread` são usadas pelo modo transmissor e recetor, respetivamente.

Em `llwrite`, serão passados o *buffer* contendo o pacote construído na *application layer* e o tamanho desse *buffer* como argumentos, procedendo a função a efetuar *byte stuffing* e *framing* antes de enviar a informação e certificar-se da sua receção (ou, caso não seja possível, um eventual *timeout*) antes de retornar.

Por outro lado, em `llread`, o único argumento passado é um endereço de memória para o qual a função deve escrever os conteúdos do pacote recebido após efetuar *destuffing* e assegurar-se da inexistência de erros, sendo o valor de retorno o tamanho do pacote.

Finalmente, a função `llclose` simplesmente fecha a ligação após uma transmissão de ficheiro bem-sucedida. Esta função é chamada em ambos os modos de execução do programa, tendo dois possíveis ramos de execução dependendo desse mesmo modo.

Seguidamente, do mesmo modo que usámos uma função auxiliar para construir pacotes, também usámos uma para construir *frames*, descrita deste modo:

```
// frame.h / frame.c
int build_frame(FrameType type, unsigned char *buffer, int bufSize);
```

Passados como argumentos são os seguintes dados:

- `type` - o tipo de frame a construir dentro dos possíveis tipos definidos no seguinte *enum*:

```
// frame.h
typedef enum {
   SET,
   UA,
   RR0,
   RR1,
   REJ0,
   REJ1,
   DISC,
   INFO
```

```
} FrameType;
```

- `buffer` - um *array* de *bytes* contendo o pacote obtido da *application layer* com *byte-stuffing* já efetuado para ser *framed*;
- `bufSize` - o tamanho do *buffer* passado como argumento.

Ainda nestes mesmos ficheiros é implementada uma máquina de estados para o processamento de *frames* recebidos:

```
typedef struct {
   enum {
        START,
        FLAG_RCV,
        A_RCV,
        C_RCV,
        BCC_OK,
        STOP
   } state;
   TransitionTable transition;
} StateMachine;
```

Como podemos ver, está definido num *enum* cada estado da máquina de estados, e também é declarada uma tabela de transições, definida da seguinte forma:

```
// frame.h
typedef void (*_trans_func)(const unsigned char);
typedef _trans_func *TransitionTable;
```

Por outras palavras, a tabela de transições é um *array* de *function pointers* com a *function signature* definida como `_trans_func`, onde cada índice do *array* corresponde a um determinado estado e à sua função de de transição, que recebe como argumento um *byte* à medida que o *frame* recebido é processado.

Estas são as funções definidas para a tabela de transições:

```
// frame.h / frame.c
void _sm_start(const unsigned char data);
void _sm_flag_rcv(const unsigned char data);
void _sm_a_rcv(const unsigned char data);
void _sm_c_rcv(const unsigned char data);
void _sm_bcc_ok(const unsigned char data);
```

Uma vez que `STOP` é o estado final, não é definida uma função de transição para este estado, tendo cada função que faz uso da máquina de estados atenção a quando este estado é atingido para terminar o processamento de dados.

Finalmente, para tratar dos *timeouts* para as retransmissões especificadas no protocolo, usamos as funções `signal` e `alarm` das bibliotecas *unistd.h* e *signal.h* em conjunto com um *handler* que é executado após o tempo especificado para alterar uma *flag* e um *counter* global que o programa usará para saber quando o tempo especificado passou e quantas retransmissões já foram tentadas.

```
// alarm.h / alarm.c
void alarmHandler(int signal);
```

5. Protocolo

Nesta secção será descrito o protocolo de funcionamento do programa em cada camada de abstração.

Application Layer

O programa começa por ser iniciado com os devidos argumentos, quer em modo de transmissão quer em modo de receção.

Seguidamente, são chamadas as funções para o modo correspondente, e o programa transferirá o controlo à *link layer* para tentar estabelecer ligação através da porta série especificada.

Ao suceder, este procederá a abrir o ficheiro especificado e obter o seu tamanho total em *bytes*, no caso do transmissor. O recetor, entretanto, ficará simplesmente à espera da restante comunicação.

De seguida, será preparado o primeiro pacote de controlo com informações como o tamanho e o nome do ficheiro, que será sucessivamente enviado para a *link layer* que tratará de processá-lo antes de o enviar. É importante ter em atenção que as funções da *link layer* só retornam após confirmação obtida da outra máquina (exceto, claro, em caso de erro), podendo assim esta camada prosseguir sem ser necessário ter conhecimento do estado interno da *link layer*.

Logo a seguir, inicia-se um ciclo de envio de pacotes de dados contendo os conteúdos do ficheiro a ser transmitido do lado do transmissor, enquanto que do lado do recetor são consumidos os pacotes válidos recebidos e devidamente escritos para o ficheiro destino, sendo que a deteção de erros cabe à *link layer*.

Finalmente, após o transmissor confirmar a receção do último pacote de dados, envia um pacote de controlo, idêntico àquele enviado no início da transmissão, após o qual irá tentar terminar a ligação devidamente. Quer seja ou não possível ambos os lados terminarem a conexão com sucesso, o transmissor tem a certeza de que o recetor recebeu todos os dados devidamente.

Link Layer

Primeiramente, é importante delinear o processo de *framing* de pacotes, pois é pertinente a todas as funções da *link layer*.

Este mesmo processo da *link layer* passa por três fases. Em primeiro lugar, é aplicada a operação *bitwise* XOR a todos os *bytes* do pacote sucessivamente para gerar o campo de deteção de erros do pacote.

Seguidamente, inicia-se o processo de *byte stuffing*. Este tem o objetivo de omitir *bytes* com um significado especial para o programa de modo a não serem erroneamente interpretados pela máquina de estados; nomeadamente, o valor que corresponde à *flag* de início de um *frame*, e o caractere de escape. O processo passa por inserir o caractere de escape antes de cada um dos *bytes* especiais presentes no pacote antes de proceder a aplicar a operação XOR ao mesmo, tornando assim possível a operação inversa que será posteriormente efetuada pelo recetor.[1]

Finalmente, este pacote já processado é colocado entre os campos de cabeçalho definidos na especificação do *frame*, tendo estes o seu próprio campo de deteção de erros (BCC1).

De volta ao protocolo em si, `llopen` é a primeira função a ser chamada. Esta irá proceder à tentativa de transmissão de um SET *frame* no caso do transmissor. Do outro lado, o recetor estará à espera deste mesmo, cuja receção confirmará com um UA *frame*.

Após o estabelecimento da ligação entre as duas máquinas, a *application layer* chamará as funções `llwrite`—no caso do transmissor—e `llread`—no caso do recetor—sucessivamente.

Esta primeira passará por efetuar o processo de *framing* no pacote que recebe da *application layer* após o qual o enviará. A segunda função, entretanto, lerá um *byte* de cada vez num ciclo, subsequentemente fornecendo-o à máquina de estados.

Do modo como o protocolo foi implementado, e de acordo com as especificações do projeto, a máquina de estados não aceitará *frames* com erros no cabeçalho, acabando por retornar ao estado inicial nesses casos, o que é logicamente equivalente a ignorar o *frame*, o que irá incitar o transmissor a reenviar o pacote.

Caso não haja erros no cabeçalho, o recetor efetuará o processo inverso ao descrito acima, acabando com o pacote original antes de ser *framed*. Este, por sua vez, será percorrido para confirmar a ausência de erros, comparando o resultado da operação XOR com o valor já presente em BCC2. Caso não seja detetado erro, o pacote é aceite e enviado para a *application layer*, e um RRX *frame* será enviado ao transmissor para incitar o envio do pacote seguinte, sendo X a paridade oposta à do pacote recebido (0 ou 1). Caso contrário, é enviado um REJX *frame*, sendo X a paridade do pacote rejeitado.

Por fim, quando `llclose` é chamada, o transmissor enviará um DISC *frame*, esperando um *frame* semelhante vindo do recetor, confirmando a receção deste com o envio de um UA *frame*, acabando o transmissor por terminar a conexão após o seu envio e o recetor após a sua receção.

6. Validação

Para garantir o correto funcionamento do programa perante diferentes tipos de possíveis erros, foram efetuados vários testes em diferentes condições: introdução de ruído no sinal através de curto-circuito, interrupção arbitrária da ligação entre as portas série, rejeição

[1] **Nota:** No protocolo implementado no nosso grupo, o campo de deteção de erros do pacote (BCC2) acaba por não passar pelo processo de *byte stuffing*. Isto, no entanto, trata-se de um erro, uma vez que numa situação ideal isto não seria desejável, por existir uma pequena chance que o resultado da operação anterior acabará por resultar num dos dois valores especiais.

aleatória de pacotes válidos, diferentes *baud rates*, diferentes tamanhos de pacote e introdução de *delay* no processamento de pacotes.

Tal como verificado pelo docente durante a apresentação do projeto, o programa funcionou corretamente face a estas dificuldades.

7. Eficiência do protocolo de ligação de dados

Para testar a eficiência do protocolo perante os diferentes cenários mencionados acima, foram recolhidas amostras do tempo médio de transmissão de um pacote nesses mesmos cenários e calculada a eficiência, através da fórmula S = R / C, onde R é o número efetivo de *bits*/s obtido a partir do tempo médio de transmissão e C a capacidade de transmissão, definida pelo *baud rate*.

É importante ter em conta que os parâmetros base usados nos testes cujos resultados são demonstrados na seguinte tabela são: FER a 0%, *baud rate* a 9600B, tamanho do pacote a 512B e nenhum *delay* introduzido. São estes os valores efetivos em cada linha exceto quando dito o contrário.

Parâmetro	Quantidade	Tempo médio de transmissão	Eficiência (S)
FER	0% 5% 10%	552ms 1036ms 970ms	77,3% 41,2% 44,0%
Baud Rate	9600B 38400B 115200B	552ms 502ms 502ms	77,3% 21,2% 7,1%
Tamanho do pacote	512B 1024B	552ms 997ms	77,3% 85,6%
Delay introduzido	0s 1s	552ms 1552ms	77,3% 27,5%

No que conta aos parâmetros FER e *delay*, é imediatamente intuitivo que estes são imediatamente proporcionais à eficiência: quanto maior o FER, maior o número de retransmissões, que por sua vez aumenta o tempo de transmissão, enquanto que a introdução de *delay* aumenta o tempo de transmissão incondicionalmente.

No entanto, verificamos que ao aumentar o *baud rate* a eficiência diminui. Isto também é espectável, uma vez que o tamanho do pacote não aumentou com o *baud rate* nestes testes, o que faz com que o protocolo não tire o total proveito da capacidade disponível, diminuindo a eficiência.

Por fim, ao aumentar o tamanho do pacote, a eficiência aumenta mesmo que o tempo médio de transmissão aumente. No entanto, é de esperar que este aumento de eficiência diminua à medida que o tamanho aproxima o valor do *baud rate*. No caso aqui visto, são testados pacotes de 4096 *bits* e 8192 *bits*, ambos valores inferiores ao *baud rate*, o que significa que este aumento permite tirar mais proveito da capacidade da ligação.

8. Conclusão

Com este trabalho foi-nos possível aprofundar os nossos conhecimentos de protocolos de comunicação ao experienciar em primeira mão os problemas e filosofia estrutural envolvidos na implementação de um protocolo destes.

Finalmente, a partir dos resultados obtidos, propomos uma futura refatoração do programa que implementa um ajuste automático do tamanho base do pacote consoante a capacidade de ligação disponível. No entanto, serão necessários subsequentes testes para determinar o ponto a partir do qual o *trade-off* entre um maior pacote e uma maior eficiência já não se justifica.

Anexo I

- ***alarm.h***

```
#ifndef _ALARM_H_
#define _ALARM_H_

#define FALSE 0
#define TRUE 1

extern int alarmEnabled;
extern int alarmCount;

// Alarm function handler
void alarmHandler(int signal);

#endif
```

- ***error.h***

```
#ifndef _ERROR_H_
#define _ERROR_H_

#include <stdio.h>

int handle_llopen(int err);
int handle_llread(int err);
int handle_llwrite(int err);
int handle_llclose(int err);
int handle_fopen(FILE *err);
int handle_open(int err);
```

```
int handle_read(int err);
int handle_write(int err);
int handle_tcgetattr(int err);
int handle_tcsetattr(int err, int new);

#endif
```

- ***frame.h***

```
#ifndef _FRAME_H_
#define _FRAME_H_

#define SU_FRAME_SIZE 5
#define DATALESS_INFO_FRAME_SIZE 6
#define INFO_FRAME_DATA_OFFSET 4

#define SF_FLAG         0x7E
#define SENDER_ADDRESS  0x03
#define SET_CONTROL     0x03
#define UA_CONTROL      0x07
#define RR0_CONTROL     0x05
#define RR1_CONTROL     0x85
#define REJ0_CONTROL    0x01
#define REJ1_CONTROL    0x81
#define DISC_CONTROL    0x0B
#define INFO0_CONTROL   0x00
#define INFO1_CONTROL   0x40

#define BLOCK_CHECK(c) ((0x03 ^ c))

typedef void (*_trans_func)(const unsigned char);
typedef _trans_func *TransitionTable;

void _sm_start(const unsigned char data);
void _sm_flag_rcv(const unsigned char data);
void _sm_a_rcv(const unsigned char data);
void _sm_c_rcv(const unsigned char data);
void _sm_bcc_ok(const unsigned char data);

typedef enum
{
   SET,
   UA,
   RR0,
   RR1,
```

```
    REJ0,
    REJ1,
    DISC,
    INFO
} FrameType;

typedef struct {
    enum {
        START,
        FLAG_RCV,
        A_RCV,
        C_RCV,
        BCC_OK,
        STOP
    } state;
    TransitionTable transition;
} StateMachine;

extern StateMachine sm;
extern unsigned char rcv_control;
extern int info_frame_counter;

int build_frame(FrameType type, unsigned char *buffer, int bufSize);

#endif
```

- ***packet.h***

```
#ifndef _PACKET_H_
#define _PACKET_H_

#define DATA_OFFSET 3
#define PACKET_DATA_CONTROL     1
#define PACKET_START_CONTROL    2
#define PACKET_END_CONTROL      3
typedef enum {
    CONTROL,
    DATA
} PacketType;

extern unsigned char *file_content_buffer;
extern int packet_data_size;

size_t build_packet(PacketType type, unsigned char *buffer, const char
*filename, long filesize);
```

```
#endif
```

- ***receiver.h***

```
#ifndef _RECEIVER_H_
#define _RECEIVER_H_

#include <termios.h>

int handle_receiver(const char *serialPort, int role, int baudRate,
                    int nTries, int timeout, const char *filename);

#endif
```

- ***sender.h***

```
#ifndef _SENDER_H_
#define _SENDER_H_

#include "../include/frame.h"

int handle_sender(const char *serialPort, int role, int baudRate,
                  int nTries, int timeout, const char *filename);

#endif
```

- ***serialio.h***

```
#ifndef _SERIALIO_H_
#define _SERIALIO_H_

#include "frame.h"

#define MAX_PAYLOAD_SIZE 1000

typedef enum
{
   LlTx,
   LlRx,
} LinkLayerRole;

typedef struct
```

```
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;

int llopen(LinkLayer *connParam);
int llwrite(const unsigned char *buffer, int bufSize);
int llread(unsigned char *packet);
int llclose(void);

#endif
```

- ***alarm.c***

```
#include <stdio.h>

#include "../include/alarm.h"

int alarmEnabled = FALSE;
int alarmCount = 0;

void alarmHandler(int signal)
{
    alarmEnabled = FALSE;
    alarmCount++;
}
```

- ***error.c***

```
#include "../include/error.h"

int handle_llopen(int err)
{
    if (err != -1) return 0;
    printf("[ERROR] Failed to establish connection.\n");
    return -1;
}

int handle_llwrite(int err)
{
    if (err != -1) return 0;
    printf("[ERROR] Failed to send frame.\n");
```

```
    return -1;
}

int handle_llread(int err)
{
    if (err != -1) return 0;
    printf("[ERROR] Failed to receive frame.\n");
    return -1;
}

int handle_llclose(int err)
{
    if (err != -1) return 0;
    printf("[ERROR] Failed to disconnect.\n");
    return -1;
}

int handle_fopen(FILE *err)
{
    if (err != NULL) return 0;

    printf("[ERROR] Failed to open the file.\n");

    #ifdef DEBUG
    perror("[DEBUG] fopen");
    #endif

    return -1;
}

int handle_open(int err)
{
    if (err != -1) return 0;

    printf("[ERROR] Unable to open serial port device.\n");

    #ifdef DEBUG
    perror("[DEBUG] open");
    #endif

    return -1;
}

int handle_read(int err)
{
    if (err != -1) return 0;

    printf("[ERROR] Unable to read from file descriptor.\n");
```

```
    #ifdef DEBUG
    perror("[DEBUG] read");
    #endif

    return -1;
}

int handle_write(int err)
{
    if (err != -1) return 0;

    printf("[ERROR] Failed to write to file descriptor");

    #ifdef DEBUG
    perror("[DEBUG] write");
    #endif

    return -1;
}

int handle_tcgetattr(int err)
{
    if (err != -1) return 0;

    printf("[ERROR] Unable to save current port settings.");

    #ifdef DEBUG
    perror("[DEBUG] tcgetattr");
    #endif

    return -1;
}

int handle_tcsetattr(int err, int new)
{
    if (err != -1) return 0;
    if (new) printf("[ERROR] Failed to set new port settings.\n");
    else printf("[ERROR] Failed to restore the old port settings.\n");

    #ifdef DEBUG
    perror("[DEBUG] tcsetattr");
    #endif

    return -1;
}
```

- ***frame.c***

```c
#include <math.h>
#include <string.h>

#include "../include/frame.h"

#define DATA_CONTROL 1
#define CONTROL_FILESIZE 0
#define CONTROL_FILENAME 1

int info_frame_counter = 0;

int build_frame(FrameType type, unsigned char *buffer, int bufSize)
{
   buffer[0] = SF_FLAG;
   buffer[1] = SENDER_ADDRESS;
   buffer[bufSize - 1] = SF_FLAG;

   switch (type) {
       case SET:
       {
           buffer[2] = SET_CONTROL;
           break;
       }
       case UA:
       {
           buffer[2] = UA_CONTROL;
           break;
       }
       case RR0:
       {
           buffer[2] = RR0_CONTROL;
           break;
       }
       case RR1:
       {
           buffer[2] = RR1_CONTROL;
           break;
       }
       case REJ0:
       {
           buffer[2] = REJ0_CONTROL;
           break;
       }
       case REJ1:
       {
```

```
            buffer[2] = REJ1_CONTROL;
            break;
        }
        case DISC:
        {
            buffer[2] = DISC_CONTROL;
            break;
        }
        case INFO:
        {
            buffer[2] = (info_frame_counter++ % 2) << 6;
            break;
        }
        default:
            return -1;
    }

    buffer[3] = BLOCK_CHECK(buffer[2]);

    return 0;
}

void _sm_start(const unsigned char data)
{
    if (data == SF_FLAG) sm.state = FLAG_RCV;
}

void _sm_flag_rcv(const unsigned char data)
{
    switch (data)
    {
        case SENDER_ADDRESS:
        {
            sm.state = A_RCV;
            return;
        }
        case SF_FLAG:
        {
            return;
        }
        default:
        {
            sm.state = START;
            return;
        }
    }
```

```
}

void _sm_a_rcv(const unsigned char data)
{
   switch (data)
   {
       case SF_FLAG:
       {
           sm.state = FLAG_RCV;
           return;
       }
       case SET_CONTROL:
       case UA_CONTROL:
       case RR0_CONTROL:
       case RR1_CONTROL:
       case REJ0_CONTROL:
       case REJ1_CONTROL:
       case INFO0_CONTROL:
       case INFO1_CONTROL:
       case DISC_CONTROL:
       {
           sm.state = C_RCV;
           rcv_control = data;
           return;
       }
       default:
       {
           sm.state = START;
           return;
       }
   }
}

void _sm_c_rcv(const unsigned char data)
{
   switch (data)
   {
       case SF_FLAG:
       {
           sm.state = FLAG_RCV;
           return;
       }
       case BLOCK_CHECK(SET_CONTROL):
       case BLOCK_CHECK(UA_CONTROL):
       case BLOCK_CHECK(RR0_CONTROL):
       case BLOCK_CHECK(RR1_CONTROL):
```

```
        case BLOCK_CHECK(REJ0_CONTROL):
        case BLOCK_CHECK(REJ1_CONTROL):
        case BLOCK_CHECK(DISC_CONTROL):
        {
            sm.state = BCC_OK;
            return;
        }
        case BLOCK_CHECK(INFO0_CONTROL):
        case BLOCK_CHECK(INFO1_CONTROL):
        {
            if (BLOCK_CHECK(rcv_control) == data) {
                sm.state = BCC_OK;
                return;
            }
        }
        default:
            sm.state = START;
            return;
    }
}

void _sm_bcc_ok(const unsigned char data)
{
    switch (data)
    {
        case SF_FLAG:
        {
            sm.state = STOP;
            return;
        }
        default:
            if (rcv_control == INFO0_CONTROL ||
                rcv_control == INFO1_CONTROL)
                return;

            sm.state = START;
            return;
    }
}
```

- ***packet.c***

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

```c
#include <stdio.h>

#include "../include/packet.h"

#define CONTROL_FILESIZE 0
#define CONTROL_FILENAME 1

int control_packet_counter = 0;

size_t build_packet(PacketType type, unsigned char *buffer, const char
*filename, long filesize)
{
   switch (type) {
       case CONTROL:
       {
           int filesize_bytes = (int)ceil(log2((double)(filesize + 1)) /
8);
           int filename_bytes = strlen(filename) + 1;
           size_t packet_size = filesize_bytes + filename_bytes + 5;

           buffer[0] = (control_packet_counter++ % 2) + 2;        // 2 -
start; 3 - end
           buffer[1] = CONTROL_FILESIZE;
           buffer[2] = filesize_bytes;
           for (int i = filesize_bytes; i > 0; i--)
           {
               buffer[3 + filesize_bytes - i] = (filesize >> (8 * (i -
1))) & 0xFF;
           }
           unsigned char next = 3 + filesize_bytes;
           buffer[next] = CONTROL_FILENAME;
           buffer[next + 1] = filename_bytes;
           (void*)strcpy((char*)(buffer + next + 2), filename);

           return packet_size;
       }
       case DATA:
       {
           buffer[0] = PACKET_DATA_CONTROL;
           buffer[1] = (packet_data_size >> 8) & 0xFF;
           buffer[2] = packet_data_size & 0xFF;
           (void*)memcpy(buffer + 3, file_content_buffer,
packet_data_size);
           return packet_data_size + 3;
       }
       default:
```

```
            return -1;
    }

    return 0;
}
```

- ***receiver.c***

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

#include "../include/receiver.h"
#include "../include/serialio.h"
#include "../include/error.h"
#include "../include/packet.h"

#define EXIT_FAILURE  1
#define EXIT_SUCCESS  0

#define TRUE 1
#define FALSE 0

int handle_receiver(const char *serialPort, int role, int baudRate,
                    int nTries, int timeout, const char *filename)
{
    LinkLayer ll;
    memset(&ll, 0, sizeof(ll));

    (void*)strncpy(ll.serialPort, serialPort, 50);
    ll.role = (LinkLayerRole)role;
    ll.baudRate = baudRate;
    ll.nRetransmissions = nTries;
    ll.timeout = timeout;
    if (handle_llopen(llopen(&ll))) return -1;

    FILE *dest = NULL;

    if (filename != NULL) {
        dest = fopen(filename, "w");
        if (handle_fopen(dest)) return -1;
    }

    unsigned char buffer[MAX_PAYLOAD_SIZE];
```

```
    int bytes;
    do {
        bytes = llread(buffer);
        if (handle_llread(bytes)) return -1;
    } while (bytes == 0);

    if (buffer[0] != PACKET_START_CONTROL) return -1;

    unsigned char filesize_size = buffer[2];
    long filesize = 0;

    for (int i = 0; i < filesize_size; i ++) {
        filesize += (buffer[3 + i] << (8 * (filesize_size - i - 1)));
    }

    unsigned char filename_size = buffer[4 + filesize_size];
    char filename_cmp[50] = {'\0'};
    (void*)strncpy(filename_cmp, (char*)(buffer + filesize_size + 5),
filename_size);

    if (dest == NULL) {
        dest = fopen(filename_cmp, "w");
        if (handle_fopen(dest)) return -1;
    }


    do {
        bytes = llread(buffer);
        if (handle_llread(bytes)) return -1;
        if (bytes == 0) continue;

        switch (buffer[0]) {
            case PACKET_DATA_CONTROL:
            {
                int data_size = 256 * buffer[1] + buffer[2];
                (void)fwrite(buffer + 3, 1, data_size, dest);
                break;
            }
            case PACKET_END_CONTROL:
            {
                (void)fclose(dest);
                goto finish;
            }
        }
    } while (1);
```

```
finish:

    if (handle_llclose(llclose())) return -1;
    printf("[INFO] Successfully disconnected.\n");
    return 0;
}
```

- ***sender.c***

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

#include "../include/sender.h"
#include "../include/serialio.h"
#include "../include/alarm.h"
#include "../include/packet.h"
#include "../include/error.h"

#define ALARM_TIMER 3
#define MAX_RETRIES 3

#define EXIT_FAILURE  1
#define EXIT_SUCCESS  0

int packet_data_size;
unsigned char *file_content_buffer;

int handle_sender(const char *serialPort, int role, int baudRate,
                  int nTries, int timeout, const char *filename)
{
    LinkLayer ll;
    memset(&ll, 0, sizeof(ll));

    (void*)strncpy(ll.serialPort, serialPort, 50);
    ll.role = (LinkLayerRole)role;
    ll.baudRate = baudRate;
    ll.nRetransmissions = nTries;
    ll.timeout = timeout;
    if (handle_llopen(llopen(&ll))) return -1;

    FILE *source = fopen(filename, "r");
    if (handle_fopen(source)) return -1;
```

```
    (void)fseek(source, 0L, SEEK_END);
    long size = ftell(source);
    (void)fseek(source, 0L, SEEK_SET);

    unsigned char buffer[MAX_PAYLOAD_SIZE];
    size_t bytes = build_packet(CONTROL, buffer, filename, size);
    if (handle_llwrite(llwrite(buffer, bytes))) return -1;

    packet_data_size = 512;
    file_content_buffer = buffer + DATA_OFFSET;

    while (!feof(source))
    {
        int read_bytes = fread(file_content_buffer, 1, packet_data_size,
source);
        if (read_bytes < packet_data_size && ferror(source)) {
            printf("[ERROR] Failed reading file.\n");
            return -1;
        }

        packet_data_size = read_bytes;

        bytes = build_packet(DATA, buffer, NULL, 0);
        if (handle_llwrite(llwrite(buffer, bytes))) return -1;
    }

    bytes = build_packet(CONTROL, buffer, filename, size);
    if (handle_llwrite(llwrite(buffer, bytes))) return -1;

    fclose(source);

    if (handle_llclose(llclose())) return -1;
    printf("[INFO] Successfully disconnected.\n");
    return 0;
}
```

- ***serialio.c***

```
#include <stdlib.h>
#include <termios.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
```

```c
#include "../include/serialio.h"
#include "../include/alarm.h"
#include "../include/error.h"

#define ESCAPE 0x7D
#define STUFF(n) (n ^ 0x20)
#define DESTUFF(n) STUFF(n)

StateMachine sm;
int fd;
unsigned char rcv_control;
LinkLayer ll;
struct termios oldtio;

int llopen(LinkLayer *connParam)
{
   ll = *connParam;

   // Set port config
   fd = open(connParam->serialPort, O_RDWR | O_NOCTTY | O_NONBLOCK);

   if (handle_open(fd)) exit(EXIT_FAILURE);

   struct termios newtio;

   // Save current port settings
   if (handle_tcgetattr(tcgetattr(fd, &oldtio))) exit(EXIT_FAILURE);

   // Clear struct for new port settings
   memset(&newtio, 0, sizeof(newtio));

   newtio.c_cflag = connParam->baudRate | CS8 | CLOCAL | CREAD;
   newtio.c_iflag = IGNPAR;
   newtio.c_oflag = 0;

   // Set input mode
   newtio.c_lflag = 0;          // Non-canonical
   newtio.c_cc[VTIME] = 0;      // Inter-character timer unused
   newtio.c_cc[VMIN] = 0;       // Non-blocking read

   // Flush data received but not read
   tcflush(fd, TCIOFLUSH);

   // Set new port settings
   if (handle_tcsetattr(tcsetattr(fd, TCSANOW, &newtio), TRUE))
```

```
exit(EXIT_FAILURE);

    printf("[INFO] New termios structure set.\n");

    // Set up state machine
    sm.transition = calloc(5, sizeof(_trans_func));

    sm.transition[START] = _sm_start;
    sm.transition[FLAG_RCV] = _sm_flag_rcv;
    sm.transition[A_RCV] = _sm_a_rcv;
    sm.transition[C_RCV] = _sm_c_rcv;
    sm.transition[BCC_OK] = _sm_bcc_ok;

    sm.state = START;

    printf("[INFO] State machine set up.\n");

    (void)signal(SIGALRM, alarmHandler);

    switch (connParam->role)
    {
        case LlTx:
        {
            unsigned char frame[SU_FRAME_SIZE];
            build_frame(SET, frame, SU_FRAME_SIZE);
            if (handle_write(write(fd, frame, SU_FRAME_SIZE))) return -1;
            while (alarmCount < connParam->nRetransmissions)
            {
                unsigned char packet;
                if (handle_read(read(fd, &packet, 1))) return -1;
                sm.transition[sm.state](packet);
                if (sm.state == STOP && rcv_control != UA_CONTROL)
sm.state = START;
                else if (sm.state == STOP) break;
                if (alarmEnabled) continue;
                if (alarmCount > 0) {
                        printf("[INFO] Resending SET frame to attempt
connection... (Tries left: %d)\n",
                                connParam->nRetransmissions - alarmCount
+ 1);
                        if (handle_write(write(fd, frame,
SU_FRAME_SIZE))) return -1;
                }
                alarm(connParam->timeout);
                alarmEnabled = TRUE;
            }
```

```
            alarm(0);
            if (rcv_control != UA_CONTROL) return -1;
            if (alarmCount < connParam->nRetransmissions) return 0;
            printf("[INFO] Max retries reached. Timed out.\n");
            return -1;
        }
        case LlRx:
        {
            // Receiver is supposed to wait
            // alarm(ll.timeout);
            // alarmEnabled = TRUE;

            while (sm.state != STOP)
            {
                // if (alarmEnabled == FALSE) return -1;

                unsigned char packet;
                int bytes = read(fd, &packet, 1);
                if (handle_read(bytes)) return -1;

                if (bytes == 0) continue;
                sm.transition[sm.state](packet);
            }

            // alarm(0);
            // alarmCount = 0;
            // alarmEnabled = FALSE;

            unsigned char frame[SU_FRAME_SIZE];
            build_frame(UA, frame, SU_FRAME_SIZE);
            if (handle_write(write(fd, frame, SU_FRAME_SIZE))) return -1;

            return 0;
        }
    }
    return -1;
}

int llwrite(const unsigned char *buffer, int bufSize)
{
    sm.state = START;

    unsigned char bcc2 = buffer[0];

    for (int i = 1; i < bufSize; i++) {
        bcc2 ^= buffer[i];
```

```
    }

    int illegal_counter = 0;
    for (int i = 0; i < bufSize; i++) {
        if (buffer[i] == ESCAPE || buffer[i] == SF_FLAG) {
            illegal_counter++;
        }
    }

    int stuffed_size = bufSize + illegal_counter +
DATALESS_INFO_FRAME_SIZE;
    unsigned char *stuffed = calloc(stuffed_size, 1);

    illegal_counter = 0;
    for (int i = 0; i < bufSize; i++) {
        if (buffer[i] == ESCAPE || buffer[i] == SF_FLAG) {
            stuffed[i + illegal_counter + INFO_FRAME_DATA_OFFSET] =
ESCAPE;
            stuffed[i + illegal_counter + INFO_FRAME_DATA_OFFSET + 1] =
STUFF(buffer[i]);
            illegal_counter++;
        } else {
            stuffed[i + illegal_counter + INFO_FRAME_DATA_OFFSET] =
buffer[i];
        }
    }

    stuffed[stuffed_size - 2] = bcc2;

    (void)build_frame(INFO, stuffed, stuffed_size);

    if (handle_write(write(fd, stuffed, stuffed_size))) {
        alarm(0);
        free(stuffed);
        return -1;
    }

    (void)signal(SIGALRM, alarmHandler);

    unsigned char expected = ((info_frame_counter - 1) % 2 == 0) ?
RR1_CONTROL : RR0_CONTROL;

    int err;

    while (alarmCount < ll.nRetransmissions) {
        unsigned char packet;
```

```
        err = handle_read(read(fd, &packet, 1));
        if (err) break;
        sm.transition[sm.state](packet);

        if (sm.state == STOP) {
            sm.state = START;
            if (rcv_control == expected) break;
            if (rcv_control != (((info_frame_counter - 1) % 2 == 0) ?
REJ0_CONTROL : REJ1_CONTROL)) continue;
            alarm(0);
            alarmCount = 0;
            alarmEnabled = FALSE;
            printf("[INFO] Frame rejected. Resending.\n");
            err = handle_write(write(fd, stuffed, stuffed_size));
            if (err) break;
        }

        if (alarmEnabled) continue;
        if (alarmCount > 0) {
                printf("[INFO] Resending I frame... (Tries left: %d)\n",
                       ll.nRetransmissions - alarmCount + 1);
                err = handle_write(write(fd, stuffed, stuffed_size));
                if (err) break;
        }
        alarm(ll.timeout);
        alarmEnabled = TRUE;
    }
    alarm(0);
    alarmCount = 0;
    alarmEnabled = FALSE;
    free(stuffed);
    if (err || rcv_control != expected) return -1;
    if (alarmCount < ll.nRetransmissions) return 0;
    printf("[INFO] Max retries reached. Timed out.\n");
    return -1;
}

int llread(unsigned char *packet)
{
    sm.state = START;

    unsigned char data;
    int bytes;
    int counter = 0;

    // No longer needed (refer below)
```

```
// alarm(ll.timeout);
// alarmEnabled = TRUE;

do {
    bytes = read(fd, &data, 1);
    if (handle_read(bytes)) return -1;

    // Wrong code in presentation
    // if (!alarmEnabled) return -1;

    if (bytes == 0) continue;

    int is_packet_data = (sm.state == BCC_OK) ? TRUE : FALSE;
    sm.transition[sm.state](data);

    if (is_packet_data == TRUE && sm.state == BCC_OK) {
        packet[counter++] = data;
    }
} while (sm.state != STOP);

// No longer needed (refer above)
// alarm(0);
// alarmCount = 0;
// alarmEnabled = FALSE;

const unsigned char bcc2 = packet[--counter];

if (counter == 0) return 0;

int escapes = 0;
for (int i = 0; i < counter; i++) {
    if (packet[i] == ESCAPE) escapes++;
}

int destuffed_size = counter - escapes;
unsigned char* destuffed = calloc(destuffed_size, 1);

escapes = 0;
for (int i = 0; i < destuffed_size; i++) {
    if (packet[i + escapes] == ESCAPE) {
        destuffed[i] = DESTUFF(packet[i + escapes + 1]);
        escapes++;
    } else {
        destuffed[i] = packet[i + escapes];
    }
}
```

```
    unsigned char bcc2_cmp = destuffed[0];

    for (int i = 1; i < destuffed_size; i++) {
        bcc2_cmp ^= destuffed[i];
    }

    memcpy(packet, destuffed, destuffed_size);
    free(destuffed);

    unsigned char frame[SU_FRAME_SIZE];
    if (bcc2 == bcc2_cmp) {
        (void)build_frame((rcv_control == INFO0_CONTROL) ? RR1 : RR0,
frame, SU_FRAME_SIZE);
        if (handle_write(write(fd, frame, SU_FRAME_SIZE))) return -1;

        return destuffed_size;
    } else {
        (void)build_frame((rcv_control == INFO0_CONTROL) ? REJ0 : REJ1,
frame, SU_FRAME_SIZE);
        if (handle_write(write(fd, frame, SU_FRAME_SIZE))) return -1;

        rcv_control = 0xFF;
        return 0;
    }
}

int llclose(void)
{
    sm.state = START;
    alarmCount = 0;
    alarmEnabled = FALSE;

    (void)signal(SIGALRM, alarmHandler);

    printf("[INFO] Transmission finished. Attempting disconnection.\n");

    switch (ll.role) {
        case LlTx:
        {
            unsigned char frame[SU_FRAME_SIZE];
            (void)build_frame(DISC, frame, SU_FRAME_SIZE);
            if (handle_write(write(fd, frame, SU_FRAME_SIZE))) return -1;

            while (alarmCount < ll.nRetransmissions)
            {
```

```
                unsigned char packet;
                if (handle_read(read(fd, &packet, 1))) return -1;
                sm.transition[sm.state](packet);
                if (sm.state == STOP && rcv_control != DISC_CONTROL)
sm.state = START;
                else if (sm.state == STOP) break;
                if (alarmEnabled) continue;
                if (alarmCount > 0) {
                    printf("[INFO] Resending DISC frame.. (Tries left:
%d)\n",
                           ll.nRetransmissions - alarmCount + 1);
                    if (handle_write(write(fd, frame, SU_FRAME_SIZE)))
return -1;
                }
                alarm(ll.timeout);
                alarmEnabled = TRUE;
            }

            alarm(0);
            free(sm.transition);

            if (alarmCount == ll.nRetransmissions) {
                printf("[INFO] Max retries reached. Disconnecting...\n");
            } else {
                (void)build_frame(UA, frame, SU_FRAME_SIZE);
                if (handle_write(write(fd, frame, SU_FRAME_SIZE))) return
-1;
                sleep(1);
            }

            if (handle_tcsetattr(tcsetattr(fd, TCSANOW, &oldtio) == -1,
FALSE)) exit(EXIT_FAILURE);

            close(fd);

            return (rcv_control == DISC_CONTROL && alarmCount <
ll.nRetransmissions) ? 0 : -1;

        }
        case LlRx:
        {
            int skip = FALSE;

            alarm(ll.timeout);
            alarmEnabled = TRUE;
```

```
            while (!skip)
            {
                unsigned char packet;
                if (handle_read(read(fd, &packet, 1))) return -1;
                sm.transition[sm.state](packet);

                if (alarmEnabled == FALSE) skip = TRUE;
                if (sm.state != STOP) continue;
                if (rcv_control == DISC_CONTROL) break;
                sm.state = START;
            }

            if (skip) printf("[INFO] DISC frame not received. Forcibly
disconnecting...\n");

            alarm(0);
            alarmCount = 0;
            alarmEnabled = FALSE;
            sm.state = START;

            unsigned char frame[SU_FRAME_SIZE];
            (void)build_frame(DISC, frame, SU_FRAME_SIZE);
            if (handle_write(write(fd, frame, SU_FRAME_SIZE))) return -1;

            while (alarmCount < ll.nRetransmissions)
            {
                unsigned char packet;
                if (handle_read(read(fd, &packet, 1))) return -1;
                sm.transition[sm.state](packet);
                if (sm.state == STOP && rcv_control != UA_CONTROL)
sm.state = START;
                else if (sm.state == STOP) break;
                if (alarmEnabled) continue;
                if (alarmCount > 0) {
                    printf("[INFO] Awaiting UA frame.. (Tries left:
%d)\n",
                           ll.nRetransmissions - alarmCount + 1);
                }
                alarm(ll.timeout);
                alarmEnabled = TRUE;
            }

            alarm(0);
            free(sm.transition);

            if (alarmCount == ll.nRetransmissions) {
```

```
                printf("[INFO] Max retries reached. Exiting...\n");
            }

            if (handle_tcsetattr(tcsetattr(fd, TCSANOW, &oldtio) == -1,
FALSE)) exit(EXIT_FAILURE);

            close(fd);

            return (rcv_control == UA_CONTROL && alarmCount <
ll.nRetransmissions) ? 0 : -1;
        }
        default:
            if (handle_tcsetattr(tcsetattr(fd, TCSANOW, &oldtio) == -1,
FALSE)) exit(EXIT_FAILURE);

            close(fd);
            free(sm.transition);

            return -1;
    }
}
```

- ***main.c***

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <termios.h>

#include "../include/sender.h"
#include "../include/receiver.h"

#define BAUDRATE 9600
#define N_TRIES 3
#define TIMEOUT 3
typedef enum {
    SENDER,
    RECEIVER
} RunMode;

int main(int argc, char *argv[]) {

    if (argc < 3)
    {
        printf("[ERROR] Incorrect program usage.\n"
```

```
            "Usage: %s <SerialPort> <mode> <filename>\n"
            "Example: %s /dev/ttyS1 sender penguin.gif\n"
            "Available modes:\n"
            "\tsender\n"
            "\treceiver\n",
            argv[0],
            argv[0]);
    exit(EXIT_FAILURE);
}

const char *serialPort = argv[1];
const char *mode_cmp = argv[2];

const char *receiver_mode = "receiver";
const char *sender_mode = "sender";

RunMode mode;

if (!strcmp(mode_cmp, receiver_mode)) mode = RECEIVER;
else if (!strcmp(mode_cmp, sender_mode)) mode = SENDER;
else {
    printf("[ERROR] Invalid mode.\n");
    exit(EXIT_FAILURE);
}

if (argc < 4 && mode == SENDER) {
    printf("[ERROR] Must specify file as sender.\n");
    exit(EXIT_FAILURE);
}

char *filename = NULL;

if (argc == 4 || mode == SENDER) filename = argv[3];

printf("[INFO] Starting link-layer protocol application\n"
       "  - Serial port: %s\n"
       "  - Role: %s\n"
       "  - Baudrate: %d\n"
       "  - Number of tries: %d\n"
       "  - Timeout: %d\n"
       "  - Filename: %s\n",
       serialPort,
       mode_cmp,
       BAUDRATE,
       N_TRIES,
       TIMEOUT,
       filename);
```

```
   int err;

   switch (mode) {
       case SENDER:
           err = handle_sender(serialPort, (int)mode, BAUDRATE, N_TRIES,
TIMEOUT, filename);
           break;
       case RECEIVER:
           err = handle_receiver(serialPort, (int)mode, BAUDRATE, N_TRIES,
TIMEOUT, filename);
           break;
   }

   return err;
}
```