7 de Novembro 2018

Faculdade de Engenharia da Universidade do Porto

Protocolo de Ligação de Dados

Redes de Computadores – Turma 3

Trabalho 1

Rui Jorge Leão Guedes – up201603854

Luís Alvela Duarte Mendes – up201605769

João Fernando da Costa Meireles Barbosa – up201604156

Índice

Sumário

No contexto da disciplina de Redes de Computadores, pretendeu-se implementar um protocolo de ligação de dados capaz de transferir ficheiros de um computador para outro, ligados por uma porta de série assíncrona.

Ao concluir o trabalho, conseguiu-se fornecer um serviço de comunicação de dados robusto e fiável entre dois sistemas, compreendendo assim como implementar e utilizar as componentes essenciais de um protocolo de ligação de dados com deteção de erros e possíveis perdas de informação.

Introdução

Este trabalho tem como objetivo principal a implementação duma aplicação que transfere ficheiros entre dois computadores, usando como meio de transmissão uma Porta Série RS-232. Para este efeito, foi criada uma pilha de dois protocolos; um protocolo de ligação de dados, responsável por fornecer um serviço de comunicação fiável a um protocolo de aplicação, responsável pela transferência do ficheiro.

Este relatório pretende documentar todo o trabalho realizado, seguindo a seguinte ordem estrutural:

- **Arquitetura e Estrutura do Código:** Apresentação e descrição dos componentes do sistema, detalhando as principais estruturas de dados, funções e interfaces de cada um.

- **Casos De Uso Principais:** Identificação destes casos, apresentando as respectivas sequências de chamada de funções.

- **Protocolo de Ligação Lógica:** Identificação dos principais aspectos funcionais com descrição da estratégia de implementação dos mesmos.

- **Protocolo de Aplicação:** Identificação dos principais aspectos funcionais com descrição da estratégia de implementação dos mesmos.

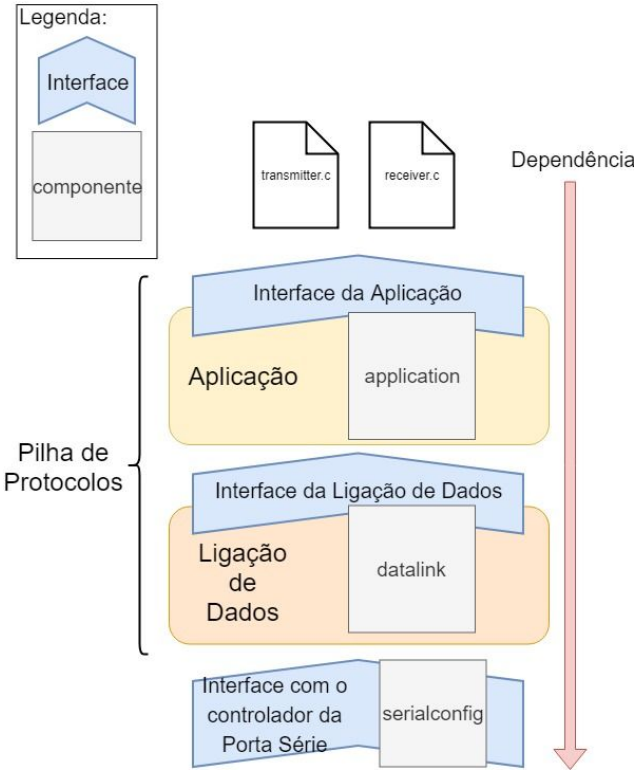
- **Validação:** Descrição dos testes efetuados.

- **Eficiência do Protocolo de Comunicação:** Análise estatística da eficiência do protocolo, usando a caracterização teórica como termo de comparação.

- **Conclusões:** Síntese da informação apresentada e reflexão sobre os objetivos de aprendizagem alcançados.

Arquitetura e Estrutura do Código

O sistema está estruturado numa arquitetura em camadas. No topo, dois programas (*transmitter.c* e *receiver.c*) acedem à pilha de protocolos para a transmissão e receção de ficheiros. Cada um dos dois protocolos da pilha tem um componente associado (*application* e *datalink*). A interface para os serviços fornecidos por estas duas camadas está contido no componente respetivo. Existe também um terceiro componente (*serialconfig*), composto por uma interface com o controlador da porta série. Cada um dos três componentes é composto por dois ficheiros; um ficheiro de cabeçalho (*header file)*, que contém a declaração das constantes, das variáveis globais e dos protótipos das funções, e um ficheiro de código fonte (*source file*), que contém a definição das funções declaradas.



Protocolo da Aplicação: *application*

A API da camada da Aplicação é constituída por duas funções:

- `int send_file(char * port, char * filename, char * file_content, int length);`
- `int receive_file(char * port);`

Cada uma destas funções, chamadas respectivamente pelo *transmitter.c* e pelo *receiver.c*, encapsulam completamente a funcionalidade de enviar/receber um ficheiro, logo não existe a necessidade de uma estrutura de dados associada ao componente. Ambas as funções necessitam da indicação da porta a usar e, no caso de envio, informações essenciais acerca do ficheiro (nome, conteúdo e o tamanho).

Associadas ao componente estão também quatro funções, cujo acesso é efetuado pelas funções anteriores, que tratam do envio e recepção dos pacotes de controlo e de dados:

- `int send_control_packet(int fd, int type, char * filename, unsigned int length);`
- `int receive_control_packet(int fd, int type, char * filename, unsigned int * file_length);`
- `int send_data_packet(int fd, unsigned int N, char * buffer, unsigned int length);`
- `unsigned int receive_data_packet(int fd, char * buffer, int * buf_len);`

Cada uma destas quatro funções necessita do serviço fornecido pela interface da camada imediatamente abaixo, isto é, a camada de Ligação de Dados.

Protocolo da Ligação de Dados: *datalink*

A API da ligação de dados segue a estrutura apresentada no guião, sendo constituída pelas seguintes funções:

- `int llopen(char *port, int user);`
- `int llwrite(int fd, char *buffer, int length);`
- `int llread(int fd, char *buffer);`
- `int llclose(int fd);`

Nesta API, como estão associadas várias chamadas, delimitadas pela abertura (**llopen**) e o respectivo fecho (**llclose**) do canal de ligação, é necessário guardar informação partilhada para as várias funções (estado do alarme, número de tentativas máximo para o envio de dados e número de sequência da trama atual):

- `extern int flag, attempts, DATA_C;`

Semelhante ao protocolo da Aplicação, a API chama funções que tratam do envio e receção das tramas de supervisão / não numeradas (*control*) e de informação (*data*):

- `void send_control_frame(int fd, int addr_byte, int ctrl_byte);`
- `unsigned char receive_control_frame(int fd, int addr_byte);`
- `int send_data_frame(int fd, char * buffer, int length);`
- `int receive_data_frame(int fd, unsigned char * data_c, char * data);`

De modo a que a porta série seja acessível a um nível razoavelmente alto, a componente *serialconfig* foi criada para estabelecer a interface com o controlador da porta série.

Interface da porta série: *serialconfig*

A API da porta série, à semelhança da ligação de dados, tem duas funções de abertura e fecho e duas funções de escrita e leitura:

- `int init_serial_n_canon(char* serialpath);`
- `int write_serial(int fd, unsigned char* buffer, int length);`
- `void read_serial(int fd, unsigned char* buffer, int length);`
- `void close_serial(int fd, int wait_time);`

Nesta componente, é guardado a estrutura original de configuração da porta série, de modo a, após o fecho, reverter à configuração original:

- `struct termios oldtio;`

Casos de Uso Principais

O trabalho realizado contém dois casos de utilização correspondentes ao emissor e ao recetor. As chamadas a serem efetuadas para estes dois casos são respetivamente **./transmitter <port> <filename>** e **./receiver <port>.** Os parâmetros passados nestas chamadas são <port> e <filename> correspondentes ao caminho da porta série a ser usado e o nome do ficheiro a transferir, respetivamente.

No caso do **transmitter** denota-se a seguinte sequência de funções:

main
  send_file
    llopen (transmitter)
      init_serial_n_cannon
      send_control_frame
        write_serial
      receive_control_frame
        read_serial
    send_control_packet
      llwrite
        send_data_frame
          write_serial
        receive_control_frame
          read_serial
    send_data_packet
      llwrite
        ...
    send_control_packet
        ...
    llclose (transmitter)
      send_control_frame
        write_serial
      receive_control_frame
        read_serial
      send_control_frame
        write_serial
      close_serial

No caso do **receiver** denota-se a seguinte sequência de funções:

main
  receive_file
    llopen (receiver)
      init_serial_n_cannon
      receive_control_frame
        ...
      send_control_frame
        ...
  receive_control_packet
    llread
      receive_data_frame
        read_serial
      send_control_frame
        write_serial
  receive_data_packet
    llread
      ...
  receive_control_packet
      ...
  llclose (receiver)
    receive_control_frame
      read_serial
    send_control_frame
      write_serial
    receive_control_frame
      read_serial
    close_serial

Protocolo de Ligação Lógica

O protocolo de ligação de dados tem como objetivo, tal como anteriormente mencionado, fornecer um serviço de comunicação fiável entre dois sistemas. Para tal, o protocolo assume as seguintes responsabilidades:

- Organização dos dados em tramas, sincronizadas através de flags no início e final de cada trama.
- Mecanismos de estabelecimento e terminação da ligação (**llopen** e **llclose**).
- Método de detecção de erros *checksum*, usando XOR LRC (Longitudinal Redundancy Check).
- Método de controlo de fluxo e de correção de erros ARQ (Automatic Repeat reQuest), usando o mecanismo Stop-and-Wait com numeração de tramas.

Sincronização das tramas (Flags e byte stuffing):

As tramas foram delimitadas por uma flag (0x7e) para garantir sincronização dos dados enviados. Para que nenhum octeto de informação seja confundido com uma flag, recorreu-se a um processo denominado de *byte stuffing*. Um octeto de escape (0x7d) é adicionado antes dum byte com valor reservado (0x7e - flag ou 0x7d - escape), que é convertido para 0x5e ou 0x5d respetivamente (XOR de si mesmo com o octeto 0x20). A operação inversa é efetuada na receção da trama.

Operação de stuffing (**write_data_frame**)

```
switch (buffer[i]) {
   case FLAG:
   case ESC:
     frame[index++] = ESC;
     frame[index++] =
buffer[i]^BST_BYTE;
     break;
   default:
     frame[index++] = buffer[i];
     break;
}
```

Operação de destuffing (**receive_data_frame**)

```
if(byte == ESC) {
   // Reads next byte
   read_serial(fd, &byte, 1);

   // Destuffs read byte
   byte ^= BST_BYTE;
}

(...)

buffer[index++] = byte;
```

Estabelecimento e terminação da ligação:

Usando as funções de escrita e leitura de tramas de controlo apresentadas anteriormente, torna-se simples a troca de mensagens para estabelecer e terminar uma ligação, conforme descrito no guião. De forma a garantir a fiabilidade desta operação, são também aplicados os métodos de controlo de erros apresentados a seguir.

Deteção de erros (Block Check Character):

Como método de detecção de erros, o XOR duma sequência de bytes transmitidos é guardado num octeto redundante acrescentado à trama, denominado de BCC (Block Check Character). Na receção da trama, o BCC é recalculado e comparado com o BCC transmitido. Nas tramas de supervisão / não numeradas, a trama é toda sujeita a uma única operação, enquanto que as tramas de informação contêm dois BCC's, um para o cabeçalho e outro para a informação a ser enviada. Este último é calculado antes da operação de stuffing, na escrita, e após a operação de destuffing, na leitura. O BCC é também sujeito a byte stuffing.

Cálculo do BCC2 (**write_data_frame**)

```
unsigned char bcc2 = 0;

(...)

for(int i = 0; i < length; i++) {
   bcc2 ^= buffer[i];

  (... Stuffing e colocação do byte ...)
}

(... Stuffing e colocação do BCC ...)
```

Recálculo do BCC2 (**receive_data_frame**)

```
unsigned char bcc2 = 0;

else {
   (... Destuffing do byte ...)

   bcc2 ^= byte;
   buffer[index++] = byte;
}
if (bcc2 == 0)
   (... Armazena os dados e retorna o
tamanho ...)
```

Mecanismo de Stop & Wait com numeração das tramas:

De modo a controlar o fluxo de tramas do emissor ao receptor, é o utilizado o mecanismo ARQ (Automatic Repeat reQuest) mais básico que existe, o mecanismo Stop & Wait. Neste mecanismo, o transmissor manda uma única trama de cada vez, prosseguindo para a próxima trama apenas quando recebe a confirmação do recetor. Do lado do transmissor, existe um mecanismo de timeout para, caso a trama de informação ou a trama de confirmação seja perdida, seja reenviada a trama de informação.

Inicialização dum alarme (**llopen**):

```
if (userType == TRANSMITTER) {
   struct sigaction sa;
   memset(&sa, 0, sizeof sa);
   sa.sa_handler = manage_alarm;

   sigaction(SIGALRM, &sa, NULL);
}
```

Controlo de timeouts usando o alarme:

```
while (attempts < MAX_ATTEMPTS) {
   //Reset alarm
   alarm(0); flag = 1;

   send_control_frame(fd, TRANS_A, SET_C);

   // Set alarm for 3 seconds
   if(flag){
   alarm(3);
   flag=0;
}
```

O mecanismo de detecção de erros, descrito anteriormente, é usado pelo receptor para verificar a validade da trama recebida. No caso de haver erros nos dados, é necessário rejeitar a trama, respondendo ao transmissor com uma trama de rejeição (De modo a antecipar o timeout, para não desperdiçar tempo).

Um problema deste método é a duplicação de tramas, ou devido à perda duma mensagem de confirmação ou devido a um tempo de timeout menor do que o tempo necessário para a receção de uma resposta (2 * Tprop + Tprocessamento). Para resolver este problema, as tramas são numeradas em módulo 2, ou seja, se o número da trama de informação recebida for igual ao número da última trama recebida, trata-se dum duplicado cujo conteúdo deve ser ignorado.

Validação e verificação do número da trama, computando a resposta apropriada:

```
if(length > -1) {
   command = data_c == DATA_C0 ? RR_C1
: RR_C0;

   if(DATA_C == data_c) {
      DATA_C = data_c == DATA_C0 ? DATA_C1
: DATA_C0;
   }
   else {
      printf("Duplicate frame\n");
   }
}
```

```
else {
   if (DATA_C == data_c) {
      command = data_c == DATA_C0 ?
REJ_C0 : REJ_C1;
   }
   else {
      command = DATA_C == DATA_C0 ?
RR_C0 : RR_C1;
      printf("Duplicate frame\n");
   }
}
```

Protocolo de Aplicação

O protocolo de aplicação assume a responsabilidade de transferir um ficheiro entre dois computadores. Apesar de usar o serviço fornecido pela ligação de dados, o protocolo necessita das seguintes funcionalidades:

- Controlo do fluxo de pacotes (Número de Sequência módulo 256);
- Sincronização dos pacotes de dados ;

Envio do comprimento de ficheiro:

Uma particularidade acerca do envio do comprimento do ficheiro (*length*) prende-se com a utilização de uma técnica que recorre a diversos shifts do valor do mesmo, permitindo guardar tamanhos superiores a 1 byte na estrutura de dados utilizada para representar o packet de controlo:

```
packet[index++] = sizeof(length);
packet[index++] = length & 0xFF;
packet[index++] = (length >> 8) & 0xFF;
packet[index++] = (length >> 16) & 0xFF;
packet[index++] = (length >> 24) & 0xFF;
```

Por sua vez, ao receber-se o control packet (*receive_control_packet*), todos esses shifts terão de ser invertidos de modo a obter o valor correspondente ao tamanho do ficheiro a ser recebido:

```
*file_length = (buffer[index++] | 0xFFFFFF00) & *file_length;
*file_length = ((buffer[index++] << 8) | 0xFFFF00FF) & *file_length;
*file_length = ((buffer[index++] << 16) | 0xFF00FFFF) & *file_length;
*file_length = ((buffer[index++] << 24) | 0x00FFFFFF) & *file_length;
```

Controlo de fluxo:

Implementou-se um mecanismo de controlo de fluxo de modo a suportar os vários envios de pacotes de dados, utilizando para tal uma técnica de numeração de tramas através do incremento da variável chamada *packet_i*, sendo que o seu valor no momento atual é recebido pela função *receive_data_packet* que posteriormente verifica a validade o pacote de dados tal como ilustrado no seguinte excerto:

```
  while (received_bytes < file_length) {
     int packet_bytes = 0;

     unsigned int packet_status = receive_data_packet(fd, buffer + received_bytes,
 &packet_bytes);

     if (packet_status < 0 || packet_status != packet_i++ % 256) {
       // Error processing occurs here (if any)
       return -1;
     }

     received_bytes += packet_bytes;
          printf("Data packet received: %d bytes of %d.\n", received_bytes, file_length);
   }
```

Sincronização da transferência de controlo:

Com o objetivo de sincronizar a transferência de dados, é utilizado o envio de pacotes de controlo que delimitam o início e fim da respetiva transferência. Assim, antes de serem enviados quaisquer pacotes de dados é enviado um pacote de controlo para demarcar o início da transferência - `send_control_packet(fd, START_C, filename, length)` - e, após o envio de todos os pacotes de dados, é enviado o pacote de controlo para demarcar o fim da mesma - `send_control_packet(fd, END_C, filename, length)` - permitindo sincronizar efetivamente todo o processo.

Uma vez que é conhecido o tamanho do ficheiro a ser transferido, a leitura dos pacotes de dados termina quando a quantidade equivalente ao tamanho do ficheiro for recebida. O pacote de controlo no final, serve como elemento redundante que, para além de repetir os dados fornecidos inicialmente, assegura o recetor de que realmente foram lidos todos os pacotes transmitidos.

Validação

Com o objectivo de validar o programa desenvolvido e consequentemente demonstrar a sua robustez, foram realizados os seguintes testes:

- Transferência de um ficheiro simples, sem e com quebra de ligação;
- Transferência de um ficheiro simples, introduzindo dados inválidos ao longo da transmissão;
- Transferência de um ficheiro simples, simultaneamente com quebra de ligação e com introdução de dados inválidos ao longo da transmissão;
- Transferência de ficheiros de maior dimensão (até 2.2 MB), tendo sido verificado sucesso mesmo executando com os testes acima descritos.

O programa ainda está provido de uma interface do utilizador que possibilita a visualização da configuração do sistema, bem como todo o processo de transferência do ficheiro. É possível visualizar esta interface através do anexo - **Interface do utilizador**.

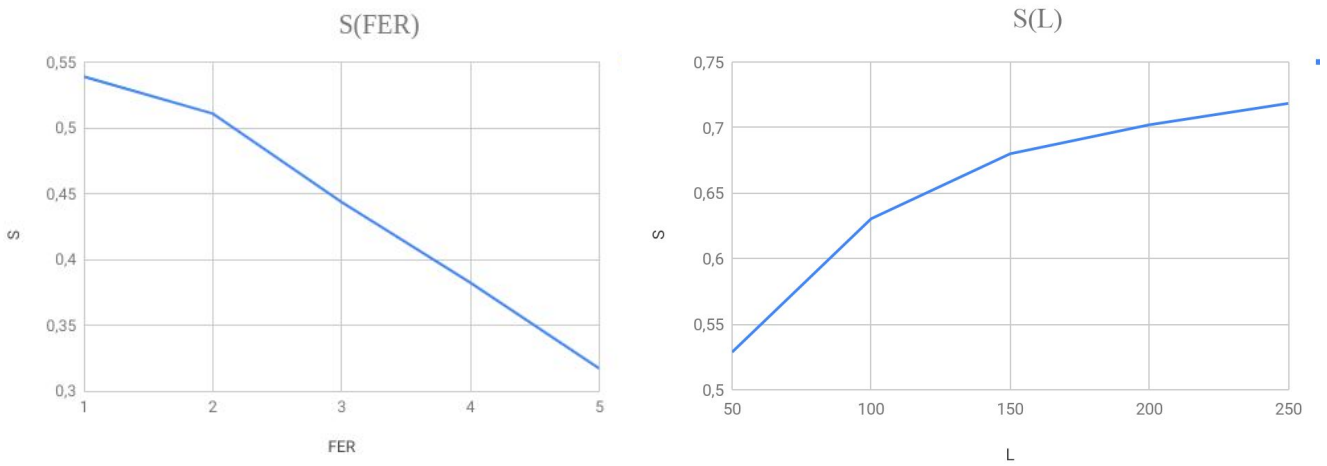
Eficiência do protocolo de ligação de dados

A eficiência, $S = \frac{1-FER}{1+2a}$, do protocolo de ligação de dados foi estimada com vista a efetuar a comparação entre o resultado prático e o resultado teórico. O cálculo da eficiência foi feito em relação a duas variáveis distintas:

- ***FER*** (Frame Error Ratio) - Usando números pseudo-aleatórios, simulamos a existência de um **FER**, rejeitando tramas a partir desta probabilidade, independentemente de terem erros ou não. É expectável, pela fórmula teórica da eficiência, que, com o aumento do **FER**, diminua o **S.**

```
length = (rand() % 10) < FER ? -1 : length;
```

- ***a*** (Tempo de propagação) - este valor depende de três variáveis, o tempo de propagação (Tprop), a capacidade do cabo (*baudrate*) e o tamanho da trama (L). Independentemente da alteração efetuada esta afetará o tempo de propagação da seguinte forma: $a = \frac{Tprop}{Tf} = \frac{Tprop \times R}{L}$. Para as nossas medições, variou-se o comprimento da trama enviada, definida numa *macro* no nosso protocolo, logo é expectável que com o aumento do **L**, **a** diminua e, consequentemente, o **S** aumente.



Conforme previsto pelo modelo teórico: $S = \frac{1 - FER}{1 + 2a}$ e $a = \frac{Tprop}{Tf} = \frac{Tprop \times R}{L}$, a eficiência diminui com o aumento do **FER** e aumenta com o aumento do **L**, que por sua vez implica a diminuição de **a**.

Conclusões

Após a realização deste trabalho, concluímos que não só o principal objectivo do trabalho foi concluído (a transferência de ficheiros entre dois computadores pela porta de série), mas também todos os restantes objectivos inerentes ao protocolo utilizado para que tal fosse possível de forma fiável e usando uma pilha de dois protocolos independentes entre si.

Na camada de ligação de dados, responsável pelo estabelecimento duma ligação unidirecional fiável entre dois sistemas, usando os mecanismos de controlo de fluxo e de erros referidos, a informação é transmitida sem processamento do seu conteúdo.

Na camada da aplicação, responsável pela transferência de um ficheiro entre dois computadores, o protocolo fia-se plenamente no serviço fornecido pelo protocolo de ligação de dados para o envio dos pacotes.

Os testes efetuados para a verificação do sistema, transferindo ficheiros de dimensão grande e colocando a teste os mecanismos de ligação de dados, asseguram a robustez planeada para o protocolo.

Considerando estas características do sistema desenvolvido, temos confiança nos conhecimentos adquiridos na sua implementação, em respeito ao aspectos funcionais dos protocolos, e temos confiança na aptidão do sistema para o segundo trabalho.

Anexos

makefile

```
CC = gcc
CFLAGS = -std=c99 -Wall -D_POSIX_C_SOURCE
DEPS = datalink.h serialconfig.h application.h
TRANSMITTER_OBJ = transmitter.o serialconfig.o datalink.o application.o
RECEIVER_OBJ = receiver.o serialconfig.o datalink.o application.o

all: transmitter receiver

%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)


transmitter: $(TRANSMITTER_OBJ)
    $(CC) -o $@ $^ $(CFLAGS)

receiver:  $(RECEIVER_OBJ)
    $(CC) -o $@ $^ $(CFLAGS)

clean:
    rm -f transmitter receiver
    rm -f $(OBJ) *.o
```

transmitter.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "serialconfig.h"
#include "datalink.h"
#include "application.h"

/*
      Main function of transmitter.

      Brief: It opens, reads and stores the file content, which name is
passed by parameter, and sends it to the serial port also passed by
```

```
parameter.
*/
int main(int argc, char** argv)
{
  if ( (argc < 3) ||
  ((strcmp("/dev/ttyS0", argv[1])!=0) &&
  (strcmp("/dev/ttyS1", argv[1])!=0) &&
  (strcmp("/dev/ttyS2", argv[1])!=0) )) {
      printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS1\n");
      exit(1);
  }

  // Open file to be sent
  struct stat buffer;
  int length = 0;

  FILE* file = fopen(argv[2], "r");
    if (!file) {
       printf("ERROR: File: %s could not be opened.\n", argv[2]);
       return -1;
    }

  // Get file size
  stat(argv[2], &buffer);
  int size = buffer.st_size;

  char file_content[size];

  while (!feof(file)) {
      length += fread(file_content, 1, sizeof(file_content), file);
  }

  // Display's initial information
  printf("\n##################\n");
  printf("## TRANSMITTER ##\n");
  printf("##################\n\n");

  printf("--- Settings ---\n\n");
  printf("Serial port: %s\n", argv[1]);
  printf("Max attempts: %d\n", 3);
  printf("Timeout: %d seconds\n", 3);
  printf("File to be transmitted: %s\n", argv[2]);
  printf("File size: %d bytes\n\n", length);


  send_file(argv[1], argv[2], file_content, length);
```

```
  return 0;
}
```

receiver.c

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

#include "serialconfig.h"
#include "datalink.h"
#include "application.h"

/*
      Main function of receiver

      Brief: Validates function call and call the respective function to
receive a certain file via serial port.
*/
int main(int argc, char** argv)
{
  if ( (argc < 2) ||
  ((strcmp("/dev/ttyS0", argv[1])!=0) &&
  (strcmp("/dev/ttyS1", argv[1])!=0) &&
  (strcmp("/dev/ttyS2", argv[1])!=0) )) {
      printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS1\n");
      exit(1);
  }

  // Display's initial information
  printf("\n################\n");
  printf("## RECEIVER ##\n");
  printf("################\n\n");
  printf("--- Settings ---\n\n");
  printf("Serial port: %s\n", argv[1]);
  printf("Max attempts: %d\n", 3);
  printf("Timeout: %d seconds\n\n", 3);

  // Initialization to avoid same random numbers
  srand(time(NULL));

    time_t begin = time(NULL);

    receive_file(argv[1]);
```

```
    printf("Execution time: %f\n", difftime(time(NULL), begin));

  return 0;
}
```

application.h

```
// Handles all of the application layer
#ifndef APPLICATION_H
#define APPLICATION_H

#define CONTROL_PACKET_LEN 5
#define DATA_PACKET_LEN 4

#define MAX_DATA_LEN 100 // Max data length to be sent in on packet: 2^16.

#define DATA_C 1
#define START_C 2
#define END_C 3

#define FILESIZE_T 0
#define FILENAME_T 1

#define MAX_ATTEMPTS 3

/*
      Sends control packet to define the beggining or ending of file
transmition

      Argument 'fd': serial port file descriptor
      Argument 'type': defines either if its the beggining or ending of
file transmition (START_C | END_C)
      Argument 'filename': name of the file to be transmitted
      Argument 'length': size of the file to be transmitted
*/
int send_control_packet(int fd, int type, char * filename, unsigned int
length);

/*
      Receives control packet containing information about the file to be
received

      Argument 'fd': serial port file descriptor
      Argument 'type': determines either if its the beggining or ending of
file transmition (START_C | END_C)
```

```
      Argument 'filename': name of the file to be received
      Argument 'file_length': size of the file to be received
*/
int receive_control_packet(int fd, int type, char * filename, unsigned int
* file_length);

/*
      Sends data packet containing file content (partial or integral)

      Argument 'fd': serial port file descriptor
      Argument 'N': data packet sequence number
      Argument 'buffer': content to be transmitted
      Argument 'length': size of the content to be transmitted
*/
int send_data_packet(int fd, unsigned int N, char * buffer, unsigned int
length);

/*
      Receives data packet containing file content (partial or integral)

      Argument 'fd': serial port file descriptor
      Argument 'buffer': buffer where content transmitted will be stored
      Argument 'length': size of the content received
*/
unsigned int receive_data_packet(int fd, char * buffer, int * buf_len);

/*
      Function responsible to send a certain file

      Argument 'port': serial port directory
      Argument 'filename': name of the file to be transmitted
      Argument 'file_content': buffer that contains content to be
transmitted
      Argument 'length': file size
*/
int send_file(char * port, char * filename, char * file_content, int
length);

/*
      Function responsible to receive a certain file

      Argument 'port': serial port directory
*/
int receive_file(char * port);

#endif
```

application.c

```
#include <stdio.h>
#include <string.h>
#include "datalink.h"
#include "application.h"

////////////////////////////////////////////////////////////////////////
////////////////////////// Control Packet Functions ////////////////////
////////////////////////////////////////////////////////////////////////

int send_control_packet(int fd, int type, char * filename, unsigned int
length) {
  unsigned int index = 0;
  char packet[CONTROL_PACKET_LEN + sizeof(filename) + sizeof(length)];

  if(strlen(filename) > 254) {
      printf("Filename: %s too large to fit in packet.\n", filename);
      return -1;
  }

  packet[index++] = type;

  packet[index++] = FILESIZE_T;
  packet[index++] = sizeof(length);
  packet[index++] = length & 0xFF;
  packet[index++] = (length >> 8) & 0xFF;
  packet[index++] = (length >> 16) & 0xFF;
  packet[index++] = (length >> 24) & 0xFF;

  packet[index++] = FILENAME_T;
  packet[index++] = strlen(filename) + 1;
  for(int i = 0; i <= strlen(filename); i++) {
      packet[index++] = filename[i];
  }

  return llwrite(fd, packet, index);
}

int receive_control_packet(int fd, int type, char * filename, unsigned int
* file_length) {
      int tmp_length = 0;
      unsigned int index = 0;
      *file_length = 0xFFFFFFFF;
      char buffer[CONTROL_PACKET_LEN + 255 + sizeof(int)];
```

```
      int packet_length = llread(fd, buffer);

      if(type != buffer[index++]) {
      return -1;
      }

      while(index < packet_length) {

      switch (buffer[index++]) {
            case FILESIZE_T:
            tmp_length = buffer[index++];
            *file_length = (buffer[index++] | 0xFFFFFF00) & *file_length;
            *file_length = ((buffer[index++] << 8) | 0xFFFF00FF) &
*file_length;
            *file_length = ((buffer[index++] << 16) | 0xFF00FFFF) &
*file_length;
            *file_length = ((buffer[index++] << 24) | 0x00FFFFFF) &
*file_length;
            break;
            case FILENAME_T:
            tmp_length = buffer[index++];
            for(int i = 0; i < tmp_length; i++){
            filename[i] = buffer[index++];
            }
            break;
            default:
            return -1;
            break;
      }
      }

      return 0;
}

////////////////////////////////////////////////////////////////////////
////////////////////////// Data Packet Functions ///////////////////////
////////////////////////////////////////////////////////////////////////

int send_data_packet(int fd, unsigned int N, char * buffer, unsigned int
length) {
  if (length > MAX_DATA_LEN) {
      printf("Length overflow\n");
      return -1;
  }
  else if (length == 0) {
      return 0; // For safety reasons, best to not do anything in this
case
```

```
  }

  unsigned int index = 0;
  char packet[DATA_PACKET_LEN + length];

  packet[index++] = DATA_C;
  packet[index++] = (unsigned char) N;
  packet[index++] = (char)(length / 256);
  packet[index++] = (char)(length % 256);

  for(int i = 0; i < length; i++) {
      packet[index++] = buffer[i];
  }

  return llwrite(fd, packet, index);
}

unsigned int receive_data_packet(int fd, char * buffer, int * buf_len) {
   unsigned int index = 0;
   char packet[DATA_PACKET_LEN + MAX_DATA_LEN];

   llread(fd, packet);

   if(packet[index++] != DATA_C) {
      printf("Wrong packet received. Should receive a data packet.\n");
      return -1;
   }

   unsigned int N = (unsigned char) packet[index++];

   *buf_len = 256 * (unsigned char)(packet[index]) + (unsigned
char)(packet[index+1]);
   index += 2;

   for(int i = 0; i < *buf_len; i++) {
      buffer[i] = packet[index++];
   }

   return N;
}

////////////////////////////////////////////////////////////////////////
////////////////////// API application functions //////////////////////
////////////////////////////////////////////////////////////////////////


int send_file(char * port, char * filename, char * file_content, int
```

```
length){

  // Establish connection
  int fd = llopen(port, TRANSMITTER);

  // In case connection establishement fails
  if(fd < 0)
      return -1;

  // Determines beggining of file transfer
  printf("--- Beggining file transfer --- \n\n");
  if (send_control_packet(fd, START_C, filename, length) < 0) {
      return -1;  // Error processing occurs here (if any)
  }

  printf("Data transmitted: 0 bytes of %d.\n", length);
  // Transfer file data (Considering the Max Data Bytes per Packet defined
in the header)
  unsigned int sent_bytes = 0, packet_i = 0;

  while (sent_bytes + MAX_DATA_LEN <= length) {
      int packet_status = send_data_packet(fd, packet_i++, file_content +
sent_bytes, MAX_DATA_LEN);

      if (packet_status < 0) { // Error Occured (Any processing of the
error occurs within this if)
      return packet_status;
      }

      sent_bytes += MAX_DATA_LEN;
      printf("Data transmitted: %d bytes of %d.\n", sent_bytes, length);
  }

  int packet_status = send_data_packet(fd, packet_i++, file_content +
sent_bytes, length - sent_bytes);
  printf("Data transmitted: %d bytes of %d.\n", length, length);
  if (packet_status < 0) {
      return packet_status;
  }

  // Determines ending of file transfer
  if (send_control_packet(fd, END_C, filename, length) < 0) {
      return -1;  // Error processing occurs here (if any)
  }

  // Terminates connection
  printf("File transfer complete.\n\n--- Ending file transfer --- \n\n");
```

```
  llclose(fd);

    return packet_i; // Return number of packets sent
}

int receive_file(char * port) {
  // Local variables
  char filename[255];
  unsigned int file_length = 0;

  // Establish connection
  int fd = llopen(port, RECEIVER);

  // Determines beggining of file transfer
  printf("--- Beggining file transfer --- \n\n");
  if (receive_control_packet(fd, START_C, filename, &file_length) < 0) {
      return -1;  // Error processing occurs here (if any)
  }

  // Receives file
  char buffer[file_length];
  unsigned int received_bytes = 0, packet_i = 0;

  while (received_bytes < file_length) {
      int packet_bytes = 0;

      unsigned int packet_status = receive_data_packet(fd, buffer +
received_bytes, &packet_bytes);

      if (packet_status < 0 || packet_status != packet_i++ % 256) {
      // Error processing occurs here (if any)
      return -1;
      }

      received_bytes += packet_bytes;
      printf("Data packet received: %d bytes of %d.\n", received_bytes,
file_length);
  }

  // Determines ending of file transfer
  if (receive_control_packet(fd, END_C, filename, &file_length) < 0) {
      return -1;  // Error processing occurs here (if any)
  }

  // Terminates connection
  printf("\n--- Ending file transfer --- \n\n");
  llclose(fd);
```

```
  // Open file to be sent
  FILE* file = fopen(filename, "w");
  if (!file) {
      printf("ERROR: File: %s could not be opened.\n", filename);
      return -1;
  }

  // Write content to file
  for(int i = 0; i < file_length; i++) {
      fwrite(buffer+i, 1, 1, file);
  }

  return 0;
}
```

datalink.h

```
// Handles all of the data link layer
#ifndef DATA_LINK_H
#define DATA_LINK_H

#define TRANSMITTER 0
#define RECEIVER  1

#define FLAG      0x7E
#define ESC       0x7D
#define BST_BYTE  0x20

#define TRANS_A   0x03 // Transmitter commands and Receiver responses
#define REC_A     0x01 // Receiver commands and Transmitter responses

// List of Commands
#define SET_C     0x03
#define DISC_C    0x0B
#define DATA_C1   0x40
#define DATA_C0   0x00


// List of Responses
#define UA_C      0x07
#define RR_C0     0x05
#define RR_C1     0x85
#define REJ_C0    0x01
#define REJ_C1    0x81
```

```
// FRAME SEND AND RECEIVER FUNCTIONS
#define CTRL_FRAME_LEN   5    // Length of the control frame (in bytes)
#define DATA_FRAME_LEN   7    // Length of the data frame header and
trailer (in bytes)
#define MAX_DATA_LEN    100   // Max data length to be sent in on packet:
2^16
#define FER             0     // Frame error probability

// Global variables
extern int flag, attempts, DATA_C;


// DataLink API

/*
      Establish connection between serial ports

      Argument 'port': serial port directory
      Argument 'user': determines the user type: transmitter or receiver
*/
int llopen(char *port, int user);

/*
      Writes content to the serial port

      Argument 'fd': serial port file descriptor
      Argument 'buffer': buffer containing data to be sent
      Argument 'length': number of bytes to be sent
*/
int llwrite(int fd, char * buffer, int length);

/*
      Reads content from the serial port.

      Argument 'fd': serial port file descriptor
      Argument 'buffer': buffer where data will be stored
*/
int llread(int fd, char* buffer);

/*
      Terminates connection between serial ports

      Argument 'fd': serial port file descriptor
*/
int llclose(int fd);
```

```
// DataLink Aux Functions

/*
      Send Supervision or Unnumbered frames

      Argument 'fd': serial port file descriptor
      Argument 'addr_byte': address byte to be sent (TRANS_A | REC_A)
      Argument 'ctrl_byte': control byte to be sent (SET_C | UA_C)
*/
void send_control_frame(int fd, int addr_byte, int ctrl_byte);

/*
      Receive Supervision or Unnumbered frames

      Argument 'fd': serial port file descriptor
      Argument 'addr_byte':  address byte to be received (TRANS_A | REC_A)
*/
unsigned char receive_control_frame(int fd, int addr_byte);

/*
      Sends frame containing the data to be sent

      Argument 'fd': serial port file descriptor
      Argument 'buffer': buffer containing the data to be sent
      Argument 'length': number of bytes to be sent
*/
int send_data_frame(int fd, char * buffer, int length);

/*
      Receives frame containing the data that was sent

      Argument 'fd': serial port file descriptor
      Argument 'data_c': received control field (0 | 1)
      Argument 'data': where data will be stored
*/
int receive_data_frame(int fd, unsigned char * data_c, char * data);

#endif
```

datalink.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
```

```c
#include "datalink.h"
#include "serialconfig.h"
#include <stdio.h>

// Init global variables
int userType;
int flag = 1;
int attempts = 1;
int DATA_C = DATA_C0;

/*
      Manages alarm interruptions
*/
void manage_alarm() {
  flag=1;
  attempts++;
}

////////////////////////////////////////////////////////////////////////
////////////////////////// Connection Establishment ////////////////////
////////////////////////////////////////////////////////////////////////

int llopen(char *port, int user) {
  printf("--- Connection Establishment ---\n\n");

  userType = user;
  int fd = init_serial_n_canon(port);

  if(userType == TRANSMITTER) {

      struct sigaction sa;
      memset(&sa, 0, sizeof sa);
      sa.sa_handler = manage_alarm;

      sigaction(SIGALRM, &sa, NULL);

  }

  switch(userType) {
      case TRANSMITTER:

      while (attempts < 4) {
      //Reset alarm
      alarm(0); flag = 1;

      // Send SET command
```

```
        send_control_frame(fd, TRANS_A, SET_C);
        printf("SET Command sent --> ");

        // Set alarm for 3 seconds
        if(flag){
              alarm(3);
              flag=0;
        }

        // Setup receiving UA message
        if (receive_control_frame(fd, TRANS_A) ==  UA_C) {
              printf("UA Command received\n");
              break;
        }
        else {
              printf("UA Command not received: Attempting to reconnect.\n");
        }
        }

        if (attempts >= 4) {
        printf("UA Command not received\n");
        return -1;
        }
        break;
        case RECEIVER:
        // Setup receiving SET message
        while(receive_control_frame(fd, TRANS_A) != SET_C);
        printf("SET Command received --> ");

        // Send UA response
        send_control_frame(fd, TRANS_A, UA_C);
        printf("UA Command sent\n");
        break;
        default:
        return -1;
        break;
    }

    printf("Connection established\n\n");

    return fd;
}

void send_control_frame(int fd, int addr_byte, int ctrl_byte) {
    unsigned char frame[CTRL_FRAME_LEN];

    frame[0] = FLAG;
```

```
  frame[1] = addr_byte;
  frame[2] = ctrl_byte;
  frame[3] = frame[1]^frame[2];
  frame[4] = FLAG;

  write_serial(fd, frame, CTRL_FRAME_LEN);
}

unsigned char receive_control_frame(int fd, int addr_byte) {
  unsigned char byte, ctrl_byte;

  enum set_states {START, FLAG_REC, A_REC, C_REC, BCC_OK, END};
  enum set_states state = START;

  while (state != END) {
      if (flag && (userType == TRANSMITTER)) {
      return 0;
      }

      read_serial(fd, &byte, 1);

      switch(state) {
      case START:
      if(byte == FLAG) {
      state = FLAG_REC;
      }
      break;
      case FLAG_REC:
      if(byte == addr_byte) {
      state = A_REC;
      }
      else if(byte != FLAG) {
      state = START;
      }
      break;
      case A_REC:
      if((byte == SET_C) || (byte == DISC_C) || (byte == UA_C) || (byte ==
RR_C0) || (byte == RR_C1) || (byte == REJ_C0) || (byte == REJ_C1)) {
      ctrl_byte = byte;
      state = C_REC;
      }
      else if (byte == FLAG) {
      state = FLAG_REC;
      }
      else {
      state = START;
      }
```

```
        break;
        case C_REC:
        if(byte == (addr_byte^ctrl_byte)) {
        state = BCC_OK;
        }
        else if (byte == FLAG) {
        state = FLAG_REC;
        }
        else {
        state = START;
        }
        break;
        case BCC_OK:
        if(byte == FLAG){
        state = END;
        }
        else {
        state = START;
        }
        break;
        case END:
        break;
        }
    }

    return ctrl_byte;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////// Data Transmission - Transmitter ////////////////
////////////////////////////////////////////////////////////////////////////

int llwrite(int fd, char * buffer, int length) {
    // Local variables
    int num_written_bytes = -1;

    // Reset global variables
    attempts = 1;
    flag = 1;

    while(attempts < 4) {
        // Send data frame
        num_written_bytes = send_data_frame(fd, buffer, length);

        // Set alarm for 3 seconds
        if(flag){
        alarm(3);
```

```
        flag = 0;
        }

        //Check for receiver response
        unsigned char command = receive_control_frame(fd, TRANS_A);

        // Reset variables
        alarm(0);
        flag = 1;

        switch(command) {
        case RR_C0:
        case REJ_C0:
        if(DATA_C == DATA_C1) {
              DATA_C = DATA_C0;
              printf("Receiver ready. ");
              return num_written_bytes;
        }
        else {
              printf("Reject. Attempting to retransmit data.\n");
        }
        break;
        case RR_C1:
        case REJ_C1:
        if(DATA_C == DATA_C0) {
              DATA_C = DATA_C1;
              printf("Receiver ready. ");
              return 0;
        }
        else {
              printf("Reject. Attempting to retransmit data.\n");
        }
        break;
        default:
        printf("Command not received. Attempting to retransmit data.\n");
        break;
        }
    }

    return -1;
}

int send_data_frame(int fd, char * buffer, int length) {
    unsigned int index = 4;
    unsigned char frame[DATA_FRAME_LEN + length*2], bcc2 = 0;

    frame[0] = FLAG;
```

```
  frame[1] = TRANS_A;
  frame[2] = DATA_C;
  frame[3] = frame[1]^frame[2];

  // Byte Stuffing - Data
  for(int i = 0; i < length; i++) {
      bcc2 ^= buffer[i];

      switch (buffer[i]) {
      case FLAG:
      case ESC:
      frame[index++] = ESC;
      frame[index++] = buffer[i]^BST_BYTE;
      break;
      default:
      frame[index++] = buffer[i];
      break;
      }
  }

  // Byte Stuffing - BCC2
  switch (bcc2) {
      case FLAG:
      case ESC:
      frame[index++] = ESC;
      frame[index++] = bcc2^BST_BYTE;
      break;
      default:
      frame[index++] = bcc2;
      break;
  }

  frame[index] = FLAG;

  return write_serial(fd, frame, index);
}

////////////////////////////////////////////////////////////////////////
////////////////////////// Data Transmission - Receiver ////////////////
////////////////////////////////////////////////////////////////////////

int llread(int fd, char* buffer) {
  int length;
  unsigned char data_c, tmp_data_c = DATA_C, command;

  while(tmp_data_c == DATA_C) {
      length = receive_data_frame(fd, &data_c, buffer);
```

```
        // Applying artificial FER
        length = (rand() % 10) < FER ? -1 : length;

        if(length > -1) {
        command = data_c == DATA_C0 ? RR_C1 : RR_C0;

        if(DATA_C == data_c) {
              DATA_C = data_c == DATA_C0 ? DATA_C1 : DATA_C0;
        }
        else {
        printf("Duplicate frame\n");
        }
        }
        else {
        if(DATA_C == data_c) {
        command = data_c == DATA_C0 ? REJ_C0 : REJ_C1;
        }
        else {
        command = DATA_C == DATA_C0 ? RR_C0 : RR_C1;
        printf("Duplicate frame\n");
        }

        }

        send_control_frame(fd, TRANS_A, command);
    }

    return length;
}

int receive_data_frame(int fd, unsigned char * data_c, char * data) {
    unsigned int index = 0;
    unsigned char byte, bcc2 = 0;
    char buffer[MAX_DATA_LEN + 5];

    enum set_states {START, FLAG_REC, A_REC, C_REC, BCC_OK, END};
    enum set_states state = START;

    while (state != END) {
        read_serial(fd, &byte, 1);

        switch(state) {
        case START:
        if(byte == FLAG) {
              state = FLAG_REC;
        }
```

```
break;
case FLAG_REC:
if(byte == TRANS_A) {
      state = A_REC;
}
else if (byte != FLAG) {
      state = START;
}
break;
case A_REC:
if(byte == FLAG){
      state = FLAG_REC;
}
else if(byte == DATA_C0 || byte == DATA_C1) {
      *data_c = byte;
      state = C_REC;
}
else
      state = START;
break;
case C_REC:
if(byte == (TRANS_A^(*data_c))) {
      state = BCC_OK;
}
else if(byte == FLAG) {
      state = FLAG_REC;
}
else {
      state = START;
}
break;
case BCC_OK:
if(byte == FLAG) {
      state = END;
}
else {
      // Destuffing byte operation
      if(byte == ESC) {
      // Reads next byte
      read_serial(fd, &byte, 1);

      // Destuffs read byte
      byte ^= BST_BYTE;
      }

      bcc2 ^= byte;
      buffer[index++] = byte;
```

```
        }
        break;
        case END:
        break;
        }
    }

    if(bcc2 == 0) {
        if(DATA_C == *data_c) { // In case of valid data (valid data frame
and not duplicated) stores content transmitted
        unsigned int iterator = 0;

        while(iterator < (index - 1)) {
        data[iterator] = buffer[iterator];
        iterator++;
        }
        }

        return (index - 1);
    }

    return -1;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////// Termination of connection //////////////////////
////////////////////////////////////////////////////////////////////////////

int llclose(int fd) {
    printf("--- Connection Termination ---\n\n");

    // Reset global variables
    attempts = 1;
    flag = 1;

    switch (userType) {
        case TRANSMITTER:
        while (attempts < 4) {
        // Send SET command
        send_control_frame(fd, TRANS_A, DISC_C);
        printf("DISC Command sent --> ");

        // Set alarm for 3 seconds
        if (flag){
              alarm(3);
              flag=0;
        }
```

```
      // Setup receiving DISC message
      if (receive_control_frame(fd, TRANS_A) ==  DISC_C) {
            printf("DISC Command received\n");
            send_control_frame(fd, TRANS_A, UA_C);
            printf("UA Command sent\n\n");
            break;
      }
      else {
            printf("DISC Command not received: Attempting to retransmit
command.\n");
      }
      }

      if (attempts >= 4) {
      printf("DISC Command not received\n");
      return -1;
      }
      break;
      case RECEIVER:
      // Setup receiving DISC message
      while (receive_control_frame(fd, TRANS_A) != DISC_C);
      printf("DISC Command received\n");

      // Send DISC response
      send_control_frame(fd, TRANS_A, DISC_C);
      printf("DISC Command sent --> ");

      // Setup receiving UA message
      while (receive_control_frame(fd, TRANS_A) != UA_C);
      printf("UA Command received\n\n");
      break;
      default:
      return -1;
      break;
  }

  close_serial(fd, 2);

  printf("--- Connection Terminated ---\n\n");

  return 0;
}
```

serialconfig.h

```
// ALL SERIAL PORT RELATED STUFF
#ifndef SERIAL_CONFIG_H
#define SERIAL_CONFIG_H

#include <termios.h>
#include <strings.h>

#define BAUDRATE B38400
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1

struct termios oldtio; // Original configuration

/*
  Sets up the serial port in non-canonical mode (Based off of source code
provided)

  Return: file descriptor  of serial port
*/
int init_serial_n_canon(char* serialpath);

/*
  Writes to serial port

  Argument 'fd': file descriptor of serial port
  Argument 'buffer': buffer to be transmitted
  Argument 'length': length of the buffer (in bytes)
*/
int write_serial(int fd, unsigned char* buffer, int length);

/*
  Reads from serial port

  Argument 'fd': file descriptor of serial port
  Argument 'buffer': buffer to store received bytes
  Argument 'length': number of bytes to be received (and stored in buffer)
*/
void read_serial(int fd, unsigned char* buffer, int length);

/*
  Closes the serial port, setting it back to original configuration

  Argument 'fd': file descriptor of serial port to close
```

```
  Argument 'wait_time': time in seconds to wait before resetting the
config (Give time for all the inserted bits to be transmitted)
*/
void close_serial(int fd, int wait_time);

#endif
```

serialconfig.c

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

#include "serialconfig.h"

int init_serial_n_canon(char* serialpath) {
  int fd;
  struct termios newtio;

  /*Open serial port device for reading and writing and not as controlling
tty because we don't want to get killed if linenoise sends CTRL-C. */

  fd = open(serialpath, O_RDWR | O_NOCTTY );
  if (fd <0) { perror(serialpath); exit(-1); }

  if ( tcgetattr(fd,&oldtio) == -1) { /* save current port settings */
      perror("tcgetattr(new)");
      exit(-1);
  }

  bzero(&newtio, sizeof(newtio));
  newtio.c_cflag = BAUDRATE | 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]    = 5;   /* inter-character timer unused */
  newtio.c_cc[VMIN]     = 0;   /* blocking read until 5 chars received */
```

```
  /* VTIME e VMIN devem ser alterados de forma a proteger com um
temporizador a leitura do(s) proximo(s) caracter(es)*/

  tcflush(fd, TCIOFLUSH);

  if ( tcsetattr(fd,TCSANOW,&newtio) == -1) {
      perror("tcsetattr");
      exit(-1);
  }

  printf("New termios structure set\n");

  return fd;
}

int write_serial(int fd, unsigned char* buffer, int length) {
  int writtenBytes = 0;

  while(writtenBytes < length)
      writtenBytes += write(fd, buffer + writtenBytes, length + 1 -
writtenBytes);

  return writtenBytes;
}

void read_serial(int fd, unsigned char* buffer, int length) {
  read(fd, buffer, length);
}

void close_serial(int fd, int wait_time) {
  sleep(wait_time);

  if (tcsetattr(fd,TCSANOW,&oldtio) == -1) {
      perror("tcsetattr(old)");
      exit(-1);
  };
  close(fd);
}
```

Interface de utilizador - Transmitter e Receiver

```
gnu53:~/RCOM1819_T3/SourceCode# ./transmitter /dev/ttyS0 pinguim.gif

################
## TRANSMITTER ##
################

--- Settings ---

Serial port: /dev/ttyS0
Max attempts: 3
Timeout: 3 seconds
File to be transmitted: pinguim.gif
File size: 10968 bytes

--- Connection Establishment ---

New termios structure set
SET Command sent --> UA Command received
Connection established

--- Beggining file transfer ---

Receiver ready. Data transmitted: 0 bytes of 10968.
Receiver ready. Data transmitted: 2000 bytes of 10968.
Receiver ready. Data transmitted: 4000 bytes of 10968.
Receiver ready. Data transmitted: 6000 bytes of 10968.
Receiver ready. Data transmitted: 8000 bytes of 10968.
Receiver ready. Data transmitted: 10000 bytes of 10968.
Receiver ready. Data transmitted: 10968 bytes of 10968.
Receiver ready. File transfer complete.

--- Ending file transfer ---

--- Connection Termination ---

DISC Command sent --> DISC Command received
UA Command sent

--- Connection Terminated ---

gnu53:~/RCOM1819_T3/SourceCode#
```

```
gnu53:~/RCOM1819_T3/SourceCode# ./receiver /dev/ttyS0

##############
## RECEIVER ##
##############

--- Settings ---

Serial port: /dev/ttyS0
Max attempts: 3
Timeout: 3 seconds

--- Connection Establishment ---

New termios structure set
SET Command received --> UA Command sent
Connection established

--- Beggining file transfer ---

Data packet received: 2000 bytes of 10968.
Data packet received: 4000 bytes of 10968.
Data packet received: 6000 bytes of 10968.
Data packet received: 8000 bytes of 10968.
Data packet received: 10000 bytes of 10968.
Data packet received: 10968 bytes of 10968.

--- Ending file transfer ---

--- Connection Termination ---

DISC Command received
DISC Command sent --> UA Command received

--- Connection Terminated ---

Execution time: 9.000000
gnu53:~/RCOM1819_T3/SourceCode#
```