FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Trabalho Laboratorial nº1:
Ligação de Dados

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA E COMPUTAÇÃO

Turma 3:

Antero Gandra, up201607926

Francisco Moreira, up201607929

Sumário

Realizado no contexto da unidade curricular de Redes de Computadores, foi requisitado aos alunos a implementação de um protocolo de comunicação assíncrona para a transmissão de um ficheiro através de uma porta série RS-232.

Introdução

Este projeto tem como objectivo a implementação de uma aplicação na linguagem C com o intuito de transmitir um ficheiro pela porta série entre 2 computadores. Mais relevante que apenas esta transmissão é a elaboração de um protocolo de ligação eficiente e robusto para lidar com possíveis erros nesta transmissão. Além disso é desejavel que a aplicação seja bastante abstrata de modo que algumas funções e funcionalidades possam ser usadas para outros fins que não apenas a transferência de um ficheiro. Por fim pretende-se analisar a solução obtida, tal é o objetivo deste relatório.

O relatório encontra-se dividido nas seguintes secções:

- Arquitetura – onde se descrevem os diferentes blocos funcionais e interfaces;
- Estrutura do código – onde se descrevem as APIs, as principais estruturas de dados, as principais funções e a relação das funções com a arquitetura);
- Casos de uso principais - onde se identificam os casos principais de uso e as sequências de chamada de funções para cada;
- Protocolo de ligação lógica- onde se identificam os principais aspetos funcionais e se descreve a sua estratégia de implementação;
- Protocolo de aplicação - onde se identificam os principais aspetos funcionais e se descreve a sua estratégia de implementação;
- Validação - onde se descrevem os testes efetuados;
- Eficiência do protocolo de ligação de dados- onde fazemos uma caraterização estatística da eficiência do protocolo, com recurso a medidas sobre o código desenvolvido.
- Conclusões - onde resumimos a informação apresentada nas secções anteriores e refletimos sobre os objetivos de aprendizagem alcançados

Arquitetura

Camadas de protocolo

O trabalho está organizado em duas camadas – layers – que permitem o correto funcionamento da aplicação: a camada do protocolo de ligação de dados e a camada de aplicação, que estão implementados em diferentes ficheiros source e header. Os ficheiros link.c e link.h representam a camada de ligação de dados enquanto os ficheiros application.c e application.h representam a camada de aplicação.

A camada do protocolo de ligação de dados contém funções genéricas que relacionadas com a porta série, tratando do estabelecimento da ligação e da transferência de dados e deteção de erros.

A camada de aplicação por outro lado contém funções mais especificas relacionadas, neste caso, com a transferência de um ficheiro pela porta serie.

Interface e opções

A interface é proporcionada pelos ficheiros da camada da aplicação que indica informação sobre o estado da transferência do ficheiro. No entanto, a camada de ligação de dados pode por vezes colocar informação na interface quando ocorrem erros.

As opções de ligação como baud rate, timeout, timeout tries e tamanho máximo da mensagem podem ser alteradas no ficheiro settings.txt. Adicionalmente existe uma opção neste ficheiro que indica a possibilidade de ser gerado um erro aleatório em cada mensagem transferida. Esta opção foi usada para testar e comparar a eficiência do protocolo, deve obviamente ser colocada a 0 para uso real.

Estrutura do código

O código encontra-se dividido em 3 ficheiros correspondentes às 2 camadas da arquitetura e um outro ficheiro alarme.c que trata de processar timeouts gerados pelo sinal SIGALRM do sistema.

Naturalmente cada um destes ficheiros possui um header file correspondente onde estão declaradas as funções e estruturas de dados usadas, assim como algumas macros.

Ligação de dados (link.c e link.h):

- Funções:
    - llopen: estabelece a ligação entre o 'writer' e o 'reader'.
    - llclose: fecha a ligação e entre o 'writer' e o 'reader'.
    - llwrite: envia uma mensagem e espera pela resposta.
    - llread: lê uma mensagem e verifica os erros.
    - stuff: realiza o stuffing das mensagens.
    - destuff: realiza o destuffing das mensagens.
    - processBCC: calcula o BCC.
    - receiveMessage: lê uma mensagem. (é chamada pelo llwrite).
    - identifyMessageControl: identifica se uma mensagem tem o tipo previsto.
    - sendCommand: prepara e envia um comando.
    - sendMessage: prepara e envia uma mensagem.
    - printStats: imprime na linha de comandos as estatisticas da ligação.
    - statisticsSetup: inicializa as estatisticas da ligação.
    - connectionSettings: lê as settings pretendidas para a ligação.
- Estruturas de dados:
    - State: Estados da máquina de estados
    - Mode: Modo da aplicação, 'writer' ou 'reader'
    - Control: Tipo do comando de controlo
    - MessageType: Tipo de mensagem
    - ErrorType: Tipo de erro
    - Data: lista de caracteres e tamanho da informação, usado em Message
    - Message: Informação de uma mensagem
    - Stats: Estatisticas da ligação
    - Settings: Opções da ligação

Aplicação (application.c e application.h):

- Funções:
    - main: Inicio do programa e processamento dos parâmetros
    - sendFile: Envio de um ficheiro
    - receiveFile: Receção de um ficheiro
    - receiveControl: Receção de um pacote de controlo
    - receiveData: Receção de um pacote de dados
    - sendControl: Envio de um pacote de controlo
    - sendData: Envio de um pacote de dados
    - fileSize: Lê o tamanho de um ficheiro
    - openFile: Abre um ficheiro
- Estruturas de dados:
    - AppControlType: Tipo de pacote
    - Parameter: Tipo de parâmetro no pacote de controlo (Size ou Name)

Alarme (alarm.c e alarm.h):

- Funções:
    - alarmHandler: Handler do alarme.
    - setAlarm: Configura o alarme.
    - stopAlarm: Para o alarme.

Casos de uso principais

Os casos de uso desta aplicação são a interface em que se escolhe o ficheiro e a porta série a usar e a transferência desse ficheiro.

Na interface o utilizador dá inicio à aplicação com os argumentos porta série e ficheiro a ser enviado. No caso do receptor apenas é necessária a porta série.

A transferência do ficheiro ocorre na seguinte sequência:

0. Configuração da ligação usando o ficheiro settings.txt
1. Recetor inicia-se com número da porta e fica à espera
2. Transmissor estabelece a ligação com o receptor
3. Transmissor envia os dados
4. Recetor recebe os dados e guarda num ficheiro
5. Ligação é terminada

Protocolo de ligação lógica

Camada de ligação de dados: camada baixo nível responsável pela interação direta com a porta série.

As funcionalidades implementadas por esta camada são:

- abertura e fecho da porta série;
- leitura e escrita de tramas de informação e controlo;
- criação de tramas de controlo;
- byte stuffing e byte destuffing, framing e deframing de uma trama;

Funções implementadas:

- llread - responsável pela leitura de informação através da porta série,
    1. Aplica destufffing e deframing na trama recebida.
    2. Se ocorrer um erro no destuffing ou na frame diferente de BCC2, a trama é descartada e espera por uma nova.
    3. Se houver um erro no BCC2, a trama é descartada, o recetor envia uma trama de controlo REJ, apesar de continuar á espera de uma nova.
    4. Se a trama for recebida corretamente, é retornada e o recetor envia uma trama de controlo tipo RR.
- llwrite - responsável pelo envio de informação através da Porta Série.
    1. Recebe a mensagem a enviar da camada superior e aplica-lhe framing e stuffing.
    2. Tenta escrever a trama e se não receber uma resposta RR durante um intervalo de tempo anteriormente definido, este reenvia a trama um determinado número de vezes.
    3. Se não tiver sucesso retorna erro.
- llopen - responsável por abrir a ligação pela porta série com as configurações escolhidas e estabelecer a ligação com SET do ‘writer’. Depois aguarda pela resposta UA do ‘reader’.
- llclose – responsável por finalizar a ligação pela porta série trocando comandos DISC entre o writer e o reader e um UA do writer para confirmar o fim da ligação. No fim as definições iniciais da porta série são repostas.
- stuff – que faz o byte stuffing da trama.
- destuff – que faz o byte destuffing da trama.

- identifyMessageControl – que verifica o tipo de uma mensagem de controlo.
- receiveMessage – que lê uma trama e faz o seu processamento, incluindo verificar todos os campos, fazer destuffing, verificar erros BCC e verificar tipo de mensagem por fim retornando uma estrutura de dados com a informação importante.
- processBCC - que recebe uma lista de caracteres e retorna o seu BCC respondente. É usada para depois comparar o BCC recebido e o esperado.
- sendCommand – que prepara e envia uma trama de controlo (Supervisão/Não Numerada).
- sendMessage – que prepara e envia uma trama com dados (Informação).
- connectionSettings – que abre o ficheiro de configurações 'settings.txt' e coloca essas configurações numa estrutura de dados usada pela camada lógica durante a sua execução.
- statisticsSetup – que inicializa as estatisticas a 0 e guarda o valor do tempo.
- printStats – que imprime no ecrã as estatísticas da ligação.

Protocolo de aplicação

Camada de aplicação: camada de alto nível que trata do envio e da receção do ficheiro fonte através do API fornecido pela camada da ligação de dados.

As funcionalidades implementadas por esta camada são:

- iniciar a ligação;
- se for emissor: ler o ficheiro fonte e dividi-lo em pacotes de dados;
- se for recetor: recompor o ficheiro fonte através de pacotes de dados;
- enviar e receber os pacotes;
- terminar a ligação.

Foi assim desenvolvido o API (application.c) que gere a interação dos pacotes de dados

Em relação aos pacotes de dados:

- sendData que cria e envia o pacote de dados (utilizando llwrite);
- receiveData que recolhe a informação do ficheiro original e recebe o pacote de dados (utilizando llread).

Em relação aos pacotes de controlo:

- sendControl que cria e envia o pacote de controlo (utilizando llwrite);
- receiveControl que recebe o pacote (utilizando llread).

Outras:

- sendFile (emissor) que inicia a ligação, lê o ficheiro fonte e divide em pacotes de dados, envia-os e termina a ligação. Utiliza 3 funções do API da ligação de dados: llopen, llwrite e llclose e API de pacotes.
- receiveFile (recetor) que inicia a ligação, lê os pacotes de dados, reconstrói o ficheiro fonte e termina a ligação. Utiliza 3 funções do API da ligação de dados: llopen, llread e llclose e API de pacotes.
- openFile (emissor) que abre o ficheiro pedido para ser enviado.
- fileSize (emissor) que retorna o tamanho de um ficheiro

Validação

Para validar do programa desenvolvido, garantindo, desta forma, que funcionava corretamente com o respetivo protocolo, foram realizados vários testes durante o desenvolvimento e demonstração do programa.

Foram testados vários ficheiros, de diferentes tipos e tamanhos e enviados com diferentes baudrates e tamanhos de pacotes de informação.

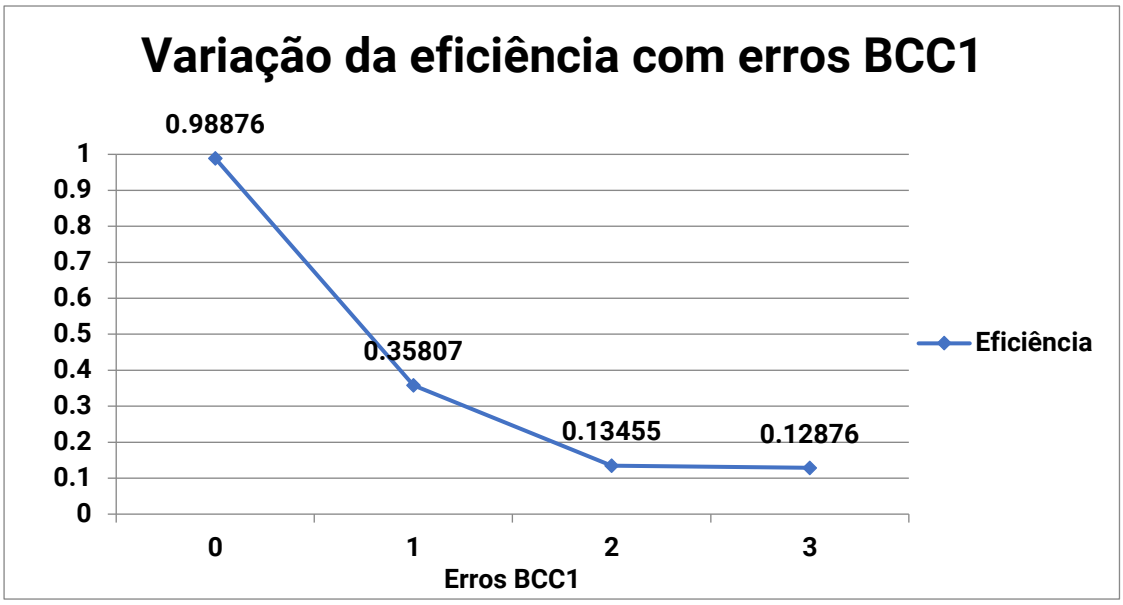
Também foram realizados testes de interrupção da comunicação na Porta Série e de introdução de erros através do curto-circuito existente nas portas.

Todos estes testes foram realizados no momento de avaliação, na presença do professor.

Eficiência do protocolo de ligação de dados

De forma a avaliar a eficiência do protocolo e tirar mais conclusões foram realizados vários testes como foi proposto. Todos os testes foram feitos fazendo a média de 3 medições para diminuir desvios e irregularidades.

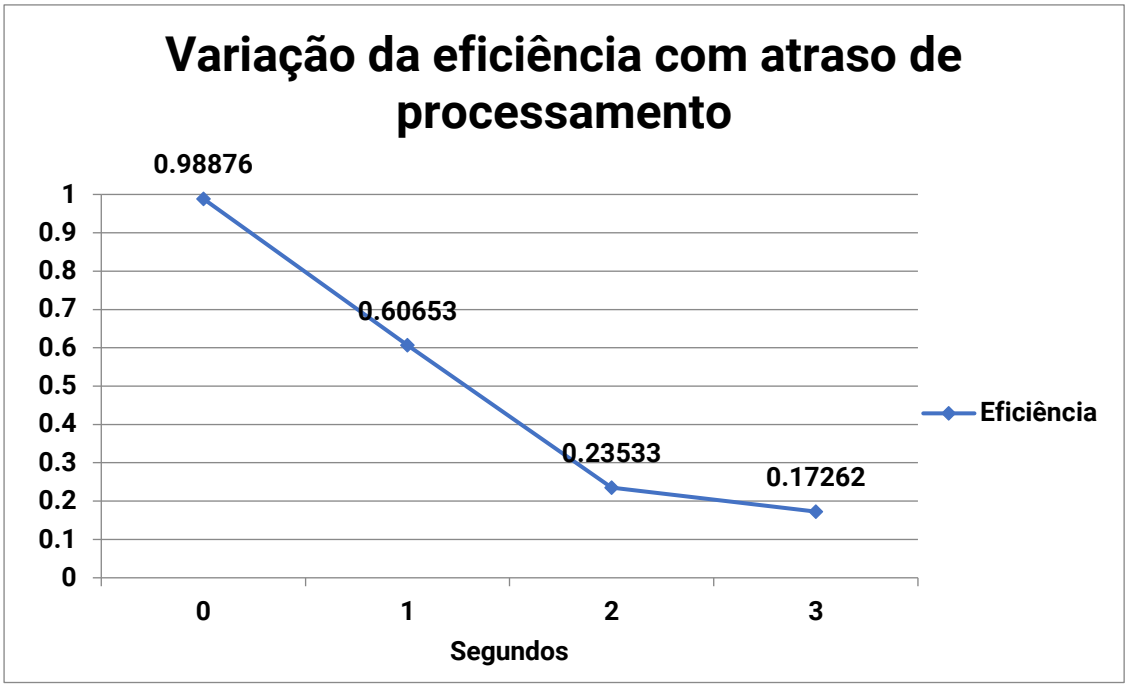
- Variar FER



A partir destes gráficos podemos verificar que tal como esperado a eficiência do protocolo baixa drasticamente em caso de erros de BCC1 visto que o recetor não responde e o emissor tem de esperar pelo timeout.

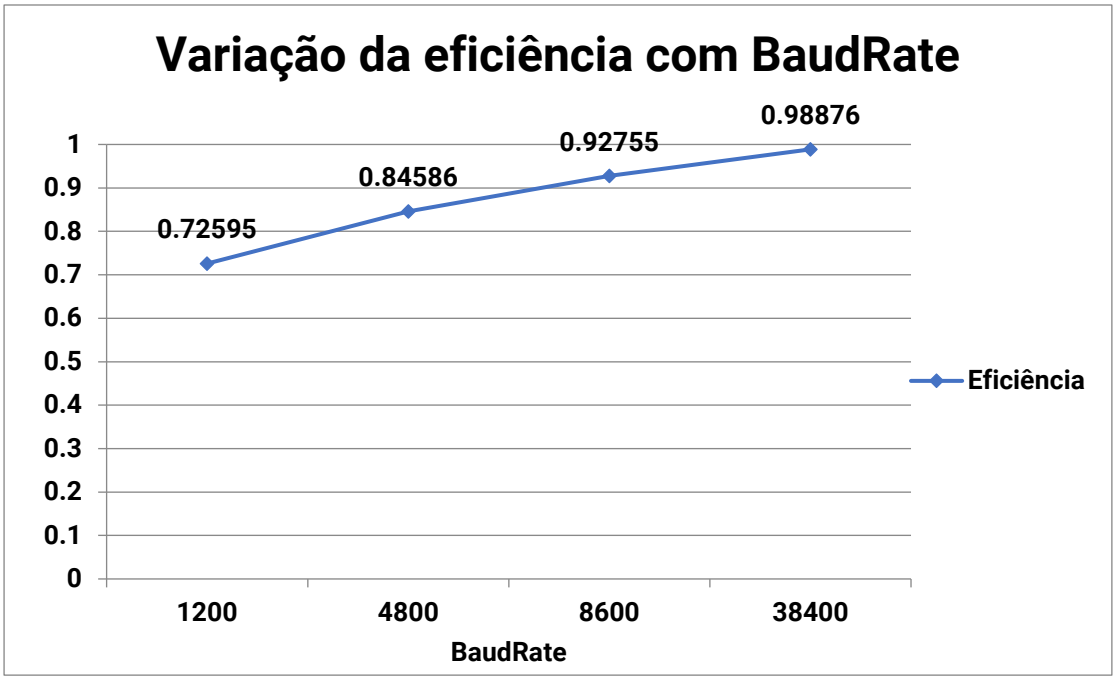
Além disso os erros de BCC2 não causam grande alteração da eficiência do protocolo visto que nestes erros a trama é reenviada imediatamente.

- Variar T_prop



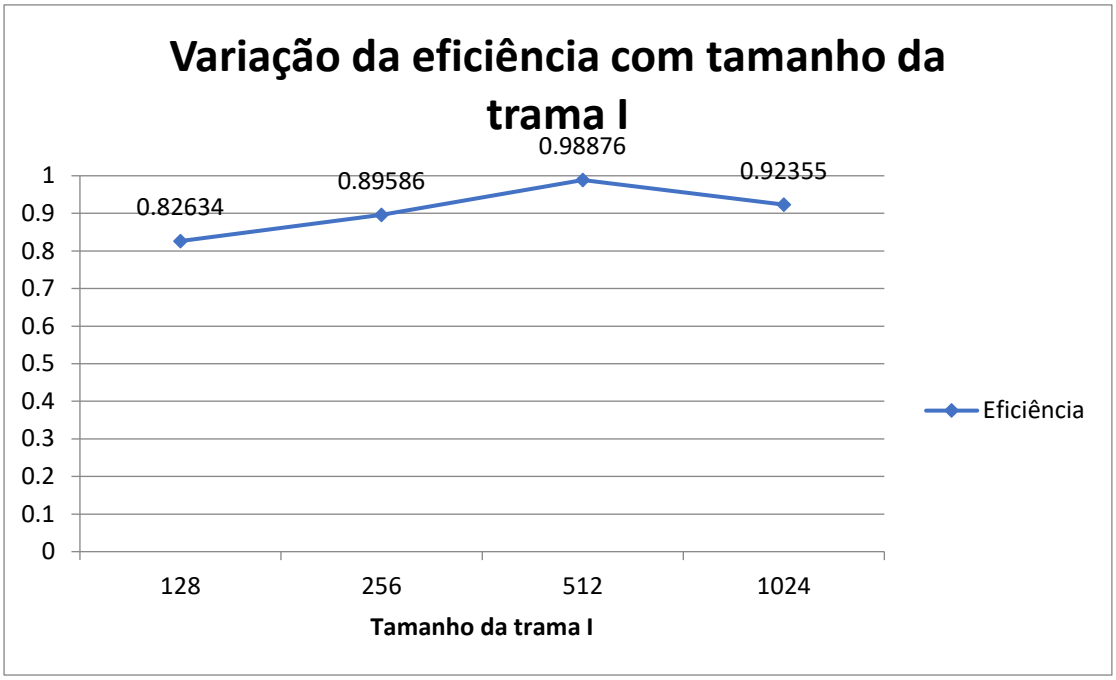
Como seria de esperar o atraso no processamento de cada trama faz diminuir muito a eficiência do protocolo.

- Variar C(BaudRate)



O aumento do BaudRate(C) faz aumentar a eficiência do protocolo visto que os dados são transmitidos mais rapidamente.

- Variar tamanho dos pacotes (tramas I)



De forma geral o aumento do tamanha da trama I faz aumentar a eficiência do protocolo até ao tamanho 512, no entanto aumentando para 1024 nota-se uma diminuição da eficiência. Isto poderá ocorrer porque o processamento dos dados demora mais tempo e tal como vimos anteriormente na variação de T_prop isto faz diminuir a eficiência.

Em relação ao protocolo utilizado, a nossa aplicação baseia-se no protocolo Stop and Wait para controlo e recuperação de erros. Quando o emissor manda qualquer tipo de tramas fica à espera de uma resposta do emissor. Essa resposta poderá ser positiva ou negativa caso se verifiquem erros e assim o emissor saberá se é necessário reenviar a mesma trama. Para tratar duplicados é usada uma numeração das tramas que pode ser 0 ou 1.

Conclusões

O grupo considera que este projeto foi concluído com sucesso tendo produzido uma aplicação eficaz para a transferência de um ficheiro pela porta série, mas também reforçado os seus conhecimentos sobre os procedimentos de mais baixo nível envolvidos nesta operação.

Adicionalmente a noção de independência entre camadas no desenvolvimento desta aplicação foi bastante elucidativa e interessante.

Relativamente ao progresso do projeto é de notar que o facto do seu desenvolvimento necessitar de uma configuração de 2 computadores ligados por porta série dificultou bastante o rapido desenvolvimento deste.

Nas primeiras semanas o grupo esteve dependente de progredir no projeto nas aulas práticas e em outros momentos na mesma sala do departamento. Apenas foi possível progredir efetivamente após configurar nos computadores próprios uma ligação de porta série simulada usando 2 máquinas virtuais.

Anexo I – Código fonte

alarm.h

```
#pragma once

#define TRUE 1
#define FALSE 0

extern int alarmFired;

void alarmHandler(int signal);

void setAlarm();

void stopAlarm();
```

alarm.c

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

#include "alarm.h"
#include "link.h"

int alarmFired = FALSE;

void alarmHandler(int signal)
{
    if (signal != SIGALRM)
        return;

    alarmFired = TRUE;

    stats->timeouts++;

    printf("Connection time out!\n");

    //Set alarm again
    alarm(settings->timeout);
}

void setAlarm()
{
```

```
    //Setup
    struct sigaction action;
    action.sa_handler = alarmHandler;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;

    sigaction(SIGALRM, &action, NULL);

    //Set alarm
    alarmFired = FALSE;
    alarm(settings->timeout);
}

void stopAlarm()
{

    //Setup
    struct sigaction action;
    action.sa_handler = alarmHandler;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;

    sigaction(SIGALRM, &action, NULL);

    //Block alarm
    alarm(0);
}
```

application.h

```
#pragma once

typedef enum
{
    CTRL_START = 1,
    CTRL_DATA = 2,
    CTRL_END = 3
} AppControlType;

typedef enum
{
    FILE_SIZE,
    FILE_NAME
} Parameter;

FILE *openFile(char *fileName);
int fileSize(FILE *file);
```

```
void sendControl(int fd, int cmd, char *fileS, char *fileName);
void sendData(int fd, int N, const char* buffer, int length);

void sendFile(char *fileName, int fd);

char* receiveControl(int fd, int* controlPackageType, int* fileLength);
void receiveData(int fd, int* N, char** buf, int* length);

void receiveFile(int fd);
```

application.c

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

#include "application.h"
#include "link.h"

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

    //Random seed
    srand(time(NULL));

    //Incorrect argument number
    if (argc < 2)
    {
        printf("Usage:\tnserial SerialPort file File\n\tex: 0
pinguim.gif\n");
        exit(1);
    }

    //Get port
    int portN = atoi(argv[1]);

    //Incorrect port identifier
    if (portN < 0 || portN > 3)
    {
```

```
        printf("Usage:\tnserial SerialPort file File\n\tex: 0
pinguim.gif\n");
        exit(1);
    }

    //Proper port format
    char port[11] = "/dev/ttyS";
    strcat(port, argv[1]);

    printf("Using port: %s\n", port);

    //File to be sent (and Mode assignment)
    Mode mode;
    if (argc == 3)
    {
        mode = WRITER;
        printf("File provided to be sent: %s\n", argv[2]);
    }
    else
    {
        mode = READER;
        printf("No file provided, receiving\n");
    }

    //Setup Link Settings
    connectionSettings(port, mode);

    //Setup Statistics
    statisticsSetup();

    //File descriptor of connection
    int fd;

    //Open connection
    fd = llopen();

    //Send/Receive file
    if (settings->mode == WRITER)
        sendFile(argv[2], fd);
    else
        receiveFile(fd);

    //Close connection
    llclose(fd);

    //Print Statistics
    printStats();
```

```
    return 0;
}

FILE *openFile(char *fileName)
{

    //Open file
    FILE *file = fopen(fileName, "rb");

    if (!file)
    {
        printf("ERROR: Could not open file, exiting...\n");
        exit(ERROR);
    }

    return file;
}

int fileSize(FILE *file)
{

    //Save start position
    long int currentPosition = ftell(file);

    //End of file
    if (fseek(file, 0, SEEK_END) == -1)
    {
        printf("ERROR: Could not get file size.\n");
        exit(ERROR);
    }

    //Size
    long int size = ftell(file);

    //Recover start position
    fseek(file, 0, currentPosition);

    return size;
}

void sendControl(int fd, int cmd, char *fileS, char *fileName)
{

    //Package size
    int packageS = strlen(fileS) + strlen(fileName) + 5;

    //Setup package
    unsigned char controlPackage[packageS];
```

```
    int index = 0;
    int i;

    //Command
    controlPackage[index++] = cmd;

    //File size
    controlPackage[index++] = FILE_SIZE;
    controlPackage[index++] = strlen(fileS);
    for (i = 0; i < strlen(fileS); i++)
        controlPackage[index++] = fileS[i];

    //File name
    controlPackage[index++] = FILE_NAME;
    controlPackage[index++] = strlen(fileName);
    for (i = 0; i < strlen(fileName); i++)
        controlPackage[index++] = fileName[i];

    //Print information
    if (cmd == CTRL_START)
    {
        printf("File name: %s\n", fileName);
        printf("File size: %s (bytes)\n", fileS);
    }

    //Send package
    if (!llwrite(fd, controlPackage, packageS))
    {
        printf("ERROR: Could not send control package to link\n");
        free(controlPackage);
        exit(ERROR);
    }
}

void sendData(int fd, int N, const char *buffer, int length)
{

    //Construct header
    unsigned char C = CTRL_DATA;
    unsigned char L2 = length / 256;
    unsigned char L1 = length % 256;

    //Package size
    int packageSize = 4 + length;

    //Allocate all space
    unsigned char *package = (unsigned char *)malloc(packageSize);
```

```
    //Package Header
    package[0] = C;
    package[1] = N;
    package[2] = L2;
    package[3] = L1;

    //Copy chunk to package
    memcpy(&package[4], buffer, length);

    //Write package
    if (!llwrite(fd, package, packageSize))
    {
        printf("ERROR: Could not send data package to link\n");
        free(package);
        exit(ERROR);
    }

    free(package);
}

void sendFile(char *fileName, int fd)
{

    //Open file to be sent
    FILE *file = openFile(fileName);

    //File size
    int fileS = fileSize(file);
    char fileSizeBuf[sizeof(int) * 3 + 2];
    snprintf(fileSizeBuf, sizeof fileSizeBuf, "%d", fileS);

    //Start Packet
    sendControl(fd, CTRL_START, fileSizeBuf, fileName);

    printf("Sending...\n");

    //File buffer
    int maxSize = settings->messageDataMaxSize;
    char *fileBuf = malloc(maxSize);

    //Read chunks
    int readBytes = 0, i = 0;
    while ((readBytes = fread(fileBuf, sizeof(char), maxSize, file)) >
0)
    {
        //Send data packet
        sendData(fd, (i++) % 255, fileBuf, readBytes);
```

```
        //Reset file buffer
        fileBuf = memset(fileBuf, 0, maxSize);
    }

    free(fileBuf);

    //Close file
    if (fclose(file) != 0)
    {
        printf("ERROR: Unable to close file.\n");
        exit(ERROR);
    }

    //End Packet
    sendControl(fd, CTRL_END, "0", "");

    printf("File successfully sent.\n");
}

char* receiveControl(int fd, int *controlPackageType, int *fileLength)
{

    //Read package
    unsigned char *package;
    if (llread(fd, &package) < 0)
    {
        printf("ERROR: Could not read control package from link\n");
        exit(ERROR);
    }

    //Control package type
    *controlPackageType = package[0];

    //Check if it's end
    if (*controlPackageType == CTRL_END)
        return NULL;

    //If not then extract information
    int i = 0, index = 1, octs = 0;
    for (i = 0; i < 2; i++)
    {
        int paramType = package[index++];

        //Parameter is file size
        if (paramType == FILE_SIZE)
        {
            octs = (int)package[index++];
```

```
            char *length = malloc(octs);
            memcpy(length, &package[index], octs);
            *fileLength = atoi(length);
            free(length);
            index+=octs;
        }
        //Parameter is file name
        else if (paramType == FILE_NAME)
        {
            octs = (int)package[index++];
            char* buf = malloc(octs);
            memcpy(buf, &package[index], octs);
            buf[octs] = '\0';
            return buf;
        }
    }

    return NULL;

}

void receiveData(int fd, int* N, char** buf, int* length) {
    unsigned char* package;

    //Read package
    int size = llread(fd, &package);
    if (size < 0) {
        printf("ERROR: Could not read data package from link\n");
        exit(ERROR);
    }

    //Extract information
    int C = package[0];
    *N = (unsigned char) package[1];
    int L2 = package[2];
    int L1 = package[3];

    //Check if it's Data
    if (C != CTRL_DATA) {
        printf("ERROR: Wrong data package received, expected
CTRL_DATA\n");
        exit(ERROR);
    }

    //Size of file chunk
    *length = L1 + 256 * L2;

    //File chunk
```

```
    *buf = malloc(*length);

    //Copy to buffer
    memcpy(*buf, &package[4], *length);

    free(package);

}

void receiveFile(int fd)
{

    //Receive control
    int controlStart;
    int fileSize;
    char *fileName = receiveControl(fd, &controlStart, &fileSize);

    //Not start control package
    if (controlStart != CTRL_START)
    {
        printf("ERROR: Wrong control package received, expected
CTRL_START");
        exit(ERROR);
    }

    //Create file
    FILE *file = fopen(fileName, "wb");
    if (file == NULL)
    {
        printf("ERROR: Failed to create output\n");
        exit(ERROR);
    }

    printf("Created output file: %s\n", fileName);
    printf("File size: %d (bytes)\n", fileSize);
    printf("Receiving...\n");

    //Receive data
    int total = 0, N = -1;
    while (total != fileSize)
    {
        int lastN = N;
        char *fileBuf = NULL;
        int length = 0;

        //Receive data
        receiveData(fd, &N, &fileBuf, &length);
```

```
        //Check sequence
        if (N != 0 && lastN + 1 != N)
        {
            printf("ERROR: Received sequence no. was %d instead of
%d.\n", N, lastN + 1);
            free(fileBuf);
            exit(ERROR);
        }

        //Write to file
        fwrite(fileBuf, sizeof(char), length, file);

        free(fileBuf);

        //Add to total
        total += length;
    }

    //Close file
    if (fclose(file) != 0)
    {
        printf("ERROR: Failed to close file\n");
        exit(ERROR);
    }

    //Receive end control
    int controlPackageTypeReceived = -1;
    receiveControl(fd, &controlPackageTypeReceived, 0);

    //Not end control package
    if (controlPackageTypeReceived != CTRL_END)
    {
        printf("ERROR: Wrong control package received, expected
CTRL_END");
        exit(ERROR);
    }

    printf("File successfully received.\n");
}
```

link.h

```
#pragma once

#include <termios.h>
#include <time.h>

#define ERROR -1

typedef enum
{
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    STOP
} State;

typedef enum
{
    WRITER,
    READER
} Mode;

typedef enum
{
    C_SET = 0x03,
    C_UA = 0x07,
    C_RR = 0x05,
    C_REJ = 0x01,
    C_DISC = 0x0B
} Control;

typedef enum
{
    COMMAND,
    DATA,
    INVALID
} MessageType;

typedef enum
{
    INPUT_OUTPUT_ERROR,
    BCC1_ERROR,
    BCC2_ERROR
} ErrorType;
```

```
struct Data
{
    unsigned char *message;
    int size;
};

typedef struct
{
    MessageType type;
    ErrorType error;
    Control control;

    int ns;
    int nr;

    struct Data data;

} Message;

typedef struct
{
    struct timespec startTime;

    int sent;
    int received;

    int timeouts;

    int sentRR;
    int sentREJ;
    int receivedRR;
    int receivedREJ;

} Stats;

#define BIT(n) (0x01 << n)

typedef struct
{

    char port[20];
    Mode mode;
    int baudRate;
    int messageDataMaxSize;
    int ns;
    int timeout;
    int numTries;
```

```
    char frame[256];
    struct termios oldtio, newtio;
    int errorChance;

} Settings;

extern Settings *settings;
extern Stats *stats;

void connectionSettings(char *port, Mode mode);
void statisticsSetup();
void printStats();

void sendCommand(int fd, Control com);
void sendMessage(int fd, const unsigned char *message, int
messageSize);

int identifyMessageControl(Message *msg, Control command);
Message *receiveMessage(int fd);
unsigned char processBCC(const unsigned char *buf, int size);

int stuff(unsigned char **buf, int bufSize);
int destuff(unsigned char **buf, int bufSize);

int llopen();
int llclose(int fd);
int llwrite(int fd, const unsigned char *buf, int bufSize);
int llread(int fd, unsigned char **message);
```

link.c

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

#include "link.h"
#include "alarm.h"

Settings *settings;
```

```
Stats *stats;

const int FLAG = 0x7E;
const int A = 0x03;
const int ESCAPE = 0x7D;

//Identify Baudrate
int findBaudrate(char *baudrateS)
{

    int baudrate = atoi(baudrateS);

    switch (baudrate)
    {
    case 0:
        return B0;
    case 50:
        return B50;
    case 75:
        return B75;
    case 110:
        return B110;
    case 134:
        return B134;
    case 150:
        return B150;
    case 200:
        return B200;
    case 300:
        return B300;
    case 600:
        return B600;
    case 1200:
        return B1200;
    case 1800:
        return B1800;
    case 2400:
        return B2400;
    case 4800:
        return B4800;
    case 9600:
        return B9600;
    case 19200:
        return B19200;
    case 38400:
        return B38400;
    default:
        return -1;
```

```
    }
}

//Setup connection settings
void connectionSettings(char *port, Mode mode)
{

    //Allocate Settings
    settings = (Settings *)malloc(sizeof(Settings));

    //Settings file
    FILE *settingsFile = fopen("settings.txt", "r");

    //Read fields
    char data[256];

    //Baud rate
    char *baud;
    if (fgets(data, 256, settingsFile) != NULL)
        baud = &data[9];
    int len = strlen(baud);
    baud[len - 1] = '\0';

    printf("Baud rate set to: %s\n", baud);

    settings->baudRate = findBaudrate(baud);

    //Max size
    char *size;
    if (fgets(data, 256, settingsFile) != NULL)
        size = &data[12];
    len = strlen(size);
    size[len - 1] = '\0';

    printf("Size set to: %s(bytes)\n", size);

    settings->messageDataMaxSize = atoi(size);

    //Timeout
    char *timeout;
    if (fgets(data, 256, settingsFile) != NULL)
        timeout = &data[8];
    len = strlen(timeout);
    timeout[len - 1] = '\0';

    printf("Timeout set to: %s(seconds)\n", timeout);

    settings->timeout = atoi(timeout);
```

```
    //Tries
    char *tries;
    if (fgets(data, 256, settingsFile) != NULL)
        tries = &data[6];
    len = strlen(tries);
    tries[len - 1] = '\0';

    printf("Tries set to: %s(attempts)\n", tries);

    settings->numTries = atoi(tries);

     //Error chance
    char *error;
    if (fgets(data, 256, settingsFile) != NULL)
        error = &data[6];
    len = strlen(error);
    error[len - 1] = '\0';

    printf("Error chance set to: %s(%% error chance per message)\n",
error);

    settings->errorChance = atoi(error);

    strcpy(settings->port, port);
    settings->mode = mode;
    settings->ns = 0;

}

//Initialize statistics
void statisticsSetup(){

    stats = (Stats *)malloc(sizeof(Stats));

    //Start clock
    clock_gettime(CLOCK_REALTIME, &stats->startTime);

    //Reset values
    stats->sent = 0;
    stats->received = 0;

    stats->timeouts = 0;

    stats->sentRR = 0;
    stats->sentREJ = 0;
    stats->receivedRR = 0;
    stats->receivedREJ = 0;
```

```
}

double timeSpecToSeconds(struct timespec* ts){
    return (double)ts->tv_sec + (double)ts->tv_nsec / 1000000000.0;
}

//Print statistics
void printStats(){
    printf("Connection statistics:\n");

    struct timespec endTime;
    clock_gettime(CLOCK_REALTIME, &endTime);
    printf("\t- Total time: %lf seconds\n",
(timeSpecToSeconds(&endTime)-timeSpecToSeconds(&stats->startTime)));

    printf("\t- Messages sent: %d\n", stats->sent);
    printf("\t- Messages received: %d\n", stats->received);
    printf("\t- Timeouts occured: %d\n", stats->timeouts);
    printf("\t- RR sent: %d\n", stats->sentRR);
    printf("\t- REJ sent: %d\n", stats->sentREJ);
    printf("\t- RR received: %d\n", stats->receivedRR);
    printf("\t- REJ received: %d\n", stats->receivedREJ);
}

//Prepares and sends command to fd
void sendCommand(int fd, Control com)
{

    //Command size
    const int commandMaxSize = 5 * sizeof(char);

    //Prepare command
    unsigned char *command = malloc(commandMaxSize);

    command[0] = FLAG;
    command[1] = A;
    command[2] = com;
    if (com == C_REJ || com == C_RR)
        command[2] |= (settings->ns << 7);
    command[3] = command[1] ^ command[2];
    command[4] = FLAG;

    //Stuffing
    int commandSize = stuff(&command, (commandMaxSize));

    //Send command
    if (write(fd, command, commandSize) != (commandMaxSize))
        printf("ERROR: Could not write %s command.\n", command);
```

```
    //Free command
    free(command);

    if (com == C_REJ)
        stats->sentREJ++;
    else if (com == C_RR)
        stats->sentRR++;

}

int identifyMessageControl(Message *msg, Control command)
{
    return msg->type == COMMAND && msg->control == command;
}

Message *receiveMessage(int fd)
{
    //Setup message
    Message *msg = (Message *)malloc(sizeof(Message));
    msg->type = INVALID;
    msg->ns = msg->nr = -1;

    State state = START;

    int size = 0;
    unsigned char *message = malloc(settings->messageDataMaxSize);

    //State Machine
    volatile int done = FALSE;
    while (!done)
    {
        unsigned char ch;

        //Not stopping yet
        if (state != STOP)
        {
            //Read
            int numReadBytes = read(fd, &ch, 1);

            //Empty
            if (!numReadBytes)
            {
                free(message);

                msg->type = INVALID;
                msg->error = INPUT_OUTPUT_ERROR;
```

```
                return msg;
            }
        }

        //State jumping
        switch (state)
        {
        case START:
            if (ch == FLAG)
            {
                message[size++] = ch;

                state = FLAG_RCV;
            }
            break;
        case FLAG_RCV:
            if (ch == A)
            {
                message[size++] = ch;

                state = A_RCV;
            }
            else if (ch != FLAG)
            {
                size = 0;

                state = START;
            }
            break;
        case A_RCV:
            if (ch != FLAG)
            {
                message[size++] = ch;

                state = C_RCV;
            }
            else if (ch == FLAG)
            {
                size = 1;

                state = FLAG_RCV;
            }
            else
            {
                size = 0;

                state = START;
            }
```

```
            break;
        case C_RCV:
            if (ch == (message[1] ^ message[2]))
            {
                message[size++] = ch;

                state = BCC_OK;
            }
            else if (ch == FLAG)
            {
                size = 1;

                state = FLAG_RCV;
            }
            else
            {
                size = 0;

                state = START;
            }
            break;
        case BCC_OK:
            if (ch == FLAG)
            {
                if (msg->type == INVALID)
                    msg->type = COMMAND;

                message[size++] = ch;

                state = STOP;
            }
            else if (ch != FLAG)
            {
                if (msg->type == INVALID)
                    msg->type = DATA;

                //Need to expand space
                if (size % settings->messageDataMaxSize == 0)
                {
                    int mFactor = size / settings->messageDataMaxSize +
1;
                    message = (unsigned char *)realloc(message, mFactor
* settings->messageDataMaxSize);
                }

                message[size++] = ch;
            }
            break;
```

```
        case STOP:
            message[size] = 0;
            done = TRUE;
            break;
        default:
            break;
        }
    }

    //Destuff
    size = destuff(&message, size);

    unsigned char A = message[1];
    unsigned char C = message[2];
    unsigned char BCC1 = message[3];

    //BCC1 check (header)
    if (BCC1 != (A ^ C))
    {
        printf("ERROR: invalid BCC1.\n");

        free(message);

        msg->type = INVALID;
        msg->error = BCC1_ERROR;

        return msg;
    }

    //Message is a command
    if (msg->type == COMMAND)
    {

        //Identify command
        switch (message[2] & 0x0F)
        {
        case C_SET:
            msg->control = C_SET;
            break;
        case C_UA:
            msg->control = C_UA;
            break;
        case C_RR:
            msg->control = C_RR;
            break;
        case C_REJ:
            msg->control = C_REJ;
            break;
```

```
        case C_DISC:
            msg->control = C_DISC;
            break;
        default:
            printf("ERROR: control field not recognized.\n");
            msg->control = C_SET;
        }

        //Control
        Control control = message[2];

        if (msg->control == C_RR || msg->control == C_REJ)
            msg->nr = (control >> 7) & BIT(0);

        if (msg->control == C_REJ)
            stats->receivedREJ++;
        else if (msg->control == C_RR)
            stats->receivedRR++;

    }
    //Message is data
    else if (msg->type == DATA)
    {

        stats->received++;

        msg->data.size = size - 6 * sizeof(char);

        //Check BCC2 (data)
        unsigned char calcBCC2 = processBCC(&message[4], msg-
>data.size);
        unsigned char BCC2 = message[4 + msg->data.size];

        if (calcBCC2 != BCC2)
        {
            printf("ERROR: invalid BCC2: 0x%02x != 0x%02x.\n",
calcBCC2, BCC2);

            free(message);

            msg->type = INVALID;
            msg->error = BCC2_ERROR;

            return msg;
        }

        msg->ns = (message[2] >> 6) & BIT(0);
```

```
        //Copy the message
        msg->data.message = malloc(msg->data.size);
        memcpy(msg->data.message, &message[4], msg->data.size);

    }

    free(message);

    return msg;
}

void sendMessage(int fd, const unsigned char *message, int messageSize)
{

    //Setup message
    unsigned char *msg = malloc(6 * sizeof(char) + messageSize);

    unsigned char C = settings->ns << 6;
    unsigned char BCC1 = A ^ C;
    unsigned char BCC2 = processBCC(message, messageSize);

    msg[0] = FLAG;
    msg[1] = A;
    msg[2] = C;
    msg[3] = BCC1;

    memcpy(&msg[4], message, messageSize);

    //Induce Error Header
    if(rand()%100 > (100-settings->errorChance)){
        printf("Randomly added header error...\n");
        msg[1] ++;
    }

    //Induce Error Data
    if(rand()%100 > (100-settings->errorChance)){
        printf("Randomly added data error...\n");
        msg[5] ++;
    }

    msg[4 + messageSize] = BCC2;
    msg[5 + messageSize] = FLAG;

    messageSize += 6 * sizeof(char);

    //Stuffing
    messageSize = stuff(&msg, messageSize);
```

```
    //Send
    int numWrittenBytes = write(fd, msg, messageSize);
    if (numWrittenBytes != messageSize)
        perror("ERROR: error while sending message.\n");

    free(msg);

    stats->sent++;

}

unsigned char processBCC(const unsigned char *buf, int size)
{
    unsigned char BCC = 0;

    int i = 0;
    for (; i < size; i++)
        BCC ^= buf[i];

    return BCC;
}

//Stuffing
int stuff(unsigned char **buf, int bufSize)
{
    int newBufSize = bufSize;

    int i;
    for (i = 1; i < bufSize - 1; i++)
        if ((*buf)[i] == FLAG || (*buf)[i] == ESCAPE)
            newBufSize++;

    *buf = (unsigned char *)realloc(*buf, newBufSize);

    for (i = 1; i < bufSize - 1; i++)
    {
        if ((*buf)[i] == FLAG || (*buf)[i] == ESCAPE)
        {
            memmove(*buf + i + 1, *buf + i, bufSize - i);

            bufSize++;

            (*buf)[i] = ESCAPE;
            (*buf)[i + 1] ^= 0x20;
        }
    }

    return newBufSize;
```

```
}

//Destuffing
int destuff(unsigned char **buf, int bufSize)
{
    int i;
    for (i = 1; i < bufSize - 1; ++i)
    {
        if ((*buf)[i] == ESCAPE)
        {
            memmove(*buf + i, *buf + i + 1, bufSize - i - 1);

            bufSize--;

            (*buf)[i] ^= 0x20;
        }
    }

    *buf = (unsigned char *)realloc(*buf, bufSize);

    return bufSize;
}

int llopen()
{

    printf("Estabilishing connection...\n");

    //=== Provided code ===

    int fd = open(settings->port, O_RDWR | O_NOCTTY);
    if (fd < 0)
    {
        perror(settings->port);
        exit(ERROR);
    }

    if (tcgetattr(fd, &settings->oldtio) == ERROR)
    {
        perror("tcgetattr");
        exit(ERROR);
    }

    bzero(&settings->newtio, sizeof(settings->newtio));
    settings->newtio.c_cflag = settings->baudRate | CS8 | CLOCAL |
CREAD;
    settings->newtio.c_iflag = IGNPAR;
    settings->newtio.c_oflag = 0;
```

```
    settings->newtio.c_lflag = 0;

    settings->newtio.c_cc[VTIME] = 1;
    settings->newtio.c_cc[VMIN] = 0;

    tcflush(fd, TCIOFLUSH);

    if (tcsetattr(fd, TCSANOW, &settings->newtio) == ERROR)
    {
        perror("tcsetattr error");
        exit(ERROR);
    }

    //========================

    int tries = 0, connected = 0;

    //Writer mode
    if (settings->mode == WRITER)
    {
        while (!connected)
        {
            if (tries == 0 || alarmFired)
            {

                //Reset alarm
                alarmFired = FALSE;

                //Number of tries exceeded
                if (tries >= settings->numTries)
                {
                    stopAlarm();
                    printf("ERROR: Maximum number of retries
exceeded.\n");
                    printf("Connection aborted\n");
                    exit(ERROR);
                }

                //Send SET command
                sendCommand(fd, C_SET);

                //Restart alarm
                if (++tries == 1)
                    setAlarm();
            }

            //Receive response
```

```
            if (identifyMessageControl(receiveMessage(fd), C_UA))
                connected = 1;
        }

        //Stop alarm
        stopAlarm();
    }
    //Reader mode
    else if (settings->mode == READER)
    {
        while (!connected)
        {
            //Receive setup and respond
            if (identifyMessageControl(receiveMessage(fd), C_SET))
            {
                sendCommand(fd, C_UA);
                connected = 1;
            }
        }
    }

    printf("Connection established\n");

    return fd;
}

int llclose(int fd)
{
    printf("Terminating connection...\n");

    int tries = 0;
    int in = TRUE;

    switch (settings->mode)
    {
    case WRITER:
    {
        while (in)
        {
            if (tries == 0 || alarmFired)
            {
                alarmFired = FALSE;

                if (tries >= settings->numTries)
                {
                    stopAlarm();
                    printf("ERROR: Maximum number of retries
exceeded.\n");
```

```
                    printf("Connection aborted\n");
                    return ERROR;
                }

                //Send disconnect
                sendCommand(fd, C_DISC);

                if (++tries == 1)
                    setAlarm();
            }

            //Receive disconnect
            if (identifyMessageControl(receiveMessage(fd), C_DISC))
                in = 0;
        }

        stopAlarm();

        //Send UA to finalize
        sendCommand(fd, C_UA);

        //Syncronize fd to make sure it sends C_UA command before
closing and resetting settings
        sync();

        printf("Connection terminated\n");

        break;
    }
    case READER:
    {
        while (in)
        {
            //Receive disconnect
            if (identifyMessageControl(receiveMessage(fd), C_DISC))
                in = 0;
        }

        int uaReceived = FALSE;
        while (!uaReceived)
        {
            if (tries == 0 || alarmFired)
            {
                alarmFired = FALSE;

                if (tries >= settings->numTries)
                {
                    stopAlarm();
```

```
                    printf("ERROR: Maximum number of retries
exceeded.\n");
                    printf("Connection aborted\n");
                    return ERROR;
                }

                //Send disconnect
                sendCommand(fd, C_DISC);

                if (++tries == 1)
                    setAlarm();
            }

            //Receive UA
            if (identifyMessageControl(receiveMessage(fd), C_UA))
                uaReceived = TRUE;
        }

        stopAlarm();
        printf("Connection terminated\n");

        break;
    }
    default:
        break;
    }

    //Reset oldtio
    if (tcsetattr(fd, TCSANOW, &settings->oldtio) == -1)
    {
        perror("tcsetattr");
        return 0;
    }

    //Close file descriptor
    close(fd);

    return ERROR;
}

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

    int tries = 0;

    while (1)
    {
        if (tries == 0 || alarmFired)
```

```
        {
            alarmFired = 0;

            if (tries >= settings->numTries)
            {
                stopAlarm();
                printf("ERROR: Maximum number of retries exceeded.\n");
                return 0;
            }

            //Send message
            sendMessage(fd, buf, bufSize);

            if (++tries == 1)
                setAlarm();
        }

        //Response
        Message *receivedMessage = receiveMessage(fd);

        //Receiver ready / positive ACK
        if (identifyMessageControl(receivedMessage, C_RR))
        {
            if (settings->ns != receivedMessage->nr)
                settings->ns = receivedMessage->nr;

            stopAlarm();
            break;
        }
        //Reject / negative ACK
        else if (identifyMessageControl(receivedMessage, C_REJ))
        {
            stopAlarm();
            tries = 0;
        }
    }

    stopAlarm();

    return 1;
}

int llread(int fd, unsigned char **message)
{
    Message *msg = NULL;

    int done = FALSE;
    while (!done)
```

```
    {
        //Read message
        msg = receiveMessage(fd);

        //Message type
        switch (msg->type)
        {
        case INVALID:
            //BCC error
            if (msg->error == BCC2_ERROR)
            {
                sendCommand(fd, C_REJ);
            }
            break;
        case COMMAND:
            //DISC command
            if (msg->control == C_DISC)
                done = TRUE;
            break;
        case DATA:
            //Check message order
            if (settings->ns == msg->ns)
            {
                *message = malloc(msg->data.size);
                memcpy(*message, msg->data.message, msg->data.size);
                free(msg->data.message);

                //Send response
                settings->ns = !msg->ns;
                sendCommand(fd, C_RR);

                done = TRUE;
            }
            else{
                printf("Wrong message ns associated: ignoring\n");
                settings->ns = msg->ns;
            }
            break;
        default:
            stopAlarm();
            return -1;
        }
    }

    //Alarm stopped by receiver
    stopAlarm();

    return 1;}
```