2022/23

Ligação de dados

Trabalho Laboratorial 1

Relatório

Rede de Computadores

Licenciatura em Engenharia Informática e Computação

Marco André Rocha up202004891@fe.up.pt

Ricardo André de Matos up202007962@fe.up.pt

Sumário

Este relatório foi realizado no âmbito da unidade curricular de Redes de Computadores (**RC**) e tem como incidência o primeiro trabalho que nos foi proposto e cuja principal meta era o desenvolvimento de uma aplicação robusta capaz de transferir dados entre duas máquinas através da porta série, tendo sido imperativa a implementação de um protocolo de comunicação.

Com este relatório visamos detalhar a implementação do trabalho bem como explicar os conceitos teóricos que foram postos em prática para o efeito, sendo de realçar que todas as metas que nos foram propostas foram alcançadas com sucesso.

Introdução

O trabalho pode ser dividido em dois objetivos concretos: elaboração da camada do protocolo de ligação de dados (**LL** - Link Layer) e elaboração da camada de aplicação (**AL** - Application Layer).

O propósito do **LL** é fornecer um serviço robusto de comunicação entre duas máquinas ligadas através de um canal de transmissão - porta série. Em adição, o **LL** deve fornecer à camada superior (mais concretamente o **AL**) uma interface de uso de forma a garantir a independência entre camadas.

No que concerne ao **AL**, este deve assegurar um simples protocolo de aplicação que permita um serviço de transferência estável de ficheiros que é oferecido pela camada inferior.

Pretendemos com este relatório explicar os conceitos teóricos aplicados e esclarecer as abordagens que tomamos durante a implementação do trabalho. Nesse sentido, e seguindo a estrutura que nos foi proposta, este relatório conta com a estrutura infra-mencionada:

- **Arquitetura -** blocos funcionais e interfaces;
- **Estrutura do código -** APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura;
- **Casos de uso principais -** identificação; sequências de chamada de funções;
- **Protocolo de ligação lógica -** identificação e descrição da estratégia de implementação dos principais aspetos funcionais;
- **Protocolo de aplicação -** identificação e descrição da estratégia de implementação dos principais aspetos funcionais;
- **Validação -** descrição dos testes efetuados e apresentação de resultados;
- **Eficiência do protocolo de ligação de dados -** caracterização estatística da eficiência do protocolo;
- **Conclusões -** síntese de informação e reflexão sobre os objetivos de aprendizagem alcançados;

Arquitetura

As duas camadas previamente mencionadas, **AL** e **LL**, encontram-se desenvolvidas nos ficheiros ***application_layer.c*** e ***link_layer.c***, com ficheiros auxiliares ***app_layer_utils.c*** e ***message.c***, respetivamente. Por sua vez, o **AL** atua como intermediário entre o utilizador e o **LL**, chamando as funções de interface oferecidas pelo **LL**.

O código, para além da divisão em duas camadas, está dividido também em dois blocos funcionais: emissor (**Tx**) e receptor (**Rx**). Ambos os blocos chamam funções das duas camadas desenvolvidas, todavia define-se uma distinção entre ambos, visto as funções **llwrite()** e **llread()** serem chamadas exclusivamente pelo **Tx** e **Rx**, respetivamente. Além disso, as funções **llopen()** e **llclose()** têm implementações diferentes para cada role para assegurar independência entre emissão e recepção.

Estrutura do código

O código-fonte do programa encontra-se dividido em **6 ficheiros** consoante as camadas e funcionalidades implementadas. A cada um destes ficheiros está associado um header file, havendo também três header files para definir ***macros*** relevantes.

link_layer.c

- **llopen() -** responsável por estabelecer a conexão (e seus parâmetros) entre o **Rx** e o **Tx**.
- **llwrite() -** responsável pela correta emissão dos dados por **Tx** para o **Rx**.
- **llread() -** responsável pela válida leitura dos dados enviados por **Rx**.
- **llclose() -** termina a ligação, dando reset aos parâmetros da serial port.

message.c

- **sendAndWaitMessage() -** escreve uma trama para a serial port esperando por uma válida resposta que é validada através de uma máquina de estados (implementada com recurso a uma virtual table). O envio da trama está protegido por um temporizador, com um número fixo de tentativas definido no estabelecimento da conexão.
- **sendInformationFrame() -** cria e envia uma trama de informação, na forma de um array de bytes, com os dados passados por parâmetro. Calcula o **BCC2** e faz o **byte stuffing** dos dados para evitar problemas de interpretação ao enviar a informação.
- **readMessageWithResponse()** -Lê uma trama até que a máquina de estados chegue a um estado final, podendo ser enviada uma resposta positiva ou negativa.

set_st.c

- **set_lookup_transitions()** - recebendo um estado da state_machine e um código de retorno do decorrer da receção e processamento de um byte em função desse estado, retorna-se o estado para qual devemos ir a seguir.
- **set_entry_state(), set_flag_state() ...** - definem a forma como reagir a um byte lido no estado apropriado tendo em conta o role do programa e o estado atual, mais propriamente se já estabelecemos conexão ou não.

utils.c

- **stuffData() e unstuffData()** - Adiciona/Remove **ESC** bytes aos dados passados como argumento e que podem ser confundidos com bytes importantes: **ESC** e **FLAG**.
- **BCC2() -** calcula o **BCC** de um dado array de dados aplicando operações XOR a cada linha do array sendo o valor resultante comparado com o **BCC2** recebido.

application_layer.c

- **applicationLayer()** - usando as funções oferecidas pelo **LL**, abre a conexão e envia/recebe os dados dos ficheiros escolhidos, acabando por fechar a ligação no fim.

app_layer_utils.c

- **makeCtrlPacket()**, **parseCtrlPacket() -** Processam tramas de controlo que são enviadas para demarcar o início e fim dos dados, contendo o nome e tamanho do ficheiro a enviar.
- **makeDataPacket(), parseDataPacket() -** Servem para encapsulamento do **AL** das tramas de informação enviadas, indicando a quantidade de bytes que contém.
- **sendFile() , rcvFile() -** Funções principais do **AL** e que asseguram a leitura de ficheiro e envio desses dados (**Tx**) e a recepção e escrita desses dados em ficheiro (**Rx**) com as funções **llread()** e **llwrite()**. Estas funções invocam as 4 funções de packets anteriores.

Casos de uso principais

A aplicação deve ser compilada usando o *Makefile* incluído e para executar o programa deve-se usar os seguintes argumentos:

Emissor: *./main* /dev/tty<porta> *tx <ficheiro>*
Recetor: *./main* /dev/tty<porta> *rx <destino>*

<porta> : Porta a ser usada na comunicação
<ficheiro> : Path do ficheiro a enviar
<destino> : Path onde guardar ficheiro a receber

Executa-se **Rx** primeiro para não haver uma trama perdida logo no início. De seguida, o **Tx** deve ser aberto e fica então estabelecida uma ligação pela porta série através da função **llopen()** (chamada por ambos os roles). Após isso, **Tx** começa a enviar através do **llwrite()** o ficheiro escolhido pelo utilizador e, por sua vez, simultaneamente, **Rx** recebe os conteúdos através da função **llread()** e vai escrevendo num ficheiro de destino. A transmissão é feita trama a trama até que todos os dados sejam transmitidos. Assim que **Tx** termina a transmissão de dados, **llclose()** é chamada por ambos os roles de modo a terminar a ligação corretamente.

Durante qualquer fase do processo acima descrito, caso **Rx** não obtenha resposta alguma durante 12s, o programa termina. De modo semelhante, o **Tx** tenta comunicar 3x com intervalos de 3s, e termina caso não obtenha resposta. De notar que estes valores são configuráveis com uso de *macros* e/ou na passagem do argumentos para o *aplicationLayer*.

Protocolo de ligação lógica

llopen()

Nesta fase é enviado um **SET**, com **temporizador** pelo **Tx** ao qual se espera receber um **UA**. O **Rx** fica portanto à espera de um **SET** ao qual responde com um **UA**. Em caso de problemas no recepcionamento do **UA** por parte do **Tx**, **Rx** poderá encontrar-se já noutra face do programa mas continuará a saber responder ao **SET** recebido.

De mencionar que a porta série é aberta e configurada nesta função. sendo também importante referir que foi configurada com o **VTIME** a 1 e o **VMIN** a 0, para que o read não seja bloqueante.

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

// Set up the connection between Tx and Rx

if (connectionParameters.role == LlRx)
{
    if (readMessageWithResponse(fd) < 0)
    {
        DEBUG_PRINT("Connection attempt to rx failed\n");
        return -1;
    }

    set_rx_ready();
}

if (connectionParameters.role == LlTx)
{
    unsigned char cmd[5] = {FLAG, ADDR_ER, SET, BCC(ADDR_ER, SET), FLAG};

    if (sendAndWaitMessage(fd, cmd, 5) < 0)
    {
        DEBUG_PRINT("Connection attempt to tx failed\n");
        return -1;
    }

    set_tx_ready();
}
```

llwrite()

Nesta função é enviado uma trama de informação em que o campo de dados é recebido por argumento. A função *sendInformationFrame*, como referido acima, é responsável por criar e enviar a trama de informação de forma correta. Caso tenha **ocorrido um REJ** (return > 0), então tenta-se enviar de novo a trama um certo número de vezes. Em caso de a trama ter sido transmitida de forma correta, e sido recebido o **RR correto,** então retorna-se corretamente da função. Caso contrário, considera-se que a ligação está instável, não sendo possível uma transmissão sólida.

```c
int llwrite(const unsigned char *buf, int bufSize)
{
    static int w_packet = 0;
    int numTries = 0, ret;

    do
    {
        numTries++;
        ret = sendInformationFrame(fd, buf, bufSize, w_packet);

        DEBUG_PRINT("\n-------------------------------------|%d|%d\n", ret, numTries);
        if (ret == 0)
        {
            w_packet = (w_packet + 1) % 2;
            DEBUG_PRINT("\nW_PACKET ::: %d\n", w_packet);
            return 0;
        }
        else if (ret < 0)
        {
            DEBUG_PRINT("Already waited 12s. Stop trying to send msg.\n");
            break;
        }

    } while (numTries < connectionParameters_cpy.nRetransmissions);

    DEBUG_PRINT("llwrite returned -1\n");
    return -1;
}
```

llread()

Nesta função é lido um pacote de dados. A função *readMessageWithResponse* é chamada para tal. Existem 3 casos: 1) a informação lida corresponde a um pacote anterior e a função *readMessageWithResponse* envia um **RR(i)** com **i** igual ao pacote que esperava receber e continua a sua leitura, de forma a que o **llread()** só retorna quando tiver o pacote na ordem correta. 2) Se a informação lida estiver errada, o **BCC2** alerta-nos disso e envia-se um **REJ** do nº de pacote correspondente. 3) No caso de a informação estiver correta envia-se um **RR** para o pacote seguinte**.**

```c
int llread(unsigned char *packet)
{
    static int r_packet = 0;
    set_rcv_packet_nr(r_packet);
    int r = readMessageWithResponse(fd);

    if (r > 0) // if rcv data is correct
    {
        r_packet = (r_packet + 1) % 2;
        DEBUG_PRINT("readMessage > 0 with r_packet= %d\n", r_packet);

        unsigned char cmd[5] = {FLAG, ADDR_ER, RR(r_packet), BCC(ADDR_ER, RR(r_packet)), FLAG};
        write(fd, cmd, 5);

        DEBUG_PRINT("Returning data from packet\n");
        return get_data(packet);
    }
    else if (r < 0) // BCC2 is not ok
    {
        unsigned char cmd[5] = {FLAG, ADDR_ER, REJ(r_packet), BCC(ADDR_ER, REJ(r_packet)), FLAG};
        write(fd, cmd, 5);
        DEBUG_PRINT("REJ was sent\n");
        return 0;
    }

    DEBUG_PRINT("LL read return 0\n");
    return -1;
}
```

llclose()

Esta função tem o propósito de terminar a ligação entre os dois programas. Para tal, **Tx** envia um comando **DISC** que deverá receber outro comando **DISC** como resposta e ao qual responde com um **UA** final (uso de *sendAndWaitMessage* e *readMessageWithResponse*)**.** Por fim, são repostas as configurações anteriores da serial port.

Protocolo de aplicação

O **AL** deve ser simples e conceder uma interface ao utilizador, utilizando as 4 funções de interface fornecidas pelo **LL**, sendo responsável por: ler ficheiros a transmitir, escrever ficheiros de destino e construir pacotes de controlo e de dados. Para tal, após estabelecida uma ligação, as funções **sendFile()** e **rcvFile()** são chamadas consoante o role para efetivamente executar a transferência do ficheiro.

```
// Open connection between TX and RX
if (llopen(connectionParameters) < 0)
{
    printf("Connection couldn't be established.\n");
    return;
}

printf("\nA connection was established.\n");

if (connectionParameters.role == LlTx)
    if (sendFile(filename) < 0)
    {
        printf("File sending failed.\n");
        return;
    }

if (connectionParameters.role == LlRx)
    if (rcvFile(filename) < 0)
    {
        printf("File receiving failed.\n");
        return;
    }

printf("File transfer complete. Starting to close connection...\n");

if (llclose(1) < 0)
{
    printf("The closing of the connection failed.\n");
    return;
}

printf("Connection closed.\n");
```

sendFile()

Abaixo segue-se a sequência de ações executadas pela função **sendFile()** e chamadas a funções auxiliares:

- Abre o ficheiro a enviar (**al_open_tx()**);
- Cria e envia trama de controlo inicial (**makeCtrlPacket()**) na qual se encontram o nome e tamanho do ficheiro a enviar;
- Leitura e envio do ficheiro em tramas de informação (através de **llwrite()**); construídas através da função **makeDataPacket()**;
- Envia trama de controlo final e fecha ficheiro lido (**al_close_tx()**);

rcvFile()

Abaixo segue-se a sequência de ações executadas pela função **rcvFile()** e chamadas a funções auxiliares:

- Espera receber a trama de controlo inicial e fazer o seu parsing com a função **parseCtrlPacket()** de onde extrai o nome e tamanho do ficheiro a receber;
- Abre o ficheiro onde escrever os conteúdos a serem recebidos (**al_open_rx()**);
- Recebe as múltiplas tramas de informação que compõem o ficheiro através do **llread()** e faz o seu parsing com **parseDataPacket()**;
- Escreve os dados extraídos do parsing anterior no ficheiro (**writeToFile()**);
- Indica a percentagem do processo de transferência pendente calculando o número de bytes já recebidos e os que ainda faltam receber;
- Espera receber a trama de controlo final e fecha ficheiro escrito (**al_close_rx()**);

Durante a transferência do ficheiro é usado um número de série em ambas as funções **rcvFile()** e **sendFile()** de modo a assegurar o envio e recepção sequencial das tramas. Caso se detectem erros na sequência ou algumas das funções **llread()** ou **llwrite()** falhem por algum motivo alheio ao **AL**, a transferência cessa.

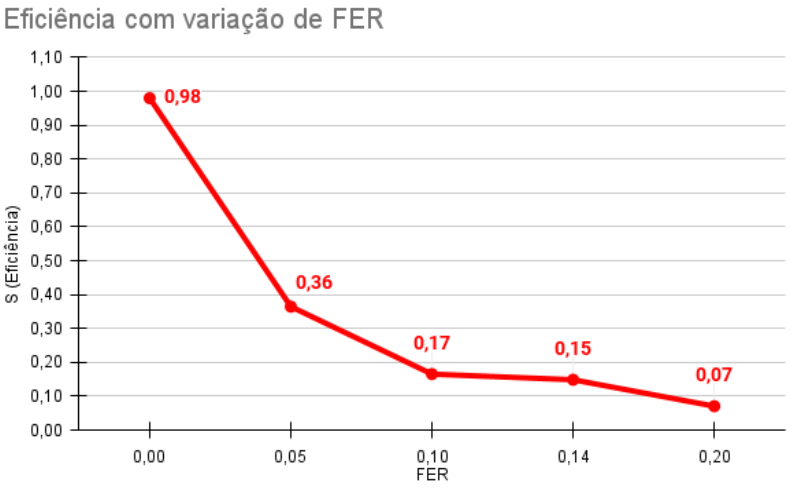
Validação e Eficiência do protocolo de ligação de dados

No sentido de avaliar a eficiência do programa desenvolvido, foram efetuados vários testes variando diversos parâmetros (variando apenas 1 e mantendo os restantes constantes em cada caso):

- Envio de vários ficheiros com diferentes tamanhos (principalmente 11 KB e 1.2MB)
- Geração de ruído no BCC1 e no BCC2 - com FER de 0, 5, 10, 15 e 20 %
- Bloqueio da ligação por cabo das porta série
- Diferentes Baudrate - 4800, 9600, 19200, 38400
- Diferentes tamanhos da trama de informação - 50, 500, 1000, 1500
- Diferentes tempos de propagação (simulados) - 2 até 8 ms, valores baixo mas que revelam a rápida perda de eficiência

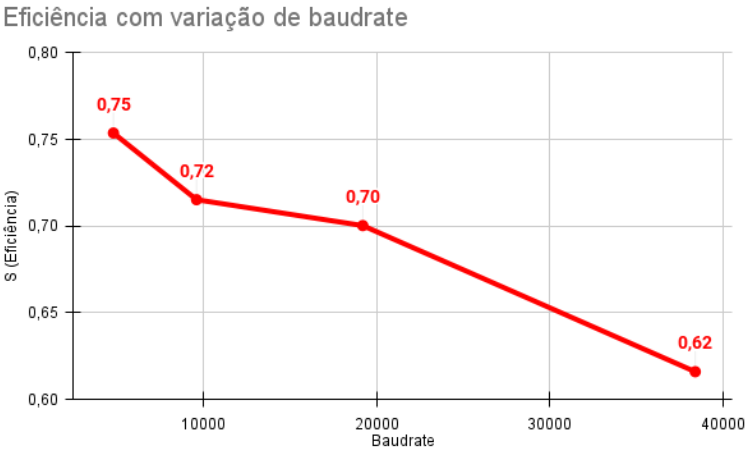
Variação do FER

O FER tem um grande impacto na eficiência do protocolo. Um erro gerado no BCC1 força a timeout e ausência de resposta do **Rx** causando uma espera de 3s para nova tentativa. Um erro no BCC2 será mais rápido de recuperar pois apenas causam o reenvio imediato da trama. Testou-se então o programa com erros tanto no BCC1 como no BCC2, em conjunto.



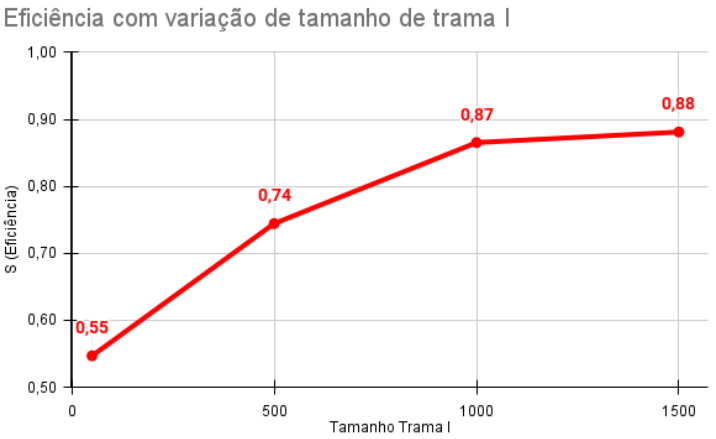
Variação do baudrate

Com este gráfico verificamos que o aumento da capacidade de ligação (baudrate) leva a uma queda não muito significativa da eficiência.



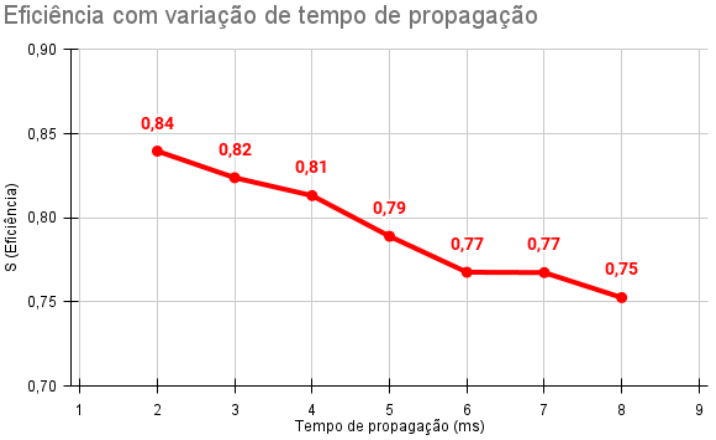
Variação do tamanho de I tramas

Quanto maior o tamanho da trama de informação maior a eficiência do programa. Isto verifica-se porque serão feitos menos envios de tramas contendo cada uma destas mais informação, levando a que timeout ocorra menos vezes.



Variação do tempo de propagação

Variando o tempo de propagação diminui-se a eficiência do programa. Isto deve-se ao facto de a aplicação passar mais tempo em idle sem enviar tramas (tempo morto) uma vez que simulamos artificialmente o delay de envio recorrendo à função usleep().



Focando agora numa visão mais teórica sobre o protocolo utilizado para controlo de erros nas tramas de informação (**Stop & Wait**), após o envio de cada trama, **Tx** aguarda resposta de confirmação (**RR**) pelo **Rx**. Caso este último detecte erros na parte dos dados da trama de informação recebida, é necessário que a trama seja novamente enviada, pelo que uma resposta negativa é retornada (**REJ**). Só após receber resposta é que **Tx** deve enviar outra trama, podendo esta ser a seguinte da sequência, em caso de **RR**, ou a mesma em caso de **REJ**. Para o Rx conseguir identificar que trama está a receber, estas encontram-se numeradas alternando valores 1 e 0 no campo de controlo.

Caso **Tx** não receba resposta, quer porque a trama não chegou a **Rx** ou a resposta deste não retornou ao emissor, **Tx** deve assegurar um mecanismo de reenvio por tentativas após 3s segundos e, por fim, após 3 tentativas, dar timeout e **Tx** termina então o programa unilateralmente. O mesmo sucede do lado do Rx que termina por timeout após não receber comunicação em 12s.

Conclusões

Síntese

Este projeto tinha como principal finalidade a implementação de um protocolo de ligação de dados entre duas máquinas através de uma porta série.

Foi de fulcral importância a identificação da necessidade de isolamento entre as duas camadas, **AL** e **LL**, a desenvolver, tendo estas, por sua vez, funcionamento e responsabilidades independentes. Em adição, a camada **LL**, por ser inferior, deve oferecer uma interface de uso à camada superior, **AL**, sendo isso assegurado através das 4 funções **llopen(), llwrite()**, **llread()** e **llclose(**).

No mesmo sentido, para além de divisão em duas camadas, o programa deve apresentar dois blocos funcionais distintos (**Tx** e **Rx**) que invocam diferentes funções conforme o role desempenhado.

O **LL** encarrega-se da comunicação efetiva pela porta série através da construção de tramas, seu envio e receção, resposta a tramas, byte stuffing/destuffing, detecção e atuação sobre erros, etc. Por sua vez, o **AL** serve de simples intermediário entre o utilizador e o **LL**, fazendo uso das 4 funções disponibilizadas pelo **LL**, permitindo ler/escrever os ficheiros enviados/recebidos conforme se atue como **Tx** ou **Rx**, respetivamente.

No sentido de avaliar a eficiência do programa desenvolvido, foram efetuados vários testes que comprovaram a eficiência e também a robustez a erros do código desenvolvido e que foram de encontro ao que seria esperado.

Reflexão

O trabalho foi concluído com sucesso tendo os objetivos principais que nos foram preconizados sido cumpridos tal como, julgamos nós, todos os objetivos secundários.

Este projeto laboratorial muniu-nos de conhecimentos práticos no que concernem protocolos de aplicação e ligação de dados, subjacente estrutura e mecanismos no envio/ receção de dados, bem como incidir sobre o respetivo processo de encapsulamento e isolamento de camadas. Em adição, podemos igualmente afirmar que foram consolidados conceitos teóricos previamente lecionados nas aulas teóricas, culminando no enriquecimento do nosso conhecimento no domínio das redes de comunicação.

Anexo - Código

link_layer.c

```
#include "link_layer.h"
#include "time.h"
extern int (*set_state[])(unsigned char c);
#define BILLION 1000000000.0
struct timespec begin;

int get_baud(int baud);

int llopen(LinkLayer connectionParameters)
{
    clock_gettime(CLOCK_REALTIME, &begin);
    signal(SIGALRM, alarm_handler);

    // 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(connectionParameters.serialPort, O_RDWR | O_NOCTTY);

    connectionParameters_cpy = connectionParameters;

    set_nr_retransmissions(connectionParameters_cpy.nRetransmissions);
    set_nr_timeout(connectionParameters_cpy.timeout);

    if (fd < 0)
    {
        perror(connectionParameters.serialPort);
        exit(-1);
    }

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

    struct termios newtio;

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

     newtio.c_cflag = get_baud(connectionParameters.baudRate) | CS8 | CLOCAL | CREAD; // Note the
concvertion to the termios Baudrate defines
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 1;
    newtio.c_cc[VMIN] = 0;
```

```
    tcflush(fd, TCIOFLUSH);

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

    // Set up the connection between Tx and Rx

    if (connectionParameters.role == LlRx)
    {
        if (readMessageWithResponse(fd) < 0)
        {
            DEBUG_PRINT("Connection attempt to rx failed\n");
            return -1;
        }

        set_rx_ready();
    }

    if (connectionParameters.role == LlTx)
    {
        unsigned char cmd[5] = {FLAG, ADDR_ER, SET, BCC(ADDR_ER, SET), FLAG};

        if (sendAndWaitMessage(fd, cmd, 5) < 0)
        {
            DEBUG_PRINT("Connection attempt to tx failed\n");
            return -1;
        }

        set_tx_ready();
    }

    return 1;
}

int llwrite(const unsigned char *buf, int bufSize)
{
    static int w_packet = 0;
    int numTries = 0, ret;

    do
    {
        numTries++;
        ret = sendInformationFrame(fd, buf, bufSize, w_packet);

        DEBUG_PRINT("\n--------------------------------------|%d|%d\n", ret, numTries);
        if (ret == 0)
        {
            w_packet = (w_packet + 1) % 2;
            DEBUG_PRINT("\nW_PACKET ::: %d\n", w_packet);
            return 0;
```

```
            }
            else if (ret < 0)
            {
                DEBUG_PRINT("Already waited 12s. Stop trying to send msg.\n");
                break;
            }

        } while (numTries < connectionParameters_cpy.nRetransmissions);

        DEBUG_PRINT("llwrite returned -1\n");
        return -1;
    }

    int llread(unsigned char *packet)
    {
        static int r_packet = 0;
        set_rcv_packet_nr(r_packet);
        int r = readMessageWithResponse(fd);

        if (r > 0) // if rcv occured in right order
        {
            r_packet = (r_packet + 1) % 2;
            DEBUG_PRINT("readMessage > 0 with r_packet= %d\n", r_packet);

            unsigned char cmd[5] = {FLAG, ADDR_ER, RR(r_packet), BCC(ADDR_ER, RR(r_packet)), FLAG};
            write(fd, cmd, 5);

            DEBUG_PRINT("Returning data from packet\n");
            return get_data(packet);
        }
        else if (r < 0) // Not the right packet, send REJ
        {
               unsigned char cmd[5] = {FLAG, ADDR_ER, REJ(r_packet), BCC(ADDR_ER, REJ(r_packet)),
FLAG};
            write(fd, cmd, 5);
            DEBUG_PRINT("REJ was sent\n");
            return 0;
        }

        DEBUG_PRINT("LL read return 0\n");
        return -1;
    }

    int llclose(int showStatistics)
    {
        // Transmitter closing
        if (connectionParameters_cpy.role == LlTx)
        {
            DEBUG_PRINT("TX Closing...\n");
            unsigned char cmd[5] = {FLAG, ADDR_ER, DISC, BCC(ADDR_ER, DISC), FLAG};

            if (sendAndWaitMessage(fd, cmd, 5) < 0)
            {
                DEBUG_PRINT("llclose tx await for response failed");
```

```
            return -1;
        }

        DEBUG_PRINT("Tx sending final UA to close\n");
        unsigned char ua_cmd[5] = {FLAG, ADDR_ER, UA, BCC(ADDR_ER, UA), FLAG};
        write(fd, ua_cmd, 5);
        sleep(1);
    }

    // Receiver closing
    if (connectionParameters_cpy.role == LlRx)
    {
        DEBUG_PRINT("RX Closing...\n");
        int r = readMessageWithResponse(fd);

        if (r < 0)
        {
            DEBUG_PRINT("Rx llclose read message failed\n");
            return -1;
        }
    }

    if (showStatistics)
    {
        struct timespec end;
        clock_gettime(CLOCK_REALTIME, &end);

        double time_spent = (end.tv_sec - begin.tv_sec) +
                            (end.tv_nsec - begin.tv_nsec) / BILLION;

        printf("TIME TO FINISH: %f\n", time_spent);
    }

    // Recover old serial port settings
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        perror("tcsetattr");
        exit(-1);
    }

    DEBUG_PRINT("closing file descriptor\n");
    close(fd);
    return 0;
}

int get_baud(int baud)
{
    switch (baud)
    {
    case 9600:
        return B9600;
    case 19200:
        return B19200;
    case 38400:
```

```
            return B38400;
        case 57600:
            return B57600;
        case 115200:
            return B115200;
        case 230400:
            return B230400;
        case 460800:
            return B460800;
        case 500000:
            return B500000;
        case 576000:
            return B576000;
        case 921600:
            return B921600;
        case 1000000:
            return B1000000;
        case 1152000:
            return B1152000;
        case 1500000:
            return B1500000;
        case 2000000:
            return B2000000;
        case 2500000:
            return B2500000;
        case 3000000:
            return B3000000;
        default:
            return B0;
        }
    }
```

message.c

```
    #include "message.h"

    extern int (*set_state[])(unsigned char c);
    static int (*set_state_fun)(unsigned char c);

    int alarm_flag = 0;
    static int rcv_paket_nr = 0;

    static int nRtr;
    static int timeout;

    void alarm_handler()
    {
        alarm_flag = 1;
    }
```

```
void set_rcv_packet_nr(int rcv_paket)
{
   rcv_paket_nr = rcv_paket;
}

int sendAndWaitMessage(int fd, unsigned char *msg, int messageSize)
{
   int numTries = 0;
   int ret;

   (void)signal(SIGALRM, alarm_handler);

   do
   {
       alarm_flag = 0;

       ret = write(fd, msg, messageSize);
       printf("\nAttempt to send messsage n°%d\n", ++numTries);
       SET_ALARM_TIME(timeout)

       if (ret < messageSize)
       {
           int restToWrite = messageSize - ret;

           do
           {
               int rt = write(fd, msg + ret, restToWrite);
               ret += rt;
               if (rt == restToWrite)
               {
                   break;
               }
               restToWrite -= rt;
           } while (!alarm_flag);
       }

       unsigned char buf = 0;
       unsigned char bytes;

       while (!alarm_flag)
       {
           bytes = read(fd, &buf, 1);
           if (bytes == 0)
               continue;

           enum set_state_codes st = get_set_state();
           set_state_fun = set_state[st];
           enum set_ret_codes rt = set_state_fun(buf);

           DEBUG_PRINT("rt:%d\n", rt);
           set_set_state(set_lookup_transitions(st, rt));

           DEBUG_PRINT("state:%d:%d:%c\n", get_set_state(), buf, buf);
```

```
            if (get_set_state() == EXIT_SET_STATE)
            {
                printf("UA/RR/REJ/DISC RECIEVED\n");
                break;
            }
        }

    } while (numTries < nRtr && get_set_state() != EXIT_SET_STATE);

    TURN_OFF_ALARM

    if (get_set_state() != EXIT_SET_STATE)
    {
        printf("\nFailed to get a response.\n");
        ret = -1;
    }

    DEBUG_PRINT("Setting State to EntryState\n");
    set_set_state(ENTRY_SET_STATE);
    return ret;
}

int sendInformationFrame(int fd, const unsigned char *data, int dataSize, int packet)
{

    unsigned char cmd[dataSize + 6];

    cmd[0] = FLAG;
    cmd[1] = ADDR_ER;
    cmd[2] = CTRL_S(packet);
    cmd[3] = BCC(cmd[1], cmd[2]);

    memcpy(cmd + 4, data, dataSize);

    unsigned char bcc2 = BCC2(data, dataSize);

    cmd[dataSize + 4] = bcc2;
    cmd[dataSize + 5] = FLAG;

    int n_mis_flags = countProblematicFlags(cmd, dataSize + 6);

    unsigned char stuffed_cmd[dataSize + 6 + n_mis_flags];

    stuffData(cmd, dataSize + 6, stuffed_cmd, dataSize + 6 + n_mis_flags);

    int ret = sendAndWaitMessage(fd, stuffed_cmd, dataSize + 6 + n_mis_flags);

    unsigned char c = get_control();
    if (ret > 0 && ((packet == 0 && c == RR(1)) || (packet == 1 && c == RR(0))))
    {
        DEBUG_PRINT("Everything ok");
        return 0;
    }
    if (ret < 0)
```

```
        {
            DEBUG_PRINT("Number of restranmissions was exceeded");
            return -1;
        }

        DEBUG_PRINT(" REJ WAS RECEIVED");
        return 1;
    }

    int readMessageWithResponse(int fd)
    {
        unsigned char buf = 0;
        unsigned char bytes;

        SET_ALARM_TIME(MAX_IDLE_TIME)
        alarm_flag = 0;

        while (!alarm_flag)
        {
            bytes = read(fd, &buf, 1);
            if (bytes == 0)
            {
                DEBUG_PRINT("Nothing was read\n");
                continue;
            }

            enum set_state_codes st = get_set_state();
            set_state_fun = set_state[st];
            enum set_ret_codes rt = set_state_fun(buf);

            if (rt == BCC2_NOT_OK)
            {
                DEBUG_PRINT("BCC2 Not ok...\n");
            }

            set_set_state(set_lookup_transitions(st, rt));

            if (get_set_state() != 6)
            {
                DEBUG_PRINT("rt:%d | state:%d:%d:%c\n", rt, get_set_state(), buf, buf);
            }
            else
            {
                DEBUG_PRINT(" message>|%c|\n", buf);
            }

            if (get_set_state() == EXIT_SET_STATE)
            {
                DEBUG_PRINT("SET RECIEVED\n");
                set_set_state(ENTRY_SET_STATE);
                TURN_OFF_ALARM
                if (get_control() == SET)
                {
                    unsigned char cmd[5] = {FLAG, ADDR_ER, UA, BCC(ADDR_ER, UA), FLAG};
```

```
                    write(fd, cmd, 5);
                    DEBUG_PRINT("Sent a SET");
                }
                else if (get_control() == CTRL_S(0) || get_control() == CTRL_S(1))
                {
                    if (get_control() != CTRL_S(rcv_paket_nr))
                        { // sends receiver ready when packet is the same as the previous packet
recieved
                            unsigned char cmd[5] = {FLAG, ADDR_ER, RR(rcv_paket_nr), BCC(ADDR_ER,
RR(rcv_paket_nr)), FLAG};
                        write(fd, cmd, 5);

                        continue;
                    }
                    return get_data_size();
                }
                else if (get_control() == DISC)
                {
                    DEBUG_PRINT("DISC recieved\n");
                    unsigned char cmd[5] = {FLAG, ADDR_ER, DISC, BCC(ADDR_ER, DISC), FLAG};
                    if (sendAndWaitMessage(fd, cmd, 5) < 0)
                    {
                        DEBUG_PRINT("Final UA from Tx wasnt received correctly\n");
                        return -1;
                    }
                }
                return 0;
            }
        }

        DEBUG_PRINT("Returning -1\n");
        TURN_OFF_ALARM
        return 0;
    }

    void set_nr_retransmissions(int nr_retransmissions_r)
    {
        nRtr = nr_retransmissions_r;
    }
    void set_nr_timeout(int timeout_r)
    {
        timeout = timeout_r;
    }
```

set_st.c

```
    #include "set_st.h"

    unsigned char msg[5] = {0};
    static unsigned char sdata[BUF_SIZE];
    static int data_size = 0;
```

```
        static enum set_state_codes set_cur_state = ENTRY_SET_STATE;

        static int tx_ready_to_send = 0;
        static int rx_RR = 0;

        int (*set_state[])(unsigned char c) = {
            set_entry_state, set_flag_state, set_a_state, set_c_state, set_bcc_state, set_stop_state,
set_data_state};

        int set_entry_state(unsigned char c)
        {
           enum set_ret_codes ret = OTHER_RCV;
           switch (c)
           {
           case FLAG:
               ret = FLAG_RCV;
               msg[0] = c;
               break;
           default:
               break;
           }
           return ret;
        }

        int set_flag_state(unsigned char c)
        {
           enum set_ret_codes ret = OTHER_RCV;
           switch (c)
           {
           case ADDR_ER:
               ret = A_RCV;
               msg[1] = c;
               break;
           case FLAG:
               ret = FLAG_RCV;
               break;
           default:
               break;
           }
           return ret;
        }

        int set_a_state(unsigned char c)
        {
           enum set_ret_codes ret = OTHER_RCV;

           switch (c)
           {
           case SET: // A UA VAI SER DIFERENTE
           case UA:
           case DISC:
               ret = C_RCV;
               msg[2] = c;
               break;
```

```
        case RR(0):
        case RR(1):
            if (tx_ready_to_send) // just in case tx has not yet made pass the "handshake" fase
            {
                ret = C_RCV;
                msg[2] = c;
            }
            break;
        case CTRL_S(1):
        case CTRL_S(0):
            if (rx_RR)
            {
                ret = C_RCV;
                msg[2] = c;
            }
            break;
        case FLAG:
            ret = FLAG_RCV;
            break;
        default:
            break;
        }
        return ret;
    }

    int set_c_state(unsigned char c)
    {
        enum set_ret_codes ret = OTHER_RCV;
        switch (c)
        {
        case FLAG:
            ret = FLAG_RCV;
            break;
        default:
            break;
        }

        unsigned char validation = BCC(msg[1], msg[2]);

        if (memcmp(&c, &validation, 1) == 0)
        {
            ret = BCC_OK;
            msg[3] = c;

            if (msg[2] == CTRL_S(1) || msg[2] == CTRL_S(0))
            {
                ret = INF_FRAME;
                DEBUG_PRINT("data_size reset OF 0\n");
                data_size = 0;
            }
        }

        return ret;
    }
```

```c
int set_bcc_state(unsigned char c)
{
   enum set_ret_codes ret = OTHER_RCV;
   switch (c)
   {
   case FLAG:
       ret = FLAG_RCV;
       msg[4] = c;
       break;
   default:
       break;
   }
   return ret;
}

int set_stop_state(unsigned char c)
{
   UNUSED(c);
   return OTHER_RCV;
}

int set_data_state(unsigned char c)
{
   enum set_ret_codes ret = OTHER_RCV;

   if (c == FLAG)
   {
       ret = FLAG_RCV;
       unsigned char usData[data_size];
       int usSize = unstuffData(sdata, data_size, usData);
       unsigned char bcc2 = BCC2(usData, usSize - 1);

       msg[3] = usData[usSize - 1]; // BCC2

       if (memcmp(&bcc2, (msg + 3), 1))
       {
           ret = BCC2_NOT_OK;
       }
   }

   if (ret == OTHER_RCV)
   {
       sdata[data_size++] = c;
   }

   msg[4] = c;
   return ret;
}

enum set_state_codes set_lookup_transitions(int cur_state, int rc)
{
   SET_ST_TRANS state_transitions[] = {
       {start, FLAG_RCV, flag_rcv},
```

```
            {start, OTHER_RCV, start},
            {flag_rcv, FLAG_RCV, flag_rcv},
            {flag_rcv, A_RCV, a_rcv},
            {flag_rcv, OTHER_RCV, start},
            {a_rcv, FLAG_RCV, flag_rcv},
            {a_rcv, OTHER_RCV, start},
            {a_rcv, C_RCV, c_rcv},
            {c_rcv, FLAG_RCV, flag_rcv},
            {c_rcv, OTHER_RCV, start},
            {c_rcv, BCC_OK, bcc_ok},
            {bcc_ok, FLAG_RCV, stop},
            {bcc_ok, OTHER_RCV, start},
            {stop, OTHER_RCV, stop},
            {c_rcv, INF_FRAME, data},
            {data, OTHER_RCV, data},
            {data, FLAG_RCV, stop},
            {data, BCC2_NOT_OK, flag_rcv}};

        for (int i = 0; i < 18; i++)
        {
                            if    (state_transitions[i].src_state    ==    (unsigned)cur_state    &&
state_transitions[i].ret_code == (unsigned)rc)
            {
                return state_transitions[i].dst_state;
            }
        }
        return cur_state;
    }

    enum set_state_codes get_set_state()
    {
        return set_cur_state;
    }

    void set_set_state(enum set_state_codes st)
    {
        set_cur_state = st;
    }

    unsigned char get_control()
    {
        return msg[2];
    }

    int get_data_size()
    {
        return data_size;
    }
    int get_data(unsigned char dt[])
    {
        unsigned char usData[data_size];
        int usSize = unstuffData(sdata, data_size, usData);
        memcpy(dt, usData, usSize - 1);
        return usSize - 1;
```

```
}

void set_tx_ready()
{
   // TX can recieve RR
   tx_ready_to_send = 1;
}
void set_rx_ready()
{
   rx_RR = 1;
}

int is_tx()
{
   return tx_ready_to_send;
}

int is_rx()
{
   return rx_RR;
}
```

utils.c

```
#include "utils.h"

int countProblematicFlags(unsigned char *data, int dataSize)
{
   int n_misleading_flags = 0;
   for (int i = 0; i < dataSize; i++)
   {
       if (data[i] == FLAG || data[i] == ESC)
       {
           n_misleading_flags++;
       }
   }
   return n_misleading_flags - 2;
}

int stuffData(unsigned char *data, int dataSize, unsigned char *stData, int stSize)
{

   stData[0] = FLAG;
   stData[stSize - 1] = FLAG;
   for (int i = 1, j = 1; i < dataSize - 1; i++, j++)
   {
       if (data[i] == FLAG)
       {
           stData[j++] = ESC;
           stData[j] = XOR_FLAG;
       }
```

```
        else if (data[i] == ESC)
        {
            stData[j++] = ESC;
            stData[j] = XOR_ESC;
        }
        else
        {
            stData[j] = data[i];
        }
    }

    return 0;
}

unsigned char BCC2(const unsigned char *data, int dataSize)
{
    unsigned char bcc = data[0];

    for (int i = 1; i < dataSize; i++)
        bcc ^= data[i];

    return bcc;
}

int unstuffData(unsigned char *data, int dataSize, unsigned char *stData)
{
    int j = 0;
    for (int i = 0; i < dataSize; i++, j++)
    {
        if (data[i] == ESC)
        {
            i++;
            stData[j] = data[i] == XOR_FLAG ? FLAG : ESC;
        }
        else
        {
            stData[j] = data[i];
        }
    }
    return j;
}
```

application_layer.c

```c
#include "application_layer.h"

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{
    // Set intial values
    LinkLayerRole LRole = (!strcmp(role, "tx")) ? LlTx : LlRx;
    LinkLayer connectionParameters = {"", LRole, baudRate, nTries, timeout};
    strcpy(connectionParameters.serialPort, serialPort);

    printf("\n\nStart of program.\nAttempting to create connection...\n");

    // Open connection between TX and RX
    if (llopen(connectionParameters) < 0)
    {
        printf("Connection couldn't be established.\n");
        return;
    }

    printf("\nA connection was established.\n");

    if (connectionParameters.role == LlTx)
        if (sendFile(filename) < 0)
        {
            printf("File sending failed.\n");
            return;
        }

    if (connectionParameters.role == LlRx)
        if (rcvFile(filename) < 0)
        {
            printf("File receiving failed.\n");
            return;
        }

    printf("File transfer complete. Starting to close connection...\n");
    if (llclose(1) < 0)
    {
        printf("The closing of the connection failed.\n");
        return;
    }
    else
    {
        printf("Connection closed.\n");
    }

    printf("End of program.\n");
}
```

app_layer_utils.c

```c
#include "app_layer_utils.h"

FILE *fp_tx = NULL;
FILE *fp_rx = NULL;

void al_close_rx()
{
   fclose(fp_rx);
}

void al_close_tx()
{
   fclose(fp_tx);
}

int al_open_tx(const char *filename_tx)
{
   fp_tx = fopen(filename_tx, "r");
   fseek(fp_tx, 0, SEEK_END);
   int file_size = ftell(fp_tx);
   fseek(fp_tx, 0, SEEK_SET);
   return file_size;
}

void al_open_rx(const char *filename_rx)
{
   fp_rx = fopen(filename_rx, "w");
}

int readFromFile(unsigned char *message_send, unsigned msg_size)
{
   return fread(message_send, 1, msg_size, fp_tx);
}

int writeToFile(unsigned char *message_rcv, unsigned msg_size)
{
   return fwrite(message_rcv, msg_size, 1, fp_rx);
}

int makeCtrlPacket(unsigned char ctrlByte, unsigned char *packet, const char *filename, int filesize)
{
```

```
    packet[0] = ctrlByte;
    packet[1] = TYPE_FILESIZE;

    int length = 0;
    int currentFileSize = filesize;

    // cicle to separate file size (v1) in bytes
    while (currentFileSize > 0)
    {
        int rest = currentFileSize % 256;
        int div = currentFileSize / 256;
        length++;

        // shifts all bytes to the right, to make space for the new byte
        for (unsigned int i = 2 + length; i > 3; i--)
            packet[i] = packet[i - 1];

        packet[3] = (unsigned char)rest;
        currentFileSize = div;
    }

    packet[2] = (unsigned char)length;
    packet[3 + length] = TYPE_FILENAME;

    int fileNameStart = 5 + length; // beginning of v2

    packet[4 + length] = (unsigned char)(strlen(filename) + 1); // adds file name length (including '\0)

    for (unsigned int j = 0; j < (strlen(filename) + 1); j++)
        packet[fileNameStart + j] = filename[j]; // strlen(fileName) + 1 in order to add the '\0' char

    return 3 + length + 2 + strlen(filename) + 1; // total length of the packet
}

int parseCtrlPacket(unsigned char *packetBuffer, int *fileSize, char *fileName)
{

    if (packetBuffer[0] != CTRL_START && packetBuffer[0] != CTRL_END)
    {
        printf("Packet being parsed doesn't correspond to Command packet.\n");
        return -1;
    }
```

```
    int length;

    if (packetBuffer[1] == TYPE_FILESIZE)
    {

        *fileSize = 0;
        length = (int)packetBuffer[2];
        for (int i = 0; i < length; i++)
            *fileSize = *fileSize * 256 + (int)packetBuffer[3 + i];
    }
    else
    {
        printf("Error during command packet filesize parsing.\n");
        return -1;
    }

    int fileNameStart = 5 + length;

    if (packetBuffer[fileNameStart - 2] == TYPE_FILENAME)
    {
        length = (int)packetBuffer[fileNameStart - 1];
        for (int i = 0; i < length; i++)
            fileName[i] = packetBuffer[fileNameStart + i];
    }
    else
    {
        printf("Error during command packet filename parsing.\n");
        return -1;
    }

    return 0;
}

int makeDataPacket(unsigned char *packet, int seqNum, unsigned char *data, int dataLen)
{

    int l1 = dataLen % 256;
    int l2 = dataLen / 256;

    packet[0] = CTRL_DATA;
    packet[1] = seqNum;
```

```
    packet[2] = l2;
    packet[3] = l1;

    // actual data packets
    for (int i = 0; i < dataLen; i++)
        packet[i + 4] = data[i];

    return dataLen + 4;
}

int parseDataPacket(unsigned char *packet, unsigned char *data)
{

    if (packet[0] != CTRL_DATA)
        return -1;

    int l1 = packet[3], l2 = packet[2];
    int data_size = 256 * l2 + l1;

    // actual data packets
    for (int i = 0; i < data_size; i++)
        data[i] = packet[i + 4];

    return packet[1];
}

int sendFile(const char *filename)
{

    printf("Opening file to be sent...\n");
    int file_send_size = al_open_tx(filename);

    unsigned char message_send[MAXSIZE_FRAME];
    unsigned char data[MAXSIZE_DATA];

    printf("Sending Start Command Packet...\n");
    int packet_size = makeCtrlPacket(CTRL_START, message_send, filename, file_send_size);
    if (llwrite(message_send, packet_size) < 0)
    {
        printf("Unable to send Start Command Packet.\n");
        return -1;
    }
```

```
   printf("Sending Main File...\n");
   int seqNum = 0, num_read_bytes;
   while ((num_read_bytes = readFromFile(data, MAXSIZE_DATA)))
   {

       int msg_size = makeDataPacket(message_send, seqNum, data, num_read_bytes);

       if (llwrite(message_send, msg_size) < 0)
       {
           DEBUG_PRINT("llwrite returned < 0\n");
           return -1;
       }
       else
       {
           DEBUG_PRINT("llwrite returned > 0\n");
       }

       seqNum = (seqNum + 1) % SEQUENCE_MODULO;

#ifdef SLOW_SEND
       sleep(1);
#endif
   }

   printf("Main file was sent.\nSending End Command Packet...\n");

   // Send End Command packet
   packet_size = makeCtrlPacket(CTRL_END, message_send, filename, file_send_size);

   if (llwrite(message_send, packet_size) < 0)
   {
       printf("Unable to send END Command Packet.\n");
       return -1;
   }

   al_close_tx();
   DEBUG_PRINT("sendFile is returning 0");
   return 0;
}

int rcvFile(const char *filename)
```

```
{

    printf("Waiting for Start Command Packet...\n");

    unsigned char message_rcv[MAXSIZE_FRAME];
    unsigned char data[MAXSIZE_DATA];

    llread(message_rcv);

    if (message_rcv[0] != CTRL_START)
    {
        printf("Expected Start Control Packet but got none.\n");
        return -1;
    }

    int file_rcv_size;
    char rcv_filename[MAXSIZE_FILE_NAME];

    if (parseCtrlPacket(message_rcv, &file_rcv_size, rcv_filename) < 0)
    {
        printf("Error parsing Start Command packet.\n");
        return -1;
    }

    // create file where to write incoming contents
    printf("Receiving file info: file '%s' with size %dB\n", rcv_filename, file_rcv_size);
    al_open_rx(filename);

    printf("Starting to write to file...\n");
    int packet_size, num_bytes_rcv = 0, seqNum = 0;
    float percentageLevel;
    while (1)
    {
        packet_size = llread(message_rcv);

        if (packet_size == 0)
        {
            continue;
        }
        else if (packet_size < 0)
        {
            // It was not possible to read anything
```

```
            return -1;
        }

        if (message_rcv[0] == CTRL_END)
            break;

        else if (message_rcv[0] == CTRL_DATA)
        {
            int rcv_seqNum = parseDataPacket(message_rcv, data);

            DEBUG_PRINT("PACKET: %d\nPACKET NR : %d", packet_size, rcv_seqNum);

            if (seqNum != rcv_seqNum)
            {
                    printf("Received packet out of order!\n Expected %d and recieved %d\n", seqNum,
rcv_seqNum);
                return -1;
            }

            seqNum = (seqNum + 1) % SEQUENCE_MODULO;

            int data_size = packet_size - 4; // removing the 4 bytes for the data packet head
            writeToFile(data, data_size);    // writing data bytes to file

            num_bytes_rcv += data_size;
            percentageLevel = (float)num_bytes_rcv / (file_rcv_size)*100;
                    printf("Current progress: %d/%d (%0.1f%%)\n", num_bytes_rcv, file_rcv_size,
percentageLevel);

            if (num_bytes_rcv == file_rcv_size)
                break; // File is complete
        }

        DEBUG_PRINT("\nPACKET_NR:: %d\n", seqNum);
    }

    printf("Write to file complete.\nWaiting for End Control Packet\n");

    // Receive End Command Packet
    llread(message_rcv);
    if (message_rcv[0] != CTRL_END)
    {
        printf("Expected End Control Packet but got none.\n");
```

```
        printf("CTRL %d", message_rcv[0]);
        return -1;
    }

    int file_rcv_size_end;
    char rcv_fileName_end[MAXSIZE_FILE_NAME];

    if (parseCtrlPacket(message_rcv, &file_rcv_size_end, rcv_fileName_end) < 0)
    {
        printf("Error parsing End Command packet.\n");
        return -1;
    }

    if (strcmp(rcv_filename, rcv_fileName_end) != 0)
    {
        printf("\nFilenames of start/end control packet dont match: ");
        printf("|%s| != |%s|\n", rcv_filename, rcv_fileName_end);
        return -1;
    }

    if (file_rcv_size != file_rcv_size_end)
    {
        printf("\nFilesize of start/end control packet dont match: ");
        printf("|%d| != |%d|\n", file_rcv_size, file_rcv_size_end);
        return -1;
    }

    al_close_rx();
    return 0;
}
```