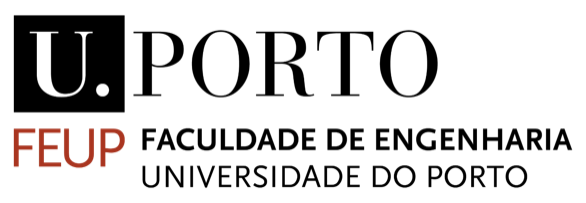
**Faculdade de Engenharia da Universidade do Porto**

**Data Link Protocol**

**Jorge Pais - up201904841**

**João Mena – up201907668**



Relatório do Trabalho Prático Laboratorial 1 no âmbito da Unidade Curricular “Redes de Computadores” da Licenciatura em Engenharia Eletrotécnica e de Computadores

Abril de 2022

**Sumário**

Este trabalho foi realizado no âmbito da Unidade Curricular “Redes de Computadores” com o objetivo de instaurar nos alunos um ponto de vista mais prático relativamente à Camada de Ligação de Dados, através do desenvolvimento de uma *API* que facilita o estabelecimento de uma ligação entre uma porta de série e a Camada de Aplicação, bem como a transferência de dados entre os dois, e a respetiva desconexão.

Os objetivos deste trabalho foram alcançados e a *API* desenvolvida funciona sem problemas, quer com introdução de erros, quer sem. Após a realização deste trabalho temos noções muito mais claras das diversas etapas que constituem um protocolo de ligação de dados e dos variados desafios que podem ser encarados na sua implementação.

**Introdução**

Neste trabalho foi-nos proposta a criação e desenvolvimento de uma *API* que representa a Camada de Ligação de Dados entre uma Camada Física, neste caso a porta de série, e uma Camada de Aplicação. A *API* deverá ter como funcionalidades a abertura da ligação, fecho da ligação, transmissão de dados e receção de dados, através das funções *llopen()*, *llclose()*, *llwrite()* e *llread()*, respetivamente.

Ao longo deste relatório será mostrado em detalhe o processo de elaboração do trabalho, desde as funções utilizadas, aos testes desenvolvidos para garantir o bom funcionamento do código e as conclusões retiradas desta experiência, de acordo com a seguinte estrutura:

1. Arquitetura – blocos funcionais e interfaces
2. Estrutura do Código – *APIs*, principais funções, estruturas de dados e a sua relação com a arquitetura
3. Casos de uso principais – identificação e sequência de chamada de funções
4. Protocolo de ligação lógica – principais elementos funcionais e estratégia de implementação
5. Validação – testes efetuados e respetivos resultados
6. Elementos de valorização – identificação destes elementos e descrição da sua implementação
7. Conclusões – síntese da informação e reflexão final

**1. Arquitetura**

O sistema que foi trabalhado neste projeto está dividido em 3 camadas: a camada física, constituída pela porta série em si e as *drivers* do sistema operativo para comunicar através desta porta, a camada de aplicação, que implementa alguma funcionalidade de comunicação, e a camada de ligação de dados, que estabelece a ligação entre as outras duas camadas, criando abstrações que permitem a programação de uma aplicação utilizando porta série sem ter que recorrer a funções de baixo nível.

A *API* do protocolo da camada de ligação de dados é composta por 4 funções principais, suportadas por um conjunto de bibliotecas e funções auxiliares implementadas por nós, e que interagem com a camada de aplicação através de 2 modos de funcionamento independentes: enquanto transmissor, ou enquanto recetor.

**2. Estrutura do Código**

O código consiste em 4 ficheiros de código fonte .c e os seus respetivos ficheiros *header*, bem como um ficheiro *header* exclusivo para a declaração de macros, o *definitions.h*. Os 4 ficheiros principais são compostos pelo ficheiro base *linklayer.c*, um ficheiro para cada modo de funcionamento, *transmitter.c* e *receiver.c*, e um ficheiro que engloba um conjunto de funções utilizadas frequentemente, *utils.c*. Para além destes ficheiros, o código ainda consta de um *makefile* para agilizar o processo de compilação.

**2.1. linklayer.c**

Este ficheiro contém as funções *llopen()* e *llclose()*, responsáveis pelo estabelecimento e terminação da ligação.

**llopen** – admite os parâmetro de conexão, guarda o *role* pretendido na variável *serialRole*, e mediante o mesmo, vai chamar a função *receiver\_llopen()* ou *transmitter\_llopen()*, definidas nos ficheiros receiver.c (2.4) e transmitter.c (2.5), respetivamente.

**llclose** – mediante o valor atribuído a *serialRole* na função *llopen()*, chama a função *receiver\_llclose()* ou  *transmitter\_llclose(),* definidas nos ficheiros receiver.c (2.4) e transmitter.c (2.5), respetivamente.

**2.2. utils.c**

Este ficheiro contém as seguintes funções de utilização frequente:

**configureSerialterminal** – configura a ligação da porta de série de acordo com os parâmetros da ligação.

**closeSerialterminal** – termina a ligação da porta de série.

**checkHeader** – implementação de uma máquina de estados que procura ler o cabeçalho de uma trama.

**readControlField** – procura e retorna o campo de controlo de uma trama de supervisão.

**generateBCC** – calcula um *Block Check Character* para um determinado vetor de dados.

**checkParameters** – verifica e copia os parâmetros de ligação para uma estrutura do tipo *linkLayer* alocada em memória.

**writeEventToFile** – função usada para enviar *logs* para o ficheiro onde é feito o *event log.*

**convertBaudRate** – converte o *baud rate* do valor definido nos parâmetros de ligação para uma variável compatível com as funções *termios*. É também verificado se o *baud rate* desejado é possível utilizar na máquina em questão.

**2.3. definitions.h**

Neste ficheiro são declaradas as seguintes macros:

**FLAG** e **ESC** – formatos de trama.

**A\_tx** e **A\_rx** – campos de endereço.

**C\_SET**, **C\_DISC**, **C\_UA**, **C\_RR**, **C\_REJ**, **C** – campos de controlo.

**SU\_SEQ** - número de sequência extraído do campo de controlo.

**I\_SEQ** – número de sequência do campo de informação.

**DEBUG\_PRINT** – usado para mensagens de *debug*.

**2.4. transmitter.c**

Este ficheiro contém as seguintes funções usadas no modo de funcionamento de transmissão de dados:

**transmitter\_llopen** – ramo da função *llopen* do lado do transmissor. Implementa o protocolo de estabelecimento de ligação: envia comando *SET* e aguarda receção do comando *UA*, com uso de *timeout*.

**prepareInfoFrame** – recolhe dados de um *buffer* e prepara uma trama de informação.

**llwrite** – transmite os dados presentes num *buffer*, lê o campo de controlo da resposta e reenvia os dados caso seja necessário.

**transmitter\_llclose** – ramo da função *llclose* do lado do transmissor. Implementa o protocolo de término de ligação: envia um comando *DISC*, espera a receção de um comando *DISC* de resposta com uso de *timeout*, e aquando da sua receção, envia um comando *UA* e termina a ligação.

**byteStuffing** – realiza uma operação de *stuffing* a um determinado vetor de dados.

**timeOut** – função auxiliar de *timeout* para gestão dos sinais *SIGALRM*.

**2.5. receiver.c**

Este ficheiro contém as seguintes funções usadas no modo de funcionamento de receção de dados:

**receiver\_llopen** – ramo da função *llopen* do lado do recetor. Espera receber o comando *SET* enviado pelo transmissor e aquando da sua receção, envia um comando *UA*.

**llread** – recebe os dados presentes num pacote e envia uma trama de resposta com controlo *RR* ou *REJ* dependendo do sucesso da receção.

**receiver\_llclose** – ramo da função *llclose* do lado do recetor. Espera a receção do comando *DISC* enviado pelo transmissor, envia um comando *DISC* de resposta e espera a receção de um comando *UA*.

**byteDestuffing** - realiza uma operação de *destuffing* a um determinado vetor de dados.

Note-se que no ficheiro *linklayer.h*, para além de declaradas as funções *llopen()*, *llclose()*, *llwrite()* e *llread()*, encontra-se também declarada a estrutura *linkLayer* e algumas macros para valores como o *role*, *baud rate*, *payload*, entre outros elementos dos parâmetros de conexão.

**3. Casos de uso principais**

O principal caso de uso deste projeto assenta no objetivo do programa que desenvolvemos: estabelecer a ligação entre uma porta de série e uma aplicação, efetuar a transmissão ou receção de dados mediante o modo de funcionamento ativo, e finalmente terminar a ligação. Para verificar a funcionalidade do programa, transferimos um ficheiro de imagem *penguin.gif* e, executando o programa em dois computadores ligados por uma porta de série, ou simulando duas máquinas independentes a correr os diferentes modos de funcionamento, verificamos que o envio e receção do ficheiro ocorre como esperado.

**4. Protocolo de ligação lógica**

O protocolo de ligação lógica de estabelece uma ligação com a camada física, neste caso a porta de série, e fornecer serviços à camada protocolar superior, neste caso a camada de aplicação. Esta camada desempenha diversas funções, desde a inicialização e fecho da ligação, ao *framing* das tramas, deteção de erros e controlo de fluxo. Neste projeto, estas funcionalidades encontram-se implementadas nas quatro funções principais da seguinte forma:

**4.1. int llopen(linkLayer connectionParameters)**

A função *llopen()* encontra-se dividida em duas componentes que correspondem aos dois modos de funcionamento possível. Ao executar a função do lado do transmissor, os parâmetros de ligação são guardados e é chamada a função *configureSerialterminal()* para estabelecer a ligação com a porta de série. Em seguida inicia-se o protocolo de ligação ao recetor: o transmissor envia uma trama com o comando *SET*. Quando a função é chamada no recetor, este espera a receção do comando *SET*, e caso o receba, envia uma trama com o comando *UA*. Quando o transmissor recebe o *UA*, a conexão está completa. Caso o transmissor não detete a receção do *UA* ao fim de um determinado intervalo de tempo, o comando *SET* é reenviado sucessivamente até que se dê a receção do *UA*, ou até que o número máximo de tentativas de conexão seja alcançado. Se isso acontecer, considera-se que o estabelecimento da ligação falhou e o programa termina. O mesmo sucede caso ocorra qualquer outro tipo de erros inesperados durante a execução das funções.

**4.2. int llwrite(char \*buf, int bufSize)**

A função *llwrite()* é utilizada no modo de funcionamento de transmissão com o objetivo de enviar tramas de informação para um recetor. Esta função começa por verificar se o número de *bytes* que se encontra no buffer para transmissão é válido, ou seja, superior a 0 e inferior à capacidade máxima, e em seguida chama a função *prepareInfoFrame()* para proceder com o *stuffing* dos dados e o cálculo do *BCC,* seguido do *framing* das tramas de informação. Com as tramas prontas para enviar, segue-se a escrita das mesmas para a porta de série, após a qual se espera pela receção de um controlo *RR* ou de um *REJ* para determinar se o envio da trama foi bem-sucedido, ou se é necessário efetuar um reenvio. No caso de não se receber nada ao fim de um determinado intervalo de tempo, é utilizado um conceito de *timeout* semelhante ao do *llopen()*, em que a retransmissão ocorre até ser recebido um *RR* ou até ao número limite de tentativas de envio definido nos parâmetros de ligação ser alcançado.

**4.3. int llread(char \*packet)**

A função *llread()* é utilizada no modo de funcionamento de receção para receber as tramas de informação enviadas pelo transmissor. Após verificar que o pacote é válido e que se trata de uma trama de informação, a função aloca memória para o campo de dados e procede com a leitura. Após a leitura ter terminado, é realizado o *destuffing* dos *bytes* recebidos, e por fim, o cálculo do *BlockCheckCharacter.* Após a leitura completa do pacote, verifica-se se o último *byte* corresponde ao *BCC* calculado com os dados que foram recebidos. Se corresponder, a leitura foi bem-sucedida e é enviada uma trama com o controlo, a informação é escrita no vetor *packet* e é retornado o tamanho do pacote. Caso contrário, é enviado uma trama de controlo REJ. No evento de o pacote recebido ser identificado como uma trama de informação repetida, é enviado na mesma um RR, mas os dados contidos na trama não são armazenados, uma vez que já foram passados anteriormente para a camada de aplicação e neste caso a função retorna 0.

**4.4 llclose(int showStatistics)**

A função *llclose()* encarrega-se de encerrar a ligação. Tal como a função *llopen()*, comporta-se de modo diferente dependendo do modo de funcionamento em que é chamada. No modo de transmissão, começa por enviar uma trama com o controlo *DISC* e espera a receção de uma trama de resposta com o controlo *DISC*, após a qual envia uma trama com o controlo *UA*. Caso a resposta com o *DISC* não seja logo recebida, o transmissor reenvia o controlo *DISC* até que a resposta pretendida seja recebida ou até o número de tentativas permitido ser alcançado. Após o envio do *UA*, é chamada a função *closeSerialterminal()* para terminar a ligação com a porta de série do lado do transmissor. Do lado do recetor, espera-se a receção do comando *DISC*, após a qual é enviada a trama de resposta também com o comando *DISC*. Finalmente, quando é recebido o *UA*, é chamada a função *closeSerialterminal()* e desta vez é terminada a ligação com a porta de série do lado do recetor, finalizando o protocolo de ligação de dados.

**5. Validação**

Com o intuito de testar a implementação desenvolvida da camada de ligação lógica, foram realizados os seguintes testes em laboratório:

* Envio do ficheiro de teste *penguin.gif* (com 11kB de tamanho), usando duas das máquinas presentes em laboratório ligadas por uma porta *RS-232.*
* Interrupção física do canal de ligação durante a transmissão do mesmo ficheiro.
* Envio do mesmo ficheiro introduzindo ruído na porta série, ligando alguns dos pinos de dados em curto-circuito.
* Variação do *baud rate*, e medição dos tempos de resposta das funções *llwrite()* e *llread()* utilizando um *payload* máximo de 1000 *bytes.* Os resultados deste teste estão disponibilizados no Anexo II, juntamente com algumas observações relativamente a este teste.

Fora do ambiente de laboratório, foi utilizado o *socat* para criar um *relay* que virtualizasse duas portas série no mesmo computador. Neste meio, foram realizados os seguintes testes:

* Envio de ficheiros com maior tamanho, com o objetivo de verificar a estabilidade da *API.*
* Corrupção dos campos *BCC* durante a transmissão e a receção de dados*,* com diferentes probabilidades,para simular a ocorrência de erros.

Durante todos os testes, foi utilizada a função da camada de aplicação *main.c* disponibilizada para este trabalho prático. Para verificar a integridade dos dados recebidos, a aplicação *sha256sum* foi utilizadapara comparar o ficheiro transmitido com o que foi recebido.

**6. Elementos de valorização**

Neste trabalho, de modo a simular a ocorrência de erros na transmissão de dados implementou-se os seguintes códigos para introduzir uma probabilidade de erros controlada por nós, tanto no envio de pacotes como na sua leitura, e verificar se o funcionamento do protocolo se mantinha operacional perante esses erros:

#ifdef test\_missing\_su\_frame

{

if(rand() % test\_missing\_su\_frame == 0)

control = 0x03; *// Simulate dropped packet*

}

#endif

O código acima corresponde à geração de erros na transmissão de dados, através da alteração forçada de certos pacotes para serem inválidos. O código seguinte gera erros na receção de dados através da alteração do *BCC2* de um pacote, o que irá invalidar a leitura do mesmo.

#ifdef test\_data\_corruption

{

if(rand() % test\_data\_corruption == 0)

BCC2++;

}

#endif

Ambos os geradores de erros podem ser ativados ou desativados no ficheiro *definitions.h* comentando ou descomentando a secção *testing controls*. Neste ficheiro também é possível configurar as probabilidades de erro.

De modo a termos uma noção dos erros a que o programa esteve sujeito durante a transferência de dados, definimos uma série de contadores, tanto no transmissor como no recetor, para guardar diversas estatísticas tais como: número de *REJs* recebidos, número de tramas de informação enviadas, número de *timeouts* ocorridos ao longo da execução do programa, número de retransmissões ocorridas, número de *REJs* enviados pelo recetor, número de tramas de informação recebidas e ainda que quantidade de tramas de informação recebidas eram repetidas. Estas estatísticas são imprimidas no *llclose()* em ambos os modos de funcionamento após o encerramento da ligação, se o parâmetro *showStatistics* for igual a 1.

Para além disso, implementámos no nosso código um *event log* onde guardamos o *timestamp* do evento em questão, erros que possam ocorrer ao longo da execução do programa e certos *checkpoints* na execução de funções importantes. De modo a efetuar a escrita destes eventos para um ficheiro, criámos a seguinte função:

int writeEventToFile(FILE \*fd, time\_t \*\_TIME, char \*str){

time(\_TIME);

struct tm \*local = localtime(\_TIME);

fprintf(fd, "[%02d:%02d:%02d] %s", local->tm\_hour, local->tm\_min, local->tm\_sec, str);

return 1;

}

**7. Conclusões**

Este trabalho prático consistia na implementação de um protocolo de ligação lógica entre dois computadores utilizando portas série, criando uma interface que permite a eficiente utilização das mesmas. Durante o desenvolvimento deste projeto, foi-nos possível perceber em detalhe o funcionamento da comunicação serial, dos protocolos de ligação de dados e as abstrações que estes providenciam às aplicações. Para além disto, foi possível perceber o que constitui uma boa ligação de dados e os mecanismos que devem ser postos em prática durante a implementação de forma a assegurar a robustez desta.

|  |  |
| --- | --- |
| Uma imagem com amarelo  Descrição gerada automaticamente  Figura 1 - Exemplo do penguin.gif após uma transmissão bem sucedida | Uma imagem com ClipArt  Descrição gerada automaticamente  Figura 2 - Exemplo do penguin.gif após uma transmissão mal sucedida |

Com este trabalho também foi possível adquirir conhecimentos relativamente ao desenvolvimento de aplicações para sistemas operativos GNU/Linux e relembrar alguns conceitos de programação em C. Durante a colaboração neste trabalho prático também pudemos aprender sobre as importantes ferramentas de controlo de versão (e.g. git) que permitiram agilizar o desenvolvimento num meio colaborativo.

Após a finalização, concluímos que todos os objetivos do trabalho foram alcançados e que este contribuiu para a consolidação de conceitos abordados nas aulas teóricas e laboratoriais.

**Anexo I - Código Fonte**

**linklayer.c**

#include "linklayer.h"

#include "transmitter.h"

#include "receiver.h"

#include "utils.h"

static int serialRole;

int llopen(linkLayer connectionParameters){

#ifdef testing

{

srand(time(NULL));

}

#endif

if(connectionParameters.role == TRANSMITTER){

serialRole = TRANSMITTER;

return transmitter\_llopen(connectionParameters);

}

else if(connectionParameters.role == RECEIVER){

serialRole = RECEIVER;

return receiver\_llopen(connectionParameters);

}

*// wrong parameter*

return -1;

}

int llclose(int showStatistics){

if(serialRole == TRANSMITTER){

return transmitter\_llclose(showStatistics);

}

else if(serialRole == RECEIVER){

return receiver\_llclose(showStatistics);

}

*//somekind of error*

return -1;

}

**linklayer.h**

#ifndef LINKLAYER

#define LINKLAYER

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include <signal.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

typedef struct linkLayer{

char serialPort[50];

int role; *//defines the role of the program: 0==Transmitter, 1=Receiver*

int baudRate;

int numTries;

int timeOut;

} linkLayer;

*//ROLE*

#define NOT\_DEFINED -1

#define TRANSMITTER 0

#define RECEIVER 1

*//SIZE of maximum acceptable payload; maximum number of bytes that application layer should send to link layer*

#define MAX\_PAYLOAD\_SIZE 1000

*//CONNECTION default values*

#define BAUDRATE\_DEFAULT B38400

#define MAX\_RETRANSMISSIONS\_DEFAULT 3

#define TIMEOUT\_DEFAULT 4

#define \_POSIX\_SOURCE 1 */\* POSIX compliant source \*/*

*//MISC*

#define FALSE 0

#define TRUE 1

*// Opens a conection using the "port" parameters defined in struct linkLayer, returns "-1" on error and "1" on sucess*

int llopen(linkLayer connectionParameters);

*// Sends data in buf with size bufSize*

int llwrite(char\* buf, int bufSize);

*// Receive data in packet*

int llread(char\* packet);

*// Closes previously opened connection; if showStatistics==TRUE, link layer should print statistics in the console on close*

int llclose(int showStatistics);

#endif

**utils.c**

#include "utils.h"

*/\**

*Globally declared termios structures*

*\*/*

struct termios oldtio, newtio;

int configureSerialterminal(linkLayer connectionParameters){

int fd = open(connectionParameters.serialPort, O\_RDWR | O\_NOCTTY);

if (fd < 0){

fprintf(stderr, "Couldn't open %s\n", connectionParameters.serialPort);

exit(-1);

}

*//copy current serial port configuration*

if(tcgetattr(fd, &oldtio) == -1){

fprintf(stderr, "couldn't save current port settings\n");

exit(-1);

}

*//Configure serial port connection*

speed\_t baud = convertBaudRate(connectionParameters.baudRate);

bzero(&newtio, sizeof(newtio));

*//newtio.c\_cflag = connectionParameters.baudRate | CS8 | CLOCAL | CREAD;*

newtio.c\_cflag = baud | CS8 | CLOCAL | CREAD;

newtio.c\_iflag = IGNPAR;

newtio.c\_oflag = 0;

*//Set local configuration*

newtio.c\_lflag = 0;

*//Set the read() timeout for 3 seconds*

newtio.c\_cc[VTIME] = (connectionParameters.timeOut \* 10);

newtio.c\_cc[VMIN] = 0; *// Set minimum of characters to be read*

tcflush(fd, TCIOFLUSH); *// flush whatever's in the buffer*

if ( tcsetattr(fd,TCSANOW,&newtio) == -1) {

fprintf(stderr, "tcsetattr");

exit(-1);

}

return fd;

}

int closeSerialterminal(int fd){

tcflush(fd, TCIOFLUSH); *// flush whatever's in the buffer*

if (tcsetattr(fd, TCSANOW, &oldtio) == -1) {

fprintf(stderr, "tcsetattr");

exit(-1);

}

close(fd);

return 1;

}

int checkHeader(int fd, u\_int8\_t \*cmd, int cmdLen){

if(cmd == NULL || cmdLen < 4 || cmdLen > 5)

return -1;

int state = 0, res;

u\_int8\_t rx\_byte;

while(state < cmdLen){

res = read(fd, &rx\_byte, 1);

if(res > 0) *//Something was read*

DEBUG\_PRINT("[checkHeader()] received byte: 0x%02x -- state: %d \n", rx\_byte, state);

else *//Nothing has been read or some kind of error*

break;

switch(state){ *//State machine*

case 0:

if(rx\_byte==cmd[0]) *//FLAG*

state = 1;

else

state = 0;

break;

case 1:

if(rx\_byte==cmd[1]) *//Address field*

state = 2;

else if(rx\_byte==cmd[0])

state = 1;

else

state = 0;

break;

case 2:

if(rx\_byte==cmd[2]) *//Control field*

state = 3;

else if(rx\_byte == cmd[0])

state = 1;

else

state = 0;

break;

case 3:

if(rx\_byte == cmd[3]) *//BCC1*

state = 4;

else if(rx\_byte == cmd[0])

state = 1;

else

state = 0;

break;

case 4: *//In case there are 5 elements*

if(rx\_byte == cmd[0]) *//FLAG*

state = 5;

else

state = 0;

break;

}

}

if(state == cmdLen) *//everything OK*

return 1;

else if(res < 0) *//somekind of error*

return -1;

*//read() had nothing to read*

return 0;

}

u\_int8\_t readControlField(int fd, int cmdLen){

u\_int8\_t cField, rx\_byte;

int state = 0, res;

while(state != cmdLen){

res = read(fd, &rx\_byte, 1);

if(res > 0) *//Something was read after 3 seconds*

DEBUG\_PRINT("[readControlField()]received byte: 0x%02x -- state: %d \n", rx\_byte, state);

else *//Nothing has been read or some kind of error*

break;

switch(state){ *//State machine*

case 0: *//Flag*

if(rx\_byte==FLAG)

state = 1;

else

state = 0;

break;

case 1: *//Address field*

if(rx\_byte==A\_tx || rx\_byte==A\_rx)

state = 2;

else if(rx\_byte==FLAG)

state = 1;

else

state = 0;

break;

case 2: *//Control field*

if(rx\_byte==FLAG)

state = 1;

else{

state = 3;

cField = rx\_byte;

}

break;

case 3:

if(rx\_byte == (cField^A\_tx) || rx\_byte == (cField^A\_rx)) *//BCC1*

state = 4;

else if(rx\_byte == FLAG)

state = 1;

else

state = 0;

break;

case 4:

if(rx\_byte == FLAG)

state = 5;

else

state = 0;

break;

}

}

if(state == cmdLen){

if((cField == C\_RR(0) || cField == C\_RR(1)) || (cField == C\_REJ(0) || cField == C\_REJ(1)) || (cField == C(0) || cField == C(1)) || cField == C\_SET || cField == C\_DISC || cField == C\_UA) *//Check*

return cField;

}

if(res == 0) *// Nothing was read for VTIME sec*

return 0xFE;

*// In case of error while reading (res < 0)*

return 0xFF;

}

u\_int8\_t generateBCC(u\_int8\_t \*data, int dataSize){

if(dataSize == 1) *//in case of only one element in the vector*

return data[0];

u\_int8\_t BCC = (data[0] ^ data[1]);

for (int i = 2; i < dataSize; i++)

BCC ^= data[i];

return BCC;

}

linkLayer \*checkParameters(linkLayer link){

linkLayer \*aux = malloc(sizeof(linkLayer));

if(aux == NULL)

return NULL;

aux->baudRate = link.baudRate;

if(link.role != TRANSMITTER && link.role != RECEIVER)

aux->role = NOT\_DEFINED;

else

aux->role = link.role;

if(link.numTries < 1)

aux->numTries = MAX\_RETRANSMISSIONS\_DEFAULT;

else

aux->numTries = link.numTries;

if(link.timeOut < 0)

aux->timeOut = TIMEOUT\_DEFAULT;

else

aux->timeOut = link.timeOut;

strcpy(aux->serialPort, link.serialPort);

return aux;

}

int writeEventToFile(FILE \*fd, time\_t \*\_TIME, char \*str){

time(\_TIME);

struct tm \*local = localtime(\_TIME);

fprintf(fd, "[%02d:%02d:%02d] %s", local->tm\_hour, local->tm\_min, local->tm\_sec, str);

return 1;

}

speed\_t convertBaudRate(int baud){

switch (baud)

{

case 0:

#ifndef B0

return BAUDRATE\_DEFAULT;

#else

return B0;

#endif

break;

case 50:

#ifndef B50

return BAUDRATE\_DEFAULT;

#else

return B50;

#endif

break;

case 75:

#ifndef B75

return BAUDRATE\_DEFAULT;

#else

return B75;

#endif

break;

case 110:

#ifndef B110

return BAUDRATE\_DEFAULT;

#else

return B110;

#endif

break;

case 134:

#ifndef B134

return BAUDRATE\_DEFAULT;

#else

return B134;

#endif

break;

case 150:

#ifndef B150

return BAUDRATE\_DEFAULT;

#else

return B150;

#endif

break;

case 200:

#ifndef B200

return BAUDRATE\_DEFAULT;

#else

return B200;

#endif

break;

case 300:

#ifndef B300

return BAUDRATE\_DEFAULT;

#else

return B300;

#endif

break;

case 600:

#ifndef B600

return BAUDRATE\_DEFAULT;

#else

return B600;

#endif

break;

case 1200:

#ifndef B1200

return BAUDRATE\_DEFAULT;

#else

return B1200;

#endif

break;

case 1800:

#ifndef B1800

return BAUDRATE\_DEFAULT;

#else

return B1800;

#endif

break;

case 2400:

#ifndef B2400

return BAUDRATE\_DEFAULT;

#else

return B2400;

#endif

break;

case 4800:

#ifndef B4800

return BAUDRATE\_DEFAULT;

#else

return B4800;

#endif

break;

case 9600:

#ifndef B9600

return BAUDRATE\_DEFAULT;

#else

return B9600;

#endif

break;

case 19200:

#ifndef B19200

return BAUDRATE\_DEFAULT;

#else

return B19200;

#endif

break;

case 38400:

#ifndef B38400

return BAUDRATE\_DEFAULT;

#else

return B38400;

#endif

break;

case 57600:

#ifndef B57600

return BAUDRATE\_DEFAULT;

#else

return B57600;

#endif

break;

case 115200:

#ifndef B115200

return BAUDRATE\_DEFAULT;

#else

return B115200;

#endif

break;

case 230400:

#ifndef B230400

return BAUDRATE\_DEFAULT;

#else

return B230400;

#endif

break;

case 460800:

#ifndef B460800

return BAUDRATE\_DEFAULT;

#else

return B460800;

#endif

break;

case 500000:

#ifndef B500000

return BAUDRATE\_DEFAULT;

#else

return B500000;

#endif

break;

case 576000:

#ifndef B576000

return BAUDRATE\_DEFAULT;

#else

return B576000;

#endif

break;

case 921600:

#ifndef B921600

return BAUDRATE\_DEFAULT;

#else

return B921600;

#endif

break;

case 1000000:

#ifndef B1000000

return BAUDRATE\_DEFAULT;

#else

return B1000000;

#endif

break;

case 1152000:

#ifndef B1152000

return BAUDRATE\_DEFAULT;

#else

return B1152000;

#endif

break;

case 1500000:

#ifndef B1500000

return BAUDRATE\_DEFAULT;

#else

return B1500000;

#endif

break;

case 2000000:

#ifndef B2000000

return BAUDRATE\_DEFAULT;

#else

return B2000000;

#endif

break;

*// SPARC architecture*

case 76800:

#ifndef B76800

return BAUDRATE\_DEFAULT;

#else

return B76800;

#endif

break;

case 153600:

#ifndef B153600

return BAUDRATE\_DEFAULT;

#else

return B153600;

#endif

break;

case 307200:

#ifndef B307200

return BAUDRATE\_DEFAULT;

#else

return B307200;

#endif

break;

case 614400:

#ifndef B614400

return BAUDRATE\_DEFAULT;

#else

return B614400;

#endif

break;

*//non-SPARC architectures (not in POSIX)*

case 2500000:

#ifndef B2500000

return BAUDRATE\_DEFAULT;

#else

return B2500000;

#endif

break;

case 3000000:

#ifndef B3000000

return BAUDRATE\_DEFAULT;

#else

return B3000000;

#endif

break;

case 3500000:

#ifndef B3500000

return BAUDRATE\_DEFAULT;

#else

return B3500000;

#endif

break;

case 4000000:

#ifndef B4000000

return BAUDRATE\_DEFAULT;

#else

return B4000000;

#endif

break;

default:

return BAUDRATE\_DEFAULT;

break;

}

return BAUDRATE\_DEFAULT;

}

**utils.h**

#ifndef UTILS\_H

#define UTILS\_H

#include <time.h>

#include "linklayer.h"

#include "definitions.h"

*/\**

*Configure serial port terminal I/O using linkLayer*

*Return values:*

*file descriptor id - If successful*

*In case of error the whole program is terminated*

*\*/*

int configureSerialterminal(linkLayer connectionParameters);

*/\**

*Close Serial Port Terminal connection upon llclose()*

*Return values:*

*1 - connection closed successfully*

*-1 - error*

*\*/*

int closeSerialterminal(int fd);

*/\**

*Tries to read a specific header for a given frame, only works*

*for valid header lenghts of either 4 or 5 bytes*

*This function is protected by a timer, after 3 seconds of*

*if nothing is read it'll return 0*

*Return values:*

*1 - the command was read successfully*

*0 - couldn´t read anything*

*-1 - error while reading*

*\*/*

int checkHeader(int fd, u\_int8\_t \*cmd, int cmdLen);

*/\**

*Tries to read and then output the control field of a*

*supervision or control frame header*

*Return values:*

*frame control field - the header was read successfully*

*0xFE - nothing was read*

*0xFF - error while reading*

*\*/*

u\_int8\_t readControlField(int fd, int cmdLen);

*/\**

*Convert an int to the apropriate speed\_t that termios understands,*

*also checks if a given baud rate is defined*

*for the current system. This check is done during compiling, so*

*one should already compile with the target architecture in mind*

*eg. 9600 -> B9600 (= 00000015)*

*If baud is an invalid value, the function will return*

*BAUDRATE\_DEFAULT configured in linklayer.h*

*\*/*

speed\_t convertBaudRate(int baud);

*/\**

*Calculate a Block Check Character for a given data vector*

*Return Values*

*BCC - BCC was correctly generated*

*-1 - somekind of error*

*\*/*

u\_int8\_t generateBCC(u\_int8\_t \*data, int dataSize);

*/\**

*Check and copy linklayer parameters*

*Invalid values are given default values assigned in linklayer.h*

*Return Values*

*pointer to new struct*

*NULL - error*

*\*/*

linkLayer \*checkParameters(linkLayer link);

*/\**

*Write current time and a given string str to a file*

*Used for logging of events*

*Return values*

*1 - always*

*\*/*

int writeEventToFile(FILE \*fd, time\_t \*\_TIME, char \*str);

#endif

**transmitter.c**

#include "transmitter.h"

*/\**

*Globally declared serial terminal file descriptor*

*and linklayer connection parameters*

*\*/*

static int tx\_fd;

linkLayer \*tx\_connectionParameters;

*//File for event logs*

FILE \*tx\_stats;

char tx\_event\_fileName[] = "tx\_statistics";

time\_t tx\_now;

*//ERROR COUNTERS for statistics*

int stat\_txRejCount = 0;

int stat\_txIFrames = 0;

int stat\_timeOutsCount = 0; *//different from timeoutCount*

int stat\_retransmittionCount = 0;

*// llwrite()*

static u\_int8\_t tx\_currSeqNumber = 0; *// Ns = 0, 1*

*//timeout related functions*

u\_int8\_t timeoutFlag, timerFlag, timeoutCount;

int transmitter\_llopen(linkLayer connectionParameters){

tx\_connectionParameters = checkParameters(connectionParameters);

tx\_fd = configureSerialterminal(\*tx\_connectionParameters);

*// open event log file*

tx\_stats = fopen(tx\_event\_fileName, "w");

if(tx\_stats == NULL){

writeEventToFile(tx\_stats, &tx\_now, "Error opening statistics file\n");

return -1;

}

writeEventToFile(tx\_stats, &tx\_now, "llopen() called\n");

*// SET frame header*

u\_int8\_t cmdSet[] = {FLAG, A\_tx, C\_SET, (A\_tx ^ C\_SET), FLAG};

*// UA frame header, what we are expecting to receive*

u\_int8\_t cmdUA[] = {FLAG, A\_tx, C\_UA, (A\_tx ^ C\_UA), FLAG};

(void) signal(SIGALRM, timeOut);

int res = write(tx\_fd, cmdSet, 5);

if(res < 0){

writeEventToFile(tx\_stats, &tx\_now, "Error writing to serial port\n");

fclose(tx\_stats);

return -1;

}

writeEventToFile(tx\_stats, &tx\_now, "Sending SET command\n");

*//printf("%d bytes written\n", res);*

timeoutFlag = 0, timeoutCount = 0, timerFlag = 1;

while (timeoutCount < tx\_connectionParameters->numTries){

if(timerFlag){

alarm(tx\_connectionParameters->timeOut);

timerFlag = 0;

}

int readResult = checkHeader(tx\_fd, cmdUA, 5);

if(readResult < 0){

writeEventToFile(tx\_stats, &tx\_now, "Error reading command from serial port\n");

return -1;

}

else if(readResult > 0){ *//Success*

writeEventToFile(tx\_stats, &tx\_now, "Connection established\n");

signal(SIGALRM, SIG\_IGN); *//disable interrupt handler*

return 1;

}

if(timeoutFlag){

int res = write(tx\_fd, cmdSet, 5);

if(res < 0){

writeEventToFile(tx\_stats, &tx\_now, "Error writing to serial port\n");

fclose(tx\_stats);

return -1;

}

writeEventToFile(tx\_stats, &tx\_now, "Sending SET command again\n");

timeoutCount++;

stat\_timeOutsCount++; *//total number of timeouts*

timeoutFlag = 0;

}

}

writeEventToFile(tx\_stats, &tx\_now, "Failed to establish connection\n");

fclose(tx\_stats);

return -1;

}

u\_int8\_t \*prepareInfoFrame(u\_int8\_t \*buf, int bufSize, int \*outputSize, u\_int8\_t sequenceBit){

if(buf == NULL || bufSize <= 0 || bufSize > MAX\_PAYLOAD\_SIZE){

writeEventToFile(tx\_stats, &tx\_now, "prepareIntoForm() - invalid parameters\n");

return NULL;

}

*// Prepare the frame data*

u\_int8\_t \*data = malloc(bufSize + 1);

if(data == NULL){

writeEventToFile(tx\_stats, &tx\_now, "prepareIntoForm() - data memory allocation failed\n");

return NULL;

}

for (int i = 0; i < bufSize; i++) *// Copy the buffer*

data[i] = buf[i];

*// Add the BCC byte*

data[bufSize] = (u\_int8\_t) generateBCC((u\_int8\_t\*) buf, bufSize);

int stuffedSize = 0;

u\_int8\_t \*stuffedData = byteStuffing(data, bufSize+1, &stuffedSize);

if(stuffedData == NULL){

writeEventToFile(tx\_stats, &tx\_now, "prepareIntoForm() - byte stuffing failed\n");

return NULL;

}

free(data);

u\_int8\_t \*outgoingData = malloc(stuffedSize + 5);

if(outgoingData == NULL){

writeEventToFile(tx\_stats, &tx\_now, "prepareIntoForm() - outgoingData memory allocation failed\n");

free(stuffedData);

return NULL;

}

outgoingData[0] = FLAG;

outgoingData[1] = A\_tx;

outgoingData[2] = C(sequenceBit);

outgoingData[3] = A\_tx ^ C(sequenceBit);

*// Copy the stuffed data*

for (int i = 0; i < stuffedSize; i++)

outgoingData[i+4] = stuffedData[i];

*// Add a trailling FLAG*

outgoingData[stuffedSize + 4] = FLAG;

free(stuffedData);

\*outputSize = stuffedSize + 5;

return outgoingData;

}

int llwrite(char \*buf, int bufSize){

fputc((int)'\n', tx\_stats);

writeEventToFile(tx\_stats, &tx\_now, "llwrite() called\n");

if(buf == NULL || bufSize > MAX\_PAYLOAD\_SIZE)

return -1;

int frameSize = 0;

u\_int8\_t \*frame = prepareInfoFrame((u\_int8\_t\*) buf, bufSize, &frameSize, tx\_currSeqNumber);

*//DEBUG\_PRINT("[llopen() start] SEQ NUMBER: %d\n", tx\_currSeqNumber);*

*// Write for the first time*

int res = write(tx\_fd, frame, frameSize);

if(res < 0){

writeEventToFile(tx\_stats, &tx\_now, "Error writing to serial port\n");

free(frame);

return -1;

}

stat\_txIFrames++;

writeEventToFile(tx\_stats, &tx\_now, "Written ");

fprintf(tx\_stats, "%d bytes to serial port\n", res);

u\_int8\_t control;

(void) signal(SIGALRM, timeOut); *// Set up signal handler*

*//Cycle through timeouts*

timeoutFlag = 0; timerFlag = 1; timeoutCount = 0;

while (timeoutCount < tx\_connectionParameters->numTries){

if(timerFlag){

alarm(tx\_connectionParameters->timeOut);

timerFlag = 0;

}

*//read the incoming frame control field*

control = readControlField(tx\_fd, 5);

#ifdef test\_missing\_su\_frame

{

if(rand() % test\_missing\_su\_frame == 0)

control = 0x03; *// Simulate dropped packet*

}

#endif

*//Check if the header and the sequence number are valid*

if(control == C\_RR(!tx\_currSeqNumber)){ *//Receive receipt*

writeEventToFile(tx\_stats, &tx\_now, "Received RR\n");

tx\_currSeqNumber = !tx\_currSeqNumber;

(void) signal(SIGALRM, SIG\_IGN); *//disable signal handler*

free(frame);

return bufSize;

}

else if(control == C\_REJ(tx\_currSeqNumber)){ *//REJ*

res = write(tx\_fd, frame, frameSize);

if(res < 0){

writeEventToFile(tx\_stats, &tx\_now, "Frame retransmission failed\n");

free(frame);

return -1;

}

writeEventToFile(tx\_stats, &tx\_now, "(Retransmission) Written ");

fprintf(tx\_stats, "%d bytes to serial port\n", res);

stat\_txRejCount++;

stat\_txIFrames++;

stat\_retransmittionCount++;

timeoutCount = 0;

alarm(tx\_connectionParameters->timeOut); *// alarm reset*

}

if(timeoutFlag){

res = write(tx\_fd, frame, frameSize);

if(res < 0){

writeEventToFile(tx\_stats, &tx\_now, "Frame retransmission failed\n");

free(frame);

return -1;

}

writeEventToFile(tx\_stats, &tx\_now, "(Retransmission) Written ");

fprintf(tx\_stats, "%d bytes to serial port\n", res);

stat\_txIFrames++;

stat\_retransmittionCount++;

stat\_timeOutsCount++;

timeoutCount++;

timeoutFlag = 0;

}

}

(void) signal(SIGALRM, SIG\_IGN); *//disable signal handler*

return -1;

}

int transmitter\_llclose(int showStatistics){

fputc((int)'\n', tx\_stats);

writeEventToFile(tx\_stats, &tx\_now, "llclose() called\n");

*// DISC frame header*

u\_int8\_t cmdDisc[] = {FLAG, A\_tx, C\_DISC, A\_tx ^ C\_DISC, FLAG};

*// UA frame header*

u\_int8\_t cmdUA[] = {FLAG, A\_rx, C\_UA, A\_rx ^ C\_UA, FLAG};

(void) signal(SIGALRM, timeOut);

int res = write(tx\_fd, cmdDisc, 5);

if(res < 0){

writeEventToFile(tx\_stats, &tx\_now, "Error writing to serial port\n");

fclose(tx\_stats);

return -1;

}

writeEventToFile(tx\_stats, &tx\_now, "Sent DISC\n");

DEBUG\_PRINT("Disconnect command sent\n");

int timeoutCount = 0;

timeoutFlag = 0, timeoutCount = 0, timerFlag = 1;

while (timeoutCount < tx\_connectionParameters->numTries){

if(timerFlag){

alarm(tx\_connectionParameters->timeOut);

timerFlag = 0;

}

res = checkHeader(tx\_fd, cmdDisc, 5);

if(res < 0){

writeEventToFile(tx\_stats, &tx\_now, "Error writing to serial port\n");

fclose(tx\_stats);

return -1;

}

else if(res > 0){ *//Success*

signal(SIGALRM, SIG\_IGN); *//disable interrupt handler*

writeEventToFile(tx\_stats, &tx\_now, "Received DISC, sending UA\n");

*//DEBUG\_PRINT("Received Disconnection confirm, sending UA\n");*

break;

}

if(timeoutFlag){

res = write(tx\_fd, cmdDisc, 5);

if(res < 0){

writeEventToFile(tx\_stats, &tx\_now, "Error writing to serial port\n");

fclose(tx\_stats);

return -1;

}

writeEventToFile(tx\_stats, &tx\_now, "DISC command sent again\n");

*//DEBUG\_PRINT("Disconnect command sent again\n");*

timeoutCount++;

stat\_timeOutsCount++;

}

}

if(write(tx\_fd, cmdUA, 5) < 0){

writeEventToFile(tx\_stats, &tx\_now, "Error writing to serial port\n");

fclose(tx\_stats);

return -1;

}

writeEventToFile(tx\_stats, &tx\_now, "UA control sent\n");

free(tx\_connectionParameters);

closeSerialterminal(tx\_fd);

writeEventToFile(tx\_stats, &tx\_now, "Connection Closed\n");

fclose(tx\_stats);

if(showStatistics){

printf("\n######## LINK LAYER STATISTICS ########\n");

printf("# of I frames sent: %d\n", stat\_txIFrames);

printf("# of total connection timeouts: %d\n", stat\_timeOutsCount);

printf("# of REJ frames received: %d\n", stat\_txRejCount);

printf("# of retransmitted frames: %d\n", stat\_retransmittionCount);

printf("Open event log using less? [y/n]\n");

res = getchar();

if(res == 'y'){

char command[100] = "less ";

strcat(command, tx\_event\_fileName);

system(command);

}

}

return 1;

}

u\_int8\_t \*byteStuffing(u\_int8\_t \*data, int dataSize, int \*outputDataSize){

if(data == NULL || outputDataSize == NULL || dataSize < 1){

writeEventToFile(tx\_stats, &tx\_now, "byteStuffing() - one or more parameters are invalid\n");

return NULL;

}

*// Maximum possible stuffed data size is twice that of the input data array*

*// We prevent having to reallocate memory during stuffing*

u\_int8\_t \*stuffedData = malloc(2\*dataSize);

if(stuffedData == NULL){

writeEventToFile(tx\_stats, &tx\_now, "byteStuffing() - stuffedData memory allocation failed\n");

return NULL;

}

int size = 0;

for (int i = 0; i < dataSize; i++){

switch (data[i])

{

case FLAG:

stuffedData[size++] = ESC;

stuffedData[size++] = FLAG ^ 0x20;

break;

case ESC:

stuffedData[size++] = ESC;

stuffedData[size++] = ESC ^ 0x20;

break;

default:

stuffedData[size++] = data[i];

break;

}

}

*// Trim the array in memory if needed*

if(size != 2\*dataSize){

stuffedData = realloc(stuffedData, size);

if(stuffedData == NULL){

writeEventToFile(tx\_stats, &tx\_now, "byteStuffing() - stuffedData memory reallocation failed\n");

return NULL;

}

}

\*outputDataSize = size;

return stuffedData;

}

void timeOut(){

writeEventToFile(tx\_stats, &tx\_now, "Connection timeout\n");

*//printf("Connection timeout\n");*

timeoutFlag = 1; *//indicate there was a timeout*

timerFlag = 1; *//restart the timer*

}

**transmitter.h**

#ifndef TRANSMITTER\_H

#define TRANSMITTER\_H

#include "utils.h"

*/\**

*Transmitter end of llopen() which is then passed on to the*

*actual function*

*Return values*

*1 - on successful connection establishment*

*-1 - on error*

*\*/*

int transmitter\_llopen(linkLayer connectionParameters);

*/\**

*Transmitter end of llclose()*

*Return values*

*1 - on success*

*-1 - on error*

*\*/*

int transmitter\_llclose(int showStatistics);

*/\**

*Auxiliary timeout function for handling SIGALRM signals*

*\*/*

void timeOut();

*/\**

*Perform a byte stuffing operation on vector data*

*Return values:*

*pointer to a new vector*

*NULL - somekind of error*

*\*/*

u\_int8\_t \*byteStuffing(u\_int8\_t \*data, int dataSize, int \*outputDataSize);

*/\**

*Prepare an Information Frame*

*Return Values:*

*pointer to frame byte array - success*

*NULL - somekind of error*

*\*/*

u\_int8\_t \*prepareInfoFrame(u\_int8\_t \*buf, int bufSize, int \*outputSize, u\_int8\_t sequenceBit);

#endif

**receiver.c**

#include "receiver.h"

*/\**

*Globally declared serial terminal file descriptor*

*and linklayer parameters*

*\*/*

static int rx\_fd;

linkLayer \*rx\_connectionParameters;

*//File and time\_t for event logs*

FILE \*rx\_stats;

char rx\_event\_fileName[] = "rx\_statistics";

time\_t rx\_now;

*//ERROR COUNTERS for statistics*

int stat\_rxRejCount = 0;

int stat\_rxIFrames = 0;

int stat\_duplicatesReceived = 0;

*//Last successfully read I frame sequence number*

static u\_int8\_t rx\_prevSeqNum = 1;

int receiver\_llopen(linkLayer connectionParameters){

*// Save connection parameters*

rx\_connectionParameters = checkParameters(connectionParameters);

rx\_fd = configureSerialterminal(\*rx\_connectionParameters);

*// open event log file*

rx\_stats = fopen(rx\_event\_fileName, "w");

if(rx\_stats == NULL){

writeEventToFile(rx\_stats, &rx\_now, "Error opening statistics file\n");

return -1;

}

writeEventToFile(rx\_stats, &rx\_now, "llopen() called\n");

*// We're expecting a SET command from tx*

u\_int8\_t cmdSET[] = {FLAG, A\_tx, C\_SET, (A\_tx ^ C\_SET), FLAG};

if(checkHeader(rx\_fd, cmdSET, 5) <= 0){

writeEventToFile(rx\_stats, &rx\_now, "Haven't received SET\n");

return -1;

}

writeEventToFile(rx\_stats, &rx\_now, "Received SET, sending UA\n");

*//printf("Received SET, sending UA\n");*

*// UA frame reply*

u\_int8\_t repUA[] = {FLAG, A\_tx, C\_UA, (A\_tx ^ C\_UA), FLAG};

if(write(rx\_fd, repUA, 5) < 0){

writeEventToFile(rx\_stats, &rx\_now, "Error writing to serial port\n");

return -1;

}

writeEventToFile(rx\_stats, &rx\_now, "Sent UA, connection secured\n");

return 1;

}

int llread(char \*packet){

fputc((int)'\n', rx\_stats);

writeEventToFile(rx\_stats, &rx\_now, "llread() called\n");

*//DEBUG\_PRINT("[llopen() call] %d \n", rx\_prevSeqNum);*

if(packet == NULL){

writeEventToFile(rx\_stats, &rx\_now, "invalid parameters\n");

return -1;

}

u\_int8\_t \*datafield = malloc(2\*MAX\_PAYLOAD\_SIZE + 3);

u\_int8\_t \*destuffedData;

if(datafield == NULL){

writeEventToFile(rx\_stats, &rx\_now, "Memory allocation failed\n");

return -1;

}

*//possible reply frames*

u\_int8\_t repRR[] = {FLAG, A\_tx, C\_RR(rx\_prevSeqNum), (A\_tx ^ C\_RR(rx\_prevSeqNum)), FLAG};

u\_int8\_t repREJ[] = {FLAG, A\_tx, C\_REJ(!rx\_prevSeqNum), (A\_tx ^ C\_REJ(!rx\_prevSeqNum)), FLAG};

u\_int8\_t STOP = 0, rx\_byte, BCC2, currSeqNum;

int res, i, destuffedDataSize;

while(!STOP){

res = readControlField(rx\_fd, 4);

if(res == 0xFF || res == C\_DISC){

writeEventToFile(rx\_stats, &rx\_now, "Error while reading frame header or DISC received \n");

return -1;

}

else if(res != C(0) && res != C(1)){ *// not an I frame*

writeEventToFile(rx\_stats, &rx\_now, "Didn't receive I frame\n");

return 0;

}

currSeqNum = I\_SEQ(res);

if (currSeqNum == !rx\_prevSeqNum){ *//This is a new I frame*

writeEventToFile(rx\_stats, &rx\_now, "New I frame received\n");

stat\_rxIFrames++;

*// Read stuffed data field, excluding FLAG*

i = 0;

res = read(rx\_fd, &rx\_byte, 1);

if(res < 0){

writeEventToFile(rx\_stats, &rx\_now, "Could not read from serial port\n");

free(datafield);

return -1;

}

do{

datafield[i++] = rx\_byte;

res = read(rx\_fd, &rx\_byte, 1);

if(res < 0){

writeEventToFile(rx\_stats, &rx\_now, "Could not read from serial port\n");

free(datafield);

return -1;

}

} while (rx\_byte != FLAG);

destuffedData = byteDestuffing(datafield, i, &destuffedDataSize);

BCC2 = generateBCC(destuffedData, destuffedDataSize - 1);

#ifdef test\_data\_corruption

{

if(rand() % test\_data\_corruption == 0)

BCC2++;

}

#endif

if(destuffedData[destuffedDataSize-1] == BCC2){

writeEventToFile(rx\_stats, &rx\_now, "BCC2 check passed, sending RR\n");

*//free(datafield);*

*//send RR*

res = write(rx\_fd, repRR, 5);

if(res < 0){

writeEventToFile(rx\_stats, &rx\_now, "Could not write to serial port\n");

free(datafield);

free(destuffedData);

return -1;

}

rx\_prevSeqNum = !rx\_prevSeqNum;

STOP = 1; *//break;*

}

else{

writeEventToFile(rx\_stats, &rx\_now, "BCC2 check failed, sending REJ\n");

free(destuffedData);

res = write(rx\_fd, repREJ, 5);

stat\_rxRejCount++;

if(res < 0){

writeEventToFile(rx\_stats, &rx\_now, "Could not write to serial port\n");

free(datafield);

return -1;

}

*//continue;*

}

}

else{ *//In case of duplicate I frame*

writeEventToFile(rx\_stats, &rx\_now, "Duplicate frame received, sending RR\n");

stat\_duplicatesReceived++;

u\_int8\_t repRR\_rej[] = {FLAG, A\_tx, C\_RR(!rx\_prevSeqNum), (A\_tx ^ C\_RR(!rx\_prevSeqNum)), FLAG};

res = write(rx\_fd, repRR\_rej, 5);

if(res < 0){

writeEventToFile(rx\_stats, &rx\_now, "Could not write to serial port\n");

free(datafield);

return -1;

}

do{ *//Dummy read*

res = read(rx\_fd, &rx\_byte, 1);

} while (rx\_byte != FLAG || res != 0);

}

}

writeEventToFile(rx\_stats, &rx\_now, "Writing read data to packet\n");

for (int i = 0; i < destuffedDataSize; i++)

packet[i] = destuffedData[i];

free(datafield);

free(destuffedData);

return (destuffedDataSize - 1);

}

u\_int8\_t \*byteDestuffing(u\_int8\_t \*data, int dataSize, int \*outputDataSize){

if(data == NULL || outputDataSize == NULL){

writeEventToFile(rx\_stats, &rx\_now, "byteDestuffing() - invalid parameters in function call\n");

return NULL;

}

u\_int8\_t \*destuffedData = malloc(dataSize);

if(destuffedData == NULL){

writeEventToFile(rx\_stats, &rx\_now, "byteDestuffing() - destuffedData memory allocation failed\n");

return NULL;

}

int size = 0;

for (int i = 0; i < dataSize; i++)

{

if(data[i] != ESC)

destuffedData[size++] = data[i];

else

switch (data[++i])

{

case FLAG^0x20:

destuffedData[size++] = FLAG;

break;

case ESC^0x20:

destuffedData[size++] = ESC;

break;

default: *//invalid escape character use*

free(destuffedData);

writeEventToFile(rx\_stats, &rx\_now, "byteDestuffing() - invalid invalid escape follow-up character used\n");

return NULL;

break;

}

}

if(size != dataSize){

destuffedData = realloc(destuffedData, size);

if(destuffedData == NULL){

writeEventToFile(rx\_stats, &rx\_now, "byteDestuffing() - memory reallocation failed\n");

return NULL;

}

}

\*outputDataSize = size;

return destuffedData;

}

int receiver\_llclose(int showStatistics){

fputc((int)'\n', rx\_stats);

writeEventToFile(rx\_stats, &rx\_now, "llclose() called\n");

*// DISC frame header*

u\_int8\_t cmdDisc[] = {FLAG, A\_tx, C\_DISC, A\_tx ^ C\_DISC, FLAG};

*// UA frame header expected to receive*

u\_int8\_t cmdUA[] = {FLAG, A\_rx, C\_UA, A\_rx ^ C\_UA, FLAG};

if(checkHeader(rx\_fd, cmdDisc, 5) < 0){

writeEventToFile(rx\_stats, &rx\_now, "Error reading from serial port\n");

return -1;

}

writeEventToFile(rx\_stats, &rx\_now, "Received DISC, sending DISC back\n");

int res = write(rx\_fd, cmdDisc, 5);

if(res < 0){

writeEventToFile(rx\_stats, &rx\_now, "Error writing to serial port\n");

return -1;

}

writeEventToFile(rx\_stats, &rx\_now, "Sent DISC\n");

if(checkHeader(rx\_fd, cmdUA, 5) < 0){

writeEventToFile(rx\_stats, &rx\_now, "Error reading from serial port\n");

return -1;

}

free(rx\_connectionParameters);

closeSerialterminal(rx\_fd);

writeEventToFile(rx\_stats, &rx\_now, "Connection Closed\n");

fclose(rx\_stats);

if(showStatistics){

printf("\n######## LINK LAYER STATISTICS ########\n");

printf("# of I frames received: %d \n", stat\_rxIFrames);

printf("# of REJ frames sent: %d \n", stat\_rxRejCount);

printf("# of duplicate frames received: %d \n", stat\_duplicatesReceived);

printf("Open event log using less? [y/n]\n");

res = getchar();

if(res=='y'){ *//Open file using less*

char command[100] = "less ";

strcat(command, rx\_event\_fileName);

system(command);

}

}

return 1;

}

**receiver.h**

#ifndef RECEIVER\_H

#define RECEIVER\_H

#include "utils.h"

*//static int rx\_fd;*

*/\**

*Receiver end of llopen() which is then passed on to the*

*actual function*

*Return values*

*1 - on successful connection establishment*

*-1 - on error*

*\*/*

int receiver\_llopen(linkLayer connectionParameters);

*/\**

*Transmitter end of llclose()*

*Return values*

*1 - on success*

*-1 - on error*

*\*/*

int receiver\_llclose(int showStatistics);

*/\**

*Perform a byte destuffing operation on vector data*

*Return values:*

*pointer to a new vector, in case of success*

*NULL, if somekind of error*

*\*/*

u\_int8\_t \*byteDestuffing(u\_int8\_t \*data, int dataSize, int \*outputDataSize);

#endif

**definitions.h**

#ifndef DEFINITIONS\_H

#define DEFINITIONS\_H

*//Frame formats*

#define FLAG 0x7E *//0b01111110*

#define ESC 0x7D *//0b01111101*

*//Address fields*

#define A\_tx 0x03 *//Commands sent by the Transmitter and Answers from the receiver*

#define A\_rx 0x01 *//Commands sent by the Receiver and Answers from the transmitter*

*//Supervision and Unnumbered Frame Control*

#define C\_SET 0x03 *//0b00000011*

#define C\_DISC 0x0B *//0b00001011*

#define C\_UA 0x07 *//0b00000111*

#define C\_RR(R) ((R<<5) + 1) *//R = Nr = 0, 1*

#define C\_REJ(R) ((R<<5) + 5) *//R = Nr = 0, 1*

#define SU\_SEQ(C) ((C & 0b00100000) >> 5) *//extract sequence number from control field*

*//Information frame control*

#define C(S) (S<<1) *//S = Ns*

#define I\_SEQ(C) (C>>1)

*//#define DEBUG*

#ifdef DEBUG

#define DEBUG\_PRINT(str, ...) printf(str, ##\_\_VA\_ARGS\_\_)

#else

#define DEBUG\_PRINT(str, ...)

#endif

*//testing controls - uncomment to use*

*/\* #define testing*

*#define test\_data\_corruption 4 //BCC2*

*#define test\_missing\_su\_frame 4 //BCC1*

*\*/*

#endif

**test.c**

#include "linklayer.h"

#include "transmitter.h"

#include "receiver.h"

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

if (argc < 3){

printf("usage: progname /dev/ttySxx tx|rx \n");

exit(1);

}

*//test stuffing*

*//stuffingTests();*

printf("%s %s\n", argv[1], argv[2]);

fflush(stdout);

struct linkLayer ll;

sprintf(ll.serialPort, "%s", argv[1]);

ll.baudRate = 9600;

ll.numTries = 3;

ll.timeOut = 3;

if(strcmp(argv[2], "tx") == 0){ *//tx mode*

printf("tx mode\n");

ll.role = TRANSMITTER;

if (llopen(ll) == 1){

char text[] = "Ola netedu!";

llwrite(text, 12);

}

}

else if(strcmp(argv[2], "rx") == 0){ *//rx mode*

printf("rx mode\n");

ll.role = RECEIVER;

if(llopen(ll)==0){

char text[MAX\_PAYLOAD\_SIZE];

llread(text);

}

}

else

printf("bad parameters\n");

return 0;

}

void stuffingTests(){

u\_int8\_t data[6] = {0x00, 0x01, 0x7E, 0x44, 0x7D, 0x74};

u\_int8\_t \*newData;

int size = 0;

newData = prepareInfoFrame(data, 6, &size, 1);

for (int i = 0; i < size; i++)

printf("0x%02x ", newData[i]);

printf("\n");

newData = byteStuffing(data, 6, &size);

for (int i = 0; i < size; i++)

printf("0x%02x ", newData[i]);

printf("\n");

newData = byteDestuffing(newData, size, &size);

for (int i = 0; i < size; i++)

printf("0x%02x ", newData[i]);

printf("\n");

return;

}

**Anexo II - Testes de Baud rate**

Durante estes testes foram analisadas 4 taxas de símbolos diferentes: 19200, 38400, 57600 e 115200 *baud*. É relevante apontar que em laboratório, *baud rates* fora destes valores eram instáveis, resultando na perda da ligação durante a transferência de dados. A seguir, nas figuras 1 e 2, encontram-se os tempos médio de respostas das duas principais funções de comunicação de dados.

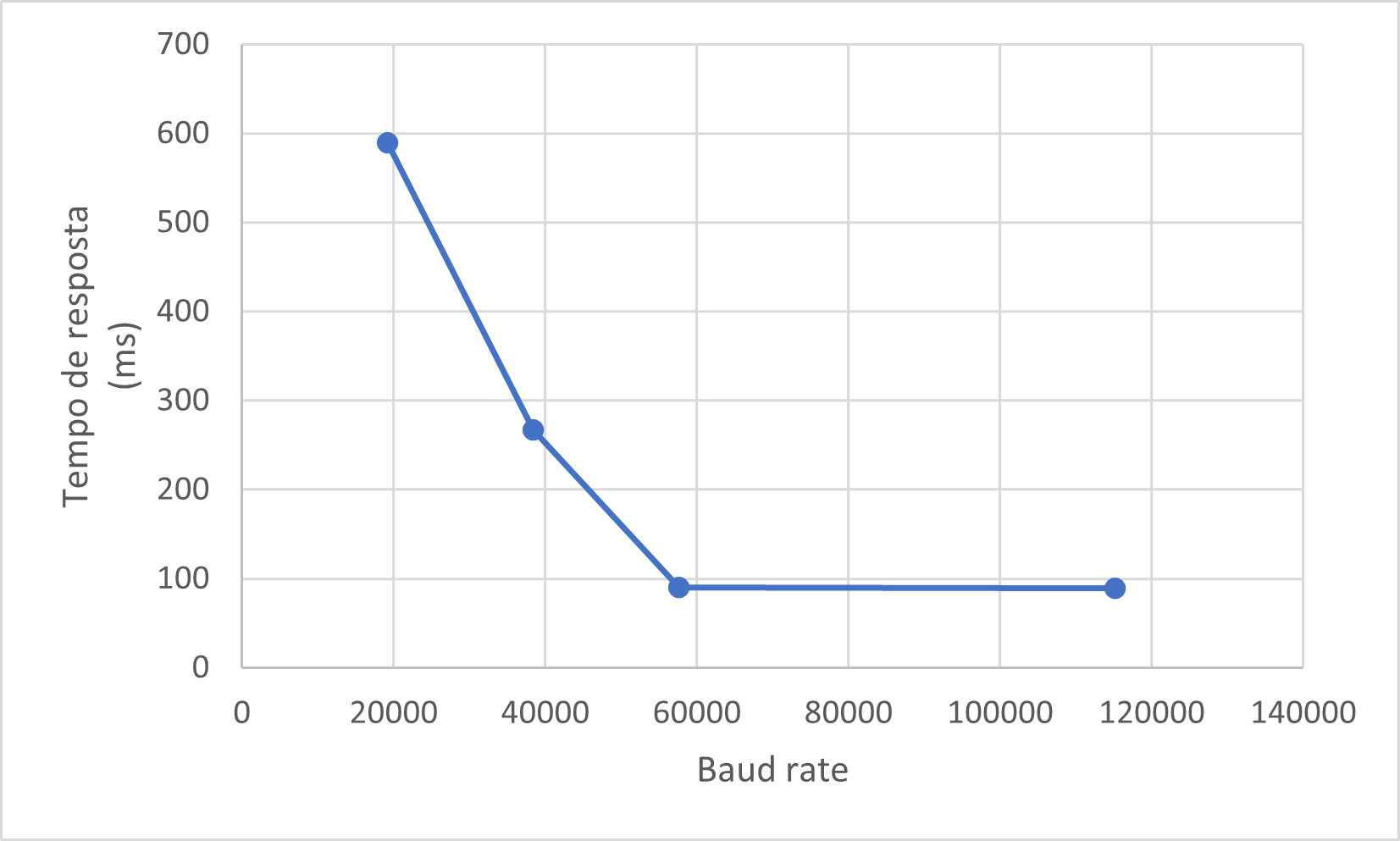
****

Figura 3 - Tempo médio de resposta do llread() para diferentes baud rates

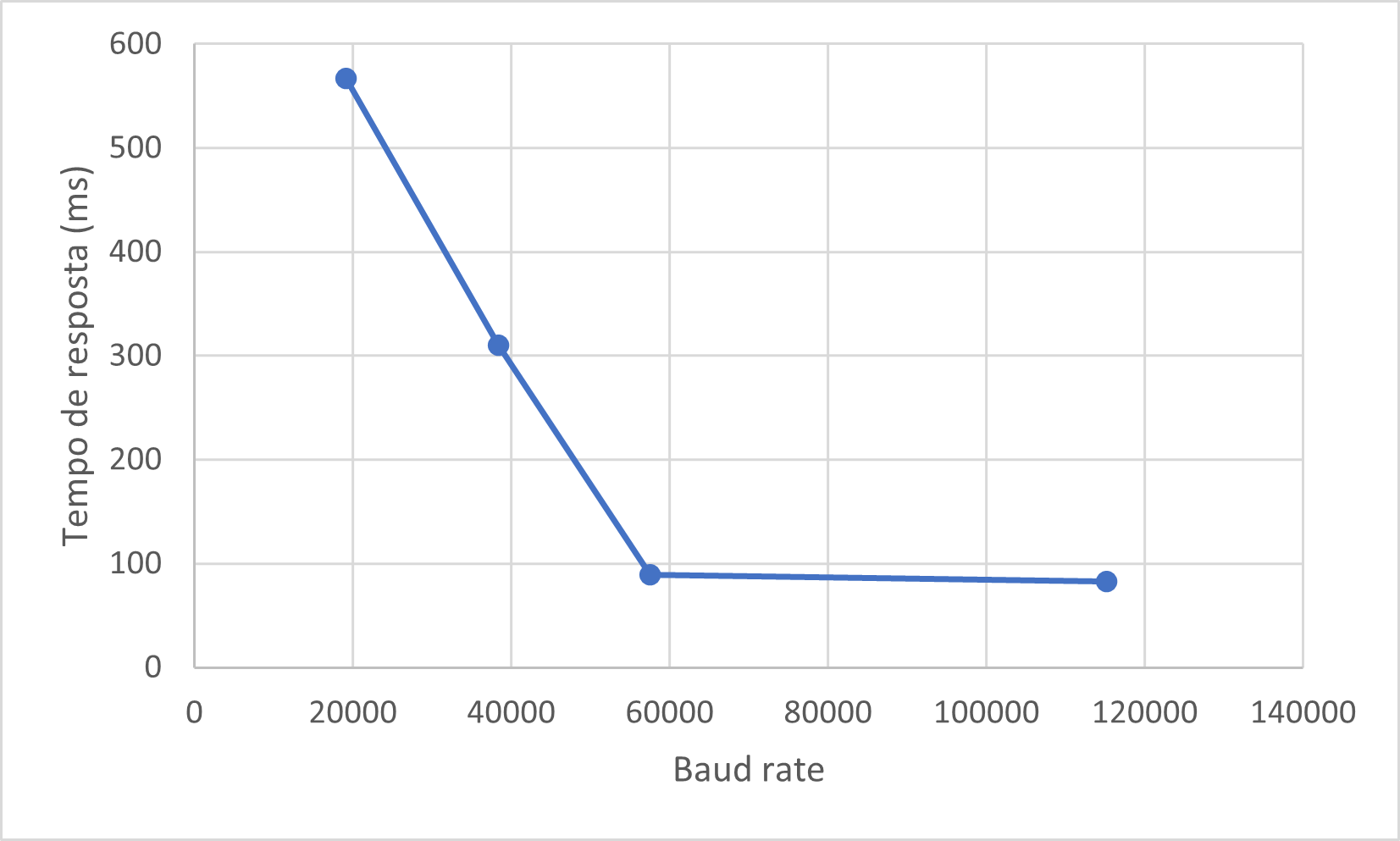


Figura 4 - Tempo médio de resposta do llwrite() para diferentes baud rates