1º Trabalho Laboratorial

Relatório



Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

Beatriz de Henriques Martins – up201502858

Nádia de Sousa Varela Carvalho – up201208223

Faculdade de Engenharia da Universidade do Porto

Rua Roberto Frias, sn, 4200-465 Porto, Portugal

31 Outubro 2017

Índice

[1 Sumário 3](#_Toc497756980)

[2 Introdução 3](#_Toc497756981)

[3 Arquitetura e Estrutura do código 4](#_Toc497756982)

[3.1 Data Link – Ligação de dados 4](#_Toc497756983)

[3.2 App Link – Aplicação 4](#_Toc497756984)

[4 Casos de uso principais 5](#_Toc497756985)

[5 Protocolo de ligação lógica 5](#_Toc497756986)

[5.1 llopen() e llclose() 5](#_Toc497756987)

[5.2 llread() e llwrite() 5](#_Toc497756988)

[6 Protocolo de aplicação 6](#_Toc497756989)

[6.1 sendData() 6](#_Toc497756990)

[6.2 receiveData() 6](#_Toc497756991)

[7 Validação 6](#_Toc497756992)

[8 Eficiência do protocolo de ligação de dados 7](#_Toc497756993)

[9 Conclusões 8](#_Toc497756994)

[Anexo – Código fonte 9](#_Toc497756995)

[A1. dataLink.h 9](#_Toc497756996)

[A2. dataLink.c 11](#_Toc497756997)

[A3. AppLink.h 23](#_Toc497756998)

[A4. AppLink.c 24](#_Toc497756999)

[A5. receiver.c 31](#_Toc497757000)

[A6. sender.c 32](#_Toc497757001)

# Sumário

Este trabalho foi realizado no âmbito da cadeira *Redes de Computadores* com o objetivo de aplicar todos os conhecimentos lecionados nas aulas teóricas sobre o *Protocolo de Ligação de Dado*. O trabalho em si consiste na transferência de dados através de uma porta de série.

O projeto foi concluído. No entanto, não está completo, uma vez que a aplicação só é capaz de enviar o ficheiro quando não ocorrem erros.

# Introdução

O objetivo principal deste trabalho é a implementação de um determinado protocolo para transferência de dados, especificado no guião do projeto, através de uma porta de série, uma das formas mais básicas para transferência de dados.

O objetivo do relatório é demonstrar como a parte teórica foi aplicada ao projeto, uma vez que na apresentação/demonstração deste não foi muito visível.

A organização do relatório será a seguinte:

* **Introdução** – breve indicação dos objetivos do projeto e do relatório;
* **Arquitetura e Estrutura do código** – explicação das API’s, principais estruturas de dados, funções e a sua relação com a arquitetura;
* **Casos principais de uso** –identificação dos principais casos de uso e sequências de chamadas
* **Protocolo de ligação de lógica** –identificação dos principais aspetos funcionais e descrição da estratégia de implementação
* **Protocolo de aplicação** –identificação dos principais aspetos funcionais e descrição da estratégia de implementação
* **Validação** – descrição dos testes efetuados com apresentação com apresentação quantificada
* **Eficiência do protocolo de ligação de dados** –caraterização estatística da  eficiência do protocolo
* **Conclusões** –síntese da informação apresentada anteriormente reflexão sobre os objetivos de aprendizagem alcançados

# Arquitetura e Estrutura do código

O nosso trabalho foi construído com base no princípio da independência entre camadas, ou seja, o tornar o programa o mais modal possível. Para uma melhor implementação optamos por dividir em 3 parte:

## Data Link – Ligação de dados

Os ficheiros dataLink.h e dataLink.c representam o nível mais baixo da aplicação. É através deste que é estabelecida a ligação entre a porta de série e aplicação. Assim sendo, podemos concluir que é este que configura, inicia e termina a ligação e também o que escreve e lê os dados da porta de série, fazendo o tratamento dos erros, stuffing e destuffing.

int open\_serial\_port(char\* port, int whoCalls);

int open\_receiver(char\* port);

int open\_sender(char\* port);

int stuffing(unsigned char\* frame, int length);

int llopen(char\* port, int whoCalls);

int llread(char\* port, int whoCalls);

int llwrite(int fd, char\* buffer, int length);

int llclose(int fd, int whoCalls);

## App Link – Aplicação

Os ficheiros *AppLink.h* e *AppLink.c* representam o nível intermédio da aplicação, ou seja é o nível que faz a ponte de ligação entre os níveis mais baixo (*dataLink.h* e *dataLink.c*) e o mais alto (*interface*). Esta é também responsável pela comunicação da camada lógica com a interface do utilizador, é nesta camada que é feita a comunicação e transferência de dados, de acordo com os parâmetros que o utilizador define.

typedef struct{

int fileDescriptor;

int status;

}app;

int connection(char\* terminal, int whoCalls);

int receiveData();

int sendControlPackage(int state, FileInfo file, unsigned char \*controlPacket);

int sendDataPackage(unsigned char \*dataPacket, FILE \*fp, int sequenceNumber, int \*length);

int sendData();

# Casos de uso principais

A aplicação desenvolvida necessita apenas de um parâmetro, 0 ou 1 que representa a porta série a utilizar (/dev/ttyS0 ou (/dev/ttyS1). Depois de executar, a aplicação estabelece a comunicação e, no modo emissor, pede ao utilizador o ficheiro a transferir, o qual envia para o recetor. Durante o estabelecimento da comunicação, se estiver em modo recetor, espera que algum emissor inicie uma ligação.

# Protocolo de ligação lógica

O protocolo de ligação implementado tem como principais aspetos, pela seguinte ordem:

1. Configuração da porta de série;
2. Estabelecimento de ligação pela porta de série;
3. Transferência de dados pela porta de série (stuffing/destuffing);
4. Tratamento de erros durante a ligação.

Foi necessário implementar as seguintes funções:

* llopen();
* llclose();
* llread();
* llwrite().

## llopen() e llclose()

Estas são as funções responsáveis por iniciar e terminar a ligação pela porta de série. Para isso, a função llopen começa por testar quem a está a chamar, se o SENDER ou o RECEIVE e, a partir daí, chamar as funções adequadas para abrir a porta série e alterar as configurações da mesma para as pretendidas.

Na função llopen do SENDER, open\_sender(), é criada uma trama SET que é enviada para o recetor, entrando de seguida num ciclo de leitura para receber a resposta pretendida, a trama UA. Enquanto não a recebe, ativa um alarme de duração 3 segundos e, sempre que o alarme for desencadeado, conta como um timeout, tentando enviar outra vez a trama. Se exceder o número de timeouts máximo, a função termina com estado de erro, indicando que o llopen não conseguiu estabelecer comunicação e para tentar outra vez mais tarde. Se a aplicação for chamada como recetor, fica à espera até receber a trama SET o que quando sucede, envia a trama UA, estabelecendo corretamente a comunicação.

A função llclose, por sua vez, tenta terminar a ligação do lado do emissor, enviando uma trama DISC que espera pela resposta do recetor, que é outra trama DISC. Ao receber, envia uma trama UA, de modo a informar o recetor que recebeu a sua intenção de por término à comunicação. No lado do recetor, este espera pelo primeiro DISC referido acima e envia a resposta (trama DISC) e depois espera pela trama UA para e repõe as configurações anteriores da porta série, finalizando a ligação.

## llread() e llwrite()

A função llwrite é responsável por enviar os dados ao recetor, recebendo um parâmetro que pode ser um pacote de dados ou de controlo para o qual constrói a trama de informação correspondente que é posteriormente enviada ao recetor. Depois de enviar, o processo emissor espera pela resposta do recetor e, no caso de não a receber, volta a enviar dentro do número de tentativas estipulado (neste caso 3). Neste caso, a função termina com erro. A resposta recebida pode ser do tipo RR que indica o sucesso da transferência e determina a continuação da mesma, enviando a trama com o próximo número da sequência desde que não repetido, ou do tipo REJ que especifica a existência de erro de leitura e determina o reenvio da mesma trama.

A função llread, lê a trama de informação enviada pelo emissor e, após realizar o destuffing do pacote contido pela trama, é processada a informação, através de uma nova máquina de estados, de modo a que sempre que um byte errado é lido, volta ao estado original. Caso a trama recebida seja inicial o ficheiro é aberto para escrita de informação. Em resposta envia uma trama do tipo RR em caso de sucesso e REJ em caso de erro.

# Protocolo de aplicação

O protocolo de aplicação implementado tem como principais aspetos, pela seguinte ordem:

1. Geração e transferência dos pacotes de controlo de dados;
2. Leitura e escrita do ficheiro transferir.

## sendData()

A função sendData tem como responsabilidade enviar os pacotes de controlo, interagir com o utilizador para perguntar qual o ficheiro a enviar, abrir, ler e enviar o mesmo nas várias tramas. Para tal utiliza as funções sendControlPackage() para, como o nome indica, enviar os pacotes de controlo de dados, quer o inicial, quer o final, e da função sendDataPackage() para escrever e enviar a informação do ficheiro.

## receiveData()

A função receiveData lê as tramas enviadas pelo emissor e, utilizando a função processingDataPacket processa o pacote da trama de informação recebida, caso seja inicial ou final, como pacote de controlo ou, nos outros casos, como pacote de dados. Neste caso, escreve a informação recebida no ficheiro de destino. Como resposta, indica qual a resposta a ser dada ao emissor, inicio, fim, REJ ou RR.

# Validação

Para verificar a robustez da ligação, foram aplicados os seguintes testes:

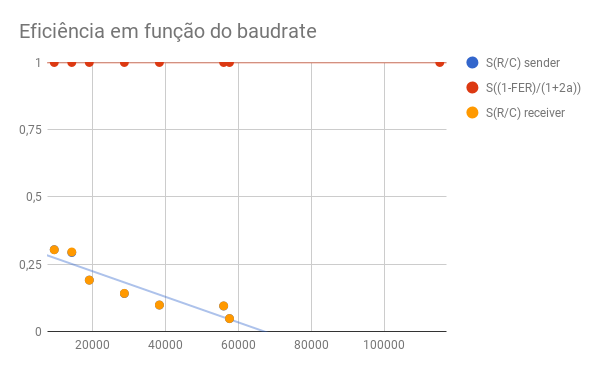
1. Enviar um ficheiro.
2. Enviar um ficheiro e introduzir erros na ligação com um cabo de cobre.

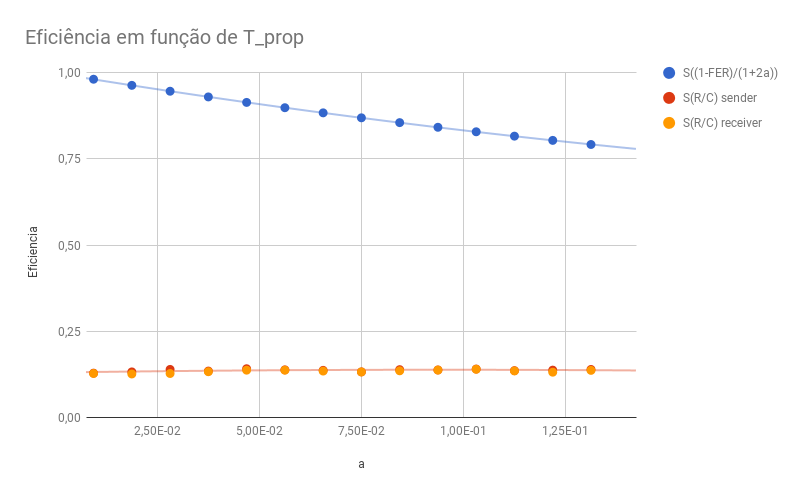
A aplicação foi capaz de superar estes testes, verificando-se isto tanto pelos bytes do ficheiro estarem corretos, assim como pela demostração no ecrã do estado de envio.

No entanto, quando se realizaram os testes de envio de um ficheiro com interrupção da ligação da porta série, o processo de timeout não ocorria de maneira correta e o ficheiro não era completamente enviado.

# Eficiência do protocolo de ligação de dados

A eficiência do protocolo de dados foi testada fazendo variar o baudrate (gráfico 1) (entre valores considerados standard: 9600, 14400, 19200, 28800, 38400, 56000, 57600, 115200) e o tempo de propagação (gráfico 2), adicionando quinze intervalos de 0,001s ao tempo de propagação calculado desde 0,0005.





# Conclusões

Por fim, concluímos que nem todos os objetivos propostos foram concluídos, uma vez que nem sempre o envio do ficheiro foi possível quando existiam erros de ligação da porta série. A nível geral, tivemos muita dificuldade em perceber como aplicar os conceitos explicados nas aulas e no guião, sentindo que este nem sempre era muito útil a responder às dúvidas que advinham das experiências realizadas a nível de código. Assim, e embora no final possamos dizer que compreendemos os conceitos necessários à realização do trabalho, bem como o sistema de independência de camadas, aonde demonstrámos a que a camada de aplicação se serve da camada de dados mas é independente do seu modo de agir, percebemos que a nossa aplicação necessita de muitas melhorias, especialmente na correção dos erros gerados pela abertura e fecho da ligação durante o processo de transferência.

# Anexo – Código fonte

## A1. dataLink.h

#ifndef DATA\_LINK

#define DATA\_LINK

#include <stdlib.h>

#include <stdio.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>

#define BAUDRATE B38400

#define SENDER 0

#define RECEIVER 1

#define A\_SENDER 0x03

#define A\_RECEIVER 0x01

#define C\_SET 0x03

#define C\_DISC 0x0B

#define C\_UA 0x07

#define C\_RR 0x05

#define C\_RR1 0x85

#define C\_REJ 0x01

#define ESC 0x7D

#define ESC\_HIDE\_BYTE 0x5D

#define FLAG 0x7E

#define FLAG\_HIDE\_BYTE 0x5E

#define FLD\_CTRL 2

#define N\_OF\_SEQ\_0 0x00

#define N\_OF\_SEQ\_1 0x40

#define DATA\_BYTE 1

#define START\_BYTE 2

#define END\_BYTE 3

/\* complete \*/

void retry();

/\* complete \*/

void atende();

/\* complete \*/

int open\_serial\_port(char\* port, int whoCalls);

/\* complete \*/

int open\_receiver(char\* port);

/\* complete \*/

int open\_sender(char\* port);

/\* complete \*/

void createControlFrame(char\* frame, char controlByte, int whoCalls);

/\* complete

maquina de estados \*/

char readingArrayStatus(int fd);

/\* complete \*/

void insertValueAt(int index, int value, unsigned char\* frame, int lenght);

/\* complete \*/

void shiftBack(int index, char\* frame, int frameSize);

/\* complete \*/

unsigned char getBCC2(unsigned char\* frame, unsigned int length);

/\* complete \*/

int stuffing(unsigned char\* frame, int length);

/\* complete \*/

int destuffing(char\* frame);

/\* complete \*/

int processingDataFrame(char\* frame);

/\* complete \*/

int readingFrame(int fd, char\* frame);

/\* complete \*/

int resetSettings(int fd);

/\* complete \*/

int llopen(char\* port,int whoCalls);

/\* complete \*/

int llread(int fd, char\* buffer);

/\* complete \*/

int llwrite(int fd, unsigned char\* buffer, int length);

/\* complete \*/

int llclose(int fd, int whoCalls);

#endif

## A2. dataLink.c

#include "dataLink.h"

int flag = 1;

int frameSize = 0;

int nTries = 3;

int outTime = 3;

int nTOuts = 0;

int tries = 0;

unsigned char frame[255];

int fdW;

struct termios oldtio, newtio;

/\*\*

\* Functions dealing with alerts

\*/

void retry() {

alarm(outTime);

write(fdW, frame, frameSize);

nTOuts++;

if (tries == nTries) {

printf(

"\n\nTIMEOUT : Lost connection to receiver\n Number of tries : %d\n\n",

nTries);

exit(1);

}

tries++;

printf("\n\nTrying to connect to receiver\nTry number : %d\n\n", tries);

}

/\*\*

\* Connection Lost

\*/

void timeout() {

printf("TIMEOUT : Connection lost, try again later\n");

exit(1);

}

/\*\*

\* atende alarme

\*/

void atende() {

printf("alarme # %d\n", tries);

flag = 1;

tries++;

nTOuts++;

}

/\*\*

\* Functions

\*/

int open\_serial\_port(char \*port, int whoCalls) {

printf("open\_serial\_port\n");

int fd;

char serialPort[10] = "/dev/ttyS";

strcat(serialPort, port);

if ((strcmp("/dev/ttyS0", serialPort) != 0) &&

(strcmp("/dev/ttyS1", serialPort) != 0)) {

printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS1\n");

exit(1);

}

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

if (fd < 0) {

perror(serialPort);

exit(-1);

}

if (tcgetattr(fd, &oldtio) == -1) { /\* save current port settings \*/

perror("tcgetattr");

exit(-1);

}

bzero(&newtio, sizeof(newtio));

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

newtio.c\_iflag = IGNPAR;

newtio.c\_oflag = 0;

/\* set input mode (non-canonical, no echo,...) \*/

newtio.c\_lflag = 0;

if (whoCalls == SENDER) {

newtio.c\_cc[VTIME] = 1;

newtio.c\_cc[VMIN] = 0;

} else if (whoCalls == RECEIVER) {

newtio.c\_cc[VTIME] = 0;

newtio.c\_cc[VMIN] = 1;

}

tcflush(fd, TCIOFLUSH);

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

perror("tcsetattr");

exit(-1);

}

return fd;

}

/\*\*

\* Function to Open Receiver and receive trama SET and send trama UA

\* @method open\_receiver

\* @param port 0 or 1 for ttyS0 or ttyS1

\* @return file descriptor for serial port opened

\*/

int open\_receiver(char \*port) {

printf("open\_receiver\n");

int fd;

(void)signal(SIGALRM, timeout);

fd = open\_serial\_port(port, RECEIVER);

// RECEIVE TRAMA SET

alarm(outTime);

char controlByte = readingArrayStatus(fd);

alarm(0);

// WRITE TRAMA UA

char tramaUA[5] = {FLAG, A\_SENDER, C\_UA, A\_SENDER ^ C\_UA, FLAG};

int res;

do {

res = write(fd, &tramaUA, sizeof(tramaUA));

} while (res == 0);

return fd;

}

/\*\*

\* Function to Open Sender and send trama SET and receive trama UA

\* @method open\_sender

\* @param port 0 or 1 for ttyS0 or ttyS1

\* @return file descriptor for serial port opened

\*/

int open\_sender(char \*port) {

printf("open\_sender\n");

char buffer[5];

char tramaUA[5];

int res = 0;

char controlByte = NULL;

int fd = open\_serial\_port(port, SENDER);

// CREATE AND WRITE TRAMA SET

createControlFrame(buffer, C\_SET, SENDER);

do {

res = write(fd, buffer, 5);

// READ TRAMA UA

alarm(outTime);

controlByte = readingArrayStatus(fd);

alarm(0);

} while (tries < nTries && flag == 1);

return fd;

}

/\*\*

\* Create Control Frame SET, DISC, etc

\* @method createControlFrame

\* @param frame pointer to frame to create

\* @param controlByte type of frame

\* @param whoCalls who calls the function: SENDER or RECEIVER

\*/

void createControlFrame(char \*frame, char controlByte, int whoCalls) {

int isAnswer = 0;

frame[0] = FLAG;

// verificação resposta ou cmd

if (controlByte == C\_UA || controlByte == C\_RR || controlByte == C\_REJ) {

isAnswer = 1;

}

if (whoCalls == SENDER) {

(isAnswer) ? (frame[1] = A\_RECEIVER) : (frame[1] = A\_SENDER);

} else {

(isAnswer) ? (frame[1] = A\_SENDER) : (frame[1] = A\_RECEIVER);

}

frame[2] = controlByte;

frame[3] = frame[1] ^ frame[2];

frame[4] = FLAG;

}

/\*\*

\* Status Machine for reading Array Status

\* @method readingArrayStatus

\* @param fd file descriptor

\* @return Control Camp

\*/

char readingArrayStatus(int fd) {

int state = 0;

unsigned char frame\_receive[5];

unsigned char var;

flag = 0;

while (state != 5 && flag == 0) {

int res = read(fd, &var, 1);

frame\_receive[state] = var;

if (res > 0) {

switch (state) {

case 0: {

if (var == FLAG) {

state = 1;

}

break;

}

case 1: {

if (var != FLAG) {

state = 2;

} else {

state = 1;

}

break;

}

case 2: {

if (var != FLAG) {

state = 3;

} else {

state = 1;

}

break;

}

case 3: {

if (var == (frame\_receive[2] ^ frame\_receive[1])) {

state = 4;

} else {

printf("Damage package\n");

return -1;

}

break;

}

case 4: {

if (var != FLAG) {

state = 0;

} else {

state = 5;

return frame\_receive[2];

}

break;

}

}

}

}

}

/\*\*

\* Shift forward method in which we insert value on frame at index

\* @method insertValueAt

\* @param index

\* @param value

\* @param frame

\* @param length

\*/

void insertValueAt(int index, int value, unsigned char \*frame, int length) {

int i;

for (i = length - 1; i >= index; i--) {

frame[i + 1] = frame[i];

}

frame[i] = value;

}

/\*\*

\* Shift Back Method

\* @method shiftBack

\* @param index

\* @param frame

\* @param frameSize

\*/

void shiftBack(int index, char \*frame, int frameSize) {

int over = 0;

index++;

do {

frame[index] = frame[index + 1];

index++;

if (frame[index] == FLAG) {

over = 1;

}

} while (!over);

}

/\*\*

\* Get Block Check Character 2 of Frame

\* @method getBCC2

\* @param frame

\* @param length

\* @return BCC2

\*/

unsigned char getBCC2(unsigned char \*frame, unsigned int length) {

unsigned char BCC = 0;

unsigned int i = 0;

for (; i < length; i++) {

BCC ^= frame[i];

}

return BCC;

}

/\*\*

\* Stuffing of frame

\* @method stuffing

\* @param frame

\* @param length

\* @return size of frame after stuffing

\*/

int stuffing(unsigned char \*frame, int length) {

int i;

for (i = 1; i < length - 1; i++) {

if (frame[i] == FLAG) {

frame[i] = ESC;

i++;

insertValueAt(i, 0, frame, length);

frameSize++;

frame[i] = FLAG\_HIDE\_BYTE;

}

if (frame[i] == ESC) {

i++;

insertValueAt(i, 0, frame, length);

frameSize++;

frame[i] = ESC\_HIDE\_BYTE;

}

}

return i;

}

/\*\*

\* Destuffing of frame

\* @method destuffing

\* @param frame

\* @return size of frame after destuffing

\*/

int destuffing(char \*frame) {

int over = 0;

int i = 1;

while (!over) {

if (frame[i] == FLAG) {

over = 1;

} else if (frame[i] == ESC && frame[i + 1] == FLAG\_HIDE\_BYTE) {

frame[i] = FLAG;

shiftBack(i, frame, 0);

} else if (frame[i] == ESC && frame[i + 1] == ESC\_HIDE\_BYTE) {

shiftBack(i, frame, 0);

}

i++;

}

return i;

}

/\*\*

\* Processing Data Frame

\* @method processingDataFrame

\* @param frame

\* @return 0 or -1 in case of success or fail

\*/

int processingDataFrame(char \*frame) {

printf("processingDataFrame\n");

if (frame[0] != FLAG) {

return -1;

}

if (frame[1] != A\_SENDER) {

return -1;

}

if (frame[2] != N\_OF\_SEQ\_0 && frame[2] != N\_OF\_SEQ\_1) {

return -1;

}

if (frame[3] != (frame[1] ^ frame[2])) {

printf("BCC1 recebido: %X\n", frame[3]);

printf("BCC1 esperado: %X\n", frame[1] ^ frame[2]);

printf("ERRO BCC1\n");

return -1;

}

return 0;

}

/\*\*

\* Reading Frame for llread

\* @method readingFrame

\* @param fd file descriptor for read

\* @param frame

\* @return size of frame read

\*/

int readingFrame(int fd, char \*frame) {

unsigned char oneByte;

int state = 0;

int over = 0;

int i = 0;

int j;

//(void)signal(SIGALRM, timeout);

while (!over) {

// alarm(outTime);

read(fd, &oneByte, 1);

// alarm(outTime);

switch (state) {

case 0:

if (oneByte == FLAG) {

frame[i] = oneByte;

i++;

state = 1;

}

break;

case 1:

if (oneByte != FLAG) {

frame[i] = oneByte;

i++;

state = 2;

}

break;

case 2:

if (oneByte != FLAG) {

frame[i] = oneByte;

i++;

state = 3;

}

break;

case 3:

if (oneByte != FLAG) {

frame[i] = oneByte;

i++;

state = 4;

}

break;

case 4:

if (oneByte != FLAG) {

frame[i] = oneByte;

i++;

} else if (oneByte == FLAG) {

frame[i] = oneByte;

i++;

over = 1;

}

break;

default:

break;

}

}

return i;

}

/\*\*

\* Reset Settings and Close File Descriptor

\* @method resetSettings

\* @param fd

\* @return 0 or -1 in case of success or fail

\*/

int resetSettings(int fd) {

printf("resetSettings\n");

if (close(fd)) {

return -1;

printf("Error closing terminal file descriptor.\n");

}

return 0;

}

/\*\*

\* LLOPEN

\* @method llopen

\* @param port 0 or 1 for ttyS0 or ttyS1

\* @param whoCalls SENDER or RECEIVER

\* @return file descriptor

\*/

int llopen(char \*port, int whoCalls) {

printf("\nllopen\n");

if (whoCalls == RECEIVER) {

return open\_receiver(port);

} else if (whoCalls == SENDER) {

return open\_sender(port);

} else {

return -1;

}

}

/\*\*

\* LLREAD

\* @method llread

\* @param fd

\* @param buffer

\* @return size after destuffing

\*/

int llread(int fd, char \*buffer) {

int ret, sizeAfterDestuffing;

readingFrame(fd, buffer);

sizeAfterDestuffing = destuffing(buffer);

if (ret == 0) {

ret = sizeAfterDestuffing;

}

return ret;

}

/\*\*

\* LLWRITE

\* @method llwrite

\* @param fd

\* @param buffer

\* @param length

\* @return number of rejections

\*/

int llwrite(int fd, unsigned char \*buffer, int length) {

fdW = fd;

int sequenceNumber = buffer[length - 1];

int nRej = 0;

length--;

frame[0] = FLAG;

frame[1] = A\_SENDER;

frame[2] = sequenceNumber;

frame[3] = frame[1] ^ frame[2];

int i;

for (i = 0; i < length; i++) {

frame[i + 4] = buffer[i];

}

frame[length + 4] = getBCC2(buffer, length);

tries = 0;

// (void)signal(SIGALRM, retry);

stuffing(frame, length + 6);

frame[length + 5] = FLAG;

unsigned char temp[5];

i = 0;

do {

if (i > 0) {

nRej++;

}

int j;

// alarm(outTime);

write(fd, frame, sizeof(frame));

read(fd, temp, 5);

// alarm(outTime);

i++;

} while (temp[2] == C\_REJ);

return nRej;

}

/\*\*

\* LLCLOSE

\* @method llclose

\* @param fd

\* @param whoCalls SENDER or RECEIVER

\* @return 0 or -1 in case of success or fail

\*/

int llclose(int fd, int whoCalls) {

printf("\nllclose\n");

char frame[5];

int res\_resetSet = 0, res = 0;

tries = 0;

(void)signal(SIGALRM, atende);

if (whoCalls == SENDER) {

createControlFrame(frame, C\_DISC, whoCalls);

if ((res = write(fd, &frame, sizeof(frame))) != 5) {

printf("llclose :: Couldn't send frame DISC on llclose().\n");

res\_resetSet = resetSettings(fd);

return -1;

}

alarm(outTime);

if ((res = readingArrayStatus(fd)) != -1) {

alarm(0);

tries = 0;

}

char tramaUA[5] = {FLAG, A\_SENDER, C\_UA, C\_UA ^ A\_SENDER, FLAG};

if (res = write(fd, &tramaUA, sizeof(tramaUA)) != 5) {

printf("llclose :: Couldn't send frame UA on llclose().\n");

res\_resetSet = resetSettings(fd);

return -1;

}

sleep(1);

res\_resetSet = resetSettings(fd);

if (res\_resetSet == 0) {

printf("llclose :: Connection successfully closed.\n");

}

} else if (whoCalls == RECEIVER) {

alarm(outTime);

res = 0;

if ((res = readingArrayStatus(fd)) != -1) {

alarm(0);

tries = 0;

}

sleep(1);

createControlFrame(frame, C\_DISC, whoCalls);

if ((res = write(fd, frame, sizeof(frame))) != 5) {

printf("llclose :: Couldn't send frame DISC on llclose().\n");

res\_resetSet = resetSettings(fd);

return -1;

}

res = 0;

alarm(outTime);

if ((res = readingArrayStatus(fd)) != -1) {

printf("UA %d\n", res);

alarm(0);

tries = 0;

}

res\_resetSet = resetSettings(fd);

if (res\_resetSet == 0) {

printf("llclose :: Connection successfully closed.\n");

}

}

printf("Number of timeouts : %d\n", nTOuts);

return 0;

}

## A3. AppLink.h

#ifndef APP\_LINK

#define APP\_LINK

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <termios.h>

#include <sys/stat.h> //biblioteca que permite obter o mode de um file

#include <sys/types.h>

#include <errno.h>

#include "dataLink.h"

#define DATA\_CTRL\_PACKET 1

#define START\_CTRL\_PACKET 2

#define END\_CTRL\_PACKET 3

#define DATA\_SIZE 100

#define MAX\_SIZE 255

#define FILE\_SIZE\_BYTE 0

#define FILE\_NAME\_BYTE 1

#define FILE\_PERMISSIONS\_BYTE 2

#define PACKET\_SIZE 256

#define PACKET\_HEADER\_SIZE 4

#define PACKET\_DATA\_SIZE PACKET\_SIZE - PACKET\_HEADER\_SIZE

#define FILE\_SIZE 10968

typedef struct{

int fileDescriptor; /\*Descritor correspondente à porta série\*/

int status; /\*TRANSMITTER | RECEIVER\*/

}app;

app application;

typedef struct {

unsigned int size;

char filename[MAX\_SIZE];

} FileInfo;

off\_t getFileSize(char\* trama, int lenghtTrama);

char\* getFileName(char\* trama, int lenghtTrama);

char\* getFileName(char\* trama, int lenghtTrama);

int connection(char\* terminal, int whoCalls);

int receiveData();

int sendControlPackage(int state, FileInfo file, unsigned char \*controlPacket);

int processingDataPacket(unsigned char \*packet, int length, FileInfo \*file, int fp);

int sendDataPackage(unsigned char \*dataPacket, FILE \*fp, int sequenceNumber, int \*length);

int sendData();

int getFile(char \*filepath);

int fileSize(FILE \*fd);

#endif

## A4. AppLink.c

#include "AppLink.h"

unsigned char prevDataCounter = 0;

int dataSize = 100;

/\*\*

\* Stablishing Connection

\* @method connection

\* @param terminal 0 or 1 for ttyS0 or ttyS1

\* @param whoCalls SENDER or RECEIVER

\* @return file descriptor

\*/

int connection(char \*terminal, int whoCalls) {

if (whoCalls != SENDER && whoCalls != RECEIVER) {

perror("AppLink :: connection() :: Invalid status.");

return -1;

}

application.status = whoCalls;

int caller = NULL;

if ((strcmp("0", terminal) == 0 || strcmp("1", terminal) == 0) &&

whoCalls == SENDER) {

caller = SENDER;

} else if ((strcmp("0", terminal) == 0 || strcmp("1", terminal) == 0) &&

whoCalls == RECEIVER) {

caller = RECEIVER;

} else {

perror("AppLink :: connection() :: terminal failed");

}

application.fileDescriptor = llopen(terminal, whoCalls);

if (application.fileDescriptor < 0) {

perror("AppLink :: connection() :: llopen failed");

return -1;

}

return application.fileDescriptor;

}

/\*\*

\* Receive Data for receiver

\* @method receiveData

\* @return -1 or 0 in case of success or fail

\*/

int receiveData() {

unsigned char frame[255];

int over = 0;

FileInfo file;

file.size = 0;

int ret;

int fp;

int packagesLost = 0;

int nRejs = 0;

int frameSize;

printf("AppLink :: Start reading\n");

while (!over) {

frameSize = llread(application.fileDescriptor, frame);

if (frameSize == -1) {

packagesLost++;

ret = -1;

} else {

ret = processingDataPacket(frame, frameSize, &file, fp);

if (ret == START\_CTRL\_PACKET) {

fp = open(file.filename, O\_CREAT | O\_WRONLY);

if (fp == -1) {

printf("AppLink :: Receive Data :: Could not open file %s\n",

file.filename);

return -1;

}

}

if (ret == END\_CTRL\_PACKET) {

over = 1;

}

}

if (ret == -1) {

write(application.fileDescriptor, C\_REJ, 5);

nRejs++;

} else {

if (frame[FLD\_CTRL] == N\_OF\_SEQ\_0) {

write(application.fileDescriptor, C\_RR1, 5);

} else

write(application.fileDescriptor, C\_RR, 5);

}

}

// OPEN FILE

char command[50] = "xdg-open ";

strcat(command, file.filename);

system(command);

printf("\nFile read\n");

printf("\nPackages lost : %d\n", packagesLost);

printf("File size : %d\n", file.size);

printf("Number of rejections sent : %d\n", nRejs);

return 0;

}

/\*\*

\* Processing Data Packet

\* @method processingDataPacket

\* @param packet Packet to Process

\* @param length Length of above

\* @param file File to Read

\* @param fp File Descriptor for file

\* @return -1 in case of fail or status

\*/

int processingDataPacket(unsigned char \*packet, int length, FileInfo \*file,

int fp) {

int index = 4;

int numberOfBytes = 0;

int ret = 0;

int checkCounterData = 0;

if (packet[index] == START\_CTRL\_PACKET || packet[index] == END\_CTRL\_PACKET) {

ret = packet[index];

index += 2;

numberOfBytes = packet[index];

index++;

memcpy(&((\*file).size), packet + index, numberOfBytes);

index += numberOfBytes + 1;

numberOfBytes = packet[index];

index++;

memcpy(&((\*file).filename), packet + index, numberOfBytes);

index += numberOfBytes;

if (ret == START\_CTRL\_PACKET) {

printf("File name : %s\n", file->filename);

printf("File size : %d\n\n", file->size);

}

} else if (packet[index] == DATA\_CTRL\_PACKET) {

ret = packet[index];

index++;

int counterIndex = index;

index++;

unsigned int l2 = packet[index];

index++;

unsigned int l1 = packet[index];

index++;

unsigned int k = l2\*256 + l1;

unsigned char expect = getBCC2(packet + 4, k + 4);

if (packet[8 + k] != expect) {

printf("BCC received: %X\n", packet[8 + k]);

printf("BCC expected: %X\n", getBCC2(packet + 4, k + 4));

return -1;

}

if (k != (length - 10)) {

return -1;

}

if (prevDataCounter == 0) {

prevDataCounter = packet[counterIndex];

} else {

if (prevDataCounter == packet[counterIndex]) {

checkCounterData = 1;

printf("Repeated packet\n");

} else {

prevDataCounter = packet[counterIndex];

checkCounterData = 0;

}

}

int i;

for (i = 0; i < k; i++) {

if (checkCounterData == 0)

write(fp, &packet[index + i], 1);

}

}

return ret;

}

/\*\*

\* Send Control Package

\* @method sendControlPackage

\* @param state

\* @param file

\* @param controlPacket

\* @return size of control package

\*/

int sendControlPackage(int state, FileInfo file, unsigned char \*ctrlPacket) {

char fileSize[50];

memcpy(fileSize, &file.size, sizeof(file.size));

int ctrlPacketSize = 0;

ctrlPacket[0] = (unsigned char)state;

ctrlPacket[1] = (unsigned char)0;

ctrlPacket[2] = (unsigned char)strlen(fileSize);

ctrlPacketSize = 3;

unsigned int i;

for (i = 0; i < strlen(fileSize); i++) {

ctrlPacket[i + 3] = fileSize[i];

}

ctrlPacketSize += strlen(fileSize);

ctrlPacket[ctrlPacketSize] = (unsigned char)1;

ctrlPacketSize++;

ctrlPacket[ctrlPacketSize] = (unsigned char)strlen(file.filename);

ctrlPacketSize++;

for (i = 0; i < strlen(file.filename); i++) {

ctrlPacket[ctrlPacketSize + i] = file.filename[i];

}

ctrlPacketSize += strlen(file.filename);

return ctrlPacketSize;

}

/\*\*

\* Send Data Package

\* @method sendDataPackage

\* @param dataPacket

\* @param fp

\* @param sequenceNumber

\* @param length

\* @return 1

\*/

int sendDataPackage(unsigned char \*dataPacket, FILE \*fp, int sequenceNumber,

int \*length) {

unsigned char buffer[dataSize];

int ret;

ret = fread(buffer, sizeof(char), dataSize, fp);

if (ret == 0) {

return 0;

} else if (ret < 0) {

return -1;

}

\*length = ret + 4;

dataPacket[0] = DATA\_CTRL\_PACKET;

dataPacket[1] = sequenceNumber;

dataPacket[2] = 0;

dataPacket[3] = ret;

int j;

for (j = 0; j < dataSize; j++) {

dataPacket[4 + j] = buffer[j];

}

return 1;

}

/\*\*

\* Send Data for sender

\* @method sendData

\* @return 1

\*/

int sendData() {

char sequenceNumber = N\_OF\_SEQ\_0;

int dataCounter = 1;

FileInfo file;

getFile(file.filename);

FILE \*fp;

fp = fopen(file.filename, "rb");

if (fp == NULL) {

printf("Could not open file test.c");

return -1;

}

printf("opened file %s\n", file.filename);

(void)signal(SIGALRM, retry);

file.size = fileSize(fp);

printf("File size : %d\n", file.size);

char fileSize[50];

memcpy(fileSize, &file.size, sizeof(file.size));

int packetSize = 5 + strlen(file.filename) + strlen(fileSize);

unsigned char controlPacket[packetSize];

int controlPacketSize =

sendControlPackage(START\_CTRL\_PACKET, file, controlPacket);

controlPacket[controlPacketSize] = sequenceNumber;

controlPacketSize++;

llwrite(application.fileDescriptor, controlPacket, controlPacketSize);

int dataPacketSize;

unsigned char dataPacket[dataSize + 4];

int ret = 1;

int nRejs = 0;

int llwriteRet;

while (ret != 0) {

ret = sendDataPackage(dataPacket, fp, dataCounter, &dataPacketSize);

if (dataCounter < 255)

dataCounter++;

else

dataCounter = 1;

if (ret != 0) {

dataPacket[dataPacketSize] = sequenceNumber;

dataPacketSize++;

llwrite(application.fileDescriptor, dataPacket, dataPacketSize);

if (sequenceNumber == N\_OF\_SEQ\_0)

sequenceNumber = N\_OF\_SEQ\_1;

else

sequenceNumber = N\_OF\_SEQ\_0;

}

}

controlPacketSize = sendControlPackage(END\_CTRL\_PACKET, file, controlPacket);

controlPacket[controlPacketSize] = sequenceNumber;

controlPacketSize++;

llwriteRet =

llwrite(application.fileDescriptor, controlPacket, controlPacketSize);

nRejs += llwriteRet;

printf("\nFile sent\n\n");

printf("Number of rejections received : %d\n", nRejs);

return 1;

}

/\*\*

\* Get File Path from command Line

\* @method getFile

\* @param filepath

\* @return 1

\*/

int getFile(char \*filepath) {

printf("Enter file path : ");

scanf("%s", filepath);

return 1;

}

/\*\*

\* Get File Size

\* @method fileSize

\* @param fd pointer to File

\* @return size of file

\*/

int fileSize(FILE \*fd) {

struct stat s;

if (fstat(fileno(fd), &s) == -1) {

return (-1);

}

return (s.st\_size);

}

## A5. receiver.c

#include "AppLink.h"

#include "dataLink.h"

#include <stdio.h>

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

int fd;

char buffer[255];

if (argc < 2) {

perror("Invalid num of arguments");

return -1;

}

fd = connection(argv[1], RECEIVER);

printf("llopen complete\n\n");

if (fd > 0)

receiveData();

llclose(fd, RECEIVER);

printf("llclose complete\n");

return 0;

}

## A6. sender.c

#include "AppLink.h"

#include "dataLink.h"

#include <stdio.h>

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

(void)signal(SIGALRM, atende); // instala rotina que atende interrupcao

int fd;

if (argc < 2) {

perror("Invalid num of arguments");

return -1;

}

fd = connection(argv[1], SENDER);

printf("llopen complete\n\n");

if (fd > 0)

sendData();

llclose(fd, SENDER);

printf("llclose complete\n");

return 0;

}