

*Protocolo de Ligação de Dados*

Relatório

Francisco Teixeira Lopes

ei11056@fe.up.pt

Nelson André Garrido da Costa

up201403128@fe.up.pt

Índice

Sumário 1

Introdução 2

Arquitetura 2

Estrutura do código 3

Casos de uso principais 4

Protocolo de ligação lógica 4

Protocolo de aplicação 4

Validação 5

Eficiência do protocolo de ligação de dados 5

Conclusões 7

Sumário

Este trabalho insere-se no âmbito da Unidade Curricular Redes de Computadores, do Mestrado Integrado em Engenharia Informática e da Computação da Faculdade de Engenharia da Universidade do Porto, abordando conteúdos programáticos, abordados nas aulas teóricas.

Entre eles, destacam-se o Nível Físico – *Physical Layer* -, a Camada de Ligação de Dados - Data *Link Layer* – e a Camada de Aplicação – *Application Layer*. A proposta de trabalho teve como base a transferência, por cabo, de ficheiros entre dois computadores, através das suas portas de série.

Este relatório prende-se na consolidação do trabalho e conhecimento adquirido até à data, fazendo-se a união entre as componentes prática e teórica sobre as quais se debruça o projeto apresentado, que se pretende mostrar bem entendidas e implementadas.

**Introdução**

O projeto foi trabalhado no contexto da Unidade Curricular de Redes de Computadores, inserida no plano do 3º ano do Mestrado Integrado em Engenharia Informática e Computação.

Os objetivos principais deste trabalho prendem-se essencialmente na implementação do protocolo de ligação de dados, através de uma aplicação responsável pela transferência de ficheiros entre dois computadores - o emissor e o recetor -, recorrendo às suas portas de série e combinando características de protocolos já existentes, como a transparência e a transmissão organizada em tramas – de informação, controlo e não numeradas - e utilizando integralmente transmissão em série assíncrona.

De modo a que o processo fosse feito em conformidade com estes objetivos, foram incluídos, na aplicação, mecanismos de deteção de erros como, por exemplo, dados enviados em duplicado ou falhas de conexão entre as máquinas, e respetiva resposta, assim como suporte para dois tipos de pacotes enviados pelo emissor: os de dados (com porções do ficheiro) e os de controlo para iniciar – *start* - e finalizar a transmissão – *end*.

Este relatório, sendo o *pós-laboral* do trabalho prático, serve como suporte à sua fácil compreensão e está estruturado da seguinte forma:

* **Introdução:** indicação dos objetivos do trabalho e do relatório; descrição da lógica do relatório;
* **Arquitetura:** blocos funcionais e interfaces;
* **Estrutura do código:** APIs consultadas e utilizadas, principais estruturas de dados usadas, principais funções e sua relação com a arquitetura;
* **Casos de uso principais;**
* **Protocolo de ligação lógica:** identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos;
* **Protocolo de aplicação:** tal como no ponto anterior, explana-se a maneira de pensar nesta fase;
* **Validação:** testes realizados e quantificação de resultados.
* **Eficiência do protocolo de ligação de dados:** caracterização estatística da eficiência do código desenvolvido comparativamente a uma caracterização teórica;
* **Conclusões:** considerações sintéticas finais sobre o trabalho realizado.

**Arquitetura**

A aplicação está organizada em duas camadas:

* **Camada do protocolo de ligação de dados:** funções do protocolo, especificando funcionalidades de sincronismo, numeração de tramas, controlo da transferência, tratamento de erros, abertura e definição das propriedades da porta de série – LinkLayer.c/LinkLayer.h
* **Camada da aplicação:** funções de escrita e leitura do ficheiro, através do processamento e envio de pacotes de controlo e dados, sendo fundamental para o funcionamento do programa. – AppLayer.c/AppLayer.h

Em relação à interface da aplicação, todos os valores globais às camadas e estruturas funcionais estão incluídos no ficheiro **Globals.h,** como, por exemplo, o *baudrate*, macros que definem valores predefinidos, como as *FLAGS*, tamanhos máximos para o nome e tamanho do ficheiro a transmitir, entre outros. É também apresentada, na linha de comandos, ao utilizador recetor, uma barra de progresso relativa à transferência com a respetiva percentagem. Mensagens *log* estão previstas, se o utilizador ativar a macro EN*ABLE\_DEBUG* no ficheiro suprarreferido.

**Estrutura do código**

* **Camada de ligação de dados:**

Utiliza-se uma ***struct*** para a sua representação, onde é guardada, entre outra informação, a porta de série, o ***baudrate***, um número de **sequência** e a última resposta, que determina qual a trama a esperar receber ou enviar de seguida, o tamanho, variáveis de ***timeout****,* para o contador de segundos de espera, e as configurações da ***termios***.

Das funções desta camada, destacam-se a ***llopen()*** - que tenta estabelecer a ligação -, ***llclose\_transmit()***/***llclose\_receive()*** - que tentam terminar a ligação -, ***llread()*** – que lê e valida pacotes de dados -, ***llwrite()*** – que processa e envia pacotes de dados.

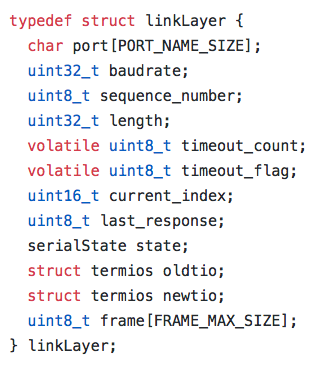
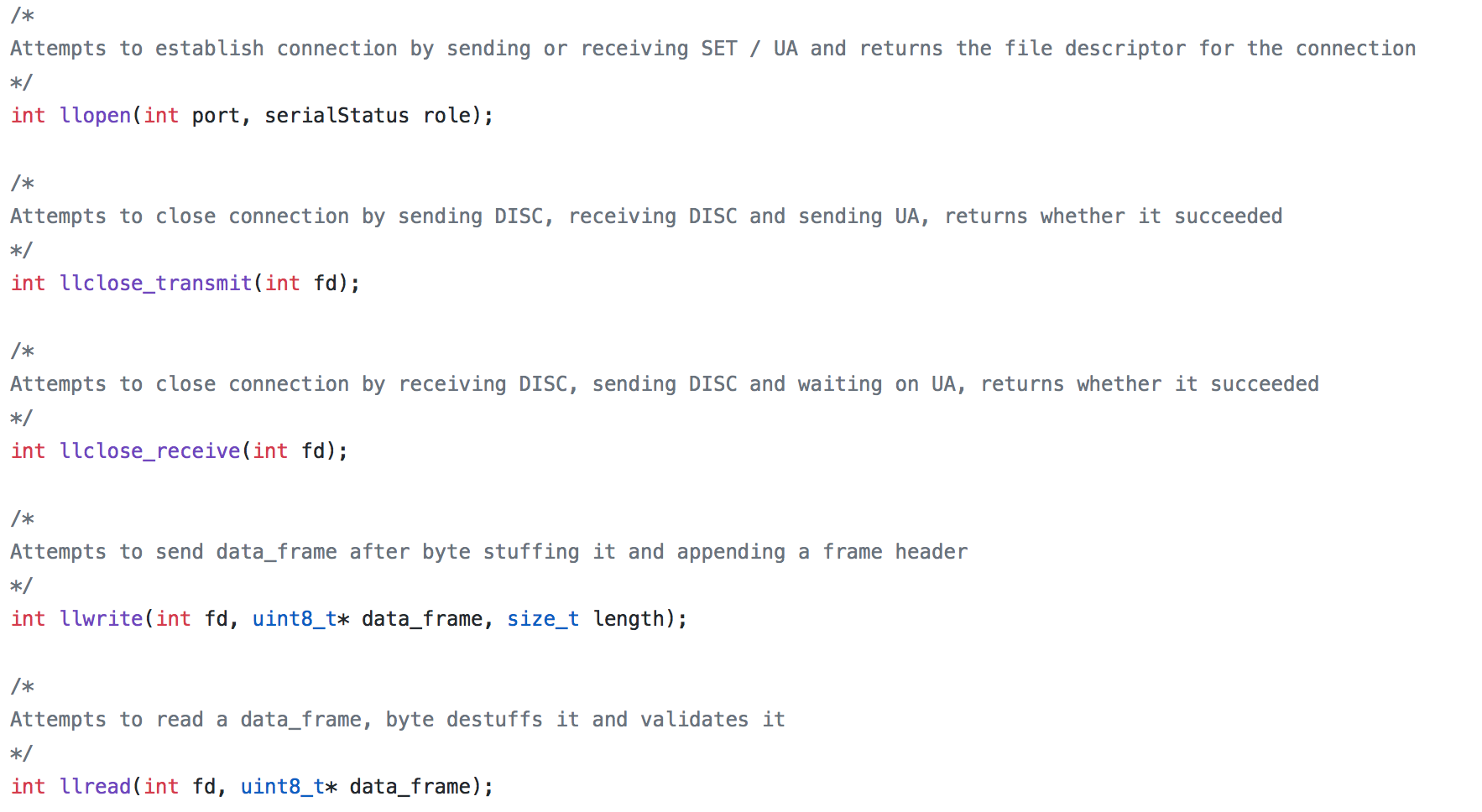


Fig1: código da *struct* da camada de ligação de dados.

Fig2: código dadeclaração das principais funções da camada de ligação de dados.

* **Camada de aplicação:**

Utiliza-se uma ***struct*** para a sua representação, onde é guardada, entre outras informações, o ***file descriptor*** da porta de série, o **tamanho** e o **nome** do ficheiro a transferir, um **contador** de envio e o **estado** do envio.

Das principais funções afetas a esta camada, destacam-se a ***alsend()*** – que abre o ficheiro a enviar e o processa em pacotes de dados-, ***alreceive()*** – que tenta ler e guardar o ficheiro no computador recetor -, ***createSEPacket()***/***createDataPacket()*** – que geram os pacotes de controlo e dados resultantes do ficheiro a enviar/receber.

Fig4: código dadeclaração das principais funções da camada de aplicação.

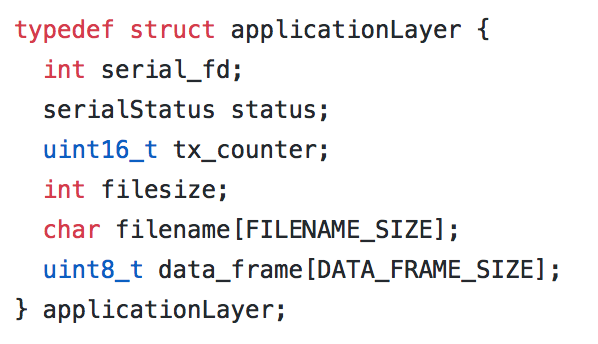
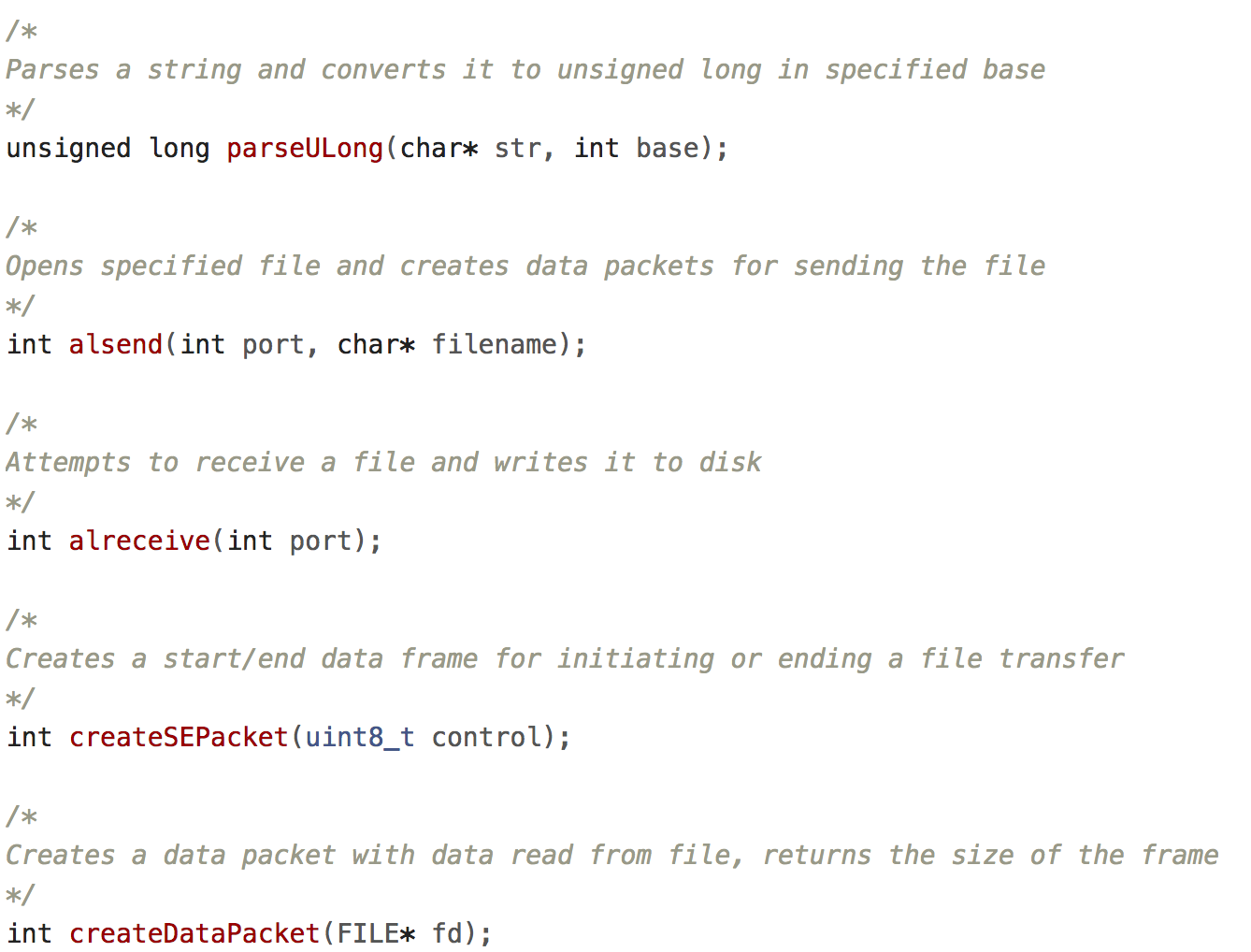


Fig3: código da *struct* da camada de aplicação.

**Casos de uso principais**

Os casos de uso do programa incluem aqueles que são responsáveis pela transferência dos ficheiros:

* Configuração da ligação com escolha do papel de emissor ou recetor e identificação do ficheiro, através do seu nome, pelo emissor.
* Estabelecimento da ligação.
* Computador emissor envia o ficheiro particionado em pacotes.
* Computador recetor recebe os dados e escreve-os no ficheiro a guardar.
* Terminação da ligação e fecho dos ficheiros até ao momento abertos.

**Protocolo da ligação lógica**

No que diz respeito ao protocolo da ligação lógica, implementa-se as principais funções relativas à camada de ligação lógica.

Ao ser invocada pela máquina emissora, *llopen()* envia o comando *SET* para tentar estabelecer a ligação, esperando pela resposta do recetor - a trama *UA* - que, se não for recebida dentro do período de tempo predefinido – 3 segundos -, o faz reenviar o *SET*. Este procedimento é replicado um número também predefinido de vezes – 3 vezes – que, se excedido, faz terminar o programa sem sucesso. Uma vez concluído este processo, a ligação é estabelecida e a execução continua.

De seguida, *llwrite()* recebe, como argumento, a informação a transmitir, que será usada para a partição em tramas, numeradas com número de sequência – 0 ou 1 – e que serão enviadas ao recetor. O programa fica, então, à espera duma resposta deste último, mais uma vez, durante um período de tempo, que findo, o faz reenviar a trama, e repetir o processo até se exceder as 3 tentativas – terminação sem sucesso. Quando se obtém, como resposta, a trama *RR*, o recetor recebeu a trama enviada sem erros e a função termina com sucesso. Se, pelo contrário, a resposta for uma trama *REJ*, o processo de envio repete-se.

A função *llread()* corre a par da *llwrite()*, na máquina recetora, e permanece em ciclo até receber uma trama, que é validada, recorrendo a uma máquina de estados que controla se os bytes recebidos são os esperados. Casos todos os campos estejam válidos, a função envia a trama *RR* e, caso contrário, envia a *REJ*, com os números de sequência da trama recebida. Por outro lado, se a função receber o comando *DISC*, a ligação deve ser terminada.

Por último, a função llclose() é invocada pelo emissor, que envia o comando *DISC*, indicando o fim da ligação, esperando receber, do recetor, um comando igual, que quando recebido, o faz enviar o comando *UA* – unnumbered acknowledgement.

**Protocolo de aplicação**

Relativamente ao protocolo de aplicação, é feita a leitura e a conversão dos argumentos passados na linha de comandos, nomeadamente o papel de cada computador, a porta de série e o nome do ficheiro a transferir.

De seguida, o computador emissor invoca *alsend()*, onde lê o ficheiro a enviar e o particiona em pacotes de dados.

O recetor, por sua vez, invoca *alreceive()*, que recebe os pacotes de dados, escrevendo o ficheiro internamente.

**Validação**

Para validar a eficácia do projeto desenvolvido, recorreu-se a vários métodos, que correram com sucesso:

* Interromper a ligação, enquanto a informação é transmitida.
* Geração de picos de tensão no cabo que liga as portas de série.
* Usar valores de *baudrate* e tamanho de pacotediferentes.
* Transmissão de diferentes ficheiros, com diferentes tamanhos.

**Eficiência do protocolo de ligação de dados**

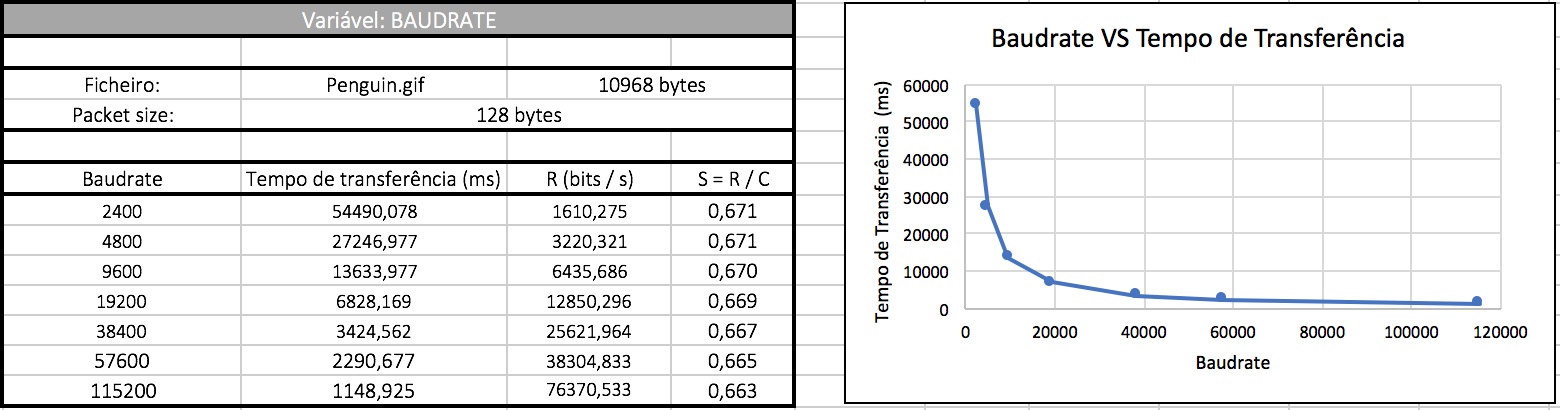


Fig5: Eficiência do programa, quando variada a taxa de transmissão, *baudrate*.

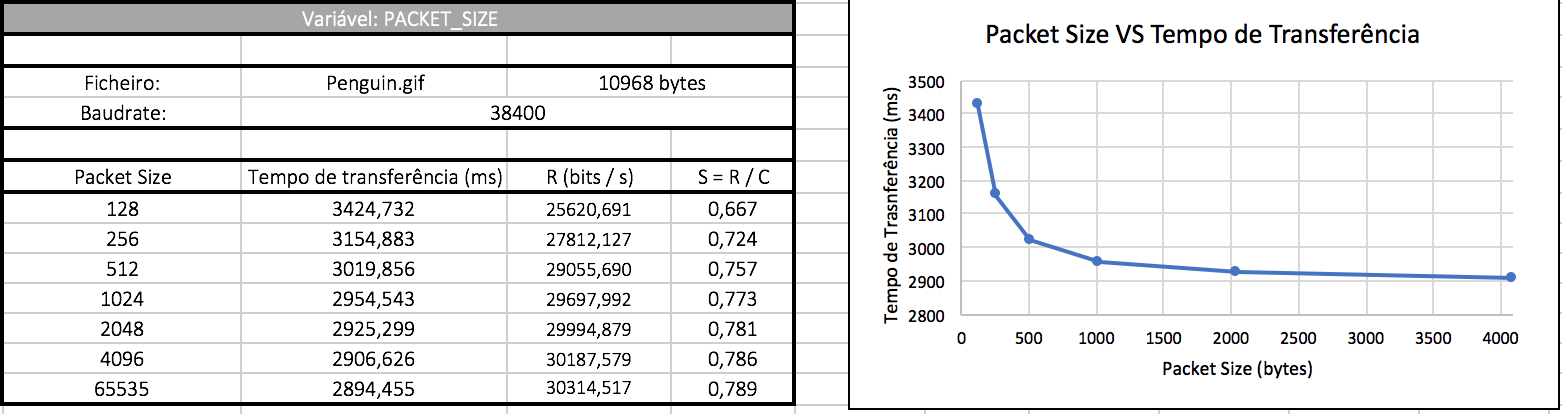


Fig6: Eficiência do programa, quando variado o tamanho dos pacotes de dados.

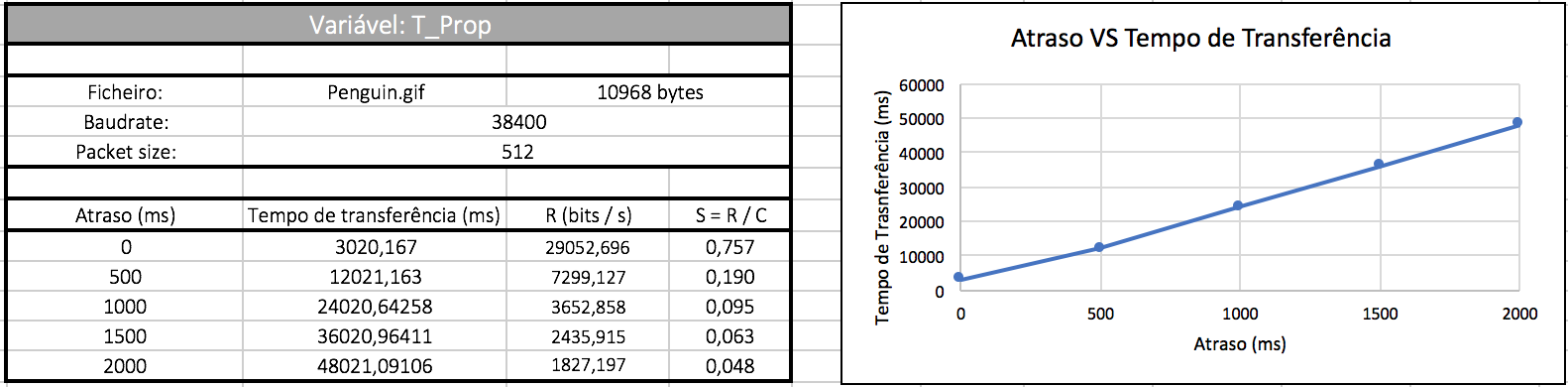
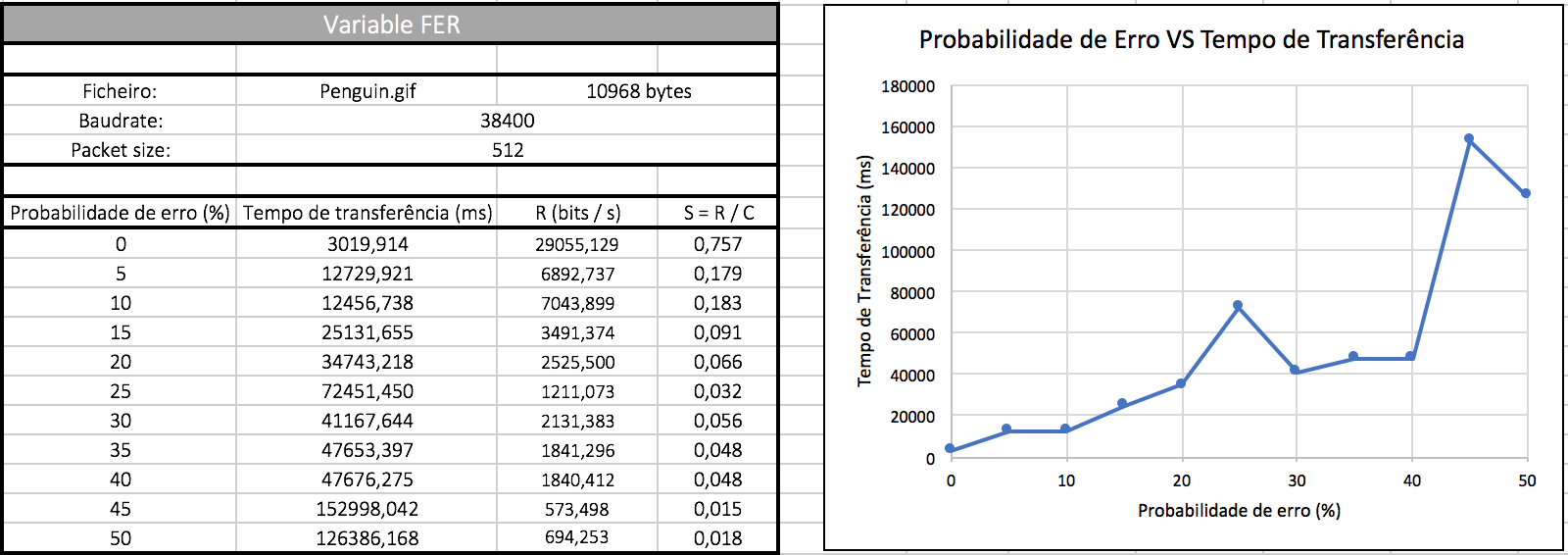


Fig8: Eficiência do programa, quando simulados erros, recorrendo a valores de probabilidade de ocorreram.

Fig7: Eficiência do programa, quando introduzido um atraso na receção das tramas, fazendo variar o tempo de propagação.

Relativamente ao teste de eficiência do programa com erros, convém referir que, para conseguirmos chegar a uma medição, alteramos o número de tentativas de *timeout*, de 3 para 10, de modo a que o programa não terminasse sem sucesso, principalmente com valores de probabilidade de erro mais elevados.

Comparando com os valores teóricos de referência relativos ao modem de linha telefónica, lecionado nas aulas teóricas, concluímos que o nosso projeto é eficiente, visto que o valor de *a* é sempre inferior a 1.

Neste contexto, não fizemos comparação teórica no teste de eficiência à T\_Prop, visto que estamos a induzir atrasos reais que afetam diretamente o tempo de execução do programa, o que resulta em valores de *a* muito díspares.

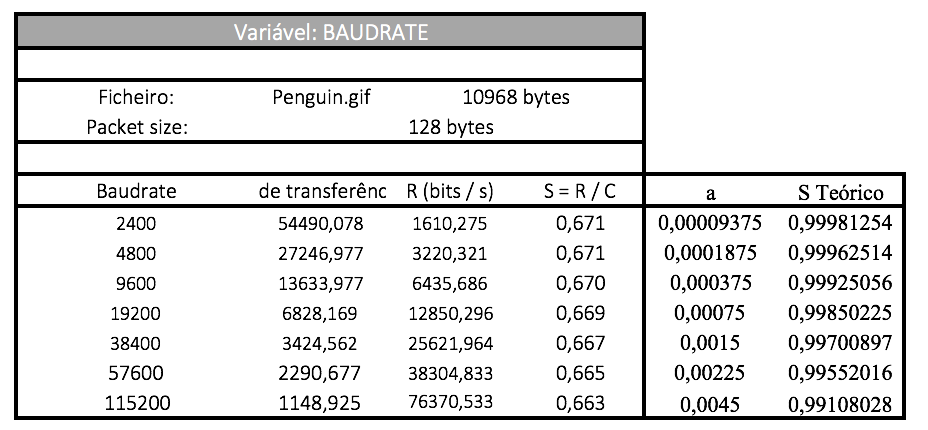
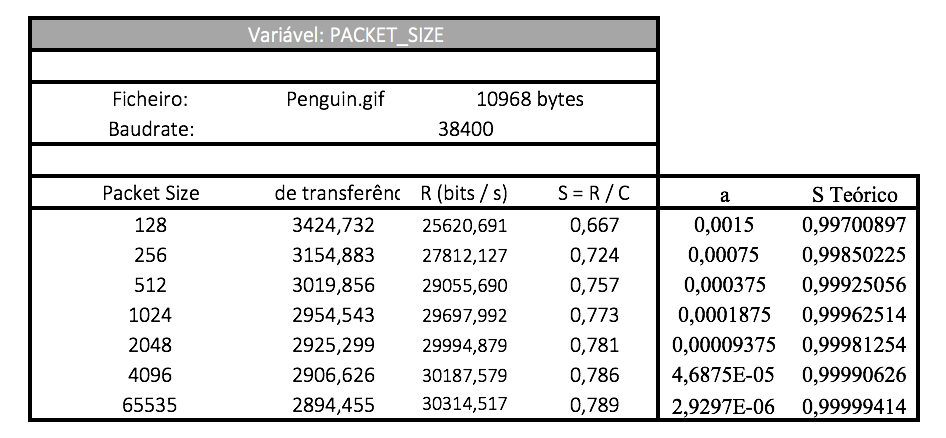
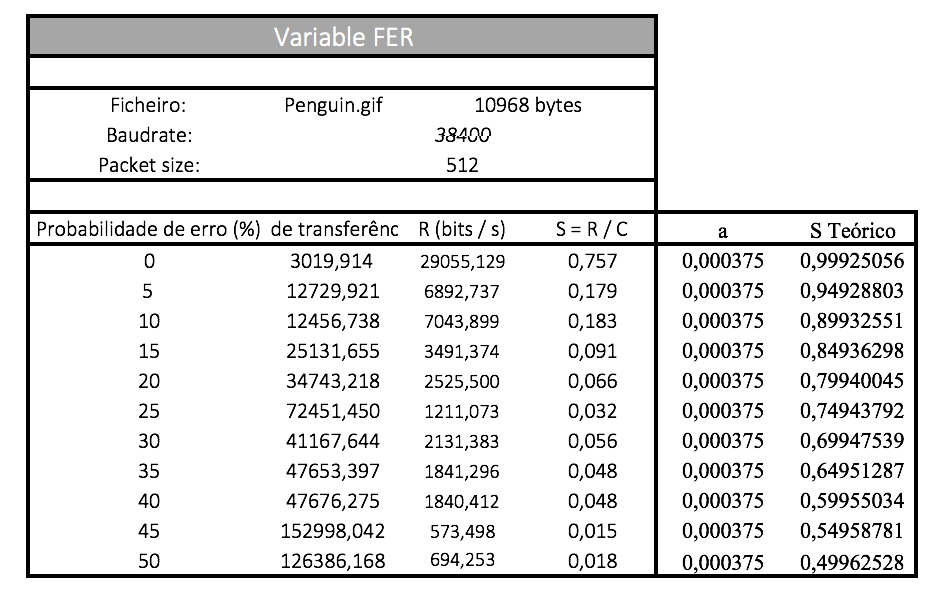


Fig9: Eficiência do programa, comparativamente aos valores teóricos.

**Conclusão**

O protocolo de ligação de dados foi correta e eficazmente implementado, tendo sido possível a transmissão resistente a erros de transferência entre dois computadores, utilizando as suas portas de série.

O programa possui duas camadas, que são inter-independentes, como estava previsto, sendo que na camada de ligação de dados não é feito qualquer processamento que incida sobre o cabeçalho do pacote de dados a enviar e na camada de aplicação não é feito qualquer processamento ou endereçamento ao cabeçalho da trama. As componentes características de outros protocolos, como a transparência, foram implementadas.

A eficiência do programa, depois de testada e analisada pode ser justificada pelo facto de se gastar uma parte considerável do tempo a processar a informação, em detrimento da sua simples transmissão.

Para concluir este relatório, considera-se por bem evidenciar que os objetivos deste projeto foram alcançados, tendo sido feita com sucesso, a união e a consolidação da componente teórica e da prática da Unidade Curricular respetiva.

**Anexo 1 – Source Code**

AppLayer.h

#ifndef APPLAYER\_H

#define APPLAYER\_H

/\*

Prints program usage

\*/

void printUsage(char\* program\_name);

/\*

Parses a string and converts it to unsigned long in specified base

\*/

unsigned long parseULong(char\* str, int base);

/\*

Opens specified file and creates data packets for sending the file

\*/

int alsend(int port, char\* filename);

/\*

Attempts to receive a file and writes it to disk

\*/

int alreceive(int port);

/\*

Get the filesize of a file with up to INT\_MAX bytes

\*/

int getFilesize(FILE\* fd);

/\*

Creates a start/end data frame for initiating or ending a file transfer

\*/

int createSEPacket(uint8\_t control);

/\*

Creates a data packet with data read from file, returns the size of the frame

\*/

int createDataPacket(FILE\* fd);

/\*

Writes received data frame to specified fd

\*/

int writeFile(FILE\* fd, uint8\_t\* data\_frame);

/\*

Update and show progress bar

\*/

void updateProgressBar(size\_t current\_bytes);

/\*

Helper function for printing a data frame

\*/

void printDataFrame(size\_t size);

/\*

Extract filesize and filename from data\_frame

\*/

int extractFileInfo(uint8\_t\* data\_frame, size\_t length);

/\*

Resets the data frame in the appLayer struct

\*/

void resetDataFrame();

#endif /\* APPLAYER\_H \*/

AppLayer.c

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <strings.h>

#include <stdint.h>

#include <unistd.h>

#include <signal.h>

#include <limits.h>

#include <errno.h>

#include <math.h>

#include "Globals.h"

#include "AppLayer.h"

#include "LinkLayer.h"

#include "Helper.h"

static applicationLayer app\_layer;

/\*

Prints program usage

\*/

void printUsage(char\* program\_name) {

printf("\tUsage : %s tx <port number [0,1]> <filename>\n", program\_name);

printf("\tUsage : %s rx <port number [0,1]>\n", program\_name);

printf("\tExample: %s tx 0 penguin.gif\n", program\_name);

printf("\tExample: %s rx 1\n", program\_name);

}

/\*

Parses a string and converts it to unsigned long in specified base

\*/

unsigned long parseULong(char\* str, int base) {

char \*endptr;

unsigned long val;

int errsv; //Preserve errno

//Convert string to unsigned long

errno = 0; //Clear errno

val = strtoul(str, &endptr, base);

errsv = errno;

//Check for conversion errors

if (errsv != 0) {

return ULONG\_MAX;

}

if (endptr == str) {

return ULONG\_MAX;

}

//Successful conversion

return val;

}

/\*

Helper function for printing a data frame

\*/

void printDataFrame(size\_t size) {

size\_t i;

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

LOG\_MSG("|0x%02X", app\_layer.data\_frame[i]);

}

LOG\_MSG("|\n");

}

/\*

Creates a start/end data frame for initiating or ending a file transfer, returns the size of the frame

\*/

int createSEPacket(uint8\_t control) {

size\_t index = 0;

app\_layer.data\_frame[0] = control;

/\* 1st parameter, filesize \*/

app\_layer.data\_frame[1] = T\_SIZE;

int filesize\_length = (int) ((ceil(log10(app\_layer.filesize)) + 1) \* sizeof(char));

app\_layer.data\_frame[2] = filesize\_length;

char filesize[FILESIZE\_MAX\_DIGITS];

snprintf(filesize, filesize\_length, "%d", app\_layer.filesize);

size\_t i;

for(i = 0; i < filesize\_length; i++) {

app\_layer.data\_frame[3 + i] = filesize[i];

}

index = i + 3;

/\* 2nd parameter, filename \*/

app\_layer.data\_frame[index] = T\_NAME;

app\_layer.data\_frame[index + 1] = strlen(app\_layer.filename) + 1;

for(i = 0; i < app\_layer.data\_frame[index + 1]; i++) {

app\_layer.data\_frame[index + 2 + i] = app\_layer.filename[i];

}

return index + 2 + i;

}

/\*

Creates a data packet with data read from file, returns the size of the frame

\*/

int createDataPacket(FILE\* fd) {

app\_layer.data\_frame[0] = C\_DATA;

app\_layer.data\_frame[1] = app\_layer.tx\_counter % 255;

int nread = fread(app\_layer.data\_frame + 4, 1, PACKET\_SIZE, fd);

app\_layer.data\_frame[2] = (nread >> 8) & 0xFF;

app\_layer.data\_frame[3] = nread & 0xFF;

return (nread + 4);

}

/\*

Resets the data frame in the appLayer struct

\*/

void resetDataFrame() {

bzero(&app\_layer.data\_frame, sizeof(app\_layer.data\_frame));

}

/\*

Extract filesize and filename from data\_frame

\*/

int extractFileInfo(uint8\_t\* data\_frame, size\_t length) {

size\_t current\_index = DCONTROL\_INDEX + 1;

if(data\_frame[current\_index] == T\_SIZE) {

size\_t par\_size = data\_frame[current\_index + 1];

char filesize[par\_size];

snprintf(filesize, par\_size, "%s", data\_frame + current\_index + 2);

app\_layer.filesize = parseULong(filesize, 10);

current\_index += par\_size + 2;

} else return -1;

if(data\_frame[current\_index] == T\_NAME) {

size\_t par\_size = data\_frame[current\_index + 1];

snprintf(app\_layer.filename, par\_size, "%s", data\_frame + current\_index + 2);

} else return -1;

return 0;

}

/\*

Writes received data frame to specified fd

\*/

int writeFile(FILE\* fd, uint8\_t\* data\_frame) {

uint8\_t MSB = data\_frame[2];

uint8\_t LSB = data\_frame[3];

size\_t nwrite = (MSB << 8) | (LSB & 0xFF);

fwrite(data\_frame + 4, 1, nwrite, fd);

return 0;

}

/\*

Update and show progress bar

\*/

void updateProgressBar(size\_t current\_bytes) {

printf("\033c");

double ratio = (double) current\_bytes / (double) app\_layer.filesize;

ratio \*= 100.0;

ratio = ceil(ratio);

uint8\_t filled\_size = ratio / 100.0 \* PROGRESS\_BAR\_SIZE;

printf("\u250C");

size\_t i;

for(i = 0; i < PROGRESS\_BAR\_SIZE; i++) {

printf("\u2500");

}

printf("\u2510\n");

printf("\u2502");

for(i = 0; i < PROGRESS\_BAR\_SIZE; i++) {

if(filled\_size > i) printf("\u2588");

else printf("\u2591");

}

printf("\u2502");

printf(" %d%%\n", (int) ratio);

printf("\u2514");

for(i = 0; i < PROGRESS\_BAR\_SIZE; i++) {

printf("\u2500");

}

printf("\u2518\n");

}

/\*

Opens specified file and creates data packets for sending the file

\*/

int alsend(int port, char\* filename) {

/\* Try to open file for sending \*/

strcpy(app\_layer.filename, filename);

FILE\* fd = fopen(app\_layer.filename, "rb");

if(fd == NULL) {

printf("%s not found!\n", app\_layer.filename);

exit(1);

}

/\* Try to establish a connection \*/

app\_layer.serial\_fd = llopen(port, TRANSMIT);

if(app\_layer.serial\_fd < 0) {

printf("alsend: couldn't open serial port for communication.\n");

exit(1);

}

printf("Connection established!\n");

/\* Validate the filesize \*/

app\_layer.filesize = getFilesize(fd);

LOG\_MSG("Filesize : %d\n", app\_layer.filesize);

if(app\_layer.filesize <= 0) {

printf("alsend: file has size 0 or exceeds 2 147 483 647 bytes (~2.14GiB).\n");

llreset(app\_layer.serial\_fd);

exit(1);

}

/\* Send the start I frame \*/

resetDataFrame();

size\_t packet\_size = createSEPacket(C\_START);

LOG\_MSG("S/E I Frame :");

printDataFrame(packet\_size);

uint8\_t end\_flag = FALSE;

size\_t bytes\_received = 0;

while(TRUE) {

uint8\_t\* data\_frame = (uint8\_t\*) malloc(packet\_size \* 2); //Allocate double amount for byte stuffing

memcpy(data\_frame, app\_layer.data\_frame, packet\_size);

int nwrite = llwrite(app\_layer.serial\_fd, data\_frame, packet\_size);

if(nwrite < 0) {

llreset(app\_layer.serial\_fd);

exit(1);

}

app\_layer.tx\_counter++;

if(end\_flag) {

free(data\_frame);

break;

}

resetDataFrame();

packet\_size = createDataPacket(fd);

free(data\_frame);

if(packet\_size <= HEADER\_SIZE - 1) {

packet\_size = createSEPacket(C\_END);

end\_flag = TRUE;

}

if(!end\_flag) {

bytes\_received += packet\_size - DPACKET\_HEADER\_SIZE;

#if ENABLE\_DEBUG == 0

updateProgressBar(bytes\_received);

#endif

}

}

llclose\_transmit(app\_layer.serial\_fd);

printf("Connection closed, %s transferred successfully!\n", app\_layer.filename);

close(app\_layer.serial\_fd);

fclose(fd);

return 0;

}

/\*

Attempts to receive a file and writes it to disk

\*/

int alreceive(int port) {

app\_layer.serial\_fd = llopen(port, RECEIVE);

if(app\_layer.serial\_fd < 0) {

printf("alreceive: couldn't open serial port for communication.\n");

exit(1);

}

printf("Connection established!\n");

uint8\_t\* data\_frame = (uint8\_t\*) malloc(FRAME\_MAX\_SIZE);

FILE\* fd;

size\_t bytes\_received = 0;

while(TRUE) {

int nread = llread(app\_layer.serial\_fd, data\_frame);

if(nread >= 0 && data\_frame[DCONTROL\_INDEX] == C\_DATA) {

bytes\_received += nread - HEADER\_SIZE;

#if ENABLE\_DEBUG == 0

updateProgressBar(bytes\_received);

#endif

}

if(nread < 0) continue;

if(data\_frame[DCONTROL\_INDEX] == C\_START) {

extractFileInfo(data\_frame, nread);

fd = fopen(app\_layer.filename, "wb");

LOG\_MSG("Filename: %s Filesize: %d\n", app\_layer.filename, app\_layer.filesize);

} else if(data\_frame[DCONTROL\_INDEX] == C\_DATA) {

writeFile(fd, data\_frame);

} else if(data\_frame[DCONTROL\_INDEX] == C\_END) break; //TODO validate filesize / filename

}

free(data\_frame);

llclose\_receive(app\_layer.serial\_fd);

printf("Connection closed, %s transferred successfully!\n", app\_layer.filename);

close(app\_layer.serial\_fd);

return 0;

}

/\*

Get the filesize of a file with up to INT\_MAX bytes

\*/

int getFilesize(FILE\* fd) {

fseek(fd, 0, SEEK\_END);

int filesize = ftell(fd);

fseek(fd, 0, SEEK\_SET);

if(filesize == INT\_MAX) return -1;

return filesize;

}

/\*

Program entry point, parses user input and continues if validation succeeded

\*/

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

if(argc < 3) {

printf("%s: wrong number of arguments!\n", argv[0]);

printUsage(argv[0]);

exit(1);

} else if((strcmp("tx", argv[1]) != 0) && (strcmp("rx", argv[1]) != 0)) {

printf("%s: mode selected does not exist, tx or rx are the available modes.\n", argv[0]);

printUsage(argv[0]);

exit(1);

} else if((strcmp("0", argv[2]) != 0) && (strcmp("1", argv[2]) != 0)) {

printf("%s: port number must be 0 or 1!\n", argv[0]);

printUsage(argv[0]);

exit(1);

} else if((strcmp("tx", argv[1]) == 0) && argc < 4) {

printf("%s: missing filename after port number!\n", argv[0]);

printUsage(argv[0]);

exit(1);

}

if((strcmp("tx", argv[1]) == 0) && strlen(argv[3]) + 1 > FILENAME\_SIZE) {

printf("Filename too long, can only be %d chars long.\n", FILENAME\_SIZE - 1);

exit(1);

}

memset(&app\_layer, 0, sizeof(app\_layer));

if(strcmp("tx", argv[1]) == 0) alsend(parseULong(argv[2], 10), argv[3]);

if(strcmp("rx", argv[1]) == 0) alreceive(parseULong(argv[2], 10));

return 0;

}

LinkLayer.h

#ifndef LINKLAYER\_H

#define LINKLAYER\_H

/\*

Attempts to establish connection by sending or receiving SET / UA and returns the file descriptor for the connection

\*/

int llopen(int port, serialStatus role);

/\*

Attempts to close connection by sending DISC, receiving DISC and sending UA, returns whether it succeeded

\*/

int llclose\_transmit(int fd);

/\*

Attempts to close connection by receiving DISC, sending DISC and waiting on UA, returns whether it succeeded

\*/

int llclose\_receive(int fd);

/\*

Attempts to send data\_frame after byte stuffing it and appending a frame header

\*/

int llwrite(int fd, uint8\_t\* data\_frame, size\_t length);

/\*

Attempts to read a data\_frame, byte destuffs it and validates it

\*/

int llread(int fd, uint8\_t\* data\_frame);

/\*

Restores previous serial port settings

\*/

void llreset(int fd);

#endif /\* LINKLAYER\_H \*/

LinkLayer.c

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <strings.h>

#include <stdint.h>

#include <unistd.h>

#include <signal.h>

#include <limits.h>

#include <errno.h>

#include "Globals.h"

#include "LinkLayer.h"

#include "Helper.h"

static linkLayer link\_layer;

/\*

Activates timeout flag and increments the timeout count

\*/

void timeoutAlarm() {

link\_layer.timeout\_flag = TRUE;

link\_layer.timeout\_count++;

signal(SIGALRM, timeoutAlarm);

}

/\*

Creates a supervision or unnumbered frame from address and control field given (F | A | C | BCC | F)

\*/

void createSUFrame(uint8\_t address, uint8\_t control) {

link\_layer.frame[0] = FLAG;

link\_layer.frame[1] = address;

link\_layer.frame[2] = control;

link\_layer.frame[3] = BCC1(address, control);

link\_layer.frame[4] = FLAG;

}

/\*

Initializes serial port and sets up the needed fiels on linkLayer struct

\*/

int initSerial(uint32\_t baudrate, serialStatus role) {

int fd;

/\* Open serial port for reading and writing and not as controlling terminal \*/

fd = open(link\_layer.port, O\_RDWR | O\_NOCTTY);

if (fd < 0) {

printf("initSerial: failed to open port %s.\n", link\_layer.port);

return -1;

}

/\* Save current port settings \*/

if(tcgetattr(fd, &link\_layer.oldtio) == -1) {

printf("initSerial: couldn't get port settings on port %s.\n", link\_layer.port);

return -1;

}

bzero(&link\_layer.newtio, sizeof(link\_layer.newtio));

link\_layer.newtio.c\_cflag = baudrate | CS8 | CLOCAL | CREAD;

link\_layer.newtio.c\_iflag = IGNPAR;

link\_layer.newtio.c\_oflag = 0;

link\_layer.newtio.c\_lflag = 0; /\* Set input mode: non-canonical / no echo \*/

link\_layer.newtio.c\_cc[VTIME] = 1; /\* Inter-character timer \*/

link\_layer.newtio.c\_cc[VMIN] = 0; /\* Minimum number of characters to read \*/

tcflush(fd, TCIOFLUSH);

if(tcsetattr(fd, TCSANOW, &link\_layer.newtio) == -1) {

printf("initSerial: couldn't set port settings on port %s.\n", link\_layer.port);

return -1;

}

link\_layer.baudrate = baudrate;

if(role == TRANSMIT) link\_layer.timeout\_flag = TRUE;

link\_layer.state = INIT;

LOG\_MSG("Using %s.\n", link\_layer.port);

return fd;

}

/\*

Link layer state machine

\*/

void llstate(serialState\* state, serialEvent event) {

switch(\*state) {

case INIT:

if(event == FLAG\_E) \*state = S\_FLAG;

else if(event == TIMEOUT\_E) \*state = INIT;

break;

case S\_FLAG:

if(event == FLAG\_E) \*state = E\_FLAG;

else if(event == TIMEOUT\_E) \*state = INIT;

break;

case E\_FLAG:

if(event == TIMEOUT\_E) \*state = INIT;

break;

}

}

/\*

Helper function for printing a full frame

\*/

void printFrame(size\_t size) {

size\_t i;

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

LOG\_MSG("|0x%02X", link\_layer.frame[i]);

}

LOG\_MSG("|\n");

}

/\*

Checks if alarm has happened and resets it if it has

\*/

void checkAlarm() {

if(link\_layer.timeout\_flag) {

LOG\_MSG("Alarm #%d\n", link\_layer.timeout\_count);

alarm(TIMEOUT\_S);

link\_layer.timeout\_flag = FALSE;

}

}

/\*

Turns off alarm and resets flag and timeout count

\*/

void downAlarm() {

alarm(0);

link\_layer.timeout\_flag = FALSE;

link\_layer.timeout\_count = 0;

}

/\*

Checks the header BCC and returns true if it checks out

\*/

int checkHeaderBCC() {

if(link\_layer.frame[BCC1\_INDEX] != (link\_layer.frame[ADDRESS\_INDEX] ^ link\_layer.frame[CONTROL\_INDEX])) {

return FALSE;

} else return TRUE;

}

/\*

Empties serial port of data

\*/

void discardFrame(int fd) {

size\_t nread;

do {

nread = read(fd, link\_layer.frame, 1);

} while(nread != 0);

LOG\_MSG("Last: 0x%02X\n", link\_layer.frame[0]);

}

/\*

Reads a full frame from the port and checks BCC1 and the control field expected, returns error type or no error on success

\*/

serialError readFrame(int fd, uint8\_t control\_expected) {

size\_t nread = 0;

size\_t total\_read = 0;

link\_layer.length = 0;

while(TRUE) {

nread = read(fd, link\_layer.frame + link\_layer.current\_index, 1);

if(nread > 0) {

LOG\_MSG("Received[%03i]: 0x%02X | Before state: %i |", link\_layer.current\_index, link\_layer.frame[link\_layer.current\_index], link\_layer.state);

if(link\_layer.current\_index == 0) {

if(link\_layer.frame[link\_layer.current\_index] != FLAG) {

return FLAG\_ERR;

}

}

if(link\_layer.frame[link\_layer.current\_index] == FLAG) {

llstate(&link\_layer.state, FLAG\_E);

}

LOG\_MSG(" After state: %i\n", link\_layer.state);

link\_layer.current\_index++;

total\_read += nread;

}

if(link\_layer.state == E\_FLAG) {

downAlarm();

link\_layer.length = total\_read;

if(!checkHeaderBCC()) return BCC1\_ERR;

if(link\_layer.frame[CONTROL\_INDEX] != control\_expected) {

link\_layer.last\_response = link\_layer.frame[CONTROL\_INDEX];

return CONTROL\_ERR;

}

return NO\_ERR;

} else if(link\_layer.timeout\_flag == TRUE) {

return TIMEOUT\_ERR;

}

}

}

/\*

Resets the linkLayer struct fields needed to run another write/read loop

\*/

void resetFrame() {

link\_layer.state = INIT;

link\_layer.current\_index = 0;

bzero(&link\_layer.frame, sizeof(link\_layer.frame));

}

/\*

Calculates the BCC corresponding to a data frame

\*/

uint8\_t calcBCC2(uint8\_t\* data\_frame, size\_t length) {

uint8\_t bcc = data\_frame[0];

size\_t i;

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

bcc ^= data\_frame[i];

}

return bcc;

}

/\*

Searches for occurrences of flag or the escape character in the data frame and replaces them

\*/

size\_t byte\_stuff(uint8\_t\* data\_frame, size\_t length) {

char result[FRAME\_MAX\_SIZE];

size\_t add\_length = 0;

size\_t i;

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

if(data\_frame[i] == FLAG) {

result[i + add\_length] = ESCAPE\_CHAR;

result[i + 1 + add\_length] = FLAG ^ XOCTET;

add\_length++;

} else if(data\_frame[i] == ESCAPE\_CHAR) {

result[i + add\_length] = ESCAPE\_CHAR;

result[i + 1 + add\_length] = ESCAPE\_CHAR ^ XOCTET;

add\_length++;

} else {

result[i + add\_length] = data\_frame[i];

}

}

bzero(data\_frame, length);

memcpy(data\_frame, result, length + add\_length);

return (length + add\_length);

}

/\*

Searches for occurrences of the escape char and replaces it with the escaped char

\*/

size\_t byte\_destuff(uint8\_t\* data\_frame, size\_t length) {

char result[FRAME\_MAX\_SIZE];

size\_t new\_length = 0;

size\_t i;

for(i = 0; i < length; i++, new\_length++) {

if(data\_frame[i] == ESCAPE\_CHAR) {

result[new\_length] = data\_frame[i + 1] ^ XOCTET;

i++;

} else result[new\_length] = data\_frame[i];

}

bzero(data\_frame, length);

memcpy(data\_frame, result, new\_length);

return new\_length;

}

/\*

Extract data frame from a full frame with BCC2 included

\*/

void extractDataFrame(uint8\_t\* data\_frame, size\_t length) {

size\_t i;

for(i = 0; i < length - HEADER\_SIZE; i++) {

data\_frame[i] = link\_layer.frame[i + 4];

}

}

/\*

Creates an information frame from given parameters (F | A | C | BCC1 | D1..DN | BCC2 | F)

\*/

void createIFrame(uint8\_t address, uint8\_t control, uint8\_t\* data\_frame, size\_t length) {

link\_layer.frame[0] = FLAG;

link\_layer.frame[1] = address;

link\_layer.frame[2] = control;

link\_layer.frame[3] = BCC1(address, control);

memcpy(link\_layer.frame + 4, data\_frame, length);

link\_layer.frame[length + 4] = FLAG;

}

/\*

Attempts to send data\_frame after byte stuffing it and appending a frame header

\*/

int llwrite(int fd, uint8\_t\* data\_frame, size\_t length) {

/\* Calculate BCC2 and append it to data\_frame \*/

uint8\_t bcc\_2 = calcBCC2(data\_frame, length);

data\_frame[length] = bcc\_2;

/\* Byte stuff the data\_frame and update length \*/

size\_t stuffed\_length = byte\_stuff(data\_frame, length + 1);

length = stuffed\_length;

LOG\_MSG("BCC2 : 0x%02X\n", bcc\_2);

uint8\_t success = FALSE;

int nwrite;

signal(SIGALRM, timeoutAlarm);

link\_layer.timeout\_flag = TRUE;

while(link\_layer.timeout\_count <= TIMEOUT) {

resetFrame();

createIFrame(A\_TX, C\_NS(link\_layer.sequence\_number), data\_frame, length);

nwrite = write(fd, link\_layer.frame, length + 5);

LOG\_MSG("Sent (b%03i) :", nwrite);

printFrame(length + 5); LOG\_MSG("\n");

checkAlarm();

resetFrame();

serialError status = readFrame(fd, C\_RR(!link\_layer.sequence\_number));

if(status == NO\_ERR) {

success = TRUE;

break;

}

if(status == BCC1\_ERR) {

link\_layer.timeout\_flag = TRUE;

} else if(status == CONTROL\_ERR) {

LOG\_MSG("Control field was not as expected.\n");

if(link\_layer.last\_response == C\_REJ(0)) link\_layer.sequence\_number = 0;

if(link\_layer.last\_response == C\_REJ(1)) link\_layer.sequence\_number = 1;

LOG\_MSG("Last response: 0x%02X\n", link\_layer.last\_response);

link\_layer.timeout\_flag = TRUE;

}

}

LOG\_MSG("Received :");

printFrame(link\_layer.current\_index);

if(success) {

link\_layer.sequence\_number = !link\_layer.sequence\_number;

return nwrite;

}

else {

LOG\_MSG("Maximum number of retries reached.\n");

return -1;

}

}

/\*

Attempts to read a data\_frame, byte destuffs it and validates it

\*/

int llread(int fd, uint8\_t\* data\_frame) {

uint8\_t success = FALSE;

uint8\_t bcc\_error = FALSE;

while(success == FALSE) {

resetFrame();

serialError status = readFrame(fd, C\_NS(link\_layer.sequence\_number));

LOG\_MSG("Received(%03d):", link\_layer.length);

printFrame(link\_layer.current\_index);

if(status == FLAG\_ERR) {

discardFrame(fd);

} else if(status == BCC1\_ERR) {

LOG\_MSG("BCC1 field was not as expected.\n");

} else if(status == CONTROL\_ERR) {

LOG\_MSG("Control field was not as expected.\n");

break;

} else success = TRUE;

}

if(success) {

/\* Extract data frame with BCC2 included \*/

extractDataFrame(data\_frame, link\_layer.length);

/\* Destuff data frame extracted \*/

size\_t destuffed\_length = byte\_destuff(data\_frame, link\_layer.length - HEADER\_SIZE);

link\_layer.length = destuffed\_length;

uint8\_t bcc\_2 = calcBCC2(data\_frame, link\_layer.length - 1);

resetFrame();

if(bcc\_2 != data\_frame[link\_layer.length - 1]) {

LOG\_MSG("BCC2 Error\n");

createSUFrame(A\_TX, C\_REJ(link\_layer.sequence\_number));

bcc\_error = TRUE;

} else createSUFrame(A\_TX, C\_RR(!link\_layer.sequence\_number));

} else {

resetFrame();

createSUFrame(A\_TX, C\_RR(link\_layer.sequence\_number));

}

/\* Send response \*/

#if ENABLE\_DEBUG == 1

int nwrite = write(fd, link\_layer.frame, SU\_FRAME\_SIZE);

#else

write(fd, link\_layer.frame, SU\_FRAME\_SIZE);

#endif

LOG\_MSG("Sent (b%03i) :", nwrite);

printFrame(SU\_FRAME\_SIZE);

if(!bcc\_error && success) link\_layer.sequence\_number = !link\_layer.sequence\_number; //Update sequence number on success

if(bcc\_error || !success) return -1;

size\_t length = link\_layer.length - 1;

link\_layer.length = 0;

return length;

}

/\*

Connection establishment specific to the transmitter, returns 0 on success, negative number otherwise

\*/

int llopen\_transmit(int fd) {

uint8\_t success = FALSE;

while(link\_layer.timeout\_count <= TIMEOUT) {

resetFrame();

createSUFrame(A\_TX, C\_SET);

#if ENABLE\_DEBUG == 1

int nwrite = write(fd, link\_layer.frame, SU\_FRAME\_SIZE);

#else

write(fd, link\_layer.frame, SU\_FRAME\_SIZE);

#endif

LOG\_MSG("Sent (%03d) :", nwrite);

printFrame(SU\_FRAME\_SIZE);

checkAlarm();

resetFrame();

serialError status = readFrame(fd, C\_UA);

if(status == NO\_ERR) {

success = TRUE;

break;

}

if(status == BCC1\_ERR || status == CONTROL\_ERR) {

LOG\_MSG("BCC1 or Control field was not as expected.\n");

return -1;

}

}

LOG\_MSG("Received(%03d):", link\_layer.length);

printFrame(link\_layer.current\_index);

if(success) return 0;

else {

LOG\_MSG("Maximum number of retries reached.\n");

return -1;

}

}

/\*

Connection establishment specific to the receiver, returns 0 on success, negative number otherwise

\*/

int llopen\_receive(int fd) {

resetFrame();

serialError status = readFrame(fd, C\_SET);

LOG\_MSG("Received(%03d):", link\_layer.length);

printFrame(link\_layer.current\_index);

if(status == BCC1\_ERR || status == CONTROL\_ERR) {

LOG\_MSG("BCC1 or Control field was not as expected.\n");

return -1;

}

resetFrame();

createSUFrame(A\_TX, C\_UA);

#if ENABLE\_DEBUG == 1

int nwrite = write(fd, link\_layer.frame, SU\_FRAME\_SIZE);

#else

write(fd, link\_layer.frame, SU\_FRAME\_SIZE);

#endif

LOG\_MSG("Sent (%03i) :", nwrite);

printFrame(SU\_FRAME\_SIZE);

return 0;

}

/\*

Attempts to establish connection by sending or receiving SET / UA and returns the file descriptor for the connection

\*/

int llopen(int port, serialStatus role) {

memset(&link\_layer, 0, sizeof(link\_layer));

signal(SIGALRM, timeoutAlarm);

int fd;

snprintf(link\_layer.port, PORT\_NAME\_SIZE + 1, "%s%d", "/dev/ttyS", port);

fd = initSerial(BAUDRATE, role);

if(fd < 0) return -1;

if(role == TRANSMIT) {

if(llopen\_transmit(fd) != 0) {

llreset(fd);

return -1;

}

} else if(role == RECEIVE) {

if(llopen\_receive(fd) != 0) {

llreset(fd);

return -1;

}

}

return fd;

}

/\*

Attempts to close connection by sending DISC, receiving DISC and sending UA, returns whether it succeeded

\*/

int llclose\_transmit(int fd) {

uint8\_t success = FALSE;

while(link\_layer.timeout\_count <= TIMEOUT) {

resetFrame();

createSUFrame(A\_TX, C\_DISC);

#if ENABLE\_DEBUG == 1

int nwrite = write(fd, link\_layer.frame, SU\_FRAME\_SIZE);

#else

write(fd, link\_layer.frame, SU\_FRAME\_SIZE);

#endif

LOG\_MSG("Sent (%03d) :", nwrite);

printFrame(SU\_FRAME\_SIZE);

checkAlarm();

resetFrame();

serialError status = readFrame(fd, C\_DISC);

if(status == NO\_ERR) {

success = TRUE;

break;

}

if(status == BCC1\_ERR || status == CONTROL\_ERR) {

LOG\_MSG("BCC1 or Control field was not as expected.\n");

llreset(fd);

return -1;

}

}

LOG\_MSG("Received(%03d):", link\_layer.length);

printFrame(link\_layer.current\_index);

resetFrame();

createSUFrame(A\_RX, C\_UA);

#if ENABLE\_DEBUG == 1

int nwrite = write(fd, link\_layer.frame, SU\_FRAME\_SIZE);

#else

write(fd, link\_layer.frame, SU\_FRAME\_SIZE);

#endif

LOG\_MSG("Sent (%03d) :", nwrite);

printFrame(SU\_FRAME\_SIZE);

sleep(CLOSE\_WAIT);

if(success) {

llreset(fd);

return 0;

} else {

LOG\_MSG("Maximum number of retries reached.\n");

llreset(fd);

return -1;

}

}

/\*

Attempts to close connection by receiving DISC, sending DISC and waiting on UA, returns whether it succeeded

\*/

int llclose\_receive(int fd) {

resetFrame();

serialError status = readFrame(fd, C\_DISC);

LOG\_MSG("Received(%03d):", link\_layer.length);

printFrame(link\_layer.current\_index);

if(status == BCC1\_ERR || status == CONTROL\_ERR) {

LOG\_MSG("BCC1 or Control field was not as expected.\n");

llreset(fd);

return -1;

}

resetFrame();

createSUFrame(A\_RX, C\_DISC);

#if ENABLE\_DEBUG == 1

int nwrite = write(fd, link\_layer.frame, SU\_FRAME\_SIZE);

#else

write(fd, link\_layer.frame, SU\_FRAME\_SIZE);

#endif

LOG\_MSG("Sent (%03i) :", nwrite);

printFrame(SU\_FRAME\_SIZE);

resetFrame();

status = readFrame(fd, C\_UA);

if(status == BCC1\_ERR || status == CONTROL\_ERR) {

LOG\_MSG("BCC1 or Control field was not as expected.\n");

llreset(fd);

return -1;

}

llreset(fd);

return 0;

}

/\*

Restores previous serial port settings

\*/

void llreset(int fd) {

if (tcsetattr(fd, TCSANOW, &link\_layer.oldtio) == -1) {

printf("llreset: couldn't set port settings on port %s.\n", link\_layer.port);

exit(1);

}

}

Helper.h

#ifndef HELPER\_H

#define HELPER\_H

/\*

Helper function for printing a generic array in hex format

\*/

void printArrayAsHex(uint8\_t\* buffer, size\_t size);

/\*

Helper function for printing a data frame as ASCII

\*/

void printArrayAsASCII(uint8\_t\* buffer, size\_t size);

#endif /\* HELPER\_H \*/

Helper.c

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <strings.h>

#include <stdint.h>

#include <unistd.h>

#include <signal.h>

#include <limits.h>

#include <errno.h>

#include <math.h>

#include "Globals.h"

/\*

Helper function for printing a generic array in hex format

\*/

void printArrayAsHex(uint8\_t\* buffer, size\_t size) {

size\_t i;

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

LOG\_MSG("|0x%02X", buffer[i]);

}

LOG\_MSG("|\n");

}

/\*

Helper function for printing a data frame as ASCII

\*/

void printArrayAsASCII(uint8\_t\* buffer, size\_t size) {

size\_t i;

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

LOG\_MSG("|%c", buffer[i]);

}

LOG\_MSG("|\n");

}

Globals.h

#ifndef GLOBALS\_H

#define GLOBALS\_H

/\*

Useful macros

\*/

#define ENABLE\_DEBUG 0

#if ENABLE\_DEBUG

#define LOG\_MSG printf

#else

#define LOG\_MSG(...)

#endif

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

#define BCC1(a, b) (a ^ b)

#define FALSE 0

#define TRUE 1

#define ABS(x) ((x) < 0 ? -(x) : (x))

/\*

Application layer constants

\*/

#define PROGRESS\_BAR\_SIZE 50

#define PORT\_NAME\_SIZE 10

#define FILESIZE\_MAX\_DIGITS 11

#define DPACKET\_HEADER\_SIZE 4

#define FILENAME\_SIZE 256

#define MAX\_START\_FRAME 272 /\* Maximum number of bytes for a start/end data frame, allows for 256 bytes filenames \*/

#define PACKET\_SIZE 256

#define DATA\_FRAME\_SIZE ((PACKET\_SIZE + 4 < MAX\_START\_FRAME) ? MAX\_START\_FRAME : PACKET\_SIZE + 4) /\* Picks the highest value a data frame will ever possibly need \*/

#define C\_NS(n) BIT(6) \* n

#define C\_DATA 0x01

#define C\_START 0x02

#define C\_END 0x03

#define T\_SIZE 0

#define T\_NAME 1

#define DCONTROL\_INDEX 0

/\*

Link layer constants

\*/

#define FLAG 0x7E /\* Start / End frame flag \*/

#define ESCAPE\_CHAR 0x7D /\* Escape character for byte stuffing \*/

#define XOCTET 0x20 /\* Octet to xor with for byte stuffing \*/

#define A\_TX 0x03 /\* Address field when sending commands as emitter or responses as receiver \*/

#define A\_RX 0x01 /\* Address field when sending commands as receiver or responses as emitter \*/

#define C\_SET 0x03 /\* Set up command \*/

#define C\_DISC 0x0B /\* Disconnect command \*/

#define C\_UA 0x07 /\* Unnumbered acknowledgment response \*/

#define C\_RR(n) (0x05 | BIT(7 \* n)) /\* Receiver ready response \*/

#define C\_REJ(n) (0x01 | BIT(7 \* n)) /\* Reject response \*/

#define SU\_FRAME\_SIZE 5 /\* Expected supervision frame size \*/

#define ADDRESS\_INDEX 1 /\* Frame header address index \*/

#define CONTROL\_INDEX 2 /\* Frame header control index \*/

#define BCC1\_INDEX 3 /\* Frame header BCC index \*/

#define HEADER\_SIZE 5 /\* Header size without BCC2 \*/

#define BAUDRATE B38400

#define TIMEOUT 3 /\* Number of retries on transmission \*/

#define FRAME\_MAX\_SIZE DATA\_FRAME\_SIZE \* 2 + 6 /\* Maximum size for a frame, usually the maximum I frame possible \*/

#define TIMEOUT\_S 3 /\* Number of seconds to wait per try \*/

#define CLOSE\_WAIT 1 /\* Number of seconds to wait before closing serial port \*/

/\*

Enumerators

\*/

typedef enum serialStatus {TRANSMIT, RECEIVE} serialStatus; /\* Current role of the program \*/

typedef enum serialState {INIT, S\_FLAG, E\_FLAG} serialState; /\* Link layer state machine states \*/

typedef enum serialEvent {FLAG\_E, TIMEOUT\_E} serialEvent; /\* Link layer state machine events \*/

typedef enum serialError {NO\_ERR, BCC1\_ERR, CONTROL\_ERR, TIMEOUT\_ERR, FLAG\_ERR} serialError; /\* Link layer possible errors \*/

/\*

Struct containing application layer data

\*/

typedef struct applicationLayer {

int serial\_fd;

serialStatus status;

uint16\_t tx\_counter;

int filesize;

char filename[FILENAME\_SIZE];

uint8\_t data\_frame[DATA\_FRAME\_SIZE];

} applicationLayer;

/\*

Struct containing link layer data

\*/

typedef struct linkLayer {

char port[PORT\_NAME\_SIZE + 1];

uint32\_t baudrate;

uint8\_t sequence\_number;

uint32\_t length;

volatile uint8\_t timeout\_count;

volatile uint8\_t timeout\_flag;

uint16\_t current\_index;

uint8\_t last\_response;

serialState state;

struct termios oldtio;

struct termios newtio;

uint8\_t frame[FRAME\_MAX\_SIZE];

} linkLayer;

#endif /\* GLOBALS\_H \*/

Makefile

CC = gcc

CFLAGS = -Wall -std=gnu99

SRCS = Helper.c AppLayer.c LinkLayer.c

all: $(SRCS)

$(CC) $(CFLAGS) $(SRCS) -lm -o rcom