Faculdade de Engenharia da Universidade do Porto

1.º projeto laboratorial

Desenvolvimento em C de um protocolo para a porta série

Redes de Computadores (L.EIC025) 2024/2025

Licenciatura em Engenharia Informática e Computação

Manuel Alberto Pereira Ricardo (Co-regente do curso)

Rui Pedro de Magalhães Claro Prior (Co-regente do curso)

Eduardo Nuno Moreira Soares de Almeida (Professor das aulas laboratoriais)

Turma 2:

Guilherme Duarte Silva Matos up202208755@up.pt

João Vítor da Costa Ferreira up202208393@up.pt

Índice

Resumo

Este projeto consiste na implementação de um protocolo de transferência de ficheiros na linguagem de programação C, com a intenção de ensinar e demonstrar como dados podem ser transmitidos a partir de uma conexão série.

O desenvolvimento deste projeto e relatório resultou numa aplicação funcional que permite a transmissão e receção de ficheiros a partir de uma porta série, bem como uma análise estatística da eficiência do mesmo protocolo.

1. Introdução

Este relatório visa documentar o projeto prático realizado ao longo das aulas laboratoriais na construção em C de um protocolo *"Stop & Wait"* para a porta série, incluindo:

- (2.) Como o código do projeto está estruturado;
- (3.) A utilização do programa;
- (4. e 5.) Como estão construídas as camadas da aplicação e de *"Logical Link"*, bem como a separação clara das suas responsabilidades;
- (6.) Os testes realizados para confirmar a validade do programa nos ambientes mais difíceis;
- (7.) A eficiência do protocolo e como esta se compara à expectativa teórica.

2. Arquitetura e Estrutura de Código

O código, funcionalmente, está dividido em duas entidades: **Emissor** e **Recetor**. Para facilitar a reutilização de código, foram identificadas as seguintes **funcionalidades comuns** às duas principais entidades:

- Análise e processamento de uma trama da camada "Logical Link";
- A capacidade de iniciar uma conexão;
- A capacidade de terminar uma conexão.

Em termos práticos, foi possível isolar a **máquina de estados** do restante código da camada *"Logical Link"*, enquanto a conexão, embora não completamente isolada e independente, também foi integrada na máquina de estados.

A máquina de estados possui, portanto, a seguinte API pública:

- `ResultFrame sm_next(unsigned char)`: interface principal que processa o byte recebido;
- `void provide_buf(DecodedData*)`: no caso de uma trama de informação, é possível dar à máquina de estados o *"buffer"* onde se escrevem os dados;
- `void start_conn()`: função que ativa a conexão interna;
- `void close_conn()`: função que desativa a conexão interna;

Em termos de arquitetura, foram definidos os seguintes tipos:

- `DecodedData`: tipo que funciona como uma *"view"* de um *"buffer"* previamente

alocado;

- `AddressTransaction`: tipo responsável por identificar qual das entidades iniciou uma transação;
- `ResultFrameState`: tipo responsável por identificar em que estado de processamento se encontra a análise de uma trama;
- `Frame`: o tipo que representa uma trama;
- `ResultFrame`: tipo que descreve a saída da máquina de estados: uma trama completa, incompleta ou um erro (caso um seja detetado).

Na camada *"Logical Link"*, existem as funções `llopen()` e `llclose()`, cujas funcionalidades são de estabelecer e terminar conexões, respetivamente, utilizando funcionalidades da máquina de estados.

Identificadas as partes comuns às duas entidades, é possível identificar a parte disjunta às mesmas. Na camada de *"Logical Link"* existem duas funções que pertencem exclusivamente a uma das entidades:

- Para o Emissor, existe a função `llwrite()` para enviar tramas de informação e receber as respostas do Recetor utilizando a máquina de estados.
- Para o Recetor, existe a função `llread()` que recebe, utilizando a máquina de estados, tramas provenientes do Emissor, e responde com a verificação de receção e de integridade dessas mesmas tramas.

Por fim, as entidades são unificadas na camada de aplicação: a função `applicationLayer()`, que seleciona como a aplicação se irá comportar: ou como um Recetor ou como um Emissor.

3. Principais Casos de Uso

O projeto tem como principais usos o envio e a receção de ficheiros a partir de uma porta série. A interação do utilizador com o programa iguala o modelo dado pela cadeira:

- Para **dar início à aplicação** como Emissor, execute na raiz do projeto `make run_tx` ou `make run_rx` para dar início como Recetor.
- Para modificar parâmetros como as portas usadas, o *"baud rate"* ou o ficheiro usado, modifique as seguintes linhas no `Makefile`:

```
Unset
TX_SERIAL_PORT = /dev/ttyS10
RX_SERIAL_PORT = /dev/ttyS11

BAUD_RATE = 9600

TX_FILE = penguin.gif
RX_FILE = penguin-received.gif
```

Ao iniciar o programa, a função `applicationLayer()` será chamada, o que resultará nesta sequência:

- Configurar a porta série a partir dos parâmetros dados;
- Abrir e assegurar uma conexão ativa entre o emissor e receptor;
- Abrir o ficheiro com as permissões adequadas;
- Iniciar a transmissão ou recepção de pacotes;
- Ler ficheiro e transmitir ou receber dados nos pacotes e gravar no ficheiro;
- Fechar a conexão e o ficheiro.

4. Protocolo “Logical Link”

A camada *“Logical Link”* tem as seguintes exclusivas responsabilidades:

- Construir tramas de informação a partir de dados;
- Recolher dados a partir de tramas de informação;
- Assegurar que a sequência de tramas é respeitada;
- Estabelecer uma conexão inicial entre o emissor e recetor;
- Indicar a ambos que a conexão irá ser fechada;
- Enviar e receber bytes da porta série;
- Enviar bytes para a máquina de estados, receber tramas e lidar com erros;
- Ativar, desativar e lidar com alarmes.

4.1. `llopen()`

Tem como função abrir a porta série e assegurar que o emissor e que o recetor estão ativos.

Emissor	Recetor

- Envia SET
- Recebe UA e ignora outras tramas
- Reinicia transmissão a cada alarme

- Recebe SET e ignora outras tramas
- Envia UA

4.2. `llwrite()`

Função invocada pelo Emissor, responsável pelo envio de uma trama de informação, verificando a receção pelo lado do Recetor.

Emissor

- Envia trama de informação (0 ou 1 dependendo da variável global `write_are_we_info_1`)
- Espera receber RR (1 ou 0):
  - Se receber o esperado, retorna sem erros;
  - Se receber RR com a numeração errada, ignora a trama;
  - Se receber Reject, reenvia a trama imediatamente;
  - Ignora outras tramas que não esteja à espera.
- Reinicia transmissão a cada alarme

4.3. `llread()`

Função invocada pelo Recetor, responsável pela receção de uma trama de informação, enviando uma verificando de receção para o Emissor.

Recetor

- Espera receber uma trama de informação (0 ou 1 dependendo da variável global read_are_we_info_1):
  - Se receber o esperado, envia RR (1 ou 0);
  - Se receber Info com a numeração errada, ignora a trama;
  - Se receber SET, envia UA (o pacote UA do `llopen()` não foi recebido);
  - Ignora outras tramas que não esteja à espera.

4.4. `llclose()`

Tem como função fechar a porta série e assegurar que o emissor e que o recetor tornam-se inativos.

Emissor	Recetor

• Envia DISC
• Espera receber DISC
  ○ Se receber o esperado, envia UA e fecha a conexão com sucesso;
  ○ Ignora outras tramas que não esteja à espera.
• Reinicia transmissão a cada alarme
	• Espera receber DISC
	  ○ Se receber UA, ignora a trama;
	  ○ Ignora outras tramas que não esteja à espera.
	• Envia DISC
	• Espera receber UA
	  ○ Se receber UA, fecha a conexão com sucesso;
	  ○ Se receber DISC, envia DISC;
	  ○ Ignora outras tramas que não esteja à espera.

5. Protocolo da "Camada de Aplicação"

A camada de aplicação é responsável pela comunicação de alto nível entre as entidades Recetor e Emissor. A sua implementação segue o protocolo descrito pela cadeira, e como tal é responsável pelas seguintes tarefas:

- Abrir um ficheiro;
- No caso do emissor:
  - Enviar um pacote de início de transmissão com o tamanho do ficheiro, representado com o número mínimo possível de bytes;
  - Repetidamente ler até 996 bytes de cada vez do ficheiro (devido à restrição de que um *"payload"* tem um tamanho máximo de 1000 bytes, e retirando os bytes do cabeçalho, tem-se 996 bytes) e enviar o pacote para a camada *"Logical Link"*;
  - Enviar um pacote de término de transmissão com os mesmos dados;
- No caso do recetor:
  - Receber o pacote de início proveniente da camada *"Logical Link"*, e processar o mesmo para obter o tamanho do ficheiro de destino;
  - Receber repetidamente os pacotes que contêm os dados e escrever os dados

no ficheiro de destino;
    - Receber o pacote de término e confirmar que as informações recebidas são iguais às do pacote de início;
- Fechar o ficheiro;

6. Validação

Ao longo do desenvolvimento do projeto foram realizados os seguintes testes com sucesso, isto é, o recetor recolher o ficheiro completo e sem falhas:

- Executar normalmente, isto é, primeiro o recetor e depois o emissor com *"baud rate"*, *"BER"* e tempo de propagação padrões;
- Executar o emissor segundos antes do recetor;
- Executar o emissor apenas. Neste caso, o resultado esperado é que os três alarmes sejam disparados e que o programa retorne −1;
- Desconectar momentaneamente a porta série, quer no mesmo período de um alarme ou entre alarmes;
- Executar para valores compatíveis de *"baud rates"*, *"BER"*, e tempos de propagação;

Para testar que a receção ocorreu como esperada, execute `make check_files`; o resultado deverá indicar que os ficheiros são idênticos:

```
Unset
$ make check_files
diff -s penguin.gif penguin-received.gif || exit 0
diff: penguin-received.gif: No such file or directory
```

O projeto tem suporte para vários níveis de mensagens de erro e de informação úteis para testar e compreender os módulos do projeto. O apêndice A entra em mais detalhes sobre como ativar estas mensagens de depuração.

7. Eficiência do protocolo "data link"

A eficiência teórica de um protocolo *"Stop & Wait"* é calculada abaixo para um dado tempo de transmissão (T_t) e um tempo de propagação (T_{prop}):

$$S_{teórico} = \frac{T_t}{T_t + 2 \times T_{prop}}$$

A eficiência experimental do protocolo é calculada medindo a capacidade de transmissão experimental recebida (R), comparada com o *"baud rate"* da porta (C). Também pode ser expressa medindo o tempo de receção de todos os bytes de uma trama pelo recetor sem atraso de propagação ($T_{útil}$) e o tempo que demora para o emissor enviar uma trama e a receber a sua confirmação (T_{trama}):

$$S_{experimental} = \frac{R}{C} = \frac{T_{útil}}{T_{trama}}$$

A medição da eficiência experimental terá as seguintes assunções e metodologia:

- O $T_{útil}$ é calculado pela duração de um recetor da porta série receber 1020 bytes;
- O T_{trama} é calculado pela duração completa do `llwrite()`. É usado uma média aritmética de 11 tramas transmitidas com o conteúdo de `penguin.gif` (incluído no modelo), o que resulta em média 1020 bytes por trama;
- Todas as medições assumem um *"BER"* nulo;
- As medições serão realizadas com o auxílio da biblioteca *"sys/time.h"* e do *"cable"*: o simulador de porta série disponibilizado pela cadeira.

Tendo isto em conta, abaixo encontram-se todas as medidas experimentais em conjunto com as eficiências teóricas respetivas:

"Baudrate" *(bits/s)*	T_{prop} (µs)	T_t (µs)	$S_{teórico}$ (%)	$T_{útil}$ (µs)	T_{trama} (µs)	$S_{experimental}$ (%)
4 800	122 916	2 083 333	89,445	2 093 590	2 366 629	88,463
4 800	10 416	2 083 333	99,010	2 093 590	2 156 286	97,092
4 800	2 083	2 083 333	99,800	2 093 590	2 139 575	97,851
9 600	122 916	1 041 667	80,906	1 046 857	1 313 756	79,684
9 600	10 416	1 041 667	98,039	1 046 857	1 088 607	96,165
9 600	2 083	1 041 667	99,602	1 046 857	1 071 856	97,668
19 200	122 916	520 833	67,935	523 403	779 620	67,136
19 200	10 416	520 833	96,154	523 403	554 679	94,361
19 200	2 083	520 833	99,206	523 403	538 006	97,286

Conclui-se que, tal como previsto nos cálculos teóricos, o tempo de propagação tem um efeito negativo na eficiência do protocolo, sendo este efeito mais notável em transmissões com "*baud rates*" elevados. Um "baud rate" elevado também diminui a eficiência do protocolo.

8. Conclusões

No âmbito da construção de um protocolo de transferência de ficheiros para uma porta série, este relatório apresentou como este projeto desenvolvido durante as aulas laboratoriais funciona, como está organizado, a sua validação e eficiência.

Este projeto foi uma ótima introdução a conceitos muito importantes em redes de computadores, não obstante, à redundância de uma trama para evitar perdas de dados, a importância da separação de responsabilidades entre as várias camadas de um protocolo e a capacidade de recuperar transmissões em conexões instáveis.

O projeto foi cumprido, tendo sido todos os objetivos principais alcançados.

Apêndices

A. Ferramentas de depuração

O projeto contém mensagens de erro e de depuração em todos os módulos do projeto com a intenção de facilitar a compreensão dos diferentes estados do programa e de possíveis falhas.

Estão disponíveis no projeto seis diferentes classes de mensagens:

- **Mensagens de erro** (visíveis obrigatoriamente);

```
Unset
[ERROR] Failed to open serial port
```

- Depuração da **camada da aplicação** (ativa por padrão): os tipos e a sequência de pacotes enviados ou à espera de serem recebidos;

```
Unset
[AL] Waiting for Data packet nº07
```

- Depuração da ***"link layer"***: os tipos de tramas enviadas ou à espera de serem recebidas pela porta série;

```
Unset
[LL] ↓ SET; ↑ UA; connection established
```

- Depuração de **alarmes** (ativa por padrão): quando um alarme foi acionado e a sua numeração;

```
Unset
[ALARM] Triggered alarm nº2
```

- Depuração da **máquina de estados**: todas as mudanças de estados, bem como o byte que causou tal mudança;

```
Unset
[SM] AwaiCtlFlag -> ParseSucc (caused by 0x7E)
```

- Depuração da **porta série**: os bytes recebidos pela porta série e o tipo de trama a que correspondem.

```
Unset
0x7E 0x03 0x07 0x04 0x7E UA frame
```

Todas as mensagens de depuração podem ser ativadas ou desativadas ao alterar o comentário da linha correspondente no debug.h:

```
C/C++
// Comment/uncomment these lines to disable/enable debug messages for each layer or component
#define DEBUG_APPLICATION_LAYER
// #define DEBUG_LINK_LAYER
#define DEBUG_ALARM
// #define DEBUG_STATE_MACHINE
// #define DEBUG_SERIAL_PORT
```

B. Código-fonte

alarm.h

```C/C++
/**
 * @file alarm.h
 * @brief Alarm abstraction used by the transmitter
 * @authors Guilherme Matos, João Ferreira
 */

#ifndef _ALARM_H_
#define _ALARM_H_

#include <stdbool.h>

// Initializes the alarm handler and sets the number of retries and timeout.
void alarmInit(int maxRetries, int timeout);
// Starts the alarm, i.e, calls alarm(TIMEOUT).
void alarmOn();
// Stops the alarm, i.e, calls alarm(0).
void alarmOff();
// Resets the alarm counter and stops the alarm.
void alarmReset();
// Returns true if the alarm is enabled.
bool alarmIsEnabled();
// Returns true if the maximum number of retries was reached.
bool alarmMaxRetriesReached();

#endif // _ALARM_H_
```

alarm.c

```C/C++
#include "../include/alarm.h"
#include "../include/debug.h"
#include <signal.h>
#include <unistd.h>
```

```c
// The maximum number of retries before giving up.
int maxRetries;
// Seconds to wait before the alarm triggers.
int timeout;
// If the alarm is enabled or not.
bool alarmEnabled = false;
// The number of times the alarm has been triggered by the SIGALRM signal.
int alarmCount = 0;

void alarmHandler(int signal) {
  alarmEnabled = false;
  alarmCount++;
  statisticsTimeout();
  debug_am("Triggered alarm nº%d", alarmCount);
}

// ═══════════════════════════════════════════════════════════

void alarmInit(int mr, int to) {
  (void)signal(SIGALRM, alarmHandler);
  maxRetries = mr;
  timeout = to;
}

void alarmOn() {
  alarm(timeout);
  alarmEnabled = true;
}

void alarmOff() {
  alarm(0);
  alarmEnabled = false;
}

void alarmReset() {
  alarmCount = 0;
  alarmOff();
}
```

```
bool alarmIsEnabled() { return alarmEnabled; }

bool alarmMaxRetriesReached() { return alarmCount ⩾ maxRetries; }
```

debug.h

```C/C++
/**
 * @file debug.h
 * @brief Auxiliary functions for printing debug messages
 * @authors Guilherme Matos, João Ferreira
 */

#ifndef _DEBUG_H_
#define _DEBUG_H_

// Comment/uncomment these lines to disable/enable debug messages for each layer or component
#define DEBUG_APPLICATION_LAYER
// #define DEBUG_LINK_LAYER
#define DEBUG_ALARM
// #define DEBUG_STATE_MACHINE
// #define DEBUG_SERIAL_PORT

#include "field.h"
#include "state_machine.h"

// Print an optional debug message for the application layer
void debug_al(const char *msg, ...);
// Print an optional debug message for the link layer
void debug_ll(const char *msg, ...);
// Print an optional debug message for the alarm
void debug_am(const char *msg, ...);
// Print an optional debug message for the state machine
void debug_sm(const char *msg, ...);
// Print an optional debug message for the serial port
void debug_sp(const char *msg, ...);
// Print a mandatory error message and exit the program
```

```
void error(const char *msg, ...);
// Convert a frame type into a human-readable string
const char *frame_command_to_string(FrameCommandField command);
// Convert a state into a human-readable string
const char *state_to_string(int state);

//
////////////////////////////////////////////////////////////////////////////////

// Struct for the statistics printed in llclose()
typedef struct {
  unsigned int packets;           // Number of packets sent/received
  unsigned int retransmissions;   // Number of retransmissions made
  unsigned int timeouts;          // Number of timeouts made by the alarm
} Statistics;

// Increment the number of packets sent/received
void statisticsPacket();
void statisticsRetransmission();
void statisticsTimeout();
void statisticsPrint(bool isTransmitter);

#endif // _DEBUG_H_
```

debug.c

```
C/C++
#include "../include/debug.h"
#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

void debug_al(const char *msg, ...) {
#ifdef DEBUG_APPLICATION_LAYER
  va_list args;
  va_start(args, msg);
  printf("[AL] ");
```

```
  vprintf(msg, args);
  printf("\n");
  va_end(args);
#endif
}

void debug_ll(const char *msg, ...) {
#ifdef DEBUG_LINK_LAYER
  va_list args;
  va_start(args, msg);
  printf("[LL] ");
  vprintf(msg, args);
  printf("\n");
  va_end(args);
#endif
}

void debug_am(const char *msg, ...) {
#ifdef DEBUG_ALARM
  va_list args;
  va_start(args, msg);
  printf("[ALARM] ");
  vprintf(msg, args);
  printf("\n");
  va_end(args);
#endif
}

void debug_sm(const char *msg, ...) {
#ifdef DEBUG_STATE_MACHINE
  va_list args;
  va_start(args, msg);
  printf("[SM] ");
  vprintf(msg, args);
  printf("\n");
  va_end(args);
#endif
}
```

```
void debug_sp(const char *msg, ...) {
#ifdef DEBUG_SERIAL_PORT
  va_list args;
  va_start(args, msg);
  // printf("[SP] ");
  vprintf(msg, args);
  // printf("\n");
  va_end(args);
#endif
}

void error(const char *msg, ...) {
  va_list args;
  va_start(args, msg);
  fprintf(stderr, "[ERROR] ");
  vfprintf(stderr, msg, args);
  fprintf(stderr, "\n");
  va_end(args);
  exit(-1);
}

const char *frame_command_to_string(FrameCommandField command) {
  switch (command) {
  case Set:
    return "Set";
  case UnNumberedAck:
    return "UA";
  case Accept0:
    return "RR0";
  case Accept1:
    return "RR1";
  case Reject0:
    return "REJ0";
  case Reject1:
    return "REJ1";
  case Disconnect:
    return "DISC";
  case Info0:
    return "Info0";
```

```
  case Info1:
    return "Info1";
  }
return "Unknown";
}

const char *state_to_string(int state) {
  switch (state) {
  case 0:
    return "AwaitStart";
  case 1:
    return "AwaitA";
  case 2:
    return "AwaitC";
  case 3:
    return "AwaitBcc1";
  case 4:
    return "AwaiCtlFlag";
  case 5:
    return "ProvideBuf";
  case 6:
    return "Data";
  case 7:
    return "Escaped";
  case 8:
    return "ParseSucc";
  case 9:
    return "ConnClosed";
  }
  return "Unknown";
}

// ═══════════════════════════════════════════════════════════════

Statistics statistics = {0, 0, 0};

void statisticsPacket() { statistics.packets++; }

void statisticsRetransmission() { statistics.retransmissions++; }
```

```
void statisticsTimeout() { statistics.timeouts++; }

void statisticsPrint(bool isTransmitter) {
  printf("Statistics:\n");
  if (isTransmitter) {
printf("- %d packets sent\n", statistics.packets);
printf("- %d retransmissions made\n", statistics.retransmissions);
printf("- %d timeouts\n", statistics.timeouts);
  } else {
printf("- %d packets received\n", statistics.packets);
printf("- %d retransmissions requested\n", statistics.retransmissions);
  }
}
```

field.h

```
C/C++
/**
 * @file field.h
 * @brief Field definitions used inside frames
 * @authors Guilherme Matos, João Ferreira
 */

#ifndef _FIELD_H_
#define _FIELD_H_

// Flag: Synchronisation for the start or end of a frame
#define FLAG        0x7E
// Address field in frames that are commands sent by the Transmitter or
replies sent by the Receiver
#define ATCV        0x03
// Address field in frames that are commands sent by the Receiver or replies
sent by the Transmitter
#define ARCV        0x01
// SET frame: sent by the transmitter to initiate a connection
#define CMD_SET     0x03
// UA frame: confirmation to the reception of a valid supervision frame
```

```
#define CMD_UA      0x07
// RR0 frame: indication sent by the Receiver that it is ready to receive an information frame number 0
#define CMD_RR0     0xAA
// RR1 frame: indication sent by the Receiver that it is ready to receive an information frame number 1
#define CMD_RR1     0xAB
// REJ0 frame: indication sent by the Receiver that it rejects an information frame number 0 (detected an error)
#define CMD_REJ0    0x54
// REJ1 frame: indication sent by the Receiver that it rejects an information frame number 1 (detected an error)
#define CMD_REJ1    0x55
// DISC frame to indicate the termination of a connection
#define CMD_DISC    0x0B
// Information frame number 0
#define CMD_INF0    0x00
// Information frame number 1
#define CMD_INF1    0x80
// Escape character (see slide 17)
#define ESCAPE      0x7D


#define __FCFsize 9
//  The types of frames that can be sent either by the Receiver or the Transmitter
typedef enum {
  Set = CMD_SET,
  UnNumberedAck = CMD_UA,
  Accept0 = CMD_RR0,
  Accept1 = CMD_RR1,
  Reject0 = CMD_REJ0,
  Reject1 = CMD_REJ1,
  Disconnect = CMD_DISC,
  Info0 = CMD_INF0,
  Info1 = CMD_INF1,
} FrameCommandField;
```

```
// Control field for the start of a transmission
#define CTRL_START  0×01
// Control field for data packets
#define CTRL_DATA   0×02
// Control field for the end of a transmission
#define CTRL_END    0×03
// Type field for the file size in a control packet
#define CTRL_FILE_SIZE 0×00
// File name
#define CTRL_FILE_NAME 0×01

#endif // _FIELD_H_
```

state_machine.h

```
C/C++
/**
 * @file state_machine.h
 * @brief General State Machine abstraction
 * @authors Guilherme Matos, João Ferreira
 */

#ifndef _STATE_MACHINE_H_
#define _STATE_MACHINE_H_
#include "field.h"
#include "link_layer.h"
#include <stdbool.h>

typedef unsigned char u8;
typedef struct conn* Connection;

typedef struct {
  u8 *buf;
  unsigned short size;
  unsigned short written;
} DecodedData;

typedef enum {
```

```
  /// Transceiver initiated
  TcvInit = 1,
  /// Receiver initiated
  RcvInit = 2,
} AddressTransaction;

typedef struct {
   AddressTransaction addr;
   FrameCommandField tag;
   // Optional, auto NULL
   DecodedData* buf;
} Frame;

typedef enum {
Ok,
Unfinished,
ErrClosed,
ErrReject,
ErrProvideBuf,
ErrStateMachineBug,
} ResultFrameState;

typedef struct {
ResultFrameState state;
// Optional, auto NULL
union {
       // OK
       struct { Frame frame; };
};
} ResultFrame;

ResultFrame sm_next(u8 byte);

void provide_buf(DecodedData* buf);

void start_conn();

void close_conn();
```

```
#endif // _STATE_MACHINE_H_
```

state_machine.c

```C/C++
#include "../include/state_machine.h"
#include "../include/debug.h"
#include <assert.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct conn *Connection;

typedef enum {
  /// Waiting for flag
  AwaitStart,
  /// Waiting for Address
  AwaitA,
  /// Waiting for Control
  AwaitC,
  /// Waiting for Bcc1
  AwaitBcc1,
  /// Waiting for flag in a supervision frame
  AwaiCtlFlag,
  /// User needs to provide a buffer to write the data to.
  ProvideBuf,
  /// Receiving Data
  Data,
  /// Escape the next byte
  Escaped,
  /// Parsing Successful
  ParseSucc,
  /// Connection Closed
  ConnClosed,
  /// Number of enum variants
  __ValidStateSize,
} ValidState;
```

```
typedef struct sm {
  u8 xor_accum;
  ValidState state;
  Frame frame;
} StateMachine;

struct conn {
  LinkLayerRole role;
};

StateMachine sm = {
.xor_accum = 0,
.state = ConnClosed,
.frame = {0},
};

struct conn connection_singleton = {0};

Connection active_connection = NULL;

ValidState prev_state = ConnClosed;

ResultFrameState __await_start(u8 byte) {
  if (sm.state != AwaitStart) {
return ErrStateMachineBug;
  }
  if (active_connection == NULL) {
sm.state = ConnClosed;
return ErrClosed;
  }
  if (byte == FLAG) {
sm.state = AwaitA;
  }
  return Ok;
}

ResultFrameState __await_a(u8 byte) {
  if (sm.state != AwaitA) {
```

```
return ErrStateMachineBug;
  }

  if (active_connection == NULL) {
return ErrClosed;
  }

  if (byte == FLAG) {
return Ok;
  }

  if (byte != ARCV && byte != ATCV) {
sm.state = AwaitStart;
return Ok;
  }
  sm.frame.addr = byte;
  sm.xor_accum ^= byte;
  sm.state = AwaitC;
  return Ok;
}

ResultFrameState __await_c(u8 byte) {
  if (sm.state != AwaitC) {
return ErrStateMachineBug;
  }
  if (active_connection == NULL) {
sm.state = ConnClosed;
return ErrClosed;
  }
  FrameCommandField command = byte;
  sm.frame.tag = command;
  sm.xor_accum ^= byte;
  switch (command) {
  case Set:
  case UnNumberedAck:
  case Accept0:
  case Accept1:
  case Reject0:
  case Reject1:
```

```
  case Disconnect:
  case Info0:
  case Info1:
sm.state = AwaitBcc1;
return Ok;
  default:
sm.frame.tag = 0;
if (byte == FLAG) {
       sm.state = AwaitA;
       sm.xor_accum = 0;
       return Ok;
} else {
       sm.state = AwaitStart;
       sm.xor_accum = 0;
       return Ok;
}
  }
}

ResultFrameState __await_bcc1(u8 byte) {
  if (sm.state != AwaitBcc1) {
return ErrStateMachineBug;
  }

  if (active_connection == NULL) {
sm.state = ConnClosed;
return ErrClosed;
  }

  sm.xor_accum ^= byte;
  if (sm.xor_accum != 0) {
sm.xor_accum = 0;
if (byte == FLAG) {
       sm.state = AwaitA;
       sm.xor_accum = sm.frame.addr;
} else {
       sm.state = AwaitStart;
       sm.xor_accum = 0;
}
```

```
return Ok;
  }
  switch (sm.frame.tag) {
  case Set:
  case UnNumberedAck:
  case Accept0:
  case Accept1:
  case Reject0:
  case Reject1:
  case Disconnect:
sm.state = AwaiCtlFlag;
return Ok;
  case Info0:
  case Info1:
sm.state = ProvideBuf;
return Ok;
  }
  return ErrStateMachineBug;
}

ResultFrameState __await_ctl_flag(u8 byte) {
  if (sm.state != AwaiCtlFlag) {
return ErrStateMachineBug;
  }
  if (active_connection == NULL) {
sm.state = ConnClosed;
return ErrClosed;
  }
  if (byte == FLAG) {
sm.state = ParseSucc;
  } else {
sm.state = AwaitStart;
  }
  return Ok;
}

ResultFrameState __data(u8 byte) {
  if (sm.state != Data) {
return ErrStateMachineBug;
```

```
  }
  if (active_connection == NULL) {
sm.state = ConnClosed;
return ErrClosed;
  }

  if (byte == FLAG) {
if (sm.xor_accum == 0) {
       sm.state = ParseSucc;
       sm.frame.buf->written--;
       return Ok;
} else {
       sm.state = AwaitA;
       sm.xor_accum = 0;
       sm.frame.buf->written = 0;
       return ErrReject;
}
  }

  if (sm.frame.buf->size == sm.frame.buf->written) {
sm.xor_accum = 0;
sm.state = AwaitStart;
sm.frame.buf->written = 0;
return ErrReject;
  }

  if (byte == ESCAPE) {
sm.state = Escaped;
return Ok;
  }

  sm.xor_accum ^= byte;
  sm.frame.buf->buf[sm.frame.buf->written] = byte;
  sm.frame.buf->written++;
  return Ok;
}

ResultFrameState __escaped(u8 byte) {
  if (sm.state != Escaped) {
```

```
return ErrStateMachineBug;
  }
  if (active_connection == NULL) {
sm.state = ConnClosed;
return ErrClosed;
  }
  u8 res = byte ^ 0x20;
  sm.xor_accum ^= res;
  sm.frame.buf->buf[sm.frame.buf->written] = res;
  sm.frame.buf->written++;
  sm.state = Data;
  return Ok;
}

ResultFrameState __parse_succ(u8 byte) {
  if (sm.state != ParseSucc) {
return ErrStateMachineBug;
  }
  if (active_connection == NULL) {
sm.state = ConnClosed;
return ErrClosed;
  }
  sm.frame.addr = 0;
  sm.frame.tag = 0;
  sm.frame.buf = NULL;
  sm.state = AwaitStart;
  __await_start(byte);
  return Ok;
}

ResultFrameState __conn_closed(u8 byte) { return ErrClosed; }

ResultFrameState __provide_buf(u8 byte) { return ErrProvideBuf; }

ResultFrameState (*const STATEFN[__ValidStateSize])(u8) = {
[AwaitStart] = __await_start,
[AwaitA] = __await_a,
[AwaitC] = __await_c,
[AwaitBcc1] = __await_bcc1,
```

```
[AwaiCtlFlag] = __await_ctl_flag,
[ProvideBuf] = __provide_buf,
[Data] = __data,
[Escaped] = __escaped,
[ParseSucc] = __parse_succ,
[ConnClosed] = __conn_closed,
};

ResultFrame sm_next(u8 byte) {
  debug_sp("0x%02X ", byte);
  ResultFrameState state = STATEFN[sm.state](byte);
  if (sm.state != prev_state) {
debug_sm("%s -> %s (caused by 0x%02X)", state_to_string(prev_state),
             state_to_string(sm.state), byte);
prev_state = sm.state;
  }
  switch (state) {
  case Ok:
switch (sm.state) {
case AwaitStart:
case AwaitA:
case AwaitC:
case AwaitBcc1:
case AwaiCtlFlag:
case Data:
case Escaped:
       return (ResultFrame){.state = Unfinished};
case ProvideBuf:
       return (ResultFrame){.state = ErrProvideBuf};
case ParseSucc:
       debug_sp("%s frame", frame_command_to_string(sm.frame.tag));
       return (ResultFrame){.state = Ok, .frame = sm.frame};
case ConnClosed:
       return (ResultFrame){.state = ErrClosed};
case __ValidStateSize:
       // Unreachable!
       exit(101);
}
break;
```

```
  case Unfinished:
return (ResultFrame){.state = Unfinished};
  case ErrClosed:
return (ResultFrame){.state = ErrClosed};
  case ErrReject:
return (ResultFrame){.state = ErrReject, .frame = sm.frame};
  case ErrProvideBuf:
return (ResultFrame){.state = ErrProvideBuf};
  case ErrStateMachineBug:
exit(101);
  }
  return (ResultFrame){.state = ErrStateMachineBug};
}

void provide_buf(DecodedData *buf) {
  sm.frame.buf = buf;
  sm.state = Data;
}

void start_conn() {
  active_connection = &connection_singleton;
  sm.state = AwaitStart;
}

void close_conn() {
  active_connection = NULL;
  sm.state = ConnClosed;
}
```

serial_port.h

Este ficheiro é fornecido no modelo e não foi alterado.

serial_port.c

Este ficheiro é fornecido no modelo, no entanto, a configuração da porta série foi alterada, tal como sugerido nas aulas práticas:

```C/C++
[...]
newtio.c_cc[VMIN] = 0;  // [!!!] Non-blocking / Instant
[...]
```

link_layer.h

Este ficheiro é fornecido no modelo e não foi alterado.

link_layer.c

```C/C++
// Link layer protocol implementation

#include "../include/link_layer.h"
#include "../include/alarm.h"
#include "../include/debug.h"
#include "../include/serial_port.h"
#include "../include/state_machine.h"
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source

// Internal Buffer
unsigned char ibuf[MAX_PAYLOAD_SIZE + 1] = {0};
LinkLayerRole role;
FrameCommandField prev_frame_type;

// Construct a supervision or unnumbered frame, given the address and control
// fields.
#define                         CREATE_BUF(addr,                         ctl)
\
  (unsigned char[5]) { FLAG, (addr), (ctl), (addr) ^ (ctl), FLAG }
```

```
// Auxiliary wrapper to write bytes to the serial port safely.
int writeBytesSP(const unsigned char *bytes, int numBytes) {
  int written = 0;
  int total_written = 0;
  while (true) {
// UNSAFE: ptr arithmetic. should be safe though.
written =
       writeBytesSerialPort(bytes + total_written, numBytes - total_written);
if (written == -1) {
       return -1;
}
total_written += written;
if (total_written == numBytes) {
       return 0;
}
  }
}

// Auxiliary function to build and write an information frame
bool write_are_we_info_1 = false;
int write_one_info_frame(const unsigned char *buf, const int bufSize) {
  unsigned char xor_accum = 0;
  FrameCommandField info = write_are_we_info_1 ? Info1 : Info0;
  unsigned char header[4] = {
       FLAG,
       ATCV,
       info,
       info ^ ATCV,
  };
  if (writeBytesSP(header, 4))
return -1;
  int totalwritten = 0;
  int ibufwritten = 0;
  int ebufread = 0;
   // Using ibuf as the internal buffer for writting allows us to keep the
size
  // smaller, since, by optimization, we should see many less FLAG or ESCAPE
   // than any other bytes, so it is safe to optimize like this. In case we
can't
```

```
  // fit everything inside, we at most write twice.
  while (ebufread < bufSize) {
xor_accum ^= buf[ebufread];
if (buf[ebufread] != ESCAPE && buf[ebufread] != FLAG) {
       ibuf[ibufwritten] = buf[ebufread];
       ibufwritten++;
} else {
       //  this  is  always  safe  because  the  actual  buffer  size  is
MAX_PAYLOAD_SIZE
       // + 1, and we cap it at MAX_PAYLOAD_SIZE. Worst case we get a full
       // buffer.
       ibuf[ibufwritten] = ESCAPE;
       ibufwritten++;
       ibuf[ibufwritten] = buf[ebufread] ^ 0x20;
       ibufwritten++;
}
ebufread++;
totalwritten++;
// Flush
if (ibufwritten >= MAX_PAYLOAD_SIZE) {
       if (writeBytesSP(ibuf, ibufwritten))
       return -1;
       ibufwritten = 0;
}
  }
  // Flush remaining
  if (ibufwritten > 0) {
if (writeBytesSP(ibuf, ibufwritten))
       return -1;
  }
  if (writeBytesSP((unsigned char[]){xor_accum, FLAG}, 2))
return -1;
  return 0;
}

// Auxiliary wrapper to read a byte with busy waiting.
static inline int readByteSP(unsigned char *byte) {
  int err = 0;
  while ((err = readByteSerialPort(byte)) == 0)
```

```
;
  return err;
}

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters) {
  debug_ll("Opening connection...");
  if (openSerialPort(connectionParameters.serialPort,
                     connectionParameters.baudRate) < 0) {
return -1;
  }
  alarmInit(connectionParameters.nRetransmissions,
            connectionParameters.timeout);
  role = connectionParameters.role;
  alarmReset();
  start_conn();

  if (role == LlTx) {
while (!alarmMaxRetriesReached()) {
       if (!alarmIsEnabled()) {
       alarmOn();
       if (writeBytesSP(CREATE_BUF(ATCV, CMD_SET), 5))
       return -1;
       debug_ll("↑ SET; ⟳ UA");
       }
       unsigned char byte = 0;
       ResultFrame frame;
       int err = 0;
       if ((err = readByteSerialPort(&byte)) == 0)
       continue;
       if (err != 1)
       return -1;
       frame = sm_next(byte);
       switch (frame.state) {
       case Ok:
       prev_frame_type = frame.frame.tag;
       switch (frame.frame.tag) {
```

```
        case UnNumberedAck:
        alarmOff();
        debug_ll("↓ UA; connection established");
        return 1;
        default: // Set, Accept0, Accept1, Reject0, Reject1, Info0, Info1,
              // Disconnect
        debug_ll("↓ %s; ignored", frame_command_to_string(frame.frame.tag));
        continue;
        }
        case ErrClosed:
        close_conn();
        return -1;
        case ErrStateMachineBug:
        exit(101);
        default: // Unfinished, ErrReject, ErrProvideBuf
        continue;
        }
}
  }

  if (role == LlRx) {
while (true) {
        unsigned char byte = 0;
        ResultFrame frame;
        int err = 0;
        if ((err = readByteSerialPort(&byte)) == 0)
        continue;
        if (err != 1)
        return -1;
        frame = sm_next(byte);
        switch (frame.state) {
        case Ok:
        prev_frame_type = frame.frame.tag;
        switch (frame.frame.tag) {
        case Set:
        debug_ll("↓ SET; ↑ UA; connection established");
        if (writeBytesSP(CREATE_BUF(ATCV, CMD_UA), 5))
              return -1;
        return 1;
```

```
        default: // UnNumberedAck, Accept0, Accept1, Reject0, Reject1, Info0,
                 // Info1, Disconnect
        debug_ll("↓ %s; ignored", frame_command_to_string(frame.frame.tag));
        continue;
        }
        case ErrClosed:
        close_conn();
        return -1;
        case ErrStateMachineBug:
        exit(101);
        default: // Unfinished, ErrReject, ErrProvideBuf
        continue;
        }
}
writeBytesSerialPort(CREATE_BUF(ATCV, CMD_UA), 5);
  }

  close_conn();
  return -1;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////

int llwrite(const unsigned char *buf, int bufSize) {
  statisticsPacket();
  debug_ll("Writing %d bytes...", bufSize);
  while (!alarmMaxRetriesReached()) {
if (!alarmIsEnabled()) {
        alarmOn();
        if (write_one_info_frame(buf, bufSize))
        return -1;
        debug_ll("↑ Info%d; ⟳ %s", write_are_we_info_1,
                 write_are_we_info_1 ? "RR0" : "RR1");
}
unsigned char byte = 0;
ResultFrame frame;
int err = 0;
```

```
if ((err = readByteSerialPort(&byte)) == 0)
        continue;
if (err != 1)
        return -1;
frame = sm_next(byte);
switch (frame.state) {
case Ok:
        prev_frame_type = frame.frame.tag;
        switch (frame.frame.tag) {
        case Accept0:
        if (!write_are_we_info_1) {
        debug_ll("↓ RR0; expected RR1; ignored");
        continue;
        }
        alarmOff();
        write_are_we_info_1 = false;
        debug_ll("↓ RR0; frame transmission successful");
        return bufSize;
        case Accept1:
        if (write_are_we_info_1) {
        debug_ll("↓ RR1; expected RR0; ignored");
        continue;
        }
        alarmOff();
        write_are_we_info_1 = true;
        debug_ll("↓ RR1; frame transmission successful");
        return bufSize;
        case Reject0:
        case Reject1:
        debug_ll("↓ %s; ↑ Info%d", frame_command_to_string(frame.frame.tag),
                write_are_we_info_1);
        statisticsRetransmission();
        alarmReset();
        continue;
        default: // Set, Info0, Info1, Disconnect
        debug_ll("↓ %s; ignored", frame_command_to_string(frame.frame.tag));
        continue;
        }
case ErrClosed:
```

```
        return -1;
case ErrStateMachineBug:
        exit(101);
case ErrProvideBuf:
        exit(169);
default: // Unfinished, ErrReject, ErrProvideBuf
        continue;
}
  }

  return -1;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
bool read_are_we_info_1 = false;

int llread(unsigned char *packet) {
  statisticsPacket();
  debug_ll("⟳ Info%d; reading a new frame...", read_are_we_info_1);
   DecodedData data = {.buf = ibuf, .size = MAX_PAYLOAD_SIZE + 1, .written =
0};
  ResultFrame frame;
  unsigned char byte = 0;
  while (true) {
int err = readByteSP(&byte);
if (err != 1)
        return err;
// advance the state machine
frame = sm_next(byte);
switch (frame.state) {
case Ok:
        prev_frame_type = frame.frame.tag;
        // See what type of frame it is and handle it.
        switch (frame.frame.tag) {
        case Set:
        debug_ll("↓ SET; ↑ UA; connection established");
        if (writeBytesSP(CREATE_BUF(ATCV, UnNumberedAck), 5))
```

```
        return -1;
        continue;
        case UnNumberedAck:
        case Accept0:
        case Accept1:
        case Reject0:
        case Reject1:
        case Disconnect:
        debug_ll("↓                    %s;                    ignoring...",
frame_command_to_string(frame.frame.tag));
        continue;
        case Info0:
        if (read_are_we_info_1) {
        debug_ll("↓ Info0; expected Info1; ignored");
        continue;
        }
        debug_ll("↓ Info0; ↑ RR1");
        memcpy(packet, data.buf, data.written);
        read_are_we_info_1 = true;
        if (writeBytesSP(CREATE_BUF(ATCV, Accept1), 5))
        return -1;
        return data.written;
        case Info1:
        if (!read_are_we_info_1) {
        debug_ll("↓ Info1; expected Info0; ignored");
        continue;
        }
        debug_ll("↓ Info1; ↑ RR0");
        memcpy(packet, data.buf, data.written);
        read_are_we_info_1 = false;
        if (writeBytesSP(CREATE_BUF(ATCV, Accept0), 5))
        return -1;
        return data.written;
        }
case Unfinished:
        continue;
case ErrClosed:
        return -1;
case ErrReject:
```

```
        statisticsRetransmission();
        if (frame.frame.tag == Info0) {
        debug_ll("↑ REJ0; ⟳ Info0; error in transmission");
        if (writeBytesSP(CREATE_BUF(ATCV, Reject0), 5))
        return -1;
        } else {
        debug_ll("↑ REJ1; ⟳ Info1; error in transmission");
        if (writeBytesSP(CREATE_BUF(ATCV, Reject1), 5))
        return -1;
        }
        continue;
case ErrProvideBuf:
        provide_buf(&data);
        continue;
case ErrStateMachineBug:
        exit(101);
}
  }
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics) {
  debug_ll("Closing connection...");
  if (role == LlTx) {
alarmReset();
while (!alarmMaxRetriesReached()) {
        if (!alarmIsEnabled()) {
        alarmOn();
        if (writeBytesSP(CREATE_BUF(ATCV, CMD_DISC), 5))
        return -1;
        debug_ll("↑ DISC; ⟳ DISC");
        }
        unsigned char byte = 0;
        ResultFrame frame;
        int err = 0;
        if ((err = readByteSerialPort(&byte)) == 0)
        continue;
```

```
        if (err != 1)
        return -1;
        frame = sm_next(byte);
        switch (frame.state) {
        case Ok:
        prev_frame_type = frame.frame.tag;
        switch (frame.frame.tag) {
        case Disconnect:
        alarmOff();
        debug_ll("↓ DISC; ↑ UA; connection closed");
        if (writeBytesSP(CREATE_BUF(ARCV, CMD_UA), 5))
              return -1;
        close_conn();
        if (closeSerialPort() == -1)
              return -1;
        if (showStatistics)
              statisticsPrint(true);
        return 1;
        continue;
        default: // Set, Accept0, Accept1, Reject0, Reject1, Info0, Info1,
              // Disconnect
        debug_ll("↓ %s; ignored", frame_command_to_string(frame.frame.tag));
        continue;
        }
        case ErrClosed:
        return -1;
        case ErrStateMachineBug:
        exit(101);
        default: // Unfinished, ErrReject, ErrProvideBuf
        continue;
        }
}
return -1;
  }

  if (role == LlRx) {
bool discReceived = false;
while (true) {
        unsigned char byte = 0;
```

```
ResultFrame frame;
int err = 0;
if ((err = readByteSerialPort(&byte)) == 0)
continue;
if (err != 1)
return -1;
frame = sm_next(byte);
switch (frame.state) {
case Ok:
prev_frame_type = frame.frame.tag;
switch (frame.frame.tag) {
case Disconnect:
debug_ll("↓ DISC; ↑ DISC; ⟳ UA");
discReceived = true;
if (writeBytesSP(CREATE_BUF(ARCV, CMD_DISC), 5))
      return -1;
continue;
case UnNumberedAck:
if (discReceived) {
      debug_ll("↓ UA; connection closed");
      close_conn();
      if (closeSerialPort() == -1)
      return -1;
      if (showStatistics)
      statisticsPrint(false);
      return 1;
}
debug_ll("↓ UA; expected DISC; ignore");
continue;
default: // Set, Accept0, Accept1, Reject0, Reject1, Info0, Info1
debug_ll("↓ %s; ignored", frame_command_to_string(frame.frame.tag));
continue;
}
case ErrClosed:
return -1;
case ErrStateMachineBug:
exit(101);
default: // Unfinished, ErrReject, ErrProvideBuf
continue;
```

```
        }
    }
    return -1;
      }
      return -1;
    }
```

application_layer.h

Este ficheiro é fornecido no modelo e não foi alterado.

application_layer.c

```C/C++
// Application layer protocol implementation

#include "../include/application_layer.h"
#include "../include/debug.h"
#include "../include/field.h"
#include "../include/link_layer.h"
#include <stdbool.h>

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

// Maximum packet data field size per payload
#define MAX_DATA_SIZE MAX_PAYLOAD_SIZE - 4

// Initialize link layer configuration via the input parameters
LinkLayer linkLayerInit(const char *serialPort, const char *role, int
baudRate,
```

```
                    int nTries, int timeout) {
  LinkLayer serialConf;
  strcpy(serialConf.serialPort, serialPort);
  serialConf.baudRate = baudRate;
  serialConf.nRetransmissions = nTries;
  serialConf.timeout = timeout;
  if (strcmp(role, "tx") == 0)
serialConf.role = LlTx;
  else if (strcmp(role, "rx") == 0)
serialConf.role = LlRx;
  else
error("Wrong role parameter");
  return serialConf;
}

// Auxiliary function to get the file size via stat()
unsigned long long getFileSize(const char *filename) {
  struct stat fileStat;
  if (stat(filename, &fileStat) < 0)
error("Failed to get file size");
  return fileStat.st_size;
}

unsigned char min_num_bytes(unsigned long long value) {
  unsigned char cnt = 1;
  value >>= 8;
  while (value) {
value >>= 8;
++cnt;
  }
  return cnt;
}

// Send a control packet with the file size. It can be a start or a end
packet.
void sendControlPacket(unsigned long long fileSize, const char *filename,
                    bool isStart) {
  if (isStart)
debug_al("Sending Control Start packet...");
```

```
  else
debug_al("Sending Control End packet...");
  unsigned char min_bytes = min_num_bytes(fileSize);
     unsigned  char  filename_len  =  strlen(filename)  >  255  ?  255  :
strlen(filename);
  unsigned char controlPacket[3 + 8 + (2 + 255)] = {0};
  if (isStart)
controlPacket[0] = CTRL_START;
  else
controlPacket[0] = CTRL_END;
  controlPacket[1] = CTRL_FILE_SIZE;
  controlPacket[2] = min_bytes;
  for (int i = 0; i < min_bytes; ++i) {
controlPacket[3 + i] = fileSize >> (8 * (min_bytes - i - 1));
  };
  controlPacket[3 + min_bytes] = CTRL_FILE_NAME;
  controlPacket[3 + min_bytes + 1] = filename_len;
  memcpy(&controlPacket[3 + min_bytes + 2], filename, filename_len);

  int bytesSent = llwrite(controlPacket, 3 + min_bytes + 2 + filename_len);
  if (bytesSent == -1)
error("Failed to write Control packet to the serial port");
  if (bytesSent != 3 + min_bytes + 2 + filename_len)
error("The number of bytes sent is not the same number of bytes of the "
      "control packet");
}

// Send a data packet to the link layer with the data
void sendDataPacket(unsigned char *data, int dataSize, int sequenceNumber) {
  debug_al("Sending Data packet nº%02d...", sequenceNumber);
  unsigned char dataPacket[MAX_PAYLOAD_SIZE] = {0};
  dataPacket[0] = CTRL_DATA;
  dataPacket[1] = sequenceNumber;
  dataPacket[2] = dataSize >> 8;
  dataPacket[3] = dataSize;
  memcpy(dataPacket + 4, data, dataSize);
  int bytesSent = llwrite(dataPacket, dataSize + 4);
  if (bytesSent == -1)
error("Failed to write Data packet to the serial port");
```

```
  if (bytesSent != dataSize + 4)
error("The number of bytes sent is not the same number of bytes of the "
       "data packet");
}

// Read a Control Start packet, check for errors and return the file size
unsigned long long readControlStartPacket(unsigned char *buf) {
  debug_al("Waiting for Control Start packet...");
  int bytesRead = llread(buf);
  if (bytesRead == -1)
error("Failed to read Control Start packet");
  if (buf[0] != CTRL_START)
error("Received packet is not a Control Start packet");
  if (buf[1] != CTRL_FILE_SIZE)
error("Control Start type field is not a file size type");
  unsigned char paramLength = buf[2];
  if (bytesRead < 3 + paramLength)
error("Control Start packet has less bytes than expected");
  unsigned long long fileSize = 0;
  for (int i = 0; i < paramLength; i++) {
fileSize = (fileSize << 8) | buf[3 + i];
  }
  return fileSize;
}

void checkControlEndPacket(unsigned char *buf, unsigned long long fileSize,
                           const char *filename) {
  if (buf[0] != CTRL_END)
error("Received packet is not a Control End packet");
  if (buf[1] != CTRL_FILE_SIZE)
error("Control End type field is not a file size type");
  unsigned char paramLength = buf[2];
  unsigned long long int endFileSize = 0;
  for (int i = 0; i < paramLength; i++) {
endFileSize = (endFileSize << 8) | buf[3 + i];
  }
  if (endFileSize != fileSize)
error("Received file size does not match the file size in the Control "
       "Start packet");
```

```
  if (fileSize != getFileSize(filename))
error("File size does not match the received file size");
}

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

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename) {
  LinkLayer serialConf =
       linkLayerInit(serialPort, role, baudRate, nTries, timeout);
  if (llopen(serialConf) != 1)
error("Failed to open serial port");
  unsigned long long fileSize;

  if (serialConf.role == LlTx) { // Transmitter: Read file and write
fileSize = getFileSize(filename);
sendControlPacket(fileSize, filename, true);

FILE *readFile = fopen(filename, "r");
if (readFile == NULL)
       error("Failed to open file");

// Read from file and send data packets, 10KB at a time
unsigned char fileData[MAX_DATA_SIZE] = {0};
int remainingBytes = fileSize;
for (int sequenceNumber = 0; remainingBytes > 0; sequenceNumber++) {
       int bytesRead = fread(fileData, 1, MAX_DATA_SIZE, readFile);
       debug_al("Read %d bytes from file...", bytesRead);
       sendDataPacket(fileData, bytesRead, sequenceNumber);
       remainingBytes -= bytesRead;
}

fclose(readFile);
sendControlPacket(fileSize, filename, false);
  }

  if (serialConf.role == LlRx) { // Receiver: Read and write to file
unsigned char buf[MAX_PAYLOAD_SIZE] = {0};
```

```
fileSize = 0;
FILE *writeFile = fopen(filename, "w");
if (writeFile == NULL)
       error("Failed to open file");
unsigned char controlByte = 0;

fileSize = readControlStartPacket(buf);

// Read data packets and write to file
unsigned char sequenceNumber = 0;
while (true) {
       debug_al("Waiting for Data packet nº%02d...", sequenceNumber);
       int bytesRead = llread(buf);
       if (bytesRead == -1)
       error("Failed to read Data/End packet");
       if (bytesRead < 7)
       error("Data/End packet has less bytes than expected");
       controlByte = buf[0];
       if (controlByte == CTRL_END) {
       debug_al("Received Control End packet; verifying values...");
       break;
       }
       if (controlByte != CTRL_DATA)
       error("Received packet is not a data packet nor a control end
packet");
       if (buf[1] != sequenceNumber)
       error("Received packet has wrong sequence number");
       unsigned short dataSize = (buf[2] << 8) | buf[3];
       debug_al("Writing %d bytes to file...", dataSize);
       fwrite(buf + 4, sizeof(char), dataSize, writeFile);
       sequenceNumber = (sequenceNumber + 1) % 100;
}

fclose(writeFile);
checkControlEndPacket(buf, fileSize, filename);
  }

  if (llclose(TRUE) != 1)
error("Failed to close serial port");
```

```
  if (serialConf.role == LlTx)
debug_al("File sent successfully!");
  else
debug_al("File received successfully!");
}
```

main.c

Este ficheiro é fornecido no modelo e não foi alterado.

Makefile

Este ficheiro é fornecido no modelo e não foi alterado.