

Redes de Computadores - Ligação de dados

Miguel Rodrigues (up201906042@edu.fe.up.pt)

Nuno Castro (up202003324@edu.fe.up.pt)

Conteúdo

Sumário	2
1. Introdução	2
2. Arquitetura e Estrutura de código	2
3. Protocolo de ligação de dados	2
3.1 <code>int llopen(int port, const uint8_t addr)</code>	3
3.2 <code>ssize_t llwrite(int fd, uint8_t *buffer, ssize_t len)</code>	3
3.3 <code>ssize_t llread(int fd, uint8_t *buffer)</code>	3
3.4 <code>int llclose(int fd)</code>	3
3.5 Opções	3
3.6 Detalhes de implementação	3
4. Protocolo de aplicação	6
5. Validação	6
6. Eficiência de protocolo de ligação	6
6.1 Aspetos de implemetação relativas a <i>ARQ (Automatic Repeate reQuest)</i>	7
6.2 Caraterização estatística da eficiência do protocolo	7
6.3 Performance	9
7. Conclusões	9
Anexos	10
Código fonte	10
<code>application.h</code>	10
<code>sender.c</code>	10
<code>receiver.c</code>	12
<code>protocol.h</code>	14
<code>protocol.c</code>	15
<code>utils.h</code>	27
<code>utils.c</code>	28

Sumário

Este projeto foi desenvolvido no âmbito da unidade curricular de Redes de Computadores e visa a implementação de um protocolo de ligação de dados e testando-o com uma aplicação de transferência de ficheiros.

1. Introdução

Existem inúmeras motivações para que existam mecanismos de transferência de dados entre computadores diferentes, por exemplo, para comunicar à distância. Além disso, é fundamental que essa transferência de dados decorra sem qualquer tipo de erros e de uma forma confiável - esta é, indubitavelmente, a principal motivação para a realização deste pequeno projeto.

Neste primeiro trabalho prático foi-nos proposto a implementação de um protocolo para a troca de dados entre 2 computadores ligados por uma porta série. As principais tecnologias utilizadas foram a linguagem C, a porta série RS-232 e ainda a *API* programática do *Linux*.

2. Arquitetura e Estrutura de código

A implementação do protocolo pode ser dividido em diferentes unidades lógicas cada uma independente entre si. Deste modo, temos um protocolo para a aplicação onde uma das partes, isto é o emissor comunica com o recetor usando uma interface que oferece uma abstração à camada de ligação de dados entre os dois programas.

Como foi expresso no parágrafo anterior, o código encontra-se dividido de modo a proporcionar diferentes camadas de abstração, isto significa que as diferentes unidades lógicas são independentes entre si. No nosso caso, essa independência é garantida com recurso à disposição do código em diferentes ficheiros - sobretudo de *header files*, mas também com o uso da *keyword static* nas declarações das funções que são internas a uma determinada unidade lógica, para que só aí possam ser utilizadas e, simultaneamente, estar escondidas do restante código.

No que concerne à estrutura dos ficheiros, esta é muito simples. Os ficheiros `protocol.h` e `protocol.c` representam a camada de ligação de dados, depois os ficheiros `sender.c` e `receiver.c` representam a camada da aplicação e, finalmente, os ficheiros `utils.h` e `utils.c` que contêm as definições das funções utilitárias.

Para utilizar os 2 programas basta executar um dos seguintes comandos, de acordo com o fluxo de transmissão, em cada um dos dispositivos:

- Para o recetor

```
$ recv <num. da porta> <nome do ficheiro a receber>
```

- Para o emissor

```
$ sndr <num. da porta> <nome do ficheiro a enviar>
```

3. Protocolo de ligação de dados

De acordo com o enunciado proposto, devem ser implementadas 4 funções que formam uma *API* a ser usada pelas aplicações, quer do emissor, quer do recetor. Eis os cabeçalhos dessa *API*:

```
int llopen(int port, const uint8_t addr);  
ssize_t llwrite(int fd, uint8_t *buffer, ssize_t len);
```

```
ssize_t llread(int fd, uint8_t *buffer);
int llclose(int fd);
```

3.1 int llopen(int port, const uint8_t addr)

Abre o canal de comunicações fornecendo o respetivo identificador. A aplicação deve fornecer o número associado à porta série e ainda um valor de modo a identificar de que “lado” da ligação se encontra. Os valores possíveis são RECEIVER e TRANSMITTER e estão definidos no ficheiro `protocol.h`:

```
#define RECEIVER 0x01
#define TRANSMITTER 0x03
```

3.2 ssize_t llwrite(int fd, uint8_t *buffer, ssize_t len)

Escreve os dados contidos no `buffer` no canal de comunicações. Retorna o número de *bytes* escritos no canal, ou então um valor negativo em caso de erro.

3.3 ssize_t llread(int fd, uint8_t *buffer)

Lê os dados disponíveis no canal de comunicações, escrevendo-os no `buffer` passado como argumento. Retorna o valor de *bytes* lidos, ou então um valor negativo em caso de erro.

3.4 int llclose(int fd)

Fecha o canal de comunicações.

3.5 Opções

O protocolo permite que se configurem algumas opções (em tempo de compilação) a partir do ficheiro `makefile`, são elas:

Tabela 1: Opções de compilação disponíveis no ficheiro `makefile`.

Opção	Descrição
BAUDRATE	Número de símbolo que fluem no canal de comunicações por segundo.
TOUT	Número de segundos de espera, no emissor, sem uma resposta do recetor até se desencadear uma retransmissão.
TPROP	Número de segundos de espera no recetor de modo a simular um atraso no tempo de propagação de uma trama.
MAX_RETRIES	Número máximo de tentativas de retransmissão até que o emissor desista de retransmitir.
MAX_PACKET_SIZE	Tamanho máximo, em <i>bytes</i> , para os pacotes da aplicação

3.6 Detalhes de implementação

Na implementação do protocolo da ligação de dados os principais desafios foram as implementações dos mecanismos de transparência e deteção de erros nos dados transmitidos e do mecanismo de leitura de dados, sobretudo por causa da panóplia de nuances a ter em conta.

O fluxo de execução é bastante simples, com a característica de que na nossa implementação é o emissor quem toma a iniciativa. Deste modo, o emissor começa por enviar o comando SET ficando logo de seguida à espera de uma resposta do recetor. Já do lado do recetor, o programa aguarda pela receção da trama SET e envia a resposta - uma trama do tipo UA.

O envio das tramas de supervisão é feito pela função `send_frame_us(int fd, uint8_t cmd, uint8_t addr)` onde `fd` descreve o indentificador do canal de comunicações, `cmd` o valor a ser enviado no campo de comando e `addr` que descreve quem envia a trama. Os valores possíveis para `addr` são os mesmos que os da função `llopen`. Nos mesmo moldes, para a `cmd` os valores possíveis são:

```
typedef enum { SET, DISC, UA, RR_0, REJ_0, RR_1, REJ_1 } frameCmd;
```

A construção das tramas de supervisão fica clara com o seguinte excerto de código:

```
unsigned char frame[5];
frame[0] = frame[4] = FLAG;
frame[1] = addr;
frame[2] = cmds[cmd];
frame[3] = frame[1] ^ frame[2];
```

Por outro lado, a receção das tramas de supervisão (e de informação) é digerida na função `read_frame_us(int fd, const uint8_t cmd_mask, const uint8_t addr)`. Esta função é mais complexa que a anterior, na medida em que existe uma máquina de estados para intepertar cada *byte* de informação lido - isto acontece porque há a necessidade de se ler os dados que chegam *byte* a *byte*. Aqui, os parâmetros, apesar de terem nomes semelhantes, tomam uma intepertação ligeiramente diferente. Assim, `fd` é o identificador do canal de comunicações, `cmd_mask` é uma máscara de *bits* para permitir que com a mesma função seja possível ler um valor dentro um conjunto valores que possam ocorrer - isto prova-se útil quando existem múltiplas possibilidades de resposta ao envio de uma trama de informação - por último, o valor `addr` representa o valor do lado que enviou a trama.

Depois, o envio e a codificação das tramas de informação é feito pelas funções `write_data` e `encode_data` chamadas por `llwrite`. No outro lado da comunicação, em `llread`, temos a leitura que é intepertada com recurso a máquina de estado - muito semelhante à presente em `send_frame_us` - e a decodificação que é da responsabilidade da função `decode_data`. No fim, após o envio de todos os dados, a conexão é terminada com a chamada a `llclose`.

Alguns excertos de código relevantes são os seguintes:

- As funções `encode_data` e `decode_data` que implementam o mecanismo de transparência de dados, muito importante, na medida em que permite que valores com significado especial possam ocorrer ao longo da informação trasmitida.

```
static ssize_t
encode_data(uint8_t **dest, const uint8_t *src, ssize_t len)
{
    ssize_t i, j;
    uint8_t bcc = src[0];
    for (i = 1; i < len; i++)
        bcc ^= src[i];
```

```

    ssize_t inc = 0;
    for (i = 0; i < len; i++)
        inc += ESCAPED_BYTE(src[i]);

    ssize_t nlen = len + inc + ESCAPED_BYTE(bcc) + 1;
    *dest = (uint8_t *)malloc(nlen);
    passert(dest != NULL, "protocol.c :: malloc", -1);

    for (i = 0, j = 0; j < len; i += ESCAPED_BYTE(src[j]) + 1, j++)
        encode_cpy(*dest, i, src[j]);
    encode_cpy(*dest, len + inc, bcc);

    return nlen;
}

```

```

static ssize_t
decode_data(uint8_t *dest, const uint8_t *src, ssize_t len)
{
    ssize_t i, j;
    ssize_t dec = 0;
    for (i = 0; i < len; i++)
        dec += IS_ESCAPE(src[i]);

    for (i = 0, j = 0; j < len - dec; i++, j++)
        dest[j] = IS_ESCAPE(src[i]) ? (src[++i] ^ KEY) : src[i];

    return len - dec;
}

```

- A função `recv_send_response` que averigua se o campo de proteção de dados está correto e que envia a resposta mais adequada ao emissor. Esta função é chamada por `llread`.

```

static int
recv_send_response(int fd, const uint8_t *buffer, const ssize_t len)
{
    ssize_t i;
    uint8_t bcc = buffer[0], expect_bcc = buffer[len-1];
    for (i = 1; i < len - 1; i++)
        bcc ^= buffer[i];

    uint8_t cmd;
    cmd = sequence_number ? RR_1 : RR_0;
    if (bcc != expect_bcc)
        cmd = sequence_number ? REJ_1 : REJ_0;

    send_frame_us(fd, cmd, RECEIVER);
    return (bcc == expect_bcc) ? len : -1;
}

```

4. Protocolo de aplicação

Como vimos na secção anterior, o protocolo da ligação de dados caracteriza-se por estar mais a baixo no modelo *OSI* do que o protocolo da aplicação. Este protocolo é mais simples e recorre à *API* descrita em cima para transferir dados.

No nosso caso implementamos 2 aplicações que representam o recetor e o transmissor dos dados. Em ambos os programas a primeira ação a ser efetuada é a abertura do canal de comunicações com a chamada a `llopen`. Depois, ocorre uma divergência na lógica dos 2 programas. Começamos pelo emissor, que envia um primeiro pacote de controlo com o valor **START** no campo de controlo e o tamanho do ficheiro, depois lê pequenos fragmentos do ficheiro fornecido como argumento e envia os respetivos pacotes de dados finalizando com um pacote de controlo semelhante ao primeiro exceto no campo de controlo onde o valor é **STOP**. Este envio dos dados acontece com recurso a chamadas a `llwrite`. Enquanto isso, do outro lado, o recetor vai lendo os pacotes de controlo e de informação e escrevendo-os no ficheiro fornecido como argumento do programa. Findo todo o processo de transmissão ambos os programas chamam a função `llclose`, libertam os recursos sobre a sua alçada e cessam a sua execução.

5. Validação

Para a validação do protocolo implementado foram executados vários testes e depois verificadas as *checksums* dos ficheiros para garantir que todos os componentes do protocolo, sobretudo os mecanismos de deteção de erros, de retransmissão e de transparência funcionavam corretamente. O tipo de testes realizados foram:

- Execução com ficheiros diferentes;
- Execução “normal” com e sem introdução de erros;
- Começo da execução tardio no lado do recetor;
- Execução com interrupções na porta série.

O *output* da execução dos teste realizados foi o seguinte:

```
$ recv 11 pingu.gif
$ sndr 10 pinguim.gif
$ sha256sum pinguim.gif pingu.gif
54da34fa5529f96c60aead3681e5ed2a53b98ce4281e62702ca2f39530c07365  pinguim.gif
54da34fa5529f96c60aead3681e5ed2a53b98ce4281e62702ca2f39530c07365  pingu.gif
```

As *checksums* foram, para todos os testes realizados, exatamente iguais, portanto o ficheiro enviado e o ficheiro recebido são exatamente iguais - o resultado pretendido. Ou seja, o protocolo é capaz de ultrapassar erros que possam ocorrer em qualquer um dos lados do eixo de comunicações.

6. Eficiência de protocolo de ligação

Segundo a definição, a eficiência de um protocolo é a razão de tempo gasto entre o envio ou leitura de dados e o tempo gasto entre a espera pelas confirmações.

6.1 Aspetos de implemetação relativas a *ARQ* (*Automatic Repeate reQuest*)

O protocolo implementado caracteriza-se pelo facto de ter a funcionalidade *ARQ*, neste caso em particular estamos perante um caso especial de *Go back N* onde

$$N = 1$$

Isto é, *Stop & Wait* - o emissor não deve avançar sem antes aguardar por uma resposta do recetor, seja ela uma resposta positiva ou uma rejeição devido a erros. Além disso, para *Go Back N* existe a necessidade de haver um número de sequência, como acontece na nossa implementação com a variável `sequence_number` definida no ficheiro `protocol.c`, e que permita ordenar as tramas de acordo com a ordem pretendida. Para *Stop & Wait* essa variável apenas precisa de alternar entre 0 e 1, visto que ocorre sempre a retransmissão para uma trama que ainda não tenha sido aceite.

Contudo, a facilidade de implementação de um sistema *Stop & Wait* impede que este faça frente à eficiência de outros mecanismos, como é o caso do *selective repeat* - onde o envio de dados prossegue mesmo em caso de erro (erros que são corrigidos alguns envios depois).

6.2 Caraterização estatística da eficiência do protocolo

Deste modo, os valores de eficiência para *Stop & Wait* são dados pelas seguintes fórmulas, disponíveis nos diapositivos apresentados nas aulas teóricas:

$$a = \frac{T_{prop}}{T_f}$$

Razão entre o tempo de propagação e o tempo de envio dos dados de um trama.

$$S = \frac{T_f}{T_f + 2T_{prop}} = \frac{1}{1 + 2a}$$

Eficiência do protocolo sem quaisquer erros.

$$S_e = \frac{T_f}{E[A](T_f + 2T_{prop})} = \frac{1}{E[A](1 + 2a)} = \frac{1 - FER}{1 + 2a}$$

Eficiência do protocolo com erros.

Onde:

- T_f : tempo entre envio de dados de uma trama;
- T_{prop} : tempo de propagação de uma trama ao longo do canal de comunicações;
- FER : probabilidade de erro de uma trama (*Frame Error Ratio*);
- $E[A]$: número médio de tentativas para se transmitir uma trama com sucesso.

Como se observa, surgem várias conclusões. A primeira é a de que se o valor de a for elevado, então, a eficiência será baixa. O principal motivo para que isto ocorra pode ser a distância entre os pontos de comunicação, bem como, o facto do tamanho da trama de informação não ser suficientemente grande - o que conduz a um tempo de envio menor, e consequentemente a um valor de a maior. Já a segunda conclusão a que chegamos é a de que se a probabilidade de uma trama conter erros - FER - for elevada, naturalmente, a eficiência do protocolo irá cair. A modelação dos valores da eficiência de acordo com a probabilidade de erro de uma trama pode ser observada no gráfico seguinte:

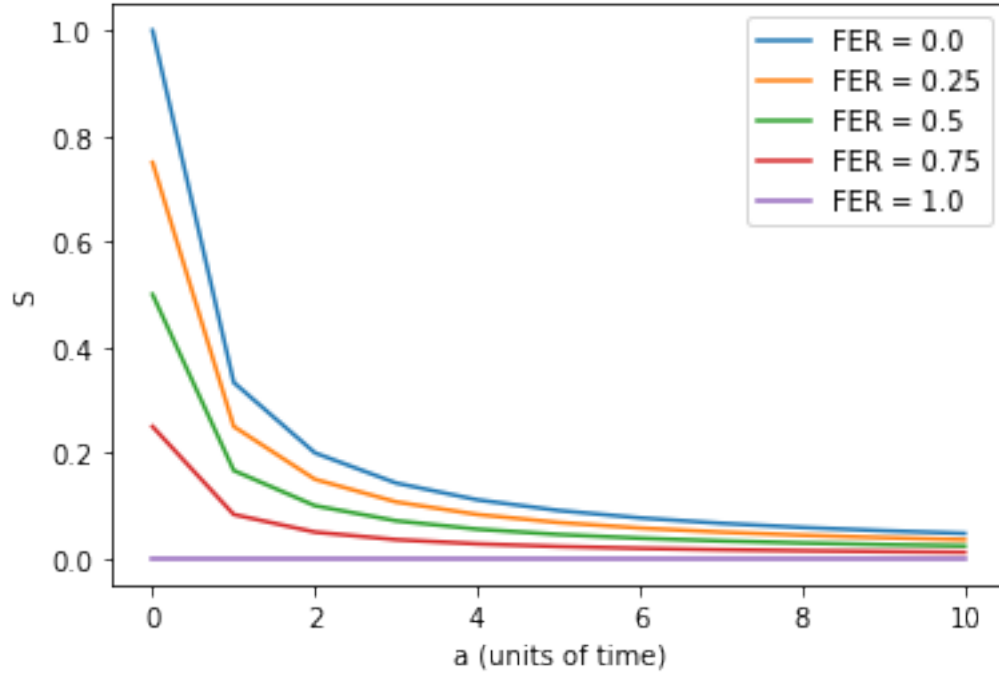


Figura 1: Valor da eficiência de acordo com o FER

Neste gráfico, importa referir que o cenário representado pela linha púrpura é hipotético, na medida, em que todas as tramas possuem erros o que impossibilita a transferência da informação, resultado, obviamente, numa eficiência nula e constante. Por outro lado, percebe-se, pela análise do gráfico, que o valor de a tem a sua influência independentemente do valor de FER . Não obstante, nota-se também que para valores baixos de a , a eficiência depende praticamente do FER .

Tendo tudo isto em conta, a escolha de *Stop & Wait* para mecanismo de *ARQ* deve ser pensada, sobretudo, de acordo com a distância entre o emissor e o recetor, mesmo que seja mais fácil de ser implementado ou que o canal tenha uma capacidade elevada e com pouca probabilidade de erros.

6.3 Performance

O gráfico seguinte mostra os tempos de envio do ficheiro fornecido `pinguim.gif` de acordo com o tamanho máximo para um pacote de dados da aplicação. Nota que este valor pode ser alterado nas opções do ficheiro `makefile`.

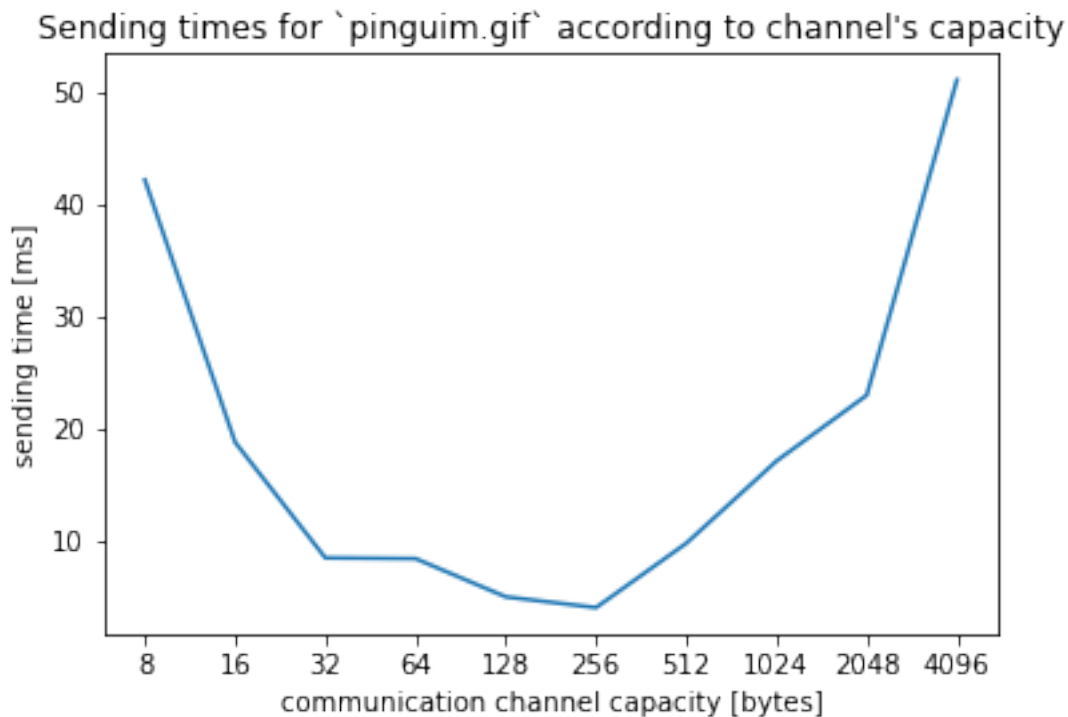


Figura 2: Tempos de envio de acordo com o tamanho dos pacotes

Como se observa, existe um valor mínimo para os tempos de envio que ronda os 256 *bytes*. Isto significa, que a eficiência do protocolo começa a cair com pacotes maiores. Podemos então assim concluir que se para pacotes mais pequenos o número de fragmentos a enviar causa um acréscimo ao tempo de envio, por outro lado, para pacotes maior o maior esforço de processamento abafa a suposta rapidez obtida de um menor número de envios de fragmentos.

7. Conclusões

Este foi um trabalho que certamente gerou um certo interesse da maioria dos alunos, sobretudo pelo facto de poderem observar fisicamente a transferência de ficheiros entre os 2 computadores no laboratório. Todavia, mesmo sendo um trabalho exigente é ótimo que assim o seja, pois obriga os estudantes a estarem a par dos conceitos teóricos leccionados nas aulas.

Agora, em retrospectiva, verificamos que com este pequeno projeto foi possível cimentar os conhecimentos prévios em C mas também descobrir, como efetivamente, a informação era transmitida por uma porta série, bem antes da internet dar os seus primeiros passos e revolucionar essa transferência da informação.

Anexos

Código fonte

application.h

```
1  /*
2   * application.h
3   * Serial port application protocol
4   * RC @ L.EIC 2122
5   * Authors: Miguel Rodrigues & Nuno Castro
6   */
7
8  #ifndef _APPLICATION_H_
9  #define _APPLICATION_H_
10
11 /* Control command for application packets */
12 typedef enum { DUMMY, DATA, START, STOP } ctrlCmd;
13 /* Parameter command for application packets */
14 typedef enum { SIZE, NAME } paramCmd;
15
16 #endif /* _APPLICATION_H_ */
```

sender.c

```
1  /*
2   * sender.c
3   * Serial port protocol sender application
4   * RC @ L.EIC 2122
5   * authors: Miguel rodrigues & Nuno castro
6   */
7
8  #include <sys/stat.h>
9
10 #include <stdint.h>
11 #include <stdio.h>
12 #include <unistd.h>
13
14 #include "application.h"
15 #include "protocol.h"
16 #include "utils.h"
17
18
19 int
20 main (int argc, char **argv)
21 {
22     if (argc < 3) {
23         fprintf(stderr, "usage: %s <port> <filename>\n", argv[0]);
24         return 1;
25     }
```

```

25     }
26
27 #ifdef DEBUG
28     clock_t begin;
29     begin = bclk();
30 #endif
31
32     int fd_file;
33     fd_file = open(argv[2], O_RDONLY);
34     passert(fd_file >= 0, "sender.c :: open", -1);
35
36     int fd;
37     fd = llopen(atoi(argv[1]), TRANSMITTER);
38     passert(fd >= 0, "sender.c :: llopen", -1);
39
40     uint8_t frag[MAX_PACKET_SIZE];
41
42     struct stat st;
43     fstat(fd_file, &st);
44     const off_t size_file = st.st_size;
45
46     frag[0] = START;
47     frag[1] = SIZE;
48     frag[2] = sizeof(off_t);
49     memcpy(frag + 3, &size_file, sizeof(off_t));
50
51     int wb;
52     wb = llwrite(fd, frag, 3 + sizeof(off_t));
53     passert(wb >= 0, "sender.c :: llwrite", -1);
54
55     uint16_t n;
56     n = size_file / (MAX_PACKET_SIZE - 4);
57     n += (size_file % (MAX_PACKET_SIZE - 4));
58
59     ssize_t rb;
60     int i;
61     for (i = 0; i < n; i++) {
62         rb = read(fd_file, frag + 4, MAX_PACKET_SIZE - 4);
63
64         frag[0] = DATA;
65         frag[1] = i % 255;
66         frag[2] = rb / 256;
67         frag[3] = rb % 256;
68
69         wb = llwrite(fd, frag, rb + 4);
70         passert(wb >= 0, "sender.c :: llwrite", -1);
71     }
72

```

```

73     frag[0] = STOP;
74     frag[1] = SIZE;
75     frag[2] = sizeof(off_t);
76     memcpy(frag + 3, &size_file, sizeof(off_t));
77
78     wb = llwrite(fd, frag, 3 + sizeof(off_t));
79     passert(wb >= 0, "sender.c :: llwrite", -1);
80
81     llclose(fd);
82     close(fd_file);
83
84     #ifdef DEBUG
85         eclk(&begin);
86     #endif
87
88     return 0;
89 }

```

receiver.c

```

1  /*
2   * receiver.c
3   * Serial port protocol receiver application
4   * RC @ L.EIC 2122
5   * Authors: Miguel Rodrigues & Nuno Castro
6   */
7
8  #include <stdint.h>
9  #include <stdio.h>
10 #include <time.h>
11 #include <unistd.h>
12
13 #include "application.h"
14 #include "protocol.h"
15 #include "utils.h"
16
17
18 int
19 main(int argc, char **argv)
20 {
21     if (argc < 3) {
22         fprintf(stderr, "usage: %s <port> <filename>\n", argv[0]);
23         return 1;
24     }
25
26     #ifdef DEBUG
27         clock_t begin;
28         begin = bclk();

```

```

29         srand(begin); /* required in order to make random errors */
30     #endif
31
32     int fd_file;
33     fd_file = open(argv[2], O_CREAT | O_WRONLY, 0666);
34     passert(fd_file >= 0, "receiver.c :: open", -1);
35
36     int fd;
37     fd = llopen(atoi(argv[1]), RECEIVER);
38     passert(fd >= 0, "receiver.c :: llopen", -1);
39
40     uint8_t pkg_n = 0;
41     uint8_t frag[MAX_PACKET_SIZE];
42     ssize_t rb, len;
43
44     while (1) {
45         rb = llread(fd, frag);
46         if (rb < 0)
47             continue;
48
49         switch (frag[0]) {
50             case DATA:
51                 if (frag[1] > (pkg_n % 256)) {
52                     len = frag[2] * 256 + frag[3];
53                     write(fd_file, frag + 4, len);
54                     pkg_n++;
55                 }
56                 break;
57             case START:
58                 break;
59             case STOP:
60                 llread(fd, frag); /* Take the last disc frame */
61                 goto finish;
62             default:
63                 break;
64         }
65     }
66
67     finish:
68         llclose(fd);
69         close(fd_file);
70
71     #ifdef DEBUG
72         eclk(&begin);
73     #endif
74
75     return 0;
76 }

```

protocol.h

```
1  /*
2   * protocol.h
3   * Serial port protocol
4   * RC @ L.EIC 2122
5   * Authors: Miguel Rodrigues & Nuno Castro
6   */
7
8  #ifndef _PROTOCOL_H_
9  #define _PROTOCOL_H_
10
11  #include <errno.h>
12  #include <fcntl.h>
13  #include <signal.h>
14  #include <stdint.h>
15  #include <stdio.h>
16  #include <stdlib.h>
17  #include <string.h>
18  #include <termios.h>
19  #include <unistd.h>
20
21  #include "utils.h"
22
23  #define RECEIVER 0x01
24  #define TRANSMITTER 0x03
25
26  /**
27   * Sets up the terminal, in order to send information packets
28   * @param int[in] - port x corresponding to the file /dev/ttySx
29   * @param const uint8_t[in] - determines whether is the RECEIVER or TRANSMITTER called
30   * @param int[out] - file descriptor corresponding to the opened file
31   */
32  int
33  llopen(int port, const uint8_t addr);
34
35  /**
36   * Writes a given chunk of information in the file pointed by the first param
37   * @param int[in] - file descriptor pointing to the file where information will be written
38   * @param uint8_t *[in] - information to be written
39   * @param ssize_t[in] - size in bytes of the chunk of information
40   * @param ssize_t[out] - number of bytes written
41   */
42  ssize_t
43  llwrite(int fd, uint8_t *buffer, ssize_t len);
44
45  /**
46   * Reads a given chunk of information in the file pointed by the first param
```

```

47  * @param int[in] - file descriptor pointing to the file where information will be read
48  * @param uint8_t *[in] - place where to place the information after performing the reading
49  * @param ssize_t[out] - number of bytes read
50  */
51  ssize_t
52  llread(int fd, uint8_t *buffer);
53
54  /**
55   * Reverts to the previous terminal settings and shutdowns all the resources in use
56   * @param int[in] - file descriptor corresponding to the opened file
57   * @param int[out] - 0 if no errors occur, negative value otherwise
58   */
59  int
60  llclose(int fd);
61
62  #endif /* _PROTOCOL_H_ */

```

protocol.c

```

1  /*
2   * protocol.c
3   * Serial port protocol
4   * RC @ L.EIC 2122
5   * Authors: Miguel Rodrigues & Nuno Castro
6   */
7
8  #include "protocol.h"
9
10 /* macros */
11 #define FLAG 0x7E
12 #define ESCAPE 0x7D
13 #define KEY 0x20
14
15 #define RESEND 1
16
17 #define IS_ESCAPE(c) (c == ESCAPE)
18 #define IS_FLAG(c) (c == FLAG)
19 #define ESCAPED_BYTE(c) (IS_ESCAPE(c) || IS_FLAG(c))
20
21 #define BITSET(m, i) (m & (1 << i))
22
23 /* commands */
24 typedef enum { SET, DISC, UA, RR_0, REJ_0, RR_1, REJ_1 } frameCmd;
25 static const uint8_t cmds[7] = { 0x3, 0xb, 0x7, 0x5, 0x1, 0x85, 0x81 };
26
27 #ifdef DEBUG
28 static const char
29 cmds_str[7][6] = { "SET", "DISC", "UA", "RR_0", "REJ_0", "RR_1", "REJ_1" };

```

```

30 #endif
31
32 /* reading */
33 typedef enum { START, FLAG_RCV, A_RCV, C_RCV, BCC_OK, DATA, STOP } readState;
34
35 /* global variables */
36 static struct termios oldtio, newtio;
37 static struct sigaction sigact;
38
39 static int port_fd;
40
41 static uint8_t connector;
42 static volatile uint8_t retries, sequence_number = 0;
43 static int connection_alive;
44
45 static uint8_t buffer_frame[2*MAX_PACKET_SIZE+5];
46 static ssize_t buffer_frame_len;
47
48 /* forward declarations */
49 static int check_resending(uint8_t cmd);
50
51 /* util funcs */
52 static void
53 install_sigalrm(void (*handler)(int))
54 {
55     sigact.sa_handler = handler;
56     sigemptyset(&sigact.sa_mask);
57     sigact.sa_flags = 0;
58     sigaction(SIGALRM, &sigact, NULL);
59 }
60
61
62
63 static int
64 term_conf_init(int port)
65 {
66     char fname[12];
67     snprintf(fname, 12, "/dev/ttyS%d", port);
68
69     port_fd = open(fname, O_RDWR | O_NOCTTY);
70     if (port_fd < 0)
71         return -1;
72
73     if (tcgetattr(port_fd, &oldtio) < 0)
74         return -1;
75
76     memset(&newtio, '\0', sizeof(newtio));
77

```



```

78     newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
79     newtio.c_iflag = IGNPAR;
80     newtio.c_oflag = 0;
81     newtio.c_lflag = 0; /* set input mode (non-canonical, no echo...) */
82
83     newtio.c_cc[VTIME] = 0;
84     newtio.c_cc[VMIN] = 1; /* 1 char required to satisfy a read */
85
86     tcflush(port_fd, TCIOFLUSH);
87     if (tcsetattr(port_fd, TCSANOW, &newtio) == -1)
88         return -1;
89 #ifdef DEBUG
90     plog("termios struct set with success\n");
91 #endif
92     return port_fd;
93 }
94
95 static int
96 term_conf_end(int fd)
97 {
98     if (tcsetattr(fd, TCSANOW, &oldtio) < 0)
99         return -1;
100
101     close(fd);
102     return 0;
103 }
104
105
106
107 static int
108 send_frame_us(int fd, const uint8_t cmd, const uint8_t addr)
109 {
110     unsigned char frame[5];
111
112     frame[0] = frame[4] = FLAG;
113     frame[1] = addr;
114     frame[2] = cmds[cmd];
115     frame[3] = frame[1] ^ frame[2];
116
117     if (write(fd, frame, sizeof(frame)) < 0)
118         return -1;
119 #ifdef DEBUG
120     if (addr == TRANSMITTER)
121         plog("frame sent with %s @ TRANSMITTER\n", cmds_str[cmd]);
122     else if (addr == RECEIVER)
123         plog("frame sent with %s @ RECEIVER\n", cmds_str[cmd]);
124 #endif
125     return 0;

```

```

126 }
127
128 static int
129 read_frame_us(int fd, const uint8_t cmd_mask, const uint8_t addr)
130 {
131     readState st = START;
132     uint8_t frame[5];
133     ssize_t i, cmd, rb;
134
135     while (st != STOP && retries < MAX_RETRIES) {
136         rb = read(fd, frame + st, 1);
137         if (rb < 0 && errno == EINTR)
138             continue;
139
140         switch (st) {
141             case START:
142                 st = IS_FLAG(frame[st]) ? FLAG_RCV : START;
143                 break;
144             case FLAG_RCV:
145                 if (frame[st] == addr)
146                     st = A_RCV;
147                 else if (frame[st] != FLAG)
148                     st = START;
149                 break;
150             case A_RCV:
151                 for (i = 0; i < 7; i++) {
152                     if (BITSET(cmd_mask, i) && frame[st] == cmds[i]) {
153                         st = C_RCV;
154                         cmd = i;
155                     }
156                 }
157
158                 if (st != C_RCV) {
159                     st = IS_FLAG(frame[st]) ? FLAG_RCV : START;
160                     frame[0] = FLAG;
161                 }
162                 break;
163             case C_RCV:
164                 if (frame[st] == (frame[st-1] ^ frame[st-2])) {
165                     st = BCC_OK;
166                 } else if (frame[st] == FLAG) {
167                     st = FLAG_RCV;
168                     frame[0] = FLAG;
169                 } else {
170                     st = START;
171                 }
172                 break;
173             case BCC_OK:

```

```

174             st = IS_FLAG(frame[st]) ? STOP : START;
175             break;
176         default:
177             break;
178     }
179 }
180
181 connection_alive = retries < MAX_RETRIES;
182 if (!connection_alive)
183     return -1;
184 #ifdef DEBUG
185     if (addr == RECEIVER)
186         plog("frame read with %s @ TRANSMITTER\n", cmds_str[cmd]);
187     else if (addr == TRANSMITTER)
188         plog("frame read with %s @ RECEIVER\n", cmds_str[cmd]);
189 #endif
190     uint8_t frame_i_ans = 1 << RR_0 | 1 << REJ_0 | 1 << RR_1 | 1 << REJ_1;
191     if (connector == TRANSMITTER && cmd_mask == frame_i_ans)
192         return check_resending(frame[2]);
193
194     return 0;
195 }
196
197
198
199 void
200 trmt_alrm_handler_open(int unused)
201 {
202     alarm(TOUT);
203     retries++;
204     send_frame_us(port_fd, SET, TRANSMITTER);
205 }
206
207 static int
208 llopen_recv(int fd)
209 {
210     read_frame_us(fd, 1 << SET, TRANSMITTER);
211     send_frame_us(fd, UA, RECEIVER);
212
213     return 0;
214 }
215
216 static int
217 llopen_trmt(int fd)
218 {
219     int conn_est;
220
221     retries = 0;

```

```

222     install_sigalrm(trmt_alm_handler_open);
223
224     send_frame_us(fd, SET, TRANSMITTER);
225     alarm(TOUT);
226     conn_est = read_frame_us(fd, 1 << UA, RECEIVER);
227     alarm(0);
228
229     if (!connection_alive) {
230         perr("can't establish a connection with the RECEIVER\n");
231         return -1;
232     }
233
234     return conn_est;
235 }
236
237 int
238 llopen(int port, const uint8_t addr)
239 {
240     int fd;
241     fd = term_conf_init(port);
242     if (fd < 0)
243         return -1;
244
245     int cnct;
246     cnct = (addr == TRANSMITTER) ? llopen_trmt(fd) : llopen_recv(fd);
247     if (cnct < 0)
248         return cnct;
249
250     connector = addr;
251     return fd;
252 }
253
254
255
256 static void
257 encode_cpy(uint8_t *dest, ssize_t off, uint8_t c)
258 {
259     dest[off] = c;
260
261     if (ESCAPED_BYTE(c)) {
262         dest[off] = ESCAPE;
263         dest[off+1] = c ^ KEY;
264     }
265 }
266
267 static ssize_t
268 encode_data(uint8_t **dest, const uint8_t *src, ssize_t len)
269 {

```

```

270     ssize_t i, j;
271     uint8_t bcc = src[0];
272     for (i = 1; i < len; i++)
273         bcc ^= src[i];
274
275     ssize_t inc = 0;
276     for (i = 0; i < len; i++)
277         inc += ESCAPED_BYTE(src[i]);
278
279     ssize_t nlen = len + inc + ESCAPED_BYTE(bcc) + 1;
280     *dest = (uint8_t *)malloc(nlen);
281     passert(dest != NULL, "protocol.c :: malloc", -1);
282
283     for (i = 0, j = 0; j < len; i += ESCAPED_BYTE(src[j]) + 1, j++)
284         encode_cpy(*dest, i, src[j]);
285     encode_cpy(*dest, len + inc, bcc);
286
287     return nlen;
288 }
289
290 static ssize_t
291 decode_data(uint8_t *dest, const uint8_t *src, ssize_t len)
292 {
293     ssize_t i, j;
294     ssize_t dec = 0;
295     for (i = 0; i < len; i++)
296         dec += IS_ESCAPE(src[i]);
297
298     for (i = 0, j = 0; j < len - dec; i++, j++)
299         dest[j] = IS_ESCAPE(src[i]) ? (src[++i] ^ KEY) : src[i];
300
301     return len - dec;
302 }
303
304
305
306 static ssize_t
307 trmt_send_data(void)
308 {
309     ssize_t wb;
310     wb = write(port_fd, buffer_frame, buffer_frame_len);
311     #ifdef DEBUG
312     uint8_t sn = sequence_number;
313     plog("sent frame no. %d of %ld bytes\n", sn, wb);
314     plog("waiting on response from RECEIVER for frame no. %d\n", sn);
315     #endif
316     return wb;
317 }

```

```

318
319 void
320 trmt_alrm_handler_write(int unused)
321 {
322     alarm(TOUT);
323     ++retries;
324     trmt_send_data();
325 }
326
327 static int
328 check_resending(const uint8_t cmd)
329 {
330     sequence_number = (cmd == cmds[RR_1] || cmd == cmds[REJ_0]);
331     return (cmd == cmds[REJ_0] || cmd == cmds[REJ_1]);
332 }
333
334 ssize_t
335 llwrite(int fd, uint8_t *buffer, ssize_t len)
336 {
337     uint8_t *data = NULL;
338     len = encode_data(&data, buffer, len);
339     if (len < 0)
340         return len;
341
342     uint8_t frame[len+5];
343
344     frame[0] = frame[len+4] = FLAG;
345     frame[1] = TRANSMITTER;
346     frame[2] = sequence_number << 6;
347     frame[3] = frame[1] ^ frame[2];
348     memcpy(frame + 4, data, len);
349
350     free(data);
351
352     buffer_frame_len = sizeof(frame);
353     memcpy(buffer_frame, frame, buffer_frame_len);
354
355     retries = 0;
356     install_sigalrm(trmt_alrm_handler_write);
357
358     ssize_t wb;
359     int rsnd;
360     uint8_t mask = 1 << RR_0 | 1 << REJ_0 | 1 << RR_1 | 1 << REJ_1;
361
362     do {
363         wb = trmt_send_data();
364         if (wb < 0)
365             return wb;

```

```

366         alarm(TOUT);
367         rsnd = read_frame_us(fd, mask, RECEIVER);
368         alarm(0);
369     } while (connection_alive && rsnd == RESEND);
370
371     if (!connection_alive) {
372         perr("can't establish a connection with RECEIVER\n");
373         return -1;
374     }
375
376     return wb;
377 }
378
379
380
381
382 static int
383 recv_send_response(int fd, const uint8_t *buffer, const ssize_t len)
384 {
385     ssize_t i;
386     uint8_t bcc = buffer[0], expect_bcc = buffer[len-1];
387     for (i = 1; i < len - 1; i++)
388         bcc ^= buffer[i];
389
390     #ifdef DEBUG
391     bcc ^= (rand() % 100 < FER) ? 0xff : 0x0; /* artificial error on bcc */
392     sleep(TPROP); /* artificial propagation time */
393     #endif
394
395     uint8_t cmd;
396     cmd = sequence_number ? RR_1 : RR_0;
397     if (bcc != expect_bcc)
398         cmd = sequence_number ? REJ_1 : REJ_0;
399
400     send_frame_us(fd, cmd, RECEIVER);
401     return (bcc == expect_bcc) ? len : -1;
402 }
403
404
405 ssize_t
406 llread(int fd, uint8_t *buffer)
407 {
408     readState st = START;
409     uint8_t frame[2*MAX_PACKET_SIZE+5];
410     uint8_t disc = 0;
411     ssize_t c = 0;
412
413     while (st != STOP) {
414         if (read(fd, frame + st + c, 1) < 0)
415             return -1;

```

```

414     switch (st) {
415     case START:
416         st = IS_FLAG(frame[st]) ? FLAG_RCV : START;
417         break;
418     case FLAG_RCV:
419         if (frame[st] == TRANSMITTER)
420             st = A_RCV;
421         else if (frame[st] != FLAG)
422             st = START;
423         break;
424     case A_RCV:
425         if (frame[st] == 0x0 || frame[st] == 0x40) {
426             sequence_number = !frame[st];
427             st = C_RCV;
428         } else if (frame[st] == cmds[DISC]) {
429             st = C_RCV;
430             disc = 1;
431         } else if (IS_FLAG(frame[st])) {
432             st = FLAG_RCV;
433             frame[0] = FLAG;
434         } else {
435             st = START;
436         }
437         break;
438     case C_RCV:
439         if (frame[st] == (frame[st-1] ^ frame[st-2])) {
440             st = BCC_OK;
441         } else if (IS_FLAG(frame[st])) {
442             st = FLAG_RCV;
443             frame[0] = FLAG;
444         } else {
445             st = START;
446         }
447         break;
448     case BCC_OK:
449         st = IS_FLAG(frame[st]) ? STOP : DATA;
450         break;
451     case DATA:
452         st = IS_FLAG(frame[st+c]) ? STOP : DATA;
453         c++;
454         break;
455     default:
456         break;
457     }
458 }
459
460 #ifdef DEBUG
461     uint8_t sn = sequence_number;

```



```

462         plog("frame no. %d read with %ld bytes\n", sn, c + 5);
463     #endif
464
465     if (disc) {
466     #ifdef DEBUG
467         plog("disconnect frame detected\n");
468     #endif
469         send_frame_us(fd, DISC, RECEIVER);
470         return -1;
471     }
472
473     ssize_t len;
474     len = decode_data(buffer, frame + 4, c);
475     len = recv_send_response(fd, buffer, len);
476
477     return len;
478 }
479
480
481
482 void
483 trmt_alm_handler_close(int unused)
484 {
485     alarm(TOUT);
486     retries++;
487     send_frame_us(port_fd, DISC, TRANSMITTER);
488 }
489
490 int
491 llclose(int fd)
492 {
493     if (connector == TRANSMITTER) {
494         retries = 0;
495         install_sigalarm(trmt_alm_handler_close);
496
497         send_frame_us(fd, DISC, TRANSMITTER);
498
499         alarm(TOUT);
500         read_frame_us(fd, 1 << DISC, RECEIVER);
501         alarm(0);
502
503         if (!connection_alive) {
504             perr("can't establish a connection with RECEIVER\n");
505             return -1;
506         }
507
508         send_frame_us(fd, UA, TRANSMITTER);
509     }

```

```
510
511     sleep(2); /* gives time to all the info flow through the communications channel */
512     return term_conf_end(fd);
513 }
```

```

utils.h

1  /*
2   * utils.h
3   * Serial port protocol utility functions
4   * RC @ L.EIC 2122
5   * Authors: Miguel Rodrigues & Nuno Castro
6   */
7
8  #ifndef _UTILS_H_
9  #define _UTILS_H_
10
11 #include <errno.h>
12 #include <stdarg.h>
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <string.h>
16 #include <time.h>
17
18 /**
19  * Writes a message to stdout
20  * @param const char *[in] - message to be written
21  */
22 void plog(const char *format, ...);
23
24 /**
25  * Writes a message to stderr
26  * @param const char *[in] - message to be written
27  */
28 void perr(const char *format, ...);
29
30 /**
31  * Verifies whether a condition is valid or not,
32  * if it valid then nothing is done, otherwise it writes the message
33  * passed as argument and finishes the program imediatly
34  * @param const int[in] - condition result
35  * @param const char *[in] - message to be written
36  * @param const int[in] - program's exit code
37  */
38 void passert(const int cond, const char *msg, const int code);
39
40 /**
41  * Begins a clock
42  * @param const clock_t[out] - clock's current timestamp
43  */
44 const clock_t bclk(void);
45
46 /**

```

```

47  * Finishes a clock
48  * @param const clock_t *[in] - clock's begin timestamp
49  */
50 void eclk(const clock_t *start);
51
52 #endif /* _UTILS_H_ */

```

utils.c

```

1  /*
2  * utils.c
3  * Serial port protocol utility functions
4  * RC @ L.EIC 2122
5  * Authors: Miguel Rodrigues & Nuno Castro
6  */
7
8  #include "utils.h"
9
10
11 void
12 plog(const char *fmt, ...)
13 {
14     fprintf(stdout, "log: ");
15     va_list args;
16
17     va_start(args, fmt);
18     vfprintf(stdout, fmt, args);
19     va_end(args);
20 }
21
22 void
23 perr(const char *fmt, ...)
24 {
25     fprintf(stderr, "err: ");
26     va_list args;
27
28     va_start(args, fmt);
29     vfprintf(stderr, fmt, args);
30     va_end(args);
31 }
32
33 void
34 passert(const int cond, const char *msg, const int code)
35 {
36     if (!cond) {
37         fprintf(stderr, "die: %s :: %s\n", msg, strerror(errno));
38         exit(code);
39     }

```

```

40  }
41
42
43  const clock_t
44  bclk(void)
45  {
46      plog("clock: began\n");
47      const clock_t start = clock();
48      return start;
49  }
50
51  void
52  eclk(const clock_t *start)
53  {
54      clock_t end = clock();
55      double elapsed = (double)(end - *start) * 1000.0 / CLOCKS_PER_SEC;
56      plog("clock: ended\n");
57      plog("clock: took %.5f ms\n", elapsed);
58  }

```

-
- Miguel Boaventura Rodrigues
 - Nuno Miguel Paiva de Melo e Castro