Universidade do Porto

Faculdade de Engenharia

FEUP

Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

Protocolo de Ligação de Dados

Trabalho 1

Segunda-feira, 10 de Novembro de 2014

Página de Rosto

Título do Projeto:
Protocolo de Ligação de Dados

Curso:
Mestrado Integrado em Engenharia Informática e Computação

Unidade Curricular / Ano Letivo:
Redes de Computadores - 2014/2015

Professor:
Manuel Alberto Pereira Ricardo
Página pessoal: http://paginas.fe.up.pt/~mricardo/
E-mail: mricardo@fe.up.pt

Elementos de Grupo:
Henrique Manuel Martins Ferrolho
201202772
ei12079@fe.up.pt

João Filipe Figueiredo Pereira
201104203
ei12023@fe.up.pt

José Pedro Vieira de Carvalho Pinto
201203811
ei12164@fe.up.pt

Miguel Ângelo Jesus Vidal Ribeiro
201100657
ei11144@fe.up.pt

Data de Entrega:
Segunda-feira, 10 de Novembro de 2014

Nota Final:
_____.

Índice

Página de Rosto 2
Índice 3
Sumário 5
Introdução 6
Arquitetura 7
  Camada de aplicação e de ligação de dados 7
  Interface da linha de comandos 7
Estrutura do código 8
  Application layer 8
  Data link layer 8
Casos de uso principais 11
Protocolo de ligação lógica 12
  API da porta de série 12
    llopen 12
    llwrite 12
    llread 12
    llclose 13
Protocolo de aplicação 14
  Pacotes do nível de aplicação 14
  Envio e receção de pacotes de controlo 14
  Envio e receção de pacotes de dados 14
  Envio e receção do ficheiro 15
Validação 16
Elementos de valorização 19
  Selecção de parâmetros pelo utilizador 19
  Implementação de REJ 19
  Verificação da integridade dos dados pela aplicação 19
  Registo de ocorrências 20
Conclusões 21
Anexo I 22
  Alarm.c 22
  Alarm.h 22
  Application.c 23
  Application.h 30
  CLI.c 31
  CLI.h 33
  ConnectionMode.h 34

ControlPackageType.h 34
DataLink.c 34
DataLink.h 47
Main.c 49
ParameterType.h 50
Utilities.c 51
Utilities.h 51
Includes 52
Sequência de chamadas de funções 55

Sumário

No âmbito da unidade curricular de *Redes de Computadores*, do curso *Mestrado Integrado em Engenharia Informática e Computação*, foi-nos proposto a elaboração de uma aplicação cujos conteúdos, até à data, estavam a ser abordados nas aulas teóricas. Entre os conteúdos abordados destacam-se a Camada de Aplicação (*Application Layer*), a Camada de Ligação de Dados (*Data Link Layer*) e a Camada Física (*Physical Layer*). A proposta de trabalho baseou-se em realizar uma transferência de ficheiros entre dois computadores. Utilizou-se para o caso, conceitos interiorizados nas aulas teóricas e laboratoriais acerca de ligação de dados, através de uma ligação por cabo pelas portas de série de cada máquina.

A elaboração de um relatório final para o trabalho tem como objetivo a consolidação do trabalho realizado ao longo desta primeira metade de semestre. Neste contexto pode-se afirmar que é necessário que haja uma ligação entre a parte prática e teórica que envolveu o projeto. Esta ligação permitirá ao docente uma avaliação correta e concisa, podendo-se dar destaque ao bom entendimento por parte do grupo das matérias lecionadas e da sua implementação.

Introdução

O trabalho realizado ao longo das aulas laboratoriais da unidade curricular de Redes de Computadores tem como objetivo a implementação um protocolo de ligação de dados, de acordo com a especificação descrita no guião.

Na sua especificação era pedida a combinação de características de protocolos de ligação de dados existentes, entre os quais a transparência na transmissão de dados e uma transmissão organizada em diferentes tipos de tramas – tais como tramas de informação, supervisão e não numeradas, que eram protegidas por código detetor de erros. De referir que o tipo de transmissão utilizado em todo o projeto é em série assíncrona. Este modelo de transmissão foi aplicado nas primeiras aulas, onde nos foi pedido a sua implementação que serviria de esboço para o desenvolvimento de uma aplicação com outro nível de complexidade.

Outro objetivo do trabalho consistia em testar o protocolo com uma aplicação simples de transferência de ficheiros, igualmente especificada. Na especificação da aplicação de teste era pedido o suporte de dois tipos de pacotes enviados pelo emissor: o de dados e o de controlo.
Nos pacotes de controlo destacam-se o de *start*, sinalizando o início da transmissão; e o de *end*, sinalizando o fim da transmissão.
Já os pacotes de dados iriam conter frações do ficheiro a transmitir.

O relatório final deverá servir, assim como já foi dito em cima - na secção de Sumário - para uma consolidação de matérias entre a parte teórica e prática do projeto. No relatório devem ser especificados todos os pontos descritos na implementação do trabalho e deve ser dividido em várias secções para uma boa organização e descrição das mesmas.
Assim sendo, o relatório subdivide-se nas seguintes partes:
- **Introdução**, onde são indicados os objetivos do trabalho e relatório;
- **Arquitetura**, contendo esta secção a explicação dos blocos funcionais e interfaces;
- **Estrutura do Código**, fazendo-se referência às API's consultadas e implementadas, a estruturas de dados elaboradas, às funções de maior importância e à sua relação com a Arquitetura;
- **Casos de uso Principais**, fazer a sua identificação e abordar as sequências de chamada de funções;
- **Protocolo de ligação lógica**, onde se descreve a estratégia de implementação abordada, com referência a excertos de código e identificação dos principais aspetos funcionais;
- **Protocolo de Aplicação**, onde a exemplo do protocolo de ligação lógica também poderá ser possível encontrar excertos de código, a sua explicação e a estratégia abordada na elaboração deste ponto do trabalho;
- Na **Validação** serão descritos os testes elaborados, ilustrando os seus resultados e possíveis comentários aos mesmos;
- Em **Elementos de valorização**, estarão descritos os pontos adicionais que eram possíveis de ser realizados no trabalho e que o grupo teve a competência de implementar;
- Em **Conclusões** estará presente o comentário e análise final do grupo perante este projeto.

Antes de prosseguirmos é de referir que o grupo desenvolveu o projeto pedido em ambiente LINUX, com linguagem de programação C e utilizando portas de série do modelo RS-232, cuja comunicação é assíncrona.

Arquitetura

Camada de aplicação e de ligação de dados

O trabalho está organizado em duas camadas – *layers* – que permitem a correta funcionalidade do projeto: a camada do *protocolo de ligação de dados* e a camada de *aplicação*, que estão implementados em diferentes ficheiros *source* e *header* (*.c e *.h). Os ficheiros *DataLink.c* e *Datalink.h* representam a camada de ligação de dados enquanto os ficheiros *Application.c* e *Application.h* representam a camada de aplicação.

A camada de ligação de dados disponibiliza funções genéricas do protocolo, estando nela especificadas as funcionalidades de sincronismo e numeração de tramas, suporte de ligação, controlo da transferência, de erros e de fluxo, e todos os aspetos relacionados com a ligação à porta série – tais como a sua abertura e definição das suas propriedades.

A camada de aplicação não é independente do bloco de ligação de dados, já que tira proveito das suas propriedades para a implementação das suas funções. Podemos dizer que a camada de aplicação é a responsável pela transferência dos ficheiros pois é nesta que são executadas as funções de receção e emissão de tramas. Os pacotes de controlo e de dados também são processados e enviados a partir desta camada, tornando-a uma peça fulcral no comando da aplicação em si.

Interface da linha de comandos

No que diz respeito à interface da aplicação, esta encontra-se implementada nos ficheiros *CLI.c* e *CLI.h*, onde existem funções que permitem uma transferência instantânea com valores predefinidos pelo grupo e que respeitam o que era pedido no guião do trabalho.
Se o utilizador especificar todos os parâmetros inicialmente, a aplicação irá predefinir o *baud rate*, tamanho máximo da mensagem, número de tentativas em caso de falha de comunicação e intervalo entre cada tentativa como foram especificados pelo grupo.
Caso o utilizador não coloque parâmetros e apenas execute a aplicação, então é requisitada ao utilizador a inserção manual dos valores lá descritos.
Para facilitar a interação com o utilizador, a interface imprime os valores válidos para inserção nos parâmetros que assim o exijam. Após a inserção manual dos valores a aplicação é executada com as propriedades definidas pelo utilizador e é iniciada a transferência.

É de referir que o módulo da *interface* interage com a camada de aplicação, com o objetivo de iniciar o programa com os valores editados pelo utilizador. Esta interação basicamente inicia o construtor da aplicação com outros valores que não os predefinidos.

Estrutura do código

Application layer

Nos ficheiros *ApplicationLayer.c* e *ApplicationLayer.h* encontra-se a implementação da *camada de aplicação* e as suas funções principais.

A camada em si é representada por uma estrutura de dados onde é guardado o descritor de ficheiro da porta de série, o modo de ligação usado, e ainda o nome do ficheiro a ser enviado ou recebido.

```
typedef struct {
      // serial port file descriptor
      int fd;

      // connection mode
      ConnnectionMode mode;

      // name of the file to be sent/received
      char* fileName;
} ApplicationLayer;
```

Figura 1 - Application layer structure.

As funções principais da camada encontram-se descritas na figura em baixo.

```
int sendFile();
int receiveFile();

int sendControlPackage(int fd, int C, char* fileSize, char* fileName);
int receiveControlPackage(int fd, int* ctrl, int* fileLength, char**
fileName);

int sendDataPackage(int fd, int N, const char* buf, int length);
int receiveDataPackage(int fd, int* N, char** buf, int* length);
```

Figura 2 - Funções principais da Application Layer.

Data link layer

Nos ficheiros *DataLink.c* e *Datalink.h* encontra-se a implementação da *camada de ligação de dados* e as suas funções principais.

A camada em si é representada por uma estrutura de dados onde é guardada a porta de série utilizada na ligação, o modo de ligação, o baud rate, o tamanho máximo de uma mensagem, o número de sequência da trama esperada, o valor de time out, o número de tentativas antes de abortar a ligação, as estruturas do termios usado pelo sistema operativo e o termios usado pelo nosso programa, um apontador para as estatísticas da ligação de dados.

```
typedef struct {
      // port /dev/ttySx
      char port[20];

      // connection mode
      ConnnectionMode mode;

      // transmission speed
      int baudRate;

      int messageDataMaxSize;

      // frame sequence number (0, 1)
      ui ns;

      // timeout value
      ui timeout;

      // number of retries in case of failure
      ui numTries;

      // old and new termios
      struct termios oldtio, newtio;

      Statistics* stats;
} LinkLayer;
```

Figura 3 - Link layer structure.

As estatísticas são representadas por uma estrutura de dados onde se registam: o número de mensagens enviadas e recebidas, o número de time outs, o número de RR enviados e recebidos, e ainda o número de REJ enviados e recebidos.

```
typedef struct {
      int sentMessages;
      int receivedMessages;

      int timeouts;

      int numSentRR;
      int numReceivedRR;

      int numSentREJ;
      int numReceivedREJ;
} Statistics;
```

Figura 4 - Statistics structure.

Finalmente, as mensagens também são representadas por uma estrutura de dados onde se guarda o tipo de mensagem – que pode ser do tipo *COMMAND*, *DATA* ou *INVALID*; o *ns* ou *nr associado*; caso seja um comando, o comando associado à mensagem; caso seja uma mensagem de informação, a mensagem e o tamanho da mensagem associada; e ainda o tipo de erro da mensagem, caso seja uma mensagem de erro.

```
typedef struct {
      MessageType type;

      int ns, nr;

      Command command;

      struct {
            unsigned char* message;
            ui messageSize;
      } data;

      MessageError error;
} Message;
```

Figura 5 - Message structure.

Casos de uso principais

A aplicação resultante do trabalho possui diversos casos de uso, existindo os casos direcionadas à parte da transferência do ficheiro como os casos em que há interação com o utilizador através da inserção dos parâmetros do programa manualmente.
Iniciando com os casos de uso referentes à parte da transferência temos uma divisão em duas sequências, sendo elas analisadas de seguida:

- No caso de o programa estar a ser executado como **Emissor** as funções mais importantes são:
  - ***sendFile***, que interage com diretamente com as funções ***llclose*** e ***llopen***, porém através de outras funções como ***sendControlPackage*** e ***sendDataPackage*** também interage com a função ***llwrite***, sendo esta a principal função de comunicação.
  - Funções como a ***sendMessage*** e a ***createMessage*** são, também elas, de extrema importância pois é a partir delas que são elaboradas as mensagens a enviar com os dados e respetivos pacotes de controlo. A função de ***stuff***, responsável pelo mecanismo de transparência da mensagem, é chamada aquando a execução de ***sendMessage***.
  - Funções já não tão relevantes mas que contribuem para o bom desenvolvimento de todo o mecanismo de transferência são ***setAlarm***, sendo aqui predefindo os tempos de espera para novas tentativas de envio, ***messageIsCommand*** onde é analisado o tipo de dados no envio ou recepção a fazer (*SET, UA, DISC*).
- No que diz respeito aos casos de um **Recetor** as funções de maior relevo são:
  - ***receiveFile***, que assim como a ***sendFile*** no caso do **Emissor** interage com o ***sendControlPackage*** e o ***sendDataPackage*** e com as funções ***llopen*** e ***llclose*** para as operações de abertura e encerramento da porta de série. Porém neste ponto há que frisar a relativa importância da função ***llread*** que será encarregue do tratamento da mensagem recebida e da comunicação do *RR* assinalando o seu sucesso de envio.
  - No caso do ***Recetor***, é bom fazer referência à função ***receiveMessage*** que atua como uma máquina de estados analisando as várias *flags* provenientes da mensagem recebida. É aqui onde se executa a função de ***destuff***, cujo objetivo é reverter o resultado produzida pela função de ***stuff***.

Na parte que diz respeito à interação com o utilizador as funções são simples mas também elas contém o seu peso para o bom funcionamento do programa. Entre elas referimos algumas como ***startCLI***, chamada no início do programa na função ***main*** quando o utilizador executa o programa sem quaisquer parâmetros, ***getIntInput*** e ***getStringInput*** que servem para obter e processar os dados introduzidos pelo utilizador durante a execução de ***startCLI***.

Na secção de Anexos é possível encontrar várias imagens ilustrativas das sequências de chamadas das funções.

Protocolo de ligação lógica

A *data link layer* é uma das camadas implementadas neste projecto, da qual a *application layer* depende. A camada de ligação de dados é responsável pelas seguintes funcionalidades:

- Estabelecer e terminar uma ligação através da porta de série, bem como escrever e ler mensagens da mesma;
- Criar e enviar comandos através da porta de série;
- Criar e enviar mensagens através da porta de série;
- Receber mensagens através da porta de série;
- Fazer *stuff* e *destuff* de pacotes da camada superior – *application layer*.

API da porta de série

As funções ***llopen***, ***llwrite***, ***llread*** e ***llclose*** constituem a *API* da porta de série e estão implementadas na camada de ligação de dados.

llopen

A função ***llopen*** é a responsável por estabelecer uma ligação através da porta de série.

Quando o emissor invoca esta função, o comando ***SET*** é enviado através da porta de série e aguarda a resposta do recetor, ou seja, o comando ***UA***. Se entretanto a resposta não chegar e o tempo de *time out* definido for excedido, com a ajuda de um alarme, é feita uma nova tentativa e o comando ***SET*** é reenviado. Este ciclo repete-se até o número de tentativas ser ultrapassado, caso em que a ligação é abortada; ou até o comando ***UA*** ser recebido como resposta do envio do comando ***SET***, caso em que a ligação foi corretamente estabelecida.

Quando o recetor invoca esta função, aguarda a receção do comando ***SET***, e quando o recebe, envia o comando de resposta ***UA*** e a ligação é estabelecida.

llwrite

A função ***llwrite*** recebe um buffer que tenta escrever para a porta de série com recurso à função ***sendMessage()*** e fica a aguardar a receção de uma resposta. Caso uma resposta não seja recebida num intervalo de tempo pré definido em *time out*, é feita uma nova tentativa de envio da mensagem. Quando uma resposta for recebida, caso essa resposta seja o comando ***RR***, a mensagem foi transmitida correctamente; caso a resposta seja o comando ***REJ***, a mensagem não foi transmitida correctamente e, por, isso, a mensagem é retransmitida e o número de tentativas efetuadas volta a zero.

llread

A função ***llread*** tenta receber uma mensagem através da porta de série. Caso receba uma mensagem inválida, envia o comando ***REJ*** através da porta de série; se a mensagem recebida for o comando ***DISC***, significa que a ligação deve ser terminada;

caso a mensagem recebida seja uma mensagem de informação, essa informação é guardada e o comando ***RR*** é enviado através da porta de série.

llclose

A função ***llclose*** é responsável por terminar a ligação através da porta de série.
Se o emissor chamar esta função, o comando ***DISC*** é enviado pela porta de série, aguarda pela receção do comando ***DISC***, e finalmente envia o comando ***UA***.
O recetor aguarda pelo comando ***DISC***, quando o receber reenvia-o e aguarda pela receção do comando ***UA***.
Finalmente, a ligação através da port de série é efetivamente terminada.

Protocolo de aplicação

A *application layer* é a camada de mais alto nível implementada neste projeto e é responsável pelas seguintes funcionalidades:

- Envio/receção de pacotes de controlo;
- Envio/receção de pacotes de dados;
- Envio/receção do ficheiro especificado.

Pacotes do nível de aplicação

Os pacotes do nível de aplicação podem ser de dois tipos: pacotes de dados ou pacotes de controlo. Estes últimos, por sua vez, podem ainda ser classificados como pacotes de controlo inicial e final.

A camada de aplicação consegue diferenciar estes três tipos de pacotes através do primeiro byte do pacote – denominado por campo de controlo – que pode tomar os valores 1, 2 ou 3.

```
typedef enum {
      CTRL_PKG_DATA = 1, CTRL_PKG_START = 2, CTRL_PKG_END = 3
} ControlPackageType;
```

Figura 6 - Control package type enumerator.

Envio e receção de pacotes de controlo

Os pacotes de controlo de início e fim são pacotes que marcam o início e o fim do envio/receção de um ficheiro, respetivamente.
O envio e receção de pacotes de controlo é mediado pelas funções abaixo ilustradas.

```
int sendControlPackage(int fd, int C, char* fileSize, char* fileName);
int receiveControlPackage(int fd, int* ctrl, int* fileLength, char**
fileName);
```

Figura 7 - Funções responsáveis pelo envio e receção de pacotes de controlo.

A função de envio encarrega-se de colocar no pacote de controlo inicial o tipo de pacote de controlo, o tamanho do ficheiro, e ainda o nome do ficheiro. Depois de ter construído o pacote, envia o mesmo recorrendo à função ***llwrite***, implementada na camada inferior, a camada de ligação de dados.
De forma semelhante, a função de receção de pacotes de controlo, lê e guarda um pacote de controlo recebido através da função ***llread***. De seguida processa o seu conteúdo: tipo de pacote de controlo; e caso seja um pacote de controlo inicial, processa o tamanho do ficheiro e o nome do ficheiro registados no mesmo.

Envio e receção de pacotes de dados

Os pacotes de dados são os pacotes que transportam as tramas de informação. É nestes pacotes que os "pedaços"/*chunks* do ficheiro a transferir são transportados.
O envio e receção de pacotes de dados é mediado pelas funções abaixo ilustradas.

```
int sendDataPackage(int fd, int N, const char* buf, int length);
int receiveDataPackage(int fd, int* N, char** buf, int* length);
```

Figura 8 - Funções responsáveis pelo envio e receção de pacotes de dados.

A função de envio de pacotes de dados recebe o apontador para um buffer que contém um *chunk* – parte do ficheiro a ser transmitido – e o seu comprimento. A função encarrega-se de construir um pacote de dados válido com o tipo de pacote no primeiro *byte*, o comprimento do *chunk* a ser transferido nos dois *bytes* seguintes (onde o primeiro *byte* tem o resultado inteiro da divisão do comprimento da trama por 256 e o segundo *byte* o resto dessa mesma divisão), e finalmente o *chunk* nos restantes *bytes*. Para terminar, o pacote é enviado, novamente, com recurso à função ***llwrite***, da camada de ligação de dados, que se encarregará de garantir a transparência no envio de tramas de informação.

De forma algo análoga, mas na ordem inversa, a função de receção de pacotes de dados lê um pacote de dados com auxílio da função ***llread*** e processa o seu conteúdo de forma a reconstruir o *chunk* do ficheiro a ser transmitido tal como ele foi enviado.

Envio e receção do ficheiro

O envio e receção de ficheiros é mediado pelas funções abaixo ilustradas.

```
int sendFile();
int receiveFile();
```

Figura 9 - Funções responsáveis pelo envio e receção de um ficheiro.

A função ***sendFile()*** encarrega-se de enviar um ficheiro através da porta de série. Primeiramente, abre o ficheiro a ser enviado em modo *read* e *binary*; de seguida inicia a ligação via porta de série através da chamada ***llopen***; envia o pacote de controlo inicial com o tamanho e o nome do ficheiro a ser transmitido; lê um *chunk* do ficheiro e envia um pacote de dados contendo esse *chunk* para a porta de série; repete o processo de leitura de um *chunk* e envio do mesmo até a totalidade do ficheiro ser enviada; finalmente, fecha o ficheiro, envia o pacote de controlo final e termina a ligação estabelecida com recurso à função ***llclose***.

A função ***receiveFile()*** é usada para receber um ficheiro através da porta de série. Esta função inicia a ligação com a porta de série; aguarda a receção de um pacote de controlo inicial, que transporta o tamanho e o nome do ficheiro que irá receber; cria e abre o ficheiro de output, onde serão depositados os *chunks* a receber; aguarda a receção de um pacote de dados com um dos *chunks* do ficheiro a receber e escreve-o para o ficheiro de output; repete a espera de um pacote de dados e a escrita do *chunk* contido nele até a totalidade do ficheiro ser recebida; fecha o ficheiro de output; aguarda a receção do pacote de controlo final; e finalmente, termina a ligação com a porta de série.

Validação

Nesta secção são abordados os testes realizados com a aplicação no que diz respeito a uma transferência de ficheiros e a sua devida apresentação de resultados e relativo comentário.

Relativamente à transferência da imagem *pinguim.png* exigida no guião do projeto, decidimos colocar aqui as imagens para visualização do ocorrido e apresentação das estatísticas registadas.

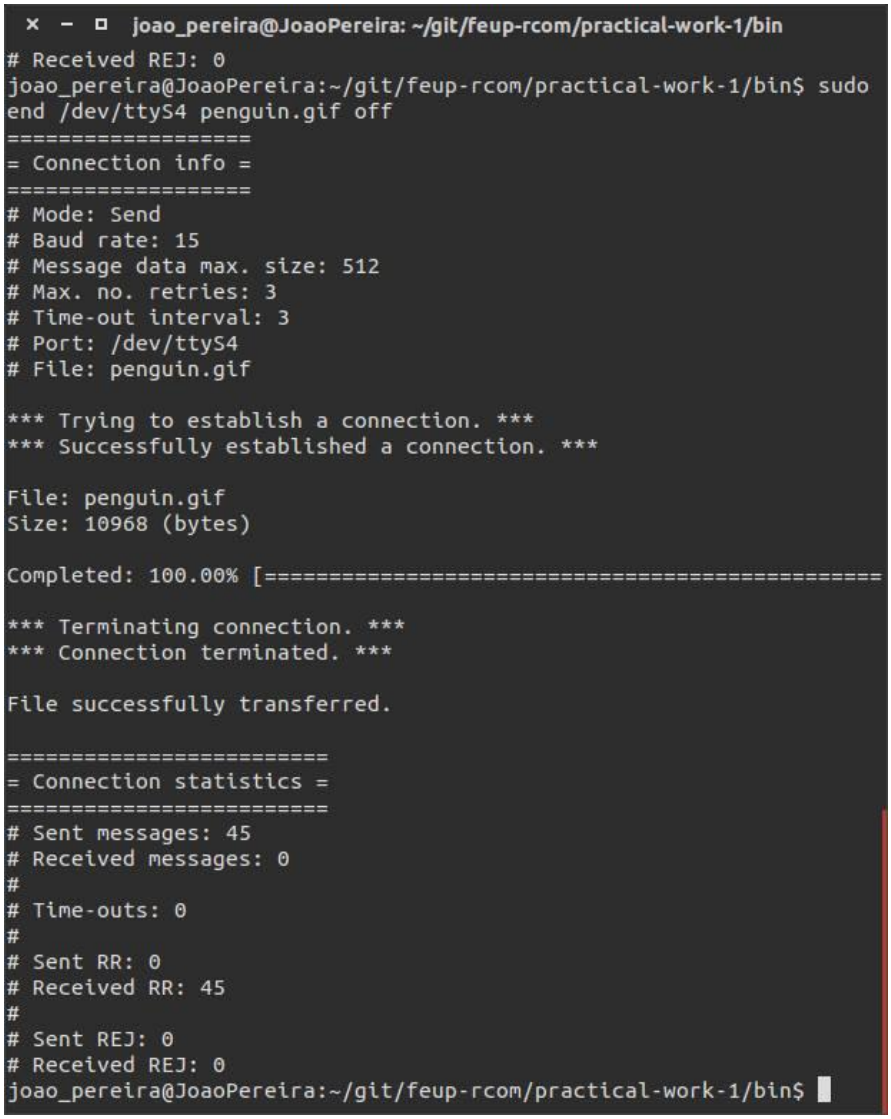


Figura 10 - Modo Emissor

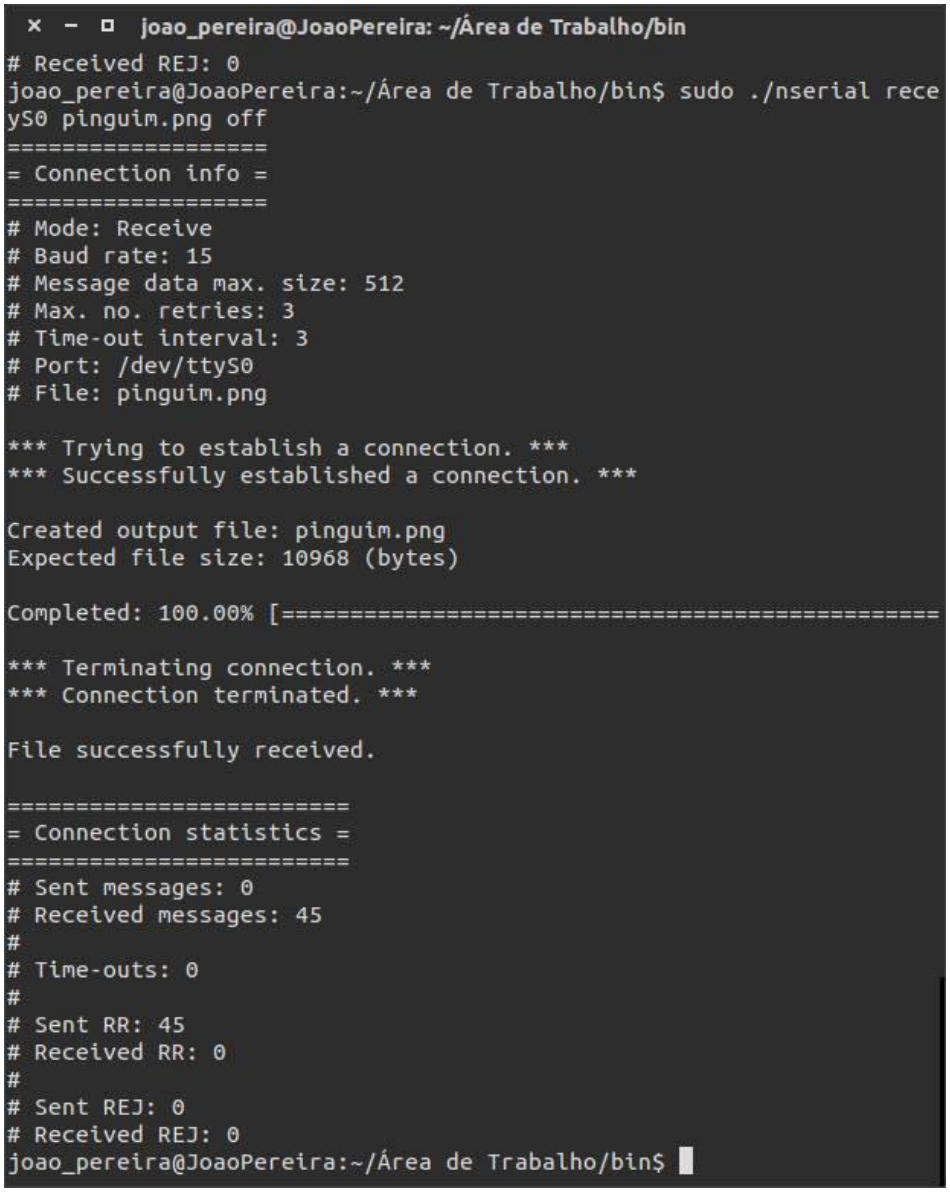


Figura 11 - Modo Recetor

O grupo, além do ficheiro acima descrito, realizou mais testes com imagens diferentes e até com uma música, havendo sucesso em todas as transferências.

Deixamos aqui os dados relativos a uma transferência de uma imagem com um tamanho bem maior que a apresentada no guião.

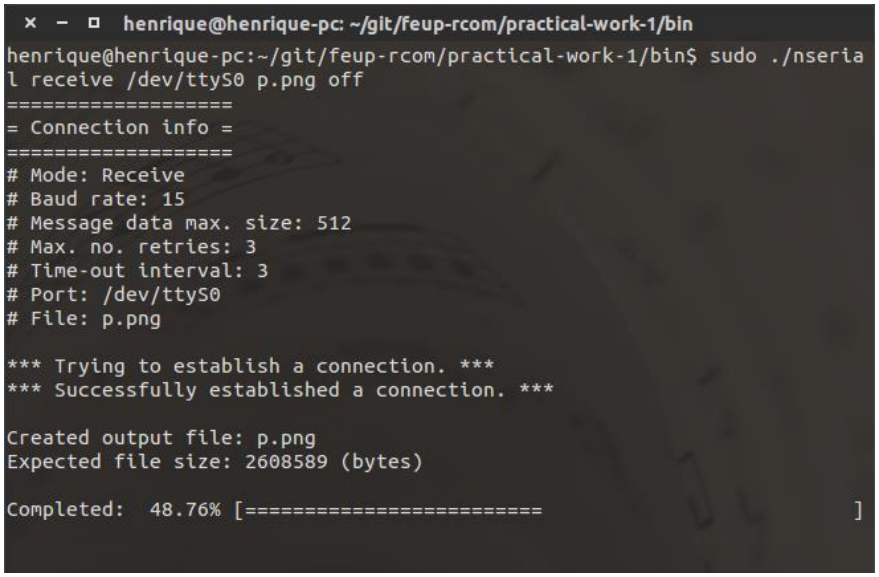


Figura 12 - Dados iniciais e barra de progresso da transferência

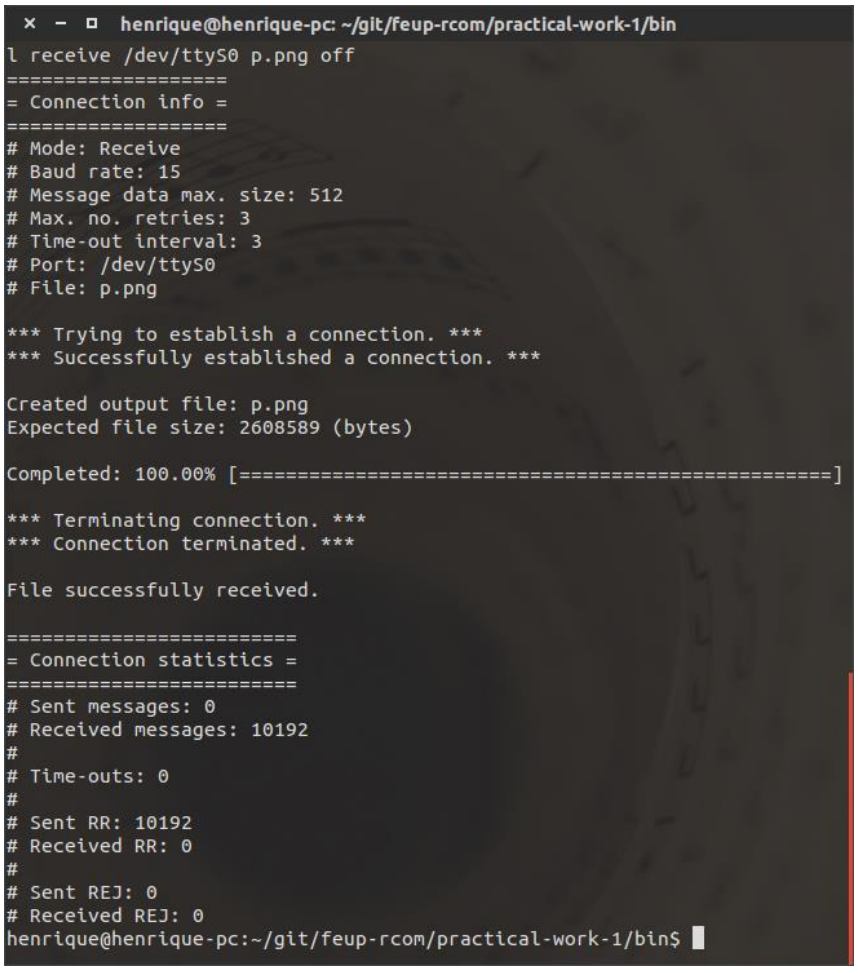


Figura 13 - Fim de transferência e apresentação das estatísticas da conexão

Relativamente aos testes executados com a extração do cabo da porta de série, não possuímos imagens, porém esse teste foi comprovado pelo docente aquando da apresentação do projeto e todos eles terem sido bem-sucedidos.

Elementos de valorização

Selecção de parâmetros pelo utilizador

Quando o utilizador corre o programa sem fornecer quaisquer argumentos através da linha de comandos, uma pequena interface surge onde podem ser especificadas variáveis de configuração da ligação através da porta de série, tais como: *baud rate*, número de tentativas, o valor de *time out*, etc.

Implementação de REJ

Quando ocorre um erro do tipo ***BCC2*** na função llread, o comando ***REJ*** é enviado para que o emissor reenvie a mensagem que não chegou ao recetor corretamente.

```
int llread(int fd, unsigned char** message) {
      Message* msg = NULL;

      int done = 0;
      while (!done) {
            msg = receiveMessage(fd);

            switch (msg->type) {
            case INVALID:
                  if (DEBUG_MODE)
                  printf("INVALID message received.\n");

                  if (msg->error == BCC2_ERROR) {
                        ll->ns = msg->ns;
                        sendCommand(fd, REJ);
                  }
                  break;
            case COMMAND:
                  (…)
                  break;
            }
      }

      stopAlarm();

      return 1;
}
```

Figura 14 - Implementação de REJ.

Verificação da integridade dos dados pela aplicação

A aplicação verifica que o tamanho do ficheiro recebido é igual ao tamanho do ficheiro enviado; também verifica a numeração de cada pacote de forma a garantir que pacotes duplicados são ignorados.

Registo de ocorrências

A aplicação também vai registando, ao longo do tempo de execução, na estrutura *Statistics*, que faz parte da estrutura *LinkLayer* as várias ocorrências de ***RR***, e ***REJ***, bem como o número de mensagens enviadas e recebidas.
Um sumário destas estatísticas é exibido após a ligação através da porta de série ter sido terminada.

Conclusões

Durante as últimas semanas de aulas o grupo teve em mãos o desenvolvimento de uma aplicação capaz de transferir ficheiros entre máquinas através de uma ligação com as portas de séries.

Nas primeiras aulas do semestre foram-nos fornecidos uns guiões iniciais que contribuíram para o bom entendimento acerca do funcionamento dos tipos de receção da porta de série. Entre os diversos tipos de ligação aprendemos o funcionamento da canónica, não canónica e a assíncrona, esta última aplicada neste trabalho desenvolvido pelo grupo.

Relativamente ao trabalho, o grupo compreendeu bem todos os pontos pedidos no guião estando este divido em duas grandes camadas como foi abordado na secção da *Arquitetura* deste relatório. As camadas possuem uma ligação unidirecional, sendo de a aplicação a exercer o controlo perante a de ligação de dados e o contrário não se sucede. Ambas as camadas sã independentes, assim como assinala o *princípio de independência de camadas*. O cabeçalho dos pacotes a transportar nas tramas de informação não é processado na camada de ligação de dados, é invisível para ela. Nesta camada, de ligação de dados, não existe distinção entre pacotes de dados nem de controlo e as numerações dos pacotes são de análise desnecessária.
No que diz respeito à camada da aplicação, esta não conhece as propriedades da camada de ligação de dados porém acede aos serviços por ela fornecidos. As estruturas, o mecanismo de delineação, proteção e retransmissões de tramas são desconhecidos por esta camada, como era abordado no guião do trabalho, assim como a existência de *stuffing* e *destuffing* são tudo processos tratados ao nível da camada ligação de dados

É bom relevar a importância do docente na ajuda ao grupo durante as aulas laboratoriais ao esclarecer pontos que estavam trémulos no que diz respeito à ligação que deveria existir entre as duas camadas. Este por ventura terá sido o grande obstáculo no desenrolar de toda a atividade produzida até aqui.

A realização deste projeto contribuiu para a consolidação dos conceitos interiorizados nas aulas teóricas e laboratoriais, para um conhecimento mais profundo de como a comunicação em redes funciona e até para um uso, novamente, da porta de série, assim como já havíamos feito em outra unidade curricular do curso.

Anexo I

Alarm.c

```
#include "Alarm.h"

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include "DataLink.h"

int alarmWentOff = 0;

void alarmHandler(int signal) {
    if (signal != SIGALRM)
        return;

    alarmWentOff = TRUE;
    ll->stats->timeouts++;
    printf("Connection time out!\n\nRetrying:\n");

    alarm(ll->timeout);
}

void setAlarm() {
    struct sigaction action;
    action.sa_handler = alarmHandler;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;

    sigaction(SIGALRM, &action, NULL);

    alarmWentOff = FALSE;

    alarm(ll->timeout);
}

void stopAlarm() {
    struct sigaction action;
    action.sa_handler = NULL;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;

    sigaction(SIGALRM, &action, NULL);

    alarm(0);
}
```

Alarm.h

```
#pragma once

extern int alarmWentOff;

void alarmHandler(int signal);

void setAlarm();
```

```
void stopAlarm();
```

Application.c

```
#include "Aplication.h"

#include <errno.h>
#include <math.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include "DataLink.h"
#include "CLI.h"
#include "ConnectionMode.h"
#include "ControlPackageType.h"
#include "ParameterType.h"
#include "Utilities.h"

int DEBUG_MODE = FALSE;

ApplicationLayer* al;

int initApplicationLayer(const char* port, ConnnectionMode mode, int
baudrate,
        int messageDataMaxSize, int numRetries, int timeout, char*
file) {
    al = (ApplicationLayer*) malloc(sizeof(ApplicationLayer));

    al->fd = openSerialPort(port);
    if (al->fd < 0) {
        printf("ERROR: Serial port could not be open.\n");
        return 0;
    }
    al->mode = mode;
    al->fileName = file;

    if (!initLinkLayer(port, mode, baudrate, messageDataMaxSize,
numRetries,
            timeout)) {
        printf("ERROR: Could not initialize Link Layer.\n");
        return 0;
    }

    printConnectionInfo();

    startConnection();

    if (!closeSerialPort(al->fd))
        return 0;

    printConnectionStatistics();

    return 1;
}

void printConnectionInfo() {
```

```
    printf("====================\n");
    printf("= Connection info =\n");
    printf("====================\n");

    switch (ll->mode) {
    case SEND:
        printf("# Mode: Send\n");
        break;
    case RECEIVE:
        printf("# Mode: Receive\n");
        break;
    default:
        printf("# ERROR: Mode: Default\n");
        break;
    }

    printf("# Baud rate: %d\n", ll->baudRate);
    printf("# Message data max. size: %d\n", ll->messageDataMaxSize);
    printf("# Max. no. retries: %d\n", ll->numTries - 1);
    printf("# Time-out interval: %d\n", ll->timeout);
    printf("# Port: %s\n", ll->port);
    printf("# File: %s\n", al->fileName);

    printf("\n");
}

int startConnection() {
    switch (al->mode) {
    case RECEIVE:
        if (DEBUG_MODE)
            printf("Starting connection in RECEIVE mode.\n");
        receiveFile();
        break;
    case SEND:
        if (DEBUG_MODE)
            printf("Starting connection in SEND mode.\n");
        sendFile();
        break;
    default:
        break;
    }

    return 1;
}

void printConnectionStatistics() {
    printf("\n");
    printf("=========================\n");
    printf("= Connection statistics =\n");
    printf("=========================\n");

    printf("# Sent messages: %d\n", ll->stats->sentMessages);
    printf("# Received messages: %d\n", ll->stats->receivedMessages);
    printf("#\n");
    printf("# Time-outs: %d\n", ll->stats->timeouts);
    printf("#\n");
    printf("# Sent RR: %d\n", ll->stats->numSentRR);
    printf("# Received RR: %d\n", ll->stats->numReceivedRR);
    printf("#\n");
    printf("# Sent REJ: %d\n", ll->stats->numSentREJ);
    printf("# Received REJ: %d\n", ll->stats->numReceivedREJ);
```

```
}

int sendFile() {
    // open file to be sent
    FILE* file = fopen(al->fileName, "rb");
    if (!file) {
        printf("ERROR: Could not open file to be sent.\n");
        return 0;
    } else if (DEBUG_MODE)
        printf("Successfully opened file to be sent.\n");

    // open connection
    int fd = llopen(ll->mode);
    if (fd <= 0)
        return 0;

    // get size of file to be sent
    int fileSize = getFileSize(file);
    char fileSizeBuf[sizeof(int) * 3 + 2];
    snprintf(fileSizeBuf, sizeof fileSizeBuf, "%d", fileSize);

    // send start control package
    if (!sendControlPackage(al->fd, CTRL_PKG_START, fileSizeBuf, al-
>fileName))
        return 0;

    // allocate space for file buffer
    char* fileBuf = malloc(MAX_SIZE);

    if (DEBUG_MODE)
        printf("*** Starting file chunks transfer. ***\n");

    // read file chunks
    ui readBytes = 0, writtenBytes = 0, i = 0;
    while ((readBytes = fread(fileBuf, sizeof(char), MAX_SIZE, file))
> 0) {
        // send those chunks inside data packages
        if (!sendDataPackage(fd, (i++) % 255, fileBuf, readBytes)) {
            free(fileBuf);
            return 0;
        }

        // reset file buffer
        fileBuf = memset(fileBuf, 0, MAX_SIZE);

        // increment no. of written bytes
        writtenBytes += readBytes;

        printProgressBar(writtenBytes, fileSize);
    }
    printf("\n\n");

    if (DEBUG_MODE)
        printf("*** File chunks transfer complete. ***\n");

    free(fileBuf);

    if (fclose(file) != 0) {
        printf("ERROR: Unable to close file.\n");
        return 0;
    }
```

```
    if (!sendControlPackage(fd, CTRL_PKG_END, "0", ""))
        return 0;

    if (!llclose(al->fd, ll->mode))
        return 0;

    printf("\n");
    printf("File successfully transferred.\n");

    return 1;
}

int receiveFile() {
    int fd = llopen(ll->mode);
    if (fd <= 0)
        return 0;

    // TODO create struct to carry these
    int controlStart, fileSize;
    char* fileName;

    if (DEBUG_MODE)
        printf("Waiting for START control package.\n");
    if (!receiveControlPackage(al->fd, &controlStart, &fileSize,
&fileName))
        return 0;

    if (controlStart != CTRL_PKG_START) {
        printf(
                "ERROR: Control package received but its control field
- %d - is not C_PKG_START",
                controlStart);
        return 0;
    }

    // create output file
    FILE* outputFile = fopen(al->fileName, "wb");
    if (outputFile == NULL) {
        printf("ERROR: Could not create output file.\n");
        return 0;
    }

    printf("\n");
    printf("Created output file: %s\n", al->fileName);
    printf("Expected file size: %d (bytes)\n", fileSize);
    printf("\n");

    if (DEBUG_MODE)
        printf("*** Starting file chunks transfer. ***\n");

    int fileSizeReadSoFar = 0, N = -1;
    while (fileSizeReadSoFar != fileSize) {
        int lastN = N;
        char* fileBuf = NULL;
        int length = 0;

        // receive data package with chunk and put chunk in fileBuf
        if (!receiveDataPackage(al->fd, &N, &fileBuf, &length)) {
            printf("ERROR: Could not receive data package.\n");
            free(fileBuf);
```

```
            return 0;
        }

        if (N != 0 && lastN + 1 != N) {
            printf("ERROR: Received sequence no. was %d instead of
%d.\n", N,
                    lastN + 1);
            free(fileBuf);
            return 0;
        }

        // write received chunk to output file
        fwrite(fileBuf, sizeof(char), length, outputFile);
        free(fileBuf);

        // increment no. of read bytes
        fileSizeReadSoFar += length;

        printProgressBar(fileSizeReadSoFar, fileSize);
    }
    printf("\n\n");

    if (DEBUG_MODE)
        printf("*** File chunks transfer complete. ***\n");

    // close output file
    if (fclose(outputFile) != 0) {
        printf("ERROR: Closing output file.\n");
        return -1;
    }

    // receive end control package
    int controlPackageTypeReceived = -1;
    if (!receiveControlPackage(al->fd, &controlPackageTypeReceived, 0,
NULL)) {
        printf("ERROR: Could not receive END control package.\n");
        return 0;
    }

    if (controlPackageTypeReceived != CTRL_PKG_END) {
        printf("ERROR: Control field received (%d) is not END.\n",
                controlPackageTypeReceived);
        return 0;
    }

    if (!llclose(al->fd, ll->mode)) {
        printf("ERROR: Serial port was not closed.\n");
        return 0;
    }

    printf("\n");
    printf("File successfully received.\n");

    return 1;
}

int sendControlPackage(int fd, int C, char* fileSize, char* fileName)
{
    if (DEBUG_MODE) {
        if (C == CTRL_PKG_START)
            printf("Sending START control package.\n");
```

```
        else if (C == CTRL_PKG_END)
            printf("Sending END control package.\n");
        else
            printf("WARNING: Sending UNKNOWN control package (C =
%d).\n", C);
    }

    // calculate control package size
    int packageSize = 5 + strlen(fileSize) + strlen(fileName);
    ui i = 0, pos = 0;

    // create control package
    unsigned char controlPackage[packageSize];
    controlPackage[pos++] = C;
    controlPackage[pos++] = PARAM_FILE_SIZE;
    controlPackage[pos++] = strlen(fileSize);
    for (i = 0; i < strlen(fileSize); i++)
        controlPackage[pos++] = fileSize[i];
    controlPackage[pos++] = PARAM_FILE_NAME;
    controlPackage[pos++] = strlen(fileName);
    for (i = 0; i < strlen(fileName); i++)
        controlPackage[pos++] = fileName[i];

    if (C == CTRL_PKG_START) {
        printf("\n");
        printf("File: %s\n", fileName);
        printf("Size: %s (bytes)\n", fileSize);
        printf("\n");
    }

    // send control package
    if (!llwrite(fd, controlPackage, packageSize)) {
        printf(
                "ERROR: Could not write to link layer while sending
control package.\n");
        free(controlPackage);

        return 0;
    }

    if (DEBUG_MODE) {
        if (C == CTRL_PKG_START)
            printf("START control package sent.\n");
        else if (C == CTRL_PKG_END)
            printf("END control package sent.\n");
        else
            printf("WARNING: UNKNOWN control package sent (C =
%d).\n", C);
    }

    return 1;
}

int receiveControlPackage(int fd, int* controlPackageType, int*
fileLength,
        char** fileName) {
    // receive control package
    unsigned char* package;
    ui totalSize = llread(fd, &package);
    if (totalSize < 0) {
        printf(
```

```
                "ERROR: Could not read from link layer while receiving
control package.\n");
        return 0;
    }

    // process control package type
    *controlPackageType = package[0];

    if (*controlPackageType == CTRL_PKG_END) {
        if (DEBUG_MODE)
            printf("END control package has been received.\n");
        return 1;
    } else if (DEBUG_MODE)
        printf("START control package has been received.\n");

    ui i = 0, numParams = 2, pos = 1, numOcts = 0;
    for (i = 0; i < numParams; i++) {
        int paramType = package[pos++];

        switch (paramType) {
        case PARAM_FILE_SIZE: {
            numOcts = (ui) package[pos++];

            char* length = malloc(numOcts);
            memcpy(length, &package[pos], numOcts);

            *fileLength = atoi(length);
            free(length);

            break;
        }
        case PARAM_FILE_NAME:
            numOcts = (unsigned char) package[pos++];
            memcpy(*fileName, &package[pos], numOcts);

            break;
        }
    }

    return 1;
}

int sendDataPackage(int fd, int N, const char* buffer, int length) {
    unsigned char C = CTRL_PKG_DATA;
    unsigned char L2 = length / 256;
    unsigned char L1 = length % 256;

    // calculate package size
    ui packageSize = 4 + length;

    // allocate space for package header and file chunk
    unsigned char* package = (unsigned char*) malloc(packageSize);

    // build package header
    package[0] = C;
    package[1] = N;
    package[2] = L2;
    package[3] = L1;

    // copy file chunk to package
    memcpy(&package[4], buffer, length);
```

```
    // write package
    if (!llwrite(fd, package, packageSize)) {
        printf(
                "ERROR: Could not write to link layer while sending
data package.\n");
        free(package);

        return 0;
    }

    free(package);

    return 1;
}

int receiveDataPackage(int fd, int* N, char** buf, int* length) {
    unsigned char* package;

    // read package from link layer
    ui size = llread(fd, &package);
    if (size < 0) {
        printf(
                "ERROR: Could not read from link layer while receiving
data package.\n");
        return 0;
    }

    int C = package[0];
    *N = (unsigned char) package[1];
    int L2 = package[2];
    int L1 = package[3];

    // assert package is a data package
    if (C != CTRL_PKG_DATA) {
        printf("ERROR: Received package is not a data package (C =
%d).\n", C);
        return 0;
    }

    // calculate size of the file chunk contained in the read package
    *length = 256 * L2 + L1;

    // allocate space for that file chunk
    *buf = malloc(*length);

    // copy file chunk to the buffer
    memcpy(*buf, &package[4], *length);

    // destroy the received package
    free(package);

    return 1;
}
```

Application.h

```
#pragma once
```

```
#include <stdio.h>
#include <termios.h>
#include "ConnectionMode.h"

typedef struct {
    // serial port file descriptor
    int fd;

    // connection mode
    ConnnectionMode mode;

    // name of the file to be sent/received
    char* fileName;
} ApplicationLayer;

extern int DEBUG_MODE;
extern ApplicationLayer* al;

int initApplicationLayer(const char* port, ConnnectionMode mode, int
baudrate,
        int messageDataMaxSize, int numRetries, int timeout, char*
file);
void printConnectionInfo();
int startConnection();
void printConnectionStatistics();

int sendFile();
int receiveFile();

int sendControlPackage(int fd, int C, char* fileSize, char* fileName);
int receiveControlPackage(int fd, int* ctrl, int* fileLength, char**
fileName);

int sendDataPackage(int fd, int N, const char* buf, int length);
int receiveDataPackage(int fd, int* N, char** buf, int* length);
```

CLI.c

```
#include "CLI.h"

#include <stdio.h>
#include <stdlib.h>
#include "Aplication.h"
#include "ConnectionMode.h"
#include "DataLink.h"

void startCLI() {
    printf("============================\n");
    printf("= RCOM - practical work 1 =\n");
    printf("============================\n");
    printf("\n");
    printf("Choose what to do:\n");
    printf("    1. Send    2. Receive\n");
    printf("\n");

    ConnnectionMode mode = getIntInput(1, 2) - 1;
    printf("\n");

    int baudrate;
```

```
    do {
        printf(
                "What Baud rate should be used? { 0, 50, 75, 110, 134,
150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600,
115200, 230400, 460800 } ");
        baudrate = getBaudrate(getIntInput(0, 460800));
        printf("\n");
    } while (baudrate == -1);

    printf("Which should be the message data maximum size? ");
    int messageDataMaxSize = getIntInput(1, 512);
    printf("\n");

    printf(
            "Which is the maximum number of retries before aborting
the connection? ");
    int numRetries = getIntInput(0, 10);
    printf("\n");

    printf("How many seconds should the program wait until a time-out?
");
    int timeout = getIntInput(1, 10);
    printf("\n");

    printf("What port - x - should be used? (/dev/ttySx) ");
    int portNum = getIntInput(0, 9);

    char port[20] = "/dev/ttySx";
    port[9] = '0' + portNum;
    printf("Using port: %s\n", port);
    printf("\n");

    switch (mode) {
    case SEND:
        printf("What file do you wish to transfer? ");
        break;
    case RECEIVE:
        printf("Which name should the received file have? ");
        break;
    default:
        printf("Unexpected error: Invalid mode.\n");
        break;
    }

    char* file = getStringInput();
    printf("\n");

    initApplicationLayer(port, mode, baudrate, messageDataMaxSize,
numRetries,
            timeout, file);
}

int getIntInput(int start, int end) {
    int input;

    int done = 0;
    while (!done) {
        printf("$ ");

        if (scanf("%d", &input) == 1 && start <= input && input <=
end)
```

```
            done = 1;
        else
            printf("Invalid input. Try again:\n");

        clearInputBuffer();
    }

    return input;
}

char* getStringInput() {
    char* input = (char*) malloc(256 * sizeof(char));

    int done = 0;
    while (!done) {
        printf("$ ");

        if (scanf("%s", input) == 1)
            done = 1;
        else
            printf("Invalid input. Try again:\n");

        clearInputBuffer();
    }

    return input;
}

void clearInputBuffer() {
    int c;
    while ((c = getchar()) != '\n' && c != EOF)
        ;
}

const int PROGRESS_BAR_LENGTH = 51;

void printProgressBar(float current, float total) {
    float percentage = 100.0 * current / total;

    printf("\rCompleted: %6.2f%% [", percentage);

    int i, len = PROGRESS_BAR_LENGTH;
    int pos = percentage * len / 100.0;

    for (i = 0; i < len; i++)
        i <= pos ? printf("=") : printf(" ");

    printf("]");

    fflush(stdout);
}
```

CLI.h

```
#pragma once

void startCLI();

int getIntInput(int start, int end);
```

```
char* getStringInput();

void clearInputBuffer();

void printProgressBar(float current, float total);
```

ConnectionMode.h

```
#pragma once

typedef enum {
    SEND, RECEIVE
} ConnnectionMode;
```

ControlPackageType.h

```
#pragma once

typedef enum {
    CTRL_PKG_DATA = 1, CTRL_PKG_START = 2, CTRL_PKG_END = 3
} ControlPackageType;
```

DataLink.c

```
#include "DataLink.h"

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include "Alarm.h"
#include "Aplication.h"

const int FLAG = 0x7E;
const int A = 0x03;
const int ESCAPE = 0x7D;

LinkLayer* ll;

int getBaudrate(int baudrate) {
    switch (baudrate) {
    case 0:
        return B0;
    case 50:
        return B50;
    case 75:
        return B75;
    case 110:
        return B110;
    case 134:
        return B134;
    case 150:
        return B150;
```

```
    case 200:
        return B200;
    case 300:
        return B300;
    case 600:
        return B600;
    case 1200:
        return B1200;
    case 1800:
        return B1800;
    case 2400:
        return B2400;
    case 4800:
        return B4800;
    case 9600:
        return B9600;
    case 19200:
        return B19200;
    case 38400:
        return B38400;
    case 57600:
        return B57600;
    case 115200:
        return B115200;
    case 230400:
        return B230400;
    case 460800:
        return B460800;
    default:
        return -1;
    }
}

int initLinkLayer(const char* port, ConnnectionMode mode, int
baudrate,
        int messageDataMaxSize, int numRetries, int timeout) {
    ll = (LinkLayer*) malloc(sizeof(LinkLayer));

    strcpy(ll->port, port);
    ll->mode = mode;
    ll->baudRate = baudrate;
    ll->messageDataMaxSize = messageDataMaxSize;
    ll->ns = 0;
    ll->timeout = timeout;
    ll->numTries = 1 + numRetries;
    ll->stats = initStatistics();

    if (!saveCurrentPortSettingsAndSetNewTermios()) {
        printf(
                "ERROR: Could not save current port settings and set
new termios.\n");
        return 0;
    }

    return 1;
}

Statistics* initStatistics() {
    Statistics* stats = (Statistics*) malloc(sizeof(Statistics));

    stats->sentMessages = 0;
```

```
    stats->receivedMessages = 0;

    stats->timeouts = 0;

    stats->numSentRR = 0;
    stats->numReceivedRR = 0;

    stats->numSentREJ = 0;
    stats->numReceivedREJ = 0;

    return stats;
}

int saveCurrentPortSettingsAndSetNewTermios() {
    if (!saveCurrentTermiosSettings()) {
        printf("ERROR: Could not save current termios settings.\n");
        return 0;
    }

    if (!setNewTermios()) {
        printf("ERROR: Could not set new termios settings.\n");
        return 0;
    }

    return 1;
}

int saveCurrentTermiosSettings() {
    if (tcgetattr(al->fd, &ll->oldtio) != 0) {
        printf("ERROR: Could not save current termios settings.\n");
        return 0;
    }

    return 1;
}

int setNewTermios() {
    bzero(&ll->newtio, sizeof(ll->newtio));
    ll->newtio.c_cflag = ll->baudRate | CS8 | CLOCAL | CREAD;
    ll->newtio.c_iflag = IGNPAR;
    ll->newtio.c_oflag = 0;
    ll->newtio.c_lflag = 0;

    // inter-character timer unused
    ll->newtio.c_cc[VTIME] = 3;

    // blocking read until x chars received
    ll->newtio.c_cc[VMIN] = 0;

    if (tcflush(al->fd, TCIOFLUSH) != 0)
        return 0;

    if (tcsetattr(al->fd, TCSANOW, &ll->newtio) != 0)
        return 0;

    if (DEBUG_MODE)
        printf("New termios structure set.\n");

    return 1;
}
```

```
int openSerialPort(const char* port) {
    // Open serial port device for reading and writing and not as
controlling
    // tty because we don't want to get killed if linenoise sends
CTRL-C.
    return open(port, O_RDWR | O_NOCTTY);
}

int closeSerialPort() {
    if (tcsetattr(al->fd, TCSANOW, &ll->oldtio) == -1) {
        perror("tcsetattr");
        return 0;
    }

    close(al->fd);

    return 1;
}

int llopen(ConnnectionMode mode) {
    printf("*** Trying to establish a connection. ***\n");

    int try = 0, connected = 0;

    switch (mode) {
    case SEND: {
        while (!connected) {
            if (try == 0 || alarmWentOff) {
                alarmWentOff = 0;

                if (try >= ll->numTries) {
                    stopAlarm();
                    printf("ERROR: Maximum number of retries
exceeded.\n");
                    printf("*** Connection aborted. ***\n");
                    return 0;
                }

                sendCommand(al->fd, SET);

                if (++try == 1)
                    setAlarm();
            }

            if (messageIsCommand(receiveMessage(al->fd), UA)) {
                connected = 1;

                printf("*** Successfully established a connection.
***\n");
            }
        }

        stopAlarm();

        break;
    }
    case RECEIVE: {
        while (!connected) {
            if (messageIsCommand(receiveMessage(al->fd), SET)) {
                sendCommand(al->fd, UA);
                connected = 1;
```

```
                printf("*** Successfully established a connection.
***\n");
            }
        }

        break;
    }
    default:
        break;
    }

    return al->fd;
}

int llwrite(int fd, const unsigned char* buf, ui bufSize) {
    int try = 0, transferring = 1;

    while (transferring) {
        if (try == 0 || alarmWentOff) {
            alarmWentOff = 0;

            if (try >= ll->numTries) {
                stopAlarm();
                printf("ERROR: Maximum number of retries
exceeded.\n");
                printf("Message not sent.\n");
                return 0;
            }

            sendMessage(fd, buf, bufSize);

            if (++try == 1)
                setAlarm();
        }

        Message* receivedMessage = receiveMessage(fd);

        if (messageIsCommand(receivedMessage, RR)) {
            if (ll->ns != receivedMessage->nr)
                ll->ns = receivedMessage->nr;

            stopAlarm();
            transferring = 0;
        } else if (messageIsCommand(receivedMessage, REJ)) {
            stopAlarm();
            try = 0;
        }
    }

    stopAlarm();

    return 1;
}

int llread(int fd, unsigned char** message) {
    Message* msg = NULL;

    int done = 0;
    while (!done) {
        msg = receiveMessage(fd);
```

```
        switch (msg->type) {
        case INVALID:
            if (DEBUG MODE)
            printf("INVALID message received.\n");

            if (msg->error == BCC2_ERROR) {
                ll->ns = msg->ns;
                sendCommand(fd, REJ);
            }

            break;
        case COMMAND:
            if (msg->command == DISC)
                done = 1;
            break;
        case DATA:
            if (ll->ns == msg->ns) {
                *message = malloc(msg->data.messageSize);
                memcpy(*message, msg->data.message, msg-
>data.messageSize);
                free(msg->data.message);

                ll->ns = !msg->ns;
                sendCommand(fd, RR);

                done = TRUE;

            } else
                printf("\tWrong message ns associated: ignoring
message.\n");

            break;
        }
    }

    stopAlarm();

    return 1;
}

int llclose(int fd, ConnnectionMode mode) {
    printf("*** Terminating connection. ***\n");

    int try = 0, disconnected = 0;

    switch (mode) {
    case SEND: {
        while (!disconnected) {
            if (try == 0 || alarmWentOff) {
                alarmWentOff = FALSE;

                if (try >= ll->numTries) {
                    stopAlarm();
                    printf("ERROR: Maximum number of retries
exceeded.\n");
                    printf("*** Connection aborted. ***\n");
                    return 0;
                }

                sendCommand(fd, DISC);
```

```
                if (++try == 1)
                    setAlarm();
            }

            if (messageIsCommand(receiveMessage(fd), DISC))
                disconnected = TRUE;
        }

        stopAlarm();
        sendCommand(fd, UA);
        sleep(1);

        printf("*** Connection terminated. ***\n");

        return 1;
    }
    case RECEIVE: {
        while (!disconnected) {
            if (messageIsCommand(receiveMessage(fd), DISC))
                disconnected = TRUE;
        }

        int uaReceived = FALSE;
        while (!uaReceived) {
            if (try == 0 || alarmWentOff) {
                alarmWentOff = FALSE;

                if (try >= ll->numTries) {
                    stopAlarm();
                    printf("ERROR: Disconnect could not be sent.\n");
                    return 0;
                }

                sendCommand(fd, DISC);

                if (++try == 1)
                    setAlarm();
            }

            if (messageIsCommand(receiveMessage(fd), UA))
                uaReceived = TRUE;
        }

        stopAlarm();
        printf("*** Connection terminated. ***\n");

        return 1;
    }
    default:
        break;
    }

    return 0;
}

unsigned char* createCommand(ControlField C) {
    unsigned char* command = malloc(COMMAND_SIZE);

    command[0] = FLAG;
    command[1] = A;
```

```
    command[2] = C;
    if (C == C_REJ || C == C_RR)
        command[2] |= (ll->ns << 7);
    command[3] = command[1] ^ command[2];
    command[4] = FLAG;

    return command;
}

int sendCommand(int fd, Command command) {
    char commandStr[MAX_SIZE];
    ControlField ctrlField = getCommandControlField(commandStr,
command);

    unsigned char* commandBuf = createCommand(ctrlField);
    ui commandBufSize = stuff(&commandBuf, COMMAND_SIZE);

    int successfullySentCommand = TRUE;
    if (write(fd, commandBuf, commandBufSize) != COMMAND_SIZE) {
        printf("ERROR: Could not write %s command.\n", commandStr);
        successfullySentCommand = FALSE;
    }

    free(commandBuf);

    if (command == REJ)
        ll->stats->numSentREJ++;
    else if (command == RR)
        ll->stats->numSentRR++;

    if (DEBUG_MODE)
        printf("Sent command: %s.\n", commandStr);

    return successfullySentCommand;
}

Command getCommandWithControlField(ControlField controlField) {
    switch (controlField & 0x0F) {
    case C_SET:
        return SET;
    case C_UA:
        return UA;
    case C_RR:
        return RR;
    case C_REJ:
        return REJ;
    case C_DISC:
        return DISC;
    default:
        printf("ERROR: control field not recognized.\n");
        return SET;
    }
}

ControlField getCommandControlField(char* commandStr, Command command)
{
    switch (command) {
    case SET:
        strcpy(commandStr, "SET");
        return C_SET;
    case UA:
```

```
        strcpy(commandStr, "UA");
        return C_UA;
    case RR:
        strcpy(commandStr, "RR");
        return C_RR;
    case REJ:
        strcpy(commandStr, "REJ");
        return C_REJ;
    case DISC:
        strcpy(commandStr, "DISC");
        return C_DISC;
    default:
        strcpy(commandStr, "* ERROR *");
        return C_SET;
    }
}

unsigned char* createMessage(const unsigned char* message, ui size) {
    unsigned char* msg = malloc(MESSAGE_SIZE + size);

    unsigned char C = ll->ns << 6;
    unsigned char BCC1 = A ^ C;
    unsigned char BCC2 = processBCC(message, size);

    msg[0] = FLAG;
    msg[1] = A;
    msg[2] = C;
    msg[3] = BCC1;
    memcpy(&msg[4], message, size);
    msg[4 + size] = BCC2;
    msg[5 + size] = FLAG;

    return msg;
}

int sendMessage(int fd, const unsigned char* message, ui messageSize)
{
    unsigned char* msg = createMessage(message, messageSize);
    messageSize += MESSAGE_SIZE;

    messageSize = stuff(&msg, messageSize);

    ui numWrittenBytes = write(fd, msg, messageSize);
    if (numWrittenBytes != messageSize)
        perror("ERROR: error while sending message.\n");

    free(msg);

    ll->stats->sentMessages++;

    return numWrittenBytes == messageSize;
}

Message* receiveMessage(int fd) {
    Message* msg = (Message*) malloc(sizeof(Message));
    msg->type = INVALID;
    msg->ns = msg->nr = -1;

    State state = START;

    ui size = 0;
```

```
    unsigned char* message = malloc(ll->messageDataMaxSize);

    volatile int done = FALSE;
    while (!done) {
        unsigned char c;

        // if not stopping
        if (state != STOP) {
            // read message
            int numReadBytes = read(fd, &c, 1);

            // if nothing was read
            if (!numReadBytes) {
                if (DEBUG_MODE)
                    printf("ERROR: nothing received.\n");

                free(message);

                msg->type = INVALID;
                msg->error = INPUT_OUTPUT_ERROR;

                return msg;
            }
        }

        switch (state) {
        case START:
            if (c == FLAG) {
                if (DEBUG_MODE)
                    printf("START: FLAG received. Going to
FLAG_RCV.\n");

                message[size++] = c;

                state = FLAG_RCV;
            }
            break;
        case FLAG_RCV:
            if (c == A) {
                if (DEBUG_MODE)
                    printf("FLAG_RCV: A received. Going to A_RCV.\n");

                message[size++] = c;

                state = A_RCV;
            } else if (c != FLAG) {
                size = 0;

                state = START;
            }
            break;
        case A_RCV:
            if (c != FLAG) {
                if (DEBUG_MODE)
                    printf("A_RCV: C received. Going to C_RCV.\n");

                message[size++] = c;

                state = C_RCV;
            } else if (c == FLAG) {
                size = 1;
```

```
                state = FLAG_RCV;
            } else {
                size = 0;

                state = START;
            }
            break;
        case C_RCV:
            if (c == (message[1] ^ message[2])) {
                if (DEBUG_MODE)
                    printf("C_RCV: BCC received. Going to BCC_OK.\n");

                message[size++] = c;

                state = BCC_OK;
            } else if (c == FLAG) {
                if (DEBUG_MODE)
                    printf("C_RCV: FLAG received. Going back to
FLAG_RCV.\n");

                size = 1;

                state = FLAG_RCV;
            } else {
                if (DEBUG_MODE)
                    printf("C_RCV: ? received. Going back to
START.\n");

                size = 0;

                state = START;
            }
            break;
        case BCC_OK:
            if (c == FLAG) {
                if (msg->type == INVALID)
                    msg->type = COMMAND;

                message[size++] = c;

                state = STOP;

                if (DEBUG_MODE)
                    printf("BCC_OK: FLAG received. Going to STOP.\n");
            } else if (c != FLAG) {
                if (msg->type == INVALID)
                    msg->type = DATA;
                else if (msg->type == COMMAND) {
                    printf("WANING?? something unexpected
happened.\n");
                    state = START;
                    continue;
                }

                // if writing at the end and more bytes will still be
received
                if (size % ll->messageDataMaxSize == 0) {
                    int mFactor = size / ll->messageDataMaxSize + 1;
                    message = (unsigned char*) realloc(message,
                            mFactor * ll->messageDataMaxSize);
```

```
                }

                message[size++] = c;
            }
            break;
        case STOP:
            message[size] = 0;
            done = TRUE;
            break;
        default:
            break;
        }
    }

    size = destuff(&message, size);

    unsigned char A = message[1];
    unsigned char C = message[2];
    unsigned char BCC1 = message[3];

    if (BCC1 != (A ^ C)) {
        printf("ERROR: invalid BCC1.\n");

        free(message);

        msg->type = INVALID;
        msg->error = BCC1_ERROR;

        return msg;
    }

    if (msg->type == COMMAND) {
        // get message command
        msg->command = getCommandWithControlField(message[2]);

        // get command control field
        ControlField controlField = message[2];

        char commandStr[MAX_SIZE];
        getCommandControlField(commandStr, msg->command);

        if (DEBUG_MODE)
            printf("Received command: %s.\n", commandStr);

        if (msg->command == RR || msg->command == REJ)
            msg->nr = (controlField >> 7) & BIT(0);

        if (msg->command == REJ)
            ll->stats->numReceivedREJ++;
        else if (msg->command == RR)
            ll->stats->numReceivedRR++;
    } else if (msg->type == DATA) {
        msg->data.messageSize = size - MESSAGE_SIZE;

        unsigned char calcBCC2 = processBCC(&message[4], msg-
>data.messageSize);
        unsigned char BCC2 = message[4 + msg->data.messageSize];

        if (calcBCC2 != BCC2) {
            printf("ERROR: invalid BCC2: 0x%02x != 0x%02x.\n",
calcBCC2, BCC2);
```

```
            free(message);

            msg->type = INVALID;
            msg->error = BCC2_ERROR;

            return msg;
        }

        msg->ns = (message[2] >> 6) & BIT(0);

        // copy message
        msg->data.message = malloc(msg->data.messageSize);
        memcpy(msg->data.message, &message[4], msg->data.messageSize);

        ll->stats->receivedMessages++;
    }

    free(message);

    return msg;
}

int messageIsCommand(Message* msg, Command command) {
    return msg->type == COMMAND && msg->command == command;
}

ui stuff(unsigned char** buf, ui bufSize) {
    ui newBufSize = bufSize;

    int i;
    for (i = 1; i < bufSize - 1; i++)
        if ((*buf)[i] == FLAG || (*buf)[i] == ESCAPE)
            newBufSize++;

    *buf = (unsigned char*) realloc(*buf, newBufSize);

    for (i = 1; i < bufSize - 1; i++) {
        if ((*buf)[i] == FLAG || (*buf)[i] == ESCAPE) {
            memmove(*buf + i + 1, *buf + i, bufSize - i);

            bufSize++;

            (*buf)[i] = ESCAPE;
            (*buf)[i + 1] ^= 0x20;
        }
    }

    return newBufSize;
}

ui destuff(unsigned char** buf, ui bufSize) {
    int i;
    for (i = 1; i < bufSize - 1; ++i) {
        if ((*buf)[i] == ESCAPE) {
            memmove(*buf + i, *buf + i + 1, bufSize - i - 1);

            bufSize--;

            (*buf)[i] ^= 0x20;
        }
```

```
    }

    *buf = (unsigned char*) realloc(*buf, bufSize);

    return bufSize;
}

unsigned char processBCC(const unsigned char* buf, ui size) {
    unsigned char BCC = 0;

    ui i = 0;
    for (; i < size; i++)
        BCC ^= buf[i];

    return BCC;
}

void cleanBuf(unsigned char* buf, ui bufSize) {
    memset(buf, 0, bufSize * sizeof(*buf));
}

void printBuf(unsigned char* buf) {
    printf("-------------------------\n");
    printf("- FLAG: 0x%02x\t\t-\n", buf[0]);
    printf("- A: 0x%02x\t\t-\n", buf[1]);
    printf("- C: 0x%02x\t\t-\n", buf[2]);
    printf("- BCC: 0x%02x = 0x%02x\t-\n", buf[3], buf[1] ^ buf[2]);
    printf("- FLAG: 0x%02x\t\t-\n", buf[4]);
    printf("-------------------------\n");
}
```

DataLink.h

```
#pragma once

#include <termios.h>
#include "ConnectionMode.h"
#include "Utilities.h"

typedef enum {
    START, FLAG_RCV, A_RCV, C_RCV, BCC_OK, STOP
} State;

typedef enum {
    SET, UA, RR, REJ, DISC
} Command;

typedef enum {
    C_SET = 0x03, C_UA = 0x07, C_RR = 0x05, C_REJ = 0x01, C_DISC =
0x0B
} ControlField;

typedef enum {
    COMMAND, DATA, INVALID
} MessageType;

typedef enum {
    INPUT_OUTPUT_ERROR, BCC1_ERROR, BCC2_ERROR
} MessageError;
```

```
typedef enum {
    COMMAND_SIZE = 5 * sizeof(char), MESSAGE_SIZE = 6 * sizeof(char)
} MessageSize;

typedef struct {
    MessageType type;

    int ns, nr;

    Command command;

    struct {
        unsigned char* message;
        ui messageSize;
    } data;

    MessageError error;
} Message;

typedef struct {
    int sentMessages;
    int receivedMessages;

    int timeouts;

    int numSentRR;
    int numReceivedRR;

    int numSentREJ;
    int numReceivedREJ;
} Statistics;

typedef struct {
    // port /dev/ttySx
    char port[20];

    // connection mode
    ConnnectionMode mode;

    // transmission speed
    int baudRate;

    int messageDataMaxSize;

    // frame sequence number (0, 1)
    ui ns;

    // timeout value
    ui timeout;

    // number of retries in case of failure
    ui numTries;

    // trama
    char frame[MAX_SIZE];

    // old and new termios
    struct termios oldtio, newtio;

    Statistics* stats;
```

```
} LinkLayer;

extern LinkLayer* ll;

int getBaudrate(int baudrate);

int initLinkLayer(const char* port, ConnnectionMode mode, int
baudrate,
        int messageDataMaxSize, int timeout, int numRetries);
Statistics* initStatistics();

int saveCurrentPortSettingsAndSetNewTermios();
int saveCurrentTermiosSettings();
int setNewTermios();

int openSerialPort(const char* port);
int closeSerialPort();

int llopen(ConnnectionMode mode);
int llwrite(int fd, const unsigned char* buf, ui bufSize);
int llread(int fd, unsigned char** message);
int llclose(int fd, ConnnectionMode mode);

unsigned char* createCommand(ControlField C);
int sendCommand(int fd, Command command);
Command getCommandWithControlField(ControlField controlField);
ControlField getCommandControlField(char* commandStr, Command
command);

unsigned char* createMessage(const unsigned char* buf, ui bufSize);
int sendMessage(int fd, const unsigned char* buf, ui bufSize);

Message* receiveMessage(int fd);
int messageIsCommand(Message* msg, Command command);

ui stuff(unsigned char** buf, ui bufSize);
ui destuff(unsigned char** buf, ui bufSize);

unsigned char processBCC(const unsigned char* buf, ui size);

void cleanBuf(unsigned char* buf, ui bufSize);
void printBuf(unsigned char* buf);
```

Main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "Aplication.h"
#include "CLI.h"
#include "Utilities.h"

const int DEFAULT_BAUDRATE = B38400;
const int DEFAULT_MESSAGE_DATA_MAX_SIZE = 512;
const int DEFAULT_NUM_RETRIES = 3;
const int DEFAULT_TIMEOUT = 3;

static void printUsage(char* argv0);
```

```
static int procArgs(int argc, char** argv);

int main(int argc, char** argv) {
    if (argc != 1 && argc != 5) {
        printf("ERROR: Wrong number of arguments.\n");
        printUsage(argv[0]);
        return 1;
    }

    if (argc == 1)
        startCLI();
    else if (argc == 5)
        procArgs(argc, argv);

    return 0;
}

static void printUsage(char* argv0) {
    printf("Usage: %s\n", argv0);
    printf("       %s <send|receive> <port> <file> <debugging>\n",
argv0);
}

static int procArgs(int argc, char** argv) {
    ConnnectionMode mode;
    char *port, *file, *debug;

    if (strncmp(argv[1], "send", strlen("send")) == 0)
        mode = SEND;
    else if (strncmp(argv[1], "receive", strlen("receive")) == 0)
        mode = RECEIVE;
    else {
        printf(
                "ERROR: Neither send nor receive specified. Did you
misspell something?\n");
        return -1;
    }

    port = argv[2];
    file = argv[3];
    debug = argv[4];

    if (strcmp(debug, "on") == 0) {
        DEBUG_MODE = TRUE;
        printf("! DEBUGGING ON !\n");
    }

    initApplicationLayer(port, mode, DEFAULT_BAUDRATE,
            DEFAULT_MESSAGE_DATA_MAX_SIZE, DEFAULT_NUM_RETRIES,
DEFAULT_TIMEOUT,
            file);

    return 0;
}
```

ParameterType.h

```
#pragma once
```

```
typedef enum {
    PARAM_FILE_SIZE, PARAM_FILE_NAME
} ParameterType;
```

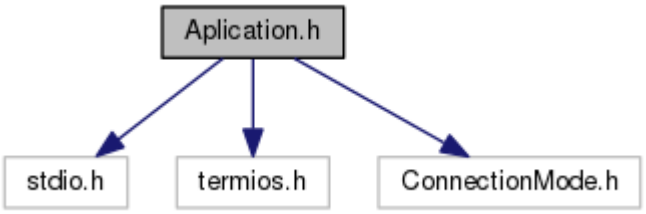
Utilities.c

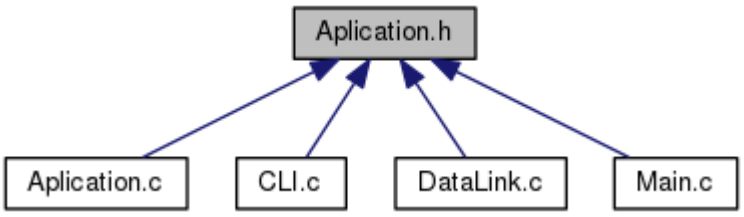
```
#include "Utilities.h"

int getFileSize(FILE* file) {
    // saving current position
    long int currentPosition = ftell(file);

    // seeking end of file
    if (fseek(file, 0, SEEK_END) == -1) {
        printf("ERROR: Could not get file size.\n");
        return -1;
    }

    // saving file size
    long int size = ftell(file);

    // seeking to the previously saved position
    fseek(file, 0, currentPosition);

    // returning size
    return size;
}
```

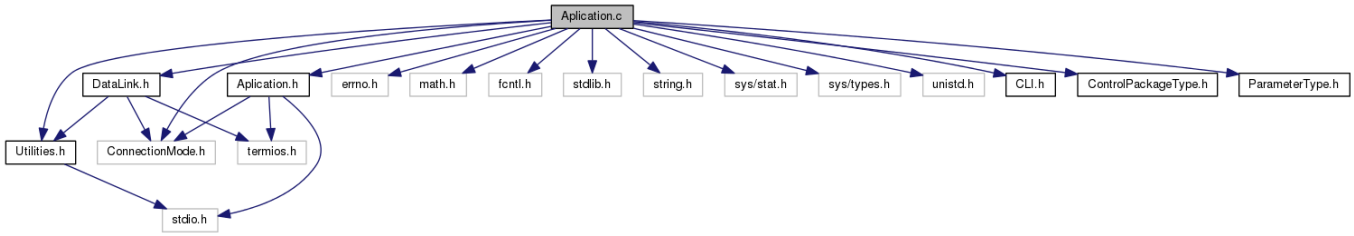
Utilities.h

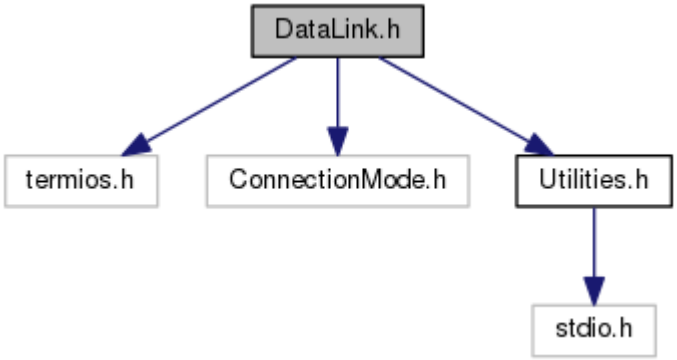
```
#pragma once

#include <stdio.h>

typedef unsigned int ui;

#define FALSE 0
#define TRUE !FALSE

#define MAX_SIZE 256
#define BIT(n) (0x01 << n)

int getFileSize(FILE* file);
```

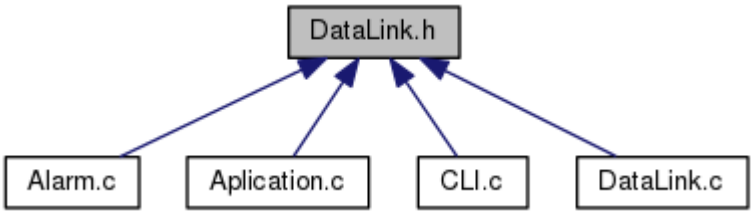
Includes

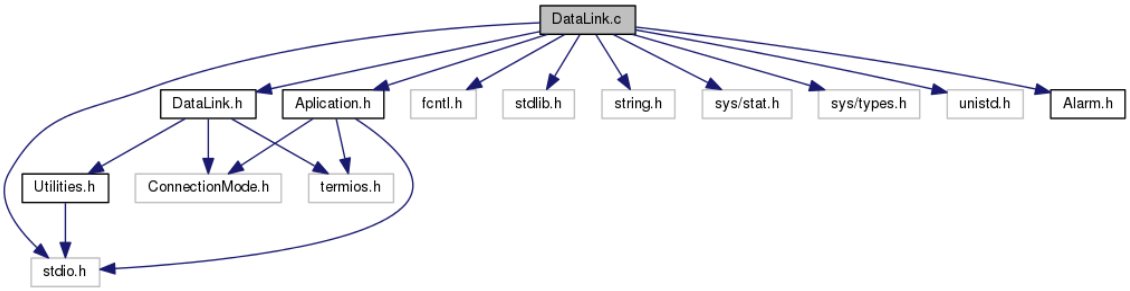

Aplication.h
stdio.h
termios.h
ConnectionMode.h

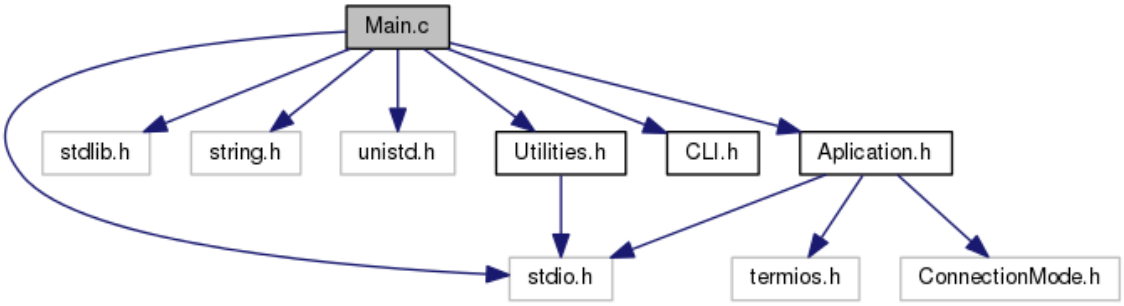

Aplication.h
Aplication.c
CLI.c
DataLink.c
Main.c

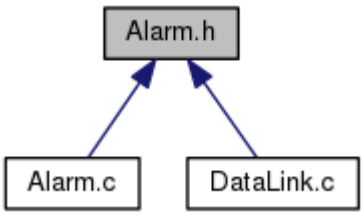

Aplication.c
DataLink.h
Aplication.h
errno.h
math.h
fcntl.h
stdlib.h
string.h
sys/stat.h
sys/types.h
unistd.h
CLI.h
ControlPackageType.h
ParameterType.h
Utilities.h
ConnectionMode.h
termios.h
stdio.h

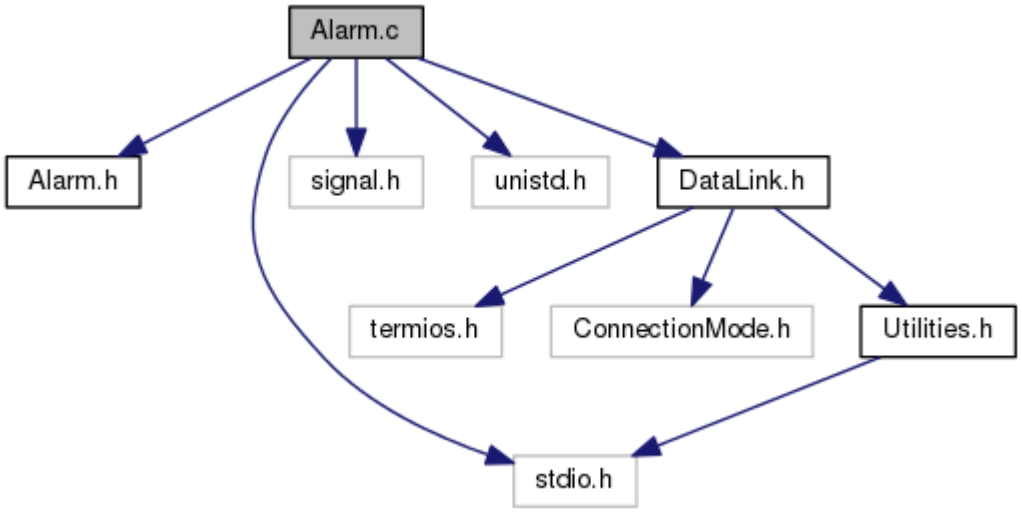

DataLink.h
termios.h
ConnectionMode.h
Utilities.h
stdio.h

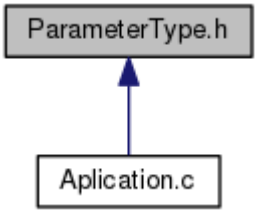

DataLink.h
Alarm.c
Aplication.c
CLI.c
DataLink.c

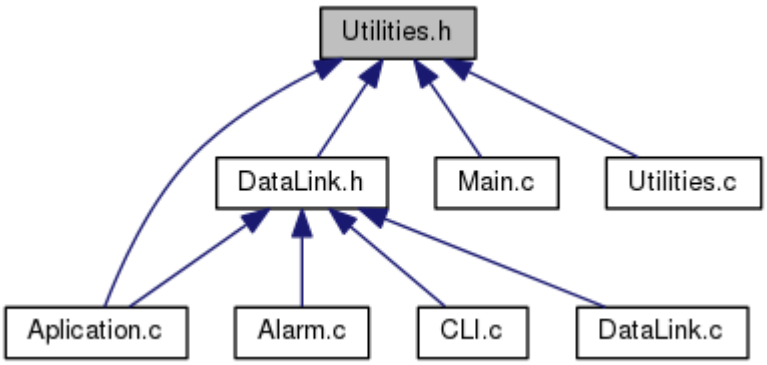

DataLink.c
DataLink.h
Aplication.h
fcntl.h
stdlib.h
string.h
sys/stat.h
sys/types.h
unistd.h
Alarm.h
Utilities.h
ConnectionMode.h
termios.h
stdio.h

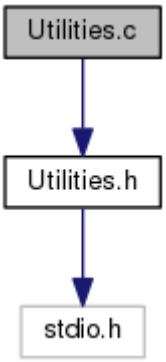

Main.c
stdlib.h
string.h
unistd.h
Utilities.h
CLI.h
Aplication.h
stdio.h
termios.h
ConnectionMode.h

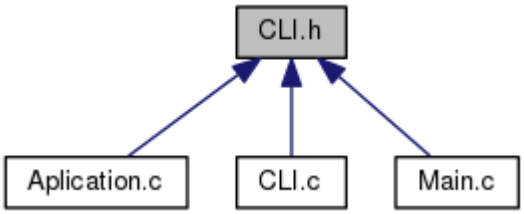

Alarm.h
Alarm.c
DataLink.c

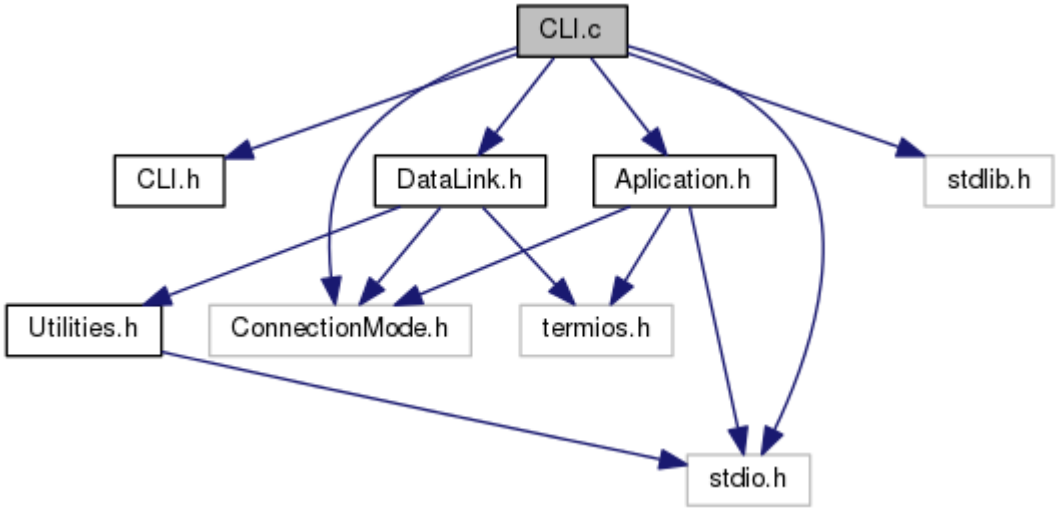

Alarm.c
Alarm.h
signal.h
unistd.h
DataLink.h
termios.h
ConnectionMode.h
Utilities.h
stdio.h


ParameterType.h
Aplication.c

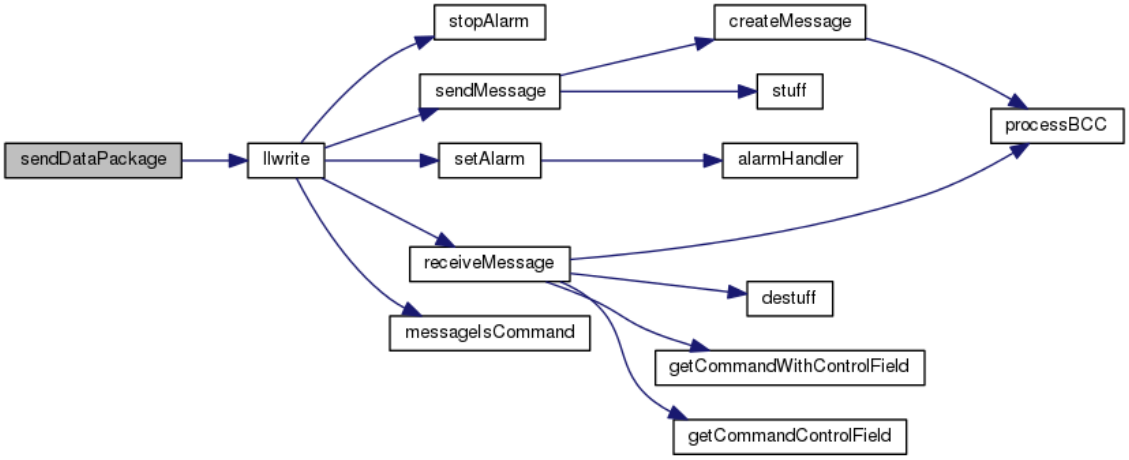

Utilities.h
DataLink.h
Main.c
Utilities.c
Aplication.c
Alarm.c
CLI.c
DataLink.c

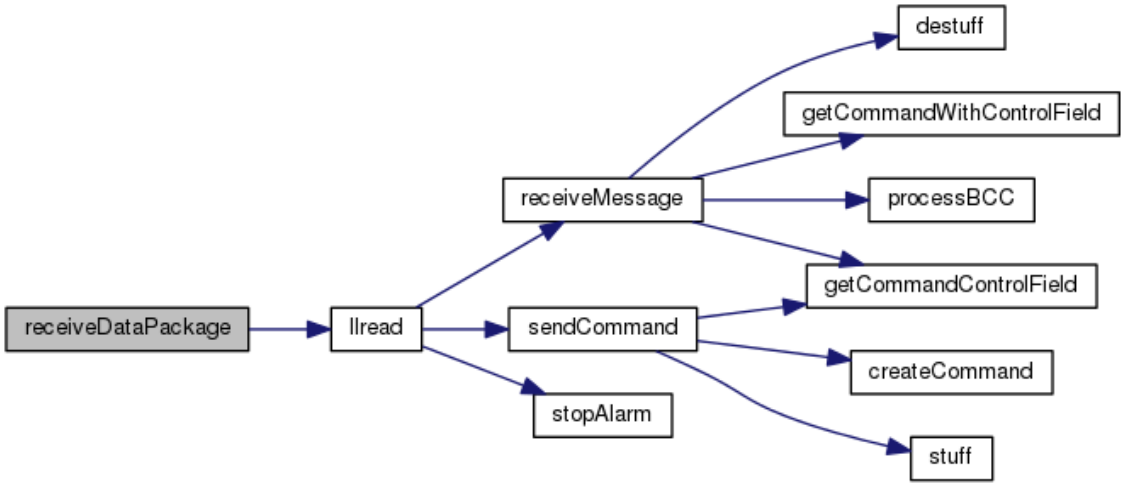

Utilities.c
Utilities.h
stdio.h

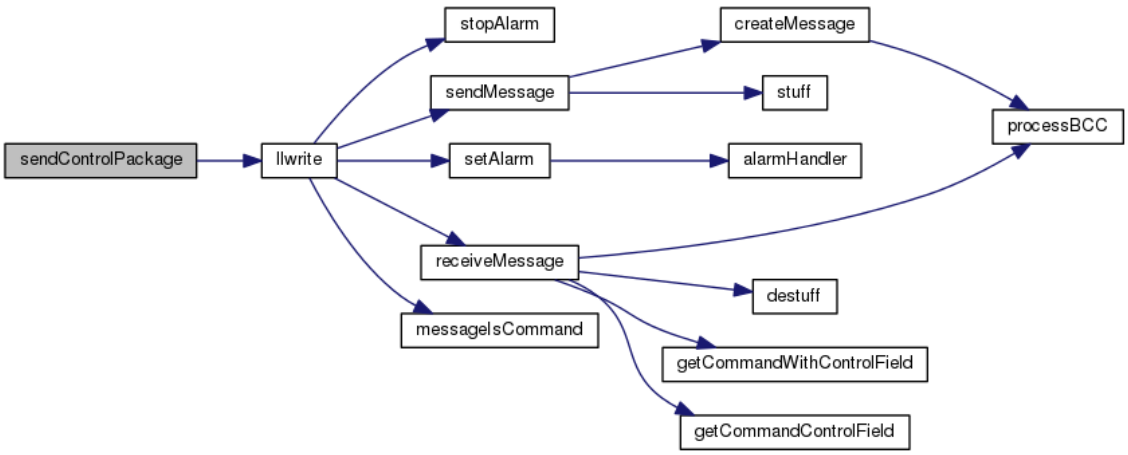

CLI.h
Aplication.c
CLI.c
Main.c

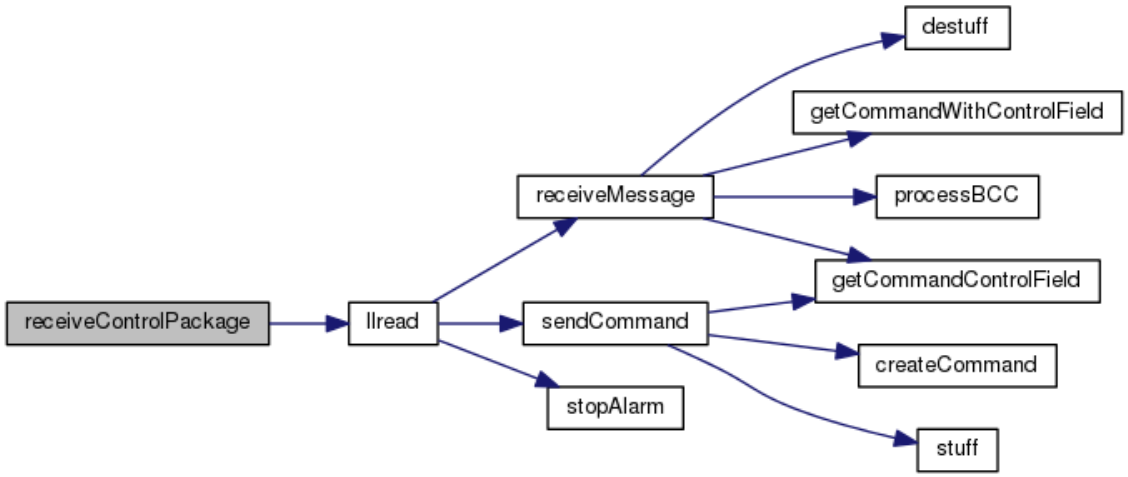

CLI.c
CLI.h
DataLink.h
Aplication.h
stdlib.h
Utilities.h
ConnectionMode.h
termios.h
stdio.h

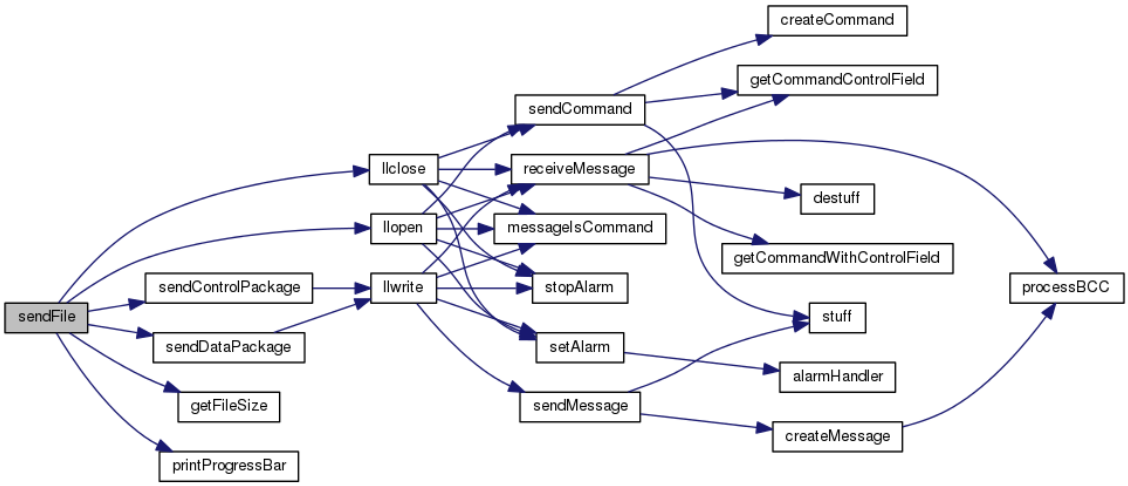
Sequência de chamadas de funções

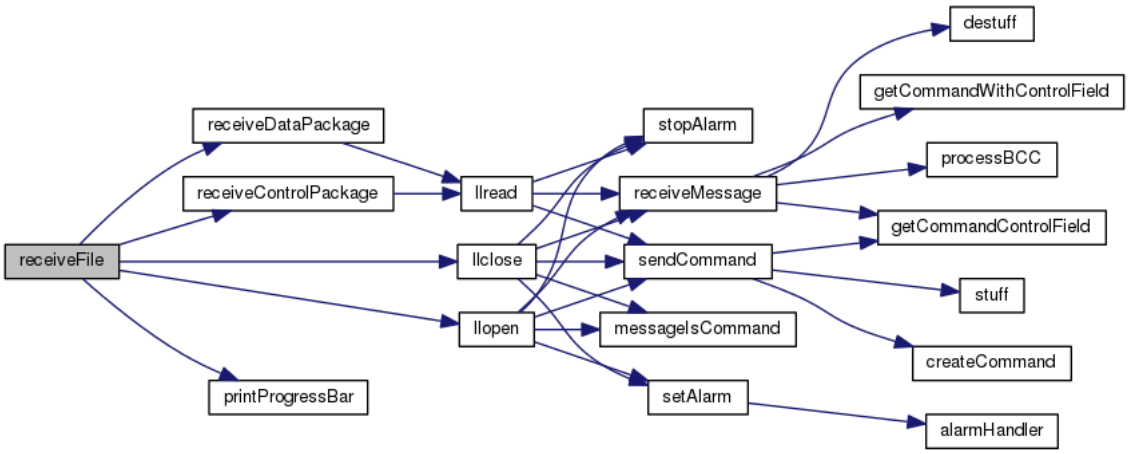


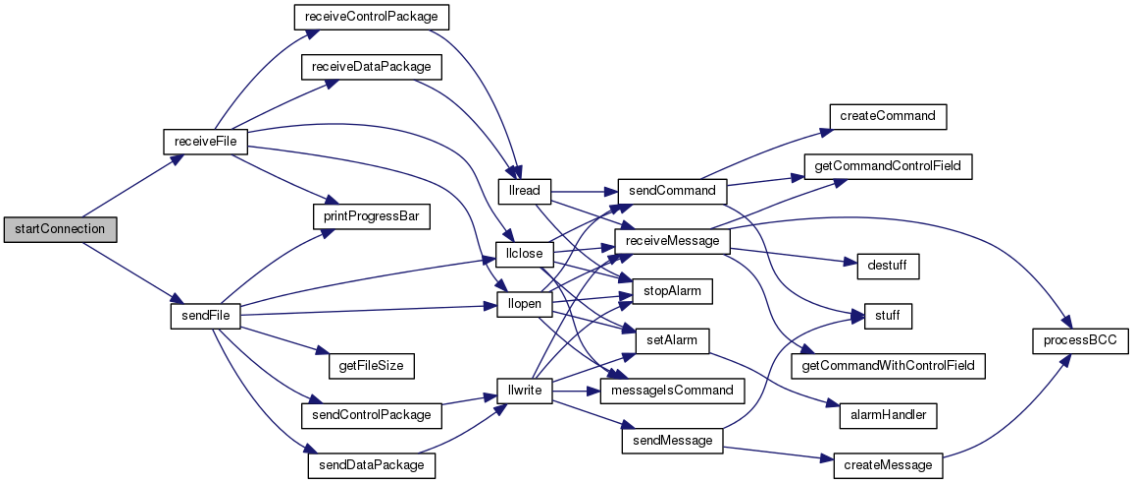


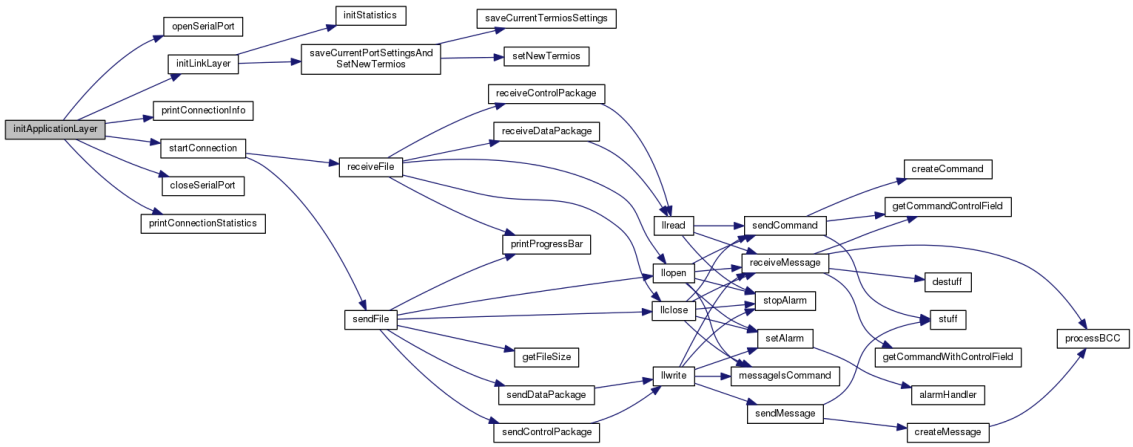


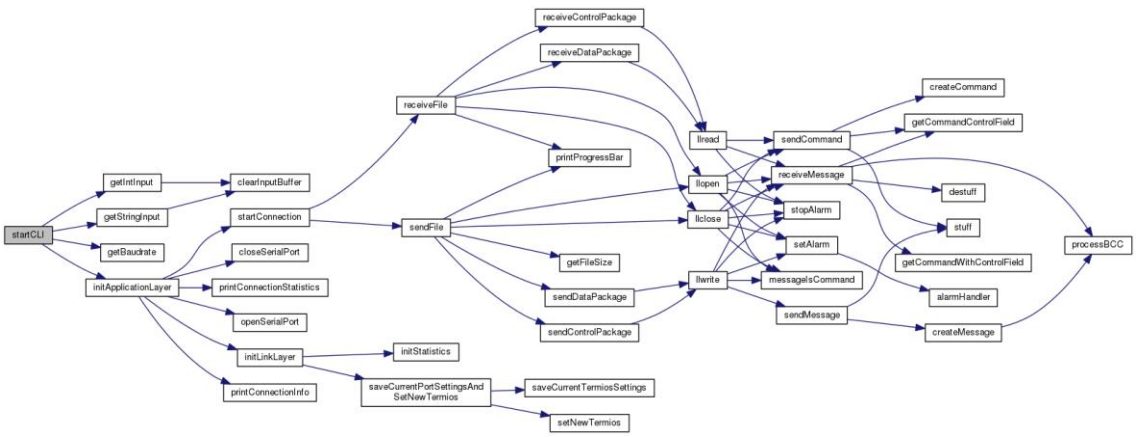

destuff
getCommandWithControlField
receiveMessage
processBCC
getCommandControlField
receiveControlPackage
llread
sendCommand
createCommand
stopAlarm
stuff

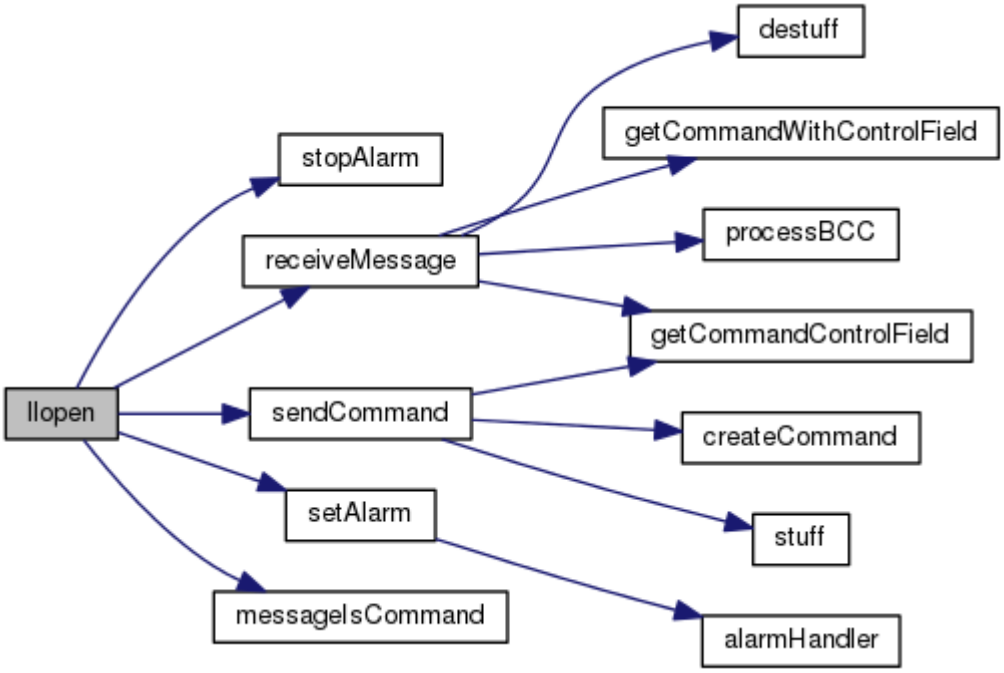

createCommand
getCommandControlField
sendCommand
llclose
receiveMessage
destuff
llopen
messageIsCommand
getCommandWithControlField
sendControlPackage
llwrite
stopAlarm
processBCC
sendFile
stuff
sendDataPackage
setAlarm
alarmHandler
getFileSize
sendMessage
createMessage
printProgressBar

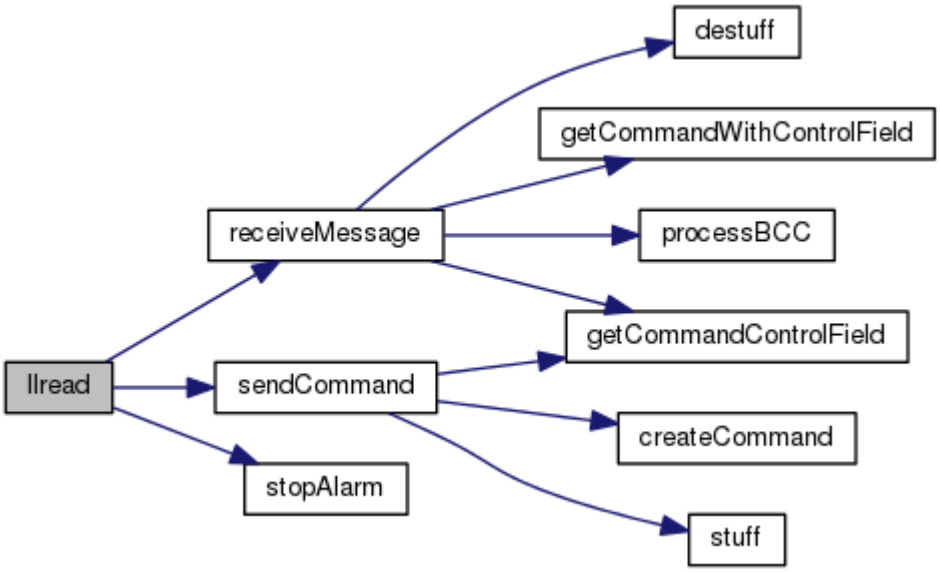

destuff
getCommandWithControlField
receiveDataPackage
stopAlarm
processBCC
receiveControlPackage
llread
receiveMessage
getCommandControlField
receiveFile
llclose
sendCommand
stuff
llopen
messageIsCommand
createCommand
printProgressBar
setAlarm
alarmHandler

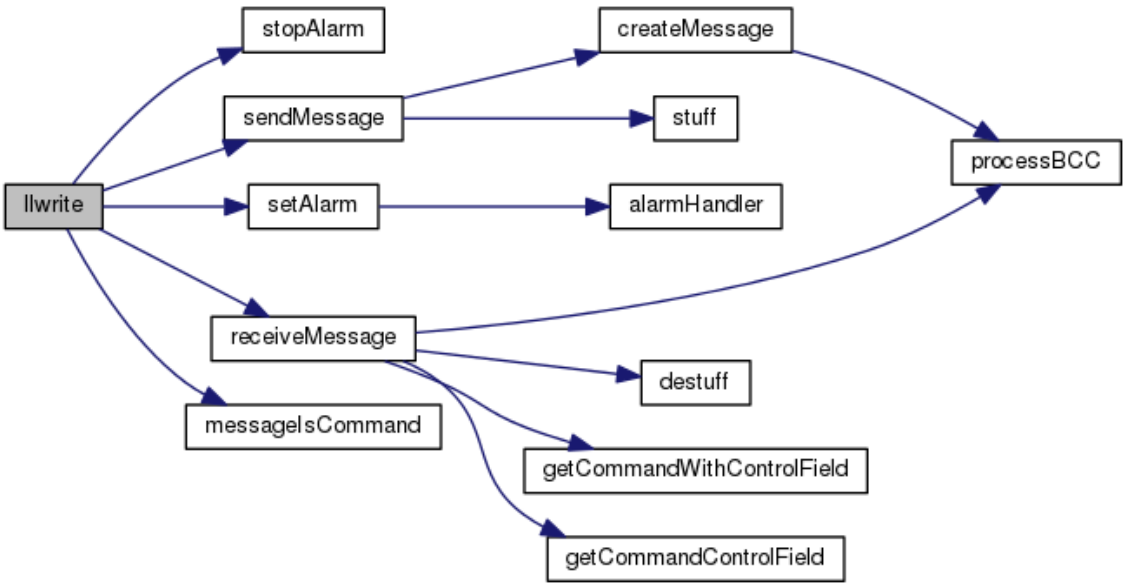

receiveControlPackage
receiveDataPackage
receiveFile
createCommand
getCommandControlField
llread
sendCommand
printProgressBar
startConnection
receiveMessage
destuff
llclose
stopAlarm
llopen
stuff
sendFile
setAlarm
processBCC
getFileSize
getCommandWithControlField
llwrite
messageIsCommand
sendControlPackage
alarmHandler
sendMessage
sendDataPackage
createMessage

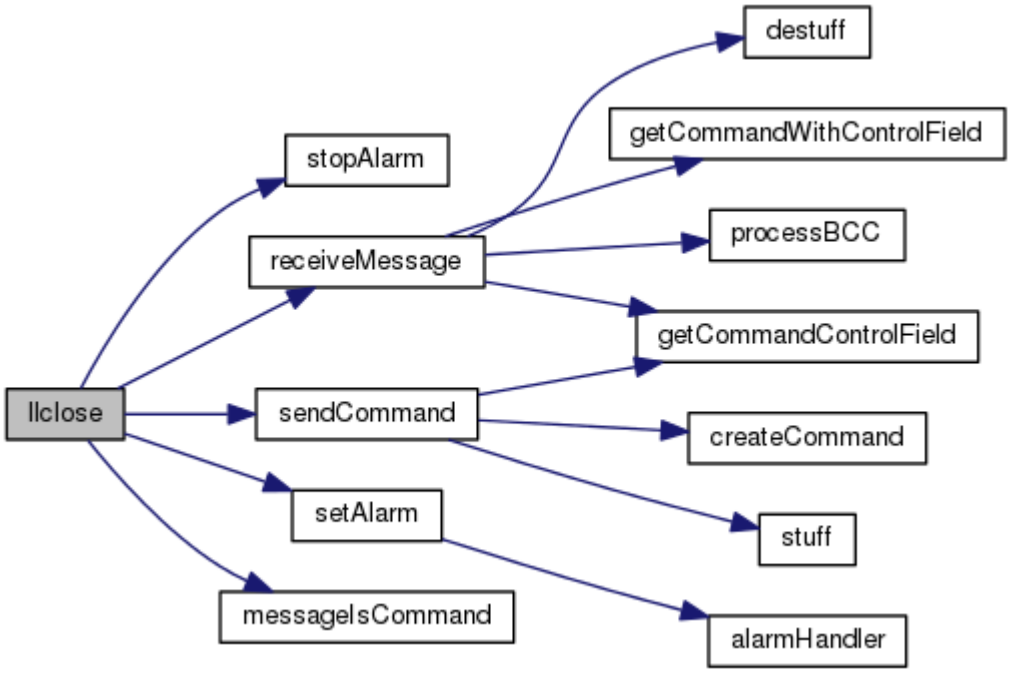

initStatistics
openSerialPort
saveCurrentTermiosSettings
initLinkLayer
saveCurrentPortSettingsAnd SetNewTermios
setNewTermios
printConnectionInfo
initApplicationLayer
receiveControlPackage
startConnection
receiveDataPackage
receiveFile
closeSerialPort
createCommand
printConnectionStatistics
getCommandControlField
llread
sendCommand
printProgressBar
receiveMessage
llopen
destuff
stopAlarm
llclose
stuff
sendFile
setAlarm
processBCC
getFileSize
getCommandWithControlField
llwrite
messageIsCommand
sendDataPackage
alarmHandler
sendMessage
sendControlPackage
createMessage

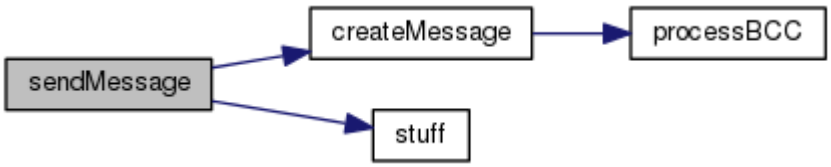

receiveControlPackage
receiveDataPackage
receiveFile
createCommand
getCommandControlField
getIntInput
clearInputBuffer
llread
sendCommand
printProgressBar
getStringInput
startCLI
startConnection
receiveMessage
llopen
destuff
getBaudrate
closeSerialPort
stopAlarm
llclose
stuff
sendFile
initApplicationLayer
printConnectionStatistics
setAlarm
processBCC
getFileSize
getCommandWithControlField
openSerialPort
llwrite
messageIsCommand
sendDataPackage
alarmHandler
initLinkLayer
initStatistics
sendMessage
sendControlPackage
createMessage
printConnectionInfo
saveCurrentPortSettingsAnd SetNewTermios
saveCurrentTermiosSettings
setNewTermios

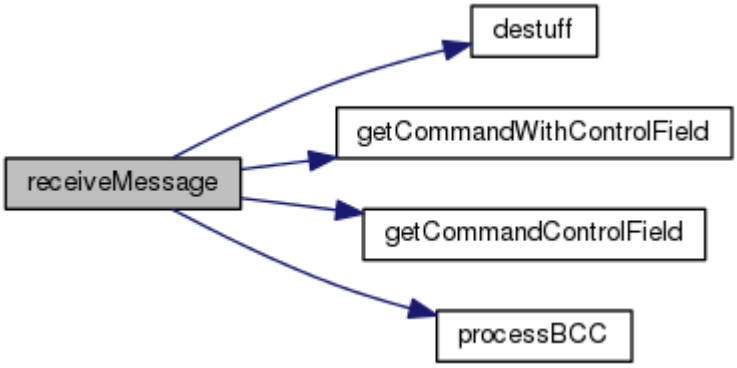

destuff
getCommandWithControlField
stopAlarm
processBCC
receiveMessage
getCommandControlField
llopen
sendCommand
createCommand
setAlarm
stuff
messageIsCommand
alarmHandler


destuff
getCommandWithControlField
receiveMessage
processBCC
getCommandControlField
llread
sendCommand
createCommand
stopAlarm
stuff

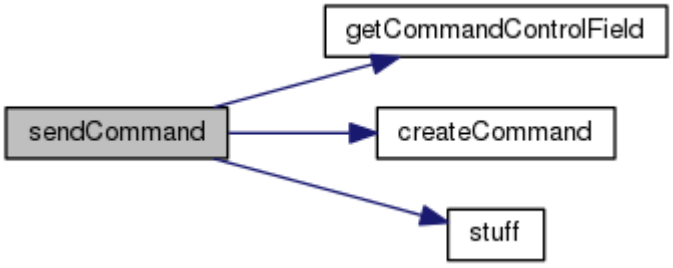

stopAlarm
createMessage
sendMessage
stuff
processBCC
llwrite
setAlarm
alarmHandler
receiveMessage
destuff
messageIsCommand
getCommandWithControlField
getCommandControlField


destuff
getCommandWithControlField
stopAlarm
processBCC
receiveMessage
getCommandControlField
llclose
sendCommand
createCommand
setAlarm
stuff
messageIsCommand
alarmHandler


sendMessage
createMessage
processBCC
stuff


receiveMessage
destuff
getCommandWithControlField
getCommandControlField
processBCC

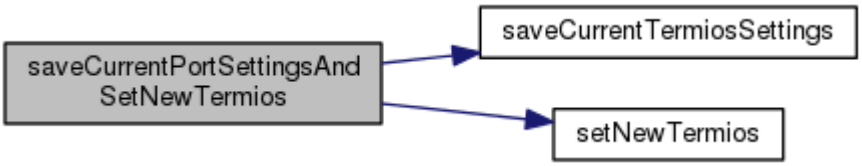

createMessage
processBCC

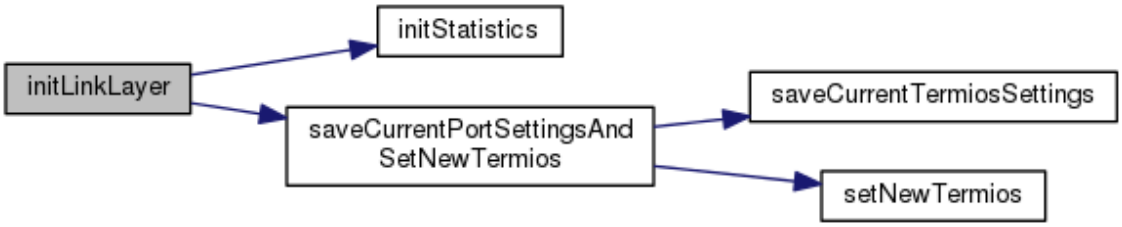

sendCommand
getCommandControlField
createCommand
stuff


setAlarm
alarmHandler

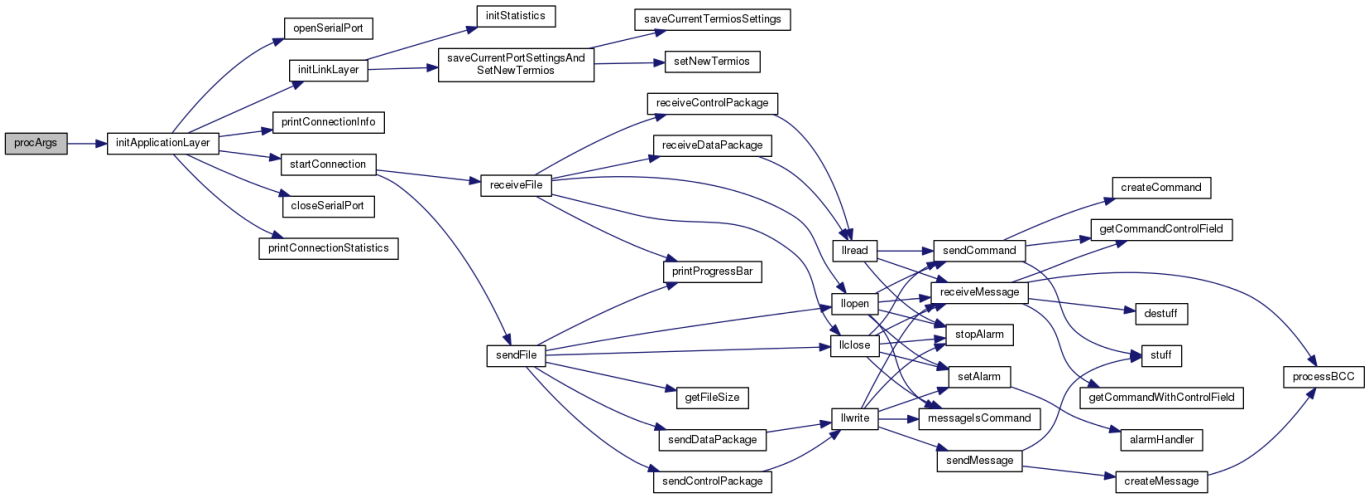

getIntInput
clearInputBuffer

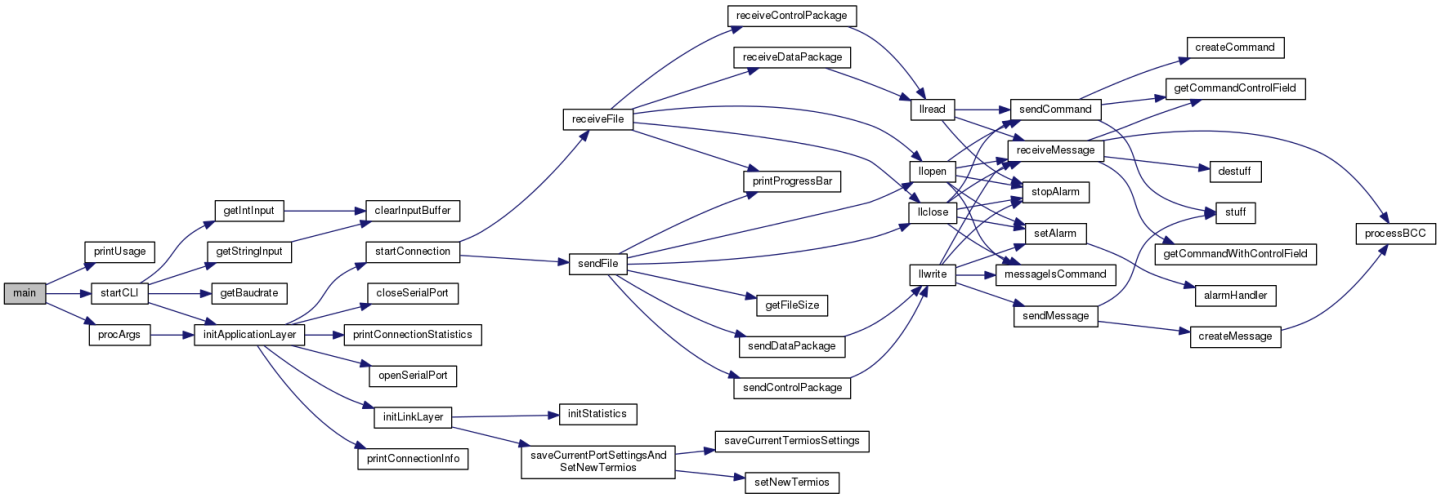

getStringInput
clearInputBuffer


saveCurrentPortSettingsAnd
SetNewTermios
saveCurrentTermiosSettings
setNewTermios


initLinkLayer
initStatistics
saveCurrentPortSettingsAnd
SetNewTermios
saveCurrentTermiosSettings
setNewTermios


Statistics
stats
LinkLayer


procArgs
initApplicationLayer
openSerialPort
initLinkLayer
initStatistics
saveCurrentPortSettingsAnd SetNewTermios
saveCurrentTermiosSettings
setNewTermios
printConnectionInfo
startConnection
closeSerialPort
printConnectionStatistics
receiveFile
receiveControlPackage
receiveDataPackage
printProgressBar
sendFile
getFileSize
sendDataPackage
sendControlPackage
llopen
llclose
llread
llwrite
sendCommand
receiveMessage
stopAlarm
setAlarm
messageIsCommand
sendMessage
createCommand
getCommandControlField
destuff
stuff
getCommandWithControlField
alarmHandler
createMessage
processBCC


main
printUsage
startCLI
procArgs
getIntInput
getStringInput
getBaudrate
clearInputBuffer
startConnection
closeSerialPort
initApplicationLayer
printConnectionStatistics
openSerialPort
initLinkLayer
printConnectionInfo
initStatistics
saveCurrentPortSettingsAnd SetNewTermios
saveCurrentTermiosSettings
setNewTermios
receiveFile
receiveControlPackage
receiveDataPackage
printProgressBar
sendFile
getFileSize
sendDataPackage
sendControlPackage
llopen
llclose
llread
llwrite
sendCommand
receiveMessage
stopAlarm
setAlarm
messageIsCommand
sendMessage
createCommand
getCommandControlField
destuff
stuff
getCommandWithControlField
alarmHandler
createMessage
processBCC