

Redes de Computadores

**PROTOCOLO DE LIGAÇÃO DE DADOS**

*1º Trabalho Laboratorial*

Diogo Santos (up201806878)

Rita Mota (up201703964)

*10 de outubro 2020*

# **Sumário**

Este relatório foi realizado no âmbito da unidade curricular Redes de Computadores e trata-se de um complemento ao primeiro trabalho prático, cujo foco foi a implementação de um protocolo de comunicação entre dois computadores através de uma porta de série.

O programa funcionou como esperado, tendo sido realizada a transferência do ficheiro teste sem quaisquer erros ou perdas de dados, mesmo com introdução de ruído ou com interrupções da ligação.

# **Introdução**

Com este trabalho pretendemos alcançar dois objetivos principais: implementar um protocolo de ligação de dados utilizando o método Stop&Wait e que permite fazer a transferência de dados entre computadores de forma assíncrona e testar o protocolo desenvolvido através da utilização de uma aplicação que permite a transferência de ficheiros.

O objetivo deste relatório é especificar a forma como desenvolvemos este protocolo e terá a seguinte estrutura:

* **Arquitetura** - Funcionamento da aplicação e interfaces;
* **Estrutura do código** - Especificação das APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura;
* **Casos de uso principais** - Identificação dos casos de uso e a sequência de chamadas a funções.
* **Protocolo de ligação lógica** - Identificação dos principais aspetos funcionais e descrição da sua estratégia de implementação.
* **Protocolo de aplicação** - Identificação dos principais aspetos funcionais e ainda a descrição da sua estratégia de implementação.
* **Validação** - descrição dos testes efetuados;
* **Eficiência do protocolo de ligação de dados** - Caracterização estatística da eficiência do protocolo;
* **Conclusão** - Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados;

# **Arquitetura**

O trabalho tem duas entidades independentes (emissor e receptor) que utilizam funções da camada de ligação de dados e da camada de aplicação, havendo então uma distinção entre as chamadas feitas uma vez que ambos desempenham papéis diferentes durante a comunicação.

# **Estrutura do Código**

O nosso código encontra-se dividido em três ficheiros: o ficheiro writenoncanonical.c onde se encontram todas as funções necessárias para o emissor, o ficheiro noncanonical.c, onde se encontram todas as funções necessárias para o receptor e o ficheiro macros.h que guarda apenas alguns valores importantes.

**writenoncanonical.c**:

Funções principais da camada de ligação de dados:

* llopen - abre a porta de série e verifica se existe comunicação através do envio de uma trama de supervisão SET e receção de uma trama de supervisão UA.
* llwrite - utiliza o mecanismo de byte stuffing para enviar tramas de informação e aguarda uma confirmação ou rejeição.
* llclose - envia uma trama de supervisão DISC e, após receber uma de volta, envia uma trama de supervisão UA e fecha a porta de série.

Funções principais da camada da aplicação:

* main - funciona como base para a camada de aplicação uma vez que faz a chamada às funções mencionadas acima.

**noncanonical.c**:

Funções principais da camada de ligação de dados:

* llopen - abre a porta de série e verifica se existe comunicação através do envio de uma trama de supervisão SET e receção de uma trama de supervisão UA.
* llread - lê a informação recebida, faz destuffing e envia uma confirmação ou rejeição.
* llclose - envia uma trama de supervisão DISC e após receber uma de volta, envia de seguida uma trama de supervisão UA e fecha a porta de série.

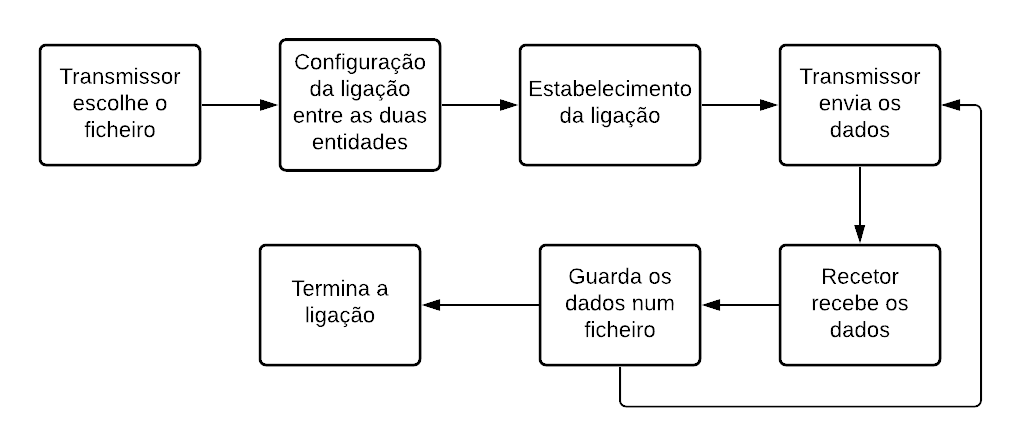
Funções principais da camada da aplicação:

* main - funciona como base para a camada de aplicação uma vez que faz a chamada às funções mencionadas acima.

# **Casos de uso principais**

Os principais casos de uso da aplicação são a interação do utilizador com a consola (interface), que lhe permite escolher a porta de série e o ficheiro a enviar e a transferência desse ficheiro através de uma porta de série.

Para poder enviar um ficheiro, o utilizador deve introduzir o número da porta de série que pretende utilizar e o caminho para o ficheiro que pretende enviar, no entanto o receptor apenas tem de introduzir o número da porta de série.

A transmissão de dados segue a seguinte sequência: 

# **Protocolo de ligação lógica**

Neste protocolo foram implementadas as funções llopen(), llwrite(), llread() e llclose(). Estas funções permitem que a camada da aplicação comunique com a camada de ligação de dados. Numa visão mais geral, o método llopen estabelece e testa a ligação entre os dois computadores, o llwrite serve de emissor do ficheiro, o llread de recetor e o llclose termina a ligação.

*Nota: Os extratos de código seguem em anexo por forma a não exceder o nº de páginas.*

##### **llopen()**

**Protótipo:** int llopen(int porta, int user);

A função llopen recebe como parâmetros um inteiro que corresponde ao número da porta série que pretendemos utilizar e um inteiro que permite a distinção entre emissor e recetor. Uma vez que dividimos o nosso trabalho por agente, cada um tem uma implementação diferente deste método, sendo que em ambas as implementações a parte inicial da função corresponde à abertura e configuração da porta de série.

###### **Emissor**

Depois de configurar a porta de série são chamadas a função connect(), que cria e envia uma trama SET para o receptor e inicia um alarme com o número de segundos estipulado em TIMEOUT, e a função waitConnect(), que espera resposta do emissor e verifica se é a correta através de uma máquina de estados.

Se o emissor receber uma trama e ela for processada na máquina de estados corretamente, a função retorna o descritor associado à porta de série.

Por outro lado, caso o receptor não responda no intervalo estipulado no alarme ou a mensagem esteja incorreta, é enviado um sinal SIGALARM (no último caso porque a máquina de estados fica em loop) e o número de tentativas falhadas é incrementado no handler timeOut().

Estes dois métodos são chamados num ciclo while cuja condição de paragem é o número de tentativas de transmissão ser inferior a MAX\_TRIES e caso o número de tentativas seja ultrapassado, a função retorna -1.

###### **Receptor**

Depois da configuração da porta de série, são invocadas a função receiveConnectionRequest(), que verifica, através de uma máquina de estados, se a mensagem recebida foi uma trama de supervisão SET, e a função sendAcknowledgment() onde é enviado para o emissor uma trama de supervisão UA, que significa que o receptor conseguiu estabelecer a ligação. Por último, e tal como no emissor, é retornado o descritor da porta série em caso de sucesso e -1 em caso de erro.

##### **llwrite()**

**Protótipo:** int llwrite(int fd, char\* buf, int length);

# 

A função llwrite() está implementada no ficheiro writenoncanonical.c e recebe como parâmetros o descritor para a porta série, o pacote de dados e o tamanho do pacote de dados.

Este método, tal como o llopen(), contém um ciclo while cuja condição de paragem é o número de tentativas ser diferente de MAX\_TRIES. Dentro desse ciclo, é invocada a função writeInfo() que calcula o BCC2 através de getBcc2() e realiza também o processo de *byte stuffing* através de uma função auxiliar doStuffing(). De seguida, transforma o pacote de dados *stuffed* numa trama de informação que envia para o receptor e, por fim, inicia um alarme com o número de segundos estipulados na macro TIMEOUT.

Depois disso, é invocada a função waitingAcknowledgmentInfo() que verifica se a mensagem recebida tem o formato correto através da utilização de uma máquina de estados.

Caso não seja recebida nenhuma mensagem durante o tempo definido em TIMEOUT, o alarme envia um sinal SIGALRM, o número de tentativas de comunicação é incrementado uma unidade através do handler timeOut e o ciclo repete-se.

Por outro lado, caso seja recebida uma mensagem, existe a possibilidade de ela ser uma rejeição (REJ) ou uma confirmação (RR). Se for um REJ, o número de tentativas não é incrementado uma vez que foi obtida uma resposta e o ciclo repete-se. Se for um RR a função retorna o comprimento do pacote de dados enviado com sucesso.

Por último, se o número de tentativas ultrapassar o valor definido em MAX\_TRIES a função retorna -1.

##### **llread()**

**Protótipo:** int llread(int fd, char\* packet);

A função llread() está implementada no ficheiro noncanonical.c e recebe como parâmetros o descritor para a porta série e um array onde vai ser guardado o pacote de dados recebido.

Inicialmente é invocada a função readInfo(), que lê uma trama de informação da porta série e faz parse da trama de informação através de uma máquina de estados. Caso a trama tenha o head e tail pretendidos, é revertido do processo de byte stuffing através da função auxiliar doDestuffing(). Na máquina de estados é também verificado, através do número de paridade, se a informação é duplicada, e, caso seja, é ativada a flag duplicated. De seguida, é efetuada a verificação do BCC2 que deverá ser igual ao XOR entre todos os bytes de dados. Se o BBC2 for diferente do esperado, é desativada a flag data\_correct. Finalmente, este método retorna o número de bytes do pacote de dados recebido.

De seguida, é invocada a função changeSequenceNumber() que muda o número de paridade que se espera receber de seguida, dependendo das variáveis data\_correct e duplicated.

Por último, é invocada a função writeAnswer() que tendo em conta os valores das variáveis referidas acima envia uma resposta para o emissor.

São enviadas mensagens de sucesso RR se:

* data\_correct for 1, o que significa que a informação está correta.
* duplicated for 1, o que significa que a informação recebida tem o mesmo número de paridade da última mensagem aceite, logo não faz sentido pedir a sua retransmissão, e os dados recebidos devem ser descartados.

Se o número de paridade da trama recebida for 0, o byte de controle da trama RR será 0x85, uma vez que varia consoante o número de paridade que espera receber a seguir. Se o número de paridade da trama recebida for 1, o byte de controle da trama RR será 0x05.

São enviadas mensagens de rejeição REJ se:

* se tanto data\_correct como duplicated forem 0, o que significa que a informação dentro da trama de informação está errada e não é duplicada, ou seja, ainda não tinha sido recebida nenhuma vez.

Na trama REJ, o byte de controlo (C) tem um número de paridade igual ao da trama recebida, ou seja C será 0x01 se a trama recebida tiver paridade 0 e 0x81 se a trama recebida tiver paridade 1.

##### **llclose()**

**Protótipo:** int llclose(int fd);

A função llclose() recebe como parâmetro o descritor para a porta série e, tal como em llopen(), também optámos por fazer uma implementação para o emissor e outra para o recetor.

###### **Emissor**

Em primeiro lugar, invocamos a função disconnect() onde é criada e enviada uma trama de supervisão DISC e é iniciado um alarme com o valor definido em TIMEOUT. De seguida, é executado o método waitDisconnect() que verifica, através de uma máquina de estados, se a mensagem recebida é uma trama DISC.

Caso receba uma trama DISC, o emissor envia uma trama UA através da função sendAcknowledgement(). No entanto, caso o receptor não responda no intervalo estipulado no alarme ou a mensagem esteja incorreta, é enviado um sinal SIGALARM (no último caso porque a máquina de estado fica em loop) e o número de tentativas falhadas é incrementado no handler timeOut().

No final, as configurações iniciais da porta série são restabelecidas e esta é fechada.

###### **Receptor**

Inicialmente é invocada a função waitingDisconnectionRequest() onde existe uma máquina de estados que faz parse do frame recebido e verifica se é uma trama de disconnect DISC. Depois disso, a função sendingDisconnectionRequest() é executada e lá é criada e enviada uma trama DISC. É ainda executada a função waitingAcknowledgment() que também possui uma máquina de estados que verifica se foi recebida uma trama UA e, por fim, é restabelecida a configuração da porta série e esta é fechada.

# **Protocolo de aplicação**

O protocolo de aplicação implementado tem como objetivos a leitura e escrita para um ficheiro, a fragmentação de um ficheiro recorrendo a números de sequência e também a criação e envio de pacotes de controlo e pacotes de dados.

*Nota: Os extratos de código seguem em anexo por forma a não exceder o nº de páginas.*

##### **Emissor**

Começa por ir buscar o tamanho do ficheiro que vai ser enviado através da função auxiliar fileSize() e envia essa informação juntamente com o nome do ficheiro nos pacotes de controlo inicial e final, que são criados no método controlPacket().

Depois de ser enviado o pacote de controlo inicial, são lidos 510 bytes do ficheiro, transformados num pacote de dados e enviados para a camada de ligação de dados através da função llwrite. Este processo é repetido até o ficheiro ter sido lido na íntegra.

##### **Receptor**

Começa por receber o pacote de controlo inicial através da função readControllPacket() lê o controll packet e inicializa as variáveis filename e filesize. Estas variáveis são utiilizadas para criar o ficheiro que vai ser recebido e para verificar se já foram recebidos todos os pacotes de dados.

De seguida, a função readDataPacket() transforma o pacote de dados recebido em dados que vão ser escritos no novo ficheiro e isto acontece até o número de bytes recebidos ser igual a filesize. Por fim, é recebido o pacote de controlo final e é processado de forma semelhante ao inicial.

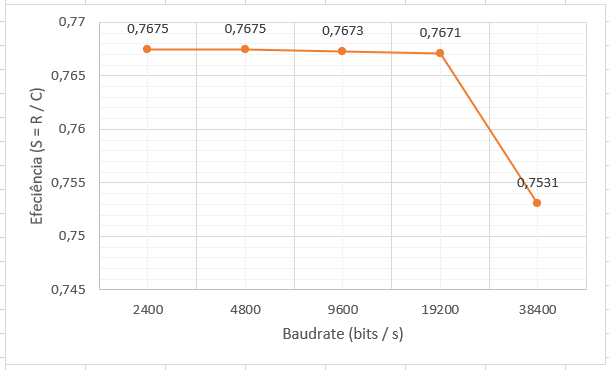
# **Validação**

Com objetivo de testar o funcionamento do nosso programa foram efetuados os seguintes testes:

* Envio de diferentes ficheiros;
* Envio de ficheiros com a desconexão da porta série a meio;
* Envio de ficheiros com introdução de “lixo” na porta série;
* Variação do comprimento da trama;
* Variação da *baudrate.*

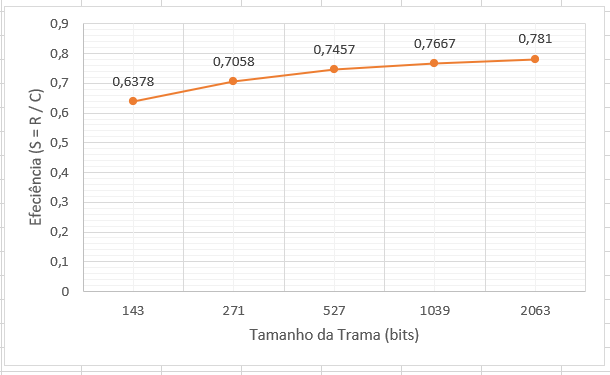
**Resultados**: Todos os testes efetuados foram concluídos com sucesso.

# **Eficiência do protocolo**

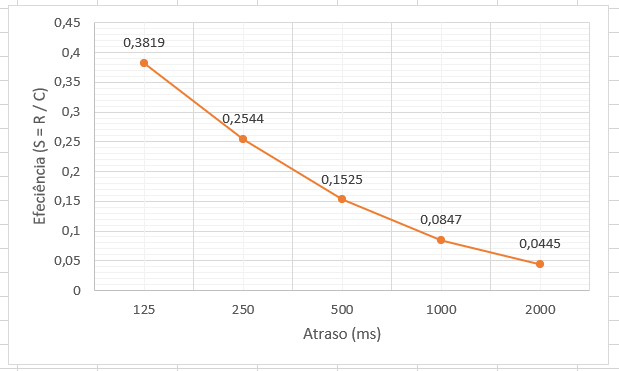


**Variação da Baudrate**

Para estes cálculos, manteve-se constante o tamanho das tramas (1035 bits). Através deste gráfico podemos concluir que quanto maior for a velocidade de transmissão, menor será a eficiência, já que muita da capacidade do cabo não será aproveitada, pois a trama tem um tamanho pequeno.

**Variação do Tamanho da Trama**

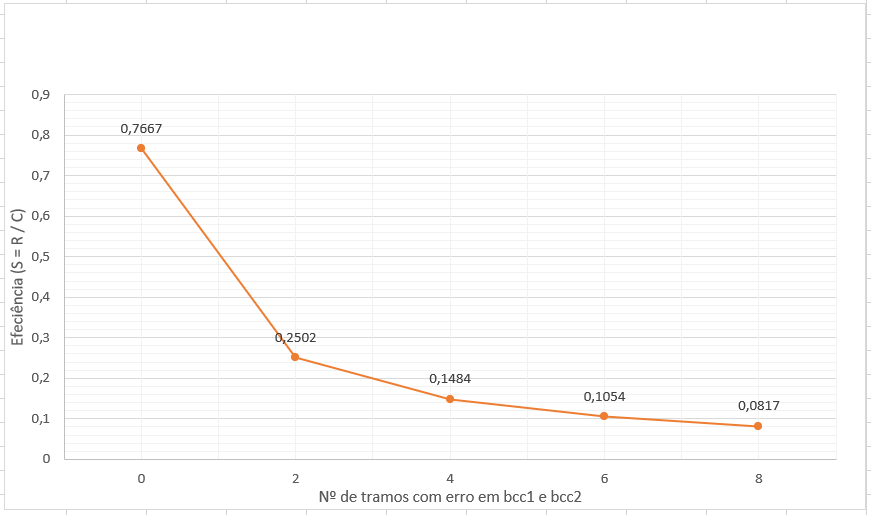
Para estes cálculos, manteve-se constante a baudrate (38400 bits/s). Através deste gráfico, é possível verificar que quanto maior for o comprimento da trama maior será a eficiência, uma vez que a capacidade do cabo será melhor aproveitada, sendo enviada mais informação de cada vez.



**Variação do tempo de propagação**

Para estes cálculos, manteve-se constante o baudrate (38400 bits/s) e o tamanho das tramas (1035 bits). Como esperado quanto maior o atraso de propagação induzido, menor será a taxa de transferência de bits/s, e como tal menor será a eficiência do protocolo.

**Variação do FER**



Manteve-se novamente constante o baudrate (38400 bits/s) e o tamanho das tramas (1035 bits).

Para a criação dos erros fez-se ocorrer um timeout e o envio de um REJ, só depois sendo enviada a mensagem corretamente.

Como era de esperar quanto maior for o FER, ou seja, quanto maior for a quantidade de erros na transmissão da mensagem, menor será a eficiência, pois terá de se reenviar a informação “perdendo” tempo de transferência.

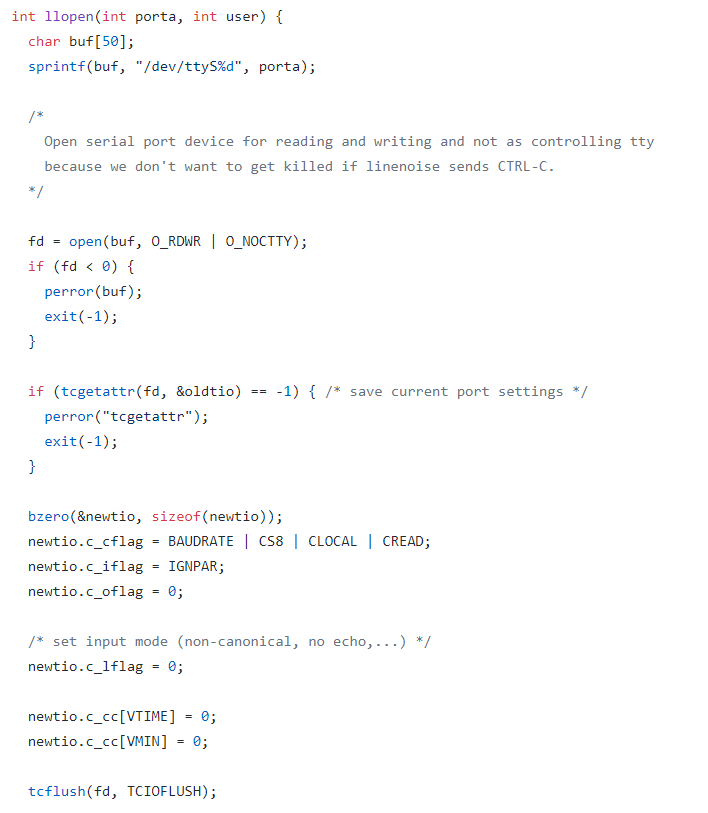
# **Conclusões**

A realização deste trabalho permitiu-nos compreender a importância dos mecanismos de *byte stuffing* e *Stop & Wait,* bem como a necessidade de independência entre camadas num protocolo de comunicação.

Relativamente ao protocolo implementado, pensamos que cumpre o esperado uma vez que conseguimos efetuar a transferência de várias imagens e ficheiros de texto sem qualquer tipo de erro e garantindo sempre a independência entre a camada da ligação de dados e a camada da aplicação.

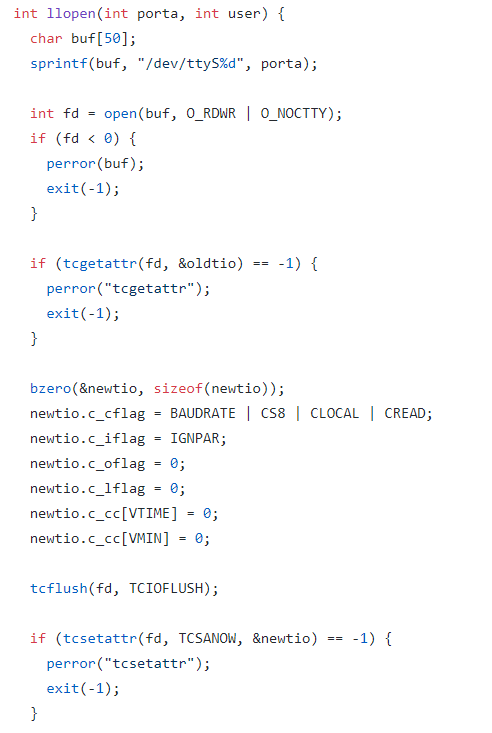
**Anexos**

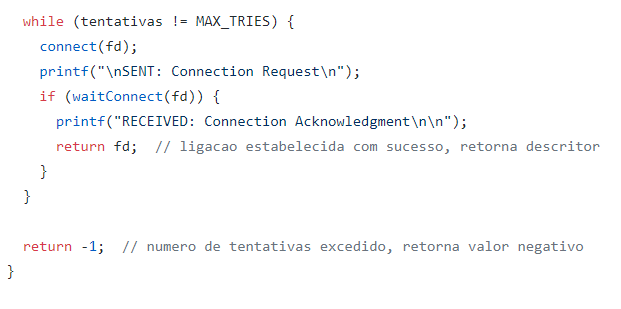
* **llopen(receptor)**

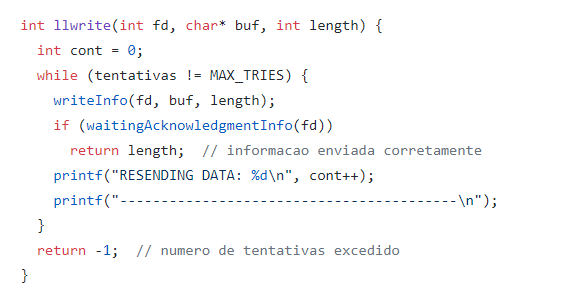


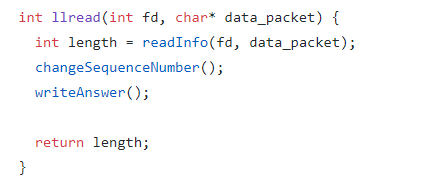


* **llopen(emissor)**





* **llwrite**
* **llread**

****

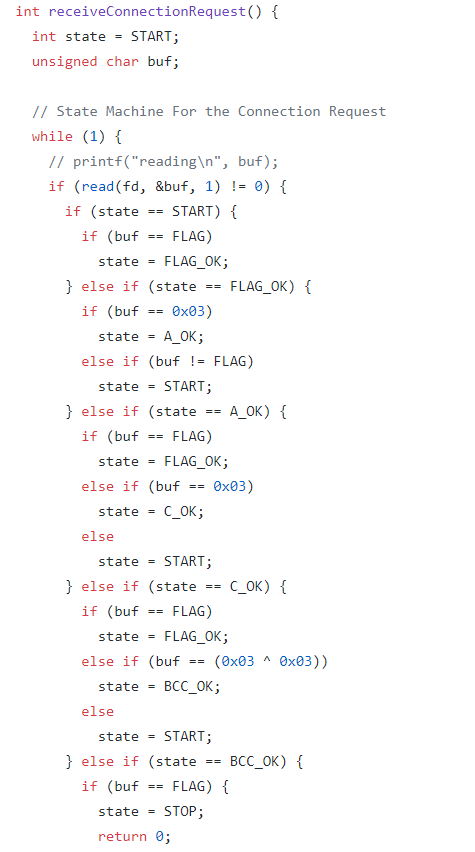
* **llclose(emissor)**

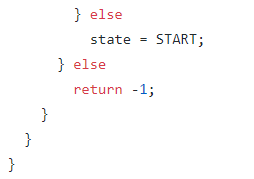


* **llclose(receptor)**

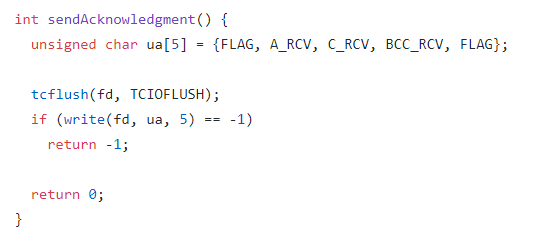


* **receiveConnectionrequest**

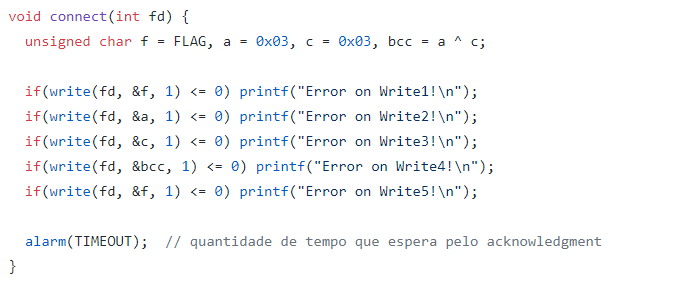


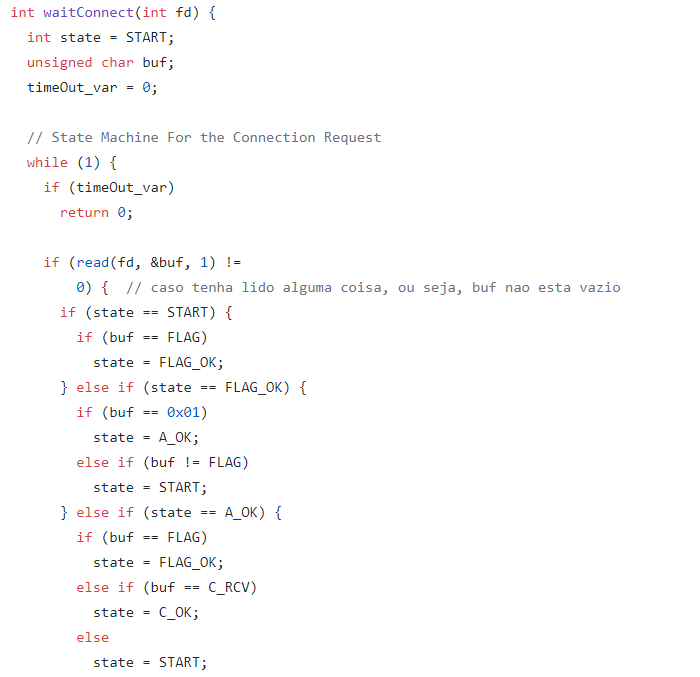
****

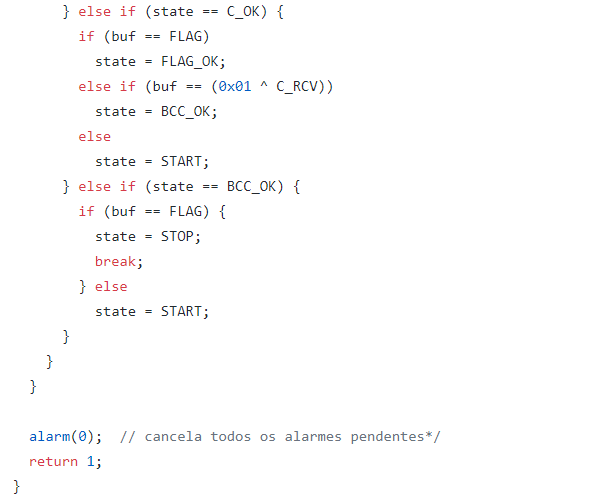
* **sendAcknowledgment**

****

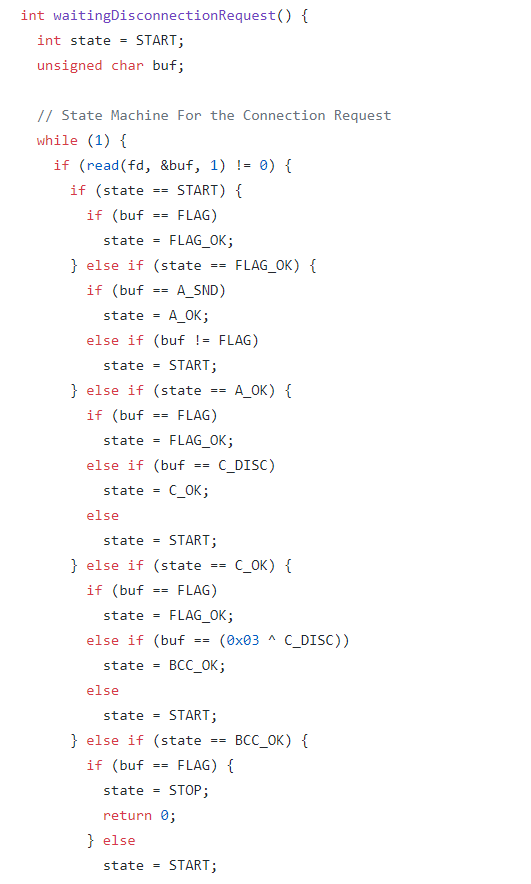
* **connect**

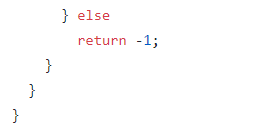
****

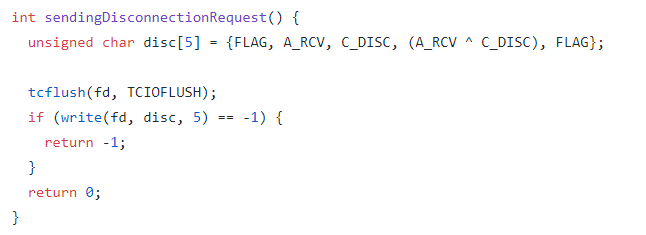
* **waitConnect**

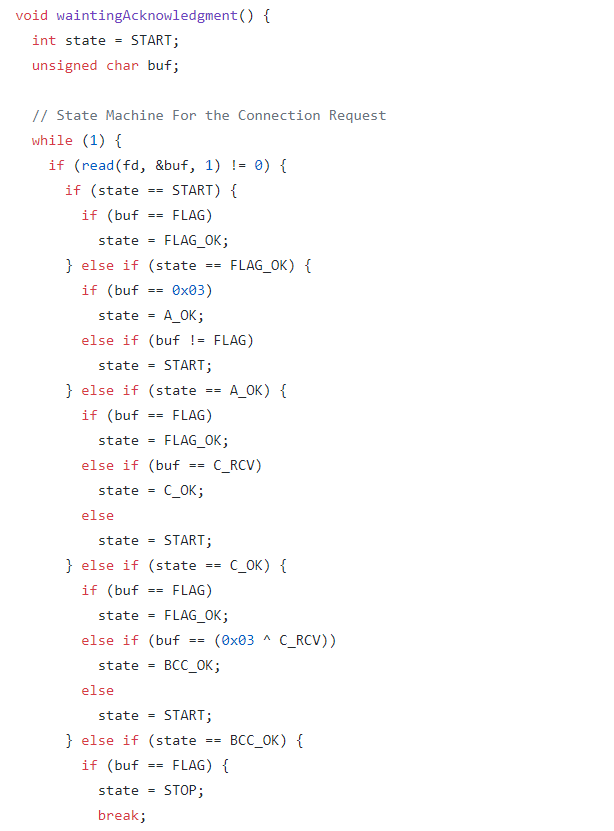
****

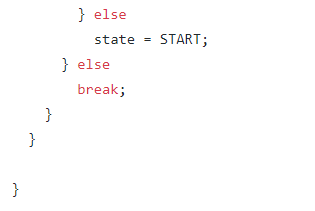
* **waitingDisconnectionRequest**



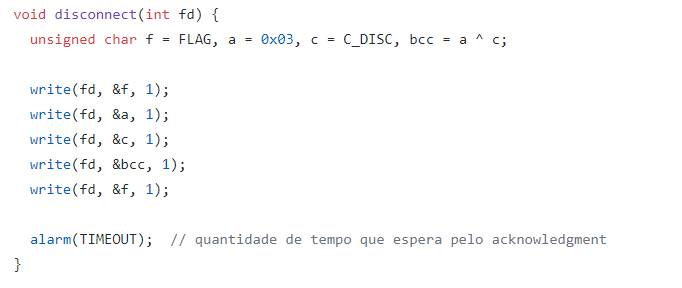
****

* **sendDisconnectionRequest** 
* **waitingAcknowledgement**

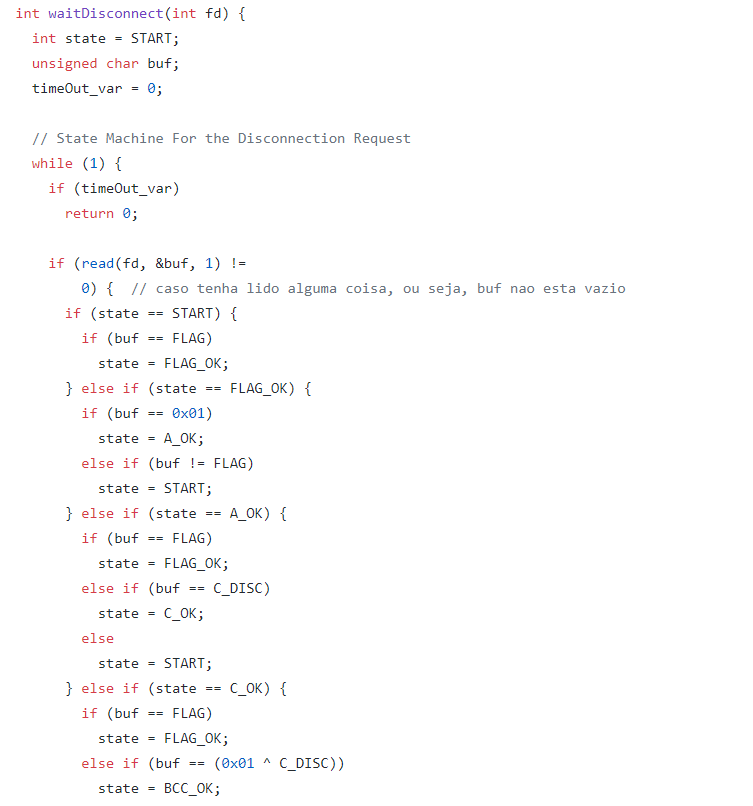


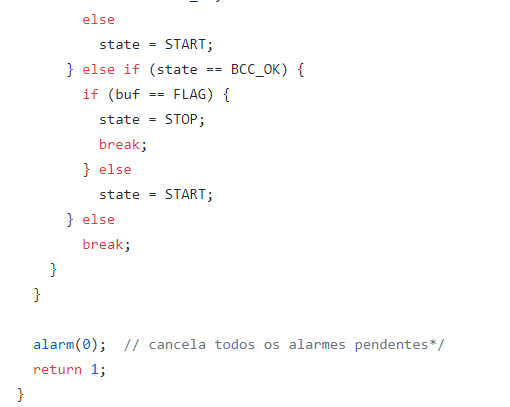


* **disconnect**

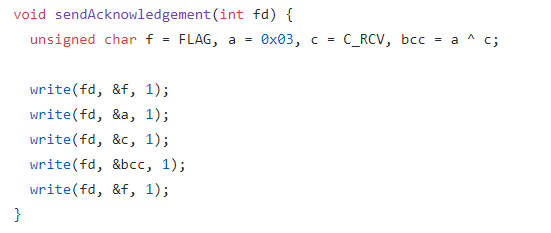
****

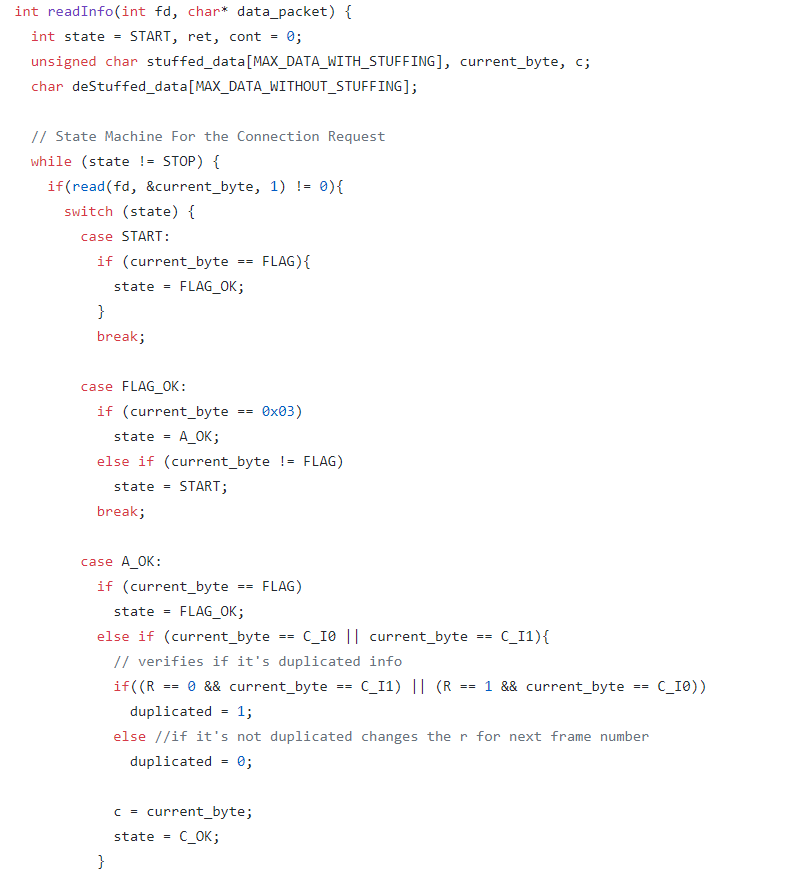
* **waitDisconnect**

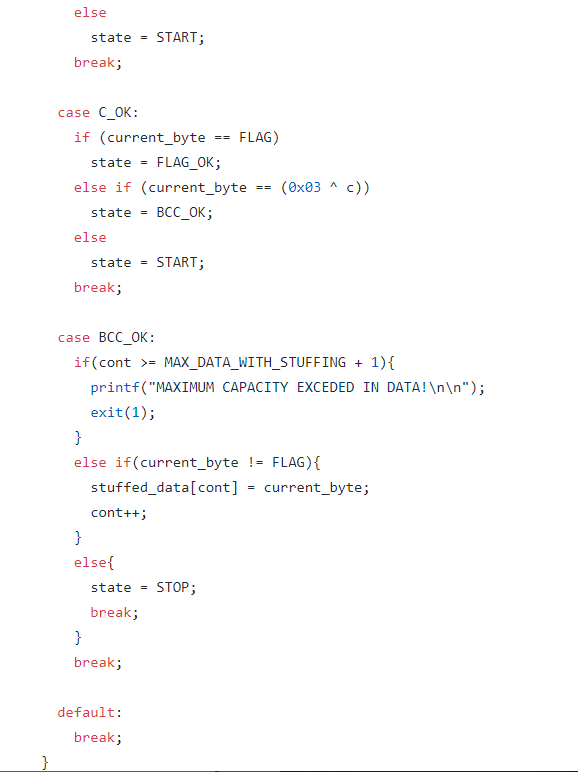
****

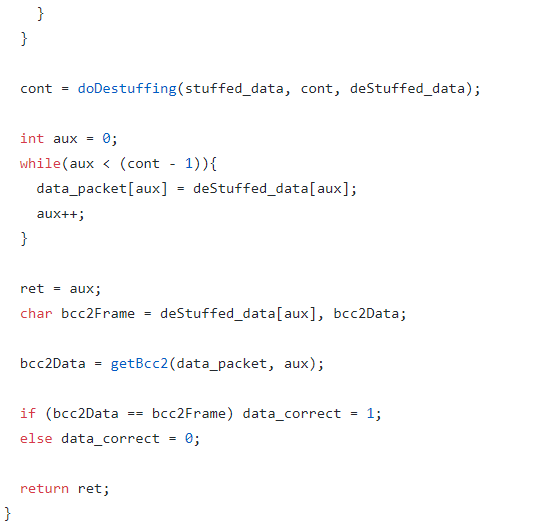


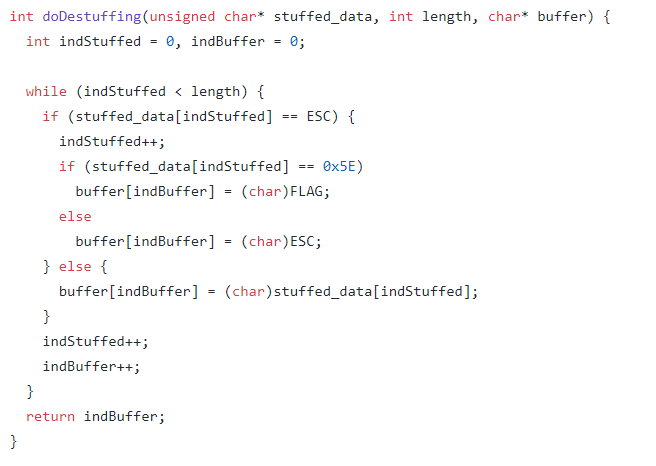
* **sendAcknowledgment**

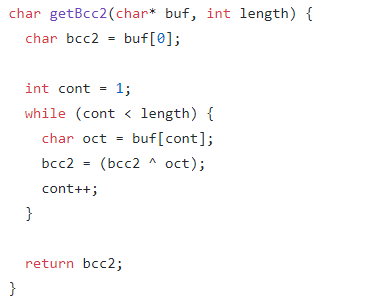


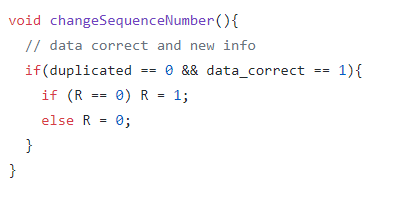
* **readInfo**

****

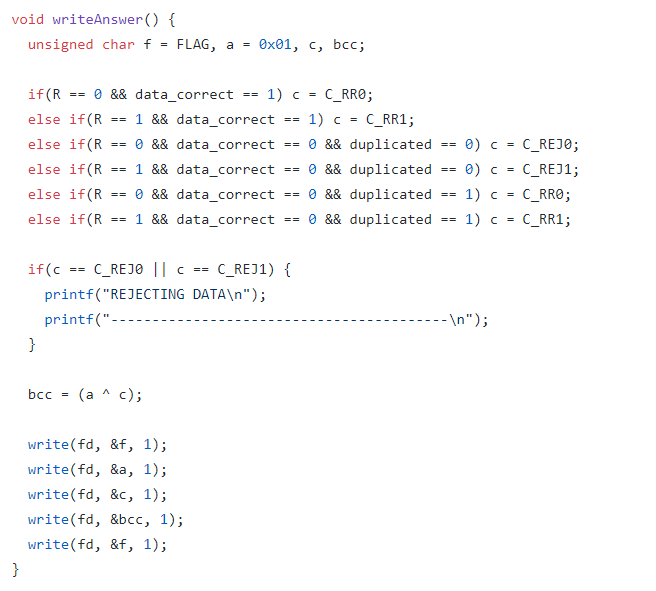
* **doDestuffing**



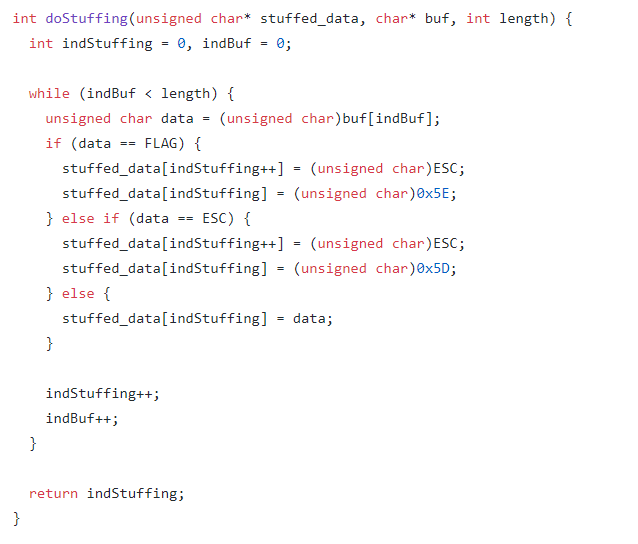
* **getBcc2**
* **changeSequenceNumber**

****

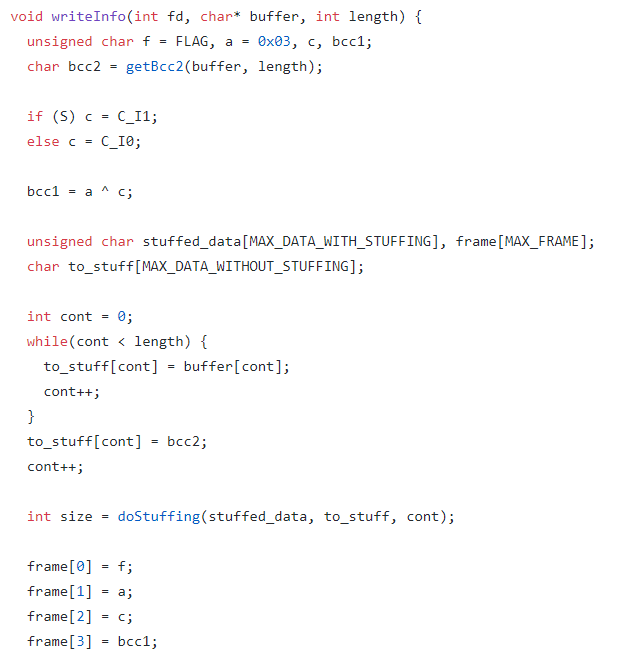
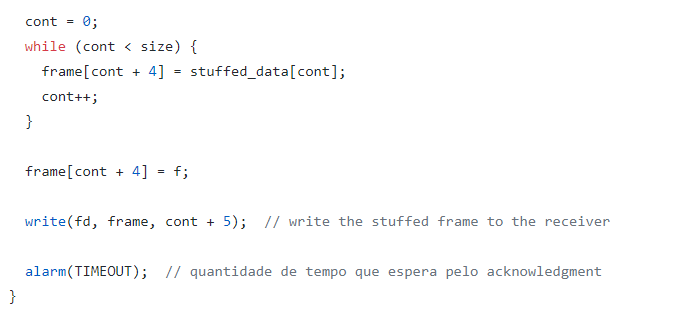
* **writeAnswer**

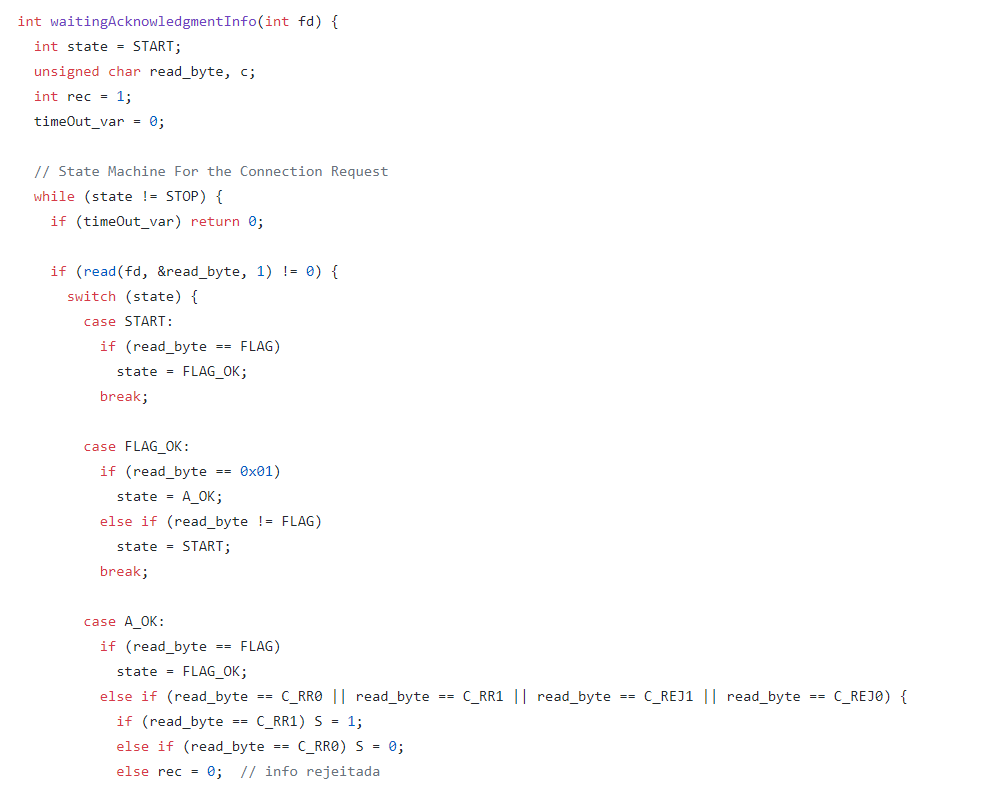
****

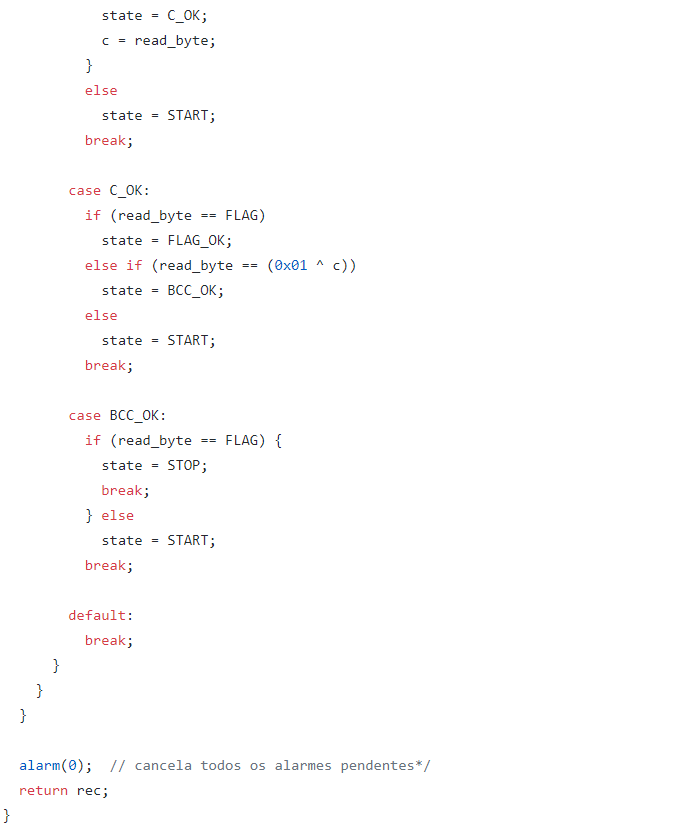
* **doStuffing**



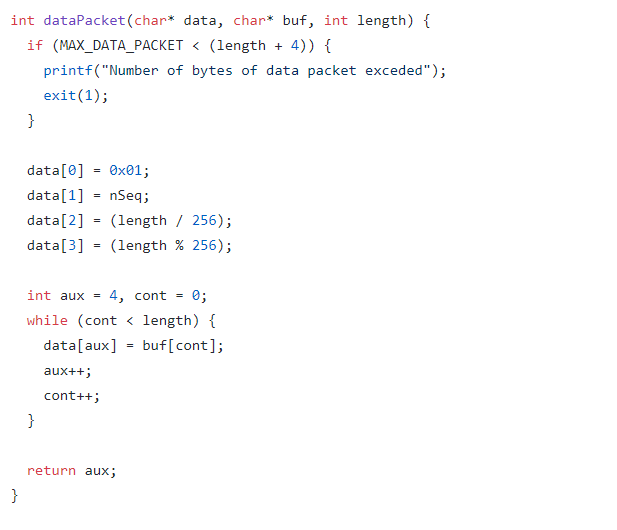
* **writeInfo**



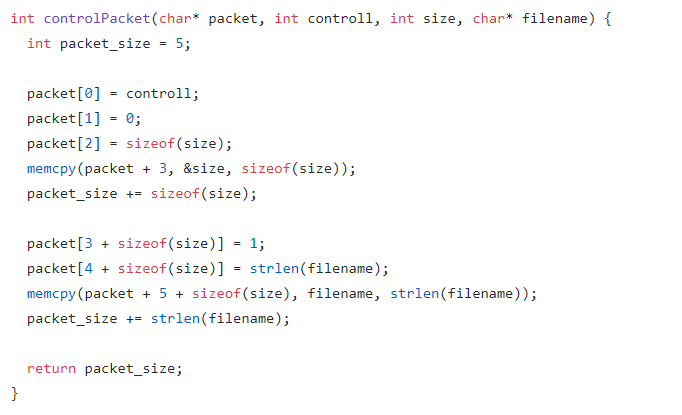
* **waitingAcknowledgmentInfo**

****

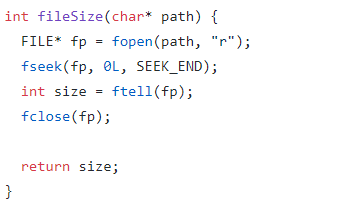
* **dataPacket**



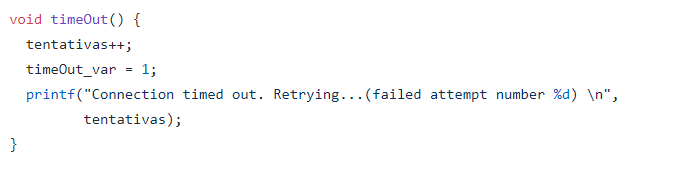
* **controlPacket**



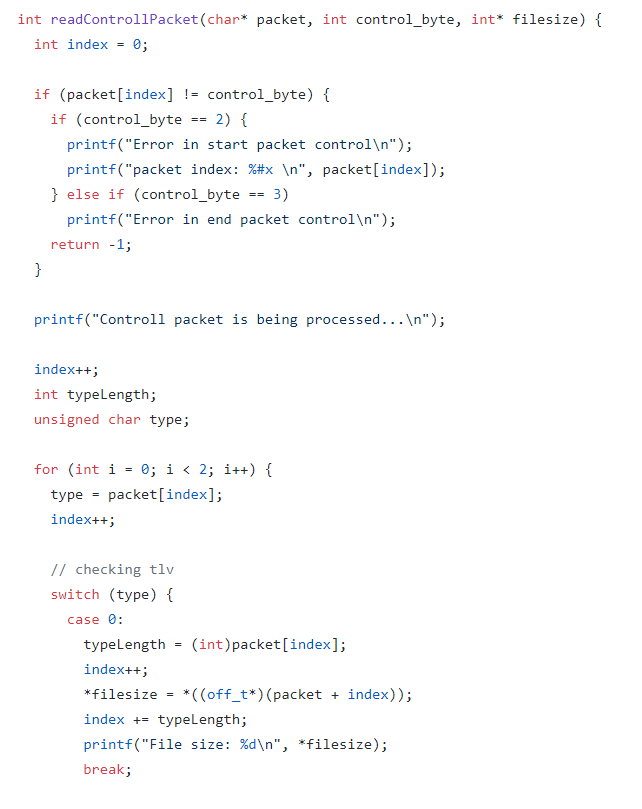
* **fileSize**

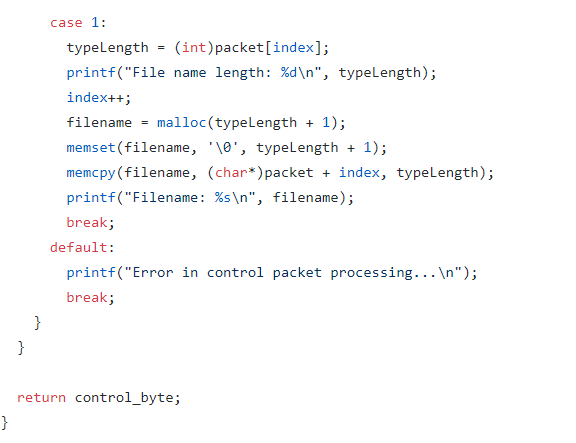
****

* **timeOut**

****

* **readControllPacket**

****



* **readDataPacket**

****