GPSTrack

Generated by Doxygen 1.9.8

Chapter 1

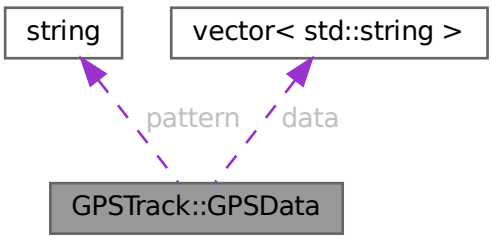
Class Documentation

1.1 GPSTrack::GPSData Class Reference

Classe responsável por representar os dados do GPS.

`#include <GPSTrack.hpp>`

Collaboration diagram for GPSTrack::GPSData:



Public Member Functions

- GPSData ()

  Construtor Default.
- bool parsing (int code_pattern, const std::vector< std::string > &data_splitted)

  Setará os dados baseado no padrão de mensagem recebido.
- std::string to_csv () const

  Retorna os dados armazenados em formato CSV.

Static Public Member Functions

- static std::string converter_lat_lon (const std::string &string_numerica, const std::string &string_hemisf)

  Função estática auxiliar para converter coordenadas NMEA (latitude/longitude) para graus decimais.

Private Attributes

- std::string pattern
- std::vector< std::string > data

  Organizaremos os dados neste vetor.

1.1.1 Detailed Description

Classe responsável por representar os dados do GPS.

Utilizar struct mostrou-se básico demais para atenter as necessidades de representação dos dados do GPS. Com isso, a evolução para Class tornou-se inevitável.

O sensor inicia a comunicação enviando mensagens do tipo $GPTXT. Essas mensagens são mensagens de texto informativas, geralmente vazias. Esse é um comportamento normal nos primeiros segundos após energizar o módulo.

Após adquirir sinais de satélites e iniciar a navegação, o módulo passa a enviar sentenças NMEA padrão, que trazem informações úteis:

- $GPRMC (Recommended Minimum Navigation Information): Fornece dados essenciais de navegação, como horário UTC, status, latitude, longitude, velocidade, curso e data.
- $GPVTG (Course over Ground and Ground Speed): Informa direção do movimento (rumo) e velocidade sobre o solo.
- $GPGGA (Global Positioning System Fix Data): Dados de fixação do GPS, incluindo número de satélites usados, qualidade do sinal, altitude e posição.
- $GPGSA (GNSS DOP and Active Satellites): Status dos satélites em uso e precisão (DOP - Dilution of Precision).
- $GPGSV (GNSS Satellites in View): Informações sobre os satélites visíveis, como elevação, azimute e intensidade de sinal.
- $GPGLL (Geographic Position - Latitude/Longitude): Posição geográfica em latitude e longitude, com horário associado.

Sendo assim, o sensor sai de um modo de inicialização para operacionalidade completa.

Como nosso próposito é apenas localização, nos interessa apenas o padrão GGA, o qual oferece dados profundos de localização.

1.1.2 Constructor & Destructor Documentation

1.1.2.1 GPSData()

```
GPSTrack::GPSData::GPSData ( )  [inline]
```

Construtor Default.

Reserva no vetor de dados 4 strings, respectivamente para horário UTC, latitude, longitude e altitude.

1.1.3 Member Function Documentation

1.1.3.1 converter_lat_lon()

```
static std::string GPSTrack::GPSData::converter_lat_lon (
            const std::string & string_numerica,
            const std::string & string_hemisf )  [inline], [static]
```

Função estática auxiliar para converter coordenadas NMEA (latitude/longitude) para graus decimais.

Parameters

string_numerica	String com a coordenada em formato NMEA (ex: "2257.34613").
string_hemisf	String com o hemisfério correspondente ("N", "S", "E", "W").

Returns

String coordenada em graus decimais (negativa para hemisférios Sul e Oeste).

Here is the caller graph for this function:

1.1.3.2 parsing()

```
bool GPSTrack::GPSData::parsing (
            int code_pattern,
            const std::vector< std::string > & data_splitted )  [inline]
```

Setará os dados baseado no padrão de mensagem recebido.

Parameters

code_pattern	Código para informar que padrão de mensagem recebeu.
data_splitted	Vetor de dados da mensagem recebida.

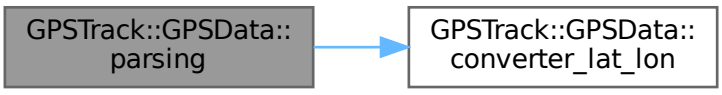
Returns

Retornará true caso seja bem sucedido. False, caso contrário.

Tradução de códigos:

- 0 == GPGGA

A partir do padrão de mensagem recebida, organizaremos o vetor com dados relevantes.
Here is the call graph for this function:



Here is the caller graph for this function:

1.1.3.3 to_csv()

```
std::string GPSTrack::GPSData::to_csv ( ) const  [inline]
```

Retorna os dados armazenados em formato CSV.

Returns

std::string Linha CSV com os valores.

Here is the caller graph for this function:

1.1.4 Member Data Documentation

1.1.4.1 data

```
std::vector<std::string> GPSTrack::GPSData::data  [private]
```

Organizaremos os dados neste vetor.

1.1.4.2 pattern

```
std::string GPSTrack::GPSData::pattern  [private]
```

The documentation for this class was generated from the following file:

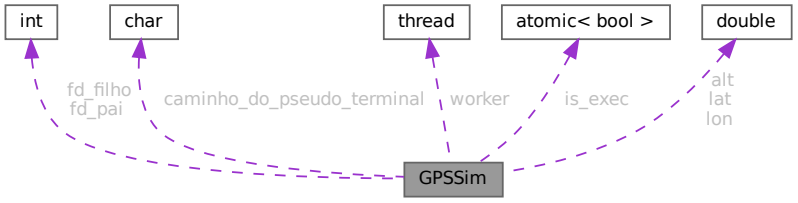
- src/GPSTrack.hpp

1.2 GPSSim Class Reference

Versão Simulada do Sensor GPS, gerando frases no padrão NMEA.

`#include <GPSSim.hpp>`

Collaboration diagram for GPSSim:



Classes

- class NMEAGenerator

  Classe responsável por agrupar as funções geradoras de sentenças NMEA.

Public Member Functions

- GPSSim (double latitude_inicial_graus, double longitude_inicial_graus, double altitude_metros)

  Construtor do GPSSim.
- ~GPSSim ()

  Destrutor do GPSSim.
- void init ()

  Inicia a geração de frases no padrão NMEA em uma thread separada.
- void stop ()

  Encerrará a geração de frases NMEA e aguardará a finalização da thread.
- std::string get_path_pseudo_term () const

  Obtém o caminho do terminal que estamos executando de forma filial.

Private Member Functions

- void loop ()

  Loop principal responsável pela geração e transmissão de dados simulados.

Static Private Member Functions

- static void degrees_to_NMEA (double graus_decimais, bool is_lat, std::string &ddmm, char &hemisf)

  Converte graus decimais para formato NMEA de localização, (ddmm.mmmm).
- static std::string format_integer (int valor, int quant_digitos)

  Formatada um número inteiro com dois dígitos, preenchendo com zero à esquerda.
- static std::tm get_utc_time ()

  Obtém o tempo UTC atual, horário em Londres.
- static std::string build_nmea_string (const std::string &corpo_frase)

  Finaliza uma sentença NMEA a partir do corpo da frase.

Private Attributes

- int fd_pai {0}
- int fd_filho {0}
- char caminho_do_pseudo_terminal [128]
- std::thread worker
- std::atomic< bool > is_exec {false}
- double lat
- double lon
- double alt

1.2.1 Detailed Description

Versão Simulada do Sensor GPS, gerando frases no padrão NMEA.

Criará um par de pseudo-terminais (PTY) para simular o funcionamento do sensor. Intervaladamente, gera frases NMEA simuladas.

1.2.2 Constructor & Destructor Documentation

1.2.2.1 GPSSim()

```
GPSSim::GPSSim (
            double latitude_inicial_graus,
            double longitude_inicial_graus,
            double altitude_metros )  [inline]
```

Construtor do GPSSim.

Inicializa alguns parâmetros de posição simulada e, criando os pseudo-terminais, configura-os para o padrão do módulo real.

Parameters

latitude_inicial_graus	Latitude inicial em graus decimais
longitude_inicial_graus	Longitude inicial em graus decimais
altitude_metros	Altitude inicial em metros, setada para 10.

1.2.2.2 ~GPSSim()

```
GPSSim::~GPSSim ( )  [inline]
```

Destrutor do GPSSim.

Chama a função `stop()` e, após verificar existência de terminal Pai, fecha-o. Here is the call graph for this

function:

1.2.3 Member Function Documentation

1.2.3.1 build_nmea_string()

```
static std::string GPSSim::build_nmea_string (
            const std::string & corpo_frase )  [inline], [static], [private]
```

Finaliza uma sentença NMEA a partir do corpo da frase.

Calcula o valor de paridade (checksum) do corpo da frase fornecida, em seguida adiciona os delimitadores e flags no formato NMEA (prefixo '$', sufixo '∗', valor de paridade em hexadecimal e "\r\n").

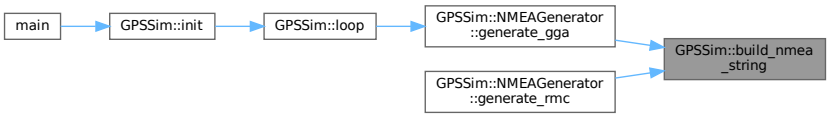
Parameters

corpo_frase	Corpo da frase NMEA sem os indicadores iniciais ('$') e finais ('∗' e checksum).

Returns

std::string Sentença NMEA completa, pronta para transmissão.

Here is the caller graph for this function:



1.2.3.2 degrees_to_NMEA()

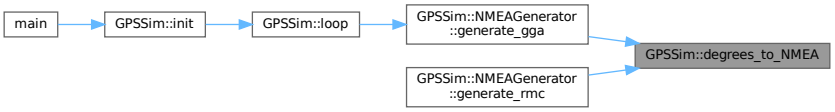
```
static void GPSSim::degrees_to_NMEA (
            double graus_decimais,
            bool is_lat,
            std::string & ddmm,
            char & hemisf )  [inline], [static], [private]
```

Converte graus decimais para formato NMEA de localização, (ddmm.mmmm).

Parameters

	graus_decimais	Valor em graus decimais
	is_lat	Flag de eixo
out	*ddmm*	String com valor formatado em graus e minutos
out	*hemisf*	Caractere indicando Hemisfério

Here is the caller graph for this function:



1.2.3.3 format_integer()

```
static std::string GPSSim::format_integer (
            int valor,
            int quant_digitos )  [inline], [static], [private]
```

Formatada um número inteiro com dois dígitos, preenchendo com zero à esquerda.

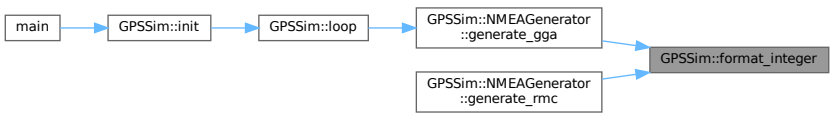
Parameters

valor	Número a ser formatado
quant_digitos	Quantidade de Dígitos presente

Returns

string formatada

Here is the caller graph for this function:



1.2.3.4 get_path_pseudo_term()

```
std::string GPSSim::get_path_pseudo_term ( ) const  [inline]
```

Obtém o caminho do terminal que estamos executando de forma filial.

Returns

String correspondendo ao caminho do dispositivo.

Here is the caller graph for this function:

1.2.3.5 get_utc_time()

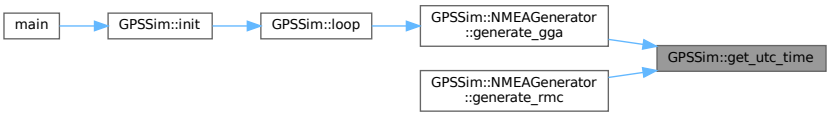
```
static std::tm GPSSim::get_utc_time ( )  [inline], [static], [private]
```

Obtém o tempo UTC atual, horário em Londres.

Returns

Struct std::tm contendo o tempo em UTC

Here is the caller graph for this function:



1.2.3.6 init()

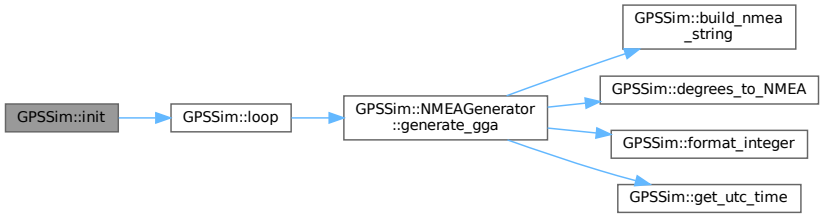
```
void GPSSim::init ( )  [inline]
```

Inicia a geração de frases no padrão NMEA em uma thread separada.

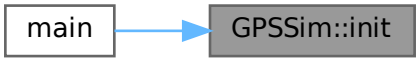
O padrão NMEA é o protocolo padrão usado por módulos GPS, cada mensagem começa com $ e termina com \r\n. Exemplo:

- $origem,codificacao_usada,dados1,dados2,...∗paridade

Here is the call graph for this function:



Here is the caller graph for this function:



1.2.3.7 loop()

```
void GPSSim::loop ( )  [inline], [private]
```

Loop principal responsável pela geração e transmissão de dados simulados.

Esta função executa um laço contínuo enquanto o simulador estiver ativo (`_is_exec`). Em cada iteração:

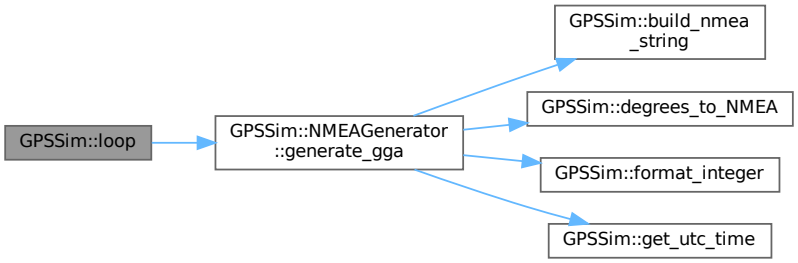
- Inicializa ou atualiza a posição simulada (latitude e longitude).
- Gera uma sentença NMEA do tipo GGA a partir da posição atual.
- Transmite a sentença gerada através do descritor de escrita `_fd_pai`.
- Aguarda o período de atualização definido em `_periodo_atualizacao`.
- Atualiza a posição simulada chamando `_update_position()`.

O loop termina automaticamente quando `_is_exec` é definido como falso.

Note

Esta função é bloqueante e deve ser executada em uma thread dedicada para não interromper o fluxo principal do programa.

Here is the call graph for this function:



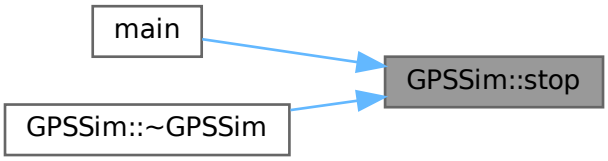
Here is the caller graph for this function:

1.2.3.8 stop()

```
void GPSSim::stop ( )  [inline]
```

Encerrará a geração de frases NMEA e aguardará a finalização da thread.

Verifica a execução da thread e encerra-a caso exista. Força a finalização da thread geradora de mensagens. Here is the caller graph for this function:



1.2.4 Member Data Documentation

1.2.4.1 alt

```
double GPSSim::alt  [private]
```

1.2.4.2 caminho_do_pseudo_terminal

```
char GPSSim::caminho_do_pseudo_terminal[128]  [private]
```

1.2.4.3 fd_filho

```
int GPSSim::fd_filho {0}  [private]
```

1.2.4.4 fd_pai

```
int GPSSim::fd_pai {0}  [private]
```

1.2.4.5 is_exec

```
std::atomic<bool> GPSSim::is_exec {false}  [private]
```

1.2.4.6 lat

```
double GPSSim::lat  [private]
```

1.2.4.7 lon

```
double GPSSim::lon  [private]
```

1.2.4.8 worker

```
std::thread GPSSim::worker  [private]
```

The documentation for this class was generated from the following file:

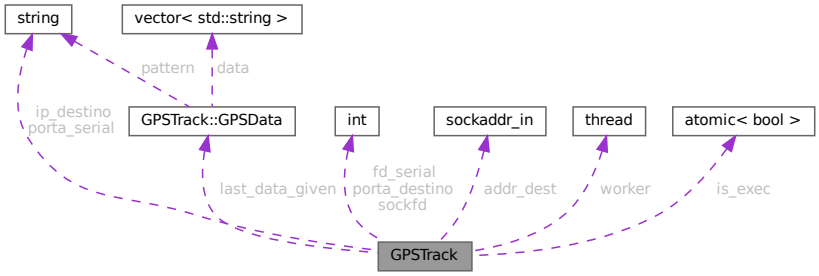
- src/GPSSim.hpp

1.3 GPSTrack Class Reference

Classe responsável por obter o tracking da carga.

`#include <GPSTrack.hpp>`

Collaboration diagram for GPSTrack:



Classes

- class GPSData

  Classe responsável por representar os dados do GPS.

Public Member Functions

- GPSTrack (const std::string &ip_destino_, int porta_destino_, const std::string &porta_serial_)

  Construtor da Classe.
- ~GPSTrack ()

  Destrutor da classe.
- void init ()

  Inicializa a thread trabalhadora.
- void stop ()

  Finaliza a thread de trabalho de forma segura.

Static Public Member Functions

- static std::vector< std::string > split (const std::string &string_de_entrada, char separador=',')

  Função estática auxiliar para separar uma string em vetores de string.

Private Member Functions

- void open_serial ()

  Abre e configura a porta serial para comunicação o sensor.
- std::string read_serial ()

  Lê dados da porta serial até encontrar uma quebra de linha.
- void loop ()

  Executa o loop principal de leitura, interpretação e envio de dados via UDP.

Private Attributes

- std::string ip_destino
- int porta_destino
- int sockfd
- sockaddr_in addr_dest
- std::thread worker
- std::atomic< bool > is_exec {false}
- GPSData last_data_given
- std::string porta_serial
- int fd_serial = -1

1.3.1 Detailed Description

Classe responsável por obter o tracking da carga.

Responsabilidades:

- Obter os dados do sensor NEO6MV2
- Interpretar esses dados, gerando informações
- Enviar as informações via socket UDP em formato CSV

Cada uma dessas responsabilidades está associada a um método da classe, respectivamente:

- read_serial()
- GPSData::parsing()
- send()

Os quais estarão sendo repetidamente executados pela thread worker a fim de manter a continuidade de informações.

Não há necessidade de mais explicações, já que o fluxo de funcionamento é simples.

1.3.2 Constructor & Destructor Documentation

1.3.2.1 GPSTrack()

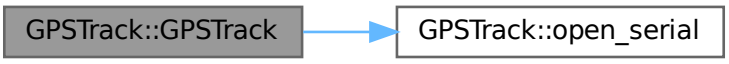
```
GPSTrack::GPSTrack (
            const std::string & ip_destino_,
            int porta_destino_,
            const std::string & porta_serial_ )  [inline]
```

Construtor da Classe.

Parameters

*ip_destino_*	Endereço IP de destino.
*porta_destino_*	Porta UDP de destino
*porta_serial_*	Caminho da porta serial

Inicializa a comunicação UDP e abre a comunicação serial. Here is the call graph for this function:



1.3.2.2 ~GPSTrack()

```
GPSTrack::~GPSTrack ( )  [inline]
```

Destrutor da classe.

Realiza a limpeza adequada dos recursos da classe, garantindo o término seguro das operações.

A ordem de operações é importante:

1. Interrompe a thread de execução
2. Fecha o socket de comunicação
3. Fecha a porta serial

Here is the call graph for this function:

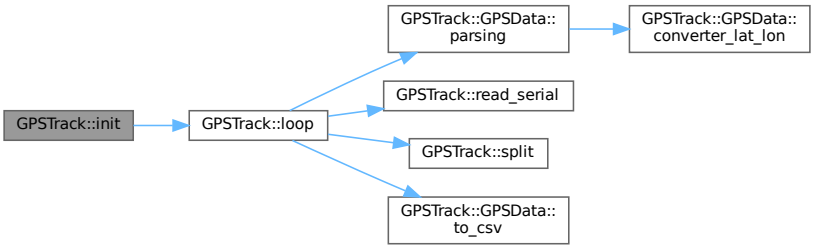
1.3.3 Member Function Documentation

1.3.3.1 init()

```
void GPSTrack::init ( )  [inline]
```

Inicializa a thread trabalhadora.

Garante que apenas uma única instância da thread de execução será iniciada, utilizando um flag atômico para controle de estado. Se a thread já estiver em execução, a função retorna imediatamente sem realizar nova inicialização. Ao iniciar, exibe uma mensagem colorida no terminal e cria uma thread worker que executa o loop principal de leitura e comunicação. Here is the call graph for this function:



Here is the caller graph for this function:

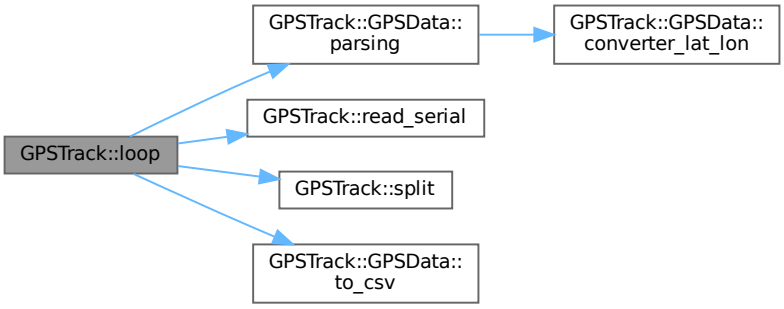
1.3.3.2 loop()

```
void GPSTrack::loop ( )  [inline], [private]
```

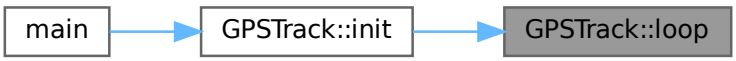
Executa o loop principal de leitura, interpretação e envio de dados via UDP.

Esta função realiza continuamente a leitura de dados da porta serial, interpreta as mensagens GPS no formato GPGGA, armazena os dados processados e, se desejado, os exibe em formato CSV.

O loop executa enquanto a flag de execução estiver ativa, com uma pausa de 1 segundo entre cada iteração para evitar consumo excessivo de CPU. Here is the call graph for this function:



Here is the caller graph for this function:



1.3.3.3 open_serial()

```
void GPSTrack::open_serial ( )  [inline], [private]
```

Abre e configura a porta serial para comunicação o sensor.

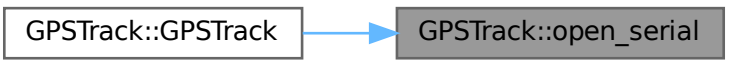
Estabelece a conexão serial utilizando a porta serial especificada. Todos os parâmetros necessários para uma comunicação estável com o dispositivo, incluindo velocidade, formato de dados e controle de fluxo são setados.

A porta é aberta em modo somente leitura (O_RDONLY) e em modo raw, no qual não há processamento adicional dos caracteres.

Aplicamos as seguintes configurações:

- 9600 bauds
- 8 bits de dados
- Sem paridade
- 1 bit de parada

Here is the caller graph for this function:



1.3.3.4 read_serial()

```
std::string GPSTrack::read_serial ( )  [inline], [private]
```

Lê dados da porta serial até encontrar uma quebra de linha.

- Caracteres de carriage return ('\r') são ignorados durante a leitura.
- A função termina quando encontra '\n' ou quando não há mais dados para ler.
- A leitura é feita caractere por caractere para garantir processamento correto dos dados do GPS que seguem protocolo NMEA.

Here is the caller graph for this function:

1.3.3.5 split()

```
static std::vector< std::string > GPSTrack::split (
            const std::string & string_de_entrada,
            char separador = ',' )  [inline], [static]
```

Função estática auxiliar para separar uma string em vetores de string.

Parameters

string_de_entrada	String que será fatiada.
separador	Caractere que será a flag de separação.

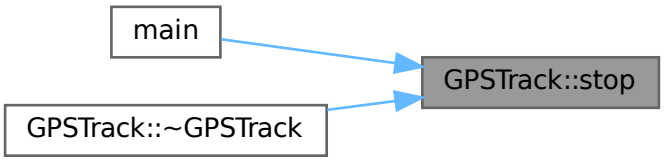
Similar ao método `split` do python, utiliza ',' como caractere separador default. Here is the caller graph for this function:

1.3.3.6 stop()

```
void GPSTrack::stop ( )  [inline]
```

Finaliza a thread de trabalho de forma segura.

Esta função realiza o desligamento controlado da thread de trabalho. Primeiro, altera o flag de execução para falso usando operação atômica. Se a thread estiver joinable (executando), imprime uma mensagem de confirmação e realiza a operação de join para aguardar a finalização segura da thread. Here is the caller graph for this function:



1.3.4 Member Data Documentation

1.3.4.1 addr_dest

```
sockaddr_in GPSTrack::addr_dest  [private]
```

1.3.4.2 fd_serial

```
int GPSTrack::fd_serial = -1  [private]
```

1.3.4.3 ip_destino

```
std::string GPSTrack::ip_destino  [private]
```

1.3.4.4 is_exec

```
std::atomic<bool> GPSTrack::is_exec {false}  [private]
```

1.3.4.5 last_data_given

```
GPSData GPSTrack::last_data_given  [private]
```

1.3.4.6 porta_destino

```
int GPSTrack::porta_destino  [private]
```

1.3.4.7 porta_serial

```
std::string GPSTrack::porta_serial  [private]
```

1.3.4.8 sockfd

```
int GPSTrack::sockfd  [private]
```

1.3.4.9 worker

```
std::thread GPSTrack::worker  [private]
```

The documentation for this class was generated from the following file:

- src/GPSTrack.hpp

1.4 GPSSim::NMEAGenerator Class Reference

Classe responsável por agrupar as funções geradoras de sentenças NMEA.

Static Public Member Functions

- static std::string generate_gga (int lat_graus, int lon_graus, int alt_metros)
  Gera uma frase GGA (Global Positioning System Fix Data)
- static std::string generate_rmc (int lat_graus, int lon_graus, int alt_metros)
  Gera uma frase GGA (Recommended Minimum Navigation Information)

1.4.1 Detailed Description

Classe responsável por agrupar as funções geradoras de sentenças NMEA.

Contém apenas os métodos estáticos que constroem a informação a ser posta na string NMEA.

1.4.2 Member Function Documentation

1.4.2.1 generate_gga()

```
static std::string GPSSim::NMEAGenerator::generate_gga (
            int lat_graus,
            int lon_graus,
            int alt_metros )  [inline], [static]
```

Gera uma frase GGA (Global Positioning System Fix Data)

Apesar de usar apenas valores de lat, long e alt, zera os demais valores.

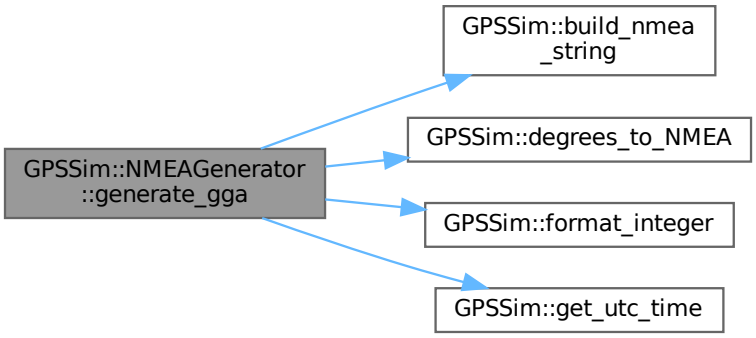
Parameters

lat_graus	Latitude em graus decimais
lon_graus	Longitude em graus decimais
alt_metros	Altitude em metros

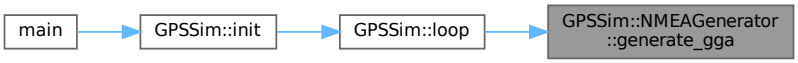
Returns

String correspondendo ao corpo de frase GGA

Here is the call graph for this function:



Here is the caller graph for this function:



1.4.2.2 generate_rmc()

```
static std::string GPSSim::NMEAGenerator::generate_rmc (
            int lat_graus,
            int lon_graus,
            int alt_metros )  [inline], [static]
```

Gera uma frase GGA (Recommended Minimum Navigation Information)

Apesar de usar apenas valores de lat, long e alt, zera os demais valores.

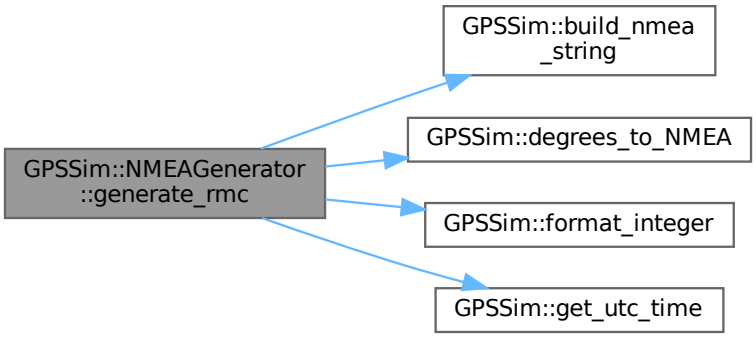
Parameters

lat_graus	Latitude em graus decimais
lon_graus	Longitude em graus decimais
alt_metros	Altitude em metros

Returns

String correspondendo ao corpo de frase GGA

Here is the call graph for this function:



The documentation for this class was generated from the following file:

- src/GPSSim.hpp

Chapter 2

File Documentation

2.1 src/debug.cpp File Reference

Responsável por prover ferramentas de debug.

```
#include <iostream>
#include "GPSTrack.hpp"
#include "GPSSim.hpp"
```

Include dependency graph for debug.cpp:

Functions

- int main ()

2.1.1 Detailed Description

Responsável por prover ferramentas de debug.

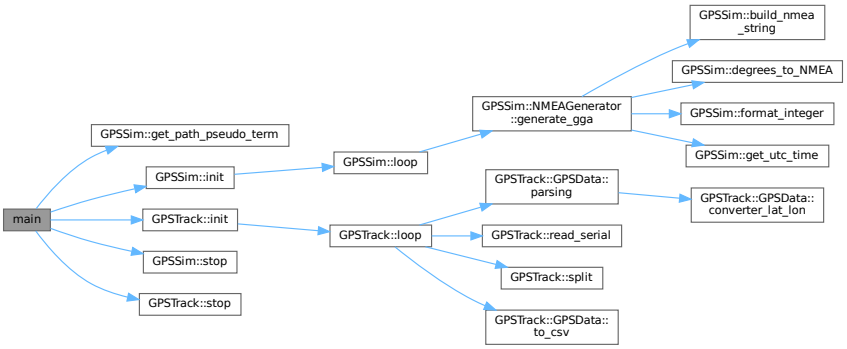
Já que a aplicação deve ser executada dentro da placa, não conseguiríamos executá-la no DeskTop. Para tanto, fez-se necessário o desenvolvimento de ferramentas que possibilitam a debugação de nosso código.

2.1.2 Function Documentation

2.1.2.1 main()

```
int main ( )
```

Here is the call graph for this function:

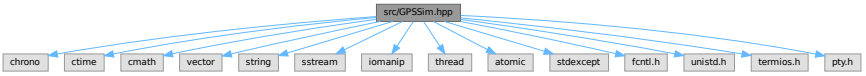


2.2 src/GPSSim.hpp File Reference

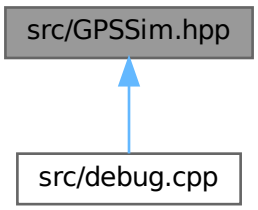
Implementação da Classe Simuladora do GPS6MV2.

```
#include <chrono>
#include <ctime>
#include <cmath>
#include <vector>
#include <string>
#include <sstream>
#include <iomanip>
#include <thread>
#include <atomic>
#include <stdexcept>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
#include <pty.h>
```

Include dependency graph for GPSSim.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class GPSSim

  Versão Simulada do Sensor GPS, gerando frases no padrão NMEA.
- class GPSSim::NMEAGenerator

  Classe responsável por agrupar as funções geradoras de sentenças NMEA.

2.2.1 Detailed Description

Implementação da Classe Simuladora do GPS6MV2.

Supondo que o módulo GPS6MV2 não esteja disponível, a classe implementada neste arquivo tem como objetivo simular todas as funcionalidades do mesmo.

2.3 GPSSim.hpp

Go to the documentation of this file.

```
00001
00008 #ifndef GPSSim_HPP
00009 #define GPSSim_HPP
00010
00011 //------------------------------------------------------------
00012
00013 // Para manipulações de Tempo e de Data
00014 #include <chrono>
00015 #include <ctime>
00016
00017 #include <cmath>
00018 #include <vector>
00019
00020 #include <string>
00021 #include <sstream>
00022 #include <iomanip>
00023
00024 // Para threads e sincronizações
00025 #include <thread>
00026 #include <atomic>
00027
00028 // Para tratamento de erros
00029 #include <stdexcept>
00030
00031 // As seguintes bibliotecas possuem relevância superior
00032 // Por se tratarem de bibliotecas C, utilizaremos o padrão de `::` para explicitar
00033 // que algumas funções advém delas.
```

```
00034 /*
00035 Fornece constantes e funções de controle de descritores de arquivos,
00036 operações de I/O de baixo nível e manipulação de flags de arquivos.
00037 */
00038 #include <fcntl.h>
00039 /*
00040 Fornece acesso a chamadas do OS de baixo nível, incluindo manipulação
00041 de processos, I/O de arquivos, controle de descritores e operações do
00042 sistema de arquivos.
00043 */
00044 #include <unistd.h>
00045 /*
00046 Fornece estruturas e funções para configurar a comunicação
00047 em sistemas Unix. Ele permite o controle detalhado sobre interfaces
00048 de terminal (TTY)
00049 */
00050 #include <termios.h>
00051 /*
00052 Fornece funções para criação e manipulação de pseudo-terminais (PTYs),
00053 um mecanismo essencial em sistemas Unix para emular terminais virtuais.
00054 */
00055 #include <pty.h>
00056
00064 class GPSSim {
00065 private:
00066
00067     // Informações ligadas ao terminal
00068     int fd_pai{0}, fd_filho{0};
00069     char caminho_do_pseudo_terminal[128];
00070
00071     // Thread de Execução Paralela e Flag de Controle
00072     std::thread worker;
00073     std::atomic<bool> is_exec{false};
00074
00075     // Informações de Localização
00076     double lat, lon, alt;
00077
00085     static void
00086     degrees_to_NMEA(
00087         double graus_decimais,
00088         bool is_lat,
00089         std::string& ddmm,
00090         char& hemisf
00091     ){
00092
00093         hemisf = (is_lat) ? (
00094                               ( graus_decimais >= 0 ) ? 'N' : 'S'
00095                              ) :
00096                              (
00097                               ( graus_decimais >= 0 ) ? 'E' : 'W'
00098                              );
00099
00100         double valor_abs = std::fabs(graus_decimais);
00101         int graus  = static_cast<int>(std::floor(valor_abs));
00102         double min = (valor_abs - graus) * 60.0;
00103
00104         // Não utilizamos apenas a função de formatar_inteiro, pois
00105         // utilizaremos o oss em seguida.
00106         std::ostringstream oss;
00107         if(
00108             is_lat
00109         ){
00110
00111             oss « std::setw(2)
00112                 « std::setfill('0')
00113                 « graus
00114                 « std::fixed
00115                 « std::setprecision(4)
00116                 « std::setw(7)
00117                 « std::setfill('0')
00118                 « min;
00119         }
00120         else{
00121
00122             oss « std::setw(3)
00123                 « std::setfill('0')
00124                 « graus
00125                 « std::fixed
00126                 « std::setprecision(4)
00127                 « std::setw(7)
00128                 « std::setfill('0')
00129                 « min;
00130         }
00131
00132         ddmm = oss.str();
00133     }
00134
```

```
00141     static std::string
00142     format_integer(
00143         int valor,
00144         int quant_digitos
00145     ){
00146 
00147         std::ostringstream oss;
00148         oss « std::setw(quant_digitos)
00149             « std::setfill('0')
00150             « valor;
00151         return oss.str();
00152     }
00153 
00158     static std::tm
00159     get_utc_time(){
00160 
00161         using namespace std::chrono;
00162         auto tempo_atual = system_clock::to_time_t(system_clock::now());
00163         std::tm tempo_utc{};
00164         gmtime_r(&tempo_atual, &tempo_utc);
00165         return tempo_utc;
00166     }
00167 
00179     static std::string
00180     build_nmea_string(
00181         const std::string& corpo_frase
00182     ){
00183 
00184         uint8_t paridade = 0;
00185         for(
00186             const char& caract : corpo_frase
00187         ){
00188 
00189             paridade ^= (uint8_t)caract;
00190         }
00191 
00192         std::ostringstream oss;
00193         oss « '$'
00194             « corpo_frase
00195             « '*'
00196             « std::uppercase
00197             « std::hex
00198             « std::setw(2)
00199             « std::setfill('0')
00200             « (int)paridade
00201             « "\r\n";
00202 
00203         return oss.str();
00204     }
00205 
00212     class NMEAGenerator {
00213     public:
00214 
00226         static std::string
00227         generate_gga(
00228             int lat_graus,
00229             int lon_graus,
00230             int alt_metros
00231         ){
00232 
00233             auto tempo_utc = get_utc_time();
00234             std::string lat_nmea, lon_nmea;
00235             char hemisferio_lat, hemisferio_lon;
00236 
00237             degrees_to_NMEA(
00238                 lat_graus,
00239                 true,
00240                 lat_nmea,
00241                 hemisferio_lat
00242             );
00243             degrees_to_NMEA(
00244                 lon_graus,
00245                 false,
00246                 lon_nmea,
00247                 hemisferio_lon
00248             );
00249 
00250             // Formato: hhmmss.ss,lat,N/S,lon,E/W,qualidade,satelites,HDOP,altitude,M,...
00251             std::ostringstream oss;
00252             oss « "GPGGA," « format_integer(tempo_utc.tm_hour, 2)
00253                 « format_integer(tempo_utc.tm_min, 2)
00254                 « format_integer(tempo_utc.tm_sec, 2) « ".00,"
00255                 « lat_nmea   « "," « hemisferio_lat « ","
00256                 « lon_nmea   « "," « hemisferio_lon « ",1,"
00257                 « -1  « "," « std::fixed « std::setprecision(1) « -1 « ","
00258                 « std::fixed « std::setprecision(1) « alt_metros « ",M,0.0,M„";
00259 
```

```
00260             return build_nmea_string(oss.str());
00261         }
00262
00274         static std::string
00275         generate_rmc(
00276             int lat_graus,
00277             int lon_graus,
00278             int alt_metros
00279         ){
00280
00281             auto tempo_utc = get_utc_time();
00282             std::string lat_nmea, lon_nmea;
00283             char hemisferio_lat, hemisferio_lon;
00284
00285             degrees_to_NMEA(
00286                 lat_graus,
00287                 true,
00288                 lat_nmea,
00289                 hemisferio_lat
00290             );
00291             degrees_to_NMEA(
00292                 lon_graus,
00293                 false,
00294                 lon_nmea,
00295                 hemisferio_lon
00296             );
00297
00298             // Formato: hhmmss.ss,A,lat,N/S,lon,E/W,velocidade,curso,data,,,
00299             std::ostringstream oss;
00300             oss « "GPRMC,"
00301                 « format_integer(tempo_utc.tm_hour, 2)
00302                 « format_integer(tempo_utc.tm_min,  2)
00303                 « format_integer(tempo_utc.tm_sec,  2) « ".00,A,"
00304                 « lat_nmea « "," « hemisferio_lat « ","
00305                 « lon_nmea « "," « hemisferio_lon « ","
00306                 « std::fixed « std::setprecision(2) « -1 « ",0.00,"
00307                 « format_integer(tempo_utc.tm_mday, 2)
00308                 « format_integer(tempo_utc.tm_mon + 1, 2)
00309                 « format_integer((tempo_utc.tm_year + 1900) % 100, 2)
00310                 « ",,A";
00311
00312             return build_nmea_string(oss.str());
00313         }
00314     };
00315
00334     void
00335     loop(){
00336
00337         while (is_exec){
00338
00339
00340             std::string saida = NMEAGenerator::generate_gga(lat, lon, alt);
00341             std::cout « "\033[7mGPS6MV2 Simulado Emitindo:\033[0m \n" « saida « std::endl;
00342
00343             // Imprimimos no terminal serial
00344             (void)!::write(fd_pai, saida.data(), saida.size());
00345
00346             // Aguarda o próximo ciclo
00347             std::this_thread::sleep_for(std::chrono::seconds(1));
00348         }
00349     }
00350
00351 public:
00352
00363     GPSSim(
00364         double latitude_inicial_graus,
00365         double longitude_inicial_graus,
00366         double altitude_metros
00367     ) : lat(latitude_inicial_graus),
00368         lon(longitude_inicial_graus),
00369         alt(altitude_metros)
00370     {
00371
00372         // Cria o par de pseudo-terminais
00373         if(
00374             ::openpty( &fd_pai, &fd_filho, caminho_do_pseudo_terminal, nullptr, nullptr ) != 0
00375         ){
00376             throw std::runtime_error("Falha ao criar pseudo-terminal");
00377         }
00378
00379         // Configura o terminal filho para simular o módulo real (9600 8N1)
00380         termios config_com{};              // Cria a estrutura vazia
00381         ::tcgetattr(fd_filho, &config_com); // Lê as configurações atuais e armazena na struct
00382         ::cfsetispeed(&config_com, B9600);   // Definimos velocidade de entrada e de saída
00383         ::cfsetospeed(&config_com, B9600);   // Essa constante está presente dentro do termios.h
00384         // Diversas operações bits a bits
00385         config_com.c_cflag = (config_com.c_cflag & ~CSIZE) | CS8;
```

```
00386         config_com.c_cflag |= (CLOCAL | CREAD);
00387         config_com.c_cflag &= ~(PARENB | CSTOPB);
00388         config_com.c_iflag = IGNPAR;
00389         config_com.c_oflag = 0;
00390         config_com.c_lflag = 0;
00391         tcsetattr(fd_filho, TCSANOW, &config_com);  // Aplicamos as configurações
00392 
00393         // Fecha o filho - será aberto pelo usuário no caminho correto
00394         // Mantemos o Pai aberto para procedimentos posteriores
00395         ::close(fd_filho);
00396     }
00397 
00404     ~GPSSim(){ stop(); if( fd_pai >= 0 ){ ::close(fd_pai); } }
00405 
00414     void
00415     init(){
00416 
00417         if( is_exec.exchange(true) ){ return; }
00418 
00419         worker = std::thread(
00420                             [this]{ loop(); }
00421                             );
00422     }
00423 
00431     void
00432     stop(){
00433 
00434         if( !is_exec.exchange(false) ){ return; }
00435 
00436         if(
00437             worker.joinable()
00438         ){
00439 
00440             worker.join();
00441         }
00442     }
00443 
00448     std::string
00449     get_path_pseudo_term() const { return std::string(caminho_do_pseudo_terminal); }
00450 };
00451 
00452 #endif // GPSSim_HPP
```

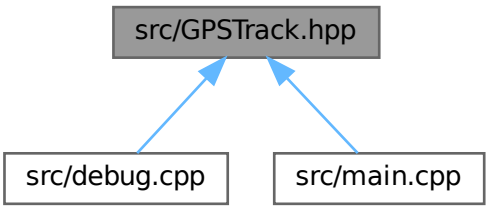
2.4 src/GPSTrack.hpp File Reference

Implementação da solução embarcada.

```
#include <string>
#include <vector>
#include <sstream>
#include <iomanip>
#include <iostream>
#include <cstring>
#include <chrono>
#include <cmath>
#include <ctime>
#include <thread>
#include <atomic>
#include <stdexcept>
#include <fcntl.h>
#include <termios.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
```

Include dependency graph for GPSTrack.hpp:

This graph shows which files directly or indirectly include this file:



Classes

- class GPSTrack

  Classe responsável por obter o tracking da carga.
- class GPSTrack::GPSData

  Classe responsável por representar os dados do GPS.

2.4.1 Detailed Description

Implementação da solução embarcada.

2.5 GPSTrack.hpp

Go to the documentation of this file.

```
00001
00005 #ifndef GPSTRACK_HPP
00006 #define GPSTRACK_HPP
00007
00008
00009 //-----------------------------------------------
00010 #include <string>
00011 #include <vector>
00012 #include <sstream>
00013 #include <iomanip>
00014 #include <iostream>
00015 #include <cstring>
00016
00017 #include <chrono>
00018 #include <cmath>
00019 #include <ctime>
00020
00021 #include <thread>
00022 #include <atomic>
00023
00024 #include <stdexcept>
00025
00026 // Específicos de Sistemas Linux
00027 #include <fcntl.h>
00028 #include <termios.h>
00029 #include <unistd.h>
00030 #include <sys/socket.h>
00031 #include <arpa/inet.h>
00032 #include <netinet/in.h>
00033
00054 class GPSTrack {
00055 public:
```

```
00056
00099     class GPSData {
00100     private:
00101
00102         std::string pattern;
00103         std::vector<std::string> data;
00104
00105     public:
00106
00114         GPSData(){ data.reserve(4); }
00115
00122         static std::string
00123         converter_lat_lon(
00124             const std::string& string_numerica,
00125             const std::string& string_hemisf
00126         ){
00127
00128             if( string_numerica.empty() ){ return ""; }
00129
00130             double valor_cru = 0;
00131             try {
00132
00133                 valor_cru = std::stod(string_numerica);
00134             }
00135             catch (std::invalid_argument&) {
00136
00137                 std::cout « "\033[1;31mErro dentro de converter_lat_lon, valor inválido para stod:
      \033[0m"
00138                           « string_numerica
00139                           « std::endl;
00140             }
00141
00142             // Parsing dos valores
00143             double graus   = floor(valor_cru / 100);
00144             double minutos = valor_cru - graus * 100;
00145             double coordenada = (graus + minutos / 60.0) * ( (string_hemisf == "S" || string_hemisf ==
      "W" ) ? -1 : 1 );
00146
00147             return std::to_string(coordenada);
00148         }
00149
00163         bool
00164         parsing(
00165             int code_pattern,
00166             const std::vector<std::string>& data_splitted
00167         ){
00168
00169             data.clear(); // Garantimos que está limpo.
00170
00171             if(
00172                 code_pattern == 0
00173             ){
00174                 // Em gga, os dados corretos estão em:
00175
00176                 int idx_data_useful[] = {
00177                                          1, // Horário UTC
00178                                          2, // Latitude  em NMEA
00179                                          4, // Longitude em NMEA
00180                                          9  // Altitude
00181                                          };
00182
00183                 for(
00184                     const auto& idx : idx_data_useful
00185                 ){
00186
00187                     if( idx == 2 || idx == 4 ){
00188                         data.emplace_back(
00189                                           std::move(converter_lat_lon(
00190                                                                       data_splitted[idx],
00191                                                                       data_splitted[idx + 1]
00192                                                                       ))
00193                                           );
00194                     }
00195                     else{
00196                         // Então basta adicionar
00197                         data.emplace_back(
00198                                           // Método bizurado para não realizarmos cópias
00199                                           std::move(data_splitted[idx])
00200                                           );
00201                     }
00202                 }
00203
00204                 return true;
00205             }
00206             // ... podemos escalar para novos padrões de mensagem
00207
00208             return false;
```

```
00209         }
00210
00215         std::string
00216         to_csv() const {
00217
00218             std::ostringstream oss;
00219             for (
00220                 int i = 0;
00221                     i < 4;
00222                     i++
00223             ){
00224                 if(i > 0){ oss « ","; }
00225
00226                 oss « data[i];
00227             }
00228
00229             return oss.str();
00230         }
00231     };
00232
00240     static std::vector<std::string>
00241     split(
00242         const std::string& string_de_entrada,
00243         char separador=','
00244     ){
00245
00246         std::vector<std::string>      elementos;
00247         std::stringstream ss(string_de_entrada);
00248
00249         std::string elemento_individual;
00250         while(
00251             std::getline(
00252                       ss,
00253                       elemento_individual,
00254                       separador
00255                      )
00256         ){
00257             if(
00258                 !elemento_individual.empty()
00259             ){
00260
00261                 elementos.push_back(elemento_individual);
00262             }
00263         }
00264
00265         return elementos;
00266     }
00267
00268 private:
00269     // Relacionadas ao Envio UDP
00270     std::string ip_destino;
00271     int      porta_destino;
00272     int             sockfd;
00273     sockaddr_in  addr_dest;
00274
00275     // Relacionados ao fluxo de funcionamento
00276     std::thread                worker;
00277     std::atomic<bool> is_exec{false};
00278
00279     // Relacionados à comunicação com o sensor
00280     GPSData   last_data_given;
00281     std::string  porta_serial;
00282     int        fd_serial = -1;
00283
00302     void
00303     open_serial(){
00304
00305         fd_serial = ::open(
00306                               porta_serial.c_str(),
00307                               // O_RDONLY: garante apenas leitura
00308                               // O_NOCTTY: impede que a porta se torne o terminal controlador do
      processo
00309                               // O_SYNC:   garante que as operações de escrita sejam completadas
      fisicamente
00310                               O_RDONLY | O_NOCTTY | O_SYNC
00311                              );
00312
00313         // Confirmação de sucesso
00314         if( fd_serial < 0 ){ throw std::runtime_error("\033[1;31mErro ao abrir porta serial do
      GPS\033[0m"); }
00315
00316         // Struct para armazenarmos os parâmetros da comunicação serial.
00317         termios tty{};
00318         if(
00319             ::tcgetattr(
00320                         fd_serial,
00321                         &tty
```

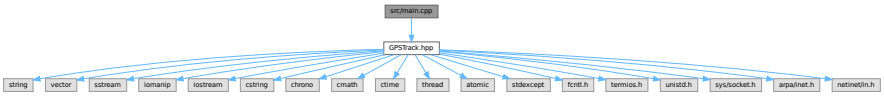
```
00322                       ) != 0
00323         ){ throw std::runtime_error("\033[1;31mErro ao tentar configurar a porta serial,
      especificamente, tcgetattr\033[0m"); }
00324
00325         // Setamos velocidade
00326         ::cfsetospeed(&tty, B9600);
00327         ::cfsetispeed(&tty, B9600);
00328
00329         // Modo raw para não haver processamento por parte do sensor.
00330         ::cfmakeraw(&tty);
00331
00332         // Configuração 8N1
00333         tty.c_cflag &= ~CSIZE;
00334         tty.c_cflag |= CS8;         // 8 bits
00335         tty.c_cflag &= ~PARENB;     // sem paridade
00336         tty.c_cflag &= ~CSTOPB;     // 1 stop bit
00337         tty.c_cflag &= ~CRTSCTS;    // sem controle de fluxo por hardware
00338
00339         // Habilita leitura na porta
00340         tty.c_cflag |= (CLOCAL | CREAD);
00341
00342         // Não encerra a comunição serial quando 'desligamos'
00343         tty.c_cflag &= ~HUPCL;
00344
00345         tty.c_iflag &= ~IGNBRK;
00346         tty.c_iflag &= ~(IXON | IXOFF | IXANY); // sem controle de fluxo por software
00347         tty.c_lflag = 0;                        // sem canonical mode, echo, signals
00348         tty.c_oflag = 0;
00349
00350         tty.c_cc[VMIN]  = 1; // lê pelo menos 1 caractere
00351         tty.c_cc[VTIME] = 1; // timeout em décimos de segundo (0.1s)
00352
00353         if(
00354             ::tcsetattr(
00355                         fd_serial,
00356                         TCSANOW,
00357                         &tty
00358                        ) != 0
00359         ){ throw std::runtime_error("\033[1;31mErro ao tentar setar configurações na comunicação
      serial, especificamente, tcsetattr\033[0m"); }
00360     }
00361
00371     std::string
00372     read_serial(){
00373
00374         std::string buffer;
00375         char caract = '\0';
00376
00377         while(
00378             true
00379         ){
00380
00381             int n = read(
00382                         fd_serial,
00383                         &caract,
00384                         1
00385                         );
00386
00387             // Confirmação de sucesso.
00388             if(n > 0){
00389
00390                 // Então é caractere válido.
00391                 if(caract == '\n'){ break; }
00392                 if(caract != '\r'){ buffer += caract; } // Ignoramos o \r
00393
00394             }
00395             else if(n == 0){ break; } // Nada a ser lido
00396             else{
00397
00398                 throw std::runtime_error("\033[1;31mErro na leitura\033[0m");
00399             }
00400         }
00401
00402         return buffer;
00403     }
00404
00415     void
00416     loop(){
00417
00418         bool parsed = false; // Apenas uma flag para sabermos se houve interpretação
00419         while(
00420             is_exec
00421         ){
00422
00423             std::string mensagem = read_serial();
00424
00425             if(mensagem.empty()){ std::cout « "Nada a ser lido..." « std::endl; }
```

```
00426             else{
00427
00428                 std::cout « "Recebendo: " « mensagem « std::endl;
00429
00430                 if( mensagem.find("GGA") != std::string::npos ){
00431
00432                     last_data_given.parsing(0, split(mensagem));
00433                     parsed = true;
00434                 }
00435                 // ... para escalarmos novos padrões de mensagem
00436                 else{
00437
00438                 }
00439
00440                 if(
00441                     parsed
00442                 ){
00443
00444                     std::cout « "Interpretando: \033[7m"
00445                               « last_data_given.to_csv()
00446                               « "\033[0m"
00447                               « std::endl;
00448                     parsed = false;
00449                     printf("\n");
00450                 }
00451             }
00452
00453             std::this_thread::sleep_for(std::chrono::seconds(1));
00454         }
00455     }
00456
00457 public:
00458
00468     GPSTrack(
00469         const std::string&   ip_destino_,
00470         int                  porta_destino_,
00471         const std::string& porta_serial_
00472
00473     ) : ip_destino(ip_destino_),
00474         porta_destino(porta_destino_),
00475         porta_serial(porta_serial_)
00476     {
00477
00478         // As seguintes definições existentes para a comunição UDP.
00479         sockfd = ::socket(AF_INET, SOCK_DGRAM, 0);
00480         if( sockfd < 0 ){
00481             throw std::runtime_error("Erro ao criar socket UDP");
00482         }
00483
00484         addr_dest.sin_family = AF_INET;
00485         addr_dest.sin_port = ::htons(porta_destino);
00486         addr_dest.sin_addr.s_addr = ::inet_addr(ip_destino.c_str());
00487
00488         open_serial();
00489     }
00490
00503     ~GPSTrack() { stop(); if( sockfd >= 0 ){ ::close(sockfd); } if( fd_serial >= 0 ){
      ::close(fd_serial); } }
00504
00515     void
00516     init(){
00517
00518         if( is_exec.exchange(true) ){ return; }
00519
00520         std::cout « "\033[1;32mIniciando Thread de Leitura...\033[0m" « std::endl;
00521         worker = std::thread(
00522                                 [this]{ loop(); }
00523                              );
00524     }
00525
00534     void
00535     stop(){
00536
00537         if( !is_exec.exchange(false) ){ return; }
00538
00539         if(
00540             worker.joinable()
00541         ){
00542
00543             std::cout « "\033[1;32mSaindo da thread de leitura.\033[0m" « std::endl;
00544             worker.join();
00545         }
00546     }
00547 };
00548
00549 #endif // GPSTRACK_HPP
```

2.6 src/main.cpp File Reference

```
#include "GPSTrack.hpp"
```

Include dependency graph for main.cpp:



Functions

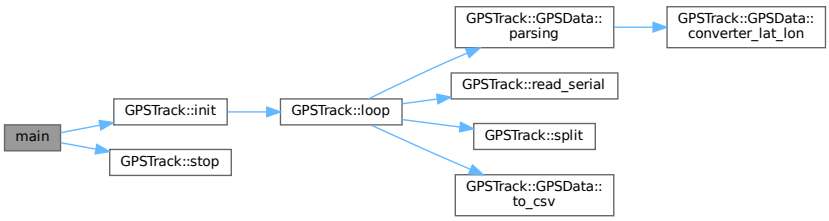
- int main ()

2.6.1 Function Documentation

2.6.1.1 main()

```
int main ( )
```

Here is the call graph for this function:



Index