RFIMS-CART

Generated by Doxygen 1.8.13

Contents

Chapter 1

RFIMS_CART

///ENGLISH///

This software is intended to run in the "RF Interference Measurement System (RFIMS)" which is going to be installed beside the China-Argentina Radio Telescope (CART) to analize the RF interference (RFI) which could reach the telescope, taking into account different azimuth angles and two antenna polarizations, horizontal and vertical.

It was designed to capture RF power measurements from a spectrum analyzer Aaronia Spectran HF-60105 V4 X, using an antenna which is mounted on a structure that allow the antenna to be rotated to point the horizon with different azimuth angles and whose polarization could be changed between horizontal and vertical. A sweep from 1 GHz (or maybe less) to 9.4 GHz is captured in each antenna position and then it is calibrated, processed to identify the RFI, saved into memory, plotted with the detected RFI and finally the measurements are sent to a remote server. The initial and stop frequencies are configurable, as many other parameters, through the files which are accesed by the software to load those parameters and which are inside the directories /home/pi/RFIMS-CART/.

To calibrate the measurements, at the beginning of each measurement cycle, which is the set of sweeps corresponding to a turning of 360 ° of the azimuth angle, a calibration of the RF front end is performed which consists in connecting a noise source (NS) at the input y an capturing two sweeps, one of them with the NS turned off and the other one with the NS turned on. With these sweeps, the curves of the front end parameters, total gain and total noise figure, versus frequency are estimated. Then, the sweeps captured with the antenna connected at the input are calibrated taking into account the estimated front end parameters, so that the distorsion produced by the front end beacuse of its no-flat frequency response, so that the power values represent the signals at the input and so that the internal noise of the front end which has been added to the measurements, to be removed.

The software has been designed to this particual purpose so this can only be run in a Raspberry Pi 3 board o later version with Raspbian Stretch or later, y the software will only work with the spectrum analyzer Aaronia Spectran HF-60105 V4 X and with the Aaronia GPS receiver with integrated sensors.

Before the installation of the software, it is necessary to install the following applications and libraries:

- Driver FTDI D2XX 2, versión 1.4.8 ARMv7 hard-float: `https://www.ftdichip.com/Drivers/D2XX.htm`
- Library libnmea: `http://nmea.io/`
- Library WiringPi 4, versión 2.46 o later: `http://wiringpi.com/`
- Software gnuplot 5 o later: `http://www.gnuplot.info/`
- The set of C++ libraries “boost” (`https://www.boost.org/`), versión 1.69.0, of which the following ones were used:
  - Library header-only Boost.Algorithm

- Library header-only Boost.Date_Time
- Library Boost.Filesystem
- Library Boost.System
- Library header-only Boost.Bimap
- Library Boost.Timer

To compile and install the software, a terminal must be opened, the current directory must be changed to the base directory of the project and the following commands must be run:

```
make all
sudo make copy-files
```

The first instruction compiles the software and it generates the binay file ./bin/rfims-cart. The second instruction copies the previous binary file to the path /usr/local/bin/, which is inside the environment variable PATH, so that the software could be run without writing the path where the binary file is; the python scritp ./scripts/client.py, which is used by the software to upload the data, is copied to the path /usr/local/; the file ./data/99-aaronia-spectran.rules is copied to the path /etc/udev/rules.d/, which allows a non-root user to run the software; and, finally, the directory tree ./data/RFIMS-CART/, which contains several files which are accesed by the software, are copied to the path /home/pi/. These directories and files are accesed by the software to load the configuration parameters and to save there the measurements. To ensure the software can read and write into the files which are in /home/pi/RFIMS-↩ CART/, it is necessary to run the following:

```
sudo chmod -r a+rw /home/pi/RFIMS-CART
```

To run the software, it must be typed "rfims-cart" in a terminal. The software has several arguments which define its behavior. To know the arguments and their usage it must be typed "rfims-cart --help" or "rims-cart -h".

To avoid interferences produced by the Raspberry Pi itself, it is very important to disable the Wi-Fi and Bluetooth interfaces, which is done editing the file /boot/config.txt.

To genearate/regenerate the software manual, you must run the following commands in a terminal:

```
cd doc
doxygen Doxyfile
cd latex
make pdf
```

After that, the linux scripts "Software_manual_pdf" and "Software_manual_html", which are in the folder doc/, will allow you to access the corresponding files.

//ESPAÑOL//

Este software está pensado para ser ejecutado en el "Sistema de Monitoreo de Interferencias de RF (RFIMS)" que será instalado junto al RadioTelescopio Chino-Argentino (CART) para analizar las interferencias de RF (RFI) que podrían alcanzar el telescopio, teniendo en cuenta diferentes ángulos azimutales y dos polarizaciones de la antena, horizontal y vertical.

El mismo está diseñado para capturar las mediciones de potencia de RF de un analizador de espectro Aaronia Spectran HF-60105 V4 X, usando una antena montada sobre una estructura que le permite rotar para apuntar al horizonte con diferentes ángulos azimutales y que le permite cambiar su polarización entre vertical y horizontal. En cada posición de la antena se captura un barrido desde 1 GHz (o quizás una menor frecuencia) hasta 9.4 GHz y luego el barrido es calibrado, procesado para identificar la RFI, se almacena en memoria no volátil, es graficado en la pantalla con la RFI detectada y, finalmente, todos los datos recolectados son enviados a un servidor remoto. Las frecuencias inicial y final son configurables, al igual que muchos otros parámetros, mediante los archivos a los cuales el software accede para levantar estos datos y que se ubican dentro de los directorios /home/pi/RFIMS-C↩ ART/.

Para calibrar las mediciones, al inicio de cada ciclo de medición, el conjunto de barridos que corresponden a un recorrido de 360° azimutal de la antena, se realiza una calibración del front end de RF que consiste en conectar un generador de ruido a la entrada y capturar dos barridos, uno con el generador apagado y otro con el generador encendido. Con estos barridos, se estiman las curvas de la ganancia total y la figura de ruido total del front end, ambas en función de la frecuencia. Luego, los barridos capturados con la antena son calibrados teniendo en cuenta los parameteros estimados anteriores, de modo que se elimine la distorsión introducida por el front end por su respuesta frecuencial no plana, las potencias estén referenciadas a la entrada del front end y de modo que se elimine el ruido interno del front end adicionado a las señales de entrada.

El software está diseñado para esta aplicación particular por lo que solo puede executarse en una placa Raspberry Pi 3 o superior con Raspbian Stretch o superior, y solo funcionará con el analizador de espectro Aaronia Spectran HF-60105 V4 X y con el receptor GPS con sensores integrados de Aaronia.

Antes de instalar este software, es necesario instalar las siguientes aplicaciones y bibliotecas:

- Driver FTDI D2XX 2, la versión 1.4.8 ARMv7 hard-float: https://www.ftdichip.com/Drivers/↩ D2XX.htm
- Biblioteca libnmea: http://nmea.io/
- Biblioteca WiringPi 4, versión 2.46 o superior: http://wiringpi.com/
- Software gnuplot 5 o superior: http://www.gnuplot.info/
- El paquete de bibliotecas de C++ “boost” (https://www.boost.org/), versión 1.69.0, de las que se utilizaron las siguientes:
  - Biblioteca header-only Boost.Algorithm
  - Biblioteca header-only Boost.Date_Time
  - Biblioteca Boost.Filesystem
  - Biblioteca Boost.System
  - Biblioteca header-only Boost.Bimap
  - Biblioteca Boost.Timer

Para compilar e instalar el software se debe abrir una terminal, ubicarse sobre el directorio base del proyecto y ejecutar los siguientes comandos:

```
make all
sudo make copy-files
```

Con la primer instrucción se compila el programa y se genera el binario ./bin/rfims-cart. Con la segunda instrucción se copia el binario anterior a la ruta /usr/local/bin/, que está dentro de la variable de entorno PATH, para que se puede ejecutar el mismo sin escribir la ruta donde se encuentra; se copia el script de python ./scripts/client.py, que es usado por el programa para enviar los datos al servidor remoto, a la ruta /usr/local/; se copia el archivo con udev rules ./data/99-aaronia-spectran.rules a la ruta /etc/udev/rules.d/, que permite que un usuario no root pueda ejecutar el software; y, por último, se copia el árbol de directorios con archivos ./data/RFIMS-CART/ a /home/pi/. Estos directorios y archivos son utilizados por el programa para cargar los parámetros de configuración y para almacenar las mediciones y datos capturados. Para asegurarse de que el software pueda escribir en los archivos correspondientes ubicados en /home/pi/RFIMS-CART es necesario modificar los permisos de este arbol de directorios, con el siguiente comando:

```
sudo chmod -R a+rw /home/pi/RFIMS-CART
```

Para ejecutar el programa se debe tipear "rfims-cart" en la terminal. El programa tiene multiples argumentos que permiten modificar su comportamiento. Para conocer los argumentos y cómo deben usarse, se debe tipear "rfims-cart --help" o "rfims-cart -h".

Para evitar interferencias producidas por la misma placa Raspberry Pi, resulta trascendental desactivar las interfaces Wi-Fi y Bluetooth, lo cual se realiza modificando el archivo /boot/config.txt.

Para generar/regenerar el manual del software, ejecutar en una terminal los siguientes comandos:

```
cd doc
doxygen Doxyfile
cd latex
make pdf
```

Luego, los scripts de linux "Software_manual_pdf" y "Software_manual_html" de la carpeta doc/ permitirán abrir los archivos correspondientes.

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

AntennaPositioner
The aim of the class *AntennaPositioner* is to handle the antenna positioning system . . . . . . **??**

BandParameters
This structure is intended to store the parameters which are used to configure the spectrum analyzer in each frequency band . . . . **??**

Command
This class builds the corresponding bytes array to send a certain command to a Aaronia Spectran V4 series spectrum analyzer . . . . **??**

CurveAdjuster
The aim of the class *CurveAdjuster* is to adjust any frequency curve, this is to interpolate and/or extrapolate the curve of a given parameter versus frequency . . . . **??**

Data3D
A structure intended to save the the values of the 3d sensors which are integrated in the GPS receiver . . . . **??**

DataLogger
The class *DataLogger* is intended to handle the storing of the generated data into memory, following the CSV (comma-separated values) format . . . . **??**

SpectranConfigurator::FixedParameters
This structure saves the fixed parameters of the spectrum analyzer, i.e. the parameters which do not change through the entire measurement cycle . . . . **??**

FloatToBytes
An union which is used to split a `float` value in its 4 bytes . . . . **??**

FreqValues
The aim of this structure is to store the curve of a determined parameter or variable versus the frequency, which is named a frequency curve here . . . . **??**

FrontEndCalibrator
The aim of this class is to calculate the total gain and total noise figure curves versus frequency of the RF front end . . . . **??**

GPSCoordinates
A structure which saves the GPS coordinates . . . . **??**

GPSInterface
The class *GPSInterace* is intended to establish the communication with the Aaronia GPS receiver, to request and capture messages from this one and extract useful data from the messages **??**

Reply
The class *Reply* is intended to receive a bytes vector sent by the spectrum analyzer and to extract its information . . . . **??**

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Class Documentation

5.1 AntennaPositioner Class Reference

The aim of the class *AntennaPositioner* is to handle the antenna positioning system.

`#include <AntennaPositioning.h>`

Public Member Functions

- AntennaPositioner (GPSInterface &gpsInterf)

  The unique constructor of the class AntennaPositioner.
- ∼AntennaPositioner ()

  The class destructor.
- void SetNumOfAzimPos (unsigned int number)

  This method allows to set the number of azimuth positions.
- bool Initialize ()

  This method performs the initialization of the antenna positioning system.
- bool NextAzimPosition ()

  This method moves the antenna to the next azimuth position.
- bool ChangePolarization ()

  This method change the antenna polarization.
- float GetAzimPosition () const

  This method returns the current antenna azimuth angle.
- std::string GetPolarizationString () const

  This method returns the current antenna polarization, as a `std::string` object.
- Polarization GetPolarization () const

  This method returns the current antenna polarization, as a value of the enumeration Polarization.
- int GetPositionIndex () const

  This method returns the current azimuth position index.
- bool IsLastPosition () const

  This method return the total number of azimuth positions.

Friends

- void **canalA** ()
- void **canalB** ()

5.1.1 Detailed Description

The aim of the class *AntennaPositioner* is to handle the antenna positioning system.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 AntennaPositioner()

```
AntennaPositioner::AntennaPositioner (
            GPSInterface & gpsInterf )
```

The unique constructor of the class *AntennaPositioner*.

Parameters

gpsInterf	A reference to the object which is responsible for the communication with the Aaronia GPS receiver.

5.1.2.2 ∼AntennaPositioner()

```
AntennaPositioner::∼AntennaPositioner ( ) [inline]
```

The class destructor.

Its implementation is empty because the attributes are implicitly destroyed. However, the destructor is defined here to allow this one to be called explicitly in any part of the code, what is used by the signals handler to destroy the objects when a signal to finish the execution of the software is received.

5.1.3 Member Function Documentation

5.1.3.1 ChangePolarization()

```
bool AntennaPositioner::ChangePolarization ( )
```

This method change the antenna polarization.

If the current polarization is horizontal, then it is changed to vertical, and vice versa.

Returns

A `true` if the operation was successful or a `false` otherwise.

5.1.3.2 GetPolarization()

```
Polarization AntennaPositioner::GetPolarization ( ) const
```

This method returns the current antenna polarization, as a value of the enumeration Polarization.

Returns

A value of the enumeration 'Polarization'.

5.1.3.3 GetPolarizationString()

```
std::string AntennaPositioner::GetPolarizationString ( ) const
```

This method returns the current antenna polarization, as a `std::string` object.

Returns

A `std::string` object with the current antenna polarization.

5.1.3.4 Initialize()

```
bool AntennaPositioner::Initialize ( )
```

This method performs the initialization of the antenna positioning system.

This initialization implies to move the antenna to its initial position, to capture the initial azimuth angle and to ensure the antenna polarization is horizontal.

Returns

A `true` if the initialization was successful or a `false` otherwise.

5.1.3.5 IsLastPosition()

```
bool AntennaPositioner::IsLastPosition ( ) const [inline]
```

This method return the total number of azimuth positions.

This method states if the current position is the last one.

5.1.3.6 NextAzimPosition()

```
bool AntennaPositioner::NextAzimPosition ( )
```

This method moves the antenna to the next azimuth position.

To move the antenna to the next position, this is rotated an angle determined by the number of positions (360/number of positions) and in a clockwise way (seen from above). If the current position is the last one, then the antenna is move to the initial position, rotating this one counterclockwise (seen from above) to avoid the cables to tangle or stretch.

Taking into account the method Initialize(), it is waited this method to be called when the antenna polarization changes from vertical to horizontal.

Returns

A `true` if the operation was successful or a `false` otherwise.

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

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/AntennaPositioning.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/AntennaPositioner.cpp

5.2 BandParameters Struct Reference

This structure is intended to store the parameters which are used to configure the spectrum analyzer in each frequency band.

```
#include <Basics.h>
```

Public Attributes

- unsigned int bandNumber

  This is an integer number which identifies the frequency band (like an index).
- bool flagEnable

  This parameter determines if the band is used or not.
- float startFreq

  Initial frequency (Fstart) in Hz.
- float stopFreq

  Final frequency (Fstop) in Hz.
- float rbw

  Resolution Bandwidth (RBW) in Hz.
- float vbw

  Video Bandwidth (VBW) in Hz.
- unsigned int sweepTime

  Time to sweep the given span, expressed in ms.
- bool flagDefaultSamplePoints

  This parameter determines if the sample points number must be configured with user-defined number or if it is left with its default value which is determined by the Spectran device.
- unsigned int samplePoints

  Number of samples points. This value can be determined by the Spectran device (default value) or it can be a forced value.
- unsigned int detector

  Display detector: ”RMS” takes the sample as the root mean square of the values present in the bucket, or “Min/Max” takes two samples as the minimum and maximum peaks in the bucket.

5.2.1 Detailed Description

This structure is intended to store the parameters which are used to configure the spectrum analyzer in each frequency band.

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

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Basics.h

5.3 Command Class Reference

This class builds the corresponding bytes array to send a certain command to a Aaronia Spectran V4 series spectrum analyzer.

```
#include <Spectran.h>
```

Public Types

- enum CommandType : char {
VERIFY =0x01, **LOGOUT**, **GETSTPVAR** =0x20, **SETSTPVAR**,
UNINITIALIZED }

    An enumeration which contains the command types which can be sent to a spectrum analyzer Aaronia Spectran HF-60105 V4 X.

Public Member Functions

- Command ()

    The default constructor.
- Command (const CommandType commType, const SpecVariable variable=SpecVariable::UNINITIALIZED, const float val=0.0)

    The most complete constructor which allows to set the internal pointers and optionally the command type.
- Command (const Command &anotherComm)

    The copy constructor.
- void SetAs (const CommandType commType, const SpecVariable variable=SpecVariable::UNINITIALIZED, const float val=0.0)

    This method is intended to provide to the object the enough data so this can configure itself to be ready to be sent.
- void SetParameters (const SpecVariable variable, const float val=0.0)

    This method is intended to set the command's parameters, so it should be used when the command type has already been set.
- CommandType GetCommandType () const

    A method to get the current command type as a value of the enumeration CommandType.
- std::string GetCommTypeString () const

    A method which returns the command type as a `std::string`.
- SpecVariable GetVariableName () const

    A method to get the variable name which is going to be modified or read, as a value of the enumeration SpecVariable.
- std::string GetVariableNameString () const

    A method which returns the name of the Spectran's variable which is related with the command (GETSTPVAR and SETSTPVAR commands) as a `std::string`.
- float GetValue () const

A method to get the value which is going to be or has been used to modify a variable.

- const std::vector< std::uint8_t > & GetBytesVector () const

  A method to obtain the bytes vector like this is implemented internally, a `std::vector` container.

- const std::uint8_t * GetBytesPointer () const

  A method to obtain the bytes vector but like a C-style array (`std::uint8_t*`).

- unsigned int GetNumOfBytes () const

  A method which returns the size of the bytes vector.

- void Clear ()

  This method allows to reset the object, cleaning the bytes vector, command type, variable name and value.

- const Command & operator= (const Command &command)

  An overloading of the assignment operator, adapted for this class.

5.3.1 Detailed Description

This class builds the corresponding bytes array to send a certain command to a Aaronia Spectran V4 series spectrum analyzer.

The *Command* class is intended to build the bytes array which will be sent a spectrum analyzer to perform one of the following tasks:

- Initialize the communication with the device.
- Set an environment variable.
- Get the value of an environment variable.
- Enable/disable the streaming of sweep points.
- Close the communication. The user have to say to the object which "command type" he wants, which "variable" he wants to write/read and which value must be used to set up the specified variable. When the object has the enough data, it will build the bytes array which will be sent to the spectrum analyzer via the USB interface. The objects of this class are interchanged between the *Spectran configurator* and the *Spectran interface*.

5.3.2 Member Enumeration Documentation

5.3.2.1 CommandType

```
enum Command::CommandType : char
```

An enumeration which contains the command types which can be sent to a spectrum analyzer Aaronia Spectran HF-60105 V4 X.

This enumeration contains just four commands from the Spectran USB Protocol: *VERIFY*, *LOGOUT*, *GETSTPVAR* and *SETSTPVAR*. There are other possible commands which are intended to modify or get information about the internal files of the spectrum analyzer, but they are not added because they will not be used. There is an extra command type which is *UNINITIALIZED* whose purpose is to state that the object is still incomplete.

5.3.3 Constructor & Destructor Documentation

5.3.3.1 Command() [1/3]

```
Command::Command ( )
```

The default constructor.

When this constructor is used, the programmer must provide the command data with the method `SetAs(commType,variable,value)`, and even with the method `SetParameters(variable,value)`.

5.3.3.2 Command() [2/3]

```
Command::Command (
            const CommandType type,
            const SpecVariable variable = SpecVariable::UNINITIALIZED,
            const float val = 0.0 )
```

The most complete constructor which allows to set the internal pointers and optionally the command type.

If this constructor is used just providing the first parameter, *command type*, then the method `SetParameters()` should be used to set the variable name (for *GETSTPVAR* and *SETSTPVAR* commands) and the value that must be used to write it (just for *SETSTPVAR* commands).

Parameters

in	*type*	The command type: VERIFY, LOGOUT, GETSTPVAR or SETSTPVAR.
in	*variable*	An optional argument which determines which variable will be read or written.
in	*val*	An optional argument which represents the value which will be used to write the given variable.

5.3.3.3 Command() [3/3]

```
Command::Command (
            const Command & anotherComm )
```

The copy constructor.

This method takes an object of the same class as argument, and it copies its attributes.

Parameters

in	*anotherComm*	A *Command* object given to copy its attributes.

5.3.4 Member Function Documentation

5.3.4.1 GetBytesPointer()

```
const std::uint8_t* Command::GetBytesPointer ( ) const  [inline]
```

A method to obtain the bytes vector but like a C-style array (`std::uint8_t*`).

This method returns a pointer to `std::uint8_t` so this allows to access directly to the memory addresses where the vector's bytes are stored. The *Spectran Interface* object uses this method because it works internally with C-style arrays.

5.3.4.2 operator=()

```
const Command & Command::operator= (
             const Command & anotherComm )
```

An overloading of the assignment operator, adapted for this class.

Parameters

in	*anotherComm*	A *Command* object given to copy its attributes.

5.3.4.3 SetAs()

```
void Command::SetAs (
             const CommandType commType,
             const SpecVariable variable = SpecVariable::UNINITIALIZED,
             const float val = 0.0 )
```

This method is intended to provide to the object the enough data so this can configure itself to be ready to be sent.

Parameters

in	*commType*	The command type: VERIFY, LOGOUT, GETSTPVAR or SETSTPVAR.
in	*variable*	An optional argument which determines which variable will be read or written.
in	*val*	An optional argument which represents the value which will be used to write the given variable.

5.3.4.4 SetParameters()

```
void Command::SetParameters (
          const SpecVariable variable,
          const float val = 0.0 )
```

This method is intended to set the command's parameters, so it should be used when the command type has already been set.

Parameters

in	*variable*	The variable which will be read or written, with the commands GETSTPVAR and SETSTPVAR.
in	*val*	An optional argument which determines the value which will be used to write the given variable.

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

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Spectran.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Command.cpp

5.4 CurveAdjuster Class Reference

The aim of the class *CurveAdjuster* is to adjust any frequency curve, this is to interpolate and/or extrapolate the curve of a given parameter versus frequency.

```
#include <SweepProcessing.h>
```

Public Member Functions

- CurveAdjuster ()

  The unique constructor of the class.
- ∼CurveAdjuster ()

  The destructor of the class.
- void SetBandsParameters (const std::vector< BandParameters > &bandsParam)

  This method allows to give the bands' parameters to the object.
- void SetRefSweep (const Sweep &swp)

  This method allow to give the reference sweep to the object.
- const FreqValues & AdjustCurve (const FreqValues &curve)

  This is the central method which allows to adjust a frequency curve.
- const FreqValues & GetAdjustedCurve () const

  This method returns the last adjusted curve.

5.4.1 Detailed Description

The aim of the class *CurveAdjuster* is to adjust any frequency curve, this is to interpolate and/or extrapolate the curve of a given parameter versus frequency.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 ∼CurveAdjuster()

```
CurveAdjuster::∼CurveAdjuster ( )  [inline]
```

The destructor of the class.

Its implementation is empty because the attributes destruction is implicitly. However, the destructor is defined here to allow this one to be called explicitly in any part of the code, what is used by the signals handler to destroy the objects when a signal to finish the execution of the software is received.

5.4.3 Member Function Documentation

5.4.3.1 AdjustCurve()

```
const FreqValues & CurveAdjuster::AdjustCurve (
            const FreqValues & curve )
```

This is the central method which allows to adjust a frequency curve.

This method takes the set of linear functions, which model the given in the frequency domain, and it generates the adjusted curve evaluating each linear function, in its range, taking into account the frequency values of the reference sweep.

The frequency curve must be a *FreqValues* structure or any of the structure derived from that one.

Parameters

in	*curve*	The frequency curve to be adjusted.

5.4.3.2 SetBandsParameters()

```
void CurveAdjuster::SetBandsParameters (
            const std::vector< BandParameters > & bandsParam )  [inline]
```

This method allows to give the bands' parameters to the object.

The bands' parameters are taken into account to perform the adjusting of a curve.

Parameters

in	*bandsParam*	The frequency bands' parameters.

5.4.3.3 SetRefSweep()

```
void CurveAdjuster::SetRefSweep (
            const Sweep & swp )  [inline]
```

This method allow to give the reference sweep to the object.

The reference sweep is used to know which are the exact frequency values which are delivered by the spectrum analyzer. This allows to correctly perform the curve adjusting.

Parameters

in	*swp*	The reference sweep.

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

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/SweepProcessing.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/CurveAdjuster.cpp

5.5 Data3D Struct Reference

A structure intended to save the the values of the 3d sensors which are integrated in the GPS receiver.

```
#include <AntennaPositioning.h>
```

Public Types

- enum SensorType : char { **GYROSCOPE**, **COMPASS**, **ACCELEROMETER**, **UNINITIALIZED** }

  An enumeration with the names of the sensors integrated in the GPS receiver.

Public Attributes

- SensorType sensor

  The specific sensor whose values are represented in the structure.
- std::string time

  The time when the measurement were made, in the format HHMMSS.NNN, where NNN is a GPS's internal counter that is reset when the time is updated by a new GPS data.
- double x

  The x-axis value.
- double y

  The y-axis value.
- double z

  The z-axis value.

5.5.1 Detailed Description

A structure intended to save the the values of the 3d sensors which are integrated in the GPS receiver.

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

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/AntennaPositioning.h

5.6 DataLogger Class Reference

The class *DataLogger* is intended to handle the storing of the generated data into memory, following the CSV (comma-separated values) format.

```
#include <SweepProcessing.h>
```

Public Types

- enum **TimestampSource** { **SWEEP**, **FRONTENDPARAM** }

Public Member Functions

- DataLogger ()

  The unique class constructor.
- ∼DataLogger ()

  The class destructor.
- void SetNumOfSweeps (unsigned int number)

  A method which allows to set the number of sweeps of a measurement cycle (the two calibration sweeps are not considered), i.e. the number of sweeps which must be store in the same file.
- void SetFilenameTimestampSrc (TimestampSource source)

  A method which allows to define the timestamp source which will be used for the name of the file where the sweeps will be stored.
- void SaveBandsParamAsCSV (const std::vector< BandParameters > &bandsParamVector)

  This method is intended to save the bands parameters in a CSV file which is more adequately to be read in the remote server.
- void SaveFrontEndParam (const FreqValues &gain, const FreqValues &noiseFigure)

  This method is intended to save the estimated front end parameters, gain and noise figure, into the non-volatile memory.
- void SaveSweep (const Sweep &sweep)

  This method is intended to save a calibrated sweep into the non-volatile memory.
- void SaveRFI (const RFI &rfi)

  This method is intended to save the detected RFI in the last sweep, into the non-volatile memory.
- void DeleteOldFiles () const

  The aim of this method is to delete the old files.
- void ArchiveAndCompress ()

  The aim of this method is to prepare, archive and compress, the files which will be send to the remote server.
- void UploadData ()

  This method is responsible for the uploading of the data files to the remote server.
- void PrepareAndUploadData ()

  This method creates a thread where the data files will be uploaded, in parallel with the capture of a new sweep.

Friends

- void ∗ UploadThreadFunc (void ∗)

  The function which is executed by the thread which is responsible for the concurrent uploading of the data files.

5.6.1 Detailed Description

The class *DataLogger* is intended to handle the storing of the generated data into memory, following the CSV (comma-separated values) format.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 DataLogger()

```
DataLogger::DataLogger ( )
```

The unique class constructor.

The constructor initializes all the internal attributes, checks if the corresponding folders exist and if any folder does not exist then it is created. Also, it checks if there is a shell available to be able to execute the external python script "client.py" to upload the data.

5.6.3 Member Function Documentation

5.6.3.1 ArchiveAndCompress()

```
void DataLogger::ArchiveAndCompress ( )
```

The aim of this method is to prepare, archive and compress, the files which will be send to the remote server.

To archive the data files the utility 'tar' is used and thr utility 'lzma' is used to compress to resulting archive file. The resulting compressed archive file is name as "rfims_data_DD-MM-YYYYTHH:MM:SS.tar.lzma", where the last part, before the extension is the timestamp of the corresponding measurement cycle.

5.6.3.2 PrepareAndUploadData()

```
void DataLogger::PrepareAndUploadData ( )
```

This method creates a thread where the data files will be uploaded, in parallel with the capture of a new sweep.

This method creates a thread where the methods `ArchiveAndCompress()` and `UploadData()` are called. After the creation of the thread, the method ends and the main thread can continues with the next operations, like the moving of the antenna, the capture of a new sweep, etc. The next time this method is called, it will control if the last thread has finished, if not, the method will wait to the thread to finish, and then it will create a new thread for the uploading of the new data.

5.6.3.3 SaveBandsParamAsCSV()

```
void DataLogger::SaveBandsParamAsCSV (
            const std::vector< BandParameters > & bandsParamVector )
```

This method is intended to save the bands parameters in a CSV file which is more adequately to be read in the remote server.

This method should be called each time the bands' parameters are reloaded (or loaded by first time), because of the file BASE_PATH/parameters/freqbands.txt was modified, to update the file BASE_PATH/parameters/csv/freqbands.csv to that has the same parameters, just in a different format. The CSV file is generated because it is easier the bands' parameters to be loaded from a file with CSV format than from a file with a more human-readable format like freqbands.txt. Each time this method is called the file freqbands.csv is regenerated. This file is then incorporated in the archive file, in the method ArchiveAndCompress().

Parameters

in	*bandsParamVector*	A vector with the parameters of all frequency bands.

5.6.3.4 SaveFrontEndParam()

```
void DataLogger::SaveFrontEndParam (
            const FreqValues & gain,
            const FreqValues & noiseFigure )
```

This method is intended to save the estimated front end parameters, gain and noise figure, into the non-volatile memory.

The given front end parameters are saved in two different files:

- BASE_PATH/calibration/frontendparam/gain_DD-MM-YYYYTHH:MM:SS.csv
- BASE_PATH/calibration/frontendparam/noisefigure_DD-MM-YYYYTHH:MM:SS.csv where DD-MM-YYYYTHH:MM:SS is the timestamp of the current measurement cycle.

Those files are then incorporated into the archive file which will be uploaded at the end of the measurement cycle. If the front end parameters were not estimated in a measurement cycle, then the default front end parameters are used, which are curves that were estimated in the laboratory and which are saved in the following files:

- BASE_PATH/calibration/frontendparam/default/gain_default.csv
- BASE_PATH/calibration/frontendparam/default/noisefigure_default.csv In that case, these files are incorporated into the archive file to be uploaded.

Parameters

in	*gain*	A structure with the estimated values of the total front end gain versus the frequency.
in	*noiseFigure*	A structure with the estimated values of the total front end noise figure versus the frequency.

5.6.3.5 SaveRFI()

```
void DataLogger::SaveRFI (
             const RFI & rfi )
```

This method is intended to save the detected RFI in the last sweep, into the non-volatile memory.

Each given structure with the RFI detected in the last sweep is saved in a different file with the filename format RF↵I_x.csv, where 'x' is an integer number between 1 and 2∗(number of azimuth positions). So each file corresponds to a sweep of the measurement cycle and all the files of determined measurement cycle are in the same folder, BA↵SE_PATH/measurement/RFI_DD-MM-YYYYTHH:MM:SS/ where the last part of the folder's name is the timestamp of the measurement cycle.

Parameters

in	*rfi*	A structure with RFI to be saved.

5.6.3.6 SaveSweep()

```
void DataLogger::SaveSweep (
             const Sweep & sweep )
```

This method is intended to save a calibrated sweep into the non-volatile memory.

The given sweep is saved in BASE_PATH/measurements/ with the filename format "sweep_DD-MM-YYYYTHH:↵MM:SS.csv" where the last part is the timestamp of the measurement cycle, which correspond to the beginning of this one.

All sweeps which corresponds to the same measurement cycle are saved in the same file and the method automatically create a new file or reopen the corresponding file each time a new sweep must be saved.

Parameters

in	*sweep*	A structure with the sweep to be saved.

5.6.3.7 UploadData()

```
void DataLogger::UploadData ( )
```

This method is responsible for the uploading of the data files to the remote server.

To upload the files the script /usr/local/client.py is called. This script try to send the archive file many times through one hour, taking into account the possibility that there is no Internet connection in the first try. If the script achieves the sending, then it wakes up the remote server to that one to read the files, and finally the script removes the local archive file. If the script ends with errors, the file is not deleted and remains in a queue waiting to be send. The idea is the uploading to perform at the end of each measurement cycle.

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

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/SweepProcessing.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/DataLogger.cpp

5.7 SpectranConfigurator::FixedParameters Struct Reference

This structure saves the fixed parameters of the spectrum analyzer, i.e. the parameters which do not change through the entire measurement cycle.

```
#include <Spectran.h>
```

Public Attributes

- int attenFactor

  Attenuator factor [dB]: -10=auto, 0=off or a value between 1 to 30.
- unsigned int displayUnit

  Measurement unit: 0=dBm (power), 1=dBuV (voltage), 2=V/m (electric field strength) or 3=A/m (magnetic field strength).
- const unsigned int demodMode =0

  Demodulator mode: 0=off, 1=am or 2=fm.
- unsigned int antennaType

  Antenna type: 0=HL7025, 1=HL7040, 2=HL7060, 3=HL6080 or 4=H60100.
- int cableType

  Cable type: -1 is none, 0 is “1m standard cable”.
- const unsigned int recvConf =0

  Receiver configuration: 0=spectrum, 1=broadband.
- bool internPreamp

  Internal preamplifier enabling: 0=off, 1=on.
- bool sweepDelayAcc

  Sweep delay for accuracy mode: 1=enable, 0=disable.
- const bool peakLevelAudioTone =0

  Peak Level Audio tone: 0=disable, 1=enable.
- const bool backBBDetector =0

  Background Broadband detector: 0=disable, 1=enable.
- float speakerVol

  Speaker volume: range from 0.0 to 1.0.

5.7.1 Detailed Description

This structure saves the fixed parameters of the spectrum analyzer, i.e. the parameters which do not change through the entire measurement cycle.

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

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Spectran.h

5.8 FloatToBytes Union Reference

An union which is used to split a `float` value in its 4 bytes.

`#include <Spectran.h>`

Public Attributes

- float floatValue

  The `float` value which must be split.
- std::uint8_t bytes [4]

  The array with the 4 bytes of the inserted `float` value.

5.8.1 Detailed Description

An union which is used to split a `float` value in its 4 bytes.

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

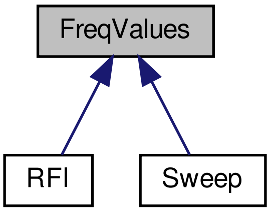
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Spectran.h

5.9 FreqValues Struct Reference

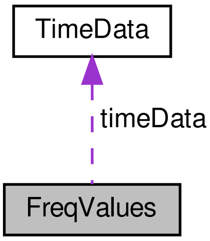
The aim of this structure is to store the curve of a determined parameter or variable versus the frequency, which is named a frequency curve here.

`#include <Basics.h>`

Inheritance diagram for FreqValues:



Collaboration diagram for FreqValues:



Public Types

- typedef float **value_type**

Public Member Functions

- FreqValues (const std::string &typ="unknown")

  The default constructor which can receive the curve type.
- FreqValues (const FreqValues &freqValues)

  The copy constructor.
- virtual ∼FreqValues ()

  This is the structure's destructor which is virtual because there are structures derived from this structure.
- bool PushBack (const FreqValues &freqValues)

  This method is intended to insert one data point (frequency,value) or a set of data points in the structure, at the end.
- virtual void Clear ()

  This method is intended to clear the structure, i.e. to delete all its data points.
- bool Empty () const

  This method allows to know if the structure is empty, i.e. it has no data points.
- const FreqValues & operator= (const FreqValues &freqValues)

  An overloading of the assignment operator adapted for this structure.
- const FreqValues & operator+= (const FreqValues &rhs)

  An overloading of the operator += adapted for this structure.
- value_type MeanValue () const

  The aim of this method is to offer the mean value of all data points, i.e. it calculates the average.

Public Attributes

- std::string type

  Type of frequency values: ”sweep”, “frequency response”, “calibration curve”, “threshold curve”, “rfi", etc.
- std::vector< value_type > values

  RF power values (dBm), gain values (dB or dBi), noise figure values (dB), etc.
- std::vector< std::uint_least64_t > frequencies

  Frequency values in Hz.
- TimeData timeData

Friends

- FreqValues operator- (const FreqValues &argument)

  An overloading of the unary operator - which negates a FreqValues object.
- FreqValues operator+ (const FreqValues &lhs, const FreqValues &rhs)

  An overloading of operator + which calculates the sum of two objects of structure FreqValues.
- FreqValues operator+ (const FreqValues &lhs, const double rhs)

  An overloading of operator + which calculates the sum of a FreqValues object and a double value, in that order.
- FreqValues operator+ (const FreqValues &lhs, const float rhs)

  An overloading of operator + which calculates the sum of a FreqValues object and a float value, in that order.
- FreqValues operator+ (const double lhs, const FreqValues &rhs)

  An overloading of operator + which calculates the sum of a double value and a FreqValues object, in that order.
- FreqValues operator+ (const float lhs, const FreqValues &rhs)

  An overloading of operator + which calculates the sum of a float value and a FreqValues object, in that order.
- FreqValues operator- (const FreqValues &lhs, const FreqValues &rhs)

  An overloading of operator - which calculates the subtraction of two objects of structure FreqValues.
- FreqValues operator- (const FreqValues &lhs, const double rhs)

  An overloading of operator - which calculates the subtraction of a FreqValues object and a double value, in that order.
- FreqValues operator- (const FreqValues &lhs, const float rhs)

  An overloading of operator - which calculates the subtraction of a FreqValues object and a float value, in that order.
- FreqValues operator- (const double lhs, const FreqValues &rhs)

  An overloading of operator - which calculates the subtraction of a double value and a FreqValues object, in that order.
- FreqValues operator- (const float lhs, const FreqValues &rhs)

  An overloading of operator - which calculates the subtraction of a float value and a FreqValues object, in that order.
- FreqValues operator∗ (const FreqValues &lhs, const FreqValues &rhs)

  An overloading of operator ∗ which multiplies two objects of structure FreqValues.
- FreqValues operator∗ (const FreqValues &lhs, const double rhs)

  An overloading of operator ∗ which multiplies a FreqValues object and a double value, in that order.
- FreqValues operator∗ (const FreqValues &lhs, const float rhs)

  An overloading of operator ∗ which multiplies a FreqValues object and a float value, in that order.
- FreqValues operator∗ (const double lhs, const FreqValues &rhs)

  An overloading of operator ∗ which multiplies a double value and a FreqValues object, in that order.
- FreqValues operator∗ (const float lhs, const FreqValues &rhs)

  An overloading of operator ∗ which multiplies a float value and a FreqValues object, in that order.
- FreqValues operator/ (const FreqValues &lhs, const FreqValues &rhs)

  An overloading of operator / which calculates the division between two objects of structure FreqValues.
- FreqValues operator/ (const FreqValues &lhs, const double rhs)

  An overloading of operator / which calculates the division between a FreqValues object and a double value, in that order.
- FreqValues operator/ (const FreqValues &lhs, const float rhs)

  An overloading of operator / which calculates the division between a FreqValues object and a float value, in that order.
- FreqValues operator/ (const double lhs, const FreqValues &rhs)

  An overloading of operator / which calculates the division between a double value and a FreqValues object, in that order.
- FreqValues operator/ (const float lhs, const FreqValues &rhs)

  An overloading of operator / which calculates the division between a float value and a FreqValues object, in that order.
- FreqValues log10 (const FreqValues &argument)

  An overloading of function `log10()`, decimal logarithm, adapted to receive an argument of type FreqValues.
- FreqValues pow (const FreqValues &base, const double exponent)

  An overloading of function `pow()`, power function, adapted to receive an argument of type FreqValues as base and an argument of type double as exponent.
- FreqValues pow (const FreqValues &base, const float exponent)

An overloading of function `pow()`, power function, adapted to receive an argument of type FreqValues as base and an argument of type float as exponent.

- FreqValues pow (const double base, const FreqValues &exponent)

  An overloading of function `pow()`, exponentiation, adapted to receive an argument of type double as base and an argument of type FreqValues as exponent.

- FreqValues pow (const float base, const FreqValues &exponent)

  An overloading of function `pow()`, exponentiation, adapted to receive an argument of type float as base and an argument of type FreqValues as exponent.

5.9.1 Detailed Description

The aim of this structure is to store the curve of a determined parameter or variable versus the frequency, which is named a frequency curve here.

5.9.2 Constructor & Destructor Documentation

5.9.2.1 FreqValues() [1/2]

```
FreqValues::FreqValues (
          const std::string & typ = "unknown" )  [inline]
```

The default constructor which can receive the curve type.

Parameters

in	*typ*	The type of parameter whose curve of values versus frequency is stored in the structure.

5.9.2.2 FreqValues() [2/2]

```
FreqValues::FreqValues (
          const FreqValues & freqValues )  [inline]
```

The copy constructor.

Parameters

in	*freqValues*	Another *FreqValues* structure which is given to copy its attributes.

5.9.3 Member Function Documentation

5.9.3.1 operator+=()

```
const FreqValues & FreqValues::operator+= (
             const FreqValues & rhs )
```

An overloading of the operator += adapted for this structure.

This method is not intended to concatenate two *FreqValues* structure, what is performed by the method `PushBack()`, but its aim is to sum each point of base structure with the corresponding point of the structure given as argument, and to save the results in the base structure.

The method returns a `const` reference to the base structure.

Parameters

in	*rhs*	A *FreqValues* structure which is given to perform the sum.

5.9.3.2 operator=()

```
const FreqValues & FreqValues::operator= (
             const FreqValues & freqValues )
```

An overloading of the assignment operator adapted for this structure.

The method returns a `const` reference to the base structure.

Parameters

in	*freqValues*	Another *FreqValues* structure which is given to copy its attributes.

5.9.3.3 PushBack()

```
bool FreqValues::PushBack (
             const FreqValues & freqValues )
```

This method is intended to insert one data point (frequency,value) or a set of data points in the structure, at the end.

Parameters

in	*freqValues*	Another *FreqValues* structure whose values must be inserted at the end of this structure.

5.9.4 Friends And Related Function Documentation

5.9.4.1 log10

```
FreqValues log10 (
            const FreqValues & argument )  [friend]
```

An overloading of function `log10()`, decimal logarithm, adapted to receive an argument of type *FreqValues*.

This function takes each point of the given structure, apply the decimal logarithm whit its value and stores the result in a different *FreqValues* structure, which is then returned. The rest of attributes (frequency, type, timestamp, etc.) are copied as-is to the object to be returned.

Parameters

in	*argument*	The *FreqValues* structure to be used as argument to the decimal logarithm operation.

5.9.4.2 operator* [1/3]

```
FreqValues operator* (
            const FreqValues & lhs,
            const FreqValues & rhs )  [friend]
```

An overloading of operator * which multiplies two objects of structure *FreqValues*.

Before performing the operation, the function checks if the frequencies of each structure match and if the "values" vectors have the same sizes as the "frequencies" vectors. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.9.4.3 operator* [2/3]

```
FreqValues operator* (
            const FreqValues & lhs,
            const double rhs )  [friend]
```

An overloading of operator * which multiplies a *FreqValues* object and a *double* value, in that order.

This function calls the function `operator*(float,FreqValues)` with the order of its argument inverted, taking into account the commutative property of the multiplication.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.9.4.4 operator* [3/3]

```
FreqValues operator* (
            const double lhs,
            const FreqValues & rhs )  [friend]
```

An overloading of operator * which multiplies a *double* value and a *FreqValues* object, in that order.

The values of the object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the right-hand argument, which is the only argument that is of type *FreqValues*.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.9.4.5 operator+ [1/3]

```
FreqValues operator+ (
            const FreqValues & lhs,
            const FreqValues & rhs )  [friend]
```

An overloading of operator + which calculates the sum of two objects of structure *FreqValues*.

Before performing the operation, the function checks if the frequencies of each structure match and if the "values" vectors have the same sizes as the "frequencies" vectors. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.9.4.6 operator+ [2/3]

```
FreqValues operator+ (
            const FreqValues & lhs,
            const double rhs )  [friend]
```

An overloading of operator + which calculates the sum of a *FreqValues* object and a *double* value, in that order.

The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument, which is the only argument that is of type *FreqValues*.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.9.4.7 operator+ [3/3]

```
FreqValues operator+ (
            const double lhs,
            const FreqValues & rhs )  [friend]
```

An overloading of operator + which calculates the sum of a *double* value and a *FreqValues* object, in that order.

This function calls the function `operator+(FreqValues,float)` with the order of its arguments inverted, taking into account the commutative property of the sum. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the right-hand side argument, which is the only argument that is of type *FreqValues*.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.9.4.8 operator- [1/4]

```
FreqValues operator- (
            const FreqValues & argument )  [friend]
```

An overloading of the unary operator - which negates a *FreqValues* object.

This function takes each point of the given structure, negates it (the stored value, not the frequency) and stores the result in a different *FreqValues* structure, which is then returned. The rest of attributes (frequency, type, timestamp, etc.) are copied as-is to the object to be returned.

Parameters

in	*argument*	The *FreqValues* structure to be negated.

5.9.4.9 operator- [2/4]

```
FreqValues operator- (
            const FreqValues & lhs,
            const FreqValues & rhs )   [friend]
```

An overloading of operator - which calculates the subtraction of two objects of structure *FreqValues*.

This function calls the function `operator+(FreqValues,FreqValues)` with the right-hand side operand negated.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.9.4.10 operator- [3/4]

```
FreqValues operator- (
            const FreqValues & lhs,
            const double rhs )   [friend]
```

An overloading of operator - which calculates the subtraction of a *FreqValues* object and a *double* value, in that order.

This function calls the function `operator+(FreqValues,float)` with the right-hand side operand negated.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.9.4.11 operator- [4/4]

```
FreqValues operator- (
             const double lhs,
             const FreqValues & rhs )  [friend]
```

An overloading of operator - which calculates the subtraction of a *double* value and a *FreqValues* object, in that order.

This function calls the function `operator+(float,FreqValues)` with the right-hand side operand negated.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.9.4.12 operator/ [1/3]

```
FreqValues operator/ (
             const FreqValues & lhs,
             const FreqValues & rhs )  [friend]
```

An overloading of operator / which calculates the division between two objects of structure *FreqValues*.

Before performing the operation, the function checks if the frequencies of each structure match and if the "values" vectors have the same sizes as the "frequencies" vectors. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.9.4.13 operator/ [2/3]

```
FreqValues operator/ (
             const FreqValues & lhs,
             const double rhs )  [friend]
```

An overloading of operator / which calculates the division between a *FreqValues* object and a *double* value, in that order.

This function calls the function `operator*(FreqValues,float)` wit the right-hand argument inverted.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.9.4.14 operator/ [3/3]

```
FreqValues operator/ (
            const double lhs,
            const FreqValues & rhs )  [friend]
```

An overloading of operator / which calculates the division between a *double* value and a *FreqValues* object, in that order.

The values of the object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the right-hand argument, which is the only argument that is of type *FreqValues*.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.9.4.15 pow [1/2]

```
FreqValues pow (
            const FreqValues & base,
            const double exponent )  [friend]
```

An overloading of function `pow()`, power function, adapted to receive an argument of type *FreqValues* as base and an argument of type *double* as exponent.

This function takes each point of the structure, raises its value to the exponent and stores the result in a different *FreqValues* structure, which is then returned. The rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument, which the only argument that is of type *FreqValues*.

Parameters

in	*base*	The *FreqValues* structure to be used as the base of the power function.
in	*exponent*	The float value which will be used as the exponent.

5.9.4.16 pow [2/2]

```
FreqValues pow (
             const double base,
             const FreqValues & exponent )  [friend]
```

An overloading of function `pow()`, exponentiation, adapted to receive an argument of type *double* as base and an argument of type *FreqValues* as exponent.

This function takes the `float` value, given as the base, and raises it to each of the values of the *FreqValues* structure given as the exponent. Each result is stored in a different *FreqValues* structure which is then returned. The rest of attributes (frequency, type, timestamp, etc.) of this structure are copied from the right-hand side argument, which is the only argument that is of type *FreqValues*.

Parameters

in	*base*	The `float` value given as the base of the exponentiation operation.
in	*exponent*	The *FreqValues* structure whose values will be used as the exponents of the exponentiation operation.

5.9.5 Member Data Documentation

5.9.5.1 timeData

```
TimeData FreqValues::timeData
```

A TimeData object which contains information about the time when the values were captured, defined, etc.

The documentation for this struct was generated from the following files:

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Basics.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/FreqValues.cpp

5.10 FrontEndCalibrator Class Reference

The aim of this class is to calculate the total gain and total noise figure curves versus frequency of the RF front end.

```
#include <SweepProcessing.h>
```

Public Member Functions

- FrontEndCalibrator (CurveAdjuster &adj)

  The default class constructor.
- FrontEndCalibrator (CurveAdjuster &adj, const std::vector< BandParameters > &bandsParam)

  A more complete constructor which allows to insert the vector with the parameters of all frequency bands.
- ∼FrontEndCalibrator ()

  The FrontEndCalibrator destructor.
- void SetBandsParameters (const std::vector< BandParameters > &bandsParam)

  This method allows to insert a vector with the parameters of all frequency bands.
- void BuildRBWCurve ()

  This method builds a curve with RBW values versus frequency, taking into account this parameter for each frequency band.
- void LoadENR ()

  This method load the curve of ENR values versus frequency of the noise generator, from the corresponding file.
- void StartCalibration ()

  The calibration is started ensuring the noise source is turned off, switching the input to this device and preparing the object to receive the sweeps.
- void TurnOnNS ()

  This method just turns on the noise generator and it internally registers this situation.
- void TurnOffNS ()

  This method just turns off the noise generator and it internally registers this situation.
- void EndCalibration ()

  The calibration process is finished turning it off the noise generator and switching the input to the antenna.
- void SetSweep (const FreqValues &sweep)

  This method is intended to insert the two sweeps which are captured during the calibration process, with the noise source turned on and off.
- void SetNSoffTemp (const float nsOffTemp)

  This method allow to set the equivalent noise temperature of the noise source when it is turned off, what matches its physical temperature.
- void EstimateParameters ()

  This is one of the central methods. It estimates the front end parameters once the two sweeps were captured.
- const FreqValues & GetGain () const

  This method returns the estimated total gain curve of the front end.
- const FreqValues & GetNoiseTemp () const

  This method returns the estimated total noise temperature curve of the front end.
- const FreqValues & GetNoiseFigure () const

  This method returns the estimated total noise figure curve of the front end.
- FreqValues GetENRcorr () const

  This method returns the corrected ENR values of the noise source.
- float GetNSoffTemp () const

  This method returns the noise temperature of the noise source when it is turned off, what matches its physical temperature.
- const FreqValues & GetSweepNSoff () const

  This method returns the sweep with output power values which were got with the noise source turned off.
- const FreqValues & GetSweepNSon () const

  This method returns the sweep with output power values which were got with the noise source turned on.
- bool IsCalibStarted () const

  This method allows to know if the calibration has been started, i.e. the noise source is connected to the input.
- bool IsNoiseSourceOff () const

  This method allows to know if the noise source is turned off or not.
- const Sweep & CalibrateSweep (const Sweep &uncalSweep)

This another central method which is intended to calibrate (correct) the sweeps obtained with the antenna, once the front end parameters have been estimated.

- const Sweep & GetCalSweep () const

  This method returns the last calibrated sweep.

- void LoadDefaultParameters ()

  When errors occur during the calibration process, a set of default parameters curves are used to calibrate sweeps. This method loads those curve from the corresponding files.

- bool AreParamEmpty ()

  This method returns a `true` if no front end parameters have been estimated or loaded from the corresponding files, and a `false` otherwise.

5.10.1 Detailed Description

The aim of this class is to calculate the total gain and total noise figure curves versus frequency of the RF front end.

5.10.2 Constructor & Destructor Documentation

5.10.2.1 FrontEndCalibrator() [1/2]

```
FrontEndCalibrator::FrontEndCalibrator (
            CurveAdjuster & adj )
```

The default class constructor.

At instantiation, the programmer must provide a reference to a *CurveAdjuster* object.

Parameters

in	*adj*	A reference to a *CurveAdjuster* object, which will be used to adjust some internal curves.

5.10.2.2 FrontEndCalibrator() [2/2]

```
FrontEndCalibrator::FrontEndCalibrator (
            CurveAdjuster & adj,
            const std::vector< BandParameters > & bandsParam )
```

A more complete constructor which allows to insert the vector with the parameters of all frequency bands.

Parameters

in	*adj*	A reference to a *CurveAdjuster* object, which will be used to adjust some internal curves.
in	*bandsParam*	A vector with the parameters of all frequency bands.

5.10.2.3 ∼FrontEndCalibrator()

```
FrontEndCalibrator::∼FrontEndCalibrator ( )  [inline]
```

The *FrontEndCalibrator* destructor.

Its implementation is empty because the attributes destruction is implicitly. However, the destructor is defined here to allow this one to be called explicitly in any part of the code, what is used by the signals handler to destroy the objects when a signal to finish the execution of the software is received.

5.10.3 Member Function Documentation

5.10.3.1 BuildRBWCurve()

```
void FrontEndCalibrator::BuildRBWCurve ( )
```

This method builds a curve with RBW values versus frequency, taking into account this parameter for each frequency band.

The aim of the RBW curve is to simplify the syntax of the equations which are used in the methods `FrontEndCalibrator::EstimateParameters()` and `FrontEndCalibrator::CalibrateSweep()`. Firstly, the RBW curve is loaded with two points for each frequency band: one point with the RBW value of that band and its start frequency (Fstart) and the other point with the same RBW value and its stop frequency (Fstop). Then, the RBW curve is adjusted using the *CurveAdjuster* object and, because of the way the first values were loaded, the end RBW curve will be formed by steps, i.e. all the frequencies which corresponds to the same band will have the same RBW value.

5.10.3.2 CalibrateSweep()

```
const Sweep & FrontEndCalibrator::CalibrateSweep (
             const Sweep & powerOut )
```

This another central method which is intended to calibrate (correct) the sweeps obtained with the antenna, once the front end parameters have been estimated.

The calibration of sweeps with output power values implies the following tasks:

- Referencing the sweep to the input of the front end (antenna's output), what is done subtracting the total gain curve to the sweep, taking the power values in dBm and the gain values in dB.
- Removing of the internal noise added to the sweep by the front end, what is done taking into account its total equivalent noise temperature. To perform these tasks the following equation are used:

$$P_{IN_{eff}[dBm]} = P_{OUT[dBm]} - G_{receiver[dB]}$$

$$P_{IN_{eff}[W]} = 10^{\frac{P_{IN_{eff}[dBm]}}{10}} * 10^{-3}$$

$$P_{IN[W]} = P_{IN_{eff}[W]} - N_{receiver[W]} = P_{IN_{eff}[W]} - k_{[J/K]} * RBW_{[Hz]} * T_{receiver[K]}$$

$$P_{IN[dBm]} = 10 * \log_{10}(P_{IN[W]}) + 30$$

Parameters

`in`	*powerOut*	A *Sweep* structure which stores an uncalibrated sweep.

Returns

The calibrated sweep.

5.10.3.3 EstimateParameters()

```
void FrontEndCalibrator::EstimateParameters ( )
```

This is one of the central methods. It estimates the front end parameters once the two sweeps were captured.

This method must be called once the calibration process finished, i.e. the method `EndCalibration()` must be called first. The front end parameters, total gain and total noise figure, are estimated using the Y-Factor method, which is exposed in the Application Note "Noise Figure Measurement Accuracy – The Y-Factor Method" of Keysight Technologies.

To perform the estimation of the front end parameters the following equations are used:

$$ENR_{CORR} = ENR_{CAL} + \frac{T_O - T_{CORR}}{T_O}$$
$$T_{SON} = T_O * ENR_{CORR} + T_{SOFF}$$
$$Y = \frac{N_{OUT_{ON}}}{N_{OUT_{OFF}}}$$
$$T_{receiver} = \frac{T_{SON} - Y * T_{SOFF}}{Y - 1}$$
$$F_{receiver} = 1 + \frac{T_{receiver}}{T_O}$$
$$NF_{receiver} = 10 * \log_{10}(F_{receiver})$$
$$G_{receiver} = \frac{1}{2 * k * RBW} * \left[\frac{N_{OUT_{ON}}}{T_{SON} + T_{receiver}} + \frac{N_{OUT_{OFF}}}{T_{SOFF} + T_{receiver}} \right]$$

Once the parameters' curves have been estimated, their mean values are checked to be reasonable.

5.10.3.4 LoadENR()

```
void FrontEndCalibrator::LoadENR ( )
```

This method load the curve of ENR values versus frequency of the noise generator, from the corresponding file.

The ENR values versus frequency of the noise generator are loaded from the file BASE_PATH/calibration/enr.txt, then those values are corrected taking into account a statistical mean physical temperature of the noise source at time of the factory calibration. That technique is exposed in the Application Note "Noise Figure Measurement Accuracy – The Y-Factor Method" of Keysight Technologies. To finish, the corrected ENR curve is adjusted to be used in the mathematical operations with the captured sweeps.

5.10.3.5 SetBandsParameters()

```
void FrontEndCalibrator::SetBandsParameters (
            const std::vector< BandParameters > & bandsParam )
```

This method allows to insert a vector with the parameters of all frequency bands.

After the bands' parameters are stored, the RBW curve is built, taking into account those parameters.

Parameters

in	*bandsParam*	A vector with the parameters of all frequency bands.

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

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/SweepProcessing.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/FrontEndCalibrator.cpp

5.11 GPSCoordinates Struct Reference

A structure which saves the GPS coordinates.

```
#include <AntennaPositioning.h>
```

Public Attributes

- double latitude

  The latitude angle represented in decimal degrees. This angle becomes negative in the southern hemisphere.
- double longitude

  The longitude angle represented in decimal degrees. This angle becomes negative in the western hemisphere.

5.11.1 Detailed Description

A structure which saves the GPS coordinates.

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

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/AntennaPositioning.h

5.12 GPSInterface Class Reference

The class *GPSInterace* is intended to establish the communication with the Aaronia GPS receiver, to request and capture messages from this one and extract useful data from the messages.

```
#include <AntennaPositioning.h>
```

Public Member Functions

- GPSInterface ()

  The default constructor of class GPSInterface.
- ∼GPSInterface ()

  The GPSInterface class' destructor.
- void Initialize ()

  This method is intended to try the communication with the Aaronia GPS receiver and configure the device.
- void EnableStreaming ()

  This method is intended to enable the streaming of data from the GPS receiver.
- void DisableStreaming ()

  This method is intended to disable the data streaming from the Aaronia GPS receiver.
- void Purge ()

  A method intended to purge the input and output buffers of the USB interface.
- TimeData UpdateTimeData ()

  A method which reads the corresponding data reply to update the time data and returns it.
- Data3D UpdateGyroData ()

  A method which reads the corresponding data reply to update the gyroscope data and returns them.
- Data3D UpdateCompassData ()

  A method which reads the corresponding data reply to update the compass data and returns them.
- Data3D UpdateAccelerData ()

  A method which reads the corresponding data reply to update the accelerometer data and returns them.
- double UpdateYaw ()

  A method which reads the corresponding data replies to update the yaw angle and returns it.
- double UpdateRoll ()

  A method which reads the corresponding data replies to update the roll angle and returns it.
- double UpdatePitch ()

  A method which reads the corresponding data replies to update the pitch angle and returns it.
- void UpdatePressAndElevat ()

  A method which reads the corresponding data reply to update the pressure and the pressure-based elevation.
- unsigned int UpdateNumOfSatellites ()

  A method which reads the corresponding data reply (GPGGA reply) to update the number of satellites and returns it.
- void UpdateAll ()

  A method which reads all data replies to update all attributes.
- bool NewTimeData () const

  A method which allows to know if there are new time data.
- bool NewCoordinates () const

  A method which allows to know if there are new GPS coordinates.
- bool NewNumOfSatellites () const

  A method which allows to know if there is a new value of the number of satellites.
- bool NewGPSElevation () const

  A method which allows to know if there is a new value of the GPS-based elevation.
- bool NewGyroData () const

  A method which allows to know if there are new gyroscope data.
- bool NewCompassData () const

  A method which allows to know if there are new compass data.
- bool NewAccelerData () const

  A method which allows to know if there are new accelerometer data.
- bool NewPressure () const

  A method which allows to know if there is a new value of pressure.
- bool NewYaw () const

A method which allows to know if there is a new value of pressure.

- bool NewRoll () const

  A method which allows to know if there is a new value of the roll angle.
- bool NewPitch () const

  A method which allows to know if there is a new value of the pitch angle.
- bool NewGPSData () const

  A method which allows to know if there are new GPS data.
- const TimeData & GetTimeData ()

  This method returns the time data (date and time) which was received from the GPS satellites.
- const GPSCoordinates & GetCoordinates ()

  This method returns the GPS coordinates.
- unsigned int GetNumOfSatellites ()

  This method returns the current number of satellites which the GPS receiver is connected with.
- float GetGPSElevation ()

  This method returns the elevation of the GPS receiver over the sea level, measured in meters (m) and based on GPS data.
- const Data3D & GetGyroData ()

  This method returns the 3D gyroscope data.
- const Data3D & GetCompassData ()

  This method returns the 3D compass data.
- const Data3D & GetAccelerData ()

  This method returns the 3D accelerometer data.
- float GetPressure ()

  This method returns the ambient temperature, measured in hectopascal (hPa).
- float GetPressElevation ()

  This method returns the elevation of the GPS receiver over the sea level, measured in meters (m) and based on the ambient pressure.
- double GetYaw ()

  This method returns the yaw angle, measured in degrees and whose range is 0 to 359. North corresponds to 0°, east to 90°, south to 180°and west to 270°.
- double GetRoll ()

  This method returns the roll angle, measured in degrees and whose range is -180 to 180. This angle is zero if the device is put on horizontal surface, positive if it turns right and negative if it turns left.
- double GetPitch ()

  This method returns the pitch angle, measured in degrees and whose range is -180 to 180. This angle is zero if the device is put on horizontal surface, positive if the elevation is over the surface and negative if the elevation is below the surface.
- unsigned int GetDataRate () const

  This method returns the data rate when the streaming is enabled.
- bool IsConnected () const

  This method states if the communication with the Aaronia GPS receiver has been initialized.
- bool IsStreamingEnabled () const

  This method states if the data streaming is enabled.

Friends

- void ∗ StreamingThread (void ∗arg)

  The function which is executed by the thread which reads each reply of the GPS Logger and extract the data from them.

5.12.1 Detailed Description

The class *GPSInterace* is intended to establish the communication with the Aaronia GPS receiver, to request and capture messages from this one and extract useful data from the messages.

5.12.2 Constructor & Destructor Documentation

5.12.2.1 GPSInterface()

```
GPSInterface::GPSInterface ( )
```

The default constructor of class GPSInterface.

The constructor has to include the custom VID and PID combination of Aaronia GPS receiver within the allowed values, then it has to open the communication with the GPS receiver and set up the UART port on the FTDI chip.

5.12.2.2 ∼GPSInterface()

```
GPSInterface::∼GPSInterface ( )
```

The GPSInterface class' destructor.

The destructor has to make sure that the data streaming and the data logging into the microSD card are stopped, and then it closes the communication with the Aaronia GPS receiver.

5.12.3 Member Function Documentation

5.12.3.1 EnableStreaming()

```
void GPSInterface::EnableStreaming ( )
```

This method is intended to enable the streaming of data from the GPS receiver.

After the streaming has been enable the method CaptureStreamData() must be used to get the stream data and update the class attributes from them.

5.12.3.2 Initialize()

```
void GPSInterface::Initialize ( )
```

This method is intended to try the communication with the Aaronia GPS receiver and configure the device.

First, this method tries the communication with an ID command, and if the reply is right it shows the hardware version, firmware version and protocol version and then it disables the data streaming and the data logging into the microSD card. After, the GPS interface set up the following variables: data rate of streaming, accelerometer range and bandwidth of the average filter of the gyroscope sensor.

5.12.3.3 NewAccelerData()

```
bool GPSInterface::NewAccelerData ( ) const  [inline]
```

A method which allows to know if there are new accelerometer data.

This method is mainly intended to be used with the streaming option. When a PAAG,DATA,T reply is received and the accelerometer data are extracted from it, it is considered these data are new. Once the accelerometer data are got with the method GetAccelerData(), then they are considered old data.

5.12.3.4 NewCompassData()

```
bool GPSInterface::NewCompassData ( ) const  [inline]
```

A method which allows to know if there are new compass data.

This method is mainly intended to be used with the streaming option. When a PAAG,DATA,C reply is received and the compass data are extracted from it, it is considered these data are new. Once the compass data are got with the method GetCompassData(), then they are considered old data.

5.12.3.5 NewCoordinates()

```
bool GPSInterface::NewCoordinates ( ) const  [inline]
```

A method which allows to know if there are new GPS coordinates.

This method is mainly intended to be used with the streaming option. When a GPRMC reply is received and the coordinates are extracted from it, it is considered the coordinates are new data. Once the GPS coordinates are got with the method GetCoordinates(), then they are considered old data.

5.12.3.6 NewGPSData()

```
bool GPSInterface::NewGPSData ( ) const  [inline]
```

A method which allows to know if there are new GPS data.

This method is mainly intended to be used with the streaming option. When the corresponding replies, GPRMC and GPGGA, are received and the GPS data (time, coordinates, etc.) are extracted from them, it is considered these data are new. Once these data are got with the corresponding methods, then they are considered old data.

5.12.3.7 NewGPSElevation()

```
bool GPSInterface::NewGPSElevation ( ) const  [inline]
```

A method which allows to know if there is a new value of the GPS-based elevation.

This method is mainly intended to be used with the streaming option. When a GPGGA reply is received and the elevation is extracted from it, it is considered this number as a new data. Once the GPS-based elevation is got with the method GetGPSElevation(), then it is considered old data.

5.12.3.8 NewGyroData()

```
bool GPSInterface::NewGyroData ( ) const  [inline]
```

A method which allows to know if there are new gyroscope data.

This method is mainly intended to be used with the streaming option. When a PAAG,DATA,G reply is received and the gyroscope data are extracted from it, it is considered these data are new. Once the gyroscope data are got with the method GetGyroData(), then they are considered old data.

5.12.3.9 NewNumOfSatellites()

```
bool GPSInterface::NewNumOfSatellites ( ) const  [inline]
```

A method which allows to know if there is a new value of the number of satellites.

This method is mainly intended to be used with the streaming option. When a GPGGA reply is received and the number of satellites is extracted from it, it is considered this number as a new data. Once the number of satellites is got with the method GetNumOfSatellites(), then it is considered old data.

5.12.3.10 NewPitch()

```
bool GPSInterface::NewPitch ( ) const  [inline]
```

A method which allows to know if there is a new value of the pitch angle.

This method is mainly intended to be used with the streaming option. When the corresponding replies are received and the pitch angle is calculated from their data, it is considered this value is new. Once this value is got with the method GetPitch(), then it is considered old data.

5.12.3.11 NewPressure()

```
bool GPSInterface::NewPressure ( ) const  [inline]
```

A method which allows to know if there is a new value of pressure.

This method is mainly intended to be used with the streaming option. When a PAAG,DATA,B reply is received and the pressure is extracted from it, it is considered this data is new. Once the pressure value (or the pressure-based elevation) is got with the method GetPressure(), then it is considered old data.

5.12.3.12 NewRoll()

```
bool GPSInterface::NewRoll ( ) const  [inline]
```

A method which allows to know if there is a new value of the roll angle.

This method is mainly intended to be used with the streaming option. When the corresponding replies are received and the roll angle is calculated from their data, it is considered this value is new. Once this value is got with the method GetRoll(), then it is considered old data.

5.12.3.13 NewTimeData()

```
bool GPSInterface::NewTimeData ( ) const  [inline]
```

A method which allows to know if there are new time data.

This method is mainly intended to be used with the streaming option. When a GPRMC reply is received and the time data are extracted from it, it is considered the time data are new data. Once the time data are got with the method GetTimeData(), then they are considered old data.

5.12.3.14 NewYaw()

```
bool GPSInterface::NewYaw ( ) const  [inline]
```

A method which allows to know if there is a new value of pressure.

This method is mainly intended to be used with the streaming option. When a PAAG,DATA,B reply is received and the pressure is extracted from it, it is considered this data is new. Once the pressure value (or the pressure-based elevation) is got with the method GetPressure(), then it is considered old data.

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

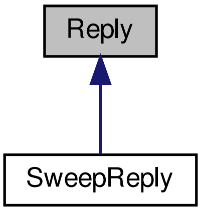
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/AntennaPositioning.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/GPSInterface.cpp

5.13 Reply Class Reference

The class *Reply* is intended to receive a bytes vector sent by the spectrum analyzer and to extract its information.

```
#include <Spectran.h>
```

Inheritance diagram for Reply:



Public Types

- enum ReplyType : char {
  VERIFY =0x01, **GETSTPVAR** =0x20, **SETSTPVAR**, **AMPFREQDAT**,
  UNINITIALIZED }

  An enumeration which contains the reply types which can be received from a Spectran HF-60105 V4 X spectrum analyzer.

Public Member Functions

- Reply ()

  The default constructor.
- Reply (const ReplyType type, const SpecVariable variable=SpecVariable::UNINITIALIZED)

  A constructor which allows to set the reply type and the variable name, in case of GETSTPVAR replies, and prepare the object to receive a spectrum analyzer's reply.
- Reply (const Reply &anotherReply)

  The copy constructor.
- virtual ∼Reply ()

  The class destructor which is defined as virtual because there are some classes derived from this one.
- virtual void PrepareTo (const ReplyType type, const SpecVariable variable=SpecVariable::UNINITIALIZED)

  This method is intended to set the reply type and variable name, in case of GETSTPVAR replies, and to ask the object to prepare itself to receive the bytes.
- virtual void InsertBytes (const std::uint8_t ∗data)

  This method is intended to insert a reply's bytes and to extract its data.
- ReplyType GetReplyType () const

  This method returns the reply type as the corresponding value of the enumeration ReplyType.
- std::string GetReplyTypeString () const

  A method which returns the reply type as a `std::string`.
- std::string GetVariableNameString () const

  A method which returns the name of the Spectran's variable which is related with the reply (GETSTPVAR reply) as a `std::string`.
- const std::vector< std::uint8_t > & GetBytesVector () const

  A method to get the bytes vector like this is implemented internally, a `std::vector` container.
- const std::uint8_t ∗ GetBytesPointer () const

  A method to get a direct pointer to the bytes of the internal vector.
- unsigned int GetNumOfBytes () const

  A method which allows to know the size of the bytes vector.
- float GetValue () const

  A method to get the value of the variable which was queried with a GETSTPVAR command or a power value (or voltage or field strength) which was received with a AMPFREQDAT reply.
- bool IsRight () const

  This method states if the received reply is right.
- virtual void Clear ()

  This method resets the object.
- virtual const Reply & operator= (const Reply &anotherReply)

  An overloading of the assignment operator, adapted to this class.

Protected Member Functions

- void FillBytesVector (const std::uint8_t ∗data)

  This method fills correctly the internal bytes vector with the received bytes.

Protected Attributes

- std::vector< std::uint8_t > bytes

  Bytes array (or vector) which has been received from the spectrum analyzer.
- float value

  The value (as a floating-point number) of a queried Spectran variable or a power value (or voltage of field strength). It has sense with GETSTPVAR and AMPFREQDAT replies.

5.13.1 Detailed Description

The class *Reply* is intended to receive a bytes vector sent by the spectrum analyzer and to extract its information.

When a command is sent to a Spectran HF-60105 V4 X spectrum analyzer, this will respond with another bytes vector (however some commands do not have reply), so the idea is to insert this in an object of class *Reply*, then the object will process and extract the info (variable id, value, etc.) of the bytes vector and finally the info will be available through the "Get" methods.

5.13.2 Member Enumeration Documentation

5.13.2.1 ReplyType

```
enum Reply::ReplyType :  char
```

An enumeration which contains the reply types which can be received from a Spectran HF-60105 V4 X spectrum analyzer.

This enumeration contains just four replies from the Spectran USB Protocol: *VERIFY*, *GETSTPVAR*, *SETSTPV*↩ *AR* and *AMPFREQDAT*. The other replies are received when an internal file of the spectrum analyzer have been queried or modified, and because that will not be done, they are not added. There is an extra command type which is *UNINITIALIZED* whose purpose is to state that the object is not prepared.

5.13.3 Constructor & Destructor Documentation

5.13.3.1 Reply() [1/2]

```
Reply::Reply (
             const ReplyType type,
             const SpecVariable variable = SpecVariable::UNINITIALIZED )
```

A constructor which allows to set the reply type and the variable name, in case of *GETSTPVAR* replies, and prepare the object to receive a spectrum analyzer's reply.

Parameters

in	*type*	The reply type: VERIFY, GETSTPVAR, SETSTPVAR or AMPFREQDAT.
in	*variable*	An optional argument which indicates the variable whose value will be received by a GETSTPVAR reply.

5.13.3.2 Reply() [2/2]

```
Reply::Reply (
             const Reply & anotherReply )
```

The copy constructor.

Parameters

in	*anotherReply*	A *Reply* object which is given to copy its attributes.

5.13.3.3 ∼Reply()

```
virtual Reply::∼Reply ( )  [inline], [virtual]
```

The class destructor which is defined as virtual because there are some classes derived from this one.

It was necessary to implement the destructor here to define this one as *virtual*. However, it was not necessary to explicitly clean, close and/or destroy any attribute and, because of that, the implementation is empty.

5.13.4 Member Function Documentation

5.13.4.1 FillBytesVector()

```
void Reply::FillBytesVector (
             const std::uint8_t * data )  [protected]
```

This method fills correctly the internal bytes vector with the received bytes.

Parameters

in	*data*	A pointer to the bytes which contain the information of the reply sent by the spectrum analyzer.

5.13.4.2 InsertBytes()

```
void Reply::InsertBytes (
             const std::uint8_t * data )  [virtual]
```

This method is intended to insert a reply's bytes and to extract its data.

The reply object must have been prepared before inserting the bytes vector.

Parameters

in	*data*	A pointer to the bytes which must be inserted in the object and from which the info must be extracted.

Reimplemented in SweepReply.

5.13.4.3 IsRight()

```
bool Reply::IsRight ( ) const
```

This method states if the received reply is right.

To do so, the method checks if the reply type (determined when the reply object was prepared) is equal to first received byte and if the reply type is VERIFY it checks if the following bytes are correct, or if the reply type is GE↩ TSTPVAR or SETSTPVAR it checks the "status" byte, which must be zero when all is right. By default the method indicates that the reply is incorrect.

5.13.4.4 operator=()

```
const Reply & Reply::operator= (
            const Reply & anotherReply )   [virtual]
```

An overloading of the assignment operator, adapted to this class.

Parameters

in	*anotherReply*	A *Reply* object which is given to copy its attributes.

5.13.4.5 PrepareTo()

```
void Reply::PrepareTo (
            const ReplyType type,
            const SpecVariable variable = SpecVariable::UNINITIALIZED )   [virtual]
```

This method is intended to set the reply type and variable name, in case of *GETSTPVAR* replies, and to ask the object to prepare itself to receive the bytes.

The tasks performed by this methods are: to clear the object, to set the reply type and then to call the private method `PrepareReply()`. The variable name must be set for the GETSTPVAR replies, for other cases it is not important.

Parameters

in	*type*	The reply type: VERIFY, GETSTPVAR, SETSTPVAR or AMPFREQDAT.
in	*variable*	An optional argument which indicates the variable whose value will be received by a GETSTPVAR reply.

Reimplemented in SweepReply.

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

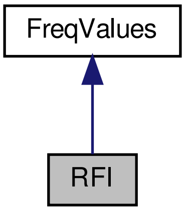
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Spectran.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Reply.cpp

5.14 RFI Struct Reference

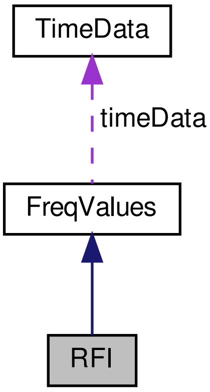
The aim of this structure is to store the data related with the detected RF interference (RFI): frequency, power, azimuth angle, polarization, time, reference norm, etc.

```
#include <Basics.h>
```

Inheritance diagram for RFI:



Collaboration diagram for RFI:



Public Types

- enum ThresholdsNorm { **ITU_RA769_2_VLBI**, **SKA_MODE1**, **SKA_MODE2** }

  Enumeration which contains the reference documents (recommendations, protocols, etc.) of harmful RFI levels (a.k.a. thresholds): the ITU recommendation RA.769-2, SKA protocol Mode 1, SKA protocol Mode 2.

Public Member Functions

- RFI ()

  The default constructor which calls the default constructor of structure FreqValues and set type to "rfi", azimuth angle to zero, number of bands to zero and set, by default, threshold norm to SKA_MODE1.
- RFI (const RFI &rfi)

  The copy constructor which receives a RFI object.
- void Clear ()

  The aim of this method is to clean the attributes of this structure.
- const RFI & operator= (const RFI &anotherRFI)

  An overloading of the assignment operator adapted to receive a RFI object.

Public Attributes

- float azimuthAngle

  The azimuth angle where this RFI was detected.
- std::string polarization

  The antenna polarization of the sweep where the RFI was detected.
- unsigned int numOfRFIBands

  The number of RFI bands defined as intervals of continuous data points where it was detected RFI.
- ThresholdsNorm threshNorm

5.14.1 Detailed Description

The aim of this structure is to store the data related with the detected RF interference (RFI): frequency, power, azimuth angle, polarization, time, reference norm, etc.

5.14.2 Constructor & Destructor Documentation

5.14.2.1 RFI()

```
RFI::RFI (
              const RFI & rfi )  [inline]
```

The copy constructor which receives a *RFI* object.

Parameters

in	*rfi*	A *RFI* structure given to copy its attributes.

5.14.3 Member Function Documentation

5.14.3.1 operator=()

```
const RFI& RFI::operator= (
             const RFI & anotherRFI )  [inline]
```

An overloading of the assignment operator adapted to receive a *RFI* object.

Parameters

in	*another↩ RFI*	Another *RFI* structure given to copy its attributes.

5.14.4 Member Data Documentation

5.14.4.1 threshNorm

```
ThresholdsNorm RFI::threshNorm
```

The norm (recommendation, protocol, etc.) which was used to define the harmful interference levels.

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

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Basics.h

5.15 RFIDetector Class Reference

The aim of this class is to compare each calibrated sweep with a threshold curve to determine where there is RF interference (RFI).

```
#include <SweepProcessing.h>
```

Public Member Functions

- RFIDetector (CurveAdjuster &adj)

  The unique class constructor.
- ∼RFIDetector ()

  The class destructor.
- void SetBandsParameters (const std::vector< BandParameters > &bandsParam)

  A method to insert a vector with the parameters of all frequency bands.
- void LoadThreshCurve (const RFI::ThresholdsNorm thrNorm)

  This method loads a determined thresholds curve from the corresponding file.
- const RFI & DetectRFI (const Sweep &sweep)

  This method detects RFI in a calibrated sweep.
- const FreqValues & GetThreshCurve () const

  This method returns the threshold curve which has been loaded.
- unsigned int GetNumOfRFIBands () const

  This method returns the number of RFI bands which were detected in the last sweep.
- const RFI & GetRFI () const

  This method returns the last detected RFI.

5.15.1 Detailed Description

The aim of this class is to compare each calibrated sweep with a threshold curve to determine where there is RF interference (RFI).

5.15.2 Constructor & Destructor Documentation

5.15.2.1 RFIDetector()

```
RFIDetector::RFIDetector (
            CurveAdjuster & adj )   [inline]
```

The unique class constructor.

At instantiation the programmer must provide a reference to a *CurveAdjuster* object.

Parameters

in	*adj*	A *CurveAdjuster* object.

5.15.2.2 ∼RFIDetector()

```
RFIDetector::∼RFIDetector ( )   [inline]
```

The class destructor.

Its implementation is empty because the attributes destruction is implicitly. However, the destructor is defined here to allow this one to be called explicitly in any part of the code, what is used by the signals handler to destroy the objects when a signal to finish the execution of the software is received.

5.15.3 Member Function Documentation

5.15.3.1 DetectRFI()

```
const RFI & RFIDetector::DetectRFI (
             const Sweep & sweep )
```

This method detects RFI in a calibrated sweep.

The given sweep should have been calibrated before.

Parameters

in	*sweep*	A calibrated sweep.

Returns

A structure with the pairs of values (frequency,power) where it was detected RFI.

5.15.3.2 LoadThreshCurve()

```
void RFIDetector::LoadThreshCurve (
             const RFI::ThresholdsNorm thrNorm )
```

This method loads a determined thresholds curve from the corresponding file.

The threshold curve is loaded from one of the fileS in the path BASE_PATH/thresholds/. The argument determines which recommendation, protocol or norm must be taken as reference to determine the threshold curve to be used, i.e. to determine from which file load that curve.

Parameters

in	*thrNorm*	The norm, recommendation or protocol that must be taken as reference.

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

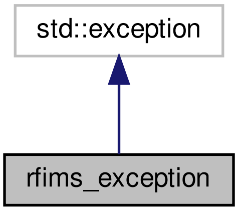
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/SweepProcessing.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/RFIDetector.cpp

5.16 rfims_exception Class Reference

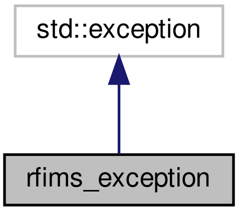
A class derived from standard class `std::exception`.

`#include <Basics.h>`

Inheritance diagram for rfims_exception:



Collaboration diagram for rfims_exception:



Public Member Functions

- rfims_exception (const std::string &msg="Error")

  The default constructor, which can set the exception message.
- void SetMessage (const std::string &msg)

  The aim of this function is to modify the exception message.
- void Prepend (const std::string &msg)

  This function is intended to add some text at the beginning of the exception message.
- void Append (const std::string &msg)

  This function is intended to add some text at the end of the exception message.
- const char ∗ what () const throw ()

  This is a standard function for classes which manage exceptions and is intended to return the exception message as a C string (`char*`).

5.16.1 Detailed Description

A class derived from standard class `std::exception`.

This class is customized to managed the exceptions in a desired way. This class has been defined to ease the appending of data to the message carried by an exception object.

5.16.2 Constructor & Destructor Documentation

5.16.2.1 rfims_exception()

```
rfims_exception::rfims_exception (
            const std::string & msg = "Error" )  [inline]
```

The default constructor, which can set the exception message.

Parameters

in	*msg*	The exception message, which can be of type `char*` or `std::string`.

5.16.3 Member Function Documentation

5.16.3.1 Append()

```
void rfims_exception::Append (
            const std::string & msg )  [inline]
```

This function is intended to add some text at the end of the exception message.

Parameters

in	*msg*	The sentence which must be appended to the exception message and which can be of type `char*` or `std::string`.

5.16.3.2 Prepend()

```
void rfims_exception::Prepend (
            const std::string & msg )  [inline]
```

This function is intended to add some text at the beginning of the exception message.

Parameters

in	*msg*	The sentence which must be prepended to the exception message and which can be of type `char*` or `std::string`.

5.16.3.3 SetMessage()

```
void rfims_exception::SetMessage (
            const std::string & msg )  [inline]
```

The aim of this function is to modify the exception message.

Parameters

in	*msg*	The exception message, which can be of type `char*` or `std::string`.

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

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Basics.h

5.17 RFPlotter Class Reference

The class *RFPlotter* is intended to plot sweeps, RF interference (RFI) and any frequency curve.

```
#include <SweepProcessing.h>
```

Public Member Functions

- RFPlotter (const std::string &titl="")

  The unique RFPloter constructor.
- ∼RFPlotter ()

  The class destructor.
- void Plot (const FreqValues &curve, const std::string &style="lines", const std::string &name="")

  This method is intended to plot any kind of frequency curve.
- void PlotSweep (const Sweep &swp)

  This method is specially intended to plot sweeps.
- void PlotRFI (const RFI &rfi)

  This method is specially intended to plot RF interference (RFI).
- void Clear ()

  This method cleans completely the plot, but the corresponding window will not be closed.

5.17.1 Detailed Description

The class *RFPlotter* is intended to plot sweeps, RF interference (RFI) and any frequency curve.

This function uses *Gnuplot* which is a C++ interface to the software *gnuplot*, a portable command-line driven graphing utility for Linux and other platforms. The interface uses a pipe to communicate with the software. Each instantiation of this class represents a different plot.

5.17.2 Constructor & Destructor Documentation

5.17.2.1 RFPlotter()

```
RFPlotter::RFPlotter (
             const std::string & titl = "" )  [inline]
```

The unique *RFPloter* constructor.

The constructor can receive a title for the plot but by default this is empty. This method set the style of the plot to "lines" and it performs an initial configuration of this one.

5.17.2.2 ∼RFPlotter()

```
RFPlotter::∼RFPlotter ( )  [inline]
```

The class destructor.

The destructor removes the temporary files which are generated by the *Gnuplot* interface.

5.17.3 Member Function Documentation

5.17.3.1 Plot()

```
void RFPlotter::Plot (
             const FreqValues & curve,
             const std::string & style = "lines",
             const std::string & name = "" )  [inline]
```

This method is intended to plot any kind of frequency curve.

Parameters

in	*curve*	The frequency curve to be plotted.
in	*style*	The style of the plot: "lines", "points", "linespoints", "impulses", "dots", "steps", "fsteps", "histeps", "boxes", "filledcurves" or "histograms".
in	*name*	The name or title of the plot.

5.17.3.2 PlotRFI()

```
void RFPlotter::PlotRFI (
             const RFI & rfi )  [inline]
```

This method is specially intended to plot RF interference (RFI).

This method is designed to plot the RFI which was detected in a determined sweep, after that sweep has already been plotted. Because of that, the plot's title and axes labels are not changed, but the style is changed to "points" to make a difference between the sweep points and the RFI points, which will be superimposed where there is RFI. Also, it is controlled the given *RFI* object has the same azimuth angle and polarization of the plotted sweep. Again, the frequencies are represented in MHz and the RFI values are assumed to be power values in dBm.

Parameters

in	*rfi*	The RFI to be plotted.

5.17.3.3 PlotSweep()

```
void RFPlotter::PlotSweep (
             const Sweep & swp )  [inline]
```

This method is specially intended to plot sweeps.

This method receives a *Sweep* object, plots it, gives a special title to the plot, which contains the info of the sweep, it sets the style to "lines" and it sets the x axis label in "Frequency (MHz)" and the y axis label in "Power (dBm)". So, the frequencies are represented in MHz and the values of the sweep are assumed to be power values in dBm. The curve label is set to "Calibrated sweep" because it is assumed that the plotting is performed after the calibration.

Parameters

in	*swp*	The sweep to be plotted.

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

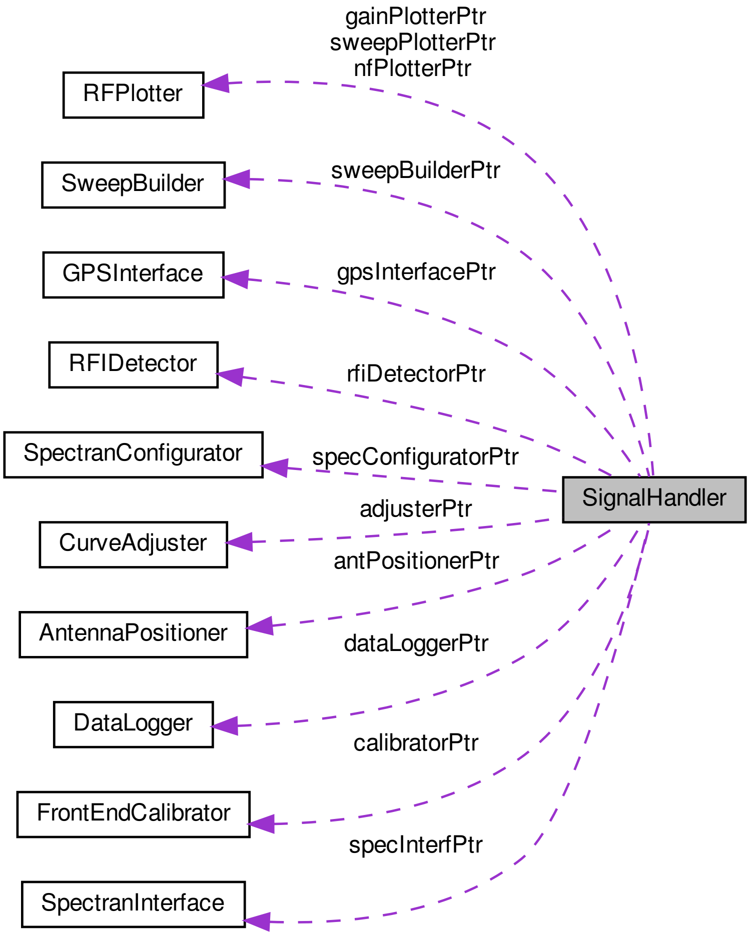
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/SweepProcessing.h

5.18 SignalHandler Class Reference

The class *SignalHandler* is intended to handle the interprocess signals (IPC) which terminates the software.

```
#include <TopLevel.h>
```

Collaboration diagram for SignalHandler:



Public Member Functions

- SignalHandler ()

  The SignalHandler constructor which set up the function which handles the signals.

- void SetAllPointers (SpectranInterface ∗specInterfPt, SpectranConfigurator ∗specConfiguratorPt, SweepBuilder ∗sweepBuilderPt, CurveAdjuster ∗adjusterPt, FrontEndCalibrator ∗calibratorPt, RFIDetector ∗rfiDetectorPt, DataLogger ∗dataLoggerPt, GPSInterface ∗gpsInterfacePt, AntennaPositioner ∗antPositionerPt, RFPlotter ∗sweepPlotterPt, RFPlotter ∗gainPlotterPt, RFPlotter ∗nfPlotterPt)

  This method is intended to set the pointers to the high-level objects and to set handler functions.

Static Public Member Functions

- static void ExitSignalHandler (int signum)

  This function is executed when a SIGINT or a SIGTERM signal arrives.

Static Public Attributes

- static SpectranInterface ∗ specInterfPtr =nullptr

  A pointer to the SpectranInterface object.
- static SpectranConfigurator ∗ specConfiguratorPtr =nullptr

  A pointer to the SpectranConfigurator object.
- static SweepBuilder ∗ sweepBuilderPtr =nullptr

  A pointer to the SweepBuilder object.
- static CurveAdjuster ∗ adjusterPtr =nullptr

  A pointer to the CurveAdjuster object.
- static FrontEndCalibrator ∗ calibratorPtr =nullptr

  A pointer to the FrontEndCalibrator object.
- static RFIDetector ∗ rfiDetectorPtr =nullptr

  A pointer to the RFIDetector object.
- static DataLogger ∗ dataLoggerPtr =nullptr

  A pointer to the DataLogger object.
- static GPSInterface ∗ gpsInterfacePtr =nullptr

  A pointer to the GPSInterface object.
- static AntennaPositioner ∗ antPositionerPtr =nullptr

  A pointer to the AntennaPositioner object.
- static RFPlotter ∗ sweepPlotterPtr =nullptr

  A pointer to the RFPlotter object which is responsible for the plotting of the last captured sweep.
- static RFPlotter ∗ gainPlotterPtr =nullptr

  A pointer to the RFPlotter object which is responsible for the plotting of the last estimated gain curve.
- static RFPlotter ∗ nfPlotterPtr =nullptr

  A pointer to the RFPlotter object which is responsible for the plotting of the last estimated noise figure curve.

5.18.1 Detailed Description

The class *SignalHandler* is intended to handle the interprocess signals (IPC) which terminates the software.

The signals which are handled by this class are SIGINT and SIGTERM. When one these signals arrived, the destructors of all the high-level objects are called, to ensure a clean closing of the software and an adequate turning off the RF front end.

5.18.2 Member Function Documentation

5.18.2.1 ExitSignalHandler()

```
static void SignalHandler::ExitSignalHandler (
              int signum )  [inline], [static]
```

This function is executed when a SIGINT or a SIGTERM signal arrives.

The function calls the destructor of almost all the high-level objects (defined in the main function) to ensure a clean closing of the software and an adequate turning off the RF front end.

Parameters

in	*signum*	This parameters must always be present and it states the number of the arrived signal.

5.18.3 Member Data Documentation

5.18.3.1 adjusterPtr

```
CurveAdjuster * SignalHandler::adjusterPtr =nullptr  [static]
```

A pointer to the *CurveAdjuster* object.

The instantiation of the pointer to the *CurveAjuster* object.

5.18.3.2 antPositionerPtr

```
AntennaPositioner * SignalHandler::antPositionerPtr =nullptr  [static]
```

A pointer to the *AntennaPositioner* object.

The instantiation of the pointer to the *AntennaPositioner* object.

5.18.3.3 calibratorPtr

```
FrontEndCalibrator * SignalHandler::calibratorPtr =nullptr  [static]
```

A pointer to the *FrontEndCalibrator* object.

The instantiation of the pointer to the *FrontEndCalibrator* object.

5.18.3.4 dataLoggerPtr

```
DataLogger * SignalHandler::dataLoggerPtr =nullptr  [static]
```

A pointer to the *DataLogger* object.

The instantiation of the pointer to the *DataLogger* object.

5.18.3.5 gainPlotterPtr

```
RFPlotter * SignalHandler::gainPlotterPtr =nullptr  [static]
```

A pointer to the *RFPlotter* object which is responsible for the plotting of the last estimated gain curve.

The instantiation of the pointer to the *RFPlotter* object which is responsible for the plotting of the last estimated gain curve.

5.18.3.6 gpsInterfacePtr

```
GPSInterface * SignalHandler::gpsInterfacePtr =nullptr  [static]
```

A pointer to the *GPSInterface* object.

The instantiation of the pointer to the *GPSInterface* object.

5.18.3.7 nfPlotterPtr

```
RFPlotter * SignalHandler::nfPlotterPtr =nullptr  [static]
```

A pointer to the *RFPlotter* object which is responsible for the plotting of the last estimated noise figure curve.

The instantiation of the pointer to the *RFPlotter* object which is responsible for the plotting of the last estimated noise figure curve.

5.18.3.8 rfiDetectorPtr

```
RFIDetector * SignalHandler::rfiDetectorPtr =nullptr  [static]
```

A pointer to the *RFIDetector* object.

The instantiation of the pointer to the *RFIDetector* object.

5.18.3.9 specConfiguratorPtr

```
SpectranConfigurator * SignalHandler::specConfiguratorPtr =nullptr  [static]
```

A pointer to the *SpectranConfigurator* object.

The instantiation of the pointer to the *SpectranConfigurator* object.

5.18.3.10 specInterfPtr

```
SpectranInterface * SignalHandler::specInterfPtr =nullptr  [static]
```

A pointer to the *SpectranInterface* object.

Global variables which are used by the SignalHandler class/////////////// The instantiation of the pointer to the *SpectranInterface* object.

5.18.3.11 sweepBuilderPtr

```
SweepBuilder * SignalHandler::sweepBuilderPtr =nullptr  [static]
```

A pointer to the *SweepBuilder* object.

The instantiation of the pointer to the *SweepBuilder* object.

5.18.3.12 sweepPlotterPtr

```
RFPlotter * SignalHandler::sweepPlotterPtr =nullptr  [static]
```

A pointer to the *RFPlotter* object which is responsible for the plotting of the last captured sweep.

The instantiation of the pointer to the *RFPlotter* object which is responsible for the plotting of the last captured sweep.

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

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/TopLevel.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/TopLevel.cpp

5.19 SpectranConfigurator Class Reference

The class *SpectranConfigurator* is intended to manage the process of configuring the Aaronia Spectran device.

```
#include <Spectran.h>
```

Classes

- struct FixedParameters

  This structure saves the fixed parameters of the spectrum analyzer, i.e. the parameters which do not change through the entire measurement cycle.

Public Member Functions

- SpectranConfigurator (SpectranInterface &interf)

  The default constructor.
- ∼SpectranConfigurator ()

  The destructor which just makes sure the input file stream is closed.
- bool LoadFixedParameters ()

  This method loads the fixed Spectran's parameters from the corresponding file.
- bool LoadBandsParameters ()

  This method loads the frequency bands parameters from the corresponding file.
- void InitialConfiguration ()

  This method is intended to configure the spectrum analyzer with the fixed parameters at the beginning of a measurement cycle.
- BandParameters ConfigureNextBand ()

  The aim of this method is to configure the spectrum analyzer with parameters of the next frequency band.
- void SetCurrBandParameters (const BandParameters &currBandParam)

  This method allows to change the parameters of the current frequency band, given a BandParameters structure as parameter.
- const FixedParameters & GetFixedParameters () const

  This method returns the fixed parameters.
- const std::vector< BandParameters > & GetBandsParameters () const

  This method returns a vector with the parameters of all frequency bands.
- const std::vector< BandParameters > & GetSubBandsParameters () const

This method returns a vector with the parameters of all frequency sub-bands.

- const BandParameters & GetCurrBandParameters () const

  This method returns the parameters of the current frequency band.
- unsigned int GetNumOfBands () const

  This method returns the number of frequency sub-bands (or bands if they were not split).
- unsigned int GetBandIndex () const

  This method returns the current value of the band index.
- bool IsLastBand () const

  This method returns a `true` if the current band is the last one and a `false` otherwise.

5.19.1 Detailed Description

The class *SpectranConfigurator* is intended to manage the process of configuring the Aaronia Spectran device.

To do this task, this component give *Command* objects to the *SpectranInterface* object, which writes them in the spectrum analyzer and then it reads the replies, the Spectran Interface passes them as *Reply* objects to the Spectran Configurator, so this one can know if the last configuration was performed successfully or not.

5.19.2 Constructor & Destructor Documentation

5.19.2.1 SpectranConfigurator()

```
SpectranConfigurator::SpectranConfigurator (
            SpectranInterface & interf )
```

The default constructor.

At instantiation of an object, a reference to the *SpectranInterface* object must be given.

Parameters

in	*interf*	A reference to the unique *SpectranInterface* object, which is responsible for the communication with the spectrum analyzer.

5.19.3 Member Function Documentation

5.19.3.1 ConfigureNextBand()

```
BandParameters SpectranConfigurator::ConfigureNextBand ( )
```

The aim of this method is to configure the spectrum analyzer with parameters of the next frequency band.

The first time this method is called, it configures the spectrum analyzer with the first band's parameters. Then, it will increase the band index to move to the parameters of the next band. Again, the streaming of sweep points should be disabled before calling this method.

This method returns the parameters of the current frequency band as a *BandParameters* structure.

5.19.3.2 InitialConfiguration()

```
void SpectranConfigurator::InitialConfiguration ( )
```

This method is intended to configure the spectrum analyzer with the fixed parameters at the beginning of a measurement cycle.

This method should be used at the beginning of the first measurement cycle and at beginning of rest ones if the file with the fixed parameters has been modified. Obviously, the sending of measurements via USB must be disable before calling this method.

5.19.3.3 LoadBandsParameters()

```
bool SpectranConfigurator::LoadBandsParameters ( )
```

This method loads the frequency bands parameters from the corresponding file.

The method loads the parameters of one band, stores them in a structure *BandsParameters* and then it splits the band in several sub-bands, if its span is bigger than 200 MHz. Then it moves to next frequency band until the end-of-file (EOF) is reached.

The method returns a `true` if the parameters were loaded from the file, either because it is the first time they are read or because the corresponding file was modified. If the parameters had been loaded before and it was found that the file was not modified from the last loading, the parameters are not read from the file and the method returns a `false`.

5.19.3.4 LoadFixedParameters()

```
bool SpectranConfigurator::LoadFixedParameters ( )
```

This method loads the fixed Spectran's parameters from the corresponding file.

The method returns a boolean value to indicate if the fixed parameters have been updated so the initial configuration should be repeated. The method returns a `true` if the parameters were loaded from the file, either because it is the first time they are read or because the corresponding file was modified. If the parameters had been loaded before and it was found that the file was not modified from the last loading, the parameters are not read from the file and the method returns a `false`.

5.19.3.5 SetCurrBandParameters()

```
void SpectranConfigurator::SetCurrBandParameters (
            const BandParameters & currBandParam )  [inline]
```

This method allows to change the parameters of the current frequency band, given a *BandParameters* structure as parameter.

Parameters

in	*currBandParam*	A *BandParameters* structure with the parameters which it is desired the current frequency band to have.

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

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Spectran.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/SpectranConfigurator.cpp

5.20 SpectranInterface Class Reference

The aim of this class is to manage the communication with the Aaronia Spectran device.

```
#include <Spectran.h>
```

Public Member Functions

- SpectranInterface ()

  The default class constructor.
- ∼SpectranInterface ()

  The class destructor.
- void Initialize ()

  This method logs in with the spectrum analyzer, once the communication has been opened and its parameters has been set up.
- unsigned int Available ()

  This method returns the number of available bytes in the input buffer.
- void ResetSweep ()

  This method is intended to restore the streaming of sweep points to the first point which corresponds to the initial frequency (Fstart).
- void EnableSweep ()

  This method allows to enable the streaming of sweep points.
- void DisableSweep ()

  This method allows to disable the streaming of sweep points.
- void Purge ()

  This method purges the input buffer.
- void SoftReset ()

  The soft reset is performed using the function `FT_ResetDevice()` of the driver D2XX.
- void HardReset ()

  The hard reset implies that the spectrum analyzer is turned off and then it is turned on again.
- void LogOut ()

  This method finishes the communication session with the spectrum analyzer.
- DWORD GetVID () const

  This method returns the Vendor ID of the spectrum analyzer.
- DWORD GetPID () const

  This method returns the Product ID of the spectrum analyzer.
- std::string GetDevDescription () const

  This method returns the device description (a string) of the spectrum analyzer.

- bool IsLogged () const

  This method returns a true value if the communication has been opened and a login has been performed, and a false value otherwise.
- bool IsSweepEnabled () const

  This method returns a true value if the streaming of sweep points has been enabled and a false otherwise.
- void SoundNewSweep ()

  This method allows to perform the a sound to state the capturing of a sweep has finished.
- void Write (const Command &command)

  This method is intended to perform all the writing operations.
- void Read (Reply &reply)

  This method is intended to perform all the reading operations.

5.20.1 Detailed Description

The aim of this class is to manage the communication with the Aaronia Spectran device.

This class establishes the communication with the Aaronia Spectran device (VID=0403, PID=E8D8) and allows to write commands to the device and read its replies. Also, it allows to know the available bytes in the input buffer, purge that buffer and reset the communication with the device.

5.20.2 Constructor & Destructor Documentation

5.20.2.1 SpectranInterface()

```
SpectranInterface::SpectranInterface ( )
```

The default class constructor.

This constructor initializes the internal flags, flagLogIn and flagSweepsEnabled, as false, includes the VID and PID of the spectrum analyzer in the list of possible values and, finally, it calls the method `OpenAndSetUp()`.

5.20.2.2 ∼SpectranInterface()

```
SpectranInterface::∼SpectranInterface ( )
```

The class destructor.

The destructor carry out the logging out and the communication closing. In each operation, this method produces messages in the `stdout` if there were errors, but it never produces exceptions.

5.20.3 Member Function Documentation

5.20.3.1 DisableSweep()

```
void SpectranInterface::DisableSweep ( )
```

This method allows to disable the streaming of sweep points.

To ensure the streaming is disabled at least two commands (USMEAS, 0) must be sent because the first one will receive a wrong reply as there is a delay until the spectrum analyzer stops sending sweep points (AMPFREQDAT), so the reply to first command will be mixed with the sweep points and the software will read it with errors. Because of that, this method is implemented with a loop where it is tried to disable the streaming times, up to 4 times. After the streaming is disabled successfully, the input buffer is purged.

5.20.3.2 EnableSweep()

```
void SpectranInterface::EnableSweep ( )
```

This method allows to enable the streaming of sweep points.

Take into account that once the streaming the sweep points is enabled, the spectrum analyzer will send these points autonomously, i.e. the communication will not follow anymore the master-slave paradigm where the slave only sends data to the master when this sends a request. So the software must be prepared to receive and process the data stream.

5.20.3.3 HardReset()

```
void SpectranInterface::HardReset ( )
```

The hard reset implies that the spectrum analyzer is turned off and then it is turned on again.

To start, a logout is performed, then the communication is closed, the entire RF front end is turned off sequentially, then it is waited 10 seconds, the entire front end is turned on again sequentially, and finally the communication is opened again. After the calling of this method, a login must be performed calling the method Initialize().

5.20.3.4 Initialize()

```
void SpectranInterface::Initialize ( )
```

This method logs in with the spectrum analyzer, once the communication has been opened and its parameters has been set up.

Firstly, this method sends two VERIFY commands to log in with the spectrum analyzer. If that operation failed, it resets the spectrum analyzer in a hard way (turn it off and then turn it on), then it tries again to send two VERIFY commands and if that failed it retries the reset and repeat this up to three times.

Secondly, it makes sure the streaming of sweep points is disabled. And finally, it set up the speaker volume, produces the login sounds and purge the input and output buffers.

5.20.3.5 LogOut()

```
void SpectranInterface::LogOut ( )
```

This method finishes the communication session with the spectrum analyzer.

To ensure the logout can be carried out successfully, first, the streaming of sweep points is disabled, then the logout sound is performed and, finally, the command to logout is sent to the spectrum analyzer.

5.20.3.6 Read()

```
void SpectranInterface::Read (
            Reply & reply )  [inline]
```

This method is intended to perform all the reading operations.

Before the calling of the function `FT_Read()`, there is a loop where it is queried if the number of waited bytes are available. After each query which states less bytes than which are waited, the method waits a certain time before the next query. The loop iterates a certain number of times and, after that, if the waited bytes are not available the method finishes and raises an exception. In the other hand, if the bytes are available before the last iteration, the method moves to a different loop where it is tried to read the bytes using the function `FT_Read()`. If an error occurs inside this loop, the method waits during a time and then retry the operation. If many errors occur the method finishes and raises an exception. But if all the bytes are read successfully, so then they are inserted in the *Reply* object.

Parameters

`in,out`	*reply*	A *Reply* object which states the number of bytes must be read and which receives these bytes to the extract the info from them.

5.20.3.7 ResetSweep()

```
void SpectranInterface::ResetSweep ( )
```

This method is intended to restore the streaming of sweep points to the first point which corresponds to the initial frequency (Fstart).

The reset of the current sweep has sense when the streaming of sweep points is enabled.

5.20.3.8 SoftReset()

```
void SpectranInterface::SoftReset ( )
```

The soft reset is performed using the function `FT_ResetDevice()` of the driver D2XX.

First, a logout is performed, then the communication is restarted using the function FT_ResetDevice() and, finally, the method sleeps 3 seconds. After calling this method the communication must be initialized again with the method Initialize().

5.20.3.9 SoundNewSweep()

```
void SpectranInterface::SoundNewSweep ( )
```

This method allows to perform the a sound to state the capturing of a sweep has finished.

The new-sweep sound is compound of just one pulse whose duration is determined by the constant NEW_SW←EEP_SOUND_DURATION. This method should be called by the object which is responsible for the capture of the sweep points and the building of the entire sweep.

5.20.3.10 Write()

```
void SpectranInterface::Write (
            const Command & command )  [inline]
```

This method is intended to perform all the writing operations.

First, the method copies the bytes of the command to an auxiliary array, because the function `FT_Write()` destroys the array of bytes which is passed to it, so this is done to avoid the modification of the *Command* object. Then the function `FT_Write()` is called with the corresponding arguments and, finally, it is controlled if there were errors and if all bytes were written.

Parameters

`in`	*command*	A *Command* object which contains the bytes must be sent to the spectrum analyzer.

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

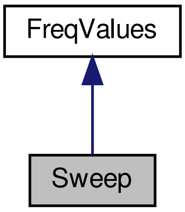
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Spectran.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/SpectranInterface.cpp

5.21 Sweep Struct Reference

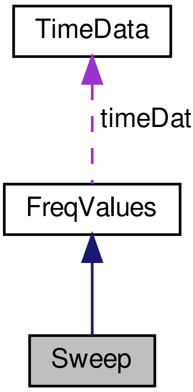
The aim of this structure is to store the data points of a sweep obtained with the spectrum analyzer in a determined azimuth position, with a specific polarization.

```
#include <Basics.h>
```

Inheritance diagram for Sweep:



Collaboration diagram for Sweep:



Public Member Functions

- Sweep ()

  The default constructor which calls the default constructor of FreqValues structure and set type to "sweep" and azimuth angle to zero.
- Sweep (const FreqValues &freqValues)

  A copy constructor which receives a FreqValues object.
- Sweep (const Sweep &sweep)

  A copy constructor which receives a Sweep object.
- void Clear ()

  The aim of this method is to clean the structure, i.e. to delete all data points, set azimuth angle to zero and clean polarization.
- const Sweep & operator= (const Sweep &sweep)

  An overloading of the assignment operator adapted to this structure.

Public Attributes

- float azimuthAngle

  The azimuth position (or angle) of the antenna when the sweep was captured.
- std::string polarization

Friends

- Sweep operator- (const Sweep &argument)

  An overloading of the unary operator - which negates a Sweep object.
- Sweep operator+ (const Sweep &lhs, const Sweep &rhs)

  An overloading of operator + which calculates the sum of two objects of structure Sweep.
- Sweep operator+ (const Sweep &lhs, const std::vector< value_type > &rhs)

An overloading of operator + which calculates the sum of a Sweep object and a `std::vector<float>` container, in that order.

- Sweep operator+ (const std::vector< value_type > &lhs, const Sweep &rhs)

  An overloading of operator + which calculates the sum of a `std::vector<float>` container and a Sweep object, in that order.
- Sweep operator+ (const Sweep &lhs, const FreqValues &rhs)

  An overloading of operator + which calculates the sum of a Sweep object and a FreqValues object, in that order.
- Sweep operator+ (const Sweep &lhs, const double rhs)

  An overloading of operator + which calculates the sum of a Sweep object and a double value, in that order.
- Sweep operator+ (const Sweep &lhs, const float rhs)

  An overloading of operator + which calculates the sum of a Sweep object and a float value, in that order.
- Sweep operator+ (const double lhs, const Sweep &rhs)

  An overloading of operator + which calculates the sum of a double value and a Sweep object, in that order.
- Sweep operator+ (const float lhs, const Sweep &rhs)

  An overloading of operator + which calculates the sum of a float value and a Sweep object, in that order.
- Sweep operator- (const Sweep &lhs, const Sweep &rhs)

  An overloading of operator - which calculates the subtraction of two objects of structure Sweep.
- Sweep operator- (const Sweep &lhs, const std::vector< value_type > &rhs)

  An overloading of operator - which calculates the subtraction of a Sweep object and a `std::vector<float>` container, in that order.
- Sweep operator- (const std::vector< value_type > &lhs, const Sweep &rhs)

  An overloading of operator - which calculates the subtraction of a `std::vector<float>` container and a Sweep object, in that order.
- Sweep operator- (const Sweep &lhs, const FreqValues &rhs)

  An overloading of operator - which calculates the subtraction of a Sweep object and a FreqValues object, in that order.
- Sweep operator∗ (const Sweep &lhs, const Sweep &rhs)

  An overloading of operator ∗ which multiplies two objects of structure Sweep.
- Sweep operator∗ (const Sweep &lhs, const double rhs)

  An overloading of operator ∗ which multiplies a Sweep object and a double value, in that order.
- Sweep operator∗ (const Sweep &lhs, const float rhs)

  An overloading of operator ∗ which multiplies a Sweep object and a float value, in that order.
- Sweep operator∗ (const double lhs, const Sweep &rhs)

  An overloading of operator ∗ which multiplies a double value and a Sweep object, in that order.
- Sweep operator∗ (const float lhs, const Sweep &rhs)

  An overloading of operator ∗ which multiplies a float value and a Sweep object, in that order.
- Sweep operator/ (const Sweep &lhs, const Sweep &rhs)

  An overloading of operator / which calculates the division between two objects of structure Sweep.
- Sweep operator/ (const Sweep &lhs, const double rhs)

  An overloading of operator / which calculates the division between a Sweep object and a double value, in that order.
- Sweep operator/ (const Sweep &lhs, const float rhs)

  An overloading of operator / which calculates the division between a Sweep object and a float value, in that order.
- Sweep operator/ (const double lhs, const Sweep &rhs)

  An overloading of operator / which calculates the division between a double value and a Sweep object, in that order.
- Sweep operator/ (const float lhs, const Sweep &rhs)

  An overloading of operator / which calculates the division between a float value and a Sweep object, in that order.
- Sweep log10 (const Sweep &argument)

  An overloading of function `log10()` adapted to receive an argument of type Sweep.
- Sweep pow (const Sweep &base, const double exponent)

  An overloading of function `pow()` adapted to receive an argument of type Sweep as base and an argument of type double as exponent.
- Sweep pow (const Sweep &base, const float exponent)

An overloading of function `pow()` adapted to receive an argument of type Sweep as base and an argument of type float as exponent.

- Sweep pow (const double base, const Sweep &exponent)

  An overloading of function `pow()` adapted to receive an argument of type double as base and an argument of type Sweep as exponent.

- Sweep pow (const float base, const Sweep &exponent)

  An overloading of function `pow()` adapted to receive an argument of type float as base and an argument of type Sweep as exponent.

Additional Inherited Members

5.21.1 Detailed Description

The aim of this structure is to store the data points of a sweep obtained with the spectrum analyzer in a determined azimuth position, with a specific polarization.

5.21.2 Constructor & Destructor Documentation

5.21.2.1 Sweep() [1/2]

```
Sweep::Sweep (
            const FreqValues & freqValues )   [inline]
```

A copy constructor which receives a *FreqValues* object.

Just the compatible attributes are copied.

Parameters

in	*freqValues*	A *FreqValues* structure which is given to copy its attributes.

5.21.2.2 Sweep() [2/2]

```
Sweep::Sweep (
            const Sweep & sweep )   [inline]
```

A copy constructor which receives a *Sweep* object.

Parameters

in	*sweep*	Another *Sweep* structure which is given to copy its attributes.

5.21.3 Member Function Documentation

5.21.3.1 operator=()

```
const Sweep & Sweep::operator= (
             const Sweep & sweep )
```

An overloading of the assignment operator adapted to this structure.

Parameters

in	*sweep*	Another *Sweep* structure which is given to copy its attributes.

5.21.4 Friends And Related Function Documentation

5.21.4.1 log10

```
Sweep log10 (
             const Sweep & argument )   [friend]
```

An overloading of function `log10()` adapted to receive an argument of type *Sweep*.

This function takes each point of the given structure, apply the decimal logarithm whit its value and stores the result in a different *Sweep* structure, which is then returned. The rest of attributes (frequency, type, timestamp, etc.) are copied as-is to the object to be returned.

Parameters

in	*argument*	The *Sweep* structure to be used as argument to the decimal logarithm operation.

5.21.4.2 operator* [1/3]

```
Sweep operator* (
             const Sweep & lhs,
             const Sweep & rhs )   [friend]
```

An overloading of operator $*$ which multiplies two objects of structure *Sweep*.

Before performing the operation, the function checks if the frequencies of each structure match and if the "values" vectors have the same sizes as the "frequencies" vectors. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.21.4.3 operator∗ [2/3]

```
Sweep operator∗ (
            const Sweep & lhs,
            const double rhs )  [friend]
```

An overloading of operator ∗ which multiplies a *Sweep* object and a *double* value, in that order.

The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument, the only one argument of type *Sweep*.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.21.4.4 operator∗ [3/3]

```
Sweep operator∗ (
            const double lhs,
            const Sweep & rhs )  [friend]
```

An overloading of operator ∗ which multiplies a *double* value and a *Sweep* object, in that order.

The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the right-hand side argument, the only one argument of type *Sweep*.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.21.4.5 operator+ [1/4]

```
Sweep operator+ (
            const Sweep & lhs,
            const Sweep & rhs )  [friend]
```

An overloading of operator + which calculates the sum of two objects of structure *Sweep*.

Before performing the operation, the function checks if the frequencies of each structure match and if the "values" vectors have the same sizes as the "frequencies" vectors. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.21.4.6 operator+ [2/4]

```
Sweep operator+ (
            const Sweep & lhs,
            const FreqValues & rhs )  [friend]
```

An overloading of operator + which calculates the sum of a *Sweep* object and a *FreqValues* object, in that order.

To perform this operation the *FreqValues* argument is casted to *Sweep*. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument, because the other argument has less attributes as it is an object of the base class *FreqValues* from which the class *Sweep* derives.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.21.4.7 operator+ [3/4]

```
Sweep operator+ (
            const Sweep & lhs,
            const double rhs )  [friend]
```

An overloading of operator + which calculates the sum of a *Sweep* object and a *double* value, in that order.

The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument, the only one argument of type *Sweep*.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.21.4.8 operator+ [4/4]

```
Sweep operator+ (
            const double lhs,
            const Sweep & rhs )  [friend]
```

An overloading of operator + which calculates the sum of a *double* value and a *Sweep* object, in that order.

The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the right-hand side argument, the only one argument of type *Sweep*.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.21.4.9 operator- [1/3]

```
Sweep operator- (
            const Sweep & argument )  [friend]
```

An overloading of the unary operator - which negates a *Sweep* object.

This function takes each point of the given structure, negates it (the stored value, not the frequency) and stores the result in a different *Sweep* structure, which is then returned. The rest of attributes (frequency, type, timestamp, etc.) are copied as-is to the object to be returned.

Parameters

in	*argument*	The *Sweep* structure to be negated.

5.21.4.10 operator- [2/3]

```
Sweep operator- (
            const Sweep & lhs,
            const Sweep & rhs )  [friend]
```

An overloading of operator - which calculates the subtraction of two objects of structure *Sweep*.

Before performing the operation, the function checks if the frequencies of each structure match and if the "values" vectors have the same sizes as the "frequencies" vectors. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.21.4.11 operator- [3/3]

```
Sweep operator- (
            const Sweep & lhs,
            const FreqValues & rhs )  [friend]
```

An overloading of operator - which calculates the subtraction of a *Sweep* object and a *FreqValues* object, in that order.

To perform this operation the *FreqValues* argument is casted to *Sweep*. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument, because the other argument has less attributes as it is an object of the base class *FreqValues* from which the class *Sweep* derives.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.21.4.12 operator/ [1/3]

```
Sweep operator/ (
            const Sweep & lhs,
            const Sweep & rhs )  [friend]
```

An overloading of operator / which calculates the division between two objects of structure *Sweep*.

Before performing the operation, the function checks if the frequencies of each structure match and if the "values" vectors have the same sizes as the "frequencies" vectors. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.21.4.13 operator/ [2/3]

```
Sweep operator/ (
            const Sweep & lhs,
            const double rhs )  [friend]
```

An overloading of operator / which calculates the division between a *Sweep* object and a *double* value, in that order.

The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument, the only one argument of type *Sweep*.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.21.4.14 operator/ [3/3]

```
Sweep operator/ (
            const double lhs,
            const Sweep & rhs )  [friend]
```

An overloading of operator / which calculates the division between a *double* value and a *Sweep* object, in that order.

The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the right-hand side argument, the only one argument of type *Sweep*.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.21.4.15 pow [1/2]

```
Sweep pow (
            const Sweep & base,
            const double exponent )   [friend]
```

An overloading of function `pow()` adapted to receive an argument of type *Sweep* as base and an argument of type *double* as exponent.

This function takes each point of the structure, raises its value to the exponent and stores the result in a different *Sweep* structure, which is then returned. The rest of attributes (frequency, type, timestamp, etc.) are copied from the base argument, which the only argument that is of type *Sweep*.

Parameters

in	*base*	The *Sweep* structure to be used as the base of the power function.
in	*exponent*	The float value which will be used as the exponent.

5.21.4.16 pow [2/2]

```
Sweep pow (
            const double base,
            const Sweep & exponent )   [friend]
```

An overloading of function `pow()` adapted to receive an argument of type *double* as base and an argument of type *Sweep* as exponent.

This function takes the `float` value, given as the base, and raises it to each of the values of the *Sweep* structure given as the exponent. Each result is stored in a different *Sweep* structure which is then returned. The rest of attributes (frequency, type, timestamp, etc.) of this structure are copied from the right-hand side argument, which is the only argument that is of type *Sweep*.

Parameters

in	*base*	The `float` value given as the base of the exponentiation operation.
in	*exponent*	The *Sweep* structure whose values will be used as the exponents of the exponentiation operation.

5.21.5 Member Data Documentation

5.21.5.1 polarization

```
std::string Sweep::polarization
```

The antenna polarization when the sweep was captured.

The documentation for this struct was generated from the following files:

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Basics.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/FreqValues.cpp

5.22 SweepBuilder Class Reference

The aim of class *SweepBuilder* is to build the complete sweep from the individual sweep points which are delivered by the Spectran Interface.

```
#include <Spectran.h>
```

Public Member Functions

- SweepBuilder (SpectranInterface &interf)

  The SweepBuilder class's constructor.
- ∼SweepBuilder ()

  The SweepBuilder class's destructor.
- const Sweep & CaptureSweep (BandParameters &bandParam)

  The aim of this method is to capture one entire sweep from the spectrum analyzer through the Spectran Interface and returns this one.
- const Sweep & GetSweep () const

  This method returns the last captured sweep, as a Sweep structure.

5.22.1 Detailed Description

The aim of class *SweepBuilder* is to build the complete sweep from the individual sweep points which are delivered by the Spectran Interface.

5.22.2 Constructor & Destructor Documentation

5.22.2.1 SweepBuilder()

```
SweepBuilder::SweepBuilder (
            SpectranInterface & interf )  [inline]
```

The SweepBuilder class's constructor.

Parameters

`in`	*interf*	A reference to the unique *SpectranInterface* object, which is responsible for the communication with the spectrum analyzer.

5.22.2.2 ∼SweepBuilder()

```
SweepBuilder::∼SweepBuilder ( )  [inline]
```

The SweepBuilder class's destructor.

Its implementation is empty because the attributes are implicitly destroyed. However, the destructor is defined here to allow this one to be called explicitly in any part of the code, what is used by the signals handler to destroy the objects when a signal to finish the execution of the software is received.

5.22.3 Member Function Documentation

5.22.3.1 CaptureSweep()

```
const Sweep & SweepBuilder::CaptureSweep (
             BandParameters & bandParam )
```

The aim of this method is to capture one entire sweep from the spectrum analyzer through the Spectran Interface and returns this one.

The method receives a *BandParameters* structure, where the parameters of the current frequency band are stored, and it uses this structure to check if the frequency values are coherent and it corrects the number of sweep points of the structure.

First, the method sends a command to reset the current sweep, it waits a moment and then it enables the streaming of sweep points. Later, the method enters in a loop where each sweep point is captured and inserted in the `std::map` container. That kind of container are ordered and unique-key, so automatically the container orders the sweep points, taking into account the frequency, and it does not allow to insert two points with the same frequency. When that happens, the container states that and the loop finishes. Later, the number of sweep points is checked and stored in the given *BandParameters* structure, the streaming of sweep points is disabled and, finally, the sweep is moved to a *Sweep* structure, which is more optimum to perform mathematical operations, and this structure is returned.

Parameters

bandParam	[in,out] The parameters of the current frequency band.

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

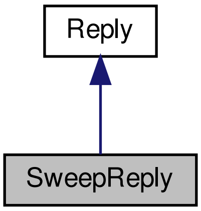
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Spectran.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/SweepBuilder.cpp

5.23 SweepReply Class Reference

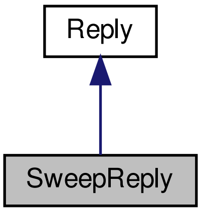
This class derives from the base class *Reply* and is intended to process in a better way replies with sweep points, i.e. *AMPFREQDAT* replies.

```
#include <Spectran.h>
```

Inheritance diagram for SweepReply:



Collaboration diagram for SweepReply:



Public Member Functions

- SweepReply ()

  The default constructor.
- SweepReply (const std::uint8_t ∗bytesPtr)

  A more complete constructor which allows to insert the received bytes of a reply.
- void PrepareTo (const ReplyType type, const SpecVariable variable=SpecVariable::UNINITIALIZED)

  An overloading of the method `PrepareTo()` which just leave it without effect because this method has no sense in this derived class.
- void InsertBytes (const std::uint8_t ∗data)

  An overloading of the corresponding method of the base class, which allows to insert the bytes of a AMPFREQDAT reply and to extract its data.
- unsigned int GetTimestamp () const

This method returns the timestamp of the corresponding sweep point.

- std::uint_least64_t GetFrequency () const

  This method returns the frequency value, which is in Hz.

- float GetMinValue () const

  This method returns the minimum power value, in case of Min/Max detector is used, or the RMS power value, in case of RMS detector is used. Even, it could be a voltage or field strength value.

- float GetMaxValue () const

  This method returns the maximum power value, in case of Min/Max detector is used, or the RMS power value, in case of RMS detector is used. Even, it could be a voltage or field strength value.

- void Clear ()

  The aim of this method is to clear the class attributes.

- const SweepReply & operator= (const SweepReply &anotherReply)

  An overloading of the assignment operator, adapted to this class.

Additional Inherited Members

5.23.1 Detailed Description

This class derives from the base class *Reply* and is intended to process in a better way replies with sweep points, i.e. *AMPFREQDAT* replies.

The purpose of this class is to handle the *AMPFREQDAT* replies which carry the sweep points. These specific replies need to be handled in a more complex way, so to simplify the base class ,which handles the others replies, a different class was made with the specific methods to extract the data from the AMPFREQDAT replies.

5.23.2 Constructor & Destructor Documentation

5.23.2.1 SweepReply()

```
SweepReply::SweepReply ( )
```

The default constructor.

This constructor clears the class attributes and calls the *Reply* constructor with an argument to set the reply type to AMPFREQDAT.

5.23.3 Member Function Documentation

5.23.3.1 InsertBytes()

```
void SweepReply::InsertBytes (
             const std::uint8_t * data )  [virtual]
```

An overloading of the corresponding method of the base class, which allows to insert the bytes of a AMPFREQDAT reply and to extract its data.

This method extracts the timestamp, frequency value and power values of a AMPFREQDAT reply.

Parameters

in	*data*	A pointer to the bytes which contains the timestamp and the frequency and power values.

Timestamp extraction: this value is received as a 4-byte unsigned integer and it represents the count of a internal timer of the spectrum analyzer. The timer period is approximately 3.5 nS and it is a 32-bit timer.

Frequency extraction: this value is received as a 4-byte unsigned integer in Hz/10 and it is stored as an unsigned integer in Hz.

Min/RMS power extraction: this value is received as a 4-byte floating point value, measured in dBm. Even, it could be a voltage value or a field strength value.

Max/RMS power extraction: this value is received as a 4-bytes floating point value, measured in dBm. Even, it could be a voltage value or a field strength value.

Reimplemented from Reply.

5.23.3.2 operator=()

```
const SweepReply & SweepReply::operator= (
            const SweepReply & anotherReply )
```

An overloading of the assignment operator, adapted to this class.

Parameters

in	*anotherReply*	A *Reply* object which is given to copy its attributes.

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

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Spectran.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Reply.cpp

5.24 TimeData Struct Reference

This structure is intended to store data related to *date* and *time* and to perform some operations with that data.

```
#include <Basics.h>
```

Public Member Functions

- TimeData ()

    The class' constructor which clear all attributes, i.e. set date and time as 00-00-0000T00:00:00.
- TimeData (const TimeData &timeData)

    The class' copy constructor.

- std::string GetDate () const

  A method to get the date as a `std::string`
- std::string GetTime () const

  A method to get the time as a `std::string`
- std::string GetTimestamp () const

  A method to get a timestamp as a `std::string` with the format DD-MM-YYYYTHH:MM:SS.
- void SetDate (const std::string &date)

  This method is intended to set just the date.
- void SetTime (const std::string &time)

  This method is intended to set just the time.
- void SetTimestamp (const std::string &timestamp)

  This method is intended to modify all attributes, giving these as a timestamp.
- void TurnBackDays (const unsigned int days)

  This method allows to turn a time point back a specified number of days.
- const TimeData & operator= (const TimeData &anotherTimeData)

  An overloading of the assignment operator.
- void Clear ()

  A method to clear all attributes, i.e. it set the object as 00-00-00T00:00:00.

Public Attributes

- unsigned int year

  This variable stores the year, taking into account the estimated birth of Jesus.
- unsigned int month

  This variable stores the month as a number that can be between 1 and 12.
- unsigned int day

  This variable stores the day as a number that can be between 1 and 31, taking into account the corresponding month.
- unsigned int hour

  This variable stores the hours as a number that can be between 0 and 23.
- unsigned int minute

  This variable stores the minutes as a number that can be between 0 and 59.
- unsigned int second

  This variable stores the seconds as a number that can be between 0 and 59.

Friends

- bool operator< (const TimeData &lhs, const TimeData &rhs)

  An overloading of the operator < to compare two TimeData objects as the first one lesser than the second one.
- bool operator> (const TimeData &lhs, const TimeData &rhs)

  An overloading of the operator > to compare two TimeData objects as the first one bigger than the second one.
- bool operator== (const TimeData &lhs, const TimeData &rhs)

  An overloading of the operator == to compare two TimeData objects as equal.

5.24.1 Detailed Description

This structure is intended to store data related to *date* and *time* and to perform some operations with that data.

This structure can store time data (hour, minute and second) and date data (year, month and date) which are obtained from a GPS receiver. It can offer just the date as a string, or just the time as a string or even can build a *timestamp* with a specific format: DD-MM-YYYYTHH:MM:SS. Also it can take a date and time and turn that back a specific number of days taking into account the *gregorian* calendar. An object of this class can even be compared as equal to, less than or bigger than another object of the same class.

5.24.2 Constructor & Destructor Documentation

5.24.2.1 TimeData()

```
TimeData::TimeData (
            const TimeData & timeData )  [inline]
```

The class' copy constructor.

Parameters

in	*timeData*	Another *TimeData* object which is given to copy its attributes.

5.24.3 Member Function Documentation

5.24.3.1 operator=()

```
const TimeData & TimeData::operator= (
            const TimeData & anotherTimeData )
```

An overloading of the assignment operator.

Parameters

in	*anotherTimeData*	Another *TimeData* object which is given to copy its attributes.

5.24.3.2 SetDate()

```
void TimeData::SetDate (
            const std::string & date )
```

This method is intended to set just the date.

Parameters

in	*date*	A date given as a `std::string` object, or even it can be inserted as `char` pointer.

5.24.3.3 SetTime()

```
void TimeData::SetTime (
             const std::string & time )
```

This method is intended to set just the time.

Parameters

in	*time*	A time given as a `std::string` object, or even it can be inserted as `char` pointer.

5.24.3.4 SetTimestamp()

```
void TimeData::SetTimestamp (
             const std::string & timestamp )
```

This method is intended to modify all attributes, giving these as a timestamp.

Parameters

in	*timestamp*	A timestamp given as a `std::string` object, or even it can be inserted as `char` pointer.

5.24.3.5 TurnBackDays()

```
void TimeData::TurnBackDays (
             const unsigned int days )
```

This method allows to turn a time point back a specified number of days.

Parameters

in	*days*	The number of days which must be used to turn back the current date.

5.24.4 Friends And Related Function Documentation

5.24.4.1 operator<

```
bool operator< (
             const TimeData & lhs,
             const TimeData & rhs )  [friend]
```

An overloading of the operator $<$ to compare two TimeData objects as the first one lesser than the second one.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.24.4.2 operator==

```
bool operator== (
            const TimeData & lhs,
            const TimeData & rhs )  [friend]
```

An overloading of the operator == to compare two TimeData objects as equal.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

5.24.4.3 operator>

```
bool operator> (
            const TimeData & lhs,
            const TimeData & rhs )  [friend]
```

An overloading of the operator > to compare two TimeData objects as the first one bigger than the second one.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

The documentation for this struct was generated from the following files:

- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Basics.h
- /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/TimeData.cpp

Chapter 6

File Documentation

6.1 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/scripts/client.py File Reference

The aim of this script is to upload the data collected by the software 'rfims-cart' (which calls this script).

Functions

- def client.Internet_on ()

  This function checks if there is Internet connection.
- def client.SendData (file_path)

  This function sends the given file to the remote server, using scp.
- def client.ServerWakeUp (file_name)

  This function wakes up the remote server to that one processes the sent file.
- def client.NameExtractor (file_path)

  This function extracts the filename from the script argument.

Variables

- string client.HOST = ""

  The public IP of the MV Amazon remote server.
- string client.USERNAME = ""

  The username which must be used to log in the remote server.
- string client.PASSWORD = ""

  The password which must be used to log in the remote server.
- int client.PORT = 22

  The port through which the data file is sent, using scp.
- string client.REMOTE_FOLDER = "/home/server/RFIMS-CART/downloads/"

  The remote folder where the file is stored.
- string client.SERVER_SCRIPT_PATH = "/home/server/RFIMS-CART/Server2.py"

  The remote path to the sript which must be executed to wake up the server.
- int client.TRIES = 6

  The number of times the Internet connection is checked, with an interval time between each try.
- int client.WAIT_SECONDS = 600

The interval time which is waited between two tries of checking Internet connection, expressed in seconds (s).

- int client.WAIT_MINS = WAIT_SECONDS / 60

  The interval time which is waited between two tries of checking Internet connection, expressed in minutes.
- client.file_path = sys.argv[1]

  The script argument which contains the path and name of the file to be sent.
- def client.file_name = NameExtractor(file_path)

  The name of the file to be sent.

6.1.1 Detailed Description

The aim of this script is to upload the data collected by the software 'rfims-cart' (which calls this script).

First, the script checks many times if there is Internet connection, through one hour, if so then it sends a compressed archive file with the rfims-cart data to the remote server, using scp, later it wakes up the server and, finally, it deletes the local archive file. If any error occurs the local archive file is not removed and it remains in a queue (in the software rfims-data) waiting to be uploaded. The path and name of the file to be sent is passed as an argument.

Author

Leandro Saldivar

6.1.2 Function Documentation

6.1.2.1 Internet_on()

```
def client.Internet_on ( )
```

This function checks if there is Internet connection.

Returns

A `true` value is returned if theres is Internet connection and a `false` value otherwise.

6.1.2.2 NameExtractor()

```
def client.NameExtractor (
              file_path )
```

This function extracts the filename from the script argument.

Returns

The name of the file to be sent.

6.1.2.3 SendData()

```
def client.SendData (
              file_path )
```

This function sends the given file to the remote server, using scp.

Returns

A `true` value is returned if the operation finished successfully, and a `false` value otherwise.

6.1.2.4 ServerWakeUp()

```
def client.ServerWakeUp (
              file_name )
```

This function wakes up the remote server to that one processes the sent file.

Returns

A `true` value is returned if the operation finished successfully, and a `false` value otherwise.

6.1.3 Variable Documentation

6.1.3.1 file_name

```
def client.file_name = NameExtractor(file_path)
```

The name of the file to be sent.

6.1.3.2 file_path

```
client.file_path = sys.argv[1]
```

The script argument which contains the path and name of the file to be sent.

6.1.3.3 HOST

```
string client.HOST = ""
```

The public IP of the MV Amazon remote server.

6.1.3.4 PASSWORD

```
string client.PASSWORD = ""
```

The password which must be used to log in the remote server.

6.1.3.5 PORT

```
int client.PORT = 22
```

The port through which the data file is sent, using scp.

6.1.3.6 REMOTE_FOLDER

```
string client.REMOTE_FOLDER = "/home/server/RFIMS-CART/downloads/"
```

The remote folder where the file is stored.

6.1.3.7 SERVER_SCRIPT_PATH

```
string client.SERVER_SCRIPT_PATH = "/home/server/RFIMS-CART/Server2.py"
```

The remote path to the sript which must be executed to wake up the server.

6.1.3.8 TRIES

```
int client.TRIES = 6
```

The number of times the Internet connection is checked, with an interval time between each try.

6.1.3.9 USERNAME

```
string client.USERNAME = ""
```

The username which must be used to log in the remote server.

6.1.3.10 WAIT_MINS

```
int client.WAIT_MINS = WAIT_SECONDS / 60
```

The interval time which is waited between two tries of checking Internet connection, expressed in minutes.

6.1.3.11 WAIT_SECONDS

```
int client.WAIT_SECONDS = 600
```

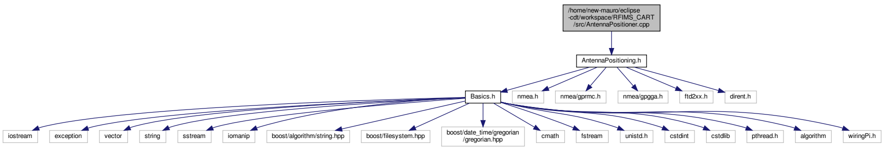
The interval time which is waited between two tries of checking Internet connection, expressed in seconds (s).

6.2 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/AntennaPositioner.cpp File Reference

This file contains the definitions of several methods of the class *AntennaPositioner*.

```
#include "AntennaPositioning.h"
```

Include dependency graph for AntennaPositioner.cpp:



Functions

- void **canalA** ()
- void **canalB** ()

6.2.1 Detailed Description

This file contains the definitions of several methods of the class *AntennaPositioner*.

Author

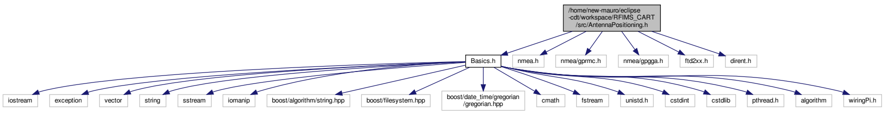
Emanuel Asencio

6.3 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/AntennaPositioning.h File Reference

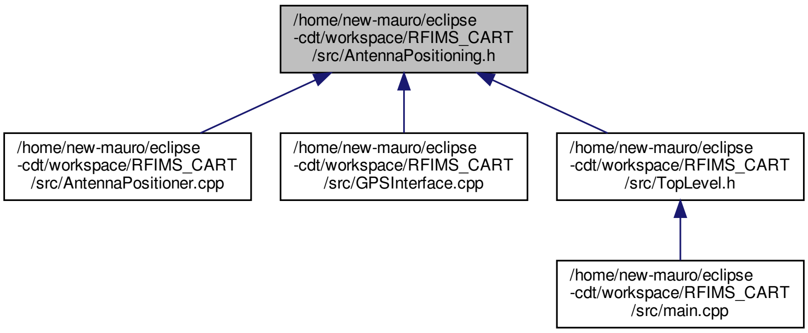
This file contains the declarations of classes AntennaPositioner and GPSInterface.

```
#include "Basics.h"
#include <nmea.h>
#include <nmea/gprmc.h>
#include <nmea/gpgga.h>
#include <ftd2xx.h>
#include <dirent.h>
```

Include dependency graph for AntennaPositioning.h:



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


/home/new-mauro/eclipse
-cdt/workspace/RFIMS_CART
/src/AntennaPositioning.h
/home/new-mauro/eclipse
-cdt/workspace/RFIMS_CART
/src/AntennaPositioner.cpp
/home/new-mauro/eclipse
-cdt/workspace/RFIMS_CART
/src/GPSInterface.cpp
/home/new-mauro/eclipse
-cdt/workspace/RFIMS_CART
/src/TopLevel.h
/home/new-mauro/eclipse
-cdt/workspace/RFIMS_CART
/src/main.cpp

Classes

- struct Data3D

  A structure intended to save the the values of the 3d sensors which are integrated in the GPS receiver.
- struct GPSCoordinates

  A structure which saves the GPS coordinates.
- class GPSInterface

  The class GPSInterace is intended to establish the communication with the Aaronia GPS receiver, to request and capture messages from this one and extract useful data from the messages.
- class AntennaPositioner

  The aim of the class AntennaPositioner is to handle the antenna positioning system.

Enumerations

- enum Polarization : char { **HORIZONTAL** =0, **VERTICAL** =1, **UNKNOWN** }

  An enumeration which contains the possible states of the antenna polarization: HORIZONTAL, VERTICAL or UNKNOWN.

6.3.1 Detailed Description

This file contains the declarations of classes AntennaPositioner and GPSInterface.

These classes are responsible for the positioning of the antenna, which can rotate its azimuth angle and its polarization. To model these rotations it were used the Cardan angles (or nautical angles): the "yaw" angle which represents the heading, the "pitch" angle which represents the elevation and the "roll" angle which models the bank angle. The yaw angle is used to represent the antenna's azimuth angle, the roll angle is used to represent the antenna polarization and the pitch angle is not used.

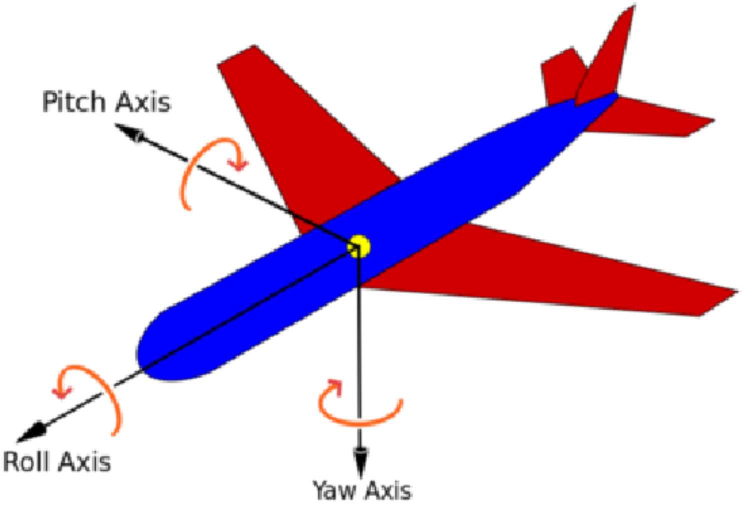


Figure 6.1 The Cardan or nautical angles

Moreover, the class GPSInterface allows to get data related with time and date, elevation (based on pressure), ambient pressure, among other things.

Also, several structures are defined here which store data related with the mentioned classes.

Author

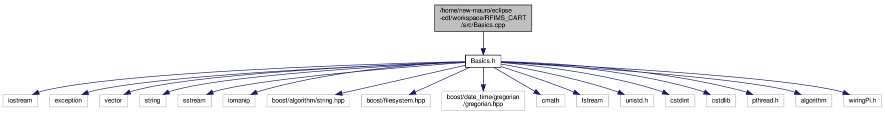
Mauro Diamantino

6.4 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Basics.cpp File Reference

This file contains the definitions of the functions and classes' methods which have been declared in file Basics.h.

```
#include "Basics.h"
```

Include dependency graph for Basics.cpp:



Functions

- bool approximatelyEqual (float a, float b)

  Function intended to compare floating-point numbers as approximately equal, taking into account the floating-point rounding errors.
- bool approximatelyEqual (std::vector< float > vectorA, std::vector< float > vectorB)

  Function intended to compare `std::vector<float>` containers as approximately equal, taking into account the floating-point rounding errors.
- void WaitForEnter ()

  This function stop the execution until any key is pressed by the user and it was used for debugging purpose.
- std::vector< FreqValues::value_type > operator- (const std::vector< FreqValues::value_type > &vect)

  An overloading of unary operator - which negates the elements of a `std::vector<float>` container.
- void InitializeGPIO ()

  This function initializes all GPIO pins which are used for the input and output signals, in the way it is described in structure piPins.
- void TurnOffLeds ()

  The aim of this function is to turn all LEDs off.
- void TurnOnFrontEnd ()

  The aim of this function is to turn on the RF front-end elements in a sequential manner, from the spectrum analyzer to the antenna..
- void TurnOffFrontEnd ()

  The aim of this function is to turn off the RF front-end elements in a sequential manner, from the antenna to the spectrum analyzer.

6.4.1 Detailed Description

This file contains the definitions of the functions and classes' methods which have been declared in file Basics.h.

Author

Mauro Diamantino

6.4.2 Function Documentation

6.4.2.1 approximatelyEqual() [1/2]

```
bool approximatelyEqual (
            float a,
            float b )
```

Function intended to compare floating-point numbers as approximately equal, taking into account the floating-point rounding errors.

This function was copied from this `link`, but it was modified. It is based in the Knuth's algorithm but it uses two epsilons, an absolute epsilon (ABS_EPSILON) which is very small and is intended to compare near-zero floating-point numbers and a relative epsilon (REL_EPSILON, which is a percentage of the biggest operand) to compare the rest of the floating-point numbers. The function returns true if the difference between a and b is less than ABS_EPSILON, or within REL_EPSILON percent of the larger of a and b.

Parameters

in	*a*	The left-hand side argument.
in	*b*	The right-hand side argument.

6.4.2.2 approximatelyEqual() [2/2]

```
bool approximatelyEqual (
            std::vector< float > vectorA,
            std::vector< float > vectorB )
```

Function intended to compare `std::vector<float>` containers as approximately equal, taking into account the floating-point rounding errors.

This function is based on the function presented in this `link`.

It is based in the Knuth's algorithm but it uses two epsilons, an absolute epsilon (ABS_EPSILON) which is very small and is intended to compare near-zero floating-point numbers and a relative epsilon (REL_EPSILON, which is a percentage of the biggest operand) to compare the rest of the floating-point numbers. The function returns true if the difference between a and b is less than ABS_EPSILON, or within REL_EPSILON percent of the larger of a and b.

Parameters

in	*vectorA*	The left-hand side argument.
in	*vectorB*	The right-hand side argument.

6.4.2.3 operator-()

```
std::vector<FreqValues::value_type> operator- (
            const std::vector< FreqValues::value_type > & vect )
```

An overloading of unary operator - which negates the elements of a `std::vector<float>` container.

Parameters

in	*vect*	A `std::vector<float>` container which must be negated.

6.4.2.4 TurnOffFrontEnd()

```
void TurnOffFrontEnd ( )
```

The aim of this function is to turn off the RF front-end elements in a sequential manner, from the antenna to the spectrum analyzer.

To turn off the RF front-end elements sequentially, this function begins ensuring the noise source is turned off, then it switches the input to that device, it waits 1 second, it turns the LNAs off, it waits again 1 second and, finally, it turns the spectrum analyzer off.

6.4.2.5 TurnOnFrontEnd()

```
void TurnOnFrontEnd ( )
```

The aim of this function is to turn on the RF front-end elements in a sequential manner, from the spectrum analyzer to the antenna..

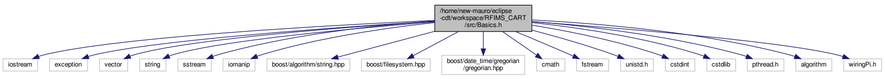
To turn on the RF front-end elements in a sequential manner, first, this function turns the spectrum analyzer on, then it waits 5 seconds, it turns the LNAs on, it waits again but this time just 1 second and, finally, it switches the input to the antenna.

6.5 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Basics.h File Reference

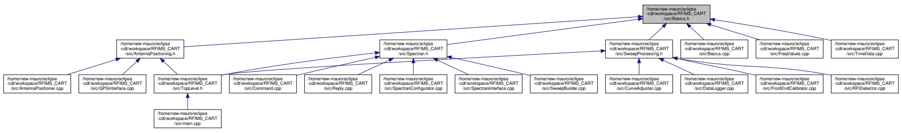
This header file contains the declarations of the most basic and global entities which are used by many others entities.

```
#include <iostream>
#include <exception>
#include <vector>
#include <string>
#include <sstream>
#include <iomanip>
#include <boost/algorithm/string.hpp>
#include <boost/filesystem.hpp>
#include <boost/date_time/gregorian/gregorian.hpp>
#include <cmath>
#include <fstream>
#include <unistd.h>
#include <cstdint>
#include <cstdlib>
#include <pthread.h>
#include <algorithm>
#include <wiringPi.h>
```

Include dependency graph for Basics.h:



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



Classes

- class rfims_exception

  A class derived from standard class `std::exception`.
- struct TimeData

  This structure is intended to store data related to date and time and to perform some operations with that data.
- struct FreqValues

  The aim of this structure is to store the curve of a determined parameter or variable versus the frequency, which is named a frequency curve here.
- struct Sweep

  The aim of this structure is to store the data points of a sweep obtained with the spectrum analyzer in a determined azimuth position, with a specific polarization.
- struct RFI

  The aim of this structure is to store the data related with the detected RF interference (RFI): frequency, power, azimuth angle, polarization, time, reference norm, etc.
- struct BandParameters

  This structure is intended to store the parameters which are used to configure the spectrum analyzer in each frequency band.

Functions

- bool approximatelyEqual (float a, float b)

  Function intended to compare floating-point numbers as approximately equal, taking into account the floating-point rounding errors.
- bool approximatelyEqual (std::vector< float > vectorA, std::vector< float > vectorB)

  Function intended to compare `std::vector<float>` containers as approximately equal, taking into account the floating-point rounding errors.
- std::vector< FreqValues::value_type > operator- (const std::vector< FreqValues::value_type > &vect)

  An overloading of unary operator - which negates the elements of a `std::vector<float>` container.
- void WaitForEnter ()

  This function stop the execution until any key is pressed by the user and it was used for debugging purpose.
- void InitializeGPIO ()

  This function initializes all GPIO pins which are used for the input and output signals, in the way it is described in structure piPins.
- void TurnOffLeds ()

  The aim of this function is to turn all LEDs off.
- void TurnOnFrontEnd ()

  The aim of this function is to turn on the RF front-end elements in a sequential manner, from the spectrum analyzer to the antenna..
- void TurnOffFrontEnd ()

  The aim of this function is to turn off the RF front-end elements in a sequential manner, from the antenna to the spectrum analyzer.

Variables

- const std::string BASE_PATH = "/home/pi/RFIMS-CART"

  This constant define the base path where there are the files and folders used by the software.
-

```
struct {
  const unsigned int SWITCH = 10
    This pin is initialized as an output in LOW state, so the RF switch output will start connected to the noise source.
  const unsigned int NOISE_SOURCE = 6
    This pin is initialized as an output in LOW state, so the noise source will start turned off.
  const unsigned int LNAS = 26
    This pin is initialized as an output in LOW state, so the LNAs will start turned off.
  const unsigned int SPECTRAN = 27
    This pin is initialized as an input with the pull-down resistor enabled, so the Spectran device will start turned off.
  const unsigned int LED_SWEEP_CAPTURE = 0
    This pin is initialized as an output in LOW state, so the led will start turned off.
  const unsigned int LED_SWEEP_PROCESS = 2
    This pin is initialized as an output in LOW state, so the led will start turned off.
  const unsigned int LED_INIT_POS = 12
    This pin is initialized as an output in LOW state, so the led will start turned off.
  const unsigned int LED_NEXT_POS = 12
    This pin is initialized as an output in LOW state, so the led will start turned off.
  const unsigned int LED_POLARIZ = 3
    This pin is initialized as an output in LOW state, so the led will start turned off.
  const unsigned int LED_ERROR = 13
  const unsigned int BUTTON_ENTER = 8
    This pin is initialized as in input wit the pull-up resistor enabled, so the button must connect the pin to GND.
  const unsigned int PUL = 28
  const unsigned int DIRECCION = 31
  const unsigned int EN = 29
  const unsigned int SENSOR_NORTE = 30
  const unsigned int POL = 5
  const unsigned int FASE_A = 22
  const unsigned int FASE_B = 21
} piPins
```

This structure is intended to store the assignment of GPIO pins to input and output external signals, taking into account the Wiring Pi numbering.

-

```
struct {
  const int SWITCH_TO_NS = HIGH
    This constant define the value the switch's pin must take to connect noise source to input, provided that the noise source i
  const int SWITCH_TO_ANT = !SWITCH_TO_NS
    This constant define the value the switch's pin must take to connect antenna to input, provided that the antenna is connec
  const int NS_ON = HIGH
  const int NS_OFF = !NS_ON
  const int LNAS_ON = HIGH
  const int LNAS_OFF = !LNAS_ON
  const int SPECTRAN_ON = HIGH
  const int SPECTRAN_OFF = !SPECTRAN_ON
  const int LED_SWP_CAPT_ON = HIGH
  const int LED_SWP_CAPT_OFF = !LED_SWP_CAPT_ON
  const int LED_SWP_PROC_ON = HIGH
  const int LED_SWP_PROC_OFF = !LED_SWP_PROC_ON
  const int LED_INIT_POS_ON = HIGH
  const int LED_INIT_POS_OFF = !LED_INIT_POS_ON
  const int LED_NEXT_POS_ON = HIGH
  const int LED_NEXT_POS_OFF = !LED_NEXT_POS_ON
  const int LED_POL_ON = HIGH
  const int LED_POL_OFF = !LED_POL_ON
  const int LED_ERROR_ON = HIGH
  const int LED_ERROR_OFF = !LED_ERROR_ON
  const int BUTTON_ON = LOW
```

const int **BUTTON_OFF** = !BUTTON_ON
const int **PUL_ON** = HIGH
const int **PUL_OFF** = !PUL_ON
const int **DIR_ANTIHOR** = HIGH
const int **DIR_HOR** = !DIR_ANTIHOR
const int **EN_ON** = HIGH
const int **EN_OFF** = !EN_ON
const int **SENS_NOR_ON** = LOW
const int **SENS_NOR_OFF** = !SENS_NOR_ON
const int **POL_HOR** = LOW
const int **POL_VERT** = !POL_HOR
const int **FASE_A_ON** = HIGH
const int **FASE_A_OFF** = !FASE_A_ON
const int **FASE_B_ON** = HIGH
const int **FASE_B_OFF** = !FASE_B_ON
} pinsValues

This structure contains the digital pins' assertive values (HIGH or LOW).

6.5.1 Detailed Description

This header file contains the declarations of the most basic and global entities which are used by many others entities.

This header file contains the declarations and definitions of the following:

- Preprocessor definitions (`define`): these elements define the blocks of code will be taking into account by the compiler, so the way the software will behave after the compilation and linking.
- Global libraries, i.e. the libraries which are used by several functions and classes from different files.
- Global constants and variables.
- Global structures which are used to interchange data between different objects and functions.
- Global structures which contain common data to all software's entities.
- A global class to handle exceptions.
- Global functions which are used in several different places.

Author

Mauro Diamantino

6.5.2 Function Documentation

6.5.2.1 approximatelyEqual() [1/2]

```
bool approximatelyEqual (
            float a,
            float b )
```

Function intended to compare floating-point numbers as approximately equal, taking into account the floating-point rounding errors.

This function was copied from this link, but it was modified. It is based in the Knuth's algorithm but it uses two epsilons, an absolute epsilon (ABS_EPSILON) which is very small and is intended to compare near-zero floating-point numbers and a relative epsilon (REL_EPSILON, which is a percentage of the biggest operand) to compare the rest of the floating-point numbers. The function returns true if the difference between a and b is less than ABS_EPSILON, or within REL_EPSILON percent of the larger of a and b.

Parameters

in	*a*	The left-hand side argument.
in	*b*	The right-hand side argument.

6.5.2.2 approximatelyEqual() [2/2]

```
bool approximatelyEqual (
            std::vector< float > vectorA,
            std::vector< float > vectorB )
```

Function intended to compare `std::vector<float>` containers as approximately equal, taking into account the floating-point rounding errors.

This function is based on the function presented in this `link`.

It is based in the Knuth's algorithm but it uses two epsilons, an absolute epsilon (ABS_EPSILON) which is very small and is intended to compare near-zero floating-point numbers and a relative epsilon (REL_EPSILON, which is a percentage of the biggest operand) to compare the rest of the floating-point numbers. The function returns true if the difference between a and b is less than ABS_EPSILON, or within REL_EPSILON percent of the larger of a and b.

Parameters

in	*vectorA*	The left-hand side argument.
in	*vectorB*	The right-hand side argument.

6.5.2.3 operator-()

```
std::vector<FreqValues::value_type> operator- (
            const std::vector< FreqValues::value_type > & vect )
```

An overloading of unary operator - which negates the elements of a `std::vector<float>` container.

Parameters

in	*vect*	A `std::vector<float>` container which must be negated.

6.5.2.4 TurnOffFrontEnd()

```
void TurnOffFrontEnd ( )
```

The aim of this function is to turn off the RF front-end elements in a sequential manner, from the antenna to the spectrum analyzer.

To turn off the RF front-end elements sequentially, this function begins ensuring the noise source is turned off, then it switches the input to that device, it waits 1 second, it turns the LNAs off, it waits again 1 second and, finally, it turns the spectrum analyzer off.

6.5.2.5 TurnOnFrontEnd()

```
void TurnOnFrontEnd ( )
```

The aim of this function is to turn on the RF front-end elements in a sequential manner, from the spectrum analyzer to the antenna..

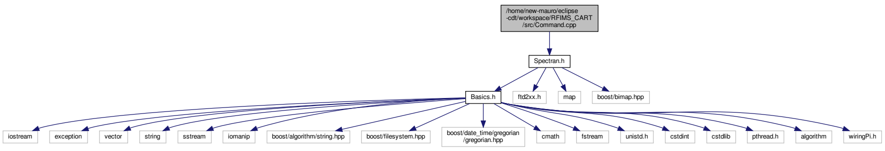
To turn on the RF front-end elements in a sequential manner, first, this function turns the spectrum analyzer on, then it waits 5 seconds, it turns the LNAs on, it waits again but this time just 1 second and, finally, it switches the input to the antenna.

6.6 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Command.cpp File Reference

This file contains the definitions of several methods of the class *Command*.

```
#include "Spectran.h"
```

Include dependency graph for Command.cpp:



6.6.1 Detailed Description

This file contains the definitions of several methods of the class *Command*.

Author

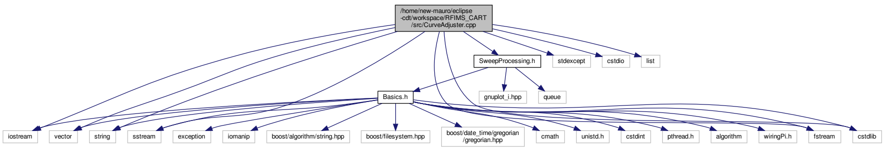
Mauro Diamantino

6.7 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/CurveAdjuster.cpp File Reference

This file contains the definitions of several methods of the class *CurveAdjuster*.

```
#include "SweepProcessing.h"
```

Include dependency graph for CurveAdjuster.cpp:



6.7.1 Detailed Description

This file contains the definitions of several methods of the class *CurveAdjuster*.

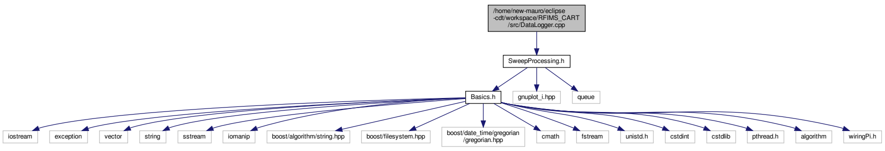
Author

Mauro Diamantino

6.8 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/DataLogger.cpp File Reference

This file contains the definitions of several methods of the class *DataLogger*.

`#include "SweepProcessing.h"`
Include dependency graph for DataLogger.cpp:



Functions

- void ∗ UploadThreadFunc (void ∗arg)

  The function which is executed by the thread which is responsible for the concurrent uploading of the data files.

6.8.1 Detailed Description

This file contains the definitions of several methods of the class *DataLogger*.

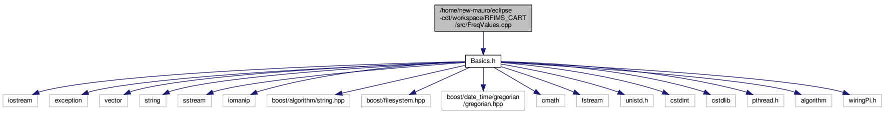
Author

Mauro Diamantino

6.9 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/FreqValues.cpp File Reference

This file contains the definitions of several methods of the structure *FreqValues* and its derived structures, and the functions related to theses ones.

`#include "Basics.h"`
Include dependency graph for FreqValues.cpp:



Functions

- FreqValues operator- (const FreqValues &argument)
- FreqValues operator+ (const FreqValues &lhs, const FreqValues &rhs)
- FreqValues operator+ (const FreqValues &lhs, const double rhs)
- FreqValues **operator+** (const FreqValues &lhs, const float rhs)
- FreqValues operator+ (const double lhs, const FreqValues &rhs)
- FreqValues **operator+** (const float lhs, const FreqValues &rhs)
- FreqValues operator- (const FreqValues &lhs, const FreqValues &rhs)
- FreqValues operator- (const FreqValues &lhs, const double rhs)
- FreqValues **operator-** (const FreqValues &lhs, const float rhs)
- FreqValues operator- (const double lhs, const FreqValues &rhs)
- FreqValues **operator-** (const float lhs, const FreqValues &rhs)
- FreqValues operator∗ (const FreqValues &lhs, const FreqValues &rhs)
- FreqValues operator∗ (const double lhs, const FreqValues &rhs)
- FreqValues **operator**∗ (const float lhs, const FreqValues &rhs)
- FreqValues operator∗ (const FreqValues &lhs, const double rhs)
- FreqValues **operator**∗ (const FreqValues &lhs, const float rhs)
- FreqValues operator/ (const FreqValues &lhs, const FreqValues &rhs)
- FreqValues operator/ (const double lhs, const FreqValues &rhs)
- FreqValues **operator**/ (const float lhs, const FreqValues &rhs)
- FreqValues operator/ (const FreqValues &lhs, const double rhs)
- FreqValues **operator**/ (const FreqValues &lhs, const float rhs)
- FreqValues log10 (const FreqValues &argument)
- FreqValues pow (const FreqValues &base, const double exponent)
- FreqValues **pow** (const FreqValues &base, const float exponent)
- FreqValues pow (const double base, const FreqValues &exponent)
- FreqValues **pow** (const float base, const FreqValues &exponent)
- Sweep operator- (const Sweep &argument)
- Sweep operator+ (const Sweep &lhs, const Sweep &rhs)
- Sweep operator+ (const Sweep &lhs, const std::vector< FreqValues::value_type > &rhs)
- Sweep operator+ (const std::vector< FreqValues::value_type > &lhs, const Sweep &rhs)
- Sweep operator+ (const Sweep &lhs, const FreqValues &rhs)
- Sweep operator+ (const Sweep &lhs, const double rhs)
- Sweep **operator+** (const Sweep &lhs, const float rhs)
- Sweep operator+ (const double lhs, const Sweep &rhs)
- Sweep **operator+** (const float lhs, const Sweep &rhs)
- Sweep operator- (const Sweep &lhs, const Sweep &rhs)
- Sweep operator- (const Sweep &lhs, const std::vector< FreqValues::value_type > &rhs)
- Sweep operator- (const std::vector< FreqValues::value_type > &lhs, const Sweep &rhs)
- Sweep operator- (const Sweep &lhs, const FreqValues &rhs)
- Sweep operator∗ (const Sweep &lhs, const Sweep &rhs)
- Sweep operator∗ (const double lhs, const Sweep &rhs)
- Sweep **operator**∗ (const float lhs, const Sweep &rhs)
- Sweep operator∗ (const Sweep &lhs, const double rhs)
- Sweep **operator**∗ (const Sweep &lhs, const float rhs)
- Sweep operator/ (const Sweep &lhs, const Sweep &rhs)
- Sweep operator/ (const double lhs, const Sweep &rhs)
- Sweep **operator**/ (const float lhs, const Sweep &rhs)
- Sweep operator/ (const Sweep &lhs, const double rhs)
- Sweep **operator**/ (const Sweep &lhs, const float rhs)
- Sweep log10 (const Sweep &argument)
- Sweep pow (const Sweep &base, const double exponent)
- Sweep **pow** (const Sweep &base, const float exponent)
- Sweep pow (const double base, const Sweep &exponent)
- Sweep **pow** (const float base, const Sweep &exponent)

6.9.1 Detailed Description

This file contains the definitions of several methods of the structure *FreqValues* and its derived structures, and the functions related to theses ones.

Author

Mauro Diamantino

6.9.2 Function Documentation

6.9.2.1 log10() [1/2]

```
FreqValues log10 (
            const FreqValues & argument )
```

This function takes each point of the given structure, apply the decimal logarithm whit its value and stores the result in a different *FreqValues* structure, which is then returned. The rest of attributes (frequency, type, timestamp, etc.) are copied as-is to the object to be returned.

Parameters

in	*argument*	The *FreqValues* structure to be used as argument to the decimal logarithm operation.

6.9.2.2 log10() [2/2]

```
Sweep log10 (
            const Sweep & argument )
```

This function takes each point of the given structure, apply the decimal logarithm whit its value and stores the result in a different *Sweep* structure, which is then returned. The rest of attributes (frequency, type, timestamp, etc.) are copied as-is to the object to be returned.

Parameters

in	*argument*	The *Sweep* structure to be used as argument to the decimal logarithm operation.

6.9.2.3 operator∗() [1/6]

```
FreqValues operator* (
            const FreqValues & lhs,
            const FreqValues & rhs )
```

Before performing the operation, the function checks if the frequencies of each structure match and if the "values" vectors have the same sizes as the "frequencies" vectors. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.4 operator*() [2/6]

```
FreqValues operator* (
             const double lhs,
             const FreqValues & rhs )
```

The values of the object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the right-hand argument, which is the only argument that is of type *FreqValues*.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.5 operator*() [3/6]

```
FreqValues operator* (
             const FreqValues & lhs,
             const double rhs )
```

This function calls the function `operator*(float,FreqValues)` with the order of its argument inverted, taking into account the commutative property of the multiplication.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.6 operator*() [4/6]

```
Sweep operator* (
            const Sweep & lhs,
            const Sweep & rhs )
```

Before performing the operation, the function checks if the frequencies of each structure match and if the "values" vectors have the same sizes as the "frequencies" vectors. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.7 operator*() [5/6]

```
Sweep operator* (
            const double lhs,
            const Sweep & rhs )
```

The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the right-hand side argument, the only one argument of type *Sweep*.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.8 operator*() [6/6]

```
Sweep operator* (
            const Sweep & lhs,
            const double rhs )
```

The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument, the only one argument of type *Sweep*.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.9 operator+() [1/9]

```
FreqValues operator+ (
             const FreqValues & lhs,
             const FreqValues & rhs )
```

Before performing the operation, the function checks if the frequencies of each structure match and if the "values" vectors have the same sizes as the "frequencies" vectors. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.10 operator+() [2/9]

```
FreqValues operator+ (
             const FreqValues & lhs,
             const double rhs )
```

The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument, which is the only argument that is of type *FreqValues*.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.11 operator+() [3/9]

```
FreqValues operator+ (
            const double lhs,
            const FreqValues & rhs )
```

This function calls the function `operator+(FreqValues,float)` with the order of its arguments inverted, taking into account the commutative property of the sum. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the right-hand side argument, which is the only argument that is of type *FreqValues*.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.12 operator+() [4/9]

```
Sweep operator+ (
            const Sweep & lhs,
            const Sweep & rhs )
```

Before performing the operation, the function checks if the frequencies of each structure match and if the "values" vectors have the same sizes as the "frequencies" vectors. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.13 operator+() [5/9]

```
Sweep operator+ (
            const Sweep & lhs,
            const std::vector< FreqValues::value_type > & rhs )
```

Before performing the operation, the function checks if the "values" vectors have the same sizes. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.14 operator+() [6/9]

```
Sweep operator+ (
            const std::vector< FreqValues::value_type > & lhs,
            const Sweep & rhs )
```

Before performing the operation, the function checks if the "values" vectors have the same sizes. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.15 operator+() [7/9]

```
Sweep operator+ (
            const Sweep & lhs,
            const FreqValues & rhs )
```

To perform this operation the *FreqValues* argument is casted to *Sweep*. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument, because the other argument has less attributes as it is an object of the base class *FreqValues* from which the class *Sweep* derives.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.16 operator+() [8/9]

```
Sweep operator+ (
            const Sweep & lhs,
            const double rhs )
```

The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument, the only one argument of type *Sweep*.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.17 operator+() [9/9]

```
Sweep operator+ (
            const double lhs,
            const Sweep & rhs )
```

The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the right-hand side argument, the only one argument of type *Sweep*.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.18 operator-() [1/9]

```
FreqValues operator- (
            const FreqValues & argument )
```

This function takes each point of the given structure, negates it (the stored value, not the frequency) and stores the result in a different *FreqValues* structure, which is then returned. The rest of attributes (frequency, type, timestamp, etc.) are copied as-is to the object to be returned.

Parameters

in	*argument*	The *FreqValues* structure to be negated.

6.9.2.19 operator-() [2/9]

```
FreqValues operator- (
             const FreqValues & lhs,
             const FreqValues & rhs )
```

This function calls the function `operator+(FreqValues,FreqValues)` with the right-hand side operand negated.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.20 operator-() [3/9]

```
FreqValues operator- (
             const FreqValues & lhs,
             const double rhs )
```

This function calls the function `operator+(FreqValues,float)` with the right-hand side operand negated.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.21 operator-() [4/9]

```
FreqValues operator- (
             const double lhs,
             const FreqValues & rhs )
```

This function calls the function `operator+(float,FreqValues)` with the right-hand side operand negated.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.22 operator-() [5/9]

```
Sweep operator- (
            const Sweep & argument )
```

This function takes each point of the given structure, negates it (the stored value, not the frequency) and stores the result in a different *Sweep* structure, which is then returned. The rest of attributes (frequency, type, timestamp, etc.) are copied as-is to the object to be returned.

Parameters

in	*argument*	The *Sweep* structure to be negated.

6.9.2.23 operator-() [6/9]

```
Sweep operator- (
            const Sweep & lhs,
            const Sweep & rhs )
```

Before performing the operation, the function checks if the frequencies of each structure match and if the "values" vectors have the same sizes as the "frequencies" vectors. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.24 operator-() [7/9]

```
Sweep operator- (
            const Sweep & lhs,
            const std::vector< FreqValues::value_type > & rhs )
```

Before performing the operation, the function checks if the "values" vectors have the same sizes. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.25 operator-() [8/9]

```
Sweep operator- (
            const std::vector< FreqValues::value_type > & lhs,
            const Sweep & rhs )
```

Before performing the operation, the function checks if the "values" vectors have the same sizes. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the right-hand side argument.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.26 operator-() [9/9]

```
Sweep operator- (
            const Sweep & lhs,
            const FreqValues & rhs )
```

To perform this operation the *FreqValues* argument is casted to *Sweep*. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument, because the other argument has less attributes as it is an object of the base class *FreqValues* from which the class *Sweep* derives.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.27 operator/() [1/6]

```
FreqValues operator/ (
            const FreqValues & lhs,
            const FreqValues & rhs )
```

Before performing the operation, the function checks if the frequencies of each structure match and if the "values" vectors have the same sizes as the "frequencies" vectors. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.28 operator/() [2/6]

```
FreqValues operator/ (
            const double lhs,
            const FreqValues & rhs )
```

The values of the object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the right-hand argument, which is the only argument that is of type *FreqValues*.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.29 operator/() [3/6]

```
FreqValues operator/ (
            const FreqValues & lhs,
            const double rhs )
```

This function calls the function `operator*(FreqValues,float)` wit the right-hand argument inverted.

The function returns a *FreqValues* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.30 operator/() [4/6]

```
Sweep operator/ (
            const Sweep & lhs,
            const Sweep & rhs )
```

Before performing the operation, the function checks if the frequencies of each structure match and if the "values" vectors have the same sizes as the "frequencies" vectors. The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.31 operator/() [5/6]

```
Sweep operator/ (
            const double lhs,
            const Sweep & rhs )
```

The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the right-hand side argument, the only one argument of type *Sweep*.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.32 operator/() [6/6]

```
Sweep operator/ (
            const Sweep & lhs,
            const double rhs )
```

The values of the of object to be returned are determined by the operation, while the rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument, the only one argument of type *Sweep*.

The function returns a *Sweep* object with the results of the operation.

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.9.2.33 pow() [1/4]

```
FreqValues pow (
            const FreqValues & base,
            const double exponent )
```

This function takes each point of the structure, raises its value to the exponent and stores the result in a different *FreqValues* structure, which is then returned. The rest of attributes (frequency, type, timestamp, etc.) are copied from the left-hand side argument, which the only argument that is of type *FreqValues*.

Parameters

in	*base*	The *FreqValues* structure to be used as the base of the power function.
in	*exponent*	The float value which will be used as the exponent.

6.9.2.34 pow() [2/4]

```
FreqValues pow (
            const double base,
            const FreqValues & exponent )
```

This function takes the `float` value, given as the base, and raises it to each of the values of the *FreqValues* structure given as the exponent. Each result is stored in a different *FreqValues* structure which is then returned. The rest of attributes (frequency, type, timestamp, etc.) of this structure are copied from the right-hand side argument, which is the only argument that is of type *FreqValues*.

Parameters

in	*base*	The `float` value given as the base of the exponentiation operation.
in	*exponent*	The *FreqValues* structure whose values will be used as the exponents of the exponentiation operation.

6.9.2.35 pow() [3/4]

```
Sweep pow (
            const Sweep & base,
            const double exponent )
```

This function takes each point of the structure, raises its value to the exponent and stores the result in a different *Sweep* structure, which is then returned. The rest of attributes (frequency, type, timestamp, etc.) are copied from the base argument, which the only argument that is of type *Sweep*.

Parameters

`in`	*base*	The *Sweep* structure to be used as the base of the power function.
`in`	*exponent*	The float value which will be used as the exponent.

6.9.2.36 pow() [4/4]

```
Sweep pow (
            const double base,
            const Sweep & exponent )
```

This function takes the `float` value, given as the base, and raises it to each of the values of the *Sweep* structure given as the exponent. Each result is stored in a different *Sweep* structure which is then returned. The rest of attributes (frequency, type, timestamp, etc.) of this structure are copied from the right-hand side argument, which is the only argument that is of type *Sweep*.

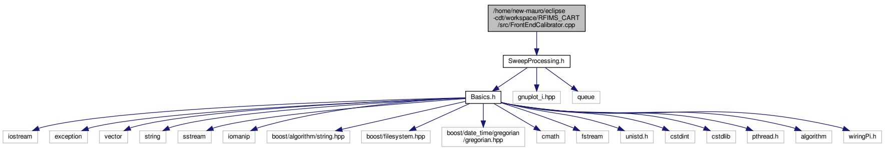
Parameters

`in`	*base*	The `float` value given as the base of the exponentiation operation.
`in`	*exponent*	The *Sweep* structure whose values will be used as the exponents of the exponentiation operation.

6.10 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/FrontEndCalibrator.cpp File Reference

This file contains the definitions of several methods of the class *FrontEndCalibrator*.

`#include "SweepProcessing.h"`
Include dependency graph for FrontEndCalibrator.cpp:



Functions

- bool **CheckNoFiniteAndNegValues** (const std::vector< FreqValues::value_type > &values)
- bool **CheckNoFiniteValues** (const std::vector< FreqValues::value_type > &values)

6.10.1 Detailed Description

This file contains the definitions of several methods of the class *FrontEndCalibrator*.

Author

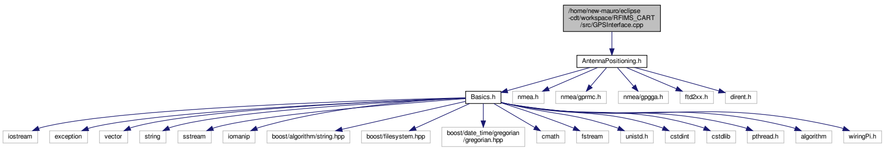
Mauro Diamantino

6.11 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/GPSInterface.cpp File Reference

This file contains the definitions of several methods of the class *GPSInterface*.

```
#include "AntennaPositioning.h"
```

Include dependency graph for GPSInterface.cpp:



Functions

- void ∗ **StreamingThread** (void ∗arg)

6.11.1 Detailed Description

This file contains the definitions of several methods of the class *GPSInterface*.

Author

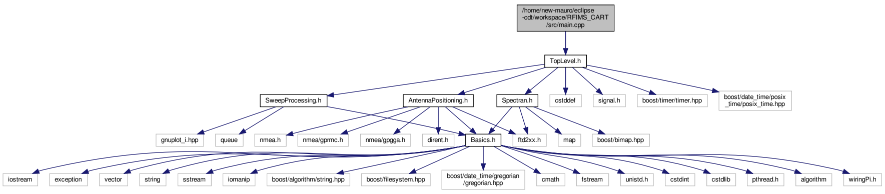
Mauro Diamantino

6.12 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/main.cpp File Reference

This file contains the main function of the RFIMS-CART software.

```
#include "TopLevel.h"
```

Include dependency graph for main.cpp:



Functions

- int main (int argc, char ∗argv[ ])

  The main function of the RFISM-CART software.

6.12.1 Detailed Description

This file contains the main function of the RFIMS-CART software.

Author

Mauro Diamantino

6.12.2 Function Documentation

6.12.2.1 main()

```
int main (
            int argc,
            char ∗ argv[ ] )
```

The main function of the RFISM-CART software.

This function instantiates all the needed objects and performs the software tasks in the corresponding order. The objects and their relations can be observed in the following components diagram:

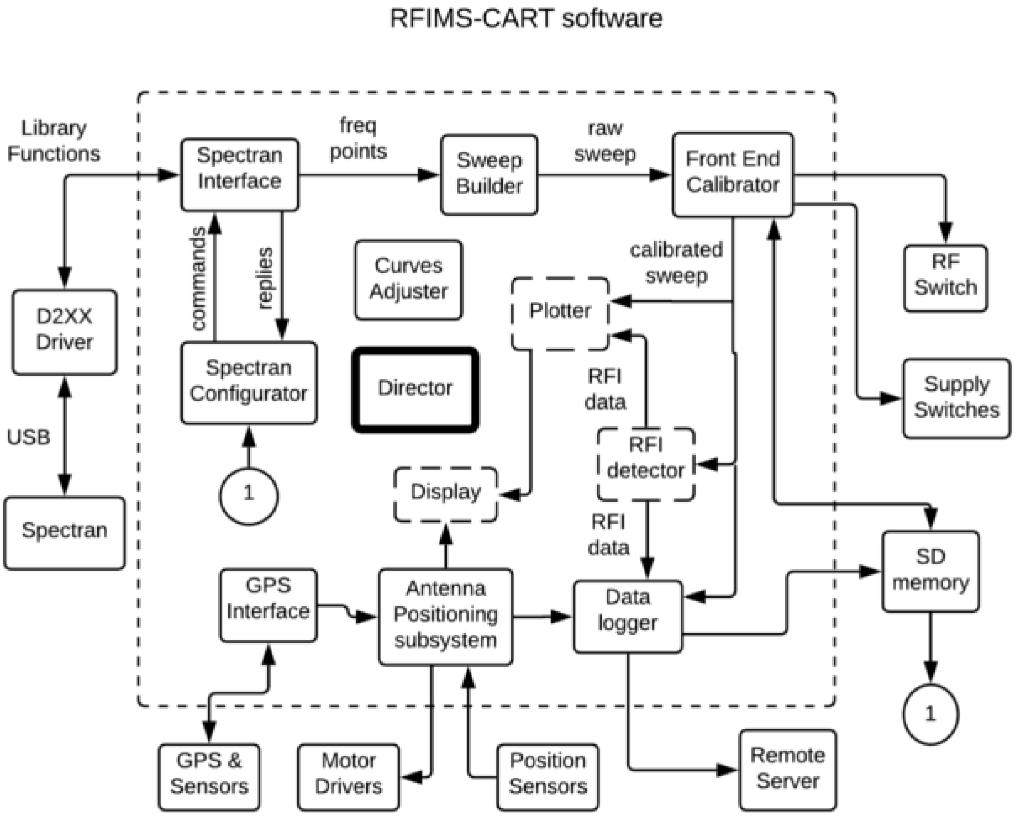


Figure 6.2 Software components diagram

The order of the operations which are performed in the main function follows the following flow diagram:

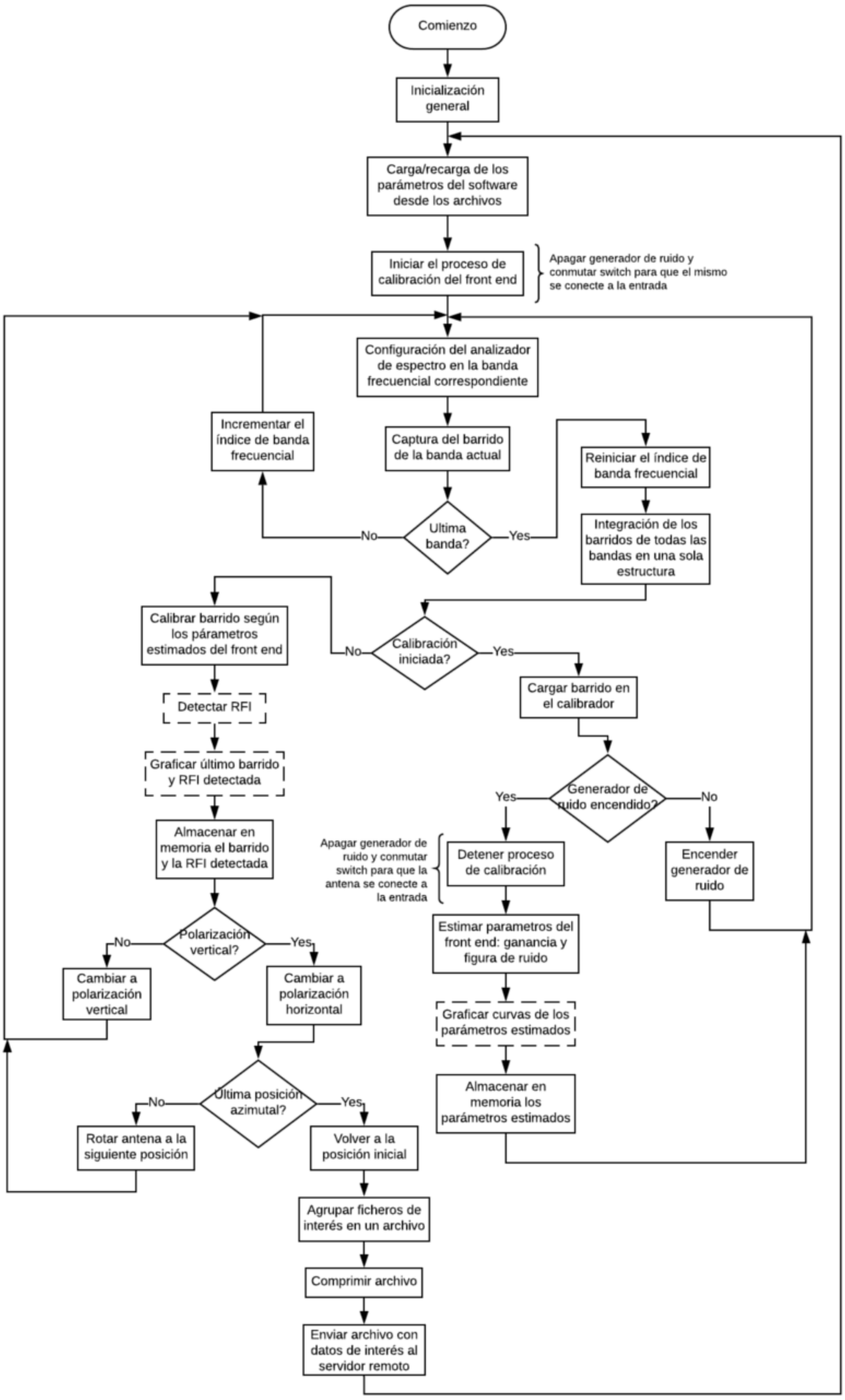

Comienzo
Inicialización general
Carga/recarga de los parámetros del software desde los archivos
Iniciar el proceso de calibración del front end
Apagar generador de ruido y conmutar switch para que el mismo se conecte a la entrada
Configuración del analizador de espectro en la banda frecuencial correspondiente
Incrementar el índice de banda frecuencial
Captura del barrido de la banda actual
Reiniciar el índice de banda frecuencial
Ultima banda?
No
Yes
Integración de los barridos de todas las bandas en una sola estructura
Calibrar barrido según los párametros estimados del front end
Calibración iniciada?
No
Yes
Cargar barrido en el calibrador
Detectar RFI
Graficar último barrido y RFI detectada
Generador de ruido encendido?
Yes
No
Apagar generador de ruido y conmutar switch para que la antena se conecte a la entrada
Detener proceso de calibración
Encender generador de ruido
Almacenar en memoria el barrido y la RFI detectada
Polarización vertical?
No
Yes
Estimar parametros del front end: ganancia y figura de ruido
Cambiar a polarización vertical
Cambiar a polarización horizontal
Graficar curvas de los parámetros estimados
Última posición azimutal?
No
Yes
Almacenar en memoria los parámetros estimados
Rotar antena a la siguiente posición
Volver a la posición inicial
Agrupar ficheros de interés en un archivo
Comprimir archivo
Enviar archivo con datos de interés al servidor remoto

Figure 6.3 High-level flow diagram

Parameters

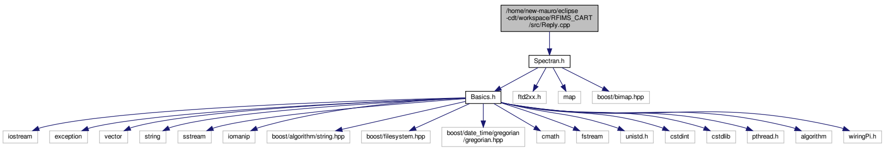
in	*argc*	The number of arguments that were received by the software.
in	*argv*	An array of C strings (`char*`) where each one is a software's argument.

6.13 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Reply.cpp File Reference

This file contains the definitions of several methods of the classes *Reply* and *SweepReply*.

```
#include "Spectran.h"
```

Include dependency graph for Reply.cpp:



6.13.1 Detailed Description

This file contains the definitions of several methods of the classes *Reply* and *SweepReply*.

Author

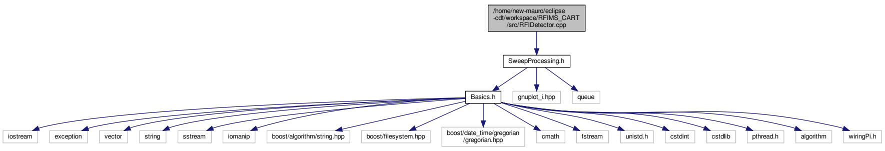
Mauro Diamantino

6.14 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/RFIDetector.cpp File Reference

This file contains the definitions of several methods of the class *RFIDetector*.

```
#include "SweepProcessing.h"
```

Include dependency graph for RFIDetector.cpp:



6.14.1 Detailed Description

This file contains the definitions of several methods of the class *RFIDetector*.

Author

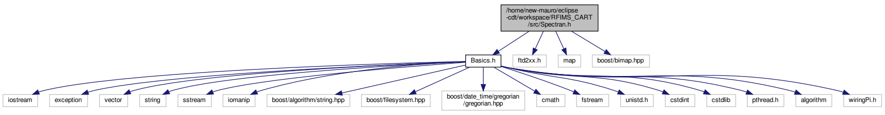
Mauro Diamantino

6.15 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/Spectran.h File Reference

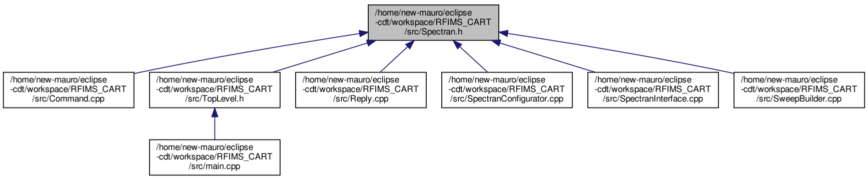
This header file contains the declarations of the classes which allow the communication with the spectrum analyzer Aaronia Spectran HF-60105 V4 X.

```
#include "Basics.h"
#include <ftd2xx.h>
#include <map>
#include <boost/bimap.hpp>
```

Include dependency graph for Spectran.h:



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



Classes

- union FloatToBytes

  An union which is used to split a `float` *value in its 4 bytes.*
- class Command

  This class builds the corresponding bytes array to send a certain command to a Aaronia Spectran V4 series spectrum analyzer.
- class Reply

  The class Reply is intended to receive a bytes vector sent by the spectrum analyzer and to extract its information.
- class SweepReply

  This class derives from the base class Reply and is intended to process in a better way replies with sweep points, i.e. AMPFREQDAT replies.
- class SpectranInterface

  The aim of this class is to manage the communication with the Aaronia Spectran device.
- class SpectranConfigurator

  The class SpectranConfigurator is intended to manage the process of configuring the Aaronia Spectran device.
- struct SpectranConfigurator::FixedParameters

  This structure saves the fixed parameters of the spectrum analyzer, i.e. the parameters which do not change through the entire measurement cycle.
- class SweepBuilder

  The aim of class SweepBuilder is to build the complete sweep from the individual sweep points which are delivered by the Spectran Interface.

Typedefs

- typedef boost::bimap< float, float > RBW_bimap

  This typedef simplifies the definitions of containers of type `boost::bimap<float,float>` which is used to store RBW values and its indexes.

Enumerations

- enum SpecVariable : uint8_t {
  STARTFREQ =0x01, **STOPFREQ**, **RESBANDW**, **VIDBANDW**,
  SWEEPTIME, **ATTENFAC**, **REFLEVEL**, **DISPRANGE**,
  DISPUNIT, **DETMODE**, **DEMODMODE**, **SPECPROC**,
  ANTTYPE, **CABLETYPE**, **RECVCONF**, **CENTERFREQ** =0x1E,
  SPANFREQ, **PREAMPEN** =0x10, **SWPDLYACC**, **SWPFRQPTS**,
  REFOFFS, **USBMEAS** =0x20, **USBSWPRST**, **USBSWPID**,
  USBRUNPROG, **LOGFILEID** =0x30, **LOGSAMPCNT**, **LOGTIMEIVL**,
  SPECDISP =0x41, **PEAKDISP**, **MARKMINPK**, **RDOUTIDX**,
  MARKCOUNT, **LEVELTONE**, **BACKBBEN**, **DISPDIS**,
  SPKVOLUME, **RBWFSTEP** =0x60, **ANTGAIN**, **PEAK1POW** =0x80,
  PEAK2POW, **PEAK3POW**, **PEAK1FREQ** =0x84, **PEAK2FREQ**,
  PEAK3FREQ, **MAXPEAKPOW** =0x90, **STDTONE** =0xC0, **UNINITIALIZED** }

  An enumeration which contains of the names of all the environment variables of the spectrum analyzer Aaronia Spectran HF-60105 V4 X.

Functions

- const std::vector< RBW_bimap::value_type > vect ({ {50e6, 0.0}, {3e6, 1.0}, {1e6, 2.0}, {300e3, 3.0}, {100e3, 4.0}, {30e3, 5.0}, {10e3, 6.0}, {3e3, 7.0}, {1e3, 8.0}, {120e3, 100.0}, {9e3, 101.0}, {200.0, 102.0}, {5e6, 103.0}, {200e3, 104}, {1.5e6, 105.0} })

  A vector which is initialized with the pairs of values {RBW(Hz), RBW index}. This vector is used to initialize a bidirectional map.

- const RBW_bimap RBW_INDEX (vect.begin(), vect.end())

  A bidirectional map (bimap) which contains the pairs of values {RBW(Hz), RBW index}.

6.15.1 Detailed Description

This header file contains the declarations of the classes which allow the communication with the spectrum analyzer Aaronia Spectran HF-60105 V4 X.

The classes defined in this header file allows to set up the spectrum analyzer, read its environment variables, enable/disable the streaming of sweep points, process its responses and to capture and store the sweep points in an orderly manner.

Author

Mauro Diamantino

6.15.2 Enumeration Type Documentation

6.15.2.1 SpecVariable

```
enum SpecVariable : uint8_t [strong]
```

An enumeration which contains of the names of all the environment variables of the spectrum analyzer Aaronia Spectran HF-60105 V4 X.

Each member (a variable name) has associated an integer number which is equal to its identification index (ID), taking into account the Spectran USB Protocol documentation.

6.15.3 Function Documentation

6.15.3.1 RBW_INDEX()

```
const RBW_bimap RBW_INDEX (
             vect.  begin(),
             vect.  end() )
```

A bidirectional map (bimap) which contains the pairs of values {RBW(Hz), RBW index}.

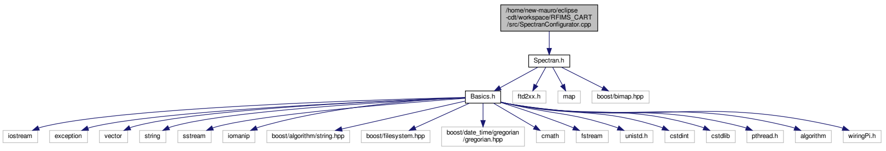
These pairs of values relates a RBW frequency value with its corresponding index taking into account the Spectran USB protocol. So this bidirectional container allows to get the corresponding index given a frequency value or to get the corresponding frequency value given an index. This allows to work with frequency values of RBW at high level, eliminating the need to learn the protocol's indexes. This container is used by the classes *Command* and *Reply*.

6.16 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/SpectranConfigurator.cpp File Reference

This file contains the definitions of several methods of the class *SpectranConfigurator*.

```
#include "Spectran.h"
```

Include dependency graph for SpectranConfigurator.cpp:



6.16.1 Detailed Description

This file contains the definitions of several methods of the class *SpectranConfigurator*.

Author

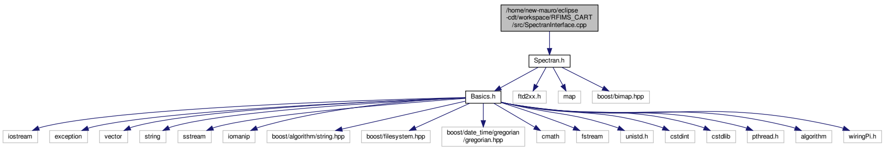
Mauro Diamantino

6.17 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/SpectranInterface.cpp File Reference

This file contains the definitions of several methods of the class *SpectranInterface*.

```
#include "Spectran.h"
```

Include dependency graph for SpectranInterface.cpp:



6.17.1 Detailed Description

This file contains the definitions of several methods of the class *SpectranInterface*.

Author

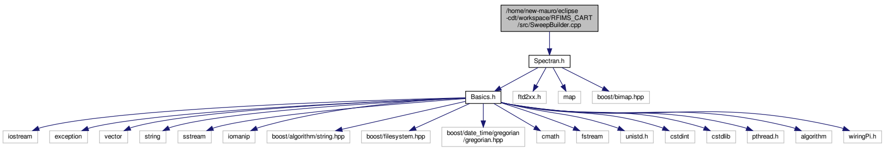
Mauro Diamantino

6.18 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/SweepBuilder.cpp File Reference

This file contains the definitions of several methods of the class *SweepBuilder*.

```
#include "Spectran.h"
```

Include dependency graph for SweepBuilder.cpp:



6.18.1 Detailed Description

This file contains the definitions of several methods of the class *SweepBuilder*.

Author

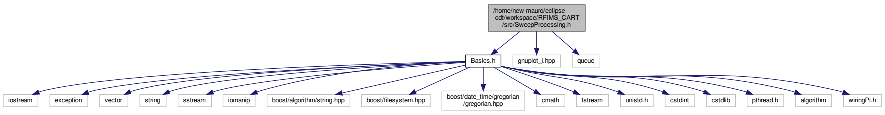
Mauro Diamantino

6.19 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/SweepProcessing.h File Reference

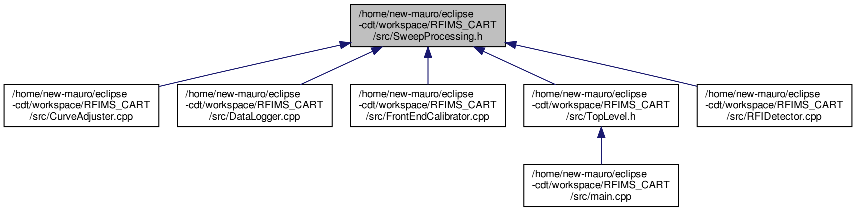
This header file contains the declarations of the classes which are responsible for the processing of each sweep, once it has been captured.

```
#include "Basics.h"
#include "gnuplot_i.hpp"
#include <queue>
```

Include dependency graph for SweepProcessing.h:



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



Classes

- class RFPlotter

  The class RFPlotter is intended to plot sweeps, RF interference (RFI) and any frequency curve.
- class CurveAdjuster

  The aim of the class CurveAdjuster is to adjust any frequency curve, this is to interpolate and/or extrapolate the curve of a given parameter versus frequency.
- class FrontEndCalibrator

  The aim of this class is to calculate the total gain and total noise figure curves versus frequency of the RF front end.
- class RFIDetector

  The aim of this class is to compare each calibrated sweep with a threshold curve to determine where there is RF interference (RFI).
- class DataLogger

  The class DataLogger is intended to handle the storing of the generated data into memory, following the CSV (comma-separated values) format.

6.19.1 Detailed Description

This header file contains the declarations of the classes which are responsible for the processing of each sweep, once it has been captured.

The tasks which are performed by the classes defined here are the following:

- Plotting of sweeps, RFI and any frequency curve.
- Adjusting (interpolation) of frequency curves.
- Front end calibration.
- RFI detection.
- Data logging.

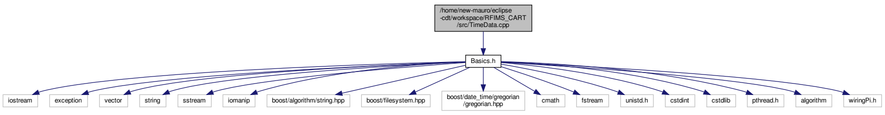
  Author

  Mauro Diamantino

6.20 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/TimeData.cpp File Reference

This file contains the definitions of several methods of the structure *TimeData*.

`#include "Basics.h"`
Include dependency graph for TimeData.cpp:



Functions

- bool operator< (const TimeData &lhs, const TimeData &rhs)
- bool operator> (const TimeData &lhs, const TimeData &rhs)
- bool operator== (const TimeData &lhs, const TimeData &rhs)

6.20.1 Detailed Description

This file contains the definitions of several methods of the structure *TimeData*.

Author

Mauro Diamantino

6.20.2 Function Documentation

6.20.2.1 operator<()

```
bool operator< (
            const TimeData & lhs,
            const TimeData & rhs )
```

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.20.2.2 operator==()

```
bool operator== (
            const TimeData & lhs,
            const TimeData & rhs )
```

Parameters

in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.20.2.3 operator>()

```
bool operator> (
            const TimeData & lhs,
            const TimeData & rhs )
```

Parameters

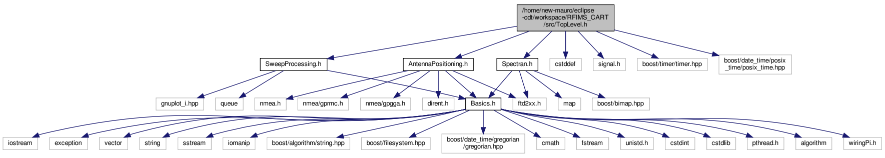
in	*lhs*	The left-hand side operand.
in	*rhs*	The right-hand side operand.

6.21 /home/new-mauro/eclipse-cdt/workspace/RFIMS_CART/src/TopLevel.h File Reference

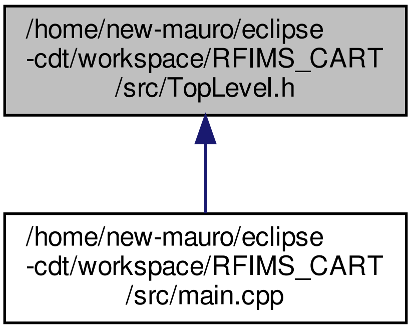
This header file includes the rest of the header files and the class of the signal handler is declared here.

```
#include "Spectran.h"
#include "SweepProcessing.h"
#include "AntennaPositioning.h"
#include <cstddef>
#include <signal.h>
#include <boost/timer/timer.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
```

Include dependency graph for TopLevel.h:



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



Classes

- class SignalHandler

  The class SignalHandler is intended to handle the interprocess signals (IPC) which terminates the software.

Functions

- void PrintHelp ()

  This function prints a message, in the `stdout`, with a software description and the descriptions of its arguments.
- bool ProcessMainArguments (int argc, char ∗argv[ ])

  This function process the software's arguments, which define the behavior of this one.
- std::string GetTimeAsString (boost::timer::cpu_timer &timer)

  A function which extracts data from a timer and returns it as a string in a human-readable format.

Variables

- const unsigned int DEF_NUM_AZIM_POS = 6

  The number of azimuth positions where the system will perform sweeps.
- bool flagCalEnabled

  The declaration of a flag which defines if the calibration of the RF front end must be done or not. By default the calibration is enabled.

- bool flagPlot

  The declaration of a flag which defines if the software has to generate plots or not. By default the plotting is not performed.
- bool flagInfiniteLoop

  The declaration of a flag which defines if the software has to perform a finite number of measurement cycles or iterate infinitely. By default the software iterates infinitely.
- bool flagRFI

  The declaration of a flag which defines if the software has to perform RFI detection or not. By this task is not performed.
- bool flagUpload

  The declaration of a flag which defines if the software has to upload the measurements or not. By default the uploading is performed.
- unsigned int numOfMeasCycles

  A variable which saves the number of measurements cycles which left to be done. It is used when the user wishes a finite number of measurements cycles.
- RFI::ThresholdsNorm rfiNorm

  A variable which saves the norm which defines the harmful RF interference levels: ska-mode1, ska-mode2, itu-ra769-2-vlbi.
- unsigned int numOfAzimPos

  A variable which receives the number of azimuth positions from the corresponding software's argument. The number of sweeps will be the double of this value.
- boost::timer::cpu_timer timer

  A timer which is used to measure the execution time when the number of iterations is finite.

6.21.1 Detailed Description

This header file includes the rest of the header files and the class of the signal handler is declared here.

This header file simplifies the include in the main.cpp file. The declaration of the class *SignalHandler* must be put in an high-level header file because this class must know the declarations of almost all the classes of the software.

Author

Mauro Diamantino

6.21.2 Function Documentation

6.21.2.1 ProcessMainArguments()

```
bool ProcessMainArguments (
             int argc,
             char * argv[] )
```

This function process the software's arguments, which define the behavior of this one.

This function determines the values of the behavior flags (flagCalEnabled, flagPlot, flagRFI, etc.) taking into account the arguments that were received and its values. The function returns a `true` value if the arguments were processed correctly and a `false` value if there was an argument which could not be recognized, and in that case it presents a message, in the `stdout`, explaining the correct use of the software arguments.

Parameters

in	*argc*	The number of arguments that were received by the software.
in	*argv*	An array of C strings (`char*`) where each one is a software's argument.