Feasa Communications SO Library

for 32 & 64 bit

reference guide

Table of Contents

1. Introduction

Feasa Communications SO (Shared Library) is a general purpose library developed by Feasa Enterprises Ltd. This library is intended to ease to the customer the access to any Feasa Device with a single SO file (LED Analyser, LED Spectrometer, Legend...), providing a robust communication method, without the need of additional components or complex port-handling code and independent of the programming language used.

This manual contains generic instructions on how to install the Library and how to get started in its implementation on the programming language of your choice.

Here you will also find detailed information for each one of the many functions provided; at the end of the document, there is a function index with links to their description.

Source code shown here is written in C, but very simple code and types are used, so it should be very easy to read. Anyway, a table of equivalences for variable types between programming languages can be found in Annex I.

An application note about Optimizing Test Times can be found in the Documentation folder of the CD, and users could find it useful.

2. Previous considerations

To be able to use this Library, you should ensure that the used programming language is able to work with SO libraries.

Remember that the SOs should be generally copied to the /usr/lib/ folder. Some compilers also accept or need the Library to be placed in the same path of the binary file.

You can notice that there are two versions of the SO a 32bit (libfeasacom.so) and a 64bit version (libfeasacom64.so); so you can get error messages or have communication problems if you are trying to use a wrong Library or your compiler is not targeting the right platform for the binary file. Almost all compilers allow to select the target, so you should set it to x86 or equivalent for the 32bit SO and Wow64, x64 or equivalent for the 64bit version.

Note that all programming examples included in the CD are targeted to 32bit platforms, so they use the 32bit version of the Library (libfeasacom.so).

As you will see in the following sections, some of the functions provided on the SO Library allow to detect automatically the baud rate of the connected Feasa device, however we do not recommend to use this method when working with Daisy Chain. We also do not recommend to open the port using the serial number method in this situations, since it is slower and also makes a polling to all the serial ports available, interrogating all connected devices, which can make to receive unexpected data in non-Feasa devices. It is also faster to use a fixed baud rate and port when opening a communication with a Feasa Device

The functions of this Library follow the CDECL calling convention. This should not be an issue since any modern programming language allow to select the calling convention when importing any Library function; others have specific importing functions depending on the calling convention.

You should be careful when handling functions that return strings by parameter, because they have a pointer structure, so you should previously reserve some memory space for the returned string. This means that if you are expecting to have an answer of 10 characters you should have a string with at least 11 characters width. Most of the modern programming languages uses Dynamic Strings, like Visual Basic, dot NET languages, Java, etc. So you need to previously ensure that these variables have a memory equal or higher that the one needed for the returned byte array, otherwise you can get critical errors during the program execution.

Depending on the programming language used, Integer variables does not match in size with the Integer described in the SO Library (32 bit). Please, check your language's reference manual in order to be sure that the proper data types are being used.

This SO Library has been written for the Linux OS and tested on Debian, Ubuntu and Mint distributions, so Feasa cannot ensure that this Library will work on different conditions than those described.

Important: *the use of USB hubs can cause issues, specially, when several USB devices are used simultaneously, being the most important ones enumeration issues, delay in opening ports or delay in communications that can lead to timeouts. These issues are specially noticeable in bad-quality USB hubs and when using multi-threading techniques. The use of mother-board attached USB ports or good quality PCI card USB ports is recommended. In case of using external USB hubs, ensure that have external power supply and are USB certified (industrial hubs preferred.*

2.1 Reliability

In order to estimate the reliability of the SO Library an intensive test has been designed.

The test consists on:

- Feasa 20-F and Feasa 10-F

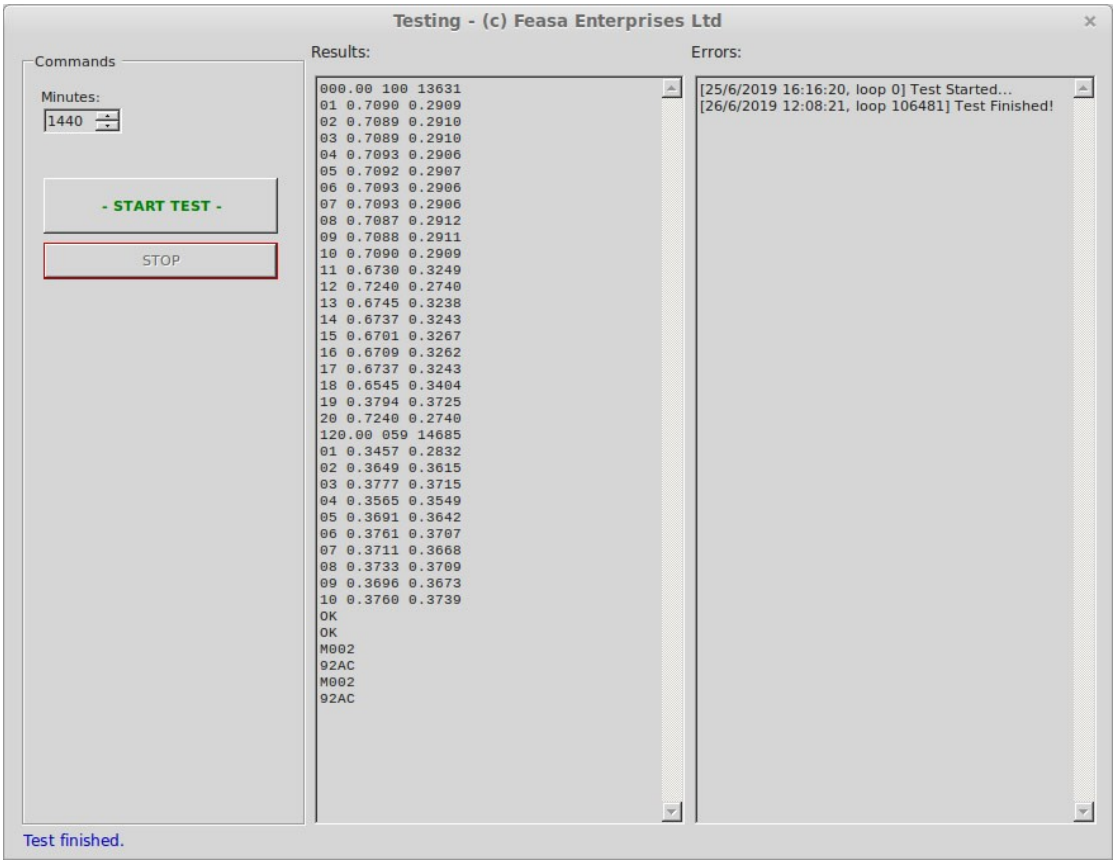
- Both devices connected through USB to a USB 3.1 hub (in order to force a non-ideal environment).

- Devices

- Sequence of commands used for each loop of the test:

1. Port enumeration (performed once every 100 loops).
2. IsConnected() for Device 1
3. IsConnected() for Device 2
4. Open() port for Device 1 + Send() GETHSI01 + Send() GETxyALL + Close()
5. Open() port for Device 2 + Send() GETHSI01 + Send() GETxyALL + Close()
6. Open_Multi() multi-threaded for Devices 1 and 2 + SendToAll() C3 + SendToAll() GETSERIAL & ensure that matches with known SNs + GetResponseByPort() & ensure that matches with the previous one + Close_Multi()

The test has been brought to life through a C# application and tested for near 24 hour in Linux, Windows and Raspberry PI using the very same setup. As a result of the test, more than 100000 loops were performed and no errors could be appreciated, as can be seen in the following screenshot:



3. First steps

3.1 Installation instructions

Before use the SO Library you should previously copy it to the system libraries folder, which can be generally found on the /usr/lib or equivalent folder.

Some programming languages allow to use the SO if you previously copy it to the same folder of the binary file or programming project, where you are calling it from, so in this situations there is no need to copy the SO Library to the system folder.

Sometimes it's required to have certain read permissions for the file, so you should change it according to your system or account policies.

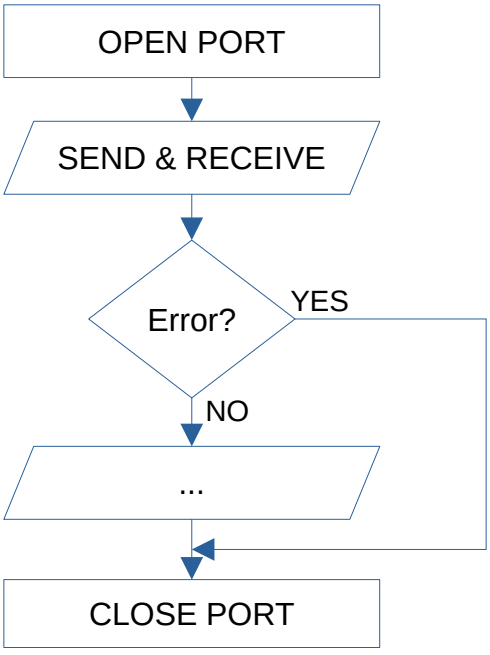
Depending on the Linux distribution could also be needed to belong to special communication groups (like 'dialout') or provide the user with certain privileges.

If you are planning to redistribute your application or you are installing it on other computers, you should repeat this installation/set-up process in all the target computers.

3.2 Usage

The SO Library works as a port interface between the computer and the Feasa Device, but does not perform any processing or does any calculation. All processing is done inside the Feasa Device so in case that results are not satisfactory, then the unit should be fine tuned using the Feasa UserCal or UserCal functions embedded in the Library or either the experiment or test station should be re-adjusted to match the required results.

The Library logic consist in opening a **communication port**, where the Feasa Device is connected to, then send commands, received the responses/results (if required) and then close the port.



If a port remains open for a long period of time and the computer is not in use, then the Operating System can put the hardware port to sleep, which can lead to the port not to be accessible in some situations, so it is highly recommended to close the port after the test batch is finished.

The communications port of your device should be listed in the Windows Device Manager and should also be accessible through the Feasa software and SO Library. A port can be serial or USB, being necessary, for this last one to have all drivers installed successfully, so if a USB port is connected and after some time is not listed in the Windows device manager or any error is reported there, then it means that the driver has not been installed successfully or not installed at all. In this case, please, use the driver provided in the Feasa CD.

Once the communication port has been identified, then this **port has to be opened** using the **baud-rate** configured in the Feasa device (default is 57600 baud; auto can be used but takes more time) through SO Library functions like FeasaCom_Open(), FeasaCom_OpenSN(), FeasaCom_OpenProject(), etc. If no error occurred, then it will be available for transmitting and receiving data from the Feasa device. In case of using a **daisy-chain configuration**, then all devices should have the same baud-rate configured. This can be easily checked by connecting to every single unit individually through the Feasa Terminal.

There are **different ways to open a port**: by the **port number** (COM1, COM2, COM3...), by the **Serial Number** printed in the Feasa device (ex: S001, A45G, 1N23...) or by **ID** (see projects). See section *4.3 Open a port* for more information.

```
FeasaCom_Open("/dev/ttyUSB0", 57600); //COM12
FeasaCom_OpenSN("A122", 0);
```

For simplicity in the handling of timeouts, the Library provides functions to **perform a measurement (Capture)**, like FeasaCom_Capture() but there are no specific functions tu run most of the features available in the Feasa devices, since there could be hundred of different commands, so all of those **commands should be transmitted** using the function FeasaCom_Send(), FeasaCom_SendSN(), FeasaCom_SendByID(), FeasaCom_SendToAll(), FeasaCom_DaisyChain_Send(), etc. All commands are listed in the User Manual of the Feasa Device. See sections *4.5 Send & receive data* and *4.14 Projects: loading a port from a file* for more information.

```
FeasaCom_Send("/dev/ttyUSB0", "GETSERIAL", ReceptionBuffer);
FeasaCom_SendSN("A122", "GETSERIAL", ReceptionBuffer);
```

As stated in the Feasa User manual, all commands should be terminated with ASCII CR + LF characters (13 + 10) but Feasa SO Library automatically appends this to all commands sent, so there is no need to include them.

Timeouts are not usually an issue for usual commands, having a default value of 3500ms which can be changed using function FeasaCom_SetResponseTimeout(), while timeouts for Capture related commands could be trickier, since depending on the capture range, exposure time, averaging value, capture type (PWM/non-PWM), etc, time needed can change significantly. This is why it could be helpful to use Capture-related functions provided in the Library, like FeasaCom_Capture(), FeasaCom_SpectrometerCapture(), etc. As a tip, as explained in the device's user manual, the

Capture/Measurement is done in all the channels simultaneously.

Once the communication has finished, then the **port has to be closed**, so it is released and can be used by other programs. This is done using commands like FeasaCom_Close(), FeasaCom_CloseSN(), FeasaCom_CloseAll(), etc. See section *4.4 Close a port* for more information.

```
FeasaCom_Close("/dev/ttyUSB0");
FeasaCom_CloseSN("A122");
```

Most functions of the Library return a value, which is helpful to find out whether **an error occurred** or not (check documentation of each function to obtain more details). The description for the error can be read using commands like FeasaCom_GetError_Description(), FeasaCom_GetError_DescriptionByPort(), or FeasaCom_GetError_DescriptionBySN().

Examples:
Think that there is a Feasa LED Analyser, connected to a port with device path /dev/tty0, which has a pre-configured baud-rate of 115200 baud. A measurement hast to be performed, in range 3 (HIGH) and chromaticity values for channel/fiber 2 need to be retrieved and shown. The code lines show a possible solution for the required target:

```
char ResponseBuffer[512];
float x, y;

if (FeasaCom_Open("/dev/tty0", 115200)==1)
{
	if (FeasaCom_Capture("/dev/tty0", 0, 3, 0)==1)
	{
		if (FeasaCom_Send("/dev/tty0", "GETXY02", ResponseBuffer))
		{
			sscanf(ResponseBuffer, "%f %f", &x, &y);
			printf("Chromaticity values are x=%f, y=%f", x, y);
		}
	}
	FeasaCom_Close("/dev/tty0");
}
```

There are many examples in the CD, so please, check them since they will be really helpful to boost the development process.

In case you think that test times are slow for your requirements, an application note about Optimizing Test Times can be found in the Documentation folder of the CD, and users could find it useful.

4. Functions

4.1 General

4.1.1 SO Library version

In order to ensure that the SO Library you are working with is the one you actually want to work with, it is necessary to check not only the path, but also the Library version. The function below provides the version of the SO Library file you are working width:

```
void FeasaCom_GetLibraryVersion(char * Version);
```

Parameters:

- Version: pointer to string that stores the version of the SO Library in a format like “x.x.x”.

Return value:

Does not return any value.

4.1 Ports Enumeration

Enumerate ports means to list all the available ports where a Feasa device is connected. This method is used for the Library to know exactly what are the ports available to work with. So you should use this function at the beginning of you program or when you connect and disconnect a USB Feasa device, otherwise a device that has just been connected will be not recognized.

```
void FeasaCom_EnumPorts(char * Version);
```

Parameters:

- Version: pointer to string array that will contain the

Return value:

Does not return any value

You may think that the Library should scan all the available Devices automatically when a port is opened, but it could affect to the speed of the production process when you have a high number of ports or Feasa Devices connected to the computer.

This action is only performed automatically the first time a port is opened, in case the port enumeration was not done before.

4.2 Find and Detect Feasa devices

4.2.1 Find out Device Path by Serial Number

This function can be used to search a Feasa device based on its Serial Number.

```
int FeasaCom_IsConnected(char * DevPath, const char * SerialNumber, const
                         int Baudrate);
```

Parameters:

- DevPath: string returned containing the path of the device, when the Feasa device has been found.
- SerialNumber: string specifying the serial number of the Feasa device to find.
- Baudrate: number specifying the baud rate used by the Feasa device: 9600, 19200, 38400, 57600, 115200, 460800, 921600. You can specify the value 0 to auto-detect the baud rate, but this method is slower.

Return value:

Returns 1 if the Feasa device is connected, 0 if not found or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

***Important**: this function perform a polling to all the ports available, trying to open those ports and sending some Feasa commands to them. Please, take this into account if you have other serial devices connected to the computer, since the usage function can make unexpected data to be received in non-Feasa devices.*

4.2.2 Detect all Feasa devices connected (and get Device Paths)

In case that the user needs to detect all Feasa devices attached to the computer and retrieve their Device Paths, the following function can be used:

```
int FeasaCom_Detect(char ** DevPaths, const int Baudrate);
```

Parameters:

- DevPaths: string array containing the list of Paths detected.
- Baudrate: number specifying the baud rate used by the Feasa device: 9600, 19200, 38400, 57600, 115200, 460800, 921600. You can specify the value 0 to auto-detect the baud rate, but this method is slower.

Return value:

Returns the number of Feasa devices detected.

You can limit the detection to certain types of Feasa devices by using the function FeasaCom_AddDetectionFilter(), explained in following section.

4.2.3 Detect all Feasa devices connected (and get Device Paths as string)

In case that the user needs to detect all Feasa devices attached to the computer and retrieve their Device Paths, the following function can be used:

```
int FeasaCom_DetectS(char * DevPaths, const char Delimiter, const int
                                Baudrate);
```

Parameters:

- DevPaths: string containing the list of Paths detected, separated by the defined delimiting character.
- Delimiter: character used to differentiate between paths returned through the DevPaths string.
- Baudrate: number specifying the baud rate used by the Feasa device: 9600, 19200, 38400,

57600, 115200, 460800, 921600. You can specify the value 0 to auto-detect the baud rate, but this method is slower.

Return value:

Returns the number of Feasa devices detected.

You can limit the detection to certain types of Feasa devices by using the function FeasaCom_AddDetectionFilter(), explained in following section.

4.2.4 Detect all Feasa devices connected (and get Serial Numbers)

In case that the user needs to detect all Feasa devices attached to the computer and retrieve their Port Numbers, the following function can be used:

```
int FeasaCom_DetectSN(char ** SerialNumbers, const int Baudrate);
```

Parameters:

- SerialNumbers: string array containing the list of Serial Numbers detected.
- Baudrate: number specifying the baud rate used by the Feasa device: 9600, 19200, 38400, 57600, 115200, 460800, 921600. You can specify the value 0 to auto-detect the baud rate, but this method is slower.

Return value:

Returns the number of Feasa devices detected.

You can limit the detection to certain types of Feasa devices by using the function FeasaCom_AddDetectionFilter(), explained in following section.

4.2.5 Add Detection Filter

It is possible to add a detection filter, so that only one type of Feasa Device is detected, based in its Hardware ID (Feasa 20-F, Feasa05-I, etc.). This way, only the devices matching with the given filters will be listed in commands Detect and DetectSN:

```
void FeasaCom_AddDetectionFilter(const char * Filter);
```

Parameters:

- Filter: string containing the hardware ID of the device to be detected or any of the following wildcards: “ANALYSER”, “FUNCTIONAL”, “ICT”, “LOWLIGHT”, “RGB”, “IR”, “SPECTROMETER”, “DISPLAY”, “LEGEND”, “RGBREF”; “ANALYSER” can be used to detect Functional or ICT devices, no matter if they are standard or high-brightness, while the other wild-cards are self-explanatory.

Return value:

none

Several filters can be added, in order to detect different Feasa Devices. Filters are not stored anywhere, so there will be no filters set every time that the SO Liobrary is loaded or the caller program run. Filters can be removed at any time using function FeasaCom_ClearDetectionFilters().

4.2.6 Clear Detection Filters

Detection filters previously set with function FeasaCom_AddDetectionFilter() can be cleared using the following function:

```
void FeasaCom_ClearDetectionFilters(void);
```

Parameters:

none

Return value:

none

4.3 Open a port

4.3.1 Open by Device Path

In order to start a communication with a Feasa device you first need to open the port to which it is connected to.

```
int FeasaCom_Open(const char * DevPath, const int Baudrate);
```

Parameters:

- `DevPath`: string containing the system path of the device to be opened.
- Baudrate: number specifying the baud rate used by the Feasa device: 9600, 19200, 38400, 57600, 115200, 460800, 921600. You can specify the value 0 to auto-detect the baud rate, but this method is slower and it is not recommended when operating in daisy-chain.

Return value:

Returns 1 if the port has been successfully opened, 0 on error. Once the port is opened, it stays in locked status, which means that can only be used by the caller application. It can not be used by other software until you close it.

4.3.2 Open by Serial Number

In order to start a communication with a Feasa device given its Serial Number, you first need to open the port to which it is connected to.

```
int FeasaCom_OpenSN(const char * SerialNumber, const int Baudrate);
```

Parameters:

- SerialNumber: Serial Number of the Feasa device with which you want to establish a communication.
- Baudrate: number specifying the baud rate used by the Feasa device: 9600, 19200, 38400, 57600, 115200, 460800, 921600. You can specify the value 0 to auto-detect the baud rate, but this method is slower and it is not recommended when operating in daisy-chain.

Return value:

Returns 1 if the port has been successfully opened, 0 on error. Once the port is opened, it stays in locked status, which means that can only be used by the caller application. It can not be used by other software until you close it.

***Important**: this function perform a polling to all the ports available, trying to open those ports and sending some Feasa commands to them. Please, take this into account if you have other serial devices connected to the computer, since the usage function can make unexpected data to be received in non-Feasa devices.*

4.3.3 Open multiple devices by Device Path port (multi-threaded)

In order to start a communication with several Feasa devices, given their Device Paths, all its ports can be opened simultaneously:

```
int FeasaCom_Open_Multi(int * ReturnValues, const char ** DevPaths, const
                  int nPorts, const char * Baudrate);
```

Parameters:

- ReturnValues: numeric array with the responses from each one of the devices.
- DevPaths: array with list of Device Paths of the Feasa devices with which you want to establish a communication.
- nPorts: numeric value indicating how many Ports are listed in the DevPaths array.
- Baudrate: number specifying the baud rate used by the Feasa device: 9600, 19200, 38400, 57600, 115200, 460800, 921600. You can specify the value 0 to auto-detect the baud rate, but this method is slower and it is not recommended when operating in daisy-chain.

Return value:

Returns 1 if all ports have been successfully opened, 0 otherwise. Once the port is opened, it stays in locked status, which means that can only be used by the caller application. It can not be used by other software until you close it.

***Note:** In case that a single port can't be opened, non of the requested ports will be opened.*

4.3.4 Open multiple devices by Serial Number (multi-threaded)

In order to start a communication with several Feasa devices, given their Serial Numbers,, all its ports can be opened simultaneously:

```
int FeasaCom_OpenSN_Multi(int * ReturnValues, const char ** SerialNumbers,
                const int nSerials, const char * Baudrate);
```

Parameters:

- ReturnValues: numeric array with the responses from each one of the devices.
- SerialNumbers: array with list of Serial Numbers of the Feasa devices with which you want to establish a communication.
- nSerials: numeric value indicating how many Serials are listed in the SerialNumbers array.
- Baudrate: number specifying the baud rate used by the Feasa device: 9600, 19200, 38400, 57600, 115200, 460800, 921600. You can specify the value 0 to auto-detect the baud rate, but this method is slower and it is not recommended when operating in daisy-chain.

Return value:

Returns 1 if all ports have been successfully opened, 0 otherwise. Once the port is opened, it stays in locked status, which means that can only be used by the caller application. It can not be used by other software until you close it.

***Note:** In case that a single port can't be opened, non of the requested ports will be opened.*

***Important**: this function perform a polling to all the ports available, trying to open those ports and sending some Feasa commands to them. Please, take this into account if you have other serial or USB (Virtual Serial Port) devices connected to the computer, since the usage function can make unexpected data to be received in non Feasa devices.*

4.4 Close port

Once the port is opened and all the commands have been sent, it is necessary to close it and free its resources in order to be able to use it from other programs.

4.4.1 Close by Device Path

A device can be closed given its Device path using the following function:

```
int FeasaCom_Close(const char * DevPath);
```

Parameters:

- `DevPath`: string containing the system path of the device.

Return value:

Returns an integer value of 1 if the port was successfully closed, 0 otherwise.

4.4.2 Close by Serial Number

A device's port can be closed given its Serial Number using the following function:

```
int FeasaCom_CloseSN(const char * SerialNumber);
```

Parameters:

- SerialNumber: the same one used to open the port.

Return value:

Returns an integer value of 1 if the port was successfully closed, 0 otherwise.

4.4.3 Close multiple devices by Device Path (multi-threaded)

Multiple device's ports can be closed simultaneously given their Device Paths using the following function:

```
int FeasaCom_Close_Multi(int * ReturnValues, const char ** DevPaths, const
                         int nPorts);
```

Parameters:

- ReturnValues: numeric array with the responses from each one of the devices.
- DevPaths: array with list of Device Paths of the Feasa devices with which you want to establish a communication.
- nPorts: numeric value indicating how many Ports are listed in the DevPaths array.

Return value:

Returns an integer value of 1 if all ports were successfully closed, 0 otherwise.

4.4.4 Close multiple devices by Serial Number (multi-threaded)

Multiple device's ports can be closed simultaneously given its Serial Number using the following function:

```
int FeasaCom_CloseSN_Multi(int * ReturnValues, const char ** SerialNumbers,
                           const int nSerials);
```

Parameters:

- ReturnValues: numeric array with the responses from each one of the devices.
- SerialNumbers: array with list of Serial Numbers of the Feasa devices with which you want to establish a communication.
- nSerials: numeric value indicating how many Serials are listed in the SerialNumbers array.

Return value:

Returns an integer value of 1 if all ports were successfully closed, 0 otherwise.

4.4.5 Close all opened ports

The following function detects and closes all the ports previously opened with the functions FeasaCom_Open or FeasaCom_OpenSN.

```
int FeasaCom_CloseAll(void);
```

Parameters:

Does not accepts parameters.

Return value:

Returns an integer value of 1 if all the ports where successfully closed and 0 otherwise.

4.5 Send & receive data

The function Send is used to transmit a command from the computer to the Feasa device. Then waits for the reception of the Feasa device response and returns it through a parameter.

4.5.1 Send & receive by Device Path port

A command can be sent to a Feasa device given its Device Path, using the following function:

```
int FeasaCom_Send(const char * DevPath, const char * Command, char *
                  ResponseText);
```

Parameters:

- `DevPath`: string containing the system path of the device.
- Command: the command to send to the Feasa device. There is no need to send the CR + LF characters at the end, because this function does it automatically.
- ResponseText: this is the response string returned by the Feasa device. Please, remember, if necessary to allocate enough memory in the destination variable. This string does not contains the ending CR+LF or CR+LF+EOT characters returned in the raw response.

Return value:

Returns an integer value of 1 if the command was sent successfully, returns 0 if a Syntax Error or a Timeout was detected and returns -1 for other errors (check FeasaCom_GetError

functions to retrieve more information about the error).

This function waits until the end-of-command characters CR + LF or CR + LF + EOT are received or there is a timeout (maximum waiting time without communication) of 3500 ms (3.5 seconds); but can you can change this last parameter using the function FeasaCom_SetResponseTimeout, that will be explained later.

4.5.2 Send & receive by Serial Number

A command can be sent to a Feasa device given its Serial Number, using the following function:

```
int FeasaCom_SendSN(const char * SerialNumber, const char * Command,
                        char * ResponseText);
```

Parameters:

- SerialNumber: the same one used to open the port.
- Command: the command to send to the Feasa Device. There is no need to send the CR + LF characters at the end, because this function does it automatically.
- ResponseText: This is the response string returned by the Feasa device. Please, remember, if necessary to allocate enough memory in the destination variable. This string does not contains the ending CR+LF or CR+LF+EOT characters returned in the raw response.

Return value:

Returns an integer value of 1 if the command was sent successfully, returns 0 if a Syntax Error or a Timeout was detected and returns -1 for other errors (check FeasaCom_GetError functions to retrieve more information about the error).

This function waits until the end-of-command characters CR + LF or CR + LF + EOT are received or there is a timeout (maximum waiting time without communication) of 3500 ms (3.5 seconds); but can you can change this last parameter using the function FeasaCom_SetResponseTimeout, that will be explained later.

4.5.3 Send to & receive from Multiple devices (multi-threaded)

Different commands command can be sent to different devices simultaneously (which port is open) using the function FeasaCom_Send_Multi(). Since the communication is made in parallel using multi-threading techniques, the communication is quick and efficient:

```
int FeasaCom_Send_Multi(int * ReturnValues, const char ** DevPaths, const
        int nPorts, const char ** Commands, char ** Responses);
```

Parameters:

- ReturnValues: numeric array with the responses from each one of the devices.
- DevPaths: string array with list of Device Paths of the Feasa devices with which you want to establish a communication.
- nPorts: numeric value indicating how many Ports are listed in the array DevPaths.
- Commands: array of strings with the commands to send to each one of the Feasa devices. The command in the first position of the array will be sent to the Feasa Device with the port listed in the first position of the array DevPaths, and so on. There is no need to send the CR

+ LF characters at the end, because this function does it automatically.

- Responses: this contains an array of response strings returned by each one of the Feasa devices. Please, remember, if necessary to allocate enough memory in the destination variable. This string does not contains the ending CR+LF or CR+LF+EOT characters returned in the raw response.

Return value:

Returns an integer value of 1 if the command was sent successfully to all the devices, returns 0 if a Syntax Error or a Timeout was detected in any of the devices and returns -1 for other errors occurred in any of the devices connected (check FeasaCom_GetError functions to retrieve more information about the error).

This function waits until the end-of-command characters CR + LF or CR + LF + EOT are received or there is a timeout (maximum waiting time without communication) of 3500 ms (3.5 seconds), but you can change this last parameter using the function FeasaCom_SetResponseTimeout, that will be explained later.

4.5.4 Send to & receive from Multiple devices by Serial Number (multi-threaded)

Different commands command can be sent to different devices simultaneously (which port is open) using the function FeasaCom_SendSN_Multi(). Since the communication is made in parallel using multi-threading techniques, the communication is quick and efficient:

```
int FeasaCom_SendSN_Multi(int * ReturnValues, const char ** SerialNumbers,
      const int nSerials, const char ** Commands, char ** Responses);
```

Parameters:

- ReturnValues: numeric array with the responses from each one of the devices.
- SerialNumbers: string array with the list of Serial Numbers of Feasa devices with which you want to establish a communication.
- nSerials: numeric value indicating how many Serial Numbers are listed in the array SerialNumbers.
- Commands: array of strings with the commands to send to each one of the Feasa devices. The command in the first position of the array will be sent to the Feasa Device with the SN listed in the first position of the array SerialNumbers, and so on. There is no need to send the CR + LF characters at the end, because this function does it automatically.
- Responses: this contains an array of response strings returned by each one of the Feasa devices. Please, remember, if necessary to allocate enough memory in the destination variable. This string does not contains the ending CR+LF or CR+LF+EOT characters returned in the raw response.

Return value:

Returns an integer value of 1 if the command was sent successfully to all the devices, returns 0 if a Syntax Error or a Timeout was detected in any of the devices and returns -1 for other errors occurred in any of the devices connected (check FeasaCom_GetError functions to retrieve more information about the error).

This function waits until the end-of-command characters CR + LF or CR + LF + EOT are received or there is a timeout (maximum waiting time without communication) of 3500 ms (3.5 seconds), but you can change this last parameter using the function FeasaCom_SetResponseTimeout, that will be explained later.

4.5.5 Send to & receive from Multiple devices (No Response)

Unlike FeasaCom_Send_Multi(), the function presented here does not retrieve responses through a parameter and function *FeasaCom_GetResponseByPort()* should be used for that purpose; Equally, list of commands to be sent is done through a plain string and those commands are differentiated with the help of a separator. The reason of this is that, in some simpler languages handling double pointers or pointers to string arrays could be complex.

```
int FeasaCom_Send_Multi_NR(int * ReturnValues, const char ** DevPaths, const
      int nPorts, const char * Commands, const char CommandSeparator);
```

Parameters:

- ReturnValues: numeric array with the responses from each one of the devices.
- DevPaths: string array with list of Device Paths of the Feasa devices with which you want to establish a communication.
- nPorts: numeric value indicating how many Ports are listed in the array DevPaths.
- Commands: strings with the list of commands to send to each one of the Feasa devices, separated using the character specified in the argument *CommandSeparator*. The first command in the list will be sent to the Feasa Device with the port listed in the first position of the array *DevPaths*, and so on. There is no need to send the CR + LF characters at the end, because this function does it automatically.
- CommandSeparator: character used to differentiate between commands in the argument string *Commands*.

Return value:

Returns an integer value of 1 if the command was sent successfully to all the devices, returns 0 if a Syntax Error or a Timeout was detected in any of the devices and returns -1 for other errors occurred in any of the devices connected (check FeasaCom_GetError functions to retrieve more information about the error).

This function waits until the end-of-command characters CR + LF or CR + LF + EOT are received or there is a timeout (maximum waiting time without communication) of 3500 ms (3.5 seconds), but you can change this last parameter using the function FeasaCom_SetResponseTimeout, that will be explained later.

4.5.6 Send to & receive from all devices

A command can be sent to all devices in use (which port is open) using the command FeasaCom_SendToAll(). Since the communication is made in parallel using multi-threading techniques, the communication is quick and efficient:

```
int FeasaCom_SendToAll(int * ReturnValues, const char * Command, char **
                              Responses);
```

Parameters:

- ReturnValues: numeric array with the responses from each one of the devices.
- Command: the command to send to all the Feasa devices. There is no need to send the CR + LF characters at the end, because this function does it automatically.
- Responses: this contains an array of response strings returned by each one of the Feasa devices. Please, remember, if necessary to allocate enough memory in the destination variable. This string does not contains the ending CR+LF or CR+LF+EOT characters returned in the raw response.

Return value:

Returns an integer value of 1 if the command was sent successfully to all the devices, returns 0 if a Syntax Error or a Timeout was detected in any of the devices and returns -1 for other errors occurred in any of the devices connected (check FeasaCom_GetError functions to retrieve more information about the error).

***Important**: the responses obtained with this command will be provided in the very same order in which the ports were opened, so if port for device with SN X001 was opened first and X002 was opened later, then the first response will be from X001 and the second one from X002. In case of using Open_Multi() functions, then the order will be the one in the parameter that contains the list of ports to open.*

This function waits until the end-of-command characters CR + LF or CR + LF + EOT are received or there is a timeout (maximum waiting time without communication) of 3500 ms (3.5 seconds), but you can change this last parameter using the function FeasaCom_SetResponseTimeout, that will be explained later.

4.5.7 Send to & receive from all devices (No Response)

A variation of the previous function has been written to ease the usage from simpler languages, like LabVIEW, in which the handling of double pointers could be complex. The main difference with the previous one is that responses are not returned through parameter; can be retrieved using the function *FeasaCom_GetResponseByPort()* instead:

```
int FeasaCom_SendToAll_NR(int * ReturnValues, const char * Command);
```

Parameters:

- ReturnValues: numeric array with the responses from each one of the devices.
- Command: the command to send to all the Feasa devices. There is no need to send the CR + LF characters at the end, because this function does it automatically.

Return value:

Returns an integer value of 1 if the command was sent successfully to all the devices, returns 0 if a Syntax Error or a Timeout was detected in any of the devices and returns -1 for other errors occurred in any of the devices connected (check FeasaCom_GetError functions to retrieve more information about the error).

This function waits until the end-of-command characters CR + LF or CR + LF + EOT are received or there is a timeout (maximum waiting time without communication) of 3500 ms (3.5 seconds), but

you can change this last parameter using the function FeasaCom_SetResponseTimeout, that will be explained later.

4.5.8 Get response by port

Responses from obtained from devices can be retrieved or recalled using the following function:

```
int FeasaCom_GetResponseByPort(const char * DevPath, char * Response);
```

Parameters:

- DevPath: Device path. The same one you used to open and communicate with the device.
- Response: this is the response string returned by the Feasa device. Please, remember, if necessary to allocate enough memory in the destination variable. This string does not contains the ending CR+LF or CR+LF+EOT characters returned in the raw response.

Return value:

Returns an integer value of 1 if the Response was successfully retrieved, or returns 0 in case that any error occurred.

4.6 Response Timeout

The function below influences the behaviour of the reception. By default, when a command is sent to the Feasa device, the reception finishes when a *timeout is detected* or a special set of characters is received. You can change this timeout with the following function:

```
int FeasaCom_SetResponseTimeout(const unsigned int Timeout);
```

Parameters:

- Timeout: integer with the value of the desired maximum Timeout (maximum waiting time) for the reception. The maximum value for this parameter is 32000 (32 seconds).

Return value:

Returns an integer value of 1 if the timeout has been saved successfully; returns 0 otherwise.

4.6.1 Automatic Timeout calculation

Some functions in the SO Library (Capture-related, etc.) can calculate automatically the timeout needed to perform its action, but this feature can be disabled if the user wish them to use a pre-set timeout given with the function FeasaCom_SetResponseTimeout().

To enable or disable automatic Timeout calculations, the function below have to be used:

```
int FeasaCom_SetResponseTimeoutAuto(const char * DevPath, const int status);
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.
- status: integer that indicates the status for the automatic timeout calculations, being 1 enabled and 0 disabled.

Return value:

Returns an integer value of 1 if the status has been saved successfully; returns 0 otherwise.

4.7 Capture

4.7.1 Capture from a single Analyser

The function Capture is used to perform a measurement on the Feasa Analyser easily, since capture type and range can be specified through the function parameters and timeout is automatically calculated based on the Analyser settings (exposure, averaging factor, etc).

```
int FeasaCom_Capture(const char * DevPath, const int isPWM, const int
                CaptureRange, const int CaptureFrames);
```

Parameters:

- DevPath: string containing the system path of the Analyser.
- isPWM: numeric value which is set to 1 to indicate that the capture performed will be performed in PWM mode (for pulse-modulated/blinking LEDs). 0 is used for continuous (normal) measurements.
- CaptureRange: numeric value for the range to be used in the capture. Use the value 0 for auto-range and other positive integer values for manual ranges (the valid range of values may differ depending on the Analyser, but usually goes from 1 to 5).
- CaptureFrames: number of frames that will be used in a PWM capture. If the value is 0, this parameter will be ignored and the default number of frames will be used. Please, refer to the Analyser manual for more information.

Return value:

Returns an integer value of 1 if the capture was performed successfully, returns 0 if a Syntax Error or a Timeout was detected and returns -1 for other errors (check FeasaCom_GetError functions to retrieve more information about the error).

Note 1: this command is not valid for Feasa LED Spectrometers.

Note 2: some of the capture types allowed in this command may not be available in your Analyser, so please, refer to its user manual. The use of values not allowed could lead to unexpected results.

Note 3: Automatic timeout calculation is enabled by default but can be disabled using function FeasaCom_SetResponseTimeoutAuto().

4.7.2 Capture from all connected Analysers

While the previous function performed a Capture on a given Analyser, the function CaptureFromAll is used to perform a measurement on all Feasa LED Analysers which are currently in use in the Library. The Capture will be performed in those Analysers which port has been opened, no matter what the hardware type or baud-rate are; the process is performed in parallel using multi-threading techniques, so the total capture time will be equal to the lowest capture time of all Analysers in use. As in the previous function the capture type and range can be specified through the function parameters and timeout is automatically calculated for each Analyser based on its settings (exposure, averaging factor, etc.).

```
int FeasaCom_CaptureFromAll(int * ReturnValues, const int isPWM, const
```

```
int CaptureRange, const int CaptureFrames);
```

Parameters:

- ReturnValues: numeric array with list of return values for each capture.
- isPWM: numeric value which is set to 1 to indicate that the capture performed will be performed in PWM mode (for pulse-modulated/blinking LEDs). 0 is used for continuous (normal) measurements.
- CaptureRange: numeric value for the range to be used in the capture. Use the value 0 for auto-range and other positive integer values for manual ranges (the valid range of values may differ depending on the Analyser, but usually goes from 1 to 5).
- CaptureFrames: number of frames that will be used in a PWM capture. If the value is 0, this parameter will be ignored and the default number of frames will be used. Please, refer to the LED Analyser manual for more information.

Return value:

Returns an integer value of 1 if the capture was successfully performed in in all the Analysers, returns 0 if a Syntax Error or a Timeout was detected in any Analyser and returns -1 for other errors occurred in any of the Analysers connected (check FeasaCom_GetError functions to retrieve more information about the error).

Note 1: this command is not valid for Feasa LED Spectrometers.

Note 2: some of the capture types allowed in this command may not be available in your Analyser, so please, refer to its user manual for more information. The use of values not allowed could lead to unexpected results.

Note 3: Automatic timeout calculation is enabled by default but can be disabled using function FeasaCom_SetResponseTimeoutAuto().

4.7.3 Capture from a single Spectrometer

The function SpectrometerCapture is used to perform a measurement on a Feasa Spectrometer easily, since capture mode and exopsure can be specified through the function parameters and timeout is automatically calculated based on the Spectrometer settings (exposure, averaging factor, etc.).

```
int FeasaCom_SpectrometerCapture(const char * DevPath, const int isPWM,
        const int UseCustomExposure, const float ExposureTime);
```

Parameters:

- DevPath: string containing the system path of the Analyser.
- isPWM: numeric value which is set to 1 to indicate that the capture performed will be performed in PWM mode (for pulse-modulated/blinking LEDs). 0 is used for continuous (normal) measurements. This makes the Library to send the command SETPWM to the Spectrometer.
- UseCustomExposure: numeric value indicating whether the exposure time is provided through the function or auto-exposure is used instead. Use the value 0 to instruct the Spectrometer to calculate the exposure time automatically (auto-exposure) or use 1 if you

wish to specify this exposure time through the parameter ExposureTime.

- ExposureTime: value of time expressed in ms, used for specifying the exposure time to perform a capture in the Spectrometer. This value is ignored when UseCustomExposure is set to 0.

Return value:

Returns an integer value of 1 if the capture successfully performed, returns 0 if a Syntax Error or a Timeout was detected; for modern firmware versions, it also returns 0 when Dark Capture has not been performed or Signal is not Stable (PWM required). Returns -1 for other errors (check FeasaCom_GetError functions to retrieve more information about the error).

Note 1: this command is not valid for Feasa LED Analysers.

Note 2: PWM mode is available in S2 Spectrometers and beyond.

Note 3: Automatic timeout calculation is enabled by default but can be disabled using function FeasaCom_SetResponseTimeoutAuto().

4.7.4 Capture from all connected Spectrometers

While the previous function performed a Capture on a single Spectrometer, the function CaptureFromAllSpectrometers is used to perform a measurement on all Feasa Spectrometers which are currently in use in the Library. The Capture will be performed in those Spectrometers which port has been opened, no matter what the hardware type or baud-rate are; the process is performed in parallel using multi-threading techniques, so the total capture time will be equal to the lowest capture time of all Spectrometers in use. As in the previous function the capture mode and exposure time can be specified through the function parameters and timeout is automatically calculated for each Spectrometer based on its settings (exposure, averaging factor, etc.).

```
int FeasaCom_CaptureFromAllSpectrometers(int * ReturnValues, const int
    isPWM, const int UseCustomExposure, const float ExposureTime);
```

Parameters:

- ReturnValues: numeric array with list of return values for each capture.
- isPWM: numeric value which is set to 1 to indicate that the capture performed will be performed in PWM mode (for pulse-modulated/blinking LEDs). 0 is used for continuous (normal) measurements. This makes the Library to send the command SETPWM to the Spectrometer.
- UseCustomExposure: numeric value indicating whether the exposure time is provided through the function or auto-exposure is used instead. Use the value 0 to instruct the Spectrometer to calculate the exposure time automatically (auto-exposure) or use 1 if you wish to specify this exposure time through the parameter ExposureTime.
- ExposureTime: value of time expressed in ms, used for performing a capture in the Spectrometer with a given exposure time. This value is ignored when UseCustomExposure is set to 0.

Return value:

Returns an integer value of 1 if the capture was successfully performed in in all the Spectrometers, returns 0 if a Syntax Error or a Timeout was detected in any Spectrometer; for modern firmware versions, it also returns 0 when Dark Capture has not been performed or Signal is not Stable (PWM required). Returns -1 for other errors occurred in any of the Spectrometers connected (check FeasaCom_GetError functions to retrieve more information about the error).

Note 1: this command is not valid for Feasa LED Analysers.

Note 2: PWM mode is available in S2 Spectrometers and beyond.

Note 3: Automatic timeout calculation is enabled by default but can be disabled using function FeasaCom_SetResponseTimeoutAuto().

4.7.5 Spectrometer dark Adjustment

The function SpectrometerDark is used to perform a dark capture or dark level adjustment on a Feasa Spectrometer easily, since capture mode and exposure can be specified through the function parameters and timeout is automatically calculated based on the Spectrometer settings (exposure, averaging factor, etc.).

```
int FeasaCom_SpectrometerDark(const char * DevPath, const int isPWM,
      const int UseCustomExposure, const float ExposureTime);
```

Parameters:

- DevPath: string containing the system path of the Feasa Spectrometer.
- isPWM: numeric value which is set to 1 to indicate that the adjustment will be performed in PWM mode (for pulse-modulated/blinking LEDs). 0 is used for continuous (normal) measurements. This makes the Library to send the command SETPWM to the Spectrometer.
- UseCustomExposure: numeric value indicating whether the exposure time is provided through the function or auto-exposure is used instead. Use the value 0 to instruct the Spectrometer to calculate the exposure time automatically (auto-exposure) or use 1 if you wish to specify this exposure time through the parameter ExposureTime.
- ExposureTime: value of time expressed in ms, used for specifying the exposure time to perform a dark adjustment in the Spectrometer. This value is ignored when UseCustomExposure is set to 0.

Return value:

Returns an integer value of 1 if the capture successfully performed, returns 0 if a Syntax Error or a Timeout was detected. Returns -1 for other errors (check FeasaCom_GetError functions to retrieve more information about the error).

Note 1: this command is not valid for Feasa LED Analysers.

Note 2: PWM mode is available in S2 Spectrometers and beyond.

Note 3: Automatic timeout calculation is enabled by default but can be disabled using function FeasaCom_SetResponseTimeoutAuto().

4.8 Sequence

4.8.1 Setting up sequence parameters

Before testing a sequence, some parameters must be configured through the following function:

```
int FeasaCom_Sequence_Setup(const char * DevPath, const int StartDelay,
const int CaptureTime, const int TimeBetweenCaptures, const int SampleCount,
                            const int toFlash);
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.
- StartDelay: time in ms to wait before performing the sequence capture, from 0ms to 999ms.
- CaptureTime: time in ms for each sample being tested, from 1ms to 999ms. Compared to the standard Capture commands, here, instead of using ranges, a fixed exposure time is used in order to allow full control over the timing of the sequence. The longer the capture time, the more sensitive is the capture, but lower the resolution and also the possible number of frames.
- TimeBetweenCaptures: delay time in ms to wait between the end of the capture of a sample and the beginning of the next one, from 0ms to 999ms. This is particularly useful for slow signals, which continuous sampling would make a buffer over-run.
- SampleCount: total number of samples to be acquired, with a new sample being recorded by each capture. The maximum number of samples depends on how many Fibers are tested in the Sequence Capture, being 9999 when only one fiber is tested or 3500 when all channels are tested simultaneously.
- toFlash: integer. Set to 0 to store changes in RAM, 1: to store changes in flash memory.

Return value:

Returns 1 if all settings were applied successfully, 0 when a timeout or syntax error was detected or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

***Note*:** the total capture time is: $T_{START} + N_{SAMPLES} * (T_{CAPT} + T_{WAIT}) + T_{COMMUNICATIONS\text{-}DELAY}$

4.8.2 Perform Sequence Capture

A Sequence Capture takes the measurement samples according to the settings previously given:

```
int FeasaCom_Sequence_Capture(const char * DevPath, const int Fiber);
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.
- Fiber: integer representing the fiber or channel number in which the sequence capture will be performed. Set this value to zero in order to perform a sequence capture in all the channels.

Return value:

Returns 1 if all settings were applied successfully, 0 when a timeout or syntax error was detected or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

***Note*:** Linear Intensity mode (PUTLIN/SETLIN) is recommended when using Capture sequence, since intensity steps in Logarithmic mode (factory default) may confuse the user; also the resolution for low intensity values is better in Linear mode.

4.8.3 Reading CIE1931 values

Once the Capture Sequence has finished, xy chromaticity values for CIE931 can be read using the following commands:

```
int FeasaCom_Sequence_ReadxyI(const char * DevPath, const int Fiber, float *
            xValues, float * yValues, int * IntensityValues);
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.
- Fiber: integer representing the fiber or channel number in which the sequence capture will be performed.
- xValues: pointer to array of float (single precision) in which values for x coordinate of each sample will be saved.
- yValues: pointer to array of float (single precision) in which values for y coordinate of each sample will be saved.
- IntensityValues: pointer to array of integer in which Intensity values for each sample will be saved.

Return value:

Returns 1 if all settings were applied successfully, 0 when a timeout or syntax error was detected or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

4.8.4 Reading RGBI values

Once the Capture Sequence has finished, xy chromaticity values for CIE931 can be read using the following commands:

```
 int FeasaCom_Sequence_ReadRGBI(const char * DevPath, const int Fiber,
unsigned char * RedValues, unsigned char * GreenValues, unsigned char *
                  BlueValues, int * IntensityValues)
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.
- Fiber: integer representing the fiber or channel number in which the sequence capture will be performed.
- RedValues: pointer to array of unsigned 8-bit integer in which Red values of each sample

will be saved.

- GreenValues: pointer to array of unsigned 8-bit integer in which Green values of each sample will be saved.
- BlueValues: pointer to array of unsigned 8-bit integer in which Green values of each sample will be saved.
- IntensityValues: pointer to array of integer in which Intensity values for each sample will be saved.

Return value:

Returns 1 if all settings were applied successfully, 0 when a timeout or syntax error was detected or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

4.8.5 Reading HSI values

Once the Capture Sequence has finished, Hue, Saturation and Intensity values can be read using the following commands:

```
int FeasaCom_Sequence_ReadHSI(const char * DevPath, const int Fiber, float *
        HueValues, int * SaturationValues, int * IntensityValues);
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.
- Fiber: integer representing the fiber or channel number in which the sequence capture will be performed.
- HueValues: pointer to array of float (single precision) in which Hue values for each sample will be saved.
- SaturationValues: pointer to array of integer in which Saturation values for each sample will be saved.
- IntensityValues: pointer to array of integer in which Intensity values for each sample will be saved.

Return value:

Returns 1 if all settings were applied successfully, 0 when a timeout or syntax error was detected or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

4.8.6 Reading Intensity values

Once the Capture Sequence has finished, Intensity values can be read using the following commands:

```
int FeasaCom_Sequence_ReadIntensity(const char * DevPath, const int Fiber,
                            int * IntensityValues);
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.
- Fiber: integer representing the fiber or channel number in which the sequence capture will be performed.
- IntensityValues: pointer to array of integer in which Intensity values for each sample will be saved.

Return value:

Returns 1 if all settings were applied successfully, 0 when a timeout or syntax error was detected or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

4.8.7 Reading CCT values

Once the Capture Sequence has finished, Intensity values can be read using the following commands:

```
int FeasaCom_Sequence_ReadCCT(const char * DevPath, const int Fiber, int *
                    CCTValues, float * deltauvValues);
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.
- Fiber: integer representing the fiber or channel number in which the sequence capture will be performed.
- CCTValues: pointer to array of integer in which CCT values for each sample will be saved.
- deltauvValues: pointer to array of float (single precision) in which delta u'v' values for each sample will be saved.

Return value:

Returns 1 if all settings were applied successfully, 0 when a timeout or syntax error was detected or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

4.8.8 Reading Wavelength values

Once the Capture Sequence has finished, Intensity values can be read using the following commands:

```
int FeasaCom_Sequence_ReadWavelength(const char * DevPath, const int Fiber,
                        int * WavelengthValues);
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.
- Fiber: integer representing the fiber or channel number in which the sequence capture will be performed.
- WavelengthValues: pointer to array of integer in which CCT values for each sample will be saved.

Return value:

Returns 1 if all settings were applied successfully, 0 when a timeout or syntax error was detected or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

4.8.9 Determine blinking pattern

Based on the intensity results of a previous Capture is possible to extract the timing and intensity pattern of a blinking LED using the following function:

```
int FeasaCom_Sequence_GetPattern(const char * DevPath, const int *
  IntensityValues, int * StatusCount, int * PatternTimes, int *
                        PatternIntensities);
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.
- IntensityValues: pointer to array of integer that contains all Intensity values from the previous Sequence Capture.
- StatusCount: pointer to integer that contains how many levels or states were extracted from the given set of Intensity values.
- PatternTimes: pointer to array of integer containing the time in ms for each level in the pattern.
- PatternIntensities: pointer to array of integer containing the intensity value for each level in the pattern.

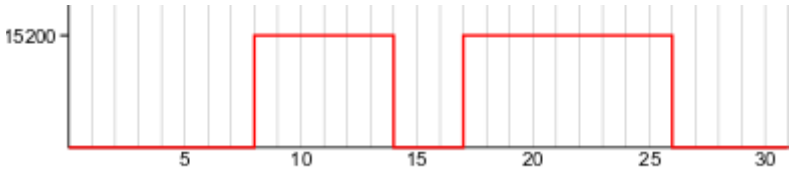
Return value:

Returns 1 if all settings were applied successfully or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

Note: there is a timing tolerance of ±2 samples.

Example:

In the following test case, there is a blinking LED, which pattern has been sampled at 15ms/sample (Capture Time), with no initial delay and no wait time between samples:



As can be seen in the diagram above, there are two single pulses, the one starting at sample 8, which means that it starts at 120ms, with a pulse width of 6 samples (90ms), a time between pulses of 3 samples (45ms), a second pulse lasting 9 samples (135ms) and 5 last samples (75ms) in which there is no light.

Based on the data exposed, the values returned by the function would be:

StatusCount = 5

PatternTimes = [120, 90, 45, 135, 75]

PatternIntensities = [0, 15200, 0, 15200, 0]

4.8.10 Determine pattern for sweeping indicators

Many modern automotive indicators follow animated sweeping patterns which need to be tested using the LED Analyser. This library contains a special function that allows to extract that timing and intensity pattern for easy further analysis:

```
int FeasaCom_Sequence_GetSweepingPattern(const char * DevPath, const int
  LEDCount, const int isOffToOn, int * LowTimes, int * HighTimes, int *
                              IntensityValues)
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.
- LEDCount: integer for indicating the number of Fibers (LEDs) that will be implicated in the measurement of the pattern.
- isOffToOn: integer, set value 1 to indicate that the pattern starts with all LEDs Off and then those LEDs start to switch on sequentially, ending with all LEDs (or the last LED) On. Use value 0 to indicate that the animation pattern is reversed, going from all LEDs On to all LEDs Off.
- LowTimes: pointer to array of integers containing the times in ms in which the intensity of he LEDs under test are Low.
- HighTimes: pointer to array of integers containing the times in ms in which the intensity of the LEDs under test are High.
- IntensityValues: pointer to array of integers containing the Intensities of the LEDs

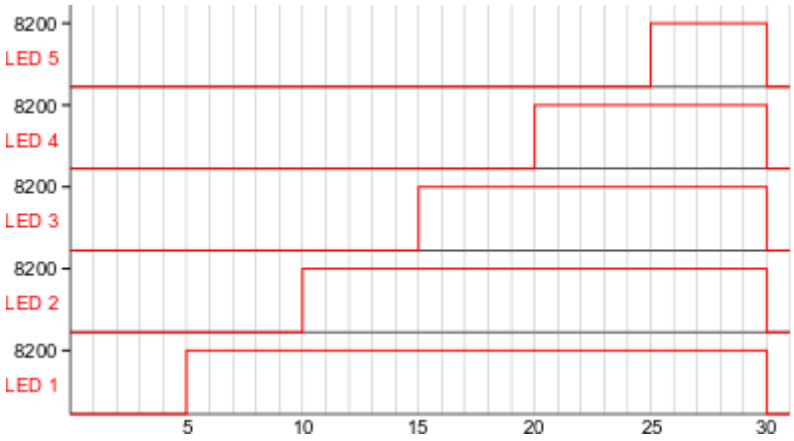
Return value:

- Returns 1 if all settings were applied successfully or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

Note: there is a timing tolerance of ±2 samples.

Example:

In the following test case, there is an indicator with five LEDs following a sweeping pattern, in which LEDs switch on sequentially, one by one:



Let's consider that the capture time is 20ms, the number of samples is 40, and as mentioned, there are 5 LEDs, so parameters should be:

LEDCount = 5

isOffToOn = 1

Based on the same logic described in the function FeasaCom_Sequence_GetPattern(), the results would be:

LowTimes = [100, 200, 300, 400, 500]

HighTimes = [500, 400, 300, 200, 100]

IntensityValues = [8200, 8200, 8200, 8200, 8200] (this is quite simplistic and values can change from LED to LED).

4.8.11 Get Frequency and Duty Cycle

Based on the intensity results of a previous Capture is possible to extract the Frequency, Duty Cycle and the number of cycles captured, using the function below:

```
int FeasaCom_Sequence_GetFrequency(const char * DevPath, const int *
  IntensityValues, float * Frequency, float * DC, int * CycleCount)
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.
- IntensityValues: pointer to array of integer that contains all Intensity values from the previous Sequence Capture.
- Frequency: pointer to integer that contains the calculated frequency of the blinking LED.
- PatternTimes: pointer to array of integer containing the time in ms for each level in the pattern.
- PatternIntensities: pointer to array of integer containing the intensity value for each level in

the pattern.

Return value:

Returns 1 if all settings were applied successfully or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

***Note*: this function will only work for periodic blinking or single pulse patterns. The higher the frequency of the blinking, the higher the number of cycles required in order to minimize the error. For testing frequency accurately, it is highly recommended to use the lower sampling time possible, in order to improve the time resolution, taking into account that there is a minimum sampling time which depends on the intensity of the LED, value below which the Analyser will not have enough sensitivity to measure any light (this is found by experimentation).**

Note: there is a timing tolerance of ±2 samples.

4.8.12 Finding out test settings

In order to ease the work of finding some valid Test settings for the Sequence Capture, this library contains a function that finds out those test parameters for repetitive signals.

```
   int FeasaCom_Sequence_FindTestSettings(const char * DevPath, const int
   TotalLEDCount, const int FiberToTest, const int SignalSpeed, const int
BlinkingSpeed, const int MinCycleCount, const int TimeResolutionIsImportant,
           int * CaptureTime, int * WaitTime, int * SampleCount);
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.
- TotalLEDCount: integer indicating the total number of LEDs you wish to measure. This function only measures one at a time, but this parameter is necessary to know the maximum number of samples available.
- FiberToTest: integer indicating the Fiber that will be used to find out the test parameters.
- SignalSpeed: integer value in the range of [0, 10] indicating the speed of the LED signal under test, being 0 a very low speed signal (<=1Hz) and 10 a high speed one (>=40Hz).
- BlinkingSpeed: integer value in the range of [0, 10] indicating the speed of the blinking (time of LED being On) of the LED under test, being 0 a very slow blinking pattern and being 1 a very quick one.
- MinCycleCount: integer indicating the minimum number of Cycles that has to be tested. If measurement of frequency or duty cycle is needed, when pattern speed is moderately high, at least 5 cycles is recommended to improve timing resolution; for slower speeds 3 might be enough, even 1 for very slow signals.
- TimeResolutionIsImportant: integer with a value of 1 to indicate that time resolution is prioritary (to measure time or freuency), while a value of 0 means that a balance between time resolution and intensity will be obtained.
- CaptureTime: pointer to integer containing the suggested Capture time per sample, in ms.
- WaitTime: pointer to integer containing the suggested wait time or interval between

samples/captures.

- SampleCount: pointer to integer containing the suggested number of samples to be used for the CaptureSequence.

Return value:

Returns 1 if all settings were applied successfully or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

***Note 1*:** this function test only the given channel number, but does not mean that this very same settings could not work in multi-channel sequence capture. So in case you wish to test more than one channel simultaneously you would need to interpret the results and extrapolate them to be used in different channels, if necessary.

***Note 2*:** test values suggested by this function are not accurate and might require a fine tuning.

4.8.13 Set pattern threshold for Zero

When calculating a pattern, by default, the SO Library uses a threshold value for zero of 15% the maximum intensity value; this value is used to determine when a logical zero begins, since below this value, the level will be considered as a zero (this can be looked as a hysteresis level). However, this threshold can be changed using the following function:

```
int FeasaCom_Sequence_SetPatternThresholdLow(const char * DevPath, const int
                                    Intensity);
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.
- Intensity: integer indicating the value below which the logical output will be considered a zero. If this value is set to “-1” the Library will calculate the threshold automatically.

Return value:

Returns 1 if the value was applied successfully or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

4.8.14 Set pattern threshold for One

When calculating a pattern, by default, the SO Library uses a threshold value for One of 85% the maximum intensity value; this value is used to determine when a logical one begins, since above this value, the level will be considered as a one (this can be looked as a hysteresis level). However, this threshold can be changed using the following function:

```
int FeasaCom_Sequence_SetPatternThresholdHigh(const char * DevPath, const
                                int Intensity);
```

Parameters:

- DevPath: string containing the system path of the Feasa Device.

- Intensity: integer indicating the value above which the logical output will be considered a one. If this value is set to “-1” the Library will calculate the threshold automatically.

Return value:

Returns 1 if the value was applied successfully or -1 if any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

4.9 Other communication tools

4.9.1 Check whether a port exists

Sometimes, it's useful to check all the ports existing on a computer in order to detect where a Feasa Feasa device is connected. In order to do this, the following function can be used:

```
int FeasaCom_IsPortAvailable(const char * DevPath);
```

Parameters:

- DevPath: string containing the the device path to check

Return value:

Returns 1 if the port exist or 0 if doesn't.

Note: *It doesn't detect if there is a Feasa device connected to that port.*

4.9.2 Retrieve ports list

The following function retrieves a list of all the ports available in the computer.

```
int FeasaCom_ListPortsDetected(char ** ListOfPortsDetected);
```

Parameters:

- ListOfPortsDetected: a pointer to an array of characters-array which is used to store the list of ports detected. For example, if ports /dev/ttyS0 and /dev/ttyS1 and detected, then the content of the array will be {“/dev/ttyS0”, “/dev/ttyS1”}.

Return value:

Returns a numeric integer with the number of ports detected.

Note: *It doesn't detect if there is a Feasa device connected to each port.*

4.9.3 Retrieve ports list in text format

Similar to the previous function, this one retrieves a list of all the ports available in the computer, but in text format.

```
int FeasaCom_ListPortsDetectedTxt(char * ListOfPortsDetected, const char *
                                  Delimiter);
```

Parameters:

- ListOfPortsDetected: a pointer to an array of characters which is used to store the list of ports detected. For example, if ports /dev/ttyS0 and /dev/ttyS1 and detected, then the content of the string will be “/dev/ttyS0;/dev/ttyS1”.
- Delimiter: this is the character that will be used as a delimiter in the string of ports detected. If Delimiter is '-', the output of the previous example will be “/dev/ttyS0;/dev/ttyS1”. If Delimiter is ' ', the output of the previous example will be “/dev/ttyS0 /dev/ttyS1”.

Return value:

Returns a numeric integer with the number of ports detected.
Note: *It doesn't detect if there is a Feasa device connected to each port.*

4.9.4 Get Baudrate

This function is used to find out the baud rate used on a port which has been previously opened. It is useful when baud rate auto-detection is used, since the current baud rate can be obtained and use it the next time the port is opened (faster connection).

```
long FeasaCom_GetBaudrate(const char * DevPath);
```

Parameters:

- `DevPath`: string containing the system path of the device. The same one you used to open the port.

Return value:

Returns a value containing the numeric baud rate, otherwise the returned value is 0.

4.9.5 Get port by Serial Number

This function is used to find out the Device Path to which is connected a Feasa device given its Serial Number. It is useful when a port has been opened by Serial Number.

```
int FeasaCom_GetPortBySN(char * DevPath, const char * SerialNumber);
```

Parameters:

- DevPath: parameter used to return a string containing the system path of the device.
- SerialNumber: Serial number of the device of interest.

Return value:

Returns 1 if the port was found or -1 if the port is not found (check FeasaCom_GetError functions to retrieve more information about the error).

4.9.6 Get Serial Number by Port

This function is used to find out the Serial Number of a Feasa device given its Device Path. It is useful when a port has been opened by Device Path.

```
int FeasaCom_GetSNByPort(char * SerialNumber, const char * DevPath);
```

Parameters:

- SerialNumber: string containing the Serial number belonging to the Port given in the parameter DevPath
- DevPath: path of the device of interest.

Return value:

Returns 1 if the Serial Number was found or -1 otherwise (check FeasaCom_GetError functions to retrieve more information about the error).

4.9.7 Get list of ports opened

It is possible to retrieve a list with the Device Paths of all the ports currently opened, using the function below:

```
int FeasaCom_GetOpenedPorts(char ** DevPaths);
```

Parameters:

- DevPaths: array of strings containing the Device Paths for the ports currently open, returned in the same order used to open them.

Return value:

Returns the number of ports opened in the SO Library or -1 in case of error (check FeasaCom_GetError functions to retrieve more information about the error).

4.9.8 Get list of ports opened (through string)

It is possible to retrieve a list with the Device Paths of all the ports currently opened, using the function below:

```
int FeasaCom_GetOpenedPortsS(char * DevPaths, const char Delimiter);
```

Parameters:

- DevPaths: single string containing list of Device Paths for the ports currently opened, returned in the same order used to open them; paths are separated with the Delimiter character given.
- Delimiter: character used to differentiate each path in the list of paths (string) returned through the parameter DevPaths.

Return value:

Returns the number of ports opened or -1 in case of error (check FeasaCom_GetError functions to retrieve more information about the error).

4.9.9 Get device type

The type of device connected to a port can be verified using the function below:

```
int FeasaCom_GetDeviceType(const char * DevPath, char * DeviceType);
```

Parameters:

- `DevPath`: string containing the system path of the device. The same one you used to open the port.
- DeviceType: pointer to string used to store the Device Type connected to the port specified by CommPort; in case of using daisy-chain, the function will return the Type for the device currently owning the bus. The possible values returned are “FUNCTIONAL”, “ICT”, “LOWLIGHT”, “IR”, “LEGEND”, “DISPLAY”, “SPECTROMETER”, “RGB”, “RGBREF” and “UNKONWN”.

Return value:

Returns 1 if the function succeed or 0 otherwise.

4.10 Daisy-Chain

The SO Library contains a set of functions that can be used on a Daisy-chain topology to simplify the testing of the bus devices.

First of all it is necessary to provide to the Library the list of devices attached to the bus, which can be done using the function FeasaCom_DaisyChain_Add, then a bus capture can performed using the function FeasaCom_DaisyChain_Capture.

***Important**: all these commands have to be executed only once the port is open. If the port is closed, the process has to be repeated if is re-opened, since the bus information will be cleared.*

4.10.1 Add device

This function is used to report a device attached to the daisy-chain bus (not the one connected to the computer).

```
int FeasaCom_DaisyChain_Add(const char * DevPath, const char *
                            SerialNumber);
```

Parameters:

- DevPath: string containing the system device path of the BUS master. The same one you used to open the port.
- SerialNumber: Serial Number of the device to add.

Return value:

Returns an integer value of 1 if the device was detected and added successfully, returns 0 otherwise.

Note: this command has to be used every time the port is opened.

4.10.2 Remove device

This function is used to remove a device previously added.

```
int FeasaCom_DaisyChain_Del(const char * DevPath, const char *
                            SerialNumber);
```

Parameters:

- DevPath: string containing the system device path of the BUS master. The same one you used to open the port.
- SerialNumber: Serial Number of the device to add.

Return value:

Returns an integer value of 1 if the device was successfully deleted, returns 0 otherwise.

4.10.3 Clear all devices

This function is used to remove all devices previously added.

```
int FeasaCom_DaisyChain_Clear(const char * DevPath);
```

Parameters:

- DevPath: string containing the system device path of the BUS master. The same one you used to open the port.

Return value:

Returns an integer value of 1 if the device was successfully deleted and 0 otherwise.

4.10.4 Send Command to daisy-chain device

This function is used to send a command to a device in the daisy-chain bus, previously added, based on its Serial Number.

```
int FeasaCom_DaisyChain_Send(const char * DevPath, char * SerialNumber,
              const char * Command, char * ResponseText);
```

Parameters:

- DevPath: string containing the system Device path of the BUS master. The same one you used to open the port.
- SerialNumber: string with the serial number of the device which you want to communicate with.
- Command: string with the Feasa Command to be sent to the device in the bus.
- ResponseText: pointer to string where the response from the Feasa Device will be stored. Be cautious to reserve enough memory space for the expected response.

Return value:

Returns an integer value of 1 if the device was successfully sent, returns 0 otherwise.

4.10.5 Capture from Analysers

Use this function to perform a broadcast-style capture in all the Analysers attached to the bus. This functions ensures that the capture is finished in all the Analysers and calculates automatically the

needed total capture timeout taking into account parameters like the exposure factor, the averaging factor, etc.

```
int FeasaCom_DaisyChain_Capture(const char * DevPath, const int isPWM, const
                int CaptureRange, const int CaptureFrames);
```

Parameters:

- DevPath: string containing the system Device path of the BUS master. The same one you used to open the port.
- isPWM: numeric value which is set to 1 to indicate that the capture performed will be performed in PWM mode (for pulse-modulated/blinking LEDs). 0 is used for continuous (normal) measurements.
- CaptureRange: numeric value for the range to be used in the capture. Use the value 0 for auto-range and other positive integer values for manual ranges (the valid range of values may differ depending on the Analyser, but usually goes from 1 to 5).
- CaptureFrames: number of frames that will be used in a PWM capture. If the value is 0, this parameter will be ignored and the default number of frames will be used. Please, refer to the LED Analyser manual for more information.

Return value:

Returns an integer value of 1 if the command was sent successfully, returns 0 if a Syntax Error or a Timeout was detected and returns -1 for other errors (check FeasaCom_GetError functions to retrieve more information about the error).

Note 1: this command is not valid for Feasa LED Spectrometers.

Note 2: some of the capture types allowed in this command may not be available in your LED Analyser, so please, refer to its user manual for more information. The use of values not allowed could lead to unexpected results.

4.10.6 Capture from Spectrometers

Use this function to perform a broadcast-style capture in all the Spectrometers attached to the bus. This functions ensures that the capture is finished in all the Spectrometers and calculates automatically the needed total capture timeout taking into account parameters like exposure time, averaging factor, etc.

```
int FeasaCom_DaisyChain_SpectrometerCapture(const * DevPath, const int
     isPWM, const int UsePresetExposure, const int ExposureTime);
```

Parameters:

- DevPath: string containing the system Device path of the BUS master. The same one you used to open the port.
- isPWM: numeric value which is set to 1 to indicate that the adjustment will be performed in PWM mode (for pulse-modulated/blinking LEDs). 0 is used for continuous (normal) measurements. This makes the Library to send the command PUTPWM to the Spectrometer.
- UsePresetExposure: numeric value used to indicate whether the exposure time used to

perform a capture is given through a parameter (0) or from a pre-set value (1). In case of using a value of 1, if parameter ExposureTime has a value greater or equal than 0, then this value will be sent (internally/automatically) to all the Spectrometers in the bus using the command PUTCUSTOMTIME, but if the parameter ExposureTime has a value lower than 0, then the value is not sent and the user should set the Exposure time for each Spectrometer manually using the command SETCUSTOMTIME OR PUTCUSTOMTIME.

- ExposureTime: value of time expressed in ms, used to specify the exposure time to perform a capture in the Spectrometer. 0 means auto. For negative values, when UsePresetExposure is 1, means that the value will not be transmitted to the Spectrometer, but when UserPresetExposure is 0 auto will be used.

Return value:

Returns an integer value of 1 if the capture was successfully performed in in all the devices, returns 0 if a Syntax Error or a Timeout was detected in any device; for modern firmware versions, it also returns 0 when Dark Capture has not been performed or Signal is not Stable (PWM required). Returns -1 for other errors occurred in any of the devices connected (check FeasaCom_GetError functions to retrieve more information about the error).

Note 1: this command is not valid for Feasa LED Analysers and is only available for S2 Spectrometers and beyond (S1 does not have daisy-chain capabilities).

Note 2: this function does not have control over the PWM mode, so in case that PWM capture is needed, it has to be previously enabled (for S2 Spectrometer and beyond).

4.10.7 Dark adjustment of Spectrometers

Use this function to perform a broadcast-style Dark adjustment in all the Spectrometers attached to the bus. This functions ensures that the adjustment is finished in all the Spectrometers and calculates automatically the needed total timeout taking into account parameters like exposure time, averaging factor, etc.

```
int FeasaCom_DaisyChain_SpectrometerDark(const * DevPath, const int
   isPWM, const int UsePresetExposure, const float ExposureTime);
```

Parameters:

- DevPath: string containing the system Device path of the BUS master. The same one you used to open the port.
- isPWM: numeric value used to indicate whether the PWM mode will be used (for PWM-modulated or blinking LEDs). This value is set to 1 if PWM test mode will be used and is set to 0 when a continuous LED want to be tested.
- UsePresetExposure: numeric value used to indicate whether the exposure time used to perform a capture is given through a parameter (0) or from a pre-set value (1). In case of using a value of 1, if parameter ExposureTime has a value greater or equal than 0, then this value will be sent (internally/automatically) to all the Spectrometers in the bus using the command PUTCUSTOMTIME, but if the parameter ExposureTime has a value lower than 0, then the value is not sent and the user should set the Exposure time for each Spectrometer manually using the command SETCUSTOMTIME OR PUTCUSTOMTIME.
- ExposureTime: value of time expressed in ms, used to specify the exposure time to perform

a capture in the Spectrometer. 0 means auto. For negative values, when UsePresetExposure is 1, means that the value is not transmitted, but when UsePresetExposure is 0 auto will be used.

Return value:

Returns an integer value of 1 if the adjustment was successfully performed in in all the Spectrometers, returns 0 if a Syntax Error or a Timeout was detected in any Spectrometer. Returns -1 for other errors occurred in any of the Spectrometers connected (check FeasaCom_GetError functions to retrieve more information about the error).

Note: this command is not valid for Feasa LED Analysers and is only available for S2 Spectrometers and beyond (S1 does not have daisy-chain capabilities).

4.11 External Trigger

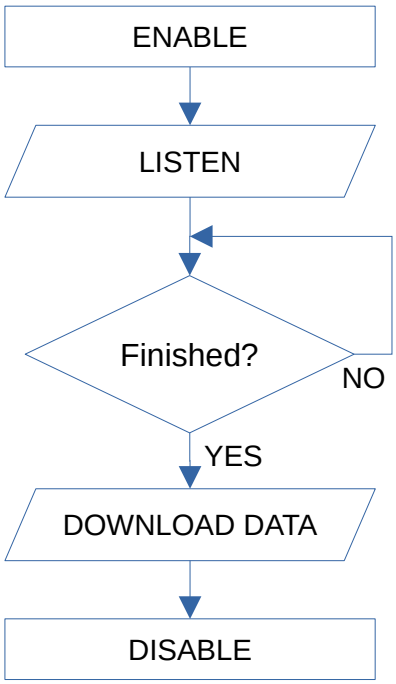
Some Feasa Devices have a feature called External trigger, which is a special hardware port that allows to start a measurement, so the functions below provide a mechanism to wait for that capture event and wait until the measurement has finished.

It is not recommended to use a push button as a trigger source, since this will create a bouncing effect/jitter, that can trigger several captures or make the behavior not stable.

When a trigger occurs in the Feasa Device, once the measurement is finished, the string “OK” is returned, but the unit can be programmed to return the measurement results after it (HSI, xyI, RGBI, etc.). Can be configured through the Library function FeasaCom_ExternalTrigger_Enable().

Note that if the trigger post-delay is used (value > 0ms) then, the Feasa Device will return OK or the desired result after the capture but the device will not attend more trigger requests until the delay has elapsed. Can be configured through the function FeasaCom_ExternalTrigger_Enable().

The basic usage schema is the following:



4.11.1 Enable and configure external trigger

The function below is used to enable the External trigger feature of the Feasa Device and configure its behavior.

```
void FeasaCom_ExternalTrigger_Enable(const char * DevPath, const int
CaptureRange, const int isPWM, const char * OutputType, const int PreDelay,
                const int PostDelay, const int toFlash);
```

Parameters:

- DevPath: string containing the system Device path of the BUS master. The same one you used to open the port.
- CaptureRange: numeric value with the capture range to use (1-5); use a value of 0 for auto.
- IsPWM: numeric value to indicate whether a PWM capture will be used (1) or not (0).
- OutputType: string to indicate what result will be returned by the Feasa device when the capture/measurement finishes. If “NONE” is passed, then the device will not return any result after the “OK” that signals the capture end. If the user wants to get the results of the measurements, one of these strings can be passed: “HSI”, “XYI”, “RGBI”, “CCTI”, “ABSINT”, “WSI”.
- PreDelay: numeric value to indicate the delay between the trigger event and the actual measurement, which can be used to compensate for the LED start-up time; this value is in millisecond, except for the Feasa Spectrometer, which is in microsecond.
- PostDelay: numeric value to indicate the delay after the trigger event + capture, which can be used as a de-bouncing mechanism; this value is in millisecond, except for the Feasa Spectrometer, which is in microsecond.

Return value:

Returns an integer value of 1 when the function succeeded, 0 if any error is detected on any parameter, or -1 in case that any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

The use of this function does not trigger a capture nor start listening for the trigger event.

4.11.2 Listen for capture

The function below is used to instruct the Library to wait for an External trigger event, which will cause the Device to return a response.

```
void FeasaCom_ExternalTrigger_Listen(const char * DevPath);
```

Parameters:

- DevPath: string containing the system Device path of the BUS master. The same one you used to open the port.

Return value:

Returns an integer value of 1 when the function succeeded, 0 if External trigger is not enabled, or -1 in case that any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

The listening process initiates an internal thread in the SO Library which constantly waits for the trigger event/response. This can be stopped at any time using the function FeasaCom_ExternalTrigger_Abort(); this is also automatically aborted when any other command is sent to the Feasa Device through the Library.

The user has to constantly poll for the event result, through the function FeasaCom_ExternalTrigger_isFinished(), in order to find out whether the external event has been detected or not. Once the process is completed, then the response from the Feasa device can be retrieved using the function FeasaCom_GetResponseByPort();

4.11.3 Check for Trigger event detection

Checks whether the internal thread of the SO Library has detected the response/external trigger and finished receiving the data:

```
void FeasaCom_ExternalTrigger_isFinished(const char * DevPath);
```

Parameters:

- DevPath: string containing the system Device path of the BUS master. The same one you used to open the port.

Return value:

Returns an integer value of 1 when the Trigger event is detected, 0 when it is still listening for the event or has been aborted, and -1 in case that any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

Note: this function takes into account the trigger post-delay, so there is no need to wait for it “manually” after the capture has been acknowledged.

4.11.4 Abort listening

Cancels the internal thread of the SO Library which is constantly waiting for the External trigger event/response:

```
void FeasaCom_ExternalTrigger_Abort(const char * DevPath);
```

Parameters:

- DevPath: string containing the system Device path of the BUS master. The same one you used to open the port.

Return value:

Returns an integer value of 1 when the function succeeded, or -1 in case that any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

4.11.5 Disable trigger

In order to disable the external trigger feature so that the Feasa Device stops listening for trigger events on the trigger port, the following function can be used:

```
void FeasaCom_ExternalTrigger_Disable(const char * DevPath, const int
                                      toFlash);
```

Parameters:

- DevPath: string containing the system Device path of the BUS master. The same one you used to open the port.
- toFlash: numeric value to indicate whether the external trigger status will be saved to RAM (0) or to flash (1). Use RAM if you are constantly using this version or you can eventually damage the flash memory.

Return value:

Returns an integer value of 1 when the function succeeded, or -1 in case that any error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

4.12 Errors

4.12.1 Get Error description

If any error occurs is possible to retrieve a description of it using the following command:

```
void FeasaCom_GetError_Description(char * ErrorDescription);
```

Parameters:

- ErrorDescription: string containing the description of the error. Please, remember, if necessary to allocate enough memory in the destination variable.

Return value:

Does not return any value.

Note 1: this function always return the description of the last error occurred, despite the last command did not generate any error so check the response the last command to ensure that no error occurred.

Note 2: In case of using multiple ports simultaneously, the content of errors description could be overwritten if multiple errors occur, so in this situation the usage of the functions described below is recommended.

4.12.2 Get Error description by Port

If any error occurs in a specific port number, is possible to retrieve a description of it using the following command:

```
void FeasaCom_GetError_DescriptionByPort(const int CommPort, char *
                        ErrorDescription);
```

Parameters:

- CommPort: integer value of the port number to check.
- ErrorDescription: string containing the description of the error. Please, remember, if necessary to allocate enough memory in the destination variable.

Return value:

Does not return any value.

4.12.3 Get Error description by Serial Number

If any error occurs in a specific port number, is possible to retrieve a description of it using the following command:

```
void FeasaCom_GetError_DescriptionBySN(const char * SerialNumber, char *
                          ErrorDescription);
```

Parameters:

- SerialNumber: string with the Serial Number of the port number to check.
- ErrorDescription: string containing the description of the error. Please, remember, if necessary to allocate enough memory in the destination variable.

Return value:

Does not return any value.

4.13 Binning

In order to help the user to classify and identify LEDs according to the binning data provided by the manufacturer of those LEDs, a function is provided.

The following function loads a VEC file, which contains the binning data, this is the set of coordinates describing the position of each bin, and then finds out if the passed color coordinates fall in any of those bins.

```
int FeasaCom_Binning_GetBinFromVECFile(const char * Path, const float x,
                  const float y, char * ResultBinName);
```

Parameters:

- Path: string containing the path to the Vector file (*.vec) containing the coordinates of the bins. The format of that file will be explained in the following section.
- x: chromaticity coordinate for either CIE1931 or CIE1976 systems.
- y: chromaticity coordinate for either CIE1931 or CIE1976 systems.
- ResultBinName: used as a return parameter. Contains the name of the bin to which those coordinates belong to.

Return value:

Returns an integer value indicating whether the bin was found (1) or not (0). The value -1 is returned in case any error occurred: file could not be opened, error in format, missing data…

4.13.1 Vector files

Vector files are ASCII files with extension "vec" (*.vec), and contain a set of coordinates that define polygons. These polygons are normally Bins of classified LEDs.

Each polygon is set in a different line following the format:
`[POLYGON_NAME]=[COORDINATE_0];[COORDINATE_1];[COORDINATE_2]...`

The field POLYGON_NAME is the name given to that polygon or Bin. Its coordinates are a set of x-y pairs separated by the semicolon character ";", and each pair has the format "[x_value], [y_value]".

Let's see an example:
TRIANGLE=0.2,0.2;0.6,0.3;0.2,0.6

Here, we define a Bin named "TRIANGLE" which coordinates are:
x_0=0.2
y_0=0.2
x_1=0.6
y_1=0.3
x_2=0.2
y_2=0.6

Note: *the decimal symbol should be a period '.'*

Let's see another example:
Here we set the bins as defined in "ANSI NEMA ANSLG C78 377 2008"

```
Q2700=0.4813,0.4319;0.4562,0.4260;0.4373,0.3893;0.4593,0.3944
Q3000=0.4562,0.4260;0.4299,0.4165;0.4147,0.3814;0.4373,0.3893
Q3500=0.4299,0.4165;0.3996,0.4015;0.3889,0.3690;0.4147,0.3814
Q4000=0.4006,0.4044;0.3736,0.3874;0.3670,0.3578;0.3898,0.3716
Q4500=0.3736,0.3874;0.3548,0.3736;0.3512,0.3465;0.3670,0.3578
Q5000=0.3551,0.3760;0.3376,0.3616;0.3366,0.3369;0.3515,0.3487
Q5700=0.3376,0.3616;0.3207,0.3462;0.3222,0.3243;0.3366,0.3369
Q6500=0.3205,0.3481;0.3028,0.3304;0.3068,0.3113;0.3221,0.3261
```

4.14 Projects: loading ports from a file.

When working in real-life projects, flexibility is needed, since ports can change and devices could be added or removed, in order to adapt the test setup to the evolving needs of the project.

For this target, Feasa provides a new set of functions in the SO Library that allows to define the ports using a file, creating a mapping between ports and known identifiers, avoiding the need to hard-code the port numbers in the code, allowing technicians to replace a Feasa device in case of failure, no matter what the port or Serial Number of that device is.

These configuration files are read using the function `FeasaCom_OpenProject()`, which will open all ports involved, but detailed information will be given in following sections.

Each new device defined in the file has to be enclosed in a structure defined between the opening tag “[DEVICE]” and the closing one “[END]”.

Inside this tags, several variables can be used to configure that device:

- port: number or string defining the port. Ex: COM12 is equivalent to 12.
- role: (optional) defines whether the device is configured as “DAISY-CHAIN” or “MASTER”. In a daisy-chain configuration, the device connected to the PC, using a serial or USB port will have the role “MASTER” while all devices attached will have the role “DAISY-CHAIN”.
- baudrate: AUTO or numeric value indicating a standard baud-rate
- id: label that will be used by the program to identify the device, instead of using the port or the serial number.
- SerialNumber: (optional) serial number of the device, which is used in daisy-chain configurations.

Those variable are case-insensitive, so no matter whether they are written in capitals or in small letters, because they will be equally accepted.

Let’s see an example of a project/configuration file:

```
[DEVICE]
path=/dev/ttyUSB0
baudrate=AUTO
id=MY_FIRST_DEVICE
[END]
```

In the example above, a device has been configured in port 12. The variable **role** can be ignored in this device entry. The baud-rate is set as AUTO and the device identifier is “MY_FIRST_DEVICE”.

Let’s see another example for a daisy-chain configuration:

```
[DEVICE]
path=/dev/ttyUSB0
role=MASTER
```

```
baudrate=57600
id=BUS_MASTER
SerialNumber=0001
[END]

[DEVICE]
path=/dev/ttyUSB0
role=DAISY-CHAIN
id=BUS_DEVICE_1
SerialNumber=0002
[END]

[DEVICE]
port=COM1
role=DAISY-CHAIN
id=BUS_DEVICE_2
SerialNumber=0003
[END]
```

In the example above, a device has been configured in port 1. The variable **role** indicates that the first device acts as a bus master, the baud-rate is set to 57600 baud, the Identifier is "BUS_MASTER" and the Serial Number is "0001". The other device is attached to the bus, which is connected to the same port, /dev/ttyUSB0, so the role is "DAISY-CHAIN" and its identifier is "BUS_DEVICE_1". There is a second device attached to the bus, with SN 0003, which uses the very same structure but a different identifier ("BUS_DEVICE_2").

Finally, the library provides of a function to get the port number of a device, given its identifier, `FeasaCom_GetPortByID()`, that will be explained further on.

4.14.1 Open project

The following function loads the device configuration from a given file, so ports and its identifiers can be mapped. All ports will be opened using the configuration provided, so they will be ready to be used when the function finishes.

```
int FeasaCom_OpenProject(const char * path);
```

Parameters:

- path: string containing the path to the configuration file containing the devices to be used and its ports.

Return value:

Returns an integer value indicating whether the project was opened successfully (1) or not (0). The value -1 is returned in case any error occurred: a port could not be opened, a syntax

error is detected, a parameter is missing... (check FeasaCom_GetError functions to retrieve more information about the error).

4.14.2 Close project

The following function close all ports opened by the function FeasaCom_OpenProject() and will clear up all internal information.

```
int FeasaCom_CloseProject();
```

Parameters:

Does not accept parameters.

Return value:

Returns an integer value indicating whether the project was closed successfully (1) or not (0). The value -1 is returned in case any error occurred: a port could not be closed, communication failed… (check FeasaCom_GetError functions to retrieve more information about the error).

4.14.3 Get port by its ID

The following function returns the port to which a device Identifier (ID) is attached.

```
int FeasaCom_GetPortByID(const char * DeviceID);
```

Parameters:

- DeviceID: string containing the Identifier of the device, as given in the configuration file.

Return value:

Returns a positive in teger value indicating the port number of the device matching the given ID. The value -1 is returned in case an identifier could not be found.

4.14.4 Send & receive by ID

A command can be sent to a Feasa device given its COM port number, using the following function:

```
int FeasaCom_SendByID(const char * DeviceID, const char * Command, char *
                      ResponseText);
```

Parameters:

- DeviceID: string containing the Identifier of the device, as given in the configuration file.
- Command: the command to send to the Feasa device. There is no need to send the CR + LF characters at the end, because this function does it automatically.
- ResponseText: this is the response string returned by the Feasa device. Please, remember, if necessary to allocate enough memory in the destination variable. This string does not contains the ending CR+LF or CR+LF+EOT characters returned in the raw response.

Return value:

Returns an integer value of 1 if the command was sent successfully, returns 0 if a Syntax Error or a Timeout was detected and returns -1 for other errors (check FeasaCom_GetError functions to retrieve more information about the error).

This function waits until the end-of-command characters CR + LF or CR + LF + EOT are received or there is a timeout (maximum waiting time without communication) of 3500 ms (3.5 seconds), but you can change this last parameter using the function FeasaCom_SetResponseTimeout, that will be explained later.

5. User calibration

Several commands have been included to allow custom adjustments of LED Analyser features without the need of having to use the Feasa UserCal program. Possible adjustments are xy offsets, Wavelength offsets and Relative and Absolute Intensities.

5.1 Relative intensity adjustment

As a typical application, some users want to balance they measurements so they get the same intensity on each fiber/channel.

Procedure to obtain reference values:

1. Reset current relative intensity adjustments for each fiber. Use function FeasaCom_UserCal_ResetInt()
2. Prepare Golden board/known-good board, for measurement.
3. Perform a measurement of the Golden board, using a certain Capture type and range.
4. Read back relative intensity values and store each individual value measured or calculate average, depending on what adjustment is needed. Keep those values as a reference.

Procedure to adjust intensities to the desired reference value:

1. Prepare the new LED Analyser to be used for the adjustment, if a different one from the reference is used.
2. Perform a Capture using the very same type and range used to obtain the reference values.
3. Use function FeasaCom_UserCal_AdjustIntensity() to adjust each fiber to achieve the desired intensity value. The desired values should be the ones obtained from the reference board; typically, the average intensity.
4. Now, a new Capture + reading, should provide the required measurements, within an error/tolerance.

5.1.1 Reset relative intensity adjustments.

```
int FeasaCom_UserCal_ResetIntensity(const char * DevPath, const int Fiber,
                                    const int toFlash)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: the fiber/channel number in which to reset the parameters
- toFlash: set to 0 to store changes in RAM, 1: to store changes in flash memory.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.1.2 Adjust relative intensities to desired values.

```
int FeasaCom_UserCal_AdjustIntensity(const char * DevPath, const int Fiber,
const int IntensityRef, const int isPWM, const int CaptureRange, const int
                                    toFlash)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters
- IntensityRef: integer representing the desired relative intensity value.
- isPWM: integer. Set to 1 if the capture used to obtain the reference values was PWM. Set to 0 otherwise.
- CaptureRange: integer representing the Capture range used while obtaining the reference parameters. Set to 0 if the Range was auto.
- toFlash: integer. Set to 0 to store changes in RAM, 1: to store changes in flash memory.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.1.3 Retrieve relative intensity gain

It is possible to read back the Gain values being applied in the relative intensity adjustment using the following function:

```
int FeasaCom_UserCal_GetIntensityGain(const char * DevPath, const int Fiber,
                                    int * Gain)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters
- Gain: pointer to integer representing the Gain value read from the Analyser.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.1.4 Store relative intensity gain

In case you wish to change the relative intensity adjustment by applying some Gain values previously saved/acquired, you can use the following function:

```
int FeasaCom_UserCal_SetIntensityGain(const char * DevPath, const int Fiber,
                    const int Gain, const int toFlash)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters
- Gain: integer representing the Gain value to store in the Analyser.
- toFlash: integer. Set to 0 to store changes in RAM, 1: to store changes in flash memory.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.2 Absolute intensity adjustment

In case that absolute intensity measurements are required, instead of relative ones, the LED Analyser can be adjusted to obtain those values using as a reference any device that could provide accurate absolute intensity measurements, like the Feasa LED Spectrometer.

Procedure to obtain reference values:

1. Measure LEDs from a golden board/known-good board using the Feasa LED Spectrometer and keep the results as reference values for further use.

Procedure to adjust intensities to the desired reference value:

1. Reset current absolute intensity adjustments for each fiber of the LED Analyser. Use function FeasaCom_UserCal_ResetAbsInt()
2. Perform a measurement (with the LED Analyser) of the very same golden board, using a certain Capture type and range.
3. Use function FeasaCom_UserCal_AdjustAbsInt() on each fiber, to obtain the desired Absolute Intensity measurement (reference value for that fiber, obtained with the LED Spectrometer).
4. Now, a new Capture + reading, should provide the required measurements, within an error/tolerance.

5.2.1 Reset absolute intensity adjustments.

```
int FeasaCom_UserCal_ResetAbsInt(const char * DevPath, const int Fiber,
                                 const int toFlash)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: the fiber/channel number in which to reset the parameters
- toFlash: set to 0 to store changes in RAM, 1: to store changes in flash memory.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.2.2 Adjust absolute intensities to desired values.

```
int FeasaCom_UserCal_AdjustAbsInt(const char * DevPath, const int Fiber,
                const double AbsIntRef, const int toFlash)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters
- AbsIntRef: floating point number, representing the desired absolute intensity value (reference value).
- toFlash: integer. Set to 0 to store changes in RAM, 1: to store changes in flash memory.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.2.3 Retrieve absolute intensity factor

It is possible to read back the adjustment values being applied in the absolute intensity using the following function:

```
int FeasaCom_UserCal_GetAbsIntFactor(const char * DevPath, const int Fiber,
                                     double * Factor)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters
- Factor: pointer to floating point number representing the adjustment factor applied that will be read from the Analyser.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.2.4 Store absolute intensity factor

In case you wish to change the absolute intensity adjustment by applying some values previously saved/acquired, you can use the following function:

```
int FeasaCom_UserCal_SetAbsIntFactor(const char * DevPath, const int Fiber,
                    const double Factor, const int toFlash)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters
- Factor: floating point number representing the adjustment factor to be applied to the Analyser.
- toFlash: integer. Set to 0 to store changes in RAM, 1: to store changes in flash memory.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.3 xy chromaticity adjustment

The LED Analyser allows to apply a fine adjustment of the xy chromaticity values in the form of offsets, so the measured values can be adjusted according to a reference.

Procedure to obtain reference values:

1. Measure LEDs from a golden board/known-good board using the Feasa LED Spectrometer and keep the results as reference values for further use.

Procedure to adjust xy chromaticity values to the desired reference value:

1. Reset current xy offsets for each fiber of the LED Analyser. Use function FeasaCom_UserCal_ResetxyOffset()
2. Perform a measurement (with the LED Analyser) of the very same golden board, using a certain Capture type and range.
3. Use function FeasaCom_UserCal_AdjustxyOffsets() on each fiber, to obtain the desired xy values, using the reference values previously obtained LED Spectrometer.
4. Now, a new Capture + reading, should provide the required measurements, within an error/tolerance.

5.3.1 Reset xy chromaticity adjustments.

```
int FeasaCom_UserCal_ResetxyOffsets(const char * DevPath, const int Fiber,
                                    const int toFlash)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters
- toFlash: integer. Set to 0 to store changes in RAM, 1: to store changes in flash memory.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.3.2 Adjust xy to desired values.

```
int FeasaCom_UserCal_AdjustxyOffsets(const char * DevPath, const int Fiber,
          const float xRef, const float yRef, const int toFlash)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters.
- xRef: floating point number of the desired chromaticity x component (reference value).
- yRef: floating point number of the desired chromaticity y component (reference value).
- toFlash: integer. Set to 0 to store changes in RAM, 1: to store changes in flash memory.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any error occurred (check FeasaCom_GetError

functions to retrieve more information about the error).

5.3.3 Retrieve xy chromaticity offsets

It is possible to read back the offset values being applied in the xy chromaticity using the following function:

```
int FeasaCom_UserCal_GetxyOffsets(const char * DevPath, const int Fiber,
                    float * xOffset, float * yOffset)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters
- xOffset: pointer to floating point number representing the x offset value to be read from the Analyser.
- yOffset: pointer to floating point number representing the y offset value to be read from the Analyser.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.3.4 Store xy chromaticity offsets

In case you wish to change the xy chromaticity by applying some offsets previously saved/acquired, you can use the following function:

```
int FeasaCom_UserCal_SetxyOffsets(const char * DevPath, const int Fiber,
      const float xOffset, const float yOffset, const int toFlash)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters
- xOffset: floating point number representing the x offset value to be applied to the Analyser.
- yOffset: floating point number representing the y offset value to be applied to the Analyser.
- toFlash: integer. Set to 0 to store changes in RAM, 1: to store changes in flash memory.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.4 Wavelength adjustment

The LED Analyser allows to apply a fine adjustment of the Wavelength values in the form of offsets, so the measured values can be adjusted according to a reference.

Procedure to obtain reference values:

1. Measure LEDs from a golden board/known-good board using the Feasa LED Spectrometer

and keep the results as reference values for further use.

Procedure to adjust Wavelength values to the desired reference value:

1. Reset current Wavelength offsets for each fiber of the LED Analyser. Use function FeasaCom_UserCal_ResetWavelengthOffset()
2. Perform a measurement (with the LED Analyser) of the very same golden board, using a certain Capture type and range.
3. Use function FeasaCom_UserCal_AdjustWavelengthOffset() on each fiber, to adjust the offset of that fiber and obtain the desired Wavelength values, using the reference values previously obtained LED Spectrometer.
4. Now, a new Capture + reading, should provide the required measurements, within an error/tolerance.

5.4.1 Reset Wavelength adjustments.

```
int FeasaCom_UserCal_ResetWavelengthOffset(const char * DevPath, const int
                        Fiber, const int toFlash)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters
- toFlash: integer. Set to 0 to store changes in RAM, 1: to store changes in flash memory.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.4.2 Adjust Wavelength to desired values.

```
int FeasaCom_UserCal_AdjustWavelengthOffset(const char * DevPath, const int
             Fiber, const int WavelengthRef, const int toFlash)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters.
- WavelengthRef: integer value of the desired wavelength (reference value).
- toFlash: integer. Set to 0 to store changes in RAM, 1: to store changes in flash memory.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.4.3 Retrieve Wavelength offsets

It is possible to read back the offsets being applied to the wavelength values using the following function:

```
int FeasaCom_UserCal_GetWavelengthOffset(const char * DevPath, const int
```

```
Fiber, int * WavelengthOffset)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters
- WavelengthOffset: pointer to integer representing the offset value to be read from the Analyser.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.4.4 Store Wavelength offsets

In case you wish to change the wavelength values by applying some offsets previously saved/acquired, you can use the following function:

```
int FeasaCom_UserCal_SetWavelengthOffset(const char * DevPath, const int
            Fiber, float WavelengthOffset, const int toFlash)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters
- WavelengthOffset: integer representing the offset value to be applied to the Analyser.
- toFlash: integer. Set to 0 to store changes in RAM, 1: to store changes in flash memory.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any error other occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.5 RGB adjustment

The LED Analyser can be adjusted for testing color and intensity of RGB LEDs, specially, when colors are mixed and PWM modulation is applied. Measured values are adjusted for RGB LEDs, using data of CIE1931 xy chromaticity and absolute intensity, according to a reference values, obtained with a LED Spectrometer. Other parameters (CCT, u'v', Wavelength, etc.) will be affected by this measurement. So if you are planning to adjust them, this has to be done after the RGB adjustment.

Procedure to obtain reference values:

1. Measure LEDs from a golden board/known-good board, when only the RED LED is switched on, using the Feasa LED Spectrometer and keep the results as reference values for further use. Use PWM measurements if LEDs are PWM-modulated.
2. Measure LEDs from a golden board/known-good board, when only the GREEN LED is switched on, using the Feasa LED Spectrometer and keep the results as reference values for further use. Use PWM measurements if LEDs are PWM-modulated.

3. Measure LEDs from a golden board/known-good board, when only the BLUE LED is switched on, using the Feasa LED Spectrometer and keep the results as reference values for further use. Use PWM measurements if LEDs are PWM-modulated.

Note: the ideal situation would be to measure, approximately, at the same time-stamp while taking the reference measurements and while adjusting the values in the LED Analyser. Ex: if LED was measured with the spectrometer, 3200ms after powering it on, then the capture with the LED Analyser, should also be taken at 3200ms after power-on.

Procedure to adjust xy and Absolute Intensity values to the desired reference value:

1. Reset current RGB adjustment each fiber of the LED Analyser. Use function FeasaCom_UserCal_ResetRGBAdj()
2. Perform a measurement (with the LED Analyser) of the very same golden board, using a certain Capture type and range (Note that PWM modulation is normally used for these kind of LEDs).
3. Use function FeasaCom_UserCal_TakeRGBCurrentValues() on each fiber, to store current values measured with the LED Anayser.
4. Use function FeasaCom_UserCal_AdjustRGB() on each fiber to preform the RGB adjustment for that fiber, passing through the parameters, the reference values for x, y and absolute intensity, for all Red, Green and Blue LEDs, previously obtained with the LED Spectrometer.

Now, a new Capture + reading, should provide the required measurements, within an error/tolerance.

5.5.1 Reset RGB adjustments.

```
int FeasaCom_UserCal_ResetRGBAdj(const char * DevPath, const int Fiber)
```

Parameters:

- `DevPath`: string containing the system path of the device.
- `Fiber`: integer representing the fiber/channel number in which to reset the parameters

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.5.2 Take current RGB values.

```
int FeasaCom_UserCal_TakeRGBCurrentValues(const char * DevPath, const int
                        Fiber, const char Color)
```

Parameters:

- DevPath: string containing the system path of the device.
- Fiber: integer representing the fiber/channel number in which to reset the parameters.
- Color: character giving information on the color tested, being this value ‘R’, ‘G’ or ‘B’.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

5.5.3 Adjust RGB.

```
int FeasaCom_UserCal_AdjustRGB(const char * DevPath, const int Fiber, const
 float xRefRed, const float yRefRed, const double AbsIntRefRed, const float
 xRefGreen, const float yRefGreen, const double AbsIntRefGreen, const float
        xRefBlue, const float yRefBlue, const double AbsIntRefBlue)
```

Parameters:

- DevPath: string containing the system path of the device..
- Fiber: integer representing the fiber/channel number in which to reset the parameters.
- xRefRed: floating point number for the reference x coordinate of the Red LED, obtained with the LED Spectrometer.
- yRefRed: floating point number for the reference y coordinate of the Red LED, obtained with the LED Spectrometer.
- AbsIntRefRed: floating point number for the reference absolute intensity of the Red LED, obtained with the LED Spectrometer.
- xRefGreen: floating point number for the reference x coordinate of the Green LED, obtained with the LED Spectrometer.
- yRefGreen: floating point number for the reference y coordinate of the Green LED, obtained with the LED Spectrometer.
- AbsIntRefGreen: floating point number for the reference absolute intensity of the Green LED, obtained with the LED Spectrometer.
- xRefBlue: floating point number for the reference x coordinate of the Blue LED, obtained with the LED Spectrometer.
- yRefBlue: floating point number for the reference y coordinate of the Blue LED, obtained with the LED Spectrometer.
- AbsIntRefBlue: floating point number for the reference absolute intensity of the Blue LED, obtained with the LED Spectrometer.

Return value:

Returns an integer value indicating whether the action was performed successfully (1) or not (0). The value -1 is returned in case any other error occurred (check FeasaCom_GetError functions to retrieve more information about the error).

6. Examples

You can find several programming examples demonstrating how to use the Feasa Communications SO Library on the CD. These examples are written on different languages like C#, Pascal, Python or C/C++ and they can be found in separate folders, depending on the level of difficulty of each exercise, from 0_ to x_, where 0_ is the simplest example.

Note: not all the examples and programming languages are available on all the CD's, so please check your device's CD to find out more.

Function Index

Annex I: Type equivalences.

Here you will find a table variable types and their equivalences between different programming languages.

Type	C	C#	Python	Delphi
Integer	`const int`	`int`	`ctypes.c_int`	`Integer`
Pointer to Integer	`int *`	`ref int`	`ctypes.POINTER(ctypes.c_int)`	`PInteger`
Integer array	`const int *`	`int[]`	`ctypes.POINTER(ctypes.c_int)`	`PInteger`
Pointer to Integer array	`int *`		`ctypes.POINTER(ctypes.c_int)`	`PInteger`
Single decimal	`const float`	`float`	`ctypes.c_float`	`Single`
Pointer to single decimal	`float *`	`ref float`	`ctypes.POINTER(ctypes.c_float)`	`PSingle`
Single decimal array	`const float *`	`float[]`	`ctypes.POINTER(ctypes.c_float)`	`PSingle`
Pointer to single decimal array	`float *`	`float[]`	`ctypes.POINTER(ctypes.c_float)`	`PSingle`
Double decimal	`const double`	`double`	`ctypes.c_double`	`Double`
Pointer to double decimal	`double *`	`ref double`	`ctypes.POINTER(ctypes.c_double)`	`PDouble`
Double decimal array	`const double *`	`double[]`	`ctypes.POINTER(ctypes.c_double)`	`PDouble`
Pointer to double decimal array	`double *`	`double[]`	`ctypes.POINTER(ctypes.c_double)`	`PDouble`
String	`const char *`	`string`	`ctypes.c_char_p`	`AnsiString`
Pointer to string	`char *`	`StringBuilder`	`ctypes.c_char_p`	`PAnsiChar`
Pointer to array of strings	`char **`	`(1)`	`ctypes.POINTER(ctypes.c_char_p)`	`PPAnsiChar`

(1) → [In, MarshalAs(UnmanagedType.LPArray, ArraySubType = UnmanagedType.LPStr)] string[]