

FR-API Reference

v1.11

ThetaMetrisis © 2019-2023

FR-API is the Application Programming Interface (API) for configuring and controlling ThetaMetrisis' FR devices.

Contents

1 Introduction

2 Overview

3 Error Handling

4 API

4.1 Top Level

4.1.1 frInitEx()

4.1.2 frInit()

4.1.3 frShutdown()

4.1.4 frGetAPIVersion()

4.2 Devices

4.2.1 frGetNumDevices()

4.2.2 frConnectRemoteDevice()

4.2.3 frCreateVirtualDevice()

4.2.4 frDestroyVirtualDevice()

4.2.5 frdevGet()/frdevSet()

4.2.6 frdevGetPeripheral()

4.2.7 frdevFetchSpectrum()

4.2.8 frdevSpectrumGetWavelengths()

4.2.9 frdevSpectrumGetSamples()

4.2.10 frdevSpectrumGetNumSamples()

4.2.11 frdevSpectrumGetFlags()

4.2.12 frdevSpectrumGetTimestamp()

4.3 Layer Stacks

4.3.1 frlsCreate()

- 4.3.2 frlsClone()
- 4.3.3 frlsDestroy()
- 4.3.4 frlsGetNumLayers()
- 4.3.5 frlsGetName()
- 4.3.6 frlsSwapLayers()
- 4.3.7 frlsAddLayerAfter()
- 4.3.8 frlsRemoveLayer()
- 4.3.9 frlsLayerSetThicknessConst()
- 4.3.10 frlsLayerSetThicknessRange()
- 4.3.11 frlsLayerSetRefrIndex()
- 4.3.12 frlsLayerSetRefrIndexFromMaterial()
- 4.3.13 frlsLayerIsThicknessConst()
- 4.3.14 frlsLayerGetThickness()
- 4.3.15 frlsLayerGetThicknessRange()
- 4.3.16 frlsLayerGetMaterial()
- 4.3.17 frlsLayerGetRefrIndexFitParamMask()
- 4.3.18 frlsLayerGetRefrIndexType()
- 4.3.19 frlsLayerGetRefrIndexParams()
- 4.3.20 frlsLayerSetType()
- 4.3.21 frlsLayerGetType()

4.4 Materials

- 4.4.1 frOpenUserMaterialDatabase()
- 4.4.2 frCloseUserMaterialDatabase()
- 4.4.3 frmCreate()
- 4.4.4 frmDelete()
- 4.4.5 frmUpdate()
- 4.4.6 frmGetName()
- 4.4.7 frmGetUri()
- 4.4.8 frmFindByUri()
- 4.4.9 frmCreate()
- 4.4.10 frmDelete()
- 4.4.11 frmUpdate()
- 4.4.12 frmGetName()
- 4.4.13 frmGetCategory()
- 4.4.14 frmGetDefaultRefrIndexType()
- 4.4.15 frmGetAvailableRefrIndexTypes()
- 4.4.16 frmGetUri()
- 4.4.17 frmFindByUri()
- 4.4.18 frmCreateRefrIndex()
- 4.4.19 frmDeleteRefrIndex()

- 4.4.20 frmUpdateRefrIndex()
- 4.4.21 frmGetRefrIndex()
- 4.4.22 frmHasRefrIndexType()
- 4.4.23 frmCalcEffectiveRefrIndex()
- 4.4.24 frmEnumBegin()
- 4.4.25 frmEnumEnd()
- 4.4.26 frmEnumReset()
- 4.4.27 frmEnumNext()
- 4.4.28 frmEnumGetCount()
- 4.4.29 frmMaterialsEnumBegin()
- 4.4.30 frmMaterialsEnumEnd()
- 4.4.31 frmMaterialsEnumReset()
- 4.4.32 frmMaterialsEnumNext()
- 4.4.33 frmMaterialsEnumGetCount()
- 4.4.34 frmSearchBegin()
- 4.4.35 frmSearchEnd()
- 4.4.36 frmSearchReset()
- 4.4.37 frmSearchNext()
- 4.4.38 frmSearchGetCount()

4.5 Fitting

- 4.5.1 frCreateFittingContextEx()
- 4.5.2 frCreateFittingContext()
- 4.5.3 frDestroyFittingContext()
- 4.5.4 frfitExecute()
- 4.5.5 frfitGetLayerStack()
- 4.5.6 frfitGetScale()
- 4.5.7 frfitGetOffset()
- 4.5.8 frfitGetAngleOfIncidence()
- 4.5.9 frfitCalcTheoreticalSpectrum()
- 4.5.10 frfitGetFrequencySpectrum()

4.6 Color Analysis

- 4.6.1 frCreateColorAnalysisContext()
- 4.6.2 frDestroyColorAnalysisContext()
- 4.6.3 frcolAnalyzeSpectrum()

4.7 FFT

- 4.7.1 frCreateFFTContext()
- 4.7.2 frDestroyFFTContext()
- 4.7.3 frfftAnalyzeSpectrum()
- 4.7.4 frfftGetFrequencySpectrum()
- 4.7.5 frfftGetPeaks()

- 4.7.6 frfftGetNumPeaks()
- 4.8 Peripherals
 - 4.8.1 frperGet()/frperSet()
 - 4.8.2 frperSendCommand()
- 4.9 Utilities
 - 4.9.1 frCreateExperimentWriter()
 - 4.9.2 frDestroyExperimentWriter()
 - 4.9.3 frexpWriteSpectrum()
 - 4.9.4 frutilNormalizeSpectrum()
 - 4.9.5 frutilCalibrateReferenceSpectrum()
 - 4.9.6 frutilCalcRefrIndex()
 - 4.9.7 frutilCalcEffectiveRefrIndex()

1 Introduction

FR-API currently supports the following platforms:

- Windows x64 (distributed as a DLL)

FR-API uses C++ internally and the publicly exposed functions are written in C. The API user (**developer**) can create wrapper functions in his/her language of choice in order to call FR-API functions. Use of complex C structures as function arguments has been kept to the minimum. Most API function arguments are POD types or pointers to POD types.

ThetaMetrisis currently provides wrappers for the following programming languages:

- C/C++: header (.h) + library (.lib) files



FR-API has been developed using Visual C++ 2022. The corresponding runtime library is expected to be installed on the machine running an application which uses the FR-API DLL.



FR-API is **not** a device driver. Depending on the spectrometer model and/or the peripherals used by your ThetaMetrisis' FR tool and the license file used to initialize the API, the related drivers are expected to be already installed on the end user's **and** the developer's PC.

Contact ThetaMetrisis to learn which C runtime library and drivers are needed for your case.

2 Overview

FR-API functions are divided into 7 modules.

1. Top Level
2. Devices
3. Layer Stacks
4. Materials
5. Fitting
6. Color Analysis
7. FFT
8. Peripherals
9. Utilities

The use of those modules depends on the license file used to initialize the API.

The high level view of the interconnection between the various API modules is shown in the following figure.

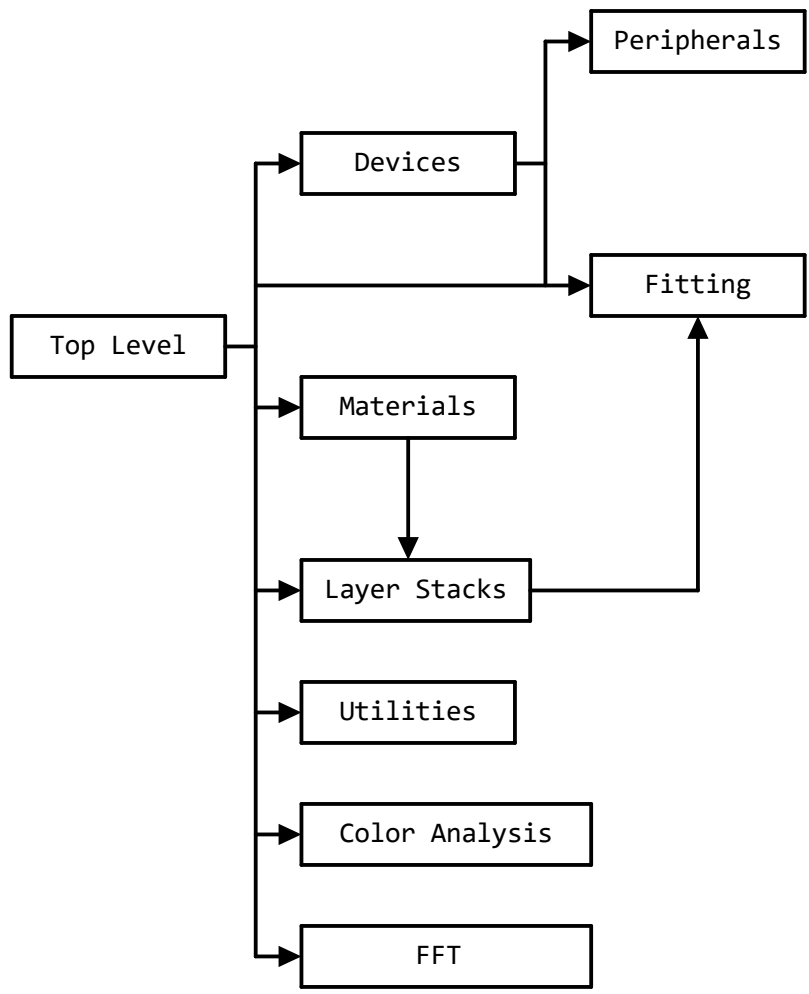


Figure 1: *Interconnection of FR-API modules*

3 Error Handling


Errors are reported back to the API user using either:

- Error codes
- Invalid handles
- Default/invalid return values

All functions which return handles to internal objects will return an invalid handle on failure. All handles are 32-bit unsigned integers wrapped in simple C structs for type-safety. An invalid handle can be checked using the `FR_HANDLE_IS_VALID(handle)` macro.

All functions which return pointers to internal objects will return the `NULL` pointer on failure.

The rest of the functions either return an error code or a default/invalid value. Error codes are 32-bit signed integers and they are all negative or zero (zero meaning “No error”).

 For example, for a `layerStack` with 4 layers, calling `fr1sLayerSetThicknessConst(layerStack, 10, 1000.0)` will return `FR_ERROR_INVALID_ARG` indicating that, at least, one of the function arguments was invalid (the layer index in this case).

The table below shows the error codes returned by the API and their meaning.


Error	Value	Meaning
FR_ERROR_NONE	0	No error. Operation was successful.
FR_ERROR_ALREADY_INITIALIZED	-1	Internal object already initialized.
FR_ERROR_DRIVER_INIT	-2	Error initializing a required device driver.
FR_ERROR_OUT_OF_MEMORY	-3	Memory allocation failed.
FR_ERROR_INVALID_ARG	-4	One or more of the function arguments was invalid.
FR_ERROR_UNSUPPORTED_OPERATION	-5	The requested operation isn't supported.
FR_ERROR_FILE_NOT_FOUND	-6	The path to the specified file wasn't found or there are not enough permissions to open the file.
FR_ERROR_CONNECTION_FAILED	-7	A remote connection failed.
FR_ERROR_INVALID_FORMAT	-8	A specified file was not in the format expected by the API.
FR_ERROR_OPERATION_FAILED	-9	A valid/supported operation failed.
FR_ERROR_UNKNOWN	-1000	Unknown error.

Error codes

4 API

4.1 Top Level

The Top Level module includes functions for the initialization and deinitialization of the API.

 Successful API initialization is mandatory before calling any other API function. If initialization fails, calling any other API function will fail with an `FR_ERROR_UNSUPPORTED_OPERATION` error. In some rare cases, if API initialization fails then the user's application might crash.

4.1.1 frInitEx()

```
int32_t frInitEx(const APIInitConfig* cfg);
```

Initializes the API using the specified configuration.

Parameters:

- `cfg`: See Remarks section for details.

Return value:

If the function succeeds, the return value will be `FR_ERROR_NONE`.

If the function fails, the return value can be one of the following (see Remarks section for more details on each error code):

- `FR_ERROR_ALREADY_INITIALIZED`: [frInitEx\(\)](#) has already been called before calling [frShutdown\(\)](#).
- `FR_ERROR_OUT_OF_MEMORY`: Failed to allocate memory.
- `FR_ERROR_FILE_NOT_FOUND`: One of the specified files wasn't found.
- `FR_ERROR_INVALID_ARG`: At least one of the function arguments was invalid.
- `FR_ERROR_DRIVER_INIT`: Failed to initialize a device driver.

Remarks:

The `APIInitConfig` struct is shown in the snippet below along with the meaning of each field.

```

struct APIInitConfig
{
    // C string with the absolute path of the license file (provided by ThetaMetrisis).
    const char* m_LicensePath;

    // C string with the absolute path of the material database file (provided by ThetaMetrisis).
    // If this parameter is `NULL`, no material database will be initialized and only user
    // materials will be available. See the Materials section for more details.
    const char* m_ThetaMaterialDBPath;

    // C string with the absolute path of the user's material database file. If this parameter is
    // `NULL`, the user's material database will be an in-memory database, until
    // `frOpenUserMaterialDatabase()` function is called. See the Materials section for more
    // details.
    const char* m_UserMaterialDBPath;

    // C string with the absolute path of an optional log file for logging API messages. If this
    // parameter is `NULL`, no log file will be opened (even if m_LogFlags include
    FR_LOG_TYPE_FILE).
    const char* m_LogFilePath;

    // Flags indicating the logging usage (see below).
    uint32_t m_LogFlags;
};


```

All members of the `APIInitConfig` struct are optional except from the license file path. Make sure to properly initialize (NULL or 0) all optional fields you don't need.

`m_LogFlags` is a bitfield of flags indicating the usage of logging functionality. The macro `FR_LOG_FLAGS_MAX_LEVEL(x)` indicates the maximum level of message logging. Potential values are shown in the table below

Log Level	Value	Meaning
FR_LOG_LEVEL_ERROR	0	Only errors will be written to the log file.
FR_LOG_LEVEL_WARNING	1	Only warnings and errors will be written to the log file.
FR_LOG_LEVEL_INFO	2	Only info messages, warnings and errors will be written to the log file.
FR_LOG_LEVEL_DEBUG	3	All the above messages including debug messages will be written to the log file.
FR_LOG_LEVEL_TRACE	4	Everything will be logged, including a detailed order of API function calls.

Log Levels

 Enabling `FR_LOG_LEVEL_TRACE` will produce really large log files. It is advised to keep this flag disabled unless you are sure API usage will be short-termed and/or you actually need a detailed report of every API function call your application performs.

The macro `FR_LOG_FLAGS_TYPE(x)` indicates the type of logging mechanism to use. Currently only file-based logging is supported. You have to include `FR_LOG_FLAGS_TYPE(FR_LOG_TYPE_FILE)` in combination with a valid `m_LogFilePath` file path in order for file logging to work correctly. By using `FR_LOG_FLAGS_TYPE(FR_LOG_TYPE_NONE)` the file specified in `m_LogFilePath` is ignored.

The code snippet below shows the way to properly enable the file-based logging mechanism, where only warnings and errors will be reported.

```
APIInitConfig cfg;
memset(&cfg, 0, sizeof(APIInitConfig)); // Zero-initialize all optional members.
cfg.m_LicensePath = "Fill in with the absolute path to your license file.";
cfg.m_LogFilePath = "Fill in with the absolute path to the log file.";
cfg.m_LogFlags = 0
    | FR_LOG_FLAGS_MAX_LEVEL(FR_LOG_LEVEL_WARNING)
    | FR_LOG_FLAGS_TYPE(FR_LOG_TYPE_FILE)
;
const int32_t res = frInitEx(&cfg);
```

`FR_ERROR_INVALID_ARG` is a generic error value indicating that one of the function arguments was incorrect. In this case `FR_ERROR_INVALID_ARG` usually means that the API tried to initialize an internal module but failed, probably due to an invalid license.

FR_ERROR_FILE_NOT_FOUND indicates that either the specified license file or the material database wasn't found, FR-API doesn't have access to it or the file was corrupted. Check the related paths where the files are stored and make sure FR-API has read access to the specified files.

FR_ERROR_OUT_OF_MEMORY can occur if the license file is corrupted. The license file also serves as the configuration file for all supported FR tools. If a device configuration is not valid (due to a corrupt license file), FR_ERROR_OUT_OF_MEMORY can occur.

FR_ERROR_DRIVER_INIT means that at least one of the required device drivers wasn't found on the system.



The necessary device drivers are determined by the license file. Drivers are dynamically loaded at runtime.

4.1.2 frInit()

```
int32_t frInit(const char* licenseFile, const char* materialDB);
```

Shortcut function for initializing the API.

Parameters:

- `licenseFile`: C string with the absolute path of the license file (provided by ThetaMetrisis).
- `materialDB`: C string with the absolute path of the material database file (provided by ThetaMetrisis). If this parameter is NULL, no material database will be initialized and only user materials will be available. See the [Materials](#) section for more details.

Return value:

See [frInitEx\(\)](#) function for details.

Remarks:

This function is a shortcut to the [frInitEx\(\)](#) function. Below is a code snippet showing the code of this function.

```
int32_t frInit(const char* licenseFile, const char* thetaMatDB)
{
    APIInitConfig cfg;
    memset(&cfg, 0, sizeof(APIInitConfig));
    cfg.m_LicensePath = licenseFile;
    cfg.m_ThetaMaterialDBPath = thetaMatDB;
    return frInitEx(&cfg);
}
```

For more details see [frInitEx\(\)](#) function.

4.1.3 frShutdown()

```
void frShutdown();
```

Deinitializes the API.

Parameters:

None.

Return value:

None.

Remarks:

This function deallocates all internally allocated memory and unloads all loaded device drivers. After the function returns, all handles previously obtained via the API are invalid. Function [frInit\(\)](#) should be called again to re-initialize the API.

4.1.4 frGetAPIVersion()

```
void frGetAPIVersion(char* str, uint32_t maxLen);
```

Get the API version

Parameters:

- `str`: Buffer to hold the version string.
- `maxLen`: The maximum number of elements the `str` buffer can hold.

Return value:

None.

Remarks:

The version string has the following format:

FR_API_VERSION_MAJOR.FR_API_VERSION_MINOR.FR_API_VERSION_PATCH.FR_API_VERSION_BUILD

4.2 Devices

The Device module includes functions for the configuration of the connected FR tools, as well as functions for capturing new spectra.

4.2.1 frGetNumDevices()

```
uint32_t frGetNumDevices();
```

Get the number of currently connected and initialized devices.

Return value:

If the function succeeds the return value is the number of FR tools that are successfully initialized and ready to be used.

If the function fails the return value will be `UINT32_MAX`.

Remarks:

Use `frdevXXX()` functions to manage the device configuration and fetch spectra.



Calling `frdevXXX` functions concurrently from multiple threads is not advised. Even though the actual device driver will probably limit access to a device to a single thread at a time, the API doesn't currently include any guards against such multi-threaded use.

4.2.2 frConnectRemoteDevice()

```
uint32_t frConnectRemoteDevice(const char* ipAddr, uint32_t spectrometerType);
```

Connect to a remote device.

Parameters:

- `ipAddr`: The IP address of the remote device.
- `spectrometerType`: The type of the device you are trying to connect to (see Remarks section for details)

Return value:

If the function succeeds, the return value is the ID of the new device.

If the function fails the return value will be `UINT32_MAX`

Remarks:

The parameter `spectrometerType` indicates what kind of device you are trying to connect to. This is required because it determines which communication protocol will be used to talk to the remote device.

Currently only the following Ocean Insights' spectrometers are supported:

- JAZ (`FR_SPECTYPE_OCEAN_JAZ_NETWORK`),
- OceanFX (`FR_SPECTYPE_OCEAN_FX`)
- OceanHDX (`FR_SPECTYPE_OCEAN_HDX`)

4.2.3 frCreateVirtualDevice()

```
uint32_t frCreateVirtualDevice(const char* spectrumFilePath);
```

Creates a virtual device without any peripherals attached, using the specified spectrum file (ThetaMetrisis' .ex2 files).



The API currently supports only 1 virtual device.

Parameters:

- spectrumFilePath: C string with the absolute path to the spectrum file to use.

Return value:

If the function succeeds, the return value is the ID of the virtual device. This ID can be used with frdevXXX() functions to query and configure the device.

If the function fails the return value will be `UINT32_MAX`.

The function can fail under the following conditions:

- A virtual device is already initialized.
- The specified spectrum file is invalid or the API doesn't have read access to it.
- A memory allocation fails.

Remarks:

Virtual devices can be used for testing the API in case the developer doesn't have a real FR device connected to the PC. Virtual devices count towards the total number of initialized devices (i.e. the value returned by [frGetNumDevices\(\)](#)).

Some limitations of virtual devices:

- Virtual devices are not connected with any peripherals, so calling [frdevGetPeripheral\(\)](#) will always return an invalid handle.
- Virtual devices are always single spectrometer devices. `frdevGetUint(vdevID, FR_DEVICE_NUM_SUBDEVICES)` will always return 1.
- Virtual devices will respond correctly to most `frdevSetXXX()` functions but the configuration has no effect on the spectra fetched using [frdevFetchSpectrum\(\)](#) except in the following cases:
 - `FR_SUBDEV_INTEGRATION_TIME` affects the rate at which each spectrum will be returned. On Windows, the delay is currently implemented using the system's `Sleep()` function. Because of this,

the actual spectrum acquisition rate might not be exactly equal to the specified integration time due to the Windows timer resolution used by `Sleep()` (usually 15ms).

- `FR_SUBDEV_SPECTRA_AVERAGING` is implemented by averaging the next N spectra from the specified `ex2` file. It is implemented as N independent acquisitions, so the total time required to fetch a single spectrum with averaging equal to N is $N * \text{integration time}$.

4.2.4 frDestroyVirtualDevice()

```
void frDestroyVirtualDevice(uint32_t devID);
```

Destroys the specified virtual device.

Parameters:

- devID: The ID of the device previously returned by [frCreateVirtualDevice\(\)](#).

Remarks:

If the specified ID doesn't correspond to a virtual device, this function does nothing. After a call to this function, the device is assumed to be detached from the system and any call to frdevXXX() functions will fail.

4.2.5 frdevGet()/frdevSet()

```
int32_t frdevGetString(uint32_t devID, uint32_t param, char* value, uint32_t maxlen);
int32_t frdevSetString(uint32_t devID, uint32_t param, const char* value);
int32_t frdevGetUint(uint32_t devID, uint32_t param, uint32_t* value);
int32_t frdevSetUint(uint32_t devID, uint32_t param, uint32_t value);
int32_t frdevGetBoolean(uint32_t devID, uint32_t param, bool* value);
int32_t frdevSetBoolean(uint32_t devID, uint32_t param, bool value);
int32_t frdevGetDouble(uint32_t devID, uint32_t param, double* value);
int32_t frdevGetDoublev(uint32_t devID, uint32_t param, double* values, uint32_t n);
int32_t frdevSetDoublev(uint32_t devID, uint32_t param, const double* values, uint32_t n);
```

Parameters:

- devID: The ID of the device
- param: The device parameter (see the Remarks section for details)
- value: The value of the parameter
- maxlen/n: The size of the buffer pointed by the value.

Return value:

If the function succeeds the return value will be FR_ERROR_NONE.

If the function fails, check the returned value for the reason of the failure.

Remarks:

Device parameters are split into two categories:

- Parameters of the device as a whole (FR_DEVICE_XXX)
- Parameters of each of the subdevices connected to the device (FR_SUBDEV_XXX)

Each parameter can only be queried using a specific set of functions. The tables below show the available parameters, their type, indicating which functions can be used to set/get them and their access permissions (read and/or write).


Parameter	Type	Access	Notes
FR_DEVICE_NUM_SUBDEVICES	UInt	Read	Number of sub-devices (spectrometers)
FR_DEVICE_NAME	String	Read	Name of the device
FR_DEVICE_SERIAL	String	Read	Serial number of the device
FR_DEVICE_MAX_INTENSITY	UInt/Double	Read	Maximum sample intensity
FR_DEVICE_NUM_PIXELS	UInt	Read	Total number of pixels per spectrum

Device Parameters

Parameter	Type	Access	Notes
FR_SUBDEV_NAME(i)	String	Read	Name of the sub-device
FR_SUBDEV_SERIAL(i)	String	Read	Serial number of the sub-device
FR_SUBDEV_INTEGRATION_TIME_MIN(i)	UInt/Double	Read	The minimum integration time in ms
FR_SUBDEV_INTEGRATION_TIME_MAX(i)	UInt/Double	Read	The maximum integration time in ms
FR_SUBDEV_INTEGRATION_TIME(i)	UInt/Double	Read/Write	The current integration time in ms
FR_SUBDEV_MAX_INTENSITY(i)	UInt/Double	Read	Maximum sample intensity
FR_SUBDEV_NUM_PIXELS(i)	UInt	Read	Number of pixels
FR_SUBDEV_SPECTRA_AVERAGING(i)	UInt	Read/Write	
FR_SUBDEV_BOXCAR_WIDTH(i)	UInt	Read/Write	
FR_SUBDEV_ELECTRIC_DARK_CORRECTION(i)	UInt/Boolean	Read/Write	
FR_SUBDEV_NON_LINEARITY_CORRECTION(i)	UInt/Boolean	Read/Write	
FR_SUBDEV_STRAY_LIGHT_CORRECTION(i)	UInt/Boolean	Read	
FR_SUBDEV_STROBE(i)	UInt/Boolean	Read/Write	
FR_SUBDEV_WAVELENGTH_COEFS(i)	Double Vec	Read/Write	Wavelength coefficients. See note below.
FR_SUBDEV_NUM_WAVELENGTH_COEFS(i)	UInt	Read	
FR_SUBDEV_IS_VIRTUAL(i)	Boolean	Read	
FR_SUBDEV_VIRTUAL_DATA(i)	String	Write	
FR_SUBDEV_EEPROM_SLOT(i, j)	String	Read/Write	
FR_SUBDEV_FW_VERSION(i)	String	Read	Firmware version of the sub-device

FR_SUBDEV_HAS_ELECTRIC_DARK_CORRECTION	UInt/Boolean	Read	Indicates whether the sub-device supports electric dark correction.
FR_SUBDEV_HAS_NON_LINEARITY_CORRECTION	UInt/Boolean	Read	Indicates whether the sub-device supports non-linearity correction.
FR_SUBDEV_HAS_STROBE	UInt/Boolean	Read	Indicates whether the sub-device supports strobing.
FR_SUBDEV_HAS_HIGH_GAIN_MODE	UInt/Boolean	Read	Indicates whether the sub-device supports high-gain mode.
FR_SUBDEV_HIGH_GAIN_MODE	UInt/Boolean	Read/Write	High gain mode.
FR_SUBDEV_HAS_EEPROM	UInt/Boolean	Read	Indicates whether the sub-device supports accessing its EEPROM.

Sub-device Parameters

 Some spectrometers support changing their wavelength coefficients by calling `frdevSetDoublev()` with the parameter `FR_SUBDEV_WAVELENGTH_COEFS(i)`. Changing the wavelength coefficients of a spectrometer will change the list of wavelengths of each spectrum.

In case of a single spectrometer device, the total number of wavelengths does not change when updating its coefficients, and the wavelength list is updated automatically. I.e. calling `frdevSpectrumGetWavelengths()` after a successful call to `frdevSetDoublev(FR_SUBDEV_WAVELENGTH_COEFS(i))` should return the new list of wavelengths.

In the case of a double spectrometer device, the user should shutdown and reinitialize the API in order to get the new list of wavelengths. This is required because the total number of wavelengths of a double spectrometer device depends on the overlap between the wavelength ranges of the two spectrometers and the overlap will almost always end up producing different number of wavelengths and different weights for the combination of the two spectra.



`frdevSetDoublev(FR_SUBDEV_WAVELENGTH_COEFS(i))` overwrites the previous wavelength coefficients in the internal memory of the spectrometer (EEPROM). If invalid calibration coefficients are written into the EEPROM, the spectrometer will no longer return valid data, so be very careful with the use of this function.

4.2.6 frdevGetPeripheral()

```
PeripheralHandle frdevGetPeripheral(uint32_t devID, uint16_t peripheralType);
```

Get the handle of the specified peripheral from a device.

Parameters:

- `devID`: The ID of the device.
- `peripheralType`: The type of the peripheral. See the Remarks section for details.

Return value:

If the function succeeds, the returned value is a handle to the specified peripheral device.

If the function failed it will return an invalid handle.

Remarks:

FR tools support several peripheral devices, including lamps, shutters, stages, cameras, etc. FR-API might not currently support all available peripherals.

The `peripheralType` parameter should be one of the `FR_PERTYPE_XXX` constants. The table below summarizes all currently supported peripheral types.

Type	Comment
<code>FR_PERTYPE_PRIMARY_LAMP</code>	The primary lamp. If available, this can be <code>FR_PER_LAMP_TYPE_HALOGEN_I2C</code> , <code>FR_PER_LAMP_TYPE_PORTABLE</code> or <code>FR_PER_LAMP_TYPE_HALOGEN_ES</code> .
<code>FR_PERTYPE_SECONDARY_LAMP</code>	The secondary lamp. If available, this can either be <code>FR_PER_LAMP_TYPE_DEUTERIUM_I2C</code> or <code>FR_PER_LAMP_TYPE_DEUTERIUM_ES</code> .
<code>FR_PERTYPE_SHUTTER</code>	Shutter. All shutters have the same set of parameters and commands so there's not a specific shutter type at the moment.

For a complete list of all peripheral parameters and commands check the Peripherals document.

4.2.7 frdevFetchSpectrum()

```
DevSpectrumHandle frdevFetchSpectrum(uint32_t devID);
```

Fetch a new spectrum from the specified device.

Parameters:

- devID: The ID of the device.

Return value:

If the function succeeds the returned value is a handle to the newly acquired spectrum.

If the function fails it will return an invalid handle.

Remarks:

Use frdevSpectrumXXX() functions to get information about the newly acquired spectrum.



In order to keep memory requirements as low as possible, the API allocates spectra from a circular buffer. The application developer is responsible for copying the acquired spectrum data to his/her own memory for use at a later time. Do not use the returned handle as a unique identifier.

If you need a unique identifier for the acquired spectrum, use the spectrum's timestamp returned by `frdevSpectrumGetTimestamp()`.

4.2.8 frdevSpectrumGetWavelengths()

```
int32_t frdevSpectrumGetWavelengths(DevSpectrumHandle handle, double* wavelengths, uint32_t n);
```

Get the wavelength array from the specified spectrum.

Parameters:

- `handle`: The spectrum handle returned by [frdevFetchSpectrum\(\)](#)
- `wavelengths`: Buffer to hold the spectrum's wavelengths
- `n`: The maximum number of elements the `wavelengths` buffer can hold

Return value:

If the function succeeds the returned value will be `FR_ERROR_NONE`.

If the function fails check the returned value for the reason of failure.

Remarks:

The `wavelengths` array should be large enough to hold spectrum's wavelengths. Use [frdevSpectrumGetNumSamples\(\)](#) to determine the total number of elements for this array.

- If the `wavelengths` array is smaller than the required size, only the first `n` wavelengths will be copied.
- If the `wavelengths` array is larger than the required size, only the first [frdevSpectrumGetNumSamples\(\)](#) wavelengths will be copied.

4.2.9 frdevSpectrumGetSamples()

```
int32_t frdevSpectrumGetSamples(DevSpectrumHandle handle, double* samples, uint32_t n);
```

Get the array of samples from the specified spectrum.

Parameters:

- `handle`: The spectrum handle returned by [frdevFetchSpectrum\(\)](#)
- `samples`: Buffer to hold the spectrum's samples
- `n`: The maximum number of elements the `samples` buffer can hold.

Return value:

If the function succeeds the returned value will be `FR_ERROR_NONE`.

If the function fails, check the returned value for the reason of failure.

Remarks:

The `samples` array should be large enough to hold all of the spectrum's samples. Use [frdevSpectrumGetNumSamples\(\)](#) to determine the total number of elements for this array.

- If the `samples` array is smaller than the required size, only the first `n` samples will be copied.
- If the `samples` array is larger than the required size, only the first [frdevSpectrumGetNumSamples\(\)](#) samples will be copied.

4.2.10 frdevSpectrumGetNumSamples()

```
uint32_t frdevSpectrumGetNumSamples(DevSpectrumHandle handle);
```

Get the number of samples of the specified spectrum.

Parameters:

- handle: The spectrum handle returned by [frdevFetchSpectrum\(\)](#)


Return value:

If the function succeeds the returned value is the total number of samples of the spectrum.

If the function fails the returned value will be 0.

Remarks:

All spectra from a specific device are expected to have the same number of samples.

-  Single spectrometer devices (`frdevGetUint(FR_DEVICE_NUM_SUBDEVICES) == 1`) will have spectra with the same amount of samples as the device's number of pixels.
- Double spectrometer devices (`frdevGetUint(FR_DEVICE_NUM_SUBDEVICES) == 2`) might have less samples than the total number of pixels of the two devices (`frdevGetUint(FR_SUBDEV_NUM_PIXELS(i))`). The actual number depends on the overlap of the two spectra.

4.2.11 frdevSpectrumGetFlags()

```
uint32_t frdevSpectrumGetFlags(DevSpectrumHandle handle);
```

Parameters:

- handle: The spectrum handle returned by [frdevFetchSpectrum\(\)](#)

Return value:

The function returns a bitfield with information about the spectrum.

Remarks:

The table below describes the meaning of each flag.

Flag	Meaning
FR_SPECTRUM_FLAG_INVALID	The spectrum is invalid
FR_SPECTRUM_FLAG_SATURATED	The spectrum is saturated
FR_SPECTRUM_FLAG_PARTIAL	The spectrum is partially loaded

Spectrum Flags

4.2.12 frdevSpectrumGetTimestamp()

```
uint64_t frdevSpectrumGetTimestamp(DevSpectrumHandle handle);
```

Get the timestamp of the specified spectrum.

Parameters:

- `handle`: The spectrum handle returned by [frdevFetchSpectrum\(\)](#)

Return value:

A 64-bit unsigned integer representing the timestamp of the spectrum.

Remarks:

On Windows, the timestamp has the same units as the [GetSystemTimePreciseAsFileTime\(\)](#) system function.

4.3 Layer Stacks

Layer stacks are used to describe the structure of the sample under test.

Layer indices are zero-based, with 0 being the top of the stack and the last layer being the substrate.

The first and last layers of a stack are considered semi-infinite by the various algorithms, so their thicknesses aren't taken into account. Only their material/refractive index is used.

4.3.1 frIsCreate()

```
LayerStackHandle frIsCreate(const char* name, uint32_t numLayers);
```

Create a new layer stack.

Parameters:

- name: A name for the layer stack.
- numLayers: The initial number of layers in the new stack.

Return value:

If the function succeeds the return value is the handle of the new stack.

If the function fails, an invalid handle is returned.

Remarks:

name isn't used by the API. It's stored along with the stack and can be used freely by the developer. The internal buffer is limited to 128 characters.

The initial number of layers of a stack must be greater than or equal to 3.

All layers of the new layer stack are initially empty (they have no material/refractive index assigned) and their thicknesses are set to 0 nm.



The developer is responsible for the configuration of the new stack by setting the layers' thicknesses and materials/refractive indices. Using a layer stack without initializing it properly might lead to crashes and/or hard-to-debug bugs.

4.3.2 frIsClone()

```
LayerStackHandle frIsClone(LayerStackHandle layerStack);
```

Clone a layer stack.

Parameters:

- `layerStack`: The handle of the layer stack to clone.

Return value:

If the function succeeds the return value will be the handle of the new layer stack.

If the function fails, an invalid handle is returned.

Remarks:

There is no link between the original and the cloned layer stacks. After a successful call to this function, the developer is free to destroy the original stack.

The new layer stack will have the string "Clone" appended to its name.

4.3.3 frlsDestroy()

```
int32_t frlsDestroy(LayerStackHandle layerStack);
```

Destroy a layer stack.

Parameters:

- `layerStack`: The handle of the layer stack to destroy.


Return value:

If the function succeeds the return value is `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of the failure.

Remarks:

After the function returns successfully the specified layer stack is considered invalid and should not be used again.

 The API is allocating layer stacks from a memory pool. The same handle can be returned again later by `frlsCreate()`. It's the developer's responsibility to make sure not to use a layer stack after calling `frlsDestroy()` on it. Using a layer stack after it has been destroyed might lead to hard-to-debug bugs.

4.3.4 frIsGetNumLayers()

```
uint32_t frIsGetNumLayers(LayerStackHandle layerStack);
```

Get the number of layers of a layer stack.

Parameters:

- `layerStack`: The handle of the layer stack.

Return value:

If the function succeeds the return value is the number of layers of the specified stack.

If the function fails the return value will be `UINT32_MAX`

Remarks:

None.

4.3.5 frlsGetName()

```
int32_t frlsGetName(LayerStackHandle layerStack, char* name, uint32_t maxLen);
```

Get the name of the layer stack.

Parameters:

- `layerStack`: The handle of the layer stack.
- `name`: A character buffer for holding the name of the stack.
- `maxLen`: The maximum number of characters `name` can hold.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE` and `name` will hold the internally stored name of the stack.

If the function fails, check the return value for the reason of the failure.

Remarks:

None.

4.3.6 frIsSwapLayers()

```
int32_t frIsSwapLayers(LayerStackHandle layerStack, uint32_t layerA, uint32_t layerB);
```

Swap two layers of a layer stack.

Parameters:

- `layerStack`: The handle of the layer stack.
- `layerA`: The zero-based index of the first layer.
- `layerB`: The zero-based index of the second layer.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of failure.

Remarks:

None.

4.3.7 frIsAddLayerAfter()

```
int32_t frIsAddLayerAfter(LayerStackHandle layerStack, uint32_t layerID);
```

Insert a new layer after the specified layer.

Parameters:

- `layerStack`: The handle of the layer stack.
- `layerID`: The zero-based index of the layer after which the new layer will be inserted.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of failure.

Remarks:

After a successful call to this function, the new layer is uninitialized. The developer is responsible for setting the thickness and material/refractive index of the new layer to valid values before using the stack.

Using the special value of `UINT32_MAX` for the `layerID` parameter will insert the new layer at the top of the stack.

4.3.8 frIsRemoveLayer()

```
int32_t frIsRemoveLayer(LayerStackHandle layerStack, uint32_t layerID);
```

Remove the specified layer from a layer stack.

Parameters:

- `layerStack`: The handle of the layer stack.
- `layerID`: The zero-based index of the layer to remove.

Return value:

If the function succeeds, the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of failure.

Remarks:

None.

4.3.9 frlsLayerSetThicknessConst()

```
int32_t frlsLayerSetThicknessConst(LayerStackHandle layerStack, uint32_t layerID, double thickness_nm);
```

Set a constant thickness for the specified layer of a stack.

Parameters:

- `layerStack`: The handle of the layer stack.
- `layerID`: The zero-based index of the layer.
- `thickness_nm`: The thickness of the layer in nanometers.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails check the return value for the reason of the failure.

Remarks:

Setting the thickness of a layer to a constant value implies that it won't be calculated during fitting. Use [frlsLayerSetThicknessRange\(\)](#) if you want to calculate the thickness of a layer during fitting.



You cannot set the thickness of the first and last layers of a layer stack. Those two layers are considered semi-infinite. Trying to set the thickness of those two layers will result in an `FR_ERROR_INVALID_ARG` error.

4.3.10 frlsLayerSetThicknessRange()

```
int32_t frlsLayerSetThicknessRange(LayerStackHandle layerStack, uint32_t layerID, double
minThickness_nm, double maxThickness_nm, uint32_t numSteps);
```

Set the thickness of a layer to a range of values.

Parameters:

- layerStack: The handle of the layer stack.
- layerID: The zero-based index of the layer.
- minThickness_nm: The minimum thickness in nanometers.
- maxThickness_nm: The maximum thickness in nanometers.
- numSteps: The number of subdivisions of the specified range.

Return value:

If the function succeeds the return value will be FR_ERROR_NONE.

If the function fails check the return value for the reason of the failure.

Remarks:

Setting the thickness of a layer to a range of values implies that it will be calculated during fitting. Use [frlsLayerSetThicknessConst\(\)](#) if you don't want to calculate the thickness of the layer during fitting.

minThickness_nm must be less than or equal to maxThickness_nm. Otherwise FR_ERROR_INVALID_ARG will be returned.

If numSteps is less than 2, then the actual number of thickness values that will be used during fitting is calculated based on the following algorithm:

```
if minThickness_nm == maxThickness_nm then
    numSteps = 1 // minimum and maximum are the same. Use only the minimum value
else if (maxThickness_nm - minThickness_nm) <= THRESHOLD
    numSteps = 2 // Only the two extremes will be used (minimum and maximum)
else
    numSteps = floor((maxThickness_nm - minThickness_nm) / THRESHOLD) + 1
```

The value of THRESHOLD depends on the real part of the layer's refractive index at the center of the specified wavelength range, based on the formula:

THRESHOLD = 150.0 / n



You cannot set the thickness of the first and last layers of a layer stack. Those two layers are considered semi-infinite. Trying to set the thickness of those two layers will result in an FR_ERROR_INVALID_ARG error.

4.3.11 frlsLayerSetRefrIndex()

```
int32_t frlsLayerSetRefrIndex(LayerStackHandle layerStack, uint32_t layerID, uint32_t refrIndexType, const double* params, uint32_t numParams, uint32_t fitParamsMask);
```

Set the refractive index of a layer to custom values.

Parameters:

- `layerStack`: The handle of the layer stack.
- `layerID`: The zero-based index of the layer.
- `refrIndexType`: The type of the refractive index equation.
- `params`: Parameters of the refractive index equation.
- `numParams`: The number of parameters.
- `fitParamsMask`: Bitfield indicating which of parameters will be calculated during fitting.

Return value:

If the function succeeds, the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of the failure.

Remarks:

The number of required parameters depends on the selected refractive index type. See the [Materials](#) section for details.

`fitParamsMask` is used to indicate which of the provided parameters will be calculated during fitting. Setting the corresponding bit to 1, will inform the fitting algorithm to treat the parameter as a variable and the provided value will be used as the initial guess.

Example: Set the 2nd layer's refractive index to use the Cauchy equation and calculate parameters A and B of the real part of the refractive index.

```
const double params[] = { 1.4, 0.0, 0.0 };
int32_t res = frlsLayerSetRefrIndex(layerStack
    , 1
    , FR_REFRACTIVE_INDEX_CAUCHY
    , params
    , 3
    , FR_FIT_RIPARAM_CAUCHY_NA | FR_FIT_RIPARAM_CAUCHY_NB
);
```

4.3.12 frlsLayerSetRefrIndexFromMaterial()

```
int32_t frlsLayerSetRefrIndexFromMaterial(LayerStackHandle layerStack, uint32_t layerID,
MaterialHandle mat, uint32_t refrIndexType, uint32_t fitParamsMask);
```

Set the refractive index of a layer from a material.

Parameters:

- layerStack: The handle of the layer stack.
- layerID: The zero-based index of the layer.
- mat: The handle of the material
- refrIndexType: The type of the refractive index equation.
- fitParamsMask: Bitfield indicating which of parameters will be calculated during fitting.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails check the return value for the reason of the failure.

Remarks:

The developer must make sure that the specified material includes parameters for the selected refractive index type (see [frmHasRefrIndexType\(\)](#) and [frmGetAvailableRefrIndexTypes\(\)](#) for details).

`fitParamsMask` has the same meaning as in [frlsLayerSetRefrIndex\(\)](#).

Example: Set the 2nd layer's refractive index to a material's raw data.

```
MaterialHandle materialHandle = ...;
int32_t res = frlsLayerSetRefrIndexFromMaterial(layerStack
    , 1
    , materialHandle
    , FR_REFRACTIVE_INDEX_RAW_DATA
    , 0 // Cannot fit raw data parameters
);
```


4.3.13 frlsLayerIsThicknessConst()

```
bool frlsLayerIsThicknessConst(LayerStackHandle layerStack, uint32_t layerID);
```

Is the specified layer's thickness constant or not.

Parameters:

- `layerStack`: The handle of the layer stack.
- `layerID`: The zero-based index of the layer.

Return value:

The function returns false if the specified layer thickness is set to a range of values (see [frlsLayerSetThicknessRange\(\)](#))

The function returns true if the specified layer thickness is set to a constant value (see [frlsLayerSetThicknessConst\(\)](#)) or if there was an error.

Remarks:

None.

4.3.14 frlsLayerGetThickness()

```
double frlsLayerGetThickness(LayerStackHandle layerStack, uint32_t layerID);
```

Get the current thickness of the specified layer.

Parameters:

- `layerStack`: The handle of the layer stack.
- `layerID`: The zero-based index of the layer.

Return value:

If the function succeeds the return value is the current thickness of the specified layer in nanometers.

If the function fails, the returned value will be negative, which is an invalid value for layer thickness.

Remarks:

If the layer has a constant thickness this function returns the value passed to [frlsLayerSetThicknessConst\(\)](#).

If the layer has a range of thicknesses, previously set using [frlsLayerSetThicknessRange\(\)](#), this function returns the last calculated thickness value. If no fitting has been performed on the specified layer stack yet, the layer's thickness will be equal to the minimum of the thickness range.

4.3.15 frlsLayerGetThicknessRange()

```
int32_t frlsLayerGetThicknessRange(LayerStackHandle layerStack, uint32_t layerID, double*
minThickness_nm, double* maxThickness_nm, uint32_t* numSteps);
```

Get the range of thicknesses for the specified layer.

Parameters:

- `layerStack`: The handle of the layer stack.
- `layerID`: The zero-based index of the layer.
- `minThickness_nm`: Pointer to a double for holding the minimum of the layer's thickness range.
- `maxThickness_nm`: Pointer to a double for holding the maximum of the layer's thickness range.
- `numSteps`: Pointer to an unsigned integer for holding the number of layer's thickness range subdivisions.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails check the return value for the reason of the failure.

Remarks:

On successful return, `minThickness_nm`, `maxThickness_nm` and `numSteps` will hold the values passed to the last successful call of [frlsLayerSetThicknessRange\(\)](#). If the current layer thickness is constant, `minThickness_nm` and `maxThickness_nm` will be set to the specified constant thickness and `numSteps` will be 0. To determine whether the layer's thickness has been previously set to a constant value or not, use [frlsLayerIsThicknessConst\(\)](#).

`numSteps` will always return the same value passed to [frlsLayerSetThicknessRange\(\)](#). The true value of the number of subdivisions, as described in [frlsLayerSetThicknessRange\(\)](#), is calculated during fitting.

4.3.16 frlsLayerGetMaterial()

```
MaterialHandle frlsLayerGetMaterial(LayerStackHandle layerStack, uint32_t layerID);
```

Get the specified layer's material.

Parameters:

- `layerStack`: The handle of the layer stack.
- `layerID`: The zero-based index of the layer.

Return value:

If the function succeeds the return value will be the handle of the material used to initialize the layer's refractive index (see [frlsLayerSetRefrIndexFromMaterial\(\)](#))

If the function fails, or if the layer has a custom refractive index set using [frlsLayerSetRefrIndex\(\)](#), the returned value will be an invalid handle.

Remarks:

None.

4.3.17 frlsLayerGetRefrIndexFitParamMask()

```
uint32_t frlsLayerGetRefrIndexFitParamMask(LayerStackHandle layerStack, uint32_t layerID);
```

Get the refractive index parameters that will be calculated during fitting.

Parameters:

- layerStack: The handle of the layer stack.
- layerID: The zero-based index of the layer.

Return value:

The function returns a bitfield indicating which parameters of the refractive index equation will be calculated during fitting.

If the function fails, 0 will be returned.

Remarks:

None.

4.3.18 frlsLayerGetRefrIndexType()

```
uint32_t frlsLayerGetRefrIndexType(LayerStackHandle layerStack, uint32_t layerID);
```

Get the selected refractive index type of the specified layer.

Parameters:

- `layerStack`: The handle of the layer stack.
- `layerID`: The zero-based index of the layer.

Return value:

If the function succeeds, the return value is the refractive index equations selected for the material using either [frlsLayerSetRefrIndex\(\)](#) or [frlsLayerSetRefrIndexFromMaterial\(\)](#).

If the function fails, the return value will be equal to `FR_REFRACTIVE_INDEX_UNKNOWN`.

Remarks:

None.

4.3.19 frlsLayerGetRefrIndexParams()

```
int32_t frlsLayerGetRefrIndexParams(LayerStackHandle layerStack, uint32_t layerID, double* params,
uint32_t* numParams);
```

Get the refractive index parameters of the specified layer.

Parameters:

- `layerStack`: The handle of the layer stack.
- `layerID`: The zero-based index of the layer.
- `params`: A buffer to hold the refractive index parameters.
- `numParams`: Pointer to an unsigned integer to hold the number of refractive index parameters.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of the failure.

Remarks:

If `params` is set to the `NULL` pointer, the function will return the number of parameters in the `numParams` argument.

`params` should be large enough to be able to hold all the refractive index parameters.

4.3.20 frlsLayerSetType()

```
int32_t frlsLayerSetType(LayerStackHandle layerStack, uint32_t layerID, uint32_t layerType);
```

Parameters:

Return value:

Remarks:

4.3.21 frlsLayerGetType()

```
uint32_t frlsLayerGetType(LayerStackHandle layerStack, uint32_t layerID);
```

Parameters:

Return value:

Remarks:

4.4 Materials

Materials are separated into two different databases:

- Build-in (provided by ThetaMetrisis)
- User

The build-in database is initialized via [frInit\(\)](#) using the specified database file, provided by ThetaMetrisis.

The user database is considered valid at all times. Initially it is an empty **in-memory** SQLite database. [frOpenUserMaterialDatabase\(\)](#) can be used to replace the in-memory database with a file-backed SQLite database, for persistent material storage. See [frOpenUserMaterialDatabase\(\)](#) and [frCloseUserMaterialDatabase\(\)](#) for details.

The build-in database has the following limitations:

- Cannot create new material categories
- Cannot create new materials or refractive index values
- Cannot change/delete existing material categories
- Cannot change/delete existing materials or refractive index values
- Cannot read material raw data refractive index values. Only the parameters of analytic refractive index formulas (e.g. Cauchy) can be read

Each one of the two databases has its own set of material categories (see [frmcXXX\(\)](#) functions) and materials (see [frmXXX\(\)](#) functions). Each material is assigned to a specific category and can have multiple different sets of refractive index parameters, one for each refractive index equation (see Refractive Index Equations table below).

Materials and material categories are assigned a Unique Resource Identifier (URI). The URI has the following format:

```
<db>:<category>/<material>
```

where:

- `db`: The database. It can either be "frapi" or "user"
- `category`: The name of the category
- `material`: The name of the material

URIs are case insensitive.



For example, the URI "frapi:semiconductors/si" identifies the material named "Si" from the category "Semiconductors" of the build-in database. The URI "user:my awesome category" identifies the material category named "My Awesome Category" from the user's database.

The available refractive index equations are shown in the table below. Each equation requires a specific number of parameters. The number of parameters and their meaning are also described in the table below.

Refractive Index	# Params	Equations
FR_REFRACTIVE_INDEX_CAUCHY	3	$n(\lambda_{\mu m}) = P_0 + \frac{P_1}{\lambda_{\mu m}^2} + \frac{P_2}{\lambda_{\mu m}^4}, k(\lambda_{\mu m}) = 0.0$
FR_REFRACTIVE_INDEX_CAUCHY	4	$n(\lambda_{\mu m}) = P_0 + \frac{P_1}{\lambda_{\mu m}^2} + \frac{P_2}{\lambda_{\mu m}^4}, k(\lambda_{\mu m}) = P_3$
FR_REFRACTIVE_INDEX_CAUCHY	6	$n(\lambda_{\mu m}) = P_0 + \frac{P_1}{\lambda_{\mu m}^2} + \frac{P_2}{\lambda_{\mu m}^4},$ $k(\lambda_{\mu m}) = P_3 e^{1.24 P_4 (\frac{1}{\lambda_{\mu m}} - \frac{1}{P_5})}$
FR_REFRACTIVE_INDEX_SELLMEIER	6	$n(\lambda_{\mu m}) = 1 + \frac{P_0 \lambda_{\mu m}^2}{\lambda_{\mu m}^2 - P_3} + \frac{P_1 \lambda_{\mu m}^2}{\lambda_{\mu m}^2 - P_4} + \frac{P_2 \lambda_{\mu m}^2}{\lambda_{\mu m}^2 - P_5},$ $k(\lambda_{\mu m}) = 0.0$
FR_REFRACTIVE_INDEX_SELLMEIER	7	$n(\lambda_{\mu m}) = 1 + \frac{P_0 \lambda_{\mu m}^2}{\lambda_{\mu m}^2 - P_3} + \frac{P_1 \lambda_{\mu m}^2}{\lambda_{\mu m}^2 - P_4} + \frac{P_2 \lambda_{\mu m}^2}{\lambda_{\mu m}^2 - P_5},$ $k(\lambda_{\mu m}) = P_6$
FR_REFRACTIVE_INDEX_DRUDE	3	$\vec{n}^2(\lambda_{\mu m}) = (P_0 - \frac{P_1^2}{(\frac{1.24}{\lambda_{\mu m}})^2 + P_2^2}) + j(\frac{P_1^2 P_2}{(\frac{1.24}{\lambda_{\mu m}})^3 + \frac{1.24}{\lambda_{\mu m}} P_2^2})$
FR_REFRACTIVE_INDEX_LORENTZ	3t + 1	TODO
FR_REFRACTIVE_INDEX_FOROUHI_BLOOMER	3t + 2	TODO
FR_REFRACTIVE_RAW_DATA	3t	Triples of lambda, n and k values. Linear interpolation between closest points is performed to calculate values at arbitrary wavelengths.

Refractive Index Equations

4.4.1 frOpenUserMaterialDatabase()

```
int32_t frOpenUserMaterialDatabase(const char* materialDB);
```

Open the specified database file as the user's database.

Parameters:

- `materialDB`: The absolute path to the database file.

Return value:

If the function succeeds, the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of failure.

Remarks:

Opening a material database using this function implicitly destroys the existing user's material database.

For example, if this function is called immediately after `frInit()`, then the default in-memory database will be destroyed and the new database will be used as the user's material database.

If the specified file is not found, a new material database will be created.

The old database is replaced only if the new database has been correctly loaded. If an error occurs during the loading or creation of the new database, no changes to the old database occur.



After a successful call to this function, all handles from the previous user database should be considered invalid and should not be used.



Do not try opening ThetaMetrisis' material database with this function! Doing so might result in database corruption and/or invalid refractive index data.

4.4.2 frCloseUserMaterialDatabase()

```
int32_t frCloseUserMaterialDatabase();
```

Replace the current user's material database with an empty in-memory database.

Parameters:

None.

Return value:

If the function succeeds, the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of failure.

Remarks:

Closing a material database using this function implicitly creates a new, empty, in-memory database.

The old database is replaced only if the new database has been correctly created. If an error occurs during the creation of the new database, no changes to the old database occur.



After a successful call to this function, all handles from the previous user database should be considered invalid and should not be used.

4.4.3 frmCreate()

```
MaterialCategoryHandle frmCreate(const char* name);
```

Create a new material category.

Parameters:

- name: The name of the category.

Return value:

If the function succeeds the returned value is the handle to the new category.

If the function fails the returned value will be an invalid handle.

Remarks:

New categories are added to the user's material database. The user's material database has a predefined material category named Default (URI: user:default).

4.4.4 frmcDelete()

```
int32_t frmcDelete(MaterialCategoryHandle handle);
```

Delete the specified material category from the user's material database.

Parameters:

- `handle`: The handle of the material category.

Return value:

If the function succeeds the returned value will be `FR_ERROR_NONE`.

If the function fails, check the returned value for the reason of the failure.

Remarks:

You cannot delete categories from the build-in material database.

4.4.5 frmUpdate()

```
int32_t frmUpdate(MaterialCategoryHandle handle, const char* name);
```

Update the specified material category in the user's material database.

Parameters:

- `handle`: The handle of the material category.
- `name`: The new name of the material category.

Return value:

If the function succeeds the returned value will be `FR_ERROR_NONE`.

If the function fails, check the returned value for the reason of the failure.

Remarks:

You cannot update categories from the build-in material database.

4.4.6 frmcGetName()

```
int32_t frmcGetName(MaterialCategoryHandle handle, char* name, uint32_t maxLen);
```

Get the name of the specified material category.

Parameters:

- `handle`: The handle of the material category.
- `name`: A buffer for holding the category name.
- `maxLen`: The maximum length of the specified name buffer.

Return value:

If the function succeeds, the returned value will be `FR_ERROR_NONE` and `name` will hold the name of the material category.

If the function fails, check the returned value for the reason of the failure.

Remarks:

The maximum length of a category's name is internally limited to 64 characters.

4.4.7 frmcGetUri()

```
int32_t frmcGetUri(MaterialCategoryHandle handle, char* uri, uint32_t maxLen);
```

Get the Unique Resource Identifier (URI) of the specified material category.

Parameters:

- `handle`: The handle of the material category.
- `uri`: A buffer for holding the category URI.
- `maxLen`: The maximum length of the specified `uri` buffer.

Return value:

If the function succeeds, the returned value will be `FR_ERROR_NONE` and `uri` will hold the URI of the specified material category.

If the function fails, check the returned value for the reason of the failure.

Remarks:

None.

4.4.8 frmcFindByUri()

```
MaterialCategoryHandle frmcFindByUri(const char* uri);
```

Find a category using its URI.

Parameters:

- `uri`: The URI of the category.

Return value:

If the function succeeds the return value is the handle to the material category.

If the function fails, an invalid handle is returned.

Remarks:

None.

4.4.9 frmCreate()

```
MaterialHandle frmCreate(MaterialCategoryHandle categoryHandle, const char* name, uint32_t
defaultRIType);
```

Create a new material.

Parameters:

- `categoryHandle`: The category of the material.
- `name`: The name of the new material.
- `defaultRIType`: The default refractive index type of the material.

Return value:

If the function succeeds, the return value is the handle to the new material.

If the function fails, an invalid handle is returned.

Remarks:

New materials can only be added to the user database.

4.4.10 frmDelete()

```
int32_t frmDelete(MaterialHandle handle);
```

Delete an existing material.

Parameters:

- `handle`: The handle of the material to delete.

Return value:

If the function succeeds the returned value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of failure.

Remarks:

You cannot delete materials from the build-in database.

4.4.11 frmUpdate()

```
int32_t frmUpdate(MaterialHandle handle, MaterialCategoryHandle categoryHandle, const char* name,
uint32_t defaultRIType);
```

Update a material.

Parameters:

- `handle`: The handle of the material.
- `categoryHandle`: The handle of the new category for the material.
- `name`: The new name of the material.
- `defaultRIType`: The new default refractive index equation for the material.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of failure.

Remarks:

You cannot update materials from the build-in database.

4.4.12 frmGetName()

```
int32_t frmGetName(MaterialHandle handle, char* name, uint32_t maxLen);
```

Get the name of the specified material.

Parameters:

- `handle`: The handle of the material.
- `name`: Buffer for holding the material name.
- `maxLen`: The maximum number of elements the buffer can hold.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE` and `name` will hold the name of the material.

If the function fails, check the return value for the reason of failure.

Remarks:

If the name buffer isn't large enough to accommodate the material name, only the first `maxLen` characters of the name will be transferred. Material names are internally limited to 128 characters.

4.4.13 frmGetCategory()

```
MaterialCategoryHandle frmGetCategory(MaterialHandle handle);
```

Get the material's category.

Parameters:

- handle: The handle of the material.

Return value:

If the function succeeds, the return value will be the handle to the material's category.

If the function fails an invalid handle will be returned.

Remarks:

None.

4.4.14 frmGetDefaultRefrIndexType()

```
uint32_t frmGetDefaultRefrIndexType(MaterialHandle handle);
```

Get the default refractive index equation of the specified material.

Parameters:

- `handle`: The handle of the material.

Return value:

If the function succeeds the return value will be the default refractive index equation of the material.

If the function fails the return value will be `FR_REFRACTIVE_INDEX_UNKNOWN`.

Remarks:

None.

4.4.15 frmGetAvailableRefrIndexTypes()

```
uint32_t frmGetAvailableRefrIndexTypes(MaterialHandle handle);
```

Get the available refractive index equations for the specified material.

Parameters:

- handle: The handle of the material.

Return value:

The function returns a bitfield, where each bit indicates whether the material has valid parameters for the corresponding refractive index equation.

In case of failure, 0 is returned (i.e. no refractive index equations are available).

Remarks:

In order to check whether a specific refractive index equation is available, test the corresponding bit of the returned bitfield.

Example: Check if the selected material has parameters for the Cauchy equation.

```
MaterialHandle matHandle = ...;
const uint32_t availableRefrIndexTypes = frmGetAvailableRefrIndexTypes(matHandle);
if((availableRefrIndexTypes & (1u << FR_REFRACTIVE_INDEX_CAUCHY)) != 0) {
    // Do something with the Cauchy parameters.
} else {
    // The specified material does not have parameters for the Cauchy equation.
}
```

4.4.16 frmGetUri()

```
int32_t frmGetUri(MaterialHandle mat, char* uri, uint32_t maxLen);
```

Get the Unique Resource Identifier of the specified material.

Parameters:

- `handle`: The handle of the material.
- `uri`: A buffer to hold the URI.
- `maxLen`: The maximum length of the buffer.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE` and `uri` will hold the URI of the material.

If the function fails, check the return value for the reason of failure.

Remarks:

None.

4.4.17 frmFindByUri()

```
MaterialHandle frmFindByUri(const char* uri);
```

Find a material by using its URI.

Parameters:

- `uri`: The URI of the material.

Return value:

If the function succeeds the return value will be the handle of the material.

If the function fails, an invalid handle will be returned.

Remarks:

None.

4.4.18 frmCreateRefrIndex()

```
int32_t frmCreateRefrIndex(MaterialHandle handle, uint32_t riType, const double* params, uint32_t numParams);
```

Create a new refractive index for the specified material.

Parameters:

- `handle`: The handle of the material.
- `riType`: The refractive index equation.
- `params`: The parameters for the refractive index equation.
- `numParam`: The number of parameters in the `params` array.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of failure.

Remarks:

You cannot create new refractive index equations for materials from the build-in database.

Check the table at the start of the [Materials](#) section for details on the number of parameters expected for each refractive index equation and their meaning.

4.4.19 frmDeleteRefrIndex()

```
int32_t frmDeleteRefrIndex(MaterialHandle handle, uint32_t riType);
```

Delete a refractive index from the specified material.

Parameters:

- `handle`: The handle of the material.
- `riType`: The refractive index equation to remove.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of the failure.

Remarks:

You cannot delete refractive index equations from material of the build-in database.

4.4.20 frmUpdateRefrIndex()

```
int32_t frmUpdateRefrIndex(MaterialHandle handle, uint32_t riType, const double* params, uint32_t numParams);
```

Update an existing refractive index from the specified material.

Parameters:

- `handle`: The handle of the material.
- `riType`: The refractive index equation to update.
- `params`: The new parameters for the refractive index equation.
- `numParams`: The number of parameters in the `params` array.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of failure.

Remarks:

You cannot update refractive index equations of materials from the build-in database.

4.4.21 frmGetRefrIndex()

```
int32_t frmGetRefrIndex(MaterialHandle handle, uint32_t riType, double* params, uint32_t* numParams);
```

Get the refractive index parameters from the specified material.

Parameters:

- `handle`: The handle of the material.
- `riType`: The refractive index equation to read.
- `params`: A buffer to hold the parameters of the refractive index equation.
- `numParams`: Pointer to an unsigned integer to hold the number of parameters copied to `params` buffer.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE` and `params` will hold the requested refractive index parameters.

If the function fails, check the return value for the reason of failure.

Remarks:

Setting `params` to `NULL` will only return the number of parameters in `numParams`.

You cannot read raw data (`FR_REFRACTIVE_INDEX_RAW_DATA`) parameters from materials from the build-in database.

4.4.22 frmHasRefrIndexType()

```
bool frmHasRefrIndexType(MaterialHandle handle, uint32_t riType);
```

Check if the specified material has parameters for the specified refractive index equation.

Parameters:

- `handle`: The handle of the material.
- `riType`: The refractive index equation to check.

Return value:

The function returns a boolean value indicating whether the specified material has parameters for the specified refractive index equation (`true`) or not (`false`).

If the function fails, `false` is returned.

Remarks:

This is a shortcut to the code presented in [frmGetAvailableRefrIndexTypes\(\)](#) and it's useful for checking the availability of a specific refractive index equation.

If you want to iterate over all available refractive index equations it's advised to use [frmGetAvailableRefrIndexTypes\(\)](#).

4.4.23 frmCalcEffectiveRefrIndex()

```
double frmCalcEffectiveRefrIndex(MaterialHandle handle, uint32_t riType, const double* wavelengths,
uint32_t numWavelengths)
```

Calculate the Effective Refractive Index of the specified material's refr. index at the specified wavelength range.

Parameters:

- `handle`: The handle of the material.
- `riType`: The refractive index equation to check.
- `wavelengths`: The list of wavelengths over which the effective refractive index will be calculated.
- `numWavelengths`: The total number of wavelengths in the `wavelengths` array.

Return value:

If the function succeeds, the return value will be the effective refractive index.

If the function fails, the return value will be 0.0 .

Remarks:

The Effective Refractive Index is useful as an estimation of a material's refractive index when converting frequencies to thicknesses during frequency analysis (see [FFT](#) section).

4.4.24 frmcEnumBegin()

```
MaterialCategoryIterator* frmcEnumBegin(uint32_t dbMask);
```

Begin material category enumeration of the specified databases.

Parameters:

- dbMask: Bitfield indicating which material databases to enumerate.

Return value:

If the function succeeds the return value will be an iterator to use with `frmcEnumXXX()` functions.

If the function fails, the return value will be the `NULL` pointer.

Remarks:

dbMask can be a combination of the following values:

- `FR_MATDB_BUILDIN_MASK` for the build-in database.
- `FR_MATDB_USER_MASK` for the user database.

The developer is responsible to free the returned iterator by calling `frmcEnumEnd()` when it's not needed anymore.

4.4.25 frmEnumEnd()

```
void frmEnumEnd(MaterialCategoryIterator* iter);
```

End a material category enumeration.

Parameters:

iter: Material category iterator previously returned by [frmEnumBegin\(\)](#).

Return value:

None.

Remarks:

After the function returns, the iterator is no longer valid and should not be used again.

4.4.26 frmEnumReset()

```
void frmEnumReset(MaterialCategoryIterator* iter);
```

Reset the material category iterator.

Parameters:

- iter: Material category iterator previously returned by [frmEnumBegin\(\)](#).

Return value:

None.

Remarks:

None.

4.4.27 frmEnumNext()

```
MaterialCategoryHandle frmEnumNext(MaterialCategoryIterator* iter);
```

Get the next material category.

Parameters:

- iter: Material category iterator previously returned by [frmEnumBegin\(\)](#).

Return value:

If the function succeeds the next material category handle will be returned.

If the function fails or there are no more material categories to enumerate, an invalid handle will be returned.

Remarks:

None.

4.4.28 frmcEnumGetCount()

```
uint32_t frmcEnumGetCount(MaterialCategoryIterator* iter);
```

Get the total number of material categories to be enumerated.

Parameters:

- iter: Material category iterator previously returned by [frmcEnumBegin\(\)](#).

Return value:

The number of material categories.

Remarks:

None.

4.4.29 frmcMaterialsEnumBegin()

```
MaterialIterator* frmcMaterialsEnumBegin(MaterialCategoryHandle handle);
```

Begin the enumeration of all the materials from a specific category.

Parameters:

- `handle`: The handle of the material category.

Return value:

If the function succeeds the return value will be an iterator to use with `frmcMaterialsEnumXXX()` functions.

If the function fails, the return value will be the `NULL` pointer.

Remarks:

The developer is responsible to free the returned iterator by calling `frmcMaterialsEnumEnd()` when it's not needed anymore.

4.4.30 frmcMaterialsEnumEnd()

```
void frmcMaterialsEnumEnd(MaterialIterator* iter);
```

End a material enumeration.

Parameters:

- iter: The material iterator.

Return value:

None.

Remarks:

After the function returns, the iterator is no longer valid and should not be used again.

4.4.31 frmcMaterialsEnumReset()

```
void frmcMaterialsEnumReset(MaterialIterator* iter);
```

Reset the material iterator.

Parameters:

- iter: The material iterator.

Return value:

None.

Remarks:

None.

4.4.32 frmcMaterialsEnumNext()

```
MaterialHandle frmcMaterialsEnumNext(MaterialIterator* iter);
```

Get the next material.

Parameters:

- iter: The material iterator.

Return value:

If the function succeeds the next material handle will be returned.

If the function fails or there are no more materials to enumerate, an invalid handle will be returned.

Remarks:

None.

4.4.33 frmcMaterialsEnumGetCount()

```
uint32_t frmcMaterialsEnumGetCount(MaterialIterator* iter);
```

Get the total number of materials to be enumerated.

Parameters:

- iter: The material iterator.

Return value:

The total number of materials.

Remarks:

None.

4.4.34 frmSearchBegin()

```
MaterialIterator* frmSearchBegin(const char* nameRegex, uint32_t dbMask);
```

Search materials by name, using regular expressions.

Parameters:

- nameRegex: Regular expression for the name of the material. See the Remarks section for details.
- dbMask: Which database to search.

Return value:

If the function succeeds, the return value will be a `MaterialIterator` pointer which can be used with `frmSearchXXX()` to enumerate the materials matching the regular expression.

If the function fails, the return value will be the `NULL` pointer.

Remarks:

`dbMask` is a bitfield indicating which of the two databases will be searched (`FR_MATDB_BUILDIN_MASK` and `FR_MATDB_USER_MASK`).

`nameRegex` is a regular expression to match against the names of the materials. `nameRegex` is case sensitive.

The regular expression engine supports the following regex operators.

Operator	Meaning
.	Dot, matches any character
^	Start anchor, matches beginning of string
\$	End anchor, matches end of string
*	Asterisk, match zero or more (greedy)
+	Plus, match one or more (greedy)
?	Question, match zero or one (non-greedy)
[abc]	Character class, match if one of {'a', 'b', 'c'}
[a-zA-Z]	Character ranges, the character set of the ranges a-z and A-Z
\s	Whitespace, \t \f \r \n \v and spaces
\S	Non-whitespace
\w	Alphanumeric, [a-zA-Z0-9_]
\W	Non-alphanumeric
\d	Digits, [0-9]
\D	Non-digits

Regular Expression Operators

The returned `MaterialIterator` should be freed using `frmSearchEnd()` when it is not needed anymore.

The following examples demonstrates how to find all materials from the build-in database starting with “Si”.

```
MaterialIterator* it = frmSearchBegin("^Si", FR_MATDB_BUILDIN_MASK);
if (it != nullptr) {
    MaterialHandle mh = frmSearchNext(it);
    while (FR_HANDLE_IS_VALID(mh)) {
        // Do something with the material...

        // Move on to the next material.
        mh = frmSearchNext(it);
    }

    frmSearchEnd(it);
}
```

4.4.35 frmSearchEnd()

```
void frmSearchEnd(MaterialIterator* iter);
```

End a search, deallocating the MaterialIterator.

Parameters:

- iter: The material iterator previously obtained using [frmSearchBegin\(\)](#)

Return value:

None.

Remarks:

None.

4.4.36 frmSearchReset()

```
void frmSearchReset(MaterialIterator* iter);
```

Reset the iterator to the start of the search results.

Parameters:

- iter: The material iterator previously obtained using [frmSearchBegin\(\)](#)

Return value:

None.

Remarks:

None.

4.4.37 frmSearchNext()

```
MaterialHandle frmSearchNext(MaterialIterator* iter);
```

Get the next found material.

Parameters:

- `iter`: The material iterator previously obtained using [frmSearchBegin\(\)](#)

Return value:

The function returns the next material from the list of found materials. If there's no more materials left, an invalid handle is returned.

Remarks:

None.

4.4.38 frmSearchGetCount()

```
uint32_t frmSearchGetCount(MaterialIterator* iter);
```

Get the number of materials found.

Parameters:

- iter: The material iterator previously obtained using [frmSearchBegin\(\)](#)

Return value:

The function returns the total number of materials found.

Remarks:

None.

4.5 Fitting

The Fitting module includes functions for fitting layer stack parameters to experimental spectra.

Each fitting operation is performed in a separate context. All fitting contexts are internally managed by a single fitting manager but they are independent of each other, so they can be executed in parallel.

FR-API does not use multiple threads to execute fitting operations. [frfitExecute\(\)](#) is executed synchronously on the calling thread. It's the developer's responsibility to move fitting calculations to separate threads and manage the results, if it is needed.

4.5.1 frCreateFittingContextEx()

```
FitContext* frCreateFittingContextEx(LayerStackHandle layerStack, const FitConfigEx* fitConfig);
```

Create a new fitting context using the specified layer stack and configuration.

Parameters:

- `layerStack`: The layer stack to fit
- `fitConfig`: Configuration parameters for the fit.


Return value:

If the function succeeds, the returned value is a new fitting context.

If the function fails, the returned value will be the NULL pointer.

Remarks:

The specified layer stack is internally cloned. The developer is free to destroy the original layer stack, if it is no longer needed.

 Immediately after the function returns, both the original layer stack and the internal clone have the exact same values. There's no connection between the two stacks. I.e. you cannot change a parameter in the original stack after calling `frCreateFittingContextEx()` and expect the new value to appear in the internal clone. The internally cloned stack is an exact copy of the original stack **at the moment** `frCreateFittingContextEx()` was called.

The specified fit configuration is copied to an internal object. The developer is free to allocate the `FitConfigEx` object on the stack or free the heap-allocated object after a successful call to this function.

The `FitConfigEx` struct is shown in the snippet below along with the meaning of each field.

```

struct FitConfigEx
{
    // Array of wavelengths over which fitting will be performed.
    const double* m_Wavelengths;

    // The (calibrated) reference spectrum corresponding to the list of wavelengths.
    const double* m_RefSpectrum;

    // The dark spectrum corresponding to the list of wavelengths.
    const double* m_DarkSpectrum;

    // Initial scale coefficients.
    double m_Scale[FR_FIT_SCALE_EQU_NUM_PARAMS]; // scale(lambda) = c0

    // Initial offset coefficients.
    double m_Offset[FR_FIT_OFFSET_EQU_NUM_PARAMS]; // offset(lambda) = c0 + c1 * lambda

    // Angle of incidence of the incoming light in radians.
    double m_IncidentAngle; // [0, PI]

    // The total number of wavelengths and samples in the m_Wavelengths, m_RefSpectrum and
    m_DarkSpectrum arrays.
    uint32_t m_NumWavelengths;

    // Fitting flags
    uint32_t m_Flags;

    // Equation to use for fitting.
    uint32_t m_Equation;

    // The fitting method to use
    uint32_t m_Method

    // Configuration parameters used when m_Method is set to FR_FIT_METHOD_LEAST_SQUARES or
    FR_FIT_METHOD_FOURIER_LEAST_SQUARES. See below for details.
    FitConfigLeastSquares m_LeastSquaresConfig;

    // Configuration parameters used when m_Method is set to FR_FIT_METHOD_FOURIER or
    FR_FIT_METHOD_FOURIER_LEAST_SQUARES. See below for details.
    FitConfigFourier m_FourierConfig;

    // Configuration parameters used when m_Method is set to FR_FIT_METHOD_FOURIER_LEAST_SQUARES.
    See below for details.
    FitConfigFourierLeastSquares m_FourierLSConfig;
};

```

`m_Wavelengths`, `m_RefSpectrum` and `m_DarkSpectrum` pointers should be valid as long as the fitting context is alive and in use. All three arrays should have the same number of elements equal to `m_NumWavelengths`.

`m_Scale` and `m_Offset` are only calculated if `m_Flags` includes `FR_FITFLAGS_FIT_SCALE` and `FR_FITFLAGS_FIT_OFFSET` respectively. Independent of whether they are calculated during fitting or not, they are expected to have valid initial values. Currently `m_Scale` is a constant function of the wavelength (`FR_FIT_SCALE_EQU_NUM_PARAMS = 1`) and its default initial value should be equal to `1.0`. `m_Offset` is a linear function of the wavelength (`FR_FIT_OFFSET_EQU_NUM_PARAMS = 2`) and its default initial values should be equal to `{ 0.0, 0.0 }`.

`m_Equation` determines the equation that will be used during fitting. If the value is equal to `FR_FIT_EQU_REFLECTANCE_NORMAL` or `FR_FIT_EQU_TRANSMITTANCE_NORMAL` a simplified reflectance/transmittance equation for normal incidence is used and `m_IncidentAngle` is ignored. If its value is equal to `FR_FIT_EQU_REFLECTANCE_GENERIC` or `FR_FIT_EQU_TRANSMITTANCE_GENERIC` a generic reflectance/transmittance equation is used and `m_IncidentAngle` should hold the angle of incidence of the incoming light, in radians. If `m_Flags` includes `FR_FITFLAGS_FIT_ANGLE` then `m_IncidentAngle` is used as the initial guess of the incident angle.

`m_Method` is used to select the preferred method for determining layer parameters.

- If the value is equal to `FR_FIT_METHOD_LEAST_SQUARES` then least squares fitting will be performed and all parameters specified as unknowns will be calculated (i.e. layer thicknesses, refractive index parameters, equation scale/offset, etc.).
- If the value is equal to `FR_FIT_METHOD_FOURIER` then only layer thicknesses are calculated and all other unknown parameters (layer refr. index, equation scale/offset, etc.) will not change and they will keep their initial values.
- If the value is equal to `FR_FIT_METHOD_FOURIER_LEAST_SQUARES` then a combination of the above two methods is used. First, the Fourier algorithm is executed to estimate all layer thicknesses and then the Least Squares algorithm is executed around those initial guesses.

Each method has its own set of configuration parameters. `FR_FIT_METHOD_LEAST_SQUARES` uses `m_LeastSquaresConfig` member shown in the code snippet below.

```
struct FitConfigLeastSquares
{
    // Maximum fitting iterations to perform.
    uint32_t m_MaxFittingIterations;
};
```

On the other hand, `FR_FIT_METHOD_FOURIER` uses the `m_FourierConfig` member shown in the code snippet below.

```

struct FitConfigFourier
{
    // The threshold (in the (0, 1] range) relative to the frequency spectrum maximum, used to
    // decide whether a
    // detected peak is valid (peak magnitude greater than the threshold) or not.
    //
    // Setting this parameter to 0.0 will use the frequency spectrum's average as the threshold.
    float m_PeakThresholdRel;

    // The range of thicknesses over which the frequency spectrum will be calculated.
    // Thickness is converted to frequency using the effective refractive index of the 1st layer
    // (the layer
    // immediately below air).
    //
    // Setting both limits to 0.0 will force the algorithm to use the largest possible frequency
    // range.
    float m_ThicknessRange[2];          // { min, max }

    // The number of points in the frequency spectrum.
    uint32_t m_FreqSpectrumNumPoints;
};

```

When `m_Method` is set to `FR_FIT_METHOD_FOURIER_LEAST_SQUARES` then each step of the combined algorithm uses the above configuration structs as well as the `m_FourierLSConfig` member shown below.

```

struct FitConfigFourierLeastSquares
{
    // The percentage (in [0, 100] range) around the thickness estimated by the Fourier step where
    // the Least Squares algorithm will be executed.
    float m_ThicknessRangePercentage;

    // The number of thickness steps inside the calculated thickness range to execute. The value has
    // the same meaning as the `numSteps` parameter
    // of frlsLayerSetThicknessRange().
    uint32_t m_NumThicknessSteps;
};

```

The function might fail for the following reasons:

- There's no parameter selected for fitting. None of the spectrum related parameters (scale, offset or incident angle) has been selected for fitting using the `m_Flags` configuration parameter nor any layer stack parameter (layer thickness or refractive index equation coefficient). There must be at least one unknown variable in order to execute the fitting algorithm.
- A memory allocation failed. This is system specific and should not happen under normal circumstances.

ⓘ It is advised to keep the number of unknown variables for fitting as small as possible. The available equations are non-linear equations and the error function minimized during fitting has many local minima. A large number of unknown variables with poor initial guesses will most likely result in a local minimum, i.e. poor fitting.

ⓘ Keep in mind that the available equations calculate *optical* thicknesses. That is, the product of a layer's thickness with its refractive index at a specific wavelength. Having both a layer's thickness and its refractive index as unknown variables, might lead to incorrect results.

4.5.2 frCreateFittingContext()

```
FitContext* frCreateFittingContext(LayerStackHandle layerStack, const FitConfig* fitConfig);
```

DEPRECATED: This internally calls [frCreateFittingContextEx\(\)](#) with `FitConfigEx::m_Method` equal to `FR_FIT_METHOD_LEAST_SQUARES`. All other parameters are copied from the specified `fitConfig`.

Parameters:

- `layerStack`: The layer stack to fit
- `fitConfig`: Configuration parameters for the fit.

Return value:

If the function succeeds, the returned value is a new fitting context.

If the function fails, the returned value will be the `NULL` pointer.

Remarks:

See [frCreateFittingContextEx\(\)](#) for details.

The `FitConfig` struct is shown in the snippet below along with the meaning of each field.

```
struct FitConfig
{
    // Array of wavelengths over which fitting will be performed.
    const double* m_Wavelengths;

    // The (calibrated) reference spectrum corresponding to the list of wavelengths.
    const double* m_RefSpectrum;

    // The dark spectrum corresponding to the list of wavelengths.
    const double* m_DarkSpectrum;

    // Initial scale coefficients.
    double m_Scale[FR_FIT_SCALE_EQU_NUM_PARAMS]; // scale(lambda) = c0

    // Initial offset coefficients.
    double m_Offset[FR_FIT_OFFSET_EQU_NUM_PARAMS]; // offset(lambda) = c0 + c1 * lambda

    // Angle of incidence of the incoming light in radians.
    double m_IncidentAngle; // [0, PI]

    // Maximum fitting iterations to perform.
    uint32_t m_MaxFittingIterations;

    // The total number of wavelengths and samples in the m_Wavelengths, m_RefSpectrum and
    m_DarkSpectrum arrays.
    uint32_t m_NumWavelengths;

    // Fitting flags
    uint32_t m_Flags;

    // Equation to use for fitting.
    uint32_t m_Equation;
};
```

4.5.3 frDestroyFittingContext()

```
void frDestroyFittingContext(FitContext* fitContext);
```

Destroy a fitting context.

Parameters:

- fitContext: The fitting context to destroy.

Return value:

None.

Remarks:

After the fitting context is destroyed, it's no longer valid. The arrays pointed by `FitConfig::m_Wavelengths`, `FitConfig::m_RefSpectrum` and `FitConfig::m_DarkSpectrum` during the creation of the fitting context are no longer needed and the developer is free to deallocate them.

4.5.4 frfitExecute()

```
int32_t frfitExecute(FitContext* fitContext, const double* spectrum, FitResult* res);
```

Execute a fit.

Parameters:

- `fitContext`: The fitting context.
- `spectrum`: The experimental spectrum.
- `res`: Pointer to a `FitResult` object for holding the results of the fitting.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of the failure.

Remarks:

`frfitExecute()` is a synchronous function. It will block until the fitting algorithm is completed. FR-API does not use multiple threads internally to execute fitting operations in parallel. The developer is responsible for moving the call to `frfitExecute()` to a separate thread in order not to block the application main/UI thread.

`spectrum` is the raw experimental spectrum as it has been acquired by a device. The developer must make sure to only pass the part of the spectrum corresponding to the wavelengths used to initialize the fitting context. In case `frfitExecute()` is called from a separate thread than the one used to acquire the spectrum, the `spectrum` pointer must be valid for the duration of the fit.

The `FitResult` struct is shown in the snippet below with information about each field.

```

struct FitResult
{
    // The error between the normalized experimental spectrum and the initial theoretical spectrum.
    double m_InitialErrorSqr;

    // The final (minimal) error between the normalized experimental spectrum and the theoretical
    spectrum.
    double m_FinalErrorSqr;


    // R^2
    double m_RSqr;

    // The number of fitting iterations performed.
    uint32_t m_NumFittingIterations;

    // The reason the minimization algorithm has been terminated.
    uint32_t m_ReasonForTerminating;

    // Statistics about the minimization algorithm.
    uint32_t m_NumFunctionEvals;
    uint32_t m_NumJacobianEvals;
    uint32_t m_NumLinearSystemSolved;
};

```

 Depending on various factors, such as the number of unknown variables, the equation used, the CPU etc., fitting might take an arbitrary amount of time to complete. In a GUI application, it is advised to use a separate thread to execute fitting operations, in order to keep the main UI thread responsive at all times.

In order to get the results of the fitting algorithm (new layer thicknesses and refractive index equation coefficients) you have to use [frfitGetLayerStack\(\)](#) to get the handle to the internal layer stack and use [fr1sXXX\(\)](#) functions to retrieve the information you want. As mentioned in [frCreateFittingContext\(\)](#) there's no connection between the original layer stack and the internal clone used by the fitting context.

4.5.5 frfitGetLayerStack()

```
LayerStackHandle frfitGetLayerStack(const FitContext* fitContext);
```

Get the internal layer stack of the fitting context.

Parameters:

- `fitContext`: The fitting context.

Return value:

If the function succeeds, the return value will be the handle to the internal layer stack.

If the function fails it will return an invalid handle.

Remarks:

The fitting context's internal layer stack is valid as long as the fitting context is alive.



To avoid any potential race conditions when using multiple threads it is advised to clone the returned layer stack after the fitting is complete.



Do not destroy the returned spectrum handle by calling `frlsDestroy()`! The internal layer stack is managed by the fitting context.

4.5.6 frfitGetScale()

```
int32_t frfitGetScale(const FitContext* fitContext, double* scale);
```

Get the fitting context's scale coefficients.

Parameters:

- `fitContext`: The fitting context.
- `scale`: Buffer to hold the scale coefficients.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE` and `scale` will hold the fitting context's scale coefficients.

If the function fails, check the return value for the reason of failure.

Remarks:

The `scale` array should be `FR_FIT_SCALE_EQU_NUM_PARAMS` long.

4.5.7 frfitGetOffset()

```
int32_t frfitGetOffset(const FitContext* fitContext, double* offset);
```

Get the fitting context's offset coefficients.

Parameters:

- `fitContext`: The fitting context.
- `offset`: Buffer to hold the offset coefficients.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE` and `offset` will hold the fitting context's offset coefficients.

If the function fails, check the return value for the reason of failure.

Remarks:

The `offset` array should be `FR_FIT_OFFSET_EQU_NUM_PARAMS` long.

4.5.8 frfitGetAngleOfIncidence()

```
double frfitGetAngleOfIncidence(const FitContext* fitContext);
```

Get the fitting context's angle of incidence.

Parameters:

- fitContext: The fitting context.

Return value:

The angle of incidence in radians.

Remarks:

None.

4.5.9 frfitCalcTheoreticalSpectrum()

```
int32_t frfitCalcTheoreticalSpectrum(FitContext* fitContext, double* spectrum);
```

Calculate the theoretical spectrum from the current parameters of the fitting context.

Parameters:

- `fitContext`: The fitting context.
- `spectrum`: Buffer to hold the calculated normalized theoretical spectrum.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE` and `spectrum` will hold the calculated normalized theoretical spectrum.

If the function fails, check the return value for the reason of failure.

Remarks:

`spectrum` should be `FitConfig::m_NumWavelengths` long. The calculated values correspond to the wavelengths passed during the creation of the fitting context.

4.5.10 frfitGetFrequencySpectrum()

```
int32_t frfitGetFrequencySpectrum(const FitContext* fitContext, double* freq, double* mag, uint32_t n);
```

Get the frequency spectrum from the fitting context.

Parameters:

- `fitContext`: The fitting context.
- `freq`: Fourier frequencies
- `mag`: Fourier magnitudes
- `n`: The number of frequencies to retrieve.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of failure.

Remarks:

If `FitConfigEx::m_Method` was set to `FR_FIT_METHOD_LEAST_SQUARES` when the fitting context was created, the function will fail.

If `n` is not equal to `FitConfigEx::FitConfigFourier::m_FreqSpectrumNumPoints` the function will fail.

Note that the frequencies returned by this function have been converted to thicknesses using the effective refractive index of the 1st layer. Also note that the frequency spectrum returned might not be exactly the same as the frequency spectrum used to determine layer thicknesses during fit. The frequency spectrum returned by this function is the frequency response in the specified thickness range (`FitConfigEx::FitConfigFourier::m_ThicknessRange`) which might not be the same frequency range used to determine layer thicknesses from peaks.

4.6 Color Analysis

The Color Analysis module includes functions for calculating the apparent color of spectra.

4.6.1 frCreateColorAnalysisContext()

```
ColorAnalysisContext* frCreateColorAnalysisContext(const ColorAnalysisConfig* cfg);
```

Creates a new Color Analysis context with the specified configuration.

Parameters:

- `cfg`: The configuration for the new context.

Return value:

If the function succeeds the return value will be an opaque pointer to the newly created context.

If the function fails, a NULL pointer will be returned.

Remarks:

The specified color analysis configuration is copied to an internal object. The developer is free to allocate the `ColorAnalysisConfig` object on the stack or free the heap-allocated object after a successful call to this function.

The `ColorAnalysisConfig` struct is shown in the snippet below along with the meaning of each field.

```
struct ColorAnalysisConfig
{
    // Array of wavelengths over which color analysis will be performed.
    const double* m_Wavelengths;

    // The tempereture (in Kelvin) of the illuminant.
    double m_IlluminantTemperature;

    // The color matching function to use when calculating CIE XYZ values.
    uint32_t m_ColorMatchingFunc;

    // The total number of wavelengths in the m_Wavelengths array
    uint32_t m_NumWavelengths;
};
```

`m_Wavelengths` pointer should have `m_NumWavelengths` elements. It is cloned into an internal array so the developer is free to release the array after the function returns. Wavelengths should be in the range [380, 780] nm, otherwise the function will fail with an `FR_ERROR_INVALID_ARG` error.



The specified wavelength values should be in ascending order and should match the wavelengths of the spectrum passed in `frcolAnalyzeSpectrum()`.

`m_IlluminantTemperature` is the temperature of the illuminant, in Kelvin. `FR_ILLUM_TEMPERATURE_XXX` macros define some temperature values for standard illuminants (e.g. `FR_ILLUM_TEMPERATURE_D65` is the temperature of a D65 illuminant). Illuminant temperature should be in the range [4000, 25000] K, otherwise the function will fail with an `FR_ERROR_INVALID_ARG` error. The Spectral Power Distribution (SPD) of the illuminant is calculated assuming a D-type illuminant.

`m_ColorMatchingFunc` is the color matching function that will be used to calculate the XYZ tristimulus values of the spectrum and the illuminant. `FR_CMF_1931_2DEG` corresponds to a 2° observer and `FR_CMF_1964_10DEG` corresponds to a 10° observer.

4.6.2 frDestroyColorAnalysisContext()

```
void frDestroyColorAnalysisContext(ColorAnalysisContext* ctx);
```

Destroy a color analysis context.

Parameters:

- ctx: The context to destroy.

Return value:

None.

Remarks:

After the function returns, no further calls to [frColAnalyzeSpectrum\(\)](#) should be performed using this context.

4.6.3 frcolAnalyzeSpectrum()

```
int32_t frcolAnalyzeSpectrum(ColorAnalysisContext* ctx, const double* spectrum,  
ColorAnalysisResult* res)
```

Analyze the specified spectrum.

Parameters:

- `ctx`: The context with which analysis will be performed.
- `spectrum`: The normalized spectrum to analyze.
- `res`: The analysis results.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails, check the returned value for the reason of the failure.

Remarks:

`spectrum` elements should correspond to the `wavelengths` array specified during context creation. All internal values have been calculated using these wavelengths, so if there is a mismatch, invalid results will be returned.



The number of elements in the `spectrum` array should be equal to `ColorAnalysisConfig::m_NumWavelengths` value specified during context creation. In case the array does not hold enough elements, the API will probably crash.

The `ColorAnalysisResult` struct is shown in the snippet below along with the meaning of each field.


```

struct ColorAnalysisResult
{
    // The illuminant's XYZ tristimulus values.
    double m_IlluminantXYZ[3]; // { X, Y, Z }

    // The spectrum's XYZ tristimulus values.
    double m_XYZ[3];          // { X, Y, Z }

    // CIE La*b* values
    double m_CIELab[3];       // { L, a*, b* }

    // CIE LCH values
    double m_CIELCH[3];      // { L, C, H }

    // Hunter Lab values
    double m_HunterLab[3];    // { L, a, b }

    // Normalized sRGB values.
    double m_RGB[3];          // { R, G, B }

    // Chromaticity coordinates
    double m_xy[2];           // { x, y }

    // Yellowness Index
    double m_YellowIndex;

    // Total Solar Reflectance
    double m_TotalSolarRefl;
};

```

`m_IlluminantXYZ` does not depend on the specified spectrum but it is included in the results for the developer's convenience.

CIE XYZ tristimulus values are calculated by integrating the specified spectrum using the specified illuminant and color matching function over the supplied wavelength list.

CIE La*b*, Hunter Lab, RGB, chromaticity and Yellowness Index values are calculated from the CIE XYZ values. CIE LCH values are calculated from the CIE La*b* values.

Yellowness Index calculations assume a D65 illuminant.

4.7 FFT

The FFT module includes functions for calculating the Discrete Fourier Transform (DFT) of a spectrum, using the Fast Fourier Transform algorithm. This might be useful for estimating the thickness of an unknown layer under certain circumstances.

4.7.1 frCreateFFTContext()

```
FFTContext* frCreateFFTContext(const FFTConfig* cfg);
```

Creates a new FFT context with the specified configuration.

Parameters:

- `cfg`: The configuration for the new context.

Return value:

If the function succeeds the return value will be an opaque pointer to the newly created context.

If the function fails, a NULL pointer will be returned.

Remarks:

The specified FFT configuration is copied to an internal object. The developer is free to allocate the `FFTConfig` object on the stack or free the heap-allocated object after a successful call to this function.

The `FFTConfig` struct is shown in the snippet below along with the meaning of each field.

```

struct FFTConfig
{
    // Array of wavelengths over which the DFT will be calculated.
    const double* m_Wavelengths;

    // A refractive index value for converting frequencies into thicknesses.
    double m_RefrIndex;

    // The total number of wavelengths in the m_Wavelengths array.
    uint32_t m_NumWavelengths;

    // The size of the DFT.
    // NOTE: The meaning of this field changed in v1.9.0.140. The name is kept for backwards
compatibility. It will be renamed in v2.x
    // See explanation below.
    uint32_t m_FFSize;

    // Window size for the peak detection algorithm.
    // DEPRECATED: Not used in v1.9.0.140. It has been kept for backwards compatibility. It will be
removed in v2.x
    uint32_t m_PeakDetectionWindowSize;

    // Threshold for the peak detection algorithm.
    // NOTE: The meaning of this field changed in v1.9.0.140. The name is kept for backwards
compatibility. It will be renamed in v2.x
    // See explanation below.
    double m_PeakDetectionThreshold;
};

```

`m_Wavelengths` pointer should have `m_NumWavelengths` elements. It is cloned into an internal array so the developer is free to release the array after the function returns.




The specified wavelength values should be in ascending order and should match the wavelengths of the spectrum passed in `frfftAnalyzeSpectrum()`.

`m_RefrIndex` is the real part of the refractive index which will be used to convert Fourier frequencies into layer thicknesses. See `frmCalcEffectiveRefrIndex()` and `frutilCalcEffectiveRefrIndex()` for a way to estimate the refractive index of a material for frequency-to-thickness conversions.

`m_FFSize` determines the total number of points in the generate frequency spectrum.

`m_PeakDetectionWindowSize` and `m_PeakDetectionThreshold` are parameters for the peak detection algorithm. The peak detection algorithm has been changed in v1.9.0.140 and `m_PeakDetectionWindowSize` is not used anymore. `m_PeakDetectionThreshold` is the peak threshold relative ([0,1]) to the maximum of the

frequency spectrum. Using 0 as the `m_PeakDetectionThreshold` will instruct the algorithm to use the average of the frequency spectrum as the threshold. Any value other than 0 is used as is.

 The frequency spectrum is computed using Type-III Non-Uniform Discrete Fourier Transform (NUDFT-III). The maximum frequency is the frequency of the largest peak multiplied by 4.

4.7.2 frDestroyFFTContext()

```
void frDestroyFFTContext(FFTContext* ctx);
```

Destroys an FFT context.

Parameters:

- ctx: The context to destroy.

Return value:

None.

Remarks:

After the function returns, the context should be considered invalid and not be used anymore.

4.7.3 frfftAnalyzeSpectrum()

```
int32_t frfftAnalyzeSpectrum(FFTContext* ctx, const double* spectrum);
```

Calculates the DFT of the specified normalized spectrum.

Parameters:

- `ctx`: The FFT context.
- `spectrum`: The normalized spectrum.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of failure.

Remarks:

`spectrum` elements should correspond to the `wavelengths` array specified during context creation. All internal values have been calculated using these wavelengths, so if there is a mismatch, invalid results will be returned.



The number of elements in the `spectrum` array should be equal to `FFTConfig::m_NumWavelengths` value specified during context creation. In case the array does not hold enough elements, the API will probably crash.

All results (frequency spectrum and peaks) will be valid until the function is called again with a new spectrum.

4.7.4 frfftGetFrequencySpectrum()

```
int32_t frfftGetFrequencySpectrum(const FFTContext* ctx, double* freq, double* mag, uint32_t n);
```

Get the frequency spectrum calculated from the last call to [frfftAnalyzeSpectrum\(\)](#).

Parameters:

- `ctx`: The FFT context
- `freq`: Fourier frequencies
- `mag`: Fourier magnitudes
- `n`: The maximum number of samples to retrieve.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of failure.

Remarks:

The size of the frequency spectrum is internally equal to `FFTConfig::m_FFTSize / 2`.

- If `n` is less than that, only the first `n` elements will be returned.
- If `n` is greater than that, only the first `FFTConfig::m_FFTSize / 2` elements will be filled.

`freq` and `mag` arrays should be large enough to hold `n` elements.

Depending on the value of `FFTConfig::m_RefrIndex` specified during context creation, the data returned in the `freq` array might be interpreted as layer thicknesses.

4.7.5 frfftGetPeaks()

```
int32_t frfftGetPeaks(const FFTContext* ctx, double* peakFreq, double* peakMag, uint32_t n);
```

Get the frequency and magnitude of all the peaks detected by the last call to [frfftAnalyzeSpectrum\(\)](#).

Parameters:

- `ctx`: The FFT context.
- `peakFreq`: The frequency of each peak.
- `peakMag`: The magnitude of each peak.
- `n`: The maximum number of peaks to retrieve.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails, check the return value for the reason of failure.

Remarks:

`n` specifies the size of the `peakFreq` and `peakMag` arrays.

- If `n` is less than the number of peaks detected (`N`), then only the first `n` peaks will be returned.
- If `n` is greater than the number of peaks detected (`N`), then only the first `N` elements will be filled.

4.7.6 frfftGetNumPeaks()

```
uint32_t frfftGetNumPeaks(const FFTContext* ctx);
```

Get the number of peaks detected by the last call to [frfftAnalyzeSpectrum\(\)](#).

Parameters:

ctx: The FFT context.

Return value:

The number of peaks detected.

Remarks:

None.

4.8 Peripherals

The Peripherals module includes functions for controlling all peripheral devices connected to the main FR tool.

Peripherals include lamps, shutters, X/Y or R/ θ stages and others. Not all peripherals are available to each FR tool.

4.8.1 frperGet()/frperSet()

```
int32_t frperSetBoolean(PeripheralHandle per, uint16_t param, bool value);
int32_t frperSetUint(PeripheralHandle per, uint16_t param, uint32_t value);
int32_t frperSetObject(PeripheralHandle per, uint16_t param, const void* value, uint32_t sz);
int32_t frperGetBoolean(PeripheralHandle per, uint16_t param, bool* value);
int32_t frperGetUint(PeripheralHandle per, uint16_t param, uint32_t* value);
int32_t frperGetString(PeripheralHandle per, uint16_t param, char* str, uint32_t sz);
int32_t frperGetObject(PeripheralHandle per, uint16_t param, void* value, uint32_t sz);
```

Parameters:

- per: The peripheral handle obtained using [frdevGetPeripheral\(\)](#)
- param: The peripheral parameter (see the Remarks section for details)
- value: The value of the parameter
- sz: The size of the buffer pointed by the value.

Return value:

If the function succeeds the return value will be FR_ERROR_NONE.

If the function fails, check the returned value for the reason of the failure.

Remarks:

See the Peripherals document for details on the parameters available to each peripheral device.

4.8.2 frperSendCommand()

```
int32_t frperSendCommand(PeripheralHandle per, uint16_t cmd, const void* buffer, uint32_t sz);
```

Send a command to the specified peripheral.

Parameters:

- per: The peripheral handle obtained using [frdevGetPeripheral\(\)](#)
- cmd: The command ID.
- buffer: Command data buffer.
- sz: The size of the command data buffer.

Return value:

If the function succeeds the return value will be `FR_ERROR_NONE`.

If the function fails, check the returned value for the reason of the failure.

Remarks:

See the Peripherals document for details on the commands available to each peripheral device.

4.9 Utilities

The Utilities module includes functions for processing spectra, calculating refractive index values and recording of experiments in ThetaMetrisis' ex2 file format.

4.9.1 frCreateExperimentWriter()

```
ExperimentWriter* frCreateExperimentWriter(const char* filename, const double* wavelengths,  
uint32_t n);
```

Create a new experiment file writer.

Parameters:

- `filename`: The absolute path to the output file.
- `wavelengths`: Array of wavelengths.
- `n`: The number of wavelengths in the `wavelengths` array.

Return value:

If the function succeeds, the return value will be an opaque pointer to the new experiment writer.

If the function fails, the returned value will be the `NULL` pointer.

Remarks:

The developer is responsible for destroying the experiment writer using [frDestroyExperimentWriter\(\)](#) when it is not needed anymore.

4.9.2 frDestroyExperimentWriter()

```
void frDestroyExperimentWriter(ExperimentWriter* writer);
```

Close an experiment writer.

Parameters:

- `writer`: Experiment writer previously obtained by calling [frCreateExperimentWriter\(\)](#).

Return value:

None.

Remarks:

Because the number of recorded spectra isn't known when the experiment writer is created, properly closing it using this function is mandatory in order to write the total recorded spectra at the correct file location. Failing to do so will produce a corrupted ex2 file.

4.9.3 frexpWriteSpectrum()

```
int32_t frexpWriteSpectrum(ExperimentWriter* writer, const double* samples, uint32_t n, float time_sec);
```

Write a new spectrum.

Parameters:

- `writer`: Experiment writer previously obtained by calling [frCreateExperimentWriter\(\)](#).
- `sample`: The samples of the new spectrum.
- `n`: The number of samples in the `samples` array.
- `time_sec`: A timestamp for the new spectrum.

Return value:

If the function succeeds, the return value will be `FR_ERROR_NONE`.

If the function fails, check the returned value for the reason of failure.

Remarks:

The number of samples `n` must be equal to the number of wavelengths passed during the creation of the experiment writer.

4.9.4 frutilNormalizeSpectrum()

```
void frutilNormalizeSpectrum(double* res, const double* spectrum, const double* ref, const double* dark, uint32_t n);
```

Calculate the normalized spectrum from the provided raw, reference and dark spectra.

Parameters:

- **res:** Buffer to hold the resulting normalized spectrum.
- **spectrum:** The raw spectrum as acquired by a device.
- **ref:** The reference spectrum.
- **dark:** The dark spectrum.
- **n:** The number of elements in all of the above arrays.

Return value:

None.

Remarks:

None.

4.9.5 frutilCalibrateReferenceSpectrum()

```
void frutilCalibrateReferenceSpectrum(uint32_t refCalMode, double* res, const double* wavelengths,  
const double* ref, const double* dark, uint32_t n);
```

Calibrate the specified reference spectrum using the specified calibration mode.

Parameters:

- `refCalMode`: The calibration mode.
- `res`: Buffer to hold the resulting spectrum.
- `wavelengths`: Buffer with the wavelengths corresponding to each sample of the ref and dark spectra.
- `ref`: The uncalibrated reference spectrum as acquired by a device.
- `dark`: The dark spectrum.
- `n`: The number of elements in all of the above arrays.

Return value:

None.

Remarks:

`refCalMode` can take one of the following values:

- `FR_REFCAL_NONE`: No calibration is performed.
- `FR_REFCAL_SILICON`
- `FR_REFCAL_BK7`
- `FR_REFCAL_ALUMINUM`
- `FR_REFCAL_AL2O3`
- `FR_REFCAL_PET`
- `FR_REFCAL_STAINLESS_STEEL`

4.9.6 frutilCalcRefrIndex()

```
int32_t frutilCalcRefrIndex(double* n, double* k, const double* wavelengths, uint32_t
numWavelengths, uint32_t riType, const double* params, uint32_t numParams);
```

Calculate the refractive index at the specified wavelengths from its parameters.

Parameters:

- **n**: Buffer to hold the real part of the refractive index.
- **k**: Buffer to hold the imaginary part of the refractive index.
- **wavelengths**: The list of wavelengths over which the refractive index will be calculated.
- **numWavelengths**: The total number of wavelengths in the **wavelengths** array.
- **riType**: The refractive index equation.
- **params**: The refractive index parameters.
- **numParams**: The number of refractive index parameters.

Return value:

If the function succeeds, the return value will be `FR_ERROR_NONE` and arrays **n** and **k** will hold the real and imaginary part of the refractive index respectively.

If the function fails, check the return value for the reason of failure.

Remarks:

Both **n** and **k** are calculated and both arrays should be valid and at least **numWavelengths** long.

Check the [Materials](#) section for details on the number of parameters required for each refractive index equation.

4.9.7 frutilCalcEffectiveRefrIndex()

```
double frutilCalcEffectiveRefrIndex(const double* wavelengths, uint32_t numWavelengths, uint32_t riType, const double* params, uint32_t numParams)
```

Calculate the Effective Refractive Index from the refractive index parameters at the specified wavelength range.

Parameters:

- `wavelengths`: The list of wavelengths over which the effective refractive index will be calculated.
- `numWavelengths`: The total number of wavelengths in the `wavelengths` array.
- `riType`: The refractive index equation.
- `params`: The refractive index parameters.
- `numParams`: The number of refractive index parameters.

Return value:

If the function succeeds, the return value will be the effective refractive index.

If the function fails, the return value will be $\emptyset . \emptyset$.

Remarks:

The Effective Refractive Index is useful as an estimation of a material's refractive index when converting frequencies to thicknesses during frequency analysis (see [FFT](#) section).

Check the [Materials](#) section for details on the number of parameters required for each refractive index equation.