

API Reference Manual
75-0048

libtrf - thinkRF devices API

API Reference Manual

version 1.9.0

Thu Jul 23 2026

Contents

1	Introduction	1
1.1	Software and System Requirements	1
1.1.1	Supported RTSA Product List	1
1.2	Using the API	2
1.2.1	Using libtrf on Windows	2
1.2.2	Using libtrf on Linux	2
1.2.3	Using Other Language Bindings	2
1.3	Quick Start with Examples	2
1.3.1	Building Example Code on Windows	3
1.3.2	Building Example Code on Linux	3
1.4	Contact Us	3
2	libtrf	5
2.1	Use of JSON for Parameters	5
2.2	Devices, Streams, Processors	6
2.2.1	Handles	6
2.2.2	Parameters	6
2.2.3	Devices	7
2.2.4	Streams	8
2.2.5	Stream Files	13
2.2.6	Processors	14
2.3	Memory considerations	16
2.3.1	IQ capture	17
2.3.2	Spectrum capture and Memory footprint	17
2.3.3	SpectrumCharacterization processor	18
2.4	Programming Antenna, Equalization and Switching	18
2.4.1	Antenna Factor tables	18
2.4.2	Equalization tables	18
2.5	libtrf Structure	19
2.5.1	Interpreting error codes	19
2.5.2	Error Handling and Diagnostic Functions	20

2.5.3	Device Handling Functions	20
2.6	Stream and Stream Data Handling Functions	20
2.6.1	Stream Frame Handling Functions	20
2.6.2	Streams and Files Functions	21
2.7	Stream Attach/Detach and State Functions	21
2.7.1	Special File Handling Functions	21
2.8	Processor Handling Functions	21
2.8.1	Parameter Handling (JSON) Functions	21
2.9	The Rest of this document	23
3	Data Structure Index	25
3.1	Data Structures	25
4	File Index	27
4.1	File List	27
5	Data Structure Documentation	29
5.1	_complex32 Struct Reference	29
5.1.1	Detailed Description	29
5.1.2	Field Documentation	29
5.2	_trfBasebandFrame Struct Reference	30
5.2.1	Detailed Description	30
5.2.2	Field Documentation	30
5.3	_trfIQFrame Struct Reference	31
5.3.1	Detailed Description	32
5.3.2	Field Documentation	32
5.4	_trfSpectrumFrame Struct Reference	34
5.4.1	Detailed Description	34
5.4.2	Field Documentation	35
6	File Documentation	37
6.1	libtrf.h File Reference	37
6.1.1	Macro Definition Documentation	51
6.1.2	Typedef Documentation	72
6.1.3	Enumeration Type Documentation	74
6.1.4	Function Documentation	78
6.2	libtrf_doxy.txt File Reference	111
Index		113

Chapter 1

Introduction

The thinkRF Real-Time Spectrum Analyzer (RTSA) has the performance of traditional high-end lab spectrum analyzers at a fraction of the cost, size, weight and power consumption and is designed for standalone, distributed or remote deployment.

thinkRF RTSA devices provide a low-level interface via SCPI and VITA49 protocols. Whilst useful in some scenarios, for many applications a higher-level interface is required that allows users to obtain high performance from the RTSA without being exposed to low-level protocol and programming details. Furthermore, future thinkRF RTSA and other devices will increasingly make use of other interfacing technologies where open protocols are less accessible or inconvenient for end-users.

To enhance user experience and provide a future-proof path forwards for users of current and future thinkRF equipment, the *libtrf* API has been developed. The *libtrf* library provides a one-stop programmer-friendly, cross-platform, expressive and powerful common interface to thinkRF devices with highly optimized performance. In addition, it provides common and advanced signal processing functions handy for end-users who wish to perform further analysis of captured signals.

The *libtrf* API is provided as a dynamic library (.dll/.lib for windows, .so for linux environments) with 'C' language bindings. thinkRF currently provides a python binding based on the 'C' shared library interface, called pyRF4.

1.1 Software and System Requirements

To use *libtrf* in your application, the following is required:

- Operating System:
 - Windows 10/11 64-bit operating system (use *libtrf.dll*, *libtrf.lib*)
 - Linux x86_64 (Intel/AMD) or aarch64 (ARM v8+). Ubuntu 20.04LTS or later is recommended (use *libtrf.so*)
- Access to an RTSA product. If you do not have access to a unit, you may request access to a demo unit by contacting support@thinkrf.com.

1.1.1 Supported RTSA Product List

libtrf version 1.6.1 and higher supports the following RTSA products and their associated models (-408, -408P, -418, -427):

- R5500
- R5550

- R5700
- R5750
- R6000

For prior-generation thinkRF receivers, the legacy APIs are available. The legacy APIs, however, are not recommended for future development.

1.2 Using the API

The API provides a 'flat' C-language interface to all functionality. The functions are published via a single header file called [libtrf.h](#) and are implemented via a single self-contained shared library binary `libtrf.[so|dll]`. Refer to [Software and System Requirements](#) section for specifications & recommendation.

Example of how to link with `libtrf` on Windows and Linux are provided in the included 'Examples' directory. In particular, see the 'README.txt' file in that directory.

1.2.1 Using libtrf on Windows

To use `libtrf` directly in your application, you should include the '[libtrf.h](#)' header file. Linking your C/C++ application with the supplied '`libtrf.lib`' will provide the required stub functions (for the [libtrf.h](#) header) to access the shared library functions and will provide for start-time. Normally, it is recommended that `libtrf.dll` be placed in the same directory as the your application binary to facilitate runtime loading.

1.2.2 Using libtrf on Linux

To use `libtrf` directly in your application, you should include the '[libtrf.h](#)' header file. The linker option '-ltrf' should be specified to cause the linker to reference `libtrf.so` - which should be available via the `LD_LIBRARY_PATH` environment variable. Note that for linking from the same directory as your binary file, executing '`export LD_LIBRARY_PATH=$↵LD_LIBRARY_PATH:.`' is typically sufficient.

1.2.3 Using Other Language Bindings

Currently, only a python binding called `pyrf4` is developed for python users. Its capabilities might be lagging behind those of `libtrf`; however, it is an open-source code and thus, is free for python end-users to modify or update as see fit. See <https://thinkrf.com/software-apis/> or reach out to support@thinkrf.com for further information.

1.3 Quick Start with Examples

The easiest way to get started with the API is to take a look at the rich set of examples provided in the "Examples" folder included with the API release package. The examples include different capture modes (spectrum sweeping, IQ capture, finite-time capture, capture with file, etc) along with build instructions for Windows and Linux environments.

The example code is heavily commented and it is recommended that you make use of this code to understand the functionality of the `libtrf` library calls and how they interact.

1.3.1 Building Example Code on Windows

1. Assuming Visual Studio is installed; in the windows start area, navigate to the install directory for your version of Visual Studio (ie. Visual Studio 2017). From there select the 'x64 Native Tools Command Prompt'.
2. In the command window that is launched, navigate to the 'examples' directory of your libtrf installation. To build any of the examples, enter (for example):
`cl CaptureToFile.cpp ../bin/x64/libtrf.lib -I ../source/inc /link /out:example.exe`
This will generate example.exe in your current directory.
3. To enable run-time linking with the libtrf shared library, libtrf.dll should be copied into the current directory:
`copy "..\bin\x64\libtrf.dll" .`
4. Type example.exe to run the example application code.

1.3.2 Building Example Code on Linux

1. Assuming that gcc/g++ is installed (version 10 or later recommended). To build any of the examples, the LD_LIBRARY_PATH environment variable should be correctly set to reference libtrf. For example, if your installation is in your home directory then in a shell you can type the following to correctly set LD_LIBRARY_PATH for build/execution with libtrf:
`export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:~/libtrf/bin/_arch_`
In the above substitute *arch* for either 'x86_64' or 'aarch64' as required. Note that this command is required every time you open a new shell to build with libtrf.
2. Build the example using the command line (for example):
`g++ CaptureToFile.cpp -o example -I ../source/inc -L ../bin/linux/x86_64 -ltrf -lpthread`
3. Type ./example to run the example application code.

1.4 Contact Us

thinkRF Support website provides online documents for resolving technical issues with thinkRF products at <https://thinkrf.com/resources>.

For all customers who hold a valid end-user license, thinkRF provides technical assistance 9 AM to 5 PM Eastern Time, Monday to Friday. Contact us at <https://support.thinkrf.com/>

© 2026 thinkRF Corporation, Ottawa, Canada, www.thinkrf.com.

Trade names are trademarks of the owners.

These specifications are preliminary, non-warranted, and subject to change without notice.

Chapter 2

libtrf

The distribution of libtrf comprises this programmers' Reference manual, a C header file and architecture-specific binary release files of the shared library.

The release package (libtrf_<version>.zip / libtrf_<version>.tar.gz) unpacks into the following structure beneath the root 'libtrf' directory:

- doc/
 - Release Notes pdf - *plaintext release notes*
 - Reference Manual pdf - *this document*
- examples/ - *source cpp code examples directory*
- inc/
 - [libtrf.h](#) - *library header file*
- bin/
 - win/x64/ - *windows binary files*
 - * libtrf.lib
 - * libtrf.dll
 - linux/
 - * x86_64/ - *64-bit x86 architecture binaries*
 - libtrf.so
 - * aarch64/ - *64-bit ARMv8+ binaries*
 - libtrf.so

2.1 Use of JSON for Parameters

A design challenge for any library with the scope of libtrf is to provide a means of accessing parameters from a set that may be modified over time for different devices, streams, processor types and in subsequent releases of the library.

To prevent an explosion of library calls to manage this parameter space, libtrf instead uses the well-known JSON (JavaScript Object Notation) open standard format (see <https://datatracker.ietf.org/doc/html/rfc8259> - RFC8259 for details).

The library reports parameter information using JSON notation and allows parameters to be examined and modified by constructing and supplying JSON strings. The library provides utility functions for the creation and management

of JSON strings where this is not supported by user code. Also, syntactic-sugar is provided in the form of 'canned' functions, which allow easy encoding of common parameters and ensure that typically-used parameters are included directly where needed.

2.2 Devices, Streams, Processors

To make best use of libtrf, it is useful to understand the core abstractions that the library is based on. This model comprises three primary classes of entities; devices, streams and processors. Collectively these are referred to as libtrf 'entities'.

Entities are typically created when needed by a user, manipulated by setting and reading parameters, and may be connected to produce user-desired processing structures.

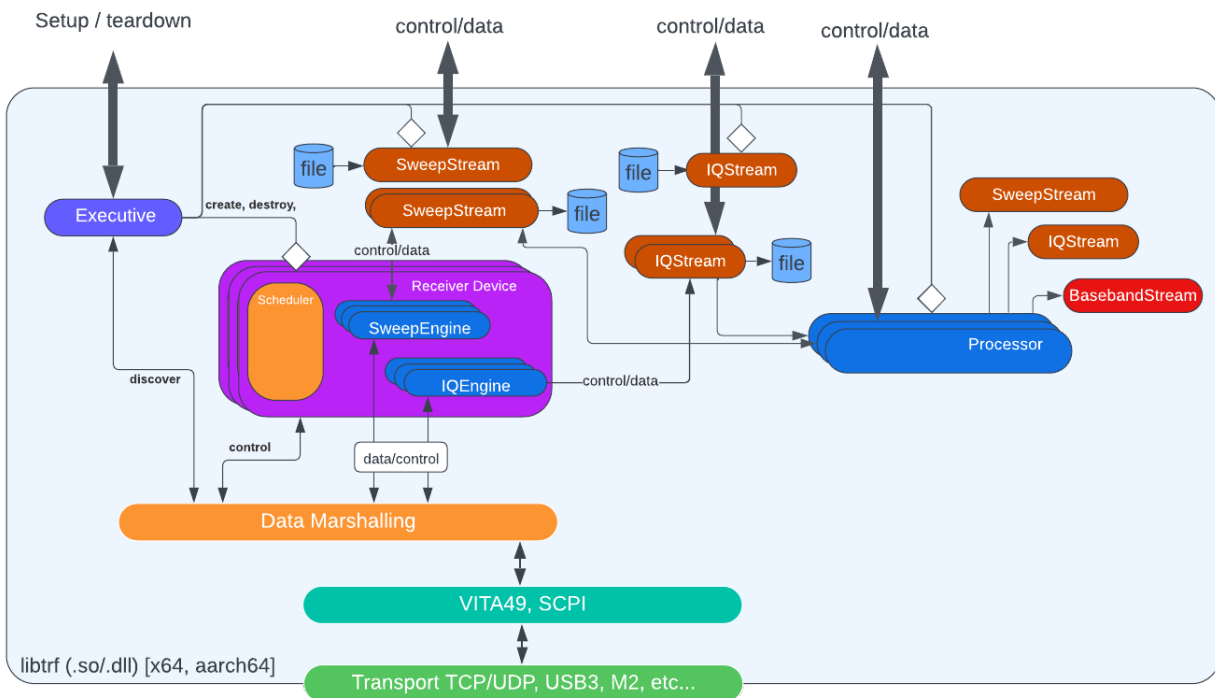


Figure 2.1 libtrf architecture

2.2.1 Handles

All entities within *libtrf* are referred to using a 'handle' (type `trfHandle`). The value of a handle is created when its matching entity is created, and its value becomes invalid when that entity is destroyed. *libtrf* supports three major entity types; Devices, Streams and Processors.

2.2.2 Parameters

All entities are controlled through modifying their parameters. All entities have documented parameters. Specialized entities, such as different types of processor, may have extended parameter sets. Parameters are referred to by names, which are defined in '[libtrf.h](#)' using macro constants for ease of reference. As noted above, the JSON notation scheme used for parameters allows for the expansion of the behaviour of *libtrf* without modifying the core API.

2.2.3 Devices

A device provides network-controllable functionality over some transport layer. Typically this is TCP/IP and, with the introduction of the R6xx0 units, USB3. After opening a device, the network supporting device functionality is transparent to a *libtrf* user.

A device may be a source of data of different kinds (ie. spectrum information, digitized radio (IQ) data). Also, it is possible that a device does not source radio data directly and implements other functionality (such as an up-converter). A device could also consume radio data (ie. a transmitter).

In the current version of *libtrf*, the only device type supported is an RTSA (ie. receiver); however, the design of the library permits other device types. Support for new devices will be added in subsequent releases of *libtrf*.

Device Parameters

The following is a list of parameters for controlling RTSA receivers via [trfSetParameters](#):

- [TRFNTPT](#) - Sets NTP server.
- [TRFLANMTU](#) - Sets LAN MTU (Maximum transmission unit).
- [TRFCurrentDate](#) - Sets receiver date.
- [TRFCurrentTime](#) - Sets receiver time.
- [TRFPLLSource](#) - Sets PLL source.
- [TRFPPSSource](#) - Sets PPS source.
- [TRFTimeSync](#) - Sets time sync.

If the receiver has a GNSS module, (check [TRFHasGNSS](#) parameter from [trfGetParameters](#)), the available parameters for configuration are:

- [TRFGNSSADelay](#) - Sets Antenna delay.
- [TRFGNSSConstellation](#) - Sets GNSS constellation to be tracked. format:
`<option 1>[,option 2]`
. Option 2 is not required. Possible Constellation options: GPS, BEIDOU and GLONASS.
- [TRFGNSSDynamicMode](#) - Sets GNSS Dynamic mode. Possible options: portable, stationary, pedestrian, automotive, atSea, airborne<1g, airborne<2g and airborne<4g.

Device parameters (options and properties) can be queried using [trfGetParameters](#), which returns parameters above as well as the ones shown below:

- [TRFRxSampleRate](#) - Receiver sample rate.
- [TRFHasFlatteningInfo](#) - Receiver flag that equalization data is available (flattening is possible).
- [TRFSweepActive](#) - sweep is active flag (Rx).
- [TRFIQActive](#) - IQ capture is active flag (Rx).
- [TRFDeviceTemperature](#) - Device temperature.
- [TRFHasGNSS](#) - Flag if receiver has GNSS module or not.
- [TRFDevice](#) - Returns a JSON string specifying device-specific parameters:

- [TRFDeviceType](#) - Device type name (ie. R5750)
- [TRFDeviceVersion](#) - Textual version of the hardware
- [TRFDeviceSerialNum](#) - Textual serial number of the device
- [TRFDeviceFirmware](#) - Textual firmware version of the device
- [TRFDeviceConnection](#) - Device connection information
- [TRFHasFlatteningInfo](#) - Flag to indicate whether the receiver has internal compensation tables populated

GNSS specific params:

- [TRFGNSSTimestamp](#) - Most recent GNSS UTC timestamp in nanosec.
- [TRFGNSSRxTime](#) - Last position report in UTC Msec.
- [TRFGNSSValid](#) - true if GNSS data is valid, false otherwise.
- [TRFLatitude](#) - Latitude information.
- [TRFLongitude](#) - Longitude information.
- [TRFAltitude](#) - Altitude information.
- [TRFGNSSSpeedOverGround](#) - Speed over ground.
- [TRFGNSSHeadingDegT](#) - Current heading in degrees.
- [TRFGNSSTrackDegT](#) - Track in degrees.
- [TRFGNSSMagVarDegT](#) - Magnetic variation in degrees E+,W-

A detailed list of parameter information and libtrf/receiver default values can be fetched via [trfGetParameterInfo\(\)](#).

2.2.4 Streams

A stream represents a time-based sequence of data. A user may access a stream to recover sampled data from a receiver device. Also, streams may write to and be generated from files. A stream may contain any type of data.

The libtrf library recognizes three types of streams: Spectrum, IQ and Baseband.

As well as a conduit of data, streams have an important control function. A receiver device can only provide data through a stream. Obtaining data involves creating a suitable stream to hold that data, and **attaching** it to a source device. If the attach is successful, the receiver device will start and continuously deliver data into the stream until the stream is detached from the receiver device. Note that functionality is provided to allow streams to auto-detach once a certain amount of data (time, frames) has been obtained, as well as for them to continue indefinitely.

A high-performance receiver can produce data at very high rates. In particular the R5xx0-family support an internal IQ sampling rate of 62.5Msamples per second, representing a raw uncompressed IQ data rate of about 500MB/sec. For R6xx0-family devices this rate increases to ~1GB/sec. For this reason, streams incorporate buffering. This buffering allows client applications to achieve continuous IQ data processing without discontinuities in the face of operating system and network delays and jitter. It should be noted that **continuous** streaming is only possible where the underlying network bandwidth is sufficient to maintain the required streaming data rate.

At the same time, this buffering is constrained to ensure that libtrf maintains a reasonable memory footprint even under high processing loads. The depth of buffering in a stream defaults to a small value (10 frames) and may be programmed by the user in terms of frames ([TRFBufferFrames](#) - most suitable for spectrum data) or time ([TRFBufferSec](#) - most suitable for IQ or baseband data) to suit application and target system requirements.

Stream Buffer overflow handling

When a buffer overflow occurs, libtrf restores the buffer limit by discarding the oldest data to restore the buffer size within programmed limits. For IQ data, this will introduce discontinuities into the stream of IQ data. By default, the minimum data required will be discarded. When operating close to performance limits this may generate a stream of frequent discontinuities. To minimize the number of such discontinuities, the [TRFDiscardProportion](#) parameter can be set. This defaults to 0.0f (minimum discarding) and can be set in the range 0.0f to 1.0f where 1.0f denotes on a buffer overflow, the entire buffer is discarded. Using this parameter the tradeoff between data loss and discontinuity rate can be optimized for the application.

If a buffer size limit is exceeded, the stream will discard the oldest data up to some 'discard proportion' limit set by the parameter [TRFDiscardProportion](#). If set to zero (default), the stream will discard only the oldest data required to bring the buffer size back within limits. Values up to 1.0 allow the proportion of the current buffer to be discarded in the case of an overrun to be controlled, up to 100% of the buffer. This allows a user to, for example minimize the number of discontinuities in received data by performing large and rare discards rather than small and frequent discards when required.

Stream functionality and control are illustrated in several key scenarios in the set of examples included with the release. These examples provide a rich illustration of the use of stream features.

Advanced stream properties

As streams program the behaviour of a device to which they are attached, special parameters are provided that control the collection process that may be utilized by a source to which a stream is connected.

- [TRFFlattenFlag](#) - By default this flag is set 'true' and ensures that the additional processing required to optimize the flatness of the spectrum output is active when the stream is attached to a receiver that supports this functionality.
- [TRFDCOffsetFilter](#) - By default this flag is set to 'false'. When 'true', signal processing is applied to remove the impacts of residual DC in IQ data retrieved from the instrument when attached to a receiver that supports this functionality. Note that a residual DC offset in a sampled signal can result in a central 'peak' in sampled data (at 0Hz).

Non-discardable streams

The discarding of frames read from files is undesirable. To account for this usage, the streams that originate from files are marked as 'non-discardable'. If these stream buffers become full, it will result in the originating file read operation being blocked until the buffer is bought back within limits. This can lead to blocking where, for example a stream is being read from a file, processed and only the frames emerging from the processor are being consumed. To allow a user to control this case, the [trfAllowDiscarding\(\)](#) function is provided. This allows a user to override the non-discarding property for streams that the user has no intention of directly servicing, hence avoiding unintended blocking from occurring when processing frames from file.

Note that the 'non-discardable' property is only relevant to streams read from file and has no semantic effect for streams originating from a live receiver.

Shutdown with non-discardable streams When shutting down streams and processing chains sourced from a file, care should be taken to avoid deadlock preventing the completion of the shutdown. A simple rule is that when shutting down a processing chain from file, the most downstream resource should be closed first. The stalling condition that can result would be, for example a call to `trfClose(...)` not completing. This condition results from the file, if not exhausted, filling stream buffers up down the chain. If these frames are not consumed (ie. at the end of processing), the buffers along the chain may become filled which stalls the upstream sources until reaching the stream sourced from the file. By closing the most downstream resource first, the cyclic dependency in the chain is broken, allowing the shutdown sequence to complete without delays or stalls.

Stream status return values

The functions used to retrieve data from streams (`trfGetNext___`) return three special information values, describing the state of the stream when no data is returned:

- `trfNotStarted` - This indicates that no data has yet been retrieved from the stream, and data is expected.
- `trfWaiting` - Some data has been received, however the next frame is not yet available.
- `trfExhausted` - No more data is expected from the stream without a configuration change. This occurs when a stream has been detached from its source (device, file) and its internal buffering is now empty.

The occurrence of these return values is *not an error*, and instead can prompt user code into responding appropriately to the conditions they indicate. Note that the timeout parameter specified in the `trfGetNext___` calls can be set large (ie. 5000 msec). This has no impact on performance as the call will return as soon as data is available, however it will minimize the return of 'nuisance' `trfNotStarted`, `trfWaiting` values if this is important to user code. Note that a timeout of 0 will return immediately if no data is available, allowing a non-blocking design approach to be used.

Spectrum Streams

A spectrum stream consists of a sequence of 'frames' of spectrum data. Each frame represents a power-spectrum between a start and stop frequency, regularly sampled. The metadata describing a spectrum frame also details how it was created; specifically its resolution bandwidth and window type. Additional parameters include a timestamp and sequence number. Spectrum frames may also contain additional spectrum data (such as min, max, average) that may be added by processors.

Spectra are created through a process of successive steps of retuning and sampling from successive IF bandwidth chunks over the tuning range of the receiver. In this process all the details of tuning and sampling are abstracted from the user, providing a small set of expressive control parameters.

Spectrum capture parameters The following parameters control how a spectrum is produced:

- Start frequency (`TRFFMinHz`). The first frequency that must be represented in the returned spectrum
- Stop frequency (`TRFFMaxHz`). The last frequency that must be represented in the returned spectrum
- Resolution bandwidth (`TRFRBWHz`). The resolution of the returned spectrum. Note that this is *not* the same as the frequency step between points returned.
- Video bandwidth (`TRFVBWHz`). The dynamic time-filtering bandwidth of the returned spectrum points.
- Dwell time (`TRFDwellUsec`). Specifies the rate at which the spectrum is to be sampled in microseconds per MHz. Increased sampling time allows refinement of output for different detector types.
- Window function (`TRFWindow`). Specifies the sampling window type to be used. Supported types are: "rectangular", "gaussian", "hann", "hamming", "cosine", "lanczos", "bartlett", "bartlettthann", "nuttall"(default), "blackmanharris", "blackmannuttall", "flattop".
- Detector type (`TRFDetector`). Specifies the detector to be used to reconstruct the spectrum. Supported types are: "signal"(default), "peak" and "rms". Note that use of detectors other than "signal" will marginally increase sampling time.
- Units (`TRFUnits`). Specifies the units for the delivered spectrum data. Supported types are: "mW", "W", "↔ V", "A", "dBm"(default), "dBW", "dBV", "dBmV", "dBuV", "dBA", "dBmA", "dBuA", "dBuV/m" and "dBW/m2". Physical unit conversions assume a 50R system, and an ideal Isotropic antenna.

Antenna, Equalization and switching To provide calibrated output values, a receiver may be programmed with both Antenna Factor (AF) and Equalization tables to allow measurements to be compensated for connected equipment and cabling. R6xx0 receivers additionally support GPIO control for antenna switching. The details of programming such compensation is detailed in [Programming Antenna, Equalization and switching](#) below.

spectrum capture The parameter [TRFDurationFrames](#) may be used to control the length of a capture sequence. By default this parameter is zero, denoting continuous capture until the user detaches the stream from the source receiver. If this parameter is non-zero, then it specifies the exact number of spectrum frames to be captured. On completion of capture, the stream will self-detach from the source receiver.

reprogramming and synchronization When using a spectrum stream, it may be desired to reprogram capture parameters on-the-fly. As there is no explicit synchronization in the case where a spectrum stream remains attached, at some point, stream parameters reflected in the received spectra will change, however there may be many frames delivered with the 'old' parameters before the 'new' frames are received. To ensure correct synchronization, either the 'incorrect' frames should be discarded, or alternatively 'finite' capture can be used to provide strong explicit synchronization between capture and reprogramming.

capture memory consumption It should be noted that the amount of memory consumed by the reconstruction process may become large if narrow resolution bandwidths and high spans are requested. Considering R5xx0 devices, an upper bound on memory requirements for capturing a particular span and resolution bandwidth is, for spans in excess of 20MHz:

$$UpperBound_{bytes} = (\frac{Span_{MHz}}{38} + 1) \times (3 \times \frac{62500}{RBW_{KHz}}) \times 8$$

For spans below 20MHz, the memory footprint for narrow RBWs is significantly reduced along with an improvement in sweep rate.

For R6000 devices, a larger 100MHz instantaneous bandwidth is available and the reconstruction method is slightly modified. The memory requirement in this case is, for spans in excess of 100MHz:

$$UpperBound_{bytes} = (\frac{Span_{MHz}}{100} + 1) \times \frac{100000}{RBW_{KHz}} \times 8$$

Again, for spans below 100MHz, the memory footprint for narrow RBWs is significantly reduced along with an improvement in sweep rate.

'Flow Control' flag and spectrum streams The binary flag [TRFFlowControl](#) may be set for any stream type. For a spectrum stream this acts as a control for a receiver to indicate that spectrum data delivery should be managed so that a new sweep sequence is only initiated once the previous one has completed and all data has been delivered. This mechanism prevents data from accumulating on the receiver, ensuring minimal latency from capture time to delivery time from the libtrf API. If this flag is not set (default), then spectrum data capture will not be restricted by network performance which may result in a faster capture speed at the expense of data latency.

IQ Streams

An IQ stream consists of a sequence of arbitrary-sized frames consisting of a contiguous sequence of IQ samples. The metadata included describes the number of samples, the usable IF bandwidth, the sample rate, frame sequence number and timestamp. For R5xx0 devices, the API supports IF bandwidths from 40MHz down to 12.5kHz across the full range of the receiver. For R6xx0 devices the IF bandwidth range extends upwards to 100MHz.

It should be noted that IQ stream parameters should be realizable for the specific receiver type for an 'attach' operation to a receiver device to be successful. For example the lower IFBW band edge (requested centre frequency minus

half IFBW) must be greater than the 9kHz lower frequency limit of the instrument, and the upper edge must be below the upper frequency limit of the instrument (ie. 27GHz for a -427 device).

Receivers perform decimation in hardware by powers of 2 with also decimation by 5 being available as a special case. When an IFBW such as 24MHz is requested, the decimation that will reliably deliver that bandwidth is selected. An allowance is made for band-edge filtering and hence the delivered IQ sample rate will always be higher than the requested IFBW. In this case an R5xx0 instrument will deliver a sample rate of 31.25MHz, and an R6xx0 instrument will deliver a sample rate of 30.7MHz. The libtrf code does not apply filtering to reduce the usable bandwidth and so the stream in this case will be reported to have an IFBW equivalent to $0.85 \times$ the delivered sample rate. For example if 22MHz IFBW is requested from an R5xx0 unit, the lowest sample rate that can deliver the $21.25/0.85$ sampling bandwidth is 25.88MHz. The nearest minimum sample rate delivering this is 31.25MHz. The IQ stream will therefore deliver a 31.25MHz sampled IQ stream, delivering a $31.25 \times 0.85 = 26.6625$ MHz usable bandwidth to satisfy this request.

R5xx0 IQ Streaming near 50MHz The top-end frequency for sampling centred at or below 50MHz is set by the hardware direct-sampling band-edge filter at 55MHz. The API will permit an upper-band edge (centre plus IFBW/2) greater than 55MHz for a centre frequency below 50MHz, however this will reveal the artefact of the direct sampled band-edge. Note that this limitation has no impact for IQ sampled with a centre frequency *above* 50MHz. No such limitation exists for R6000 devices.

Sample bit-depth To make optimal use of the available network bandwidth when streaming IQ data, ThinkRF units support the use of multiple bit-depths allowing the user to trade-off inherent digital sampling noise for transmission speed. Using the [TRFBitDepth](#) stream parameter, the resolution of the signal samples can be set from the set 8, 10, 12 and 16 bits. If a value is specified not in this set, the closest value will be chosen. The available signal dynamic range resulting from the change in sample width can be calculated as:

$$DynamicRange(dB) = 20 \times \log_{10}(2^{bits}) \approx 6.02 \times bits$$

The increase in available sample throughput is in inverse proportion to the bit width - for example an 8-bit depth signal will have twice the sample throughput over the same network as a 16-bit sampled signal.

Digital Gain parameter (R5xx0 only) To allow the further optimization of dynamic range, R5xx0 instruments support the [TRFDigitalGain](#) parameter which should be used in concert with the [TRFBitDepth](#) parameter to minimize quantization noise. The gain can be thought of as a bit-shift of the sampled signal towards the most significant bit, and hence changes in 6dB increments. The available combinations of [TRFBitDepth](#) and [TRFDigitalGain](#) on R5xx0 instruments are:

TRFBitDepth	TRFDigitalGain
16 bits	0
12 bits	0, 6, 12
10 bits	6, 12, 18, 24
8 bits	18, 24, 30, 36

Dynamic stream capture parameter changes Whilst an IQ or Spectrum stream is attached, it is possible to modify stream parameters. In this case the frames with the previous parameters will asynchronously cease delivery and packets corresponding to the new parameters will start to be delivered. During the transition, the parameters read back from the stream will momentarily be out of synchronization. Once the delivery of the first frame of the new stream has occurred, the stream parameters will accurately reflect the frame parameters. Good practice therefore will always rely on the frame metadata as the reliable source of stream parameter information.

Baseband Streams

Through the actions of a demodulator (processor), IQ data may be processed to produce baseband (ie. audio) data. The baseband frame type provides a sequence of contiguous real samples. The metadata in a baseband frame describes its size, usable bandwidth, sample rate, sequence number and timestamp.

2.2.5 Stream Files

All stream types can be used to generate files to capture the stream data. Similarly a stream may be 'sourced' from a file to support off-line processing and analysis. The stream file format is designed to be fast, comprising a JSON-formatted 'metadata' .json file, and a companion binary '.data' file. For most purposes, users should be able to treat these files as stand-alone units. For special cases where it is necessary to interpret the underlying data format, please contact thinkRF support.

Stream file JSON metadata

Radio data (IQ, Spectrum) written by libtrf consists of a pair of files with a common filename prefix. The .json file contains file metadata in JSON format. The .data file contains the actual captured data in an efficient binary format.

Example IQ file metadata (.json) IQ file metadata comprises five variables: 'dataType', 'device', 'gnssInfo', 'sampling' and 'iq'. The first three of these are generic. The sampling variable contains generic radio data descriptive information including centre frequency, bandwidth and units. The 'iq' section describes the encoding, frame format and data timing for the captured data.

```
{
  "dataType": "iq",
  "device": {
    "firmware": "1.2.5-dev6",
    "hasGNSS": true,
    "pllSource": "gnss",
    "serial": "190225-725",
    "type": "R5700-427",
    "version": "v1"
  },
  "gnssInfo": {
    "altitude": 106.7,
    "gnssAntDelay": 50,
    "gnssDynamic": "stationary",
    "gnssPosnEpoch": 1784245021477,
    "gnssSpeed": 0.000000,
    "gnssTrack": 0.000000,
    "latitude": 45.33300,
    "longitude": -75.9000
  },
  "iq": {
    "encoding": "Complex<_float32> LE",
    "endTimestamp": 1784269344704056016,
    "frames": 78,
    "framesRateHz": 1914.828491,
    "samples": 2545920,
    "samplesPerFrame": 32640,
    "startTimestamp": 1784269344668543712,
    "timeResolution": "ns"
  },
  "sampling": {
    "SampleRateHz": 62500000.000000,
    "bandwidthHz": 40000000.000000,
    "fCentreHz": 1000000000.000000,
    "refdBm": -50.000000,
    "units": "sqrt-mw"
  }
}
```

Example Spectrum file metadata (.json) For spectrum metadata, the 'device' and 'gnssInfo' sections remain the same as for IQ data. The 'sampling' section contains an additional resolution bandwidth parameter and the 'spectrum'

section replaces the 'iq' section.

```
{
  "dataType":"spectrum",
  "device":... ,
  "gnssInfo":... ,
  "sampling":{
    "bandwidthHz":7000006144.000000,
    "fCentreHz":4500002816.000000,
    "rbwHz":99948.281250,
    "refdBm":-50.000000,
    "units":"dBm",
    "windowFn":"nuttall"
  },
  "spectrum":{
    "encoding":"_float32 LE",
    "endTimestamp":1784269323468975839,
    "frames":100,
    "pointsPerFrame":141569,
    "startTimestamp":1784269304058977984,
    "timeResolution":"ns"
  }
}
```

2.2.6 Processors

A processor encapsulates functionality processing a stream (typically IQ or Spectrum) to produce an output stream (IQ, Spectrum, Baseband) and optionally output parameters. Examples in the current version of libtrf include AM and FM demodulators, IQ-to-spectrum conversion and spectrum characterization functionality.

AMDemodulator Processor

See examples/AMDemodulatorExample.cpp

The AMDemodulator processor implements a conventional Dual-sideband, Upper-sideband (USB) and Lower-sideband (LSB) AM demodulator. The demodulator operates on an input IQ stream and produces an output baseband stream. The parameters of the demodulation are controlled by the [TRFAMSubtype](#) parameter and [TRFOutputSampleRate](#) via [trfSetParameters](#).

The parameter [TRFFramesProduced](#) can be read via [trfGetParameters](#).

FMDemodulator Processor

See examples/FMDemodulatorExample.cpp

The FMDemodulator processor implements a conventional FM demodulator. The demodulator operates on an input IQ stream and produces an output baseband stream. The output sample rate is the only control parameter, set by [TRFOutputSampleRate](#).

The parameter [TRFFramesProduced](#) can be read via [trfGetParameters](#).

IQToSpectrum Processor

See examples/IQToSpectrumExample.cpp

This processor takes an IQ stream as input, and produces a spectrum stream as output. The conversion from IQ to spectrum data is controlled by the following set of parameters:

- [TRFRBWHz](#) - sets the Resolution Bandwidth.
- [TRFFollowIQ](#) - if set 'true' then one spectrum frame will be produced for each input IQ frame. this effectively sets [TRFOverlap](#) to zero and [TRFRBWHz](#) is ignored.
- [TRFWindow](#) - selects the window function to use in spectrum reconstruction.

- [TRFOverlap](#) - sets the degree of overlap between successive spectra from the input stream. A value of 0.5 implies 50% overlap of the input data stream in successive output spectra. Default is zero.

The processor produces the following read-only parameters:

- [TRFMatchIQ](#) - the sequence number of the most recently-processed input IQ frame.
- [TRFOutputSize](#) - the expected spectrum output size.

SpectrumCharacterization Processor

See `examples/SpectrumCharacterisationProcessorExample.cpp` and `SpectrumCharacterisationProcessorFromFileExample.cpp`

This processor implements the commonly used functionality of producing max, average and minimum spectrum traces. The output spectrum stream thus consists of the input spectrum, plus 3 'passenger' spectra for the average, minimum and maximum. The operation of this processor is controlled by the following parameters via [trfSetParameters](#):

- [TRFClearAverageTrace](#) - Setting this to 'true' will cause the average trace to be reset.
- [TRFClearMaxHoldTrace](#) - Setting this to 'true' will cause the max-hold trace to be reset.
- [TRFClearMinHoldTrace](#) - Setting this to 'true' will cause the min-hold trace to be reset.

The output spectrum frame will hold 3 'passenger' spectra. The indexes of these are contained in the output spectrum 'pJSONInfo' field. This field will also contain the current state of the [TRFClearAverageTrace](#), [TRFClearMaxHoldTrace](#), [TRFClearMinHoldTrace](#) valid when the output frame was created. These are included to aid in downstream synchronization.

BasebandResample Processor

See `FMDemodulationExample.cpp` and `BasebandFileExample.cpp`

This processor accepts an incoming baseband stream and converts from the input sample rate to a user-specified sample rate utilizing a linear-interpolation resampler.

The control parameters for this process are the output sample rate, specified by [TRFOutputSampleRate](#) (default 44.1kHz) and a requested output frame size specified using [TRFOutputFrameSize](#) (default 4096 samples).

BasebandToSpectrum Processor

See `FMDemodulationExample.cpp`

This processor accepts an incoming baseband stream and produces an output spectrum stream at a 1:1 frame ratio with the input stream. The spectrum is specified from 0Hz to the Nyquist bandwidth (ie. half sample rate) of the incoming baseband data. There are no control parameters for this process.

PeakFinder Processor

PeakFinder detects and reports the strongest peaks in an incoming spectrum stream.

PeakFinder is a terminal spectrum-processing stage: it attaches to a spectrum data stream and, for each incoming frame, scans the spectrum for local maxima - points whose level exceeds both immediate neighbours - that are above a configurable minimum level. Qualifying peaks are ranked by level and the strongest are retained, up to

a configurable maximum count. PeakFinder does not emit frames downstream; results are retrieved by polling its parameters.

Parameters:

- TRFPeakCount (RW) Maximum number of peaks to report, strongest first. Range 1-100, default 5.
- TRFMinimumPeakdBm (RW) Minimum level, in dBm, a point must exceed to be considered a peak. Range -140 to -20 dBm, default -90 dBm.
- TRFPeaks (RO) Array of peaks detected in the most recently processed frame, each as an object with TRF←PeakHz and TRFPeakdBm, sorted strongest first.
- TRFLastSequence (RO) Sequence number of the frame the current peaks were derived from, so a caller can detect when new results are available.

Alarms Processor

AlarmsProcessor monitors an incoming spectrum stream against one or more programmed frequency ranges and reports threshold-exceedance and signal-loss events.

AlarmsProcessor watches each incoming SpectrumFrame against a set of independently configured "ranges" - named frequency spans, each with its own threshold and event behaviour. For each range, two kinds of event are detected:

- Exceedance: the signal rises more than "exceeddB" above the range's threshold.
- Loss: the signal falls below the threshold by "sustaindB" (i.e. no signal remains above the sustain level) - only evaluated if sustaindB is set.

A range's threshold can be established three ways:

- Fixed: a single dB level, or a piecewise-linear shape (frequency/level pairs), given directly at configuration time.
- Trained: the processor observes the incoming spectrum for "trainSec" seconds and adopts the observed min/max envelope as a fixed threshold from that point on.
- Adaptive: with "observeSec" set, the processor continuously tracks a rolling min/max envelope over the trailing window of that duration, so the threshold follows slow changes in the background signal rather than staying fixed.

Adjacent/overlapping exceedance and loss events are combined into single ongoing events rather than being reported as a burst of short fragments, and closed events are retained briefly so their end can be reported once, avoiding lost or duplicated event records across frame boundaries. Each event is reported as a record carrying its ID, range name, exceed/loss flag, centre frequency, span, level, and start/end timestamps and frame sequence numbers.

Ranges are managed dynamically via addRange/removeRange/modifyRange, so the monitored set can be reconfigured at runtime without restarting the stream. Detected events are available by polling parameters at any time; optionally, the processor can also emit a spectrum frame per input frame (frameOutput) carrying the current threshold shape and an event-record annotation, either for every frame or only frames with an active event (gated).

See 'AlarmsProcessor_WhitePaper.pdf' in the examples folder for a detailed description.

2.3 Memory considerations

High-bandwidth receivers such as the thinkRF devices that libtrf supports can generate large data rates and hence care should be taken to manage the storage requirements and lifetime of data to ensure that memory constraints on any particular platform and for any particular application are respected.

2.3.1 IQ capture

Capturing IQ data will consume memory at the capture sampling rate, where each sample is 8 bytes (2 32-bit floating point numbers representing a complex sample). Inside libtrf, IQ data is buffered within streams up to a default value of 10 frames which minimizes overhead whilst minimizing the possibility of frame loss for most applications. An upper limit for the memory footprint of this default buffer is 64k x 8 bytes x 10 frames yielding 5MBytes. This buffering limit can be changed using either the [TRFBufferFrames](#) parameter or, more meaningfully the [TRFBufferSec](#) parameter of IQ streams. For example setting [TRFBufferSec](#) for an IQ stream to 0.5 will guarantee that the libtrf internal buffering will expand as needed to accommodate up to 0.5sec of IQ data.

Finite IQ Capture and buffering

If an IQ stream is created with a non-zero value for [TRFDurationSec](#), then that stream once attached will run until the required capture time has been achieved (finite capture). The R5xx0 and R6xx0 units both contain significant internal sample storage. Depending on the bit-depth chosen for capture this may provide more recording time. If all samples from a finite capture can be held in internal memory, then it can be guaranteed that the samples from the receiver unit will be contiguous; ie. no discontinuities.

$$T(MaxContiguous)_{sec} = \frac{128MB}{(IQSampleSize_{bytes} \times SampleRate_{Hz})}$$

If [TRFDurationSec](#) is determined to be below this limit, after attaching the stream to a device the [TRFIsContiguous](#) flag for the stream will be set. However it is still possible that discontinuities may be introduced when the underlying mechanisms recovering data from the RTSA accumulate data in local memory which is subject to constraints set by stream parameters. There are two solutions to eliminating the possibility of discontinuities in this case; buffer sizing and flow control.

The buffer sizing solution requires that the [TRFBufferSec](#) parameter for the finite IQ stream be set in excess of the capture duration. It is recommended that an excess sizing factor is used for [TRFBufferSec](#) as [TRFDurationSec](#) sets the minimum guaranteed duration, and the actual duration returned may be slightly longer. A factor of 1.2 should be sufficient for all capture durations.

The flow-control solution is to set the 'discarding' flag (see [trfAllowDiscarding\(\)](#)) to 'false' for the stream. This ensures that end-to-end flow control is used to ensure no data is discarded without requiring the user to specify potentially large buffering for the stream.

```
trfHandle hFiniteIQStream(trfInvalidHandle);
float fDuration(0.25f); // Seconds
trfCreateIQStream(&hFiniteIQStream, 1.0e9f, 20.0e6f, -40.0f);
if (bufferSizingSolution) {
    char *pJSON(nullptr);
    trfAddNumericParameter(&pJSON, TRFDurationSec, fDuration);
    trfAddNumericParameter(&pJSON, TRFBufferSec, fDuration*1.2f); // Allow for slight overflow
    trfSetParameters(hFiniteIQStream, &pJSON);
}
else if (nonDiscardingSolution) {
    trfAllowDiscarding(hFiniteIQStream, false);
}
...
```

2.3.2 Spectrum capture and Memory footprint

libtrf can produce very fine resolution spectra. It is easily possible for a user to create data structures with a very large memory footprint. For example, a 27GHz spectrum at 10kHz resolution bandwidth with a Nuttall window is represented by approximately 21MB of data. The memory requirement is directly proportional to the spectrum width, and inversely proportional to the resolution bandwidth chosen. In this case a 10-frame buffer ('[TRFBufferFrames](#)' parameter) will occupy 210MB of memory when filled. The libtrf functions do not constrain the user from specifying values that may require excessive amounts of memory, and therefore the user should be aware of memory requirements.

2.3.3 SpectrumCharacterization processor

Spectrum frames from this processor contain additional 'min', 'max' and 'average' frame data. It should therefore be noted that such frames will have 4x the size of a 'normal' spectrum frame.

2.4 Programming Antenna, Equalization and Switching

libtrf incorporates mechanisms for applying antenna factor and RF path equalization to calibrate output data from spectrum streams. Additionally, for R6xx0 devices, GPIO settings can be associated with different frequency ranges. A detailed example is included in 'SpectrumCSVCapture.cpp' in the examples directory.

2.4.1 Antenna Factor tables

To convert from received power to field units (dBuV/m, dBW/m²) the efficiency of an antenna must be incorporated. This is conventionally done by using tables that either express an Antenna Factor as a function of frequency in either dBi (dB relative to ideal isotropic), or dB/m (conventional Antenna Factor). Two example tables (comma-separated values) are provided in the examples directory taken from manufacturer datasheets (ETS_↔ Lindgren-3148.ant, Schwarzbeck_BBA_9106.ant) to illustrate the approach. As well as supporting loading tables from file, libtrf allows for Antenna Factor tables to be loaded programmatically as JSON objects.

2.4.2 Equalization tables

To compensate for the frequency response of cabling an antenna equipment, equalization tables can additionally be loaded. An equalization table expresses a loss in dB as a function of frequency. An example is included in the examples directory (RG316_3m_SMA.eq). Again, libtrf supports Equalization tables being loaded programmatically as JSON objects.

Loading tables from file Antenna and equalization tables can be loaded from a specified path using [trfLoadAntennaFiles](#) and [trfLoadEQFiles](#) calls. These functions discover .ant and .eq files respectively, loading them into internal session tables from which they can subsequently be referenced by name. These files specify the tables in a .csv format with a prefixed metadata header. In this header, lines prefixed with '#' are comments. Those prefixed with '!' specify metadata. The filename is used as the key for subsequent reference.

```
# Antenna file (example)
! "name": "Schwarzbeck BBA 9106"
! "manufacturer": "Schwarzbeck Mess-Elektronik"
! "model": "BBA 9106 Biconical"
! "pattern": "broadband biconical, co-pol"
! "notes": "Illustrative AF curve (smoothed); approximate, not for calibration."
Hz, dB/m
20e6,14.0
30e6,16.0
50e6,18.0
80e6,20.5
100e6,21.5
150e6,24.0
200e6,26.0
250e6,27.3
300e6,28.0
```

Loading tables programmatically A table may be specified as a JSON object with name, type ('antenna' or 'eq') and an array of frequency/value entries. An example is (eq table):

```
{ "name": "myEQ", "type": "eq", "units": "dB", "values": [
  { "fCentreHz": 100.0e6, "value": 1.0 },
  { "fCentreHz": 200.0e6, "value": 3.0 },
  { "fCentreHz": 500.0e6, "value": 3.2 },
  { "fCentreHz": 1.0e9, "value": 1.1 }
]
```

2.5 libtrf Structure

```
}
```

Programmatically, this can be generated and loaded using the following code. Note that [trfInvalidHandle](#) is used to specify programming a global parameter in the libtrf API.

```
// Program in an EQ table using JSON. Note - antenna tables can be loaded in a similar way
// using TRFType:TRFAntenna and units of "dBi" or "dB/m"
char *pTable(nullptr), *pEqualizer(nullptr), *pPoint(nullptr);
trfAddTextParameter(&pEqualizer, TRFName, "myEQ");
trfAddTextParameter(&pEqualizer, TRFType, TRFEqualization);
trfAddTextParameter(&pEqualizer, TRFUnits, "dB");
std::list<std::pair<float32, _float32>> lEQ({
    {100.0e6f, 1.0f}, {200.0e6f, 3.0f}, {500.0e6f, 3.2f}, {1.0e9f, 1.1f},
});
for (auto &cPair : lEQ) {
    trfAddNumericParameter(&pPoint, TRFFCentreHz, cPair.first);
    trfAddNumericParameter(&pPoint, TRFValue, cPair.second);
    trfAddJSONElementToArray(&pEqualizer, TRFValues, pPoint);
    trfDisposeString(&pPoint);
}
trfAddJSONElement(&pTable, TRFTable, pEqualizer);
trfDisposeString(&pEqualizer);
eStatus = trfSetParameters(trfInvalidHandle, &pTable);
```

Specifying Antenna Factor and Equalization table use Once tables have been specified, they are referenced in a [TRFRFSelect](#) object that specifies global parameters to use, and zero or more sub-ranges where other parameters are to be used. Once programmed, all captures will be subject to the specified parameters. In the case of R6x devices, associated GPIO states can also be programmed which would permit the control of, for example, an external antenna switch. The GPIO option does not exist for R5x devices. An example object is:

```
{ "rfPath": {
    "gpio":0, "range":[
        {"fStart":20.0e6, "fStop":100.0e6, "antenna":"Schwarzbeck_BBA-9106", "eq":"RG316_3m_SMA", "gaindB":10.0,
        "gpio":1},
        {"fStart":140.0e6, "fStop":800.0e6, "antenna":"ETS_Lindgren-3148", "eq":"RG316_3m_SMA", "gaindB":5.0,
        "gpio":2}
    ]
}
```

Note in the above, default settings (gpio, antenna, eq) can be specified outside of the 'range' array. This can be created programmatically using the following sequence:

```
// Configure receiver antenna usage (with intentional gaps)
char *pRanges(nullptr), *pRange(nullptr), *pRangesProgram(nullptr);
trfAddNumericParameter(&pRange, TRFRFFStart, 20.0e6f);
trfAddNumericParameter(&pRange, TRFRFFStop, 100.0e6f);
trfAddTextParameter(&pRange, TRFAntenna, "Schwarzbeck_BBA-9106");
trfAddTextParameter(&pRange, TRFEqualization, "RG316_3m_SMA");
trfAddNumericParameter(&pRange, TRFGaindB, 10.0f);
trfAddNumericParameter(&pRange, TRFGPIO, 0x01);
trfAddJSONElementToArray(&pRanges, TRFRFRange, pRange);
trfAddNumericParameter(&pRange, TRFRFFStart, 140.0e6f);
trfAddNumericParameter(&pRange, TRFRFFStop, 800.0e6f);
trfAddTextParameter(&pRange, TRFAntenna, "ETS_Lindgren-3148");
trfAddTextParameter(&pRange, TRFEqualization, "RG316_3m_SMA");
trfAddNumericParameter(&pRange, TRFGaindB, 5.0f);
trfAddNumericParameter(&pRange, TRFGPIO, 0x02);
trfAddJSONElementToArray(&pRanges, TRFRFRange, pRange);
trfAddNumericParameter(&pRanges, TRFGPIO, 0x0);
trfAddJSONElement(&pRangesProgram, TRFRFSelect, pRanges);
eStatus = trfSetParameters(hDevice, &pRangesProgram);
```

2.5 libtrf Structure

The libtrf library calls are divided into error handling and diagnostics, device functions, stream functions, processor functions and parameter manipulation.

2.5.1 Interpreting error codes

Most libtrf functions return an error code of type [trfStatus](#). A status code of '0' is, by convention the 'no error' response. Status codes greater than zero should be interpreted as warnings or status information rather than errors. Codes

below zero indicate actual error conditions.

2.5.2 Error Handling and Diagnostic Functions

The following functions provide support for debugging.

- [trfGetTextForStatus\(\)](#) - Interpret a libtrf status code as text.
- [trfGetNextStatus\(\)](#) - Get any additional status codes relating to the last error returned.
- [trfGetDetailedStatus\(\)](#) - Returns a string providing more detailed, potentially nested error information.

2.5.3 Device Handling Functions

The following provide means of discovering, interrogating and managing access to devices:

- [trfAddDeviceAddress\(\)](#) - Add a (remote) device to the device addressing table.
- [trfEnumerateDevices\(\)](#) - Enumerate all devices, returning a count. Any directly-connected devices will be automatically returned in this list.
- [trfExamineDevice\(\)](#) - Return all information known about an enumerated device.
- [trfOpenDevice\(\)](#) - Open an enumerated device, returning a device handle.
- [trfResetDeviceConnection\(\)](#) - Reset the network connection with a device. This provides a means of recovering a device in the case of a network or other system error.
- [trfClose\(\)](#) - Close any libtrf handle (device, stream, processor).

2.6 Stream and Stream Data Handling Functions

Receiver devices deliver data in streams of different kinds. These family of functions provide means of creating, destroying and obtaining data from streams.

- [trfCreateIQStream\(\)](#) - Create an unattached IQ stream.
- [trfCreateSpectrumStream\(\)](#) - Create an unattached spectrum stream. Note: Instances of BasebandStream are only supported as output of Processor instances and hence are only obtained from querying a processor instance.

2.6.1 Stream Frame Handling Functions

- [trfGetNextSpectrum\(\)](#) - Get the next spectrum (frame) from the specified stream.
- [trfFreeSpectrum\(\)](#) - Dispose of a spectrum frame returned by [trfGetNextSpectrum](#).
- [trfGetNextIQ\(\)](#) - Get the next IQ (frame) from the specified stream.
- [trfFreeIQ\(\)](#) - Free an IQ frame returned by [trfGetNextIQ](#).
- [trfGetNextBaseband\(\)](#) - Get the next baseband frame from the specified stream.
- [trfFreeBaseband\(\)](#) - Dispose of a baseband frame returned by [trfGetNextBaseband](#).

2.6.2 Streams and Files Functions

- [trfSetStreamOutputFile\(\)](#) - Attach a file to a stream to receive its contents or detach the stream from the current file.
- [trfAttachStreamToFile\(\)](#) - Source a stream from the specified file (non-discarding by default).

2.7 Stream Attach/Detach and State Functions

- [trfAttachStreamToDevice\(\)](#) - Attach a specified stream to a given device. This operation if successful will start the flow of data into the stream.
- [trfDetachStream\(\)](#) - Detach a stream from any entity it is attached to, halting the flow of data.
- [trfFlushStream\(\)](#) - Flush all data currently in the stream.
- [trfIsAttached\(\)](#) - Determine if the stream is attached to a data source and hence is expected to continue to deliver data.
- [trfAllowDiscarding\(\)](#) - Controls the discarding behaviour of a stream. By default all streams from receiver devices are 'discarding' - that is if the stream buffer size overruns its limits (see [TRFBufferFrames](#), [TRFBufferSec](#)) then the oldest frames will be discarded (subject to the discarding policy set by [TRFDiscardProportion](#)) to return the buffer within size limits. Conversely, streams sourced from files are set as 'non-discarding' which will cause the file reading process to block if frames are not consumed from the stream sufficiently quickly. In the case a user knows that a file-sourced stream will not be read directly (ie. it is sourcing a processor), then the user should set that stream to be 'discarding' to allow the read process to proceed without blocking. For an example of usage, see the `SpectrumCharacterizationProcessorExampleFromFile.cpp` example code.

2.7.1 Special File Handling Functions

- [trfConvertBasebandFileToWavFile\(\)](#) - Converts a captured baseband data file into an AIFF 'wav' file suitable for use with typical audio-handling utilities.

2.8 Processor Handling Functions

- [trfGetProcessorTypes\(\)](#) - Returns a description of the set of available processor type names.
- [trfCreateProcessor\(\)](#) - Given a valid processor type name, creates an instance of that processor type.
- [trfAttachProcessorToStream\(\)](#) - Attach the specified processor to a source stream. Note that the stream type must be that accepted by the processor or an error will be returned.
- [trfDetachProcessor\(\)](#) - Detaches the specified processor from its source stream.
- [trfGetProcessorOutputStream\(\)](#) - Return the output stream (if any) supported by the processor. Different processor types will support different output stream types.

2.8.1 Parameter Handling (JSON) Functions

Parameter management is performed by passing and obtaining strings from libtrf entities using the JSON notation. The following calls provide for the creation, examination and application of JSON parameter sets. All parameters may be read/written using their 'TRFxxx' parameter name as defined in [libtrf.h](#).

A user is not required to use these functions as libtrf will recognize and generate valid JSON strings in UTF-8 encoding, captured as conventional C (ie. `char *`, null-terminated) strings, allocated on the heap (ie. using `'new char[]'`). Use of stack-based storage will result in undefined behaviour. All strings returned by libtrf functions are heap-allocated and may be conveniently disposed of using the `trfDisposeString` function.

- `trfGetParameterInfo()` - Return all information known about the public parameters of a libtrf entity (device, stream, processor).
- `trfGetParameters()` - Return the values of all entity public parameters.
- `trfSetParameters()` - Apply a JSON parameter set to a libtrf entity (device, stream, processor).
- `trfReadParameterInfoElement()` - Extract a nested JSON description of a particular parameter from a parameter info string.
- `trfReadParameterAsJSON()` - Extract a JSON parameter as a new JSON string.
- `trfDisposeString()` - Utility function to dispose of JSON (or other) strings returned by libtrf functions. Note that this function supersedes `trfDisposeJSON` from earlier versions of libtrf.

All parameter names recognized by libtrf entities are listed in [libtrf.h](#) and are of the form 'TRF<name>'. The following functions allow the creation of JSON strings from parameter names and values:

- `trfAddNumericParameter()` - Add a named numeric parameter to a JSON parameter set string.
- `trfAddTextParameter()` - Add a named textual parameter to a JSON parameter set string.
- `trfAddBooleanParameter()` - Add a named boolean parameter to a JSON parameter set string.

The following functions allow the examination of JSON strings, extracting named parameters:

- `trfReadNumericParameter()` - Extract a named numeric parameter from a JSON parameter set string.
- `trfReadTextParameter()` - Extract a named textual parameter from a JSON parameter set string.
- `trfReadBooleanParameter()` - Extract a named boolean parameter from a JSON parameter set string.

Commonly-used Parameter Functions

To aid with creating self-documenting code, the following 'syntactic-sugar' functions are provided that encapsulate the setting of particular named parameters. This functionality can also be realized using `trfAdd<...>` and `trfRead<...>` calls above using the appropriate parameter names. The following functions are provided for ease of programming when encoding and examining commonly-used parameters.

- `trfAddFCentre()`
- `trfAddIFBWHz()`
- `trfAddFMinMax()`
- `trfAddRBWHz()`
- `trfAddWindowType()`
- `trfReadFCentre()`
- `trfReadIFBWHz()`
- `trfReadFMinMax()`

- [trfReadRBWHz\(\)](#)
- [trfReadWindowType\(\)](#)
- [trfReadSampleRateHz\(\)](#)
- [trfReadRefLeveldBm\(\)](#)

2.9 The Rest of this document

The remainder of this document provides a detailed description of all the functions, structures and definitions provided by libtrf. It also documents the example code provided with libtrf and provides detailed build instructions for windows and linux environments.

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

_complex32	Complex type consisting of 32-bit float Real (In-phase) and Imaginary (Quadrature) components .	29
_trfBasebandFrame	Baseband sampled data (ie. audio) frame. Basedband data can be of any type representable by an array of _float32 values. Multiple channels are supported	30
_trfIQFrame	IQ data frame. This structure contains the actual data and all descriptive metadata	31
_trfSpectrumFrame	Spectrum data frame. This structure contains the actual spectral data and all descriptive metadata	34

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

libtrf.h	37
------------------------------------	----

Chapter 5

Data Structure Documentation

5.1 `_complex32` Struct Reference

Complex type consisting of 32-bit float Real (In-phase) and Imaginary (Quadrature) components.

```
#include <libtrf.h>
```

Data Fields

- `_float32 fl`
32-bit floating point (IEEE-754) In-phase (I) sample value
- `_float32 fQ`
32-bit floating point (IEEE-754) Quadrature (Q) sample value

5.1.1 Detailed Description

Complex type consisting of 32-bit float Real (In-phase) and Imaginary (Quadrature) components.

5.1.2 Field Documentation

`fl`

```
_float32 _complex32::fl
```

32-bit floating point (IEEE-754) In-phase (I) sample value

`fQ`

```
_float32 _complex32::fQ
```

32-bit floating point (IEEE-754) Quadrature (Q) sample value

5.2 _trfBasebandFrame Struct Reference

Baseband sampled data (ie. audio) frame. Basedband data can be of any type representable by an array of `_float32` values. Multiple channels are supported.

```
#include <libtrf.h>
```

Data Fields

- [_uint32 uSequenceNumber](#)
Frame sequence number (monotonic unit-step)
- [_uint64 uTimestampNanosec](#)
Frame origination timestamp.
- [_float32 fSampleRateHz](#)
Sample rate in Hz.
- [_float32 fUsableBWHz](#)
Usable bandwidth of sampled data (is less than fSampleRateHz/2)
- [_uint32 uChannels](#)
Channels in pData. Channel0 at offset 0, 1 at offset uSamples etc...
- [_uint32 uSamples](#)
Samples in frame.
- `const _float32 ** ppData`
ppData[0] - channel0, ppData[1] - channel 1 etc... Frame data (range +/-1.0f). Array terminates with a NULL pointer.
- `const char * pJSONInfo`
Any additional information for this frame (ie. from processing)

5.2.1 Detailed Description

Baseband sampled data (ie. audio) frame. Basedband data can be of any type representable by an array of `_float32` values. Multiple channels are supported.

5.2.2 Field Documentation

fSampleRateHz

```
_float32 _trfBasebandFrame::fSampleRateHz
```

Sample rate in Hz.

fUsableBWHz

```
_float32 _trfBasebandFrame::fUsableBWHz
```

Usable bandwidth of sampled data (is less than fSampleRateHz/2)

5.3 _trfIQFrame Struct Reference

pJSONInfo

`const char* _trfBasebandFrame::pJSONInfo`

Any additional information for this frame (ie. from processing)

ppData

`const _float32** _trfBasebandFrame::ppData`

ppData[0] - channel0, ppData[1] - channel 1 etc... Frame data (range +/-1.0f). Array terminates with a NULL pointer.

uChannels

`_uint32 _trfBasebandFrame::uChannels`

Channels in pData. Channel0 at offset 0, 1 at offset uSamples etc...

uSamples

`_uint32 _trfBasebandFrame::uSamples`

Samples in frame.

uSequenceNumber

`_uint32 _trfBasebandFrame::uSequenceNumber`

Frame sequence number (monotonic unit-step)

uTimestampNanosec

`_uint64 _trfBasebandFrame::uTimestampNanosec`

Frame origination timestamp.

5.3 _trfIQFrame Struct Reference

IQ data frame. This structure contains the actual data and all descriptive metadata.

```
#include <libtrf.h>
```

Data Fields

- [trfStatus eFlags](#)

Any warning/error for this frame only.

- [_uint32 uSequenceNumber](#)
Frame sequence number (monotonic unit-step)
- [_uint64 uTimestampNanosec](#)
Frame origination timestamp.
- [_float64 fCentreHz](#)
Centre frequency.
- [_float32 fFBWHz](#)
Usable IF Bandwidth - assumed symmetrical around fCentreHz.
- [_float32 fSampleRateHz](#)
Sample rate in Hz - fSampleRateHz is always greater than fFBWHz.
- [bool bDiscontinuity](#)
'true' if frame is not contiguous with previous frame. Will not be set for the first frame of a new stream when first attached. If detached and re-attached the first frame will have the discontinuity flag set.
- [bool bSubOptimalDRFlag](#)
'true' if a low or high excursion outside target dynamic range was detected in this frame
- [_uint32 uSamples](#)
*Complex IQ samples in *pData.*
- [const _complex32 * pData](#)
IQ Frame data as calibrated sqrt(mW)
- [const char * pJSONInfo](#)
Any additional information for this frame (ie. from processing)

5.3.1 Detailed Description

IQ data frame. This structure contains the actual data and all descriptive metadata.

5.3.2 Field Documentation

bDiscontinuity

```
bool _trfIQFrame::bDiscontinuity
```

'true' if frame is not contiguous with previous frame. Will not be set for the first frame of a new stream when first attached. If detached and re-attached the first frame will have the discontinuity flag set.

bSubOptimalDRFlag

```
bool _trfIQFrame::bSubOptimalDRFlag
```

'true' if a low or high excursion outside target dynamic range was detected in this frame

eFlags

`trfStatus` `_trfIQFrame::eFlags`

Any warning/error for this frame only.

fCentreHz

`_float64` `_trfIQFrame::fCentreHz`

Centre frequency.

fFBWHz

`_float32` `_trfIQFrame::fFBWHz`

Usable IF Bandwidth - assumed symmetrical around fCentreHz.

fSampleRateHz

`_float32` `_trfIQFrame::fSampleRateHz`

Sample rate in Hz - fSampleRateHz is always greater than fFBWHz.

pData

`const _complex32*` `_trfIQFrame::pData`

IQ Frame data as calibrated sqrt(mW)

pJSONInfo

`const char*` `_trfIQFrame::pJSONInfo`

Any additional information for this frame (ie. from processing)

uSamples

`_uint32` `_trfIQFrame::uSamples`

Complex IQ samples in *pData.

uSequenceNumber

`_uint32` `_trfIQFrame::uSequenceNumber`

Frame sequence number (monotonic unit-step)

uTimestampNanosec

```
_uint64 _trfIQFrame::uTimestampNanosec
```

Frame origination timestamp.

5.4 _trfSpectrumFrame Struct Reference

Spectrum data frame. This structure contains the actual spectral data and all descriptive metadata.

```
#include <libtrf.h>
```

Data Fields

- [trfStatus eFlags](#)
Any warning/error for this frame only.
- [_uint32 uSequenceNumber](#)
Frame sequence number (monotonic unit-step)
- [_uint64 uTimestampNanosec](#)
Timestamp of first sample used in spectrum reconstruction.
- [_int64 iDurationNanosec](#)
Delta from first-sample to last-sample timestamp used in spectrum reconstruction. Note that this is not the same as the frame rate which can be derived from a stream by comparing the timestamps of successive trfSpectrumFrame instances in the stream.
- [_float64 fMinHz](#)
Centre frequency of first bin in sweep.
- [_float64 fMaxHz](#)
Centre frequency of last bin in sweep.
- [_float32 fRBWHz](#)
Resolution Bandwidth in Hz.
- [_uint32 uPoints](#)
*Number of data points in *pData.*
- `const _float32 * pData`
Frame data in stream-specified units (default is dBm)
- `const char * pUnits`
- [_uint32 uAdditionalSpectra](#)
Number of entries in pAdditionalSpectra.
- `const _float32 ** pAdditionalSpectra`
Array of uAdditionalSpectra entries.
- `const char * pJSONInfo`
Any additional information for this frame (ie. from processing)

5.4.1 Detailed Description

Spectrum data frame. This structure contains the actual spectral data and all descriptive metadata.

5.4.2 Field Documentation

eFlags

`trfStatus _trfSpectrumFrame::eFlags`

Any warning/error for this frame only.

fMaxHz

`_float64 _trfSpectrumFrame::fMaxHz`

Centre frequency of last bin in sweep.

fMinHz

`_float64 _trfSpectrumFrame::fMinHz`

Centre frequency of first bin in sweep.

fRBWHz

`_float32 _trfSpectrumFrame::fRBWHz`

Resolution Bandwidth in Hz.

iDurationNanosec

`_int64 _trfSpectrumFrame::iDurationNanosec`

Delta from first-sample to last-sample timestamp used in spectrum reconstruction. Note that this is not the same as the frame rate which can be derived from a stream by comparing the timestamps of successive `trfSpectrumFrame` instances in the stream.

pAdditionalSpectra

`const _float32** _trfSpectrumFrame::pAdditionalSpectra`

Array of `uAdditionalSpectra` entries.

pData

`const _float32* _trfSpectrumFrame::pData`

Frame data in stream-specified units (default is dBm)

pJSONInfo

```
const char* _trfSpectrumFrame::pJSONInfo
```

Any additional information for this frame (ie. from processing)

pUnits

```
const char* _trfSpectrumFrame::pUnits
```

uAdditionalSpectra

```
_uint32 _trfSpectrumFrame::uAdditionalSpectra
```

Number of entries in pAdditionalSpectra.

uPoints

```
_uint32 _trfSpectrumFrame::uPoints
```

Number of data points in *pData.

uSequenceNumber

```
_uint32 _trfSpectrumFrame::uSequenceNumber
```

Frame sequence number (monotonic unit-step)

uTimestampNanosec

```
_uint64 _trfSpectrumFrame::uTimestampNanosec
```

Timestamp of first sample used in spectrum reconstruction.

Chapter 6

File Documentation

6.1 libtrf.h File Reference

Data Structures

- struct [_complex32](#)
Complex type consisting of 32-bit float Real (In-phase) and Imaginary (Quadrature) components.
- struct [_trfIQFrame](#)
IQ data frame. This structure contains the actual data and all descriptive metadata.
- struct [_trfSpectrumFrame](#)
Spectrum data frame. This structure contains the actual spectral data and all descriptive metadata.
- struct [_trfBasebandFrame](#)
Baseband sampled data (ie. audio) frame. Basedband data can be of any type representable by an array of `_float32` values. Multiple channels are supported.

Macros

- `#define trfInvalidHandle (0xffffffff)`
Invalid handle value for any libtrf entity. As good practice, all unused or uninitialized handle values should as good practice be set to this value - ie.
- `#define trfInvalidSequence (0xfffffffffe)`
Invalid sequence number for any frame type.
- `#define TRFDefault "default"`
Generic JSON parameter names for libtrf entities; devices, streams, processors. These parameter names are used in conjunction with JSON string construction/ examination functions to allow parameters to be set for and obtained from libtrf entities. In the following documentation 'RO' denotes 'read-only', and RW denotes read/write. Setting a parameter for an entity that does not support that parameter is an error and will typically return `trfParameterSetError` when calling `trfSetParameters` with a JSON string including an unsupported parameter. Detailed information can be obtained using `trfGetDetailedStatus`. Note that these names may change across future releases and therefore it is highly recommended that the following symbolic names `TRF__` are used instead of their textual equivalent. RO - Read-only, RW - Read and Write.
- `#define TRFMin "min"`
RO minimum value tag.
- `#define TRFMax "max"`
RO maximum value tag.
- `#define TRFType "type"`

- RO type of unit tag (user readable)*

 - #define **TRFAddress** "address"

RO network address tag.
- #define **TRFDevice** "device"

RO device information section.
- #define **TRFDeviceType** "type"

RO parameter - device textual type.
- #define **TRFDeviceVersion** "version"

RO parameter - device textual version.
- #define **TRFDeviceSerialNum** "serial"

RO parameter - device textual serial number.
- #define **TRFDeviceFirmware** "firmware"

RO parameter - device textual firmware version.
- #define **TRFDeviceConnection** "connection"

RO parameter - device textual connection information.
- #define **TRFHasFlatteningInfo** "iqEqualization"

RO receiver flag that equalization data is available (flattening is possible)
- #define **TRFSequence** "sequence"

RO sequence number parameter (internal)
- #define **TRFTimestamp** "timestamp"

RO timestamp parameter (internal)
- #define **TRFTimeResolution** "timeResolution"

RO time resolution parameter (internal)
- #define **TRFBandwidthHz** "bandwidthHz"

RO generic bandwidth parameter.
- #define **TRFWriteStartSequence** "startSequence"

RW sequence number for first frame to record.
- #define **TRFWriteStopSequence** "stopSequence"

RW sequence number for last frame to record.
- #define **TRFFilename** "filename"

RW parameter - filename parameter (stream)
- #define **TRFLooping** "loopingFlag"

RW set to cause file to loop.
- #define **TRFSweepActive** "sweepActive"

RO parameter - sweep is active flag (Rx)
- #define **TRFIQActive** "iqActive"

RO parameter - IQ capture is active flag (Rx)
- #define **TRFBufferSec** "bufferSec"

RW parameter - user-settable maximum buffering duration in seconds. If set this will override TRFBufferFrames.
- #define **TRFBufferFrames** "bufferFrames"

RW parameter - user-settable maximum stream buffering in frames. If set this will override TRFBufferSec.
- #define **TRFDiscardProportion** "bufferDiscard"

RW parameter - On overflow (TRFBufferSec, TRFBufferFrames) this sets the proportion of the buffer that will be discarded. The oldest frames will be discarded first. Setting to zero will minimize the number of frames discarded. Setting to 1.0 will discard the whole buffer. All other values 0,1 are acceptable. This allows a user to manage the tradeoff between discontinuities and data latency.
- #define **TRFFMinHz** "fMinHz"

RW parameter - Min extent of sweep range.

- `#define TRFFMaxHz "fMaxHz"`
RW parameter - Max extent of sweep range.
- `#define TRFRBWHz "rbwHz"`
RW parameter - Resolution Bandwidth in Hz.
- `#define TRFVBWHz "vbwHz"`
RW parameter - Video Bandwidth in Hz.
- `#define TRFFFrameDuration "frameDuration"`
RO parameter - duration of a recorded frame in time units.
- `#define TRFDwellUsec "dwellUsec"`
RW parameter - minimum dwell time usec per MHz.
- `#define TRFWindow "windowFn"`
RW parameter - Window function for spectrum reconstruction.
- `#define TRFIQCapture "iqCapture"`
RW parameter - data structure holding complete IQ capture specification.
- `#define TRFDetector "detector"`
RW parameter - Detector type for spectrum reconstruction.
- `#define TRFFCentreHz "fCentreHz"`
RW parameter - IQ Capture centre frequency.
- `#define TRFSampleRateHz "sampleRateHz"`
RO parameter - IQ Capture sample rate.
- `#define TRFIFBWHz "ifbwHz"`
RW parameter - IF Bandwidth for IQ capture.
- `#define TRFUsableBWHz "usableBwHz"`
RO parameter - Usable bandwidth (baseband).
- `#define TRFChannels "channels"`
RO parameter - Channels of data, if more than one presented (baseband).
- `#define TRFUnits "units"`
RW parameter - Units of data ("mW", "W", "V", "A", "dBm", "dBW", "dBV", "dBmV", "dBuV", "dBA", "dBmA", "dBuA", "dBuV/m", "dBW/m2")
- `#define TRFEqualization "eq"`
RW parameter - Equalization to apply to the stream. Names a pre-loaded equalization table.
- `#define TRFGaindB "gaindB"`
RW parameter - Gain to apply to the channel.
- `#define TRFRefLevel "refdBm"`
RW parameter - User-supplied expected max signal dBm.
- `#define TRFUserCalibrationOffset "userCalDb"`
RW parameter - User-supplied correction-factor for both Spectrum and calibrated-IQ.
- `#define TRFFlattenFlag "flatten"`
RW parameter - apply spectrum equalization if available. By default this flag is set 'false'.
- `#define TRFDurationFrames "captureFrames"`
RW parameter - Optional capture duration in frames (0 indicates continuous capture).
- `#define TRFDurationSec "captureSec"`
RW parameter - IQ Stream specified capture duration.
- `#define TRFMaxContiguousSec "maxContiguousSec"`
RO parameter - IQ Stream max limit of continuous streaming. Note -only valid after attaching the stream to a receiver device.
- `#define TRFIsContiguous "contiguous"`

- RO parameter - Indicates this IQ stream is guaranteed contiguous.*

 - #define [TRFFramesExpected](#) "framesExpected"
- RO parameter - Number of frames expected in this stream, or 0 if configured for continuous capture. If the stream is.*

 - #define [TRFFramelnFile](#) "frameInFile"

*Number of this frame within a file (playback stream only)
guaranteed contiguous (TRFIsContiguous is set) then this number will be the exact number of frames delivered before exhaustion. Otherwise this will be a lower-bound on the number of frames delivered before the stream is terminated by the API.*
- RO parameter - Dynamic range is non-optimal flag.*

 - #define [TRFSubOptimalFlag](#) "subOptimalDr"
- RO parameter - Full adapt flag - if set on stream will cause DR adaption before IQ streaming (~500msec)*

 - #define [TRFFullAdaptFlag](#) "adaptFull"
- RW parameter - Step adapt flag - if set on stream will cause DR adaption before IQ streaming.*

 - #define [TRFStepAdaptFlag](#) "adaptStep"
- RO parameter - Measured average power - only valid if TRFRefLevelAdapt is true.*

 - #define [TRFMeasuredAvgDbm](#) "avgDbm"
- RO parameter - Proportion of dynamic range utilized - expressed in dB relative to full-scale.*

 - #define [TRFFlowControl](#) "flowControl"

RW parameter - If set, ensures tight flow control with a receiver for spectrum data.
- RW parameter - default 'true'. Removes residual DC offset signal from IQ data.*

 - #define [TRFDCOffsetFilter](#) "dcOffsetFilter"
- RW parameter - default 'false'. Perform spur suppression (R5xx0 only)*

 - #define [TRFSpurReject](#) "spurReject"
- RO parameter - Attenuation used in dB.*

 - #define [TRFAttenuation](#) "attenuation"
- RO parameter - Packet duration in seconds.*

 - #define [TRFPacketDurationMsec](#) "packetDuration"
- RO parameter - IF Filter ratio applied.*

 - #define [TRFFilterRatio](#) "ifFilterRatio"
- RW parameter - NTP servers list.*

 - #define [TRFNTP](#) "ntpServers"
- RW parameter - LAN MTU (maximum transfer unit, in bytes) value.*

 - #define [TRFLANMTU](#) "lanMtu"
- RW parameter - SCPI transaction timeout.*

 - #define [TRFSCPIQueryTimeout](#) "scpiQueryTimeout"
- RO parameter - native 'base' sample rate of a receiver.*

 - #define [TRFRxSampleRate](#) "rxSampleRate"
- RO parameter - true if Receiver has a GNSS module.*

 - #define [TRFHasGNSS](#) "hasGNSS"
- RO parameter - true if GNSS data is valid, false otherwise.*

 - #define [TRFGNSSValid](#) "gnssValid"
- RO parameter - Latitude returned by GNSS.*

 - #define [TRFLatitude](#) "latitude"
- RO parameter - Longitude returned by GNSS.*

 - #define [TRFLongitude](#) "longitude"
- RO parameter - Altitude returned by GNSS.*

 - #define [TRFAltitude](#) "altitude"

- `#define TRFGNSSRxTime "gnssPosnEpoch"`
RO parameter - Last position report in UTC Msec.
- `#define TRFGNSSTimestamp "gnssTimeNanosec"`
RO parameter - Most recent GNSS UTC timestamp in nanosec.
- `#define TRFGNSSADelay "gnssAntDelay"`
RW parameter - GNSS Antenna delay parameter.
- `#define TRFGNSSConstellation "gnssCons"`
RW parameter - GNSS Constellation to track format: "<option 1>[,option 2]".
- `#define TRFGNSSDynamicMode "gnssDynamic"`
RW parameter - GNSS Dynamic mode (textual)
- `#define TRFGNSSSpeedOverGround "gnssSpeed"`
RO parameter - GNSS Speed over ground in m/sec.
- `#define TRFGNSSHeadingDegT "gnssHeading"`
RO parameter - GNSS Heading in degrees true.
- `#define TRFGNSSTrackDegT "gnssTrack"`
RO parameter - GNSS Track in degrees true.
- `#define TRFGNSSMagVarDegT "gnssMagVar"`
RO parameter - GNSS Magnetic variation in degrees E+, W-.
- `#define TRFDeviceTemperature "deviceTemperature"`
RO parameter - Reported device internal temperatures.
- `#define TRFDeviceLockStatus "deviceClockLock"`
RO parameter - Reported device clock lock statuses.
- `#define TRFPLLSource "pllSource"`
RW parameter - PLL clock source (internal, external, gnss)
- `#define TRFPPSSource "ppsSource"`
RW parameter - PPS clock source (external, gnss)
- `#define TRFTimeSync "timeSync"`
RW parameter - clock reference source (disable, "ntp,once", "ntp,continuous")
- `#define TRFCurrentDate "date"`
RW parameter - current date as year,month,date.
- `#define TRFCurrentTime "time"`
RW parameter - current time as hour, minute, second [,millisecond].
- `#define TRFBitDepth "bitDepth"`
RW - Bit-depth size 8,10,12,16.
- `#define TRFDigitalGain "digitalGain"`
RW - Digital Gain in dB, for 5xx0 products only.
- `#define TRFGPIO "gpio"`
RW parameter - Explicitly set GPIO bits (GPIO [0:2]) now.
- `#define TRFRFSelect "rfPath"`
RW parameter - identifies RFPATH control group follows.
- `#define TRFRFRRange "range"`
RW parameter - identifies RF path GPIO range group follows.
- `#define TRFRFFStart "fStart"`
RW parameter - start of frequency range.
- `#define TRFRFFStop "fStop"`
RW parameter - end of frequency range.
- `#define TRFRFGPIO "gpio"`

- RW parameter - associated GPIO setting (GPIO [0:2])*

 - #define TRFAntenna "antenna"

RW parameter - associated antenna table.

 - #define TRFTable "table"
 - #define TRFName "name"
 - #define TRFType "type"

RO type of unit tag (user readable)

 - #define TRFValues "values"
 - #define TRFValue "value"
 - #define TRFAMSubtype "amType"

RW parameter - type of AM demodulation to perform; [DSB(default), LSB, USB].

 - #define TRFFramesProduced "frames"

RO parameter - count of frames produced.

 - #define TRFOutputSampleRate "outputSampleRate"

RW parameter - requested baseband output sample rate.

 - #define TRFOutputFrameSize "outputFrameSize"

RW parameter - requested baseband output frame size.

 - #define TRFFollowIQ "followIQ"

RW parameter - if 'true', then one spectrum frame will be produced for each input IQ frame with RBW dictated by the window function and input IQ frame size.

 - #define TRFOverlap "fftOverlap"

RW parameter - sets the proportion 0.0-1.0 of overlap in successive output spectra.

 - #define TRFMatchIQ "matchIQ"

RO parameter - Most recently processed input IQ frame sequence number.

 - #define TRFOutputSize "outputSize"

RO parameter - Size of output spectra expected with current parameters.

 - #define TRFRecentFrames "recentFrames"

RO parameter - Number of frames processed since last trfGetParameters call.

 - #define TRFAverageCount "average"

RW parameter - number of spectra over which to create 'average' spectrum.

 - #define TRFClearAverageTrace "clearAverageTrace"

RW parameter - On the next frame, clear the average trace.

 - #define TRFClearMaxHoldTrace "clearMaxHoldTrace"

RW parameter - On the next frame, clear the max-hold trace.

 - #define TRFClearMinHoldTrace "clearMinHoldTrace"

RW parameter - On the next frame, clear the min-hold trace.

 - #define TRFWeightedFilter "weightedFilter"

RW parameter - Defines smoothing aggressiveness 0 to 1.

 - #define TRFOccupiedPercentage "occupiedPercentage"

RO - Percentage of the total integrated power.

 - #define TRFSpan "spanHz"

RW parameter - span over which to integrate power.

 - #define TRFTimeConstant "timeConstant"

RW parameter - averaging time constant.

 - #define TRFMinimumPeakdBm "minPeakdBm"

RW parameter - minimum power for any returned peak.

 - #define TRFPeakCount "peakCount"

- RW parameter - maximum number of peaks to return.*
- `#define TRFLastSequence "lastSequence"`
RO parameter - sequence number of last processed frame.
- `#define TRFPeaks "peaks"`
RO parameter - vector of detected peaks of the form {TRFPeakHz: peakFHz, TRFPeakdBm: peakLeveldBm}.
- `#define TRFPeakHz "peakFrequencyHz"`
element of TRFPeaksVector - peak frequency in Hz
- `#define TRFPeakdBm "peakLeveldBm"`
element of TRFPeaksVector - peak level in dBm
- `#define TRFStartTimeStamp "startTimestamp"`
RO - StartTimestamp param in recorded file.
- `#define TRFEndTimeStamp "endTimestamp"`
RO - EndTimestamp param in recorded file.
- `#define TRFDataType "dataType"`
RO - DataType param in recorded file.
- `#define TRFSampling "sampling"`
RO - Sampling param in recorded file.
- `#define TRFEncoding "encoding"`
RO - Encoding param in recorded file.
- `#define TRFSamples "samples"`
RO - Samples param in recorded file.
- `#define TRFPointsPerFrame "pointsPerFrame"`
RO - PointsPerFrame param in recorded file.
- `#define TRFReferenceLeveldBm "referenceLeveldBm"`
RO - ReferenceLeveldBm param in recorded file.
- `#define TRFIQ "iq"`
RO - IQ group in recorded file.
- `#define TRFSpectrum "spectrum"`
RO - Spectrum group in recorded file.
- `#define TRFBaseband "baseband"`
RO - Baseband group in recorded file.
- `#define TRFIOnly "iOnly"`
RO - I-only datatype.
- `#define TRFGNSSInfo "gnssInfo"`
RO - GNSSInfo group in recorded file.
- `#define TRFAddRange "addRange"`
W - specification for range follows.
- `#define TRFRemoveRange "removeRange"`
W - remove specified range.
- `#define TRFModifyRange "modifyRange"`
W - modify specified range.
- `#define TRFFrameOutputEnabled "frameOutput"`
RW - if true, alarms processor frame output is active.
- `#define TRFGatedOutput "gated"`
RW - if true, alarms processor only passes on spectra in case of exceedance.
- `#define TRFEventID "id"`
RO - unique event ID.

- `#define TRFExceedNotLoss "exceedNotLoss"`
RO - boolean indicating if this is an exceedance or loss event.
- `#define TRFRRangeName "name"`
RW - name of range.
- `#define TRFRRangeActive "active"`
RW - range active flag.
- `#define TRFFrequency "freq"`
RW - range frequency point.
- `#define TRFLevel "level"`
RW - range level point.
- `#define TRFExceeddB "exceeddB"`
RW - alarm 'exceeds' threshold by this amount.
- `#define TRFSustaindB "sustaindB"`
RW - No alarm if any signal sustains above this threshold.
- `#define TRFThreshold "threshold"`
RW - specification (scalar, vector) for upper threshold.
- `#define TRFTrainSeconds "trainSec"`
RW - pre-training time for static thresholds.
- `#define TRFObserveSeconds "observeSec"`
RW - observation time for dynamic thresholds.
- `#define TRFEventDuration "duration"`
RO - observed event duration (milliseconds)
- `#define TRFExceedRecords "exceed"`
RO - observed event records.
- `#define TRFStartSequence "startSequence"`
RO - sequence number of first frame in event.
- `#define TRFEndSequence "endSequence"`
RO - sequence number of last frame in event.

Typedefs

- `typedef unsigned char _uint8`
Basic types used in the API.
- `typedef unsigned short _uint16`
Unsigned 16-bit integer.
- `typedef unsigned int _uint32`
Unsigned 32-bit integer.
- `typedef unsigned int _uint`
Unsigned integer.
- `typedef unsigned long long int _uint64`
Signed 64-bit integer.
- `typedef float _float32`
32-bit floating point (IEEE-754) value
- `typedef double _float64`
64-bit floating point (IEEE-754) value
- `typedef _uint32 trfHandle`
Device or processor handle type.

- typedef enum [_trfStatus](#) [trfStatus](#)
General operation status reporting.
- typedef struct [_trfIQFrame](#) [trfIQFrame](#)
IQ data frame. This structure contains the actual data and all descriptive metadata.
- typedef struct [_trfSpectrumFrame](#) [trfSpectrumFrame](#)
Spectrum data frame. This structure contains the actual spectral data and all descriptive metadata.
- typedef struct [_trfBasebandFrame](#) [trfBasebandFrame](#)
Baseband sampled data (ie. audio) frame. Basedband data can be of any type representable by an array of `_float32` values. Multiple channels are supported.

Enumerations

- enum [_trfStatus](#) {
[trfOk](#) = 0, [trfWaiting](#) = 1, [trfExhausted](#) = 2, [trfNotStarted](#) = 3,
[trfDiscontinuousWithPreviousFrame](#) = 4, [trfDetached](#) = 5, [trfPingFailed](#) = 6, [trfDuplicateAddress](#) = 7,
[trfUnimplemented](#) = -1, [trfAPINotInitialized](#) = -2, [trfDeviceAddressAlreadyKnown](#) = -3, [trfUnallocatedUserData](#) = -4,
[trfDeviceIndexOutOfRange](#) = -5, [trfNoDeviceInformation](#) = -6, [trfCannotOpenDevice](#) = -7, [trfBadDeviceHandle](#) = -8,
[trfBadUnitHandle](#) = -9, [trfNoParametersSpecified](#) = -11, [trfBadReceiverHandle](#) = -14, [trfIQStreamStartFailure](#) = -15,
[trfMemoryAllocationError](#) = -16, [trfInvalidStreamHandle](#) = -17, [trfNoAssociatedRxOrFile](#) = -18, [trfNotAnIQStream](#) = -19,
[trfNoDataSpecified](#) = -20, [trfInvalidHandleSpecified](#) = -21, [trfBadStreamShutdown](#) = -22, [trfUnsupportedParameter](#) = -23,
[trfCannotCloseStream](#) = -24, [trfNotASpectrumStream](#) = -25, [trfCannotRestartStream](#) = -26, [trfSweepStartFailure](#) = -27,
[trfFrameTypeMismatch](#) = -28, [trfFileOpenFailed](#) = -29, [trfFileTypesNotIQ](#) = -30, [trfFileTypesNotSpectrum](#) = -31,
[trfNoAssociatedSource](#) = -32, [trfUnknownParameter](#) = -33, [trfParameterSetError](#) = -34, [trfDeviceHandlesInvalid](#) = -35,
[trfSCPISendFailed](#) = -36, [trfSCPIQueryFailed](#) = -37, [trfBadStreamHandle](#) = -38, [trfCannotAttachIQStream](#) = -39,
[trfCannotAttachSpectrumStream](#) = -40, [trfCannotAttachStream](#) = -41, [trfCannotDetachIQStream](#) = -42,
[trfCannotDetachSpectrumStream](#) = -43,
[trfCannotDetachStream](#) = -44, [trfConnectionResetFailed](#) = -45, [trfBadFrequencyParameters](#) = -46,
[trfBadRBWParameter](#) = -47,
[trfBadReferenceLevelParameter](#) = -48, [trfSpectrumStreamCreateFailure](#) = -49, [trfBufferSecOutOfRange](#) = -50,
[trfBufferFramesOutOfRange](#) = -51,
[trfBufferDiscardProportionOutOfRange](#) = -52, [trfFrequencyRangesInvalid](#) = -53, [trfBadGNSSDynamicMode](#) = -54, [trfBadPLLSource](#) = -55,
[trfPLLSourceWithNoGNSS](#) = -56, [trfFCentreInvalid](#) = -57, [trfIFBWInvalid](#) = -58, [trfFMinInvalid](#) = -59,
[trfFMaxInvalid](#) = -60, [trfResolutionBWInvalid](#) = -61, [trfFCentreOutOfRange](#) = -62, [trfIFBWOutOfRange](#) = -63,
[trfReferenceLevelOutOfRange](#) = -64, [trfResolutionBWOutOfRange](#) = -65, [trfUnknownWindowType](#) = -66,
[trfStartFrequencyOutOfRange](#) = -67,
[trfStopFrequencyOutOfRange](#) = -68, [trfDeviceUnreachable](#) = -69, [trfInvalidFilename](#) = -70, [trfCannotOpenOutputFile](#) = -71,
[trfUninitializedUserData](#) = -72, [trfBadProcessorHandle](#) = -73, [trfCannotCreateProcessor](#) = -74, [trfProcessorAttachFailed](#) = -75,
[trfProcessorDetachFailed](#) = -76, [trfProcessorUnattached](#) = -77, [trfNotABasebandStream](#) = -78, [trfFileTypesNotBaseband](#) = -79,
[trfBasebandStreamStartFailure](#) = -80, [trfSampleRateInvalid](#) = -81, [trfCannotObtainOutputStream](#) = -82,
[trfCannotConnectOutputStream](#) = -83,
}

trfBadFilename = -84, trfWindowTypeInvalid = -85, trfDeviceCommunicationsLost = -86, trfUnknownStreamType = -87,
 trfStreamHandleAlreadyOpen = -88, trfInvalidStreamSource = -89, trfBadPPSSource = -90, trfStreamFailedToStart = -91,
 trfStreamParametersError = -92, trfBasebandStreamCreateFailure = -93, trfInvalidIP = -94, trfUnknownAddress = -95,
 TRFBitDepthOutOfRange = -96, trfFatalReceiverError = -97, trfDigitalGainOutOfRange = -98, trfJSONEmpty = -99,
 trfNoUniqueArrayParameter = -100, trfNoArrayParameter = -101, trfInvalidArrayIndex = -102, trfNoOp = -103,
 trfSpectrumCompressFailed = -104, trfOnCallPointerShouldBeNull = -105, trfBadJSONElement = -106,
 trfBadJSONStructure = -107,
 trfBadJSONName = -108, trfUnknownDetectorType = -109, trfDwellTimeOutOfRange = -110, trfVideoBWOutOfRange = -111,
 trfUnitsInvalid = -112, trfGainInvalid = -113, trfEQInvalid = -114, trfVBWInvalid = -115,
 trfDwellTimeInvalid = -116, trfAntennaInvalid = -117, trfEqualizerInvalid = -118, trfRebootedStatus = -998,
 trfGenericFailStatus = -999, trfLastErrorSentinel = -1000 }

General operation status reporting.

- enum **trfDecimationType** { **trfMaximum** = 0, **trfMinimum** = 1, **trfLogAverage** = 2, **trfLinAverage** = 3 }

Type of decimation to perform, selecting peak, minimum or average from a set of points as the combined result point.

Functions

- _TRFAPI char * **trfGetVersion** (void)
Return a C-string describing this version of the library.
- const _TRFAPI char * **trfGetTextForStatus** (trfStatus eStatus)
Return textual (UTF-8, ISO-Latin-1) status description matching with the passed status code.
- _TRFAPI trfStatus **trfGetNextStatus** (void)
Return any additional (descriptive) error codes arising from the last issued libtrf function on the calling thread. To return all error status a user can iterate on this call until it returns trfOk to indicate no further error information is available.
- _TRFAPI char * **trfGetDetailedStatus** (void)
Return textual (UTF-8, ISO-Latin-1) error details for the last significant status on the calling thread. Clears error string for this thread on on call.
- _TRFAPI char * **trfGetUserResourcesReport** (void)
Return textual (UTF-8, ISO-Latin-1) report of current user resource utilization within libtrf. Intended as an aid to help a user determine correct operation and identify any resource leakage.
- _TRFAPI trfStatus **trfLoadAntennaFiles** (const char *pFullPath)
Inteprets pFullPath as either a directory or specific file to load an antenna table from.
- _TRFAPI trfStatus **trfLoadEQFiles** (const char *pFullPath)
Inteprets pFullPath as either a directory or specific file to load an EQ table from.
- _TRFAPI trfStatus **trfAddDeviceAddress** (const char *pRemoteAddress)
Add a potential device address for use by trfEnumerateDevices. Note that local devices that may be discovered directly (LAN, USB3 etc...) will not require their addresses to be added through this call, however it is not an error if they are.
- _TRFAPI trfStatus **trfRemoveDeviceAddress** (const char *pRemoteAddress)
Removes a potential device address from later enumeration.
- _TRFAPI trfStatus **trfEnumerateDevices** (_uint *puDevices, bool bAutoDiscover)
Enumerate all accessible devices and return a count. Device enumeration may be initiated by a user at any time to enumerate newly connected devices or update the list of devices.
- _TRFAPI trfStatus **trfExamineDevice** (_uint uDeviceIndex, char **ppJSON)
Return known information about a discovered device (uDeviceIndex in the range [0 .. (uDevices-1)]) into user-allocated trfDeviceInfo structure.

- `_TRFAPI trfStatus trfOpenDevice` (`_uint uDeviceIndex`, `trfHandle *pDevice`)
*Given a device index, attempt to open it. On success sets the device handle *pHandle.*
- `_TRFAPI trfStatus trfResetDeviceConnection` (`trfHandle cDevice`)
If a communications error is suspected, issuing this call will cause a connection shutdown and re-open to the device without losing any operational state. On success the device will continue with programmed operation. If the return value is not 'trfOk' then it is likely communications are unrecoverable and the device handle should be closed (see `trfClose`).
- `_TRFAPI trfStatus trfClose` (`trfHandle *pHandle`)
*Close any open handle (device, processor, stream) On successful return, *pHandle==kInvalidHandle.*
- `_TRFAPI trfStatus trfCreateIQStream` (`trfHandle *pStream`, `_float32 fCentreHz`, `_float32 fIFBWHz`, `_float32 fRefLeveldBm`)
Create an IQ stream with the given parameters. This entity will be inactive until successfully attached to a device capable of servicing the stream. Note that in the current implementation of libtrf, IF bandwidths narrower than 100kHz are not supported. Additionally IF bandwidths of less than 1MHz below 50MHz FCentre exhibit significant distortion and their use is not recommended. This limitation will be addressed in planned future releases.
- `_TRFAPI trfStatus trfGetNextIQ` (`trfHandle cStream`, `trfIQFrame **ppFrame`, `_uint32 uTimeoutMsec`)
*Obtain the next IQ frame (if available) from the stream. If no frame is yet available trfStatus will be 'trfWaiting'. If the stream has been halted then the status 'trfExhausted' may be returned to indicate there are no more buffered frames to retrieve. If the retrieval succeeds, trfStatus will be trfOk and *ppFrame will point to the returned IQ frame. The frame should be disposed of using `trfFreeIQ`. The trfExhausted return indicates that capture for this 'attach' is complete and the stream has been detached.*
- `_TRFAPI trfStatus trfFreeIQ` (`trfIQFrame **ppFrame`)
Free the given IQ frame, which will no longer be valid after this call (NULL). The user should ensure that frames are disposed when no longer required to avoid STORAGE_CAPACITY_VIOLATION errors.
- `_TRFAPI trfStatus trfCreateSpectrumStream` (`trfHandle *pStream`, `_float32 fMinHz`, `_float32 fMaxHz`, `_float32 fRBWHz`, `const char *pWindowType`, `_float32 fRefLeveldBm`)
Create a spectrum stream with the given parameters. This entity will be inactive until successfully attached to a device capable of servicing the stream.
- `_TRFAPI trfStatus trfGetNextSpectrum` (`trfHandle cStream`, `trfSpectrumFrame **ppFrame`, `_uint32 uTimeout←Msec`)
*Obtain the next frame (if available) from the stream. If no frame is yet available trfStatus will be 'WAITING' and *pp←Frame will be NULL otherwise trfStatus will be trfOk and *ppFrame will point to the returned data frame. The frame should be disposed of using `trfFreeSpectrum`.*
- `_TRFAPI trfStatus trfFreeSpectrum` (`trfSpectrumFrame **ppFrame`)
Free the given spectrum frame, which will no longer be valid after this call. The user should ensure that frames are disposed when no longer required to avoid STORAGE_CAPACITY_VIOLATION errors.
- `_TRFAPI trfStatus trfCreateBasebandStream` (`trfHandle *pStream`)
Create a baseband stream with null parameters.
- `_TRFAPI trfStatus trfGetNextBaseband` (`trfHandle cStream`, `trfBasebandFrame **ppFrame`, `_uint32 u←TimeoutMsec`)
*Obtain the next Baseband frame (if available) from the stream. If no frame is yet available trfStatus will be 'trfWaiting'. If the stream has been halted then the status 'trfExhausted' may be returned to indicate there are no more buffered frames to retrieve. If the retrieval succeeds, trfStatus will be trfOk and *ppFrame will point to the returned Baseband frame. The frame should be disposed of using `trfFreeBaseband`. The trfExhausted return indicates that capture for this 'attach' is complete and the stream has been detached.*
- `_TRFAPI trfStatus trfFreeBaseband` (`trfBasebandFrame **ppFrame`)
Free the given IQ frame, which will no longer be valid after this call (NULL). The user should ensure that frames are disposed when no longer required to avoid STORAGE_CAPACITY_VIOLATION errors.
- `_TRFAPI trfStatus trfAttachStreamToDevice` (`trfHandle cDevice`, `trfHandle cStream`)
Attach the given stream to the specified device. A stream may only be attached to one device at a time. If the device supports the stream, the call will return trfOk and the stream will start to produce data, subject to resource availability on the specified device.

- `_TRFAPI trfStatus trfDetachStream (trfHandle cStream)`
Detach the given stream from any device to which it is connected. This will stop any capture associated with the stream - however the stream will still contain any buffered data that has been previously delivered.
- `_TRFAPI trfStatus trfFlushStream (trfHandle cStream)`
Remove all frames currently present in a stream. If the stream is attached and delivering data, this will potentially create a break in the data. In the case that the stream is detached, this will remove any outstanding frames. After this call any call to `trfGetNextXXX` will return `trfExhausted`.
- `_TRFAPI trfStatus trfIsAttached (trfHandle cStream)`
Test if the specified stream is attached. Note that a finite capture stream will detach itself after capture is completed. This change will occur at some point in time after the completion of capture, and hence it is valid for this function to return `trfOk` whilst an associated `trfGetNextXXXFrame` has returned `trfExhausted` (finite capture).
- `_TRFAPI trfStatus trfAllowDiscarding (trfHandle cStream, bool bAllowDiscarding)`
By default, the data contained in streams will be discarded to ensure buffer size constraints are respected. In certain cases - such as streaming from file - it is necessary to control the flow of data to keep a processing chain synchronized and not arbitrarily drop frames. This call allows streams that would normally contain non-discardable frames to allow discarding to occur. By setting this flag on (say) an IQ stream from a file that is being processed downstream, the user will not have to loop to keep this stream emptied to allow the processor to proceed - otherwise the stream would block waiting for the stream buffer to be emptied.
- `_TRFAPI trfStatus trfSetStreamOutputFile (trfHandle cStream, const char *pFilePath)`
Attaches the specified output file - or if the file exists and is not empty, a numbered sibling file to the specified stream. The self-contained output file may be replayed using the `trfCreateFileSourceStream` function. Note that the specified file path may be modified to ensure that it contains no special or problematic characters before use; these will be replaced with the underscore '_' character. The output binary file created has the extension '.data' (ThinkRF Spectrum, IQ, Baseband), The binary file is accompanied by a metadata file (.json). If `pFilePath` is NULL, any file writing for the stream shall be immediately halted.
- `_TRFAPI trfStatus trfAttachStreamToFile (trfHandle cStream, const char *pFilePath)`
Attaches the given stream to an input file. Assumes that the file is valid and of the correct type for the stream - otherwise the call will fail. Note that any special or problematic characters in the file path will be replaced with '_' prior to use. Playback from the stream is controllable through the following parameters that may be set for the stream: `TRFLooping` - boolean: if set true then the file will be played continuously by looping back to the start once the file has been completely delivered. Function parameters:
- `_TRFAPI trfStatus trfConvertBasebandFileToWavFile (const char *pSourceFilePath, const char *pDestinationFilePath)`
Stand-alone function to convert a previously created Baseband stream file and produce a WAV file equivalent output file. Any problematic characters in `pSourceFilePath` will be replaced with the underscore '_' character prior to use.
- `_TRFAPI trfStatus trfGetProcessorTypes (char **ppJSON)`
Returns a JSON-formatted array names 'processortypes' of the available processor types.
- `_TRFAPI trfStatus trfCreateProcessor (trfHandle *pProcessor, const char *pType)`
Create a processor entity of the given type. Returns a handle to the newly created processor.
- `_TRFAPI trfStatus trfAttachProcessorToStream (trfHandle hProcessor, trfHandle hSourceStream)`
Attach a processor to a source stream.
- `_TRFAPI trfStatus trfDetachProcessor (trfHandle hProcessor)`
Detach a processor from its current source stream.
- `_TRFAPI trfStatus trfGetProcessorOutputStream (trfHandle cProcessor, trfHandle *pStream)`
Return the output stream from the given processor.
- `_TRFAPI trfStatus trfDecimateSpectrum (trfSpectrumFrame **ppResult, const trfSpectrumFrame *pSource, trfDecimationType eType, _uint32 uDecimation)`
Decimate the given spectrum frame by the given factor. Note that if the spectrum does not contain an integral number of decimation intervals, the last interval will be ignored, truncating the range of the resulting spectrum below the original endpoint. Note that the RBW of the resulting frame will not be affected however the delta between points will increase for `uDecimation > 1`.

- `_TRFAPI trfStatus trfCompressSpectrum` (`_uint8 **ppBlock`, `_uint32 *pBlockSize`, `const trfSpectrumFrame *pSource`, `_float32 fThresholdBm`)
Compresses the given source frame into an opaque data block. If the threshold parameter is not 0.0f, it is interpreted as a threshold below which all exact spectrum values are ignored. On decompression all such values will be reproduced as the threshold value.
- `_TRFAPI trfStatus trfDecompressSpectrum` (`trfSpectrumFrame **ppResult`, `const _uint8 *pBlock`, `_uint32 uBlockBytes`)
Decompresses the given data block, produced by `trfCompressSpectrum` to produce a valid spectrum frame.
- `_TRFAPI trfStatus trfGetParameterInfo` (`trfHandle cEntity`, `char **ppJSON`)
Obtains all available parameters for the specified device along with the valid ranges, type and any associated information through `ppJSON`.
- `_TRFAPI trfStatus trfReadParameterInfoElement` (`char **ppJSON`, `const char *pParameter`, `const char *pElement`, `_float32 *pfValue`)
Read a sub-element from a JSON element as a `_float32` value. Useful for obtaining 'min', 'max' and 'default' values from parameter information.
- `_TRFAPI trfStatus trfReadParameterAsJSON` (`char **pJSON`, `const char *pParameter`, `char **ppJSONExtract`)
Read a sub-element from a JSON element as another JSON string. Useful for extracting substructure from a JSON string.
- `_TRFAPI trfStatus trfReadVectorElementAsJSON` (`char **pJSON`, `_uint32 uIndex`, `char **ppJSONExtract`)
Assuming `pJSON` is a valid JSON array, extracts the specified index from the array as a JSON string.
- `_TRFAPI trfStatus trfGetParameters` (`trfHandle cEntity`, `char **ppJSON`)
Obtain the current values of device/processor parameters and make accessible through `ppJSON`.
- `_TRFAPI trfStatus trfSetParameters` (`trfHandle cEntity`, `char **ppJSON`)
Set the current value of a device/stream/processor parameter.
- `_TRFAPI void trfDisposeString` (`char **ppString`)
*Dispose of a previously allocated `ppJSON` string. Note that only strings that are returned via a parameter of the form 'char **' should be disposed of using this call. This is provided for syntactic convenience and is equivalent to: if (`ppJSON != nullptr`) { if (`*ppJSON != nullptr`) { delete [>(*ppJSON); *ppJSON = nullptr; } }.*
- `_TRFAPI trfStatus trfAddNumericParameter` (`char **ppJSON`, `const char *pParameterName`, `_float32 fValue`)
For parameters not named above - add parameter by name Add named numeric parameter to `ppJSON`.
- `_TRFAPI trfStatus trfAddIntegerParameter` (`char **ppJSON`, `const char *pParameterName`, `int iValue`)
For parameters not named above - add parameter by name Add named numeric parameter to `ppJSON`.
- `_TRFAPI trfStatus trfAddTextParameter` (`char **ppJSON`, `const char *pParameterName`, `const char *pValue`)
Add named textual parameter to `ppJSON`.
- `_TRFAPI trfStatus trfAddArrayElement` (`char **ppJSON`, `const char *pArrayName`, `const char *pJSONElement`)
Add contained JSON element to the named array contained in `ppJSON`.
- `_TRFAPI trfStatus trfAddBooleanParameter` (`char **ppJSON`, `const char *pParameterName`, `bool bValue`)
Add named boolean parameter to `ppJSON`.
- `_TRFAPI trfStatus trfAddJSONElement` (`char **ppJSON`, `const char *pElementName`, `const char *pElement`)
Add given parameter (pre-constructed JSON) as a named object in `ppJSON` If `ppJSON` is empty - creates it. If `ppJSON` does not contain the element - adds it. If `ppJSON` does contain the element - replaces it.
- `_TRFAPI trfStatus trfAddJSONElementToArray` (`char **ppJSON`, `const char *pArrayName`, `const char *pElement`)
Add given parameter (pre-constructed JSON) to the given array element in `ppJSON` If `ppJSON` is empty - creates it. If `ppJSON` does not contain the array - adds it. If `ppJSON` already contains the array - adds to it.
- `_TRFAPI trfStatus trfReadJSONElementFromArray` (`char **ppJSON`, `const char *pArrayName`, `char **ppFirstElement`)

Extract the first element in the passed ppJSON to be interpreted as a JSON array, removing that item from the array and returning it in *ppElement. If on return *ppElement is nullptr,.

- `_TRFAPI trfStatus trfReadNumericParameter` (char **ppJSON, const char *pParameterName, `_float32` *pfValue)
Read named numeric parameter from ppJSON.
- `_TRFAPI trfStatus trfReadIntegerParameter` (char **ppJSON, const char *pParameterName, `_int32` *piValue)
Read named numeric parameter from ppJSON.
- `_TRFAPI trfStatus trfReadLongIntegerParameter` (char **ppJSON, const char *pParameterName, `_int64` *piValue)
Read named numeric parameter from ppJSON.
- `_TRFAPI trfStatus trfReadTextParameter` (char **ppJSON, const char *pParameterName, char **ppvValue)
Read named textual parameter from ppJSON.
- `_TRFAPI trfStatus trfReadBooleanParameter` (char **ppJSON, const char *pParameterName, bool *pbValue)
Read named boolean parameter from ppJSON.
- `_TRFAPI trfStatus trfReadJSONElement` (const char *ppJSON, const char *pElementName, char **ppElement)
*Read a named element from the passed JSON string into *ppElement. The element itself must be either a JSON object, or a JSON array. If ppJSON does not contain the element - *ppElement will be nullptr on exit and the function will return trfUnknownParameter.*
- `_TRFAPI trfStatus trfAddFCentre` (char **ppJSON, `_float32` fFCentreHz)
Add Centre frequency parameter to ppJSON. Centre frequency parameter is required by IQ Streams.
- `_TRFAPI trfStatus trfAddIFBWHZ` (char **ppJSON, `_float32` fFIFBWHZ)
Add IF Bandwidth (IFBW) parameter to ppJSON. IFBW is required by IQ Streams.
- `_TRFAPI trfStatus trfAddFMinMax` (char **ppJSON, `_float32` fFMinHz, `_float32` fFMaxHz)
Add min/max frequency range to ppJSON. The min and max frequencies specified are required by Spectrum streams.
- `_TRFAPI trfStatus trfAddRBWHZ` (char **ppJSON, `_float32` fRBWHZ)
Add resolution bandwidth parameter to ppJSON. RBW parameter is required by spectrum streams.
- `_TRFAPI trfStatus trfAddWindowType` (char **ppJSON, const char *pWindow)
Add window type specification to ppJSON. A window type is optional for spectrum streams.
- `_TRFAPI trfStatus trfAddRefLeveldBm` (char **ppJSON, `_float32` fRefLeveldBm)
Add reference level to ppJSON The reference level is the strongest expected signal. Setting the reference level instructs a receiver to optimize its signal path, attenuation, gain settings to maximize available dynamic range for this expected signal level. A reference level is optional but recommended for IQ and Spectrum streams.
- `_TRFAPI trfStatus trfAddIQCaptureSec` (char **ppJSON, `_float32` fIQCaptureSec)
Add capture duration parameter ppJSON The duration is expressed in seconds and for an IQ stream represents the amount of data in seconds to be captured. After this capture period has expired from attaching to a device, the stream will automatically detach from the device and report 'trfExhausted' once all captured data has been retrieved. All devices will attempt to meet fIQCaptureSec at best-effort. In the case that the captured time will fit in internal buffering, the captured IQ sequence is guaranteed to be contiguous. If a stream duration is short enough to be fully buffered, the read-only stream parameter 'contiguous' will be present (and reads as 'true' (boolean)). A stream for which the 'contiguous' flag is not present may contain discontinuities. The read-only parameter 'maxContiguousSec' (`_float32`) gives the maximum capture length that can fit in device buffering and thus for which the 'contiguous' flag will be set with the current capture parameters.
- `_TRFAPI trfStatus trfReadFCentre` (char **ppJSON, `_float32` *pfFCentreHz)
Read Centre frequency parameter from ppJSON.
- `_TRFAPI trfStatus trfReadIFBWHZ` (char **ppJSON, `_float32` *pfFIFBWHZ)
Read IFBW parameter from ppJSON.
- `_TRFAPI trfStatus trfReadFMinMax` (char **ppJSON, `_float32` *pfFMinHz, `_float32` *pfFMaxHz)
Read min/max frequency range from ppJSON.
- `_TRFAPI trfStatus trfReadRBWHZ` (char **ppJSON, `_float32` *pfRBWHZ)

Read resolution bandwidth parameter from ppJSON.

- `_TRFAPI trfStatus trfReadWindowType` (char **ppJSON, char **ppWindow)

Read window type parameter from ppJSON.

- `_TRFAPI trfStatus trfReadSampleRateHz` (char **ppJSON, `_float32` *pfSampleRateHz)

Read sample rate parameter from ppJSON.

- `_TRFAPI trfStatus trfReadRefLeveldBm` (char **ppJSON, `_float32` *pfRefLeveldBm)

Read reference level parameter from ppJSON.

6.1.1 Macro Definition Documentation

TRFAddRange

```
#define TRFAddRange "addRange"
```

W - specification for range follows.

TRFAddress

```
#define TRFAddress "address"
```

RO network address tag.

TRFAltitude

```
#define TRFAltitude "altitude"
```

RO parameter - Altitude returned by GNSS.

TRFAMSubtype

```
#define TRFAMSubtype "amType"
```

RW parameter - type of AM demodulation to perform; [DSB(default), LSB, USB].

TRFAntenna

```
#define TRFAntenna "antenna"
```

RW parameter - associated antenna table.

TRFAttenuation

```
#define TRFAttenuation "attenuation"
```

RO parameter - Attenuation used in dB.

TRFAverageCount

```
#define TRFAverageCount "average"
```

RW parameter - number of spectra over which to create 'average' spectrum.

TRFBandwidthHz

```
#define TRFBandwidthHz "bandwidthHz"
```

RO generic bandwidth parameter.

TRFBaseband

```
#define TRFBaseband "baseband"
```

RO - Baseband group in recorded file.

TRFBitDepth

```
#define TRFBitDepth "bitDepth"
```

RW - Bit-depth size 8,10,12,16.

TRFBufferFrames

```
#define TRFBufferFrames "bufferFrames"
```

RW parameter - user-settable maximum stream buffering in frames. If set this will override TRFBufferSec.

TRFBufferSec

```
#define TRFBufferSec "bufferSec"
```

RW parameter - user-settable maximum buffering duration in seconds. If set this will override TRFBufferFrames.

TRFChannels

```
#define TRFChannels "channels"
```

RO parameter - Channels of data, if more than one presented (baseband).

TRFClearAverageTrace

```
#define TRFClearAverageTrace "clearAverageTrace"
```

RW parameter - On the next frame, clear the average trace.

TRFClearMaxHoldTrace

```
#define TRFClearMaxHoldTrace "clearMaxHoldTrace"
```

RW parameter - On the next frame, clear the max-hold trace.

TRFClearMinHoldTrace

```
#define TRFClearMinHoldTrace "clearMinHoldTrace"
```

RW parameter - On the next frame, clear the min-hold trace.

TRFCurrentDate

```
#define TRFCurrentDate "date"
```

RW parameter - current date as year,month,date.

TRFCurrentTime

```
#define TRFCurrentTime "time"
```

RW parameter - current time as hour, minute, second [,millisecond].

TRFDataType

```
#define TRFDataType "dataType"
```

RO - DataType param in recorded file.

TRFdBFS

```
#define TRFdBFS "dbfsd"
```

RO parameter - Proportion of dynamic range utilized - expressed in dB relative to full-scale.

TRFDCOffsetFilter

```
#define TRFDCOffsetFilter "dcOffsetFilter"
```

RW parameter - default 'true'. Removes residual DC offset signal from IQ data.

TRFDefault

```
#define TRFDefault "default"
```

Generic JSON parameter names for libtrf entities; devices, streams, processors. These parameter names are used in conjunction with JSON string construction/ examination functions to allow parameters to be set for and obtained from libtrf entities. In the following documentation 'RO' denotes 'read-only', and RW denotes read/write. Setting a parameter for an entity that does not support that parameter is an error and will typically return `trfParameterSetError` when calling `trfSetParameters` with a JSON string including an unsupported parameter. Detailed information can be obtained using `trfGetDetailedStatus`. Note that these names may change across future releases and therefore it is highly recommended that the following symbolic names `TRF__` are used instead of their textual equivalent. RO - Read-only, RW - Read and Write.

RO default parameter value tag

TRFDetector

```
#define TRFDetector "detector"
```

RW parameter - Detector type for spectrum reconstruction.

TRFDevice

```
#define TRFDevice "device"
```

RO device information section.

TRFDeviceConnection

```
#define TRFDeviceConnection "connection"
```

RO parameter - device textual connection information.

TRFDeviceFirmware

```
#define TRFDeviceFirmware "firmware"
```

RO parameter - device textual firmware version.

TRFDeviceLockStatus

```
#define TRFDeviceLockStatus "deviceClockLock"
```

RO parameter - Reported device clock lock statuses.

TRFDeviceSerialNum

```
#define TRFDeviceSerialNum "serial"
```

RO parameter - device textual serial number.

TRFDeviceTemperature

```
#define TRFDeviceTemperature "deviceTemperature"
```

RO parameter - Reported device internal temperatures.

TRFDeviceType

```
#define TRFDeviceType "type"
```

RO parameter - device textual type.

TRFDeviceVersion

```
#define TRFDeviceVersion "version"
```

RO parameter - device textual version.

TRFDigitalGain

```
#define TRFDigitalGain "digitalGain"
```

RW - Digital Gain in dB, for 5xx0 products only.

TRFDiscardProportion

```
#define TRFDiscardProportion "bufferDiscard"
```

RW parameter - On overflow (TRFBufferSec, TRFBufferFrames) this sets the proportion of the buffer that will be discarded. The oldest frames will be discarded first. Setting to zero will minimize the number of frames discarded. Setting to 1.0 will discard the whole buffer. All other values 0,1 are acceptable. This allows a user to manage the tradeoff between discontinuities and data latency.

TRFDurationFrames

```
#define TRFDurationFrames "captureFrames"
```

RW parameter - Optional capture duration in frames (0 indicates continuous capture).

TRFDurationSec

```
#define TRFDurationSec "captureSec"
```

RW parameter - IQ Stream specified capture duration.

TRFDwellUsec

```
#define TRFDwellUsec "dwellUsec"
```

RW parameter - minimum dwell time usec per MHz.

TRFEncoding

```
#define TRFEncoding "encoding"
```

RO - Encoding param in recorded file.

TRFEndSequence

```
#define TRFEndSequence "endSequence"
```

RO - sequence number of last frame in event.

TRFEndTimeStamp

```
#define TRFEndTimeStamp "endTimestamp"
```

RO - EndTimestamp param in recorded file.

TRFEqualization

```
#define TRFEqualization "eq"
```

RW parameter - Equalization to apply to the stream. Names a pre-loaded equalization table.

TRFEventDuration

```
#define TRFEventDuration "duration"
```

RO - observed event duration (milliseconds)

TRFEventID

```
#define TRFEventID "id"
```

RO - unique event ID.

TRFExceeddB

```
#define TRFExceeddB "exceeddB"
```

RW - alarm 'exceeds' threshold by this amount.

TRFExceedNotLoss

```
#define TRFExceedNotLoss "exceedNotLoss"
```

RO - boolean indicating if this is an exceedance or loss event.

TRFExceedRecords

```
#define TRFExceedRecords "exceed"
```

RO - observed event records.

TRFFCentreHz

```
#define TRFFCentreHz "fCentreHz"
```

RW parameter - IQ Capture centre frequency.

TRFFilename

```
#define TRFFilename "filename"
```

RW parameter - filename parameter (stream)

TRFFilterRatio

```
#define TRFFilterRatio "ifFilterRatio"
```

RO parameter - IF Filter ratio applied.

TRFFlattenFlag

```
#define TRFFlattenFlag "flatten"
```

RW parameter - apply spectrum equalization if available. By default this flag is set 'false'.

TRFFlowControl

```
#define TRFFlowControl "flowControl"
```

RW parameter - If set, ensures tight flow control with a receiver for spectrum data.

TRFFMaxHz

```
#define TRFFMaxHz "fMaxHz"
```

RW parameter - Max extent of sweep range.

TRFFMinHz

```
#define TRFFMinHz "fMinHz"
```

RW parameter - Min extent of sweep range.

TRFFollowIQ

```
#define TRFFollowIQ "followIQ"
```

RW parameter - if 'true', then one spectrum frame will be produced for each input IQ frame with RBW dictated by the window function and input IQ frame size.

TRFFFrameDuration

```
#define TRFFFrameDuration "frameDuration"
```

RO parameter - duration of a recorded frame in time units.

TRFFFrameInFile

```
#define TRFFFrameInFile "frameInFile"
```

Number of this frame within a file (playback stream only)
guaranteed contiguous (TRFIsContiguous is set) then this number will be the exact number of frames delivered before exhaustion. Otherwise this will be a lower-bound on the number of frames delivered before the stream is terminated by the API.

TRFFrameOutputEnabled

```
#define TRFFrameOutputEnabled "frameOutput"
```

RW - if true, alarms processor frame output is active.

TRFFramesExpected

```
#define TRFFramesExpected "framesExpected"
```

RO parameter - Number of frames expected in this stream, or 0 if configured for continuous capture. If the stream is.

TRFFramesProduced

```
#define TRFFramesProduced "frames"
```

RO parameter - count of frames produced.

TRFFrequency

```
#define TRFFrequency "freq"
```

RW - range frequency point.

TRFFullAdaptFlag

```
#define TRFFullAdaptFlag "adaptFull"
```

RW parameter - Full adapt flag - if set on stream will cause DR adaption before IQ streaming (~500msec)

TRFGaindB

```
#define TRFGaindB "gaindB"
```

RW parameter - Gain to apply to the channel.

TRFGatedOutput

```
#define TRFGatedOutput "gated"
```

RW - if true, alarms processor only passes on spectra in case of exceedance.

TRFGNSSADelay

```
#define TRFGNSSADelay "gnssAntDelay"
```

RW parameter - GNSS Antenna delay parameter.

TRFGNSSConstellation

```
#define TRFGNSSConstellation "gnssCons"
```

RW parameter - GNSS Constellation to track format: "<option 1>[,option 2]".

TRFGNSSDynamicMode

```
#define TRFGNSSDynamicMode "gnssDynamic"
```

RW parameter - GNSS Dynamic mode (textual)

TRFGNSSHeadingDegT

```
#define TRFGNSSHeadingDegT "gnssHeading"
```

RO parameter - GNSS Heading in degrees true.

TRFGNSSInfo

```
#define TRFGNSSInfo "gnssInfo"
```

RO - GNSSInfo group in recorded file.

TRFGNSSMagVarDegT

```
#define TRFGNSSMagVarDegT "gnssMagVar"
```

RO parameter - GNSS Magnetic variation in degrees E+,W-.

TRFGNSSRxTime

```
#define TRFGNSSRxTime "gnssPosnEpoch"
```

RO parameter - Last position report in UTC Msec.

TRFGNSSSpeedOverGround

```
#define TRFGNSSSpeedOverGround "gnssSpeed"
```

RO parameter - GNSS Speed over ground in m/sec.

TRFGNSSTimestamp

```
#define TRFGNSSTimestamp "gnssTimeNanosec"
```

RO parameter - Most recent GNSS UTC timestamp in nanosec.

TRFGNSSTrackDegT

```
#define TRFGNSSTrackDegT "gnssTrack"
```

RO parameter - GNSS Track in degrees true.

TRFGNSSValid

```
#define TRFGNSSValid "gnssValid"
```

RO parameter - true if GNSS data is valid, false otherwise.

TRFGPIO

```
#define TRFGPIO "gpio"
```

RW parameter - Explicitly set GPIO bits (GPIO [0:2]) now.

TRFHasFlatteningInfo

```
#define TRFHasFlatteningInfo "iqEqualization"
```

RO receiver flag that equalization data is available (flattening is possible)

TRFHasGNSS

```
#define TRFHasGNSS "hasGNSS"
```

RO parameter - true if Receiver has a GNSS module.

TRFIFBWHz

```
#define TRFIFBWHz "ifbwHz"
```

RW parameter - IF Bandwidth for IQ capture.

trfInvalidHandle

```
#define trfInvalidHandle (0xffffffff)
```


Invalid handle value for any libtrf entity. As good practice, all unused or uninitialized handle values should as good practice be set to this value - ie.

```
trfHandle hNewDevice(trfInvalidHandle);
```

trfInvalidSequence

```
#define trfInvalidSequence (0xfffffffffe)
```

Invalid sequence number for any frame type.

TRFIOnly

```
#define TRFIOnly "iOnly"
```

RO - I-only datatype.

TRFIQ

```
#define TRFIQ "iq"
```

RO - IQ group in recorded file.

TRFIQActive

```
#define TRFIQActive "iqActive"
```

RO parameter - IQ capture is active flag (Rx)

TRFIQCapture

```
#define TRFIQCapture "iqCapture"
```

RW parameter - data structure holding complete IQ capture specification.

TRFIsContiguous

```
#define TRFIsContiguous "contiguous"
```

RO parameter - Indicates this IQ stream is guaranteed contiguous.

TRFLANMTU

```
#define TRFLANMTU "lanMtu"
```

RW parameter - LAN MTU (maximum transfer unit, in bytes) value.

TRFLastSequence

```
#define TRFLastSequence "lastSequence"
```

RO parameter - sequence number of last processed frame.

TRFLatitude

```
#define TRFLatitude "latitude"
```

RO parameter - Latitude returned by GNSS.

TRFLevel

```
#define TRFLevel "level"
```

RW - range level point.

TRFLongitude

```
#define TRFLongitude "longitude"
```

RO parameter - Longitude returned by GNSS.

TRFLooping

```
#define TRFLooping "loopingFlag"
```

RW set to cause file to loop.

TRFMatchIQ

```
#define TRFMatchIQ "matchIQ"
```

RO parameter - Most recently processed input IQ frame sequence number.

TRFMax

```
#define TRFMax "max"
```

RO maximum value tag.

TRFMaxContiguousSec

```
#define TRFMaxContiguousSec "maxContiguousSec"
```

RO parameter - IQ Stream max limit of continuous streaming. Note -only valid after attaching the stream to a receiver device.

TRFMeasuredAvgdBm

```
#define TRFMeasuredAvgdBm "avgDbm"
```

RO parameter - Measured average power - only valid if TRFRefLevelAdapt is true.

TRFMin

```
#define TRFMin "min"
```

RO minimum value tag.

TRFMinimumPeakdBm

```
#define TRFMinimumPeakdBm "minPeakdBm"
```

RW parameter - minimum power for any returned peak.

TRFModifyRange

```
#define TRFModifyRange "modifyRange"
```

W - modify specified range.

TRFName

```
#define TRFName "name"
```

TRFNTP

```
#define TRFNTP "ntpServers"
```

RW parameter - NTP servers list.

TRFOBOccupiedPercentage

```
#define TRFOBOccupiedPercentage "occupiedPercentage"
```

RO - Percentage of the total integrated power.

TRFObserveSeconds

```
#define TRFObserveSeconds "observeSec"
```

RW - observation time for dynamic thresholds.

TRFOutputFrameSize

```
#define TRFOutputFrameSize "outputFrameSize"
```

RW parameter - requested baseband output frame size.

TRFOutputSampleRate

```
#define TRFOutputSampleRate "outputSampleRate"
```

RW parameter - requested baseband output sample rate.

TRFOutputSize

```
#define TRFOutputSize "outputSize"
```

RO parameter - Size of output spectra expected with current parameters.

TRFOverlap

```
#define TRFOverlap "fftOverlap"
```

RW parameter - sets the proportion 0.0-1.0 of overlap in successive output spectra.

TRFPacketDurationMsec

```
#define TRFPacketDurationMsec "packetDuration"
```

RO parameter - Packet duration in seconds.

TRFPeakCount

```
#define TRFPeakCount "peakCount"
```

RW parameter - maximum number of peaks to return.

TRFPeakdBm

```
#define TRFPeakdBm "peakLeveldBm"
```

element of TRFPeaksVector - peak level in dBm

TRFPeakHz

```
#define TRFPeakHz "peakFrequencyHz"
```

element of TRFPeaksVector - peak frequency in Hz

TRFPeaks

```
#define TRFPeaks "peaks"
```

RO parameter - vector of detected peaks of the form {TRFPeakHz: peakFHz, TRFPeakdBm: peakLeveldBm}.

TRFPLLSource

```
#define TRFPLLSource "pllSource"
```

RW parameter - PLL clock source (internal, external, gnss)

TRFPointsPerFrame

```
#define TRFPointsPerFrame "pointsPerFrame"
```

RO - PointsPerFrame param in recorded file.

TRFPPSSource

```
#define TRFPPSSource "ppsSource"
```

RW parameter - PPS clock source (external, gnss)

TRFRangeActive

```
#define TRFRangeActive "active"
```

RW - range active flag.

TRFRangeName

```
#define TRFRangeName "name"
```

RW - name of range.

TRFRBWHz

```
#define TRFRBWHz "rbwHz"
```

RW parameter - Resolution Bandwidth in Hz.

TRFRecentFrames

```
#define TRFRecentFrames "recentFrames"
```

RO parameter - Number of frames processed since last trfGetParameters call.

TRFReferenceLeveldBm

```
#define TRFReferenceLeveldBm "referenceLeveldBm"
```

RO - ReferenceLeveldBm param in recorded file.

TRFRefLevel

```
#define TRFRefLevel "refdBm"
```

RW parameter - User-supplied expected max signal dBm.

TRFRemoveRange

```
#define TRFRemoveRange "removeRange"
```

W - remove specified range.

TRFRFFStart

```
#define TRFRFFStart "fStart"
```

RW parameter - start of frequency range.

TRFRFFStop

```
#define TRFRFFStop "fStop"
```

RW parameter - end of frequency range.

TRFRFGPIO

```
#define TRFRFGPIO "gpio"
```

RW parameter - associated GPIO setting (GPIO [0:2])

TRFRFRange

```
#define TRFRFRange "range"
```

RW parameter - identifies RF path GPIO range group follows.

TRFRFSelect

```
#define TRFRFSelect "rfPath"
```

RW parameter - identifies RFPath control group follows.

TRFRxSampleRate

```
#define TRFRxSampleRate "rxSampleRate"
```

RO parameter - native 'base' sample rate of a receiver.

TRFSampleRateHz

```
#define TRFSampleRateHz "sampleRateHz"
```

RO parameter - IQ Capture sample rate.

TRFSamples

```
#define TRFSamples "samples"
```

RO - Samples param in recorded file.

TRFSampling

```
#define TRFSampling "sampling"
```

RO - Sampling param in recorded file.

TRFSCPIQueryTimeout

```
#define TRFSCPIQueryTimeout "scpiQueryTimeout"
```

RW parameter - SCPI transaction timeout.

TRFSequence

```
#define TRFSequence "sequence"
```

RO sequence number parameter (internal)

TRFSpan

```
#define TRFSpan "spanHz"
```

RW parameter - span over which to integrate power.

TRFSpectrum

```
#define TRFSpectrum "spectrum"
```

RO - Spectrum group in recorded file.

TRFSpurReject

```
#define TRFSpurReject "spurReject"
```

RW parameter - default 'false'. Perform spur suppression (R5xx0 only)

TRFStartSequence

```
#define TRFStartSequence "startSequence"
```

RO - sequence number of first frame in event.

TRFStartTimeStamp

```
#define TRFStartTimeStamp "startTimestamp"
```

RO - StartTimestamp param in recorded file.

TRFStepAdaptFlag

```
#define TRFStepAdaptFlag "adaptStep"
```

RW parameter - Step adapt flag - if set on stream will cause DR adaption before IQ streaming.

TRFSubOptimalFlag

```
#define TRFSubOptimalFlag "subOptimalDr"
```

RO parameter - Dynamic range is non-optimal flag.

TRFSustaindB

```
#define TRFSustaindB "sustaindB"
```

RW - No alarm if any signal sustains above this threshold.

TRFSweepActive

```
#define TRFSweepActive "sweepActive"
```

RO parameter - sweep is active flag (Rx)

TRFTable

```
#define TRFTable "table"
```

TRFThreshold

```
#define TRFThreshold "threshold"
```

RW - specification (scalar, vector) for upper threshold.

TRFTimeConstant

```
#define TRFTimeConstant "timeConstant"
```

RW parameter - averaging time constant.

TRFTimeResolution

```
#define TRFTimeResolution "timeResolution"
```

RO time resolution parameter (internal)

TRFTimestamp

```
#define TRFTimestamp "timestamp"
```

RO timestamp parameter (internal)

TRFTimeSync

```
#define TRFTimeSync "timeSync"
```

RW parameter - clock reference source (disable,"ntp,once","ntp,continuous")

TRFTrainSeconds

```
#define TRFTrainSeconds "trainSec"
```

RW - pre-training time for static thresholds.

TRFType [1/2]

```
#define TRFType "type"
```

RO type of unit tag (user readable)

TRFType [2/2]

```
#define TRFType "type"
```

RO type of unit tag (user readable)

TRFUnits

```
#define TRFUnits "units"
```

RW parameter - Units of data ("mW", "W", "V", "A", "dBm", "dBW", "dBV", "dBmV", "dBuV", "dBA", "dBmA", "dBuA", "dBuV/m", "dBW/m2")

TRFUsableBWHz

```
#define TRFUsableBWHz "usableBwHz"
```

RO parameter - Usable bandwidth (baseband).

TRFUserCalibrationOffset

```
#define TRFUserCalibrationOffset "userCalDb"
```

RW parameter - User-supplied correction-factor for both Spectrum and calibrated-IQ.

TRFValue

```
#define TRFValue "value"
```

TRFValues

```
#define TRFValues "values"
```

TRFVBWHz

```
#define TRFVBWHz "vbwHz"
```

RW parameter - Video Bandwidth in Hz.

TRFWeightedFilter

```
#define TRFWeightedFilter "weightedFilter"
```

RW parameter - Defines smoothing aggressiveness 0 to 1.

TRFWindow

```
#define TRFWindow "windowFn"
```

RW parameter - Window function for spectrum reconstruction.

TRFWriteStartSequence

```
#define TRFWriteStartSequence "startSequence"
```

RW sequence number for first frame to record.

TRFWriteStopSequence

```
#define TRFWriteStopSequence "stopSequence"
```

RW sequence number for last frame to record.

6.1.2 Typedef Documentation

`_float32`

```
typedef float _float32
```

32-bit floating point (IEEE-754) value

`_float64`

```
typedef double _float64
```

64-bit floating point (IEEE-754) value

`_uint`

```
typedef unsigned int _uint
```

Unsigned integer.

`_uint16`

```
typedef unsigned short _uint16
```

Unsigned 16-bit integer.

`_uint32`

```
typedef unsigned int _uint32
```

Unsigned 32-bit integer.

`_uint64`

```
typedef unsigned long long int _uint64
```

Signed 64-bit integer.

`_uint8`

```
typedef unsigned char _uint8
```

Basic types used in the API.

Unsigned 8-bit integer

`trfBasebandFrame`

```
typedef struct _trfBasebandFrame trfBasebandFrame
```

Baseband sampled data (ie. audio) frame. Basedband data can be of any type representable by an array of `_float32` values. Multiple channels are supported.

trfHandle

```
typedef _uint32 trfHandle
```

Device or processor handle type.

Handle for accessing libtrf resource (device, processor etc...)

trfIQFrame

```
typedef struct _trfIQFrame trfIQFrame
```

IQ data frame. This structure contains the actual data and all descriptive metadata.

trfSpectrumFrame

```
typedef struct _trfSpectrumFrame trfSpectrumFrame
```

Spectrum data frame. This structure contains the actual spectral data and all descriptive metadata.

trfStatus

```
typedef enum _trfStatus trfStatus
```

General operation status reporting.

6.1.3 Enumeration Type Documentation

_trfStatus

```
enum _trfStatus
```

General operation status reporting.

Enumerator

trfOk	No errors, no warnings - operation has completed successfully.
trfWaiting	Warnings or status (greater than 0) No error - data delivery is underway but momentarily unavailable
trfExhausted	No more data is available.
trfNotStarted	Data is expected but is not yet available.
trfDiscontinuousWithPreviousFrame	Indicator that a discontinuity is detected in IQ or baseband data.
trfDetached	Indicates that a stream is in a 'detached' state.
trfPingFailed	Warning that a 'ping' attempt failed.

6.1 libtrf.h File Reference

Enumerator

trfDuplicateAddress	Duplicate address specified.
trfUnimplemented	Errors codes (less than 0) Function or behaviour is not currently available
trfAPINotInitialized	API initialization is required before executing this function.
trfDeviceAddressAlreadyKnown	An already-known device address has been added.
trfUnallocatedUserData	Expected user-allocated storage has not been provided.
trfDeviceIndexOutOfRange	Provided device index is not in the range of enumerated devices.
trfNoDeviceInformation	No information is available for the specified device.
trfCannotOpenDevice	Unable to open the specified device.
trfBadDeviceHandle	The given device handle is invalid or corrupted.
trfBadUnitHandle	The given entity handle is invalid or corrupted.
trfNoParametersSpecified	No parameters where parameters were expected have been specified.
trfBadReceiverHandle	The given receiver handle is invalid or corrupted.
trfIQStreamStartFailure	The IQ stream has failed to start.
trfMemoryAllocationError	An internal memory allocation has failed.
trfInvalidStreamHandle	The given stream handle is invalid or corrupted.
trfNoAssociatedRxOrFile	The specified stream has no associated data source.
trfNotAnIQStream	An IQ-specific call is made on a non IQ-stream entity handle.
trfNoDataSpecified	No data has been specified where data was expected.
trfInvalidHandleSpecified	The given entity handle is invalid or corrupted.
trfBadStreamShutdown	An error was encountered during stream shutdown.
trfUnsupportedParameter	A parameter was specified which is not supported.
trfCannotCloseStream	An error occurred preventing the specified stream being closed.
trfNotASpectrumStream	A stream handle for a non-spectrum stream was specified.
trfCannotRestartStream	A failure has occurred preventing the stream from restarting.
trfSweepStartFailure	A failure has occurred preventing the spectrum stream from starting.
trfFrameTypeMismatch	An incorrect frame type has been specified.
trfFileOpenFailed	A failure has occurred when attempting to open a file.
trfFileTypesIsNotIQ	The file specified was expected to contain IQ data.
trfFileTypesIsNotSpectrum	The file specified was expected to contain spectra.
trfNoAssociatedSource	No source is associated with this entity.
trfUnknownParameter	The parameter specified is not known.
trfParameterSetError	An error was encountered when setting a parameter.
trfDeviceHandleIsInvalid	An invalid or corrupted device handle was specified.
trfSCPISendFailed	An error in sending SCPI data has occurred.
trfSCPIQueryFailed	An error in making a SCPI query has occurred.
trfBadStreamHandle	An invalid or corrupted stream handle was specified.
trfCannotAttachIQStream	A failure occurred when attempting to attach an IQ stream.
trfCannotAttachSpectrumStream	A failure occurred when attempting to attach a spectrum stream.
trfCannotAttachStream	A failure occurred when attempting to attach a stream.
trfCannotDetachIQStream	A failure occurred when attempting to detach an IQ stream.
trfCannotDetachSpectrumStream	A failure occurred when attempting to detach a spectrum stream.
trfCannotDetachStream	A failure occurred when attempting to detach a stream.

Enumerator

trfConnectionResetFailed	Failed to reset the connection to the specified device.
trfBadFrequencyParameters	'Impossible' frequency parameters specified (ie. FStart >= FEnd)
trfBadRBWParameter	'Impossible' RBW parameter specified
trfBadReferenceLevelParameter	'Impossible' reference level parameter specified
trfSpectrumStreamCreateFailure	General failure to create a spectrum stream.
trfBufferSecOutOfRange	Buffer seconds parameter specified is invalid.
trfBufferFramesOutOfRange	Buffer frames parameter specified is invalid.
trfBufferDiscardProportionOutOfRange	Buffer discard proportion parameter specified is out of range (valid is 0.0 to 1.0)
trfFrequencyRangelsInvalid	Invalid frequency range specified.
trfBadGNSSDynamicMode	Invalid GNSS dynamic mode specified.
trfBadPLLSource	Invalid PLL Source parameter specified.
trfPLLSourceWithNoGNSS	PLL source specified for a non-GNSS enabled device.
trfCentreInvalid	Centre frequency specified is invalid.
trfIFBWInvalid	IFBW specified is invalid.
trfFMinInvalid	Minimum frequency specified is invalid.
trfFMaxInvalid	Maximum frequency specified is invalid.
trfResolutionBWInvalid	Resolution bandwidth specified is invalid.
trfCentreOutOfRange	Centre frequency specified is out of range for this device.
trfIFBWOutOfRange	IFBW specified is out of range for this device.
trfReferenceLevelOutOfRange	Reference level is out-of-range for this device.
trfResolutionBWOutOfRange	Resolution bandwidth is out-of-range for this device.
trfUnknownWindowType	Window type specified is not supported by this device.
trfStartFrequencyOutOfRange	Minimum frequency specified is not supported by this device.
trfStopFrequencyOutOfRange	Maximum frequency specified is not supported by this device.
trfDeviceUnreachable	Unable to connect with device through network address.
trfInvalidFilename	Filename passed does not specify a valid file path.
trfCannotOpenOutputFile	Failed to open specified output file for writing.
trfUninitializedUserData	Expected data was found uninitialized.
trfBadProcessorHandle	Unknown processor handle specified.
trfCannotCreateProcessor	Failed to create processor.
trfProcessorAttachFailed	Attempt to attach processor to stream failed (wrong stream type?)
trfProcessorDetachFailed	Attempt to detach processor from source stream failed.
trfProcessorUnattached	Attempt to detach processor when not attached.
trfNotABasebandStream	Stream handle given is not of a baseband stream.
trfFileTypesNotBaseband	Stream file given is not a baseband file.
trfBasebandStreamStartFailure	Failed to start baseband file stream from file.
trfSampleRateInvalid	Invalid sample rate specified.
trfCannotObtainOutputStream	Unable to obtain processor output stream handle.
trfCannotConnectOutputStream	Failed to connect output stream to processor.
trfBadFilename	Filename / file path is invalid.
trfWindowTypeInvalid	Window type specified is not known.
trfDeviceCommunicationsLost	Device is not responding over network.

6.1 libtrf.h File Reference

Enumerator

trfUnknownStreamType	Stream type is not known - internal error (should never happen)
trfStreamHandleAlreadyOpen	Stream handle is already open - attempt to re-open failed.
trfInvalidStreamSource	Stream source is invalid inside a connection operation.
trfBadPPSSource	Invalid PPS Source parameter specified.
trfStreamFailedToStart	A stream failed to start for some undefined reason.
trfStreamParametersError	A stream failed to start because of bad programming.
trfBasebandStreamCreateFailure	Could not create a baseband stream.
trfInvalidIP	Invalid IP address specified.
trfUnknownAddress	Specified address is not in the list of known addresses.
TRFBitDepthOutOfRange	Specified bit-packing parameter is out of valid range.
trfFatalReceiverError	A receiver has encountered a fatal error (ie. loss of comms)
trfDigitalGainOutOfRange	Specified digital gain parameter is out of valid range.
trfJSONEmpty	Specified JSON string contains no elements.
trfNoUniqueArrayParameter	Specified JSON string does not contain a single array element.
trfNoArrayParameter	Specified JSON element name does not match to an array element.
trfInvalidArrayIndex	Specified JSON array index is out-of-range.
trfNoOp	With the specified parameters, there is no work to be done.
trfSpectrumCompressFailed	Failed to compress the spectrum. Undefined error.
trfOnCallPointerShouldBeNull	Value indirected through pointer value is not null.
trfBadJSONElement	Element passed is not valid JSON.
trfBadJSONStructure	Structure passed (primary argument) is not valid JSON.
trfBadJSONName	Expected a valid name, and found nothing.
trfUnknownDetectorType	Detector type is not known.
trfDwellTimeOutOfRange	Specified dwell time is invalid.
trfVideoBWOutOfRange	Specified VBW is out of range.
trfUnitsInvalid	Specified units are not recognized.
trfGainInvalid	Invalid gain value.
trfEQInvalid	Invalid equalizer name.
trfVBWInvalid	Video bandwidth specified is invalid.
trfDwellTimeInvalid	Dwell time specified is invalid.
trfAntennaInvalid	The named antenna does not exist.
trfEqualizerInvalid	The named equalizer does not exist.
trfRebootedStatus	Special error code to be used if receiver is forcefully rebooted.
trfGenericFailStatus	Generic 'failed' status code.
trfLastErrorSentinel	

trfDecimationType

enum `trfDecimationType`

Type of decimation to perform, selecting peak, minimum or average from a set of points as the combined result point.

Enumerator

trfMaximum	
trfMinimum	
trfLogAverage	
trfLinAverage	

6.1.4 Function Documentation**trfAddArrayElement()**

```
_TRFAPI trfStatus trfAddArrayElement (
    char ** ppJSON,
    const char * pArrayName,
    const char * pJSONElement )
```

Add contained JSON element to the named array contained in ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string - accumulating parameters.
<i>pArrayName</i>	Name of array to add to.
<i>pJSONElement</i>	JSON element to add.

Returns

error/warning information (on success == trfOk)

trfAddBooleanParameter()

```
_TRFAPI trfStatus trfAddBooleanParameter (
    char ** ppJSON,
    const char * pParameterName,
    bool bValue )
```

Add named boolean parameter to ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string - accumulating parameters.
<i>pParameterName</i>	Name of parameter to add.
<i>bValue</i>	Boolean value to add.

Returns

error/warning information (on success == trfOk)

trfAddDeviceAddress()

```
_TRFAPI trfStatus trfAddDeviceAddress (
    const char * pRemoteAddress )
```

Add a potential device address for use by trfEnumerateDevices. Note that local devices that may be discovered directly (LAN, USB3 etc...) will not require their addresses to be added through this call, however it is not an error if they are.

Parameters

<i>pRemoteAddress</i>	IPv4/IPv6 address for remote device.
-----------------------	--------------------------------------

Returns

error/warning information (on success == trfOk) If the address cannot be successfully pinged - will return the warning trfCannotPingDevice.

Example:

```
eStatus = trfAddDeviceAddress("10.126.110.114");
if ((eStatus != trfOk) && (eStatus != trfPingFailed)) {
    cout << "trfAddDeviceAddress failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) << endl;
    return false;
}
```

trfAddFCentre()

```
_TRFAPI trfStatus trfAddFCentre (
    char ** ppJSON,
    _float32 fFCentreHz )
```

Add Centre frequency parameter to ppJSON. Centre frequency parameter is required by IQ Streams.

Parameters

<i>ppJSON</i>	Pointer to JSON string - accumulating parameters.
<i>fFCentreHz</i>	Centre frequency to add to JSON parameter string.

Returns

error/warning information (on success == trfOk)

trfAddFMinMax()

```
_TRFAPI trfStatus trfAddFMinMax (
    char ** ppJSON,
```

```

_float32 fFMinHz,
_float32 fFMaxHz )

```

Add min/max frequency range to ppJSON. The min and max frequencies specified are required by Spectrum streams.

Parameters

<i>ppJSON</i>	Pointer to JSON string - accumulating parameters.
<i>fFMinHz</i>	Minimum extent frequency to add to JSON parameter string.
<i>fFMaxHz</i>	Maximum extent frequency to add to JSON parameter string.

Returns

error/warning information (on success == trfOk)

trfAddIFBWHz()

```

_TRFAPI trfStatus trfAddIFBWHz (
    char ** ppJSON,
    _float32 fFIFBWHz )

```

Add IF Bandwidth (IFBW) parameter to ppJSON. IFBW is required by IQ Streams.

Parameters

<i>ppJSON</i>	Pointer to JSON string - accumulating parameters.
<i>fFIFBWHz</i>	IF Bandwidth parameter to add to JSON parameter string.

Returns

error/warning information (on success == trfOk)

trfAddIntegerParameter()

```

_TRFAPI trfStatus trfAddIntegerParameter (
    char ** ppJSON,
    const char * pParameterName,
    int iValue )

```

For parameters not named above - add parameter by name Add named numeric parameter to ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string - accumulating parameters.
<i>pParameterName</i>	Name of parameter to add.
<i>iValue</i>	Integer value to add.

Returns

error/warning information (on success == trfOk)

trfAddIQCaptureSec()

```
_TRFAPI trfStatus trfAddIQCaptureSec (  
    char ** ppJSON,  
    _float32 fIQCaptureSec )
```

Add capture duration parameter ppJSON The duration is expressed in seconds and for an IQ stream represents the amount of data in seconds to be captured. After this capture period has expired from attaching to a device, the stream will automatically detach from the device and report 'trfExhausted' once all captured data has been retrieved. All devices will attempt to meet fIQCaptureSec at best-effort. In the case that the captured time will fit in internal buffering, the captured IQ sequence is guaranteed to be contiguous. If a stream duration is short enough to be fully buffered, the read-only stream parameter 'contiguous' will be present (and reads as 'true' (boolean)). A stream for which the 'contiguous' flag is not present may contain discontinuities. The read-only parameter 'maxContiguousSec' (_float32) gives the maximum capture length that can fit in device buffering and thus for which the 'contiguous' flag will be set with the current capture parameters.

Parameters

<i>ppJSON</i>	Pointer to JSON string - accumulating parameters.
<i>fIQCaptureSec</i>	Number of seconds of data to capture.

Returns

error/warning information (on success == trfOk)

trfAddJSONElement()

```
_TRFAPI trfStatus trfAddJSONElement (  
    char ** ppJSON,  
    const char * pElementName,  
    const char * pElement )
```

Add given parameter (pre-constructed JSON) as a named object in ppJSON If ppJSON is empty - creates it. If ppJSON does not contain the element - adds it. If ppJSON does contain the element - replaces it.

Parameters

<i>ppJSON</i>	Pointer to JSON string - accumulating parameters.
<i>pArrayName</i>	Name of array to add to. Cannot be null.
<i>pElement</i>	Well-constructed JSON element.

Returns

error/warning information (on success == trfOk)

trfAddJSONElementToArray()

```
_TRFAPI trfStatus trfAddJSONElementToArray (
    char ** ppJSON,
    const char * pArrayName,
    const char * pElement )
```

Add given parameter (pre-constructed JSON) to the given array element in ppJSON. If ppJSON is empty - creates it. If ppJSON does not contain the array - adds it. If ppJSON already contains the array - adds to it.

Parameters

<i>ppJSON</i>	Pointer to JSON string - accumulating parameters.
<i>pArrayName</i>	Name of array to add to. Cannot be null.
<i>pElement</i>	Well-constructed JSON element.

Returns

error/warning information (on success == trfOk)

trfAddNumericParameter()

```
_TRFAPI trfStatus trfAddNumericParameter (
    char ** ppJSON,
    const char * pParameterName,
    _float32 fValue )
```

For parameters not named above - add parameter by name. Add named numeric parameter to ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string - accumulating parameters.
<i>pParameterName</i>	Name of parameter to add.
<i>fValue</i>	Numeric value to add.

Returns

error/warning information (on success == trfOk)

trfAddRBWHZ()

```
_TRFAPI trfStatus trfAddRBWHZ (
```

6.1 libtrf.h File Reference

```
char ** ppJSON,  
_float32 fRBWHz )
```

Add resolution bandwidth parameter to ppJSON. RBW parameter is required by spectrum streams.

Parameters

<i>ppJSON</i>	Pointer to JSON string - accumulating parameters.
<i>fRBWHz</i>	Resolution Bandwidth parameter to add to JSON parameter string.

Returns

error/warning information (on success == trfOk)

trfAddRefLeveldBm()

```
_TRFAPI trfStatus trfAddRefLeveldBm (  
    char ** ppJSON,  
    _float32 fRefLeveldBm )
```

Add reference level to ppJSON The reference level is the strongest expected signal. Setting the reference level instructs a receiver to optimize its signal path, attenuation, gain settings to maximize available dynamic range for this expected signal level. A reference level is optional but recommended for IQ and Spectrum streams.

Parameters

<i>ppJSON</i>	Pointer to JSON string - accumulating parameters.
<i>fRefLeveldBm</i>	Level in dBm of strongest expected signal.

Returns

error/warning information (on success == trfOk)

trfAddTextParameter()

```
_TRFAPI trfStatus trfAddTextParameter (  
    char ** ppJSON,  
    const char * pParameterName,  
    const char * pValue )
```

Add named textual parameter to ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string - accumulating parameters.
<i>pParameterName</i>	Name of parameter to add.
<i>pValue</i>	Textual value to add (zero-terminated string).

Returns

error/warning information (on success == trfOk)

trfAddWindowType()

```
_TRFAPI trfStatus trfAddWindowType (
    char ** ppJSON,
    const char * pWindow )
```

Add window type specification to ppJSON. A window type is optional for spectrum streams.

Parameters

<i>ppJSON</i>	Pointer to JSON string - accumulating parameters.
<i>pWindow</i>	Textual window type to add to JSON parameter string.

Returns

error/warning information (on success == trfOk)

trfAllowDiscarding()

```
_TRFAPI trfStatus trfAllowDiscarding (
    trfHandle cStream,
    bool bAllowDiscarding )
```

By default, the data contained in streams will be discarded to ensure buffer size constraints are respected. In certain cases - such as streaming from file - it is necessary to control the flow of data to keep a processing chain synchronized and not arbitrarily drop frames. This call allows streams that would normally contain non-discardable frames to allow discarding to occur. By setting this flag on (say) an IQ stream from a file that is being processed downstream, the user will not have to loop to keep this stream emptied to allow the processor to proceed - otherwise the stream would block waiting for the stream buffer to be emptied.

Parameters

<i>cStream</i>	Stream handle
<i>bAllowDiscarding</i>	If set true, then this ensures normal discarding behaviour is followed.

Returns

error/warning information. Expect trfOk.

Example:

```
trfHandle hIQStream(trfInvalidHandle);
trfCreateIQStream(&hIQStream, 0.0f, 0.0f, 0.0f);
trfAllowDiscarding(hIQStream, true); // Stream not serviced - prevents blocking
char *pJSON(nullptr);
trfAddBooleanParameter(&pJSON, TRFLooking, true);
trfSetParameters(hIQStream, &pJSON);
```

trfAttachProcessorToStream()

```
_TRFAPI trfStatus trfAttachProcessorToStream (  
    trfHandle hProcessor,  
    trfHandle hSourceStream )
```

Attach a processor to a source stream.

Parameters

<i>hProcessor</i>	Processor handle
<i>hSourceStream</i>	Stream handle

Returns

error/warning information (on success == trfOk)

Example:

```
// Attach hProcessor to its source IQ stream  
eStatus = trfAttachProcessorToStream(hProcessor, hIQStream);  
if (eStatus != trfOk) {  
    cout << "trfAttachProcessorToStream failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) << endl;  
    return false;  
}
```

trfAttachStreamToDevice()

```
_TRFAPI trfStatus trfAttachStreamToDevice (  
    trfHandle cDevice,  
    trfHandle cStream )
```

Attach the given stream to the specified device. A stream may only be attached to one device at a time. If the device supports the stream, the call will return trfOk and the stream will start to produce data, subject to resource availability on the specified device.

Parameters

<i>cDevice</i>	Device handle
<i>cStream</i>	Stream handle

Returns

error/warning information (on success == trfOk)

Example:

```
eStatus = trfAttachStreamToDevice(hDevice, hStream);  
if (eStatus != trfOk) {  
    cout << "trfAttachStreamToDevice failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) << endl;  
    char *pFullError(trfGetDetailedStatus());  
    cout << "error info: " << std::string(pFullError) << endl;  
    trfDisposeString(&pFullError);  
    trfStatus eStatus(trfOk);  
    while ((eStatus = trfGetNextStatus()) != trfOk) {  
        cout << "Error code " << (int)eStatus << " - " << trfGetTextForStatus(eStatus) << endl;  
    }  
    trfClose(&hStream);  
    trfClose(&hDevice);  
    return false;  
}
```


trfAttachStreamToFile()

```

__TRFAPI trfStatus trfAttachStreamToFile (
    trfHandle cStream,
    const char * pFilePath )

```

Attaches the given stream to an input file. Assumes that the file is valid and of the correct type for the stream - otherwise the call will fail. Note that any special or problematic characters in the file path will be replaced with '_' prior to use. Playback from the stream is controllable through the following parameters that may be set for the stream: TRFLooing - boolean: if set true then the file will be played continuously by looping back to the start once the file has been completely delivered. Function parameters:

Parameters

<i>cStream</i>	Handle to stream to attach to file
<i>*pFilePath</i>	Fully qualified input path for json/data file.

Returns

error/warning information (on success == trfOk)

Example:

```

trfHandle hStream(trfInvalidHandle);
std::cout << "Playing back Spectra from file " << sFile << std::endl;
trfStatus eStatus(trfCreateSpectrumStream(&hStream, 0.0f, 0.0f, 0.0f, nullptr, 0.0f));
if (eStatus != trfOk) {
    std::cout << "trfCreateSpectrumStream failed:" << eStatus << ": " << trfGetTextForStatus(eStatus) <<
    std::endl;
    return false;
}
eStatus = trfAttachStreamToFile(hStream, sFile.c_str());
if (eStatus != trfOk) {
    std::cout << "trfAttachStreamToFile failed:" << eStatus << ": " << trfGetTextForStatus(eStatus) << std::endl;
    return false;
}
char *pJSON(nullptr);
trfGetParameters(hStream, &pJSON);
std::cout << "Stream parameters:" << std::endl << "---" << std::endl << pJSON << std::endl;

```

trfClose()

```

__TRFAPI trfStatus trfClose (
    trfHandle * pHandle )

```

Close any open handle (device, processor, stream) On successful return, *pHandle==kInvalidHandle.

Parameters

<i>*pHandle</i>	Pointer to Device, processor, stream handle
-----------------	---

Returns

error/warning information (on success == trfOk)

Example:

```
eStatus = trfClose(&pHandles[i]);  
if (eStatus != trfOk) {  
    cout << "trfClose [ " << i << " ] failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) << endl;  
    continue;  
}
```

trfCompressSpectrum()

```
_TRFAPI trfStatus trfCompressSpectrum (  
    _uint8 ** ppBlock,  
    _uint32 * pBlockSize,  
    const trfSpectrumFrame * pSource,  
    _float32 fThresholddBm )
```

Compresses the given source frame into an opaque data block. If the threshold parameter is not 0.0f, it is interpreted as a threshold below which all exact spectrum values are ignored. On decompression all such values will be reproduced as the threshold value.

Parameters

<i>ppBlock</i>	Pointer to nullptr to receive the compressed heap-allocated data block.
<i>pBlockSize</i>	Pointer to _uint32 to receive size of compressed data block.
<i>pSource</i>	Pointer to source frame (should be a valid spectrum frame).
<i>fThresholddBm</i>	Threshold parameter.

Returns

error/warning information (on success == trfOk)

trfConvertBasebandFileToWavFile()

```
_TRFAPI trfStatus trfConvertBasebandFileToWavFile (  
    const char * pSourceFilePath,  
    const char * pDestinationFilePath )
```

Stand-alone function to convert a previously created Baseband stream file and produce a WAV file equivalent output file. Any problematic characters in pSourceFilePath will be replaced with the underscore '_' character prior to use.

Parameters

<i>*pSourceFilePath</i>	Fully qualified path to previously generated Baseband data file.
<i>*pDestinationFilePath</i>	Fully qualified output path for equivalent wav file.

Returns

error/warning information (on success == trfOk)

Example:

```
const char *pBasebandFilePath(_TempFile);  
eStatus = trfConvertBasebandFileToWavFile(pBasebandFilePath, pOutputFile);  
if (eStatus != trfOk) {
```

```

        cout << "trfConvertBasebandFileToWavFile failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) <<
endl;
        return false;
    }

```

trfCreateBasebandStream()

```

_TRFAPI trfStatus trfCreateBasebandStream (
    trfHandle * pStream )

```

Create a baseband stream with null parameters.

Parameters

<i>*pStream</i>	Stream handle (returned on success)
-----------------	-------------------------------------

Returns

error/warning information (on success == trfOk)

trfCreateIQStream()

```

_TRFAPI trfStatus trfCreateIQStream (
    trfHandle * pStream,
    _float32 fCentreHz,
    _float32 fIFBWHz,
    _float32 fRefLeveldBm )

```

Create an IQ stream with the given parameters. This entity will be inactive until successfully attached to a device capable of servicing the stream. Note that in the current implementation of libtrf, IF bandwidths narrower than 100kHz are not supported. Additionally IF bandwidths of less than 1MHz below 50MHz FCentre exhibit significant distortion and their use is not recommended. This limitation will be addressed in planned future releases.

Parameters

<i>*pStream</i>	Stream handle (returned on success)
<i>fCentreHz</i>	Centre frequency of frames to be returned by this stream
<i>fIFBWHz</i>	IF Bandwidth of frames to be returned by this stream
<i>fRefLeveldBm</i>	Reference Level in dBm of maximum expected signal

Returns

error/warning information (on success == trfOk)

Example:

```

int iBitDepth(8);
_float32 fFCentreHz(140.0e6f), fIFBWHz(20.0e6f);
_float32 fInitialRefLeveldBm(-50.0f);
trfHandle hStream(trfInvalidHandle);
eStatus = trfCreateIQStream(&hStream, fFCentreHz, fIFBWHz, fInitialRefLeveldBm);
if (eStatus != trfOk) {
    cout << "trfCreateIQStream failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) << endl;
}

```

```

        return false;
    }
    char *pFlag(nullptr);
    trfAddBooleanParameter(&pFlag, TRFDCOffsetFilter, false);
    trfAddIntegerParameter(&pFlag, TRFBitDepth, iBitDepth);
    trfSetParameters(hStream, &pFlag);

```

trfCreateProcessor()

```

_TRFAPI trfStatus trfCreateProcessor (
    trfHandle * pProcessor,
    const char * pType )

```

Create a processor entity of the given type. Returns a handle to the newly created processor.

Parameters

<i>*pProcessor</i>	Processor handle (returned on success)
<i>*pType</i>	Textual type of processor to be created

Returns

error/warning information (on success == trfOk)

Example:

```

// Instantiate an AM demodulator
trfHandle hProcessor(trfInvalidHandle);
eStatus = trfCreateProcessor(&hProcessor, "AMDemodulator");
if (eStatus != trfOk) {
    cout << "trfCreateProcessor failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) << endl;
    return false;
}

```

trfCreateSpectrumStream()

```

_TRFAPI trfStatus trfCreateSpectrumStream (
    trfHandle * pStream,
    _float32 fMinHz,
    _float32 fMaxHz,
    _float32 fRBWHz,
    const char * pWindowType,
    _float32 fRefLeveldBm )

```

Create a spectrum stream with the given parameters. This entity will be inactive until successfully attached to a device capable of servicing the stream.

Parameters

<i>*pStream</i>	Stream handle (returned on success)
<i>fMinHz</i>	Minimum extent of swept frequency range
<i>fMaxHz</i>	Maximum extent of swept frequency range
<i>fRBWHz</i>	Resolution bandwidth of returned sweeps
<i>pWindowType</i>	Window type to use for spectrum reconstruction (nullptr - default)
<i>fRefLeveldBm</i>	Reference Level in dBm of maximum expected signal

Returns

error/warning information (on success == trfOk)

Example:

```
// Create a spectrum (sweep) stream from 1 to 2GHz with RBW of 20kHz
cout << "Creating a spectrum stream." << endl;
_float32 fFStartHz(1300000.0f), fFStopHz(14400000.0f), fRBWHz(1.0e3f), fRefLeveldBm(-40.0f);
const char *pWindow("hann");
trfHandle hStream(trfInvalidHandle);
eStatus = trfCreateSpectrumStream(&hStream, fFStartHz, fFStopHz, fRBWHz, pWindow, fRefLeveldBm);
if (eStatus != trfOk) {
    cout << "trfCreateSpectrumStream failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) << endl;
    return false;
}
```

trfDecimateSpectrum()

```
_TRFAPI trfStatus trfDecimateSpectrum (
    trfSpectrumFrame ** ppResult,
    const trfSpectrumFrame * pSource,
    trfDecimationType eType,
    _uint32 uDecimation )
```

Decimate the given spectrum frame by the given factor. Note that if the spectrum does not contain an integral number of decimation intervals, the last interval will be ignored, truncating the range of the resulting spectrum below the original endpoint. Note that the RBW of the resulting frame will not be affected however the delta between points will increase for $uDecimation > 1$.

Parameters

<i>ppResult</i>	Pointer to resulting frame (which should be nullptr)
<i>pSource</i>	Pointer to source frame (should be a valid spectrum frame)
<i>eType</i>	Type of decimation (maximum, average, minimum)
<i>uDecimation</i>	Decimation factor to apply (0, 1 ignored)

Returns

error/warning information (on success == trfOk)

trfDecompressSpectrum()

```
_TRFAPI trfStatus trfDecompressSpectrum (
    trfSpectrumFrame ** ppResult,
    const _uint8 * pBlock,
    _uint32 uBlockBytes )
```

Decompresses the given data block, produced by trfCompressSpectrum to produce a valid spectrum frame.

Parameters

<i>ppResult</i>	Pointer to null trfSpectrumFrame to receive the uncompressed spectrum.
<i>pBlock</i>	Pointer to pointer to return data block. Should be nullptr when called, and will be a heap-allocated block on successful return.

6.1 libtrf.h File Reference

Parameters

<i>uBlockBytes</i>	Numer of bytes in pBlock.
--------------------	---------------------------

Returns

error/warning information (on success == trfOk)

trfDetachProcessor()

```
_TRFAPI trfStatus trfDetachProcessor (  
    trfHandle hProcessor )
```

Detach a processor from its current source stream.

Parameters

<i>hProcessor</i>	Processor handle
-------------------	------------------

Returns

error/warning information (on success == trfOk)

Example:

```
// Detach processor from its source stream  
eStatus = trfDetachProcessor(hProcessor);  
if (eStatus != trfOk) {  
    cout << "Whups! " << eStatus << endl;  
}
```

trfDetachStream()

```
_TRFAPI trfStatus trfDetachStream (  
    trfHandle cStream )
```

Detach the given stream from any device to which it is connected. This will stop any capture associated with the stream - however the stream will still contain any buffered data that has been previously delivered.

Parameters

<i>cStream</i>	Receives stream handle on success
----------------	-----------------------------------

Returns

error/warning information (on success == trfOk)

Example:

```
// Detach stream halts capture  
trfDetachStream(hStream);  
if (trfIsAttached(hStream) != trfDetached) {  
    cout << "Not showing stream as detached - weird" << endl;  
    return false;  
}
```

```
}
```

trfDisposeString()

```
_TRFAPI void trfDisposeString (
    char ** ppString )
```

Dispose of a previously allocated ppJSON string. Note that only strings that are returned via a parameter of the form 'char **' should be disposed of using this call. This is provided for syntactic convenience and is equivalent to: if (ppJSON != nullptr) { if (*ppJSON != nullptr) { delete [](*ppJSON); *ppJSON = nullptr; } }.

trfEnumerateDevices()

```
_TRFAPI trfStatus trfEnumerateDevices (
    _uint * puDevices,
    bool bAutoDiscover )
```

Enumerate all accessible devices and return a count. Device enumeration may be initiated by a user at any time to enumerate newly connected devices or update the list of devices.

Parameters

<i>puDevices</i>	Address of _uint to receive the number of devices known.
<i>bAutoDiscover</i>	If 'true', then attempt to find any accessible devices. Otherwise only devices explicitly added through trfAddDeviceAddress or USB3 devices are included.

Returns

error/warning information (on success == trfOk)

Example:

```
_uint uDevices(0);
eStatus = trfEnumerateDevices(&uDevices, true);
if (eStatus != trfOk) {
    cout << "trfEnumerateDevices failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) << endl;
    return false;
}
if (0 == uDevices) {
    cout << "No ThinkRF devices enumerated" << endl;
    return false;
}
```

trfExamineDevice()

```
_TRFAPI trfStatus trfExamineDevice (
    _uint uDeviceIndex,
    char ** ppJSON )
```

Return known information about a discovered device (uDeviceIndex in the range [0 .. (uDevices-1)]) into user-allocated trfDeviceInfo structure.

Parameters

<i>uDeviceIndex</i>	Index of device
<i>ppJSON</i>	Pointer to receive UTF8 JSON string

Returns

error/warning information (on success == trfOk)

Example:

```
eStatus = trfExamineDevice(i, &pJSON);
if (eStatus != trfOk) {
    cout << "trfExamineDevice [ " << i << " ] failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) << endl;
    continue;
}
cout << "Device:" << i << " information: " << endl << "---" << endl << pJSON << endl;
trfDisposeString(&pJSON);
```

trfFlushStream()

```
_TRFAPI trfStatus trfFlushStream (
    trfHandle cStream )
```

Remove all frames currently present in a stream. If the stream is attached and delivering data, this will potentially create a break in the data. In the case that the stream is detached, this will remove any outstanding frames. After this call any call to trfGetNextXXX will return trfExhausted.

Parameters

<i>cStream</i>	Receives stream handle on success
----------------	-----------------------------------

Returns

error/warning information (on success == trfOk)

trfFreeBaseband()

```
_TRFAPI trfStatus trfFreeBaseband (
    trfBasebandFrame ** ppFrame )
```

Free the given IQ frame, which will no longer be valid after this call (NULL). The user should ensure that frames are disposed when no longer required to avoid STORAGE_CAPACITY_VIOLATION errors.

Parameters

<i>ppFrame</i>	Address of pointer to IQ frame to dispose of
----------------	--

Returns

error/warning information (on success == trfOk)

trfFreeIQ()

```
_TRFAPI trfStatus trfFreeIQ (
    trfIQFrame ** ppFrame )
```

Free the given IQ frame, which will no longer be valid after this call (NULL). The user should ensure that frames are disposed when no longer required to avoid STORAGE_CAPACITY_VIOLATION errors.

Parameters

<i>ppFrame</i>	Address of pointer to IQ frame to dispose of
----------------	--

Returns

error/warning information (on success == trfOk)

trfFreeSpectrum()

```
_TRFAPI trfStatus trfFreeSpectrum (
    trfSpectrumFrame ** ppFrame )
```

Free the given spectrum frame, which will no longer be valid after this call. The user should ensure that frames are disposed when no longer required to avoid STORAGE_CAPACITY_VIOLATION errors.

Parameters

<i>ppFrame</i>	Address of pointer to Spectrum frame to dispose of
----------------	--

Returns

error/warning information (on success == trfOk)

trfGetDetailedStatus()

```
_TRFAPI char* trfGetDetailedStatus (
    void )
```

Return textual (UTF-8, ISO-Latin-1) error details for the last significant status on the calling thread. Clears error string for this thread on on call.

Returns

Error string - should be disposed of using trfDisposeString.

trfGetNextBaseband()

```
_TRFAPI trfStatus trfGetNextBaseband (
    trfHandle cStream,
    trfBasebandFrame ** ppFrame,
    _uint32 uTimeoutMsec )
```

Obtain the next Baseband frame (if available) from the stream. If no frame is yet available trfStatus will be 'trfWaiting'. If the stream has been halted then the status 'trfExhausted' may be returned to indicate there are no more buffered frames to retrieve. If the retrieval succeeds, trfStatus will be trfOk and *ppFrame will point to the returned Baseband frame. The frame should be disposed of using trfFreeBaseband. The trfExhausted return indicates that capture for this 'attach' is complete and the stream has been detached.

Parameters

<i>cStream</i>	Stream handle
<i>ppFrame</i>	Address of pointer to frame to recieve IQ data.
<i>uTimeoutMsec</i>	Milliseconds to wait for the next packet

Returns

error/warning information (on success == trfOk)

Example:

```
_uint32 uSamplesRemaining((_uint32)roundf(1.0f * kAudioRateHz));
while(uSamplesRemaining > 0) {
    trfBasebandFrame *pFrame(nullptr);
    while (true) {
        eStatus = trfGetNextBaseband(hAudioOut, &pFrame, 1000);
        if (eStatus == trfOk) {
            break;
        }
    }
    if (pFrame->pJSONInfo != nullptr) {
        cout << "Annotation JSON: " << pFrame->pJSONInfo << endl;
    }
    uSamplesRemaining = (pFrame->uSamples > uSamplesRemaining) ? 0: (uSamplesRemaining - pFrame->uSamples);
    cout << "Samples remaining:" << uSamplesRemaining << endl;
    trfFreeBaseband(&pFrame);
}
```

trfGetNextIQ()

```
_TRFAPI trfStatus trfGetNextIQ (
    trfHandle cStream,
    trfIQFrame ** ppFrame,
    _uint32 uTimeoutMsec )
```

Obtain the next IQ frame (if available) from the stream. If no frame is yet available trfStatus will be 'trfWaiting'. If the stream has been halted then the status 'trfExhausted' may be returned to indicate there are no more buffered frames to retrieve. If the retrieval succeeds, trfStatus will be trfOk and *ppFrame will point to the returned IQ frame. The frame should be disposed of using trfFreeIQ. The trfExhausted return indicates that capture for this 'attach' is complete and the stream has been detached.

Parameters

<i>cStream</i>	Stream handle
<i>ppFrame</i>	Address of pointer to frame to recieve IQ data.
<i>uTimeoutMsec</i>	Milliseconds to wait for the next packet

Returns

error/warning information (on success == trfOk)

Example:

```
// Report the first uFramesRemaining IQFrames received and then detach
_uint32 uFramesCaptured(0), uSamplesCaptured(0);
_uint32 uExpectedSequence(0);
bool bFirstDiscontinuity(false);
while (uFramesRemaining > 0) { // could poll until trfExhausted instead to get whole capture as configured
    in stream
    trfIQFrame *pIQFrame(nullptr);
    eStatus = trfGetNextIQ(hStream, &pIQFrame, 5000);    // Obtain next frame
    if (trfExhausted == eStatus) {
        cout << uFramesCaptured << " frames captured - total " << uSamplesCaptured << " samples" << endl;
        break;
    }
    else if (eStatus != trfOk && eStatus > 0) {
        cout << "\n current status: " << eStatus << ", " << trfGetTextForStatus(eStatus) << endl;
        std::cout << std::endl << trfGetVerboseInternalErrorState(100) << std::endl;
        break;
    }
    else if (eStatus < 0) { // error codes are negative values
        cout << "Failed with code: " << eStatus << ", " << trfGetTextForStatus(eStatus) << endl;
        trfClose(&hStream);
        trfClose(&hDevice);
        return false;
    }
    // Report the received frame
    uFramesCaptured++;
    if ((uExpectedSequence != 0) && (pIQFrame->uSequenceNumber != uExpectedSequence)) {
        std::cout << "D" << std::flush;
        if (false == bFirstDiscontinuity) {
            bFirstDiscontinuity = true;
            std::cout << "[" << uExpectedSequence << "]" << std::flush;
        }
    }
    else {
        std::cout << "." << std::flush;
    }
    uExpectedSequence = pIQFrame->uSequenceNumber + 1;
    uFramesRemaining--;
    // Release the received frame
    trfFreeIQ(&pIQFrame);
}
std::cout << ::endl;
```

trfGetNextSpectrum()

```
_TRFAPI trfStatus trfGetNextSpectrum (
    trfHandle cStream,
    trfSpectrumFrame ** ppFrame,
    _uint32 uTimeoutMsec )
```

Obtain the next frame (if available) from the stream. If no frame is yet available trfStatus will be 'WAITING' and *pp↵Frame will be NULL otherwise trfStatus will be trfOk and *ppFrame will point to the returned data frame. The frame should be disposed of using trfFreeSpectrum.

Parameters

<i>cStream</i>	Stream handle
<i>ppFrame</i>	Address of pointer to receive frame pointer
<i>uTimeoutMsec</i>	Maximum time in milliseconds to wait for frame

Returns

error/warning information (on success == trfOk)

Example:

```
bool bFirstFrame(true);
for (_uint32 i = 0; i < 100; i++) {
    trfSpectrumFrame *pSpectrumFrame(nullptr);
    do {
        eStatus = trfGetNextSpectrum(hStream, &pSpectrumFrame, 5000);
        if (eStatus < trfOk) {
            cout << "trfGetNextSpectrum failed: " << trfGetTextForStatus(eStatus) << ":" << (int)eStatus <<
endl;
            break;
        }
        else if (eStatus == trfNotStarted) {
            cout << "s";
            std::cout << trfGetVerboseInternalErrorState(1000) << std::endl;
            break;
        }
        else if (eStatus == trfWaiting) {
            cout << "w";
        }
        else if (eStatus == trfExhausted) {
            cout << "!";
            break;
        }
    } while (pSpectrumFrame == nullptr);
    if (nullptr == pSpectrumFrame) {
        break;
    }
    if (true == bFirstFrame) {
        bFirstFrame = false;
        cout << "First frame FMin:" << pSpectrumFrame->fMinHz << ", FMax:" << pSpectrumFrame->fMaxHz << ",
RBWHZ:" << pSpectrumFrame->fRBWHZ << endl;
    }
    if (pSpectrumFrame->fRBWHZ != fLastRBWHZ) {
        cout << "At frame " << i << ", RBW changes from " << fLastRBWHZ << "Hz to " << pSpectrumFrame->fRBWHZ
<< endl;
    }
    fLastRBWHZ = pSpectrumFrame->fRBWHZ;
    // Compression/decompression check
    _uint8 *pBlob(nullptr);
    _uint32 uBlobSize(0);
    trfCompressSpectrum(&pBlob, &uBlobSize, pSpectrumFrame, -95.0f);
    trfSpectrumFrame *pDecompressed(nullptr);
    trfDecompressSpectrum(&pDecompressed, pBlob, uBlobSize);
    delete[]pBlob; pBlob = nullptr;
    trfFreeSpectrum(&pDecompressed);
    // Free spectrum
    trfFreeSpectrum(&pSpectrumFrame);
    cout << (i%10) << flush;
}
```

trfGetNextStatus()

```
_TRFAPI trfStatus trfGetNextStatus (
    void )
```

Return any additional (descriptive) error codes arising from the last issued libtrf function on the calling thread. To return all error status a user can iterate on this call until it returns trfOk to indicate no further error information is available.

Returns

API status code (trfOk if no further status available)

trfGetParameterInfo()

```
_TRFAPI trfStatus trfGetParameterInfo (
    trfHandle cEntity,
    char ** ppJSON )
```

Obtains all available parameters for the specified device along with the valid ranges, type and any associated information through ppJSON.

Parameters

<i>cEntity</i>	Device/Processor/Stream handle
<i>ppJSON</i>	Pointer to receive UTF8 JSON string

Returns

error/warning information (on success == trfOk)

Example:

```
// Display demodulator parameter set
char *pParameterInfo(nullptr), *pParameters(nullptr);
eStatus = trfGetParameterInfo(hProcessor, &pParameterInfo);
cout << "Parameter info : " << endl << pParameterInfo << endl;
```

trfGetParameters()

```
_TRFAPI trfStatus trfGetParameters (
    trfHandle cEntity,
    char ** ppJSON )
```

Obtain the current values of device/processor parameters and make accessible through ppJSON.

Parameters

<i>cEntity</i>	Device/Processor/Stream handle
<i>ppJSON</i>	Pointer to receive UTF8 JSON string describing parameters

Returns

error/warning information (on success == trfOk)

Example:

```
trfGetParameters(hStream, &pJSON);
std::cout << "Stream parameters:" << std::endl << "----" << std::endl << pJSON << std::endl;
```

trfGetProcessorOutputStream()

```
_TRFAPI trfStatus trfGetProcessorOutputStream (
    trfHandle cProcessor,
    trfHandle * pStream )
```

Return the output stream from the given processor.

6.1 libtrf.h File Reference

Parameters

<i>cProcessor</i>	Processor handle
<i>pStream</i>	Pointer to trfHandle to receive stream handle.

Returns

error/warning information (on success == trfOk)

Example:

```
// Obtain baseband output stream from processor (AM Demodulator)
trfHandle hAudioOut(trfInvalidHandle);
eStatus = trfGetProcessorOutputStream(hProcessor, &hAudioOut);
if (eStatus != trfOk) {
    cout << "trfGetProcessorOutputStream failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) << endl;
    return false;
}
```

trfGetProcessorTypes()

```
_TRFAPI trfStatus trfGetProcessorTypes (
    char ** ppJSON )
```

Returns a JSON-formatted array names 'procesortypes' of the available processor types.

Parameters

<i>**ppJSON</i>	*pJSON contains the list of available processors on exit.
-----------------	---

Returns

error/warning information (on success == trfOk)

Example:

```
// Get the list of available processor types
char *pTypes(nullptr);
eStatus = trfGetProcessorTypes(&pTypes);
cout << "trfGetProcessorTypes returns: " << endl << pTypes << endl;
trfDisposeString(&pTypes);
```

trfGetTextForStatus()

```
const _TRFAPI char* trfGetTextForStatus (
    trfStatus eStatus )
```

Return textual (UTF-8, ISO-Latin-1) status description matching with the passed status code.

Parameters

<i>eStatus</i>	API status code
----------------	-----------------

Returns

Error string - remains owned by API.

trfGetUserResourcesReport()

```
_TRFAPI char* trfGetUserResourcesReport (
    void )
```

Return textual (UTF-8, ISO-Latin-1) report of current user resource utilization within libtrf. Intended as an aid to help a user determine correct operation and identify any resource leakage.

Returns

Error string - should be disposed of using trfDisposeString.

trfGetVersion()

```
_TRFAPI char* trfGetVersion (
    void )
```

Return a C-string describing this version of the library.

Returns

textual description of the library version. Should be disposed of using trfDisposeString using delete[].

Example:

```
char *pVersion(trfGetVersion());
cout << pVersion << endl;
trfDisposeString(&pVersion);
```

trfIsAttached()

```
_TRFAPI trfStatus trfIsAttached (
    trfHandle cStream )
```

Test if the specified stream is attached. Note that a finite capture stream will detach itself after capture is completed. This change will occur at some point in time after the completion of capture, and hence it is valid for this function to return trfOk whilst an associated trfGetNextXXXFrame has returned trfExhausted (finite capture).

Parameters

<i>cStream</i>	Stream handle
----------------	---------------

Returns

error/warning information. If attached returns trfOk. If detached returns the warning trfDetached. trfBadHandle may also be returned.

Example:

6.1 libtrf.h File Reference

```
if (trfIsAttached(hStream) != trfDetached) {  
    cout << "Stream is still attached to source" << endl;  
    return false;  
}
```

trfLoadAntennaFiles()

```
_TRFAPI trfStatus trfLoadAntennaFiles (  
    const char * pFullPath )
```

Inteprets pFullPath as either a directory or specific file to load an antenna table from.

Parameters

<i>pFullPath</i>	File system path to a directory containing .ant files (.csv) or a specific antenna csv file.
------------------	--

Returns

API status code (trfOk on success)

trfLoadEQFiles()

```
_TRFAPI trfStatus trfLoadEQFiles (  
    const char * pFullPath )
```

Inteprets pFullPath as either a directory or specific file to load an EQ table from.

Parameters

<i>pFullPath</i>	File system path to a directory containing .eq files (.csv) or a specific antenna csv file.
------------------	---

Returns

API status code (trfOk on success)

trfOpenDevice()

```
_TRFAPI trfStatus trfOpenDevice (  
    _uint uDeviceIndex,  
    trfHandle * pDevice )
```

Given a device index, attempt to open it. On success sets the device handle *pHandle.

Parameters

<i>uDeviceIndex</i>	Index of device from most recent call to trfEnumerateDevices
<i>pDevice</i>	Pointer to device handle to be used in future operations

Returns

error/warning information (on success == trfOk)

Example:

```
eStatus = trfOpenDevice(i, &pHandles[i]);
if (eStatus != trfOk) {
    cout << "trfOpenDevice [" << i << "] failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) << endl;
    continue;
}
```

trfReadBooleanParameter()

```
_TRFAPI trfStatus trfReadBooleanParameter (
    char ** ppJSON,
    const char * pParameterName,
    bool * pbValue )
```

Read named boolean parameter from ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string containing parameter(s).
<i>pParameterName</i>	Name of numeric parameter to read.
<i>pbValue</i>	Pointer to bool to receive value.

Returns

error/warning information (on success == trfOk)

trfReadFCentre()

```
_TRFAPI trfStatus trfReadFCentre (
    char ** ppJSON,
    _float32 * pfFCentreHz )
```

Read Centre frequency parameter from ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string containing parameter(s).
<i>pfFCentreHz</i>	Pointer to _float32 to receive value.

Returns

error/warning information (on success == trfOk)

trfReadFMinMax()

```
_TRFAPI trfStatus trfReadFMinMax (
    char ** ppJSON,
    _float32 * pfFMinHz,
    _float32 * pfFMaxHz )
```

Read min/max frequency range from ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string containing parameter(s).
<i>pfFMinHz</i>	Pointer to _float32 to receive value.
<i>pfFMaxHz</i>	Pointer to _float32 to receive value.

Returns

error/warning information (on success == trfOk)

trfReadIFBWHz()

```
_TRFAPI trfStatus trfReadIFBWHz (
    char ** ppJSON,
    _float32 * pfFIFBWHz )
```

Read IFBW parameter from ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string containing parameter(s).
<i>pfFIFBWHz</i>	Pointer to _float32 to receive value.

Returns

error/warning information (on success == trfOk)

trfReadIntegerParameter()

```
_TRFAPI trfStatus trfReadIntegerParameter (
    char ** ppJSON,
    const char * pParameterName,
    _int32 * piValue )
```

Read named numeric parameter from ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string containing parameter(s).
<i>pParameterName</i>	Name of integer parameter to read.
<i>piValue</i>	Pointer to <code>_int32</code> to receive value.

Returns

error/warning information (on success == `trfOk`)

trfReadJSONElement()

```
_TRFAPI trfStatus trfReadJSONElement (
    const char * ppJSON,
    const char * pElementName,
    char ** ppElement )
```

Read a named element from the passed JSON string into *ppElement. The element itself must be either a JSON object, or a JSON array. If ppJSON does not contain the element - *ppElement will be nullptr on exit and the function will return `trfUnknownParameter`.

Parameters

<i>pJSON</i>	Pointer to JSON string.
<i>pElementName</i>	Name of element to extract.
<i>ppElement</i>	Pointer to char pointer to receive value - allocated as <code>[]char</code> .

Returns

error/warning information (on success == `trfOk`)

trfReadJSONElementFromArray()

```
_TRFAPI trfStatus trfReadJSONElementFromArray (
    char ** ppJSON,
    const char * pArrayName,
    char ** ppFirstElement )
```

Extract the first element in the passed ppJSON to be interpreted as a JSON array, removing that item from the array and returning it in *ppElement. If on return *ppElement is nullptr,.

Parameters

<i>ppJSON</i>	Pointer to JSON string containing a named array.
<i>pArrayName</i>	Name of array
<i>pFirstElement</i>	First element of array

Returns

error/warning information (on success == trfOk)

trfReadLongIntegerParameter()

```
_TRFAPI trfStatus trfReadLongIntegerParameter (
    char ** ppJSON,
    const char * pParameterName,
    _int64 * piValue )
```

Read named numeric parameter from ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string containing parameter(s).
<i>pParameterName</i>	Name of integer parameter to read.
<i>piValue</i>	Pointer to _int64 to receive value.

Returns

error/warning information (on success == trfOk)

trfReadNumericParameter()

```
_TRFAPI trfStatus trfReadNumericParameter (
    char ** ppJSON,
    const char * pParameterName,
    _float32 * pfValue )
```

Read named numeric parameter from ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string containing parameter(s).
<i>pParameterName</i>	Name of numeric parameter to read.
<i>pfValue</i>	Pointer to _float32 to receive value.

Returns

error/warning information (on success == trfOk)

trfReadParameterAsJSON()

```
_TRFAPI trfStatus trfReadParameterAsJSON (
    char ** pJSON,
```

```
const char * pParameter,
char ** ppJSONExtract )
```

Read a sub-element from a JSON element as another JSON string. Useful for extracting substructure from a JSON string.

Parameters

<i>pJSON</i>	Source JSON string
<i>pParameter</i>	Name of parameter to extract
<i>ppJSONExtract</i>	Pointer to receive JSON string

Returns

error/warning information (on success == trfOk)

trfReadParameterInfoElement()

```
_TRFAPI trfStatus trfReadParameterInfoElement (
    char ** ppJSON,
    const char * pParameter,
    const char * pElement,
    _float32 * pfValue )
```

Read a sub-element from a JSON element as a `_float32` value. Useful for obtaining 'min', 'max' and 'default' values from parameter information.

Parameters

<i>ppJSON</i>	Source JSON string
<i>pParameter</i>	Name of parameter to be extracted from
<i>pElement</i>	Name of element (ie. "min", "max", "default")
<i>pfValue</i>	Pointer to <code>_float32</code> to receive value

Returns

error/warning information (on success == trfOk)

trfReadRBWHZ()

```
_TRFAPI trfStatus trfReadRBWHZ (
    char ** ppJSON,
    _float32 * pfRBWHZ )
```

Read resolution bandwidth parameter from ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string containing parameter(s).
---------------	---

6.1 libtrf.h File Reference

Parameters

<i>pfRBWHz</i>	Pointer to _float32 to receive value.
----------------	---------------------------------------

Returns

error/warning information (on success == trfOk)

trfReadRefLeveldBm()

```
_TRFAPI trfStatus trfReadRefLeveldBm (  
    char ** ppJSON,  
    _float32 * pfRefLeveldBm )
```

Read reference level parameter from ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string containing parameter(s).
<i>pfRefLeveldBm</i>	Pointer to _float32 to receive value.

Returns

error/warning information (on success == trfOk)

trfReadSampleRateHz()

```
_TRFAPI trfStatus trfReadSampleRateHz (  
    char ** ppJSON,  
    _float32 * pfSampleRateHz )
```

Read sample rate parameter from ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string containing parameter(s).
<i>pfSampleRateHz</i>	Pointer to _float32 to receive value.

Returns

error/warning information (on success == trfOk)

trfReadTextParameter()

```
_TRFAPI trfStatus trfReadTextParameter (
```

```
char ** ppJSON,
const char * pParameterName,
char ** ppValue )
```

Read named textual parameter from ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string containing parameter(s).
<i>pParameterName</i>	Name of textual parameter to read.
<i>ppValue</i>	Pointer to char pointer to receive value - allocated as []char.

Returns

error/warning information (on success == trfOk)

trfReadVectorElementAsJSON()

```
_TRFAPI trfStatus trfReadVectorElementAsJSON (
    char ** pJSON,
    _uint32 uIndex,
    char ** ppJSONExtract )
```

Assuming pJSON is a valid JSON array, extracts the specified index from the array as a JSON string.

Parameters

<i>pJSON</i>	Source JSON string
<i>uIndex</i>	Array index to extract
<i>ppJSONExtract</i>	Pointer to receive JSON string

Returns

error/warning information (on success == trfOk)

trfReadWindowType()

```
_TRFAPI trfStatus trfReadWindowType (
    char ** ppJSON,
    char ** ppWindow )
```

Read window type parameter from ppJSON.

Parameters

<i>ppJSON</i>	Pointer to JSON string containing parameter(s).
<i>ppWindow</i>	Pointer to char pointer to receive value - allocated as []char.

Returns

error/warning information (on success == trfOk)

trfRemoveDeviceAddress()

```
_TRFAPI trfStatus trfRemoveDeviceAddress (
    const char * pRemoteAddress )
```

Removes a potential device address from later enumeration.

Parameters

<i>pRemoteAddress</i>	IPv4/IPv6 address for remote device.
-----------------------	--------------------------------------

Returns

error/warning information (on success == trfOk)

Example:

```
eStatus = trfAddDeviceAddress("10.126.110.114");
if ((eStatus != trfOk) && (eStatus != trfPingFailed)) {
    cout << "trfAddDeviceAddress failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) << endl;
    return false;
}
```

trfResetDeviceConnection()

```
_TRFAPI trfStatus trfResetDeviceConnection (
    trfHandle cDevice )
```

If a communications error is suspected, issuing this call will cause a connection shutdown and re-open to the device without losing any operational state. On success the device will continue with programmed operation. If the return value is not 'trfOk' then it is likely communications are unrecoverable and the device handle should be closed (see trfClose).

Parameters

<i>cDevice</i>	Device handle to be used in future operations
----------------	---

Returns

error/warning information (on success == trfOk)

Example:

```
eStatus = trfResetDeviceConnection(hHandle);
if (eStatus != trfOk) {
    cout << "trfResetDeviceConnection [" << i << "] failed:" << eStatus << " - " <<
    trfGetTextForStatus(eStatus) << endl;
}
```


trfSetParameters()

```
_TRFAPI trfStatus trfSetParameters (
    trfHandle cEntity,
    char ** ppJSON )
```

Set the current value of a device/stream/processor parameter.

Parameters

<i>cEntity</i>	Device/Processor/Stream handle
<i>ppJSON</i>	Pointer to JSON string pointer, describing parameter(s) to be set. On return *ppJSON will be nullptr and storage will be disposed of.

Returns

error/warning information (on success == trfOk)

Example:

```
// Set AM demodulator parameters (obtain using trfGetParameterInfo)
trfAddNumericParameter(&pParameters, TRFOutputSampleRate, kAudioRateHz); // Program output sample rate -
44.1kHz
trfAddTextParameter(&pParameters, TRFAMSubtype, "DSB"); // Program AM demodulator type
eStatus = trfSetParameters(hProcessor, &pParameters);
if (eStatus != trfOk) {
    cout << "trfSetParameters failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) << endl;
    return false;
}
trfDisposeString(&pParameters);
```

trfSetStreamOutputFile()

```
_TRFAPI trfStatus trfSetStreamOutputFile (
    trfHandle cStream,
    const char * pFilePath )
```

Attaches the specified output file - or if the file exists and is not empty, a numbered sibling file to the specified stream. The self-contained output file may be replayed using the trfCreateFileSourceStream function. Note that the specified file path may be modified to ensure that it contains no special or problematic characters before use; these will be replaced with the underscore '_' character. The output binary file created has the extension '.data' (ThinkRF Spectrum, IQ, Baseband), The binary file is accompanied by a metadata file (.json). If pFilePath is NULL, any file writing for the stream shall be immediately halted.

Parameters

<i>cStream</i>	Stream handle
<i>pFilePath</i>	Fully qualified output path for data file.

Returns

error/warning information (on success == trfOk)

Example:

```
trfSetStreamOutputFile(hStream, pFilename);
if (eStatus != trfOk) {
    std::cout << "trfSetStreamOutputFile failed:" << eStatus << " - " << trfGetTextForStatus(eStatus) <<
    std::endl;
```

6.2 libtrf_doxy.txt File Reference

```
    trfClose(&hStream);  
    return false;  
}
```

6.2 libtrf_doxy.txt File Reference

Index

- [_complex32, 29](#)
 - [fl, 29](#)
 - [fQ, 29](#)
 - [_float32](#)
 - [libtrf.h, 72](#)
 - [_float64](#)
 - [libtrf.h, 73](#)
 - [_trfBasebandFrame, 30](#)
 - [fSampleRateHz, 30](#)
 - [fUsableBWHHz, 30](#)
 - [pJSONInfo, 30](#)
 - [ppData, 31](#)
 - [uChannels, 31](#)
 - [uSamples, 31](#)
 - [uSequenceNumber, 31](#)
 - [uTimestampNanosec, 31](#)
 - [_trfIQFrame, 31](#)
 - [bDiscontinuity, 32](#)
 - [bSubOptimalDRFlag, 32](#)
 - [eFlags, 32](#)
 - [fCentreHz, 33](#)
 - [fIFBWHHz, 33](#)
 - [fSampleRateHz, 33](#)
 - [pData, 33](#)
 - [pJSONInfo, 33](#)
 - [uSamples, 33](#)
 - [uSequenceNumber, 33](#)
 - [uTimestampNanosec, 34](#)
 - [_trfSpectrumFrame, 34](#)
 - [eFlags, 35](#)
 - [fMaxHz, 35](#)
 - [fMinHz, 35](#)
 - [fRBWHHz, 35](#)
 - [iDurationNanosec, 35](#)
 - [pAdditionalSpectra, 35](#)
 - [pData, 35](#)
 - [pJSONInfo, 36](#)
 - [pUnits, 36](#)
 - [uAdditionalSpectra, 36](#)
 - [uPoints, 36](#)
 - [uSequenceNumber, 36](#)
 - [uTimestampNanosec, 36](#)
 - [_trfStatus](#)
 - [libtrf.h, 74](#)
 - [_uint](#)
 - [libtrf.h, 73](#)
 - [_uint16](#)
 - [libtrf.h, 73](#)
 - [_uint32](#)
 - [libtrf.h, 73](#)
 - [_uint64](#)
 - [libtrf.h, 73](#)
 - [_uint8](#)
 - [libtrf.h, 73](#)
 - [bDiscontinuity](#)
 - [_trfIQFrame, 32](#)
 - [bSubOptimalDRFlag](#)
 - [_trfIQFrame, 32](#)
 - [eFlags](#)
 - [_trfIQFrame, 32](#)
 - [_trfSpectrumFrame, 35](#)
 - [fCentreHz](#)
 - [_trfIQFrame, 33](#)
 - [fl](#)
 - [_complex32, 29](#)
 - [fIFBWHHz](#)
 - [_trfIQFrame, 33](#)
 - [fMaxHz](#)
 - [_trfSpectrumFrame, 35](#)
 - [fMinHz](#)
 - [_trfSpectrumFrame, 35](#)
 - [fQ](#)
 - [_complex32, 29](#)
 - [fRBWHHz](#)
 - [_trfSpectrumFrame, 35](#)
 - [fSampleRateHz](#)
 - [_trfBasebandFrame, 30](#)
 - [_trfIQFrame, 33](#)
 - [fUsableBWHHz](#)
 - [_trfBasebandFrame, 30](#)
 - [iDurationNanosec](#)
 - [_trfSpectrumFrame, 35](#)
 - [libtrf.h, 37](#)
 - [_float32, 72](#)
 - [_float64, 73](#)
 - [_trfStatus, 74](#)

_uint, 73
 _uint16, 73
 _uint32, 73
 _uint64, 73
 _uint8, 73
 trfAddArrayElement, 78
 trfAddBooleanParameter, 78
 trfAddDeviceAddress, 79
 trfAddFCentre, 79
 trfAddFMinMax, 79
 trfAddIFBWHZ, 80
 trfAddIntegerParameter, 80
 trfAddIQCaptureSec, 81
 trfAddJSONElement, 81
 trfAddJSONElementToArray, 82
 trfAddNumericParameter, 82
 TRFAddRange, 51
 trfAddRBWHZ, 82
 trfAddRefLeveldBm, 83
 TRFAddress, 51
 trfAddTextParameter, 83
 trfAddWindowType, 84
 trfAllowDiscarding, 84
 TRFAltitude, 51
 TRFAMSubtype, 51
 TRFAntenna, 51
 trfAntennaInvalid, 77
 trfAPINotInitialized, 75
 trfAttachProcessorToStream, 84
 trfAttachStreamToDevice, 85
 trfAttachStreamToFile, 86
 TRFAttenuation, 51
 TRFAverageCount, 52
 trfBadDeviceHandle, 75
 trfBadFilename, 76
 trfBadFrequencyParameters, 76
 trfBadGNSSDynamicMode, 76
 trfBadJSONElement, 77
 trfBadJSONName, 77
 trfBadJSONStructure, 77
 trfBadPLLSource, 76
 trfBadPPSSource, 77
 trfBadProcessorHandle, 76
 trfBadRBWParameter, 76
 trfBadReceiverHandle, 75
 trfBadReferenceLevelParameter, 76
 trfBadStreamHandle, 75
 trfBadStreamShutdown, 75
 trfBadUnitHandle, 75
 TRFBandwidthHz, 52
 TRFBaseband, 52
 trfBasebandFrame, 73
 trfBasebandStreamCreateFailure, 77
 trfBasebandStreamStartFailure, 76
 TRFBitDepth, 52
 TRFBitDepthOutOfRange, 77
 trfBufferDiscardProportionOutOfRange, 76
 TRFBufferFrames, 52
 trfBufferFramesOutOfRange, 76
 TRFBufferSec, 52
 trfBufferSecOutOfRange, 76
 trfCannotAttachIQStream, 75
 trfCannotAttachSpectrumStream, 75
 trfCannotAttachStream, 75
 trfCannotCloseStream, 75
 trfCannotConnectOutputStream, 76
 trfCannotCreateProcessor, 76
 trfCannotDetachIQStream, 75
 trfCannotDetachSpectrumStream, 75
 trfCannotDetachStream, 75
 trfCannotObtainOutputStream, 76
 trfCannotOpenDevice, 75
 trfCannotOpenOutputFile, 76
 trfCannotRestartStream, 75
 TRFChannels, 52
 TRFClearAverageTrace, 53
 TRFClearMaxHoldTrace, 53
 TRFClearMinHoldTrace, 53
 trfClose, 86
 trfCompressSpectrum, 87
 trfConnectionResetFailed, 76
 trfConvertBasebandFileToWavFile, 87
 trfCreateBasebandStream, 88
 trfCreateIQStream, 88
 trfCreateProcessor, 89
 trfCreateSpectrumStream, 89
 TRFCurrentDate, 53
 TRFCurrentTime, 53
 TRFDataType, 53
 TRFdBFSD, 53
 TRFDOffsetFilter, 53
 trfDecimateSpectrum, 90
 trfDecimationType, 77
 trfDecompressSpectrum, 90
 TRFDefault, 54
 trfDetached, 74
 trfDetachProcessor, 91
 trfDetachStream, 91
 TRFDetector, 54
 TRFDevice, 54
 trfDeviceAddressAlreadyKnown, 75
 trfDeviceCommunicationsLost, 76
 TRFDeviceConnection, 54
 TRFDeviceFirmware, 54
 trfDeviceHandlesInvalid, 75
 trfDeviceIndexOutOfRange, 75
 TRFDeviceLockStatus, 54
 TRFDeviceSerialNum, 55

[TRFDeviceTemperature](#), 55
[TRFDeviceType](#), 55
[trfDeviceUnreachable](#), 76
[TRFDeviceVersion](#), 55
[TRFDigitalGain](#), 55
[trfDigitalGainOutOfRange](#), 77
[TRFDiscardProportion](#), 55
[trfDiscontinuousWithPreviousFrame](#), 74
[trfDisposeString](#), 92
[trfDuplicateAddress](#), 75
[TRFDurationFrames](#), 55
[TRFDurationSec](#), 56
[trfDwellTimeInvalid](#), 77
[trfDwellTimeOutOfRange](#), 77
[TRFDwellUsec](#), 56
[TRFEncoding](#), 56
[TRFEndSequence](#), 56
[TRFEndTimeStamp](#), 56
[trfEnumerateDevices](#), 92
[trfEQInvalid](#), 77
[TRFEqualization](#), 56
[trfEqualizerInvalid](#), 77
[TRFEventDuration](#), 56
[TRFEventID](#), 56
[trfExamineDevice](#), 92
[TRFExceeddB](#), 57
[TRFExceedNotLoss](#), 57
[TRFExceedRecords](#), 57
[trfExhausted](#), 74
[trfFatalReceiverError](#), 77
[TRFFCentreHz](#), 57
[trfFCentreInvalid](#), 76
[trfFCentreOutOfRange](#), 76
[TRFFFilename](#), 57
[trfFileOpenFailed](#), 75
[trfFileTypelsNotBaseband](#), 76
[trfFileTypelsNotIQ](#), 75
[trfFileTypelsNotSpectrum](#), 75
[TRFFFilterRatio](#), 57
[TRFFflattenFlag](#), 57
[TRFFlowControl](#), 58
[trfFlushStream](#), 93
[TRFFMaxHz](#), 58
[trfFMaxInvalid](#), 76
[TRFFMinHz](#), 58
[trfFMinInvalid](#), 76
[TRFFollowIQ](#), 58
[TRFFrameDuration](#), 58
[TRFFrameInFile](#), 58
[TRFFrameOutputEnabled](#), 58
[TRFFramesExpected](#), 59
[TRFFramesProduced](#), 59
[trfFrameTypeMismatch](#), 75
[trfFreeBaseband](#), 93
[trfFreeIQ](#), 94
[trfFreeSpectrum](#), 94
[TRFFFrequency](#), 59
[trfFrequencyRangesInvalid](#), 76
[TRFFFullAdaptFlag](#), 59
[TRFGaindB](#), 59
[trfGainInvalid](#), 77
[TRFGatedOutput](#), 59
[trfGenericFailStatus](#), 77
[trfGetDetailedStatus](#), 94
[trfGetNextBaseband](#), 94
[trfGetNextIQ](#), 95
[trfGetNextSpectrum](#), 96
[trfGetNextStatus](#), 97
[trfGetParameterInfo](#), 97
[trfGetParameters](#), 98
[trfGetProcessorOutputStream](#), 98
[trfGetProcessorTypes](#), 99
[trfGetTextForStatus](#), 99
[trfGetUserResourcesReport](#), 100
[trfGetVersion](#), 100
[TRFGNSSADelay](#), 59
[TRFGNSSConstellation](#), 60
[TRFGNSSDynamicMode](#), 60
[TRFGNSSHeadingDegT](#), 60
[TRFGNSSInfo](#), 60
[TRFGNSSMagVarDegT](#), 60
[TRFGNSSRxTime](#), 60
[TRFGNSSSpeedOverGround](#), 60
[TRFGNSSTimestamp](#), 60
[TRFGNSSTrackDegT](#), 61
[TRFGNSSValid](#), 61
[TRFGPIO](#), 61
[trfHandle](#), 74
[TRFHasFlatteningInfo](#), 61
[TRFHasGNSS](#), 61
[TRFIFBWHz](#), 61
[trfIFBWInvalid](#), 76
[trfIFBWOutOfRange](#), 76
[trfInvalidArrayIndex](#), 77
[trfInvalidFilename](#), 76
[trfInvalidHandle](#), 61
[trfInvalidHandleSpecified](#), 75
[trfInvalidIP](#), 77
[trfInvalidSequence](#), 62
[trfInvalidStreamHandle](#), 75
[trfInvalidStreamSource](#), 77
[TRFIOnly](#), 62
[TRFIQ](#), 62
[TRFIQActive](#), 62
[TRFIQCapture](#), 62
[trfIQFrame](#), 74
[trfIQStreamStartFailure](#), 75
[trfIsAttached](#), 100

TRFIsContiguous, 62
trfJSONEmpty, 77
TRFLANMTU, 62
trfLastErrorSentinel, 77
TRFLastSequence, 63
TRFLatitude, 63
TRFLevel, 63
trfLinAverage, 78
trfLoadAntennaFiles, 101
trfLoadEQFiles, 101
trfLogAverage, 78
TRFLongitude, 63
TRFLooking, 63
TRFMatchIQ, 63
TRFMax, 63
TRFMaxContiguousSec, 63
trfMaximum, 78
TRFMeasuredAvgdBm, 64
trfMemoryAllocationError, 75
TRFMin, 64
trfMinimum, 78
TRFMinimumPeakdBm, 64
TRFModifyRange, 64
TRFName, 64
trfNoArrayParameter, 77
trfNoAssociatedRxOrFile, 75
trfNoAssociatedSource, 75
trfNoDataSpecified, 75
trfNoDeviceInformation, 75
trfNoOp, 77
trfNoParametersSpecified, 75
trfNotABasebandStream, 76
trfNotAnIQStream, 75
trfNotASpectrumStream, 75
trfNotStarted, 74
trfNoUniqueArrayParameter, 77
TRFNTP, 64
TRFOBOccupiedPercentage, 64
TRFObserveSeconds, 65
trfOk, 74
trfOnCallPointerShouldBeNull, 77
trfOpenDevice, 101
TRFOutputFrameSize, 65
TRFOutputSampleRate, 65
TRFOutputSize, 65
TRFOverlap, 65
TRFPacketDurationMsec, 65
trfParameterSetError, 75
TRFPeakCount, 65
TRFPeakdBm, 65
TRFPeakHz, 66
TRFPeaks, 66
trfPingFailed, 74
TRFPLLSource, 66
trfPLLSourceWithNoGNSS, 76
TRFPointsPerFrame, 66
TRFPPSSource, 66
trfProcessorAttachFailed, 76
trfProcessorDetachFailed, 76
trfProcessorUnattached, 76
TRFRangeActive, 66
TRFRangeName, 66
TRFRBWHz, 67
trfReadBooleanParameter, 102
trfReadFCentre, 102
trfReadFMinMax, 103
trfReadIFBWHz, 103
trfReadIntegerParameter, 103
trfReadJSONElement, 104
trfReadJSONElementFromArray, 104
trfReadLongIntegerParameter, 105
trfReadNumericParameter, 105
trfReadParameterAsJSON, 105
trfReadParameterInfoElement, 106
trfReadRBWHz, 106
trfReadRefLeveldBm, 107
trfReadSampleRateHz, 107
trfReadTextParameter, 107
trfReadVectorElementAsJSON, 108
trfReadWindowType, 108
trfRebootedStatus, 77
TRFRecentFrames, 67
TRFReferenceLeveldBm, 67
trfReferenceLevelOutOfRange, 76
TRFRefLevel, 67
trfRemoveDeviceAddress, 109
TRFRemoveRange, 67
trfResetDeviceConnection, 109
trfResolutionBWInvalid, 76
trfResolutionBWOutOfRange, 76
TRFRFFStart, 67
TRFRFFStop, 67
TRFRFGPIO, 67
TRFRFRRange, 68
TRFRFRSelect, 68
TRFRxSampleRate, 68
TRFSampleRateHz, 68
trfSampleRateInvalid, 76
TRFSamples, 68
TRFSampling, 68
trfSCPIQueryFailed, 75
TRFSCPIQueryTimeout, 68
trfSCPISendFailed, 75
TRFSequence, 69
trfSetParameters, 109
trfSetStreamOutputFile, 110
TRFSpan, 69
TRFSpectrum, 69

- trfSpectrumCompressFailed, [77](#)
- trfSpectrumFrame, [74](#)
- trfSpectrumStreamCreateFailure, [76](#)
- TRFSpurReject, [69](#)
- trfStartFrequencyOutOfRange, [76](#)
- TRFStartSequence, [69](#)
- TRFStartTimeStamp, [69](#)
- trfStatus, [74](#)
- TRFStepAdaptFlag, [69](#)
- trfStopFrequencyOutOfRange, [76](#)
- trfStreamFailedToStart, [77](#)
- trfStreamHandleAlreadyOpen, [77](#)
- trfStreamParametersError, [77](#)
- TRFSubOptimalFlag, [69](#)
- TRFSustaindB, [70](#)
- TRFSweepActive, [70](#)
- trfSweepStartFailure, [75](#)
- TRFTable, [70](#)
- TRFThreshold, [70](#)
- TRFTimeConstant, [70](#)
- TRFTimeResolution, [70](#)
- TRFTimestamp, [70](#)
- TRFTimeSync, [71](#)
- TRFTrainSeconds, [71](#)
- TRFType, [71](#)
- trfUnallocatedUserData, [75](#)
- trfUnimplemented, [75](#)
- trfUninitializedUserData, [76](#)
- TRFUnits, [71](#)
- trfUnitsInvalid, [77](#)
- trfUnknownAddress, [77](#)
- trfUnknownDetectorType, [77](#)
- trfUnknownParameter, [75](#)
- trfUnknownStreamType, [77](#)
- trfUnknownWindowType, [76](#)
- trfUnsupportedParameter, [75](#)
- TRFUsableBWHZ, [71](#)
- TRFUserCalibrationOffset, [71](#)
- TRFValue, [71](#)
- TRFValues, [72](#)
- TRFVBWHZ, [72](#)
- trfVBWInvalid, [77](#)
- trfVideoBWOutOfRange, [77](#)
- trfWaiting, [74](#)
- TRFWeightedFilter, [72](#)
- TRFWindow, [72](#)
- trfWindowTypeInvalid, [76](#)
- TRFWriteStartSequence, [72](#)
- TRFWriteStopSequence, [72](#)
- libtrf_doxy.txt, [111](#)
- pAdditionalSpectra
 - _trfSpectrumFrame, [35](#)
- pData
 - _trfIQFrame, [33](#)
 - _trfSpectrumFrame, [35](#)
- pJSONInfo
 - _trfBasebandFrame, [30](#)
 - _trfIQFrame, [33](#)
 - _trfSpectrumFrame, [36](#)
- ppData
 - _trfBasebandFrame, [31](#)
- pUnits
 - _trfSpectrumFrame, [36](#)
- trfAddArrayElement
 - libtrf.h, [78](#)
- trfAddBooleanParameter
 - libtrf.h, [78](#)
- trfAddDeviceAddress
 - libtrf.h, [79](#)
- trfAddFCentre
 - libtrf.h, [79](#)
- trfAddFMinMax
 - libtrf.h, [79](#)
- trfAddIFBWHZ
 - libtrf.h, [80](#)
- trfAddIntegerParameter
 - libtrf.h, [80](#)
- trfAddIQCaptureSec
 - libtrf.h, [81](#)
- trfAddJSONElement
 - libtrf.h, [81](#)
- trfAddJSONElementToArray
 - libtrf.h, [82](#)
- trfAddNumericParameter
 - libtrf.h, [82](#)
- TRFAddRange
 - libtrf.h, [51](#)
- trfAddRBWHZ
 - libtrf.h, [82](#)
- trfAddRefLeveldBm
 - libtrf.h, [83](#)
- TRFAddress
 - libtrf.h, [51](#)
- trfAddTextParameter
 - libtrf.h, [83](#)
- trfAddWindowType
 - libtrf.h, [84](#)
- trfAllowDiscarding
 - libtrf.h, [84](#)
- TRFAltitude
 - libtrf.h, [51](#)
- TRFAMSubtype
 - libtrf.h, [51](#)
- TRFAntenna
 - libtrf.h, [51](#)
- trfAntennaInvalid

- libtrf.h, [77](#)
- trfAPINotInitialized
 - libtrf.h, [75](#)
- trfAttachProcessorToStream
 - libtrf.h, [84](#)
- trfAttachStreamToDevice
 - libtrf.h, [85](#)
- trfAttachStreamToFile
 - libtrf.h, [86](#)
- TRFAttenuation
 - libtrf.h, [51](#)
- TRFAverageCount
 - libtrf.h, [52](#)
- trfBadDeviceHandle
 - libtrf.h, [75](#)
- trfBadFilename
 - libtrf.h, [76](#)
- trfBadFrequencyParameters
 - libtrf.h, [76](#)
- trfBadGNSSDynamicMode
 - libtrf.h, [76](#)
- trfBadJSONElement
 - libtrf.h, [77](#)
- trfBadJSONName
 - libtrf.h, [77](#)
- trfBadJSONStructure
 - libtrf.h, [77](#)
- trfBadPLLSource
 - libtrf.h, [76](#)
- trfBadPPSSource
 - libtrf.h, [77](#)
- trfBadProcessorHandle
 - libtrf.h, [76](#)
- trfBadRBWParameter
 - libtrf.h, [76](#)
- trfBadReceiverHandle
 - libtrf.h, [75](#)
- trfBadReferenceLevelParameter
 - libtrf.h, [76](#)
- trfBadStreamHandle
 - libtrf.h, [75](#)
- trfBadStreamShutdown
 - libtrf.h, [75](#)
- trfBadUnitHandle
 - libtrf.h, [75](#)
- TRFBandwidthHz
 - libtrf.h, [52](#)
- TRFBaseband
 - libtrf.h, [52](#)
- trfBasebandFrame
 - libtrf.h, [73](#)
- trfBasebandStreamCreateFailure
 - libtrf.h, [77](#)
- trfBasebandStreamStartFailure
 - libtrf.h, [76](#)
- TRFBitDepth
 - libtrf.h, [52](#)
- TRFBitDepthOutOfRange
 - libtrf.h, [77](#)
- trfBufferDiscardProportionOutOfRange
 - libtrf.h, [76](#)
- TRFBufferFrames
 - libtrf.h, [52](#)
- trfBufferFramesOutOfRange
 - libtrf.h, [76](#)
- TRFBufferSec
 - libtrf.h, [52](#)
- trfBufferSecOutOfRange
 - libtrf.h, [76](#)
- trfCannotAttachIQStream
 - libtrf.h, [75](#)
- trfCannotAttachSpectrumStream
 - libtrf.h, [75](#)
- trfCannotAttachStream
 - libtrf.h, [75](#)
- trfCannotCloseStream
 - libtrf.h, [75](#)
- trfCannotConnectOutputStream
 - libtrf.h, [76](#)
- trfCannotCreateProcessor
 - libtrf.h, [76](#)
- trfCannotDetachIQStream
 - libtrf.h, [75](#)
- trfCannotDetachSpectrumStream
 - libtrf.h, [75](#)
- trfCannotDetachStream
 - libtrf.h, [75](#)
- trfCannotObtainOutputStream
 - libtrf.h, [76](#)
- trfCannotOpenDevice
 - libtrf.h, [75](#)
- trfCannotOpenOutputFile
 - libtrf.h, [76](#)
- trfCannotRestartStream
 - libtrf.h, [75](#)
- TRFChannels
 - libtrf.h, [52](#)
- TRFClearAverageTrace
 - libtrf.h, [53](#)
- TRFClearMaxHoldTrace
 - libtrf.h, [53](#)
- TRFClearMinHoldTrace
 - libtrf.h, [53](#)
- trfClose
 - libtrf.h, [86](#)
- trfCompressSpectrum
 - libtrf.h, [87](#)
- trfConnectionResetFailed

- libtrf.h, [76](#)
- trfConvertBasebandFileToWavFile
 - libtrf.h, [87](#)
- trfCreateBasebandStream
 - libtrf.h, [88](#)
- trfCreateIQStream
 - libtrf.h, [88](#)
- trfCreateProcessor
 - libtrf.h, [89](#)
- trfCreateSpectrumStream
 - libtrf.h, [89](#)
- TRFCurrentDate
 - libtrf.h, [53](#)
- TRFCurrentTime
 - libtrf.h, [53](#)
- TRFDataType
 - libtrf.h, [53](#)
- TRFdBFSD
 - libtrf.h, [53](#)
- TRFDOffsetFilter
 - libtrf.h, [53](#)
- trfDecimateSpectrum
 - libtrf.h, [90](#)
- trfDecimationType
 - libtrf.h, [77](#)
- trfDecompressSpectrum
 - libtrf.h, [90](#)
- TRFDefault
 - libtrf.h, [54](#)
- trfDetached
 - libtrf.h, [74](#)
- trfDetachProcessor
 - libtrf.h, [91](#)
- trfDetachStream
 - libtrf.h, [91](#)
- TRFDetector
 - libtrf.h, [54](#)
- TRFDevice
 - libtrf.h, [54](#)
- trfDeviceAddressAlreadyKnown
 - libtrf.h, [75](#)
- trfDeviceCommunicationsLost
 - libtrf.h, [76](#)
- TRFDeviceConnection
 - libtrf.h, [54](#)
- TRFDeviceFirmware
 - libtrf.h, [54](#)
- trfDeviceHandlesInvalid
 - libtrf.h, [75](#)
- trfDeviceIndexOutOfRange
 - libtrf.h, [75](#)
- TRFDeviceLockStatus
 - libtrf.h, [54](#)
- TRFDeviceSerialNum
 - libtrf.h, [55](#)
- TRFDeviceTemperature
 - libtrf.h, [55](#)
- TRFDeviceType
 - libtrf.h, [55](#)
- trfDeviceUnreachable
 - libtrf.h, [76](#)
- TRFDeviceVersion
 - libtrf.h, [55](#)
- TRFDigitalGain
 - libtrf.h, [55](#)
- trfDigitalGainOutOfRange
 - libtrf.h, [77](#)
- TRFDiscardProportion
 - libtrf.h, [55](#)
- trfDiscontinuousWithPreviousFrame
 - libtrf.h, [74](#)
- trfDisposeString
 - libtrf.h, [92](#)
- trfDuplicateAddress
 - libtrf.h, [75](#)
- TRFDurationFrames
 - libtrf.h, [55](#)
- TRFDurationSec
 - libtrf.h, [56](#)
- trfDwellTimeInvalid
 - libtrf.h, [77](#)
- trfDwellTimeOutOfRange
 - libtrf.h, [77](#)
- TRFDwellUsec
 - libtrf.h, [56](#)
- TRFEncoding
 - libtrf.h, [56](#)
- TRFEndSequence
 - libtrf.h, [56](#)
- TRFEndTimeStamp
 - libtrf.h, [56](#)
- trfEnumerateDevices
 - libtrf.h, [92](#)
- trfEQInvalid
 - libtrf.h, [77](#)
- TRFEqualization
 - libtrf.h, [56](#)
- trfEqualizerInvalid
 - libtrf.h, [77](#)
- TRFEventDuration
 - libtrf.h, [56](#)
- TRFEventID
 - libtrf.h, [56](#)
- trfExamineDevice
 - libtrf.h, [92](#)
- TRFExceeddB
 - libtrf.h, [57](#)
- TRFExceedNotLoss

- libtrf.h, [57](#)
- TRFExceedRecords
 - libtrf.h, [57](#)
- trfExhausted
 - libtrf.h, [74](#)
- trfFatalReceiverError
 - libtrf.h, [77](#)
- TRFFCentreHz
 - libtrf.h, [57](#)
- trfCentreInvalid
 - libtrf.h, [76](#)
- trfCentreOutOfRange
 - libtrf.h, [76](#)
- TRFFilename
 - libtrf.h, [57](#)
- trfFileOpenFailed
 - libtrf.h, [75](#)
- trfFileTypesNotBaseband
 - libtrf.h, [76](#)
- trfFileTypesNotIQ
 - libtrf.h, [75](#)
- trfFileTypesNotSpectrum
 - libtrf.h, [75](#)
- TRFFilterRatio
 - libtrf.h, [57](#)
- TRFFlattenFlag
 - libtrf.h, [57](#)
- TRFFlowControl
 - libtrf.h, [58](#)
- trfFlushStream
 - libtrf.h, [93](#)
- TRFFMaxHz
 - libtrf.h, [58](#)
- trfMaxInvalid
 - libtrf.h, [76](#)
- TRFFMinHz
 - libtrf.h, [58](#)
- trfMinInvalid
 - libtrf.h, [76](#)
- TRFFollowIQ
 - libtrf.h, [58](#)
- TRFFrameDuration
 - libtrf.h, [58](#)
- TRFFrameInFile
 - libtrf.h, [58](#)
- TRFFrameOutputEnabled
 - libtrf.h, [58](#)
- TRFFramesExpected
 - libtrf.h, [59](#)
- TRFFramesProduced
 - libtrf.h, [59](#)
- trfFrameTypeMismatch
 - libtrf.h, [75](#)
- trfFreeBaseband
 - libtrf.h, [93](#)
- trfFreeIQ
 - libtrf.h, [94](#)
- trfFreeSpectrum
 - libtrf.h, [94](#)
- TRFFrequency
 - libtrf.h, [59](#)
- trfFrequencyRangelsInvalid
 - libtrf.h, [76](#)
- TRFFullAdaptFlag
 - libtrf.h, [59](#)
- TRFGaindB
 - libtrf.h, [59](#)
- trfGainInvalid
 - libtrf.h, [77](#)
- TRFGatedOutput
 - libtrf.h, [59](#)
- trfGenericFailStatus
 - libtrf.h, [77](#)
- trfGetDetailedStatus
 - libtrf.h, [94](#)
- trfGetNextBaseband
 - libtrf.h, [94](#)
- trfGetNextIQ
 - libtrf.h, [95](#)
- trfGetNextSpectrum
 - libtrf.h, [96](#)
- trfGetNextStatus
 - libtrf.h, [97](#)
- trfGetParameterInfo
 - libtrf.h, [97](#)
- trfGetParameters
 - libtrf.h, [98](#)
- trfGetProcessorOutputStream
 - libtrf.h, [98](#)
- trfGetProcessorTypes
 - libtrf.h, [99](#)
- trfGetTextForStatus
 - libtrf.h, [99](#)
- trfGetUserResourcesReport
 - libtrf.h, [100](#)
- trfGetVersion
 - libtrf.h, [100](#)
- TRFGNSSADelay
 - libtrf.h, [59](#)
- TRFGNSSConstellation
 - libtrf.h, [60](#)
- TRFGNSSDynamicMode
 - libtrf.h, [60](#)
- TRFGNSSHeadingDegT
 - libtrf.h, [60](#)
- TRFGNSSInfo
 - libtrf.h, [60](#)
- TRFGNSSMagVarDegT

[libtrf.h, 60](#)
 TRFGNSSRxTime
 [libtrf.h, 60](#)
 TRFGNSSSpeedOverGround
 [libtrf.h, 60](#)
 TRFGNSSTimestamp
 [libtrf.h, 60](#)
 TRFGNSSTrackDegT
 [libtrf.h, 61](#)
 TRFGNSSValid
 [libtrf.h, 61](#)
 TRFGPIO
 [libtrf.h, 61](#)
 trfHandle
 [libtrf.h, 74](#)
 TRFHasFlatteningInfo
 [libtrf.h, 61](#)
 TRFHasGNSS
 [libtrf.h, 61](#)
 TRFIFBWHz
 [libtrf.h, 61](#)
 trfIFBWInvalid
 [libtrf.h, 76](#)
 trfIFBWOutOfRange
 [libtrf.h, 76](#)
 trfInvalidArrayIndex
 [libtrf.h, 77](#)
 trfInvalidFilename
 [libtrf.h, 76](#)
 trfInvalidHandle
 [libtrf.h, 61](#)
 trfInvalidHandleSpecified
 [libtrf.h, 75](#)
 trfInvalidIP
 [libtrf.h, 77](#)
 trfInvalidSequence
 [libtrf.h, 62](#)
 trfInvalidStreamHandle
 [libtrf.h, 75](#)
 trfInvalidStreamSource
 [libtrf.h, 77](#)
 TRFIOnly
 [libtrf.h, 62](#)
 TRFIQ
 [libtrf.h, 62](#)
 TRFIQActive
 [libtrf.h, 62](#)
 TRFIQCapture
 [libtrf.h, 62](#)
 trfIQFrame
 [libtrf.h, 74](#)
 trfIQStreamStartFailure
 [libtrf.h, 75](#)
 trfIsAttached
 [libtrf.h, 100](#)
 TRFIsContiguous
 [libtrf.h, 62](#)
 trfJSONEmpty
 [libtrf.h, 77](#)
 TRFLANMTU
 [libtrf.h, 62](#)
 trfLastErrorSentinel
 [libtrf.h, 77](#)
 TRFLastSequence
 [libtrf.h, 63](#)
 TRFLatitude
 [libtrf.h, 63](#)
 TRFLevel
 [libtrf.h, 63](#)
 trfLinAverage
 [libtrf.h, 78](#)
 trfLoadAntennaFiles
 [libtrf.h, 101](#)
 trfLoadEQFiles
 [libtrf.h, 101](#)
 trfLogAverage
 [libtrf.h, 78](#)
 TRFLongitude
 [libtrf.h, 63](#)
 TRFLooking
 [libtrf.h, 63](#)
 TRFMatchIQ
 [libtrf.h, 63](#)
 TRFMax
 [libtrf.h, 63](#)
 TRFMaxContiguousSec
 [libtrf.h, 63](#)
 trfMaximum
 [libtrf.h, 78](#)
 TRFMeasuredAvgdBm
 [libtrf.h, 64](#)
 trfMemoryAllocationError
 [libtrf.h, 75](#)
 TRFMin
 [libtrf.h, 64](#)
 trfMinimum
 [libtrf.h, 78](#)
 TRFMinimumPeakdBm
 [libtrf.h, 64](#)
 TRFModifyRange
 [libtrf.h, 64](#)
 TRFName
 [libtrf.h, 64](#)
 trfNoArrayParameter
 [libtrf.h, 77](#)
 trfNoAssociatedRxOrFile
 [libtrf.h, 75](#)
 trfNoAssociatedSource

libtrf.h, [75](#)
 trfNoDataSpecified
 libtrf.h, [75](#)
 trfNoDeviceInformation
 libtrf.h, [75](#)
 trfNoOp
 libtrf.h, [77](#)
 trfNoParametersSpecified
 libtrf.h, [75](#)
 trfNotABasebandStream
 libtrf.h, [76](#)
 trfNotAnIQStream
 libtrf.h, [75](#)
 trfNotASpectrumStream
 libtrf.h, [75](#)
 trfNotStarted
 libtrf.h, [74](#)
 trfNoUniqueArrayParameter
 libtrf.h, [77](#)
 TRFNTF
 libtrf.h, [64](#)
 TRFOBOccupiedPercentage
 libtrf.h, [64](#)
 TRFObserveSeconds
 libtrf.h, [65](#)
 trfOk
 libtrf.h, [74](#)
 trfOnCallPointerShouldBeNull
 libtrf.h, [77](#)
 trfOpenDevice
 libtrf.h, [101](#)
 TRFOutputFrameSize
 libtrf.h, [65](#)
 TRFOutputSampleRate
 libtrf.h, [65](#)
 TRFOutputSize
 libtrf.h, [65](#)
 TRFOverlap
 libtrf.h, [65](#)
 TRFPacketDurationMsec
 libtrf.h, [65](#)
 trfParameterSetError
 libtrf.h, [75](#)
 TRFPeakCount
 libtrf.h, [65](#)
 TRFPeakdBm
 libtrf.h, [65](#)
 TRFPeakHz
 libtrf.h, [66](#)
 TRFPeaks
 libtrf.h, [66](#)
 trfPingFailed
 libtrf.h, [74](#)
 TRFPLLSource
 libtrf.h, [66](#)
 trfPLLSourceWithNoGNSS
 libtrf.h, [76](#)
 TRFPointsPerFrame
 libtrf.h, [66](#)
 TRFPSSource
 libtrf.h, [66](#)
 trfProcessorAttachFailed
 libtrf.h, [76](#)
 trfProcessorDetachFailed
 libtrf.h, [76](#)
 trfProcessorUnattached
 libtrf.h, [76](#)
 TRFRangeActive
 libtrf.h, [66](#)
 TRFRangeName
 libtrf.h, [66](#)
 TRFRBWHZ
 libtrf.h, [67](#)
 trfReadBooleanParameter
 libtrf.h, [102](#)
 trfReadFCentre
 libtrf.h, [102](#)
 trfReadFMinMax
 libtrf.h, [103](#)
 trfReadIFBWHZ
 libtrf.h, [103](#)
 trfReadIntegerParameter
 libtrf.h, [103](#)
 trfReadJSONElement
 libtrf.h, [104](#)
 trfReadJSONElementFromArray
 libtrf.h, [104](#)
 trfReadLongIntegerParameter
 libtrf.h, [105](#)
 trfReadNumericParameter
 libtrf.h, [105](#)
 trfReadParameterAsJSON
 libtrf.h, [105](#)
 trfReadParameterInfoElement
 libtrf.h, [106](#)
 trfReadRBWHZ
 libtrf.h, [106](#)
 trfReadRefLeveldBm
 libtrf.h, [107](#)
 trfReadSampleRateHz
 libtrf.h, [107](#)
 trfReadTextParameter
 libtrf.h, [107](#)
 trfReadVectorElementAsJSON
 libtrf.h, [108](#)
 trfReadWindowType
 libtrf.h, [108](#)
 trfRebootedStatus

- libtrf.h, [77](#)
- TRFRecentFrames
 - libtrf.h, [67](#)
- TRFReferenceLevelBm
 - libtrf.h, [67](#)
- trfReferenceLevelOutOfRange
 - libtrf.h, [76](#)
- TRFRefLevel
 - libtrf.h, [67](#)
- trfRemoveDeviceAddress
 - libtrf.h, [109](#)
- TRFRemoveRange
 - libtrf.h, [67](#)
- trfResetDeviceConnection
 - libtrf.h, [109](#)
- trfResolutionBWInvalid
 - libtrf.h, [76](#)
- trfResolutionBWOutOfRange
 - libtrf.h, [76](#)
- TRFRFFStart
 - libtrf.h, [67](#)
- TRFRFFStop
 - libtrf.h, [67](#)
- TRFRFGPIO
 - libtrf.h, [67](#)
- TRFRFRRange
 - libtrf.h, [68](#)
- TRFRFSelect
 - libtrf.h, [68](#)
- TRFRxSampleRate
 - libtrf.h, [68](#)
- TRFSampleRateHz
 - libtrf.h, [68](#)
- trfSampleRateInvalid
 - libtrf.h, [76](#)
- TRFSamples
 - libtrf.h, [68](#)
- TRFSampling
 - libtrf.h, [68](#)
- trfSCPIQueryFailed
 - libtrf.h, [75](#)
- TRFSCPIQueryTimeout
 - libtrf.h, [68](#)
- trfSCPISendFailed
 - libtrf.h, [75](#)
- TRFSequence
 - libtrf.h, [69](#)
- trfSetParameters
 - libtrf.h, [109](#)
- trfSetStreamOutputFile
 - libtrf.h, [110](#)
- TRFSpan
 - libtrf.h, [69](#)
- TRFSpectrum
 - libtrf.h, [69](#)
- trfSpectrumCompressFailed
 - libtrf.h, [77](#)
- trfSpectrumFrame
 - libtrf.h, [74](#)
- trfSpectrumStreamCreateFailure
 - libtrf.h, [76](#)
- TRFSpurReject
 - libtrf.h, [69](#)
- trfStartFrequencyOutOfRange
 - libtrf.h, [76](#)
- TRFStartSequence
 - libtrf.h, [69](#)
- TRFStartTimeStamp
 - libtrf.h, [69](#)
- trfStatus
 - libtrf.h, [74](#)
- TRFStepAdaptFlag
 - libtrf.h, [69](#)
- trfStopFrequencyOutOfRange
 - libtrf.h, [76](#)
- trfStreamFailedToStart
 - libtrf.h, [77](#)
- trfStreamHandleAlreadyOpen
 - libtrf.h, [77](#)
- trfStreamParametersError
 - libtrf.h, [77](#)
- TRFSubOptimalFlag
 - libtrf.h, [69](#)
- TRFSustaindB
 - libtrf.h, [70](#)
- TRFSweepActive
 - libtrf.h, [70](#)
- trfSweepStartFailure
 - libtrf.h, [75](#)
- TRFTable
 - libtrf.h, [70](#)
- TRFThreshold
 - libtrf.h, [70](#)
- TRFTimeConstant
 - libtrf.h, [70](#)
- TRFTimeResolution
 - libtrf.h, [70](#)
- TRFTimestamp
 - libtrf.h, [70](#)
- TRFTimeSync
 - libtrf.h, [71](#)
- TRFTrainSeconds
 - libtrf.h, [71](#)
- TRFType
 - libtrf.h, [71](#)
- trfUnallocatedUserData
 - libtrf.h, [75](#)
- trfUnimplemented

- libtrf.h, [75](#)
- trfUninitializedUserData
 - libtrf.h, [76](#)
- TRFUnits
 - libtrf.h, [71](#)
- trfUnitsInvalid
 - libtrf.h, [77](#)
- trfUnknownAddress
 - libtrf.h, [77](#)
- trfUnknownDetectorType
 - libtrf.h, [77](#)
- trfUnknownParameter
 - libtrf.h, [75](#)
- trfUnknownStreamType
 - libtrf.h, [77](#)
- trfUnknownWindowType
 - libtrf.h, [76](#)
- trfUnsupportedParameter
 - libtrf.h, [75](#)
- TRFUsableBWHZ
 - libtrf.h, [71](#)
- TRFUserCalibrationOffset
 - libtrf.h, [71](#)
- TRFValue
 - libtrf.h, [71](#)
- TRFValues
 - libtrf.h, [72](#)
- TRFVBWHZ
 - libtrf.h, [72](#)
- trfVBWInvalid
 - libtrf.h, [77](#)
- trfVideoBWOutOfRange
 - libtrf.h, [77](#)
- trfWaiting
 - libtrf.h, [74](#)
- TRFWeightedFilter
 - libtrf.h, [72](#)
- TRFWindow
 - libtrf.h, [72](#)
- trfWindowTypeInvalid
 - libtrf.h, [76](#)
- TRFWriteStartSequence
 - libtrf.h, [72](#)
- TRFWriteStopSequence
 - libtrf.h, [72](#)
- uAdditionalSpectra
 - _trfSpectrumFrame, [36](#)
- uChannels
 - _trfBasebandFrame, [31](#)
- uPoints
 - _trfSpectrumFrame, [36](#)
- uSamples
 - _trfBasebandFrame, [31](#)
- _trfIQFrame, [33](#)
- uSequenceNumber
 - _trfBasebandFrame, [31](#)
 - _trfIQFrame, [33](#)
 - _trfSpectrumFrame, [36](#)
- uTimestampNanosec
 - _trfBasebandFrame, [31](#)
 - _trfIQFrame, [34](#)
 - _trfSpectrumFrame, [36](#)