JBLopen

Embedded Software Insight

TREEspan File System API Reference Manual

RM0002

February 12, 2020

Contents

Chapter

1 Overview

Welcome to the TREEspan™ File System API reference manual. This reference manual covers the core, file and directory API functions, data types and preprocessor definitions along with a description and usage information for each API element. The API is written in ISO/IEC 9899:1999 (C99) compliant C and designed to be portable between platforms and toolchains.

For convenience during development, all the information related to each individual API elements is also reproduced within the relevant header source files in human readable format.

1.1 About TREEspan File System

TSFS is an embedded transactional file system, supporting a wide range of storage technologies, including native flash support with both dynamic and static wear-leveling. Through its support for snapshots and write transactions, TSFS provides the application with flexible, robust and fail-safe data storage. Being RTOS and platform agnostic, with a minimum RAM requirement of less than 4KiB, TSFS can be deployed on almost any platform.

1.2 Elements of the API reference

Each documented API element, be it a function, data type or preprocessor definition is presented using a similar layout which is described below. This section briefly describes the various elements of the API reference.

1.2.1 Functions

The most numerous and important API elements documented are functions. Below is an example of API reference for a hypothetical function named `example_func()`:

Function

`example_func()`

<example/example.h>

Example function description.

Prototype

```
int  example_func  ( uint32_t  arg1,
                     uint32_t  arg2 );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✗	✓	✓	✓

Parameters

`arg1`	First argument's description.
`arg2`	Second argument's description.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Example

```
uint32_t a = 0u;
uint32_t b = 1u;
int result;

result = example_func(a, b);
if(result != RTNC_SUCCESS) {
    // Handle error.
}
```

Function Name

At the top of each API is the name of the function or object as it appears in the source code. TSFS functions are always prefixed with `tsfs_` followed by the function's specific name.

Header

Following the name is the header file where the declaration of the documented API can be found. It is recommended to use the displayed path relative to the root of the TSFS source directory when including the TSFS headers.

For example, to include the file module header file `tsfs_file.h` the following include directive is recommended.

```
#include <tsfs_file.h>
```

The root of the TSFS source directory should be added to the include path of the compiler.

Description

A description of the API element including basic usage information.

Prototype

For functions, the full signature of the API along with parameter names, types, and function return type.

Attributes

For functions only, this section lists the relevant function attributes. See Section 1.3 for a detailed description of each attribute.

Parameters

Function parameters list along with a short description of each parameter.

Returned Errors or Return Value

For functions returning an error code, this section is named ”Returned Errors” and lists the relevant errors that can be returned. See Section 1.4.2 for more information on TSFS error handling.

For other functions that do not return an error code, this section lists the possible output values of the function. In this case, the section is named ”Returned Value”.

Example

Some API functions may include a small code example to illustrate usage. Note that these examples are for documentation purposes and may not include error handling and checking to keep the examples concise.

1.2.2 Data Types

Data types include structure definitions, enumerated types as well as scalar type definitions. They all follow a similar documentation layout, below is an example of API reference for a hypothetical structure definition named `example_struct_t`:

Data Type

`example_struct_t`

<example/example.h>

Example structure description.

Members

`member1`	`uint32_t`	First member’s description.
`member2`	`uint32_t`	Second member’s description.

Data Type Name

At the top of each API is the name of the data type as it appears in the source code. In the case of structures and enumerated types, this is always the `typedef`’d data type. TSFS types always prefixed with `tsfs_` followed by the type’s specific name. Types are also always suffixed with `_t` to differentiate them from other definitions.

Header

Following the name is the header file where the declaration of the documented API can be found. It is recommended to use the displayed path relative to the root of the source directory of TSFS when including TSFS headers.

Description

A description of the data type including basic usage information.

Members/Enumeration Values

If documented, the API reference will list the structure members or the list of enumeration constants along with a short description of each member. The list of members for opaque types with no publicly accessible members are omitted from the list of members in the API documentation.

1.2.3 Macros

Relevant and preprocessor macros that are part of the public API are documented in the API reference. This includes function-like macros as well as object-like macros. The latter is often referred to as preprocessor definitions or simply defines. Below is an example of function-like macro named `TSFS_EXAMPLE_MACRO()`:

Macro

`TSFS_EXAMPLE_MACRO()`

<example/tsfs_example.h>

Example macro description.

Prototype

```
TSFS_EXAMPLE_MACRO ( arg1,
                     arg2 );
```

Parameters

`arg1`	First argument's description.
`arg2`	First argument's description.

Expansion Macro expansion's description.

Macro Name

At the top of each API is the name of the macro as it appears in the source code. TSFS preprocessor definitions are always in capital letters and prefixed with `TSFS_` followed by the macro's specific name.

Header

Following the name is the header file where the declaration of the documented API can be found. It is recommended to use the displayed path relative to the root of the source directory of TSFS when including TSFS headers.

Description

A description of the macro including basic usage information.

Parameters

Macro parameters list along with a short description of each parameter.

Expansion

For function-like macros an expansion section describes the macro's expansion including the type if applicable.

1.3 Function Attributes

The API reference documentation for API functions includes a set of attributes that clarifies in which context it is safe to call a specific API function. The attributes are as follows:

- Blocking
- ISR-safe
- Critical-safe
- Thread-safe

1.3.1 Blocking

The function is potentially blocking, which means it can wait or pend on a kernel object such as a semaphore or mutex, in order to wait for a resource to be available or for an operation to complete. Some functions may be optionally blocking depending on the function's arguments. Those functions are always marked as blocking in the API reference regardless.

In a bare-metal environment, any function marked as blocking can potentially suspend the background task while waiting for a specific interrupt. Many of those functions take a timeout parameter that can be set to 0 to make them non-blocking (polling) if suspension of the background task is undesired.

As a general rule, blocking functions should not be called from an interrupt service routine, also known as interrupt handler or while the CPU interrupts are disabled. In addition, some RTOSes allow suspending or locking the scheduler, when this is the case, blocking functions should not be called while the scheduler is suspended or locked.

1.3.2 ISR-Safe

An ISR-safe function can be called from within an interrupt service routine. This also includes callback functions that are called from an interrupt context. Note that while an ISR-safe function is usually critical-safe this is not always the case. Also an ISR-safe function may not necessarily be thread-safe.

1.3.3 Critical Safe

Critical safe functions can be called when the CPU interrupts are disabled, this is also called a critical context or sometimes a critical section. Critical sections are usually entered by calling a spinlock acquire or critical section enter function. Calling a non-critical-safe function from within a critical section can corrupt the state of the CPU's interrupt disable flags and cause runtime faults or data corruption.

1.3.4 Thread-Safe

A thread safe function guarantees correct operations between multiple threads or tasks when running under a multitasking kernel. In the context of the TSFS API, thread-safe also implies thread safety on an SMP system, which means it is safe to use the API function from different threads in parallel. Due to the design of TSFS, thread-safe functions are also re-entrant, provided that the other function attributes, such as ISR safety, are respected.

1.3.5 Function Attributes in Header Files

Function attributes are documented slightly differently in the source header files in order to be more concise and easier to maintain. The attributes are documented under an ”Attributes” section and are named as follows:

- non-blocking
- non-thread-safe
- ISR-safe
- critical-section-safe

Absence of an attribute implies that the opposite attribute applies to the function. For example, in the absence of any explicit function attribute in the header documentation, a function is assumed to be blocking, thread-safe and not safe to call from ISRs and critical sections.

1.4 API Conventions

TSFS API adheres to a few conventions with respect to the naming, error handling and timeouts that are useful for the application developers.

1.4.1 Naming

The TSFS API function names are all written in lower case, except preprocessor macros which are in upper case. Words within an object name are separated by underscores and the whole name is prefixed with `tsfs_` followed by the function specific part of the name.

For example, the core module function to create a snapshot is as follows:

```
tsfs_sshot_create()
```

And the maximum path length is named as follows:

```
TSFS_MAX_PATH_LEN
```

1.4.2 Error Handling

Most API functions return a status in the form of a plain `int` as the function’s return value. As a general exception, some functions that cannot fail are allowed to return nothing (`void`) or another value.

In general, TSFS attempts to minimize the number of different error codes to simplify the application’s error handling and improve performance. The list of possible error codes is included within every

function's documentation. The meaning of each error code is also documented in a function's description. A list of all defined error codes is given in here.

As with other preprocessor macros and enumeration constants, the application should never rely on the exact numerical value of any specific error code. However, two guarantees are made with respect to the error code numerical values. The first is that `RTNC_SUCCESS` will always expand to 0. The second is that all other error codes are negative. Positive values are not used for any valid error code. Any undefined or unexpected error code returned by a function should be treated as a fatal error.

Two error codes have the exact same meaning for all the functions, namely `RTNC_SUCCESS` and `RTNC_FATAL`.

`RTNC_SUCCESS` is returned when a function completed successfully without issue.

`RTNC_FATAL` is returned if and only if an unexpected situation that should not happen at runtime is detected. This includes invalid function arguments, internal data corruption and assertion failures within the code. In addition, any unexpected error code returned from a function should be treated as a fatal error. It is up to the application to decide on the proper action to perform upon receiving a fatal error. As a general rule, the application should not perform any other calls to that module instance. Safety critical applications should consider an `RTNC_FATAL` error code as a severe assertion failure and act accordingly.

Some modules, especially IO modules such as UART and I2C, provides a reset API call that can be used to reset the internal state of a module as well as the underlying peripheral. This can be used to attempt to recover from a fatal error in case the error condition is temporary.

1.4.3 Numerical Values of Macros and Enumeration Constants

To ease maintainability and ensure compatibility with future versions, the application should never rely on enumeration constants and macros numerical value.

Chapter

2 API Reference

Function

tsfs_commit()

<tsfs.h>

Commits all the updates performed on the given file system instance since the last commit, including file updates, snapshot creations and deletions.

In the event of an unexpected interruption (e.g. power loss) the file system is returned to the state it was in after the last successful call to `tsfs_commit()`.

Prototype

```
int  tsfs_commit  ( const char *  p_fs_name );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the file system instance to be committed.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_create()

<tsfs.h>

Creates a new file system instance.

Prototype

```
int tsfs_create ( const char *       p_fs_name,
                  const tsfs_cfg_t * p_cfg );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the created file system instance.
`p_cfg`	TSFS configuration structure.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_destroy()

<tsfs.h>

Frees all the memory tied to the given file system instance.

Prototype

```
int tsfs_destroy ( const char * p_fs_name );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the created file system instance.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_dir_close()

<tsfs_dir.h>

Closes the given directory. The handle becomes invalid after the directory is closed. Using a directory handle after closing it yields undefined behavior.

Prototype

```
int tsfs_dir_close ( tsfs_dir_hndl_t hndl );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters `hndl` Handle to the directory to be closed.

Returned Errors
`RTNC_SUCCESS`
`RTNC_FATAL`

Function

tsfs_dir_create()

<tsfs_dir.h>

Creates a directory at the given path.

The given path must lead to a location within the working state.

If a file or directory already exists at this location, `RTNC_ALREADY_EXIST` is returned and the original file or directory is left untouched.

If the parent directory does not exist, `RTNC_NOT_FOUND` is returned. If the path is outside the working state, `RTNC_INVALID_OP` is returned.

Prototype

```
int  tsfs_dir_create  ( const char *  p_path );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters `p_path` Path of the directory to be created.

Returned Errors
`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_OP`
`RTNC_ALREADY_EXIST`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_dir_delete()

<tsfs_dir.h>

Deletes the directory located at the given path.

The path must lead to an existing directory of the working state. If the directory does not exist, `RTNC_NOT_FOUND` is returned. If the directory is not in the working state or the given path leads to a file, `RTNC_INVALID_OP` is returned. Also, if the directory is not empty, `RTNC_INVALID_OP` is returned.

An opened directory may safely be deleted. In this case, any further read access to this directory will return `RTNC_NOT_FOUND`.

Prototype

```
int  tsfs_dir_delete  ( const char *  p_path );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path of the directory to be deleted.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_OP`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_dir_exists()

<tsfs_dir.h>

Verifies whether the directory located at the given path exists. The function returns `true` (through the `p_exist` parameter) if the directory exists and `false` otherwise (including when the given path leads to a file).

Prototype

```
int  tsfs_dir_exists  ( const char *  p_path,
                        bool *        p_exist );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path to the directory which existence is to be tested.
`p_exist`	Whether the directory located at the given path exists.

Returned Errors

`RTNC_SUCCESS`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_dir_open()

<tsfs_dir.h>

Opens the directory located at the given path. If the directory does not exist, `RTNC_NOT_FOUND` is returned. If the given path leads to a file, `RTNC_INVALID_OP` is returned.

Prototype

```
int  tsfs_dir_open  ( const char *        p_path,
                      tsfs_dir_hndl_t *   p_hndl );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path to the directory to be opened.
`p_hndl`	Handle to the opened directory instance.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_OP`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_dir_read()`

<tsfs_dir.h>

Reads the content of the given directory, one directory entry at a time. The name of the current directory entry is copied in the given name buffer. Calling `tsfs_dir_read()` after the last directory entry has been reached will return an empty string.

Prototype

```
int  tsfs_dir_read  ( tsfs_dir_hndl_t  hndl,
                      char *           p_name,
                      size_t           name_sz );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`hndl`	Handle to the directory to read from.
`p_name`	Buffer to receive the current entry name.
`name_sz`	Size of the given name buffer in bytes.

Returned Errors

`RTNC_SUCCESS`
`RTNC_IO_ERR`
`RTNC_OVERFLOW`
`RTNC_FATAL`

Function

`tsfs_drop()`

<tsfs.h>

Reverts the file system's state to that of the latest commit. All modifications performed since the latest commit are discarded, including file updates, snapshot creations and deletions.

Prototype

```
int  tsfs_drop  ( const char *  p_fs_name );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the file system instance.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_file_append()

<tsfs_file.h>

Writes the supplied buffer at the end of the given file.

The number of bytes written is returned through `p_append_sz`. The returned value may be smaller than `append_sz` if, and only if, the file system is full. In this case `RTNC_FULL` is returned and the value pointed to by `p_append_sz` indicates the number of bytes written before the file system becomes full.

If an error occurs, other than `RTNC_FULL`, the value pointed to by `p_append_sz` is unspecified. Otherwise, if the function completes successfully, the requested number of bytes is guaranteed to have been written. In this case, the value pointed to by `p_append_sz` is always equal to `append_sz`.

Prototype

```
int  tsfs_file_append  ( tsfs_file_hndl_t    hndl,
                         const void *        p_buf,
                         tsfs_file_size_t    append_sz,
                         tsfs_file_size_t *  p_append_sz );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`hndl`	Handle to the file to append data.
`p_buf`	Buffer to be written.
`append_sz`	Size of the buffer to be written in bytes.
`p_append_sz`	Size of the written data in bytes.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FULL`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_file_close()

<tsfs_file.h>

Closes the given file. The handle becomes invalid after the file is closed. Using a file handle after closing it yields undefined behaviour.

Prototype

```
int tsfs_file_close ( tsfs_file_hndl_t hndl );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`hndl` Handle to the file to be closed.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

tsfs_file_create()

<tsfs_file.h>

Creates a file at the given path.

The given path must lead to a location within the working state.

If a file or directory already exists at the same location, `RTNC_ALREADY_EXIST` is returned and the original file or directory is left untouched.

If the parent directory does not exist, `RTNC_NOT_FOUND` is returned. If the path is outside the working state, `RTNC_INVALID_OP` is returned.

Prototype

```
int tsfs_file_create ( const char * p_path );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path` Path of the file to be created.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_OP`
`RTNC_ALREADY_EXIST`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_file_delete()

<tsfs_file.h>

Deletes the file located at the given path.

The path must lead to an existing file of the working state. If the file does not exist, `RTNC_NOT_FOUND` is returned. If the file is not in the working state or the given path leads to a directory, `RTNC_INVALID_OP` is returned.

A opened file may safely be deleted. In this case, any further read/write access to this file will return `RTNC_NOT_FOUND`.

Prototype

```
int  tsfs_file_delete  ( const char *  p_path );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path to the file to be deleted.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_OP`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_file_exists()

<tsfs_file.h>

Verifies whether the file located at the given path exists. The function returns `true` (through the `p_exist` parameter) if the file exists and `false` otherwise (including when the given path leads to a directory).

Prototype

```
int  tsfs_file_exists  ( const char *  p_path,
                         bool *        p_exist );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path of the file which existence is to be tested.
`p_exist`	Whether the file located at the given path exists.

Returned Errors

`RTNC_SUCCESS`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_file_extent_min_sz_set()

<tsfs_file.h>

Sets the minimum extent size for the given file. The file must be empty, otherwise RTNC_INVALID_OP is returned.

Prototype

```
void  tsfs_file_extent_min_sz_set  ( const char *  p_path,
                                     uint8_t       min_sz_log2 );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

p_path	File path.
min_sz_log2	Minimum extent size (base-2 logarithm).

Function

tsfs_file_mode_reset()

<tsfs_file.h>

Resets an opened file access mode.

Prototype

```
void  tsfs_file_mode_reset  ( tsfs_file_hndl_t  hndl,
                              int               mode );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

hndl	Handle to the file which mode is to be altered.
mode	Access mode to be reset (only TSFS_FILE_MODE_RD_ONLY is currently supported).

Function

tsfs_file_mode_set()

<tsfs_file.h>

Sets an opened file access mode.

Prototype

```
void  tsfs_file_mode_set  ( tsfs_file_hndl_t  hndl,
                            int               mode );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`hndl`	Handle to the file which mode is to be altered.
`mode`	Access mode to be set (only `TSFS_FILE_MODE_RD_ONLY` is currently supported).

Function

tsfs_file_open()

<tsfs_file.h>

Opens the file located at the given path. If the file does not exist `RTNC_NOT_FOUND` is returned. If the given path leads to a directory, `RTNC_INVALID_OP` is returned.

Prototype

```
int  tsfs_file_open  ( const char *        p_path,
                       tsfs_file_hndl_t *  p_hndl );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path of the file to be opened.
`p_hndl`	Opened file handle.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_OP`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_file_read()

<tsfs_file.h>

Reads the requested amount of bytes from the given file.

The number of bytes read is returned through `p_rd_sz`. The returned value may be smaller than `rd_sz` if the end of the file is reached. If an error occurs, the value pointed to by `p_rd_sz` is unspecified.

Prototype

```
int  tsfs_file_read  ( tsfs_file_hndl_t    hndl,
                       void *              p_buf,
                       tsfs_file_size_t    rd_sz,
                       tsfs_file_size_t *  p_rd_sz );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`hndl`	Handle to the file to read from.
`p_buf`	Buffer to read into.
`rd_sz`	Number of bytes to be read from the file.
`p_rd_sz`	Actual number of bytes read from the file.

Returned Errors

`RTNC_SUCCESS`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_file_seek()

<tsfs_file.h>

Sets the current read/write position of the given file to the specified position.

Prototype

```
int  tsfs_file_seek  ( tsfs_file_hndl_t        hndl,
                       tsfs_file_pos_offset_t  offset,
                       int                     whence );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`hndl`	Handle to the file to seek into.
`offset`	New position relative to the supplied reference position (the `whence` parameter).
`whence`	Reference position to which `offset` is to be added.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

tsfs_file_size_get()

<tsfs_file.h>

Gets the size of the file located at the given path.

If the file does not exist, `RTNC_NOT_FOUND` is returned. If the given path leads to a directory, `RTNC_INVALID_OP` is returned.

Prototype

```
int  tsfs_file_size_get  ( const char *        p_path,
                           tsfs_file_size_t *  p_sz );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path of the file to get the size of.
`p_sz`	Size of the file in bytes.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_OP`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_file_truncate()`

<tsfs_file.h>

Shrinks or extends the file located at the given path.

The path must lead to an existing file of the working state. If the file does not exist, `RTNC_NOT_FOUND` is returned. If the file is not in the working state or the given path leads to a directory, `RTNC_INVALID_OP` is returned.

A file may safely be truncated while it is opened.

Prototype

```
int  tsfs_file_truncate  ( const char *       p_path,
                           tsfs_file_size_t   new_sz );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path to the file to be truncated.
`new_sz`	Size of the file after truncation in bytes.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_OP`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_file_write()`

<tsfs_file.h>

Writes the supplied buffer to the given file.

The number of bytes written is returned through `p_wr_sz`. The returned value may be smaller than `wr_sz` if, and only if, the file system is full. In this case `RTNC_FULL` is returned and the value pointed to by `p_wr_sz` indicates the number of bytes written before the file system becomes full.

Prototype

```
int  tsfs_file_write  ( tsfs_file_hndl_t    hndl,
                        const void *        p_buf,
                        tsfs_file_size_t    wr_sz,
                        tsfs_file_size_t *  p_wr_sz );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`hndl`	File handle.
`p_buf`	Buffer to be written.
`wr_sz`	Size of the buffer to be written in bytes.
`p_wr_sz`	Size of the written data in bytes.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FULL`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_format()

<tsfs.h>

Formats the given file system instance.

Prototype

```
int  tsfs_format  ( const char *  p_fs_name,
                    const void *  p_params );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the file system instance to be formatted.
`p_params`	Optional format parameters (set to NULL for default).

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_media_get()

<tsfs.h>

Gets the media used by the given file system instance.

Prototype

```
int  tsfs_media_get  ( const char *        p_fs_name,
                       bp_media_hndl_t *  p_media_hndl );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the file system instance.
`p_media_hndl`	Retrieved media handle.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_FATAL`

Function

`tsfs_mount()`

<tsfs.h>

Mounts the file system residing on the given media. If the media has not been previously formatted using `tsfs_format()` or the on-disk format is invalid, `RTNC_INVALID_FMT` is returned.

Prototype

```
int  tsfs_mount  ( const char *  p_fs_name );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the file system instance to be mounted.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_FMT`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_revert()`

<tsfs.h>

Returns the file system to the state it was in at the time of the given snapshot. If the given snapshot does not exist, `RTNC_NOT_FOUND` is returned.

Snapshots created after the revert snapshot are deleted. The revert operation is ended by an implicit commit.

Prototype

```
int  tsfs_revert  ( const char *  p_path );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path` Path to the snapshot to revert to.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_sshot_create()`

<tsfs.h>

Takes a snapshot of the current file system state. If the snapshot already exists, `RTNC_ALREADY_EXIST` is returned.

Prototype

```
int  tsfs_sshot_create  ( const char *  p_fs_name,
                          const char *  p_sshot_name );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name` Name of the file system instance.
`p_sshot_name` Name of the newly created snapshot.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_ALREADY_EXIST`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_sshot_delete()`

<tsfs.h>

Discards the given snapshot.

Prototype

```
int  tsfs_sshot_delete  ( const char *  p_fs_name,
                          const char *  p_sshot_name );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the file system instance.
`p_sshot_name`	Name of the snapshot to be deleted.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_sshot_exists()`

<tsfs.h>

Verifies whether the snapshot located at the given path exists. The function returns `true` (through the `p_exist` parameter) if the file exists and `false` otherwise.

Prototype

```
int  tsfs_sshot_exists  ( const char *  p_path,
                            bool *      p_exist );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path of the snapshot which existence is to be tested.
`p_exist`	Whether the snapshot located at the given path exists.

Returned Errors

`RTNC_SUCCESS`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_trace_data_get()`

<tsfs.h>

Gets the trace data used by the given file system instance.

Prototype

```
int  tsfs_trace_data_get  ( const char *  p_fs_name,
                              void **     pp_tdata );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the file system instance.
`pp_tdata`	Retrieved trace data.

Returned Errors
RTNC_SUCCESS
RTNC_NOT_FOUND
RTNC_FATAL

Function

tsfs_unmount()

<tsfs.h>

Unmounts the given file system instance.

Prototype
```
int tsfs_unmount ( const char * p_fs_name );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

p_fs_name	Name of the file system instance to be unmounted.

Returned Errors
RTNC_SUCCESS
RTNC_NOT_FOUND
RTNC_IO_ERR
RTNC_FATAL

Data Type

tsfs_file_pos_offset_t

<tsfs_file.h>

TSFS file position offset. This type is used to represent file position offsets. It may contain positive or negative position offsets.

Data Type

tsfs_file_size_t

<tsfs_file.h>

TSFS file size. This type is used to represent file sizes and positions.

Data Type

tsfs_cfg_t

<tsfs.h>

File system configuration structure.

Members

media_hndl	bp_media_hndl_t	Media handle to be bound to the created file system instance.

`max_entry_cnt`	`size_t`	Maximum number of simultaneously opened files or directories.
`p_seg`	`void *`	Memory segment for allocating internal file system structures.
`seg_sz`	`size_t`	Size of the memory segment provided to the file system.
`p_tdata`	`void *`	Trace data.
`p_ext_cfg`	`const void *`	Optional extended configuration. Set to `NULL` for default.

Data Type

tsfs_dir_hndl_t

<tsfs_dir.h>

TSFS directory handle. A directory handle is obtained through `tsfs_dir_open()`.

Data Type

tsfs_file_hndl_t

<tsfs_file.h>

TSFS file handle. A file handle is obtained through `tsfs_file_open()`. The file handle is internally tied to a file descriptor that contains the current read/write position.

Many file handles can be obtained for the same file, each handle being tied to a different file descriptor and thus a different and independent file position.

Macro

TSFS_FILE_MODE_RD_ONLY

<tsfs_file.h>

File access mode flags. Access mode flags only affect opened file instances. They do not alter on-disk file attributes.Allows for write protection on a per-file basis.

Macro

TSFS_MAX_INSTANCE_NAME_LEN

<tsfs.h>

Maximum number of characters in an instance name excluding the terminating null character.

Macro

TSFS_MAX_PATH_LEN

<tsfs.h>

Maximum number of characters in a file path excluding the terminating null character.

Macro

RTNC_*

<util/rtnc.h>

Return codes.

RTNC_SUCCESS	Function completed successfully.
RTNC_FATAL	Fatal error occurred.
RTNC_NO_RESOURCE	Resource allocation failure.
RTNC_IO_ERR	Transfer or peripheral operation failed.
RTNC_TIMEOUT	Function timed out.
RTNC_NOT_SUPPORTED	API, feature or configuration is not supported.
RTNC_NOT_FOUND	Requested object not found.
RTNC_ALREADY_EXIST	Object already created or allocated.
RTNC_ABORT	Operation aborted by software.
RTNC_INVALID_OP	Invalid operation.
RTNC_WANT_READ	Read operation requested.
RTNC_WANT_WRITE	Write operation requested.
RTNC_INVALID_FMT	Invalid format.
RTNC_INVALID_PATH	Invalid path.
RTNC_CORRUPT	Data corrupted.
RTNC_FULL	Container full.
RTNC_OVERFLOW	Overflow

Macro

TSFS_FILE_SEEK_*

<tsfs_file.h>

File seek flags. Indicate where the file offset should be applied from.

TSFS_FILE_SEEK_SET	Seek from the beginning.
TSFS_FILE_SEEK_CUR	Seek from the current position.
TSFS_FILE_SEEK_END	Seek from the end of the file.

Chapter

3

Document Revision History

The revision history of the TSFS user manual and reference manuals can be found within the TSFS source package.