

Reliance Edge
v2.5.1 (Build 864)

Generated by Doxygen

Contents

1 Reliance Edge Developer's Guide	1
2 Product Introduction	3
2.1 What is Reliance Edge?	3
2.1.1 Reliance Edge Features	3
2.2 Reliance Edge vs. FAT	4
2.3 Reliance Edge vs. Journaling File Systems	4
2.4 How Reliance Edge Works	4
2.5 Transaction Points	5
2.5.1 How Reliance Edge Preserves Data	5
2.5.2 Atomicity	5
2.5.3 Manual and Automatic Transaction Points	6
3 Porting Guide	7
3.1 Porting Process	7
3.2 System Requirements	7
3.2.1 C Language Implementation	7
3.2.2 Resource Requirements	7
3.2.3 Storage Medium	8
3.2.4 Application Interaction	8
3.3 OS Services	8
3.3.1 Introduction to the OS Services	8
3.3.2 Implementing the OS Services	9
4 FreeRTOS Integration	23
4.1 Assumptions	23
4.2 FreeRTOS Configuration	23
4.3 OS Services	23
4.3.1 Implementing Block Device Service for FreeRTOS	24

4.4 Building	25
4.5 Memory Allocation in FreeRTOS Version 9	25
5 INTEGRITY Integration	27
5.1 Reliance Edge: unified or client-server	27
5.2 Reliance Edge configuration on INTEGRITY	27
5.2.1 Configuration considerations	28
5.2.2 Additional configuration macros	28
5.3 Projects	29
5.3.1 Example projects	29
5.3.2 INTEGRITY ARM Simulator (simarm)	29
5.3.3 Renesas R-Car H3 Starter Kit (rcar-h3sk)	30
5.3.4 BeagleBone Black Application, Unified (bbb-app-unified)	30
5.3.5 BeagleBone Black Application, Client--Server (bbb-app-clientserv)	30
5.3.6 BeagleBone Black INTEGRITY Shell (bbb-shell)	30
5.3.7 Updating an existing MULTI project	30
5.4 Building	30
5.4.1 Building the file system	30
5.4.2 Building the tools for the target	32
5.4.3 Building the tools for the host	32
5.4.4 Building the tests	32
5.4.5 Building for the INTEGRITY Shell	32
5.5 Using Reliance Edge from an application	33
5.5.1 Mounting Reliance Edge	33
5.5.2 Reliance Edge paths	35
5.5.3 Recovering from a server fault	35
5.5.4 Reliance Edge INTEGRITY ioctls	35
5.5.5 Reliance Edge tool APIs	36
5.6 Reliance Edge and Special-Purpose Storage Drivers	37
5.6.1 Reliance Edge and the Multiplexor Driver	37
5.6.2 Reliance Edge and the Proxy Driver	38
5.7 Limitations	38
5.7.1 No support for root file system	38
5.7.2 Unimplemented functionality	38
5.7.3 Unintuitive CWD behavior	39
5.8 Behavioral Differences from Native File Systems	40
5.9 Legacy Block Device Interfaces	41

5.9.1	Device names	41
5.9.2	Enabling IDE/SATA implementation	41
5.9.3	Flushing the block device	41
5.9.4	Sharing block device with FFS or MSDOSFS	41
6	MQX Integration	43
6.1	MQX VFS Overview	43
6.2	Building Reliance Edge for MQX	43
6.2.1	The Reliance Edge KDS Project	43
6.2.2	The SD Card Demo KDS Projects	44
6.3	OS Services and Configuration	44
6.3.1	Mutex service	44
6.3.2	Timestamp service	44
6.3.3	Real-time clock service.	44
6.3.4	Block device service.	45
6.4	Supported Block Devices	45
6.5	Tests	45
7	ARM mbed Integration	47
7.1	Overview	47
7.2	ARM mbed filesystem abstraction	47
7.3	Reliance Edge configuration on ARM mbed	48
7.4	OS Services	48
7.4.1	Mutex and task services	48
7.4.2	Block device service	48
7.4.3	Assert service	48
7.5	Building	48
7.5.1	Creating and building an mbed classic project	49
7.5.2	Building for mbed OS	49
7.6	Testing	49
8	U-Boot Integration	51
8.1	Reliance Edge configuration for U-Boot	51
8.2	U-Boot project	51
8.3	Building U-Boot	52
8.4	Boot script for U-Boot	52
9	Building	53
9.1	Building the Reliance Edge Driver	53

9.1.1	Source Files and Include Directories	53
9.1.2	Building Reliance Edge with an IDE	56
9.1.3	Building Reliance Edge with a Makefile	56
9.1.4	Linking and Building as a Library	56
9.2	Building the Host Tools	57
9.3	Dealing with Compiler Warnings	57
10	POSIX-Like API Guide	59
10.1	When to Use the POSIX-Like API	59
10.2	Path Prefixes	60
10.3	Path Strings	60
10.3.1	Current Working Directory Support	61
10.3.2	Path Limits	61
10.4	File Descriptor Features	62
10.5	Errno and Task Count	62
10.6	Differences from POSIX	62
10.6.1	General Differences	62
10.6.2	Final Dot or Dot-Dot Errno	63
10.6.3	Individual APIs	64
10.7	Porting POSIX Apps	74
11	File System Essentials API Guide	75
11.1	Introduction	75
11.1.1	Characteristics	75
11.1.2	Benefits	75
11.1.3	Selection	75
11.2	Managing File Numbers	76
11.3	Limitations	76
11.3.1	No Handles	76
11.3.2	No Directories	76
11.3.3	No Creation or Deletion	76
11.3.4	No Querying File Attributes (fstat)	76
12	Product Configuration	77
12.1	Creating a Project	77
12.2	How to Configure Your Project	77
12.3	Product Configuration Guide	78
12.3.1	Read Only	78

12.3.2 Automatic Discards	79
12.3.3 POSIX-like API Configuration	79
12.3.4 File System Essentials API Configuration	82
12.3.5 Text Output	84
12.3.6 Assertion Handling	84
12.3.7 Block Size	84
12.3.8 Volume Configuration	85
12.3.9 Endianness	89
12.3.10 Alignment Size	89
12.3.11 CRC Algorithm	89
12.3.12 Inode st_blocks Field	90
12.3.13 Inode Timestamps	90
12.3.14 Access Time	90
12.3.15 Inode Pointer Configuration Values	91
12.3.16 Block Buffer Count	91
12.3.17 Memory and String Functions	92
12.3.18 Default Transaction Settings	92
12.3.19 Options Set Automatically	93
13 Discard Operations	95
13.1 Discard Availability in Reliance Edge SDKs	95
13.2 Discards 101: A Brief Introduction	95
13.2.1 Discard Definition	95
13.2.2 Discard Availability	95
13.2.3 Discard Benefits	95
13.2.4 Discard Drawbacks	96
13.3 Discards in Reliance Edge	96
13.3.1 Automatic Discards	97
13.3.2 File System Trim (Manual Discards)	97
14 Configuration Utility	101
14.1 Introduction	101
14.1.1 Installation and Setup	101
14.1.2 Versions and Compatibility	101
14.2 Editing settings	102
14.2.1 General	102
14.2.2 Volumes	104
14.2.3 Data	105

14.2.4 Memory	106
14.2.5 Transactions	107
14.2.6 Limits Tab	108
14.2.7 Other Settings	108
14.3 Saving, Loading, and Using the Configuration Files	109
14.3.1 Saving and using the configuration	109
14.3.2 Resolving invalid settings	110
14.3.3 Editing previously generated redconf files	110
14.3.4 Using the redconf files	110
14.4 Source Code	111
15 Command-Line Host Tools	113
15.1 Introduction	113
15.2 Availability	113
15.3 Formatter	113
15.3.1 Command-Line Syntax	113
15.4 Checker	114
15.4.1 Command-Line Syntax	114
15.5 Image Builder	115
15.5.1 Command-Line Syntax	115
15.5.2 Constraints	117
15.5.3 Copying an Image File	117
15.6 Image Copier	117
15.6.1 Command-Line Syntax	117
15.6.2 Constraints	118
15.7 Linux FUSE Implementation	118
15.7.1 Command-Line Syntax	118
15.7.2 Using the FUSE Implementation	119
16 By the Numbers	121
16.1 Limits	121
16.1.1 Maximum File Sizes	121
16.1.2 Maximum Volume Sizes	122
16.1.3 Maximum Path Length	123
16.1.4 Maximum Name Length	123
16.1.5 Allowable Characters in File Names	123
16.2 Resource Consumption	124
16.2.1 RAM Usage	124

16.2.2 ROM Size	125
16.2.3 Maximum Stack Depth	125
16.3 Worst-Case I/O Counts	125
16.3.1 Mount	126
16.3.2 Transaction Point	126
16.3.3 File I/O	126
17 Use Case Guide	131
17.1 Introduction	131
17.2 Use Case A: System Update	131
17.2.1 Description	131
17.2.2 Goal	131
17.2.3 Planning	131
17.2.4 Approach	131
17.2.5 Benefits	131
17.2.6 Process in Detail	132
17.3 Use Case B: Logging Application	132
17.3.1 Description	132
17.3.2 Goal	132
17.3.3 Planning	132
17.3.4 Approach	133
17.3.5 Process in Detail	133
17.4 Use Case C: Controlling Data-at-Risk on Power Failure	133
17.4.1 Description	133
17.4.2 Goal	133
17.4.3 Approach	133
17.4.4 Benefits	134
18 Testing	135
18.1 Introduction	135
18.2 BDevTest: Block Device Performance Test	135
18.2.1 Command-Line Syntax	136
18.3 Disk Full Tests	137
18.3.1 Command-Line Syntax	137
18.4 FSE API Test Suite	138
18.4.1 Command-Line Syntax	138
18.5 FSE API Stress Test	138
18.5.1 Command-Line Syntax	138

18.6 FSIOTest: File System Performance Test	139
18.6.1 Command-Line Syntax	139
18.7 Multi-Volume Stress Test	141
18.7.1 Command-Line Syntax	141
18.8 fsstress Test	141
18.8.1 Command-Line Syntax	141
18.9 POSIX-like API Test Suite	142
18.9.1 Command-Line Syntax	142
18.10 OS-Specific API Test Suite	143
18.10.1 Command-Line Syntax	143
18.11 Stochastic Test	143
18.11.1 Command-Line Syntax	144
18.12 Running the Tests on Your Target Hardware	144
18.13 Specialized Test Projects	145
18.13.1 Simulated Power Interruption Project	145
18.13.2 I/O Error Injection Project	145
18.13.3 Discard Tests Project	145
18.13.4 FSTrim Test Project	145
19 Module Index	147
19.1 API References	147
20 Hierarchical Index	149
20.1 Class Hierarchy	149
21 Data Structure Index	151
21.1 Data Structures	151
22 File Index	153
22.1 File List	153
23 Module Documentation	155
23.1 The POSIX-like File System Interface	155
23.2 The File System Essentials Interface	157
24 Data Structure Documentation	159
24.1 BDEVINFO Struct Reference	159
24.1.1 Detailed Description	159
24.1.2 Field Documentation	159

24.2 RED_DIR_PARAM Struct Reference	160
24.2.1 Detailed Description	160
24.3 red_dirent Struct Reference	160
24.3.1 Detailed Description	160
24.4 RED_RENAME_PARAM Struct Reference	160
24.4.1 Detailed Description	160
24.5 REDDIRENT Struct Reference	161
24.5.1 Detailed Description	161
24.5.2 Field Documentation	161
24.6 RedDirHandle Class Reference	162
24.6.1 Detailed Description	162
24.6.2 Member Function Documentation	162
24.7 RedFileHandle Class Reference	163
24.7.1 Detailed Description	164
24.7.2 Member Function Documentation	164
24.8 RedFileSystem Class Reference	168
24.8.1 Detailed Description	169
24.8.2 Constructor & Destructor Documentation	169
24.8.3 Member Function Documentation	170
24.9 RedMemFileSystem Class Reference	175
24.9.1 Detailed Description	175
24.10 RedSDFileSystem Class Reference	176
24.10.1 Detailed Description	177
24.10.2 Member Enumeration Documentation	178
24.10.3 Constructor & Destructor Documentation	179
24.10.4 Member Function Documentation	179
24.11 REDSTAT Struct Reference	182
24.11.1 Detailed Description	182
24.11.2 Field Documentation	182
24.12 REDSTATFS Struct Reference	184
24.12.1 Detailed Description	185
24.12.2 Field Documentation	185
25 File Documentation	189
25.1 fse/fse.c File Reference	189
25.1.1 Detailed Description	190
25.1.2 Function Documentation	190

25.2 include/redapimacs.h File Reference	198
25.2.1 Detailed Description	199
25.2.2 Macro Definition Documentation	199
25.3 include/rederrno.h File Reference	202
25.3.1 Detailed Description	204
25.3.2 Macro Definition Documentation	204
25.4 include/redfse.h File Reference	208
25.4.1 Detailed Description	209
25.4.2 Macro Definition Documentation	209
25.4.3 Function Documentation	209
25.5 include/redoserv.h File Reference	217
25.5.1 Enumeration Type Documentation	218
25.5.2 Function Documentation	220
25.6 include/redposix.h File Reference	226
25.6.1 Detailed Description	228
25.6.2 Macro Definition Documentation	229
25.6.3 Enumeration Type Documentation	231
25.6.4 Function Documentation	231
25.7 include/redstat.h File Reference	257
25.7.1 Macro Definition Documentation	257
25.8 os/mbed/RedFileSystem/RedDirHandle.cpp File Reference	258
25.8.1 Detailed Description	258
25.9 os/mbed/RedFileSystem/RedDirHandle.h File Reference	258
25.9.1 Detailed Description	258
25.10 os/mbed/RedFileSystem/RedFileHandle.cpp File Reference	258
25.10.1 Detailed Description	259
25.11 os/mbed/RedFileSystem/RedFileHandle.h File Reference	259
25.11.1 Detailed Description	259
25.12 os/mbed/RedFileSystem/RedFileSystem.cpp File Reference	259
25.12.1 Detailed Description	259
25.13 os/mbed/RedFileSystem/RedFileSystem.h File Reference	259
25.13.1 Detailed Description	259
25.13.2 Macro Definition Documentation	260
25.14 os/mbed/RedFileSystem/RedMemFileSystem.h File Reference	260
25.14.1 Detailed Description	260
25.15 os/mbed/RedSDFFileSystem/RedSDFFileSystem.cpp File Reference	260
25.15.1 Detailed Description	260

25.16os/mbed/RedSDFFileSystem/RedSDFFileSystem.h File Reference	260
25.16.1 Detailed Description	261
25.17os/mqx/include/redfs_mqx.h File Reference	261
25.17.1 Detailed Description	262
25.17.2 Macro Definition Documentation	262
25.17.3 Function Documentation	266
25.17.4 Variable Documentation	267
25.18os/mqx/vfs/redfs_fio.c File Reference	268
25.18.1 Detailed Description	268
25.19os/mqx/vfs/redfs_nio.c File Reference	268
25.19.1 Detailed Description	268
25.19.2 Function Documentation	268
25.20os/stub/include/redostypes.h File Reference	270
25.20.1 Detailed Description	270
25.20.2 Typedef Documentation	270
25.21os/stub/services/osassert.c File Reference	270
25.21.1 Detailed Description	271
25.21.2 Function Documentation	271
25.22os/stub/services/osbdev.c File Reference	271
25.22.1 Detailed Description	271
25.22.2 Function Documentation	272
25.23os/stub/services/osclock.c File Reference	276
25.23.1 Detailed Description	276
25.23.2 Function Documentation	277
25.24os/stub/services/osmutex.c File Reference	278
25.24.1 Detailed Description	278
25.24.2 Function Documentation	278
25.25os/stub/services/osoutput.c File Reference	280
25.25.1 Detailed Description	280
25.25.2 Function Documentation	280
25.26os/stub/services/ostask.c File Reference	280
25.26.1 Detailed Description	281
25.26.2 Function Documentation	281
25.27os/stub/services/ostimestamp.c File Reference	281
25.27.1 Detailed Description	281
25.27.2 Function Documentation	282
25.28posix posix.c File Reference	283

25.28.1 Detailed Description	285
25.28.2 Function Documentation	285
25.29 projects/doxygen/redconf.h File Reference	310
25.29.1 Macro Definition Documentation	312
25.30 projects/doxygen/redtypes.h File Reference	318
25.30.1 Detailed Description	318
25.30.2 Typedef Documentation	319
25.31 projects/mqx/TWR-K65F180M/RelianceEdge/redosconf.c File Reference	321
25.31.1 Variable Documentation	321
Index	323

Chapter 1

Reliance Edge Developer's Guide

This manual is divided into the following chapters:

- [Product Introduction](#)
- [Porting Guide](#)
- [FreeRTOS Integration](#)
- [INTEGRITY Integration](#)
- [MQX Integration](#)
- [ARM mbed Integration](#)
- [U-Boot Integration](#)
- [Building](#)
- [POSIX-Like API Guide](#)
- [File System Essentials API Guide](#)
- [Product Configuration](#)
- [Discard Operations](#)
- [Configuration Utility](#)
- [Command-Line Host Tools](#)
- [By the Numbers](#)
- [Use Case Guide](#)
- [Testing](#)

The [Product Introduction](#) should be read first; the rest can be read in any order.

Other Documentation

Please read the README.md and doc/release_notes.md files that are included with Reliance Edge for a concise introduction and for other pertinent information about the release.

Copyright and Trademark Notice

Copyright © 2021 Tuxera Inc. and/or its affiliates. All rights reserved. TUXERA, Tuxera's logo, Datalight, FlashFX, FlashFX Tera, Fusion File Share by Tuxera, GravityCS by Tuxera, Reliance, Reliance Nitro, Reliance Assure, Reliance Velocity, ROM-DOS, are trademarks or registered trademarks of Tuxera Inc. All other product names are trademarks of their respective holders. Specification and price change privileges reserved.

Chapter 2

Product Introduction

This chapter is an introduction to Reliance Edge and its features.

2.1 What is Reliance Edge?

Reliance Edge is a small, portable, reliable file system for resource-constrained embedded systems like microcontrollers. It is *small* because it does not require much code space on your ROM or NOR flash and because it does not need much RAM (see [Resource Consumption](#)); it is *portable* because it is written in portable C and is designed to work in a wide variety of embedded environments; it is *reliable* because it implements an atomic, transactional model where changes either occur in their entirety or not at all, which is to say, the file system reverts to a "known good state" after system failures such as power loss.

2.1.1 Reliance Edge Features

- Transaction-based update model prevents data loss due to file system corruption.
- No need to check (CHKDSK/fsck) the file system at boot time and no journal to replay at mount time, resulting in consistently fast boot and mount times.
- Provides a POSIX-like File System API; or alternatively, a no-frills File System Essentials API for applications with simple storage needs.
- Highly configurable: can be tuned to be optimal for your application. File system code not used by your application can be disabled to reduce code size and eliminate dead code.
- File system driver conforms to MISRA-C:2012 (guidelines for the use of the C language in critical systems).
- Deterministic performance for key file system operations: see [Worst-Case I/O Counts](#).
- Readily portable to new embedded environments.
- Media independent: can use any storage media with a block device interface.
- Supports large files, large volumes, and long file names: see [Limits](#).
- Supports multiple file system volumes.
- Supports locking to allow multiple tasks to use the file system.

2.2 Reliance Edge vs. FAT

Many developers choose a FAT file system for their embedded devices without understanding its shortcomings. The FAT file system was developed in the early 1980s and has been modified since then primarily to support larger storage media. Its metadata includes the file names and attributes, the directory names and attributes, and a large data structure that tracks the location of the sectors that make up the specific files contained on the disk. This data structure is the file allocation table, from which the acronym FAT is derived, and is the key to accessing every file on the disk.

The application program, after reading user data (a file) from the disk drive into the cache buffers, can modify the contents of the buffers by changing, manipulating, and replacing data. At some point these cache buffers are flushed to the disk in order to store the modified user data, and also metadata, on the disk. Multiple separate writes to the disk are required in order to get the on-disk state to reflect the cached state. If the system fails due to a power loss or system crash prior to completing all of these writes to disk, the FAT file system could suffer from some degree of corruption. The symptoms of file system corruption include lost files, cross-linked data clusters, more than one file of a given name, files without names, and so on; usually a complicated mess that causes the file system to be deemed unreliable. Any subsequent file-recovery process can be quite lengthy and may not be entirely successful.

While conventional file systems are prone to data loss or corruption during the writing of data to the associated storage media, Reliance Edge protects the file system and maintains data integrity. Reliance Edge avoids the need to perform data-recovery operations because the original data is not overwritten, but is preserved until the new data is confirmed as written to the storage media. If an interruption does occur during the writing of data, the previous state of the file system is still available on the media, allowing the file system to always be in a valid state. This method of file system protection is similar in concept to that of a bank transaction, where a determination is made that a transaction was successfully completed and new data is in place, or the transaction failed during the process, and the original data has been preserved and is available.

2.3 Reliance Edge vs. Journaling File Systems

The most common method of avoiding the reliability problems of FAT and other conventional file systems is to use a *journal*. In the embedded space, this includes offerings from several vendors that sell versions of FAT which have been retrofitted to use a journal. Data loss and corruption in a conventional file system such as FAT is usually due to a failure within the embedded device such as a system crash or loss of power during a data write operation which leaves the file system in an inconsistent state. A file system that maintains a detailed log (or journal) of file system modifications offers protection against such data loss; however, the journaling/logging of file system change data requires several additional write operations to complete a file I/O operation. This additional overhead can have a significant impact on system performance and can result in additional media wear which reduces the lifespan of the device.

Reliance Edge uses a system of transactions to provide a comparable level of file system reliability without the overhead of a journal. The reliability of a transactional file system, such as Reliance Edge, is based on the preservation of the original data until a transaction event. This scheme provides a high degree of reliability, but with less overhead and improved performance compared to a journaling file system.

2.4 How Reliance Edge Works

The reliability of Reliance Edge is based on the preservation of the original data until the writing of new data is complete and a transaction point is committed. All file system modifications, including changes to directories, files, and metadata, are stored on an area of the storage media that is currently unused and does not contain "live" data. The possibility of corruption is eliminated because the data of the new transaction is written to an unused portion of the media, preserving the file system's original state. Consequently, in the event that a system error, such as loss of power, occurs during a write operation to the file system, the valid data from the previous transaction is still available.

In addition to providing complete reliability for file I/O operations, Reliance Edge eliminates slow boot times caused by

lengthy file system checks, especially those performed on large media. Because the Reliance Edge file system is always in a valid state, even at start up, checks or scans of the storage media as part of system initialization are unnecessary. The elimination of the storage media check can reduce the boot time, especially in systems which have a large amount of storage. This allows fast startup times for the target device.

2.5 Transaction Points

A significant event in the operation of the Reliance Edge file system is the setting of a transaction point, which atomically makes the current working state of the file system into the transacted state. To put it more verbosely, when Reliance Edge is mounted the state of the file system (that is, all the data and metadata) is exactly as it was at the time of the last transaction point. When changes are made to the file system, either adding new data/metadata or modifying or deleting existing data/metadata, these deltas (changes) are working state data. If there is a system failure like power loss, the working state data is lost, and the file system reverts to the transacted state. A transaction point is the process of atomically incorporating the working state deltas into the transacted state. The transaction point is termed *atomic* since it either completes entirely or it has no effect at all; the transaction point, even if interrupted, *never* transacts only a portion of the working state deltas; it is all or nothing.

The working state deltas, although not yet permanent, are still reflected in the state of the file system as seen the application. Thus, for example, if a file is created, deleted, or rewritten, those changes are visible to the application even if no transaction point has incorporated those changes into the transacted state.

2.5.1 How Reliance Edge Preserves Data

Reliance Edge keeps track of unused or free data blocks on the storage media and always writes data to these blocks. Reliance Edge does not overwrite existing data blocks (it is a *copy-on-write* file system), so the previous state of the file system remains intact on the storage media as updates are made.

After some amount of new data has been successfully written, a transaction point can be set, and then all new data is committed to the file system and storage media. After the transaction point is set, the previous data blocks are now made available for a subsequent write operation. If there is a system failure, only the new data written since the last transaction point is lost; the original data is preserved and available when the device returns to operation.

If there is insufficient available free space on the storage media in order to complete a write operation, Reliance Edge detects the "disk full" condition and sets a transaction point. With a new transaction point set, Reliance Edge then frees any blocks previously marked as unused and continues the write operation. This functionality within Reliance Edge is configurable and thus can be disabled if required.

At each transaction point, the previously-valid data blocks which were deleted or superseded become available for reuse.

2.5.2 Atomicity

Transaction points are fully atomic since all metadata is pointed-at by a metadata block called the *metaroot* (metadata root node), of which there are two copies, each with a counter, allowing the file system to determine the more recent copy. Writing a metaroot block to disk is the final step in setting a transaction point, and the transaction point has not occurred until the metaroot is written. Writing the metaroot is itself atomic, even if it spans multiple sectors, since the entire metaroot has a checksum (or more accurately, a CRC), and if the metaroot block was not written completely, the checksum will fail to correspond to the contents of the block. If the metaroot is thus detected to be incomplete, the transaction did not finish and the previous transacted state as pointed-at by the other metaroot is used.

2.5.3 Manual and Automatic Transaction Points

A transaction point can be set by the application manually, by calling the `red_transact()` or `RedFseTransact()` APIs. Optionally, a number of events can be individually configured to cause a transaction; these are termed *automatic transaction events*, and most users will have some automatic transaction events enabled. There are events for several types of write operations (like closing a file) and (as mentioned earlier) an event for a "disk full" condition. See [Default Transaction Settings](#) for a discussion of configuring default automatic transaction events. Both the POSIX-like and File System Essentials APIs provide an API that lets the application change the automatic transaction event mask at run-time.

Chapter 3

Porting Guide

Reliance Edge is a portable file system capable of running in a wide variety of embedded hardware and software environments, and can be used with numerous RTOSes and even in simple no-RTOS systems. Read this section to learn how to get Reliance Edge up and running on your hardware and interfacing with your drivers and systems software.

3.1 Porting Process

There are four steps involved in a Reliance Edge porting effort:

1. Determine whether the target system is capable of running Reliance Edge. This is discussed in the next section, [System Requirements](#).
2. Create a project tuned for the use-case and hardware. This is detailed in the separate [Product Configuration](#) chapter.
3. Cross-compile Reliance Edge with stubbed OS services. This is detailed in the separate [Building](#) chapter.
4. Implement the OS services for the operating environment. This is covered below, in the [OS Services](#) section.

3.2 System Requirements

3.2.1 C Language Implementation

Reliance Edge is written entirely in portable C (ISO C90 / ANSI C89). So, in order to use the file system, you must have a C implementation (a C run-time library) and a cross-compiler for your hardware. A freestanding C implementation is acceptable: Reliance Edge does *not* depend on the standard C library (for example, it does *not* use `malloc`). If you will be or already are running software on your target hardware written in C, this requirement is met.

3.2.2 Resource Requirements

Your system must have enough RAM, stack space, and code storage for Reliance Edge. The exact resource requirements depend on how Reliance Edge is configured and upon the compiler, but typical minimums are given in the [Resource Consumption](#) section.

3.2.3 Storage Medium

Reliance Edge, like any file system, needs a storage medium to keep its data on, for example, an SD/MMC card, eMMC, CompactFlash, USB flash drive, non-volatile RAM, or a rotating hard disk. Any type of media will work provided that it is possible to read sectors, write sectors, and (if write caching exists) flush sectors—essentially, Reliance Edge needs a standard block device interface. If this interface is not natively supported by the storage medium, it must be emulated: for example, if using "raw" flash memory (with a page read/program and block erase interface) a flash translation layer (FTL) will be needed to emulate a block device interface.

Note

If you are using raw NAND flash memory (such as SPI NAND) as your storage device, Reliance EdgeNAND might be a better fit for your project than the standard release of Reliance Edge. Contact sales@tuxera.com for details.

See the [block device](#) section for further detail on how Reliance Edge will interact with the storage medium.

The storage medium plays a key role in the reliability of the overall storage solution. Using a reliable file system like Reliance Edge with an unreliable storage medium will result in a less-than-reliable solution. If you are using an SD card or eMMC device with buggy firmware, or a solution that uses flash memory that allows power failures to interrupt page program operations, file system corruption is a possibility regardless of which file system you use. That said, putting Reliance Edge on an unreliable storage medium may nonetheless result in a solution which is at least *more* reliable than putting an unreliable file system on that medium, since a source of failure is nonetheless removed from the system.

3.2.4 Application Interaction

Reliance Edge is designed to be called directly by applications which need file system functionality. This works well in monolithic environments, where the application code and the file system code run in a common address space—most embedded systems fit this description. Even if the address space is segmented, provided that the applications using the file system can run in the same address space as Reliance Edge, it will be possible for those applications to directly invoke Reliance Edge APIs.

A few comments addressed to users of real-time operating systems (RTOSes) that includes a Virtual File System (VFS). A VFS is an operating system abstraction that allows different file systems (for example, Reliance Edge and FAT) to be accessed via a common interface. If your RTOS has a VFS, it may be possible to integrate Reliance Edge into it, though this requires a great deal more expertise and labor than using Reliance Edge directly. However, just because a VFS exists does not mean it must be used. In a monolithic environment, an application can bypass the VFS and call Reliance Edge APIs directly. In most cases this will be considerably easier than VFS integration. If VFS integration is a requirement, you will need to determine how to translate between the VFS interface and Reliance Edge. VFS interfaces differ considerably in their functionality and complexity, so the optimal integration approach will vary. VFS integration is beyond the scope of this manual; if you need help with this, contact support@tuxera.com.

3.3 OS Services

3.3.1 Introduction to the OS Services

Reliance Edge requires a handful of basic services from the operating environment (that is, from the RTOS or equivalent systems software), but the details of accessing these services differ from environment to environment. Directly interfacing with the RTOS, system drivers, or hardware is inherently non-portable. Instead, Reliance Edge defines an OS Services API—an abstraction which can be implemented in a wide variety of environments. This abstraction hides the differences between operating environments and allows almost all of the file system driver code to be the same in all environments.

The OS Services API is designed to be simple and easy to implement for a developer who is familiar with the operating environment.

3.3.1.1 Undefined Behavior

The documentation of many OS Services APIs specifies *undefined behavior*. Reliance Edge never uses the OS services in such a way that these undefined behaviors occur. The reason for having undefined behavior is to make it easier to implement the OS services. For example, the behavior of attempting to recursively acquire the mutex is undefined (see [RedOsMutexAcquire\(\)](#)). This allows the mutex to be implemented with a semaphore (which does not support recursive acquisition) or a mutex (which does), whichever is available. In many cases, the undefined behavior reduces the burden of checking for errors. For example, calling [RedOsMutexAcquire\(\)](#) without having first initialized the mutex via [RedOsMutexInit\(\)](#) is undefined behavior, so the implementation of [RedOsMutexAcquire\(\)](#) can assume this never happens and is not required to check that the mutex has been initialized.

To explicitly define expected behavior for use cases Reliance Edge does not actually need just makes the OS services harder to implement, without benefiting anyone.

In brief, documented "undefined behavior" represents conditions that never happen and which you do not need to worry about when implementing the OS Services API.

3.3.1.2 Required OS Services

Not all users need to implement every OS service. The only OS service which is always required is the block device service (since a file system cannot function without storage). The others may or may not be needed, depending on use case: in the detailed discussion of each service (below), the circumstances under which it is required are given.

3.3.1.3 Starting Point: Stubbed Services

In the os/stub directory, there are files and subdirectories which provide a "stubbed" implementation of the OS services. This stubbed implementation includes the necessary files populated with stubbed functions and documentation comments for all the OS service APIs. The stubbed implementation will compile without error in any environment but is not intended to actually work—it is designed to assist with building Reliance Edge before the port is complete, and as a starting point for implementing the OS Services API.

The recommended way to start working on the OS services is to copy the os/stub directory to os/OSNAME, where OSNAME is the name of the RTOS or operating environment you are using. The convention is to use lowercase letters, for example, os/threadx if using ThreadX.

After copying that directory, it might be a good time to figure out how you will be building Reliance Edge, so that you can compile the OS Services code as you write it. See the [Building](#) section for recommendations.

3.3.2 Implementing the OS Services

This section reviews each of the OS services and provides tips on how to implement them. In addition to the text in this manual, it is recommended to read the API descriptions for each OS service function (linked from the function name). Implementing these services involves editing the following source files (where OSNAME is the name of the RTOS directory you created, copying from os/stub):

- os/OSNAME/services/osbdev.c – [Block device](#) service.
- os/OSNAME/services/osmutex.c – [Mutex](#) service.
- os/OSNAME/services/ostask.c – [Task ID](#) service.

- os/OSNAME/services/osclock.c – [Real-time clock](#) service.
- os/OSNAME/services/osoutput.c – [Text output](#) service.
- os/OSNAME/services/osassert.c – [Assertion handler](#) service.
- os/OSNAME/services/ostimestamp.c – [Timestamp](#) service.
- os/OSNAME/include/redostypes.h – OS types, currently used just for the [Timestamp](#) service and described in that section.
- os/OSNAME/include/redosdeviations.h – OS MISRA C:2012 deviations. This only needs to be modified when creating a MISRA C compliant port (see the FreeRTOS version for an example); otherwise it can be left empty.

Renaming or refactoring these source files is not recommended, since it will break the supplied Makefiles and make your port harder to understand by others.

3.3.2.1 Block Device (osbdev.c)

The block device service implements an interface that allows Reliance Edge to issue commands to the storage device (or multiple storage devices). If you are using an RTOS that already has a block device abstraction, the implementations of these functions will simply need to translate between that interface and the Reliance Edge block device interface. If no block device abstraction exists, then the implementation of these functions will need to interact with the storage media driver directly. This interaction either involves adding calls to your driver functions (assuming the driver is already written), or implementing the driver in-place in [osbdev.c](#).

The FreeRTOS implementation of the block device service (in os/freertos/services/osbdev.c) is a good reference. Since FreeRTOS does not have a block device abstraction, a generic FreeRTOS implementation is not possible; instead, there are several separate versions of the block device implemented in that file, conditioned on preprocessor logic. One implementation is designed to re-use block device code written for a FAT file system implementation; this demonstrates block device interface translation, and for former users of that version of FAT, may have utility for re-using existing code. A second implementation interfaces with the Atmel Software Framework SD/MMC driver. A fourth implementation is a RAM disk. Early in the project, especially if the storage media drivers are incomplete, it might make sense to use a RAM disk implementation as a temporary solution to allow some degree of testing.

All of the block device functions take a volume number parameter which indicates the volume for which an I/O operation is being done. If you plan on using multiple Reliance Edge volumes, you are responsible—as part of implementing this service—for mapping the volume numbers to the separate storage devices or partitions where each volume lives. For example, suppose a system with three volumes and three storage devices (such as on-board eMMC, a removable SD card, and a small RAM disk): the implementations of these functions would be responsible for looking at the volume number and using the correct device. If there is partitioning, Reliance Edge supports specifying a [Sector Offset](#) for each volume. If you choose not to use that feature, you may implement partitioning yourself by having the implementations of these functions be responsible for using the volume numbers to add the necessary offsets to the sector numbers, such that the requests for each volume are directed to the correct portion of the media (these offsets could be hard-coded or read from an MBR in sector zero during initialization).

Both [RedOsBDevRead\(\)](#) and [RedOsBDevWrite\(\)](#) support multi-sector transfers, and Reliance Edge makes use of that feature—it always reads or writes at least a block at a time, so if the block size is larger than the sector size, there will be multi-sector requests. Furthermore, Reliance Edge will read and write multiple blocks at a time (and hence multiple sectors) when sufficiently large chunks of file data are read or written. Ideally, the implementation of the block device service will be capable of handling these multi-sector requests efficiently. While it is valid to read or write one sector at a time in a loop to handle a multi-sector request, it is typically several times faster to transfer all the data in one storage media request. So if your use case will result in multi-sector requests, and performance is important to your application, an effort should be made to natively support multi-sector transfers.

If your storage medium suffers from transient I/O failures, you might consider setting [Block I/O Retries](#) to be greater than zero. That way, if one of the block device I/O functions—[RedOsBDevRead\(\)](#), [RedOsBDevWrite\(\)](#), [RedOsBDevFlush\(\)](#),

or [RedOsBDevDiscard\(\)](#)—fails, then Reliance Edge will automatically try it again, up to the maximum amount of retries. However, be aware that such an approach impacts [Worst-Case I/O Counts](#). If your implementation has transient I/O failures in [RedOsBDevOpen\(\)](#) or [RedOsBDevClose\(\)](#), then you can add retry-loop logic to your implementation.

Six functions must be implemented for the block device service: [RedOsBDevOpen\(\)](#), [RedOsBDevClose\(\)](#), [RedOsBDevGetGeometry\(\)](#), [RedOsBDevRead\(\)](#), [RedOsBDevWrite\(\)](#), and [RedOsBDevFlush\(\)](#). These are discussed below.

[RedOsBDevOpen](#)

[RedOsBDevOpen\(\)](#) is called during mount or format to initialize the block device. If it is necessary to initialize the storage device, the code for doing so can be implemented in or called from this function. For example, the Atmel SD/MMC implementation calls an ASF function which readies the SD card:

```
uint32_t      ulTries;
Ctrl_status cs;

// The first time the disk is opened, the SD card can take a while to get
// ready, in which time sd_mmc_test_unit_ready() returns either CTRL_BUSY
// or CTRL_NO_PRESENT. Try numerous times, waiting half a second after
// each failure. Empirically, this has been observed to succeed on the
// second try, so trying 10x more than that provides a margin of error.
//
for(ulTries = 0U; ulTries < 20U; ulTries++)
{
    cs = sd_mmc_test_unit_ready(0U);
    if((cs != CTRL_NO_PRESENT) && (cs != CTRL_BUSY))
    {
        break;
    }

    vTaskDelay(500U / portTICK_PERIOD_MS);
}
```

If the storage media is partitioned, [RedOsBDevOpen\(\)](#) can read the MBR to determine the absolute starting sector number for each volume. Alternatively, if the partitions are always the same, the offsets can be hard-coded into the implementation.

[RedOsBDevOpen\(\)](#) includes an [BDEVOOPENMODE](#) parameter, with three possible values: [BDEV_O_RDONLY](#), [BDEV_O_WRONLY](#), and [BDEV_O_RDWR](#). The most common value is [BDEV_O_RDWR](#) and most implementations can ignore the open mode. However, the [BDEV_O_RDONLY](#) mode is used if the volume is mounted as read-only and for the checker (which is primarily a host tool). If the media is read-only (say a write-protected SD card), an implementation may choose to report an error for [BDEV_O_RDWR](#) or [BDEV_O_WRONLY](#) but allow [BDEV_O_RDONLY](#), but this is not required. Likewise, an implementation may choose to (for example) disallow write operations when opened with the [BDEV_O_RDONLY](#) mode, but this is not required.

[RedOsBDevClose](#)

[RedOsBDevOpen\(\)](#) is called when Reliance Edge is finished with the block device, either when the volume is unmounted or at the conclusion of format. The implementation of this function can free any resources allocated by [RedOsBDevOpen\(\)](#). When [RedOsBDevClose\(\)](#) completes, it must be safe to call [RedOsBDevOpen\(\)](#) again. The implementation of this function may "release", "close", or "uninitialize" the driver or abstraction it is working with, though in some cases the implementation will do nothing and return success.

[RedOsBDevGetGeometry](#)

[RedOsBDevGetGeometry\(\)](#) is called during mount or format to retrieve the sector geometry of the block device, namely, the sector size and the sector count. If autodetection of the sector geometry is enabled (see [Sector Size](#) and [Sector Count](#)), this function is how that information is reported to Reliance Edge. If the sector geometry is statically configured, the values returned by this function are used to ensure that the statically configured geometry is compatible with the block device.

To implement this function, the underlying block device API or block device driver must have a means of querying the

sector count and possibly also the sector size. Usually, there will be a method to query the sector count or another value from which the sector count can be derived, such as the device capacity or size. The sector size is usually either query-able or implicitly fixed: as an example of the latter, many SD card drivers only support a sector size of 512, so that would be the sector size value to return.

The sector count which is returned should normally be the sector count of the entire storage device: if the [Sector Offset](#) feature is used to implement partitioning, there is shared code that will take care of subtracting the sector offset from the sector count to yield the volume's sector count.

If [RedOsBDevGetGeometry\(\)](#) cannot be implemented in your environment, it can be stubbed to return `-RED_ENOTSUPP`, meaning "not supported". If this function is not supported, then sector geometry autodetection cannot be used: the sector count and sector size for each volume must be statically configured.

For a RAM disk, there is not an obvious value to return for the sector count and sector size, since there is no storage device to query. If there are volume numbers which are implemented as RAM disks, it would be reasonable to return `-RED_ENOTSUPP` for those volume numbers, thus requiring that their sector geometries are statically configured. The several RAM disk implementations that are included in Reliance Edge all take this approach.

RedOsBDevRead

[RedOsBDevRead\(\)](#) is called to read data from a specified range of sectors into the given buffer. The read must be synchronous: [RedOsBDevRead\(\)](#) cannot return until all the data is copied into the buffer.

Regarding the sector numbers, `ullSectorStart` has a minimum value of the [Sector Offset](#) for the volume and `(ullSectorStart + ulSectorCount)` has a maximum value of the [Sector Offset](#) added to the [Sector Count](#).

RedOsBDevWrite

[RedOsBDevWrite\(\)](#) is called to write data from the given buffer to a specified range of sectors. The write is not required to be synchronous; it is acceptable to return after copying the write into a cache or passing it along to an I/O scheduler, provided that [RedOsBDevFlush\(\)](#) is implemented properly.

Regarding the sector numbers, `ullSectorStart` has a minimum value of the [Sector Offset](#) for the volume and `(ullSectorStart + ulSectorCount)` has a maximum value of the [Sector Offset](#) added to the [Sector Count](#).

RedOsBDevFlush

[RedOsBDevFlush\(\)](#) is called to flush all data written to a volume by [RedOsBDevWrite\(\)](#) to a permanent location on the storage medium. This means that all software and hardware caches underlying the specified volume must be flushed. Implementing this function correctly is critically important—failure to do so may undermine the reliability of Reliance Edge and corrupt the file system.

Reliance Edge uses [RedOsBDevFlush\(\)](#) to enforce write I/O ordering. Suppose that the driver writes sector A, calls [RedOsBDevFlush\(\)](#), and then writes sector B. If power failure interrupts this sequence, after reboot, the media should be in one of three states: 1) neither sector A nor sector B were written; 2) only sector A was written; or 3) both sector A and sector B were written. Under no circumstances should sector A have been unwritten and sector B have been written—this is known as an *out-of-order write*, and it has the potential to corrupt Reliance Edge. If [RedOsBDevFlush\(\)](#) is implemented correctly, such that the data is truly flushed, out-of-order writes will not occur.

If [RedOsBDevWrite\(\)](#) is entirely synchronous (meaning there is no caching in either software or hardware), [RedOsBDevFlush\(\)](#) can do nothing and return success. Likewise if the storage medium is inherently an impermanent form of storage (like a RAM disk).

If using partitioning, a [RedOsBDevFlush\(\)](#) call issued for a single volume might need to flush a storage device shared by multiple volumes. While not ideal from a performance perspective, doing so should not cause problems.

RedOsBDevDiscard

[RedOsBDevDiscard\(\)](#) is an optional interface that is needed only if the file system is configured to issue discards. It is called to advise the block device of sectors that are no longer needed by the file system and may be treated by the driver as free space. This is not required, but is recommended on many storage devices to optimize performance and media lifespan. For guidance on whether you should enable discards and implement this interface, see [Discard Operations](#) chapter.

Discard support is only included in the commercial version of Reliance Edge. You can view commercial licensing options [here](#) or contact sales@tuxera.com for more info.

If you do enable discard support, you should implement [RedOsBDevDiscard\(\)](#) to forward the request to the underlying driver.

Regarding the sector numbers, `ullSectorStart` has a minimum value of the [Sector Offset](#) for the volume and `(ullSectorStart + ullSectorCount)` has a maximum value of the [Sector Offset](#) added to the [Sector Count](#). The `ullSectorCount` value is potentially very large: especially during format, when the whole storage medium is discarded. Some storage devices struggle with large discard requests, so your implementation might consider breaking a large request into smaller requests, if the underlying driver is not already doing that.

When [RedOsBDevDiscard\(\)](#) is called for [Automatic Discards](#) or during format, the return status is ignored—the success or failure of such a discard operation does not impact file system correctness; it is only an optimization. When [RedOsBDevDiscard\(\)](#) is called as part of [red_fstrim\(\)](#), the return status is propagated, so that the application is notified if the discards it explicitly requested are failing.

Note

Discard/trim should not be confused with "secure trim" (sometimes called "secure delete" or just "trim" as well), which actively erases and overwrites the given sectors instead of just marking them as free. [RedOsBDevDiscard\(\)](#) should *not* be implemented as a secure trim, since doing so may result in additional flash wear and performance degradation. Certain types of storage media (such as eMMC) support an alternative to secure trim where data is discarded (normal discards) followed by a "sanitize" command to eliminate data remanence in the discarded sectors; see [red_fstrim\(\)](#), which can be used to implement the first half of that use case.

3.3.2.2 Mutex (osmutex.c)

Only one task or thread is allowed to run Reliance Edge code at any given time; if there are multiple volumes, at most one volume is active at any given time. The mutex service implements a single synchronization object which is used to enforce this mutual exclusion. This service is normally implemented using a synchronization object provided by the RTOS, such as a semaphore or mutex. If necessary, it is also possible to implement a mutex in software, either in C or assembly, but doing so is beyond the scope of this chapter.

You need to implement this service if: multiple independent execution contexts (that is, tasks or threads) will be calling into Reliance Edge. In other words, if the task count is greater than one, you must implement the mutex service.

If you determine you do not need this service, the stubbed versions can be left as-is, since when the task count is one, the mutex code is not even compiled.

Four functions need to be implemented for the mutex service: [RedOsMutexInit\(\)](#), [RedOsMutexUninit\(\)](#), [RedOsMutexAcquire\(\)](#), and [RedOsMutexRelease\(\)](#). These are discussed below.

RedOsMutexInit

[RedOsMutexInit\(\)](#) is called when the Reliance Edge driver is initialized, and does whatever is necessary to create the mutex and prepare it to be used by [RedOsMutexAcquire\(\)](#) and [RedOsMutexRelease\(\)](#). The created mutex must be in the released state.

Typically the implementation of this function will allocate and initialize a semaphore or mutex provided by the RTOS, using the appropriate functions and types. Below is an example taken from the FreeRTOS port:

```
#include <FreeRTOS.h>
#include <semphr.h>

static SemaphoreHandle_t xMutex;

REDSTATUS RedOsMutexInit(void)
{
    REDSTATUS ret;

    xMutex = xSemaphoreCreateMutex();
    if(xMutex != NULL)
    {
        ret = 0;
    }
    else
    {
        ret = -RED_ENOMEM;
    }

    return ret;
}
```

RedOsMutexUninit

[RedOsMutexUninit\(\)](#) is called when the Reliance Edge driver is uninitialized, and does whatever is necessary to uninitialized the mutex and free associated resources. The mutex must be put into a state such that [RedOsMutexInit\(\)](#) can be called again. In some environments, it may be reasonable to stub this function (do nothing and return zero), and write [RedOsMutexInit\(\)](#) in such a way that the mutex is initialized once, and subsequent calls do nothing.

Typically the implementation of this function will simply undo the work done by [RedOsMutexInit\(\)](#), as in this example taken from the FreeRTOS port:

```
REDSTATUS RedOsMutexUninit(void)
{
    vSemaphoreDelete(xMutex);
    xMutex = NULL;

    return 0;
}
```

RedOsMutexAcquire

[RedOsMutexAcquire\(\)](#) is called by Reliance Edge APIs as their first step, in order to secure exclusive access to the Reliance Edge driver for the calling task.

Typically the implementation of this function will "acquire", "take", or "lock" the synchronization object created in [RedOsMutexInit\(\)](#).

Note that [RedOsMutexAcquire\(\)](#) returns void: no errors are allowed, so it should be implemented in such a way that errors are not expected. This should not pose a problem in most environments, where the mutex or semaphore is a highly reliable primitive.

There is also no time-out. If the underlying synchronization supports a time-out, a correct implementation will retry when the time-out expires. This is demonstrated in this example taken from the FreeRTOS port:

```
void RedOsMutexAcquire(void)
{
    while(xSemaphoreTake(xMutex, portMAX_DELAY) != pdTRUE)
    {
    }
}
```

RedOsMutexRelease

[RedOsMutexRelease\(\)](#) is called by Reliance Edge APIs as their last step, in order to relinquish exclusive access to the Reliance Edge driver and allow other tasks to use the file system.

Typically the implementation of this function will "release", "give", "put", or "unlock" the synchronization object created in [RedOsMutexInit\(\)](#).

Below is an example taken from the FreeRTOS port:

```
void RedOsMutexRelease(void)
{
    BaseType_t xSuccess;

    xSuccess = xSemaphoreGive(xMutex);
    REDASSERT(xSuccess == pdTRUE);
    (void)xSuccess;
}
```

3.3.2.3 Task ID (ostask.c)

The Task ID service provides an integer identifier which is nonzero and unique among all tasks or threads (that is, execution contexts) calling into Reliance Edge. It is used to store certain per-task information (like [red_errno](#)) in an identifiable way.

You need to implement this service if: You are using the POSIX-like API and the task count is greater than one.

RedOsTaskId

[RedOsTaskId\(\)](#) is called in order to retrieve a task ID, which is both nonzero and unique among all tasks. Typically the implementation of this function will call an RTOS function with a similar purpose.

If the RTOS already uses nonzero integer task IDs, this function is trivial to implement, as in the below example:

```
#include <vxWorks.h>
#include <taskLib.h>

uint32_t RedOsTaskId(void)
{
    int id = taskIdSelf();

    return (uint32_t)id;
}
```

If the RTOS uses pointers as task identifiers, the pointer should be cast to integer. If zero (or NULL) is a valid task ID for the RTOS, steps must be taken to avoid returning zero. The below example taken from the FreeRTOS port (and slightly abbreviated) demonstrates this:

```
#include <FreeRTOS.h>
#include <task.h>

uint32_t RedOsTaskId(void)
{
    uint32_t ulTaskPtr = (uint32_t)(uintptr_t)xTaskGetCurrentTaskHandle();

    // NULL is a valid task handle in FreeRTOS, so add one to all task IDs.
    //
    REDASSERT((ulTaskPtr + 1U) != 0U);
    return ulTaskPtr + 1U;
}
```

3.3.2.4 Real-Time Clock (osclock.c)

The Real-Time Clock (RTC) service provides a representation of the actual date and time. Reliance Edge makes limited use of the RTC. The atime, mtime, and ctime inode timestamps are populated with the timestamp provided by this service. In addition, an RTC timestamp is stored in the master block to indicate when the volume was formatted. However, many use cases will not use these timestamps, and most embedded hardware does not include a real-time clock.

You need to implement this service if: You are using the POSIX-like API and plan to use the atime, mtime, and ctime inode timestamps in your application. In order to do this, your target system must have hardware support for an RTC. If your hardware has an RTC, but your application will not use the inode timestamps, then there is limited value in implementing this service (it has some value as a debugging aid).

If you determine you do not need this service, the stubbed functions can be left as-is.

Three functions need to be implemented for the RTC service: [RedOsClockInit\(\)](#), [RedOsClockUninit\(\)](#), [RedOsClockGetTime\(\)](#). These are discussed below.

[RedOsClockInit](#)

[RedOsClockInit\(\)](#) is called when the Reliance Edge driver is initialized, and does whatever is necessary—if anything—to initialize the RTC and prepare [RedOsClockGetTime\(\)](#) to return the current date and time. In many cases, no initialization will be required, or the initialization will be done elsewhere, and this function can do nothing and return zero (success). However, the RTC on some embedded hardware will require explicit initialization logic (for example, by manipulating hardware registers in a prescribed way) and if this is not being done elsewhere, [RedOsClockInit\(\)](#) is the proper place to implement this logic.

[RedOsClockUninit](#)

[RedOsClockUninit\(\)](#) is called when the Reliance Edge driver is uninitialized, and can—if desired—uninitialize the RTC. If [RedOsClockInit\(\)](#) allocated any resources, they can be freed here. The key point is that after [RedOsClockUninit\(\)](#) is called, it must be safe to call [RedOsClockInit\(\)](#) again. In many cases the logic in [RedOsClockInit\(\)](#) will be safe to run repeatedly, and this function will do nothing and return success.

[RedOsClockGetTime](#)

[RedOsClockGetTime\(\)](#) is called to retrieve a timestamp that encodes the current date and time. The prescribed encoding is the number of seconds since January 1, 1970, excluding leap seconds—in other words, standard Unix time. If the resolution or epoch of the hardware RTC differ from this, the implementation will need to convert the time to the expected representation.

If your system includes a C library with a working implementation of `time()`, this function can be implemented easily since `time()` returns Unix time:

```
#include <time.h>

uint32_t RedOsClockGetTime(void)
{
    return (uint32_t)time();
}
```

Below is an example which converts DOS date/time (which is a bitfield with bits for the several values) to Unix time:

```
uint32_t RedOsClockGetTime(void)
{
    uint32_t ulTime = DOS_Date_Time();
    uint16_t uYear = (((uint16_t)(ulTime >> 25U)) & 0x7FU) + 1980U;
    uint8_t bMonth = (((uint8_t)(ulTime >> 21U)) & 0x0FU);
    uint8_t bDay = (((uint8_t)(ulTime >> 16U)) & 0x1FU);
    uint8_t bHour = (((uint8_t)(ulTime >> 11U)) & 0x1FU);
    uint8_t bMinute = (((uint8_t)(ulTime >> 5U)) & 0x3FU;
    uint8_t bSecond = (((uint8_t)(ulTime) & 0x1FU) * 2U;

    return SecondsSince(uYear, bMonth, bDay, bHour, bMinute, bSecond);
}
```

Here is a similar example, using the Atmel Software Framework RTC functions:

```
#include <rtc.h>
```

```

uint32_t RedOsClockGetTime(void)
{
    uint32_t ulHour;
    uint32_t ulMinute;
    uint32_t ulSecond;
    uint32_t ulYear;
    uint32_t ulMonth;
    uint32_t ulDay;

    rtc_get_time(RTC, &ulHour, &ulMinute, &ulSecond);
    rtc_get_date(RTC, &ulYear, &ulMonth, &ulDay, NULL);

    return SecondsSince((uint16_t)ulYear, (uint8_t)ulMonth,
        (uint8_t)ulDay, (uint8_t)ulHour, (uint16_t)ulMinute,
        (uint16_t)ulSecond);
}

```

Here is the SecondsSince() function used by both examples, which you are free to use in your implementation if it is helpful:

```

// Leap years occur every 4 years, but not at 100 year boundaries,
// unless it is a 400 year boundary. Therefore, the year 2000 is
// a leap year, but the year 2100 is not.
//
#define ISLEAPYEAR(y) \
    (((y) % 4U) == 0U) && (((y) % 400U) == 0U) || (((y) % 100U) != 0U))

#define LEAPDAYS_BEFORE_YEAR(y) \
    (((y) / 4U) - ((y) / 100U)) + ((y) / 400U))

#define BASELINE_YEAR 1970U
#define BASELINE_LEAPDAYS LEAPDAYS_BEFORE_YEAR(BASELINE_YEAR)

#define DAYS_PER_YEAR 365U
#define DAYS_PER_PRESIDENT ((DAYS_PER_YEAR * 4U) + 1U)
#define DAYS_PER_CENTURY ((DAYS_PER_PRESIDENT * 25U) - 1U)
#define DAYS_PER_QUADCENTURY ((DAYS_PER_CENTURY * 4U) + 1U)

#define DAYS_BEFORE_BASELINE_YEAR \
    ((BASELINE_YEAR-1U) * DAYS_PER_YEAR) + BASELINE_LEAPDAYS

static uint32_t SecondsSince(
    uint16_t uYear,
    uint8_t bMonth,
    uint8_t bDay,
    uint8_t bHour,
    uint8_t bMinute,
    uint8_t bSecond)
{
    static const uint8_t abDaysPerMonth[] =
        {31U, 28U, 31U, 30U, 31U, 30U, 31U, 30U, 31U, 30U, 31U};
    uint32_t ulTotalDays = 0U;
    uint32_t uWholeYears;
    uint16_t bIndex;

    uWholeYears = uYear - 1U; // Current year not complete.

    if(uWholeYears >= 400U)
    {
        ulTotalDays += ((uWholeYears / 400U) * DAYS_PER_QUADCENTURY);
        uWholeYears %= 400U;
    }

    if(uWholeYears >= 100U)
    {
        ulTotalDays += ((uWholeYears / 100U) * DAYS_PER_CENTURY);
        uWholeYears %= 100U;
    }

    if(uWholeYears >= 4U)
    {
        ulTotalDays += (uWholeYears / 4U) * DAYS_PER_PRESIDENT;
        uWholeYears %= 4U;
    }

    if(uWholeYears >= 1U)
    {
        ulTotalDays += uWholeYears * DAYS_PER_YEAR;
    }
}

```

```

}

ulTotalDays -= DAYS_BEFORE_BASELINE_YEAR;

for(bIndex = 0U; bIndex < (bMonth - 1U); bIndex++)
{
    ulTotalDays += abDaysPerMonth[bIndex];

    // For February, if it is a leap year, set 29 days
    //
    if((bIndex == 1U) && ISLEAPYEAR(uYear))
    {
        ulTotalDays++;
    }
}

ulTotalDays += bDay - 1U;

// Determine the number of seconds up to this day
//
ulTotalSeconds = ulTotalDays * 24U * 60U * 60U;

// Add the number of seconds for this day
//
ulTotalSeconds += ((uint32_t)bHour * 60U * 60U);
ulTotalSeconds += ((uint32_t)bMinute * 60U);
ulTotalSeconds += bSecond;

return ulTotalSeconds;
}

```

A note on the "year 2038 problem". Traditionally, Unix time has been stored in a signed 32-bit integer, which will overflow in 2038 (that is, the number of seconds since 1970 will exceed INT32_MAX in 2038). [RedOsClockGetTime\(\)](#) uses an unsigned 32-bit integer instead, which is good until the year 2106. However, if your implementation of [RedOsClockGetTime\(\)](#) uses time(), and time() returns a signed 32-bit time_t, then your implementation may be vulnerable to year 2038 issues.

3.3.2.5 Text Output (osoutput.c)

The text output service writes a null-terminated string to a serial port, console, terminal, or other display device, such that the string is visible to the user. This is used by tests to display results, and by the driver to print a sign-on message and error notifications.

You need to implement this service if: You will be running any of the tests provided with Reliance Edge on your target hardware; or if you want to see the sign-on message and error notifications outputted by the driver.

If not implementing this service, the stubbed version can be left as-is.

[RedOsOutputString](#)

[RedOsOutputString\(\)](#) is called in order to output a null-terminated string.

If the standard C library is available, it may be possible to output the string with printf():

```
#include <stdio.h>

void RedOsOutputString(
    const char *pszString)
{
    printf("%s", pszString);
}
```

In other environments, the implementation may need to use custom output routines.

The inputted string will represent newlines with a single '\n' character. In some environments, a serial terminal will only display newlines correctly when they are followed by a subsequent carriage return character ('\r'). These characters can be added in this function, as in the following example:

```
#include <stdio.h>

void RedOsOutputString(
    const char *pszString)
{
    uint32_t     ulIdx = 0U;

    while (pszString[ulIdx] != '\0')
    {
        putchar(pszString[ulIdx]);

        if (pszString[ulIdx] == '\n')
        {
            putchar('\r');
        }

        ulIdx++;
    }
}
```

3.3.2.6 Assertion Handling (osassert.c)

The assertion handling service determines what happens when one of the Reliance Edge assertions fails. When assertions are disabled the assertion handler is not used.

You need to implement this service if: You are enabling assertions, whether for testing, debugging, or as a extra set of checks in production.

RedOsAssertFail

[RedOsAssertFail\(\)](#) is called when an assertion fails. The file name and line number are passed in, and it is recommended to print this information if possible to aid in debugging. Beyond that, the handling of assertion failures is somewhat of a philosophical matter. You can return from [RedOsAssertFail\(\)](#), and Reliance Edge will attempt to execute the statement after the assertion: in some cases this will fail cleanly, but in other cases the code after the failing assertion will fault or fail in more subtle ways. The alternative to returning is to enter an infinite loop, which has the advantage that the assertion is more obvious, instead of (for example) being a single line of output in a long test log. If you decide to leave asserts enabled in production code, the more conservative option would be to return from [RedOsAssertFail\(\)](#), since that is essentially what would have happened if assertions were disabled.

In some environments, it may be possible to invoke more advanced fault handling from the assertion handler: for example, in a system with task and memory partitioning, it may be possible to restart the file system and the tasks using it, getting things back to normal without human intervention.

3.3.2.7 Timestamp (ostimestamp.c)

The timestamp service—not to be confused with the [real-time clock](#) service—provides timestamps that can be used to accurately and precisely measure the passage of time. These timestamps are used by Reliance Edge tests, and a good implementation is essential for the performance tests (like FSIOTest).

You need to implement this service if: You will be running Reliance Edge tests which use the timestamp service. These tests include FSIOTest, the POSIX-like API Test, the FSE API Test, the Disk Full Test, and BDevTest.

Timestamp resolution is the minimum time elapsed between two adjacent timestamps. The resolution supported by the timestamp service will depend on the implementation and what is possible on the target system. [RedOsTimePassed\(\)](#) reports the passage of time in microseconds, so a one microsecond resolution is the best supported resolution; most environments will use a lower resolution. A timer resolution of one millisecond is sufficient for most testing purposes, although a less accurate timer is better than nothing—crude performance results are possible even with a one second resolution timer.

There are a number of ways the timestamp service could be implemented. Some RTOSes have a tick count with a known frequency that can be used to measure the passage of time. If this tick count does not exist, it may be possible

to create one using interrupts or RTOS timer functions. If the system has an RTC, it may be possible to use that to implement the timestamp service, although usually the RTC is of low resolution.

The code in os/freertos/ostimestamp.c is an example which uses the RTOS tick count. It uses a FreeRTOS function to retrieve the current tick count, which is used for the timestamps, and uses a FreeRTOS macro expressing the tick count interval to convert ticks to time passed.

If you determine you do not need this service, the stubbed functions can be left as-is.

Four functions need to be implemented for the timestamp service: [RedOsTimestampInit\(\)](#), [RedOsTimestampUninit\(\)](#), [RedOsTimestamp\(\)](#), and [RedOsTimePassed\(\)](#). These are discussed below.

[RedOsTimestampInit](#)

[RedOsTimestampInit\(\)](#) is called at the start of a test which uses timestamps. The function must do whatever is necessary such that [RedOsTimestamp\(\)](#) can return a valid timestamp and [RedOsTimePassed\(\)](#) can determine how much time has passed since a given timestamp. In many cases, the implementation can simply do nothing and return success (zero). For example, the FreeRTOS implementation uses the FreeRTOS tick count and requires no initialization. However, if the tick count is being maintained manually, setup is probably required.

For example, here is an example that uses the Atmel Software Framework to setup interrupts for the Atmel SAM4E-EK board to maintain a value representing the number of milliseconds that have passed, supporting resolutions as high as one millisecond:

```
#include <asf.h>
#include <genclk.h>
#include <conf_board.h>

// Resolution of the timer.
//
#define TIMER_RESOLUTION_MS 1U

#if (1000U % TIMER_RESOLUTION_MS) != 0U
    #error "1000U % TIMER_RESOLUTION_MS != 0U"
#endif

static volatile uint32_t ulMilliseconds = 0;

REDSTATUS RedOsTimestampInit(void)
{
    struct genclk_config     conf;
    REDSTATUS                 ret;

    if(SysTick_Config(SystemCoreClock / (1000U / TIMER_RESOLUTION_MS)))
    {
        ret = -RED_ENOSYS;
    }
    else
    {
        // Enable PIO module related clock.
        //
        sysclk_enable_peripheral_clock(PIN_PUSHBUTTON_1_ID);

        // Configure specific CLKOUT pin.
        //
        ioport_set_pin_mode(PIO_PA6_IDX, IOPORT_MODE_MUX_B);
        ioport_disable_pin(PIO_PA6_IDX);

        // Configure the output clock source and frequency.
        //
        genclk_config_defaults(&conf, GENCLK_PCK_0);
        genclk_config_set_source(&conf, GENCLK_PCK_SRC_PLLACK);
        genclk_config_set_divider(&conf, GENCLK_PCK_PRES_1);
        genclk_enable(&conf, GENCLK_PCK_0);

        ret = 0;
    }

    return ret;
}

void SysTick_Handler(void)
```

```
    ulMilliseconds += TIMER_RESOLUTION_MS;
}
```

RedOsTimestampUninit

[RedOsTimestampUninit\(\)](#) is called at the conclusion of a test which uses timestamps. If [RedOsTimestampInit\(\)](#) has allocated resources, they should be freed here. [RedOsTimestampUninit\(\)](#) must leave things in such a state that it is safe to call [RedOsTimestampInit\(\)](#) again. In many cases the implementation of [RedOsTimestampUninit\(\)](#) can do nothing and return success, even if [RedOsTimestampInit\(\)](#) does setup. For example, the above Atmel interrupt example is safe to run repeatedly, so [RedOsTimestampUninit\(\)](#) could elect to do nothing and let the millisecond counter continue to increment.

RedOsTimestamp

[RedOsTimestamp\(\)](#) is called to retrieve a timestamp. For example, if a performance test is about to start an operation, it will call [RedOsTimestamp\(\)](#) to retrieve and save a timestamp right before the operation starts, and once the operation completes, pass the saved timestamp into [RedOsTimePassed\(\)](#) to determine how long the operation took.

The return value of [RedOsTimestamp\(\)](#) is implementation defined. The return type REDTIMESTAMP is defined in the os/OSNAME/include/redostypes.h header; you can set REDTIMESTAMP to any type which is convenient for the implementation that can be legally returned. In most cases REDTIMESTAMP will be an integer of some sort, but using a pointer or structure type is also allowed. All code which uses the timestamp service uses REDTIMESTAMP as an opaque type, with no assumptions about the underlying type or the encoding or representation of the timestamp.

In most implementations, [RedOsTimestamp\(\)](#) will return a tick count, and the REDTIMESTAMP typedef should be updated to an appropriate integer type for that tick count.

RedOsTimePassed

[RedOsTimePassed\(\)](#) is called to determine how much time has passed since the provided timestamp was returned from [RedOsTimestamp\(\)](#).

The resolution of the return value is microseconds (one millionth of a second). If the implementation has a lower resolution (for example, milliseconds, 10 millisecond ticks, et cetera) or a higher resolution (for example, nanoseconds) that result must be multiplied or divided to match the expected return value. If the implementation fails to use the correct units—for example, reporting the number of milliseconds that have passed, rather than microseconds—performance testing will produce grievously incorrect results.

Below is an example from the FreeRTOS implementation, which uses a FreeRTOS macro (configTICK_RATE_HZ) to determine the number of microseconds per tick, and multiplies the number of elapsed ticks by that value:

```
#define MICROSECS_PER_TICK (1000000U / configTICK_RATE_HZ)

uint64_t RedOsTimePassed(
    REDTIMESTAMP tsSince)
{
    // This works even if the tick count has wrapped around, provided it
    // has only wrapped around once.
    //
    uint32_t ulTicksPassed = (uint32_t)xTaskGetTickCount() - tsSince;
    uint64_t ullMicrosecs = (uint64_t)ulTicksPassed * MICROSECS_PER_TICK;

    return ullMicrosecs;
}
```


Chapter 4

FreeRTOS Integration

Reliance Edge has a FreeRTOS port—which should also work for OpenRTOS and SafeRTOS, both FreeRTOS derivatives—which provides those parts of the port which can be provided generically. So while FreeRTOS users do not need to do everything described in the [Porting Guide](#), some porting work is still required.

4.1 Assumptions

The Reliance Edge FreeRTOS port assumes that Reliance Edge API functions (including the initialization functions, `red_init()` and `RedFseInit()`) are being called from a FreeRTOS task (that is, tasks created with `xTaskCreate()`). Thus it is also assumed that the FreeRTOS task scheduler is active (that is, `vTaskStartScheduler()` has been called).

4.2 FreeRTOS Configuration

The FreeRTOS configuration value `INCLUDE_xTaskGetCurrentTaskHandle` should be set to 1 if Reliance Edge task count is configured to be two or more.

4.3 OS Services

See [Introduction to the OS Services](#) if you are unfamiliar with the OS services.

The following services have been implemented for FreeRTOS and will not normally need to be modified:

- [Mutex](#) service.
- [Task ID](#) service.
- [Timestamp](#) service.

The following services have also been implemented and will work for most FreeRTOS users:

- [Text output](#) service. Assumes `stdio.h` and `putchar` are available. If this is not true, follow the link to determine whether you need this service and for tips on implementing it.
- [Assertion handler](#) service. Assumes `stdio.h` and `printf` are available. Also enters an infinite loop if an assertion fails. If these assumptions are not true, or the infinite loop is undesirable, follow the link to determine whether you need this service and for tips on implementing it.

The following services have not been implemented:

- [Block device](#) service. Implementing this service is required. The next section goes into more detail.
- [Real-time clock](#) service. Reliance Edge does not depend on this service, so implementing it is optional. It requires a real-time clock, which many systems lack. Follow the link for more details.

4.3.1 Implementing Block Device Service for FreeRTOS

Implementing the block device service for FreeRTOS is much the same as implementing it for any system. The main difference is that os/freertos/services/osbdev.c includes several example implementations, and some users might find that one of these works as-is for their system.

If none of the example implementations are useful, you can copy the contents of [os/stub/services/osbdev.c](#) into os/freertos/services/osbdev.c to start from scratch, following the instructions in the [block device](#) service section.

4.3.1.1 Implementations

The FreeRTOS implementation in os/freertos/services/osbdev.c includes seven example implementations of the block device service. The `BDEV_EXAMPLE_IMPLEMENTATION` macro near the start of the file determines which of the examples is compiled and used: it can be set to one of six values (`BDEV_CUSTOM`, `BDEV_FLASHFX`, `BDEV_FATFS`, `BDEV_ATMEL_SDMMC`, `BDEV_STM32_SDIO` or `BDEV_RAM_DISK`) for the different examples, described in detail below. To ease comprehension, each example is implemented in a separate header, and when that example is enabled, it is included via the preprocessor into [osbdev.c](#).

Custom implementation

This is the default value. This is actually not an example implementation: all of its functions are stubbed. It includes an `#error` directive to make it obvious that changes to [osbdev.c](#) are required. If you wish, you can use this as a starting point for your own custom implementation: alternatively, as suggested above, you can copy [os/stub/services/osbdev.c](#) to get the stubs without the other examples.

FlashFX Tera example implementation

This implementation uses Tuxera's FlashFX Tera driver to use raw flash storage with Reliance Edge.

This option is only available in commercial releases of Reliance Edge.

FatFs example implementation

This implementation is designed to reuse an existing block device driver that was written for FatFs (which is an open-source implementation of FAT). If you have such a driver, it can be linked in and used immediately. The FatFs `diskio.h` header must be in the include directory path.

Atmel Software Framework SD/MMC driver example implementation

This implementation uses a modified version of the open source SD/MMC driver included in the Atmel Software Framework (ASF) and will work as-is for many varieties of Atmel hardware. This example assumes relatively minor modifications to the ASF SD/MMC driver to make it support multi-sector read and write requests, which greatly improves performance. The modified driver is distributed with the Reliance Edge commercial kit and is included in FreeRTOS Atmel projects that come with the commercial kit (such as in `projects/freertos/atmel/sam4e-ek/src/ASF`).

This example can easily be altered to work with an unmodified version of the ASF SD/MMC driver. Simply replace `sd_mmc_mem_2_ram_multi()` and `sd_mmc_ram_2_mem_multi()` with `sd_mmc_mem_2_ram()` and

`sd_mmc_ram_2_mem()` respectively, and add a for loop to loop over each sector in the request. However, as described in the [block device](#) service section, there are considerable performance advantages to issuing real multi-sector requests, so using the modified driver is recommended.

ST Microelectronics STM32 SDIO driver example implementation

This implementation interfaces with the STM32 BSP (Board Support Package) utilities to access the SD card as a block device. The BSP utilities are implemented as an abstraction layer over the STM32 HAL drivers, and they are distributed with the STM32Cube software packages. Note that the older STM32 StdPeriph drivers are not supported at this time.

This Reliance Edge implementation will work as-is with the BSP utilities provided for the STM324xG-EVAL and the STM32F746NG-Discovery boards. Other boards supported by STM32Cube may also be compatible if the correct header files are included in `os/freertos/services/osbdev.c`, but have not been tested by Tuxera.

The STM32 SDIO drivers require buffers to be aligned on a 4-byte boundary when using DMA (Direct Memory Access) for sector transfers. In order to meet this requirement, Reliance Edge statically allocates an aligned buffer and copies non-aligned data through this buffer. This increases the RAM requirement of Reliance Edge by 512 bytes.

The STM32 drivers do not support using multiple SD cards simultaneously, so this implementation only supports one Reliance Edge volume.

RAM disk example implementation

This implementation uses a RAM disk. It will allow you to compile and test Reliance Edge even if your storage driver is not yet ready. On typical target hardware, the amount of spare RAM will be limited so generally only very small disks will be available.

4.4 Building

The considerations in the [Building](#) section generally apply. To compile the Reliance Edge FreeRTOS port, the FreeRTOS/Source/include directory must be one of the include directories. Reliance Edge must eventually link against FreeRTOS: this can be accomplished by compiling FreeRTOS and Reliance Edge at the same time, or by compiling Reliance Edge as a library and linking it with FreeRTOS later.

4.5 Memory Allocation in FreeRTOS Version 9

In compliance with MISRA C:2012 directive 4.12, Reliance Edge does not internally allocate memory dynamically (unless the RAM disk example block device service implementation is used). Starting with version 9, FreeRTOS may also be configured to avoid all dynamic memory allocation. Functions like `xTaskCreate()` or `xMutexCreate()` that do allocate memory from the heap may now be substituted for functions such as `xMutexCreateStatic()`, which allow a statically allocated segment of memory to be provided instead. The Reliance Edge mutex service implementation uses one of these mutex create functions, depending on the FreeRTOS configuration. To ensure the filesystem does use any methods that dynamically allocate memory, set `configSUPPORT_STATIC_ALLOCATION` to 1.

Chapter 5

INTEGRITY Integration

Reliance Edge has been ported to the INTEGRITY RTOS from Green Hills Software. This port includes integration into the INTEGRITY file system layer, which means that normal INTEGRITY system calls like `mount()`, `open()`, `unlink()`, `read()`, `write()`, etc., can be used to access the Reliance Edge file system.

The INTEGRITY port is only available in the commercial kit.

5.1 Reliance Edge: unified or client-server

Reliance Edge for INTEGRITY comes in two configurations: unified and client–server. The unified configuration is a single library that contains the entirety of the file system. The client–server configuration has two libraries: a client library to be linked into every AddressSpace using Reliance Edge, and a server library to be linked into an AddressSpace dedicated to hosting the Reliance Edge server. From an application perspective, both configurations are functionally identical. The choice between unified or client–server should be guided by the use case and priorities of the system.

The client–server configuration is required if Reliance Edge is to be used by more than one AddressSpace. If only one AddressSpace needs to use Reliance Edge, the client–server configuration can still be used, with the advantage that the application and file system are protected from one another by being in separate AddressSpaces. For example, if the application and file system share an AddressSpace (as with the unified library), it is theoretically possible for the application to write to a rogue pointer and thereby corrupt file system memory; and if the memory is corrupted in certain ways, this may lead to data loss or file system corruption—whereas that class of error is not possible with the client–server configuration.

The unified configuration can only be used from a single AddressSpace. If that limitation is ignored, and the unified library is linked into more than one AddressSpace, the result will be data loss and file system corruption. However, if one AddressSpace is enough, the unified configuration offers faster performance (due to the lack of client–server communication overhead) and lower resource requirements (both RAM and code space). The unified configuration also has fewer moving parts than the client–server configuration, and this simplicity might be preferred for reliability reasons.

5.2 Reliance Edge configuration on INTEGRITY

Reliance Edge on INTEGRITY is mostly configured the same way as it is in other environments; see the [Configuration Utility](#) and [Product Configuration](#) chapters. However, for INTEGRITY, there are the following restrictions upon the configuration:

- Ensure "Use POSIX API" is selected (which sets `REDCONF_API_POSIX` to 1); this is the default setting. Only the [POSIX-like API](#) provides the necessary functionality to integrate into the INTEGRITY file system layer. While

using the [File System Essentials API](#) on INTEGRITY is technically possible, that use-case is not covered in this documentation.

- Ensure the path separator character is / (the forward slash), which is also the default. INTEGRITY uses forward-slashes in its paths, so to fit in with the rest of the system, Reliance Edge should do the same.
- Ensure the "Maximum file name length" ([REDCONF_NAME_MAX](#)) is not greater than 255; the default value, 12, is well under this limit. Names longer than 255 bytes are not supported by INTEGRITY for certain system calls, like `readdir()`. For now, we disallow names longer than this system limit. This restriction should be acceptable; even in other environments, where a maximum name length longer than 255 bytes is allowed, it is not recommended for space efficiency reasons (see the [Maximum Name Length](#) section).
- The volume path prefixes must be the same as the INTEGRITY mount point for each volume. The default volume path prefixes used by the Configuration Utility (for example, "VOL0:", "VOL1:", "VOL2:", etc.) are not suitable for this purpose. According to Green Hills documentation on implementing a file system, for a file system integrating into INTEGRITY as Reliance Edge does, the mount points must be named in a quasi-directory style with two leading path separators, similar to the `POSIX_EXT` mount style described in the PJFS documentation. Example volume path prefixes conforming to this recommendation are "//mnt", "//sdcard", "//data", "//data/bak", etc. If `mount()` is provided a mount point which does not match any volume path prefix, `mount()` will fail with an `ENOENT` error. If `mount()` is provided a mount point which does not start with "//", `mount()` will fail with an `EINVAL` error.

The above restrictions will not be enforced by the Configuration Utility. Some are enforced at compile-time with #error directives.

5.2.1 Configuration considerations

When configuring the [Handle Count](#), be aware that opening a directory with `open()` or `opendir()` will use two of Reliance Edge's internal handles. Thus, for your application to have ten directories open simultaneously, there must be at least twenty handles.

If using the client–server configuration of Reliance Edge, then the [Task Count](#) must be large enough for all tasks in all AddressSpaces. Likewise, the [Handle Count](#) must be large enough for all tasks in all AddressSpaces, as the handles are a shared resource managed by the server.

5.2.2 Additional configuration macros

There are additional macros, particular to the INTEGRITY port, which exist outside of the standard configuration header, that may be modified for additional configuration.

The following macros, both defined in `os/integrity/driver/server/redfs_server.h`, are specific to the client–server configuration of Reliance Edge for INTEGRITY:

- `REDFS_ADDRESSSPACE_COUNT`. This is the number of client AddressSpaces which will be connecting to the server. The default value for this macro is the [Task Count](#), but that represents the worst-case; if multiple file system tasks run in the same AddressSpace, then the true AddressSpace count is lower. Lowering this value will reduce the amount of RAM needed by the server.
- `REDFS_RECEIVE_BUFFER_SIZE`. The size in bytes of the buffer that is statically allocated for each client AddressSpace; it represents the maximum transfer size between the client and server. The default size is 128KB. To reduce the amount of RAM needed by the server, the buffer size can be decreased, as far down as the [Block Size](#), although this may decrease performance.

The following macros, all defined in `os/integrity/driver/server/server.c`, are specific to the client–server configuration of Reliance Edge for INTEGRITY:

- REDFS_SERVER_PRIORITY. Task priority for the main server task which listens for client requests; the default is 150. The server's exception monitoring task will use a task priority of REDFS_SERVER_PRIORITY + 1.
- REDFS_SERVER_STACK_SIZE. Stack size for the main server task. Also the stack size for the exception monitoring task. The default is 4096.
- REDFS_SERVER_TASK_NAME. Task name for the main server task; the default is "RedfsServerTask".
- REDFS_SERVER_EM_TASK_NAME. Task name for the exception monitoring task; the default is "RedfsServerExceptionM

The following macros, all defined in os/integrity/services/osbdev_sdrv.c, are applicable to all configurations of Reliance Edge when using INTEGRITY v11.7 or later:

- REDFS_SDRV_BOOUNCE_BUFFER_SIZE: The size in bytes of the bounce buffer used for aligning read/write requests that were made with an unaligned buffer. The default size is 128KB. It can be decreased to reduce RAM usage, with a minimum value of the sector size; smaller values may hurt performance if large I/O operations are using unaligned buffers. If using the client-server configuration, using the same value as the REDFS_RECEIVE_BUFFER_SIZE is optimal.
- REDFS_SDRV_BOOUNCE_BUFFER_ALIGNMENT: The minimum required buffer alignment for block device I/O (reads and writes). If the buffer handed to osbdev_sdrv.c does not have this alignment, the bounce buffer will be used, imposing an extra `memcpy` operation. Alignment is often required with storage drivers using DMA. Typical values might be 4 or 8 (processor word size aligned), 512 (sector aligned), or 0 (no alignment needed). The default value is MAXIMUM_CACHE_LINE_SIZE.
- REDFS_SDRV_TASK_STACK: Stack size in bytes for the storage I/O task; the default is 4096.
- REDFS_SDRV_TASK_PRIORITY: Task priority for the storage I/O task; the default is 152. **If you are using the unified configuration, this must be at least two higher than the priority of any task which accesses Reliance Edge.**
- REDFS_SDRV_DRIVER_PRIORITY: Priority for the driver notifier job; the default is 10.

If desired, the values of the above macros can be updated from within the MULTI project, rather than by editing the source file. Right-click on a project component (such as the server's AddressSpace), select "Set Build Options...", pick the Preprocessor category, right-click on "Define Preprocessor Symbol" and select "Set Define Preprocessor Symbol...", and from there add entries in the form of MACRO_NAME=value.

5.3 Projects

5.3.1 Example projects

The following example projects can be found in the projects/integrity directory:

5.3.2 INTEGRITY ARM Simulator (simarm)

This project is for using Reliance Edge from the INTEGRITY Shell on the INTEGRITY ARM Simulator. It is setup to use the client-server configuration, but can be updated to use the unified configuration, as described in the project README. To use this project, you must have an INTEGRITY installation with the necessary source to build the mount library (see [Building for the INTEGRITY Shell](#)). This project is a good way to test and play with Reliance Edge on INTEGRITY in a dynamic fashion, without any hardware setup. See the README.txt file in that project's directory for further details.

This project has been tested with INTEGRITY v11.7.x.

5.3.3 Renesas R-Car H3 Starter Kit (rcar-h3sk)

This project is for using Reliance Edge from the INTEGRITY Shell on the Renesas R-Car H3 Starter Kit. It is setup to use the client–server configuration, but can be updated to use the unified configuration, as described in the project README. To use this project, you must have an INTEGRITY installation which includes the R-Car patches, along with the necessary source to build the mount library (see [Building for the INTEGRITY Shell](#)). See the README.txt file in that project's directory for further details.

This project has been tested with INTEGRITY v11.7.x.

5.3.4 BeagleBone Black Application, Unified (bbb-app-unified)

This project contains an example application which uses Reliance Edge via the unified library. It runs on a BeagleBone Black; to use it, you must have an INTEGRITY installation which includes the patches for the BeagleBone Black BSP. See the README.docx file in that project's directory for further details.

This project has been tested with INTEGRITY v11.4.x.

5.3.5 BeagleBone Black Application, Client--Server (bbb-app-clientserv)

This project contains example applications which use Reliance Edge via the client–server libraries. There are three client AddressSpaces, with a fourth AddressSpace for the server. The project runs on a BeagleBone Black; to use it, you must have an INTEGRITY installation which includes the patches for the BeagleBone Black BSP. See the README.docx file in that project's directory for further details.

This project has been tested with INTEGRITY v11.4.x.

5.3.6 BeagleBone Black INTEGRITY Shell (bbb-shell)

This project is for using Reliance Edge from the INTEGRITY Shell on a BeagleBone Black. It is setup to use the client–server configuration, but can easily be updated to use the unified configuration, as described in the project README. To use this project, you must have an INTEGRITY installation which includes the patches for the BeagleBone Black BSP, along with the necessary source to build the mount library (see [Building for the INTEGRITY Shell](#)). See the README.docx file in that project's directory for further details.

This project has been tested with INTEGRITY v11.4.x.

5.3.7 Updating an existing MULTI project

The [Building](#) section describes how an existing MULTI project can be updated to include the Reliance Edge libraries.

5.4 Building

Reliance Edge for INTEGRITY can be built with MULTI, the Green Hills compiler and IDE. The os/integrity/libraries directory contains several library projects designed to make it very easy to build Reliance Edge from MULTI; these are described in further detail in the following subsections.

5.4.1 Building the file system

5.4.1.1 Unified configuration

`os/integrity/libraries/libredfs_unified.gpj` will build the Reliance Edge unified library, `libredfs_unified.a`. To add it to your MULTI project, right-click on the Virtual AddressSpace for your application and select "Add File into XXXXX.gpj..." and find `libredfs_unified.gpj`. That Virtual AddressSpace will automatically be linked with `libredfs_unified.a`.

Remember that the unified library must only be used from a single AddressSpace. If you need to use the file system from multiple AddressSpaces, you should be using the client-server configuration.

5.4.1.2 Client-server configuration

`os/integrity/libraries/libredfs.gpj` will build the Reliance Edge client library, `libredfs.a`. To add it to your MULTI project, right-click on a Virtual AddressSpace for an application that will use the file system and select "Add File into XXXXX.gpj..." and find `libredfs.gpj`. That Virtual AddressSpace will automatically be linked with `libredfs.a`. Repeat as desired for each AddressSpace that will be using Reliance Edge.

`os/integrity/libraries/libredfsserver.gpj` will build the Reliance Edge server library, `libredfsserver.a`. To add it to your MULTI project, create a new Virtual AddressSpace to host the file system server, and delete the C file created by MULTI for `main()`, since the server library already has a `main()`. Right-click on the server Virtual AddressSpace and select "Add File into RelianceEdgeServer.gpj..." (or whatever you named the server) and find `libredfsserver.gpj`. The Virtual AddressSpace will be automatically linked with `libredfsserver.a`.

5.4.1.3 Linking with storage libraries

The Virtual AddressSpace to which `libredfs_unified.gpj` or `libredfsserver.gpj` was added may need to be modified to link with additional libraries. Assuming INTEGRITY v11.7 or later, the AddressSpace must link with `libstorage-core.a` and `libstorage-driver-pseudo.a`. To do this, right-click on the Virtual AddressSpace, select "Set Build Options...", pick the Project category, right-click on Libraries and select "Set Libraries...", and add `storage-core` and `storage-driver-pseudo` (note that the `lib*.a` part of the name is implicit).

Depending on the target, additional libraries may be required. For example, in the [Renesas R-Car H3 Starter Kit \(rcar-h3sk\)](#) project, which uses the client-server configuration, the Reliance Edge server also links with:

- `libvirt_iodevice_interface.a`
- `libstorage-driver-sdio-mmc.a`
- `libsdh_i_mmc_rcar_gen3.a`

5.4.1.4 Heap size adjustments

The default heap size for a Virtual AddressSpace is often relatively small, such as `0x20000`. This might be insufficient for Reliance Edge and the storage drivers it uses. To increase it, double-click on `integrity.int`, find the virtual AddressSpace where `libredfs_unified.gpj` or `libredfsserver.gpj` resides, right-click it (in an empty area, *not* the "Task Initial") and select Edit, go to the Attributes tab, and change the Heap Size field. The default is usually specified in hexadecimal bytes, but a decimal value with a postfix (e.g., "16M") also works.

5.4.1.5 Configuration files

The `redconf.h` and `redconf.c` files generated by the [Configuration Utility](#) should be saved into the MULTI project directory. If you want a starting point, open these files from one of the [example project directories](#) and save-as into your MULTI project directory.

Additionally, [redtypes.h](#) should also be in the MULTI project directory. This can be copied from one of the [example project directories](#). [redtypes.h](#) should not need any customization.

5.4.2 Building the tools for the target

If you want to run the formatter (newfs) and checker (fsck) tools on the target, you need to build their libraries. Add `os/integrity/libraries/libnewfs_redfs.gpj` and/or `os/integrity/libraries/libfsck_redfs.gpj` into your MULTI project, in the same fashion that `libredfs.gpj` or `libredfs_unified.gpj` was added (see above). An AddressSpace can only use the tools libraries if it links against either `libredfs.a` or `libredfs_unified.a`.

These tools can also be run from the host machine; see the [Command-Line Host Tools](#) chapter.

Note

`libfsck_redfs.gpj` requires that `REDCONF_CHECKER` is set to 1 in `redconf.h`. The [Configuration Utility](#) will always set `REDCONF_CHECKER` to 0, so if you want to use `libfsck_redfs.gpj`, you must edit `redconf.h` with a text editor to update `REDCONF_CHECKER`.

5.4.3 Building the tools for the host

To build [Command-Line Host Tools](#), copy the `host/` subdirectory from one of the [example project directories](#) into your MULTI project directory (which should already contain `redconf.h` and `redconf.c`). Open `host/Makefile` and edit `P_BASEDIR` to point at the Reliance Edge source and `P_PROJDIR` to point at the directory containing the Makefile. The host tools are built separately outside of MULTI (using Make and either Visual Studio or GCC), as detailed in [Building the Host Tools](#).

5.4.4 Building the tests

The following test libraries can be found in `os/integrity/libraries`:

- `libbdevtest.gpj`
- `libfsiotest.gpj`
- `libporttest.gpj`
- `libmultifstest.gpj`

These can be added to your MULTI project in the same fashion that `libredfs.gpj` or `libredfs_unified.gpj` was added (see above). Doing so allows you to invoke the tests from your application. See the [Testing](#) chapter for details on the tests. An AddressSpace can only use the tests libraries if it links against either `libredfs.a` or `libredfs_unified.a`.

5.4.5 Building for the INTEGRITY Shell

`os/integrity/libraries/libmount_redfs.gpj` is for INTEGRITY Shell integration; it builds the mount library, needed to mount Reliance Edge with the Shell's mount command. This library depends upon INTEGRITY source files not provided with the base installation; it needs either the FFS/MSDOSFS source or the "GHS support files for Reliance Edge" patch, available from Green Hills.

To use the mount library, right-click on `net_server_module.gpj` (the OS Module for the INTEGRITY Shell), select "Add File into `net_server_module.gpj...`", and find `libmount_redfs.gpj`. The main Reliance Edge library (either `libredfs.a` or `libredfs_unified.a`) should also be added into `net_server_module.gpj`. Once the mount command works, the other file system commands in the INTEGRITY Shell should work also.

There are several libraries in os/integrity/libraries which add new commands to the INTEGRITY Shell. These new commands invoke tools or tests provided with Reliance Edge. All of the libraries which add new commands are named in the pattern *_command.gpj. To use these commands, add their libraries into net_server_module.gpj, in the same fashion that libmount_redfs.gpj was added. For example, to add the newfs_redfs command to the Shell, add libnewfs_redfs_command.gpj and libnewfs_redfs.gpj into net_server_module.gpj.

Note

With INTEGRITY v11.7+, adding BDevTest (libbdevtest.gpj and libbdevtest_command.gpj) to the Shell requires that the Shell links with the storage driver libraries. If using the client-server configuration of Reliance Edge, these libraries will already be linked into another AddressSpace (that of the Reliance Edge Server), which can result in failures with drivers that assume they have exclusive ownership of the storage hardware. In such cases, either exclude BDevTest from the Shell, reboot before and after running BDevTest, or consider using the [proxy driver](#).

Be aware that net_server_module.gpj lives in your INTEGRITY installation, not your MULTI project; any changes made to it affect all INTEGRITY projects using that installation.

5.5 Using Reliance Edge from an application

For the most part, using Reliance Edge from your application is the same as using any file system on INTEGRITY: use the POSIX file system APIs like `open()`, `read()`, `write()`, and `close()`; or C library APIs like `fopen()`, `fread()`, `fwrite()`, and `fclose()`.

Consult the "INTEGRITY File System Services" chapter of the *INTEGRITY Libraries and Utilities User's Guide* and the "INTEGRITY File System Services Reference" chapter of the *INTEGRITY Libraries and Utilities Reference Guide* for further details on INTEGRITY's file system interfaces.

5.5.1 Mounting Reliance Edge

Reliance Edge cannot be automatically mounted via the mount table (the `vfs_MountTable` array, typically defined in a mounttable.c), since that table only works for file systems implemented with the low-level IVFS, such as MSDOSFS or FFS. Instead, your application must mount Reliance Edge via the `mount()` system call.

```
// Prototype for the mount() system call
int mount(const char *fsType, const char *mountPoint, int mountOpts, void *data);
```

If using the client-server configuration of Reliance Edge, each client AddressSpace must mount each file system volume it uses, even if the volume was already mounted by another AddressSpace. Internally, the server only mounts the volume once, but the client has mount-time setup to do even if the server has the volume mounted. When a client unmounts a file system volume, the server will unmount it only if no other client has it mounted.

The first parameter to `mount()`, the file system type (`fsType`), must be a pointer to the string "redfs" when mounting Reliance Edge.

The second parameter to `mount()`, the mount point (`mountPoint`), must match a volume path prefix given when Reliance Edge was configured. In the example below, "//mnt" is used, so "//mnt" would have been the volume path prefix given to the Configuration Utility, written into redconf.c. The mount point must start with "//", so all of the volume path prefixes should be configured to start with "//".

The third parameter to `mount()`, the mount option flags (`mountOpts`), is used for INTEGRITY mount flags, defined in `sys/mnttab.h`. There are many such flags; Reliance Edge supports `MNT_RDONLY` (which mounts the file system read-only) and `MNT_NOTRIM` (which disables [Automatic Discards](#)).

The forth parameter to `mount()`, the file system data (`data`), has a unique meaning for each file system type. Reliance Edge uses the data parameter in a similar manner to FFS or MSDOSFS, that is, to pass in a structure which specifies the name of the block device, and also includes any optional parameters. This structure is `struct redfs_args` from `os/integrity/include/redfs_integrity.h`; that header must be included to use `mount()` with Reliance Edge.

```
// Arguments to mount redfs filesystems
struct redfs_args
{
    const char *fspec; // Name of device to mount
    unsigned int flags; // REDFSMNT_* flags
};

#define REDFSMNT_FMTONCE 0x1U // format if volume cannot be mounted
#define REDFSMNT_FMTALWAYS 0x2U // format always
```

An example of using `struct redfs_args` with `mount()`:

```
struct redfs_args mo = {0};
int err;

mo.fspec = "<ram:4M>ramdisk"; // The name of the storage medium
mo.flags = REDFSMNT_FMTONCE; // Format if unformatted
err = mount("redfs", "/mnt", 0, &mo);
if(err)
{
    perror("mount");
    Exit(1);
}
```

Note that the `struct redfs_args` is zero-initialized even though all of its fields are explicitly initialized. This is a good practice, so that if any field is added to `struct redfs_args` in the future, it will inherit a default value of zero without the need to update application code.

The `fspec` field of `struct redfs_args` is the storage device name. It should not include any partition identifier, since that is handled by the volume definition in `redconf.c`. If a partition is included, such as "`<ram:4M>ramdisk:b`", `mount()` will fail.

The `flags` field of `struct redfs_args` recognizes two flags, both pertaining to automatic formatting:

- If `REDFSMNT_FMTONCE` is specified and `red_format()` is enabled, then if `mount` initially fails because the storage device appears to be unformatted, or is formatted with an invalid configuration, or there is an I/O error reading the metadata, then the file system will be reformatted and the `mount` will be retried. (Think carefully before using this option in production: it might unexpectedly reformat the file system due to a misbehaving storage device. Can you accept that data loss? Can your application recover from starting with an empty volume that has not been populated with any files? If the answer to either question is no, then you should probably not use `REDFSMNT_FMTONCE`.)
- If `REDFSMNT_FMTALWAYS` is specified and `red_format()` is enabled, then prior to mounting the file system will be reformatted.

If `red_format()` has been disabled, then specifying either flag will cause `mount()` to fail with an `EOPNOTSUPP` error.

5.5.1.1 Unmounting

A Reliance Edge volume which has been mounted with `mount()` can be unmounted with `umount()`.

```
// Prototype for the umount() system call
int umount(const char *mountPoint, int unmountOpts);
```

The only option flag (`unmountOpts`) for `umount()` is `MNT_FORCE`. If `MNT_FORCE` is specified, all open handles for the volume are closed prior to `umount`. If `MNT_FORCE` is omitted, attempting to `umount` a volume with open handles will fail, setting `errno` to `EBUSY`.

5.5.2 Reliance Edge paths

Reliance Edge paths behave much like the native file system paths, except that the mount point—such as “//mnt”—which is the initial part of an absolute path, looks a little different due to the two leading path separators. But the behavior is much the same as any other path. When using path-based APIs like `open()`, `fopen()`, or `unlink()`, you can include the mount point in an absolute path or `chdir` into the mount point directory (or a subdirectory) and use a relative path. Thus, to delete the “temp.txt” file from the root directory of a Reliance Edge volume with the name “//mnt”, you can either `unlink("//mnt/temp.txt")`; or `chdir("//mnt")` and `unlink("temp.txt")`.

One note about dot-dot (..)—a dot-dot from a Reliance Edge root directory stays in the root directory. So if “//mnt” is the root directory, “//mnt/..” is also the root directory, not “/” or any higher-level path. This differs from the native file systems: if FFS is mounted on “/” and MSDOSFS on “/mnt”, then “/mnt/..” would be a reference to “/”.

5.5.3 Recovering from a server fault

This section applies to the client–server configuration only.

After file system clients have connected to the file system server, if the server should happen to fault or crash, provisions have been made to allow for recovery—at least theoretically. The server AddressSpace includes an exception monitoring task which will jump into action if the main server task dies; most importantly it will close the client connections so that the clients do not block indefinitely waiting for a server response that will never come.

If the server is restarted (for example, if a new server instance is dynamically downloaded), the client library will establish a connection to the new server. The client library remembers which file system volumes it had mounted prior to the crash, and will instruct the server to mount them once again. Similar to power loss, the file system reverts to the committed state, meaning all changes since the last transaction point are reverted; which means that changes the applications thinks it made might be gone. The client library is incapable of restoring open handles (file descriptors or directory handles), so after a server crash all handles are marked stale; all operations on a stale handle, except `close()` or `closedir()`, will fail with an `ESTALE` errno; the application must close and reopen its handles to recover. Furthermore, because the current working directory is stored on the server side, and not saved by the client library, it will reset to the root directory of volume zero after a server crash; something to keep in mind when using working directories in your application.

If the server is not restarted, or in the interim between the crash and the restart, file system operations will fail. As mentioned above, operations on stale handles will fail with `ESTALE`; all other operations will attempt to reconnect to the server, which will eventually time out and fail with an `EIO` errno.

While these provisions for recovering from a server fault do exist, it is not recommended to put too much faith in them; there is too much that can go wrong in both the application and the file system for wholly reliable recovery. A server fault, if seen, would constitute a bug; rather than working around it, report it to support@tuxera.com.

5.5.4 Reliance Edge INTEGRITY ioctls

Reliance Edge implements several ioctls to provide functionality not available via the normal file system interfaces. These ioctls are defined in `os/integrity/include/redfs_integrity.h`.

The `ioctl()` system call needs a file descriptor. For the current set of ioctls, this can be the file descriptor for any file or directory (including the root directory) on the Reliance Edge volume, and the open mode is not important. All of the current ioctls operate on a file system volume, either doing something to it or retrieving information about it; if there are multiple volumes, the ioctl pertains to the volume which contains the file or directory that was opened for the file descriptor.

5.5.4.1 IORE_TRANSACT

Commit a transaction point. This is implemented by calling `red_transact()`; see the description of that API for details.

```

int err;

err = ioctl(fd, IORE_TRANSACT);
if(err)
{
    perror("ioctl(IORE_TRANSACT)");
    Exit(1);
}

```

5.5.4.2 IORE_TRANSACT_MASK_GET

Get the transaction mask. This is implemented by calling [red_gettransmask\(\)](#); see the description of that API for details.

```

uint32_t transmask;
int err;

err = ioctl(fd, IORE_TRANSACT_MASK_GET, &transmask);
if(err)
{
    perror("ioctl(IORE_TRANSACT_MASK_GET)");
    Exit(1);
}

```

5.5.4.3 IORE_TRANSACT_MASK_SET

Set the transaction mask. This is implemented by calling [red_settransmask\(\)](#); see the description of that API for details.

```

uint32_t transmask;
int err;

transmask = RED_TRANSACT_MANUAL; // Or any other mask.
err = ioctl(fd, IORE_TRANSACT_MASK_SET, &transmask);
if(err)
{
    perror("ioctl(IORE_TRANSACT_MASK_SET)");
    Exit(1);
}

```

5.5.4.4 IORE_MAX_FILE_SIZE_GET

Get the maximum file size of the file system volume. This is the same value that [red_statvfs\(\)](#) puts into **REDSTATFS::f_maxsize**.

```

uint64_t maxfilesize;
int err;

err = ioctl(fd, IORE_MAX_FILE_SIZE_GET, &maxfilesize);
if(err)
{
    perror("ioctl(IORE_MAX_FILE_SIZE_GET)");
    Exit(1);
}

```

5.5.5 Reliance Edge tool APIs

The Reliance Edge tools, including a checker and formatter, can be run from a Windows or Linux host machine. See [Command-Line Host Tools](#) for details.

For use on the INTEGRITY target, Reliance Edge provides a set of APIs for the formatter (newfs) and checker (fsck):

```

Error newfs_redfs_async(int argc, char *argv[], FILE *input, FILE *output, Task *theTask);
Error newfs_redfs_async_cleanup(Task theTask, Value *exitStatus);
Error newfs_redfs(int argc, char *argv[], FILE *input, FILE *output, Value *exitStatus);

```

```
Error fsck_redfs_async(int argc, char *argv[], FILE *input, FILE *output, Task *theTask);
Error fsck_redfs_async_cleanup(Task theTask, Value *exitStatus);
Error fsck_redfs(int argc, char *argv[], FILE *input, FILE *output, Value *exitStatus);
```

These are modeled after similar APIs provided by FFS and MSDOSFS, which you can find described in the *INTEGRITY Libraries and Utilities Reference Guide*; search for `newfs_ffs`, `newfs_msdos`, etc. To use the above APIs, include `os/integrity/include/redfs_integrity.h` in your source (and include the relevant libraries in your application, as described in [Building the tools for the target](#).)

Caveats:

- The argc/argv syntax is the same as the [Command-Line Host Tools](#), which differs from the syntax for the similar FFS and MSDOSFS functions. Thus, to format with MSDOSFS you would (assuming each space-delimited word is a separate element in the argv array) use: "`newfs_msdos <ram:4M>ramdisk`". Whereas with Reliance Edge you would use: "`newfs_redfs <vol> --dev=<ram:4M>ramdisk`", where "`<vol>`" is a volume name (like "`//mnt`") or volume number (like "0").
- The `input` parameter for the `newfs` and `fsck` functions is inoperative because the Reliance Edge tools do not accept user input.
- The Reliance Edge `fsck` was designed to run from the host machine (desktop Windows or desktop Linux), not the embedded target; depending on the file system configuration (notably the file system size), it may need a lot of RAM to run successfully.
- The Reliance Edge `fsck` is only available on the target if `REDCONF_CHECKER` is set to 1 in `redconf.h`; this cannot be done with the [Configuration Utility](#), only by editing `redconf.h` manually.

5.6 Reliance Edge and Special-Purpose Storage Drivers

This section applies to INTEGRITY v11.7+ only.

5.6.1 Reliance Edge and the Multiplexor Driver

Reliance Edge can partition the storage device internally, via the [Sector Offset](#) and [Sector Count](#) settings that are saved in `redconf.c`. This internal partitioning is sufficient when Reliance Edge is the only file system using the storage device from INTEGRITY. This code assumes it has exclusive access to the storage device and will use the storage driver API to lock the device. This is problematic when the storage device is shared with other INTEGRITY file systems (like FFS, MSDOSFS, or PJFS). When such sharing is desired, the partitioning should instead be accomplished with the multiplexor ("mux") driver.

Within the INTEGRITY v11.7+ installation, the mux driver can be found at `modules/ghs/bpsrc/driver/storage/gh_mux_driver.c`. The comment at the top of the source file has instructions on using the driver. In addition to those instructions, the `AddressSpace` which hosts `libredfs_unified.gpj` or `libredfsserver.gpj` needs to be updated to link with `libstorage-driver-mux.a`.

Here is an example of a mux device string: If the device string is normally `<sdhc_mmc:rcar_sdhi_gen3>SDHIDev` and you want Reliance Edge to use a partition that starts at sector 264192 (0x40800) and is 30851072 (0x1D6C000) sectors long, the mux device string would be `<mux:0x40800>0x1D6C000+<sdhc_mmc:rcar_sdhi_gen3>SDHIDev`.

When the mux driver is used for partitioning, the [Sector Offset](#) setting for the partition should always be zero: the mux driver will take care of adding the sector offset into the I/O requests. When Reliance Edge is configured with a [Sector Offset](#) of zero, make sure you do not use a non-mux device string by accident, since Reliance Edge would then be reading and writing to the wrong place on the storage device.

5.6.2 Reliance Edge and the Proxy Driver

While the main storage driver library (libstorage-core.a) can be linked into multiple AddressSpaces, the library or libraries which provides the low-level driver for a real storage device (like the libstorage-driver-sdio mmc.a and libsdhi mmc rcar_gen3.a libraries used for SD/MMC in the R-Car project) should only be linked into one AddressSpace. If these libraries are in multiple AddressSpaces, independently attempting to manipulate the storage hardware, then failures are likely.

The proxy driver allows the low-level storage driver libraries to exist in one AddressSpace but to be used from many AddressSpaces. The proxy driver is described in the "Proxy Devices and Proxy Servers" section of the *INTEGRITY BSP User's Guide* (bspguide.pdf), which can be found in the manuals subdirectory of the INTEGRITY installation. The general idea is to create a proxy server which owns the block device; other AddressSpaces use proxy device strings which result in requests that are forwarded to the proxy server.

To create a proxy server, add a new AddressSpace to the MULTI project and have it invoke `gh_storage_Proxy_Server()` from its `main()`, as in the following example:

```
#include <INTEGRITY.h>
#include <stdio.h>
#include <device/storagedriver.h>

int main(void)
{
    Error err;

    // The name "proxyserv" is arbitrary, but it affects the device strings.
    err = gh_storage_Proxy_Server("proxyserv");

    // gh_storage_Proxy_Server() never returns except on error.
    printf("gh_storage_Proxy_Server() failed with error %d\n", (int)err);
    Exit(1);
}
```

Update the proxy server AddressSpace to link with libstorage-core.a, libstorage-proxy-server.a, and whatever low-level storage drivers it needs (such as libstorage-driver-sdio-mmc.a). Depending on the drivers used, it may also be necessary to increase the heap size of the AddressSpace.

The AddressSpace which hosts libredfs_unified.gpj or libredfsserver.gpj would link with libstorage-core.a and libstorage-driver-proxy.a. It would *not* link with any low-level drivers. Likewise for the AddressSpaces of other file systems, like the IVFS Server.

The device strings must be changed when the proxy device is in use. The device string must be in the form `<proxy:name_of_proxy_server,basedev>`, where `name_of_proxy_server` is the name passed into `gh_storage_Proxy_Server()` and `basedev` is the base device string. For example, if the proxy server name is `proxyserv` and the base device string is `<sdhc_mmc:rcar_sdhi_gen3>SDHIDev`, then to use the proxy driver device string would be `<proxy:proxyserv,<sdhc_mmc:rcar_sdhi_gen3>SDHIDev>`.

5.7 Limitations

5.7.1 No support for root file system

Reliance Edge cannot be the root file system (that is, the file system mounted on "/"). Reliance Edge is separate from the main file system hierarchy, so it can be used in a setup which lacks a root file system; or it can be used alongside an MSDOSFS or FFS root file system.

5.7.2 Unimplemented functionality

INTEGRITY provides numerous POSIX system calls for accessing a file system, and Reliance Edge implements the most commonly used functions. However, there is some functionality that is not implemented. The following functions

are not supported:

- `symlink()`, `readlink()`
- `mkfifo()`, `mknod()`
- `chmod()`, `lchmod()`, `fchmod()`
- `chown()`, `lchown()`, `fchown()`
- `chroot()`, `fchroot()`
- `utimes()`, `lutimes()`, `futimes()`
- `setuid()`, `setrgid()`, `seteuid()`, `setegid()`, `setsupgroups()`
- `realpath()`
- `fchdir()`
- `dup()`
- `flock()`
- `fcntl()`
- `pipe()`
- `poll()`
- All socket functions are unsupported, including: `socket()`, `accept()`, `bind()`, `connect()`, `listen()`, `recv()`, `send()`, and `shutdown()`

Attempting to use any of the above functions with a Reliance Edge path or file descriptor will fail with an `ENOSYS` error.

Furthermore, when Reliance Edge functionality is disabled in your configuration, this moves the corresponding file system function into the unsupported category. This should be mostly intuitive: for example, if your configuration has `REDCONF_API_POSIX_MKDIR` set to 0, which removes `red_mkdir()`, then the `mkdir()` system call will no longer work with Reliance Edge, but will instead fail with an `ENOSYS` error.

For `mount()`, only the `MNT_RDONLY` and (in INTEGRITY v11.7+) `MNT_NOTRIM` flags are supported. All the other `MNT_*` flags—such as `MNT_UPDATE`, `MNT_RELOAD`, `MNT_ASYNC`, and `MNT_NOATIME`—are unsupported. (The behavior of `MNT_NOATIME` can be achieved via compile-time product configuration: see the [Access Time](#) section.)

If any of this missing functionality is required for your project, please contact support@tuxera.com.

5.7.3 Unintuitive CWD behavior

When Reliance Edge and a native INTEGRITY file system are used simultaneously, the current working directory (CWD) behavior may be unintuitive, namely when the CWD moves from Reliance Edge to a native file system. This is because Reliance Edge is not notified when the CWD (current working directory) changes to another file system. For example, if Reliance Edge has a volume mounted at "`//redfs`" and FFS has a volume mounted at "`/ffs`", consider the following sequence of operations:

```
chdir("//redfs/subdir");
chdir("//ffs/subdir";
rmdir("//redfs/subdir");
```

Reliance Edge is not notified when the CWD moves to "/ffs/subdir"; as far as Reliance Edge is concerned, the CWD is still "//redfs/subdir". This means that the `rmdir()` will fail, since Reliance Edge does not allow the CWD to be deleted.

As a workaround, when the CWD is on a Reliance Edge volume, before `chdir()`ing to another file system, you can `chdir()` to the root directory of the Reliance Edge volume. For example:

```
chdir("//redfs/subdir");
chdir("//redfs");
chdir("//ffs/subdir");
rmdir("//redfs/subdir");
```

5.8 Behavioral Differences from Native File Systems

In addition to the [unimplemented functionality](#) discussed above, the behavior of Reliance Edge differs in minor ways from FFS and MSDOSFS. For example:

- The `errno` value in error cases is not always the same. For example, the native file systems use `EFAULT` for `NULL` pointer parameters, but Reliance Edge will use `EINVAL`.
- Attempting to `unlink()` or `rmdir()` a file or directory which is currently open fails, returning -1 and setting `errno` to `EBUSY`.
- Attempting to `unlink()` or `rmdir()` a directory which is the current working directory of any task fails, returning -1 and setting `errno` to `EBUSY`.
- Reliance Edge allows an empty directory to be deleted with `unlink()`, whereas the native file systems do not.
- Reliance Edge is stricter with open flags; it does not allow `O_TRUNC` or `O_CREAT` with `O_RDONLY`, nor does it allow `O_EXCL` without `O_CREAT`; whereas the native file systems allow those flag combinations.
- Reliance Edge returns an error if `read()` or `pread()` are used with a directory file descriptor, while the native file systems permit the read.
- Reliance Edge returns an error if `fstat()` or `statvfs()` are given a `NULL` parameter for the buffer, while the native file systems will return success, provided that the path is valid.
- Reliance Edge does not have physical directory entries for "." or "..", while the native file systems do, which affects behavior in two ways. First, it means that `readdir()` will not return "." or ".." entries with Reliance Edge. Secondly, the link count of a directory will always be one with Reliance Edge, since there are no ".." directory entries to increase it.
- Reliance Edge allows extra path separators after a file name, like "foobar.txt///", while the native file systems do not.
- POSIX says that when `rename()` is given two paths which point to the same inode, the operation should succeed but have no effect. Reliance Edge implements this behavior, unlike FFS which deletes the old name.

See also the [Differences from POSIX](#) section, which describes how the Reliance Edge POSIX-like API differs from POSIX. While an INTEGRITY application would not normally use the POSIX-like API directly, under the hood, Reliance Edge on INTEGRITY is using the Reliance Edge POSIX-like API. For example, an `open()` call is internally translated into a `red_open()` call with the equivalent flags; and any resulting `red_errno` (such as `RED_ENOENT`) is translated into the equivalent INTEGRITY `errno` (such as `ENOENT`). Thus, in many cases the behavior of Reliance Edge on INTEGRITY is similar to the POSIX-like API.

5.9 Legacy Block Device Interfaces

INTEGRITY v11.7 introduced the storage driver API, a new standard interface for block device access. It superseded several older interfaces which were used to interact with different types of block devices. Unlike some of the older interfaces, the storage driver API supports discard and flush operations. When used with INTEGRITY v11.7+, Reliance Edge always uses the storage driver API for its block device access.

Reliance Edge also supports pre-v11.7 versions of INTEGRITY. For those older versions, it provides block device implementations for IODevice (tested with the SD/MMC driver on the BeagleBone Black) and IDE/SATA (tested with SATA hard drives on an x86 PC).

The rest of this section has some notes that pertain to using these legacy block device interfaces. If you are using INTEGRITY v11.7+, they do not apply.

5.9.1 Device names

The legacy block device interfaces may differ in their conventions for block device names. For example, the old IODevice name for an SD card might be `SDCardDev` whereas the new storage driver name might be `<sdhc_mmc:generic_sdhc>SDCardDev`. Consult your BSP documentation or ask Green Hills if you do not know the name of your block device.

5.9.2 Enabling IDE/SATA implementation

The IDE/SATA interface is disabled by default since compiling it requires linking with an extra library, `libblockdriver.a`. To enable the interface, set `REDCONF_INTEGRITY_IDE` in `os/integrity/services/osbdev_ide.c` to 1. Alternatively, rather than editing the file, the macro value can be set in your MULTI project (as discussed in [Additional configuration macros](#)).

5.9.3 Flushing the block device

Neither the IODevice nor IDE/SATA device interfaces have a standard way to flush the underlying storage device, so flushing is not implemented. If the storage device has a hardware cache, there *must* be a way to flush it, or Reliance Edge cannot operate reliably. If you know that your storage device has a hardware cache, you need to work with Tuxera and/or Green Hills to implement a method of flushing the storage device.

5.9.4 Sharing block device with FFS or MSDOSFS

With the legacy block devices, file systems implemented with the low-level IVFS interface, such as FFS or MSDOSFS, need a multiplexor to share a block device with a file system implemented at a higher level, such as Reliance Edge or PJFS. For further details, contact Green Hills Software. Without a multiplexor, if Reliance Edge is on the same block device as MSDOSFS or FFS, then Reliance Edge should be unmounted when using the other file system, and vice versa.

Chapter 6

MQX Integration

This chapter describes the MQX port of Reliance Edge.

Note

The Reliance Edge MQX port is released as part of the commercial version. It is not available in the GPL version on GitHub. See [the Reliance Edge website](#) for commercial licensing options.

6.1 MQX VFS Overview

MQX is unique in the extensivity of its Virtual File System (VFS). In MQX, filesystems and I/O drivers are both abstracted through the VFS. Thus, Reliance Edge may be installed on one or more volume in the VFS (e.g. "VOL0:"), but it will open another device in the VFS for block I/O (e.g. "sdcard:"). The block device driver may then use the VFS to access even lower-level drivers (e.g. "esdhc:"). Thus, Reliance Edge uses the VFS for both input and output. This should be understood in order to avoid confusion when reading the sections below.

Freescale offers two variations of the MQX operating system. MQX may be used as a standalone operating system or it may be run on top of the Kinetis SDK (KSDK). The version of MQX that is shipped with the KSDK has several new features, including a redesigned virtual file system (VFS). The older VFS is referred to as the Formatted I/O system (FIO), and the newer VFS is referred to as the I/O Subsystem, or NIO. Reliance Edge is compatible with both FIO and NIO, and detects which VFS to use through a macro which MQX defines.

The Reliance Edge VFS interface is available only if the POSIX-like API [is enabled](#). For documentation on the VFS interface, see [redfs_mqx.h](#).

6.2 Building Reliance Edge for MQX

6.2.1 The Reliance Edge KDS Project

Note

If you do not have the Kinetis Development Studio, you may download it from the [Freescale website](#). The Reliance Edge MQX port has been tested on KDS version 3.0.0.

The supported method of building Reliance Edge for MQX is to use the supplied Kinetis Development Studio (KDS) project, which may be found in projects/mqx/TWR-K65F180M/RelianceEdge. This project should be opened in the same KDS workspace as the MQX operating system projects (and KSDK, if used) and any user projects. This setup is the same used by the MQX and KSDK example projects for KDS, and is assumed to be familiar to the user.

In order for Reliance Edge to be built, the path to the MQX source code and output must be specified. These values can be edited by opening the project's Properties page in KDS and selecting "C/C++ Build", "Build Variables." Set KsdkSrcPath to the KSDK install directory if using KSDK; otherwise set MqxSrcPath to point directly to the MQX install directory. Check and ensure the other variables are correct as well.

When the Reliance Edge KDS project is built, it generates an intermediate file libredfs.a. This file should be linked into the final executable.

6.2.2 The SD Card Demo KDS Projects

In addition to the Reliance Edge project, two demo projects are provided in the projects/mqx/TWR-K65F180M folder. These projects illustrate how to initialize the SD card driver, and how to install and use Reliance Edge through the VFS.

The KsdkDemo project requires KSDK (tested with versions 1.2.0 and 1.3.0), and uses the NIO VFS. Unlike the other two projects mentioned here, the KsdkDemo must link directly against several source files in the KSDK install directory. The easiest way to correctly specify these is to open the project's properties page, click Resources -> Linked Resources, and set the variable KSDK_SRC_PATH to point to the KSDK install directory.

The FioDemo project is designed to run on MQX 4.2.0, without the KSDK; it uses the older FIO VFS.

As in the Reliance Edge KDS project, the Build Variables will need to be checked and edited appropriately before attempting to build either demo project. For more details, see the README.txt file included with each project.

6.3 OS Services and Configuration

The MQX port provides a full implementation of the Reliance Edge OS services that is expected to be sufficient for most developers. However, some details require the developer's attention. See the [porting guide](#) for a generic description of the OS services.

6.3.1 Mutex service

Reliance Edge does not allow failures in the mutex acquire and release functions. However, the MQX_mutex_lock and _mutex_unlock methods are allowed to return errors. There are no circumstances in Reliance Edge where an error can be expected from either of these functions, but an assert is used to check for the unexpected. MISRA C:2012 requires that error information be checked; therefor the mutex services are in deviation if asserts are disabled (REDCONF_ASSERTS is set to 0).

MQX allows the user to disable mutexes through the configuration macro MQX_USE_MUTEXES. This macro must be set to 1 if the [Reliance Edge task count](#) is set to two or more.

6.3.2 Timestamp service

The timestamp service in the MQX port relies on MQX_HAS_TICK being set to 1. If this is not the case, then [RedOsTimestampInit\(\)](#) will return -RED_ENOSYS.

6.3.3 Real-time clock service.

The real-time clock service also relies on MQX_HAS_TICK. If it is not set to 1, then RedOsClockGetTime always returns 0.

This service relies on the MQX_time_get API function, which is documented to return the time since MQX startup unless _time_set has been called. If REDCONF_INODE_TIMESTAMPS is set to 1, then the user must call

`_time_set` or `_time_set_ticks` to set the clock before initializing Reliance Edge. Otherwise the inode timestamps will be filled with inaccurate information.

6.3.4 Block device service.

In the MQX operating system, all I/O devices are abstracted by the VFS. This allows Reliance Edge to work with a variety of devices without significant modification. The MQX os-specific configuration file ([redosconf.c](#)) specifies the device name to be used for block I/O for each volume. The user must install the specified block device in the MQX VFS *before* mounting the volume.

The os-specific configuration file must also specify whether each volume supports flush commands; i.e. whether `IOCTL_FLUSH_OUTPUT` is implemented by the block device and is needed to ensure that data has been permanently written to the disk. If the underlying device does cache any data, then flushing must be enabled in order to ensure that critical data has been written to the disk after a Reliance Edge transaction.

In order to support devices that require memory buffers to be aligned, the block device service statically allocates an aligned RAM buffer. Unaligned data is copied through the buffer when required by the underlying block device. This increases the RAM requirement of Reliance Edge by one block - `REDCONF_BLOCK_SIZE` bytes.

6.4 Supported Block Devices

Reliance Edge for MQX uses the VFS to perform block I/O underneath the filesystem. In MQX, any I/O device may be installed in the VFS, but not any I/O device is compatible with the Reliance Edge block device service.

A compatible I/O device must implement open, close, read, write, and seek functions. The read, write, and seek functions may perform I/O in bytes or in sectors (as identified by `IO_IOCTL_DEVICE_IDENTIFY`). Compatible devices may require buffer alignment of up to 8 bytes (as identified by `IO_IOCTL_GET_REQ_ALIGNMENT`). The seek function must be able to seek to sector boundaries. Raw Flash devices are not supported; like most other file systems, Reliance Edge requires a Flash Translation Layer (FTL) in order to use raw Flash memory. SD card and eMMC devices are supported because they implement an FTL in their hardware.

6.5 Tests

See [Testing](#) for a description of Reliance Edge testing suite.

All of the tests may be run on the MQX port. However, only the POSIX-like API test suite is ported to install Reliance Edge in the MQX VFS and access it through the VFS. All other tests access Reliance Edge directly through the POSIX-like or File System Essentials API.

In addition, two Kinetis Design Studio projects are supplied which exercise the API test suite on MQX. The projects are located at `projects/mqx/TWR-K65F180M/KsdkNioTests` (for MQX for KSDK 1.x with the NIO VFS) and `projects/mqx/TWR-K65F180M/FioTests` (for MQX standalone with the Formatted I/O VFS). See the included `README.txt` files for more details.

Chapter 7

ARM mbed Integration

Reliance Edge is ported for use as part of the ARM mbed ecosystem. This chapter describes the integration of Reliance Edge and ARM mbed.

7.1 Overview

ARM mbed is an ecosystem of software and hardware that is built around the ARM Cortex-M architecture. There are two forms of mbed which are available: mbed classic (mbed 2) and mbed OS (mbed 3). Although these two iterations of mbed share much of the same core, there are significant differences. Two differences are pertinent to this product:

- ARM mbed is not in and of itself an RTOS; however, an RTOS module is available for mbed classic. This module has not yet been ported to mbed OS. This means that the OS mutex and task modules of Reliance Edge are not implemented on mbed OS, and the user should set the configured [task count](#) to 1.
- The primary method of compiling an mbed classic project is through the online compiler at developer.mbed.org. Projects can also be exported from the online compiler to environments such as Eclipse/GCC. There is no sample Reliance Edge project for mbed classic. On the other hand, with mbed OS, ARM released [Yotta](#), a package management and build system for mbed OS projects. Yotta can automatically download required libraries from its online repository or from GitHub, although Reliance Edge is not built this way. Yotta then uses CMake and other tools to build the project. A sample project is included for mbed OS on the NXP FRDM-K64F board in the folder `projects/mbed/frdm-k64f`.

7.2 ARM mbed filesystem abstraction

ARM mbed includes a C Standard I/O (stdio) implementation, allowing users to access filesystems through methods such as `fopen()` and `fread()`. To integrate with this Virtual File System (VFS), a filesystem must inherit and implement the class `mbed::FileSystemLike`.

If the [POSIX-like API](#) is selected in the Reliance Edge configuration, then the `RedFileSystem` class implements the `FileSystemLike` class. Once a `RedFileSystem` instance is created, the filesystem may be accessed through methods such as `RedFileSystem::open("file1")` or through the VFS calls such as `fopen ("redvolume:/file1")` (assuming a volume named "redvolume:" which contains a file "file1"). Note that the VFS functions `telldir()` and `seekdir()` are not supported by Reliance Edge at this time.

If the [FSE API](#) is selected in the Reliance Edge configuration, then the `RedFileSystem` type is much simpler and does not integrate with the VFS. A `RedFileSystem` instance should still be created (which will provide access to the block device and calls `RedFsInit()`), but the filesystem must be accessed directly through the [File System Essentials Interface](#).

See the documentation for [RedSDFileSystem](#) for sample usage of Reliance Edge on mbed with the POSIX-like or FSE API options.

7.3 Reliance Edge configuration on ARM mbed

- The POSIX-like API call [red_rmdir\(\)](#) is not called from the ARM mbed port, so it is recommended that the [the corresponding configuration value](#) be disabled. The mbed filesystem classes do not implement a rmdir function; to remove a directory, [FileSystemLike::remove\(\)](#) may be called with the directory path (which resolves to [red_unlink\(\)](#)). The mbed VFS also does not implement a link function, so it is recommended that [red_link\(\)](#) be disabled in the Reliance Edge configuration.
- When using mbed OS, the Reliance Edge [task count](#) must be set to 1.
- The directory entry structure in mbed ([dirent](#)) allocates enough space to store NAME_MAX characters, which is generally 255. It is strongly recommended that the Reliance Edge [maximum name length](#) not be set higher than this value when [POSIX readdir](#) is enabled. (Typical use cases for Reliance Edge would not require filenames to be anywhere near 255 characters in length.)

7.4 OS Services

The ARM mbed port provides a full implementation of the Reliance Edge OS services that is expected to be sufficient for most developers. However, some details require the developer's attention. See the [porting guide](#) for a generic description of the OS services.

7.4.1 Mutex and task services

As of the date of the initial release of Reliance Edge for ARM mbed, mbed OS does not have an RTOS implementation, so multi-threaded filesystem access is not possible. For this reason, the mutex and task services are not available when using mbed OS, and the [task count](#) must be set to 1 (as mentioned above).

7.4.2 Block device service

The block device service for ARM mbed is a wrapper for the block device virtual methods in the [RedFileSystem](#) class. Block device I/O is ultimately handled by a child class of [RedFileSystem](#). Two example classes are provided: [RedSDFileSystem](#) (for using Reliance Edge on SD/MMC via SPI) and [RedMemFileSystem](#) (for using Reliance Edge on a ramdisk). The class interface required by [RedFileSystem](#) is nearly identical to the interface of [mbed::FATFileSystem](#), so any other block device classes that have been implemented for FAT on mbed should be portable to Reliance Edge with only trivial effort.

7.4.3 Assert service

Even when [assertion handling](#) is enabled, Reliance Edge calls the mbed [error\(\)](#) function only if [text output](#) is also enabled. If output is not enabled, the system will quietly enter an infinite loop on assert failure. It is expected that users who enable assertion handling will generally want to leave text output enabled as well.

7.5 Building

7.5.1 Creating and building an mbed classic project

Adding Reliance Edge to an mbed classic project requires adding all the required source code and include directories from the Reliance Edge repository to the IDE project. The list of files and folders required is found in the section [Source Files and Include Directories](#) of this guide. Note that the names of some OS services files are different for mbed; make sure you add all the files and folders in os/mbed.

7.5.2 Building for mbed OS

ARM mbed OS projects are built using Yotta (<http://yottadocs.mbed.com/>), which is a command line tool for managing dependencies and executing the build. Normally libraries are included in Yotta projects by adding a line to the dependencies list in the file `module.json` that specifies the public location of the library source code. However, the ARM mbed port of Reliance Edge is only available under proprietary licensing, so it is not available from public repositories. Instead, Reliance Edge uses a custom CMake script to build a library (`reliance-edge-fs`) and link it to the desired target.

An example project (projects/mbed/frdm-k64f) is provided to show how Reliance Edge may be built with Yotta for mbed OS. To add Reliance Edge to an existing Yotta module or project, follow these steps:

1. Include the Reliance Edge CMake build script

Create a `.cmake` file in your Yotta project's source folder and add these three lines, substituting "path/to/redfs" with the path to the Reliance Edge source tree (use `${CMAKE_CURRENT_LIST_DIR}` for relative paths):

```
set(REDFS_ROOT "path/to/redfs")
set(TARGET_NAME "${YOTTA_MODULE_NAME}")
include("${REDFS_ROOT}/os/mbed/reliance-edge-fs-local.cmake")
```

This will instruct the build system to create a library `reliance-edge-fs` and link it to `TARGET_NAME`.

If you have any additional test targets, you will need to repeat this step for those as well, editing `TARGET_NAME` appropriately (see the CMake scripts in `projects/mbed/frdm-k64f/test` for example).

2. Add the Reliance Edge project configuration files

Copy the files `redconf.h` and `redtypes.h` from the Reliance Edge example project (`projects/mbed/frdm-k64f`) to your project root folder. Copy the file `redconf.c` from `projects/mbed/frdm-k64f/source` to your source folder (and to any additional test target folders).

3. Customize your project configuration

Run the Reliance Edge [configuration utility](#) to customize your filesystem configuration. Load the two `redconf` files from the above step, and make sure to save them to their original locations when finished.

Once Reliance Edge has been added to your project, you can access it by creating an instance of a `RedFileSystem` class (either `RedSDFileSystem` or `RedMemFileSystem`). The project will build when `yotta build` is invoked.

7.6 Testing

Note

See the [Testing](#) page for a description of Reliance Edge testing suite.

All of the Reliance Edge tests may be run on the ARM mbed port. However, most of the tests are designed to validate the behavior of the Reliance Edge core and will not exercise the `mbed VFS` and related port code. The exception

is the [OS-specific API test](#), which does use the native VFS to exercise Reliance Edge. The OS API test is built as part of the sample mbed OS project at `projects/mbed/frdm-k64f`. The code which invokes the test, located in `projects/mbed/frdm-k64f/test/os-api-test`, may easily be adapted for use on mbed classic. See `main.cpp` for the code, and `osapitest.cmake` for a list of extra C files required to build the test (beyond what is required to [build](#) Reliance Edge itself).

Expected failures

Because the OS API test runs on the native VFS, there may be some test failures due to unexpected behavior in the VFS layer that are not related to Reliance Edge. On ARM mbed OS, there is one test that fails for this reason. The first set of test cases, "Reliance Edge Fopen Tests", fails with the following message in the detailed output:

```
Call freopen with NULL filepath and same mode
FAIL  "freopen"  Actual value 'hTestFile != NULL' is false
```

This failure occurs when the test to call `freopen()` with a NULL file path and a valid open file handle. The ISO C Standard behavior is to reopen the same file as the handle refers to, but the mbed VFS does not support this, so the test case fails.

In mbed classic, the test "Reliance Edge OS API Write Tests" also fails, with the following message in the output:

```
Perform zero-length writes.
FAIL  "write"  Actual value 0x00000001 mismatches expected 0x00000000
```

This again happens due to a failure of the VFS to conform to the C Standard behavior. Although `fwrite()` is documented to return 0 when asked to write 1 item that is 0 bytes long, the mbed VFS returns 1 instead in this case.

In both mbed classic and mbed OS, there are also two tests that are marked as skipped because all of their test cases are disabled when running on mbed in order to avoid hard faults. These tests are "Reliance Edge Readdir Error Tests" and "Reliance Edge Rewinddir Error Tests".

Chapter 8

U-Boot Integration

Reliance Edge has been ported to the Universal Boot Loader (U-Boot). This allows a system using U-Boot to boot from an operating system image or kernel image stored on a Reliance Edge file system volume. Storing the OS image on Reliance Edge has the advantage that the OS image file can be atomically updated via the transaction point mechanism in Reliance Edge. It also avoids the need to mount another file system (such as FAT) from the RTOS to update the OS; this means it is not necessary to find, purchase, or integrate another file system other than Reliance Edge for OS updates.

The U-Boot port is only available in the open-source kit, since only the GPLv2 license of the open-source kit is compatible with the GPLv2 license of U-Boot. If you are using the commercial kit for your primary RTOS, you can separately acquire the corresponding open-source release in order to use the U-Boot port.

8.1 Reliance Edge configuration for U-Boot

Reliance Edge for U-Boot is mostly configured the same way as it is in other environments; see the [Configuration Utility](#) and [Product Configuration](#) chapters. However, for U-Boot, there are the following restrictions upon the configuration:

- Ensure "Use POSIX API" is selected (which sets `REDCONF_API_POSIX` to 1); this is the default setting. Only the [POSIX-like API](#) provides the necessary functionality to integrate into the U-Boot file system layer.
- Ensure "Enable readonly configuration" is selected (which sets `REDCONF_READ_ONLY` to 1). Presently writing to a Reliance Edge disk is not supported in U-Boot.
- Ensure "Maximum task count" is set to 1 (which sets `REDCONF_TASK_COUNT` to 1).
- On the Memory tab, be aware that you cannot include string.h for the string and memory functions. Selecting "Use Reliance functions" will always work.

The above restrictions will not be enforced by the Configuration Utility. Some are enforced at compile-time with `#error` directives.

8.2 U-Boot project

Before building Edge into U-Boot, the redconf.c and `redconf.h` files that reside in the projects/u-boot directory must be configured. The U-Boot configuration must match the configuration used by the primary project for all settings that affect the on-disk layout. It may be easiest to copy your main project configuration and update the settings described in the

previous section to be correct for U-Boot. If you wish to minimize memory usage in the U-Boot environment, you can set [Handle Count](#) to 1 and [Block Buffer Count](#) to its minimum.

Note that the path prefix for the volume will need to precede the name of the any file or directory being accessed through U-Boot.

8.3 Building U-Boot

Building is performed in the U-Boot installation. But first, a patch must be applied in order to integrate Reliance Edge into build process. Multiple patches are available for different versions of U-Boot. The patch files reside in projects/u-boot and are named redfs_uboot-{ver}.patch. For example, the redfs_uboot-2017.11.patch file should be used for U-Boot version 2017.11 and later. Run this patch in the root of the U-Boot build directory. The patch command should resemble:

```
patch -p1 < redfs_uboot-2017.11.patch
```

Next, a symbolic link must be added in the U-Boot tree that points to the Edge installation. Run the following command from the root of the U-Boot development tree while substituting the location where Reliance Edge is installed in your environment:

```
ln -s ~/edge fs/redfs/red
```

You should now be able to clean and build U-Boot that has Reliance Edge integrated. Note that U-Boot requires the MLO to be contiguous and the first file in a FAT partition. Format the FAT partition, copy the MLO file, sync, then copy the u-boot.img file.

8.4 Boot script for U-Boot

The boot script for U-Boot will be unique to your environment. The following example is for a BeagleBone Black that has two partitions. The first partition is FAT and contains the U-Boot files. The second partition is formatted with Reliance Edge and has a boot image file named int.bin. Reliance Edge was configured with //mnt as the mount point name. In this example, the boot script is loading a file int.bin from the second partition of an MMC device.

```
load mmc 0:2 ${loadaddr} //mnt/int.bin
mmc dev 1 2
mmc read 0x402f0400 0 0x80
go ${loadaddr}
```

The boot script must be compiled. If the script was named boot.scr.txt, the command to compile it would look similar to the following:

```
mkimage -n 'INTEGRITY Boot Script' -A arm -O linux -T script -C none -d boot.scr.txt boot.scr
```

The resulting boot.scr file should be copied to the FAT partition where the MLO and u-boot.img files reside.

Chapter 9

Building

This chapter provides guidance on building Reliance Edge. This includes both [building the file system driver](#) and [building the host tools](#).

If you are using an RTOS for which Tuxera provides a port, please also refer to the chapter in this manual for that RTOS, since it will have more specific advice about building for that RTOS.

9.1 Building the Reliance Edge Driver

There are a number of approaches to building Reliance Edge. It can be built from an IDE like Eclipse, Atmel Studio, or Code Composer Studio; or it can be built from the command line, using Makefiles or modern alternatives. The linking can be done all at once (by building Reliance Edge, the application, and the RTOS at the same time) or later, by building Reliance Edge as a library that will be linked into the application and RTOS. The first step, required for any of these approaches, is to figure out which Reliance Edge source files need to be cross-compiled, and which Reliance Edge directories must be listed as include directories.

9.1.1 Source Files and Include Directories

Not all source files included in the Reliance Edge repository are necessary to build the file system driver; some are only needed for tests, host tools, or other environments. The following list is sufficient to build the file system driver in any configuration:

```
bdev/bdev.c
core/driver/blockio.c
core/driver/buffer.c
core/driver/core.c
core/driver/dir.c
core/driver/discard.c [commercial customers only]
core/driver/format.c
core/driver/imap.c
core/driver/imapextern.c
core/driver/imapinline.c
core/driver/inode.c
core/driver/inodedata.c
core/driver/volume.c
fse/fse.c
os/YOURRTOS/services/osassert.c
os/YOURRTOS/services/osbdev.c
os/YOURRTOS/services/osclock.c
os/YOURRTOS/services/osmutex.c
os/YOURRTOS/services/osoutput.c
os/YOURRTOS/services/ostask.c
os/YOURRTOS/services/ostimestamp.c
posix/path.c
```

```
posix posix.c
util bitmap.c
util crc.c
util memory.c
util namelen.c
util sign.c
util string.c
projects/YOURPROJ/redconf.c
```

Substitute YOURRTOS with the name of directory where you implemented the [OS Services](#), or with stub if you have not yet implemented the OS services; and substitute YOURPROJ with the name of your project directory.

In the Reliance Edge MQX port, the following files are necessary as well:

```
os/mqx/vfs/redfs_fio.c
os/mqx/vfs/redfs_nio.c
projects/YOURPROJ/redosconf.c
```

In the Reliance Edge mbed port, the following OS-specific code files are required (the files in os/mbed/RedSDFileSystem are required for using Reliance Edge to access SD/MMC cards):

```
os/mbed/services/osassert.cpp
os/mbed/services/osbdev.cpp
os/mbed/services/osclock.c
os/mbed/services/osmutex.cpp
os/mbed/services/osoutput.cpp
os/mbed/services/ostask.cpp
os/mbed/services/ostimestamp.c
os/mbed/RedFileSystem/RedFileSystem.cpp
os/mbed/RedFileSystem/RedFileHandle.cpp
os/mbed/RedFileSystem/RedDirHandle.cpp
os/mbed/RedSDFileSystem/RedSDFileSystem.cpp
os/mbed/RedSDFileSystem/SDCRC.cpp
```

It is possible but not necessary to remove driver files which are not needed for a given configuration. For example, if you are not using the File System Essentials API, there is no need to build [fse/fse.c](#). However, all of the source files have preprocessor logic that conditions out all of the code if it is not needed, so removing files you do not need will not shrink the size of the object code, and will only marginally improve build times. The advantage of building all the source files is it allows you to change the Reliance Edge configuration without altering the build process.

The following Reliance Edge include directories are required to build the above source files:

```
include
core/include
os/YOURRTOS/include
projects/YOURPROJ
```

To this you must add any include directories necessary to build your implementation of the OS services. For example, the FreeRTOS implementation of the OS services includes FreeRTOS.h and other FreeRTOS headers, so FreeRTOS/Source/include would need to be an include directory.

For the Reliance Edge mbed port, you must add `os/mbed/RedFileSystem` and `os/mbed/RedSDFileSystem` to your include directories as well.

9.1.1.1 Files and Include Directories for Test Code

The source files and include directories given above are sufficient to build the file system driver, but more source is required if you wish to use the test code distributed with Reliance Edge.

Generally, compiling the test code will increase build times, but will only increase object size if the test code is referenced and thus pulled into the binary by the linker; so it is safe to include the test code even when it is not used. However, it should be noted that the test code is less portable than the driver code (for example, the driver does not use the C library, but the test code does), and hence more prone to compilation failures—see the [Testing](#) chapter for details.

GPL Distribution

One test program is included with the GPL distribution of Reliance Edge: `fsstress`. To build this, in addition to the source files for the file system driver, you must also compile:

```
tests posix/fsstress.c  
tests util/atoi.c  
tests util/math.c  
tests util/printf.c  
tests util/rand.c
```

If your IDE requires header files included with quotation marks to be in an include directory, the following must be added as an include directory: `tests posix`.

Commercial Distribution

A number of test programs and test suites are included in the commercial distribution of Reliance Edge. To build all of them, in addition to the source files for the file system driver, you must also compile:

```
core checker/checker.c  
tests bdevtest.c  
tests disk_full/tdiskfull.c  
tests disk_full/tdiskfulltestcases.c  
tests fse/stress.c  
tests fse_api/tfse.c  
tests fse_api/tfsetestcases.c  
tests fsiotest.c  
tests posix/mvstresstest.c  
tests posix_api/tinit.c  
tests posix_api/tread.c  
tests posix_api/twrite.c  
tests posix_api/tlink.c  
tests posix_api/tclose.c  
tests posix_api/tfsync.c  
tests posix_api/tlseek.c  
tests posix_api/ttruncate.c  
tests posix_api/tfstat.c  
tests posix_api/tfstrim.c  
tests posix_api/tclosedir.c  
tests posix_api/tmkdir.c  
tests posix_api/topen.c  
tests posix_api/topendir.c  
tests posix_api/treaddir.c  
tests posix_api/trename.c  
tests posix_api/trewinddir.c  
tests posix_api/trmdir.c  
tests posix_api/tspecialcases.c  
tests posix_api/tstatvfs.c  
tests posix_api/ttransact.c  
tests posix_api/tunlink.c  
tests posix_api/tsync.c  
tests posix_api/tcwd.c  
tests posix_api/tposix.c  
tests pseudovfs/vfsfatfs.c  
tests pseudovfs/vfsred.c  
tests stochposix/stochposix.c  
tests stochposix/stochposix_create.c  
tests stochposix/stochposix_dir.c  
tests stochposix/stochposix_file.c  
tests stochposix/stochposix_misc.c  
tests testfw/stochfw.c  
tests testfw/testfw.c  
tests testfw/tfwdate.c  
tests util/atoi.c  
tests util/math.c  
tests util/printf.c  
tests util/rand.c  
tests util/scale.c  
tests util/stringi.c  
tools getopt.c  
tools toolcmn.c
```

`tests/testfw` must be added to the list of include directories.

If your IDE requires header files included with quotation marks to be in an include directory, the following must also be added as include directories:

```
tests/disk_full
tests/fse_api
tests posix
tests posix_api
tests/stochposix
```

9.1.2 Building Reliance Edge with an IDE

If you are using an IDE (Integrated Development Environment) for your project, it might make sense to use it to build Reliance Edge. The exact details will vary, but in most cases this will involve adding the files and include directories listed above to the project or a subproject.

It is highly desirable to preserve the Reliance Edge directory structure, so that you can create and receive patches, and update to newer versions. Some IDEs (like Atmel Studio) will by default make a copy of a file which is added to the project, meaning the directory structure is lost unless effort is made to duplicate it. There are two possible solutions. One is to have a Reliance Edge directory outside of the IDE project, and add the Reliance Edge files as a "link" of some sort rather than copying the file into the project. For example, with the Atmel Studio add existing file dialog, this can be accomplished by clicking the triangle on the Add button and selecting "Add As Link". The second solution is to copy the Reliance Edge directory into the IDE project directory, and maintain the subdirectory structure as files are added to the IDE project.

9.1.3 Building Reliance Edge with a Makefile

Provided you follow certain conventions in your Makefile, you can reuse the build/reliance.mk file, which defines the objects to be built, and provides the rules to build those objects, including header file dependencies. build/reliance.mk defines several variables which are potentially useful in a Makefile, but the two most important are REDDRIVOBJ (list of objects necessary to build the driver); and REDTESTOBJ, the additional objects necessary to build the tests.

For an example, see projects/win32/Makefile. It defines variables for the Reliance Edge product directory (P_BASEDIR), the project directory (P_PROJDIR), the OS name (P_OS), and the object file extension (B_OBJEXT). It then includes build/reliance.mk, and uses the variables defined therein in its build logic.

The Makefile must include a rule for compiling the object files, as well as logic to supply the necessary include directories and compiler flags.

The advantage of using build/reliance.mk is that it reduces maintenance, particularly when upgrading to new versions. That said, there is no rule against rewriting the Makefile from scratch. In some cases this might be the most sensible move, such as when integrating with an existing system of Makefiles. Use the information in the [Source Files and Include Directories](#) section to figure out which files to build; and, if not automatically generating the header dependencies, consult build/reliance.mk to find them.

9.1.4 Linking and Building as a Library

The simplest approach to linking is to build all the source code in the system—the RTOS, the application, and Reliance Edge—at the same time, and let the compiler link everything together at the end. For some applications, this might involve adding Reliance Edge and RTOS code to the build process used by the application. Or some RTOSes themselves have build systems into which applications integrate, in which case Reliance Edge can be likewise integrated.

Another approach is to compile the Reliance Edge source files into object files (which typically have a *.o or *.obj extension), and then use a tool like GCC libtool to bundle those object files into a library, which can be linked to the application and RTOS later. Makefiles generally create the necessary object files anyway, so adapting this approach just involves changing how the object files are used. Most IDEs can be made to compile Reliance Edge as a library, but the method varies.

9.2 Building the Host Tools

Each project has a "host" subdirectory from which the host tools should be compiled. This host subproject inherits configuration settings from the parent project; the tools will not work correctly unless they are using the same configuration settings as the driver, so it is important to use the host subdirectory from your project.

The host tools can be compiled from the same folder on either Windows or Linux using a compatible make utility and compiler.

Building the host tools on Windows

On Windows, the easiest way to compile the host tools is to use GNU make and Visual Studio. The Makefiles supplied with Reliance Edge are written to work with GNU make, for portability reasons. Other implementations of make are not supported and might be incompatible; in particular, Microsoft nmake is known to be incompatible. GNU make for Windows can be obtained [from sourceforge.net](http://sourceforge.net). It is recommended to download and install the setup program, rather than the ZIP files, since the program includes the dependencies. Once GNU make is installed, you want to add it to the PATH (its normal install location is C:\Program Files (x86)\GnuWin32\bin), otherwise the full path to `make.exe` will need to be used on the command line.

Any relatively recent version of Visual Studio will work (Visual Studio 2008 or later), including the free editions, Visual Studio Express or Visual Studio Community. The latter edition is newer and has more features. To obtain a free version of Visual Studio, visit visualstudio.com.

Once Visual Studio is installed, open a Visual Studio Command Prompt. If you need help finding it, consult [this MSDN article](#). (If link goes dead, as MSDN links are prone to do, consult your search engine of choice.) From the Visual Studio Command Prompt, CD (change directory) to the host subproject, and type `make` to compile. Compiler output will be printed to the console, and when complete the host tools executables will have been created. You can use `make clean` to delete the object files and the executables.

Building the host tools on Linux

On Linux, the host tools can be compiled using GNU make and GCC. Simply CD (change directory) to the host subproject, and type `make` to compile.

The FUSE implementation is not built by default because it relies on the package `libfuse-dev`. After installing `libfuse-dev` (e.g. `sudo apt-get install libfuse-dev` on Ubuntu), you can build the FUSE implementation with the command `make redfuse`. You can also build the FUSE implementation and all other available tools by calling `make all`.

9.3 Dealing with Compiler Warnings

Ideally Reliance Edge will compile without warnings, but given the sheer quantity of compilers, compiler versions, and compiler options, you may see compiler warnings. If you determine that a warning indicates a real problem, you may attempt to fix it or report the issue to Tuxera. Otherwise, you can take steps to suppress the warning, either by changing the compilation flags or altering the code. If you make code changes to suppress warnings, consider providing them to Tuxera to be incorporated into the product, so that you do not see the problem again with future versions and to eliminate potential merge conflicts. To be accepted, changes to suppress warnings must be portable and compiler agnostic (no pragmas or compiler extensions); furthermore, if the warning being suppressed is excessively silly (for example, compiling with the GCC `-Wtraditional` flag), your changes will probably not be incorporated.

Chapter 10

POSIX-Like API Guide

The Reliance Edge POSIX-like API is one of two API sets supported by Reliance Edge. It is lightweight, but provides a more complete interface than the [File System Essentials](#) options.

[Here](#) is the documentation for the POSIX-like API calls.

10.1 When to Use the POSIX-Like API

The Reliance Edge POSIX-like API includes several features that are not supported by the more minimal File System Essentials API. The following features are unique to the POSIX-like API:

- The ability to create and delete files at runtime.
- The ability to access files by name and create multiple hard links to a file if needed.
- The ability to query for file information, such as date modified.
- Use of file descriptors as an interface to access files.

If any feature listed above is a key requirement of your project, then the POSIX-like API should be used. The sections below provide some examples of where the POSIX-like API may be preferable.

Porting existing code

If an existing embedded software project uses POSIX calls to interface with the file system, then it will likely be much easier to port the project to Reliance Edge if the POSIX-like API is used. The [Porting POSIX Apps](#) section gives guidance to help with this process.

Porting to a new RTOS

When Reliance Edge is ported to an operating system, care must be taken in adapting to the requirements of the OS. If the OS expects POSIX-like functionality from the underlying file system, then the POSIX-like API may be the only reasonable option.

Using files across multiple processes

The POSIX-like API allows files to be referenced by file names. This could provide a safer way for applications to share Reliance Edge files by forcing the applications to name which files they want to access.

Additionally, the ability for the applications to create and delete files provides stability by returning an error if a read request is done on an obsolete file.

10.2 Path Prefixes

Reliance Edge can be used as a file system for multiple volumes. When using the POSIX-like API, each volume must be assigned a unique path prefix, set in redconf.c.

- There is no limit to the length of the path prefix.
- One volume and no more may be given an empty string as a path prefix. If a volume is assigned an empty path prefix, that volume is considered to be the default volume. Absolute file paths which do not begin with another path prefix are assumed to refer to this volume. A path separator character, set by [REDCONF_PATH_SEPARATOR](#), is optional at the beginning of a path on the default volume.
- The path prefix may contain path separator characters, and may overlap other path prefixes. For example, if the prefix of the first volume is "foo", the prefix of the second volume may be "foo/bar" if desired. If a path matches multiple prefixes, the longest match is selected; thus "foo/bar/data/c.txt" will match the "foo/bar" volume rather than the "foo" volume. Care must be taken with overlapping prefixes of this type: for example, if a "bar" directory is created on the "foo" volume, it will be inaccessible since the path would be matched to the "foo/bar" volume.

10.3 Path Strings

The POSIX-like API is provided to access file data through file names and paths. If [REDCONF_API_POSIX_CWD](#) is true, Reliance Edge supports a per-task current working directory and will parse dot and dot-dot components in paths. If [REDCONF_API_POSIX_CWD](#) is false, all paths must be absolute and fully-qualified.

The format of an absolute path (one which is not relative to the CWD) is as follows:

- The first element of a path is a volume path prefix.
- The path prefix is followed by a path separator character ([REDCONF_PATH_SEPARATOR](#)).
- Further elements must follow the standard convention: a directory name precedes the name of the desired file or directory it contains, separated by a path separator.
- Multiple adjacent path separator characters are allowed. They are treated as identical to a single path separator.
- Any trailing path separators at the end of the path are ignored. This applies to paths that name files or directories.

As a special case, a volume path prefix by itself is interpreted as a reference to the root directory of that volume, just as if the path prefix had been followed by a path separator. Thus, if there is a volume which uses an empty string as a path prefix, then an empty string will be a valid path, referring to the root directory of the default volume. If no volume uses an empty string as a path prefix, then an empty string is always an invalid path, eliciting a [RED_ENOENT](#) error, as specified by POSIX.

10.3.1 Current Working Directory Support

If `REDCONF_API_POSIX_CWD` is true, Reliance Edge also supports relative paths, which are parsed relative to the current working directory (CWD). A path is treated as a relative path if it does not start with a volume path prefix or a path separator. Parsing of relative paths starts at the CWD set by `red_chdir()`; if that function has never been used, the default CWD is the root directory of volume zero; furthermore, if the CWD is on a volume which is unmounted, it resets to the root directory of the unmounted volume.

A directory which is the CWD for any task cannot be deleted; any attempt to do so, via `red_unlink()`, `red_rmdir()`, or `red_rename()`, will result in an `RED_EBUSY` error.

A directory which is a CWD can be renamed or moved, in which case the CWD path changes along with the directory. For example, assuming the path separator is a forward slash and the volume prefix is an empty string, if the CWD is `"/foo/bar"` and that directory is renamed to `"/foo/abc"`, then the CWD is implicitly updated to be `"/foo/abc"`. The CWD is similarly updated if any of its parent directories are renamed or moved. So, if the CWD is `"/foo/bar"` and `"/foo"` is renamed to `"/abc"`, then the CWD becomes `"/abc/bar"`.

The CWD is stored for each task as a volume number and directory inode number, rather than as a path string. Enabling the feature thus consumes only a small amount of additional RAM.

10.3.1.1 Dot and Dot-Dot

When `REDCONF_API_POSIX_CWD` is true, Reliance Edge also supports dot `(".")` and dot-dot `("..")` path components. For example, assuming the path separator is a forward slash, `"a/b/c/.."` and `"a/b/."` both resolve to `"a/b"`.

A dot-dot in the root directory resolves to the root directory.

A path whose last component is dot or dot-dot (like `"a/b/c/.."` or `"a/b/."`) and whose second-to-last component is a directory is allowed for `red_open()` (without `RED_O_CREAT`), `red_opendir()`, and `red_chdir()`; but disallowed for `red_open()` (with `RED_O_CREAT`), `red_unlink()`, `red_mkdir()`, `red_rmdir()`, `red_rename()` (both paths), and `red_link()` (both paths).

Although dot and dot-dot are supported in path parsing, Reliance Edge does not store dot or dot-dot directory entries. `red_readdir()` will never return an entry for dot or dot-dot. This is consistent with POSIX, which allows, but does not require, directory entries for dot and dot-dot.

When `REDCONF_API_POSIX_CWD` is false, Reliance Edge does not parse dot or dot-dot; however, to avoid confusion, the file system disallows creating a file or directory named `".` or `.."`. This restriction was not enforced in earlier versions of Reliance Edge; an old file or directory named `".` or `.."` can, when `REDCONF_API_POSIX_CWD` is false, still be accessed or renamed to something else.

10.3.2 Path Limits

Reliance Edge imposes no limit to the length of a path string. However, the image builder and image copier utilities do enforce path length limits based on Windows or Linux limitations. If a Reliance Edge volume contains files with abnormally long paths, the image copier may encounter an error and abort when called.

Reliance Edge imposes no limit on the directory depth of a path. However, deep directory structures may make traversing difficult because the number of open files and directories cannot exceed the set `REDCONF_HANDLE_COUNT`.

The path prefix and other path components may use any UTF-8 characters, but the path separator character must be a single byte, standard ASCII character.

The maximum length of a file or directory name is set by `REDCONF_NAME_MAX`.

10.4 File Descriptor Features

The POSIX-like API uses file descriptors as handles for both files and directories. The following are properties of file descriptors in Reliance Edge.

- Multiple file descriptors may be open on the same file, including multiple file descriptors open for writing.
- The maximum number of open file descriptors is set by [REDCONF_HANDLE_COUNT](#).
- A file with one link cannot be unlinked if it has an open file descriptor.
- File descriptor values never set the high bit (bit 31), so a file descriptor will never be a negative number. File descriptors are also never 0, 1, or 2, to avoid confusion with POSIX I/O file descriptors (STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO).
- File descriptors encode a "mount generation" to guard against staleness. Thus, if a file descriptor is closed and the volume is unmounted and mounted again, if that closed file descriptor number is erroneously saved and used later, it will automatically be recognized as invalid even as other file descriptors are opened. This feature is available only for file descriptors, not directory streams.

10.5 Errno and Task Count

The macro [red_errno](#) is used to represent the most recent error encountered while performing POSIX-like API calls. It is set to the values of one of the error macros defined in [rederrno.h](#).

The macro [red_errno](#) expands to a dereferenced pointer to the errno variable for the current task. As such, it can be used as an lvalue or rvalue; that is, it may be assigned a number as well as being evaluated. This allows a user to clear the errno value before an API call and check afterwards to see if it has been changed. The [red_readdir\(\)](#) function requires this behavior in order to check for errors.

The POSIX-like API has a maximum on how many simultaneous file system users are allowed. This maximum is set by [REDCONF_TASK_COUNT](#).

The API calls use a mutex to ensure that no concurrent operations are allowed that would put data at risk. If [REDCONF_TASK_COUNT](#) is set to 1, then the mutex is disabled because no concurrent use is allowed.

10.6 Differences from POSIX

The Reliance Edge POSIX-like API differs from POSIX in numerous particulars. Most of these differences are intended to make the POSIX-like API more suited to small embedded systems than the full POSIX file system API.

10.6.1 General Differences

The following list summarizes the most significant general deviations of the Reliance Edge POSIX-like API from the POSIX file system API specifications.

- Function names, type names, and macro names used by Reliance Edge are prefixed with `RED` or `red`. Names may also be modified to follow Reliance Edge naming conventions. For example, the POSIX-like API's equivalent to the POSIX `struct stat` is named [REDSSTAT](#).
- Except for ASCII characters and strings, the Reliance Edge code uses C99-style fixed width integer types such as `int32_t` for integral values, in conformance with the MISRA-C:2012 guidelines. These are used in place of POSIX integer types such as `size_t`.

- Reliance Edge does not store or enforce file permissions or ownership.
- Symbolic links are not supported by Reliance Edge.
- Non-blocking I/O is not supported by Reliance Edge.
- The support for current working directories (CWD), along with parsing of dot and dot-dot components in paths, is optional in Reliance Edge, enabled by [REDCONF_API_POSIX_CWD](#). If that macro is false, all file paths given to any POSIX-like API call must be absolute paths.
- The support of timestamps is optional in Reliance Edge. Timestamps are enabled by [REDCONF_INODE_TIMESTAMPS](#). The atime timestamp is enabled separately by [REDCONF_ATIME](#). When enabled, timestamps are set according to POSIX specifications. Otherwise they are not set and the timestamp fields are not included in the [REDSSTAT](#) structure.
- The support of file block count is optional in Reliance Edge, and is enabled by [REDCONF_INODE_BLOCKS](#). If the block count is not enabled, then the field [REDSSTAT::st_blocks](#) is not included in the [REDSSTAT](#) structure.
- The POSIX-like API allows only a limited number of file system users, set by [REDCONF_TASK_COUNT](#). If a task which is not a file system user calls a Reliance Edge API, and there are no more task slots available, the call will fail with a [RED_EUSERS](#) error.
- Many of the functions in the POSIX-like API may be configured away to save resources. If the corresponding [REDCONF_API_POSIX_\[API call name\]](#) macro is set to false, then the API call will not be available.
- The POSIX-like API uses some different semantics from POSIX in processing file paths. For example, Reliance Edge ignores trailing path separator characters in a path, where POSIX only accepts a trailing path separator on a directory name. See [Path Strings](#) for more information.

10.6.2 Final Dot or Dot-Dot Errno

When [REDCONF_API_POSIX_CWD](#) is true, and the second-to-last path component is a directory and the last path component is a dot or dot-dot, the APIs listed below will fail and set [red_errno](#) to [RED_EINVAL](#). In some cases this differs from POSIX, which either allows a final dot or dot-dot or would return a different errno. These differences exist in order to simplify the implementation.

- [red_open\(\)](#) with [RED_O_CREAT](#): For a final dot or dot-dot, POSIX requires a failure with the errnos [EEXIST](#) or [EISDIR](#), rather than [EINVAL](#).
- [red_unlink\(\)](#): POSIX does not specify the behavior of unlinking directories, including dot or dot-dot.
- [red_mkdir\(\)](#): For a final dot or dot-dot, POSIX requires a failure with the errno [EEXIST](#), rather than [EINVAL](#).
- [red_rmdir\(\)](#): For a final dot, POSIX requires a failure with the errno [EINVAL](#), which is what Reliance Edge does. For a final dot-dot, POSIX requires a failure with the errno [ENOTEMPTY](#), rather than [EINVAL](#).
- [red_rename\(\)](#): For a final dot or dot-dot in the source path, or a final dot in the destination path, POSIX would succeed unless the paths were erroneous for some other reason. (The Linux implementation of `rename()` fails with `EBUSY` in the previous cases, which POSIX allows for.) For a final dot-dot in the destination path, POSIX requires a failure with [ENOTEMPTY](#) rather than [EINVAL](#).
- [red_link\(\)](#): For a final dot or dot-dot in the destination path, POSIX requires a failure with the errno [EEXIST](#), rather than [EINVAL](#).

10.6.3 Individual APIs

This section reviews each function member of the POSIX-like API and describes pertinent differences between the function and the POSIX equivalent. For each function with a POSIX equivalent, a list of the possible `red_errno` values is presented. Any error conditions that are not specified by POSIX are explained. Any POSIX-defined error that is not listed here will not be thrown by the function, even if the specified error condition is encountered in the function.

Not POSIX defined

The following functions are implemented as part of the Reliance Edge POSIX-like API, though equivalent functions are not specified by POSIX.

- `red_init()`
- `red_uninit()`
- `red_mount()`
- `red_mount2()`
- `red_umount()`
- `red_umount2()`
- `red_format()`
- `red_transact()`
- `red_settransmask()`
- `red_gettransmask()`
- `red_errnoptr()`

These functions are extensions to the POSIX API which provide for file system functionality not covered by POSIX (like mounting and unmounting) and to enable unique Reliance Edge capabilities (like transaction points).

sync

```
int32_t red_sync(void);
```

A distinguishing feature of Reliance Edge is that data written to the disk remains in an uncommitted state until a transaction is done. This protects the system from data corruption on power failure. For this reason, uncommitted data is effectively identical to unsynchronized data in memory. Because of this, the `red_sync()` function does not flush dirty data for a given volume unless that volume is configured to perform an automatic transaction (that is, unless the transaction flag `RED_TRANSACT_SYNC` is set). If sync automatic transactions have been disabled on all volumes, this function does nothing and returns success. This is a deviation from the POSIX standard.

`red_sync()` returns a value indicating success/fail, and sets `red_errno` on failure. This is a deviation from the POSIX standard, which does not allow for a return value, nor errors.

statvfs

```
int32_t red_statvfs(const char *pszVolume, REDSTATFS *pStatvfs);
```

The argument `pszVolume` must point to a string of characters representing the path prefix of the volume to query. This deviates from POSIX, which specifies that the first argument can be the path to any file on the desired volume.

red_errno values

- **RED_EINVAL:** Volume is not mounted, or one of the supplied arguments is null. This deviates from the POSIX standard, which does not allow the EINVAL errno.
- **RED_ENOENT:** `pszVolume` is not a valid volume path prefix.
- **RED_EUSERS:** Cannot become a file system user: too many users. This deviates from the POSIX standard, which does not allow the EUSERS errno.

RESTATFS

This structure conforms to the POSIX requirements for a `statvfs` structure. The fields `RESTATFS::f_maxsize` and `RESTATFS::f_dev` are added to report the maximum file size and the volume number of a Reliance Edge volume.

open

```
int32_t red_open(const char *pszPath, uint32_t ulOpenMode)
```

The argument `ulOpenMode` must contain exactly one of the file access mode flags `RED_O_RDONLY`, `RED_O_WRONLY`, or `RED_O_RDWR`. The POSIX access mode flags `O_EXEC` and `O_SEARCH` are not supported.

The following combinable POSIX open flags are also not supported: `O_CLOEXEC`, `O_DIRECTORY`, `O_DSYNC`, `O_NOCTTY`, `O_NOFOLLOW`, `O_NONBLOCK`, `O_RSYNC`, `O_SYNC`, and `O_TTY_INIT`.

The `RED_O_TRUNC` flag is only available if `REDCONF_API_POSIX_FTRUNCATE` is set to 1 (true).

If the volume or file system is read only, `ulOpenMode` must equal `RED_O_RDONLY`. This behavior deviates from POSIX.

Unlike the POSIX equivalent, `red_open()` does not support an optional third argument.

Upon successful completion, the function returns a file descriptor. The returned file descriptor is not guaranteed to be the lowest-numbered file descriptor not currently open for the calling process. This is a deviation from the POSIX specification.

red_errno values

- **RED_EEXIST**
- **RED_EINVAL:** `ulOpenMode` is invalid; or `pszPath` is NULL; or the volume containing the path is not mounted; or `REDCONF_API_POSIX_CWD` is true, and `RED_O_CREAT` is included in `ulOpenMode`, and the path ends with dot or dot-dot. Only the first condition is allowed by the POSIX standard.
- **RED_EIO:** A disk I/O error occurred. This deviates from the POSIX standard, which specifies a different use of EIO that is not applicable to Reliance Edge.
- **RED_EISDIR**
- **RED_EMFILE**
- **RED_ENAMETOOLONG**
- **RED_ENFILE:** Attempting to create a file but the file system has used all available inode slots. This deviates from POSIX, which applies ENFILE to the number of open files in the system, not total number of files.
- **RED_ENOENT:** `RED_O_CREAT` is not set and the named file does not exist; or `RED_O_CREAT` is set and the parent directory does not exist; or the volume does not exist; or the `pszPath` argument points to an empty string (and there is no volume with an empty path prefix). This differs slightly from POSIX.

- [RED_ENOSPC](#)
- [RED_ENOTDIR](#): A component of the prefix in `pszPath` does not name a directory. Unlike POSIX, this error is not encountered when the path to a file contains a trailing path separator.
- [RED_EROFS](#): The path resides on a read-only file system and a write operation was requested.
- [RED_EUSERS](#): Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the `errno EUSERS`.

unlink

```
int32_t red_unlink(const char *pszPath)
```

Unlike POSIX unlink, deleting a file or directory with open handles (file descriptors or directory streams) will fail with an [RED_EBUSY](#) error. This only applies when deleting an inode with a link count of one; if a file has multiple names (hard links), all but the last name may be deleted even if the file is open.

Unlike POSIX unlink, this function can fail when the disk is full. To fix this, transact and try again: Reliance Edge guarantees that it is possible to delete at least one file or directory after a transaction point. If disk full automatic transactions are enabled, this will happen automatically.

This function may be used to unlink an empty directory. POSIX leaves the definition of this behavior to the implementation.

[red_errno](#) values

- [RED_EBUSY](#)
- [RED EINVAL](#): `pszPath` is `NULL`; or the volume containing the path is not mounted; or [REDCONF_API_POSIX_CWD](#) is true and the path ends with dot or dot-dot. This deviates from POSIX, which does not allow the `EINVAL` `errno`.
- [RED_EIO](#): A disk I/O error occurred. This deviates from POSIX, which does not allow the `EIO` `errno`.
- [RED_ENAMETOOLONG](#)
- [RED_ENOENT](#)
- [RED_ENOTDIR](#)
- [RED_ENOTEMPTY](#): The path names a directory which is not empty. This deviates from POSIX, which does not allow the `ENOTEMPTY` `errno`.
- [RED_ENOSPC](#): The file system does not have enough space to modify the parent directory to perform the deletion. This deviates from POSIX, which does not allow the `ENOSPC` `errno`.
- [RED_EUSERS](#): Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the `errno EUSERS`.

mkdir

```
int32_t red_mkdir(const char *pszPath)
```

Unlike POSIX mkdir, this function has no second argument for the permissions (which Reliance Edge does not use).

red_errno values

- [RED_EEXIST](#)
- [RED_EINVAL](#): `pszPath` is `NULL`; or the volume containing the path is not mounted; or [REDCONF_API_POSIX_CWD](#) is true and the path ends with dot or dot-dot. This deviates from POSIX, which does not allow the `EINVAL` errno.
- [RED_EIO](#): A disk I/O error occurred. This deviates from POSIX, which does not allow the `EIO` errno.
- [RED_ENAMETOOLONG](#)
- [RED_ENOENT](#)
- [RED_ENOSPC](#)
- [RED_ENOTDIR](#)
- [RED_ERofs](#)
- [RED_EUSERS](#): Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the `errno EUSERS`.

rmdir

```
int32_t red_rmdir(const char *pszPath)
```

Unlike POSIX `rmdir`, deleting a directory with open handles (file descriptors or directory streams) will fail with an [RED_EBUSY](#) error.

If [REDCONF_API_POSIX_CWD](#) is false, then the operation does not check for a closing dot or dot-dot component at the end of the given path; this is a deviation from POSIX.

Unlike POSIX `unlink`, this function can fail when the disk is full. To fix this, transact and try again: Reliance Edge guarantees that it is possible to delete at least one file or directory after a transaction point. If disk full automatic transactions are enabled, this will happen automatically.

red_errno values

- [RED_EBUSY](#)
- [RED_EINVAL](#): `pszPath` is `NULL`; or the volume containing the path is not mounted; or [REDCONF_API_POSIX_CWD](#) is true and the path ends with dot or dot-dot. This deviates from the POSIX standard, which specifies a different use of `EINVAL` that is not applicable to Reliance Edge.
- [RED_EIO](#)
- [RED_ENAMETOOLONG](#)
- [RED_ENOENT](#)
- [RED_ENOTDIR](#)
- [RED_ENOTEMPTY](#)
- [RED_ENOSPC](#): The file system does not have enough space to modify the parent directory to perform the deletion. This deviates from POSIX, which does not allow the `ENOSPC` errno.
- [RED_ERofs](#)
- [RED_EUSERS](#): Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the `errno EUSERS`.

rename

```
int32_t red_rename(const char *pszOldPath, const char *pszNewPath)
```

If `pszNewPath` names an existing file or directory, the behavior depends on the configuration. If `REDCONF_RENAME_ATOMIC` is false, and if the destination name exists, this function always fails and sets `red_errno` to `RED_EEXIST`. This behavior deviates from POSIX.

Unlike POSIX rename, if `pszNewPath` points to an inode with a link count of one and open handles (file descriptors or directory streams), the rename will fail with `RED_EBUSY`.

If `REDCONF_API_POSIX_CWD` is false, then the operation does not check for a closing dot or dot-dot component at the end of the given path; this is a deviation from POSIX.

`red_errno` values

- `RED_EBUSY`
- `RED_EEXIST`: `REDCONF_RENAME_ATOMIC` is false and `pszNewPath` exists. This deviates from POSIX, which allows `EEXIST` as a synonym for `ENOTEMPTY`.
- `RED_EINVAL`: `pszOldPath` is `NULL`; or `pszNewPath` is `NULL`; or the volume containing the path is not mounted; or `REDCONF_API_POSIX_CWD` is true and either path ends with dot or dot-dot. This deviates from POSIX, which uses `EINVAL` to report other errors.
- `RED_EIO`
- `RED_EISDIR`
- `RED_ENAMETOOLONG`
- `RED_ENOENT`
- `RED_ENOTDIR`: A component of either path prefix is not a directory; or `pszOldPath` names a directory and `pszNewPath` names a file. Unlike POSIX, this error is not encountered when the path to a file contains a trailing path separator.
- `RED_ENOTEMPTY`
- `RED_ENOSPC`
- `RED_EROFS`
- `RED_EUSERS`: Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the `errno` `EUSERS`.
- `RED_EXDEV`: `pszOldPath` and `pszNewPath` are on different file system volumes. This behavior is left by POSIX for the implementation to define.

link

```
int32_t red_link(const char *pszPath, const char *pszHardLink)
```

The path given by `pszPath` must name a file. Hard linking directories is not supported. POSIX leaves the implementation to define whether hard linking directories is supported.

red_errno values

- [RED_EEXIST](#)
- [RED_EINVAL](#): `pszPath` or `pszHardLink` is `NULL`; or the volume containing the paths is not mounted. This deviates from the POSIX standard, which specifies a different use of `EINVAL` that is not applicable to Reliance Edge.
- [RED_EIO](#): A disk I/O error occurred. This deviates from POSIX, which does not allow the `errno EIO`.
- [RED_EMLINK](#): Creating the link would exceed the maximum link count of the inode named by `pszPath`. This varies slightly from POSIX specifications.
- [RED_ENAMETOOLONG](#)
- [RED_ENOENT](#)
- [RED_ENOSPC](#)
- [RED_ENOTDIR](#): A component of either path prefix is not a directory. Unlike POSIX, this error is not encountered when the path to a file contains a trailing path separator.
- [RED_EPERM](#): The `pszPath` argument names a directory. This behavior is left for the implementation to define in POSIX.
- [RED_EROFS](#)
- [RED_EUSERS](#): Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the `errno EUSERS`.
- [RED_EXDEV](#): `pszOldPath` and `pszNewPath` are on different file system volumes. This behavior is left by POSIX for the implementation to define.

close

```
int32_t red_close(int32_t iFildes)
```

Unlike POSIX, `red_close` never results in the freeing of a file with no links, because Reliance Edge does not support removing the last link to an open file.

red_errno values

- [RED_EBADF](#)
- [RED_EIO](#)
- [RED_EUSERS](#): Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the `errno EUSERS`.

read

```
int32_t red_read(int32_t iFildes, void *pBuffer, uint32_t ulLength)
```

The read takes place at the file offset associated with `iFildes` and advances the file offset by the number of bytes actually read. This is the POSIX specified behavior for files that support seeking, and applies to all files within Reliance Edge.

If the requested read length is greater than `INT32_MAX`, the read fails with an `RED_EINVAL` `errno`. This type of behavior is left by POSIX to be implementation defined.

POSIX allows implementations to define whether a `read` call is allowed on directories. This behavior is not supported by Reliance Edge, and a `red_read` request with a file descriptor which refers to a directory will fail with `RED_EISDIR`.

[red_errno](#) values

- [RED_EBADF](#)
- [RED_EINVAL](#): pBuffer is NULL; or ulLength exceeds INT32_MAX and cannot be returned properly. This deviates from the POSIX standard, which specifies a different use of EINVAL that is not applicable to Reliance Edge.
- [RED_EIO](#)
- [RED_EISDIR](#): The iFildes is a file descriptor for a directory. This behavior is left by POSIX for the implementation to define.
- [RED_EUSERS](#): Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the errno EUSERS.

write

```
int32_t red_write(int32_t iFildes, const void *pBuffer,
                  uint32_t ulLength)
```

The write takes place at the file offset associated with iFildes and advances the file offset by the number of bytes actually written. Alternatively, if iFildes was opened with [RED_O_APPEND](#), the file offset is set to the end-of-file before the write begins, and likewise advances by the number of bytes actually written. This is the POSIX specified behavior for files that support seeking, and applies to all files within Reliance Edge.

If the requested write length is greater than INT32_MAX, the write fails with an [RED_EINVAL](#) errno. This type of behavior is left by POSIX to be implementation defined.

[red_errno](#) values

- [RED_EBADF](#)
- [RED_EFBIG](#)
- [RED_EINVAL](#): pBuffer is NULL; or ulLength exceeds INT32_MAX and cannot be returned properly. This deviates from the POSIX standard, which specifies a different use of EINVAL that is not applicable to Reliance Edge.
- [RED_EIO](#)
- [RED_ENOSPC](#)
- [RED_EUSERS](#): Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the errno EUSERS.

fsync

```
int32_t red_fsync(int32_t iFildes)
```

A distinguishing feature of Reliance Edge is that data written to the disk remains in an uncommitted state until a transaction is done. This protects the system from data corruption on power failure. For this reason, uncommitted data is effectively identical to unsynchronized data in memory. Because of this, the [red_fsync](#) function does not flush dirty file data unless it is configured to perform an automatic transaction (that is, unless the transaction flag [RED_TRANSACT_FSYNC](#) is set). If fsync automatic transactions have been disabled, this function does nothing and returns success. This is a deviation from the POSIX standard.

In the current implementation, this function has global effect. All dirty buffers are flushed and a transaction point is committed. Fsyncing one file effectively fsyncs all files. However, applications written for portability should avoid assuming this behavior.

red_errno values

- [RED_EBADF](#)
- [RED_EIO](#)
- [RED_EUSERS](#): Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the `errno` EUSERS.

lseek

```
int64_t red_lseek(int32_t iFildes, int64_t lloffset,  
                   REDWHENCE whence)
```

red_errno values

- [RED_EBADF](#)
- [RED_EINVAL](#)
- [RED_EIO](#): A disk I/O error occurred. This deviates from POSIX, which does not allow the `EIO` `errno`.
- [RED_EISDIR](#): The `iFildes` argument is a file descriptor for a directory. This deviates from POSIX, which does not allow the `EISDIR` `errno`.
- [RED_EUSERS](#): Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the `errno` EUSERS.

REDWHENCE

This enum defines [RED_SEEK_SET](#), [RED_SEEK_CUR](#), and [RED_SEEK_END](#), which correspond to the POSIX macros SEEK_SET, SEEK_CUR, and SEEK_END.

ftruncate

```
int32_t red_ftruncate(int32_t iFildes, uint64_t ullSize)
```

Unlike POSIX ftruncate, this function can fail when the disk is full if `ullSize` is non-zero. When decreasing the file size, this can be fixed by transacting and trying again: Reliance Edge guarantees that it is possible to perform a truncate of at least one file that decreases the file size after a transaction point. If disk full transactions are enabled, this will happen automatically.

red_errno values

- [RED_EBADF](#)
- [RED_EFBIG](#)
- [RED_EIO](#)
- [RED_ENOSPC](#): Insufficient free space to perform the truncate. This deviates from POSIX, which does not allow the `errno` ENOSPC.
- [RED_EUSERS](#): Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the `errno` EUSERS.

fstat

```
int32_t red_fstat(int32_t iFildes, REDSTAT *pStat)
```

red_errno values

- [RED_EBADF](#)
- [RED_EINVAL](#): pStat is NULL. This deviates from POSIX, which does not allow the errno EINVAL.
- [RED_EIO](#)
- [RED_EUSERS](#): Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the errno EUSERS.

REDSTAT

This struct is comparable to the POSIX stat structure, but does not include the fields st_uid, st_gid, st_rdev, and st_blksize. Additionally, the struct type fields st_atim, st_mtim, and st_ctim are omitted in favor of integral fields [REDSTAT::st_atime](#), [REDSTAT::st_mtime](#), and [REDSTAT::st_ctime](#) (which were present in the stat structure in earlier versions of the POSIX standard).

Some of the members of this structure are not defined or set if the Reliance Edge project configuration makes them inapplicable. For example, the setting [REDCONF_INODE_BLOCKS](#) specifies whether the field [REDSTAT::st_blocks](#) should be included.

opendir

```
REDDIR *red_opendir(const char *pszPath)
```

The type [REDDIR](#) is implemented using a file descriptor so the number of open directories and files must not exceed the number of available file descriptors, as set by [REDCONF_HANDLE_COUNT](#). This is comparable to the POSIX specification.

red_errno values

- [RED_EINVAL](#): pszPath is NULL; or the volume containing the path is not mounted. This deviates from POSIX, which does not allow the errno EINVAL.
- [RED_EIO](#): A disk I/O error occurred. This deviates from POSIX, which does not allow the EIO errno.
- [RED_ENOENT](#)
- [RED_ENOTDIR](#)
- [RED_EMFILE](#)
- [RED_EUSERS](#): Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the errno EUSERS.

readdir

```
REDDIRENT *red_readdir(REDDIR *pDirStream)
```

If files are added to the directory referred to by pDirStream after it is opened, the new files may or may not be returned by this function. If files are deleted, the deleted files will not be returned. Behavior in these scenarios is explicitly unspecified by POSIX.

red_errno values

- **RED_EBADF**
- **RED_EIO**: A disk I/O error occurred. This deviates from POSIX, which does not allow the `EIO` errno.
- **RED_EUSERS**: Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the `errno EUSERS`.

rewinddir

```
void red_rewinddir(REDDIR *pDirStream)
```

Since this function (like its POSIX equivalent) cannot return an error, it takes no action in error conditions, such as when `pDirStream` is invalid. Behavior in the case of an invalid argument is explicitly undefined by POSIX.

closedir

```
int32_t red_closedir(REDDIR *pDirStream)
```

red_errno values

- **RED_EBADF**
- **RED_EUSERS**: Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the `errno EUSERS`.

chdir

```
int32_t red_chdir(const char *pszPath)
```

red_errno values

- **RED_EINVAL**: `pszPath` is `NULL`; or the volume containing the path is not mounted. This deviates from POSIX, which does not allow the `errno EINVAL`.
- **RED_EIO**: A disk I/O error occurred. This deviates from POSIX, which does not allow the `EIO` errno.
- **RED_ENAMETOOLONG**
- **RED_ENOENT**
- **RED_ENOTDIR**
- **RED_EUSERS**: Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the `errno EUSERS`.

getcwd

```
char *red_getcwd(char *pszBuffer, uint32_t ulBufferSize)
```

The behavior when the buffer (`pszBuffer`) is `NULL` is explicitly unspecified by POSIX. Most Unix implementations implement an extension where if the buffer is `NULL`, a sufficiently large buffer is dynamically allocated via `malloc()`. Reliance Edge does not use dynamic memory allocation, and thus does not support this extension; it is an error if `pszBuffer` is `NULL`.

red_errno values

- [RED_EINVAL](#)
- [RED_EIO](#): A disk I/O error occurred. This deviates from POSIX, which does not allow the EIO errno.
- [RED_ERANGE](#)
- [RED_EUSERS](#): Cannot become a file system user: too many users. This deviates from POSIX, which does not allow the errno EUSERS.

10.7 Porting POSIX Apps

The [Differences from POSIX](#) notwithstanding, it is often quite feasible to take an existing POSIX application program and update it to run against the Reliance Edge POSIX-like API instead. It becomes more difficult if the application makes extensive use of features Reliance Edge does not support.

The recommended approach is to first replace the includes for the POSIX headers (like `unistd.h` and `sys/stat.h`) with an include for `redposix.h`. Then start replacing all the function calls, macros, and types. After changing the headers, compiling the code will generate errors or warnings for most things that need to change. For example, `open` needs to be replaced by `red_open()`, `O_CREAT` by `RED_O_CREAT`, `struct stat` by `REDSTAT`, and so on.

As an alternate approach, macros and typedefs can do a lot of this work for you. See `tests posix/redposixcompat.h` for an example. This should really only be a temporary solution, particularly for important application code. Using macros and typedefs makes changes automatically that should really be done manually so that the ramifications are fully considered on an individual basis.

A POSIX application which uses C streams (like `fprintf` and `fwrite`) to read or write to the file system will need to be modified to use the `red_read()` and `red_write()` APIs. For something like `fprintf`, a wrapper can print the content to a buffer before calling `red_write()`.

The POSIX file system API is much larger than the POSIX-like API implemented by Reliance Edge, so many POSIX applications will need to be altered to use a different API. For example, Reliance Edge does not implement `unlinkat`, so if an application uses that function, it will need to be modified to work with `red_unlink()` instead. In some cases this may be inconvenient, but it is not practical for an embedded file system like Reliance Edge to implement all the numerous POSIX file system APIs in all their variants.

Chapter 11

File System Essentials API Guide

The Reliance Edge File System Essentials (FSE) is one of two API sets supported by Reliance Edge. It is a minimalistic but reliable alternative to the [POSIX-like option](#).

[Here](#) is the documentation for the FSE API calls.

11.1 Introduction

11.1.1 Characteristics

With the FSE API, files are identified by a number, rather than a path and file name. These files can be read, written, and truncated. All available files are created when the volume is formatted and are never deleted.

11.1.2 Benefits

Many small systems today must accomplish sophisticated functions while using few resources. The FSE configuration enables applications to store and retrieve data in many files, just like a full-featured file system, with fewer resources and lines of code.

11.1.3 Selection

Before choosing between the POSIX-like and FSE APIs for your application, consider both options against your requirements. If your application will be written from scratch, you may be able to design in the FSE APIs as your file system solution. However, if your application includes pre-existing components which depend on POSIX file services, it may be prohibitively difficult to adapt your application to the FSE API.

The FSE API is designed for use cases where each file number can be assigned a fixed purpose: for example, file 2 is a log of data type A, file 3 is a log of data type B, file 4 is a database for purpose C, and so on. If the file system requirements are more dynamic—where files are created and deleted, and the number of files used for a particular purpose is variable—the POSIX-like API might be a better choice.

The FSE API is better suited to use cases with a relatively small number of files. Each file is identified by a number, which in practice will be a macro named to identify the file; if there are too many files, managing all the macros can become tedious. The point at which the FSE API becomes tedious depends on how good you are at naming and organizing macros, but as a rule of thumb if you have hundreds of files, the POSIX-like API is probably more appropriate.

11.2 Managing File Numbers

The general notion is that rather than hard-coding file numbers in your application, a macro will be used to give each file number a symbolic name. [redfse.h](#) defines a macro, `RED_FILENO_FIRST_VALID`, for the first valid file number, which can be used in defining the file macros, as in the below example:

```
#define LOG_FILE      (RED_FILENO_FIRST_VALID)
#define DATABASE_FILE (RED_FILENO_FIRST_VALID + 1U)
#define ICON1_FILE    (RED_FILENO_FIRST_VALID + 2U)
#define ICON2_FILE    (RED_FILENO_FIRST_VALID + 3U)
```

Then these macros would be used to access the corresponding files:

```
written = RedFseWrite(0, LOG_FILE, logSize, captureLen, pCaptureData);
```

The [Image Builder](#) is capable of producing a set of macros for you, though you are free to change the names.

11.3 Limitations

Several features of the POSIX-like configuration are not included with FSE, including:

- Handles
- Names and Directories
- Creation and Deletion
- Querying File Attributes (fstat)

11.3.1 No Handles

There is no concept of handles in the interface, and files are never "opened" or "closed". Instead, files are accessed via a unique number. Any file can be accessed at any time. Because there are no file handles, there is also no tracking of file offset, thus each file read, write and truncate call must specify the file offset for the operation.

11.3.2 No Directories

There are no names or directories in the FSE configuration. Instead, files are identified by unique file number.

11.3.3 No Creation or Deletion

All files are created with zero size during format. File data can be truncated to free up disk space, but the files themselves always exist and are never created or deleted.

11.3.4 No Querying File Attributes (fstat)

The only attribute of a file that can be queried is its size (via `RedFseSizeGet()`). Other values, like timestamps, are not available, but would be available with a POSIX-like API configuration via `red_fstat()`.

Chapter 12

Product Configuration

This chapter describes the process of creating and configuring a Reliance Edge project. The *project* is a directory containing a set of source files which allows you to configure Reliance Edge as required for your use case, compiler, and hardware.

12.1 Creating a Project

Make a copy of the projects/newproj directory; give it a name appropriate for your project. Let's assume you made your copy in projects/foobar. Two of the copied files need to be modified. The first is projects/foobar/redtypes.h, which defines types (mostly fixed-width integers) used by Reliance Edge. Open the file and define the types as is appropriate for your environment and compiler; the [redtypes.h](#) file itself contains documentation with further details.

The second file to modify is projects/foobar/host/Makefile. Open that file and edit the P_PROJDIR variable to point to the directory containing the Makefile (instead of the newproj/host directory it was copied from). The other files in the host subdirectory do not need to be modified. This directory will eventually be used to [build the host tools](#), but the project configuration files need to be put in place before the host tools will build.

After this, the [Configuration Utility](#) will be used to create [redconf.h](#) and [redconf.c](#) files that will be saved into projects/foobar. This is discussed in further detail in the rest of this chapter.

You may add additional files to projects/foobar in order to build the project (like a Makefile). Or if desired, the project can be reorganized to better integrate with an IDE. See [Building the Reliance Edge Driver](#).

12.2 How to Configure Your Project

The officially supported way to create and edit your project configuration files is to use the [Configuration Utility](#) (see that chapter for information on using the utility). The recommended approach is to run the utility and start working through the configuration settings, consulting the [Product Configuration Guide](#) given below for explanations, recommendations, and guidance. The configuration utility can also edit your configuration files after they have been created.

Editing the configuration files by hand is discouraged. The configuration settings are interrelated, and without intimate knowledge of Reliance Edge it is difficult to configure everything correctly without the assistance that the configuration utility provides.

Note

Make sure you have the correct version of the configuration utility for your release of Reliance Edge; otherwise the files produced may not be compatible. See the [configuration utility page](#) for more info.

12.3 Product Configuration Guide

This section provides details on each of the configuration options; it is recommended you read this section with configuration utility running. The settings appear here in the same order as they do in the configuration utility.

12.3.1 Read Only

Where to find it: In the configuration utility, this setting is found under the General tab in the checkbox labeled "Enable readonly configuration". The corresponding macro in `redconf.h` is `REDCONF_READ_ONLY`.

What it means: If enabled, this option disables all write APIs and all the code that exists to support writing to the file system. It will still be possible to read from the file system, but not to write anything. If selected, the configuration utility will automatically gray-out numerous options which are no longer relevant.

Below is the complete list of POSIX-like and FS Essentials APIs which are disabled when this setting is checked:

- `red_format()`
- `red_transact()`
- `red_settransmask()`
- `red_fstrim()`
- `red_unlink()`
- `red_mkdir()`
- `red_rmdir()`
- `red_rename()`
- `red_link()`
- `red_write()`
- `red_fsync()`
- `red_ftruncate()`
- `RedFseFormat()`
- `RedFseWrite()`
- `RedFseTruncate()`
- `RedFseTransMaskSet()`
- `RedFseTransact()`

Guidance: This setting exists to allow Reliance Edge to be built for environments that do not need to write, like bootloaders. Most of the time this should be left unchecked.

12.3.2 Automatic Discards

Where to find it: In the configuration utility, this setting is found under the General tab in the checkbox labeled "Enable automatic discards". The corresponding macro in `redconf.h` is `REDCONF_DISCARDS`.

What it means: When this setting is enabled, Reliance Edge will be built with support for automatically issuing discard operations (also known as trim operations) when blocks are freed.

Guidance: Automatic discards are an optional feature available in the commercial version of Reliance Edge. Automatic discards can improve performance and increase device lifetime, particular with storage devices based on flash memory. See the [Discard Operations](#) chapter for details.

When this option is enabled, there will be an error (which prevents the configuration from being saved) if none of the volumes is on a block device which [supports discards](#).

12.3.3 POSIX-like API Configuration

12.3.3.1 POSIX-like File System API

Where to find it: In the configuration utility, this setting is found under the General tab in the radio button labeled "Use POSIX API". The corresponding macro in `redconf.h` is `REDCONF_API_POSIX`.

What it means: Reliance Edge provides two mutually exclusive file system APIs: the POSIX-like API and the File System Essentials API. This setting indicates your application will be using the POSIX-like API.

Guidance: See the [POSIX-Like API Guide](#) for a discussion of this API and the [API documentation](#) for the list of POSIX-like APIs. Most users will use the POSIX-like API since it is familiar and provides the most features. However, users looking for a smaller and simpler API might consider the File System Essentials API; see the [File System Essentials API Guide](#) for a discussion of when the Essentials API might be appropriate.

12.3.3.2 `red_format()`

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "format". The corresponding macro in `redconf.h` is `REDCONF_API_POSIX_FORMAT`.

What it means: This setting determines whether the `red_format()` API will be available.

Guidance: If you plan to format or reformat the storage media on the target, enable this setting; otherwise, disable it to reduce code size. Often the media is formatted only once, when the system is "flashed", and never again; in this case there is no need for `red_format()`. Enabling `red_format()` will be required to run most of the tests provided with Reliance Edge.

12.3.3.3 `red_link()`

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "link". The corresponding macro in `redconf.h` is `REDCONF_API_POSIX_LINK`.

What it means: This setting determines whether the `red_link()` API will be available.

Guidance: If you plan to create hard links in your application, enable this setting; otherwise, disable it to reduce code size.

12.3.3.4 `red_unlink()`

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "unlink". The corresponding macro in `redconf.h` is `REDCONF_API_POSIX_UNLINK`.

What it means: This setting determines whether the [red_unlink\(\)](#) API will be available.

Guidance: If you plan to delete files in your application, enable this setting; otherwise, disable it to reduce code size. [red_unlink\(\)](#) shares most of its code with [red_rmdir\(\)](#), so the difference in code size is small unless both APIs are enabled or disabled.

12.3.3.5 red_mkdir()

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "mkdir". The corresponding macro in [redconf.h](#) is [REDCONF_API_POSIX_MKDIR](#).

What it means: This setting determines whether the [red_mkdir\(\)](#) API will be available.

Guidance: If you plan to create directories in your application, enable this setting; otherwise, disable it to reduce code size.

12.3.3.6 red_rmdir()

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "rmdir". The corresponding macro in [redconf.h](#) is [REDCONF_API_POSIX_RMDIR](#).

What it means: This setting determines whether the [red_rmdir\(\)](#) API will be available.

Guidance: If you plan to delete directories in your application, enable this setting; otherwise, disable it to reduce code size. [red_rmdir\(\)](#) shares most of its code with [red_unlink\(\)](#), so the difference in code size is small unless both APIs are enabled or disabled.

12.3.3.7 red_rename()

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "rename". The corresponding macro in [redconf.h](#) is [REDCONF_API_POSIX_RENAME](#).

What it means: This setting determines whether the [red_rename\(\)](#) API will be available.

Guidance: If you **need** to rename files or directories in your application, enable this setting; otherwise, disable it to reduce code size and memory usage. [red_rename\(\)](#) substantially increases the minimum number of block buffers (the increase is between 3 and 6 block size buffers, depending on other settings), thereby increasing the amount of RAM needed to use Reliance Edge. Therefore it is recommended to disable this setting unless [red_rename\(\)](#) is actually needed.

12.3.3.8 Atomic Rename

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "atomic rename". The corresponding macro in [redconf.h](#) is [REDCONF_RENAME_ATOMIC](#).

What it means: This setting determines whether [red_rename\(\)](#) will support atomic rename functionality, as described in its documentation.

Guidance: If you want to replace a file with another file, one method is to [red_unlink\(\)](#) the original file and [red_rename\(\)](#) the new file to have the same name as the original file. But there are several things that could go wrong: it leaves an interval where the file does not exist with its original name, and if another task tries to open it, the open will fail; if another task commits a transaction point, a transacted state could be created where the original file does not exist, which could possibly cause problems if power is lost and the file is missing after reboot; or the rename could fail, and the original file would have been deleted without a replacement. Atomic rename allows [red_rename\(\)](#) to rename onto an existing file, removing the need for the [red_unlink\(\)](#) and guaranteeing that the file will exist with either its old or new contents. If

this sounds like a useful feature for your application, enable this setting; otherwise, disable it to reduce code size and decrease the minimum block buffer count by one.

12.3.3.9 `red_ftruncate()` and `RED_O_TRUNC`

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "ftruncate". The corresponding macro in `redconf.h` is `REDCONF_API_POSIX_FTRUNCATE`.

What it means: This setting determines whether truncation functionality will be available, namely the `red_ftruncate()` API and the `RED_O_TRUNC` flag for `red_open()`.

Guidance: If you plan to truncate files in your application, enable this setting; otherwise, disable it to reduce code size.

12.3.3.10 Read Directory APIs

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "readdir". The corresponding macro in `redconf.h` is `REDCONF_API_POSIX_READDIR`.

What it means: This setting determines whether the `red_opendir()`, `red_readdir()`, `red_rewinddir()`, and `red_closedir()` APIs will be available.

Guidance: If you plan to enumerate the contents of a directory in your application, enable this setting; otherwise, disable it to reduce code size and reduce the amount of memory needed for each file system handle (see [Handle Count](#)). `red_readdir()` is useful for discovering the contents of a directory, and for writing code which parses directories recursively, but most embedded applications will not need it.

12.3.3.11 Current Working Directory APIs and Relative Paths

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "relative paths". The corresponding macro in `redconf.h` is `REDCONF_API_POSIX_CWD`.

What it means: This setting determines whether the `red_chdir()` and `red_getcwd()` APIs will be available; and determines whether relative path parsing, including dot and dot-dot handling, will be supported for the other path-based POSIX-like APIs.

Guidance: If you would like to be able to use current working directories, relative paths, or dot and dot-dot path components, enable this setting; otherwise, disable it to reduce code size. With use cases that are more dynamic or involve deeper directory hierarchies, these features can simplify application logic.

12.3.3.12 `red_fstrim()`

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "fstrim". The corresponding macro in `redconf.h` is `REDCONF_API_POSIX_FSTRIM`.

What it means: This setting determines whether the `red_fstrim()` API will be available.

Guidance: The `red_fstrim()` API is only available in the commercial version of Reliance Edge; users of the open source version must leave this setting disabled. `red_fstrim()` allows the application to control when free space is discarded and can be a useful alternative to automatic discards. For details, click on `red_fstrim()` to see its API documentation and review the [Discard Operations](#) chapter. If you are not planning to use `red_fstrim()`, it should be disabled to reduce code size.

12.3.3.13 Maximum Name Length

Where to find it: In the configuration utility, this setting is found under the General tab in the spinbox labeled "Maximum file name length". The corresponding macro in `redconf.h` is `REDCONF_NAME_MAX`.

What it means: This determines the maximum length for a file or directory name; it is equivalent to the `NAME_MAX` macro in POSIX.

Guidance: For efficient, compact directories, this should be kept as small as possible. Directory entries have a fixed size, so all names consume the same amount of directory space as the maximum-length name. If your application code has already been written, it is reasonable to search your application for the longest file name, and use that; or if you previously used a FAT file system without long file names, you can set this to 12 (sufficient for 8.3 names). Internally, directory entry sizes must be a multiple of four; if the maximum name length is not divisible by four, directory entries will be padded with unused bytes. To avoid this padding, use a name length which is divisible by four. The maximum legal name length is block size minus 4, but using such large names is not recommended.

12.3.3.14 Path Separator Character

Where to find it: In the configuration utility, this setting is found under the General tab in the drop-down menu labeled "Path separator character". The corresponding macro in `redconf.h` is `REDCONF_PATH_SEPARATOR`.

What it means: This is the character which will separate names in a path. For example, if set to `/`, then a path like `foo/bar` means `foo` is a directory and `bar` is a file or subdirectory within `foo`.

Guidance: This is usually `/` (for Unix-style paths) or `\` (for DOS- or Windows-style paths); both these options are provided in the drop-down menu. Alternate path separator characters can be entered if "Custom" is selected from the drop-down. For example, you could set the path separator to `:` (like classic Mac OS) or `.` (like VMS or RISC OS). If your application code has been written, it is reasonable to use whatever separator your application is already using. The default choice of `/` has the advantage of being the same as POSIX and unlike `\` it does not need to be escaped in a C string (as in "C:\\foo\\bar" to yield `C:\foo\bar`).

For simplicity reasons, having multiple path separator characters is not supported; you cannot mix `/` and `\` in your application.

If relative path support is enabled, `.` (a period) is not allowed as a path separator, since it would interfere with parsing of dot and dot-dot path components.

12.3.3.15 Handle Count

Where to find it: In the configuration utility, this setting is found under the General tab in the spinbox labeled "Handle count". The corresponding macro in `redconf.h` is `REDCONF_HANDLE_COUNT`.

What it means: This determines the number of file system handles that will be available. This total is shared among all file system volumes. A "handle" is a file descriptor (as returned by `red_open()`) or a directory stream (as returned by `red_opendir()`). Each allocated handle is available for either purpose (file descriptor or directory stream), but not at the same time.

Guidance: The number of handles needs to be sufficient for the needs of your application. For example, if your application never opens more than three handles at once, this can be set to three. There is little harm in having too many handles; the main drawback is wasted memory. Handles are not very big. If the [Read Directory APIs](#) is enabled, each handle is about 40–60 bytes (depending on other settings) plus the [maximum name length](#); if that setting is disabled, the handles are about 16 bytes each.

12.3.4 File System Essentials API Configuration

12.3.4.1 File System Essentials API

Where to find it: In the configuration utility, this setting is found under the General tab in the radio button labeled "Use File System Essentials API". The corresponding macro in `redconf.h` is `REDCONF_API_FSE`.

What it means: Reliance Edge provides two mutually exclusive file system APIs: the POSIX-like API and the File System Essentials API. This setting indicates your application will be using the Essentials API.

Guidance: See the [File System Essentials API Guide](#) for a discussion of this API (including reasons why you might use it instead of the POSIX-like API) and the [API documentation](#) for the list of the Essentials APIs.

12.3.4.2 RedFseFormat()

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "Format". The corresponding macro in `redconf.h` is `REDCONF_API_FSE_FORMAT`.

What it means: This setting determines whether the `RedFseFormat()` API will be available.

Guidance: If you plan to format or reformat the storage media on the target, enable this setting; otherwise, disable it to reduce code size. Often the media is formatted only once, when the system is "flashed", and never again; in this case there is no need for `RedFseFormat()`. Enabling `RedFseFormat()` will be required to run most of the tests provided with Reliance Edge.

12.3.4.3 RedFseTruncate()

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "Truncate". The corresponding macro in `redconf.h` is `REDCONF_API_FSE_TRUNCATE`.

What it means: This setting determines whether the `RedFseTruncate()` API will be available.

Guidance: If you plan to truncate files in your application, enable this setting; otherwise, disable it to reduce code size. Note that since the Essentials API does not support deletion, truncating data is the only way to free space, so most applications will want this enabled.

12.3.4.4 RedFseTransMaskGet()

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "Get transaction mask". The corresponding macro in `redconf.h` is `REDCONF_API_FSE_TRANSMASKGET`.

What it means: This setting determines whether the `RedFseTransMaskGet()` API will be available.

Guidance: If you plan to retrieve the transaction mask at run-time in your application, enable this setting; otherwise, disable it to reduce code size.

12.3.4.5 RedFseTransMaskSet()

Where to find it: In the configuration utility, this setting is found under the General tab beneath the "Enabled operations" label in a checkbox labeled "Set transaction mask". The corresponding macro in `redconf.h` is `REDCONF_API_FSE_TRANSMASKSET`.

What it means: This setting determines whether the `RedFseTransMaskSet()` API will be available.

Guidance: If you plan to change the transaction mask at run-time in your application, enable this setting; otherwise, disable it to reduce code size.

12.3.4.6 Task Count

Where to find it: In the configuration utility, this setting is found under the General tab in a spinbox labeled "Maximum task count". The corresponding macro in `redconf.h` is `REDCONF_TASK_COUNT`.

What it means: This determines the number of file system tasks supported by the driver. It must be large enough for every task which will ever use the file system; not just the number which will use it at any given time. If set to 1, all mutex code is conditioned out and there is no protection against multiple tasks entering the file system.

Guidance: If your system is a simple environment without tasks (only one thread of execution), then this can safely be set to 1. If you know you have (for example) four tasks which will be calling into Reliance Edge, you can set this to four. Note that if you set the task count to 1, all locking is removed; if you do so and multiple tasks erroneously use Reliance Edge, this will go undetected and lead to race conditions which can corrupt the file system. So if your system has multiple tasks, only one of which uses the file system, it might be a good idea to set the task count to 2, to preserve the locking.

If using the POSIX-like API, the task count is also used to allocate an array of per-task information, primarily for `red_errno`. Since there are only so many task slots available, an error will be returned if too many tasks attempt to use Reliance Edge. For example, if the task count is set to four, and five tasks attempt to use Reliance Edge, the fifth task will be denied access. Thus, care should be taken to make sure the task count is accurate. If the task count is set to 1, in addition to removing all locking, `red_errno` is no longer stored on a per-task basis.

12.3.5 Text Output

Where to find it: In the configuration utility, this setting is found under the General tab in the checkbox labeled "Enable text output". The corresponding macro in `redconf.h` is `REDCONF_OUTPUT`.

What it means: This setting determines whether to enable text output, namely `RedOsOutputString()` and its test code front-ends such as `RedPrintf()`.

Guidance: If enabled, the Reliance Edge file system driver will print a sign-on message when first initialized, and print notifications after certain critical errors (such as I/O failures). If these seem like useful features, and `RedOsOutputString()` has been implemented, enable the setting. Test code also uses `RedOsOutputString()` (mainly via the `RedPrintf()` front-end, which is similar to `printf()`), so enabling this setting is necessary to use the tests.

12.3.6 Assertion Handling

Where to find it: In the configuration utility, this setting is found under the General tab in the checkbox labeled "Process asserts". The corresponding macro in `redconf.h` is `REDCONF_ASSERTS`.

What it means: This setting determines whether to process assertions. If enabled, Reliance Edge will include assertions which invoke the user-implemented `RedOsAssertFail()` function when an assertion fails. If disabled, assertions are reduced to no-ops which generate no code.

Guidance: Enable assertions during testing and debugging in order to make sure that the code is not encountering situations it did not expect. Enabling assertions will increase the code size and slightly decrease the performance. Assertions are traditionally disabled for release builds, but this is not required; some engineers advocate leaving assertions enabled, so that problems in the field can be more easily caught and diagnosed. For additional discussion of assertions, see the [assertion handling](#) section of the [Porting Guide](#) chapter.

12.3.7 Block Size

Where to find it: In the configuration utility, this setting is found under the Volumes tab in the drop-down menu labeled "Block size". The corresponding macro in `redconf.h` is `REDCONF_BLOCK_SIZE`.

What it means: The block size is the unit of allocation for file data and metadata. All read and write requests issued against the storage media are always at least a block in size. A single block size is shared by all Reliance Edge volumes.

Guidance: The value chosen for the block size is important due to its impact on memory usage, performance, and file system limits. Smaller block sizes use less memory; while larger block sizes tend to yield better performance and larger maximum file and volume sizes (see the [Limits](#) section for details on those maximums).

Reliance Edge needs a certain number of block buffers in order to operate; as the name implies, each block buffer consumes a block sized chunk of memory. So if the block size is doubled, and the [Block Buffer Count](#) is not modified, the amount of memory used by Reliance Edge will almost double. Thus, given that the number of block buffers cannot be decreased below a certain point, the block size is a key variable dictating how much memory Reliance Edge will use.

The minimum value for the block size is the [sector size](#); to be precise, the block size must be greater than or equal to the sector size of every volume. But setting the block size to a value larger than the sector size can yield improved performance. For example, on many types of managed flash media (such as many SD cards and eMMC devices), the sector size is 512 bytes; but internally, the media uses 2048, 4096, or 8192 byte pages, and writes smaller than that, including 512 byte sector writes, are slow. On such media, using a block size that is at least as large as the internal media page size will improve performance.

There are also disadvantages to using a block size which is too large. If the application writes one byte, this will be increased to a block sized write: and on some types of media, writing a 512-byte block will be faster than a 4096-byte block. Another problem with larger block sizes is internal fragmentation: assuming there are direct pointers in the inode, a one-byte file will consume 512 bytes of disk space if the block size is 512 (for 511 wasted bytes), or 4096 bytes of disk space if the block size is 4096 (for 4095 wasted bytes). So smaller block sizes make better use of disk space, especially if there are lots of small files. For these reasons, in addition to high memory usage, block sizes above 8192 are uncommon.

In summary, setting the block size to the sector size is usually a good starting point; but if using flash media, setting it to the page size (or a decent guess at the page size) will improve performance and is usually worth it unless memory usage is a concern. If performance is not good enough, experimenting with other block sizes is a good starting point.

12.3.8 Volume Configuration

12.3.8.1 Path Prefix

Where to find it: In the configuration utility, this setting is found under the Volumes tab in the text box labeled "Volume name (path prefix)"; it is configured separately for each volume. The corresponding values in redconf.c are the last member (the seventh or eighth member, depending on other settings) of every gaRedVolConf element (.pszPathPrefix).

What it means: This setting names the volume and provides a prefix that is used in paths. For example, if the path prefix is "VOL0:", that string would be used with APIs that operate on a volume (like [red_mount\(\)](#)), and "VOL0:/foo/bar" would be a path on that volume (assuming '/' is the path separator character). Or, if there are two volumes with prefixes "/system" and "/data", then "/system/foo/bar" and "/data/foo/bar" would be paths on those volumes.

Guidance: The exact identifier to use for a path prefix is a matter of preference. An empty string is a valid path prefix; if the system only has one volume, an empty prefix is a good choice since it avoids the need to prefix all the paths. For more discussion of path prefixes, see the [Path Prefixes](#) section.

12.3.8.2 Sector Size

Where to find it: In the configuration utility, this setting is found under the Volumes tab in the drop-down menu labeled "Sector size"; or, alternatively, there is an "Auto" checkbox which disables the drop-down menu. It is configured separately for each volume. The corresponding values in redconf.c are the first member of every gaRedVolConf element (.ulSectorSize).

What it means: This is the sector size of the storage medium underlying the volume being configured. The sector

size is the unit for reads and writes to the storage medium; while the file system allocates in terms of blocks, the block numbers and counts are converted to sector numbers and counts before being passed into the [block device service](#).

Guidance: The correct value here is either "Auto" or the sector size of the storage medium. For example, if the storage medium underlying the volume has 512-byte sectors, this should be set to 512. The most common value here will be 512. Supported values are powers-of-two between 128 and 65536, inclusive.

If "Auto" is selected, the redconf.c will have a placeholder value (#SECTOR_SIZE_AUTO) as its sector size and [RedOsBDevGetGeometry\(\)](#) will be used at run-time to query the sector size. This will work so long as [RedOsBDevGetGeometry\(\)](#) has been implemented, which it is for the pre-ported RTOSes. However, note that "Auto" is unsupported with RAM disks provided as part of Reliance Edge, so for those the sector size must be explicitly specified.

The "Auto" option is valuable in situations where the sector size is not known in advance. For example, all eMMC parts default to a 512-byte sector size but some of them can be configured to switch to a 4096-byte sector size. The "Auto" option would allow the same configuration to work with either sector size.

12.3.8.3 Sector Count

Where to find it: In the configuration utility, this setting is found under the Volumes tab in the spinbox labeled "Volume size"; or, alternatively, there is an "Auto" checkbox which disables the spinbox. The sector count is configured separately for each volume. The corresponding values in redconf.c are the second member of every gaRedVolConf element (.ullSectorCount).

What it means: This is the size of the volume, in units of sectors.

Guidance: Selecting "Auto" is often the correct choice, since it provides more flexibility and avoids the hassle of manually determining the sector count of the storage device. When "Auto" is selected, the redconf.c will have a placeholder value (#SECTOR_COUNT_AUTO) as its sector count and [RedOsBDevGetGeometry\(\)](#) will be used at run-time to query the sector count. This will work so long as [RedOsBDevGetGeometry\(\)](#) has been implemented, which it is for the pre-ported RTOSes. The one downside to using "Auto" is that it may slightly increase the code size, relative to what it would have been if the sector count was explicitly configured. Also note that "Auto" is unsupported for RAM disks provided as part of Reliance Edge, so for those the sector count must be explicitly specified. For a file disks provided as part of Reliance Edge, "Auto" is only supported if the file disk already exists, since the file size will be used to determine the sector count.

Otherwise, to set this sector count explicitly, you need to know the exact size of your storage media. For example, if your medium is an SD card which is *about* 2 GB and you assume the sector count is 4194304, this will result in errors if the SD card is actually 1.86 GB with 3900702 sectors. This information may be available in the datasheet for your storage medium; with SD and eMMC, the sector count can be derived from values in the CSD (Card-Specific Data) register.

If you can connect your storage medium to a Linux machine, `fdisk -lu /dev/FOO` (where FOO is replaced with the device or partition name) will print the sector count. Alternatively, `sfdisk -l -uS /dev/FOO` will also print the same information.

If you can connect your storage medium to a Windows machine, `wmic partition get BlockSize, Name, NumberOfBlocks` will report the size of every partition of every disk. The "NumberOfBlocks" is reported in terms of "BlockSize" (which is usually 512; if this does not match the true sector size of the media, scale the "NumberOfBlocks" value accordingly). The output of `wmic` names each disk by number (e.g., "Disk #6"); you can use the "Disk Management" utility (available from the Control Panel) to figure out which disk numbers corresponds to which drive letters.

Sometimes the reported sizes will be inaccurate, and sectors near the end of the media will fail to read or write with I/O errors. It is prudent to test your media to ensure that all the sectors can be read and written without error (for example, on Linux this could be done with `dd`).

If you tell Reliance Edge that the media has fewer sectors than it really does, things will work correctly—the only drawback is that the unreported sectors will be unusable.

If you are using partitioning on your storage media, then the sector count should be the size of the partition which will underlie the volume, rather than the size of the entire storage device.

12.3.8.4 Sector Offset

Where to find it: In the configuration utility, this setting is found under the Volumes tab in the box labeled "Volume offset"; it is configured separately for each volume. The corresponding values in redconf.c are the second member of every gaRedVolConf element (.ullSectorOffset).

What it means: This is the number of sectors before the partition that contains the Reliance Edge volume. More generically, it is the number of sectors on the block device that should be skipped over when doing I/O to this volume.

Guidance: This value, in combination with the [Sector Count](#), can be used to place multiple file system volumes on the same block device, in manually configured partitions. It can also be used to skip over sectors that are being used for other purposes, such as storing boot code. To set this value correctly, you need to know the exact starting sector of the partition. You must ensure that this offset and the [Sector Count](#) do not result in overlapping partitions; neither the configuration utility nor the driver are capable of detecting such overlap, since they are not aware of which volumes share the same block device.

Reliance Edge does not require that its volumes reside on partitions that are defined in an on-disk partition table, such as an MBR. However, you are free to use an on-disk partition table if it helps use third-party tools or other file systems with the storage medium.

If an on-disk partition table exists, and if you can connect your storage medium to a Linux machine, `fdisk -lu /dev/FOO` (where FOO is replaced with the device or partition name) will print the sector offsets and sector counts of the partitions on `/dev/FOO`. Alternatively, `sfdisk -l -uS /dev/FOO` will print the same information.

If an on-disk partition table exists, and if you can connect your storage medium to a Windows machine, `wmic partition get BlockSize, Name, NumberOfBlocks, StartingOffset` will report the size and starting offset of every partition of every disk. The "NumberOfBlocks" and "StartingOffset" are reported in terms of "BlockSize" (which is usually 512; if this does not match the true sector size of the media, scale the other values accordingly). The output of `wmic` names each disk by number (e.g., "Disk #6"); you can use the "Disk Management" utility (available from the Control Panel) to figure out which disk numbers corresponds to which drive letters.

Note that if the partitioning is handled by your [osbdev.c](#) implementation, or by the RTOS itself, then this value should be set to zero.

This value must be zero for volumes using the RAM disk implementations supplied as part of Reliance Edge. This includes the RAM disks in the Win32, Linux, FreeRTOS, and ARM mbed ports. This is enforced when the RAM disk is opened, as happens during format or mount. The justification is that a nonzero sector offset does not make sense for RAM disks, which cannot be shared or partitioned, so to avoid wasting RAM or complicating the implementation, nonzero sector offsets are disallowed.

If the [Sector Size](#) is set to "Auto", be aware that the sector offset specified here will result in different offsets into the storage device depending on what the run-time sector size is detected to be. For example, a sector offset of 2048 is 1 MB with a 512-byte sector size but is 8 MB with a 4096-byte sector size. If it is important for the volume offset to be at a fixed location, then it would be advisable to hard-code the sector size.

If there are multiple volumes on the same storage device and the sector offset is being used to partition that block device, make sure that the [Sector Count](#) is *not* set to "Auto" for any volume other than the last volume on the block device. When "Auto" is used for the [Sector Count](#), the volume will extend from the sector offset to the end of the storage device; so to avoid volumes with overlapping sector ranges, volumes prior to the final volume must not use an auto-detected sector count.

12.3.8.5 Atomic Sector Write

Where to find it: In the configuration utility, this setting is found under the Volumes tab in the drop-down menu labeled

"Atomic sector write"; it is configured separately for each volume. The corresponding values in redconf.c are the third member of every gaRedVolConf element (.fAtomicSectorWrite).

What it means: This tells the file system driver that when a sector is written to the storage medium underlying the volume, the sector write is atomic. *Atomic* means that if the sector write is interrupted by power failure, after reboot, the sector being written is *absolutely guaranteed* to contain either the old contents or the new contents: in other words, the sector write finishes completely or had no effect at all. On media without atomic sector writes, an interrupted sector write might leave the sector with a mix of old and new data; or it might corrupt the contents of the sector entirely.

Guidance: Reliance Edge does *not* require atomic sector writes; it can operate reliably even if interrupting the sector write corrupts the contents of the sector. However, if atomic sector writes are supported, Reliance Edge can enable stronger error checks at mount time, which might catch media corruption errors that would otherwise go unnoticed and perhaps manifest as problems later on. Most modern storage media support atomic sector writes. If unsure, it is safer to assume atomic sector writes are not supported.

12.3.8.6 Discards Supported by Block Device

Where to find it: In the configuration utility, this setting is found under the Volumes tab in the drop-down menu labeled "Discard (trim) support"; it is configured separately for each volume. If both **Automatic Discards** and **red_fstrim()** are disabled, then it is irrelevant whether the block device supports discards: this option will be grayed-out in the GUI and its boolean is not included in the gaRedVolConf array. Otherwise, this setting will be inserted as the seventh member of every gaRedVolConf element (.fDiscardsSupported) in the file redconf.c.

What it means: This value indicates whether discards are supported by the block device for the volume.

Guidance: If this option is not grayed-out (i.e., if either automatic discards or **red_fstrim()** have been enabled), this setting should reflect whether discards are supported by the underlying block device for the volume. If this setting is false, no discards will be issued for this volume. If this setting is true, and either automatic discards or **red_fstrim()** are used with this volume, then discards will be issued. See the [Discard Operations](#) for details on discards.

Discards are only supported by the commercial version of Reliance Edge. The configuration utility will warn you of this when you set "Discard (trim) support" to "Supported." You may safely ignore the warning if you have purchased a license for Reliance Edge and received the commercial source kit. But if you save the configuration and use it with source code downloaded from the Reliance Edge GitHub repository, you will encounter build errors. See [here](#) or contact sales@tuxera.com for Reliance Edge commercial licensing options.

12.3.8.7 Inode Count

Where to find it: In the configuration utility, this setting is found under the Volumes tab in the spinbox labeled "Inode count"; it is configured separately for each volume. The corresponding values in redconf.c are the fifth member of every gaRedVolConf element (.ulInodeCount).

What it means: Every time a file or directory is created, one *inode* is consumed; when a file or directory is deleted, the inode is freed and available for reuse. Also, when using the POSIX-like API, one inode is consumed by the root directory. In other words, the inode count dictates the maximum number of files and directories that can reside on the volume.

Guidance: The number of inodes will depend on how many files and directories will be created on the file system either by your application or during image creation by the [Image Builder](#). Some applications have very fixed use cases (for example, using exactly twelve files with well-defined purposes) which makes it easy to determine this value; use cases where the File System Essentials API is appropriate generally fall into this category. If you have a root file system, you can set the inode count to the number of files and directories in that root file system (including the root directory) plus the number of files and directories created at run-time. You can configure Reliance Edge with more inodes than are actually needed; the only downside is that each inode consumes two blocks of space which cannot be used for anything else.

12.3.8.8 Block I/O Retries

Where to find it: In the configuration utility, this setting is found under the Volumes tab. Two elements are used to specify this setting. If the checkbox labelled "Retry block device I/O on failure" is not selected, then it is set to 0. Otherwise it is set to the number specified in the spinbox labelled "Maximum retries." The corresponding values in redconf.c are the sixth member of every `gaRedVolConf` element (`.bBlockIoRetries`).

What it means: In Reliance Edge, a block device I/O failure is normally a critical error. However, if block I/O retries are enabled, then Reliance Edge will not immediately abort when a read, write, or flush call fails. Instead it will try the same operation again up to the specified number of times before returning an error. The maximum number of times an operation may be attempted is one plus the configured max I/O retries.

Guidance: Block device I/O retries should be enabled when Reliance Edge is interfacing with drivers or hardware that are known to fail intermittently without negative side effects (i.e. the failures do not compromise the device state or any data that has been successfully written). Tuxera has observed such failures while testing SD hardware and drivers on a Freescale Tower board and on an ST Microelectronics STM32 board. If retries are enabled, the ideal maximum number of retries should be found by testing the block device. The likelihood of a sporadic failure should be weighed against the lifespan of the part, the likelihood of a true failure (such as permanent media failure), and other considerations.

If you are unsure whether block I/O retries are necessary for your project, then it is recommended that you leave this feature disabled.

12.3.9 Endianness

Where to find it: In the configuration utility, this setting is found under the Data tab in the drop-down menu labeled "System byte order". The corresponding macro in `redconf.h` is `REDCONF_ENDIAN_BIG`.

What it means: This setting is used to specify whether the byte order of the target system is little endian or big endian.

Guidance: The correct value is determined by your hardware. The majority of embedded systems are little endian. For example, ARM is bi-endian, but uses little endian by default. If this value is set incorrectly, the file system driver will not work: `red_init()` or `RedFsInit()` will return an error.

12.3.10 Alignment Size

Where to find it: In the configuration utility, this setting is found under the Data tab in the drop-down menu labeled "Native alignment size". The corresponding macro in `redconf.h` is `REDCONF_ALIGNMENT_SIZE`.

What it means: This indicates the pointer alignment required to safely and efficiently access a pointer in an "aligned" fashion. In practice, "aligned" means dereferencing a pointer as if it were a pointer to `uint32_t`. A pointer is assumed to be sufficiently aligned to be accessed in this fashion if its address is evenly divisible by the alignment size.

Guidance: The default value of 4 will work for most systems (since that is the size of a `uint32_t`). On some architectures (like x86), an unaligned pointer can be accessed in an aligned fashion, but it is less efficient, so the alignment value should still be set. At this time, this setting is only used by the slice-by-8 CRC algorithm. If you are using an idiosyncratic architecture where the pointer address is not a reliable indicator of pointer alignment, this setting can be ignored, and the slice-by-8 CRC algorithm should not be used.

12.3.11 CRC Algorithm

Where to find it: In the configuration utility, this setting is found under the Data tab in the drop-down menu labeled "CRC algorithm". The corresponding macro in `redconf.h` is `REDCONF_CRC_ALGORITHM`.

What it means: Reliance Edge stores a CRC-32 value in each of its metadata blocks to allow the driver to check for media errors that have corrupted the metadata. There are several CRC algorithms, all of which compute identical CRC

values given identical data and are thus equally good at detecting errors; but the different algorithms allow for trade-offs between size and speed.

Guidance: The default slice-by-8 algorithm is the fastest and the largest; it computes the CRC in 8-byte slices, taking advantage of the ability of many modern processors to load memory from several locations in parallel. It has the most code and it uses an 8 KB read-only table. If you have enough code space (on your ROM or NOR flash), the slice-by-8 algorithm is recommended since it provides the best performance. The slice-by-8 algorithm also relies on aligned memory access; see notes on [Alignment Size](#).

The Sarwate algorithm is a fairly fast implementation which is much smaller than slice-by-8; it computes the CRC one byte at a time and uses a 1 KB read-only table. If the slice-by-8 algorithm is too big, the Sarwate algorithm is a good choice that provides decent performance in a much smaller package.

The bitwise algorithm is a slow but very small implementation; it computes the CRC one bit at a time and does not use a table. This algorithm is only recommended if space is very limited and Reliance Edge needs to be as small as possible.

Since all three algorithms compute the same CRC result, the CRC algorithm can be changed at any time without breaking media compatibility.

12.3.12 Inode st_blocks Field

Where to find it: In the configuration utility, this setting is found under the Data tab in the checkbox labeled "Enable inode block count". The corresponding macro in `redconf.h` is `REDCONF_INODE_BLOCKS`.

What it means: This setting determines whether to track the number of data blocks allocated to an inode. Since Reliance Edge supports *sparse files* (files with unwritten areas that have no data allocated and which read as zeroes), dividing the file size by the block size is not an accurate indicator of how much data is allocated to the inode; this setting enables a field and code to track the number of data blocks actually allocated. This information is made available in the `REDSSTAT::st_blocks` member, returned by `red_fstat()` (and as an extension, by `red_readdir()`).

Guidance: If you plan to use `REDSSTAT::st_blocks` in your application, enable this setting; otherwise, disable it to reduce code size and free up space in the inode.

12.3.13 Inode Timestamps

Where to find it: In the configuration utility, this setting is found under the Data tab in the checkbox labeled "Enable inode timestamps". The corresponding macro in `redconf.h` is `REDCONF_INODE_TIMESTAMPS`.

What it means: This setting determines whether to store timestamps with the inode, namely atime (time of last access), mtime (time of last modification), and ctime (time of last status change). If enabled, these timestamps are accessible in the `REDSSTAT` structure. The values assigned to these timestamps rely on `RedOsClockGetTime()` (see the [real-time clock service](#) section for details).

Guidance: If you plan to use the inode timestamps in your application, or like having them available to support file system analysis, enable this setting; otherwise, disable it to reduce code size and free up space in the inode. If you have not implemented the [real-time clock service](#), there is little point in enabling this setting since the timestamps will always be zero.

12.3.14 Access Time

Where to find it: In the configuration utility, this setting is found under the Data tab in the checkbox labeled "Update atime on read". The corresponding macro in `redconf.h` is `REDCONF_ATIME`.

What it means: This setting determines whether to update the atime (time of last access) timestamp of an inode when it is read. For a file, this would be updated during `red_read()` or `RedFseRead()`; for a directory, it would be updated during `red_readdir()`.

Guidance: Keeping atime up-to-date has the disadvantage of adding write operations to read operations—for example, `red_read()` will include a write to disk, instead of just reads. While disabling this setting reduces POSIX compliance, many systems do so (as with the `noatime` mount option in Linux). If your application has a need for atime to be kept up-to-date, enable this setting; otherwise, disable it to improve performance and limit writes to the media which may reduce device lifetime.

12.3.15 Inode Pointer Configuration Values

Where to find it: In the configuration utility, these settings are found under the Data tab in the checkboxes labeled "Direct pointers per inode" and "Indirect pointers per inode". The corresponding macros in `redconf.h` are `REDCONF_DIRECT_POINTERS` and `REDCONF INDIRECT_POINTERS`.

What it means: Inodes (file system objects used to implement files and directories) point to data blocks, either directly or indirectly. A greater degree of indirection allows for larger files, while a lesser degree of indirection is more efficient. Like traditional Unix file systems, Reliance Edge offers a compromise by providing a mix of indirection levels, with lower file offsets having less indirection and higher file offsets having more. Inodes have three pointer types: direct pointers, which store file data block numbers; indirect pointers, which store block numbers of indirect blocks which in turn store file data block numbers; and double indirect pointers, which store block numbers of double indirect blocks which in turn store indirect block numbers.

Unlike other file systems, Reliance Edge allows the number of each pointer type to be configured to allow for different use cases. The number of direct and indirect pointers are configured in these settings, and leftover pointers are designated double indirect pointers.

Guidance: These settings are a powerful way to optimize Reliance Edge for your use case. For example, if most of your files are under 20 KB and you have a 1024 byte block size, you can put 20 direct pointers in the inode so that most of your files will read and write very efficiently due to the lack of indirection; the first 20 KB of large files would have the same efficiency improvement, which may be useful for file types with metadata headers that are accessed more often than later areas of the file. Adding indirect pointers allows for more efficient access to midsized files. Each indirect stores $(B - 20) / 4$ data block pointers (where B is the block size); so with a block size of 1024, adding 32 indirect pointers would allow files up to 8 MB to be accessed with only one level of indirection. As described in the [Maximum File Sizes](#) section, reducing the amount of indirection in the inode has the effect of reducing the maximum file size; the Info tab in the configuration utility reports the maximum file size with the current settings. If the maximum file size is too small, setting the direct and indirect pointer counts to zero (resulting in an inode which stores nothing but double indirect pointers) will increase the maximum; if it is still too small, then the only option is to increase the block size.

This setting affects directory inodes as well as file inodes. Since directories are usually small, keeping a few direct pointers in the inode is useful for improving the performance of directory operations.

12.3.16 Block Buffer Count

Where to find it: In the configuration utility, this setting is found under the Memory tab in the spinbox labeled "Allocated buffers". The corresponding macro in `redconf.h` is `REDCONF_BUFFER_COUNT`.

What it means: This setting determines the number of block buffers in the block buffer cache.

Guidance: Reliance Edge needs a certain minimum number of block buffers in the cache in order to operate. This minimum can be anywhere from 2 to 12, depending on other settings; the configuration utility will require this minimum is met. Using a buffer count above the minimum will improve performance (often substantially) at the cost of using more RAM. Thus, if the performance of Reliance Edge is not satisfactory, increasing the buffer count is a good experiment. The improvement in performance produced by increasing the buffer count is subject to diminishing returns; if there are already several dozen buffers, adding more will have little effect. The maximum buffer count is 255, but it should be noted that the buffer cache is optimized for a relatively small number of buffers, and that using hundreds of buffers (if such a large amount of RAM is available) may actually decrease performance.

12.3.17 Memory and String Functions

Where to find it: In the configuration utility, these setting are found under the Memory tab in the area labeled "Memory Management Methods". The corresponding macros in `redconf.h` are `RedMemCpyUnchecked`, `RedMemMoveUnchecked`, `RedMemSetUnchecked`, `RedMemCmpUnchecked`, `RedStrLenUnchecked`, `RedStrCmpUnchecked`, `RedStrNCmpUnchecked`, `RedStrNCopyUnchecked`, although in some configurations, some of these macros (or possibly all of them) will be left undefined.

What it means: Reliance Edge needs functions like `memcpy()` and `strlen()`, but it also needs to work in freestanding environments where the C library functions might be unavailable. This setting tells Reliance Edge which functions to use: the C library versions, its own implementations, or custom functions. Regardless of which functions are used, Reliance Edge always invokes them via wrapper functions which checks for invalid parameters; for example, there is a `RedMemCpy()` function which wraps `RedMemCpyUnchecked()`.

Guidance: This function has two presets. The first is to use the standard C library functions, where `string.h` is included and functions with the standard names (`memcpy()`, `memmove()`, `strlen()`, `strcmp()`, and so on) are used. This is a reasonable choice if you know that header and those functions are available in your system. The advantage of using the C library functions, if available, is they tend to be optimized, and code size is not increased since those functions were already present on the target.

The second preset is to use Reliance Edge implementations of these functions (see the code in `util/memory.c` and `util/string.c`). This has the advantage that it works anywhere, without external dependencies, and these implementations are known not to have any bugs which will cause problems for the file system. The disadvantage is that these are very simple implementations without any optimization, and if these functions are already implemented elsewhere in the system, code size is increased by unnecessarily including two copies of these functions.

The third option allows for customization. For example, perhaps you already wrote versions of these functions for your application; you can tell Reliance Edge to use these same versions, by providing the names and the header file to include. You can also mix-and-match: for example, if you have most of these functions written for your application, but not `strncmp()`, you can fill out the other text boxes but leave the "Bounded string compare" box empty so that the Reliance Edge implementation of `strncmp()` will be used.

12.3.18 Default Transaction Settings

Where to find it: In the configuration utility, these settings are found under the Transactions tab in the area labeled "Transaction Settings". The corresponding macro in `redconf.h` is `REDCONF_TRANSACT_DEFAULT`.

What it means: These settings determine which events automatically trigger a transaction point by default; these transaction settings can be changed at run-time by the application. For an explanation of transaction points and automatic transaction points, see [How Reliance Edge Works](#).

Guidance: If you are unsure, the default automatic transaction events will work fine for most use cases. If your application causes too many transaction points (for example, if it is constantly opening and closing files, and the close automatic transaction point is enabled) then consider turning off the offending event. While there are times where it is useful to disable all transaction points (such as the [system update use case](#)), normally this is done at run-time when needed, rather than disabling all automatic transaction points all of the time.

A few notes on specific automatic transaction events:

- Write automatic transactions should usually be *disabled*. Committing a transaction point every time a file is written to is detrimental to performance and storage media lifetime.
- Volume full automatic transactions should usually be enabled. Reliance Edge can often avoid returning a disk full error by automatically transacting when free space is low; if these automatic transactions are disabled, disk full errors will be returned even when transacting would have freed enough space to have allowed the operation to succeed.

- Volume unmount automatic transactions should usually be enabled. If you unmount without transacting, any changes that have not been transacted are discarded. While this is occasionally a useful feature (it allows the file system to revert to the last committed state), usually a user does not want unmount to change the contents of the file system.

12.3.19 Options Set Automatically

Several macros in `redconf.h` are set automatically by the configuration utility based on other settings. You do not need to read this section to configure your project; it is included only for completeness.

12.3.19.1 Volume Count

`REDCONF_VOLUME_COUNT` is set based on the number of volumes created in the Volumes tab.

12.3.19.2 Imap Configuration Options

`REDCONF_IMAP_INLINE` and `REDCONF_IMAP_EXTERNAL` are set based on the size and block size of the configured volumes. Reliance Edge uses a free space bitmap called the *imap*. When the *imap* is small enough to fit in the metaroot block (which tends to be the case with smaller volumes or larger block sizes), the *inline imap* code is used; this is an optimization for smaller volumes. When the *imap* is too large to be inlined, the *external imap* code is used. If there are no volumes using the *inline imap* code, it is disabled by setting `REDCONF_IMAP_INLINE` to zero to reduce code size. Likewise, if there are no volumes using the *external imap* code, it is disabled by setting `REDCONF_IMAP_EXTERNAL` to zero to reduce code size.

12.3.19.3 Image Builder

`REDCONF_IMAGE_BUILDER` is always 0 in the generated `redconf.h`; only in the host subprojects is it ever 1.

12.3.19.4 Checker

`REDCONF_CHECKER` is always 0 in the generated `redconf.h`; only in the host subprojects is it ever 1.

Chapter 13

Discard Operations

13.1 Discard Availability in Reliance Edge SDKs

Discards are only supported by the commercial version of Reliance Edge. See [here](#) or contact sales@tuxera.com for Reliance Edge commercial licensing options.

The rest of this chapter assumes the reader is using a commercially-licensed Reliance Edge SDK, which has support for discards.

13.2 Discards 101: A Brief Introduction

This section explains discards in a generic way (not specific to Reliance Edge).

13.2.1 Discard Definition

Discard (also called trim, unmap, or erase in various command sets) is a command which informs the storage media that the contents of a sector or range of sectors are no longer important, and thus the sectors' contents do not need to be preserved, until such time that the sectors are rewritten.

13.2.2 Discard Availability

Not all storage media support discards. Most modern media built on flash memory will support some sort of discard; older generations might not. Rotating hard drives often lack discards because they are not very useful for that media type.

Not all operating environments support discards. Many real-time operating systems lack support for discards in their block device interfaces and media drivers.

13.2.3 Discard Benefits

This section explains why issuing discard commands can be a good thing, in the context of storage media using flash memory. The reader needs a basic understanding of flash memory concepts to understand this explanation, so a quick review of flash memory is included.

Flash memory is not like a hard disk: it does not have sectors which can be individually written and rewritten at any time in any order. A flash memory device consists of one or more chips, which in turn consist of erase blocks, which in turn consist of pages. As a concrete example, consider a chip with 1024 erase blocks, and each erase block has 128 pages, and each page is 4KB (not including out-of-band area); thus each erase block has 512KB worth of pages, and the chip has 512MB of erase blocks. Erase blocks can be erased, which resets every page in the block to the erased state, a process which wipes out the data in those pages. Only pages in the erased state can be written; a page, once written, cannot be rewritten without re-erasing the erase block and losing all the page data. Normally there is also a requirement that pages within an erase block are written sequentially. Most file systems are not designed to follow these rules (to say nothing of other flash memory issues like bad block handling and wear leveling), so a flash translation layer (FTL) is run atop of the raw flash memory and simulates a hard disk with rewritable sectors. The FTL can be a piece of software, like Tuxera FlashFX Tera, or it can exist in firmware, as with eMMC, SD cards, SATA SSDs, and so on.

A modern high-performance FTL (a page-based FTL as opposed to the older block-based FTLs), when it gets a write request, will write into an erase block with erased pages and maintain metadata to record the physical location of the written sectors. When a sector is rewritten, it gets written to a new page (copy on write) and the metadata indicating its location is updated. Eventually, after enough has been written, the FTL will begin to run low on erase blocks which have not been written. Many of the erase blocks will contain pages with obsolete data (dead pages), data for sectors that have since been rewritten. To reclaim space, the FTL will pick an erase block with dead pages, copy out the live (not dead) pages, and then erase the block so it can be reused. This process is called garbage collection (or compaction in FlashFX terminology). Garbage collection can be an expensive process, substantially slowing down writes to the disk. The more live pages there are, the longer it takes for garbage collection to reclaim space.

Discards help garbage collection by decreasing the number of live pages. If all the sectors in a page have been overwritten, the FTL knows the page is dead. But if the sectors in the page were written for a file that was since deleted or truncated, but have not been rewritten, the page is still live – unless the file system discards the sectors during or after the delete or truncate. The FTL knows that the contents of a discarded sector are no longer important, so pages containing discarded sectors can be dead pages which are not copied when the containing erase block is garbage collected. This makes garbage collection faster, and thus makes writing to the disk faster when garbage collection is active; it also reduces FTL write amplification and thus increases how long the flash memory will last before it wears out.

Discards have little, if any, positive impact until the disk starts garbage collecting. Attempting to gauge the impact of discards by formatting the disk (which usually discards the whole media) and running a short benchmark is almost always misleading, because FTL has enough unused erase blocks to make it through the whole test without any garbage collection. Longer and smarter testing is required when testing discards.

13.2.4 Discard Drawbacks

Discards are not free. They take time to execute, which can impose a performance penalty for those operations which issue discards.

Most FTLs need to write a record to indicate that a sector has been discarded; thus, in certain scenarios, discards can result in writing more to the flash memory instead of less. These downsides to discarding tend to be more pronounced when the discards are small; thus deleting or truncating or overwriting small or fragmented files can result in especially expensive discards.

The severity of these drawbacks depends on the use case and how well the storage media processes discards. In some cases, it is cheaper to turn discards off and pay the penalty during garbage collection.

13.3 Discards in Reliance Edge

Reliance Edge supports two types of discards: *automatic discards* and *manual discards*. It is possible to use these in combination or to disable both.

13.3.1 Automatic Discards

Automatic discards is the term used for discards as they are traditionally implemented by file systems: discards are issued for sectors when those sectors become part of free space.

13.3.1.1 Enabling Automatic Discards

To enable automatic discards, first enable [Automatic Discards](#), then enable [Discards Supported by Block Device](#) for each volume which resides on a block device that supports discards. With that configuration, volumes which are mounted via `red_mount()` or `RedFseMount()` will have automatic discards enabled; while volumes which are mounted with `red_mount2()` will have automatic discards enabled if (and only if) the `RED_MOUNT_DISCARD` flag is specified.

13.3.1.2 When are Automatic Discards Issued?

Reliance Edge will issue automatic discards for the sectors of allocable blocks that become free. Allocable blocks which are working state (i.e., *not* part of the transacted state) may be freed directly, for example when a file is deleted or truncated to a smaller size, and the discards will happen immediately. Allocable blocks which are committed state (i.e., part of the transacted state) are never freed immediately, because the blocks are still a valid constituent of the most recent transaction point. Instead, when committed state blocks are overwritten, truncated, or deleted, the blocks become "almost free". After a transaction point, the allocation bitmap is parsed to find blocks that were almost free, and discards are issued for those blocks. In other words, for blocks that are part of the committed state, discards will be postponed until after a transaction point that establishes a new committed state.

13.3.1.3 Performance Impact of Automatic Discards

Automatic discards are intended to increase performance and flash media lifespan. However, the process of determining which sectors should be discarded and discarding them does take time. As a result, automatic discards may actually cause a small decrease in performance on a freshly-formatted volume. The benefits of discards will show up later after most of the storage medium has been written to by the file system.

In some cases, automatic discards may never yield a benefit. If the storage medium does not need to do garbage collection (e.g., a rotating hard drive), then discards do not have much positive impact. If you have partitioned your storage medium and leave a significant percentage of it as unused free space, then garbage collection may be reasonably efficient even without discards.

13.3.1.4 Discards and FlashFX Tera

If you are using Reliance Edge with Tuxera FlashFX Tera, enabling automatic discards is strongly recommended.

13.3.2 File System Trim (Manual Discards)

Manual discards is the term used for discards issued by `red_fstrim()`, part of the POSIX-like API. The name "fstrim" is a reference to the `fstrim` utility on Linux (part of the `util-linux` package); `red_fstrim()` provides functionality similar to `fstrim` and is useful for the same reasons.

13.3.2.1 Enabling `red_fstrim()`

To enable `red_fstrim()`, you must enable the [POSIX-like File System API](#). Then, you must separately enable the configuration option for `red_rmdir()`. Finally, at least one volume must have the [Discards Supported by Block Device](#) setting enabled.

13.3.2.2 `red_fstrim()` vs. Automatic Discards

Compared to automatic discards, `red_fstrim()` requires more effort on the part of the end-user, since the application must have logic to invoke `red_fstrim()` at a suitable frequency (see next section). The reward is that `red_fstrim()` can be used to achieve the long-term benefits of discards with less run-time overhead. Compared to automatic discards, judicious use of `red_fstrim()` can lead to discarding less often in larger chunks. Disabling automatic discards will also speed up transaction points, since the allocation bitmap no longer needs to be scanned for newly freed blocks.

13.3.2.3 `red_fstrim()` frequency

On Linux, `fstrim` is typically invoked with a relatively low frequency, such as once a week. A similar infrequent interval is advisable for `red_fstrim()` to minimize flash wear. One way this can be implemented on systems with an RTC is to simply store a timestamp in a file representing the last time that `red_fstrim()` was used; periodically read from that file, and if sufficient time has elapsed, use `red_fstrim()` and write an updated timestamp into the file.

13.3.2.4 `red_fstrim()` latency

`red_fstrim()` can potentially take a long time, depending on its arguments. You can discard everything at once, as in the following snippet:

```
// Issue discards for every free block in the allocable area.
//
ret = red_fstrim(volumeName, 0U, UINT32_MAX);
```

But be aware that such an invocation may take a very long time to complete, depending on the size of the volume and the efficiency of discards on the storage medium. This may pose a problem if there are other tasks which need access to Reliance Edge. Thus, it might make sense reduce latency by invoking `red_fstrim()` repeatedly with a smaller range, for example:

```
const uint32_t ulBlocksPerFstrim = 256U;
uint32_t ulBlockStart = 0U;
REDSSTATFS fsinfo;
int32_t ret;

ret = red_statvfs(volumeName, &fsinfo);
if(ret == -1)
{
    // Error handling omitted...
}

while(ulBlockStart < fsinfo.f_blocks)
{
    // It's okay if ulBlockStart + ulBlocksPerFstrim is beyond the end of
    // the volume: red_fstrim() ignores out-of-range blocks.
    //
    ret = red_fstrim(volumeName, ulBlockStart, ulBlocksPerFstrim);
    if(ret == -1)
    {
        break; // Error handling omitted...
    }

    // Avoid incrementing ulBlockStart above UINT32_MAX, thereby causing it
    // to wrap-around to a smaller value, leading to an infinite loop.
    //
    ulBlockStart += MIN(ulBlocksPerFstrim, UINT32_MAX - ulBlockStart);
}
```

13.3.2.5 Eliminating Data Remanence with `red_fstrim()`

Data remanence exists when data that has been deleted, truncated, or overwritten (from a file system perspective) persists on the storage medium. This remanence might be externally visible, as when freed sectors still contain their

former data; or the remanence might be internal, as on flash memory when a spare erase block has old data. At times, eliminating such data remanence may be desirable for security reasons: for example, wiping all user data as part of a factory reset.

Discards do not, by themselves, guarantee that the discarded data has been eliminated from the underlying storage medium. However, some storage media, such as eMMC, implement a "sanitize" command which guarantees that all data in discarded sectors is wiped. Thus, [red_fstrim\(\)](#) can be used in combination with "sanitize" to wipe the storage medium of any data which exists, or formerly existed, in the free space sectors.

Using [red_fstrim\(\)](#) in this fashion is a three-step process:

1. Delete (or truncate to zero size) the files with sensitive data.
2. Invoke [red_fstrim\(\)](#) for every block on the volume. Either code snippet in the preceding section would accomplish this.
3. Send the "sanitize" command to the storage media.

The [Reliance Edge block device API](#) does not include a function for sanitize, so the third step will involve invoking a custom function: perhaps a function implemented by the low-level block device driver.

Even if [Automatic Discards](#) are enabled, [red_fstrim\(\)](#) should still be used when eliminating data remanence is the goal. Automatic discards are not guaranteed to result in discards for every single sector that becomes free. In particular, power interruption (or unmounting without an unmount automatic transaction) can create sectors in free space which are not discarded. Calling [red_fstrim\(\)](#) is the only way to ensure that all free space sectors are discarded.

Chapter 14

Configuration Utility

14.1 Introduction

In order to support a wide variety of devices while minimizing storage and memory footprints, Reliance Edge requires certain device characteristics and other options to be specified at compile-time. The Reliance Edge configuration utility is provided to help make this process easy and reliable. This chapter is mostly about the tool itself; for more details on the process of configuring Reliance Edge, including a more detailed discussion of the individual configuration settings, see the [Product Configuration](#) chapter.

14.1.1 Installation and Setup

The configuration utility is distributed as a Windows executable which can be downloaded from the [Reliance Edge website](#). It does not require any installation procedures or administrative privileges. The executable is tested to work on the Windows 7 and 10 operating systems. The executable file is named redconfig.exe. Its folder also contains several support library files. These files must exist in the working directory of the configuration utility.

The [source code](#) for the configuration utility is distributed with Reliance Edge in the folder tools/config. The file tools/config/README.txt describes how to build the configuration utility on Windows or Linux.

14.1.2 Versions and Compatibility

In order to produce the correct configuration format for the chosen version of Reliance Edge, the correct version of the configuration utility must be used. Whenever a Reliance Edge update involves configuration changes, the version of the configuration utility will be updated to match it. (The version of the utility may be identified by opening the *File* menu and selecting *About*.)

The following table identifies which version of the configuration utility for each version of Reliance Edge since v1.0.

Reliance Edge version	Configuration Utility version required
2.5 or later	2.5
2.4	2.4
2.3	2.3
2.2	2.02
2.1	2.01
2.0	2.0
1.0.2 - 1.0.4	1.0.2
1.0 - 1.0.1	1.0

Current and obsolete versions of the configuration utility are available on the utility's [download page](#).

14.2 Editing settings

Settings are organized in tabbed sections.

14.2.1 General

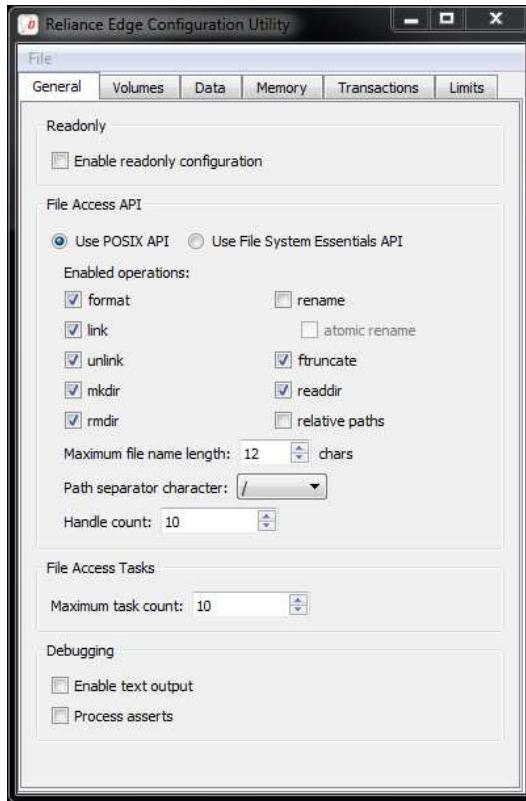


Figure 14.1: General Settings

The General tab contains the read-only setting, file access API-related settings, and debugging-related settings. The following macros are set within the General tab:

- [REDCONF_READ_ONLY](#)
- [REDCONF_DISCARDS](#)
- [REDCONF_API_POSIX](#)
- [REDCONF_API_FSE](#)
- [REDCONF_API_POSIX_FORMAT](#)
- [REDCONF_API_POSIX_LINK](#)
- [REDCONF_API_POSIX_UNLINK](#)

- [REDCONF_API_POSIX_MKDIR](#)
- [REDCONF_API_POSIX_RMDIR](#)
- [REDCONF_API_POSIX_RENAME](#)
- [REDCONF_RENAME_ATOMIC](#)
- [REDCONF_API_POSIX_FTRUNCATE](#)
- [REDCONF_API_POSIX_REaddir](#)
- [REDCONF_API_POSIX_CWD](#)
- [REDCONF_API_POSIX_FSTRIM](#)
- [REDCONF_NAME_MAX](#)
- [REDCONF_PATH_SEPARATOR](#)
- [REDCONF_TASK_COUNT](#)
- [REDCONF_HANDLE_COUNT](#)
- [REDCONF_API_FSE_FORMAT](#)
- [REDCONF_API_FSE_TRUNCATE](#)
- [REDCONF_API_FSE_TRANSMASKGET](#)
- [REDCONF_API_FSE_TRANSMASKSET](#)
- [REDCONF_OUTPUT](#)
- [REDCONF_ASSERTS](#)

Read-only configuration

The read-only configuration option disables all write access functions and procedures, producing a read-only file system. Several settings are disabled when this option is selected, including the Transactions tab.

POSIX or File System Essentials API

The user may choose which API to use with configure Reliance Edge. When an API is selected, options are shown below which are specific to that API. This includes options such as which API function calls are enabled.

Path separator character

The path separator character may be configured under the POSIX API. A drop-down menu is displayed with common path separators (slashes) and a Custom option. If Custom is selected, the user may type any ASCII character or escape sequence into the text field which is revealed. A null character (\0) is not accepted. If relative paths are enabled ([REDCONF_API_POSIX_CWD](#)), the path separator cannot be . (a period), since that would interfere with parsing of dot and dot-dot path components.

14.2.2 Volumes

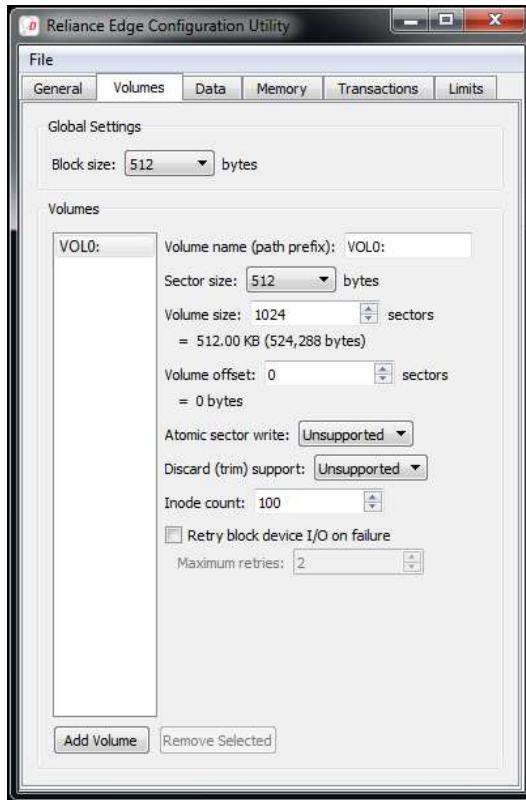


Figure 14.2: Volume Settings

The Volumes tab controls the partitioning of the disk and all volume-specific settings, as well as the block size to use. The global array of volume information, `gaRedVolConf`, is set by this tab, along with the macro [REDCONF_BLOCK_SIZE](#).

Volume name (path prefix)

Under the POSIX API, each volume must be assigned a path prefix, which is also displayed in the left-hand column. If the File System Essentials API is selected (on the [General tab](#)), this option is disabled, as volumes are accessed by indexes. These indexes are then displayed in the left-hand column.

14.2.3 Data

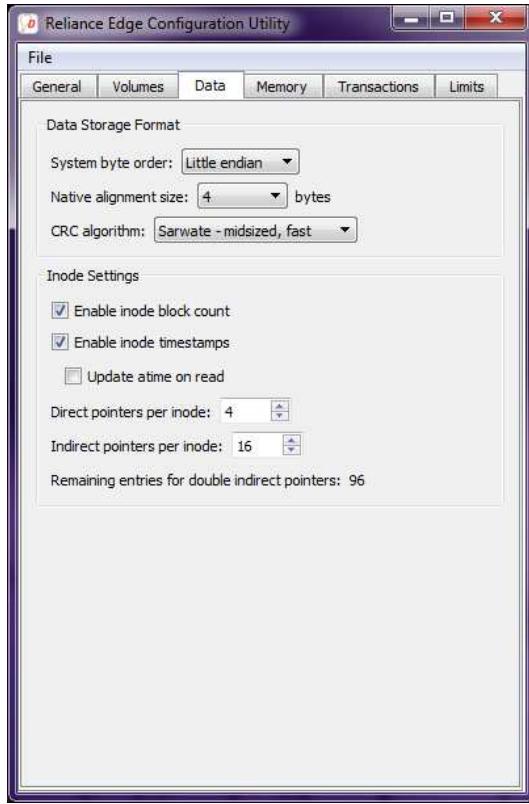


Figure 14.3: Data Settings

The Data tab contains data storage settings, including inode configuration. The following macros are set within the Data tab:

- [REDCONF_ENDIAN_BIG](#)
- [REDCONF_ALIGNMENT_SIZE](#)
- [REDCONF_CRC_ALGORITHM](#)
- [REDCONF_INODE_BLOCKS](#)
- [REDCONF_INODE_TIMESTAMPS](#)
- [REDCONF_ATIME](#)
- [REDCONF_DIRECT_POINTERS](#)
- [REDCONF INDIRECT_POINTERS](#)

Direct and indirect pointers

The number of direct and indirect pointers per inode may be set. The combined number of direct and indirect pointers may not exceed the number of available inode entries in a block. The remaining entries are used for double indirect pointers, as reported in the user interface.

14.2.4 Memory

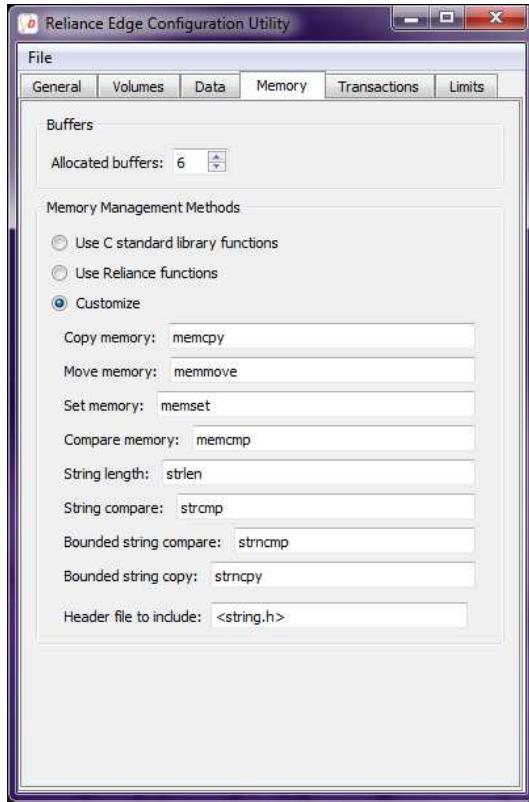


Figure 14.4: Memory Settings

The Memory tab controls the number of buffers kept in memory as well as the functions used for memory and string manipulation. The following macros are set by this tab:

- [REDCONF_BUFFER_COUNT](#)
- `RedMemcpyUnchecked`
- `RedMemMoveUnchecked`
- `RedMemSetUnchecked`
- `RedMemCmpUnchecked`
- `RedStrLenUnchecked`
- `RedStrCmpUnchecked`
- `RedStrNCmpUnchecked`
- `RedStrNCPyUnchecked`

Allocated buffers

The number of buffers kept in memory is set at compile time. The maximum number of buffers is 255, and the minimum number is determined by other settings, such as whether the POSIX API is selected and if rename operations are

enabled. If you set this too low, an error icon will appear. Hover over it to view the minimum number in your current configuration.

Memory management functions

Three options are given for memory management and string manipulation functions. For compilers which have a satisfactory implementation of the C standard library, the default option to *Use C standard library functions* may be selected. If the compiler or target platform does not support these, then the Reliance implementations may be used by selecting the *Use Reliance functions* option. The *Customize* option allows each function to be set distinctly.

When customizing these functions, leave a line blank to use the Reliance Edge implementation of that function. (Reliance Edge uses a `#ifndef` block to determine whether to compile each function implementation.)

If a header file is required to declare any of the customized functions, then its name should be placed in the *Header file to include* field. The header name should be enclosed in quotation marks or angle brackets, based on its relative location to the saved `redconf.h` file. For example, the file `<string.h>` must be included if any C standard library string manipulation functions are specified.

The configuration utility does not check the viability of any customized input in these fields.

14.2.5 Transactions

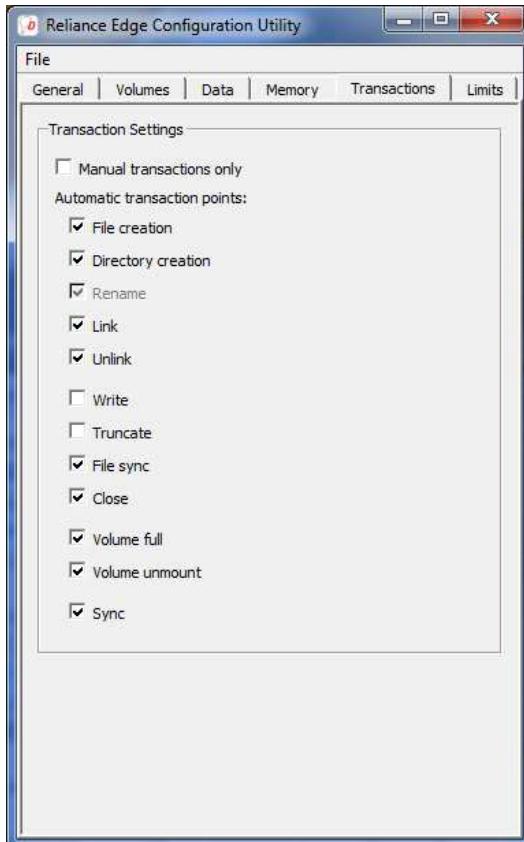


Figure 14.5: Transaction Settings

The Transactions tab sets the default transaction mask, which controls which events trigger automatic transaction points. The macro `REDCONF_TRANSACT_DEFAULT` is set within this tab. If the *Enable readonly configuration* option is selected on the *General* tab, then this tab is disabled.

Each automatic transaction setting corresponds to a file system API call. If the corresponding API call is deselected on the *General* tab, then the transaction settings will be disabled. If the File System Essentials API is selected, then all POSIX-specific API calls are disabled with their corresponding transaction settings. This includes all transaction settings except *Write*, *Truncate*, *Volume full*, and *Volume unmount*.

Leaving the *Volume full* transaction setting selected is recommended, unless all automatic transactions must be avoided for special purposes. If the *Volume full* transaction setting is disabled, a warning icon will be displayed.

14.2.6 Limits Tab

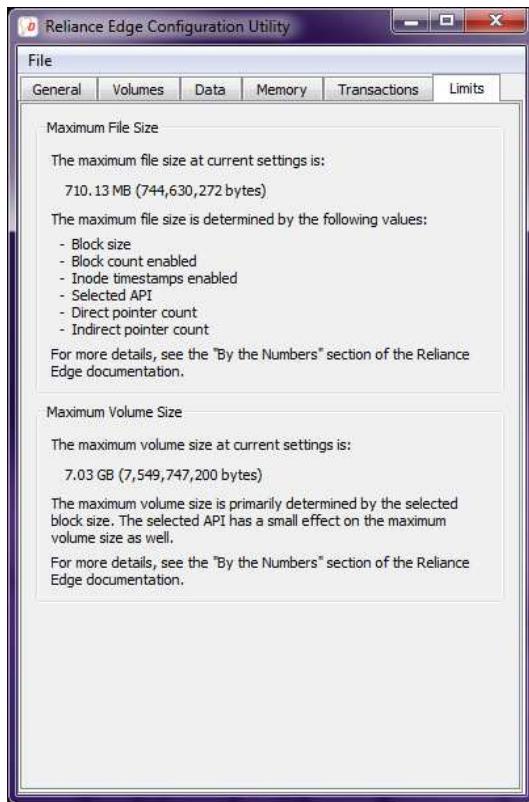


Figure 14.6: Limits Tab

The Limits tab shows the current maximum file and volume sizes. These limits are determined based on other settings, as described in the tab. For further information on the maximum file and volume sizes in Reliance Edge, see the [Limits](#) section of the documentation.

14.2.7 Other Settings

The following macros are generated based on the values of other settings in the configuration tool:

- `REDCONF_VOLUME_COUNT`

- [REDCONF_IMAP_INLINE](#)
- [REDCONF_IMAP_EXTERNAL](#)

The value of [REDCONF_VOLUME_COUNT](#) is determined by the number of volumes added in the *Volumes* tab.

The values of [REDCONF_IMAP_INLINE](#) and [REDCONF_IMAP_EXTERNAL](#) are determined by the sizes of the volumes.

The macros [REDCONF_IMAGE_BUILDER](#) and [REDCONF_CHECKER](#) are always set to 0 (false); only in the host subprojects is it ever 1 (true).

14.3 Saving, Loading, and Using the Configuration Files

14.3.1 Saving and using the configuration

Once the Reliance Edge configuration is satisfactory, the menu option *File -> Save* or *File -> Save As* may be selected in order to use the configuration to build Reliance Edge. If the *File -> Save* option is selected, the two file save dialogs will not be shown if the files already have been written to the disk and can be overwritten by the utility.

If there are incompatible settings, then the user will be prompted to change these settings before saving. If there are settings with values that are not recommended, then a confirmation message will be displayed, prompting the user to continue or return to the interface.

The user is then presented two file save dialogs, first to save the file `redconf.h` and then to save `redconf.c`. The `.h` file contains all of the macro values which are used to compile Reliance Edge under the specified configuration. The `.c` file contains a list of volumes and their settings, as specified in the [Volumes tab](#).

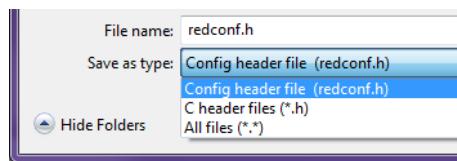


Figure 14.7: Changing the Save File Type

The dialogs only show files with the exact `redconf.[h/c]` filename by default. This may be changed within the dialog via the *Save as type* drop down menu below the file name input box.

14.3.2 Resolving invalid settings

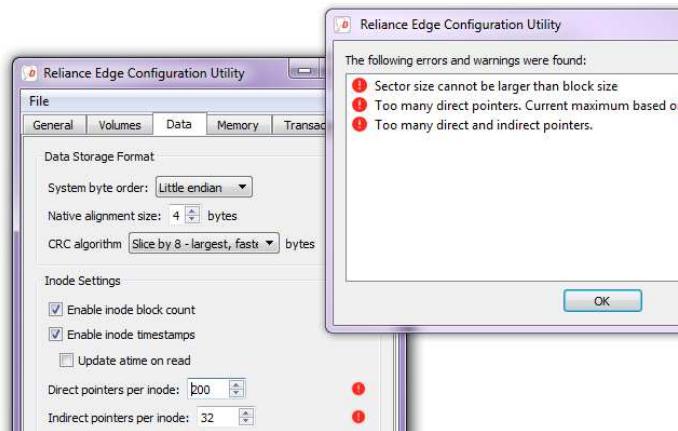


Figure 14.8: Using the Error Dialog to Resolve Settings

When a setting in the configuration is invalid or conflicts with another setting, an error icon is displayed beside the offending setting. If the user attempts to save the configuration, an error dialog is presented, listing all error and warning messages. The same dialog may be shown by clicking an error or warning icon that shows up beside a setting.

The error dialog does not necessarily block user interaction with the parent window; that is, the user may choose to resolve the invalid settings the error dialog is open on the side. However, the list of errors will not be updated until the dialog is closed and re-opened.

14.3.3 Editing previously generated redconf files

Once the `redconf.h` and `redconf.c` files have been saved, they may be loaded into the configuration utility again using the *File -> Load* menu command.

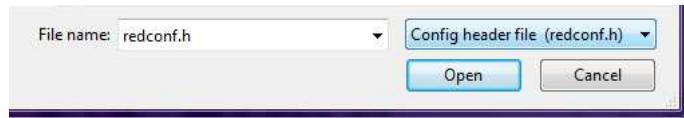


Figure 14.9: Changing the File Type While Loading a File

Two file load dialogs are shown to the user, first asking for an `redconf.h` file and then asking for an `redconf.c` file. The dialog boxes have options to show more file types in case the configuration files were saved under different names.

If any values in the `redconf` files given were edited by hand or the `redconf` files given were not produced by the configuration tool, the settings may not be loaded correctly. A warning will be displayed if there were any settings that the tool was not able to load; however, the tool is not guaranteed to correctly read any settings from `redconf` files that were edited by hand or by another tool.

14.3.4 Using the redconf files

Once the files are saved, the compiler should be configured to look for header files in the folder containing `redconf.h` and to compile `redconf.c` as part of the Reliance Edge project.

14.4 Source Code

The source code of the configuration utility is available under the same license as the rest of the product. The code is written in C++ and uses the Qt libraries for data structures, user interface components, and event handling.

Chapter 15

Command-Line Host Tools

15.1 Introduction

Reliance Edge is packaged with several tools for accessing Reliance Edge file system volumes from the command-line on a host computer running either Windows or Linux. The tools can be used with physical volumes (such as an SD card) or with image files. Instructions for building these tools from source code are found in the [Building the Host Tools](#) section.

15.2 Availability

All of the command-line host tools are included in the commercial kit. The [checker](#) and [image copier](#) are excluded from the GPL kit. See [here](#) or contact sales@tuxera.com for Reliance Edge commercial licensing options.

15.3 Formatter

The Reliance Edge formatter is a command-line tool used to format a Reliance Edge volume on a disk image or physical device. Formatting a volume removes all existing data and prepares it for use by Reliance Edge.

Warning

Formatting a device will permanently delete any data on the device.

15.3.1 Command-Line Syntax

The syntax of the formatter command is as follows:

```
redfmt VolumeID --dev=devname
```

Volume parameter

The parameter `VolumeID` specifies the target volume. Under the FSE API configuration, this must be the unsigned integer volume number. If the POSIX API is enabled, a volume path prefix (such as "VOL1:") will also be accepted.

Device parameter

The parameter `--dev=devname` specifies the block device underlying the volume. The `devname` may be any one of the following:

- The path and name of a disk image file (such as "redimg.bin")
- On Windows, a drive letter (such as "G:") or a Win32 device name (such as "\.\PhysicalDrive7")
- On Linux, a file naming a raw device (such as "/dev/sdb") or a partition (such as "/dev/sdb1")

The Win32 device name format ("\.\PhysicalDriveX") is recommended over a drive letter when formatting physical media that may have been previously formatted for use with another file system. If a master boot record (MBR) is present on the media, the Windows driver letter ("X:") will refer only to the partition, excluding the MBR sector and unpartitioned space. This causes the format operation to write the metadata at an offset location, which could break the ability of Reliance Edge to use the media on a target device, unless the block device implementation in the port is also MBR-aware.

In order to write to a drive by its device name, the image builder must be run from an administrator command prompt. To find the desired drive number (the X in "\.\PhysicalDriveX"), run `wmic diskdrive list brief`.

On Linux, a raw device (such as "/dev/sdb") is recommended over a partition name for the same reasons described above. To view available devices, run `lsblk`.

15.4 Checker

The Reliance Edge checker utility is a command-line tool used to check the system integrity of a Reliance Edge volume. The checker reports success if the volume is valid and no internal errors occur. If volume corruption is found, the specific issue is reported, and the checker exits with an error status.

This tool is only available in the commercial kit.

15.4.1 Command-Line Syntax

The syntax of the checker command is as follows:

```
redchk VolumeID --dev=devname
```

Volume parameter

The parameter `VolumeID` specifies the target volume. Under the FSE API configuration, this must be the unsigned integer volume number. If the POSIX API is enabled, a volume path prefix (such as "VOL1:") will also be accepted.

The volume specified must be unmounted, or the checker will report an error.

Device parameter

The parameter `--dev=devname` specifies the block device underlying the volume. The `devname` may be any one of the following:

- The path and name of a disk image file (such as "redimg.bin")
- On Windows, a drive letter (such as "G:") or a Win32 device name (such as "\.\PhysicalDrive7")
- On Linux, a file naming a raw device (such as "/dev/sdb") or a partition (such as "/dev/sdb1")

15.5 Image Builder

The Reliance Edge image builder is a command-line tool used to build a Reliance Edge image. The tool creates and formats an image file or formats a device on the host system. It then copies specified files from the host system to the target volume for use with the Reliance Edge file system.

Warning

Writing a file system image to a device will permanently delete any data on the device.

15.5.1 Command-Line Syntax

The syntax of the image builder command is as follows:

```
redimgbld VolumeID --dev=devname --dir=inputDir [--map=mappath]
[--defines=file] [--no-warn]
```

The `--map=mappath` and `--defines=file` options are applicable only if the File System Essentials configuration is enabled.

Volume parameter

The parameter `VolumeID` specifies the target volume. Under the FSE API configuration, this must be the unsigned integer volume number. If the POSIX API is enabled, a volume path prefix (such as "VOL1:") will also be accepted.

Device parameter

The parameter `--dev=devname` specifies the block device underlying the volume. The `devname` may be any one of the following:

- The path and name of a disk image file (such as "redimg.bin")
- On Windows, a drive letter (such as "G:") or a Win32 device name (such as "\.\PhysicalDrive7")
- On Linux, a file naming a raw device (such as "/dev/sdb") or a partition (such as "/dev/sdb1")

The Win32 device name format ("\.\PhysicalDriveX") is recommended over a drive letter when creating an image on physical media that may have been previously formatted for use with another file system. If a master boot record (MBR) is present on the media, the Windows driver letter ("X:") will refer only to the partition, excluding the MBR sector and unpartitioned space. This causes the format operation to write the metadata at an offset location, which could break the ability of Reliance Edge to use the media on a target device, unless the block device implementation in the port is also MBR-aware.

In order to write to a drive by its Win32 device name, the image builder must be run from an administrator command prompt. To find the desired drive number (the X in "\.\PhysicalDriveX"), run `wmic diskdrive list brief`.

On Linux, a raw device (such as "/dev/sdb") is recommended over a partition name for the same reasons described above. To view available devices, run `lsblk`.

Input directory parameter

The parameter `--dir=inputDir` specifies a path to a directory that contains all of the files to be copied into the image.

If the POSIX configuration is enabled, then the directory and all subdirectories will be copied to the target volume with file and directory names intact.

If the File System Essentials configuration is enabled and no map file is specified, then only files in the root directory will be copied. Subdirectories will be ignored. Each file will be copied to an assigned index, and a #define statement will be output to refer to that file using a C macro representation of its original file name.

If the File System Essentials configuration is enabled and a map file is specified, then the tool will look in the input directory for all files specified in the map file. The input directory parameter may be excluded in this case. If the input directory is excluded, the map file must contain absolute paths to all files.

Map path parameter (File System Essentials API only)

The optional parameter --map=mappath is a path to a text file which maps input files on the host system to file indexes in the target volume. Because the Reliance Edge File System Essentials API accesses files by index, the file names on the host system are not preserved.

Specifying a map file allows custom assignment of which file index each file in the input directory is assigned. Additionally, it allows files from other directories to be used. If all file paths specified in the map file are absolute paths, then the parameter --dir=inputDir is not required.

Map file syntax

The map file referred to by --map=mappath must follow the following syntax rules.

- Except for blank lines, each line must be either a comment or a map entry.
- Leading whitespace is ignored on each line.
- A comment line must start with a pound character (#).
- A map entry line must begin with the target file index (a positive decimal integer, such as 2), followed by a tab character, followed by the file name or path on the host system.
- File indexes 0 and 1 are reserved for internal file system use. Thus, the target file index must be a positive decimal integer greater than or equal to 2 and less than the configured inode count of the target volume plus 2.
- File indexes must be specified in ascending order.
- The file name or path cannot contain whitespace unless enclosed in quotation marks (such as "my file.txt").
- The same host system file path may be used in multiple entries, but each file index for the target volume may not appear more than once.

An example map file is shown below.

```
# This is an example map file for the Reliance Edge
# image builder tool.

# Indexes 0 and 1 reserved

2 data1.bin
3 "data 2.bin"

# Leave file 4 empty for future use

5 doc/readme
```

Output #define's file parameter (File System Essentials API only)

The optional parameter `--defines=file` specifies a file to which to write `#define` statements to use to refer to file indexes. If specified, the file will be filled with C macros derived from the original file names on the host system. Each macro will be defined to equal the file index to which that file was copied.

If two files names conflict when converted to compatible characters, a number is appended to the macro name of one of them. For example, if two files "file.txt" and "file_txt" are copied by the tool, one will produce a macro named "FILE_TXT" and the other will have a name such as "FILE_TXT1".

If the file referred to by the parameter `--defines=file` cannot be opened for writing, then a message will be displayed and the macros will be written to the console or standard output instead.

If neither the parameter `--map=mappath` nor the parameter `--defines=file` is specified, then the macros will be written to the console or standard output.

By default, if the `--defines` file already exists, the image builder will issue a prompt to confirm whether it should be overwritten. To suppress the prompt and always overwrite, specify `--no-warn`.

15.5.2 Constraints

The image builder tool may report an error and abort if given excessively long file paths. This is due to compatibility with Windows API file path limits.

15.5.3 Copying an Image File

If the target volume is in an image file on a host system, then the file must be copied from the host system to the target device for use. This may be done using a command-line binary copy utility, such as the Unix `dd` utility.

15.6 Image Copier

The Reliance Edge image copier is a command-line tool used to copy all of the files from a Reliance Edge volume to a host system.

This tool is only available in the commercial kit.

15.6.1 Command-Line Syntax

The syntax of the image copier command is as follows:

```
redimcopy VolumeID --dev=devname --dir=outDir [--no-warn]
```

Volume parameter

The parameter `VolumeID` specifies the target volume. Under the FSE API configuration, this must be the unsigned integer volume number. If the POSIX API is enabled, a volume path prefix (such as "VOL1:") will also be accepted.

Device parameter

The parameter `--dev=devname` specifies the block device underlying the volume. The `devname` may be any one of the following:

- The path and name of a disk image file (such as "redimg.bin")
- On Windows, a drive letter (such as "G:") or a Win32 device name (such as "\.\PhysicalDrive7")
- On Linux, a file naming a raw device (such as "/dev/sdb") or a partition (such as "/dev/sdb1")

Output directory parameter

The parameter `--dir=outDir` specifies a path to a directory to which to copy the files. If the POSIX API is enabled, then all files and directories will be copied into this directory, preserving their file names. If the File System Essentials API is enabled, then each file will be named after the index it occupies on the Reliance Edge volume. For example, if the volume has 3 files, the output directory will contain files named 2, 3, and 4.

By default, the image copier utility will prompt if the output directory exists and is a non-empty directory. To automatically delete and replace an existing directory without prompting, use the `--no-warn` parameter.

15.6.2 Constraints

The POSIX API of Reliance Edge is case sensitive, but Windows is not. On Windows, if the Reliance Edge volume has directories that contain file names that differ only by case, the image copier will encounter an error and abort.

The POSIX API of Reliance Edge has no path length limits, but Windows and Linux both do. If the Reliance Edge volume contains very long paths, the image copier may encounter an error and abort.

The image copier has no special handling for hard links, which may be created under the POSIX API configuration. Each link to a file will be copied out as a wholly separate file.

There is no special handling for sparse files. Sparse areas will be read from the Reliance Edge file and copied out as zeroes to the host file.

15.7 Linux FUSE Implementation

Reliance Edge can be installed as a FUSE driver (File System in User Space) on Linux. This allows a user to mount a Reliance Edge volume within a folder so that it appears like a native Linux file system. The contents of the volume may then be accessed with a file browser or any other Linux program.

Note that the FUSE implementation depends on `libfuse-dev`, and it is built as a separate `make` target from the other host tools listed here (`make redfuse`). When built, an executable (`redfuse`) is produced, which can be run to install Reliance Edge as a FUSE driver.

The FUSE implementation is not available on Windows. Currently the FUSE implementation only supports the Reliance Edge POSIX-like API.

15.7.1 Command-Line Syntax

The syntax of the FUSE executable is as follows:

```
redfuse MountPoint --vol=volumeid --dev=devname [--format] [fuseoptions]
```

where `MountPoint` names an existing empty directory at which to mount the volume.

Volume parameter

The parameter `--vol=volumeid` specifies the target volume. A volume path prefix (such as "VOL1:") or an unsigned integer volume number will be accepted. This field is optional if Reliance Edge is configured to use only one volume.

Device parameter

The parameter `--dev=devname` specifies the block device underlying the volume. The `devname` may be any one of the following:

- The path and name of a disk image file (such as "redimg.bin")
- A file naming a raw device (such as "/dev/sdb") or a partition (such as "/dev/sdb1")

If the `--format` flag is specified, Reliance Edge will attempt to format the given device before mounting it as a volume.

FUSE parameters

The FUSE library adds a number of options that are not listed here. To view help for these options, run `redfuse` with no arguments or with `-h`.

15.7.2 Using the FUSE Implementation

When accessing a Reliance Edge volume through the FUSE driver, you may notice certain errors or behaviors that are not common in other file systems. This is because the FUSE driver is intended to allow developers to access their Reliance Edge volumes on a host device; it is not intended to be used as a Linux filesystem replacement. Here are some behaviors you may encounter:

- Slow performance: the FUSE driver runs in userspace and does not support multi-threaded access, so it is expected to run slower than standard Linux filesystems.
- "File name too long": the default maximum name length in Reliance Edge is 12 characters (see [REDCONF_NAME_MAX](#)), which is not enough to store "Untitled Folder" or "Untitled Document." So, expect to see this error if you try to create a new folder or file with a file manager such as GNOME Files or KDE Dolphin. (Try creating the file elsewhere, renaming it, and then moving it instead.)
- "Function not implemented": Reliance Edge does not support some Unix features, such as symlinks, file permissions, and extended attributes. Additionally, many of the Reliance Edge APIs may be disabled in the configuration. This means that functions like `symlink()` and `chmod()` are always expected to fail, and other functions like `ftruncate()` may fail if the corresponding API is disabled in the Reliance Edge configuration.
- "Transport endpoint is not connected": this error may occur if `redfuse` crashes without being properly unmounted. You can run `fusermount [MountPoint] -u` and then re-start `redfuse` to fix this. Try adding a debug argument (`-d`) if you continue to experience this error.

Chapter 16

By the Numbers

16.1 Limits

This section provides details on Reliance Edge file system limits.

Note

This section uses binary sizes. Thus kilobytes (KB) refers to 1024 bytes (not 1000), and so on for megabytes (MB; 1024 KB), gigabytes (GB; 1024 MB), and terabytes (TB; 1024 GB).

16.1.1 Maximum File Sizes

The maximum file size is configuration-dependent. Indeed, no less than six configuration options impact the maximum file size, and a very wide range of maximum file sizes are possible depending on the configuration: from as low as 10 KB to as high as 256 TB.

The limiting factor for file sizes is the number of block pointers that an inode block can point to, either directly or indirectly. This is controlled by three main variables. The first is the size of inode block and the size of the objects that the inode points to: this is controlled by the block size ([REDCONF_BLOCK_SIZE](#)), and larger block sizes allow for larger files.

The second variable is the number of data pointers in the inode. Several configuration values add or remove fields from the inode header, and the larger the inode header is, the less room there is for data pointers and the smaller the maximum file size. The configuration values which effect the inode metadata are:

- [REDCONF_INODE_BLOCKS](#): This adds one field to the inode header (st_blocks).
- [REDCONF_INODE_TIMESTAMPS](#): This adds three fields to the inode header (st_atime, st_mtime, and st_ctime).
- [REDCONF_API_POSIX](#): This adds one field to the inode header (used internally).

The third variable is how the data pointers in the inode are allocated. Some fixed number of pointers are allocated for three roles: direct pointers, which store file data block numbers; indirect pointers, which store block numbers of indirect blocks which in turn store file data block numbers; and double indirect pointers, which store block numbers of double indirect blocks which in turn store indirect block numbers. Reliance Edge allows the number of pointers to be configured to allow for different use cases. Direct pointers are always allocated for the beginning of the file and using them allows small files to be very efficient. Indirect pointers follow direct pointers and allow for more efficient small to medium sized files. Double indirect pointers are the least efficient, but are capable of addressing the most data. In other words, direct pointers optimize small files, double indirect pointers increase the maximum file size, and indirect pointers

are a compromise. The number of inode pointers for each type can be optimized based on the expected file sizes and maximum required file size; for further details, see [this section](#).

It is not possible to list the maximum file size for all possible configurations in this space. The [Configuration Utility](#) will report the maximum file size for your current configuration in its Limits tab. Furthermore, at run time, the POSIX-like `red_statvfs()` API will return the maximum file size in `REDSSTATFS::f_maxsize`.

Below are some maximum file sizes in example configurations. This first table shows the maximum file size at various block sizes, assuming all inode pointers have been allocated to double indirect pointers (to maximize the maximum file size), and that all inode header fields are present:

Block Size	Maximum File Size
128	~1.78 MB (1,866,240 bytes)
256	~44.2 MB (46,339,072 bytes)
512	~857 MB (898,541,568 bytes)
1024	~14.7 GB (15,741,177,856 bytes)
2048	~245 GB (263,218,176,000 bytes)
4096	~3.91 TB (4,304,164,175,872 bytes)
8192	~32.0 TB (35,184,372,080,640 bytes)
16384	~64.0 TB (70,368,744,161,280 bytes)
32768	~128 TB (140,737,488,322,560 bytes)
65536	~256 TB (281,474,976,645,120 bytes)

This second table is the same, except that the number of pointers allocated as direct, indirect, or double indirect has been set approximately equal:

Block Size	Maximum File Size
128	~0.73 MB (768,000 bytes)
256	~15.6 MB (16,301,568 bytes)
512	~298 MB (312,254,464 bytes)
1024	~4.95 GB (5,310,969,856 bytes)
2048	~82.5 GB (88,614,010,880 bytes)
4096	~1.31 TB (1,438,964,768,768 bytes)
8192	~21.1 TB (23,262,029,021,184 bytes)
16384	~64.0 TB (70,368,744,161,280 bytes)
32768	~128 TB (140,737,488,322,560 bytes)
65536	~256 TB (281,474,976,645,120 bytes)

16.1.2 Maximum Volume Sizes

Maximum volume size is mainly dependent on block size: larger block sizes allow larger volumes. Reliance Edge uses a free space bitmap, and for most common block sizes the limiting factor is the number of blocks which can be addressed by this bitmap. However, at very large block sizes (16 KB and up), the limiting factor is the number of blocks which can be addressed by 32-bit block numbers.

If using the POSIX-like API, the maximum volume size is as follows for each of the legal block sizes:

Block Size	Maximum Volume Size
128	84.0 MB (88,080,384 bytes)
256	840 MB (880,803,840 bytes)

Block Size	Maximum Volume Size
512	~7.27 GB (7,801,405,440 bytes)
1024	~61.0 GB (65,531,805,696 bytes)
2048	~500 GB (536,938,020,864 bytes)
4096	~3.95 TB (4,346,641,121,280 bytes)
8192	~31.8 TB (34,978,482,094,080 bytes)
16384	~64.0 TB (70,368,744,161,280 bytes)
32768	~128 TB (140,737,488,322,560 bytes)
65536	~256 TB (281,474,976,645,120 bytes)

If using the File System Essentials API, due to a very slight difference in on-disk metadata, the maximum volume size is a tiny bit larger for block sizes under 16 KB:

Block Size	Maximum Volume Size
128	87.5 MB (91,750,400 bytes)
256	855 MB (896,532,480 bytes)
512	~7.33 GB (7,866,417,152 bytes)
1024	~61.3 GB (65,796,046,848 bytes)
2048	~501 GB (538,003,374,080 bytes)
4096	~3.96 TB (4,350,919,311,360 bytes)
8192	~31.8 TB (34,995,628,408,832 bytes)
16384	~64.0 TB (70,368,744,161,280 bytes)
32768	~128 TB (140,737,488,322,560 bytes)
65536	~256 TB (281,474,976,645,120 bytes)

16.1.3 Maximum Path Length

Reliance Edge places no restrictions on maximum path length. Very deep directory structures are possible. The POSIX-like API uses path buffers that are always provided by the caller, and are never copied or stored, so the path length can as long as you want, provided there is sufficient memory for it. The inode count may also restrict how many directories can be created, limiting the path length.

16.1.4 Maximum Name Length

The maximum name length is configured via the `REDCONF_NAME_MAX` macro. The maximum legal size for `REDCONF_NAME_MAX` is block size minus four.

16.1.5 Allowable Characters in File Names

Reliance Edge places no restrictions on which characters or bytes are allowed in file names, except that a file name cannot contain a null byte (since it would terminate the string) or a path separator character (since that would terminate the file name). This means file names can contain UTF-8 byte sequences, unprintable characters, and characters like : or * which some file systems disallow.

16.2 Resource Consumption

This section explains Reliance Edge resource consumption and gives some examples. Example data was generated using the Keil toolset targeting ARM/Thumb and Cortex-M4.

Three configurations are presented in this section as examples. These settings are common between each of the configurations:

Setting	Value
Block Size	512 bytes
CRC Algorithm	Sarwate
Imap	External Only

FSE Configuration

The FSE configuration uses the FSE API with all optional APIs enabled and the minimum number of buffers (5).

Small POSIX Configuration

The Small POSIX configuration uses the POSIX-like API with all optional APIs enabled except rename, the minimum number of buffers (6), 10 handles and 10 tasks.

Full POSIX Configuration

The Full POSIX configuration uses the POSIX-like API with all optional APIs enabled, the minimum number of buffers (12), 10 handles and 10 tasks.

16.2.1 RAM Usage

RAM usage depends primarily on these configuration options:

- Block size
- Number of buffers

When using the POSIX-like API, these configuration options also affect RAM usage:

- Number of tasks
- Number of handles

Configuration	RAM Usage (bytes)
FSE	3898
Small POSIX	5321
Full POSIX	8449

Note

Some Reliance Edge ports implement an additional RAM buffer in port code. The [MQX block device service](#) and the STM32 SDIO implementation of the [FreeRTOS block device](#) service" increase the RAM requirement in this way.

16.2.2 ROM Size

ROM size includes binary code and read-only data, and depends on the API (FSE or POSIX-like) used, as well as which optional APIs and features are enabled.

Configuration	ROM Size (bytes)
FSE	11800
Small POSIX	17446
Full POSIX	18476

16.2.3 Maximum Stack Depth

Like ROM size, maximum stack depth depends on the API used and which optional APIs and features are enabled.

Configuration	Maximum Stack Depth (bytes)
FSE	440
Small POSIX	624
Full POSIX	704

16.3 Worst-Case I/O Counts

One of the features of Reliance Edge is that the run-times of its operations are very predictable. In particular, for several important file system operations, it is possible to compute the worst-case number of I/Os that will be issued. In some systems, where the worst-case response times to I/O requests can be determined, this means the worst-case response times of these file system operations can be determined also and as such their operations are deterministic. This may allow these operations to be used by tasks which have real-time requirements.

For example, suppose a periodic task needs to read 512 bytes of data (possibly at an unaligned offset) every time it runs, and the task is granted a 200 ms time slice by the scheduler, by which time the task must have finished its periodic work. Further suppose the block size of the Reliance Edge file system is 512, the buffer count is 6, and it takes the media at most 10 ms to read or write a sector. To complete the read, there will be at most 8 sector reads and 6 sector writes, which will take at most 140 ms. Assuming the CPU time can be shown to be less than 60 ms, then the task will complete within its allotted time slot.

Note

The equations below use file system blocks. If the block size is larger than the sector size, then each block I/O involves multiple sector I/Os, which should be accounted for when determining I/O response time.

Reliance Edge has a configuration option which allows reads and writes to be retried on failure (see [Block I/O Retries](#)). When this option is enabled, all worst case I/O counts are essentially multiplied by the maximum number of times an operation may be attempted (i.e. the configured number of retries plus one).

16.3.1 Mount

Mount reads exactly three blocks, every time. No other I/O is issued.

16.3.2 Transaction Point

A transaction point involves a sequence of four operations:

1. Dirty block buffers are written to disk.
2. The block device is flushed.
3. A metaroot block is written to disk.
4. The block device is flushed.

The first step involves a variable number of I/Os: all of the dirty block buffers are written to disk. However, the worst case is that all of the block buffers are dirty; so, at most, the first step writes `REDCONF_BUFFER_COUNT` blocks. Thus, the worst case I/O count for a transaction point is `REDCONF_BUFFER_COUNT` plus one block writes, along with the two flushes. No other I/O is issued.

However, if discard (trim) support is enabled, Reliance Edge must determine which blocks (if any) have become free after the transaction point and discard them. On larger volumes (ones that use an *external imap* for the free space bitmap), some or all of the free space bitmap must be read to determine which blocks have been freed. This operation is not deterministic, since in the worst case the entire free space bitmap must be read and any number of allocable blocks may need to be discarded.

16.3.3 File I/O

16.3.3.1 File Read

Let R denote the length of the read in bytes, B the block size, and C the buffer count. For the read operation, the worst-case number of read I/Os (r_{reads}) and write I/Os (r_{writes}) are as follows:

$$r_{reads} = \left\lceil \frac{R - 1}{B} \right\rceil + \left\lceil \frac{\lceil (R - 1)/B \rceil + 1}{(B - 20)/4} \right\rceil + \left\lceil \frac{\lceil (R - 1)/B \rceil + 1}{(B - 20)/4} \right\rceil + 5$$

$$r_{writes} = \min(C, r_{reads})$$

(A note on notation: $\lceil x \rceil$ denotes the "ceiling" of x , rounding it up to the nearest whole number. Furthermore, within these sections, division should be assumed to be real division, not truncated integer division. So $\frac{1}{2} = 0.5$.)

The equation for (r_{reads}) can be broken down, making it easier to understand, and to show how, in some circumstances, components of the equation can be subtracted.

First, let us pessimistically compute the number of file data blocks which will be read from disk:

$$r_{file} = \left\lceil \frac{R - 1}{B} \right\rceil + 1$$

The above accounts for unaligned reads which may cross block boundaries. Consider several key values (and assume B is 512 for illustration). When $R = 0$, $r_{file} = 0$. When $R = 1$, $r_{file} = 1$. When $R = 2$, $r_{file} = 2$ (since the two-byte

read could cross a block boundary). And not until $R = 514$ does $r_{file} = 3$, since a 514-byte read is the smallest read could cross two block boundaries: the last byte of one block, a whole block, then the first byte of the next block.

In order to compute the amount of metadata which may be read, let us define a constant, E , the number of block numbers that can be stored in an indirect or double indirect block (both blocks have 20 byte header and each block number requires 4 bytes):

$$E = \frac{B - 20}{4}$$

Now we can pessimistically compute the number of indirect and double indirect blocks which will be read from disk:

$$r_{indir} = \left\lceil \frac{r_{file} - 1}{E} \right\rceil + 1$$

$$r_{dindir} = \left\lceil \frac{r_{indir} - 1}{E} \right\rceil + 1$$

This is pessimistically assuming the read is at an offset where both indirects and double indirects need to be read. In some cases, this is not necessary. For example, if you have configured your inodes such that there are no double indirect entries, then r_{dindir} can be zero; likewise for r_{indir} if the inode consists solely of direct pointers. Technically, when the inode pointers are mixed, some offsets are cheaper to read than others; factoring in the offset would make this equation too complex, which is why we pessimistically assume the read starts at an offset where double indirects are in use.

The number of reads to buffer the inode block (assuming nothing relevant is buffered) is a constant:

$$r_{imap} = 1$$

$$r_{inode} = 1 + r_{imap}$$

(Technically, on small volumes, where the imap is *inlined*, $r_{imap} = 0$.)

Thus, we now have all the components to rewrite the equations for r_{reads} :

$$r_{reads} = r_{file} + r_{indir} + r_{dindir} + r_{inode}$$

When the terms are expanded and the constants combined, the above is the same as the original equation.

The equation for the worst-case number of writes was given earlier and is duplicated below:

$$r_{writes} = \min(C, r_{reads})$$

The explanation for r_{writes} is simple. Reading from a file never directly writes any blocks, but if there are dirty block buffers, they may be written to disk as they are replaced with buffers needed by the read (or flushed to disk to allow a multiblock read of file data). In the worst case, if all the block buffers are dirty, each block read might result in another write, up until every dirty block buffer has been written.

16.3.3.2 File Write

Note

The equations given below assume disk full automatic transaction points are disabled. If they are enabled, and the write encounters a disk full condition, some of the work detailed below happens twice. The simple correction is to double w_{writes} and w_{reads} .

The equations given below assume the buffer count **exceeds** the minimum buffer count by one or more. When the buffer count is exactly equal to the minimum buffer count, there is theoretically a very bad worst-case condition that could involve a very large number of reads. To determine whether your buffer count exceeds the minimum, open the [Configuration Utility](#), find the [Block Buffer Count](#) spinbox, and try decreasing the buffer count by one. If the configuration tool complains, you are already at the minimum, and these equations do not hold unless the buffer count is increased.

The worst-case I/O equations for file writes are too complicated to be legibly written as a single equation, so from the outset they will be broken up into their component parts. To keep things manageable, generally the most expensive path is considered at all points, even though these are sometimes mutually exclusive, to produce a worst-case I/O count which is accurate albeit pessimistic. Thus, the write is assumed to be overwriting existing transacted state data, since this is costlier than appending new data to a file.

Now for the parameters. Let W denote the length of the write in bytes, B the block size, C the buffer count, S the sector size, and Q the sector count. As before, in order to compute the amount of metadata affected, let us define a constant, E , the number of block numbers that can be stored in an indirect or double indirect block:

$$E = \frac{B - 20}{4}$$

Before considering the number of blocks which will be written to disk in the course of the write operation, let us first consider a superset, those blocks which are *dirtied* (updated) by the write.

The number of file data, indirect, double indirect, and inode blocks that are dirtied resembles the equations used in the previous read section. The main difference is that an additional file data, indirect, and double indirect block may be dirtied if the file size is being expanded from a size that is not block aligned, which may require zeroing data beyond the old end-of-file. (This is an example of where these series of equations are pessimistic: since if the file is being expanded, it is not being overwritten, as is assumed later on.)

$$\begin{aligned} d_{file} &= \left\lceil \frac{W - 1}{B} \right\rceil + 2 \\ d_{indir} &= \left\lceil \frac{d_{file} - 2}{E} \right\rceil + 2 \\ d_{dindir} &= \left\lceil \frac{d_{indir} - 2}{E} \right\rceil + 2 \\ d_{inode} &= 1 \end{aligned}$$

The free space bitmap (the imap) is updated during the course of the write. For each dirtied block, there are two imap updates: one to mark a new block as allocated, and (assuming existing data is being overwritten) another to mark the old block as free. These two changes may dirty separate inode blocks, except for the inode, since both inode blocks always reside in the same imap block. Since there is not guaranteed to be enough buffers to keep the relevant imap blocks buffered, each imap update could result in I/O, including repeated I/Os of the same block.

$$d_{imap} = 2(d_{file} + d_{indir} + d_{dindir}) + d_{inode}$$

Adding together those figures yields the total number of blocks dirtied:

$$d_{total} = d_{file} + d_{indir} + d_{dindir} + d_{inode} + d_{imap}$$

Not all of the dirtied blocks will be written; some will remain buffered at the conclusion of the operation. At a minimum, the inode block and the last used indirect and double indirect will still be buffered; the file data block might not be, if the write ended on an aligned boundary. All others may be potentially flushed by the last block allocation, which could churn the buffers in an expensive operation such as scanning the imap. So, in total, there are three blocks out of those dirtied which will definitely still be buffered. Thus, in the worst case, the number of dirtied blocks which will be written to disk is:

$$w_{wdirty} = d_{total} - 3$$

However, in the worst case, when the write operation started, there were already dirty block buffers in the buffer cache, and during the course of the operation, as those buffers were replaced, they are written to disk. In most cases this means every buffer in the buffer cache (C) could be written; but if the buffer count is high, the limiting factor is the number of buffer operations (reads or writes) that could result in a flush. The number of reads (w_{reads} ; computed below) accurately represents the number of buffer operations, since in the worst case, all block writes are preceded by block reads, except for aligned file data blocks. Thus, the worst-case number of block write I/Os resulting from a write is:

$$w_{writes} = w_{wdirty} + \min(C, w_{reads})$$

Next let us consider the number of block read I/Os that could result from a write. For indirect, double indirect, and inode blocks, this is the same as the number of blocks dirtied. For file data it is less, since when entire blocks of file are overwritten, there is no need to read the original file data beforehand. There are potentially two file data blocks which could be overwritten only partially and thus need to be read from disk first. A third case where a file data block is read would be when the file size is being expanded, but this is mutually exclusive with overwriting two blocks partially, and thus never are more than two file data blocks read.

$$w_{rfile} = 2$$

$$w_{rindir} = d_{indir}$$

$$w_{rdindir} = d_{dindir}$$

$$w_{rinode} = d_{inode}$$

Let us define N , the number of imap nodes in the imap bitmap (where each node comprises two blocks). The imap does not include bits for the first three blocks (master block and metaroots) or for the imap nodes themselves.

$$N = \left\lceil \frac{Q(B/S) - 3}{8(B - 16) + 2} \right\rceil$$

The total number of imap blocks read is given below. When allocating, the imap is scanned, and both blocks from each imap node may be read. The scan starts where the last allocation left off, and in the worst-case it wraps around to end in the same node. An imap block may also need to be read from disk to free a block; no scan is required, since the exact imap block which needs to be modified is known. The number of possibly freed blocks is equal to the number of dirtied blocks, sans the dirtied imap blocks.

$$w_{rimap} = 2(N + 1) + d_{file} + d_{indir} + d_{dindir} + d_{inode}$$

Adding everything together, the worst-case number of block read I/Os resulting from a write is:

$$w_{reads} = w_{rfile} + w_{rindir} + w_{rdindir} + w_{rinode} + w_{rimap}$$

Chapter 17

Use Case Guide

17.1 Introduction

How to Use This Guide: This collection of tips and techniques is designed to help you identify how to best configure Reliance Edge for your use case. Find the use case that most closely matches yours to use as your starting configuration and then feel free to experiment with the various settings using the tools provided with Reliance Edge to validate the results.

17.2 Use Case A: System Update

17.2.1 Description

Use when a group of files must be updated in a batch. For instance, critical system components are being upgraded, and the system may not function if the upgrade process is interrupted when only partially completed.

17.2.2 Goal

Update all of the files atomically, and as quickly as possible.

17.2.3 Planning

In order to perform an atomic update, the file system will require sufficient free space.

17.2.4 Approach

The file system is temporarily configured not to commit any transactions, some changes are made to the file system, and then the file system is restored to its normal transaction configuration and transacted to commit all changes at once.

17.2.5 Benefits

The update will be atomic: all changes will be committed or none of them. If power failure occurs before the final transaction completes, the file system will revert to its prior state upon reboot. Otherwise, all of the changes will be

persistent. By committing all of the changes at once, the operation will avoid having to write multiple transactions and will thus complete as fast as possible.

17.2.6 Process in Detail

Step 1: Disable Transactions

Disabling transactions can be done calling the API call `red_settransmask()` (or `RedFseTransMaskSet()` if the File System Essentials API is enabled) and passing `RED_TRANSACT_MANUAL` as the event mask.

After this operation, system calls will not result in transactions. The only way the file system will transact is via a manual transaction. Thus, it is imperative that the update application does not execute manual transactions while the update is in progress to ensure that the update is atomic.

Step 2: Perform the Update

What constitutes the update will vary, but it will involve some file write operations, and may involve file and directory creation and deleting. Because transactions have been disabled, none of the changes made during this step will be persisted if power is lost before it completes.

Step 3: Restore the Configuration

The configuration is restored by calling `red_settransmask()` (or `RedFseTransMaskSet()` if the File System Essentials API is enabled), passing in the normal transaction event mask.

Step 4: Transact the Update

The operation in Step 3 will reset the transaction mask but will not transact the file system. The file system can be manually transacted by calling `red_transact()` (or `RedFseTransact()` if the File System Essentials API is enabled).

17.3 Use Case B: Logging Application

17.3.1 Description

Substantial amounts of data are written to one or more log files over time. It is acceptable for the written data to be cached for a limited duration.

17.3.2 Goal

To ensure the data is committed sufficiently often for the use case while maximizing performance.

17.3.3 Planning

To properly tune the system for the logging application requires an awareness of the trade-offs. Transacting too often may lower throughput and increase flash wear. Transacting infrequently may put too much data at risk—a term for data sitting in caches on the disk which will be lost after power interruption. The ideal balance will depend upon the needs of the project.

17.3.4 Approach

Below, we consider two things which can be adjusted to control how often the logged data is written to disk and committed in a transaction point:

1. The transaction settings of Reliance Edge; and
2. The use of manual transaction calls in the application itself.

17.3.5 Process in Detail

Step 1: Transaction Configuration

When data is written to the disk, a new transaction point is needed for that data to become part of the committed state. This may occur through automatic transaction points or through manual sync or transaction calls. The transaction configuration can be set to determine whether automatic transaction points will occur.

If the transaction flag `RED_TRANSACT_WRITE` is set, then each write operation will be transacted automatically. This ensures all data written will be available in the event of power failure, but has a negative impact on performance. Other transaction settings allow for automatic transactions on other events, such as on file close.

Step 2: Manual Transaction

The application can manually place the logged data in the committed state by calling `red_transact()` (or `RedFseTransact()` if the File System Essentials API is enabled). If the transaction flag `RED_TRANSACT_FSYNC` is set, a `red_fsync()` call has the same effect by triggering an automatic transaction.

17.4 Use Case C: Controlling Data-at-Risk on Power Failure

17.4.1 Description

Critical data needs to be committed immediately to non-volatile media to minimize the risk of losing data remaining in the cache in the event of a power loss. The normal mechanisms which ensure data is eventually committed leave this important data at risk for too long.

17.4.2 Goal

To take the necessary steps for the data to be written to disk and committed in a transaction point.

17.4.3 Approach

File data is immediately written to the disk by Reliance Edge, but remains in an uncommitted state—since Reliance Edge is a copy-on-write file system, the data is not part of the committed state until a transaction point has been performed.

The application should call `red_transact` (or `RedFseTransact` if the File System Essentials API is enabled) to perform manual transaction as frequently as needed. To ensure critical data is transacted, one of these API functions should be called at regular intervals.

An alternate solution would be to transact automatically each time data is written—that is, each time `red_write()` is called (or `RedFseWrite()` if the File System Essentials API is enabled). This can be achieved by ensuring the transaction flag `RED_TRANSACT_WRITE` is set.

17.4.4 Benefits

After the process is complete, the data is guaranteed to be present on the media after power loss.

Chapter 18

Testing

18.1 Introduction

Reliance Edge comes with a number of tests that are designed to help you troubleshoot problems and provide a high degree of confidence that everything is functioning properly.

Tests can be found in the tests subdirectory of the Reliance Edge development tree. The win32 and the linux projects, found in the projects subdirectory of the Reliance Edge development tree, contain sample entry points for tests: e.g., fsstress_main.c.

If you are using the GPL version of Reliance Edge, the following tests are included:

- [fsstress](#)

If you are using the commercial version of Reliance Edge, the following tests are included:

- [BDevTest: Block Device Performance Test](#)
- [Disk Full Tests](#)
- [FSE API Test Suite](#)
- [FSE API Stress Test](#)
- [FSIOTest: File System Performance Test](#)
- [Multi-Volume Stress Test](#)
- [POSIX-like API Test Suite](#)
- [Stochastic Test](#)
- [OS-Specific API Test](#)

18.2 BDevTest: Block Device Performance Test

BDevTest is a performance test for the block device (that is, for the storage media). It does not use Reliance Edge or any other file system; it reads and writes to the storage media directly, using the [block device OS service](#). It is useful for understanding the performance characteristics of a storage medium (for example, how the disk handles random writes versus sequential writes, or small versus large writes), which in turn is useful in analyzing file system performance (such

as FSIOTest results). BDevTest is also useful for configuring Reliance Edge; for example, if the block device is flash memory, the performance results of various write sizes can be used to determine the internal page size, which is often an optimal choice for the Reliance Edge [block size](#).

Although a Win32 or Linux executable is built for this test, the performance data it gathers is not very meaningful unless it is run from the target hardware. This holds true even if using removable media; the caching which exists in a desktop operating system such as Windows or Linux is drastically more advanced than that of a typical embedded system, leading to very different results even with the same media.

This test is destructive—since it writes directly to the media, it will most likely corrupt any file system format which is there, and the media will need to be reformatted. However, most of the file system tests reformat the media, so usually another test can be run right after BDevTest without manually reformatting.

Users familiar with the version of BDevTest included in Reliance Nitro and FlashFX should note that this is an adapted version which includes only a subset of the tests and features of the original test.

18.2.1 Command-Line Syntax

The syntax of BDevTest is as follows:

```
bdevtest VolumeID Tests [Options]
```

The parameter **VolumeID** specifies the target volume. Although BDevTest does not use the file system, it still looks at the volume configuration to determine the sector size and sector count of the media. Under the FSE API configuration, the **VolumeID** must be the unsigned integer volume number. If the POSIX API is enabled, a volume path prefix (such as "VOL1:") will also be accepted.

Tests is a stand-in for one or more of the following test parameters:

```
--all, -A
    Run all of the tests implemented by BDevTest.
--rand[=ops], -R[ops]
    Run the random I/O tests. ops is an optional argument which may be either
    'r' or 'w' or 'd' to run just the read or write or discard portion of the
    test.
--seq=[ops], -S[ops]
    Run the sequential I/O tests. See the above description of 'ops'.
```

Options can be any of the following:

```
--count=sects[:max], -c sects[:max]
    The count of sectors per I/O operation. If a max sectors value is
    provided, then the test will iterate and double the count each time. The
    default value is 1:16.
--max=size, -m size
    The maximum amount of data to touch, starting at sector 0. If not
    specified, the test uses all the space remaining on the device.
--passes=n, -p n
    Specifies the number of passes to perform over the data range specified by
    --max (per iteration). The default is 1. Specify 0 to cause the passes
    to be treated as infinite -- the iteration will quit only when the time
    limit has been reached.
--time=n[s|m|h], -t n[s|m|h]
    Specifies the amount of time (in seconds, minutes, or hours) to run each
    iteration (default is 90 seconds). Specify 0 to cause time to be ignored;
    the iteration will quit when the passes limit has been reached.
```

```
--verify, -v
    Causes the sequential read test to verify that the sector data read
    matches what was written.
--async, -a
    Causes the write tests to be potentially asynchronous by not flushing
    after each write.
--no-wipe, -w
    Prevents the media from being wiped (discarded) between tests. Only
    meaningful if the block device supports discards.
--sample-rate=rate, -r rate
    Specifies the sample rate in seconds at which intermediate results will be
    displayed. The default is 2, 0 to disable.
--seed=n, -s n
    Specifies the random seed to use (default is 12345678; 0 for timestamp).
--dev=devname, -D devname
    Specifies the device name. This is typically only meaningful when
    running the test on a host machine. This can be "ram" to test on a RAM
    disk, the path and name of a file disk (e.g., red.bin); or an OS-specific
    reference to a device (on Windows, a drive letter like G: or a device name
    like \\.\PhysicalDrive7; on Linux, a device file like /dev/sdb).
--help, -H
    Prints this usage text and exits.
```

18.3 Disk Full Tests

The Reliance Edge disk full tests validate the file system when the disk is full and nearly full. These tests are destructive and should only be run on a volume which does not contain important data. At present, this test only runs on the POSIX-like API.

18.3.1 Command-Line Syntax

The syntax of the Disk Full Tests command is as follows:

```
diskfull VolumeID [Options]
```

The parameter `VolumeID` specifies the target volume. This can be an unsigned integer volume number or a volume path prefix (such as "VOL1:").

The optional `Options` parameters are as follows:

```
--quit-on-failure, -q
    Stop the test once a failure has been encountered.
--debug, -d
    Break into the debugger (where supported) on error.
--dev=devname, -D devname
    Specifies the device name. This is typically only meaningful when
    running the test on a host machine. This can be "ram" to test on a RAM
    disk, the path and name of a file disk (e.g., red.bin); or an OS-specific
    reference to a device (on Windows, a drive letter like G: or a device name
    like \\.\PhysicalDrive7; on Linux, a device file like /dev/sdb).
--help, -H
    Prints this usage text and exits.
```

18.4 FSE API Test Suite

The Reliance Edge FSE API Test Suite validates the File System Essentials API. These tests are destructive and should only be run on a volume which does not contain important data.

18.4.1 Command-Line Syntax

The syntax of the FSE API Test Suite command is as follows:

```
fsetest VolumeID [Options]
```

The parameter `VolumeID` specifies the target volume. This must be an unsigned integer volume number. For example, if your configuration defines two volumes, valid volume numbers will be "0" for the first disk and "1" for the second disk.

Options can be any of the following:

```
--quit-on-failure, -q
    Stop the test once a failure has been encountered.
--debug, -d
    Break into the debugger (if possible) on error.
--dev=devname, -D devname
    Specifies the device name. This is typically only meaningful when
    running the test on a host machine. This can be "ram" to test on a RAM
    disk, the path and name of a file disk (e.g., red.bin); or an OS-specific
    reference to a device (on Windows, a drive letter like G: or a device name
    like \\.\PhysicalDrive7; on Linux, a device file like /dev/sdb).
--help, -H
    Prints this usage text and exits.
```

18.5 FSE API Stress Test

This is a stress test which exercises the file system via the FSE API. This test is destructive and should only be run on a volume which does not contain important data.

18.5.1 Command-Line Syntax

The syntax of the FSE API Stress Test command is as follows:

```
fse_stress VolumeID [Options]
```

The parameter `VolumeID` specifies the target volume. This must be an unsigned integer volume number. For example, if your configuration defines two volumes, valid volume numbers will be "0" for the first disk and "1" for the second disk.

Options can be any of the following:

```
--files=n, -f n
    The number of files to use in the test (default 2).
--max=size, -m size
    The maximum size of each file during the test (default 4096KB).
--buffer-size=size, -b size
    The buffer size to allocate, which will be the maximum size for read and
    write operations (default 64KB).
```

```
--nops=count, -n count
    Specifies the number of operations to run (default 1000000).
--loops=count, -l count
    Specifies the number of times the entire test should loop. Each time a
    new loop starts, the seed is incremented and printed. Using a large
    --loops value and a relatively small --nops value is useful for finding
    problems that can be reproduced quickly. Use 0 for infinite. Default 1.
--sample-rate=rate, -r rate
    How many seconds to wait before displaying an indication of progress.
    Higher values reduce log file sizes. Use 0 to display every iteration;
    default is 2.
--seed=n, -s n
    Specifies the random seed to use (default is 1; 0 to use timestamp).
--dev=devname, -D devname
    Specifies the device name. This is typically only meaningful when
    running the test on a host machine. This can be "ram" to test on a RAM
    disk, the path and name of a file disk (e.g., red.bin); or an OS-specific
    reference to a device (on Windows, a drive letter like G: or a device name
    like \\.\PhysicalDrive7; on Linux, a device file like /dev/sdb).
--help, -H
    Prints this usage text and exits.
```

18.6 FSIOTest: File System Performance Test

This is a file system performance test, capable of testing the performance of a variety of file system operations, but with a focus on read and write operations. FSIOTest uses a file system abstraction, so (with a bit of effort) it can be run against file systems other than Reliance Edge.

Although a Win32 or Linux executable is built for this test, the performance data it gathers is not very meaningful unless it is run from the target hardware. This holds true even if using removable media; the caching which exists in a desktop operating system such as Windows or Linux is drastically more advanced than that of a typical embedded system, leading to very different results even with the same media.

These tests are destructive and should only be run on a volume which does not contain important data.

Users familiar with the version of FSIOTest included in Reliance Nitro and FlashFX should note that this is an adapted version which includes only a subset of the tests and features of the original test.

18.6.1 Command-Line Syntax

The syntax of the FSIOTest command is as follows:

```
fsiotest VolumeID Tests [Options]
```

The parameter `VolumeID` specifies the target volume. This can be an unsigned integer volume number or a volume path prefix (such as "VOL1:").

`Tests` is a stand-in for one or more of the following test parameters:

```
--all, -A
    Run all of the tests implemented by FSIOTest.
--rand[=ops], -R[ops]
    Run the random I/O tests. ops is an optional argument which may be either
    'r' or 'w' to run just the read or write portion of the test.
--seq=[ops], -S[ops]
```

```

Run the sequential I/O tests. ops is an optional argument which may be
'r', 'w', or 'e' to run just the read, write, or rewrite portion of the
test.
--mixed, -M
    Run the mixed (alternating random and sequential) write I/O tests.
--scan, -N
    Run the directory scan tests.

```

Options can be any of the following:

```

--buffer-size=size, -b size
    Specifies the I/O buffer size (default is 1/128th of --max-size).
--max-size, -m size
    The maximum test file size (default is 1/16th free disk space).
--sample-rate=rate, -r rate
    Specifies the sample rate in seconds, for those tests which support
    sampling. Specify 0 to sample every buffer processed. Default is 2,
    maximum is 3600.
--seed=n, -s n
    Specifies the random seed to use (default is 12345678; 0 for timestamp).
--scan-files=n, -f n
    The number of files to use for the --scan test (default=1000).
--rand-fow=n, -u n
    Flush-on-write, flush after every Nth (default=128) write in the --rand
    and --mixed test. If negative, flush after a random number of writes
    modulus the absolute value of the argument to this parameter.
--rand-pass=r:w, -p r:w
    Random test passes to perform for reads and writes. Defaults to 2 for
    reads and 1 for writes.
--mixed-pass=n, -P n
    Mixed test passes to perform. Defaults to 4.
--start=n, -o n
    This option causes the --seq test to iterate through a range of I/O buffer
    sizes, starting with 'n' up to the size specified by the --buffer-size
    parameter, incrementing by doubling the value. 'n' is always specified
    in bytes. If not specified, 'n' defaults to the buffer size.
--verify, -v
    Read back and verify writes (will skew performance results.)
--block-size=n, -z n
    The file system block size to use, in bytes. If not specified, the block
    size is detected automatically. Used as a unit of operation for some
    tests, including --rand and --seq.
--fs=name, -F name
    Picks which file system to run against. Options are "RED" (Reliance
    Edge, the default), and "FatFs". Options other than "RED" are not
    supported out of the box.
--dev=devname, -D devname
    Specifies the device name. This is typically only meaningful when
    running the test on a host machine. This can be "ram" to test on a RAM
    disk, the path and name of a file disk (e.g., red.bin); or an OS-specific
    reference to a device (on Windows, a drive letter like G: or a device name
    like \\.\PhysicalDrive7; on Linux, a device file like /dev/sdb).
--help, -H
    Prints this usage text and exits.
--xhelp, -J
    Prints an extended description of the --rand, --seq, --mixed, and --scan
    tests; and then exits.

```

18.7 Multi-Volume Stress Test

This is a file system exerciser for the POSIX-like API. It can be provided with multiple volume IDs, in which case it mounts and exercises all of the volumes. This test is destructive and should only be run on volumes which do not contain important data.

18.7.1 Command-Line Syntax

The syntax of the Multi-Volume Stress Test command is as follows:

```
mvstress test VolumeID [AdditionalVolumeIDs...] [Options]
```

The parameter `VolumeID` specifies the target volume. This can be an unsigned integer volume number or a volume path prefix (such as "VOL1:"). Unlike the other tests, you can specify multiple volume IDs and the test will exercise all of those volumes.

Options can be any of the following:

```
--file-count=n, -f n
    The number of files to use on each volume (default 4).
--file-size=size, -z size
    The size of each file during the test (default 256KB).
--buffer-size=size, -b size
    The buffer size to allocate, which will be the maximum size for read and
    write operations (default 16KB).
--iterations=count, -n count
    Specifies the number of test iterations to run (default 10000).
--seed=n, -s n
    Specifies the random seed to use (default is 1; 0 to use timestamp).
--dev=devname, -D devname
    Specifies device names for the test volumes. Because this is a multivolume
    test, this parameter may be specified multiple times: the device names are
    associated with the volumes in the order they are given on the command line.
    For example, the first device name is associated with the first volume ID,
    the second device name with the second volume ID, etc. Device names are
    typically only meaningful when running the test on a host machine. This can
    be "ram" to test on a RAM disk, the path and name of a file disk (e.g.,
    red.bin); or an OS-specific reference to a device (on Windows, a drive
    letter like G: or a device name like \.; on Linux, a
    device file like /dev/sdb).
--help, -H
    Prints this usage text and exits.
```

18.8 fsstress Test

This is a version of fsstress, a (quite good) file system stress test written by SGI. It has been modified to use the Reliance Edge POSIX-like API, and to be single-threaded. This test is destructive and should only be run on a volume which does not contain important data.

18.8.1 Command-Line Syntax

The syntax of the fsstress test command is as follows:

```
fsstress VolumeID [Options]
```

The parameter `VolumeID` specifies the target volume. This can be an unsigned integer volume number or a volume path prefix (such as "VOL1:").

Options can be any of the following:

```
--no-cleanup, -c
    Specifies not to remove files (cleanup) after execution
--loops=count, -l count
    Specifies the number of times the entire test should loop. Use 0 for
    infinite. Default 1.
--nops=count, -n count
    Specifies the number of operations to run (default 10000).
--namepad, -r
    Specifies to use random name padding.
--seed=value, -s value
    Specifies the seed for the random number generator (default timestamp).
--verbose, -v
    Specifies verbose mode (without this, test is very quiet).
--dev=devname, -D devname
    Specifies the device name. This is typically only meaningful when
    running the test on a host machine. This can be "ram" to test on a RAM
    disk, the path and name of a file disk (e.g., red.bin); or an OS-specific
    reference to a device (on Windows, a drive letter like G: or a device name
    like \\.\PhysicalDrive7; on Linux, a device file like /dev/sdb).
--help, -H
    Prints this usage text and exits.
```

18.9 POSIX-like API Test Suite

The Reliance Edge POSIX-like API Test Suite validates the POSIX-like API. These tests are destructive and should only be run on a volume which does not contain important data.

18.9.1 Command-Line Syntax

The syntax of the POSIX-like API Test Suite command is as follows:

```
posixtest VolumeID [Options]
```

The parameter `VolumeID` specifies the target volume. This can be an unsigned integer volume number or a volume path prefix (such as "VOL1:").

Options can be any of the following:

```
--quick, -u
    Skip tests that are slow on large volumes (like disk full tests).
--quit-on-failure, -q
    Stop the test once a failure has been encountered.
--debug, -d
    Break into the debugger (if possible) on error.
--no-corrupt, -k
    Skip tests which deliberately introduce temporary metadata corruption.
    Normally there is no need to specify this option; it exists to prevent
```

```

failures in the simulated power interruption tests.

--dev=devname, -D devname
    Specifies the device name. This is typically only meaningful when
    running the test on a host machine. This can be "ram" to test on a RAM
    disk, the path and name of a file disk (e.g., red.bin); or an OS-specific
    reference to a device (on Windows, a drive letter like G: or a device name
    like \\.\PhysicalDrive7; on Linux, a device file like /dev/sdb).

--help, -H
    Prints this usage text and exits.

```

18.10 OS-Specific API Test Suite

The Reliance Edge OS-specific API Test Suite validates the integration of Reliance Edge with the native Virtual File System (VFS). It runs tests on standardized C APIs such as `fopen()`, `fread()`, and `POSIX mkdir()`; and may also test other OS-specific interfaces (such as `ioctl` filesystem commands). This test is not built directly as part of the Win32 or Linux projects.

Currently the OS API test is only implemented for two platforms: ARM mbed and Linux. However, the test may be easily ported to other platforms that support Reliance Edge and implement a C Standard Library VFS. To run the OS API test on a new platform, the file `redtosapiconfig.h` should be copied and adapted for the platform. This file can be found in the folder `os/mbed/include`, and is well commented to explain its usage.

The OS-specific API test is ported to Linux for the purpose of verifying the behavior of the [Reliance Edge FUSE driver](#). To do this, `redfuse` should first be run to mount a Reliance Edge volume, and then the OS API test should be run against the mountpoint. To build the OS API test for Linux, `cd` to `projects/linux/host/fusetest` and run `make`.

18.10.1 Command-Line Syntax

The OS API test is not run from a command line on ARM mbed. The syntax for running it on Linux is as follows:

```
./osapitest Mountpoint [Options]
```

The parameter `Mountpoint` specifies the folder supplied to `redfuse` as a mountpoint—i.e. a folder that is currently mounted as Reliance Edge FUSE volume.

`Options` can be any of the following:

```

--quick, -u
    Skip tests that are slow on large volumes (like disk full tests).
--quit-on-failure, -q
    Stop the test once a failure has been encountered.
--debug, -d
    Break into the debugger (if possible) on error.
--verbose, -v
    Specifies verbose mode (sets verbosity to obnoxious level).
--help, -H
    Prints this usage text and exits.

```

18.11 Stochastic Test

The Stochastic Test exercises the POSIX-like API in a random fashion. Informally this test is also called the "monkey", since it monkeys with the file system. This test is destructive and should only be run on a volume which does not contain important data.

18.11.1 Command-Line Syntax

The syntax of the Stochastic Test is as follows:

```
stochposix VolumeID [Options]
```

The parameter `VolumeID` specifies the target volume. This can be an unsigned integer volume number or a volume path prefix (such as "VOL1:").

Options can be any of the following:

```
--iterations=count, -i count
    The number of iterations to run. Default is 10000000.
--files=count, -n count
    The number of tracked files. Used to select a random file for the tests
    that require one. Only the given number of files may exist on the volume
    at a time. Default is 128.
--dirs=count, -m count
    The number of tracked directories. Used to select a random directory for
    the tests that require one. Only the given number of directories may
    exist on the volume at a time. Default is 64.
--open-files=count, -o count
    The maximum number of open files at a time. Used to select a random open
    file for those tests that require it. Default is 2.
--open-dirs=count, -p count
    The maximum number of open directories at a time. Used to select a random
    open directory for those tests that require one. Default is 2.
--seed=n, -s n
    Specifies the random seed to use (default is 53; 0 to use timestamp).
--dev=devname, -D devname
    Specifies the device name. This is typically only meaningful when
    running the test on a host machine. This can be "ram" to test on a RAM
    disk, the path and name of a file disk (e.g., red.bin); or an OS-specific
    reference to a device (on Windows, a drive letter like G: or a device name
    like \\.\PhysicalDrive7; on Linux, a device file like /dev/sdb).
--help, -H
    Prints this usage text and exits.
```

18.12 Running the Tests on Your Target Hardware

All of the tests described above can be run from the host machine, such as a Windows machine using the projects/win32 project. This is a convenient way of running the tests, but it does not provide the same level of assurance as running them on the target hardware. The command-line syntax given above is convenient for interactive environments, but often on the target hardware the tests will be run non-interactively. All of the tests are designed so that they can be invoked with all of their options specified programmatically and non-interactively. For example, the below code demonstrates running `fsstress`:

```
#include <redposix.h>
#include <redtests.h>

void test(void)
{
    FSSTRESSPARAM param;
    int status;

    // Fill in the default fsstress parameters.
    //
    FsstressDefaultParams(&param);
```

```
// Override default parameters as desired.  
//  
param.fVerbose = true;  
param.ulSeed = 42U;  
  
// Start the test.  
//  
status = FsstressStart(&param);  
}
```

All of the tests described in this chapter have a `*DefaultParams()` and `*Start()` function. The prototypes and parameter structures are all in `include/redtests.h`.

One pitfall to be aware of is that some tests (like `fsstress`) expect the driver to be initialized and the volume mounted, while others (like the POSIX-like API Test Suite) expect an uninitialized driver. The example entry points in `projects/win32` (the `*_main.c` files) provide examples of setting things up like the respective `*Start()` functions expect.

18.13 Specialized Test Projects

The commercial kit also includes the below specialized test projects:

18.13.1 Simulated Power Interruption Project

This project, found in `projects/powerint`, tests the integrity of the file system after power interruption in a Win32 simulation environment. See `projects/powerint/README.txt` for details.

18.13.2 I/O Error Injection Project

This project, found in `projects/errinject`, tests the behavior of the file system when I/O errors are encountered by injecting I/O errors in a Win32 simulation environment. See `projects/errinject/README.txt` for details.

18.13.3 Discard Tests Project

This subproject, found in `projects/win32/discardtests`, builds a copy of the Stochastic test and the FSE Stress test using a modified ramdisk block device implementation. The modified ramdisk reports an error if the filesystem attempts to read invalid data from a discarded sector. See `projects/win32/discardtests/README.txt` for details.

18.13.4 FSTrim Test Project

This project, found in `projects/fstrim`, tests the use case where `red_fstrim()`, in combination with a "sanitize" storage device command, is used to [eliminate data remanence](#). It implements a RAM disk with a simulated "sanitize" command.

Chapter 19

Module Index

19.1 API References

The following two API sets are available for use with Reliance Edge.

The POSIX-like File System Interface	155
The File System Essentials Interface	157

Chapter 20

Hierarchical Index

20.1 Class Hierarchy

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

BDEVINFO	159
RED_DIR_PARAM	160
red_dirent	160
RED_RENAME_PARAM	160
REDDIRECT	161
RedDirHandle	162
RedFileHandle	163
RedFileSystem	168
RedMemFileSystem	175
RedSDFileSystem	176
REDSTAT	182
REDSTATFS	184

Chapter 21

Data Structure Index

21.1 Data Structures

Here are the data structures with brief descriptions:

BDEVINFO	Block device geometry information	159
RED_DIR_PARAM	Structure used as an argument for the directory traversing MQX IOCTLs	160
red_dirent	Extends the POSIX dirent structure on ARM mbed by adding REDDIR information for the current directory entry	160
RED_RENAME_PARAM	Structure used as an argument for the MQX IOCTLs RED_IOCTL_RENAME and RED_IOCTL_LINK	160
REDDIRECT	Directory entry information	161
RedDirHandle	The Reliance Edge file handle type for ARM mbed	162
RedFileHandle	The Reliance Edge file handle type for ARM mbed	163
RedFileSystem	The core implementation of the ARM mbed port of Tuxera's Reliance Edge failsafe filesystem for IoT	168
RedMemFileSystem	A ramdisk implementation of the ARM mbed port of Tuxera's Reliance Edge failsafe filesystem for IoT	175
RedSDFileSystem	An SD/MMC implementation of the ARM mbed port of Tuxera's Reliance Edge failsafe filesystem for IoT	176
REDDIR	Status information on an inode	182
REDDIRFS	Status information on a file system volume	184

Chapter 22

File Index

22.1 File List

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

fse/fse.c	Implementation of the Reliance Edge FSE API	189
include/redapimacs.h	Defines macros used to interact with the Reliance Edge API	198
include/rederrno.h	Error values for Reliance Edge APIs	202
include/redfse.h	Interface for the Reliance Edge FSE API	208
include/redoserv.h	217
include/redposix.h	Interface for the Reliance Edge POSIX-like API	226
include/redstat.h	257
os/mbed/RedFileSystem/RedDirHandle.cpp	Implementation of the RedDirHandle type for Reliance Edge on ARM mbed	258
os/mbed/RedFileSystem/RedDirHandle.h	Definition of the RedDirHandle type for Reliance Edge on ARM mbed	258
os/mbed/RedFileSystem/RedFileHandle.cpp	Implementation of the RedFileHandle type for Reliance Edge on mbed	258
os/mbed/RedFileSystem/RedFileHandle.h	Declaration of the RedFileHandle type for Reliance Edge on ARM mbed	259
os/mbed/RedFileSystem/RedFileSystem.cpp	Implementation of the RedFileSystem type for Reliance Edge on ARM mbed	259
os/mbed/RedFileSystem/RedFileSystem.h	Declaration of the RedFileSystem type for Reliance Edge on ARM mbed	259
os/mbed/RedMemFileSystem.h	Implementation of the RedMemFileSystem type for Reliance Edge on ARM mbed	260
os/mbed/RedSDFileSystem/RedSDFileSystem.cpp	Implementation of the RedSDFileSystem type for Reliance Edge on ARM mbed	260
os/mbed/RedSDFileSystem/RedSDFileSystem.h	Declaration of the RedSDFileSystem type for Reliance Edge on ARM mbed	260
os/mqx/include/redfs_mqx.h	This file includes the headers needed to use Reliance Edge on MQX and declares any public MQX-specific Reliance Edge methods	261

os/mqx/vfs/ redfs_fio.c	Implements the VFS integration for Reliance Edge in the MQX Formatted I/O library	268
os/mqx/vfs/ redfs_nio.c	Implements the VFS integration for Reliance Edge in the MQX NIO library	268
os/stub/include/ redostypes.h	Defines OS-specific types for use in common code	270
os/stub/services/ osassert.c	Implements assertion handling	270
os/stub/services/ osbdev.c	Implements block device I/O	271
os/stub/services/ osclock.c	Implements real-time clock functions	276
os/stub/services/ osmutex.c	Implements a synchronization object to provide mutual exclusion	278
os/stub/services/ osoutput.c	Implements outputting a character string	280
os/stub/services/ ostask.c	Implements task functions	280
os/stub/services/ ostimestamp.c	Implements timestamp functions	281
posix/ posix.c	Implementation of the Reliance Edge POSIX-like API	283
projects/doxygen/ redconf.h	310
projects/doxygen/ redtypes.h	Defines basic types used by Reliance Edge	318
projects/mqx/TWR-K65F180M/RelianceEdge/ redosconf.c	321

Chapter 23

Module Documentation

23.1 The POSIX-like File System Interface

Functions

- `int32_t red_init (void)`
Initialize the Reliance Edge file system driver.
- `int32_t red_uninit (void)`
Uninitialize the Reliance Edge file system driver.
- `int32_t red_sync (void)`
Commits file system updates.
- `int32_t red_mount (const char *pszVolume)`
Mount a file system volume.
- `int32_t red_mount2 (const char *pszVolume, uint32_t ulFlags)`
Mount a file system volume with flags.
- `int32_t red_umount (const char *pszVolume)`
Unmount a file system volume.
- `int32_t red_umount2 (const char *pszVolume, uint32_t ulFlags)`
Unmount a file system volume with flags.
- `int32_t red_format (const char *pszVolume)`
Format a file system volume.
- `int32_t red_transact (const char *pszVolume)`
Commit a transaction point.
- `int32_t red_settransmask (const char *pszVolume, uint32_t ulEventMask)`
Update the transaction mask.
- `int32_t red_gettransmask (const char *pszVolume, uint32_t *pulEventMask)`
Read the transaction mask.
- `int32_t red_statvfs (const char *pszVolume, REDSTATFS *pStatvfs)`
Query file system status information.
- `int32_t red_ftrim (const char *pszVolume, uint32_t ulBlockStart, uint32_t ulBlockCount)`
Discard (trim) free blocks within a range of a volume's blocks.
- `int32_t red_open (const char *pszPath, uint32_t ulOpenMode)`
Open a file or directory.
- `int32_t red_unlink (const char *pszPath)`

- `int32_t red_mkdir (const char *pszPath)`
Create a new directory.
- `int32_t red_rmdir (const char *pszPath)`
Delete a directory.
- `int32_t red_rename (const char *pszOldPath, const char *pszNewPath)`
Rename a file or directory.
- `int32_t red_link (const char *pszPath, const char *pszHardLink)`
Create a hard link.
- `int32_t red_close (int32_t iFildes)`
Close a file descriptor.
- `int32_t red_read (int32_t iFildes, void *pBuffer, uint32_t ulLength)`
Read from an open file.
- `int32_t red_write (int32_t iFildes, const void *pBuffer, uint32_t ulLength)`
Write to an open file.
- `int32_t red_fsync (int32_t iFildes)`
Synchronizes changes to a file.
- `int64_t red_lseek (int32_t iFildes, int64_t llOffset, REDWHENCE whence)`
Move the read/write file offset.
- `int32_t red_ftruncate (int32_t iFildes, uint64_t ullSize)`
Truncate a file to a specified length.
- `int32_t red_fstat (int32_t iFildes, REDSTAT *pStat)`
Get the status of a file or directory.
- `REDDIR * red_opendir (const char *pszPath)`
Open a directory stream for reading.
- `REDDIRENT * red_readdir (REDDIR *pDirStream)`
Read from a directory stream.
- `void red_rewinddir (REDDIR *pDirStream)`
Rewind a directory stream to read it from the beginning.
- `int32_t red_closedir (REDDIR *pDirStream)`
Close a directory stream.
- `int32_t red_chdir (const char *pszPath)`
Change the current working directory (CWD).
- `char * red_getcwd (char *pszBuffer, uint32_t ulBufferSize)`
Get the path of the current working directory (CWD).
- `REDSTATUS * red_errnoptr (void)`
Pointer to where the last file system error (`errno`) is stored.

23.2 The File System Essentials Interface

Functions

- **REDSTATUS RedFseInit (void)**
Initialize the Reliance Edge file system driver.
- **REDSTATUS RedFseUninit (void)**
Uninitialize the Reliance Edge file system driver.
- **REDSTATUS RedFseMount (uint8_t bVolNum)**
Mount a file system volume.
- **REDSTATUS RedFseUnmount (uint8_t bVolNum)**
Unmount a file system volume.
- **REDSTATUS RedFseFormat (uint8_t bVolNum)**
Format a file system volume.
- **int32_t RedFseRead (uint8_t bVolNum, uint32_t ulFileNum, uint64_t ullFileOffset, uint32_t ulLength, void *pBuffer)**
Read from a file.
- **int32_t RedFseWrite (uint8_t bVolNum, uint32_t ulFileNum, uint64_t ullFileOffset, uint32_t ulLength, const void *pBuffer)**
Write to a file.
- **REDSTATUS RedFseTruncate (uint8_t bVolNum, uint32_t ulFileNum, uint64_t ullNewFileSize)**
Truncate a file (set the file size).
- **int64_t RedFseSizeGet (uint8_t bVolNum, uint32_t ulFileNum)**
Retrieve the size of a file.
- **REDSTATUS RedFseTransMaskSet (uint8_t bVolNum, uint32_t ulEventMask)**
Update the transaction mask.
- **REDSTATUS RedFseTransMaskGet (uint8_t bVolNum, uint32_t *pulEventMask)**
Read the transaction mask.
- **REDSTATUS RedFseTransact (uint8_t bVolNum)**
Commit a transaction point.

Chapter 24

Data Structure Documentation

24.1 BDEVINFO Struct Reference

Block device geometry information.

Data Fields

- `uint32_t ulSectorSize`
The sector size for the block device: the basic unit for reading and writing to the storage media.
- `uint64_t ullSectorCount`
The number of sectors in this block device.

24.1.1 Detailed Description

Block device geometry information.

Definition at line 21 of file redoserv.h.

24.1.2 Field Documentation

24.1.2.1 ulSectorSize

`uint32_t ulSectorSize`

The sector size for the block device: the basic unit for reading and writing to the storage media.

This value is either taken from the #VOLCONF, or queried from the block device.

Definition at line 27 of file redoserv.h.

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

- `include/redoserv.h`

24.2 RED_DIR_PARAM Struct Reference

Structure used as an argument for the directory traversing MQX IOCTLs.

24.2.1 Detailed Description

Structure used as an argument for the directory traversing MQX IOCTLs.

Definition at line 161 of file [redfs_mqx.h](#).

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

- [os/mqx/include/redfs_mqx.h](#)

24.3 red_dirent Struct Reference

Extends the POSIX dirent structure on ARM mbed by adding [REDSSTAT](#) information for the current directory entry.

Inherits dirent.

Data Fields

- [REDSSTAT * d_stat](#)
Pointer to file information (extension)

24.3.1 Detailed Description

Extends the POSIX dirent structure on ARM mbed by adding [REDSSTAT](#) information for the current directory entry.

Definition at line 57 of file [RedDirHandle.h](#).

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

- [os/mbed/RedFileSystem/RedDirHandle.h](#)

24.4 RED_RENAME_PARAM Struct Reference

Structure used as an argument for the MQX IOCTLs [RED_IOCTL_RENAME](#) and [RED_IOCTL_LINK](#).

24.4.1 Detailed Description

Structure used as an argument for the MQX IOCTLs [RED_IOCTL_RENAME](#) and [RED_IOCTL_LINK](#).

Definition at line 152 of file [redfs_mqx.h](#).

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

- [os/mqx/include/redfs_mqx.h](#)

24.5 REDDIRENT Struct Reference

Directory entry information.

Data Fields

- `uint32_t d_ino`
File serial number (inode number).
- `char d_name [12U+1U]`
Name of entry.
- `REDSSTAT d_stat`
File information (POSIX extension).

24.5.1 Detailed Description

Directory entry information.

Definition at line 103 of file redposix.h.

24.5.2 Field Documentation

24.5.2.1 d_ino

`uint32_t d_ino`

File serial number (inode number).

Definition at line 105 of file redposix.h.

24.5.2.2 d_name

`char d_name[12U+1U]`

Name of entry.

Definition at line 106 of file redposix.h.

Referenced by RedDirHandle::readdir().

24.5.2.3 d_stat

`REDSSTAT d_stat`

File information (POSIX extension).

Definition at line 107 of file redposix.h.

Referenced by RedDirHandle::readdir().

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

- include/redposix.h

24.6 RedDirHandle Class Reference

The Reliance Edge file handle type for ARM mbed.

Inherits DirHandle.

Public Member Functions

- virtual int [closedir\(\)](#)
Close the directory handle.
- virtual struct [red_dirent](#) * [readdir\(\)](#)
Return the next directory entry, and move to the next entry in the directory stream.
- virtual void [rewinddir\(\)](#)
Rewind a directory stream to read it from the beginning.
- virtual off_t [telldir\(\)](#)
Return the current position in the directory stream: not implemented.

24.6.1 Detailed Description

The Reliance Edge file handle type for ARM mbed.

A [RedDirHandle](#) pointer is returned by a call to [RedFileSystem::opendir\(\)](#). [RedFileHandle](#) objects should not be otherwise instantiated directly.

In addition to the methods provided by [mbed::DirHandle](#) are implemented except for [seekdir\(\)](#) and [telldir\(\)](#).

This class is not implemented if Reliance Edge is configured to use its File System Essentials API or if directory operations are disabled ([REDCONF_API_POSIX_READDIR](#)).

Definition at line 76 of file RedDirHandle.h.

24.6.2 Member Function Documentation

24.6.2.1 [closedir\(\)](#)

```
int closedir ( ) [virtual]
```

Close the directory handle.

On success, the memory allocated to the directory handle will be freed.

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_closedir\(\)](#).

Definition at line 43 of file RedDirHandle.cpp.

24.6.2.2 readdir()

```
struct red_dirent * readdir ( ) [virtual]
```

Return the next directory entry, and move to the next entry in the directory stream.

Returns

On success, returns a pointer to a [red_dirent](#) struct, containing the name and stat information of the next file or subdirectory. On failure, returns NULL and sets red_errno to the appropriate value. If the end of the directory has been reached, returns NULL and leaves red_errno unmodified.

For a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_readdir\(\)](#).

Definition at line 57 of file RedDirHandle.cpp.

24.6.2.3 telldir()

```
virtual off_t telldir ( ) [inline], [virtual]
```

Return the current position in the directory stream: not implemented.

Returns

Sets red_errno to RED_ENOSYS and returns -1.

Definition at line 117 of file RedDirHandle.h.

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

- os/mbed/RedFileSystem/[RedDirHandle.h](#)
- os/mbed/RedFileSystem/[RedDirHandle.cpp](#)

24.7 RedFileHandle Class Reference

The Reliance Edge file handle type for ARM mbed.

Inherits FileHandle.

Public Member Functions

- virtual int [close \(\)](#)

Close the file handle.

- virtual ssize_t [write](#) (const void *buffer, size_t length)
Write to an open file.
- virtual ssize_t [read](#) (void *buffer, size_t length)
Read from an open file.
- virtual int [isatty](#) ()
Check whether this handle is associated with a terminal device.
- virtual off_t [lseek](#) (off_t position, int whence)
Move the read/write file offset.
- virtual int [fsync](#) ()
Synchronizes changes to a file.
- virtual off_t [flen](#) ()
Retrieve the length of the file.
- virtual int [ftruncate](#) (off_t size)
Truncate a file to a specified length (FileHandle extension).
- virtual int [fstat](#) (REDSTAT *stat)
Get file or directory information (FileHandle extension)

24.7.1 Detailed Description

The Reliance Edge file handle type for ARM mbed.

A [RedFileHandle](#) pointer is returned by a call to [RedFileSystem::open\(\)](#). [RedFileHandle](#) objects should not be otherwise instantiated directly.

In addition to the methods provided by [mbed::FileHandle](#), [RedFileHandle](#) implements [ftruncate\(\)](#) and [fstat\(\)](#).

This class is not implemented if Reliance Edge is configured to use its File System Essentials API.

Definition at line 56 of file RedFileHandle.h.

24.7.2 Member Function Documentation

24.7.2.1 [close\(\)](#)

```
int close ( ) [virtual]
```

Close the file handle.

On success, the memory allocated to the handle will be freed.

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_close\(\)](#).

Definition at line 43 of file RedFileHandle.cpp.

24.7.2.2 `flen()`

```
off_t flen ( ) [virtual]
```

Retrieve the length of the file.

Returns

On success, returns 0. On error, returns -1 and sets red_errno to one of the following values:

- [RED_EIO](#): A disk I/O error occurred.
- [RED_EUSERS](#): Cannot become a file system user: too many users.
- [RED_EFBIG](#): the file is too large to fit its size in the return value. Try using [fstat\(\)](#) to get the file size instead.

Definition at line 152 of file RedFileHandle.cpp.

24.7.2.3 `fstat()`

```
int fstat (
    REDSTAT * stat ) [virtual]
```

Get file or directory information (FileHandle extension)

Parameters

<code>stat</code>	Pointer to a REDSSTAT structure to populate.
-------------------	--

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_fstat\(\)](#).

Definition at line 184 of file RedFileHandle.cpp.

24.7.2.4 `fsync()`

```
int fsync ( ) [virtual]
```

Synchronizes changes to a file.

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_fsync\(\)](#).

Definition at line 142 of file RedFileHandle.cpp.

24.7.2.5 ftruncate()

```
int ftruncate (
    off_t size ) [virtual]
```

Truncate a file to a specified length (FileHandle extension).

Parameters

<i>size</i>	The new file size. If this is smaller than the current file size, then remaining data will be discarded. If this is larger than the current size, then the file will be padded with zeros.
-------------	--

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_ftruncate\(\)](#).

Definition at line 168 of file RedFileHandle.cpp.

24.7.2.6 isatty()

```
virtual int isatty ( ) [inline], [virtual]
```

Check whether this handle is associated with a terminal device.

Returns

Returns 0 to indicate that this is not a terminal device handle. Reliance Edge does not support terminal device files.

This function does not set red_errno.

Definition at line 116 of file RedFileHandle.h.

24.7.2.7 lseek()

```
off_t lseek (
    off_t position,
    int whence ) [virtual]
```

Move the read/write file offset.

Parameters

<i>position</i>	The new file offset, relative to whence.
<i>whence</i>	The base seek offset, one of SEEK_SET, SEEK_CUR, or SEEK_END.

Returns

On success, returns the new file offset in bytes. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_lseek\(\)](#).

In addition to the failure cases documented for red_lseek, this may fail with RED_EINVAL if the resulting file offset is greater than would fit in the return value.

Definition at line 92 of file RedFileHandle.cpp.

24.7.2.8 read()

```
ssize_t read (
    void * buffer,
    size_t length ) [virtual]
```

Read from an open file.

Parameters

<i>buffer</i>	The buffer in which to store data.
<i>length</i>	Number of bytes to attempt to read.

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_read\(\)](#).

Definition at line 75 of file RedFileHandle.cpp.

24.7.2.9 write()

```
ssize_t write (
    const void * buffer,
    size_t length ) [virtual]
```

Write to an open file.

Parameters

<i>buffer</i>	The buffer containing the data to be written.
<i>length</i>	Number of bytes to attempt to write.

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_write\(\)](#).

Definition at line 57 of file RedFileHandle.cpp.

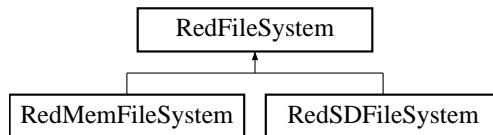
The documentation for this class was generated from the following files:

- os/mbed/RedFileSystem/[RedFileHandle.h](#)
- os/mbed/RedFileSystem/[RedFileHandle.cpp](#)

24.8 RedFileSystem Class Reference

The core implementation of the ARM mbed port of Tuxera's Reliance Edge failsafe filesystem for IoT.

Inheritance diagram for RedFileSystem:



Public Member Functions

- [RedFileSystem](#) (const char *n)
Construct a Reliance Edge filesystem object and initialize Reliance Edge if not already initited.
- virtual [RedFileHandle](#) * [open](#) (const char *name, int flags)
Open a file or directory.
- virtual int [remove](#) (const char *filename)
Remove a file or directory.
- virtual int [rename](#) (const char *oldname, const char *newname)
Rename or move a file or directory.
- virtual int [format](#) ()
Formats this volume (FileSystemLike extension)
- virtual [RedDirHandle](#) * [opendir](#) (const char *name)
Open a directory stream for reading.
- virtual int [mkdir](#) (const char *name, mode_t mode)
Create a new directory.
- virtual int [mount](#) ()
Mount this Reliance Edge volume (FileSystemLike extension)
- virtual int [umount](#) ()
Unmount this Reliance Edge volume (FileSystemLike extension)
- virtual int [transact](#) ()
Transact this Reliance Edge volume (FileSystemLike extension)
- virtual int [set_trans_mask](#) (uint32_t event_mask)

- virtual int `get_trans_mask (uint32_t *event_mask)`
Update the transaction mask (FileSystemLike extension)
- virtual int `statvfs (REDSTATFS *statvfs)`
Get the transaction mask (FileSystemLike extension)
- virtual int `link (const char *path, const char *hard_link)`
Queries file system status information (FileSystemLike extension)
- virtual int `link (const char *path, const char *hard_link)`
Make a hard link to a file (RedFileSystem extension)

Static Public Attributes

- static `RedFileSystem * _rfs [2U]`
RedFileSystem objects, as parallel to gaRedVolConf in redconf.c.

24.8.1 Detailed Description

The core implementation of the ARM mbed port of Tuxera's Reliance Edge failsafe filesystem for IoT.

To use Reliance Edge on ARM mbed, do not instantiate this virtual class directly. Instead, use one of the child classes that implement the block device interface calls, such as `RedSDFileSystem` or `RedMemFileSystem`.

When Reliance Edge is configured to use its POSIX-like API, this class plugs into mbed the same way as the FATFileSystem class does, inheriting from `mbed::FileSystemLike` so that `redfs` can be used with C standard library I/O (`fopen`, `fread`, etc). In addition to the methods provided by `mbed::FileSystemLike`, `RedFileSystem` implements `format()`, `mount()`, `unmount()`, `transact()`, `set_trans_mask()`, `get_trans_mask()`, `statvfs()`, and `link()`. Some API's may be disabled in the Reliance Edge configuration.

When Reliance Edge is configured to use its File System Essentials API (FSE), this class no longer inherits from `mbed::FileSystemLike` or maintains compatibility with the FATFileSystem class. With the FSE configuration, users should instantiate a `RedFileSystem` child class (such as `RedSDFileSystem`) to initialize the file system and provide block device I/O, but then the filesystem must be accessed through the RedFse APIs (declared in `redfse.h`). See the documentation comments in `RedSDFileSystem.h` for example code.

Unlike the mbed official FATFileSystem, Reliance Edge must be explicitly mounted after a `RedFileSystem` object is created before the filesystem is accessed.

Definition at line 86 of file `RedFileSystem.h`.

24.8.2 Constructor & Destructor Documentation

24.8.2.1 RedFileSystem()

```
RedFileSystem (
    const char * n )
```

Construct a Reliance Edge filesystem object and initialize Reliance Edge if not already initted.

If there is an error initializing or if `n` does not name a Reliance Edge volume, then the mbed `error()` function will be invoked.

Parameters

<code>n</code>	The volume path prefix; must match one of configured Reliance Edge volumes in <code>redconf.c</code> .
----------------	--

Definition at line 50 of file RedFileSystem.cpp.

24.8.3 Member Function Documentation

24.8.3.1 format()

```
int format ( ) [virtual]
```

Formats this volume (FileSystemLike extension)

If the volume is currently mounted, it will be unmounted before formatting and re-mounted afterward.

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_format\(\)](#). If [format\(\)](#) is called while the volume is still mounted, then any of the error conditions of [mount\(\)](#) and [unmount\(\)](#) may also be encountered.

Definition at line 199 of file RedFileSystem.cpp.

24.8.3.2 get_trans_mask()

```
int get_trans_mask (
    uint32_t * event_mask ) [virtual]
```

Get the transaction mask (FileSystemLike extension)

Parameters

<code>event_mask</code>	Populated with a bitwise-OR'd mask of RED_TRANSACT_ values which represent the current transaction mode.
-------------------------	--

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_gettransmask\(\)](#).

Definition at line 312 of file RedFileSystem.cpp.

24.8.3.3 link()

```
int link (
    const char * path,
    const char * hard_link ) [virtual]
```

Make a hard link to a file ([RedFileSystem](#) extension)

Creating a hard link to a directory is not supported.

Parameters

<i>path</i>	The path to the existing file to link to, not including the volume path prefix.
<i>hard_link</i>	The path at which to create the hard link, not including the volume path prefix.

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_link\(\)](#).

Definition at line 331 of file RedFileSystem.cpp.

24.8.3.4 mkdir()

```
int mkdir (
    const char * name,
    mode_t mode ) [virtual]
```

Create a new directory.

Parameters

<i>name</i>	The path for directory, not including the volume path prefix.
<i>mode</i>	Unused.

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_mkdir\(\)](#).

Definition at line 248 of file RedFileSystem.cpp.

24.8.3.5 mount()

```
int mount ( ) [virtual]
```

Mount this Reliance Edge volume (FileSystemLike extension)

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_mount\(\)](#).

Definition at line 270 of file RedFileSystem.cpp.

Referenced by [format\(\)](#), and [RedFileSystem\(\)](#).

24.8.3.6 open()

```
RedFileHandle * open (
    const char * name,
    int flags ) [virtual]
```

Open a file or directory.

Parameters

<i>name</i>	The path to the file or directory, not including the volume path prefix.
<i>flags</i>	The open flags: one of O_RDONLY, O_WRONLY, or O_RDWR, plus any of O_CREAT, O_APPEND, and/or O_TRUNC.

Returns

On success, returns a pointer to a newly allocated [RedFileHandle](#) object. On error, sets red_errno and returns NULL.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_open\(\)](#).

Definition at line 108 of file RedFileSystem.cpp.

24.8.3.7 opendir()

```
RedDirHandle * opendir (
    const char * name ) [virtual]
```

Open a directory stream for reading.

Parameters

<i>name</i>	The path to the directory, not including the volume path prefix.
-------------	--

Returns

On success, returns a pointer to a newly allocated [RedDirHandle](#) object. On error, sets red_errno and returns NULL.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_opendir\(\)](#).

Definition at line 226 of file RedFileSystem.cpp.

24.8.3.8 remove()

```
int remove (
    const char * filename ) [virtual]
```

Remove a file or directory.

Parameters

<i>filename</i>	The path to the file or directory, not including the volume path prefix.
-----------------	--

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_unlink\(\)](#).

Definition at line 153 of file RedFileSystem.cpp.

24.8.3.9 rename()

```
int rename (
    const char * oldname,
    const char * newname ) [virtual]
```

Rename or move a file or directory.

Parameters

<i>oldname</i>	The existing path to the file or directory, not including the volume path prefix.
<i>newname</i>	The new path, not including the volume path prefix.

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_rename\(\)](#).

Definition at line 174 of file RedFileSystem.cpp.

24.8.3.10 set_trans_mask()

```
int set_trans_mask (
    uint32_t event_mask ) [virtual]
```

Update the transaction mask (FileSystemLike extension)

Parameters

<code>event_mask</code>	A bitwise-OR'd mask of RED_TRANSACT_ values to be set as the current transaction mode.
-------------------------	--

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_settransmask\(\)](#).

Definition at line 302 of file RedFileSystem.cpp.

24.8.3.11 statvfs()

```
int statvfs (
    REDSTATFS * statvfs ) [virtual]
```

Queries file system status information (FileSystemLike extension)

Parameters

<code>statvfs</code>	Populated with file system information.
----------------------	---

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_statvfs\(\)](#).

Definition at line 321 of file RedFileSystem.cpp.

24.8.3.12 transact()

```
int transact ( ) [virtual]
```

Transact this Reliance Edge volume (FileSystemLike extension)

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_transact\(\)](#).

Definition at line 291 of file RedFileSystem.cpp.

24.8.3.13 unmount()

```
int unmount ( ) [virtual]
```

Unmount this Reliance Edge volume (FileSystemLike extension)

Returns

On success, returns 0. On error, sets red_errno and returns -1.

For further documentation, including a list of error conditions and red_errno values, see the Reliance Edge documentation for [red_unmount\(\)](#).

Definition at line 280 of file RedFileSystem.cpp.

Referenced by [format\(\)](#).

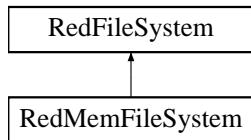
The documentation for this class was generated from the following files:

- os/mbed/RedFileSystem/[RedFileSystem.h](#)
- os/mbed/RedFileSystem/[RedFileSystem.cpp](#)

24.9 RedMemFileSystem Class Reference

A ramdisk implementation of the ARM mbed port of Tuxera's Reliance Edge failsafe filesystem for IoT.

Inheritance diagram for RedMemFileSystem:



Additional Inherited Members

24.9.1 Detailed Description

A ramdisk implementation of the ARM mbed port of Tuxera's Reliance Edge failsafe filesystem for IoT.

This class enables Reliance Edge, Tuxera's open-source power-safe filesystem, to be used on a virtual disk created in dynamically allocated memory.

Definition at line 32 of file RedMemFileSystem.h.

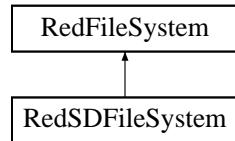
The documentation for this class was generated from the following file:

- os/mbed/RedFileSystem/RedMemFileSystem.h

24.10 RedSDFileSystem Class Reference

An SD/MMC implementation of the ARM mbed port of Tuxera's Reliance Edge failsafe filesystem for IoT.

Inheritance diagram for RedSDFileSystem:



Public Types

- enum `SwitchType` {
 `SWITCH_NONE`, `SWITCH_POS_NO`, `SWITCH_POS_NC`, `SWITCH_NEG_NO`,
`SWITCH_NEG_NC` }
- Represents the different card detect switch types.*
- enum `CardType` {
 `CARD_NONE`, `CARD_MMC`, `CARD_SD`, `CARD_SDHC`,
`CARD_UNKNOWN` }
- Represents the different SD/MMC card types.*

Public Member Functions

- `RedSDFileSystem` (PinName mosi, PinName miso, PinName sclk, PinName cs, const char *name, PinName cd=NC, `SwitchType` cdtype=`SWITCH_NONE`, int hz=1000000)
- Create a virtual file system for accessing SD/MMC cards via SPI.*
- `bool card_present ()`
- Determine whether or not a card is present.*
- `RedSDFileSystem::CardType card_type ()`
- Get the detected SD/MMC card type.*
- `bool crc ()`
- Get whether or not CRC is enabled for commands and data.*
- `void crc (bool enabled)`
- Set whether or not CRC is enabled for commands and data.*
- `bool large_frames ()`
- Get whether or not 16-bit frames are enabled for data read/write operations.*
- `void large_frames (bool enabled)`
- Set whether or not 16-bit frames are enabled for data read/write operations.*
- `bool write_validation ()`
- Get whether or not write validation is enabled for data write operations.*
- `void write_validation (bool enabled)`
- Set whether or not write validation is enabled for data write operations.*

Additional Inherited Members

24.10.1 Detailed Description

An SD/MMC implementation of the ARM mbed port of Tuxera's Reliance Edge failsafe filesystem for IoT.

This class enables Reliance Edge, Tuxera's open-source power-safe filesystem, to be used on SD/MMC cards via SPI.

If Reliance Edge is configured to use its POSIX-like API, then this class will integrate with the OS, allowing access to the volume through standard I/O, as demonstrated in the example below.

```
#include "mbed.h"
#include "RedSDFileSystem.h"

//Create a RedSDFileSystem object
RedSDFileSystem sd(p5, p6, p7, p20, "sd");

int main()
{
    //Mount the filesystem
    sd.mount();

    //Perform a write test
    printf("\nWriting to SD card...");
    FILE *fp = fopen("/sd/sdtest.txt", "w");
    if (fp != NULL) {
        fprintf(fp, "We're writing to an SD card!");
        fputc('W', fp);
        fclose(fp);
        printf("success!\n");
    } else {
        printf("failed!\n");
    }

    //Manually transact to ensure the information written has been
    //committed to the disk (specific to Reliance Edge)
    sd.transact();

    //Perform a read test
    printf("Reading from SD card...");
    fp = fopen("/sd/sdtest.txt", "r");
    if (fp != NULL) {
        char c = fgetc(fp);
        if (c == 'W')
            printf("success!\n");
        else
            printf("incorrect char (%c)!\n", c);
        fclose(fp);
    } else {
        printf("failed!\n");
    }

    //Unmount the filesystem
    sd.unmount();
}
```

If the Reliance Edge is configured to use its File System Essentials API (FSE), then the filesystem must be accessed through the RedFse APIs, as in this example:

```
#include "mbed.h"
#include "RedSDFileSystem.h"

//Create a RedSDFileSystem object for volume 0.
RedSDFileSystem sd(p5, p6, p7, p20, 0);

int main()
{
    //Mount the filesystem
    RedFseMount(0);

    //Perform a write test: write 1 byte ("W") to file 2 on
    //volume 0 at offset 0.
    printf("\nWriting to SD card...");
    if (RedFseWrite(0, 2, 0, 1, "W") == 1) {
```

```

        printf("success!\n");
    } else {
        printf("failed!\n");
    }

    //Manually transact to commit data
    RedFseTransact(0);

    //Perform a read test
    printf("Reading from SD card...");
    char c;
    if (RedFseRead(0, 2, 0, 1, &c) == 1) {
        if (c == 'W')
            printf("success!\n");
        else
            printf("incorrect char (%c)!\n", c);
    } else {
        printf("failed!\n");
    }

    //Unmount the filesystem
    RedFseUnmount(0);
}

```

Definition at line 126 of file RedSDFileSystem.h.

24.10.2 Member Enumeration Documentation

24.10.2.1 CardType

enum `CardType`

Represents the different SD/MMC card types.

Enumerator

<code>CARD_NONE</code>	No card is present.
<code>CARD_MMC</code>	MMC card.
<code>CARD_SD</code>	Standard capacity SD card.
<code>CARD_SDHC</code>	High capacity SD card.
<code>CARD_UNKNOWN</code>	Unknown or unsupported card.

Definition at line 141 of file RedSDFileSystem.h.

24.10.2.2 SwitchType

enum `SwitchType`

Represents the different card detect switch types.

Enumerator

<code>SWITCH_NONE</code>	No card detect switch (assumes socket is always occupied)
<code>SWITCH_POS_NO</code>	Switch shorts to VDD when the socket is occupied (positive logic, normally open)
<code>SWITCH_POS_NC</code>	Switch shorts to VDD when the socket is empty (positive logic, normally closed)

Enumerator

<code>SWITCH_NEG_NO</code>	Switch shorts to GND when the socket is occupied (negative logic, normally open)
<code>SWITCH_NEG_NC</code>	Switch shorts to GND when the socket is empty (negative logic, normally closed)

Definition at line 131 of file RedSDFileSystem.h.

24.10.3 Constructor & Destructor Documentation

24.10.3.1 RedSDFileSystem()

```
RedSDFileSystem (
    PinName mosi,
    PinName miso,
    PinName sclk,
    PinName cs,
    const char * name,
    PinName cd = NC,
    SwitchType cdtype = SWITCH_NONE,
    int hz = 1000000 )
```

Create a virtual file system for accessing SD/MMC cards via SPI.

Parameters

<code>mosi</code>	The SPI data out pin.
<code>miso</code>	The SPI data in pin.
<code>sclk</code>	The SPI clock pin.
<code>cs</code>	The SPI chip select pin.
<code>name</code>	The name used to access the virtual filesystem. Must match a volume path prefix in redconf.c. When Reliance Edge is configured to use FSE, this parameter is instead the uint8 volume number.
<code>cd</code>	The card detect pin.
<code>cdtype</code>	The type of card detect switch.
<code>hz</code>	The SPI bus frequency (defaults to 1MHz).

Definition at line 29 of file RedSDFileSystem.cpp.

24.10.4 Member Function Documentation

24.10.4.1 card_present()

```
bool card_present ( )
```

Determine whether or not a card is present.

Returns

'true' if a card is present, 'false' if no card is present.

Definition at line 86 of file RedSDFileSystem.cpp.

24.10.4.2 card_type()

```
RedSDFileSystem::CardType card_type ( )
```

Get the detected SD/MMC card type.

Returns

The detected card type as a CardType enum.

Note

Valid after the filesystem has been mounted.

Definition at line 95 of file RedSDFileSystem.cpp.

24.10.4.3 crc() [1/2]

```
bool crc ( )
```

Get whether or not CRC is enabled for commands and data.

Returns

'true' if CRC is enabled for commands and data, 'false' if CRC is disabled for commands and data.

Definition at line 104 of file RedSDFileSystem.cpp.

24.10.4.4 crc() [2/2]

```
void crc (
    bool enabled )
```

Set whether or not CRC is enabled for commands and data.

Parameters

<code>enabled</code>	Whether or not to enable CRC for commands and data.
----------------------	---

Definition at line 110 of file RedSDFileSystem.cpp.

24.10.4.5 large_frames() [1/2]

```
bool large_frames ( )
```

Get whether or not 16-bit frames are enabled for data read/write operations.

Returns

'true' if 16-bit frames will be used during data read/write operations, 'false' if 8-bit frames will be used during data read/write operations.

Definition at line 133 of file RedSDFileSystem.cpp.

24.10.4.6 large_frames() [2/2]

```
void large_frames (
    bool enabled )
```

Set whether or not 16-bit frames are enabled for data read/write operations.

Parameters

<code>enabled</code>	Whether or not 16-bit frames are enabled for data read/write operations.
----------------------	--

Definition at line 139 of file RedSDFileSystem.cpp.

24.10.4.7 write_validation() [1/2]

```
bool write_validation ( )
```

Get whether or not write validation is enabled for data write operations.

Returns

'true' if data writes will be verified using CMD13, 'false' if data writes will not be verified.

Definition at line 145 of file RedSDFileSystem.cpp.

24.10.4.8 write_validation() [2/2]

```
void write_validation (
    bool enabled )
```

Set whether or not write validation is enabled for data write operations.

Parameters

<code>enabled</code>	Whether or not write validation is enabled for data write operations.
----------------------	---

Definition at line 151 of file RedSDFFileSystem.cpp.

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

- os/mbed/RedSDFFileSystem/[RedSDFFileSystem.h](#)
- os/mbed/RedSDFFileSystem/[RedSDFFileSystem.cpp](#)

24.11 REDSTAT Struct Reference

Status information on an inode.

Data Fields

- `uint8_t st_dev`
Volume number of volume containing file.
- `uint32_t st_ino`
File serial number (inode number).
- `uint16_t st_mode`
Mode of file.
- `uint16_t st_nlink`
Number of hard links to the file.
- `uint64_t st_size`
File size in bytes.
- `uint32_t st_atime`
Time of last access (seconds since 01-01-1970).
- `uint32_t st_mtime`
Time of last data modification (seconds since 01-01-1970).
- `uint32_t st_ctime`
Time of last status change (seconds since 01-01-1970).
- `uint32_t st_blocks`
Number of blocks allocated for this object.

24.11.1 Detailed Description

Status information on an inode.

Definition at line 31 of file redstat.h.

24.11.2 Field Documentation

24.11.2.1 st_atime

`uint32_t st_atime`

Time of last access (seconds since 01-01-1970).

Definition at line 39 of file redstat.h.

24.11.2.2 st_blocks

`uint32_t st_blocks`

Number of blocks allocated for this object.

Definition at line 44 of file redstat.h.

24.11.2.3 st_ctime

`uint32_t st_ctime`

Time of last status change (seconds since 01-01-1970).

Definition at line 41 of file redstat.h.

24.11.2.4 st_dev

`uint8_t st_dev`

Volume number of volume containing file.

Definition at line 33 of file redstat.h.

24.11.2.5 st_ino

`uint32_t st_ino`

File serial number (inode number).

Definition at line 34 of file redstat.h.

24.11.2.6 st_mode

`uint16_t st_mode`

Mode of file.

Definition at line 35 of file redstat.h.

24.11.2.7 st_mtime

`uint32_t st_mtime`

Time of last data modification (seconds since 01-01-1970).

Definition at line 40 of file redstat.h.

24.11.2.8 st_nlink

`uint16_t st_nlink`

Number of hard links to the file.

Definition at line 36 of file redstat.h.

24.11.2.9 st_size

`uint64_t st_size`

File size in bytes.

Definition at line 37 of file redstat.h.

Referenced by RedFileHandle::flen(), and RedFileHandle::lseek().

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

- [include/redstat.h](#)

24.12 REDSTATFS Struct Reference

Status information on a file system volume.

Data Fields

- `uint32_t f_bsize`
File system block size.
- `uint32_t f_frsize`
Fundamental file system block size.
- `uint32_t f_blocks`
Total number of blocks on file system in units of f_frsize.
- `uint32_t f_bfree`
Total number of free blocks.
- `uint32_t f_bavail`
Number of free blocks available to non-privileged process.
- `uint32_t f_files`
Total number of file serial numbers.
- `uint32_t f_ffree`
Total number of free file serial numbers.
- `uint32_t f_favail`
Number of file serial numbers available to non-privileged process.
- `uint32_t f_fsid`
File system ID (useless, populated with zero).
- `uint32_t f_flag`
Bit mask of f_flag values.
- `uint32_t f_namemax`

Maximum filename length.

- `uint64_t f_maxsize`

Maximum file size (POSIX extension).

- `uint32_t f_dev`

Volume number (POSIX extension).

24.12.1 Detailed Description

Status information on a file system volume.

Definition at line 51 of file redstat.h.

24.12.2 Field Documentation

24.12.2.1 f_bavail

`uint32_t f_bavail`

Number of free blocks available to non-privileged process.

Definition at line 57 of file redstat.h.

24.12.2.2 f_bfree

`uint32_t f_bfree`

Total number of free blocks.

Definition at line 56 of file redstat.h.

24.12.2.3 f_blocks

`uint32_t f_blocks`

Total number of blocks on file system in units of f_frsize.

Definition at line 55 of file redstat.h.

24.12.2.4 f_bsize

`uint32_t f_bsize`

File system block size.

Definition at line 53 of file redstat.h.

24.12.2.5 f_dev

`uint32_t f_dev`

Volume number (POSIX extension).

Definition at line 65 of file redstat.h.

24.12.2.6 f_favail

`uint32_t f_favail`

Number of file serial numbers available to non-privileged process.

Definition at line 60 of file redstat.h.

24.12.2.7 f_ffree

`uint32_t f_ffree`

Total number of free file serial numbers.

Definition at line 59 of file redstat.h.

24.12.2.8 f_files

`uint32_t f_files`

Total number of file serial numbers.

Definition at line 58 of file redstat.h.

24.12.2.9 f_flag

`uint32_t f_flag`

Bit mask of f_flag values.

Includes read-only file system flag.

Definition at line 62 of file redstat.h.

24.12.2.10 f_frsiz

`uint32_t f_frsiz`

Fundamental file system block size.

Definition at line 54 of file redstat.h.

24.12.2.11 f_fsid

`uint32_t f_fsid`

File system ID (useless, populated with zero).

Definition at line 61 of file redstat.h.

24.12.2.12 f_maxfsize

`uint64_t f_maxfsize`

Maximum file size (POSIX extension).

Definition at line 64 of file redstat.h.

24.12.2.13 f_namemax

`uint32_t f_namemax`

Maximum filename length.

Definition at line 63 of file redstat.h.

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

- [include/redstat.h](#)

Chapter 25

File Documentation

25.1 fse/fse.c File Reference

Implementation of the Reliance Edge FSE API.

Functions

- **REDSSTATUS RedFseInit (void)**
Initialize the Reliance Edge file system driver.
- **REDSSTATUS RedFseUninit (void)**
Uninitialize the Reliance Edge file system driver.
- **REDSSTATUS RedFseMount (uint8_t bVolNum)**
Mount a file system volume.
- **REDSSTATUS RedFseUnmount (uint8_t bVolNum)**
Unmount a file system volume.
- **REDSSTATUS RedFseFormat (uint8_t bVolNum)**
Format a file system volume.
- **int32_t RedFseRead (uint8_t bVolNum, uint32_t ulFileNum, uint64_t ullFileOffset, uint32_t ulLength, void *pBuffer)**
Read from a file.
- **int32_t RedFseWrite (uint8_t bVolNum, uint32_t ulFileNum, uint64_t ullFileOffset, uint32_t ulLength, const void *pBuffer)**
Write to a file.
- **REDSSTATUS RedFseTruncate (uint8_t bVolNum, uint32_t ulFileNum, uint64_t ullNewFileSize)**
Truncate a file (set the file size).
- **int64_t RedFseSizeGet (uint8_t bVolNum, uint32_t ulFileNum)**
Retrieve the size of a file.
- **REDSSTATUS RedFseTransMaskSet (uint8_t bVolNum, uint32_t ulEventMask)**
Update the transaction mask.
- **REDSSTATUS RedFseTransMaskGet (uint8_t bVolNum, uint32_t *pulEventMask)**
Read the transaction mask.
- **REDSSTATUS RedFseTransact (uint8_t bVolNum)**
Commit a transaction point.

25.1.1 Detailed Description

Implementation of the Reliance Edge FSE API.

25.1.2 Function Documentation

25.1.2.1 RedFseFormat()

```
REDSTATUS RedFseFormat (
    uint8_t bVolNum )
```

Format a file system volume.

Uses the statically defined volume configuration. After calling this function, the volume needs to be mounted – see [RedFseMount\(\)](#).

An error is returned if the volume is mounted.

Parameters

<i>bVolNum</i>	The volume number of the volume to be formatted.
----------------	--

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EBUSY</i>	The volume is mounted.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number; or the driver is uninitialized.
<i>-RED_EIO</i>	I/O error formatting the volume.

Definition at line 230 of file fse.c.

25.1.2.2 RedFseInit()

```
REDSTATUS RedFseInit (
    void )
```

Initialize the Reliance Edge file system driver.

Prepares the Reliance Edge file system driver to be used. Must be the first Reliance Edge function to be invoked: no volumes can be mounted until the driver has been initialized.

If this function is called when the Reliance Edge driver is already initialized, it does nothing and returns success.

This function is not thread safe: attempting to initialize from multiple threads could leave things in a bad state.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
----------	---------------------------

Definition at line 40 of file fse.c.

25.1.2.3 RedFseMount()

```
REDSTATUS RedFseMount (
    uint8_t bVolNum )
```

Mount a file system volume.

Prepares the file system volume to be accessed. Mount will fail if the volume has never been formatted, or if the on-disk format is inconsistent with the compile-time configuration.

If the volume is already mounted, this function does nothing and returns success.

Parameters

<i>bVolNum</i>	The volume number of the volume to be mounted.
----------------	--

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number; or the driver is uninitialized.
<i>-RED_EIO</i>	Volume not formatted, improperly formatted, or corrupt.

Definition at line 141 of file fse.c.

25.1.2.4 RedFseRead()

```
int32_t RedFseRead (
    uint8_t bVolNum,
    uint32_t ulFileNum,
    uint64_t ullFileOffset,
    uint32_t ulLength,
    void * pBuffer )
```

Read from a file.

Data which has not yet been written, but which is before the end-of-file (sparse data), shall read as zeroes. A short read – where the number of bytes read is less than requested – indicates that the requested read was partially or, if zero bytes were read, entirely beyond the end-of-file.

If *ullFileOffset* is at or beyond the maximum file size, it is treated like any other read entirely beyond the end-of-file: no data is read and zero is returned.

Parameters

<i>bVolNum</i>	The volume number of the file to read.
<i>ulFileNum</i>	The file number of the file to read.
<i>ullFileOffset</i>	The file offset to read from.
<i>ulLength</i>	The number of bytes to read.
<i>pBuffer</i>	The buffer to populate with the data read. Must be at least <i>ulLength</i> bytes in size.

Returns

The number of bytes read (nonnegative) or a negated [REDSTATUS](#) code indicating the operation result (negative).

Return values

<i>>=0</i>	The number of bytes read from the file.
<i>-RED_EBADF</i>	<i>ulFileNum</i> is not a valid file number.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number or not mounted; or <i>pBuffer</i> is NULL; or <i>ulLength</i> exceeds INT32_MAX and cannot be returned properly.
<i>-RED_EIO</i>	A disk I/O error occurred.

Definition at line 277 of file fse.c.

25.1.2.5 RedFseSizeGet()

```
int64_t RedFseSizeGet (
    uint8_t bVolNum,
    uint32_t ulFileNum )
```

Retrieve the size of a file.

Parameters

<i>bVolNum</i>	The volume number of the file whose size is being read.
<i>ulFileNum</i>	The file number of the file whose size is being read.

Returns

The size of the file (nonnegative) or a negated [REDSTATUS](#) code indicating the operation result (negative).

Return values

<i>>=0</i>	The size of the file.
---------------	-----------------------

Return values

<i>-RED_EBADF</i>	ulFileNum is not a valid file number.
<i>-RED_EINVAL</i>	bVolNum is an invalid volume number or not mounted.
<i>-RED_EIO</i>	A disk I/O error occurred.

Definition at line 453 of file fse.c.

25.1.2.6 RedFseTransact()

```
REDSHARE RedFseTransact (
    uint8_t bVolNum )
```

Commit a transaction point.

Reliance Edge is a transactional file system. All modifications, of both metadata and filedata, are initially working state. A transaction point is a process whereby the working state atomically becomes the committed state, replacing the previous committed state. Whenever Reliance Edge is mounted, including after power loss, the state of the file system after mount is the most recent committed state. Nothing from the committed state is ever missing, and nothing from the working state is ever included.

Parameters

<i>bVolNum</i>	The volume number of the volume to transact.
----------------	--

Returns

A negated **REDSHARE** code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	bVolNum is an invalid volume number or not mounted.
<i>-RED_EIO</i>	A disk I/O error occurred.
<i>-REDEROFS</i>	The file system volume is read-only.

Definition at line 591 of file fse.c.

25.1.2.7 RedFseTransMaskGet()

```
REDSHARE RedFseTransMaskGet (
    uint8_t bVolNum,
    uint32_t * pulEventMask )
```

Read the transaction mask.

If the volume is read-only, the returned event mask is always zero.

Parameters

<i>bVolNum</i>	The volume number of the volume whose transaction mask is being retrieved.
<i>pulEventMask</i>	Populated with a bitwise-OR'd mask of automatic transaction events which represent the current transaction mode for the volume.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number or not mounted; or <i>pulEventMask</i> is NULL.

Definition at line 551 of file fse.c.

25.1.2.8 RedFseTransMaskSet()

```
REDSTATUS RedFseTransMaskSet (
    uint8_t bVolNum,
    uint32_t ulEventMask )
```

Update the transaction mask.

The following events are available:

- [RED_TRANSACT_UMOUNT](#)
- [RED_TRANSACT_WRITE](#)
- [RED_TRANSACT_TRUNCATE](#)
- [RED_TRANSACT_VOLFULL](#)

The [RED_TRANSACT_MANUAL](#) macro (by itself) may be used to disable all automatic transaction events. The [RED_TRANSACT_MASK](#) macro is a bitmask of all transaction flags, excluding those representing excluded functionality.

Attempting to enable events for excluded functionality will result in an error.

Parameters

<i>bVolNum</i>	The volume number of the volume whose transaction mask is being changed.
<i>ulEventMask</i>	A bitwise-OR'd mask of automatic transaction events to be set as the current transaction mode.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	bVolNum is an invalid volume number or not mounted; or ulEventMask contains invalid bits.
<i>-RED_EROFS</i>	The file system volume is read-only.

Definition at line 514 of file fse.c.

25.1.2.9 RedFseTruncate()

```
REDSTATUS RedFseTruncate (
    uint8_t bVolNum,
    uint32_t ulFileNum,
    uint64_t ullNewFileSize )
```

Truncate a file (set the file size).

Allows the file size to be increased, decreased, or to remain the same. If the file size is increased, the new area is sparse (will read as zeroes). If the file size is decreased, the data beyond the new end-of-file will return to free space once it is no longer part of the committed state (either immediately or after the next transaction point).

This function can fail when the disk is full if `ullNewFileSize` is non-zero. If decreasing the file size, this can be fixed by transacting and trying again: Reliance Edge guarantees that it is possible to perform a truncate of at least one file that decreases the file size after a transaction point. If disk full transactions are enabled, this will happen automatically.

Parameters

<i>bVolNum</i>	The volume number of the file to truncate.
<i>ulFileNum</i>	The file number of the file to truncate.
<i>ullNewFileSize</i>	The new file size, in bytes.

Returns

A negated `REDSTATUS` code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EBADF</i>	ulFileNum is not a valid file number.
<i>-RED_EFBIG</i>	ullNewFileSize exceeds the maximum file size.
<i>-RED_EINVAL</i>	bVolNum is an invalid volume number or not mounted.
<i>-RED_EIO</i>	A disk I/O error occurred.
<i>-RED_ENOSPC</i>	Insufficient free space to perform the truncate.
<i>-RED_EROFS</i>	The file system volume is read-only.

Definition at line 419 of file fse.c.

25.1.2.10 RedFseUninit()

```
REDSTATUS RedFseUninit (
    void )
```

Uninitialize the Reliance Edge file system driver.

Tears down the Reliance Edge file system driver. Cannot be used until all Reliance Edge volumes are unmounted. A subsequent call to [RedFseInit\(\)](#) will initialize the driver again.

If this function is called when the Reliance Edge driver is already uninitialized, it does nothing and returns success.

This function is not thread safe: attempting to uninitialized from multiple threads could leave things in a bad state.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EBUSY</i>	At least one volume is still mounted.

Definition at line 79 of file fse.c.

25.1.2.11 RedFseUnmount()

```
REDSTATUS RedFseUnmount (
    uint8_t bVolNum )
```

Unmount a file system volume.

This function discards the in-memory state for the file system and marks it as unmounted. Subsequent attempts to access the volume will fail until the volume is mounted again.

If unmount automatic transaction points are enabled, this function will commit a transaction point prior to unmounting. If unmount automatic transaction points are disabled, this function will unmount without transacting, effectively discarding the working state.

Before unmounting, this function will wait for any active file system thread to complete by acquiring the FS mutex. The volume will be marked as unmounted before the FS mutex is released, so subsequent FS threads will possibly block and then see an error when attempting to access a volume which is unmounting or unmounted.

If the volume is already unmounted, this function does nothing and returns success.

Parameters

<i>bVolNum</i>	The volume number of the volume to be unmounted.
----------------	--

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	bVolNum is an invalid volume number; or the driver is uninitialized.
<i>-RED_EIO</i>	I/O error during unmount automatic transaction point.

Definition at line 191 of file fse.c.

25.1.2.12 RedFseWrite()

```
int32_t RedFseWrite (
    uint8_t bVolNum,
    uint32_t ulFileNum,
    uint64_t ullFileOffset,
    uint32_t ulLength,
    const void * pBuffer )
```

Write to a file.

If the write extends beyond the end-of-file, the file size will be increased.

A short write – where the number of bytes written is less than requested – indicates either that the file system ran out of space but was still able to write some of the request; or that the request would have caused the file to exceed the maximum file size, but some of the data could be written prior to the file size limit.

If an error is returned (negative return), either none of the data was written or a critical error occurred (like an I/O error) and the file system volume will be read-only.

Parameters

<i>bVolNum</i>	The volume number of the file to write.
<i>ulFileNum</i>	The file number of the file to write.
<i>ullFileOffset</i>	The file offset to write at.
<i>ulLength</i>	The number of bytes to write.
<i>pBuffer</i>	The buffer containing the data to be written. Must be at least <i>ulLength</i> bytes in size.

Returns

The number of bytes written (nonnegative) or a negated [REDSTATUS](#) code indicating the operation result (negative).

Return values

<i>>0</i>	The number of bytes written to the file.
<i>-RED_EBADF</i>	<i>ulFileNum</i> is not a valid file number.
<i>-RED_EFBIG</i>	No data can be written to the given file offset since the resulting file size would exceed the maximum file size.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number or not mounted; or <i>pBuffer</i> is NULL; or <i>ulLength</i> exceeds INT32_MAX and cannot be returned properly.
<i>-RED_EIO</i>	A disk I/O error occurred.

Return values

<code>-RED_ENOSPC</code>	No data can be written because there is insufficient free space.
<code>-REDEROFS</code>	The file system volume is read-only.

Definition at line 352 of file fse.c.

25.2 include/redapimacs.h File Reference

Defines macros used to interact with the Reliance Edge API.

Macros

- `#define RED_MOUNT_READONLY`
Mount the volume as read-only.
- `#define RED_MOUNT_DISCARD`
Mount the volume with automatic discards enabled.
- `#define RED_MOUNT_MASK`
Mask of all supported mount flags.
- `#define RED_MOUNT_DEFAULT`
Default mount flags.
- `#define RED_UNMOUNT_FORCE`
Force unmount, closing all open handles.
- `#define RED_UNMOUNT_MASK`
Mask of all supported unmount flags.
- `#define RED_UNMOUNT_DEFAULT`
Default unmount flags.
- `#define RED_TRANSACT_MANUAL`
Clear all events: manual transactions only.
- `#define RED_TRANSACT_UNMOUNT`
Transact prior to unmounting in `red_umount()`, `red_umount2()`, or `RedFseUnmount()`.
- `#define RED_TRANSACT_CREAT`
Transact after a successful `red_open()` which created a file.
- `#define RED_TRANSACT_UNLINK`
Transact after a successful `red_unlink()` or `red_rmdir()`.
- `#define RED_TRANSACT_MKDIR`
Transact after a successful `red_mkdir()`.
- `#define RED_TRANSACT_RENAME`
Transact after a successful `red_rename()`.
- `#define RED_TRANSACT_LINK`
Transact after a successful `red_link()`.
- `#define RED_TRANSACT_CLOSE`
Transact after a successful `red_close()`.
- `#define RED_TRANSACT_WRITE`
Transact after a successful `red_write()` or `RedFseWrite()`.
- `#define RED_TRANSACT_FSYNC`

Transact after a successful [red_fsync\(\)](#).

- `#define RED_TRANSACT_TRUNCATE`

Transact after a successful [red_ftruncate\(\)](#), [RedFseTruncate\(\)](#), or [red_open\(\)](#) with `RED_O_TRUNC` that actually truncates.

- `#define RED_TRANSACT_VOLFULL`

Transact to free space in disk full situations.

- `#define RED_TRANSACT_SYNC`

Transact after a successful os sync.

- `#define RED_TRANSACT_MASK`

Mask of all supported automatic transaction events.

25.2.1 Detailed Description

Defines macros used to interact with the Reliance Edge API.

25.2.2 Macro Definition Documentation

25.2.2.1 RED_MOUNT_DEFAULT

```
#define RED_MOUNT_DEFAULT
```

Default mount flags.

These are the mount flags that are used when Reliance Edge is mounted via an API which does not allow mount flags to be specified: viz., [red_mount\(\)](#) or [RedFseMount\(\)](#). If [red_mount2\(\)](#) is used, the flags provided to it supersede these flags.

Definition at line 29 of file redapimacs.h.

Referenced by [red_mount\(\)](#).

25.2.2.2 RED_MOUNT_DISCARD

```
#define RED_MOUNT_DISCARD
```

Mount the volume with automatic discards enabled.

Definition at line 12 of file redapimacs.h.

25.2.2.3 RED_MOUNT_MASK

```
#define RED_MOUNT_MASK
```

Mask of all supported mount flags.

Definition at line 19 of file redapimacs.h.

25.2.2.4 RED_MOUNT_READONLY

```
#define RED_MOUNT_READONLY
```

Mount the volume as read-only.

Definition at line 9 of file redapimacs.h.

25.2.2.5 RED_TRANSACT_CLOSE

```
#define RED_TRANSACT_CLOSE
```

Transact after a successful [red_close\(\)](#).

Definition at line 64 of file redapimacs.h.

25.2.2.6 RED_TRANSACT_CREAT

```
#define RED_TRANSACT_CREAT
```

Transact after a successful [red_open\(\)](#) which created a file.

Definition at line 49 of file redapimacs.h.

25.2.2.7 RED_TRANSACT_FSYNC

```
#define RED_TRANSACT_FSYNC
```

Transact after a successful [red_fsync\(\)](#).

Definition at line 70 of file redapimacs.h.

25.2.2.8 RED_TRANSACT_LINK

```
#define RED_TRANSACT_LINK
```

Transact after a successful [red_link\(\)](#).

Definition at line 61 of file redapimacs.h.

25.2.2.9 RED_TRANSACT_MANUAL

```
#define RED_TRANSACT_MANUAL
```

Clear all events: manual transactions only.

Definition at line 43 of file redapimacs.h.

25.2.2.10 RED_TRANSACT_MKDIR

```
#define RED_TRANSACT_MKDIR
```

Transact after a successful [red_mkdir\(\)](#).

Definition at line 55 of file redapimacs.h.

25.2.2.11 RED_TRANSACT_RENAME

```
#define RED_TRANSACT_RENAME
```

Transact after a successful [red_rename\(\)](#).

Definition at line 58 of file redapimacs.h.

25.2.2.12 RED_TRANSACT_SYNC

```
#define RED_TRANSACT_SYNC
```

Transact after a successful os sync.

Definition at line 79 of file redapimacs.h.

25.2.2.13 RED_TRANSACT_TRUNCATE

```
#define RED_TRANSACT_TRUNCATE
```

Transact after a successful [red_ftruncate\(\)](#), [RedFseTruncate\(\)](#), or [red_open\(\)](#) with [RED_O_TRUNC](#) that actually truncates.

Definition at line 73 of file redapimacs.h.

25.2.2.14 RED_TRANSACT_UNMOUNT

```
#define RED_TRANSACT_UNMOUNT
```

Transact prior to unmounting in [red_umount\(\)](#), [red_umount2\(\)](#), or [RedFseUnmount\(\)](#).

Definition at line 46 of file redapimacs.h.

25.2.2.15 RED_TRANSACT_UNLINK

```
#define RED_TRANSACT_UNLINK
```

Transact after a successful [red_unlink\(\)](#) or [red_rmdir\(\)](#).

Definition at line 52 of file redapimacs.h.

25.2.2.16 RED_TRANSACT_VOLFULL

```
#define RED_TRANSACT_VOLFULL
```

Transact to free space in disk full situations.

Definition at line 76 of file redapimacs.h.

25.2.2.17 RED_TRANSACT_WRITE

```
#define RED_TRANSACT_WRITE
```

Transact after a successful [red_write\(\)](#) or [RedFseWrite\(\)](#).

Definition at line 67 of file redapimacs.h.

25.2.2.18 RED_UNMOUNT_DEFAULT

```
#define RED_UNMOUNT_DEFAULT
```

Default umount flags.

Definition at line 39 of file redapimacs.h.

Referenced by [red_umount\(\)](#).

25.2.2.19 RED_UNMOUNT_FORCE

```
#define RED_UNMOUNT_FORCE
```

Force umount, closing all open handles.

Definition at line 33 of file redapimacs.h.

25.2.2.20 RED_UNMOUNT_MASK

```
#define RED_UNMOUNT_MASK
```

Mask of all supported umount flags.

Definition at line 36 of file redapimacs.h.

25.3 include/rederrno.h File Reference

Error values for Reliance Edge APIs.

Macros

- [#define RED_EPERM](#)

Operation not permitted.

- #define RED_ENOENT

No such file or directory.

- #define RED_EIO

I/O error.

- #define RED_EBADF

Bad file number.

- #define RED_ENOMEM

Out of memory.

- #define RED_EBUSY

Device or resource busy.

- #define RED_EEXIST

File exists.

- #define RED_EXDEV

Cross-device link.

- #define RED_ENOTDIR

Not a directory.

- #define RED_EISDIR

Is a directory.

- #define RED_EINVAL

Invalid argument.

- #define RED_ENFILE

File table overflow.

- #define RED_EMFILE

Too many open files.

- #define RED_EFBIG

File too large.

- #define RED_ENOSPC

No space left on device.

- #define RED_EROFS

Read-only file system.

- #define RED_EMLINK

Too many links.

- #define RED_ERANGE

Math result not representable.

- #define RED_ENAMETOOLONG

File name too long.

- #define RED_ENOSYS

Function not implemented.

- #define RED_ENOTEMPTY

Directory not empty.

- #define RED_ENODATA

No data available.

- #define RED_EUSERS

Too many users.

- #define RED_ENOTSUPP

Operation is not supported.

- #define RED_EFUBAR

Nothing will be okay ever again.

Typedefs

- `typedef int32_t REDSTATUS`
Return type for Reliance Edge error values.

25.3.1 Detailed Description

Error values for Reliance Edge APIs.

25.3.2 Macro Definition Documentation

25.3.2.1 RED_EBADF

```
#define RED_EBADF
```

Bad file number.

Definition at line 26 of file rederrno.h.

25.3.2.2 RED_EBUSY

```
#define RED_EBUSY
```

Device or resource busy.

Definition at line 32 of file rederrno.h.

Referenced by `redfs_format()`.

25.3.2.3 RED_EEXIST

```
#define RED_EEXIST
```

File exists.

Definition at line 35 of file rederrno.h.

25.3.2.4 RED_EFBIG

```
#define RED_EFBIG
```

File too large.

Definition at line 56 of file rederrno.h.

Referenced by `RedFileHandle::flen()`, and `RedFileHandle::ftruncate()`.

25.3.2.5 RED_EFUBAR

```
#define RED_EFUBAR
```

Nothing will be okay ever again.

Definition at line 89 of file rederrno.h.

25.3.2.6 RED_EINVAL

```
#define RED_EINVAL
```

Invalid argument.

Definition at line 47 of file rederrno.h.

Referenced by RedFileSystem::link(), RedFileHandle::lseek(), RedFileSystem::mkdir(), RedFileSystem::open(), RedFileSystem::opendir(), RedFileHandle::read(), red_getcwd(), red_read(), red_write(), RedFseRead(), RedFseWrite(), RedOsBDevClose(), RedOsBDevDiscard(), RedOsBDevFlush(), RedOsBDevGetGeometry(), RedOsBDevOpen(), RedOsBDevRead(), RedOsBDevWrite(), RedFileSystem::remove(), RedFileSystem::rename(), and RedFileHandle::write().

25.3.2.7 RED_EIO

```
#define RED_EIO
```

I/O error.

Definition at line 23 of file rederrno.h.

25.3.2.8 RED_EISDIR

```
#define RED_EISDIR
```

Is a directory.

Definition at line 44 of file rederrno.h.

25.3.2.9 RED_EMFILE

```
#define RED_EMFILE
```

Too many open files.

Definition at line 53 of file rederrno.h.

25.3.2.10 RED_EMLINK

```
#define RED_EMLINK
```

Too many links.

Definition at line 65 of file rederrno.h.

25.3.2.11 RED_ENAMETOOLONG

```
#define RED_ENAMETOOLONG
```

File name too long.

Definition at line 71 of file rederrno.h.

Referenced by RedDirHandle::readdir().

25.3.2.12 RED_ENFILE

```
#define RED_ENFILE
```

File table overflow.

Definition at line 50 of file rederrno.h.

25.3.2.13 RED_ENODATA

```
#define RED_ENODATA
```

No data available.

Definition at line 80 of file rederrno.h.

25.3.2.14 RED_ENOENT

```
#define RED_ENOENT
```

No such file or directory.

Definition at line 20 of file rederrno.h.

25.3.2.15 RED_ENOSPC

```
#define RED_ENOSPC
```

No space left on device.

Definition at line 59 of file rederrno.h.

25.3.2.16 RED_ENOSYS

```
#define RED_ENOSYS
```

Function not implemented.

Definition at line 74 of file rederrno.h.

Referenced by RedOsBDevClose(), RedOsBDevDiscard(), RedOsBDevFlush(), RedOsBDevGetGeometry(), RedOsBDevOpen(), RedOsBDevRead(), RedOsBDevWrite(), RedOsMutexInit(), RedOsMutexUninit(), RedOsTimestampInit(), RedOsTimestampUninit(), and RedDirHandle::telldir().

25.3.2.17 RED_ENOTDIR

```
#define RED_ENOTDIR
```

Not a directory.

Definition at line 41 of file rederrno.h.

25.3.2.18 RED_ENOTEMPTY

```
#define RED_ENOTEMPTY
```

Directory not empty.

Definition at line 77 of file rederrno.h.

25.3.2.19 RED_ENOTSUPP

```
#define RED_ENOTSUPP
```

Operation is not supported.

Definition at line 86 of file rederrno.h.

25.3.2.20 RED_EPERM

```
#define RED_EPERM
```

Operation not permitted.

Definition at line 17 of file rederrno.h.

25.3.2.21 RED_ERANGE

```
#define RED_ERANGE
```

Math result not representable.

Definition at line 68 of file rederrno.h.

25.3.2.22 RED_EROFS

```
#define RED_EROFS
```

Read-only file system.

Definition at line 62 of file rederrno.h.

25.3.2.23 RED_EUSERS

```
#define RED_EUSERS
```

Too many users.

Definition at line 83 of file rederrno.h.

25.3.2.24 RED_EXDEV

```
#define RED_EXDEV
```

Cross-device link.

Definition at line 38 of file rederrno.h.

25.4 include/redfse.h File Reference

Interface for the Reliance Edge FSE API.

Macros

- [#define RED_FILENO_FIRST_VALID](#)

First valid file number.

Functions

- [REDSTATUS RedFseInit \(void\)](#)
Initialize the Reliance Edge file system driver.
- [REDSTATUS RedFseUninit \(void\)](#)
Uninitialize the Reliance Edge file system driver.
- [REDSTATUS RedFseMount \(uint8_t bVolNum\)](#)
Mount a file system volume.
- [REDSTATUS RedFseUnmount \(uint8_t bVolNum\)](#)
Unmount a file system volume.
- [REDSTATUS RedFseFormat \(uint8_t bVolNum\)](#)

- Format a file system volume.
- `int32_t RedFseRead (uint8_t bVolNum, uint32_t ulFileNum, uint64_t ullFileOffset, uint32_t ulLength, void *pBuffer)`
 - Read from a file.*
- `int32_t RedFseWrite (uint8_t bVolNum, uint32_t ulFileNum, uint64_t ullFileOffset, uint32_t ulLength, const void *pBuffer)`
 - Write to a file.*
- `REDSSTATUS RedFseTruncate (uint8_t bVolNum, uint32_t ulFileNum, uint64_t ullNewFileSize)`
 - Truncate a file (set the file size).*
- `int64_t RedFseSizeGet (uint8_t bVolNum, uint32_t ulFileNum)`
 - Retrieve the size of a file.*
- `REDSSTATUS RedFseTransMaskSet (uint8_t bVolNum, uint32_t ulEventMask)`
 - Update the transaction mask.*
- `REDSSTATUS RedFseTransMaskGet (uint8_t bVolNum, uint32_t *pulEventMask)`
 - Read the transaction mask.*
- `REDSSTATUS RedFseTransact (uint8_t bVolNum)`
 - Commit a transaction point.*

25.4.1 Detailed Description

Interface for the Reliance Edge FSE API.

The FSE (File Systems Essentials) API is a minimalist file system API intended for simple use cases with a fixed number of statically-defined files. It does not support creating or deleting files dynamically. Files are referenced by a fixed file number, rather than by name; there are no file names and no directories. There are also no file handles: files are not opened or closed, and file offsets are given explicitly.

If the FSE API is too limited to meet the needs of your application, consider using the more feature-rich POSIX-like file system API instead.

25.4.2 Macro Definition Documentation

25.4.2.1 RED_FILENO_FIRST_VALID

```
#define RED_FILENO_FIRST_VALID
```

First valid file number.

This macro can be used to statically define file numbers for given files, as in the below example:

```
#define LOG_FILE      (RED_FILENO_FIRST_VALID)
#define DATABASE_FILE (RED_FILENO_FIRST_VALID + 1U)
#define ICON1_FILE    (RED_FILENO_FIRST_VALID + 2U)
#define ICON2_FILE    (RED_FILENO_FIRST_VALID + 3U)
```

Definition at line 45 of file redfse.h.

25.4.3 Function Documentation

25.4.3.1 RedFseFormat()

```
REDSTATUS RedFseFormat (
    uint8_t bVolNum )
```

Format a file system volume.

Uses the statically defined volume configuration. After calling this function, the volume needs to be mounted – see [RedFseMount\(\)](#).

An error is returned if the volume is mounted.

Parameters

<i>bVolNum</i>	The volume number of the volume to be formatted.
----------------	--

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EBUSY</i>	The volume is mounted.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number; or the driver is uninitialized.
<i>-RED_EIO</i>	I/O error formatting the volume.

Definition at line 230 of file fse.c.

25.4.3.2 RedFseInit()

```
REDSTATUS RedFseInit (
    void )
```

Initialize the Reliance Edge file system driver.

Prepares the Reliance Edge file system driver to be used. Must be the first Reliance Edge function to be invoked: no volumes can be mounted until the driver has been initialized.

If this function is called when the Reliance Edge driver is already initialized, it does nothing and returns success.

This function is not thread safe: attempting to initialize from multiple threads could leave things in a bad state.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
----------	---------------------------

Definition at line 40 of file fse.c.

25.4.3.3 RedFseMount()

```
REDSTATUS RedFseMount (
    uint8_t bVolNum )
```

Mount a file system volume.

Prepares the file system volume to be accessed. Mount will fail if the volume has never been formatted, or if the on-disk format is inconsistent with the compile-time configuration.

If the volume is already mounted, this function does nothing and returns success.

Parameters

<i>bVolNum</i>	The volume number of the volume to be mounted.
----------------	--

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number; or the driver is uninitialized.
<i>-RED_EIO</i>	Volume not formatted, improperly formatted, or corrupt.

Definition at line 141 of file fse.c.

25.4.3.4 RedFseRead()

```
int32_t RedFseRead (
    uint8_t bVolNum,
    uint32_t ulFileNum,
    uint64_t ullFileOffset,
    uint32_t ulLength,
    void * pBuffer )
```

Read from a file.

Data which has not yet been written, but which is before the end-of-file (sparse data), shall read as zeroes. A short read – where the number of bytes read is less than requested – indicates that the requested read was partially or, if zero bytes were read, entirely beyond the end-of-file.

If *ullFileOffset* is at or beyond the maximum file size, it is treated like any other read entirely beyond the end-of-file: no data is read and zero is returned.

Parameters

<i>bVolNum</i>	The volume number of the file to read.
<i>ulFileNum</i>	The file number of the file to read.

Parameters

<i>ullFileOffset</i>	The file offset to read from.
<i>ulLength</i>	The number of bytes to read.
<i>pBuffer</i>	The buffer to populate with the data read. Must be at least <i>ulLength</i> bytes in size.

Returns

The number of bytes read (nonnegative) or a negated [REDSTATUS](#) code indicating the operation result (negative).

Return values

<i>>=0</i>	The number of bytes read from the file.
<i>-RED_EBADF</i>	<i>ulFileNum</i> is not a valid file number.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number or not mounted; or <i>pBuffer</i> is NULL; or <i>ulLength</i> exceeds INT32_MAX and cannot be returned properly.
<i>-RED_EIO</i>	A disk I/O error occurred.

Definition at line 277 of file fse.c.

25.4.3.5 RedFseSizeGet()

```
int64_t RedFseSizeGet (
    uint8_t bVolNum,
    uint32_t ulFileNum )
```

Retrieve the size of a file.

Parameters

<i>bVolNum</i>	The volume number of the file whose size is being read.
<i>ulFileNum</i>	The file number of the file whose size is being read.

Returns

The size of the file (nonnegative) or a negated [REDSTATUS](#) code indicating the operation result (negative).

Return values

<i>>=0</i>	The size of the file.
<i>-RED_EBADF</i>	<i>ulFileNum</i> is not a valid file number.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number or not mounted.
<i>-RED_EIO</i>	A disk I/O error occurred.

Definition at line 453 of file fse.c.

25.4.3.6 RedFseTransact()

```
REDSTATUS RedFseTransact (
    uint8_t bVolNum )
```

Commit a transaction point.

Reliance Edge is a transactional file system. All modifications, of both metadata and filedata, are initially working state. A transaction point is a process whereby the working state atomically becomes the committed state, replacing the previous committed state. Whenever Reliance Edge is mounted, including after power loss, the state of the file system after mount is the most recent committed state. Nothing from the committed state is ever missing, and nothing from the working state is ever included.

Parameters

<i>bVolNum</i>	The volume number of the volume to transact.
----------------	--

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number or not mounted.
<i>-RED_EIO</i>	A disk I/O error occurred.
<i>-REDEROFS</i>	The file system volume is read-only.

Definition at line 591 of file fse.c.

25.4.3.7 RedFseTransMaskGet()

```
REDSTATUS RedFseTransMaskGet (
    uint8_t bVolNum,
    uint32_t * pulEventMask )
```

Read the transaction mask.

If the volume is read-only, the returned event mask is always zero.

Parameters

<i>bVolNum</i>	The volume number of the volume whose transaction mask is being retrieved.
<i>pulEventMask</i>	Populated with a bitwise-OR'd mask of automatic transaction events which represent the current transaction mode for the volume.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number or not mounted; or <i>ulEventMask</i> is NULL.

Definition at line 551 of file fse.c.

25.4.3.8 RedFseTransMaskSet()

```
REDSTATUS RedFseTransMaskSet (
    uint8_t bVolNum,
    uint32_t ulEventMask )
```

Update the transaction mask.

The following events are available:

- [RED_TRANSACT_UMOUNT](#)
- [RED_TRANSACT_WRITE](#)
- [RED_TRANSACT_TRUNCATE](#)
- [RED_TRANSACT_VOLFULL](#)

The [RED_TRANSACT_MANUAL](#) macro (by itself) may be used to disable all automatic transaction events. The [RED_TRANSACT_MASK](#) macro is a bitmask of all transaction flags, excluding those representing excluded functionality.

Attempting to enable events for excluded functionality will result in an error.

Parameters

<i>bVolNum</i>	The volume number of the volume whose transaction mask is being changed.
<i>ulEventMask</i>	A bitwise-OR'd mask of automatic transaction events to be set as the current transaction mode.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number or not mounted; or <i>ulEventMask</i> contains invalid bits.
<i>-RED_ERROFS</i>	The file system volume is read-only.

Definition at line 514 of file fse.c.

25.4.3.9 RedFseTruncate()

```
REDSTATUS RedFseTruncate (
    uint8_t bVolNum,
    uint32_t ulFileNum,
    uint64_t ullNewFileSize )
```

Truncate a file (set the file size).

Allows the file size to be increased, decreased, or to remain the same. If the file size is increased, the new area is sparse (will read as zeroes). If the file size is decreased, the data beyond the new end-of-file will return to free space once it is no longer part of the committed state (either immediately or after the next transaction point).

This function can fail when the disk is full if `ullNewFileSize` is non-zero. If decreasing the file size, this can be fixed by transacting and trying again: Reliance Edge guarantees that it is possible to perform a truncate of at least one file that decreases the file size after a transaction point. If disk full transactions are enabled, this will happen automatically.

Parameters

<code>bVolNum</code>	The volume number of the file to truncate.
<code>ulFileNum</code>	The file number of the file to truncate.
<code>ullNewFileSize</code>	The new file size, in bytes.

Returns

A negated `REDSTATUS` code indicating the operation result.

Return values

<code>0</code>	Operation was successful.
<code>-RED_EBADF</code>	<code>ulFileNum</code> is not a valid file number.
<code>-RED_EFBIG</code>	<code>ullNewFileSize</code> exceeds the maximum file size.
<code>-RED_EINVAL</code>	<code>bVolNum</code> is an invalid volume number or not mounted.
<code>-RED_EIO</code>	A disk I/O error occurred.
<code>-RED_ENOSPC</code>	Insufficient free space to perform the truncate.
<code>-RED_EROFS</code>	The file system volume is read-only.

Definition at line 419 of file fse.c.

25.4.3.10 RedFseUninit()

```
REDSTATUS RedFseUninit (
    void )
```

Uninitialize the Reliance Edge file system driver.

Tears down the Reliance Edge file system driver. Cannot be used until all Reliance Edge volumes are unmounted. A subsequent call to `RedFseInit()` will initialize the driver again.

If this function is called when the Reliance Edge driver is already uninitialized, it does nothing and returns success.

This function is not thread safe: attempting to uninitialized from multiple threads could leave things in a bad state.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EBUSY</i>	At least one volume is still mounted.

Definition at line 79 of file fse.c.

25.4.3.11 RedFseUnmount()

```
REDSTATUS RedFseUnmount (
    uint8_t bVolNum )
```

Unmount a file system volume.

This function discards the in-memory state for the file system and marks it as unmounted. Subsequent attempts to access the volume will fail until the volume is mounted again.

If unmount automatic transaction points are enabled, this function will commit a transaction point prior to unmounting. If unmount automatic transaction points are disabled, this function will unmount without transacting, effectively discarding the working state.

Before unmounting, this function will wait for any active file system thread to complete by acquiring the FS mutex. The volume will be marked as unmounted before the FS mutex is released, so subsequent FS threads will possibly block and then see an error when attempting to access a volume which is unmounting or unmounted.

If the volume is already unmounted, this function does nothing and returns success.

Parameters

<i>bVolNum</i>	The volume number of the volume to be unmounted.
----------------	--

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number; or the driver is uninitialized.
<i>-RED_EIO</i>	I/O error during unmount automatic transaction point.

Definition at line 191 of file fse.c.

25.4.3.12 RedFseWrite()

```
int32_t RedFseWrite (
    uint8_t bVolNum,
    uint32_t ulFileNum,
    uint64_t ullFileOffset,
    uint32_t ulLength,
    const void * pBuffer )
```

Write to a file.

If the write extends beyond the end-of-file, the file size will be increased.

A short write – where the number of bytes written is less than requested – indicates either that the file system ran out of space but was still able to write some of the request; or that the request would have caused the file to exceed the maximum file size, but some of the data could be written prior to the file size limit.

If an error is returned (negative return), either none of the data was written or a critical error occurred (like an I/O error) and the file system volume will be read-only.

Parameters

<i>bVolNum</i>	The volume number of the file to write.
<i>ulFileNum</i>	The file number of the file to write.
<i>ullFileOffset</i>	The file offset to write at.
<i>ulLength</i>	The number of bytes to write.
<i>pBuffer</i>	The buffer containing the data to be written. Must be at least <i>ulLength</i> bytes in size.

Returns

The number of bytes written (nonnegative) or a negated [REDSTATUS](#) code indicating the operation result (negative).

Return values

>0	The number of bytes written to the file.
-RED_EBADF	<i>ulFileNum</i> is not a valid file number.
-RED_EFBIG	No data can be written to the given file offset since the resulting file size would exceed the maximum file size.
-RED_EINVAL	<i>bVolNum</i> is an invalid volume number or not mounted; or <i>pBuffer</i> is NULL; or <i>ulLength</i> exceeds INT32_MAX and cannot be returned properly.
-RED_EIO	A disk I/O error occurred.
-RED_ENOSPC	No data can be written because there is insufficient free space.
-RED_EROFS	The file system volume is read-only.

Definition at line 352 of file fse.c.

25.5 include/redoserv.h File Reference

Data Structures

- struct **BDEVINFO**

Block device geometry information.

Enumerations

- enum **BDEVOOPENMODE** { **BDEV_O_RDONLY**, **BDEV_O_WRONLY**, **BDEV_O_RDWR** }

Type of access requested when opening a block device.

Functions

- **REDSSTATUS RedOsBDevOpen (uint8_t bVolNum, BDEVOOPENMODE mode)**

Initialize a block device.
- **REDSSTATUS RedOsBDevGetGeometry (uint8_t bVolNum, BDEVINFO *pInfo)**

Return the block device geometry.
- **REDSSTATUS RedOsBDevClose (uint8_t bVolNum)**

Uninitialize a block device.
- **REDSSTATUS RedOsBDevRead (uint8_t bVolNum, uint64_t ullSectorStart, uint32_t ulSectorCount, void *pBuffer)**

Read sectors from a physical block device.
- **REDSSTATUS RedOsBDevWrite (uint8_t bVolNum, uint64_t ullSectorStart, uint32_t ulSectorCount, const void *pBuffer)**

Write sectors to a physical block device.
- **REDSSTATUS RedOsBDevFlush (uint8_t bVolNum)**

Flush any caches beneath the file system.
- **REDSSTATUS RedOsClockInit (void)**

Initialize the real time clock.
- **REDSSTATUS RedOsClockUninit (void)**

Uninitialize the real time clock.
- **uint32_t RedOsClockGetTime (void)**

Get the date/time.
- **REDSSTATUS RedOsTimestampInit (void)**

Initialize the timestamp service.
- **REDSSTATUS RedOsTimestampUninit (void)**

Uninitialize the timestamp service.
- **REDTIMESTAMP RedOsTimestamp (void)**

Retrieve a timestamp.
- **uint64_t RedOsTimePassed (REDTIMESTAMP tsSince)**

Determine how much time has passed since a timestamp was retrieved.

25.5.1 Enumeration Type Documentation

25.5.1.1 BDEOPENMODE

enum `BDEOPENMODE`

Type of access requested when opening a block device.

Enumerator

BDEV_O_RDONLY	Open block device for read access.
BDEV_O_WRONLY	Open block device for write access.
BDEV_O_RDWR	Open block device for read and write access.

Definition at line 12 of file redoserv.h.

25.5.2 Function Documentation

25.5.2.1 RedOsBDevClose()

```
REDSTATUS RedOsBDevClose (
    uint8_t bVolNum )
```

Uninitialize a block device.

This function is called when the file system no longer needs access to a block device. If any resource were allocated by [RedOsBDevOpen\(\)](#) to service block device requests, they should be freed at this time.

Upon successful return, the block device must be in such a state that it can be opened again.

The behavior of calling this function on a block device which is already closed is undefined.

Parameters

<i>bVolNum</i>	The volume number of the volume whose block device is being uninitialized.
----------------	--

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

0	Operation was successful.
-RED_EINVAL	<i>bVolNum</i> is an invalid volume number.

Definition at line 71 of file osbdev.c.

25.5.2.2 RedOsBDevFlush()

```
REDSTATUS RedOsBDevFlush (
    uint8_t bVolNum )
```

Flush any caches beneath the file system.

This function must synchronously flush all software and hardware caches beneath the file system, ensuring that all sectors written previously are committed to permanent storage.

If the environment has no caching beneath the file system, the implementation of this function can do nothing and return success.

The behavior of calling this function is undefined if the block device is closed or if it was opened with [BDEV_O_RDONLY](#).

Parameters

<i>bVolNum</i>	The volume number of the volume whose block device is being flushed.
----------------	--

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number.
<i>-RED_EIO</i>	A disk I/O error occurred.

Definition at line 236 of file osbdev.c.

25.5.2.3 RedOsBDevGetGeometry()

```
REDSTATUS RedOsBDevGetGeometry (
    uint8_t bVolNum,
    BDEVINFO * pInfo )
```

Return the block device geometry.

The behavior of calling this function is undefined if the block device is closed.

Parameters

<i>bVolNum</i>	The volume number of the volume whose block device geometry is being queried.
<i>pInfo</i>	On successful return, populated with the geometry of the block device.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number, or <i>pInfo</i> is NULL.
<i>-RED_EIO</i>	A disk I/O error occurred.
<i>-RED_ENOTSUPP</i>	The geometry cannot be queried on this block device.

Definition at line 108 of file osbdev.c.

25.5.2.4 RedOsBDevOpen()

```
REDSTATUS RedOsBDevOpen (
    uint8_t bVolNum,
    BDEVOOPENMODE mode )
```

Initialize a block device.

This function is called when the file system needs access to a block device.

Upon successful return, the block device should be fully initialized and ready to service read/write/flush/close requests.

The behavior of calling this function on a block device which is already open is undefined.

Parameters

<i>bVolNum</i>	The volume number of the volume whose block device is being initialized.
<i>mode</i>	The open mode, indicating the type of access required.

Returns

A negated **REDSTATUS** code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number.
<i>-RED_EIO</i>	A disk I/O error occurred.

Definition at line 30 of file osbdev.c.

25.5.2.5 RedOsBDevRead()

```
REDSTATUS RedOsBDevRead (
    uint8_t bVolNum,
    uint64_t ullSectorStart,
    uint32_t ulSectorCount,
    void * pBuffer )
```

Read sectors from a physical block device.

The behavior of calling this function is undefined if the block device is closed or if it was opened with **BDEV_O_WRONLY**.

Parameters

<i>bVolNum</i>	The volume number of the volume whose block device is being read from.
<i>ullSectorStart</i>	The starting sector number.
<i>ulSectorCount</i>	The number of sectors to read.
<i>pBuffer</i>	The buffer into which to read the sector data.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number, <i>pBuffer</i> is NULL, or <i>ullStartSector</i> and/or <i>ulSectorCount</i> refer to an invalid range of sectors.
<i>-RED_EIO</i>	A disk I/O error occurred.

Definition at line 147 of file osbdev.c.

25.5.2.6 RedOsBDevWrite()

```
REDSTATUS RedOsBDevWrite (
    uint8_t bVolNum,
    uint64_t ullSectorStart,
    uint32_t ulSectorCount,
    const void * pBuffer )
```

Write sectors to a physical block device.

The behavior of calling this function is undefined if the block device is closed or if it was opened with [BDEV_O_RDONLY](#).

Parameters

<i>bVolNum</i>	The volume number of the volume whose block device is being written to.
<i>ullSectorStart</i>	The starting sector number.
<i>ulSectorCount</i>	The number of sectors to write.
<i>pBuffer</i>	The buffer from which to write the sector data.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number, <i>pBuffer</i> is NULL, or <i>ullStartSector</i> and/or <i>ulSectorCount</i> refer to an invalid range of sectors.
<i>-RED_EIO</i>	A disk I/O error occurred.

Definition at line 191 of file osbdev.c.

25.5.2.7 RedOsClockGetTime()

```
uint32_t RedOsClockGetTime (
    void )
```

Get the date/time.

The behavior of calling this function when the RTC is not initialized is undefined.

Returns

The number of seconds since January 1, 1970 excluding leap seconds (in other words, standard Unix time). If the resolution or epoch of the RTC is different than this, the implementation must convert it to the expected representation.

Definition at line 47 of file osclock.c.

25.5.2.8 RedOsClockInit()

```
REDSTATUS RedOsClockInit (
    void )
```

Initialize the real time clock.

The behavior of calling this function when the RTC is already initialized is undefined.

Returns

A negated **REDSTATUS** code indicating the operation result.

Return values

0	Operation was successful.
---	---------------------------

Definition at line 16 of file osclock.c.

25.5.2.9 RedOsClockUninit()

```
REDSTATUS RedOsClockUninit (
    void )
```

Uninitialize the real time clock.

The behavior of calling this function when the RTC is not initialized is undefined.

Returns

A negated **REDSTATUS** code indicating the operation result.

Return values

0	Operation was successful.
---	---------------------------

Definition at line 31 of file osclock.c.

25.5.2.10 RedOsTimePassed()

```
uint64_t RedOsTimePassed (
    REDTIMESTAMP tsSince )
```

Determine how much time has passed since a timestamp was retrieved.

The behavior of invoking this function when timestamps are not initialized is undefined.

Parameters

<i>tsSince</i>	A timestamp acquired earlier via RedOsTimestamp() .
----------------	---

Returns

The number of microseconds which have passed since *tsSince*.

Definition at line 74 of file ostimestamp.c.

25.5.2.11 RedOsTimestamp()

```
REDTIMESTAMP RedOsTimestamp (
    void )
```

Retrieve a timestamp.

The behavior of invoking this function when timestamps are not initialized is undefined

Returns

A timestamp which can later be passed to [RedOsTimePassed\(\)](#) to determine the amount of time which passed between the two calls.

Definition at line 58 of file ostimestamp.c.

25.5.2.12 RedOsTimestampInit()

```
REDSTATUS RedOsTimestampInit (
    void )
```

Initialize the timestamp service.

The behavior of invoking this function when timestamps are already initialized is undefined.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_ENOSYS</i>	The timestamp service has not been implemented.

Definition at line 20 of file ostimestamp.c.

25.5.2.13 RedOsTimestampUninit()

```
REDSTATUS RedOsTimestampUninit (
    void )
```

Uninitialize the timestamp service.

The behavior of invoking this function when timestamps are not initialized is undefined.

Returns

A negated **REDSTATUS** code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
----------	---------------------------

Definition at line 39 of file ostimestamp.c.

25.6 include/redposix.h File Reference

Interface for the Reliance Edge POSIX-like API.

Data Structures

- struct **REDDIRECT**
Directory entry information.

Macros

- #define **RED_O_RDONLY**
Open for reading only.
- #define **RED_O_WRONLY**
Open for writing only.
- #define **RED_O_RDWR**
Open for reading and writing.
- #define **RED_O_APPEND**
File offset for all writes is end-of-file.
- #define **RED_O_CREAT**

- `#define RED_O_EXCL`

Create the file.
- `#define RED_O_TRUNC`

Error if path already exists.
- `#define red_errno`

Truncate file to size zero.
- `#define red_errno`

Last file system error (errno).

Typedefs

- `typedef struct sREDHANDLE REDDIR`

Opaque directory handle.

Enumerations

- `enum REDWHENCE { RED_SEEK_SET, RED_SEEK_CUR, RED_SEEK_END }`

Positions from which to seek within a file.

Functions

- `int32_t red_init (void)`

Initialize the Reliance Edge file system driver.
- `int32_t red_uninit (void)`

Uninitialize the Reliance Edge file system driver.
- `int32_t red_mount (const char *pszVolume)`

Mount a file system volume.
- `int32_t red_mount2 (const char *pszVolume, uint32_t ulFlags)`

Mount a file system volume with flags.
- `int32_t red_umount (const char *pszVolume)`

Unmount a file system volume.
- `int32_t red_umount2 (const char *pszVolume, uint32_t ulFlags)`

Unmount a file system volume with flags.
- `int32_t red_format (const char *pszVolume)`

Format a file system volume.
- `int32_t red_transact (const char *pszVolume)`

Commit a transaction point.
- `int32_t red_settransmask (const char *pszVolume, uint32_t ulEventMask)`

Update the transaction mask.
- `int32_t red_gettransmask (const char *pszVolume, uint32_t *pulEventMask)`

Read the transaction mask.
- `int32_t red_statvfs (const char *pszVolume, REDSTATFS *pStatvfs)`

Query file system status information.
- `int32_t red_fstrim (const char *pszVolume, uint32_t ulBlockStart, uint32_t ulBlockCount)`

Discard (trim) free blocks within a range of a volume's blocks.
- `int32_t red_sync (void)`

Commits file system updates.

- `int32_t red_open (const char *pszPath, uint32_t ulOpenMode)`
Open a file or directory.
- `int32_t red_unlink (const char *pszPath)`
Delete a file or directory.
- `int32_t red_mkdir (const char *pszPath)`
Create a new directory.
- `int32_t red_rmdir (const char *pszPath)`
Delete a directory.
- `int32_t red_rename (const char *pszOldPath, const char *pszNewPath)`
Rename a file or directory.
- `int32_t red_link (const char *pszPath, const char *pszHardLink)`
Create a hard link.
- `int32_t red_close (int32_t iFildes)`
Close a file descriptor.
- `int32_t red_read (int32_t iFildes, void *pBuffer, uint32_t ulLength)`
Read from an open file.
- `int32_t red_write (int32_t iFildes, const void *pBuffer, uint32_t ulLength)`
Write to an open file.
- `int32_t red_fsync (int32_t iFildes)`
Synchronizes changes to a file.
- `int64_t red_lseek (int32_t iFildes, int64_t llOffset, REDWHENCE whence)`
Move the read/write file offset.
- `int32_t red_ftruncate (int32_t iFildes, uint64_t ullSize)`
Truncate a file to a specified length.
- `int32_t red_fstat (int32_t iFildes, REDSTAT *pStat)`
Get the status of a file or directory.
- `REDDIR * red_opendir (const char *pszPath)`
Open a directory stream for reading.
- `REDDIRENT * red_readdir (REDDIR *pDirStream)`
Read from a directory stream.
- `void red_rewinddir (REDDIR *pDirStream)`
Rewind a directory stream to read it from the beginning.
- `int32_t red_closedir (REDDIR *pDirStream)`
Close a directory stream.
- `int32_t red_chdir (const char *pszPath)`
Change the current working directory (CWD).
- `char * red_getcwd (char *pszBuffer, uint32_t ulBufferSize)`
Get the path of the current working directory (CWD).
- `REDSTATUS * red_errno (void)`
Pointer to where the last file system error (errno) is stored.

25.6.1 Detailed Description

Interface for the Reliance Edge POSIX-like API.

The POSIX-like file system API is the primary file system API for Reliance Edge, which supports the full functionality of the file system. This API aims to be compatible with POSIX where reasonable, but it is simplified considerably to meet the needs of resource-constrained embedded systems. The API has also been extended to provide access to the unique features of Reliance Edge, and to cover areas (like mountins and formatting) which do not have APIs in the POSIX specification.

25.6.2 Macro Definition Documentation

25.6.2.1 red_errno

```
#define red_errno
```

Last file system error (errno).

Under normal circumstances, each task using the file system has an independent `red_errno` value. Applications do not need to worry about one task obliterating an error value that another task needed to read. The value is initially zero. When one of the POSIX-like APIs return an indication of error, `red_errno` is set to an error value.

In some circumstances, `red_errno` will be a global errno location which is shared by multiple tasks. If the calling task is not registered as a file system user and all of the task slots are full, there can be no task-specific errno, so the global errno is used. Likewise, if the file system driver is uninitialized, there are no registered file system users and `red_errno` always refers to the global errno. Under these circumstances, multiple tasks manipulating `red_errno` could be problematic. When the task count is set to one, `red_errno` always refers to the global errno.

Note that `red_errno` is usable as an lvalue; i.e., in addition to reading the error value, the error value can be set:

```
red_errno = 0;
```

Definition at line 78 of file redposix.h.

Referenced by RedFileHandle::flen(), RedFileSystem::format(), RedFileHandle::fstat(), RedFileHandle::fsync(), RedFileHandle::ftruncate(), RedFileSystem::get_trans_mask(), RedFileSystem::link(), RedFileHandle::lseek(), RedFileSystem::mkdir(), RedFileSystem::mount(), RedFileSystem::open(), RedFileSystem::opendir(), RedFileHandle::read(), RedDirHandle::readdir(), RedFileSystem::RedFileSystem(), redfs_format(), RedFileSystem::remove(), RedFileSystem::rename(), RedFileSystem::set_trans_mask(), RedFileSystem::statvfs(), RedDirHandle::telldir(), RedFileSystem::transact(), RedFileSystem::unmount(), and RedFileHandle::write().

25.6.2.2 RED_O_APPEND

```
#define RED_O_APPEND
```

File offset for all writes is end-of-file.

Definition at line 41 of file redposix.h.

Referenced by RedFileSystem::open().

25.6.2.3 RED_O_CREAT

```
#define RED_O_CREAT
```

Create the file.

Definition at line 44 of file redposix.h.

Referenced by RedFileSystem::open().

25.6.2.4 RED_O_EXCL

```
#define RED_O_EXCL
```

Error if path already exists.

Definition at line 47 of file redposix.h.

Referenced by RedFileSystem::open().

25.6.2.5 RED_O_RDONLY

```
#define RED_O_RDONLY
```

Open for reading only.

Definition at line 32 of file redposix.h.

Referenced by RedFileSystem::open().

25.6.2.6 RED_O_RDWR

```
#define RED_O_RDWR
```

Open for reading and writing.

Definition at line 38 of file redposix.h.

Referenced by RedFileSystem::open().

25.6.2.7 RED_O_TRUNC

```
#define RED_O_TRUNC
```

Truncate file to size zero.

Definition at line 50 of file redposix.h.

Referenced by RedFileSystem::open().

25.6.2.8 RED_O_WRONLY

```
#define RED_O_WRONLY
```

Open for writing only.

Definition at line 35 of file redposix.h.

Referenced by RedFileSystem::open().

25.6.3 Enumeration Type Documentation

25.6.3.1 REDWHENCE

enum [REDWHENCE](#)

Positions from which to seek within a file.

Enumerator

RED_SEEK_SET	Set file offset to given offset.
RED_SEEK_CUR	Set file offset to current offset plus signed offset.
RED_SEEK_END	Set file offset to EOF plus signed offset.

Definition at line 83 of file redposix.h.

25.6.4 Function Documentation

25.6.4.1 [red_chdir\(\)](#)

```
int32_t red_chdir (
    const char * pszPath )
```

Change the current working directory (CWD).

The default CWD, if it has never been set since the file system was initialized, is the root directory of volume zero. If the CWD is on a volume that is unmounted, it resets to the root directory of that volume.

Parameters

pszPath	The path to the directory which will become the current working directory.
-------------------------	--

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EINVAL](#): `pszPath` is NULL; or the volume containing the path is not mounted.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_ENAMETOOLONG](#): The length of a component of `pszPath` is longer than [REDCONF_NAME_MAX](#).
- [RED_ENOENT](#): A component of `pszPath` does not name an existing directory; or the volume does not exist; or the `pszPath` argument points to an empty string (and there is no volume with an empty path prefix).
- [RED_ENOTDIR](#): A component of `pszPath` does not name a directory.

- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 2315 of file posix.c.

25.6.4.2 red_close()

```
int32_t red_close (
    int32_t iFildes )
```

Close a file descriptor.

Parameters

<i>iFildes</i>	The file descriptor to close.
----------------	-------------------------------

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBADF](#): *iFildes* is not a valid file descriptor.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 1484 of file posix.c.

Referenced by RedFileHandle::close().

25.6.4.3 red_closedir()

```
int32_t red_closedir (
    REDDIR * pDirStream )
```

Close a directory stream.

After calling this function, *pDirStream* should no longer be used.

Parameters

<i>pDirStream</i>	The directory stream to close.
-------------------	--------------------------------

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBADF](#): *pDirStream* is not an open directory stream.

- **RED_EUSERS**: Cannot become a file system user: too many users.

Definition at line 2262 of file posix.c.

Referenced by RedDirHandle::closedir().

25.6.4.4 red_errnoptr()

```
REDSTATUS* red_errnoptr (
    void )
```

Pointer to where the last file system error (errno) is stored.

This function is intended to be used via the `red_errno` macro, or a similar user-defined macro, that can be used both as an lvalue (writable) and an rvalue (readable).

Under normal circumstances, the errno for each task is stored in a different location. Applications do not need to worry about one task obliterating an error value that another task needed to read. This task errno for is initially zero. When one of the POSIX-like APIs returns an indication of error, the location for the calling task will be populated with the error value.

In some circumstances, this function will return a pointer to a global errno location which is shared by multiple tasks. If the calling task is not registered as a file system user and all of the task slots are full, there can be no task-specific errno, so the global pointer is returned. Likewise, if the file system driver is uninitialized, there are no registered file system users and this function always returns the pointer to the global errno. Under these circumstances, multiple tasks manipulating errno could be problematic.

This function never returns `NULL` under any circumstances. The `red_errno` macro unconditionally dereferences the return value from this function, so returning `NULL` could result in a fault.

Returns

Pointer to where the errno value is stored for this task.

Definition at line 2642 of file posix.c.

25.6.4.5 red_format()

```
int32_t red_format (
    const char * pszVolume )
```

Format a file system volume.

Uses the statically defined volume configuration. After calling this function, the volume needs to be mounted – see `red_mount()`.

An error is returned if the volume is mounted.

Parameters

<code>pszVolume</code>	A path prefix identifying the volume to format.
------------------------	---

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBUSY](#): Volume is mounted.
- [RED_EINVAL](#): pszVolume is NULL; or the driver is uninitialized.
- [RED_EIO](#): I/O error formatting the volume.
- [RED_ENOENT](#): pszVolume is not a valid volume path prefix.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 637 of file posix.c.

Referenced by RedFileSystem::format(), and redfs_format().

25.6.4.6 red_fstat()

```
int32_t red_fstat (
    int32_t iFildes,
    REDSTAT * pStat )
```

Get the status of a file or directory.

See the [REDSSTAT](#) type for the details of the information returned.

Parameters

<i>iFildes</i>	An open file descriptor for the file whose information is to be retrieved.
<i>pStat</i>	Pointer to a REDSSTAT buffer to populate.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBADF](#): The *iFildes* argument is not a valid file descriptor.
- [RED_EINVAL](#): *pStat* is NULL.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 2035 of file posix.c.

Referenced by RedFileHandle::flen(), RedFileHandle::fstat(), and RedFileHandle::lseek().

25.6.4.7 red_fstrim()

```
int32_t red_fstrim (
    const char * pszVolume,
    uint32_t ulBlockStart,
    uint32_t ulBlockCount )
```

Discard (trim) free blocks within a range of a volume's blocks.

A transaction point will be unconditionally committed prior to discarding any free blocks.

Discards can reduce the garbage collection workload for flash memory storage devices, thereby improving the performance of write operations. However, the cost of discarding at run-time can be high on some devices; and in some cases, poorly designed devices may even wear out faster if discards are issued continuously. This function is an alternative that allows these discards to happen less frequently, and in larger batches, to provide their benefit while reducing their cost. On Linux, the fstrim ioctl and utility provide similar functionality.

A second application of this function is to execute the first step of wiping the volume of any data that previously resided in blocks that are now free space, for example, deleted files. The second step is for the application to issue a command such as eMMC sanitize. On eMMC, the discards by themselves are not guaranteed to remove the discarded data from the underlying flash memory, but the sanitize command *is* guaranteed to remove any discarded data.

If the range specifies blocks which are beyond the end of the volume, such blocks will be ignored.

Parameters

<i>pszVolume</i>	The path prefix of the volume to fstrim.
<i>ulBlockStart</i>	The start of the range of blocks to discard if free.
<i>ulBlockCount</i>	The length of the range of blocks to discard if free.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EINVAL](#): Volume is not mounted; or *pszVolume* is NULL; or *ulBlockCount* is zero.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_ENOENT](#): *pszVolume* is not a valid volume path prefix.
- [RED_ENOTSUPP](#): Discards are not supported by the volume's block device.
- [RED_EROFS](#): The file system volume is read-only.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 894 of file posix.c.

25.6.4.8 red_fsync()

```
int32_t red_fsync (
    int32_t iFildes )
```

Synchronizes changes to a file.

Commits all changes associated with a file or directory (including file data, directory contents, and metadata) to permanent storage. This function will not return until the operation is complete.

In the current implementation, this function has global effect. All dirty buffers are flushed and a transaction point is committed. Fsyncing one file effectively fsyncs all files.

If fsync automatic transactions have been disabled, this function does nothing and returns success. In the current implementation, this is the only real difference between this function and [red_transact\(\)](#): this function can be configured to do nothing, whereas [red_transact\(\)](#) is unconditional.

Applications written for portability should avoid assuming [red_fsync\(\)](#) effects all files, and use [red_fsync\(\)](#) on each file that needs to be synchronized.

Passing read-only file descriptors to this function is permitted.

Parameters

<i>iFildes</i>	The file descriptor to synchronize.
----------------	-------------------------------------

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBADF](#): The *iFildes* argument is not a valid file descriptor.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 1755 of file posix.c.

Referenced by RedFileHandle::fsync().

25.6.4.9 red_ftruncate()

```
int32_t red_ftruncate (
    int32_t iFildes,
    uint64_t ullSize )
```

Truncate a file to a specified length.

Allows the file size to be increased, decreased, or to remain the same. If the file size is increased, the new area is sparse (will read as zeroes). If the file size is decreased, the data beyond the new end-of-file will return to free space once it is no longer part of the committed state (either immediately or after the next transaction point).

The value of the file offset is not modified by this function.

Unlike POSIX ftruncate, this function can fail when the disk is full if *ullSize* is non-zero. If decreasing the file size, this can be fixed by transacting and trying again: Reliance Edge guarantees that it is possible to perform a truncate of at least one file that decreases the file size after a transaction point. If disk full transactions are enabled, this will happen automatically.

Parameters

<i>iFildes</i>	The file descriptor of the file to truncate.
<i>ullSize</i>	The new size of the file.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBADF](#): The *iFildes* argument is not a valid file descriptor open for writing. This includes the case where the file descriptor is for a directory.
- [RED_EFBIG](#): *ullSize* exceeds the maximum file size.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_ENOSPC](#): Insufficient free space to perform the truncate.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 1973 of file posix.c.

Referenced by RedFileHandle::ftruncate().

25.6.4.10 red_getcwd()

```
char* red_getcwd (
    char * pszBuffer,
    uint32_t ulBufferSize )
```

Get the path of the current working directory (CWD).

The default CWD, if it has never been set since the file system was initialized, is the root directory of volume zero. If the CWD is on a volume that is unmounted, it resets to the root directory of that volume.

Note

Reliance Edge does not have a maximum path length; paths, including the CWD path, can be arbitrarily long. Thus, no buffer is guaranteed to be large enough to store the CWD. If it is important that calls to this function succeed, you need to analyze your application to determine the maximum length of the CWD path. Alternatively, if dynamic memory allocation is used, this function can be called in a loop, with the buffer size increasing if the function fails with a RED_ERANGE error; repeat until the call succeeds.

Parameters

<i>pszBuffer</i>	The buffer to populate with the CWD.
<i>ulBufferSize</i>	The size in bytes of <i>pszBuffer</i> .

Returns

On success, `pszBuffer` is returned. On error, `NULL` is returned and `red_errno` is set appropriately.

Errno values

- `RED_EINVAL`: `pszBuffer` is `NULL`; or `ulBufferSize` is zero.
- `RED_EIO`: A disk I/O error occurred.
- `RED_ERANGE`: `ulBufferSize` is greater than zero but too small for the CWD path string.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 2409 of file posix.c.

25.6.4.11 red_gettransmask()

```
int32_t red_gettransmask (
    const char * pszVolume,
    uint32_t * pulEventMask )
```

Read the transaction mask.

If the volume is read-only, the returned event mask is always zero.

Parameters

<code>pszVolume</code>	The path prefix of the volume whose transaction mask is being retrieved.
<code>pulEventMask</code>	Populated with a bitwise-OR'd mask of automatic transaction events which represent the current transaction mode for the volume.

Returns

On success, zero is returned. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EINVAL`: Volume is not mounted; or `pszVolume` is `NULL`; or `pulEventMask` is `NULL`.
- `RED_ENOENT`: `pszVolume` is not a valid volume path prefix.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 788 of file posix.c.

Referenced by `RedFileSystem::get_trans_mask()`.

25.6.4.12 red_init()

```
int32_t red_init (
    void )
```

Initialize the Reliance Edge file system driver.

Prepares the Reliance Edge file system driver to be used. Must be the first Reliance Edge function to be invoked: no volumes can be mounted or formatted until the driver has been initialized.

If this function is called when the Reliance Edge driver is already initialized, it does nothing and returns success.

This function is not thread safe: attempting to initialize from multiple threads could leave things in a bad state.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EINVAL](#): The volume path prefix configuration is invalid.

Definition at line 181 of file posix.c.

Referenced by RedFileSystem::RedFileSystem(), and redfs_format().

25.6.4.13 red_link()

```
int32_t red_link (
    const char * pszPath,
    const char * pszHardLink )
```

Create a hard link.

This creates an additional name (link) for the file named by `pszPath`. The new name refers to the same file with the same contents. If a name is deleted, but the underlying file has other names, the file continues to exist. The link count (accessible via [red_fstat\(\)](#)) indicates the number of names that a file has. All of a file's names are on equal footing: there is nothing special about the original name.

If `pszPath` names a directory, the operation will fail.

Parameters

<code>pszPath</code>	The path indicating the inode for the new link.
<code>pszHardLink</code>	The name and location for the new link.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EEXIST](#): `pszHardLink` resolves to an existing file.
- [RED_EINVAL](#): `pszPath` or `pszHardLink` is NULL; or the volume containing the paths is not mounted; or `REDCONF_API_POSIX_CWD` is true and `pszHardLink` ends with dot or dot-dot.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_EMLINK](#): Creating the link would exceed the maximum link count of the inode named by `pszPath`.

- [RED_ENAMETOOLONG](#): The length of a component of either `pszPath` or `pszHardLink` is longer than `REDCONF_NAME_MAX`.
- [RED_ENOENT](#): A component of either path prefix does not exist; or the file named by `pszPath` does not exist; or either `pszPath` or `pszHardLink` point to an empty string (and there is no volume with an empty path prefix).
- [RED_ENOSPC](#): There is insufficient free space to expand the directory that would contain the link.
- [RED_ENOTDIR](#): A component of either path prefix is not a directory.
- [RED_EPERM](#): The `pszPath` argument names a directory.
- [RED_EROFS](#): The requested link requires writing in a directory on a read-only file system.
- [RED_EUSERS](#): Cannot become a file system user: too many users.
- [RED_EXDEV](#): `pszPath` and `pszHardLink` are on different file system volumes.

Definition at line 1418 of file posix.c.

Referenced by `RedFileSystem::link()`.

25.6.4.14 red_lseek()

```
int64_t red_lseek (
    int32_t iFildes,
    int64_t llOffset,
    REDWHENCE whence )
```

Move the read/write file offset.

The file offset of the `iFildes` file descriptor is set to `llOffset`, relative to some starting position. The available positions are:

- [RED_SEEK_SET](#) Seek from the start of the file. In other words, `llOffset` becomes the new file offset.
- [RED_SEEK_CUR](#) Seek from the current file offset. In other words, `llOffset` is added to the current file offset.
- [RED_SEEK_END](#) Seek from the end-of-file. In other words, the new file offset is the file size plus `llOffset`.

Since `llOffset` is signed (can be negative), it is possible to seek backward with [RED_SEEK_CUR](#) or [RED_SEEK_END](#).

It is permitted to seek beyond the end-of-file; this does not increase the file size (a subsequent `red_write()` call would).

Unlike POSIX `lseek`, this function cannot be used with directory file descriptors.

Parameters

<code>iFildes</code>	The file descriptor whose offset is to be updated.
<code>lOffset</code>	The new file offset, relative to <code>whence</code> .
<code>whence</code>	The location from which <code>lOffset</code> should be applied.

Returns

On success, returns the new file position, measured in bytes from the beginning of the file. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EBADF`: The `iFildes` argument is not an open file descriptor.
- `RED_EINVAL`: whence is not a valid `RED_SEEK_` value; or the resulting file offset would be negative or beyond the maximum file size.
- `RED_EIO`: A disk I/O error occurred.
- `RED_EISDIR`: The `iFildes` argument is a file descriptor for a directory.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 1834 of file posix.c.

Referenced by `RedFileHandle::lseek()`.

25.6.4.15 `red_mkdir()`

```
int32_t red_mkdir (
    const char * pszPath )
```

Create a new directory.

Unlike POSIX `mkdir`, this function has no second argument for the permissions (which Reliance Edge does not use).

Parameters

<code>pszPath</code>	The name and location of the directory to create.
----------------------	---

Returns

On success, zero is returned. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EEXIST`: `pszPath` points to an existing file or directory.
- `RED_EINVAL`: `pszPath` is NULL; or the volume containing the path is not mounted; or the path ends with dot or dot-dot.
- `RED_EIO`: A disk I/O error occurred.
- `RED_ENAMETOOLONG`: The length of a component of `pszPath` is longer than `REDCONF_NAME_MAX`.
- `RED_ENOENT`: A component of the path prefix does not name an existing directory; or the `pszPath` argument points to an empty string (and there is no volume with an empty path prefix).
- `RED_ENOSPC`: The file system does not have enough space for the new directory or to extend the parent directory of the new directory.

- [RED_ENOTDIR](#): A component of the path prefix is not a directory.
- [RED_EROFS](#): The parent directory resides on a read-only file system.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 1129 of file posix.c.

Referenced by RedFileSystem::mkdir().

25.6.4.16 red_mount()

```
int32_t red_mount (
    const char * pszVolume )
```

Mount a file system volume.

Prepares the file system volume to be accessed. Mount will fail if the volume has never been formatted, or if the on-disk format is inconsistent with the compile-time configuration.

An error is returned if the volume is already mounted.

Parameters

<i>pszVolume</i>	A path prefix identifying the volume to mount.
------------------	--

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBUSY](#): Volume is already mounted.
- [RED_EINVAL](#): *pszVolume* is NULL; or the driver is uninitialized.
- [RED_EIO](#): Volume not formatted, improperly formatted, or corrupt.
- [RED_ENOENT](#): *pszVolume* is not a valid volume path prefix.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 378 of file posix.c.

Referenced by RedFileSystem::mount().

25.6.4.17 red_mount2()

```
int32_t red_mount2 (
    const char * pszVolume,
    uint32_t ulFlags )
```

Mount a file system volume with flags.

Prepares the file system volume to be accessed. Mount will fail if the volume has never been formatted, or if the on-disk format is inconsistent with the compile-time configuration.

An error is returned if the volume is already mounted.

The following mount flags are available:

- [RED_MOUNT_READONLY](#): If specified, the volume will be mounted read-only. All write operations will fail, setting [red_errno](#) to [RED_EROFS](#).
- [RED_MOUNT_DISCARD](#): If specified, and if the underlying block device supports discards, discards will be issued for blocks that become free. If the underlying block device does *not* support discards, then this flag has no effect.

The [RED_MOUNT_DEFAULT](#) macro can be used to mount with the default mount flags, which is equivalent to mounting with [red_mount\(\)](#).

Parameters

<i>pszVolume</i>	A path prefix identifying the volume to mount.
<i>ulFlags</i>	A bitwise-OR'd mask of mount flags.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBUSY](#): Volume is already mounted.
- [RED_EINVAL](#): *pszVolume* is NULL; or the driver is uninitialized; or *ulFlags* includes invalid mount flags.
- [RED_EIO](#): Volume not formatted, improperly formatted, or corrupt.
- [RED_ENOENT](#): *pszVolume* is not a valid volume path prefix.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 419 of file posix.c.

Referenced by [red_mount\(\)](#).

25.6.4.18 red_open()

```
int32_t red_open (
    const char * pszPath,
    uint32_t ulOpenMode )
```

Open a file or directory.

Exactly one file access mode must be specified:

- [RED_O_RDONLY](#): Open for reading only.
- [RED_O_WRONLY](#): Open for writing only.

- [RED_O_RDWR](#): Open for reading and writing.

Directories can only be opened with [RED_O_RDONLY](#).

The following flags may also be used:

- [RED_O_APPEND](#): Set the file offset to the end-of-file prior to each write.
- [RED_O_CREAT](#): Create the named file if it does not exist.
- [RED_O_EXCL](#): In combination with [RED_O_CREAT](#), return an error if the path already exists.
- [RED_O_TRUNC](#): Truncate the opened file to size zero. Only supported when [REDCONF_API_POSIX_FTRUNCATE](#) is true.

[RED_O_CREAT](#), [RED_O_EXCL](#), and [RED_O_TRUNC](#) are invalid with [RED_O_RDONLY](#). [RED_O_EXCL](#) is invalid without [RED_O_CREAT](#).

If the volume is read-only, [RED_O_RDONLY](#) is the only valid open flag; use of any other flag will result in an error.

If [RED_O_TRUNC](#) frees data which is in the committed state, it will not return to free space until after a transaction point.

The returned file descriptor must later be closed with [red_close\(\)](#).

Unlike POSIX open, there is no optional third argument for the permissions (which Reliance Edge does not use) and other open flags (like [O_SYNC](#)) are not supported.

Parameters

<i>pszPath</i>	The path to the file or directory.
<i>ulOpenMode</i>	The open flags (mask of RED_O_ values).

Returns

On success, a nonnegative file descriptor is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EEXIST](#): Using [RED_O_CREAT](#) and [RED_O_EXCL](#), and the indicated path already exists.
- [RED_EINVAL](#): *ulOpenMode* is invalid; or *pszPath* is NULL; or the volume containing the path is not mounted; or [RED_O_CREAT](#) is included in *ulOpenMode*, and the path ends with dot or dot-dot.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_EISDIR](#): The path names a directory and *ulOpenMode* includes [RED_O_WRONLY](#) or [RED_O_RDWR](#).
- [RED_EMFILE](#): There are no available file descriptors.
- [RED_ENAMETOOLONG](#): The length of a component of *pszPath* is longer than [REDCONF_NAME_MAX](#).
- [RED_ENFILE](#): Attempting to create a file but the file system has used all available inode slots.
- [RED_ENOENT](#): [RED_O_CREAT](#) is not set and the named file does not exist; or [RED_O_CREAT](#) is set and the parent directory does not exist; or the volume does not exist; or the *pszPath* argument points to an empty string (and there is no volume with an empty path prefix).

- **RED_ENOSPC**: The file does not exist and **RED_O_CREAT** was specified, but there is insufficient free space to expand the directory or to create the new file.
- **RED_ENOTDIR**: A component of the prefix in **pszPath** does not name a directory.
- **RED_EROFS**: The path resides on a read-only file system and a write operation was requested.
- **RED_EUSERS**: Cannot become a file system user: too many users.

Definition at line 987 of file posix.c.

Referenced by RedFileSystem::open().

25.6.4.19 red_opendir()

```
REDDIR* red_opendir (
    const char * pszPath )
```

Open a directory stream for reading.

Parameters

<i>pszPath</i>	The path of the directory to open.
----------------	------------------------------------

Returns

On success, returns a pointer to a **REDDIR** object that can be used with **red_readdir()** and **red_closedir()**. On error, returns **NULL** and **red_errno** is set appropriately.

Errno values

- **RED EINVAL**: **pszPath** is **NULL**; or the volume containing the path is not mounted.
- **RED EIO**: A disk I/O error occurred.
- **RED ENOENT**: A component of **pszPath** does not exist; or the **pszPath** argument points to an empty string (and there is no volume with an empty path prefix).
- **RED ENOTDIR**: A component of **pszPath** is not a directory.
- **RED EMFILE**: There are no available file descriptors.
- **RED EUSERS**: Cannot become a file system user: too many users.

Definition at line 2087 of file posix.c.

Referenced by RedFileSystem::opendir().

25.6.4.20 red_read()

```
int32_t red_read (
    int32_t iFildes,
```

```
void * pBuffer,
uint32_t ulLength )
```

Read from an open file.

The read takes place at the file offset associated with `iFildes` and advances the file offset by the number of bytes actually read.

Data which has not yet been written, but which is before the end-of-file (sparse data), will read as zeroes. A short read – where the number of bytes read is less than requested – indicates that the requested read was partially or, if zero bytes were read, entirely beyond the end-of-file.

Parameters

<code>iFildes</code>	The file descriptor from which to read.
<code>pBuffer</code>	The buffer to populate with data read. Must be at least <code>ulLength</code> bytes in size.
<code>ulLength</code>	Number of bytes to attempt to read.

Returns

On success, returns a nonnegative value indicating the number of bytes actually read. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EBADF`: The `iFildes` argument is not a valid file descriptor open for reading.
- `RED_EINVAL`: `pBuffer` is NULL; or `ulLength` exceeds INT32_MAX and cannot be returned properly.
- `RED_EIO`: A disk I/O error occurred.
- `RED_EISDIR`: The `iFildes` is a file descriptor for a directory.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 1529 of file posix.c.

Referenced by `RedFileHandle::read()`.

25.6.4.21 `red_readdir()`

```
REDDIRENT* red_readdir (
    REDDIR * pDirStream )
```

Read from a directory stream.

The `REDDIRENT` pointer returned by this function will be overwritten by subsequent calls on the same `pDir`. Calls with other `REDDIR` objects will *not* modify the returned `REDDIRENT`.

If files are added to the directory after it is opened, the new files may or may not be returned by this function. If files are deleted, the deleted files will not be returned.

This function (like its POSIX equivalent) returns NULL in two cases: on error and when the end of the directory is reached. To distinguish between these two cases, the application should set `red_errno` to zero before calling this function, and if NULL is returned, check if `red_errno` is still zero. If it is, the end of the directory was reached; otherwise, there was an error.

Parameters

<i>pDirStream</i>	The directory stream to read from.
-------------------	------------------------------------

Returns

On success, returns a pointer to a [REDDIRECT](#) object which is populated with directory entry information read from the directory. On error, returns `NULL` and [red_errno](#) is set appropriately. If at the end of the directory, returns `NULL` but [red_errno](#) is not modified.

Errno values

- [RED_EBADF](#): *pDirStream* is not an open directory stream.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 2150 of file posix.c.

Referenced by [RedDirHandle::readdir\(\)](#).

25.6.4.22 red_rename()

```
int32_t red_rename (
    const char * pszOldPath,
    const char * pszNewPath )
```

Rename a file or directory.

Both paths must reside on the same file system volume. Attempting to use this API to move a file to a different volume will result in an error.

If *pszNewPath* names an existing file or directory, the behavior depends on the configuration. If [REDCONF_RENAME_ATOMIC](#) is false, and if the destination name exists, this function always fails and sets [red_errno](#) to [RED_EEXIST](#). This behavior is contrary to POSIX.

If [REDCONF_RENAME_ATOMIC](#) is true, and if the new name exists, then in one atomic operation, *pszNewPath* is unlinked and *pszOldPath* is renamed to *pszNewPath*. Both *pszNewPath* and *pszOldPath* must be of the same type (both files or both directories). As with [red_unlink\(\)](#), if *pszNewPath* is a directory, it must be empty. The major exception to this behavior is that if both *pszOldPath* and *pszNewPath* are links to the same inode, then the rename does nothing and both names continue to exist. Unlike POSIX rename, if *pszNewPath* points to an inode with a link count of one and open handles (file descriptors or directory streams), the rename will fail with [RED_EBUSY](#).

If the rename deletes the old destination, it may free data in the committed state, which will not return to free space until after a transaction point. Similarly, if the deleted inode was part of the committed state, the inode slot will not be available until after a transaction point.

Parameters

<i>pszOldPath</i>	The path of the file or directory to rename.
<i>pszNewPath</i>	The new name and location after the rename.

Returns

On success, zero is returned. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- **`RED_EBUSY`**: `pszOldPath` or `pszNewPath` names the root directory; or `REDCONF_RENAME_ATOMIC` is true and either a) `pszNewPath` points to an inode with open handles and a link count of one or b) `REDCONF_API_POSIX_CWD` is true and the `pszNewPath` points to an inode which is the CWD of at least one task.
- **`RED_EEXIST`**: `REDCONF_RENAME_ATOMIC` is false and `pszNewPath` exists.
- **`RED_EINVAL`**: `pszOldPath` is NULL; or `pszNewPath` is NULL; or the volume containing the path is not mounted; or `REDCONF_API_POSIX_CWD` is true and either path ends with dot or dot-dot; or `REDCONF_API_POSIX_CWD` is false and `pszNewPath` ends with dot or dot-dot; or the rename is cyclic.
- **`RED_EIO`**: A disk I/O error occurred.
- **`RED_EISDIR`**: The `pszNewPath` argument names a directory and the `pszOldPath` argument names a non-directory.
- **`RED_ENAMETOOLONG`**: The length of a component of either `pszOldPath` or `pszNewPath` is longer than `REDCONF_NAME_MAX`.
- **`RED_ENOENT`**: The link named by `pszOldPath` does not name an existing entry; or either `pszOldPath` or `pszNewPath` point to an empty string (and there is no volume with an empty path prefix).
- **`RED_ENOTDIR`**: A component of either path prefix is not a directory; or `pszOldPath` names a directory and `pszNewPath` names a file.
- **`RED_ENOTEMPTY`**: The path named by `pszNewPath` is a directory which is not empty.
- **`RED_ENOSPC`**: The file system does not have enough space to extend the directory that would contain `pszNewPath`.
- **`RED_EROFS`**: The directory to be removed resides on a read-only file system.
- **`RED_EUSERS`**: Cannot become a file system user: too many users.
- **`RED_EXDEV`**: `pszOldPath` and `pszNewPath` are on different file system volumes.

Definition at line 1298 of file posix.c.

Referenced by `RedFileSystem::rename()`.

25.6.4.23 `red_rewinddir()`

```
void red_rewinddir (
    REDDIR * pDirStream )
```

Rewind a directory stream to read it from the beginning.

Similar to closing the directory object and opening it again, but without the need for the path.

Since this function (like its POSIX equivalent) cannot return an error, it takes no action in error conditions, such as when `pDirStream` is invalid.

Parameters

<i>pDirStream</i>	The directory stream to rewind.
-------------------	---------------------------------

Definition at line 2234 of file posix.c.

Referenced by RedDirHandle::rewinddir().

25.6.4.24 red_rmdir()

```
int32_t red_rmdir (
    const char * pszPath )
```

Delete a directory.

The given directory name is deleted and the corresponding directory inode will be deleted.

Unlike POSIX rmdir, deleting a directory with open handles (file descriptors or directory streams) will fail with an [RED_EBUSY](#) error.

If the path names a directory which is not empty, the deletion will fail. If the path names the root directory of a file system volume, the deletion will fail.

If the path names a regular file, the deletion will fail. This provides type checking and may be useful in cases where an application knows the path to be deleted should name a directory.

If the deletion frees data in the committed state, it will not return to free space until after a transaction point.

Unlike POSIX rmdir, this function can fail when the disk is full. To fix this, transact and try again: Reliance Edge guarantees that it is possible to delete at least one file or directory after a transaction point. If disk full automatic transactions are enabled, this will happen automatically.

Parameters

<i>pszPath</i>	The path of the directory to delete.
----------------	--------------------------------------

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBUSY](#): *pszPath* names the root directory; or *pszPath* points to a directory with open handles; or [REDCONF_API_POSIX_CWD](#) is true and *pszPath* points to the CWD of a task.
- [RED_EINVAL](#): *pszPath* is NULL; or the volume containing the path is not mounted; or [REDCONF_API_POSIX_CWD](#) is true and the path ends with dot or dot-dot.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_ENAMETOOLONG](#): The length of a component of *pszPath* is longer than [REDCONF_NAME_MAX](#).
- [RED_ENOENT](#): The path does not name an existing directory; or the *pszPath* argument points to an empty string (and there is no volume with an empty path prefix).
- [RED_ENOTDIR](#): A component of the path is not a directory.

- [RED_ENOTEMPTY](#): The path names a directory which is not empty.
- [RED_ENOSPC](#): The file system does not have enough space to modify the parent directory to perform the deletion.
- [RED_EROFS](#): The directory to be removed resides on a read-only file system.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 1214 of file posix.c.

25.6.4.25 red_settransmask()

```
int32_t red_settransmask (
    const char * pszVolume,
    uint32_t ulEventMask )
```

Update the transaction mask.

The following events are available:

- [RED_TRANSACT_SYNC](#)
- [RED_TRANSACT_UMOUNT](#)
- [RED_TRANSACT_CREAT](#)
- [RED_TRANSACT_UNLINK](#)
- [RED_TRANSACT_MKDIR](#)
- [RED_TRANSACT_RENAME](#)
- [RED_TRANSACT_LINK](#)
- [RED_TRANSACT_CLOSE](#)
- [RED_TRANSACT_WRITE](#)
- [RED_TRANSACT_FSYNC](#)
- [RED_TRANSACT_TRUNCATE](#)
- [RED_TRANSACT_VOLFULL](#)

The [RED_TRANSACT_MANUAL](#) macro (by itself) may be used to disable all automatic transaction events. The [RED_TRANSACT_MASK](#) macro is a bitmask of all transaction flags, excluding those representing excluded functionality.

Attempting to enable events for excluded functionality will result in an error.

Parameters

<i>pszVolume</i>	The path prefix of the volume whose transaction mask is being changed.
<i>ulEventMask</i>	A bitwise-OR'd mask of automatic transaction events to be set as the current transaction mode.

Returns

On success, zero is returned. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EINVAL`: Volume is not mounted; or `pszVolume` is NULL; or `ulEventMask` contains invalid bits.
- `RED_ENOENT`: `pszVolume` is not a valid volume path prefix.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 745 of file posix.c.

Referenced by `RedFileSystem::set_trans_mask()`.

25.6.4.26 red_statvfs()

```
int32_t red_statvfs (
    const char * pszVolume,
    REDSTATFS * pStatvfs )
```

Query file system status information.

`pszVolume` should name a valid volume prefix or a valid root directory; this differs from POSIX `statvfs`, where any existing file or directory is a valid path.

Parameters

<code>pszVolume</code>	The path prefix of the volume to query.
<code>pStatvfs</code>	The buffer to populate with volume information.

Returns

On success, zero is returned. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EINVAL`: Volume is not mounted; or `pszVolume` is NULL; or `pStatvfs` is NULL.
- `RED_ENOENT`: `pszVolume` is not a valid volume path prefix.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 829 of file posix.c.

Referenced by `RedFileSystem::statvfs()`.

25.6.4.27 red_sync()

```
int32_t red_sync (
    void )
```

Commits file system updates.

Commits all changes on all file system volumes to permanent storage. This function will not return until the operation is complete.

If sync automatic transactions have been disabled for one or more volumes, this function does not commit changes to those volumes, but will still commit changes to any volumes for which automatic transactions are enabled.

If sync automatic transactions have been disabled on all volumes, this function does nothing and returns success.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EIO](#): I/O error during the transaction point.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 312 of file posix.c.

25.6.4.28 red_transact()

```
int32_t red_transact (
    const char * pszVolume )
```

Commit a transaction point.

Reliance Edge is a transactional file system. All modifications, of both metadata and filedata, are initially working state. A transaction point is a process whereby the working state atomically becomes the committed state, replacing the previous committed state. Whenever Reliance Edge is mounted, including after power loss, the state of the file system after mount is the most recent committed state. Nothing from the committed state is ever missing, and nothing from the working state is ever included.

Parameters

<code>pszVolume</code>	A path prefix identifying the volume to transact.
------------------------	---

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EINVAL](#): Volume is not mounted; or `pszVolume` is NULL.
- [RED_EIO](#): I/O error during the transaction point.
- [RED_ENOENT](#): `pszVolume` is not a valid volume path prefix.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 682 of file posix.c.

Referenced by RedFileSystem::transact().

25.6.4.29 red_umount()

```
int32_t red_umount (
    const char * pszVolume )
```

Unmount a file system volume.

This function discards the in-memory state for the file system and marks it as unmounted. Subsequent attempts to access the volume will fail until the volume is mounted again.

If unmount automatic transaction points are enabled, this function will commit a transaction point prior to unmounting. If unmount automatic transaction points are disabled, this function will unmount without transacting, effectively discarding the working state.

Before unmounting, this function will wait for any active file system thread to complete by acquiring the FS mutex. The volume will be marked as unmounted before the FS mutex is released, so subsequent FS threads will possibly block and then see an error when attempting to access a volume which is unmounting or unmounted. If the volume has open handles, the unmount will fail.

An error is returned if the volume is already unmounted.

Parameters

<i>pszVolume</i>	A path prefix identifying the volume to unmount.
------------------	--

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBUSY](#): There are still open handles for this file system volume.
- [RED_EINVAL](#): *pszVolume* is NULL; or the driver is uninitialized; or the volume is already unmounted.
- [RED_EIO](#): I/O error during unmount automatic transaction point.
- [RED_ENOENT](#): *pszVolume* is not a valid volume path prefix.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 506 of file posix.c.

Referenced by RedFileSystem::unmount().

25.6.4.30 red_umount2()

```
int32_t red_umount2 (
    const char * pszVolume,
    uint32_t ulFlags )
```

Unmount a file system volume with flags.

This function is the same as [red_umount\(\)](#), except that it accepts a flags parameter which can change the unmount behavior.

The following unmount flags are available:

- **RED_UNMOUNT_FORCE**: If specified, if the volume has open handles, the handles will be closed. Without this flag, the behavior is to return an **RED_EBUSY** error if the volume has open handles.

The **RED_UNMOUNT_DEFAULT** macro can be used to unmount with the default unmount flags, which is equivalent to unmounting with **red_umount()**.

Parameters

<i>pszVolume</i>	A path prefix identifying the volume to unmount.
<i>ulFlags</i>	A bitwise-OR'd mask of unmount flags.

Returns

On success, zero is returned. On error, -1 is returned and **red_errno** is set appropriately.

Errno values

- **RED_EBUSY**: There are still open handles for this file system volume and the **RED_UNMOUNT_FORCE** flag was *not* specified.
- **RED_EINVAL**: *pszVolume* is NULL; or *ulFlags* includes invalid unmount flags; or the driver is uninitialized; or the volume is already unmounted.
- **RED_EIO**: I/O error during unmount automatic transaction point.
- **RED_ENOENT**: *pszVolume* is not a valid volume path prefix.
- **RED_EUSERS**: Cannot become a file system user: too many users.

Definition at line 543 of file posix.c.

Referenced by **red_umount()**.

25.6.4.31 red_uninit()

```
int32_t red_uninit (
    void )
```

Uninitialize the Reliance Edge file system driver.

Tears down the Reliance Edge file system driver. Cannot be used until all Reliance Edge volumes are unmounted. A subsequent call to **red_init()** will initialize the driver again.

If this function is called when the Reliance Edge driver is already uninitialized, it does nothing and returns success.

This function is not thread safe: attempting to uninitialized from multiple threads could leave things in a bad state.

Returns

On success, zero is returned. On error, -1 is returned and **red_errno** is set appropriately.

Errno values

- **RED_EBUSY**: At least one volume is still mounted.

Definition at line 230 of file posix.c.

Referenced by `redfs_format()`.

25.6.4.32 `red_unlink()`

```
int32_t red_unlink (
    const char * pszPath )
```

Delete a file or directory.

The given name is deleted and the link count of the corresponding inode is decremented. If the link count falls to zero (no remaining hard links), the inode will be deleted.

Unlike POSIX unlink, deleting a file or directory with open handles (file descriptors or directory streams) will fail with an `RED_EBUSY` error. This only applies when deleting an inode with a link count of one; if a file has multiple names (hard links), all but the last name may be deleted even if the file is open.

If the path names a directory which is not empty, the unlink will fail.

If the deletion frees data in the committed state, it will not return to free space until after a transaction point.

Unlike POSIX unlink, this function can fail when the disk is full. To fix this, transact and try again: Reliance Edge guarantees that it is possible to delete at least one file or directory after a transaction point. If disk full automatic transactions are enabled, this will happen automatically.

Parameters

<code>pszPath</code>	The path of the file or directory to delete.
----------------------	--

Returns

On success, zero is returned. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EBUSY`: `pszPath` names the root directory; or `pszPath` points to an inode with open handles and a link count of one; or `REDCONF_API_POSIX_CWD` is true and `pszPath` points to the CWD of a task.
- `RED_EINVAL`: `pszPath` is NULL; or the volume containing the path is not mounted; or `REDCONF_API_POSIX_CWD` is true and the path ends with dot or dot-dot.
- `RED_EIO`: A disk I/O error occurred.
- `RED_ENAMETOOLONG`: The length of a component of `pszPath` is longer than `REDCONF_NAME_MAX`.
- `RED_ENOENT`: The path does not name an existing file; or the `pszPath` argument points to an empty string (and there is no volume with an empty path prefix).
- `RED_ENOTDIR`: A component of the path prefix is not a directory.
- `RED_ENOTEMPTY`: The path names a directory which is not empty.
- `RED_ENOSPC`: The file system does not have enough space to modify the parent directory to perform the deletion.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 1084 of file posix.c.

Referenced by RedFileSystem::remove().

25.6.4.33 red_write()

```
int32_t red_write (
    int32_t iFildes,
    const void * pBuffer,
    uint32_t ulLength )
```

Write to an open file.

The write takes place at the file offset associated with `iFildes` and advances the file offset by the number of bytes actually written. Alternatively, if `iFildes` was opened with `RED_O_APPEND`, the file offset is set to the end-of-file before the write begins, and likewise advances by the number of bytes actually written.

A short write – where the number of bytes written is less than requested – indicates either that the file system ran out of space but was still able to write some of the request; or that the request would have caused the file to exceed the maximum file size, but some of the data could be written prior to the file size limit.

If an error is returned (-1), either none of the data was written or a critical error occurred (like an I/O error) and the file system volume will be read-only.

Parameters

<code>iFildes</code>	The file descriptor to write to.
<code>pBuffer</code>	The buffer containing the data to be written. Must be at least <code>ulLength</code> bytes in size.
<code>ulLength</code>	Number of bytes to attempt to write.

Returns

On success, returns a nonnegative value indicating the number of bytes actually written. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EBADF`: The `iFildes` argument is not a valid file descriptor open for writing. This includes the case where the file descriptor is for a directory.
- `RED_EFBIG`: No data can be written to the current file offset since the resulting file size would exceed the maximum file size.
- `RED_EINVAL`: `pBuffer` is NULL; or `ulLength` exceeds `INT32_MAX` and cannot be returned properly.
- `RED_EIO`: A disk I/O error occurred.
- `RED_ENOSPC`: No data can be written because there is insufficient free space.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 1635 of file posix.c.

Referenced by RedFileHandle::write().

25.7 include/redstat.h File Reference

Data Structures

- struct **REDSTAT**
Status information on an inode.
- struct **REDSTATFS**
Status information on a file system volume.

Macros

- #define **RED_S_IFDIR**
Mode bit for a directory.
- #define **RED_S_IFREG**
Mode bit for a regular file.
- #define **RED_S_ISDIR(m)**
Test for a directory.
- #define **RED_S_ISREG(m)**
Test for a regular file.
- #define **RED_ST_RDONLY**
File system is read-only.
- #define **RED_ST_NOSUID**
File system ignores suid and sgid bits.

25.7.1 Macro Definition Documentation

25.7.1.1 **RED_S_IFDIR**

```
#define RED_S_IFDIR
```

Mode bit for a directory.

Definition at line 8 of file redstat.h.

25.7.1.2 **RED_S_IFREG**

```
#define RED_S_IFREG
```

Mode bit for a regular file.

Definition at line 11 of file redstat.h.

25.7.1.3 RED_ST_NOSUID

```
#define RED_ST_NOSUID
```

File system ignores uid and sgid bits.

Definition at line 26 of file redstat.h.

25.7.1.4 RED_ST_RDONLY

```
#define RED_ST_RDONLY
```

File system is read-only.

Definition at line 23 of file redstat.h.

25.8 os/mbed/RedFileSystem/RedDirHandle.cpp File Reference

Implementation of the [RedDirHandle](#) type for Reliance Edge on ARM mbed.

25.8.1 Detailed Description

Implementation of the [RedDirHandle](#) type for Reliance Edge on ARM mbed.

25.9 os/mbed/RedFileSystem/RedDirHandle.h File Reference

Definition of the [RedDirHandle](#) type for Reliance Edge on ARM mbed.

Data Structures

- struct [red_dirent](#)

Extends the POSIX dirent structure on ARM mbed by adding REDSTAT information for the current directory entry.

- class [RedDirHandle](#)

The Reliance Edge file handle type for ARM mbed.

25.9.1 Detailed Description

Definition of the [RedDirHandle](#) type for Reliance Edge on ARM mbed.

25.10 os/mbed/RedFileSystem/RedFileHandle.cpp File Reference

Implementation of the [RedFileHandle](#) type for Reliance Edge on mbed.

25.10.1 Detailed Description

Implementation of the [RedFileHandle](#) type for Reliance Edge on mbed.

25.11 os/mbed/RedFileSystem/RedFileHandle.h File Reference

Declaration of the [RedFileHandle](#) type for Reliance Edge on ARM mbed.

Data Structures

- class [RedFileHandle](#)

The Reliance Edge file handle type for ARM mbed.

25.11.1 Detailed Description

Declaration of the [RedFileHandle](#) type for Reliance Edge on ARM mbed.

25.12 os/mbed/RedFileSystem/RedFileSystem.cpp File Reference

Implementation of the [RedFileSystem](#) type for Reliance Edge on ARM mbed.

25.12.1 Detailed Description

Implementation of the [RedFileSystem](#) type for Reliance Edge on ARM mbed.

25.13 os/mbed/RedFileSystem/RedFileSystem.h File Reference

Declaration of the [RedFileSystem](#) type for Reliance Edge on ARM mbed.

Data Structures

- class [RedFileSystem](#)

The core implementation of the ARM mbed port of Tuxera's Reliance Edge failsafe filesystem for IoT.

Macros

- #define [REDFS_DEBUG](#)

Enable debug output.

25.13.1 Detailed Description

Declaration of the [RedFileSystem](#) type for Reliance Edge on ARM mbed.

25.13.2 Macro Definition Documentation

25.13.2.1 REDFS_DEBUG

```
#define REDFS_DEBUG
```

Enable debug output.

Set REDFS_DEBUG to 1 to enable debug output when a Reliance Edge call fails.

Definition at line 53 of file RedFileSystem.h.

Referenced by RedFileHandle::flen(), RedFileSystem::format(), RedFileHandle::fstat(), RedFileHandle::fsync(), RedFileHandle::ftruncate(), RedFileSystem::get_trans_mask(), RedFileSystem::link(), RedFileHandle::lseek(), RedFileSystem::mkdir(), RedFileSystem::mount(), RedFileSystem::open(), RedFileSystem::opendir(), RedFileHandle::read(), RedDirHandle::readdir(), RedFileSystem::RedFileSystem(), RedFileSystem::remove(), RedFileSystem::rename(), RedFileSystem::set_trans_mask(), RedFileSystem::statvfs(), RedFileSystem::transact(), RedFileSystem::unmount(), and RedFileHandle::write().

25.14 os/mbed/RedFileSystem/RedMemFileSystem.h File Reference

Implementation of the [RedMemFileSystem](#) type for Reliance Edge on ARM mbed.

Data Structures

- class [RedMemFileSystem](#)

A ramdisk implementation of the ARM mbed port of Tuxera's Reliance Edge failsafe filesystem for IoT.

25.14.1 Detailed Description

Implementation of the [RedMemFileSystem](#) type for Reliance Edge on ARM mbed.

25.15 os/mbed/RedSDFileSystem/RedSDFileSystem.cpp File Reference

Implementation of the [RedSDFileSystem](#) type for Reliance Edge on ARM mbed.

25.15.1 Detailed Description

Implementation of the [RedSDFileSystem](#) type for Reliance Edge on ARM mbed.

25.16 os/mbed/RedSDFileSystem/RedSDFileSystem.h File Reference

Declaration of the [RedSDFileSystem](#) type for Reliance Edge on ARM mbed.

Data Structures

- class [RedSDFileSystem](#)

An SD/MMC implementation of the ARM mbed port of Tuxera's Reliance Edge failsafe filesystem for IoT.

25.16.1 Detailed Description

Declaration of the [RedSDFileSystem](#) type for Reliance Edge on ARM mbed.

25.17 os/mqx/include/redfs_mqx.h File Reference

This file includes the headers needed to use Reliance Edge on MQX and declares any public MQX-specific Reliance Edge methods.

Data Structures

- struct [RED_RENAME_PARAM](#)

Structure used as an argument for the MQX IOCTLs RED_IOCTL_RENAME and RED_IOCTL_LINK.

- struct [RED_DIR_PARAM](#)

Structure used as an argument for the directory traversing MQX IOCTLs.

Macros

- #define [RED_IOCTL_BASE](#)

An abstract number as a base for our IOCTLs.

- #define [RED_IOCTL_TRANSACT](#)

IOCTL code for transacting the volume.

- #define [RED_IOCTL_SET_TRANSMASK](#)

IOCTL code for setting the transaction mask.

- #define [RED_IOCTL_GET_TRANSMASK](#)

IOCTL code for getting the transaction mask.

- #define [RED_IOCTL_STATVFS](#)

IOCTL code for getting volume statistics.

- #define [RED_IOCTL_UNLINK](#)

IOCTL code for unlinking a file or directory.

- #define [RED_IOCTL_MKDIR](#)

IOCTL code for creating a directory.

- #define [RED_IOCTL_RMDIR](#)

IOCTL code for removing a directory.

- #define [RED_IOCTL_RENAME](#)

IOCTL code for renaming a file or directory.

- #define [RED_IOCTL_LINK](#)

IOCTL code for creating a hard link to a file.

- #define [RED_IOCTL_FTRUNCATE](#)

IOCTL code for truncating a file.

- #define [RED_IOCTL_FSTAT](#)

- `#define RED_IOCTL_OPENDIR`
Open a directory for reading.
- `#define RED_IOCTL_READDIR`
Read the next directory entry from an open directory.
- `#define RED_IOCTL_REWINDDIR`
Reset the directory pointer to start at the beginning of the directory.
- `#define RED_IOCTL_CLOSEDIR`
Open a directory for reading.

Functions

- `int32_t redfs_install (const char *pszVolume)`
Install Reliance Edge and mount the given volume.
- `int32_t redfs_uninstall (const char *pszVolume)`
Unmount Reliance Edge from the given volume.
- `int32_t redfs_format (const char *pszVolume)`
Format a Reliance Edge volume.

Variables

- `const REDMQXDEVICE gaRedDeviceConf [2U]`
List of block device properties, in the same order as the VOLCONF array in redconf.c.

25.17.1 Detailed Description

This file includes the headers needed to use Reliance Edge on MQX and declares any public MQX-specific Reliance Edge methods.

In addition to the custom RED_IOCTL commands declared here, Reliance Edge also implements IO_IOCTL_FLUSH_OUTPUT as an fsync command if REDCONF_READ_ONLY is not enabled. No argument is taken.

25.17.2 Macro Definition Documentation

25.17.2.1 RED_IOCTL_BASE

```
#define RED_IOCTL_BASE
```

An abstract number as a base for our IOCTLs.

In FIO, the IOCTL command is encoded as an _mqx_uint, which is not expected to be narrower than 16 bits. It is thought unlikely that any other device would use IOCTL codes within the 0xD1Ax range. (D1A designed to resemble "Datalight," but using three hex digits.)

Definition at line 24 of file redfs_mqx.h.

25.17.2.2 RED_IOCTL_CLOSEDIR

```
#define RED_IOCTL_CLOSEDIR
```

Open a directory for reading.

Argument: RED_DIR_PARAM* (input/output)

- pszDirPath unused.
- pDir (input/output): the directory to close. Set to NULL on success.
- pDrent (output): set to NULL on success.

Definition at line 146 of file redfs_mqx.h.

25.17.2.3 RED_IOCTL_FSTAT

```
#define RED_IOCTL_FSTAT
```

IOCTL code for getting file statistics.

Argument: REDSTAT* (output)

Definition at line 107 of file redfs_mqx.h.

25.17.2.4 RED_IOCTL_FTRUNCATE

```
#define RED_IOCTL_FTRUNCATE
```

IOCTL code for truncating a file.

Argument: _file_size* (input) if using Formatted I/O; _nio_off_t* (input) if using the I/O Subsystem (NIO)

Definition at line 100 of file redfs_mqx.h.

25.17.2.5 RED_IOCTL_GET_TRANSMASK

```
#define RED_IOCTL_GET_TRANSMASK
```

IOCTL code for getting the transaction mask.

Argument: uint32_t* (output)

Definition at line 46 of file redfs_mqx.h.

25.17.2.6 RED_IOCTL_LINK

```
#define RED_IOCTL_LINK
```

IOCTL code for creating a hard link to a file.

Argument: const RED_LINK_PARAM* (input)

Definition at line 91 of file redfs_mqx.h.

25.17.2.7 RED_IOCTL_MKDIR

```
#define RED_IOCTL_MKDIR
```

IOCTL code for creating a directory.

Argument: const char* (null-terminated) (input)

Definition at line 67 of file redfs_mqx.h.

25.17.2.8 RED_IOCTL_OPENDIR

```
#define RED_IOCTL_OPENDIR
```

Open a directory for reading.

Argument: RED_DIR_PARAM* (input/output)

- pszDirPath (input): specifies the path to the directory to open.
- pDir (output): set to the open directory pointer.
- pDrent (output): set to NULL.

Definition at line 117 of file redfs_mqx.h.

25.17.2.9 RED_IOCTL_READDIR

```
#define RED_IOCTL_READDIR
```

Read the next directory entry from an open directory.

Argument: RED_DIR_PARAM* (input/output)

- pszDirPath: unused.
- pDir (input): the directory to read from.
- pDrent (output): set to point to the directory entry information; set to NULL if the end of the directory has been reached.

Definition at line 128 of file redfs_mqx.h.

25.17.2.10 RED_IOCTL_RENAME

```
#define RED_IOCTL_RENAME
```

IOCTL code for renaming a file or directory.

Argument: const RED_RENAME_PARAM* (input)

Definition at line 83 of file redfs_mqx.h.

25.17.2.11 RED_IOCTL_REWINDDIR

```
#define RED_IOCTL_REWINDDIR
```

Reset the directory pointer to start at the beginning of the directory.

Argument: const RED_DIR_PARAM* (input)

- pszDirPath unused.
- pDir (input): the directory to rewind.
- pDirent: unused.

Definition at line 137 of file redfs_mqx.h.

25.17.2.12 RED_IOCTL_RMDIR

```
#define RED_IOCTL_RMDIR
```

IOCTL code for removing a directory.

Argument: const char* (null-terminated) (input)

Definition at line 75 of file redfs_mqx.h.

25.17.2.13 RED_IOCTL_SET_TRANSMASK

```
#define RED_IOCTL_SET_TRANSMASK
```

IOCTL code for setting the transaction mask.

Argument: uint32_t* (input)

Definition at line 39 of file redfs_mqx.h.

25.17.2.14 RED_IOCTL_STATVFS

```
#define RED_IOCTL_STATVFS
```

IOCTL code for getting volume statistics.

Argument: REDSTATFS* (output)

Definition at line 52 of file redfs_mqx.h.

25.17.2.15 RED_IOCTL_TRANSACT

```
#define RED_IOCTL_TRANSACT  
IOCTL code for transacting the volume.  
Argument: none  
Definition at line 31 of file redfs_mqx.h.
```

25.17.2.16 RED_IOCTL_UNLINK

```
#define RED_IOCTL_UNLINK  
IOCTL code for unlinking a file or directory.  
Argument: const char* (null-terminated) (input)  
Definition at line 59 of file redfs_mqx.h.
```

25.17.3 Function Documentation

25.17.3.1 redfs_format()

```
int32_t redfs_format (const char * pszVolume )
```

Format a Reliance Edge volume.

This function is not thread-safe. Attempting to install, uninstall, and/or format volumes from multiple threads could leave things in a bad state.

Parameters

<i>pszVolume</i>	A Reliance Edge volume path prefix.
------------------	-------------------------------------

Returns

On success, zero is returned. On error, a negated NIO_E... (NIO) or a positive MQX_E... (FIO) error code is returned.

Definition at line 182 of file redfs_nio.c.

25.17.3.2 redfs_install()

```
int32_t redfs_install (const char * pszVolume )
```

Install Reliance Edge and mount the given volume.

This function initializes the Reliance Edge driver (if not already initialized), mounts the given volume, and installs the volume in the VFS. This function may be used in succession to install multiple Reliance Edge volumes on the system.

This function is not thread-safe. Attempting to install, uninstall, and/or format volumes from multiple threads could leave things in a bad state.

Parameters

<i>pszVolume</i>	A path prefix identifying the volume to install Reliance Edge on.
------------------	---

Returns

On success, zero is returned. On error, a negated NIO_E... (NIO) or a positive MQX_E... (FIO) error code is returned.

Definition at line 78 of file redfs_nio.c.

25.17.3.3 redfs_uninstall()

```
int32_t redfs_uninstall (
    const char * pszVolume )
```

Unmount Reliance Edge from the given volume.

This function is an alias for `_io_dev_uninstall()`. It uninstalls the given volume from the VFS, unmounts the volume, and uninitializes the filesystem if no other volumes are mounted.

Warning: `_nio_dev_uninstall` will hang indefinitely if there are any open files on this volume.

This function is not thread-safe. Attempting to install, uninstall, and/or format volumes from multiple threads could leave things in a bad state.

Parameters

<i>pszVolume</i>	A Reliance Edge volume path prefix.
------------------	-------------------------------------

Returns

On success, zero is returned. On error, a negated NIO_E... (NIO) or a positive MQX_E... (FIO) error code is returned.

Definition at line 153 of file redfs_nio.c.

25.17.4 Variable Documentation

25.17.4.1 gaRedDeviceConf

```
const REDMQXDEVICE gaRedDeviceConf[2U]
```

List of block device properties, in the same order as the VOLCONF array in redconf.c.

Each element in this array should define the block device name (as given to `_nio_dev_install` or `_io_dev_install[...]` before initializing Reliance Edge) and whether the flushing the device with `IO_IOCTL_FLUSH_OUTPUT` is supported.

Definition at line 13 of file `redosconf.c`.

Referenced by `redfs_format()`, and `redfs_install()`.

25.18 os/mqx/vfs/redfs_fio.c File Reference

Implements the VFS integration for Reliance Edge in the MQX Formatted I/O library.

25.18.1 Detailed Description

Implements the VFS integration for Reliance Edge in the MQX Formatted I/O library.

25.19 os/mqx/vfs/redfs_nio.c File Reference

Implements the VFS integration for Reliance Edge in the MQX NIO library.

Functions

- `int32_t redfs_install (const char *pszVolume)`
Install Reliance Edge and mount the given volume.
- `int32_t redfs_uninstall (const char *pszVolume)`
Unmount Reliance Edge from the given volume.
- `int32_t redfs_format (const char *pszVolume)`
Format a Reliance Edge volume.

25.19.1 Detailed Description

Implements the VFS integration for Reliance Edge in the MQX NIO library.

25.19.2 Function Documentation

25.19.2.1 redfs_format()

```
int32_t redfs_format (
    const char * pszVolume )
```

Format a Reliance Edge volume.

This function is not thread-safe. Attempting to install, uninstall, and/or format volumes from multiple threads could leave things in a bad state.

Parameters

<i>pszVolume</i>	A Reliance Edge volume path prefix.
------------------	-------------------------------------

Returns

On success, zero is returned. On error, a negated NIO_E... (NIO) or a positive MQX_E... (FIO) error code is returned.

Definition at line 182 of file redfs_nio.c.

25.19.2.2 redfs_install()

```
int32_t redfs_install (
    const char * pszVolume )
```

Install Reliance Edge and mount the given volume.

This function initializes the Reliance Edge driver (if not already initialized), mounts the given volume, and installs the volume in the VFS. This function may be used in succession to install multiple Reliance Edge volumes on the system.

This function is not thread-safe. Attempting to install, uninstall, and/or format volumes from multiple threads could leave things in a bad state.

Parameters

<i>pszVolume</i>	A path prefix identifying the volume to install Reliance Edge on.
------------------	---

Returns

On success, zero is returned. On error, a negated NIO_E... (NIO) or a positive MQX_E... (FIO) error code is returned.

Definition at line 78 of file redfs_nio.c.

25.19.2.3 redfs_uninstall()

```
int32_t redfs_uninstall (
    const char * pszVolume )
```

Unmount Reliance Edge from the given volume.

This function is an alias for `_io_dev_uninstall()`. It uninstalls the given volume from the VFS, unmounts the volume, and ununits the filesystem if no other volumes are mounted.

Warning: `_io_dev_uninstall` will hang indefinitely if there are any open files on this volume.

This function is not thread-safe. Attempting to install, uninstall, and/or format volumes from multiple threads could leave things in a bad state.

Parameters

<code>pszVolume</code>	A Reliance Edge volume path prefix.
------------------------	-------------------------------------

Returns

On success, zero is returned. On error, a negated NIO_E... (NIO) or a positive MQX_E... (FIO) error code is returned.

Definition at line 153 of file redfs_nio.c.

25.20 os/stub/include/redostypes.h File Reference

Defines OS-specific types for use in common code.

Typedefs

- `typedef uint64_t REDTIMESTAMP`
Implementation-defined timestamp type.

25.20.1 Detailed Description

Defines OS-specific types for use in common code.

25.20.2 Typedef Documentation

25.20.2.1 REDTIMESTAMP

`typedef uint64_t REDTIMESTAMP`

Implementation-defined timestamp type.

This can be an integer, a structure, or a pointer: anything that is convenient for the implementation. Since the underlying type is not fixed, common code should treat this as an opaque type.

Definition at line 14 of file redostypes.h.

25.21 os/stub/services/osassert.c File Reference

Implements assertion handling.

Functions

- `void RedOsAssertFail (const char *pszFileName, uint32_t ulLineNum)`
Invoke the native assertion handler.

25.21.1 Detailed Description

Implements assertion handling.

25.21.2 Function Documentation

25.21.2.1 RedOsAssertFail()

```
void RedOsAssertFail (
    const char * pszFileName,
    uint32_t ulLineNum )
```

Invoke the native assertion handler.

Parameters

<i>pszFileName</i>	Null-terminated string containing the name of the file where the assertion fired.
<i>ulLineNum</i>	Line number in <i>pszFileName</i> where the assertion fired.

Definition at line 16 of file osassert.c.

25.22 os/stub/services/osbdev.c File Reference

Implements block device I/O.

Functions

- [REDSHIFT RedOsBDevOpen \(uint8_t bVolNum, BDEVOOPENMODE mode\)](#)
Initialize a block device.
- [REDSHIFT RedOsBDevClose \(uint8_t bVolNum\)](#)
Uninitialize a block device.
- [REDSHIFT RedOsBDevGetGeometry \(uint8_t bVolNum, BDEVINFO *plInfo\)](#)
Return the block device geometry.
- [REDSHIFT RedOsBDevRead \(uint8_t bVolNum, uint64_t ullSectorStart, uint32_t ulSectorCount, void *pBuffer\)](#)
Read sectors from a physical block device.
- [REDSHIFT RedOsBDevWrite \(uint8_t bVolNum, uint64_t ullSectorStart, uint32_t ulSectorCount, const void *pBuffer\)](#)
Write sectors to a physical block device.
- [REDSHIFT RedOsBDevFlush \(uint8_t bVolNum\)](#)
Flush any caches beneath the file system.
- [REDSHIFT RedOsBDevDiscard \(uint8_t bVolNum, uint64_t ullSectorStart, uint64_t ullSectorCount\)](#)
Discard (trim) sectors on a physical block device.

25.22.1 Detailed Description

Implements block device I/O.

25.22.2 Function Documentation

25.22.2.1 RedOsBDevClose()

```
REDSTATUS RedOsBDevClose (
    uint8_t bVolNum )
```

Uninitialize a block device.

This function is called when the file system no longer needs access to a block device. If any resource were allocated by [RedOsBDevOpen\(\)](#) to service block device requests, they should be freed at this time.

Upon successful return, the block device must be in such a state that it can be opened again.

The behavior of calling this function on a block device which is already closed is undefined.

Parameters

<i>bVolNum</i>	The volume number of the volume whose block device is being uninitialized.
----------------	--

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	bVolNum is an invalid volume number.

Definition at line 71 of file osbdev.c.

25.22.2.2 RedOsBDevDiscard()

```
REDSTATUS RedOsBDevDiscard (
    uint8_t bVolNum,
    uint64_t ullSectorStart,
    uint64_t ullSectorCount )
```

Discard (trim) sectors on a physical block device.

This function alerts the block device that the given sectors no longer contain information that is important to the filesystem.

The behavior of calling this function is undefined if the block device is closed or if it was opened with [BDEV_O_RDONLY](#).

If discarding fails, the integrity of the volume should not be affected and there is nothing that needs to be done to recover or report the error. So no errors are returned from this function.

Parameters

<i>bVolNum</i>	The volume number of the volume whose block device is being accessed.
----------------	---

Parameters

<i>ullSectorStart</i>	The starting sector number.
<i>ullSectorCount</i>	The number of sectors to discard.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number, or <i>ullStartSector</i> and/or <i>ullSectorCount</i> refer to an invalid range of sectors.
<i>-RED_EIO</i>	A disk I/O error occurred.

Definition at line 281 of file osbdev.c.

25.22.2.3 RedOsBDevFlush()

```
REDSTATUS RedOsBDevFlush (
    uint8\_t bVolNum )
```

Flush any caches beneath the file system.

This function must synchronously flush all software and hardware caches beneath the file system, ensuring that all sectors written previously are committed to permanent storage.

If the environment has no caching beneath the file system, the implementation of this function can do nothing and return success.

The behavior of calling this function is undefined if the block device is closed or if it was opened with [BDEV_O_RDONLY](#).

Parameters

<i>bVolNum</i>	The volume number of the volume whose block device is being flushed.
----------------	--

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number.
<i>-RED_EIO</i>	A disk I/O error occurred.

Definition at line 236 of file osbdev.c.

25.22.2.4 RedOsBDevGetGeometry()

```
REDSTATUS RedOsBDevGetGeometry (
    uint8_t bVolNum,
    BDEVINFO * pInfo )
```

Return the block device geometry.

The behavior of calling this function is undefined if the block device is closed.

Parameters

<i>bVolNum</i>	The volume number of the volume whose block device geometry is being queried.
<i>pInfo</i>	On successful return, populated with the geometry of the block device.

Returns

A negated **REDSTATUS** code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
-RED_EINVAL	<i>bVolNum</i> is an invalid volume number, or <i>pInfo</i> is NULL.
-RED_EIO	A disk I/O error occurred.
-RED_ENOTSUPP	The geometry cannot be queried on this block device.

Definition at line 108 of file osbdev.c.

25.22.2.5 RedOsBDevOpen()

```
REDSTATUS RedOsBDevOpen (
    uint8_t bVolNum,
    BDEOPENMODE mode )
```

Initialize a block device.

This function is called when the file system needs access to a block device.

Upon successful return, the block device should be fully initialized and ready to service read/write/flush/close requests.

The behavior of calling this function on a block device which is already open is undefined.

Parameters

<i>bVolNum</i>	The volume number of the volume whose block device is being initialized.
<i>mode</i>	The open mode, indicating the type of access required.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	bVolNum is an invalid volume number.
<i>-RED_EIO</i>	A disk I/O error occurred.

Definition at line 30 of file osbdev.c.

25.22.2.6 RedOsBDevRead()

```
REDSTATUS RedOsBDevRead (
    uint8_t bVolNum,
    uint64_t ullSectorStart,
    uint32_t ulSectorCount,
    void * pBuffer )
```

Read sectors from a physical block device.

The behavior of calling this function is undefined if the block device is closed or if it was opened with [BDEV_O_WRONLY](#).

Parameters

<i>bVolNum</i>	The volume number of the volume whose block device is being read from.
<i>ullSectorStart</i>	The starting sector number.
<i>ulSectorCount</i>	The number of sectors to read.
<i>pBuffer</i>	The buffer into which to read the sector data.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	bVolNum is an invalid volume number, pBuffer is NULL, or ullStartSector and/or ulSectorCount refer to an invalid range of sectors.
<i>-RED_EIO</i>	A disk I/O error occurred.

Definition at line 147 of file osbdev.c.

25.22.2.7 RedOsBDevWrite()

```
REDSTATUS RedOsBDevWrite (
    uint8_t bVolNum,
    uint64_t ullSectorStart,
    uint32_t ulSectorCount,
    const void * pBuffer )
```

Write sectors to a physical block device.

The behavior of calling this function is undefined if the block device is closed or if it was opened with [BDEV_O_RDONLY](#).

Parameters

<i>bVolNum</i>	The volume number of the volume whose block device is being written to.
<i>ullSectorStart</i>	The starting sector number.
<i>ulSectorCount</i>	The number of sectors to write.
<i>pBuffer</i>	The buffer from which to write the sector data.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_EINVAL</i>	<i>bVolNum</i> is an invalid volume number, <i>pBuffer</i> is NULL, or <i>ullStartSector</i> and/or <i>ulSectorCount</i> refer to an invalid range of sectors.
<i>-RED_EIO</i>	A disk I/O error occurred.

Definition at line 191 of file osbdev.c.

25.23 os/stub/services/osclock.c File Reference

Implements real-time clock functions.

Functions

- [REDSTATUS RedOsClockInit \(void\)](#)

Initialize the real time clock.
- [REDSTATUS RedOsClockUninit \(void\)](#)

Uninitialize the real time clock.
- [uint32_t RedOsClockGetTime \(void\)](#)

Get the date/time.

25.23.1 Detailed Description

Implements real-time clock functions.

25.23.2 Function Documentation

25.23.2.1 RedOsClockGetTime()

```
uint32_t RedOsClockGetTime (
    void )
```

Get the date/time.

The behavior of calling this function when the RTC is not initialized is undefined.

Returns

The number of seconds since January 1, 1970 excluding leap seconds (in other words, standard Unix time). If the resolution or epoch of the RTC is different than this, the implementation must convert it to the expected representation.

Definition at line 47 of file osclock.c.

25.23.2.2 RedOsClockInit()

```
REDSTATUS RedOsClockInit (
    void )
```

Initialize the real time clock.

The behavior of calling this function when the RTC is already initialized is undefined.

Returns

A negated **REDSTATUS** code indicating the operation result.

Return values

0	Operation was successful.
---	---------------------------

Definition at line 16 of file osclock.c.

25.23.2.3 RedOsClockUninit()

```
REDSTATUS RedOsClockUninit (
    void )
```

Uninitialize the real time clock.

The behavior of calling this function when the RTC is not initialized is undefined.

Returns

A negated [REDSSTATUS](#) code indicating the operation result.

Return values

0	Operation was successful.
---	---------------------------

Definition at line 31 of file osclock.c.

25.24 os/stub/services/osmutex.c File Reference

Implements a synchronization object to provide mutual exclusion.

Functions

- [REDSSTATUS RedOsMutexInit \(void\)](#)
Initialize the mutex.
- [REDSSTATUS RedOsMutexUninit \(void\)](#)
Uninitialize the mutex.
- void [RedOsMutexAcquire \(void\)](#)
Acquire the mutex.
- void [RedOsMutexRelease \(void\)](#)
Release the mutex.

25.24.1 Detailed Description

Implements a synchronization object to provide mutual exclusion.

25.24.2 Function Documentation

25.24.2.1 RedOsMutexAcquire()

```
void RedOsMutexAcquire (
    void )
```

Acquire the mutex.

The behavior of calling this function when the mutex is not initialized is undefined; likewise, the behavior of recursively acquiring the mutex is undefined.

Definition at line 57 of file osmutex.c.

25.24.2.2 RedOsMutexInit()

```
REDSTATUS RedOsMutexInit (
    void )
```

Initialize the mutex.

After initialization, the mutex is in the released state.

The behavior of calling this function when the mutex is still initialized is undefined.

Returns

A negated **REDSTATUS** code indicating the operation result.

Return values

0	Operation was successful.
---	---------------------------

Definition at line 20 of file osmutex.c.

25.24.2.3 RedOsMutexRelease()

```
void RedOsMutexRelease (
    void )
```

Release the mutex.

The behavior is undefined in the following cases:

- Releasing the mutex when the mutex is not initialized.
- Releasing the mutex when it is not in the acquired state.
- Releasing the mutex from a task or thread other than the one which acquired the mutex.

Definition at line 72 of file osmutex.c.

25.24.2.4 RedOsMutexUninit()

```
REDSTATUS RedOsMutexUninit (
    void )
```

Uninitialize the mutex.

The behavior of calling this function when the mutex is not initialized is undefined; likewise, the behavior of uninitialized the mutex when it is in the acquired state is undefined.

Returns

A negated **REDSTATUS** code indicating the operation result.

Return values

0	Operation was successful.
---	---------------------------

Definition at line 40 of file osmutex.c.

25.25 os/stub/services/osoutput.c File Reference

Implements outputting a character string.

Functions

- void [RedOsOutputString](#) (const char *pszString)

Write a string to a user-visible output location.

25.25.1 Detailed Description

Implements outputting a character string.

25.25.2 Function Documentation

25.25.2.1 RedOsOutputString()

```
void RedOsOutputString (
    const char * pszString )
```

Write a string to a user-visible output location.

Write a null-terminated string to the serial port, console, terminal, or other display device, such that the text is visible to the user.

Parameters

pszString	A null-terminated string.
-----------	---------------------------

Definition at line 16 of file osoutput.c.

25.26 os/stub/services/ostask.c File Reference

Implements task functions.

Functions

- [uint32_t RedOsTaskId](#) (void)

Get the current task ID.

25.26.1 Detailed Description

Implements task functions.

25.26.2 Function Documentation

25.26.2.1 RedOsTaskId()

```
uint32_t RedOsTaskId (
    void )
```

Get the current task ID.

This task ID must be unique for all tasks using the file system.

Returns

The task ID. Must not be 0.

Definition at line 15 of file ostask.c.

25.27 os/stub/services/ostimestamp.c File Reference

Implements timestamp functions.

Functions

- [REDSSTATUS RedOsTimestampInit \(void\)](#)
Initialize the timestamp service.
- [REDSSTATUS RedOsTimestampUninit \(void\)](#)
Uninitialize the timestamp service.
- [REDTIMESTAMP RedOsTimestamp \(void\)](#)
Retrieve a timestamp.
- [uint64_t RedOsTimePassed \(REDTIMESTAMP tsSince\)](#)
Determine how much time has passed since a timestamp was retrieved.

25.27.1 Detailed Description

Implements timestamp functions.

The functionality implemented herein is not needed for the file system driver, only to provide accurate results with performance tests.

25.27.2 Function Documentation

25.27.2.1 RedOsTimePassed()

```
uint64_t RedOsTimePassed (
    REDTIMESTAMP tsSince )
```

Determine how much time has passed since a timestamp was retrieved.

The behavior of invoking this function when timestamps are not initialized is undefined.

Parameters

<code>tsSince</code>	A timestamp acquired earlier via RedOsTimestamp() .
----------------------	---

Returns

The number of microseconds which have passed since `tsSince`.

Definition at line 74 of file ostimestamp.c.

25.27.2.2 RedOsTimestamp()

```
REDTIMESTAMP RedOsTimestamp (
    void )
```

Retrieve a timestamp.

The behavior of invoking this function when timestamps are not initialized is undefined

Returns

A timestamp which can later be passed to [RedOsTimePassed\(\)](#) to determine the amount of time which passed between the two calls.

Definition at line 58 of file ostimestamp.c.

25.27.2.3 RedOsTimestampInit()

```
REDSTATUS RedOsTimestampInit (
    void )
```

Initialize the timestamp service.

The behavior of invoking this function when timestamps are already initialized is undefined.

Returns

A negated [REDSTATUS](#) code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
<i>-RED_ENOSYS</i>	The timestamp service has not been implemented.

Definition at line 20 of file ostimestamp.c.

25.27.2.4 RedOsTimestampUninit()

```
REDSTATUS RedOsTimestampUninit (
    void )
```

Uninitialize the timestamp service.

The behavior of invoking this function when timestamps are not initialized is undefined.

Returns

A negated **REDSTATUS** code indicating the operation result.

Return values

<i>0</i>	Operation was successful.
----------	---------------------------

Definition at line 39 of file ostimestamp.c.

25.28 posix posix.c File Reference

Implementation of the Reliance Edge POSIX-like API.

Functions

- **int32_t red_init (void)**
Initialize the Reliance Edge file system driver.
- **int32_t red_uninit (void)**
Uninitialize the Reliance Edge file system driver.
- **int32_t red_sync (void)**
Commits file system updates.
- **int32_t red_mount (const char *pszVolume)**
Mount a file system volume.
- **int32_t red_mount2 (const char *pszVolume, uint32_t ulFlags)**
Mount a file system volume with flags.
- **int32_t red_umount (const char *pszVolume)**
Unmount a file system volume.
- **int32_t red_umount2 (const char *pszVolume, uint32_t ulFlags)**
Unmount a file system volume with flags.

- `int32_t red_format (const char *pszVolume)`
Format a file system volume.
- `int32_t red_transact (const char *pszVolume)`
Commit a transaction point.
- `int32_t red_settransmask (const char *pszVolume, uint32_t ulEventMask)`
Update the transaction mask.
- `int32_t red_gettransmask (const char *pszVolume, uint32_t *pulEventMask)`
Read the transaction mask.
- `int32_t red_statvfs (const char *pszVolume, REDSTATFS *pStatvfs)`
Query file system status information.
- `int32_t red_fstrim (const char *pszVolume, uint32_t ulBlockStart, uint32_t ulBlockCount)`
Discard (trim) free blocks within a range of a volume's blocks.
- `int32_t red_open (const char *pszPath, uint32_t ulOpenMode)`
Open a file or directory.
- `int32_t red_unlink (const char *pszPath)`
Delete a file or directory.
- `int32_t red_mkdir (const char *pszPath)`
Create a new directory.
- `int32_t red_rmdir (const char *pszPath)`
Delete a directory.
- `int32_t red_rename (const char *pszOldPath, const char *pszNewPath)`
Rename a file or directory.
- `int32_t red_link (const char *pszPath, const char *pszHardLink)`
Create a hard link.
- `int32_t red_close (int32_t iFildes)`
Close a file descriptor.
- `int32_t red_read (int32_t iFildes, void *pBuffer, uint32_t ulLength)`
Read from an open file.
- `int32_t red_write (int32_t iFildes, const void *pBuffer, uint32_t ulLength)`
Write to an open file.
- `int32_t red_fsync (int32_t iFildes)`
Synchronizes changes to a file.
- `int64_t red_lseek (int32_t iFildes, int64_t llOffset, REDWHENCE whence)`
Move the read/write file offset.
- `int32_t red_ftruncate (int32_t iFildes, uint64_t ullSize)`
Truncate a file to a specified length.
- `int32_t red_fstat (int32_t iFildes, REDSTAT *pStat)`
Get the status of a file or directory.
- `REDDIR * red_opendir (const char *pszPath)`
Open a directory stream for reading.
- `REDDIRENT * red_readdir (REDDIR *pDirStream)`
Read from a directory stream.
- `void red_rewinddir (REDDIR *pDirStream)`
Rewind a directory stream to read it from the beginning.
- `int32_t red_closedir (REDDIR *pDirStream)`
Close a directory stream.
- `int32_t red_chdir (const char *pszPath)`

Change the current working directory (CWD).

- `char * red_getcwd (char *pszBuffer, uint32_t ulBufferSize)`

Get the path of the current working directory (CWD).

- `REDSTATUS * red_errno (void)`

Pointer to where the last file system error (errno) is stored.

25.28.1 Detailed Description

Implementation of the Reliance Edge POSIX-like API.

25.28.2 Function Documentation

25.28.2.1 red_chdir()

```
int32_t red_chdir (
    const char * pszPath )
```

Change the current working directory (CWD).

The default CWD, if it has never been set since the file system was initialized, is the root directory of volume zero. If the CWD is on a volume that is unmounted, it resets to the root directory of that volume.

Parameters

<code>pszPath</code>	The path to the directory which will become the current working directory.
----------------------	--

Returns

On success, zero is returned. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EINVAL`: `pszPath` is NULL; or the volume containing the path is not mounted.
- `RED_EIO`: A disk I/O error occurred.
- `RED_ENAMETOOLONG`: The length of a component of `pszPath` is longer than `REDCONF_NAME_MAX`.
- `RED_ENOENT`: A component of `pszPath` does not name an existing directory; or the volume does not exist; or the `pszPath` argument points to an empty string (and there is no volume with an empty path prefix).
- `RED_ENOTDIR`: A component of `pszPath` does not name a directory.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 2315 of file posix.c.

25.28.2.2 red_close()

```
int32_t red_close (
    int32_t iFildes )
```

Close a file descriptor.

Parameters

iFildes	The file descriptor to close.
---------	-------------------------------

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBADF](#): iFildes is not a valid file descriptor.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 1484 of file posix.c.

Referenced by RedFileHandle::close().

25.28.2.3 red_closedir()

```
int32_t red_closedir (
    REDDIR * pDirStream )
```

Close a directory stream.

After calling this function, pDirStream should no longer be used.

Parameters

pDirStream	The directory stream to close.
------------	--------------------------------

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBADF](#): pDirStream is not an open directory stream.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 2262 of file posix.c.

Referenced by RedDirHandle::closedir().

25.28.2.4 red_errnoptr()

```
REDSTATUS* red_errnoptr (
    void )
```

Pointer to where the last file system error (errno) is stored.

This function is intended to be used via the `red_errno` macro, or a similar user-defined macro, that can be used both as an lvalue (writable) and an rvalue (readable).

Under normal circumstances, the errno for each task is stored in a different location. Applications do not need to worry about one task obliterating an error value that another task needed to read. This task errno for is initially zero. When one of the POSIX-like APIs returns an indication of error, the location for the calling task will be populated with the error value.

In some circumstances, this function will return a pointer to a global errno location which is shared by multiple tasks. If the calling task is not registered as a file system user and all of the task slots are full, there can be no task-specific errno, so the global pointer is returned. Likewise, if the file system driver is uninitialized, there are no registered file system users and this function always returns the pointer to the global errno. Under these circumstances, multiple tasks manipulating errno could be problematic.

This function never returns `NULL` under any circumstances. The `red_errno` macro unconditionally dereferences the return value from this function, so returning `NULL` could result in a fault.

Returns

Pointer to where the errno value is stored for this task.

Definition at line 2642 of file posix.c.

25.28.2.5 red_format()

```
int32_t red_format (
    const char *pszVolume )
```

Format a file system volume.

Uses the statically defined volume configuration. After calling this function, the volume needs to be mounted – see `red_mount()`.

An error is returned if the volume is mounted.

Parameters

<code>pszVolume</code>	A path prefix identifying the volume to format.
------------------------	---

Returns

On success, zero is returned. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EBUSY`: Volume is mounted.
- `RED_EINVAL`: `pszVolume` is `NULL`; or the driver is uninitialized.
- `RED_EIO`: I/O error formatting the volume.

- [RED_ENOENT](#): pszVolume is not a valid volume path prefix.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 637 of file posix.c.

Referenced by RedFileSystem::format(), and redfs_format().

25.28.2.6 red_fstat()

```
int32_t red_fstat (
    int32_t iFildes,
    REDSTAT * pStat )
```

Get the status of a file or directory.

See the [REDSSTAT](#) type for the details of the information returned.

Parameters

<i>iFildes</i>	An open file descriptor for the file whose information is to be retrieved.
<i>pStat</i>	Pointer to a REDSSTAT buffer to populate.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBADF](#): The iFildes argument is not a valid file descriptor.
- [RED_EINVAL](#): pStat is NULL.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 2035 of file posix.c.

Referenced by RedFileHandle::flen(), RedFileHandle::fstat(), and RedFileHandle::lseek().

25.28.2.7 red_ftrim()

```
int32_t red_ftrim (
    const char * pszVolume,
    uint32_t ulBlockStart,
    uint32_t ulBlockCount )
```

Discard (trim) free blocks within a range of a volume's blocks.

A transaction point will be unconditionally committed prior to discarding any free blocks.

Discards can reduce the garbage collection workload for flash memory storage devices, thereby improving the performance of write operations. However, the cost of discarding at run-time can be high on some devices; and in some cases,

poorly designed devices may even wear out faster if discards are issued continuously. This function is an alternative that allows these discards to happen less frequently, and in larger batches, to provide their benefit while reducing their cost. On Linux, the fstrim ioctl and utility provide similar functionality.

A second application of this function is to execute the first step of wiping the volume of any data that previously resided in blocks that are now free space, for example, deleted files. The second step is for the application to issue a command such as eMMC sanitize. On eMMC, the discards by themselves are not guaranteed to remove the discarded data from the underlying flash memory, but the sanitize command *is* guaranteed to remove any discarded data.

If the range specifies blocks which are beyond the end of the volume, such blocks will be ignored.

Parameters

<i>pszVolume</i>	The path prefix of the volume to fstrim.
<i>ulBlockStart</i>	The start of the range of blocks to discard if free.
<i>ulBlockCount</i>	The length of the range of blocks to discard if free.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EINVAL](#): Volume is not mounted; or *pszVolume* is NULL; or *ulBlockCount* is zero.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_ENOENT](#): *pszVolume* is not a valid volume path prefix.
- [RED_ENOTSUPP](#): Discards are not supported by the volume's block device.
- [RED_EROFS](#): The file system volume is read-only.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 894 of file posix.c.

25.28.2.8 red_fsync()

```
int32_t red_fsync (
    int32_t iFildes )
```

Synchronizes changes to a file.

Commits all changes associated with a file or directory (including file data, directory contents, and metadata) to permanent storage. This function will not return until the operation is complete.

In the current implementation, this function has global effect. All dirty buffers are flushed and a transaction point is committed. Fsyncing one file effectively fsyncs all files.

If fsync automatic transactions have been disabled, this function does nothing and returns success. In the current implementation, this is the only real difference between this function and [red_transact\(\)](#): this function can be configured to do nothing, whereas [red_transact\(\)](#) is unconditional.

Applications written for portability should avoid assuming [red_fsync\(\)](#) effects all files, and use [red_fsync\(\)](#) on each file that needs to be synchronized.

Passing read-only file descriptors to this function is permitted.

Parameters

<i>iFildes</i>	The file descriptor to synchronize.
----------------	-------------------------------------

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBADF](#): The *iFildes* argument is not a valid file descriptor.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 1755 of file posix.c.

Referenced by RedFileHandle::fsync().

25.28.2.9 red_ftruncate()

```
int32_t red_ftruncate (
    int32_t iFildes,
    uint64_t ullSize )
```

Truncate a file to a specified length.

Allows the file size to be increased, decreased, or to remain the same. If the file size is increased, the new area is sparse (will read as zeroes). If the file size is decreased, the data beyond the new end-of-file will return to free space once it is no longer part of the committed state (either immediately or after the next transaction point).

The value of the file offset is not modified by this function.

Unlike POSIX ftruncate, this function can fail when the disk is full if *ullSize* is non-zero. If decreasing the file size, this can be fixed by transacting and trying again: Reliance Edge guarantees that it is possible to perform a truncate of at least one file that decreases the file size after a transaction point. If disk full transactions are enabled, this will happen automatically.

Parameters

<i>iFildes</i>	The file descriptor of the file to truncate.
<i>ullSize</i>	The new size of the file.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBADF](#): The *iFildes* argument is not a valid file descriptor open for writing. This includes the case where the file descriptor is for a directory.
- [RED_EFBIG](#): *ullSize* exceeds the maximum file size.

- [RED_EIO](#): A disk I/O error occurred.
- [RED_ENOSPC](#): Insufficient free space to perform the truncate.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 1973 of file posix.c.

Referenced by RedFileHandle::ftruncate().

25.28.2.10 red_getcwd()

```
char* red_getcwd (
    char * pszBuffer,
    uint32_t ulBufferSize )
```

Get the path of the current working directory (CWD).

The default CWD, if it has never been set since the file system was initialized, is the root directory of volume zero. If the CWD is on a volume that is unmounted, it resets to the root directory of that volume.

Note

Reliance Edge does not have a maximum path length; paths, including the CWD path, can be arbitrarily long. Thus, no buffer is guaranteed to be large enough to store the CWD. If it is important that calls to this function succeed, you need to analyze your application to determine the maximum length of the CWD path. Alternatively, if dynamic memory allocation is used, this function can be called in a loop, with the buffer size increasing if the function fails with a RED_ERANGE error; repeat until the call succeeds.

Parameters

<i>pszBuffer</i>	The buffer to populate with the CWD.
<i>ulBufferSize</i>	The size in bytes of <i>pszBuffer</i> .

Returns

On success, *pszBuffer* is returned. On error, `NULL` is returned and [red_errno](#) is set appropriately.

Errno values

- [RED EINVAL](#): *pszBuffer* is `NULL`; or *ulBufferSize* is zero.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_ERANGE](#): *ulBufferSize* is greater than zero but too small for the CWD path string.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 2409 of file posix.c.

25.28.2.11 red_gettransmask()

```
int32_t red_gettransmask (
    const char * pszVolume,
    uint32_t * pulEventMask )
```

Read the transaction mask.

If the volume is read-only, the returned event mask is always zero.

Parameters

<i>pszVolume</i>	The path prefix of the volume whose transaction mask is being retrieved.
<i>pulEventMask</i>	Populated with a bitwise-OR'd mask of automatic transaction events which represent the current transaction mode for the volume.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EINVAL](#): Volume is not mounted; or *pszVolume* is NULL; or *pulEventMask* is NULL.
- [RED_ENOENT](#): *pszVolume* is not a valid volume path prefix.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 788 of file posix.c.

Referenced by RedFileSystem::get_trans_mask().

25.28.2.12 red_init()

```
int32_t red_init (
    void )
```

Initialize the Reliance Edge file system driver.

Prepares the Reliance Edge file system driver to be used. Must be the first Reliance Edge function to be invoked: no volumes can be mounted or formatted until the driver has been initialized.

If this function is called when the Reliance Edge driver is already initialized, it does nothing and returns success.

This function is not thread safe: attempting to initialize from multiple threads could leave things in a bad state.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EINVAL](#): The volume path prefix configuration is invalid.

Definition at line 181 of file posix.c.

Referenced by RedFileSystem::RedFileSystem(), and redfs_format().

25.28.2.13 red_link()

```
int32_t red_link (
    const char * pszPath,
    const char * pszHardLink )
```

Create a hard link.

This creates an additional name (link) for the file named by `pszPath`. The new name refers to the same file with the same contents. If a name is deleted, but the underlying file has other names, the file continues to exist. The link count (accessible via `red_fstat()`) indicates the number of names that a file has. All of a file's names are on equal footing: there is nothing special about the original name.

If `pszPath` names a directory, the operation will fail.

Parameters

<code>pszPath</code>	The path indicating the inode for the new link.
<code>pszHardLink</code>	The name and location for the new link.

Returns

On success, zero is returned. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EEXIST`: `pszHardLink` resolves to an existing file.
- `RED_EINVAL`: `pszPath` or `pszHardLink` is NULL; or the volume containing the paths is not mounted; or `REDCONF_API_POSIX_CWD` is true and `pszHardLink` ends with dot or dot-dot.
- `RED_EIO`: A disk I/O error occurred.
- `RED_EMLINK`: Creating the link would exceed the maximum link count of the inode named by `pszPath`.
- `RED_ENAMETOOLONG`: The length of a component of either `pszPath` or `pszHardLink` is longer than `REDCONF_NAME_MAX`.
- `RED_ENOENT`: A component of either path prefix does not exist; or the file named by `pszPath` does not exist; or either `pszPath` or `pszHardLink` point to an empty string (and there is no volume with an empty path prefix).
- `RED_ENOSPC`: There is insufficient free space to expand the directory that would contain the link.
- `RED_ENOTDIR`: A component of either path prefix is not a directory.
- `RED_EPERM`: The `pszPath` argument names a directory.
- `RED_EROFS`: The requested link requires writing in a directory on a read-only file system.
- `RED_EUSERS`: Cannot become a file system user: too many users.
- `RED_EXDEV`: `pszPath` and `pszHardLink` are on different file system volumes.

Definition at line 1418 of file `posix.c`.

Referenced by `RedFileSystem::link()`.

25.28.2.14 red_lseek()

```
int64_t red_lseek (
    int32_t iFildes,
    int64_t lloffset,
    REDWHENCE whence )
```

Move the read/write file offset.

The file offset of the `iFildes` file descriptor is set to `lloffset`, relative to some starting position. The available positions are:

- `RED_SEEK_SET` Seek from the start of the file. In other words, `lloffset` becomes the new file offset.
- `RED_SEEK_CUR` Seek from the current file offset. In other words, `lloffset` is added to the current file offset.
- `RED_SEEK_END` Seek from the end-of-file. In other words, the new file offset is the file size plus `lloffset`.

Since `lloffset` is signed (can be negative), it is possible to seek backward with `RED_SEEK_CUR` or `RED_SEEK_END`.

It is permitted to seek beyond the end-of-file; this does not increase the file size (a subsequent `red_write()` call would).

Unlike POSIX `lseek`, this function cannot be used with directory file descriptors.

Parameters

<code>iFildes</code>	The file descriptor whose offset is to be updated.
<code>lloffset</code>	The new file offset, relative to <code>whence</code> .
<code>whence</code>	The location from which <code>lloffset</code> should be applied.

Returns

On success, returns the new file position, measured in bytes from the beginning of the file. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EBADF`: The `iFildes` argument is not an open file descriptor.
- `RED_EINVAL`: `whence` is not a valid `RED_SEEK_` value; or the resulting file offset would be negative or beyond the maximum file size.
- `RED_EIO`: A disk I/O error occurred.
- `RED_EISDIR`: The `iFildes` argument is a file descriptor for a directory.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 1834 of file `posix.c`.

Referenced by `RedFileHandle::lseek()`.

25.28.2.15 red_mkdir()

```
int32_t red_mkdir (
    const char * pszPath )
```

Create a new directory.

Unlike POSIX mkdir, this function has no second argument for the permissions (which Reliance Edge does not use).

Parameters

<code>pszPath</code>	The name and location of the directory to create.
----------------------	---

Returns

On success, zero is returned. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EEXIST`: `pszPath` points to an existing file or directory.
- `RED_EINVAL`: `pszPath` is NULL; or the volume containing the path is not mounted; or the path ends with dot or dot-dot.
- `RED_EIO`: A disk I/O error occurred.
- `RED_ENAMETOOLONG`: The length of a component of `pszPath` is longer than `REDCONF_NAME_MAX`.
- `RED_ENOENT`: A component of the path prefix does not name an existing directory; or the `pszPath` argument points to an empty string (and there is no volume with an empty path prefix).
- `RED_ENOSPC`: The file system does not have enough space for the new directory or to extend the parent directory of the new directory.
- `RED_ENOTDIR`: A component of the path prefix is not a directory.
- `RED_EROFS`: The parent directory resides on a read-only file system.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 1129 of file posix.c.

Referenced by `RedFileSystem::mkdir()`.

25.28.2.16 red_mount()

```
int32_t red_mount (
    const char * pszVolume )
```

Mount a file system volume.

Prepares the file system volume to be accessed. Mount will fail if the volume has never been formatted, or if the on-disk format is inconsistent with the compile-time configuration.

An error is returned if the volume is already mounted.

Parameters

<i>pszVolume</i>	A path prefix identifying the volume to mount.
------------------	--

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBUSY](#): Volume is already mounted.
- [RED_EINVAL](#): *pszVolume* is NULL; or the driver is uninitialized.
- [RED_EIO](#): Volume not formatted, improperly formatted, or corrupt.
- [RED_ENOENT](#): *pszVolume* is not a valid volume path prefix.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 378 of file posix.c.

Referenced by [RedFileSystem::mount\(\)](#).

25.28.2.17 red_mount2()

```
int32_t red_mount2 (
    const char * pszVolume,
    uint32_t ulFlags )
```

Mount a file system volume with flags.

Prepares the file system volume to be accessed. Mount will fail if the volume has never been formatted, or if the on-disk format is inconsistent with the compile-time configuration.

An error is returned if the volume is already mounted.

The following mount flags are available:

- [RED_MOUNT_READONLY](#): If specified, the volume will be mounted read-only. All write operations will fail, setting [red_errno](#) to [RED_EROFS](#).
- [RED_MOUNT_DISCARD](#): If specified, and if the underlying block device supports discards, discards will be issued for blocks that become free. If the underlying block device does *not* support discards, then this flag has no effect.

The [RED_MOUNT_DEFAULT](#) macro can be used to mount with the default mount flags, which is equivalent to mounting with [red_mount\(\)](#).

Parameters

<i>pszVolume</i>	A path prefix identifying the volume to mount.
<i>ulFlags</i>	A bitwise-OR'd mask of mount flags.

Returns

On success, zero is returned. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EBUSY`: Volume is already mounted.
- `RED_EINVAL`: `pszVolume` is NULL; or the driver is uninitialized; or `ulFlags` includes invalid mount flags.
- `RED_EIO`: Volume not formatted, improperly formatted, or corrupt.
- `RED_ENOENT`: `pszVolume` is not a valid volume path prefix.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 419 of file posix.c.

Referenced by `red_mount()`.

25.28.2.18 red_open()

```
int32_t red_open (
    const char * pszPath,
    uint32_t ulOpenMode )
```

Open a file or directory.

Exactly one file access mode must be specified:

- `RED_O_RDONLY`: Open for reading only.
- `RED_O_WRONLY`: Open for writing only.
- `RED_O_RDWR`: Open for reading and writing.

Directories can only be opened with `RED_O_RDONLY`.

The following flags may also be used:

- `RED_O_APPEND`: Set the file offset to the end-of-file prior to each write.
- `RED_O_CREAT`: Create the named file if it does not exist.
- `RED_O_EXCL`: In combination with `RED_O_CREAT`, return an error if the path already exists.
- `RED_O_TRUNC`: Truncate the opened file to size zero. Only supported when `REDCONF_API_POSIX_FTRUNCATE` is true.

`RED_O_CREAT`, `RED_O_EXCL`, and `RED_O_TRUNC` are invalid with `RED_O_RDONLY`. `RED_O_EXCL` is invalid without `RED_O_CREAT`.

If the volume is read-only, `RED_O_RDONLY` is the only valid open flag; use of any other flag will result in an error.

If `RED_O_TRUNC` frees data which is in the committed state, it will not return to free space until after a transaction point.

The returned file descriptor must later be closed with `red_close()`.

Unlike POSIX open, there is no optional third argument for the permissions (which Reliance Edge does not use) and other open flags (like `O_SYNC`) are not supported.

Parameters

<i>pszPath</i>	The path to the file or directory.
<i>ulOpenMode</i>	The open flags (mask of <code>RED_O_</code> values).

Returns

On success, a nonnegative file descriptor is returned. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EEXIST`: Using `RED_O_CREAT` and `RED_O_EXCL`, and the indicated path already exists.
- `RED_EINVAL`: *ulOpenMode* is invalid; or *pszPath* is NULL; or the volume containing the path is not mounted; or `RED_O_CREAT` is included in *ulOpenMode*, and the path ends with dot or dot-dot.
- `RED_EIO`: A disk I/O error occurred.
- `RED_EISDIR`: The path names a directory and *ulOpenMode* includes `RED_O_WRONLY` or `RED_O_RDWR`.
- `RED_EMFILE`: There are no available file descriptors.
- `RED_ENAMETOOLONG`: The length of a component of *pszPath* is longer than `REDCONF_NAME_MAX`.
- `RED_ENFILE`: Attempting to create a file but the file system has used all available inode slots.
- `RED_ENOENT`: `RED_O_CREAT` is not set and the named file does not exist; or `RED_O_CREAT` is set and the parent directory does not exist; or the volume does not exist; or the *pszPath* argument points to an empty string (and there is no volume with an empty path prefix).
- `RED_ENOSPC`: The file does not exist and `RED_O_CREAT` was specified, but there is insufficient free space to expand the directory or to create the new file.
- `RED_ENOTDIR`: A component of the prefix in *pszPath* does not name a directory.
- `RED_EROFS`: The path resides on a read-only file system and a write operation was requested.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 987 of file posix.c.

Referenced by `RedFileSystem::open()`.

25.28.2.19 red_opendir()

```
REDDIR* red_opendir (
    const char * pszPath )
```

Open a directory stream for reading.

Parameters

<i>pszPath</i>	The path of the directory to open.
----------------	------------------------------------

Returns

On success, returns a pointer to a [REDDIR](#) object that can be used with [red_readdir\(\)](#) and [red_closedir\(\)](#). On error, returns `NULL` and [red_errno](#) is set appropriately.

Errno values

- [RED_EINVAL](#): `pszPath` is `NULL`; or the volume containing the path is not mounted.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_ENOENT](#): A component of `pszPath` does not exist; or the `pszPath` argument points to an empty string (and there is no volume with an empty path prefix).
- [RED_ENOTDIR](#): A component of `pszPath` is not a directory.
- [RED_EMFILE](#): There are no available file descriptors.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 2087 of file `posix.c`.

Referenced by `RedFileSystem::opendir()`.

25.28.2.20 red_read()

```
int32_t red_read (
    int32_t iFildes,
    void * pBuffer,
    uint32_t ulLength )
```

Read from an open file.

The read takes place at the file offset associated with `iFildes` and advances the file offset by the number of bytes actually read.

Data which has not yet been written, but which is before the end-of-file (sparse data), will read as zeroes. A short read – where the number of bytes read is less than requested – indicates that the requested read was partially or, if zero bytes were read, entirely beyond the end-of-file.

Parameters

<code>iFildes</code>	The file descriptor from which to read.
<code>pBuffer</code>	The buffer to populate with data read. Must be at least <code>ulLength</code> bytes in size.
<code>ulLength</code>	Number of bytes to attempt to read.

Returns

On success, returns a nonnegative value indicating the number of bytes actually read. On error, `-1` is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBADF](#): The `iFildes` argument is not a valid file descriptor open for reading.

- **RED_EINVAL**: pBuffer is NULL; or ulLength exceeds INT32_MAX and cannot be returned properly.
- **RED_EIO**: A disk I/O error occurred.
- **RED_EISDIR**: The iFildes is a file descriptor for a directory.
- **RED_EUSERS**: Cannot become a file system user: too many users.

Definition at line 1529 of file posix.c.

Referenced by RedFileHandle::read().

25.28.2.21 red_readdir()

```
REDDIRENT* red_readdir (
    REDDIR * pDirStream )
```

Read from a directory stream.

The **REDDIRENT** pointer returned by this function will be overwritten by subsequent calls on the same **pDir**. Calls with other **REDDIR** objects will *not* modify the returned **REDDIRENT**.

If files are added to the directory after it is opened, the new files may or may not be returned by this function. If files are deleted, the deleted files will not be returned.

This function (like its POSIX equivalent) returns **NULL** in two cases: on error and when the end of the directory is reached. To distinguish between these two cases, the application should set **red_errno** to zero before calling this function, and if **NULL** is returned, check if **red_errno** is still zero. If it is, the end of the directory was reached; otherwise, there was an error.

Parameters

pDirStream	The directory stream to read from.
-------------------	------------------------------------

Returns

On success, returns a pointer to a **REDDIRENT** object which is populated with directory entry information read from the directory. On error, returns **NULL** and **red_errno** is set appropriately. If at the end of the directory, returns **NULL** but **red_errno** is not modified.

Errno values

- **RED_EBADF**: **pDirStream** is not an open directory stream.
- **RED_EIO**: A disk I/O error occurred.
- **RED_EUSERS**: Cannot become a file system user: too many users.

Definition at line 2150 of file posix.c.

Referenced by RedDirHandle::readdir().

25.28.2.22 red_rename()

```
int32_t red_rename (
    const char * pszOldPath,
    const char * pszNewPath )
```

Rename a file or directory.

Both paths must reside on the same file system volume. Attempting to use this API to move a file to a different volume will result in an error.

If `pszNewPath` names an existing file or directory, the behavior depends on the configuration. If `REDCONF_RENAME_ATOMIC` is false, and if the destination name exists, this function always fails and sets `red_errno` to `RED_EEXIST`. This behavior is contrary to POSIX.

If `REDCONF_RENAME_ATOMIC` is true, and if the new name exists, then in one atomic operation, `pszNewPath` is unlinked and `pszOldPath` is renamed to `pszNewPath`. Both `pszNewPath` and `pszOldPath` must be of the same type (both files or both directories). As with `red_unlink()`, if `pszNewPath` is a directory, it must be empty. The major exception to this behavior is that if both `pszOldPath` and `pszNewPath` are links to the same inode, then the rename does nothing and both names continue to exist. Unlike POSIX rename, if `pszNewPath` points to an inode with a link count of one and open handles (file descriptors or directory streams), the rename will fail with `RED_EBUSY`.

If the rename deletes the old destination, it may free data in the committed state, which will not return to free space until after a transaction point. Similarly, if the deleted inode was part of the committed state, the inode slot will not be available until after a transaction point.

Parameters

<code>pszOldPath</code>	The path of the file or directory to rename.
<code>pszNewPath</code>	The new name and location after the rename.

Returns

On success, zero is returned. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- **RED_EBUSY**: `pszOldPath` or `pszNewPath` names the root directory; or `REDCONF_RENAME_ATOMIC` is true and either a) `pszNewPath` points to an inode with open handles and a link count of one or b) `REDCONF_API_POSIX_CWD` is true and the `pszNewPath` points to an inode which is the CWD of at least one task.
- **RED_EEXIST**: `REDCONF_RENAME_ATOMIC` is false and `pszNewPath` exists.
- **RED_EINVAL**: `pszOldPath` is NULL; or `pszNewPath` is NULL; or the volume containing the path is not mounted; or `REDCONF_API_POSIX_CWD` is true and either path ends with dot or dot-dot; or `REDCONF_API_POSIX_CWD` is false and `pszNewPath` ends with dot or dot-dot; or the rename is cyclic.
- **RED_EIO**: A disk I/O error occurred.
- **RED_EISDIR**: The `pszNewPath` argument names a directory and the `pszOldPath` argument names a non-directory.
- **RED_ENAMETOOLONG**: The length of a component of either `pszOldPath` or `pszNewPath` is longer than `REDCONF_NAME_MAX`.
- **RED_ENOENT**: The link named by `pszOldPath` does not name an existing entry; or either `pszOldPath` or `pszNewPath` point to an empty string (and there is no volume with an empty path prefix).

- **RED_ENOTDIR**: A component of either path prefix is not a directory; or `pszOldPath` names a directory and `pszNewPath` names a file.
- **RED_ENOTEMPTY**: The path named by `pszNewPath` is a directory which is not empty.
- **RED_ENOSPC**: The file system does not have enough space to extend the directory that would contain `pszNewPath`.
- **RED_EROFS**: The directory to be removed resides on a read-only file system.
- **RED_EUSERS**: Cannot become a file system user: too many users.
- **RED_EXDEV**: `pszOldPath` and `pszNewPath` are on different file system volumes.

Definition at line 1298 of file posix.c.

Referenced by `RedFileSystem::rename()`.

25.28.2.23 red_rewinddir()

```
void red_rewinddir (
    REDDIR * pDirStream )
```

Rewind a directory stream to read it from the beginning.

Similar to closing the directory object and opening it again, but without the need for the path.

Since this function (like its POSIX equivalent) cannot return an error, it takes no action in error conditions, such as when `pDirStream` is invalid.

Parameters

<code>pDirStream</code>	The directory stream to rewind.
-------------------------	---------------------------------

Definition at line 2234 of file posix.c.

Referenced by `RedDirHandle::rewinddir()`.

25.28.2.24 red_rmdir()

```
int32_t red_rmdir (
    const char * pszPath )
```

Delete a directory.

The given directory name is deleted and the corresponding directory inode will be deleted.

Unlike POSIX `rmdir`, deleting a directory with open handles (file descriptors or directory streams) will fail with an **RED_EBUSY** error.

If the path names a directory which is not empty, the deletion will fail. If the path names the root directory of a file system volume, the deletion will fail.

If the path names a regular file, the deletion will fail. This provides type checking and may be useful in cases where an application knows the path to be deleted should name a directory.

If the deletion frees data in the committed state, it will not return to free space until after a transaction point.

Unlike POSIX rmdir, this function can fail when the disk is full. To fix this, transact and try again: Reliance Edge guarantees that it is possible to delete at least one file or directory after a transaction point. If disk full automatic transactions are enabled, this will happen automatically.

Parameters

<i>pszPath</i>	The path of the directory to delete.
----------------	--------------------------------------

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBUSY](#): *pszPath* names the root directory; or *pszPath* points to a directory with open handles; or [REDCONF_API_POSIX_CWD](#) is true and *pszPath* points to the CWD of a task.
- [RED_EINVAL](#): *pszPath* is NULL; or the volume containing the path is not mounted; or [REDCONF_API_POSIX_CWD](#) is true and the path ends with dot or dot-dot.
- [RED_EIO](#): A disk I/O error occurred.
- [RED_ENAMETOOLONG](#): The length of a component of *pszPath* is longer than [REDCONF_NAME_MAX](#).
- [RED_ENOENT](#): The path does not name an existing directory; or the *pszPath* argument points to an empty string (and there is no volume with an empty path prefix).
- [RED_ENOTDIR](#): A component of the path is not a directory.
- [RED_ENOTEMPTY](#): The path names a directory which is not empty.
- [RED_ENOSPC](#): The file system does not have enough space to modify the parent directory to perform the deletion.
- [RED_EROFS](#): The directory to be removed resides on a read-only file system.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 1214 of file posix.c.

25.28.2.25 red_settransmask()

```
int32_t red_settransmask (
    const char * pszVolume,
    uint32_t ulEventMask )
```

Update the transaction mask.

The following events are available:

- [RED_TRANSACT_SYNC](#)
- [RED_TRANSACT_UMOUNT](#)

- [RED_TRANSACT_CREAT](#)
- [RED_TRANSACT_UNLINK](#)
- [RED_TRANSACT_MKDIR](#)
- [RED_TRANSACT_RENAME](#)
- [RED_TRANSACT_LINK](#)
- [RED_TRANSACT_CLOSE](#)
- [RED_TRANSACT_WRITE](#)
- [RED_TRANSACT_FSYNC](#)
- [RED_TRANSACT_TRUNCATE](#)
- [RED_TRANSACT_VOLFULL](#)

The [RED_TRANSACT_MANUAL](#) macro (by itself) may be used to disable all automatic transaction events. The [RED_TRANSACT_MASK](#) macro is a bitmask of all transaction flags, excluding those representing excluded functionality.

Attempting to enable events for excluded functionality will result in an error.

Parameters

<i>pszVolume</i>	The path prefix of the volume whose transaction mask is being changed.
<i>ulEventMask</i>	A bitwise-OR'd mask of automatic transaction events to be set as the current transaction mode.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EINVAL](#): Volume is not mounted; or *pszVolume* is NULL; or *ulEventMask* contains invalid bits.
- [RED_ENOENT](#): *pszVolume* is not a valid volume path prefix.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 745 of file posix.c.

Referenced by [RedFileSystem::set_trans_mask\(\)](#).

25.28.2.26 red_statvfs()

```
int32_t red_statvfs (
    const char * pszVolume,
    REDSTATFS * pStatvfs )
```

Query file system status information.

pszVolume should name a valid volume prefix or a valid root directory; this differs from POSIX statvfs, where any existing file or directory is a valid path.

Parameters

<i>pszVolume</i>	The path prefix of the volume to query.
<i>pStatvfs</i>	The buffer to populate with volume information.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EINVAL](#): Volume is not mounted; or *pszVolume* is NULL; or *pStatvfs* is NULL.
- [RED_ENOENT](#): *pszVolume* is not a valid volume path prefix.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 829 of file posix.c.

Referenced by RedFileSystem::statvfs().

25.28.2.27 red_sync()

```
int32_t red_sync (
    void )
```

Commits file system updates.

Commits all changes on all file system volumes to permanent storage. This function will not return until the operation is complete.

If sync automatic transactions have been disabled for one or more volumes, this function does not commit changes to those volumes, but will still commit changes to any volumes for which automatic transactions are enabled.

If sync automatic transactions have been disabled on all volumes, this function does nothing and returns success.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EIO](#): I/O error during the transaction point.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 312 of file posix.c.

25.28.2.28 red_transact()

```
int32_t red_transact (
    const char * pszVolume )
```

Commit a transaction point.

Reliance Edge is a transactional file system. All modifications, of both metadata and filedata, are initially working state. A transaction point is a process whereby the working state atomically becomes the committed state, replacing the previous committed state. Whenever Reliance Edge is mounted, including after power loss, the state of the file system after mount is the most recent committed state. Nothing from the committed state is ever missing, and nothing from the working state is ever included.

Parameters

<i>pszVolume</i>	A path prefix identifying the volume to transact.
------------------	---

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EINVAL](#): Volume is not mounted; or *pszVolume* is NULL.
- [RED_EIO](#): I/O error during the transaction point.
- [RED_ENOENT](#): *pszVolume* is not a valid volume path prefix.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 682 of file posix.c.

Referenced by RedFileSystem::transact().

25.28.2.29 red_umount()

```
int32_t red_umount (
    const char * pszVolume )
```

Unmount a file system volume.

This function discards the in-memory state for the file system and marks it as unmounted. Subsequent attempts to access the volume will fail until the volume is mounted again.

If unmount automatic transaction points are enabled, this function will commit a transaction point prior to unmounting. If unmount automatic transaction points are disabled, this function will unmount without transacting, effectively discarding the working state.

Before unmounting, this function will wait for any active file system thread to complete by acquiring the FS mutex. The volume will be marked as unmounted before the FS mutex is released, so subsequent FS threads will possibly block and then see an error when attempting to access a volume which is unmounting or unmounted. If the volume has open handles, the unmount will fail.

An error is returned if the volume is already unmounted.

Parameters

<i>pszVolume</i>	A path prefix identifying the volume to unmount.
------------------	--

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBUSY](#): There are still open handles for this file system volume.
- [RED_EINVAL](#): *pszVolume* is NULL; or the driver is uninitialized; or the volume is already unmounted.
- [RED_EIO](#): I/O error during unmount automatic transaction point.
- [RED_ENOENT](#): *pszVolume* is not a valid volume path prefix.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 506 of file posix.c.

Referenced by [RedFileSystem:::unmount\(\)](#).

25.28.2.30 red_umount2()

```
int32_t red_umount2 (
    const char * pszVolume,
    uint32_t ulFlags )
```

Unmount a file system volume with flags.

This function is the same as [red_umount\(\)](#), except that it accepts a flags parameter which can change the unmount behavior.

The following unmount flags are available:

- [RED_UNMOUNT_FORCE](#): If specified, if the volume has open handles, the handles will be closed. Without this flag, the behavior is to return an [RED_EBUSY](#) error if the volume has open handles.

The [RED_UNMOUNT_DEFAULT](#) macro can be used to unmount with the default unmount flags, which is equivalent to unmounting with [red_umount\(\)](#).

Parameters

<i>pszVolume</i>	A path prefix identifying the volume to unmount.
<i>ulFlags</i>	A bitwise-OR'd mask of unmount flags.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBUSY](#): There are still open handles for this file system volume and the [RED_UNMOUNT_FORCE](#) flag was not specified.
- [RED_EINVAL](#): *pszVolume* is NULL; or *ulFlags* includes invalid unmount flags; or the driver is uninitialized; or the volume is already unmounted.

- [RED_EIO](#): I/O error during unmount automatic transaction point.
- [RED_ENOENT](#): pszVolume is not a valid volume path prefix.
- [RED_EUSERS](#): Cannot become a file system user: too many users.

Definition at line 543 of file posix.c.

Referenced by [red_umount\(\)](#).

25.28.2.31 red_uninit()

```
int32_t red_uninit (
    void )
```

Uninitialize the Reliance Edge file system driver.

Tears down the Reliance Edge file system driver. Cannot be used until all Reliance Edge volumes are unmounted. A subsequent call to [red_init\(\)](#) will initialize the driver again.

If this function is called when the Reliance Edge driver is already uninitialized, it does nothing and returns success.

This function is not thread safe: attempting to uninitialized from multiple threads could leave things in a bad state.

Returns

On success, zero is returned. On error, -1 is returned and [red_errno](#) is set appropriately.

Errno values

- [RED_EBUSY](#): At least one volume is still mounted.

Definition at line 230 of file posix.c.

Referenced by [redfs_format\(\)](#).

25.28.2.32 red_unlink()

```
int32_t red_unlink (
    const char * pszPath )
```

Delete a file or directory.

The given name is deleted and the link count of the corresponding inode is decremented. If the link count falls to zero (no remaining hard links), the inode will be deleted.

Unlike POSIX unlink, deleting a file or directory with open handles (file descriptors or directory streams) will fail with an [RED_EBUSY](#) error. This only applies when deleting an inode with a link count of one; if a file has multiple names (hard links), all but the last name may be deleted even if the file is open.

If the path names a directory which is not empty, the unlink will fail.

If the deletion frees data in the committed state, it will not return to free space until after a transaction point.

Unlike POSIX unlink, this function can fail when the disk is full. To fix this, transact and try again: Reliance Edge guarantees that it is possible to delete at least one file or directory after a transaction point. If disk full automatic transactions are enabled, this will happen automatically.

Parameters

<code>pszPath</code>	The path of the file or directory to delete.
----------------------	--

Returns

On success, zero is returned. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EBUSY`: `pszPath` names the root directory; or `pszPath` points to an inode with open handles and a link count of one; or `REDCONF_API_POSIX_CWD` is true and `pszPath` points to the CWD of a task.
- `RED_EINVAL`: `pszPath` is NULL; or the volume containing the path is not mounted; or `REDCONF_API_POSIX_CWD` is true and the path ends with dot or dot-dot.
- `RED_EIO`: A disk I/O error occurred.
- `RED_ENAMETOOLONG`: The length of a component of `pszPath` is longer than `REDCONF_NAME_MAX`.
- `RED_ENOENT`: The path does not name an existing file; or the `pszPath` argument points to an empty string (and there is no volume with an empty path prefix).
- `RED_ENOTDIR`: A component of the path prefix is not a directory.
- `RED_ENOTEMPTY`: The path names a directory which is not empty.
- `RED_ENOSPC`: The file system does not have enough space to modify the parent directory to perform the deletion.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 1084 of file posix.c.

Referenced by `RedFileSystem::remove()`.

25.28.2.33 red_write()

```
int32_t red_write (
    int32_t iFildes,
    const void * pBuffer,
    uint32_t ulLength )
```

Write to an open file.

The write takes place at the file offset associated with `iFildes` and advances the file offset by the number of bytes actually written. Alternatively, if `iFildes` was opened with `RED_O_APPEND`, the file offset is set to the end-of-file before the write begins, and likewise advances by the number of bytes actually written.

A short write – where the number of bytes written is less than requested – indicates either that the file system ran out of space but was still able to write some of the request; or that the request would have caused the file to exceed the maximum file size, but some of the data could be written prior to the file size limit.

If an error is returned (-1), either none of the data was written or a critical error occurred (like an I/O error) and the file system volume will be read-only.

Parameters

<i>iFildes</i>	The file descriptor to write to.
<i>pBuffer</i>	The buffer containing the data to be written. Must be at least <i>ulLength</i> bytes in size.
<i>ulLength</i>	Number of bytes to attempt to write.

Returns

On success, returns a nonnegative value indicating the number of bytes actually written. On error, -1 is returned and `red_errno` is set appropriately.

Errno values

- `RED_EBADF`: The *iFildes* argument is not a valid file descriptor open for writing. This includes the case where the file descriptor is for a directory.
- `RED_EFBIG`: No data can be written to the current file offset since the resulting file size would exceed the maximum file size.
- `RED_EINVAL`: *pBuffer* is NULL; or *ulLength* exceeds INT32_MAX and cannot be returned properly.
- `RED_EIO`: A disk I/O error occurred.
- `RED_ENOSPC`: No data can be written because there is insufficient free space.
- `RED_EUSERS`: Cannot become a file system user: too many users.

Definition at line 1635 of file posix.c.

Referenced by `RedFileHandle::write()`.

25.29 projects/doxygen/redconf.h File Reference

Macros

- `#define REDCONF_ENDIAN_BIG`
Whether the target system is big endian.
- `#define REDCONF_ALIGNMENT_SIZE`
The native memory alignment size of the target system.
- `#define REDCONF_OUTPUT`
Whether to enable text output.
- `#define REDCONF_ASSERTS`
Whether to enable assertion processing.
- `#define REDCONF_CRC_ALGORITHM`
Which CRC algorithm to use.
- `#define REDCONF_VOLUME_COUNT`
The number of volumes implemented by the driver.
- `#define REDCONF_TASK_COUNT`
The number of file system tasks supported by the driver.
- `#define REDCONF_HANDLE_COUNT`
Number of file system handles available to the POSIX-like API.

- #define REDCONF_BLOCK_SIZE
The logical block size.
- #define REDCONF_NAME_MAX
Maximum length for a file or directory name.
- #define REDCONF_PATH_SEPARATOR
Path separator character.
- #define REDCONF_DIRECT_POINTERS
The number of direct pointers in an inode.
- #define REDCONF_INDIRECT_POINTERS
The number of indirect pointers in an inode.
- #define REDCONF_INODE_BLOCKS
Whether to track the number of blocks allocated to each inode.
- #define REDCONF_INODE_TIMESTAMPS
Whether to store POSIX timestamps (ctime, mtime, atime) for each inode.
- #define REDCONF_ATIME
Whether to keep the atime (time of last access) inode timestamp up to date.
- #define REDCONF_BUFFER_COUNT
The number of block buffers allocated for Reliance Edge.
- #define REDCONF_TRANSACT_DEFAULT
Default automatic transaction mask for all volumes.
- #define REDCONF_RENAME_ATOMIC
Whether to support atomic rename functionality in [red_rename\(\)](#).
- #define REDCONF_IMAP_EXTERNAL
Whether to include support for the external imap.
- #define REDCONF_IMAP_INLINE
Whether to include support for the inline imap.
- #define REDCONF_READ_ONLY
Whether to exclude write operations.
- #define REDCONF_API_POSIX
Whether to include the POSIX-like API.
- #define REDCONF_API_POSIX_FORMAT
Whether to include [red_format\(\)](#).
- #define REDCONF_API_POSIX_UNLINK
Whether to include [red_unlink\(\)](#).
- #define REDCONF_API_POSIX_MKDIR
Whether to include [red_mkdir\(\)](#).
- #define REDCONF_API_POSIX_RMDIR
Whether to include [red_rmdir\(\)](#).
- #define REDCONF_API_POSIX_RENAME
Whether to include [red_rename\(\)](#).
- #define REDCONF_API_POSIX_LINK
Whether to include [red_link\(\)](#).
- #define REDCONF_API_POSIX_FTRUNCATE
Whether to include [red_ftruncate\(\)](#) and support for [RED_O_TRUNC](#).
- #define REDCONF_API_POSIX_READDIR
Whether to include [red_opendir\(\)](#), [red_readdir\(\)](#), [red_rewinddir\(\)](#), and [red_closedir\(\)](#).
- #define REDCONF_API_POSIX_CWD

- Whether to include current working directory handling and the related APIs, `red_getcwd()` and `red_chdir()`.
- `#define REDCONF_API_POSIX_FSTRIM`
Whether to include `red_ftrim()`.
- `#define REDCONF_API_FSE`
Whether to include the File System Essentials API.
- `#define REDCONF_API_FSE_FORMAT`
Whether to include `RelFseFormat()`.
- `#define REDCONF_API_FSE_TRUNCATE`
Whether to include `RelFseTruncate()`.
- `#define REDCONF_API_FSE_TRANSMASKSET`
Whether to include `RelFseTransMaskSet()`.
- `#define REDCONF_API_FSE_TRANSMASKGET`
Whether to include `RelFseTransMaskGet()`.
- `#define REDCONF_IMAGE_BUILDER`
Whether to enable code needed by the image builder.
- `#define REDCONF_CHECKER`
Whether to enable the file system integrity checker.
- `#define REDCONF_DISCARDS`
Whether the file system driver supports discards.

25.29.1 Macro Definition Documentation

25.29.1.1 REDCONF_ALIGNMENT_SIZE

```
#define REDCONF_ALIGNMENT_SIZE
```

The native memory alignment size of the target system.

Must be a power of two. If a pointer has an address which is evenly divisible by this value, it is assumed the pointer can safely be read as an array of `uint32_t`.

Definition at line 17 of file redconf.h.

25.29.1.2 REDCONF_API_FSE

```
#define REDCONF_API_FSE
```

Whether to include the File System Essentials API.

If set to false, this overrides all other exclusion macros for the FSE API and excludes the entire API set. If set to true, `REDCONF_API_POSIX` must be false: only one API can be used at a time.

Definition at line 281 of file redconf.h.

25.29.1.3 REDCONF_API_POSIX

```
#define REDCONF_API_POSIX
```

Whether to include the POSIX-like API.

If set to false, this overrides all other exclusion macros for the POSIX-like API and excludes the entire API set. If set to true, REDCONF_API_FSE must be false: only one API can be used at a time.

Definition at line 226 of file redconf.h.

25.29.1.4 REDCONF_API_POSIX_RENAME

```
#define REDCONF_API_POSIX_RENAME
```

Whether to include [red_rename\(\)](#).

Excluding [red_rename\(\)](#) is recommended unless it is truly required, since enabling rename considerably increases the minimum number of block buffers (and thus the minimum amount of RAM) needed by Reliance Edge.

Definition at line 250 of file redconf.h.

25.29.1.5 REDCONF_ASSERTS

```
#define REDCONF_ASSERTS
```

Whether to enable assertion processing.

If enabled, Reliance Edge will include assertions which invoke the user-implemented [RedOsAssertFail\(\)](#) function when an assertion fails. If disabled, asserts reduced to no-ops which generate no code.

Definition at line 29 of file redconf.h.

25.29.1.6 REDCONF_ATIME

```
#define REDCONF_ATIME
```

Whether to keep the atime (time of last access) inode timestamp up to date.

If true, atime is updated by every file read and readdir operation; this turns read operations into write operations, reducing performance.

Definition at line 127 of file redconf.h.

25.29.1.7 REDCONF_BLOCK_SIZE

```
#define REDCONF_BLOCK_SIZE
```

The logical block size.

This is the unit of allocation for file data and metadata. Must be a power of two value between 128 and 65,536, and must not be smaller than the sector size of any volume.

Definition at line 73 of file redconf.h.

25.29.1.8 REDCONF_BUFFER_COUNT

```
#define REDCONF_BUFFER_COUNT
```

The number of block buffers allocated for Reliance Edge.

Higher values may increase performance in some use cases. The minimum value is variable, anywhere from 2 to 12, depending on the other configuration values.

Definition at line 135 of file redconf.h.

25.29.1.9 REDCONF_CRC_ALGORITHM

```
#define REDCONF_CRC_ALGORITHM
```

Which CRC algorithm to use.

The options are:

- **CRC_BITWISE**. A very slow, very small implementation that calculates the CRC one bit at a time.
- **CRC_SARWATE**. A fast, mid-sized implementation that calculates the CRC one byte at a time. Requires a 1 KB table in the data area.
- **CRC_SLICEBY8**. A very fast, large implementation that calculates the CRC eight bytes at a time. Requires an 8 KB table in the data area.

Definition at line 41 of file redconf.h.

25.29.1.10 REDCONF_DISCARDS

```
#define REDCONF_DISCARDS
```

Whether the file system driver supports discards.

If set to true, each volume may configure whether or not a discard (trim) interface is supported.

Discard support is only available in the commercial version of Reliance Edge. Commercial licensing options are listed [here](#).

Definition at line 319 of file redconf.h.

25.29.1.11 REDCONF_HANDLE_COUNT

```
#define REDCONF_HANDLE_COUNT
```

Number of file system handles available to the POSIX-like API.

With the POSIX-like API, a file descriptor (see [red_open\(\)](#)) or directory stream (see [red_opendir\(\)](#)) is a handle. Each allocated handle is available for either purpose (file descriptor or directory stream), but not at the same time. The total number of handles that will be available for all volumes is determined here.

Definition at line 65 of file redconf.h.

25.29.1.12 REDCONF_IMAP_EXTERNAL

```
#define REDCONF_IMAP_EXTERNAL
```

Whether to include support for the external imap.

If the external imap is required by one or more volumes (based on the size of the volume and the block size), then it must be included. Otherwise, it must be excluded to avoid dead code. Generally, this must be enabled if unless all the volumes are relatively small.

Definition at line 175 of file redconf.h.

25.29.1.13 REDCONF_IMAP_INLINE

```
#define REDCONF_IMAP_INLINE
```

Whether to include support for the inline imap.

If the inline imap is required by one or more volumes (based on the size of the volume and the block size), then it must be included. Otherwise, it must be excluded to avoid dead code. Generally, this must be enabled if at least one of the volumes is relatively small.

Definition at line 184 of file redconf.h.

25.29.1.14 REDCONF_INODE_BLOCKS

```
#define REDCONF_INODE_BLOCKS
```

Whether to track the number of blocks allocated to each inode.

This determines whether [REDSSTAT::st_blocks](#) will be available. Disabling this field removes a small amount of code and frees up a small amount of space in the inode block.

Definition at line 110 of file redconf.h.

25.29.1.15 REDCONF_INODE_TIMESTAMPS

```
#define REDCONF_INODE_TIMESTAMPS
```

Whether to store POSIX timestamps (ctime, mtime, atime) for each inode.

This determines whether [REDSSTAT::st_ctime](#), [REDSSTAT::st_mtime](#), and [REDSSTAT::st_atime](#) will be available. Disabling this field removes a small amount of code and frees up space in the inode block.

Definition at line 119 of file redconf.h.

25.29.1.16 REDCONF_NAME_MAX

```
#define REDCONF_NAME_MAX
```

Maximum length for a file or directory name.

This is equivalent to the NAME_MAX macro in POSIX. For efficient, compact directories, this should be kept as small as possible. Directory entries have a fixed size, so all names consume the same amount of directory space as the maximum-length name. Directory entry sizes must be a multiple of four; if the maximum name length is not divisible by four, directory entries will be padded with unused bytes. To avoid this padding, use a name length which is divisible by four.

Definition at line 85 of file redconf.h.

25.29.1.17 REDCONF_PATH_SEPARATOR

```
#define REDCONF_PATH_SEPARATOR
```

Path separator character.

This is the character which will separate names in a path. At this time, having multiple path separator characters is not supported.

Usually '/' (for Unix-style paths) or '\' (for DOS-style paths).

Definition at line 94 of file redconf.h.

Referenced by RedFileSystem::link(), RedFileSystem::mkdir(), RedFileSystem::open(), RedFileSystem::opendir(), RedFileSystem::remove(), and RedFileSystem::rename().

25.29.1.18 REDCONF_READ_ONLY

```
#define REDCONF_READ_ONLY
```

Whether to exclude write operations.

This configuration allows the Reliance Edge driver to be configured to support read operations only, excluding support for all write operations. If set to true, it overrides all conflicting macros: for example, if this macro is true, and REDCONF_API_POSIX_UNLINK is true, the former takes precedence and [red_unlink\(\)](#) is excluded.

The following POSIX-like and FSE APIs are completely disabled when this configuration option is true:

- [red_format\(\)](#)
- [red_transact\(\)](#)
- [red_settransmask\(\)](#)
- [red_unlink\(\)](#)
- [red_mkdir\(\)](#)
- [red_rmdir\(\)](#)
- [red_rename\(\)](#)
- [red_link\(\)](#)

- [red_write\(\)](#)
- [red_fsync\(\)](#)
- [red_ftruncate\(\)](#)
- [RedFseFormat\(\)](#)
- [RedFseWrite\(\)](#)
- [RedFseTruncate\(\)](#)
- [RedFseTransMaskSet\(\)](#)
- [RedFseTransact\(\)](#)

In addition, the behavior of some APIs is modified on read-only volumes: these differences are described in the API documentation.

Definition at line 217 of file redconf.h.

25.29.1.19 REDCONF_RENAME_ATOMIC

```
#define REDCONF_RENAME_ATOMIC
```

Whether to support atomic rename functionality in [red_rename\(\)](#).

See the documentation of [red_rename\(\)](#) for a description of what this macro enables. If enabled, the file system will require more memory.

Definition at line 165 of file redconf.h.

25.29.1.20 REDCONF_TASK_COUNT

```
#define REDCONF_TASK_COUNT
```

The number of file system tasks supported by the driver.

Must be large enough for every task which will ever use the file system; not just the number which will use it at any given time.

If set to 1, all mutex code is conditioned out and there is no protection against multiple tasks entering the file system.

Definition at line 55 of file redconf.h.

25.29.1.21 REDCONF_TRANSACT_DEFAULT

```
#define REDCONF_TRANSACT_DEFAULT
```

Default automatic transaction mask for all volumes.

These are the default events in the automatic transaction mask. The possible values are defined in [redapimacs.h](#).

This transaction mask can be changed at run time with [red_transmaskset\(\)](#) or [RedFseTransMaskSet\(\)](#). If there are multiple volumes and they need to be configured with different transaction masks, this must be setup at run time.

Definition at line 146 of file redconf.h.

25.30 projects/doxygen/redtypes.h File Reference

Defines basic types used by Reliance Edge.

Typedefs

- `typedef int bool`
Boolean type; either true or false.
- `typedef unsigned char uint8_t`
Unsigned 8-bit integer.
- `typedef signed char int8_t`
Signed 8-bit integer.
- `typedef unsigned short uint16_t`
Unsigned 16-bit integer.
- `typedef short int16_t`
Signed 16-bit integer.
- `typedef unsigned int uint32_t`
Unsigned 32-bit integer.
- `typedef int int32_t`
Signed 32-bit integer.
- `typedef unsigned long long uint64_t`
Unsigned 64-bit integer.
- `typedef long long int64_t`
Signed 64-bit integer.
- `typedef uint32_t uintptr_t`
Unsigned integer capable of storing a pointer.

25.30.1 Detailed Description

Defines basic types used by Reliance Edge.

The following types *must* be defined by this header, either directly (using `typedef`) or indirectly (by including other headers, such as the C99 headers `stdint.h` and `stdbool.h`):

- `bool`: Boolean type, capable of storing true (1) or false (0)
- `uint8_t`: Unsigned 8-bit integer
- `int8_t`: Signed 8-bit integer
- `uint16_t`: Unsigned 16-bit integer
- `int16_t`: Signed 16-bit integer
- `uint32_t`: Unsigned 32-bit integer
- `int32_t`: Signed 32-bit integer
- `uint64_t`: Unsigned 64-bit integer
- `int64_t`: Signed 64-bit integer

- `uintptr_t`: Unsigned integer capable of storing a pointer, preferably the same size as pointers themselves.

These types deliberately use the same names as the standard C99 types, so that if the C99 headers `stdint.h` and `stdbool.h` are available, they may be included here.

If the user application defines similar types, those may be reused. For example, suppose there is an application header `apptypes.h` which defines types with a similar purpose but different names. That header could be reused to define the types Reliance Edge needs:

```
~~~{.c} #include <apptypes.h>  
typedef BOOL bool; typedef BYTE uint8_t; typedef INT8 int8_t; And so on... ~~~
```

If there are neither C99 headers nor suitable types in application headers, this header should be populated with `typedef`s that define the required types in terms of the standard C types. This requires knowledge of the size of the C types on the target hardware (e.g., how big is an "int" or a pointer). Below is an example which assumes the target has 8-bit chars, 16-bit shorts, 32-bit ints, 32-bit pointers, and 64-bit long longs:

```
~~~{.c} typedef int bool; typedef unsigned char uint8_t; typedef signed char int8_t; typedef unsigned short uint16_t;  
typedef short int16_t; typedef unsigned int uint32_t; typedef int int32_t; typedef unsigned long long uint64_t; typedef  
long long int64_t; typedef uint32_t uintptr_t; ~~~
```

25.30.2 Typedef Documentation

25.30.2.1 `bool`

```
typedef int bool
```

Boolean type; either true or false.

Definition at line 62 of file `redtypes.h`.

25.30.2.2 `int16_t`

```
typedef short int16_t
```

Signed 16-bit integer.

Definition at line 68 of file `redtypes.h`.

25.30.2.3 `int32_t`

```
typedef int int32_t
```

Signed 32-bit integer.

Definition at line 71 of file `redtypes.h`.

25.30.2.4 int64_t

```
typedef long long int64_t
```

Signed 64-bit integer.

Definition at line 74 of file redtypes.h.

25.30.2.5 int8_t

```
typedef signed char int8_t
```

Signed 8-bit integer.

Definition at line 65 of file redtypes.h.

25.30.2.6 uint16_t

```
typedef unsigned short uint16_t
```

Unsigned 16-bit integer.

Definition at line 67 of file redtypes.h.

25.30.2.7 uint32_t

```
typedef unsigned int uint32_t
```

Unsigned 32-bit integer.

Definition at line 70 of file redtypes.h.

25.30.2.8 uint64_t

```
typedef unsigned long long uint64_t
```

Unsigned 64-bit integer.

Definition at line 73 of file redtypes.h.

25.30.2.9 uint8_t

```
typedef unsigned char uint8_t
```

Unsigned 8-bit integer.

Definition at line 64 of file redtypes.h.

25.30.2.10 uintptr_t

```
typedef uint32_t uintptr_t
```

Unsigned integer capable of storing a pointer.

Definition at line 76 of file redtypes.h.

25.31 projects/mqx/TWR-K65F180M/RelianceEdge/redosconf.c File Reference

Variables

- const REDMQXDEVICE gaRedDeviceConf [REDCONF_VOLUME_COUNT]

List of block device properties, in the same order as the VOLCONF array in redconf.c.

25.31.1 Variable Documentation

25.31.1.1 gaRedDeviceConf

```
const REDMQXDEVICE gaRedDeviceConf[REDCONF_VOLUME_COUNT]
```

List of block device properties, in the same order as the VOLCONF array in redconf.c.

Each element in this array should define the block device name (as given to _nio_dev_install or _io_dev_install [...] before initializing Reliance Edge) and whether the flushing the device with IO_IOCTL_FLUSH_OUTPUT is supported.

Definition at line 13 of file redosconf.c.

Referenced by redfs_format(), and redfs_install().

Index

ARM mbed, 47
Accessing Reliance Edge Data on a PC, 117
atime, 90
Atomic sector writes, 87
Atomicity, 5
Automatic discards, 79, 97

BDEVINFO, 159
 ulSectorSize, 159

BDEVOOPENMODE
 redoserv.h, 218

BDevTest, 135

BDEV_O_RDONLY
 redoserv.h, 220

BDEV_O_RDWR
 redoserv.h, 220

BDEV_O_WRONLY
 redoserv.h, 220

Block size, 84
bool
 redtypes.h, 319

Build warnings, 57
Building, 53
Building, file system driver, 53
Building, host tools, 57

card_present
 RedSDFFileSystem, 179

card_type
 RedSDFFileSystem, 180

CardType
 RedSDFFileSystem, 178

CARD_MMC
 RedSDFFileSystem, 178

CARD_NONE
 RedSDFFileSystem, 178

CARD_SD
 RedSDFFileSystem, 178

CARD_SDHC
 RedSDFFileSystem, 178

CARD_UNKNOWN
 RedSDFFileSystem, 178

Checker, 114
close
 RedFileHandle, 164

closedir

 RedDirHandle, 162

Command-line host tools, 113
Configuration, 77
Copying data to a PC, 117
crc
 RedSDFFileSystem, 180

d_ino
 REDDIRENT, 161

d_name
 REDDIRENT, 161

d_stat
 REDDIRENT, 161

Determinism, 125
Determinism, reads, 126
Determinism, writes, 127
Discard, 79, 88

f_bavail
 REDSTATFS, 185

f_bfree
 REDSTATFS, 185

f_blocks
 REDSTATFS, 185

f_bsize
 REDSTATFS, 185

f_dev
 REDSTATFS, 185

f_favail
 REDSTATFS, 186

f_ffree
 REDSTATFS, 186

f_files
 REDSTATFS, 186

f_flag
 REDSTATFS, 186

f_frsize
 REDSTATFS, 186

f_fsid
 REDSTATFS, 187

f_maxfsize
 REDSTATFS, 187

f_namemax
 REDSTATFS, 187

FSE API test suite, 138
FSIOTest, 139

FSE, 75
 FUSE implementation, 118
 File System Essentials, 75
 flen
 RedFileHandle, 164
 format
 RedFileSystem, 170
 Formatter, 113
 FreeRTOS, 23
 fse.c
 RedFseFormat, 190
 RedFselInit, 190
 RedFseMount, 191
 RedFseRead, 191
 RedFseSizeGet, 192
 RedFseTransMaskGet, 193
 RedFseTransMaskSet, 194
 RedFseTransact, 193
 RedFseTruncate, 195
 RedFseUninit, 195
 RedFseUnmount, 196
 RedFseWrite, 197
 fse/fse.c, 189
 fstat
 RedFileHandle, 165
 fstrim, 97
 fsync
 RedFileHandle, 165
 ftruncate
 RedFileHandle, 165
 gaRedDeviceConf
 redfs_mqx.h, 267
 redosconf.c, 321
 get_trans_mask
 RedFileSystem, 170
 Host tools, 113
 INTEGRITY, 27
 Image builder, 115
 Image copier, 117
 include/redapimacs.h, 198
 include/rederrno.h, 202
 include/redfse.h, 208
 include/redoserv.h, 217
 include/redposix.h, 226
 include/redstat.h, 257
 int16_t
 redtypes.h, 319
 int32_t
 redtypes.h, 319
 int64_t
 redtypes.h, 319
 int8_t
 redtypes.h, 320
 isatty
 RedFileHandle, 166
 large_frames
 RedSDFileSystem, 180, 181
 Limits, 121
 link
 RedFileSystem, 170
 Linux FUSE port, 118
 lseek
 RedFileHandle, 166
 MQX, 43
 Maximum file sizes, 121
 Maximum stack depth, 125
 Maximum volume sizes, 122
 mbed, 47
 mkdir
 RedFileSystem, 171
 mount
 RedFileSystem, 171
 mvstress, 141
 OS-specific API test suite, 143
 open
 RedFileSystem, 172
 opendir
 RedFileSystem, 172
 os/mbed/RedFileSystem/RedDirHandle.cpp, 258
 os/mbed/RedFileSystem/RedDirHandle.h, 258
 os/mbed/RedFileSystem/RedFileHandle.cpp, 258
 os/mbed/RedFileSystem/RedFileHandle.h, 259
 os/mbed/RedFileSystem/RedFileSystem.cpp, 259
 os/mbed/RedFileSystem/RedFileSystem.h, 259
 os/mbed/RedFileSystem/RedMemFileSystem.h, 260
 os/mbed/RedSDFileSystem/RedSDFileSystem.cpp, 260
 os/mbed/RedSDFileSystem/RedSDFileSystem.h, 260
 os/mqx/include/redfs_mqx.h, 261
 os/mqx/vfs/redfs_fio.c, 268
 os/mqx/vfs/redfs_nio.c, 268
 os/stub/include/redostypes.h, 270
 os/stub/services/osassert.c, 270
 os/stub/services/osbdev.c, 271
 os/stub/services/osclock.c, 276
 os/stub/services/osmutex.c, 278
 os/stub/services/osoutput.c, 280
 os/stub/services/ostask.c, 280
 os/stub/services/ostimestamp.c, 281
 osassert.c
 RedOsAssertFail, 271
 osbdev.c
 RedOsBDevClose, 272
 RedOsBDevDiscard, 272
 RedOsBDevFlush, 273

RedOsBDevGetGeometry, 274
RedOsBDevOpen, 274
RedOsBDevRead, 275
RedOsBDevWrite, 275
osclock.c
 RedOsClockGetTime, 277
 RedOsClockInit, 277
 RedOsClockUninit, 277
osmutex.c
 RedOsMutexAcquire, 278
 RedOsMutexInit, 278
 RedOsMutexRelease, 279
 RedOsMutexUninit, 279
osoutput.c
 RedOsOutputString, 280
ostask.c
 RedOsTaskId, 281
ostimestamp.c
 RedOsTimePassed, 282
 RedOsTimestamp, 282
 RedOsTimestampInit, 282
 RedOsTimestampUninit, 283
POSIX-like API test suite, 142
POSIX, 59
Paths, 60
Porting, 7
posix.c
 red_chdir, 285
 red_close, 285
 red_closedir, 286
 red_errno, 286
 red_format, 287
 red_fstat, 288
 red_fstrim, 288
 red_fsync, 289
 red_ftruncate, 290
 red_getcwd, 291
 red_gettransmask, 291
 red_init, 292
 red_link, 292
 red_lseek, 293
 red_mkdir, 294
 red_mount, 295
 red_mount2, 296
 red_open, 297
 red_opendir, 298
 red_read, 299
 red_readdir, 300
 red_rename, 300
 red_rewinddir, 302
 red_rmdir, 302
 red_settransmask, 303
 red_statvfs, 304
 red_sync, 305
 red_transact, 305
 red_umount, 306
 red_umount2, 307
 red_uninit, 308
 red_unlink, 308
 red_write, 309
 posix/posix.c, 283
 Project creation, 77
 projects/doxygen/redconf.h, 310
 projects/doxygen/redtypes.h, 318
 projects/mqx/TWR-K65F180M/RelianceEdge/redosconf.c, 321
 RAM usage, 124
 RED_DIR_PARAM, 160
 RED_EBADF
 rederrno.h, 204
 RED_EBUSY
 rederrno.h, 204
 RED_EEXIST
 rederrno.h, 204
 RED_EFBIG
 rederrno.h, 204
 RED_EFUBAR
 rederrno.h, 204
 RED_EINVAL
 rederrno.h, 205
 RED_EISDIR
 rederrno.h, 205
 RED_EIO
 rederrno.h, 205
 RED_EMFILE
 rederrno.h, 205
 RED_EMLINK
 rederrno.h, 205
 RED_ENAMETOOLONG
 rederrno.h, 206
 RED_ENFILE
 rederrno.h, 206
 RED_ENODATA
 rederrno.h, 206
 RED_ENOENT
 rederrno.h, 206
 RED_ENOSPC
 rederrno.h, 206
 RED_ENOSYS
 rederrno.h, 206
 RED_ENOTDIR
 rederrno.h, 207
 RED_ENOTEMPTY
 rederrno.h, 207
 RED_ENOTSUPP
 rederrno.h, 207

RED_EPERM
 rederrno.h, 207
RED_ERANGE
 rederrno.h, 207
RED_EROFS
 rederrno.h, 208
RED_EUSERS
 rederrno.h, 208
RED_EXDEV
 rederrno.h, 208
RED_FILENO_FIRST_VALID
 redfse.h, 209
RED_IOCTL_BASE
 redfs_mqx.h, 262
RED_IOCTL_CLOSERDIR
 redfs_mqx.h, 262
RED_IOCTL_FSTAT
 redfs_mqx.h, 263
RED_IOCTL_FTRUNCATE
 redfs_mqx.h, 263
RED_IOCTL_GET_TRANSMASK
 redfs_mqx.h, 263
RED_IOCTL_LINK
 redfs_mqx.h, 263
RED_IOCTL_MKDIR
 redfs_mqx.h, 264
RED_IOCTL_OPENDIR
 redfs_mqx.h, 264
RED_IOCTL_READDIR
 redfs_mqx.h, 264
RED_IOCTL_RENAME
 redfs_mqx.h, 264
RED_IOCTL_REWINDDIR
 redfs_mqx.h, 265
RED_IOCTL_RMDIR
 redfs_mqx.h, 265
RED_IOCTL_SET_TRANSMASK
 redfs_mqx.h, 265
RED_IOCTL_STATVFS
 redfs_mqx.h, 265
RED_IOCTL_TRANSACT
 redfs_mqx.h, 265
RED_IOCTL_UNLINK
 redfs_mqx.h, 266
RED_MOUNT_DEFAULT
 redapimacs.h, 199
RED_MOUNT_DISCARD
 redapimacs.h, 199
RED_MOUNT_MASK
 redapimacs.h, 199
RED_MOUNT_READONLY
 redapimacs.h, 199
RED_O_APPEND
 redposix.h, 229
RED_O_CREAT
 redposix.h, 229
RED_O_EXCL
 redposix.h, 229
RED_O_RDONLY
 redposix.h, 230
RED_O_RDWR
 redposix.h, 230
RED_O_TRUNC
 redposix.h, 230
RED_O_WRONLY
 redposix.h, 230
RED_RENAME_PARAM, 160
RED_S_IFDIR
 redstat.h, 257
RED_S_IFREG
 redstat.h, 257
RED_ST_NOSUID
 redstat.h, 257
RED_ST_RDONLY
 redstat.h, 258
RED_TRANSACT_CLOSE
 redapimacs.h, 200
RED_TRANSACT_CREAT
 redapimacs.h, 200
RED_TRANSACT_FSYNC
 redapimacs.h, 200
RED_TRANSACT_LINK
 redapimacs.h, 200
RED_TRANSACT_MANUAL
 redapimacs.h, 200
RED_TRANSACT_MKDIR
 redapimacs.h, 200
RED_TRANSACT_RENAME
 redapimacs.h, 201
RED_TRANSACT_SYNC
 redapimacs.h, 201
RED_TRANSACT_TRUNCATE
 redapimacs.h, 201
RED_TRANSACT_UNMOUNT
 redapimacs.h, 201
RED_TRANSACT_UNLINK
 redapimacs.h, 201
RED_TRANSACT_VOLFULL
 redapimacs.h, 201
RED_TRANSACT_WRITE
 redapimacs.h, 202
RED_UNMOUNT_DEFAULT
 redapimacs.h, 202
RED_UNMOUNT_FORCE
 redapimacs.h, 202
RED_UNMOUNT_MASK
 redapimacs.h, 202
REDCONF_ALIGNMENT_SIZE

redconf.h, 312
REDCONF_API_FSE
 redconf.h, 312
REDCONF_API_POSIX_RENAME
 redconf.h, 313
REDCONF_API_POSIX
 redconf.h, 312
REDCONF_ASSERTS
 redconf.h, 313
REDCONF_ATIME
 redconf.h, 313
REDCONF_BLOCK_SIZE
 redconf.h, 313
REDCONF_BUFFER_COUNT
 redconf.h, 314
REDCONF_CRC_ALGORITHM
 redconf.h, 314
REDCONF_DISCARDS
 redconf.h, 314
REDCONF_HANDLE_COUNT
 redconf.h, 314
REDCONF_IMAP_EXTERNAL
 redconf.h, 315
REDCONF_IMAP_INLINE
 redconf.h, 315
REDCONF_INODE_BLOCKS
 redconf.h, 315
REDCONF_INODE_TIMESTAMPS
 redconf.h, 315
REDCONF_NAME_MAX
 redconf.h, 315
REDCONF_PATH_SEPARATOR
 redconf.h, 316
REDCONF_READ_ONLY
 redconf.h, 316
REDCONF_RENAME_ATOMIC
 redconf.h, 317
REDCONF_TASK_COUNT
 redconf.h, 317
REDCONF_TRANSACT_DEFAULT
 redconf.h, 317
REDDIRENT, 161
 d_ino, 161
 d_name, 161
 d_stat, 161
REDFS_DEBUG
 RedFileSystem.h, 260
REDSTATFS, 184
 f_bavail, 185
 f_bfree, 185
 f_blocks, 185
 f_bsize, 185
 f_dev, 185
 f_favail, 186
 f_ffree, 186
 f_files, 186
 f_flag, 186
 f_frsize, 186
 f_fsid, 187
 f_maxfsiz, 187
 f_namemax, 187
REDSTAT, 182
 st_atime, 182
 st_blocks, 182
 st_ctime, 183
 st_dev, 183
 st_ino, 183
 st_mode, 183
 st_mtime, 183
 st_nlink, 183
 st_size, 184
REDTIMESTAMP
 redostypes.h, 270
REDWHENCE
 redposix.h, 231
ROM size, 125
read
 RedFileHandle, 167
Read determinism, 126
readdir
 RedDirHandle, 163
red_chdir
 posix.c, 285
 redposix.h, 231
red_close
 posix.c, 285
 redposix.h, 232
red_closedir
 posix.c, 286
 redposix.h, 232
red_dirent, 160
red_errno
 redposix.h, 229
red_errnoptr
 posix.c, 286
 redposix.h, 233
red_format
 posix.c, 287
 redposix.h, 233
red_fstat
 posix.c, 288
 redposix.h, 234
red_fstrim
 posix.c, 288
 redposix.h, 234
red_fsync
 posix.c, 289
 redposix.h, 235

red_ftruncate
 posix.c, 290
 redposix.h, 236
red_getcwd
 posix.c, 291
 redposix.h, 237
red_gettransmask
 posix.c, 291
 redposix.h, 238
red_init
 posix.c, 292
 redposix.h, 238
red_link
 posix.c, 292
 redposix.h, 239
red_lseek
 posix.c, 293
 redposix.h, 240
red_mkdir
 posix.c, 294
 redposix.h, 241
red_mount
 posix.c, 295
 redposix.h, 242
red_mount2
 posix.c, 296
 redposix.h, 242
red_open
 posix.c, 297
 redposix.h, 243
red_opendir
 posix.c, 298
 redposix.h, 245
red_read
 posix.c, 299
 redposix.h, 245
red_readdir
 posix.c, 300
 redposix.h, 246
red_rename
 posix.c, 300
 redposix.h, 247
red_rewinddir
 posix.c, 302
 redposix.h, 248
red_rmdir
 posix.c, 302
 redposix.h, 249
red_settransmask
 posix.c, 303
 redposix.h, 250
red_statvfs
 posix.c, 304
 redposix.h, 251
red_sync
 posix.c, 305
 redposix.h, 251
red_transact
 posix.c, 305
 redposix.h, 252
red_umount
 posix.c, 306
 redposix.h, 252
red_umount2
 posix.c, 307
 redposix.h, 253
red_uninit
 posix.c, 308
 redposix.h, 254
red_unlink
 posix.c, 308
 redposix.h, 255
red_write
 posix.c, 309
 redposix.h, 256
RedDirHandle, 162
 closedir, 162
 readdir, 163
 telldir, 163
RedFileHandle, 163
 close, 164
 flen, 164
 fstat, 165
 fsync, 165
 ftruncate, 165
 isatty, 166
 lseek, 166
 read, 167
 write, 167
RedFileSystem, 168
 format, 170
 get_trans_mask, 170
 link, 170
 mkdir, 171
 mount, 171
 open, 172
 opendir, 172
 RedFileSystem, 169
 remove, 173
 rename, 173
 set_trans_mask, 174
 statvfs, 174
 transact, 174
 unmount, 175
RedFileSystem.h
 REDFS_DEBUG, 260
RedFseFormat
 fse.c, 190

redfse.h, 209
RedFseInit
 fse.c, 190
 redfse.h, 210
RedFseMount
 fse.c, 191
 redfse.h, 211
RedFseRead
 fse.c, 191
 redfse.h, 211
RedFseSizeGet
 fse.c, 192
 redfse.h, 212
RedFseTransMaskGet
 fse.c, 193
 redfse.h, 213
RedFseTransMaskSet
 fse.c, 194
 redfse.h, 214
RedFseTransact
 fse.c, 193
 redfse.h, 212
RedFseTruncate
 fse.c, 195
 redfse.h, 214
RedFseUninit
 fse.c, 195
 redfse.h, 215
RedFseUnmount
 fse.c, 196
 redfse.h, 216
RedFseWrite
 fse.c, 197
 redfse.h, 216
RedMemFileSystem, 175
RedOsAssertFail
 osassert.c, 271
RedOsBDevClose
 osbdev.c, 272
 redoserv.h, 220
RedOsBDevDiscard
 osbdev.c, 272
RedOsBDevFlush
 osbdev.c, 273
 redoserv.h, 220
RedOsBDevGetGeometry
 osbdev.c, 274
 redoserv.h, 221
RedOsBDevOpen
 osbdev.c, 274
 redoserv.h, 222
RedOsBDevRead
 osbdev.c, 275
 redoserv.h, 222
RedOsBDevWrite
 osbdev.c, 275
 redoserv.h, 223
RedOsClockGetTime
 osclock.c, 277
 redoserv.h, 223
RedOsClockInit
 osclock.c, 277
 redoserv.h, 224
RedOsClockUninit
 osclock.c, 277
 redoserv.h, 224
RedOsMutexAcquire
 osmutex.c, 278
RedOsMutexInit
 osmutex.c, 278
RedOsMutexRelease
 osmutex.c, 279
RedOsMutexUninit
 osmutex.c, 279
RedOsOutputString
 osoutput.c, 280
RedOsTaskId
 ostask.c, 281
RedOsTimePassed
 ostimestamp.c, 282
 redoserv.h, 225
RedOsTimestamp
 ostimestamp.c, 282
 redoserv.h, 225
RedOsTimestampInit
 ostimestamp.c, 282
 redoserv.h, 225
RedOsTimestampUninit
 ostimestamp.c, 283
 redoserv.h, 226
RedSDFileSystem, 176
 card_present, 179
 card_type, 180
 CardType, 178
 crc, 180
 large_frames, 180, 181
 RedSDFileSystem, 179
 SwitchType, 178
 write_validation, 181
RED_SEEK_CUR
 redposix.h, 231
RED_SEEK_END
 redposix.h, 231
RED_SEEK_SET
 redposix.h, 231
redapimacs.h
 RED_MOUNT_DEFAULT, 199
 RED_MOUNT_DISCARD, 199

RED_MOUNT_MASK, 199
 RED_MOUNT_READONLY, 199
 RED_TRANSACT_CLOSE, 200
 RED_TRANSACT_CREAT, 200
 RED_TRANSACT_FSYNC, 200
 RED_TRANSACT_LINK, 200
 RED_TRANSACT_MANUAL, 200
 RED_TRANSACT_MKDIR, 200
 RED_TRANSACT_RENAME, 201
 RED_TRANSACT_SYNC, 201
 RED_TRANSACT_TRUNCATE, 201
 RED_TRANSACT_UNMOUNT, 201
 RED_TRANSACT_UNLINK, 201
 RED_TRANSACT_VOLFULL, 201
 RED_TRANSACT_WRITE, 202
 RED_UNMOUNT_DEFAULT, 202
 RED_UNMOUNT_FORCE, 202
 RED_UNMOUNT_MASK, 202
redconf.h
 REDCONF_ALIGNMENT_SIZE, 312
 REDCONF_API_FSE, 312
 REDCONF_API_POSIX_RENAME, 313
 REDCONF_API_POSIX, 312
 REDCONF_ASSERTS, 313
 REDCONF_ATIME, 313
 REDCONF_BLOCK_SIZE, 313
 REDCONF_BUFFER_COUNT, 314
 REDCONF_CRC_ALGORITHM, 314
 REDCONF_DISCARDS, 314
 REDCONF_HANDLE_COUNT, 314
 REDCONF_IMAP_EXTERNAL, 315
 REDCONF_IMAP_INLINE, 315
 REDCONF_INODE_BLOCKS, 315
 REDCONF_INODE_TIMESTAMPS, 315
 REDCONF_NAME_MAX, 315
 REDCONF_PATH_SEPARATOR, 316
 REDCONF_READ_ONLY, 316
 REDCONF_RENAME_ATOMIC, 317
 REDCONF_TASK_COUNT, 317
 REDCONF_TRANSACT_DEFAULT, 317
rederrno.h
 RED_EBADF, 204
 RED_EBUSY, 204
 RED_EEXIST, 204
 RED_EBBIG, 204
 RED_EFUBAR, 204
 RED EINVAL, 205
 RED_EISDIR, 205
 RED_EIO, 205
 RED_EMFILE, 205
 RED_EMLINK, 205
 RED_ENAMETOOLONG, 206
 RED_ENFILE, 206
 RED_ENODATA, 206
 RED_ENOENT, 206
 RED_ENOSPC, 206
 RED_ENOSYS, 206
 RED_ENOTDIR, 207
 RED_ENOTEMPTY, 207
 RED_ENOTSUPP, 207
 RED_EPERM, 207
 RED_ERANGE, 207
 RED_EROFS, 208
 RED_EUSERS, 208
 RED_EXDEV, 208
redfs_format
 redfs_mqx.h, 266
 redfs_nio.c, 268
redfs_install
 redfs_mqx.h, 266
 redfs_nio.c, 269
redfs_mqx.h
 gaRedDeviceConf, 267
 RED_IOCTL_BASE, 262
 RED_IOCTL_CLOSERDIR, 262
 RED_IOCTL_FSTAT, 263
 RED_IOCTL_FTRUNCATE, 263
 RED_IOCTL_GET_TRANSMASK, 263
 RED_IOCTL_LINK, 263
 RED_IOCTL_MKDIR, 264
 RED_IOCTL_OPENDIR, 264
 RED_IOCTL_READDIR, 264
 RED_IOCTL_RENAME, 264
 RED_IOCTL_REWINDDIR, 265
 RED_IOCTL_RMDIR, 265
 RED_IOCTL_SET_TRANSMASK, 265
 RED_IOCTL_STATVFS, 265
 RED_IOCTL_TRANSACT, 265
 RED_IOCTL_UNLINK, 266
redfs_uninstall
 redfs_mqx.h, 267
 redfs_nio.c, 269
redfs_nio.c
 redfs_format, 268
 redfs_install, 269
 redfs_uninstall, 269
redfs_uninstall
 redfs_mqx.h, 267
 redfs_nio.c, 269
redfse.h
 RED_FILENO_FIRST_VALID, 209
 RedFseFormat, 209
 RedFseInit, 210
 RedFseMount, 211
 RedFseRead, 211
 RedFseSizeGet, 212
 RedFseTransMaskGet, 213
 RedFseTransMaskSet, 214

RedFseTransact, 212
RedFseTruncate, 214
RedFseUninit, 215
RedFseUnmount, 216
RedFseWrite, 216
redosconf.c
 gaRedDeviceConf, 321
redoserv.h
 BDEOPENMODE, 218
 RedOsBDevClose, 220
 RedOsBDevFlush, 220
 RedOsBDevGetGeometry, 221
 RedOsBDevOpen, 222
 RedOsBDevRead, 222
 RedOsBDevWrite, 223
 RedOsClockGetTime, 223
 RedOsClockInit, 224
 RedOsClockUninit, 224
 RedOsTimePassed, 225
 RedOsTimestamp, 225
 RedOsTimestampInit, 225
 RedOsTimestampUninit, 226
redoserv.h
 BDEV_O_RDONLY, 220
 BDEV_O_RDWR, 220
 BDEV_O_WRONLY, 220
redostypes.h
 REDTIMESTAMP, 270
redposix.h
 RED_O_APPEND, 229
 RED_O_CREAT, 229
 RED_O_EXCL, 229
 RED_O_RDONLY, 230
 RED_O_RDWR, 230
 RED_O_TRUNC, 230
 RED_O_WRONLY, 230
 REDWHENCE, 231
 red_chdir, 231
 red_close, 232
 red_closedir, 232
 red_errno, 229
 red_errno, 233
 red_format, 233
 red_fstat, 234
 red_fstrim, 234
 red_fsync, 235
 red_ftruncate, 236
 red_getcwd, 237
 red_gettransmask, 238
 red_init, 238
 red_link, 239
 red_lseek, 240
 red_mkdir, 241
 red_mount, 242
 red_mount2, 242
 red_open, 243
 red_opendir, 245
 red_read, 245
 red_readdir, 246
 red_rename, 247
 red_rewinddir, 248
 red_rmdir, 249
 red_settransmask, 250
 red_statvfs, 251
 red_sync, 251
 red_transact, 252
 red_umount, 252
 red_umount2, 253
 red_uninit, 254
 red_unlink, 255
 red_write, 256
redposix.h
 RED_SEEK_CUR, 231
 RED_SEEK_END, 231
 RED_SEEK_SET, 231
RedSDFileSystem
 CARD_MMC, 178
 CARD_NONE, 178
 CARD_SD, 178
 CARD_SDHC, 178
 CARD_UNKNOWN, 178
 SWITCH_NEG_NC, 179
 SWITCH_NEG_NO, 179
 SWITCH_NONE, 178
 SWITCH_POS_NC, 178
 SWITCH_POS_NO, 178
redstat.h
 RED_S_IFDIR, 257
 RED_S_IFREG, 257
 RED_ST_NOSUID, 257
 RED_ST_RDONLY, 258
redtypes.h
 bool, 319
 int16_t, 319
 int32_t, 319
 int64_t, 319
 int8_t, 320
 uint16_t, 320
 uint32_t, 320
 uint64_t, 320
 uint8_t, 320
 uintptr_t, 320
remove
 RedFileSystem, 173
rename
 RedFileSystem, 173
Resource consumption, 124

Sector size, 85
set_trans_mask
 RedFileSystem, 174
st_atime
 REDSSTAT, 182
st_blocks
 REDSSTAT, 182
st_ctime
 REDSSTAT, 183
st_dev
 REDSSTAT, 183
st_ino
 REDSSTAT, 183
st_mode
 REDSSTAT, 183
st_mtime
 REDSSTAT, 183
st_nlink
 REDSSTAT, 183
st_size
 REDSSTAT, 184
statvfs
 RedFileSystem, 174
Stochastic Test, 143
Storage medium, 8
SwitchType
 RedSDFFileSystem, 178
SWITCH_NEG_NC
 RedSDFFileSystem, 179
SWITCH_NEG_NO
 RedSDFFileSystem, 179
SWITCH_NONE
 RedSDFFileSystem, 178
SWITCH_POS_NC
 RedSDFFileSystem, 178
SWITCH_POS_NO
 RedSDFFileSystem, 178

telldir
 RedDirHandle, 163
Testing, 135
The File System Essentials Interface, 157
The POSIX-like File System Interface, 155
transact
 RedFileSystem, 174
Transaction points, 5
Trim, 79, 88

U-Boot, 51
uint16_t
 redtypes.h, 320
uint32_t
 redtypes.h, 320
uint64_t
 redtypes.h, 320