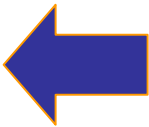

Rtfs

Failsafe Technical Reference

©2007 EBS, Inc
Revised June 2009



For best online viewing experience we recommend using Adobe Acrobat's **Bookmarks** tab for navigating



EBS Inc. 39 Court Street Groton MA 01450 USA
<http://www.ebseembeddedsoftware.com>

TABLE OF CONTENTS

Introduction	4
Synopsis	4
Failsafe architecture	4
Overview	4
Improvements over earlier versions of Failsafe.	5
Transaction Files	7
The Failsafe algorithm	8
Failsafe Journal file structure	10
Locating the Journal file in specific sectors	10
Locating the Journal file in free sectors	10
The Master Record	11
The Frame Record	11
Compile time configuration options	12
Run time configuration options	13
Application level API	14
fs_api_enable	15
fs_api_disable	17
fs_api_commit	18
fs_api_async_commit_start	20
fs_api_restore	22
fs_api_info	23
pc_diskio_failsafe_stats	25
Failsafe Callback API	28
RTFS_CB_FS_RETRIEVE_FIXED_JOURNAL_LOCATION	28

RTFS_CB_FS_FAIL_ON_JOURNAL_RESIZE_____	28
RTFS_CB_FS_FAIL_ON_JOURNAL_FULL_____	29
RTFS_CB_FS_RETRIEVE_JOURNAL_SIZE_____	29
RTFS_CB_FS_RETRIEVE_RESTORE_STRATEGY _____	29
RTFS_CB_FS_FAIL_ON_JOURNAL_CHANGED_____	30
RTFS_CB_FS_CHECK_JOURNAL_BEGIN_NOT _____	30
RTFS_CB_FS_RETRIEVE_FLUSH_STRATEGY _____	31
Failsafe During Application Development_____	33
Failsafe Testing _____	35
Basic Tests _____	35
Unit Tests _____	36
Transaction File Tests _____	36

Introduction

Synopsis

- Failsafe architecture is explained.
- A Failsafe API is provided.
- A fully transparent journaling mode, with no API calls required, is also provided.
- Several callback routines are available that modify the behavior of Failsafe, this document discusses callbacks that perform the following tasks:
 - Select Journal File placement
 - Select Journal File size
 - Determine auto restore strategy
 - Determines the strategy if auto-restore fails
 - Automatically enable failsafe journaling when a volume is mounted
- Failsafe is configured by initializing a Failsafe context block and returning it from **device_configure_volume**.
- This document discusses how configuration parameters affect performance.
 - Block map size impacts the number of blocks that may be remapped during a session.
 - By default 512 remap structures (4 kilobytes). Are assigned to each drive. This should be adequate for any application but you can make API calls to learn the maximum number of structures your application consumes.
 - The size of the restore buffer impacts performance.
 - For a typical format, applications that manipulate files less than 200 megabytes, performance should increase until the restore buffer size reaches approximately 70 (35 K), and then flatten out.
 - For a typical format, applications that manipulate files one Gigabyte or larger, performance should increase until the restore buffer size reaches approximately 130 (65 K), and then flatten out.
 - These values depend somewhat on how the device is formatted and will vary depending on the format.
 - If memory consumption is a problem the restore buffer size can be reduced. This will result in a linear decline in synchronization performance as the buffer size is reduced.
 - The minimum restore buffer size is 2 sectors. (1024 for typical devices with 512 byte sectors)

Failsafe architecture

Overview

- Failsafe eliminates the risk of file system corruption caused by unexpected power interruptions and media removal events.
- Changes to directory entries and FAT tables are written to a Journal file instead of directly to the volume.

- After the Journal file is flushed all preceding file system operations are safe.
- Journal file flush may be invoked manually or automatically.
- A separate synchronization step updates the volume from the Journal file.
- If the synchronization step is interrupted by a power failure it can be resumed when the system is restarted (restored).
- After a restore completes the volume structure is guaranteed to contain the same information as the flushed Journal file.
- A special file open mode is available which forces file writes to be transactional in nature. When a transactional file write returns, the volume is guaranteed to contain the data that was written.
- Sequence numbers, session IDs, and replacement block checksums are maintained to detect corruption of journal file blocks.
- FAT free space information is stored in the journal file to detect if volume changes occurred while Failsafe was not active.
- Automatic or manual operation – Failsafe may be configured in such a way that Failsafe operation is completely transparent to the application layer, or it may be configured such that the API must be called to flush, synchronize, or restore from the Journal. A mix of automatic and manual operations may also be created. For example, the callbacks may be configured to specify that journaling is automatically enabled, but API calls must be performed to flush the Journal and synchronize with the volume.
- Programmers Interface – An API is provided to perform necessary operations including enable and disable journaling, restore a volume from the Journal, and retrieve the status of a session or Journal file.
- Callback Interface – A callback API is provided to control the Failsafe algorithm.
- Journal file placement -The Journal file size and placement may be specified through a callback function. This allows system integrators to place the Journal data blocks in a separate partition or in a reserved section of the media.
- If the Journal file size and placement are not specified, the Journal file contents are placed in free sectors that RtfS reserves for the Journal file. This makes the Journal file clusters un-allocated when media is placed in another PC or device.
- When the disk volume fills. The Journal file is automatically resized and reserved free clusters are released to RtfS for allocation.
- Journal file access functions are segregated so that alternative schemes, such as journaling to on-board NVRAM, may be devised.

Improvements over earlier versions of Failsafe.

- Journal data is hidden in free sectors
- Journaling performance - Failsafe performs multi-block transfers to the Journal file whenever possible. This mirrors the access characteristics of RtfS making operating speed when Journaling comparable to normal operation.
- Journal flushes are much faster, requiring only two additional sector writes per flush. This makes frequent Journal flushing practical.
- Journal synchronization is much faster. During synchronization, Failsafe performs multi-block read of the Journal file and writes to the volume whenever possible. This provides a significant performance improvement and makes frequent synchronization practical.

- The Journal file is organized as a circular linked list of frames. This allows efficient scheduling of separate file flush and volume synchronization steps.
- Synchronization may be deferred, or performed from a background thread.
- Transaction processing - Failsafe's auto-commit feature combined with its fast Journal file flush feature allows it to perform high performance transaction based operations. At the application's discretion, Rtfs operations may be performed as atomic transactions, such that when an API call returns the disk image is guaranteed to reflect the operation requested.
- Transaction file IO – A new extended file open option is available that provides transactional file write and overwrite capability. Overwrites are a challenge that distinguish transactional file systems from Journaling file systems. To perform file region overwrites, a transactional file system must provide a data rollback mechanism in case a file region overwrite operation is interrupted. Failsafe provides this capability with minimal data copying. Zero data copying is needed if data is cluster aligned, if the data is not cluster aligned as few bytes as necessary are copied to implement the rollback feature.

Transaction Files

Transaction files are files opened with the **PCE_TRANSACTION_FILE** option selected.

- Transaction files provide a way to perform write operations in such a way that when a write call returns, the data is guaranteed to be on the media and available even if power is interrupted.
- If the write call does not succeed then the data is guaranteed to not be a part of the file and the file is guaranteed unchanged.
- Transaction file writes and overwrites perform more slowly than normal writes but they are fast enough, adding only a few additional block writes per call, that applications requiring transaction level assurance for certain files can use them practically.
- Transaction file metadata changes are written to the Journal file and the Journal file is flushed. If the power is interrupted before the volume is synchronized the metadata changes are synchronized with the volume by a restore.

Transaction file write operations that overwrite existing sections of the file perform the following operations:

- New clusters are allocated to hold the data to be overwritten.
- If the file pointer or the write count is not cluster aligned, then one or possibly two overlapping clusters are copied from the file to the new clusters.
- Cluster aligned data is written directly to the newly allocated clusters.
- The file's cluster chain is modified to exclude the overwritten clusters and to contain the new clusters.
- Before the write call returns the new cluster chain information is flushed to the journal file and the journal file is flushed.
- Once the write call returns you can be assured that the data that was written is present and the cluster chain information is preserved and the file size is updated.
- If the over-write call fails, the contents of the file are guaranteed to be the same as they were before the overwrite call was made.

The Failsafe algorithm

- The Journal file consists of a one block Journal file header followed by one or more Journal file segments.
- Journal file segments consist of one or more frames.
- Journal file frames contain a one block frame header and replacement block index that maps replacement blocks to blocks on the volume. The frame header is followed by a variable number of replacement blocks. If any transaction file overwrites have occurred the replacement blocks contain updated FAT table blocks, directory blocks, and possibly file data.
- The frame header is written to the disk when either the replacement block index is full or a Journal flush occurs.
- Only the current frame header is written when the Journal is flushed, but this puts all frames in the current segment into the flushed state.
- A Journal file segment may be synchronized with the volume anytime after it is flushed.
- After a Journal file segment is flushed, a new frame is created, to start a new segment, to hold subsequent journal requests.
- A session may contain as many flushed segments as will fit in the Journal file. The synchronization step processes outstanding flushed segments sequentially.

Each frame cycles between the following states:

- **OPEN** - The frame is receiving volume blocks. When a block is recorded in the Journal the frame's block remap table is updated and the block is written to the Journal file at the end of the current frame. If power is interrupted during this phase, transactions performed so far on this frame and previous frames in the current segment are lost. The volume structure suffers no corruption and all operations performed on the current segment are lost. All previous flushed segments are still valid and may be restored to the volume. Various flush strategies are available to allow the application to determine the number of outstanding operations in this state (down to as low as one).
- **CLOSED** - The frame's block remap index is full and the frame header has been flushed, but the segment containing this frame is not in the flushed state, so if power is interrupted during this phase, transactions performed so far on this segment are lost. The volume structure suffers no corruption and all operations performed on the current segment are lost.
- **FLUSHED** - The segment that this frame terminates is flushed, but the FAT volume has not yet been synchronized. Once a segment enters this state, Failsafe guarantees that the volume can be restored to include all operations performed on the segment.

- **SYNCHRONIZING** - The segment that this frame terminates is flushed and the volume is currently being synchronized from it. This state is entered after the **FLUSHED** state. It updates FAT tables and directory blocks from the Journal file. If power is interrupted during this phase, the FAT volume will be corrupted, but Failsafe guarantees that the Failsafe restore process will complete the synchronization step. If check disk or scan disk are run before the restore is executed they will show lost cluster chains. After the Failsafe restore procedure is executed, check disk or scan disk will show no lost cluster chains and the FAT volume structure will be identical to the view of the file system the application had when the segment was flushed.
- **SYNCHRONIZED** - The segment that this frame terminates and the volume are synchronized. The frames in the segment will be automatically re-used.

Failsafe Journal file structure

The Journal file contains a master record in the first sector followed by one or more segments, each consisting of one or more frames. Frames contain a frame record in the first sector followed by a variable number of replacement blocks. Replacement blocks contain raw data to be copied to the volume structure. The next frame follows immediately after the current frame record and its raw replacement blocks.

Locating the Journal file in specific sectors

The **RTFS_CB_FS_RETRIEVE_FIXED_JOURNAL_LOCATION** callback may be used to specify that a Journal file of a specific size be placed at a specific sector offset on the disk. If this method is used then the Journal file is created and accessed at these locations only. None of the placement and resizing operations described in the next section occur.

Locating the Journal file in free sectors

If the **RTFS_CB_FS_RETRIEVE_FIXED_JOURNAL_LOCATION** callback is not being used the Journal file is placed in free sectors using the following algorithm:

- Create - Find enough free clusters to hold the Journal and write that cluster number into location one of FAT copy 1.
- Reopen - Read the cluster number from location one of FAT copy 1 and read the block at that cluster and check for a Failsafe signature.
- Remove the link after a synchronize when the Journal file is no longer needed
 - Read the first block of FAT copy 0 and write it to FAT copy 1.

The algorithm performs the following steps.

- The **RTFS_CB_FS_RETRIEVE_JOURNAL_SIZE** callback is used to retrieve the recommend Journal size.
- The Journal is placed at the first location on the volume having this many contiguous free sectors.
- If not enough contiguous free sectors are found The **RTFS_CB_FS_FAIL_ON_JOURNAL_RESIZE** callback is made and if resizing is allowed the size is reduced until enough free contiguous clusters are found.
- If resizing is not allowed then the **RTFS_CB_FS_FAIL_ON_JOURNAL_FULL** callback is made to determine if Journaling should be disabled.
- When the Journal file is flushed the cluster location of the free sectors is written to the second reserved cluster (cluster 1) in FAT copy 1.
- At least two FAT copies are required. If there is only one FAT copy Journaling is automatically disabled.
- When the Journal file is reopened, the cluster number is retrieved from the reserved location in FAT copy 1. If the cluster number is valid and not reserved the Journal data is accessed from the cluster at that location.
- The location of the Journal file is cleared from FAT copy 1 by overwriting the first block of FAT copy 1 with the contents from FAT copy 0. This happens when a volume restore completes or when the volume synchronization step completes and no new frames have been flushed.

The Master Record

There is one master record for the Journal file. It is contained in the first sector of the Journal file. It is initialized when journaling starts and it is only updated in the rare case of a Journal file wrap, which requires a change to the start record field.

Master Record Contents	
Offset in dwords	Description
0	Not a valid Journal file if this field does not contain 'F','A','I','L' .
1	Not a valid Journal file if this field does not contain 'S','A','F','E'
2	Not a valid Journal file if this field does not contain the current failsafe version number
3	The size of the journal file in 512 byte blocks.
4	Session Id. This value is incremented every time a new Journal session is started. All frames written during this session will have this session ID.
5	Start Record. Contains the offset of the first valid frame in this session. This will initially contain the value one, but it changes to contain the first frame of the first unsynchronized segment if the Journal wraps the end of file.
6-127	These fields are unused

The Frame Record

This record is contained in the first block of each frame. The frame record contains a few words of session information followed by the volume block numbers associated with the FAT and directory replacement blocks to follow. The number of replacement blocks per frame is limited by the number of block numbers that can be stored in the frame record. The next frame record may be found by adding the number of replacement blocks in the current frame record to the block number of the current frame record. If the next frame record calculated by this method is beyond the journal file size, then the next frame record may be found at block one of the journal file.

Frame Record Contents	
Offset in dwords	Description
0	<p>Frame Type – The frame type field contains the current operating state for this frame. The type field is used to coordinate Journal, synchronize, and restore operations.</p> <p>1 - FS_FRAME_TYPE_OPEN – The record is written with this value when a frame is the current frame and blocks are currently being written to it.</p> <p>2 - FS_FRAME_TYPE_NULL - This frame is reserved and ignored. Used to create an empty frame if the frame record lands on the last sector in the journal file and no replacement blocks can be allocated to it.</p> <p>3 - FS_FRAME_TYPE_CLOSED – The record is full, all possible index blocks in the frame have been used and the frame index has been flushed. A new frame is opened to continue the segment.</p>

	<p>4 - FS_FRAME_TYPE_FLUSHED - This value written by a Failsafe Journal flush. This frame and all closed frames that precede it make up a valid segment that may be synchronized or restored.</p> <p>5 - FS_FRAME_TYPE_RESTOREING - A frame that was previously marked FS_FRAME_TYPE_FLUSHED is currently being restored. If the system is interrupted during this state the volume will have some degree of corruption, and the restore process should be restarted and completed to eliminate the corruption.</p> <p>6 - FS_FRAME_TYPE_RESTORED - The segment terminated by this frame has been successfully restored. The frame records and all replacement blocks in this segment will be re-used when the Journal file wraps and reaches the segment.</p>
1	Sequence - Sequence number within the current session. This value is incremented for each new frame that is created. Failsafe uses the frame sequence number and session ID to identify valid frames for the current session.
2	Frame checksum - Contains the ones complement checksum of all block numbers stored in the current frame. This field is used during restore and synchronize operations to detect corruption of the index block.
3	Segment checksum - Contains ones complement checksum of all block numbers stored in the current segment. This field is not used.
4	Frame record count - Contains the count of replacement blocks contained in this frame. These replacement blocks are stored in the sectors immediately following the frame record. The next frame record can be found after the last replacement block of the frame. If the current frame reaches the end of the Journal file the next frame may be found at block offset one.
5	FAT free space signature - Contains the free cluster count of the volume when the segment containing this frame was opened. If a session is flushed, but not yet restored, this value is compared against the current volume free space to determine if the volume was changed on another system or by Rtfs while Failsafe was disabled.
6	Session id - Contains the same value as the session ID in the master record. If these values do not match then the frame is not part of the current session and it marks the end of the session.
7-127	Replacement block index table - The rest of the frame record contains a table of block numbers. These are the block numbers on the volume where the replacement blocks that immediately follow are to be copied by the synchronize process.
Up to 121 blocks	Replacement block contents - Blocks following the frame record contain raw replacement blocks. These blocks are copied to the offsets in the volume stored in the replacement block index table.

Compile time configuration options

Several compile time constants in `rtfsconf.h` are used to configure Failsafe.

- **INCLUDE_FAILSAFE_CODE** - Rtfs must be compiled with this option enabled to use Failsafe.
- **INCLUDE_TRANSACTION_FILES** This option must be enabled if the transaction file feature of **pc_efilio_open()** is to be used. Disabling this option will save a small amount of code space if transaction files are not being used.

Run time configuration options

Memory for context structures and buffering are provided to Failsafe when a device is inserted or enabled and **pc_rtfs_media_insert()** is called.

The **rtfs_failsafe_callback()** function provides additional run-time control. The options are described below. Failsafe behavior may be changed by modifying the actions of the callback handlers.

Application level API

The following API calls are provided for Failsafe users.

- **fs_api_enable()** - Manually enable Failsafe.
- **fs_api_disable()** - Manually disable Failsafe.
- **fs_api_commit()** - Manually flush the Journal file and optionally synchronize the volume with the Journal.
- **fs_api_commit_start()** - Schedules a Journal flush and optional volume synchronization for later completion by **pc_async_continue()**.
- **fs_api_restore()** - Restore the volume from the Journal file.
- **fs_api_info()** - Retrieve information about the current active Failsafe session or the current on disk Journal file.
- **pc_diskio_failsafe_stats()** - Retrieve information about the current active Failsafe session or the current on disk Journal file.

fs_api_enable

FUNCTION

Enable or re-enable Failsafe for the specified drive.

SUMMARY

BOOLEAN **fs_api_enable** (drive_name, clear_journal)

byte * drive_name	Drive identifier, A:, B: C: etc
BOOLEAN clear_journal	Clear the current Journal contents

DESCRIPTION

Note: Except when clearing the journal file, fs_api_enable() should only be used when the RTFS_CB_FS_CHECK_JOURNAL_BEGIN_NOT callback handler is programmed to return TRUE or when the drive operating policy is set to DRVPOL_DISABLE_AUTOFAILSAFE. (see the documentation for device_configure_volume() in the API reference guide.)

fs_api_enable manually enables Failsafe if it is not already enabled. If clear_journal is **TRUE** the contents of the Journal file is cleared. If the Journal file contains valid data to be restored the data is cleared, if the Journal file has errors the errors are cleared.

If the drive is currently mounted, its buffers are flushed, and journaling is enabled immediately. If the drive is not currently mounted, internal state variables are set such that when the drive is next accessed, journaling will be enabled. If the drive is not currently mounted and clear_journal is **TRUE**, internal state variables are set to force RtfS to bypass the auto-restore on mount procedure(if it is currently enabled). If clear_journal is **FALSE** normal processing will occur on mount. This means auto-restoring from the Journal, if that feature is enabled, and detecting Journal errors.

If journaling was automatically enabled for the drive by **pc_failsafe_callback(RTFS_CB_FS_CHECK_JOURNAL_BEGIN_NOT)**, it should not be necessary to call **fs_api_enable()** except in the following case:

fs_api_enable() must be used to clear Journal errors that were detected by the mount process or **fs_api_restore()**, or **fs_api_info()**. To clear these errors call **fs_api_enable()** with clear_journal set to **TRUE**.

RETURNS

TRUE	The operation was a success
FALSE	The operation failed consult errno

Application Level Error Return Codes

PEFSBUSY	Failsafe already enabled. Call fs_api_disable() first
PENOINIT	The drive was not configured to support Failsafe by

	device_configure_volume.
An Rdfs system error	See Appendix for a description of system errors

fs_api_disable

FUNCTION

Disable Failsafe for the specified drive.

SUMMARY

BOOLEAN **fs_api_disable** (drive_name, abort)

byte *drive_name	Drive identifier, A:, B: C: etc
BOOLEAN abort	If TRUE force an immediate shut-down

DESCRIPTION

Note: fs_api_disable() should only be used with the drive operating policy set to DRVPOL_DISABLE_AUTOFAILSAFE. (see the documentation for device_configure_volume() in the API reference guide.)

fs_api_disable() stops journaling for the specified drive and dismounts the device. If abort is **FALSE** the Journal file is flushed and the volume is synchronized. If abort is **TRUE** the Journal is not flushed and no volume synchronization is done, and all recorded Journal session information is lost.

After **fs_api_disable()** is called, if journaling is automatically enabled, it will be restarted the next time the volume is accessed, if it is not, **fs_api_enable()** must be called to restart journaling.

RETURNS

TRUE	If no errors were encountered
FALSE	An error occurred

If an error occurred: errno is set to one of the following:

An RTFS system error	See Appendix for a description of system errors
-----------------------------	---

fs_api_commit

FUNCTION

Flush the Journal and optionally synchronize the volume.

SUMMARY

BOOLEAN **fs_api_commit**(drive_name, synch_volume)

byte *drive_name	Drive identifier, A:, B: C: etc
BOOLEAN synch_volume	If TRUE synchronize the volume after flushing the Journal.

DESCRIPTION

Note: fs_api_commit() may be used in conjunction with schemes that use the RTFS_CB_FS_RETRIEVE_FLUSH_STRATEGY callback handler to control the journal algorithm so that journal flush and volume synchronization are not performed on every API call. By default it will have no effect because the journal is flushed and synchronized every time. Fs_api_commit() or fs_api_async_commit_start() must be used if the drive operating policy is set to DRVPOL_DISABLE_AUTOFAILSAFE. (see the documentation for device_configure_volume() in the API reference guide.)

fs_api_commit() flushes the Journal file. When it returns, the Journal file is guaranteed to be synchronized with the application's view of the file system. If synch_volume is **TRUE** the volume is synchronized from the volume structure.

*Note: If synch_volume is **TRUE** and fs_api_commit() does not return because of some power or removal event there is no way to tell if the journal file was flushed. If this assurance is necessary call fs_api_commit() twice, once with synch_volume set to **FALSE**, and the second time with synch_volume set to **TRUE**. If the first call succeeds, the contents are guaranteed preserved. If the second call succeeds the volume structure is guaranteed to be up to date. If the first call succeeded but the second call did not, your application can restore the volume by calling fs_api_restore().*

*Note: If the disk is configured with **DRVPOL_ASYNC_AJFLUSH** and **DRVPOL_ASYNC_AJRESTORE** enabled, the functionality of fs_api_commit() may also be achieved by calling pc_async_continue().*

To emulate **pc_api_commit()** with no volume synchronization, execute:

```
pc_async_continue(drive_number, DRV_ASYNC_DONE_JOURNALFLUSH, 0);
```

To emulate **pc_api_commit()** with volume synchronization, execute:

```
pc_async_continue(drive_number, DRV_ASYNC_DONE_RESTORE, 0);
```

RETURNS

TRUE	If no errors were encountered
FALSE	An error occurred

If an error occurred: `errno` is set to one of the following:

An RTFS system error	See Appendix for a description of system errors
-----------------------------	---

fs_api_async_commit_start

FUNCTION

Schedule asynchronous Journal flush and volume synchronization.

SUMMARY

int **fs_api_async_commit_start** (drive_name)

byte *drive_name	Drive identifier "A:", "B:" etc.
-------------------------	----------------------------------

DESCRIPTION

Note: **fs_api_commit_start()** may be used in conjunction with schemes that use the **RTFS_CB_FS_RETRIEVE_FLUSH_STRATEGY** callback handler to control the journal algorithm so that journal flush and volume synchronization are not performed on every API call. By default it will have no effect because the journal is flushed and synchronized every time. **Fs_api_commit()** or **fs_api_async_commit_start()** must be used if the drive operating policy is set to **DRVPOL_DISABLE_AUTOFAILSAFE**. (see the documentation for **device_configure_volume()** in the API reference guide.)

fs_api_async_commit_start() – Schedules a Journal flush and volume synchronization for later completion by **pc_async_continue()**. This function is provided for rare cases when Failsafe is not configured to support asynchronous completion, but a single instance of asynchronous completion is required. To perform an asynchronous flush and synchronize when **DRVPOL_ASYNC_AJFLUSH** and **DRVPOL_ASYNC_AJRESTORE** were not enabled in **device_configure_volume**, execute the function and then call **pc_async_continue()** with a target state of **DRV_ASYNC_DONE_RESTORE**.

Note: ***This function is deprecated and should not be used.*** A preferred method is to configure the drive with **DRVPOL_ASYNC_AJFLUSH** and **DRVPOL_ASYNC_AJRESTORE** enabled and control when the flush and restore occur by controlling the target state when calling **pc_async_continue()**, setting **target_state** to **DRV_ASYNC_DONE_FLUSH** to flush the journal and **DRV_ASYNC_DONE_RESTORE** to synchronize the volume.

RETURNS

PC_ASYNC_CONTINUE	Success, call pc_async_continue() to complete processing
PC_ASYNC_ERROR	Error, drive not valid or Failsafe is not active.

Application Level Error Return Codes

PENONIT	Failsafe is not active
An RTFS system error	Invalid drive parameter, drive not available, etc.

fs_api_restore

FUNCTION

Restore the volume from the Journal file.

SUMMARY

BOOLEAN **fs_api_restore** (drive_name)

byte *drive_name	Name of the volume "A:", "B:" etc.
-------------------------	------------------------------------

DESCRIPTION

Note: **fs_api_restore()** may be used in conjunction with schemes that program the **RTFS_CB_FS_RETRIEVE_RESTORE_STRATEGY** callback handler to return **FS_CB_CONTINUE** or **FS_CB_ABORT**. By default this routine will have no effect because **RTFS_CB_FS_RETRIEVE_RESTORE_STRATEGY** instructs Failsafe to automatically perform a restore when the volume is mounted. **fs_api_restore** must be used if the drive operating policy is set to **DRVPOL_DISABLE_AUTOFAILSAFE**. (see the documentation for **device_configure_volume()** in the API reference guide.)

fs_api_restore() checks the Journal file and if it is valid and contains data to be restored, it restores the volume from the recorded Journal data.

If the Journal is valid but contains no data it returns **TRUE** immediately.

If the Journal file contains invalid data or an IO error occurs during processing, it return **FALSE**. In this event the best strategy is to call **fs_api_info()** and check the status values returned in the info structure to determine if the error is due to stale or overwritten data in the Journal file or if the error is due to an actual error. If the error is due to stale or overwritten data in the journal file you should call **fs_api_enable()** with the **clear_error** parameter set to **TRUE** to clear the error condition.

RETURNS

TRUE	If no errors were encountered
FALSE	An error occurred

If an error occurred: **errno** is set to one of the following:

0	Probably bad data in the journal, call fs_api_info
PENOINIT	Failsafe is not active
PEEINPROGRESS	Restore operation already in progress
An RTFS system error	Some read, write, or drive not available condition

fs_api_info

FUNCTION

Return information about the current active Failsafe session or the current on disk Journal file.

SUMMARY

BOOLEAN **fs_api_info** (drive_name, pfsinfo)

byte *drive_name	Drive identifier, A:, B: C: etc
FSINFO *pinfo	Address of a failsafe information structure to be filled in by the function.

typedef struct fsinfo {

BOOLEAN	journaling;	TRUE if Journaling is currently enabled
BOOLEAN	journal_file_valid;	TRUE if a Journal file was found and the master header contained a valid signature.
Dword	version;	Failsafe major and minor number (ie 0x31) If Journaling, the current version, if not, the version in the file.
Dword	numblocksremapped;	Number of volume replacement blocks currently in the Journal file
Dword	journaledfreespace;	Volume of free-space when the Journal was created
Dword	currentfreespace;	Current volume of free-space
Dword	journal_block_number;	Raw block number on the volume where the Journal resides
Dword	filesize;	Size of the Journal file in blocks
BOOLEAN	needsflush;	TRUE if journaling is active and a flush is required
BOOLEAN	out_of_date;	TRUE if journaling is inactive, the Journal contains data, but the saved free-space does not match the current free-space
BOOLEAN	check_sum_fails;	TRUE if journaling is inactive, the Journal contains data, but a checksum indicates that one or more Journal indices are corrupted.
BOOLEAN	restore_required;	TRUE if journaling is inactive, the Journal contains data, and a volume restore was started at some point but not completed. If this field is TRUE it indicates that the volume is probably corrupted and fs_api_restore() must be called to repair it.
BOOLEAN	restore_recommended;	TRUE if journaling is inactive, the Journal contains data, and a volume restore was not started or completed. If this field is TRUE it indicates that the Journal file was flushed but the

		synchronization never occurred. fs_api_restore() may be called to synchronize the volume with current recorded Journal data.
--	--	--

} FSINFO;

DESCRIPTION

If automatic restore processing is enabled you may call this when a disk mount fails because of a Journal file error (see the callback routine descriptions to learn how to control this). One of the error fields should be set and **fs_api_enable()** must be called with clear_journal set to **TRUE** to clear the condition.

If automatic restore processing is not enabled you may call this routine at power up to determine if a call to **fs_api_restore()** is required or recommended. If the field restore_required is **TRUE** this indicates a volume synchronization step was interrupted and **fs_api_restore()** must be executed or you will see volume corruption. If the field restore_recommended is **TRUE** this indicates that the Journal file was flushed but the volume synchronization step was not executed. If **fs_api_restore()** is not executed the recorded Journal information will be lost, but you will see no volume corruption. If one of the error fields is set, the volume can not be restored and **fs_api_enable()** must be called with clear_journal set to **TRUE** to clear the condition.

If Failsafe Journaling is currently active this routine may be called to query the number of blocks currently containing data, the size of the Journal file and whether the Journal file needs to be flushed. This latter fact may be used by a background process to periodically force a Journal flush based on that status information

RETURNS

TRUE	No errors occurred and the information in the FSINFO structure is valid.
FALSE	Some error occurred and the information in the FSINFO structure is not valid.

Application Level Error Return Codes

PEINVALIDDRIVEID	Not a valid drive identifier
PENOINIT	Drive is not configured for Failsafe use
An Rtfs system error	See Appendix for a description of system errors

pc_diskio_failsafe_stats

FUNCTION

Return failsafe buffering and usage patterns for a drive.

*Note: To get the full benefit of **pc_diskio_failsafe_stats()** you must compile the RtfsProPlus library with **INCLUDE_DEBUG_RUNTIME_STATS** enabled. If it is not enabled only some fields of the **DRIVE_RUNTIME_STATS** structure will be populated.*

SUMMARY

BOOLEAN **pc_diskio_failsafe_stats** (driveid, pstats)

byte *driveid	Name of a mounted volume "A," "B," etc.
FAILSAFE_RUNTIME_STATS *pstats	Usage statistics are placed into this structure.

See the function description for a detailed description of the **FAILSAFE_RUNTIME_STATS** structure and the meaning of the fields.

DESCRIPTION

These statistics are provided for determining if you have optimally configured Rtfs Failsafe and Journaling for your application. Please see the description of the pstats structure below to learn how to interpret these statistics.

Detailed description of the stats structure fields. All fields are of type dword .	
Failsafe internal statistics. If Failsafe is not enabled these fields will all be zero valued. Some of the information is obscure, some other fields contain information you may wish to analyze.	
journaling_active	If this value is 1, journaling is enabled
sync_in_progress	If this value is 1, an asynchronous Journal to volume synchronization is in progress.
journal_file_size	The size of the Journal file in blocks
journal_file_used	The number of blocks in the Journal file, in blocks, currently in use
journal_max_used	The maximum number of blocks in the Journal that have been used since the drive was initialized
restore_buffer_size	The size of the restore buffer in blocks provided to Failsafe by device_configure_volume .
num_blockmaps	The number of block map structures provided to Failsafe by device_configure_volume .
num_blockmaps_used	The number of block map structures currently in use
max_blockmaps_used	The maximum number of block map structures that have ever been in use at one time.
reserved_free_clusters	This is the total number of free clusters that are being reserved for use by Failsafe. If the disk fills, this value will be automatically reduced. If Journaling is disabled these clusters will be available for allocation.

	<i>Note: This value will be zero if Failsafe has been configured to place the Journal file in fixed sectors.</i>
cluster_frees_pending	<p>This is the total number of clusters that were released by file deletes and directory removals since the last volume synchronization. The clusters are not released to free-space until the volume is synchronized or serious volume corruption would result.</p> <p>Because of this, when Failsafe is journaling you will not see an increase in free-space after file deletes. To release the clusters for reuse you must synchronize.</p>
freestore_pass_count	Count of steps the restore state machine has made in the current synchronize phase. If no synchronize is occurring, the field contains the steps required to complete the last synchronization.
frames_restored	How many frames are being restored during the current active synchronize.
frames_closed	Number of frame index blocks that have been written to the disk since the last flush. Each frame indexes approximately 128 blocks of Metadata.
frames_flushed	Number of frame index blocks that have been written to the disk up to the last flush. These frames will be synchronized the next time synchronization is executed. They can be restored if a sudden system halt occurs.
current_frame	Current frame number where Metadata is being recorded in the Journal.
current_index	Current offset into the Frame where Metadata is being recorded in the Journal.
flushed_blocks	Total number of Metadata blocks contained within the region in flushed_frames.
open_blocks	Total number of Metadata blocks recorded to the Journal since the last flush.
restoring_blocks	Number of combined Metadata and index blocks that will be restored during the current synchronize.
restored_blocks	Number of combined Metadata and index blocks restored so far during the current synchronize.
current_restoring_block	The starting block of the last group of blocks read during the current active synchronize
<p>Journaling access statistics. These values will be zero if Failsafe Journaling is not enabled for the drive. These fields are mainly informational and will track the various directory block, and FAT table statistics in previous fields.</p> <p><i>Note: journal_data_reads, journal_data_blocks_read, journal_data_writes and journal_data_blocks_written log directory block, and FAT table accesses that are recorded to the Journal and do not represent additional disk accesses because Journaling is enabled.</i></p> <p>These fields are valid only when INCLUDE_DEBUG_RUNTIME_STATS is enabled.</p>	
journal_data_reads	Number of block read calls executed to access remapped Meta-data from the Journal file.
journal_data_blocks_read	Number of blocks of remapped Meta-data transferred from the Journal file.

journal_data_writes	Number of block write calls executed to place remapped Meta-data in the Journal file.
journal_data_blocks_written	Number of blocks of remapped Meta-data transferred to the Journal file.
journal_index_writes	Number of additional single block write calls made to update Journal index blocks.
journal_index_reads	Number of additional single block read calls made to Journal index blocks.
<p>These fields contain the number of additional disk blocks read from the Journal and written to the volume during Journal to volume synchronizations. To achieve optimal performance the ratio of blocks read, to the number of read calls should be maximized. This can be achieved by adjusting the restore buffer size as explained in the Failsafe User's manual.</p> <p>These fields are valid only when INCLUDE_DEBUG_RUNTIME_STATS is enabled.</p>	
restore_data_reads	Number of block read calls executed to access Journal index blocks and remapped Meta-data
restore_data_blocks_read	Number of index blocks and remapped Meta-data transferred from the Journal.
fat_synchronize_writes	Number of writes made to the FAT region of the disk by Failsafe.
fat_synchronize_blocks_written	Total number of FAT blocks written Failsafe.
dir_synchronize_writes	Number of writes made by Failsafe to regions of the disk other than the FAT.
dir_synchronize_blocks_written	Total number of blocks written by Failsafe to regions of the disk other than the FAT.
<p>Transaction file overwrite statistics.</p> <p>RtfsProPlus provides a rollback mechanism for files opened in Transaction file mode to restore the file to its previous state if the write call is interrupted. The rollback mechanism is a "zero copy" algorithm if the file data being overwritten is aligned on cluster boundaries. If the accesses are not cluster aligned, one and sometimes two, clusters must be copied. The following fields provide statistics on the number of such copies that are required by your application. The only way to optimize them is to adjust application access patterns.</p> <p>These fields are valid only when INCLUDE_DEBUG_RUNTIME_STATS is enabled.</p>	
transaction_buff_hits	Number of unaligned file block read accesses that were fulfilled by data caches during previous unaligned transaction file overwrites.
transaction_buff_reads	Number of unaligned file block disk reads that were required during transaction file overwrites.
transaction_buff_writes	Number of unaligned file block disk reads that were required during transaction file overwrites.

RETURNS

TRUE	The operation was a success
FALSE	The operation failed consult errno

Application Level Error Return Codes

PEINVALIDPARMS	Missing or invalid parameters
PEINVALIDDRIVEID	Invalid drive specified in an argument
An RTFS system error	See Appendix for a description of system errors

Failsafe Callback API

Failsafe uses a callback based interface for retrieving operating instructions and configuration values from the application layer. The callback interface also provides status updates to the application layer. When Failsafe has status changes to report or it needs instructions it calls the function named **rtfs_failsafe_callback()** and passes an operation code indicating what request or update it is performing. **rtfs_failsafe_callback()** must be provided by the application because it is not part of the RtfS library. A sample file named `rtfscallbacks.c` is provided, this file contains a version of **rtfs_failsafe_callback()** that may be used un-modified in most environments. **rtfs_failsafe_callback()** contains case statements for each possible request or status update.

`rtfscallbacks.c` provides default implementations for the operation codes listed below the source code may be modified to change the default behavior and to implement application extensions and diagnostics.

RTFS_CB_FS_RETRIEVE_FIXED_JOURNAL_LOCATION

rtfs_failsafe_callback() is called with this request when Failsafe is about to create or re-open the Journal.

This callback may be used to place Journal data at a fixed location on the disk. This may be in reserved sectors, on a separate partition, or in contiguous sectors within a file.

- To use Failsafe's default Journal file placement algorithm
 - return 0
- To place the Journal in a fixed location:
 - Return the start sector and size of the fixed region following the example provided in the source code.
 - return 1

RTFS_CB_FS_FAIL_ON_JOURNAL_RESIZE

rtfs_failsafe_callback() is called with this request when RtfS detects that there are not enough contiguous free space to hold a Journal file as large as the size prescribed by the **RTFS_CB_FS_RETRIEVE_JOURNAL_SIZE** callback. This may happen during a volume mount or when during cluster allocations when not enough free clusters are available.

The default behavior is to automatically reduce the journal file if enough space can be made available by doing so.

The **RTFS_CB_FS_FAIL_ON_JOURNAL_RESIZE** callback handler may be changed to force Failsafe to abandon journaling when free space gets low. RtfS then call s **rtfs_failsafe_callback(RTFS_CB_FS_FAIL_ON_JOURNAL_FULL..)** to determine it's next step.

Note: If the drive policy is set `DRVPOL_DISABLE_AUTOFAILSFAE`. (see `device_configure_volume()`) This routine is not called and resizing is

allowed.

RTFS_CB_FS_FAIL_ON_JOURNAL_FULL

rtfs_failsafe_callback() is called with this request when it can't resize or create a journal file because the drive does not have enough space to hold a minimum sized Journal file or it needed to free space by resizing the journal file but the **RTFS_CB_FS_FAIL_ON_JOURNAL_RESIZE** callback instructed it to fail instead.

The default behavior for Failsafe file fills is to disable Journaling and continue operating without journaling.

The **RTFS_CB_FS_FAIL_ON_JOURNAL_FULL** callback handler may be changed to force Rtfs to report a disk full condition instead.

The **RTFS_CB_FS_FAIL_ON_JOURNAL_FULL** callback may be also be used to monitor journal file full conditions without changing the default behavior.

Note: If the drive policy is set DRVPOL_DISABLE_AUTOFAILSAFE. (see device_configure_volume()) This routine is not called and the default behavior is to disable Journaling and continue operating without journaling.

RTFS_CB_FS_RETRIEVE_JOURNAL_SIZE

rtfs_failsafe_callback() is called with this request when Failsafe is about to create a Journal file. Failsafe passes the volume size in blocks, and the routine may override default values and recommended Journal file.

Notes:

- *The default method returns zero, which instructs Rtfs to calculate a size using a default algorithm that creates a journal file that is 1 / 128th the size of the volume or 0x100000 sectors, whichever is less.*
- *The **RTFS_CB_FS_RETRIEVE_JOURNAL_SIZE** callback handler may be changed to assign a file size instead of using the default algorithm.*
- *Failsafe does not call this routine if **RTFS_CB_FS_RETRIEVE_FIXED_JOURNAL_LOCATION** is being used to place the journal at a fixed location and size.*
- *The default Journal file placement algorithm uses the returned size when initializing the journal file. Journal size is reduced.*
- *If there are not enough free blocks available, the default Journal file placement algorithm attempts to automatically reduce the Journal file size.*
- *If the **RTFS_CB_FS_FAIL_ON_JOURNAL_RESIZE** callback is selected the file is not resized, the API call fails and errno is set to **PENOSPC**.*

RTFS_CB_FS_RETRIEVE_RESTORE_STRATEGY

rtfs_failsafe_callback() is called with this request when Rfs is about to mount a volume but it detects that the volume should be restored from the Journal file.

The default implementation instructs Failsafe to automatically restore the volumes from the Journal.

The **RTFS_CB_FS_RETRIEVE_RESTORE_STRATEGY** callback handler may be modified to change the behavior by returning one of the following:

FS_CB_CONTINUE - Tells Rtfs to proceed with the mount without restoring the volume.

FS_CB_ABORT - Tells Rtfs to terminate the mount, report failure to the API and set *errno* to **PEFSRESTORENEEDED**.

FS_CB_RESTORE - Tells Rtfs to restore the volume before it proceeds with the mount.

Note: If the drive policy is set DRVPOL_DISABLE_AUTOFailsafe. (see device_configure_volume()) This routine is not called and the default behavior is to proceed as if FS_CB_CONTINUE was returned.

Note: If the automatic restore is disabled, fs_api_restore() may be called from the application to perform the restore.

RTFS_CB_FS_FAIL_ON_JOURNAL_CHANGED

rtfs_failsafe_callback() is called with this request when the volume mount procedure detects a that a restore from Journal file procedure is required but it also detects that the volume was modified by a system not running Failsafe after the Journal file was last flushed.

If this routine returns zero (0) no restore processing will be done and the mount process will continue.

If this routine returns one (1) the mount will fail causing the API call to fail with *errno* set to **PEFSRESTOREERROR**.

RTFS_CB_FS_CHECK_JOURNAL_BEGIN_NOT

rtfs_failsafe_callback() is called with this request when a volume has just been mounted and it is configured for Failsafe operation.

The return value from this callback instructs Rtfs to either automatically start journaling or to wait for the API to initiate journaling.

The default behavior is to automatically start journaling.

The **RTFS_CB_FS_CHECK_JOURNAL_BEGIN_NOT** callback may change this

behavior and instruct RtfS to proceed with the mount, but not to automatically start journaling.

Note: If the drive policy is set `DRVPOL_DISABLE_AUTOFailsafe`. (see `device_configure_volume()`) This routine is not called and the default behavior is to proceed as if `TRUE` was returned, instructing RTFS to not automatically enable journaling and instead wait until `fs_api_enable()` is called from the application.

`fs_api_enable()` may be called from the application to start journaling when automatic journaling is disabled.

RTFS_CB_FS_RETRIEVE_FLUSH_STRATEGY

`rtfs_failsafe_callback()` is called with this request when an API call has caused a change in to the journal file and is about to return control to the user. The default return value is **`FS_CB_SYNC`**. See below for a description of this option and the other possible choices.

Auto-flush makes operations like file create, directory create, file delete, and directory removal transactional application calls, but, it does not automatically make file write operations transactional. Files must be opened with the **`PCE_TRANSACTION_FILE`** option enabled for file write operations to be transactional because additional special processing is required to implement transactional file writes.

`FS_CB_FLUSH` - The Journal file is flushed before the API call will return. If power is lost before the API call returns to the application layer, it is guaranteed that the transaction is lost as if it never executed. (note this is not true for file region overwrites, this requires transaction files). If API does return to the application, it may be assumed that if the power were lost, the operation is in the journal file and may be completed when the system is restarted, and a restore procedure is completed. **`FS_CB_FLUSH`** provides a means to quickly flush the journal file and insure to the application that the transaction is completed, but the application must at some point call `fs_api_commit()` to synchronize the FAT.

`FS_CB_SYNC` The Journal file is flushed and the FAT volume is synchronized with the Journal. When control returns to the application it may be assumed that if power were lost now the FAT volume contains the completed transaction. If power is lost before the API call returns to the application the transaction may be either lost or restorable, depending on when power was lost. If the Journal was not flushed when power was lost it will appear as if the operation never occurred. If the Journal was flushed when power was lost the transaction may be completed by executing a restore procedure. If power was lost during the FAT synchronization step then the FAT volume will be corrupted until a restore is executed.

`FS_CB_CONTINUE` The journal file is not flushed or synchronized, the application must call `fs_api_commit()` to flush the Journal file and optionally synchronize the volume.

Note: If the drive policy is set `DRVPOL_DISABLE_AUTOFailsafe`. (see

device_configure_volume()) This routine is not called and the default behavior is to proceed as if FS_CB_CONTINUE was returned, instructing RTFS to not perform any automatic journal file flush or synchronization and instead wait until fs_api_commit() is called from the application.

Failsafe During Application Development

You can use the function named **pc_diskio_failsafe_stats()** during application development and testing to check your configuration for both performance and capacity. This section discusses how some of the returned values can be used to tune your application. Please review the manual page for detailed information about what other information is provided by this function.

Monitoring worst case Journal file block consumption – API calls will fail if the amount of data written to the Journal ever exceeds the maximum number of free contiguous clusters on the volume. You can monitor worst case consumption to determine among other things, the largest amount of free space you will require to host your application with Journaling enabled. Check the field `journal_max_used` to determine the worst case number of blocks that have been allocated for journaling. This indicates how many free contiguous blocks must be available to run the application under these circumstances.

If `journal_max_used` is excessive you can modify your application to reduce Journal block consumption in one of the following ways.

- Flush the Journal and synchronize the volume more often – The Journal file continues to grow as API calls are made. The blocks are not released until the volume is synchronized
- Synchronize more often – If you are flushing frequently but synchronizing infrequently, the Journal file will grow by one or two additional blocks each time it is flushed. These blocks are only re-usable after they have been synchronized.
- Use **pc_deltree()** carefully – **pc_deltree()** can write large amounts of modified Meta-data to the Journal file in a single operation. It will require a Journal file at least as large as the total blocks in the FAT occupied by all released cluster chains and all modified directory blocks.

Monitoring worst case block map consumption – API calls will fail if the number of remapped (un-synchronized) block fragments ever exceeds the number of ram based block map structures that are provided by **device_configure_volume**. Check the field named `max_blockmaps_used` to verify that enough block map structures have been provided. Compare this field with `num_blockmaps` to determine if your default block map buffer is large enough.

Monitoring disk access patterns – Most of the additional IO overhead of Failsafe is consumed during the synchronize state when blocks are copied from the Journal file to the volume. This additional overhead may be reduced by:

- Decoupling the Journal flush and volume synchronize steps - You can perform one synchronize after performing many Journal flushes. This reduces the overhead but increases the likelihood that a restore will be needed if a power outage occurs.
- Adjusting the restore buffer size - The data is transferred using the largest multi-block transfers possible, constrained by the restore buffer size and how contiguous the Meta-data is. Increasing the restore buffer size will generally increase performance until other considerations dominate this effect. Sections to follow discuss this relationship in greater detail.

The following fields are provided to help analyze disk access patterns during synchronization: `restore_data_reads`, `restore_data_blocks_read`, `fat_synchronize_writes`, `fat_synchronize_blocks_written`, `dir_synchronize_writes`, and `dir_synchronize_blocks_written`

Failsafe Testing

Failsafe can be tested manually by running an application or simulated application while forcing IO errors or manually forcing media removals.

Rtfs also provides full suite of automated tests to validate the operation of Failsafe. These tests simulate data block corruptions, removal events and read and write errors to validate Failsafe operation. In addition to this method a set of tests have been devised that test boundary conditions, operating modes and recovery from simulated errors. These tests are provided in the files name prfstest.c and prasytest.c. The tests that are performed by these files are described below.

Basic Tests

The following basic tests are performed by source code in the file named **rtfscommon/apps/prfstest.c**.

- Test that Journaling works correctly when the Journal file wraps due to separately scheduled Journal flush and volume synchronization steps.
- Test the use of the **RTFS_CB_FS_RETRIEVE_FIXED_JOURNAL_LOCATION** callback to place the Journal at a fixed place and size. This test creates a contiguous file on a volume and then sets the fixed Journal placement callback coordinates to the file's extents. It then starts journaling and verifies the correct size and location of the Journal.
- Test that Failsafe properly resizes the initial Journal file size when journaling is started on a nearly full disk.
- Test that Failsafe properly resizes the active Journal file size when journaling is active on a nearly full disk.
- Test Journal full conditions. Verify correct operation and proper error handling when all records in the Journal file are consumed.
- Several tests are performed to check error conditions and correct operation for simulated error conditions.
 - Test out of date condition when a volume is changed with journaling disabled but there are flushed records in the file.
 - Test bad master record errors. Verify correct operation and error conditions when fields in the master record are manually corrupted.
 - Test bad frame record errors. Verify correct operation and error conditions when fields in frame records are manually corrupted.
 - Test that un-flushed volume changes are lost.
 - Test that flushed but unsynchronized volume changes are restored
 - Test that flushed volume changes can be synchronized
 - Test that aborted synchronizes may be restored
 - Test proper operation of separately scheduled Journal and synchronize operations. Verify that volume changes can be recorded to the Journal at one segment while another segment is being synchronized.

Unit Tests

The following tests are performed by source code in the file named `rtfspackages/apps/prasytest.c`

- Test that interrupting after a Journal flush leaves the disk undamaged and unchanged and that the recorded Journal changes can be restored to the volume by a restore.
- Test that when a write IO error occurs during the first Journal flush of a session, the Journal file is not valid.
- Test that volume corruption occurs when a write IO error occurs during a synchronize. Verify that the volume can be restored and the error condition is cleared.
- Test that Failsafe behaves properly when IO errors occur on all types of meta-data reads and writes. This coverage test simulates an IO error on every block that is read or written and verifies that Failsafe responds appropriately.
 - A test sequence creates subdirectories, populates files, and issues Journal flush and synchronize requests.
 - The test is calibrated by capturing all blocks of IO activity.
 - Benchmarks log the correct state of the volume after each synchronization.
 - The same operations are then performed in a repetitive loop.
 - ***An IO error is simulated for every block read and write.***
 - For each simulated error it is verified that Failsafe is performing correctly, by either leaving the volume unchanged if no flushes have occurred, or by restoring the volume so it matches the benchmark state associated with last successful flush operation.

Transaction File Tests

- The following tests are performed by source code in the file named `rtfspackages/apps/prtranstest.c`.
- Tests argument handling for **`pc_efilio_open()`** with the **`PCE_TRANSACTION_FILE`** option selected. Tests both 32 bit and 64 bit files.
- A set of tests verify correct behavior when reading, appending and overwriting transaction file data.
- The tests simulate power interruptions and then verify that transaction file data is in the correct state.
- The tests are performed using random write sizes at random offsets.
- The tests are also performed at various boundary conditions.