librsync Reference Manual

Generated by Doxygen 1.2.5

Sun Mar 18 16:54:50 2001

Contents

1 The librsync delta-encoding library

Author(s):
Martin Pool <mbp@samba.org>

Id:
main.dox,v 1.13 2001/03/12 01:30:50 mbp Exp

This document is also available in printable form:

- http://rproxy.samba.org/doxygen/librsync/refman.ps.gz
- http://rproxy.samba.org/doxygen/librsync/refman.pdf

More information about librsync can be found on the rproxy web site:

- http://rproxy.samba.org/

librsync can be downloaded from http://rproxy.samba.org/download.html, and used, modified and redistributed under the terms of the GNU Lesser General Public License (version 2.1 or later).

2 Introduction

librsync is a library for calculating and applying network deltas, with an interface designed to ease integration into diverse network applications. *librsync* is being developed as part of the **rproxy** <http://rproxy.samba.org/> and **rsync** <http://rsync.samba.org/> projects.

librsync encapsulates the core algorithms of the rsync protocol, which help with efficient calculation of the differences between two files. The rsync algorithm is different from most differencing algorithms because it does not require the presence of the two files to calculate the delta. Instead, it requires a set of checksums of each block of one file, which together form a signature for that file. Blocks at any in the other file which have the same checksum are likely to be identical, and whatever remains is the difference.

The library does not deal with file metadata or structure, such as filenames, permissions, or directories. To this library, a file is just a stream of bytes. Higher-level tools, such as rsync <http://rsync.samba.org> can deal with such issues in a way appropriate to their users.

The library supports three basic operations:

1. Generating the signature S of a file A .
2. Calculating a delta D from S and a new file B.
3. Applying D to A to reconstruct B.

The library also provides the rdiff network delta tool command-line tool, which makes this functionality available to users and scripting languages.

3 Programming interface

The public interface to librsync (rsync.h) has functions in several main areas:

- Data streaming
- Generating and applying deltas
- Buffers
- Processing whole files
- Debugging trace and error logging
- Encoding statistics
- Utility functions

All external symbols have the prefix "`rs_`", or "`RS_`" in the case of preprocessor symbols.

3.1 Data streaming

A key design requirement for librsync is that it should handle data as and when the hosting application requires it. librsync can be used inside applications that do non-blocking IO or filtering of network streams, because it never does IO directly, or needs to block waiting for data.

The programming interface to librsync is similar to that of zlib and bzlib. Arbitrary-length input and output buffers are passed to the library by the application, through an instance of rs_buffers_t. The library proceeds as far as it can, and returns an rs_result value indicating whether it needs more data or space.

All the state needed by the library to resume processing when more data is available is kept in a small opaque rs_job_t structure. After creation of a job, repeated calls to rs_job_iter() in between filling and emptying the buffers keeps data flowing through the stream. The rs_result_t values returned may indicate

- RS_DONE: processing is complete
- RS_BLOCKED: processing has blocked pending more data
- one of various possible errors in processing

These can be converted to a human-readable string by rs_strerror().

3.2 Generating and applying deltas

All encoding operations are performed by using a `*_begin` function to create a rs_job_t object, passing in any necessary initialization parameters. The various jobs available are:

- rs_sig_begin(): Calculate the signature of a file.
- rs_loadsig_begin(): Load a signature into memory.
- rs_delta_begin(): Calculate the delta between a signature and a new file.
- rs_patch_begin(): Apply a delta to a basis to recreate the new file.

3.3 Buffers

After creating a job, input and output buffers are passed to rs_job_iter() in an rs_buffers_t structure.

On input, the buffers structure must contain the address and length of the input and output buffers. The library updates these values to indicate the amount of **remaining** buffer. So, on return, `avail_out` is not the amount of output data produced, but rather the amount of output buffer space unfilled. This means that the values on return are consistent with the values on entry, but not necessarily what you would expect.

A similar system is used by `libz` and `libbz2`.

Warning:

The input may not be completely consumed by the iteration if there is not enough output space. The application must retain unused input data, and pass it in again when it is ready for more output.

3.4 Processing whole files

Some applications do not require fine-grained control over IO, but rather just want to process a whole file with a single call. librsync provides 'whole-file' functionality to do exactly that.

Processing of a whole file begins with creation of a rs_job_t object for the appropriate operation, just as if the application was going to do buffering itself. After creation, the job may be passed to rs_whole_run(), which will feed it to and from two FILEs as necessary until end of file is reached or the operation completes.

3.5 Debugging trace and error logging

librsync can output trace or log messages as it proceeds. These follow a fairly standard priority-based filtering system (rs_trace_set_level()), using the same severity levels as UNIX syslog. Messages by default are sent to stderr, but may be passed to an application-provided callback (rs_trace_to(), rs_trace_fn_t).

3.6 Encoding statistics

Encoding and decoding routines accumulate compression performance statistics in a rs_stats_t structure as they run. These may be converted to human-readable form or written to the log file using rs_format_stats() or rs_log_stats() respectively.

3.7 Utility functions

Some additional functions are used internally and also exposed in the API:

- encoding/decoding binary data: rs_base64(), rs_unbase64(), rs_hexify().
- MD4 message digests: rs_mdfour(), rs_mdfour_begin(), rs_mdfour_update(), rs_mdfour_result().

4 librsync Data Structure Index

4.1 librsync Data Structures

Here are the data structures with brief descriptions:

5 librsync File Index

5.1 librsync File List

Here is a list of all documented files with brief descriptions:

6 librsync Page Index

6.1 librsync Related Pages

Here is a list of all related documentation pages:

7 librsync Class Documentation

7.1 rs_buffers_s Struct Reference

Stream through which the calling application feeds data to and from the library.

`#include <rsync.h>`

Data Fields

- char* next_in
  Next input byte.
- size_t avail_in
  Number of bytes available at next_in.
- int eof_in
  True if there is no more data after this.
- char* next_out
  Next output byte should be put there.
- size_t avail_out
  Remaining free space at next_out.

7.1.1 Detailed Description

Stream through which the calling application feeds data to and from the library.

On each call to rs_job_iter, the caller can make available

- avail_in bytes of input data at next_in
- avail_out bytes of output space at next_out
- some of both

Buffers must be allocated and passed in by the caller. This routine never allocates, reallocates or frees buffers.

Pay attention to the meaning of the returned pointer and length values. They do **not** indicate the location and amount of returned data. Rather, if `*out_ptr` was originally set to `out_buf`, then the output data begins at `out_buf`, and has length `*out_ptr - out_buf`.

Note also that if *avail_in is nonzero on return, then not all of the input data has been consumed. The caller should either provide more output buffer space and call rs_work() again passing the same next_in and avail_in, or put the remaining input data into some persistent buffer and call rs_work() with it again when there is more output space.

Parameters:

next_in References a pointer which on entry should point to the start of the data to be encoded. Updated to point to the byte after the last one consumed.

avail_in References the length of available input. Updated to be the number of unused data bytes, which will be zero if all the input was consumed. May be zero if there is no new input, but the caller just wants to drain output.

next_out References a pointer which on entry points to the start of the output buffer. Updated to point to the byte after the last one filled.

avail_out References the size of available output buffer. Updated to the size of unused output buffer.

Returns:

The rs_result that caused iteration to stop.

See also:

rs_buffers_t , Buffers

7.1.2 Field Documentation

7.1.2.1 char * rs_buffers_s::next_in

Next input byte.

7.1.2.2 size_t rs_buffers_s::avail_in

Number of bytes available at next_in.

7.1.2.3 int rs_buffers_s::eof_in

True if there is no more data after this.

7.1.2.4 char * rs_buffers_s::next_out

Next output byte should be put there.

7.1.2.5 size_t rs_buffers_s::avail_out

Remaining free space at next_out.

The documentation for this struct was generated from the following file:

- rsync.h

7.2 rs_stats Struct Reference

Performance statistics from a librsync encoding or decoding operation.

`#include <rsync.h>`

Data Fields

- char const* op

  Human-readable name of current operation.

- int lit_cmds

  Number of literal commands.

- rs_long_t lit_bytes

  Number of literal bytes.

- rs_long_t lit_cmdbytes

  Number of bytes used in literal command headers.

- rs_long_t **copy_cmds**
- rs_long_t **copy_bytes**
- rs_long_t **copy_cmdbytes**
- rs_long_t **sig_cmds**
- rs_long_t **sig_bytes**
- int **false_matches**
- rs_long_t in_bytes

  Total bytes read from input.

- rs_long_t out_bytes

  Total bytes written to output.

7.2.1 Detailed Description

Performance statistics from a librsync encoding or decoding operation.

See also:
rs_format_stats(), rs_log_stats()

7.2.2 Field Documentation

7.2.2.1 char const * rs_stats::op

Human-readable name of current operation.

For example, ”delta”.

7.2.2.2 int rs_stats::lit_cmds

Number of literal commands.

7.2.2.3 rs_long_t rs_stats::lit_bytes

Number of literal bytes.

7.2.2.4 rs_long_t rs_stats::lit_cmdbytes

Number of bytes used in literal command headers.

7.2.2.5 rs_long_t rs_stats::in_bytes

Total bytes read from input.

7.2.2.6 rs_long_t rs_stats::out_bytes

Total bytes written to output.

The documentation for this struct was generated from the following file:

- rsync.h

7.3 rs_target Struct Reference

Description of the match described by a signature.

`#include <sumset.h>`

Data Fields

- unsigned short **t**
- int **i**

7.3.1 Detailed Description

Description of the match described by a signature.

The documentation for this struct was generated from the following file:

- sumset.h

8 librsync File Documentation

8.1 rsync.h File Reference

Main public interface to librsync.

Data Structures

- struct rs_stats

  Performance statistics from a librsync encoding or decoding operation.

- struct **rs_mdfour**
- struct rs_buffers_s

  Stream through which the calling application feeds data to and from the library.

Defines

- #define **RS_MD4_LENGTH** 16
- #define RS_DEFAULT_STRONG_LEN 8

  Default length of strong signatures, in bytes.

- #define RS_DEFAULT_BLOCK_LEN 2048

  Default block length, if not determined by any other factors.

Typedefs

- typedef long rs_long_t

  A long integer type that can handle the largest file offsets.

- typedef void rs_trace_fn_t (int level, char const *msg)

  Callback to write out log messages.

- typedef struct rs_stats rs_stats_t

  Performance statistics from a librsync encoding or decoding operation.

- typedef struct rs_mdfour rs_mdfour_t

  MD4 message-digest accumulator.

- typedef unsigned int **rs_weak_sum_t**
- typedef unsigned char **rs_strong_sum_t** [RS_MD4_LENGTH]
- typedef struct rs_signature **rs_signature_t**

- typedef struct rs_buffers_s rs_buffers_t

  Stream through which the calling application feeds data to and from the library.

- typedef struct rs_job rs_job_t

  Job of work to be done.

- typedef enum rs_work_options rs_work_options

  Bitmask values that may be passed to the options parameter of rs_work().

- typedef rs_result **rs_driven_cb** (rs_job_t *job, rs_buffers_t *buf, void *opaque)
- typedef rs_result rs_copy_cb (void *opaque, off_t pos, size_t *len, void **buf)

  Callback used to retrieve parts of the basis file.

Enumerations

- enum rs_loglevel { RS_LOG_EMERG = 0, RS_LOG_ALERT = 1, RS_LOG_CRIT = 2, RS_LOG_ERR = 3, RS_LOG_WARNING = 4, RS_LOG_NOTICE = 5, RS_LOG_INFO = 6, RS_LOG_DEBUG = 7 }

  Log severity levels.

- enum rs_result { RS_DONE = 0, RS_BLOCKED = 1, RS_RUNNING = 2, RS_TEST_SKIPPED = 77, RS_IO_ERROR = 100, RS_SYNTAX_ERROR = 101, RS_MEM_ERROR = 102, RS_INPUT_ENDED = 103, RS_BAD_MAGIC = 104, RS_UNIMPLEMENTED = 105, RS_CORRUPT = 106, RS_INTERNAL_ERROR = 107, RS_PARAM_ERROR = 108 }

  Return codes from nonblocking rsync operations.

- enum rs_work_options { RS_END = 0x01 }

  Bitmask values that may be passed to the options parameter of rs_work().

Functions

- void rs_trace_set_level (rs_loglevel level)

  Set the least important message severity that will be output.

- void rs_trace_to (rs_trace_fn_t *)

  Set trace callback.

- void rs_trace_stderr (int level, char const *msg)

  Default trace callback that writes to stderr.

- int rs_supports_trace (void)

  Check whether the library was compiled with debugging trace suport.

- void rs_hexify (char *to_buf, void const *from_buf, int from_len)

  Convert FROM_LEN bytes at FROM_BUF into a hex representation in TO_BUF, which must be twice as long plus one byte for the null terminator.

- size_t rs_unbase64 (char *s)

Decode a base64 buffer in place.

- void rs_base64 (unsigned char const *buf, int n, char *out)

  Encode a buffer as base64.

- char const* rs_strerror (rs_result r)

  Return an English description of a rs_result value.

- void **rs_mdfour** (unsigned char *out, void const *in, int n)
- void **rs_mdfour_begin** (rs_mdfour_t *md)
- void **rs_mdfour_update** (rs_mdfour_t *md, void const *, size_t n)
- void **rs_mdfour_result** (rs_mdfour_t *md, unsigned char *out)
- char* rs_format_stats (rs_stats_t const *, char *, size_t)

  Return a human-readable representation of statistics.

- int **rs_log_stats** (rs_stats_t const *stats)
- void rs_free_sumset (rs_signature_t *)

  Deep deallocation of checksums.

- void rs_sumset_dump (rs_signature_t const *)

  Dump signatures to the log.

- rs_result rs_job_iter (rs_job_t *, rs_buffers_t *)

  Run a rs_job_t state machine until it blocks (RS_BLOCKED), returns an error, or completes (RS_COMPLETE).

- rs_result rs_job_drive (rs_job_t *job, rs_buffers_t *buf, rs_driven_cb in_cb, void *in_opaque, rs_driven_cb out_cb, void *out_opaque)

  Actively process a job, by making callbacks to fill and empty the buffers until the job is done.

- rs_result **rs_job_free** (rs_job_t *)
- int **rs_accum_value** (rs_job_t *, char *sum, size_t sum_len)
- rs_job_t* rs_sig_begin (size_t new_block_len, size_t strong_sum_len)

  Set up a new encoding job.

- rs_job_t* rs_delta_begin (rs_signature_t *)

  Prepare to compute a streaming delta.

- rs_job_t* rs_loadsig_begin (rs_signature_t **)

  Read a signature from a file into an rs_signature_t structure in memory.

- rs_job_t* rs_patch_begin (rs_copy_cb *, void *copy_arg)

  Apply a delta to a basis to recreate the new file.

- rs_result **rs_build_hash_table** (rs_signature_t *sums)
- void rs_mdfour_file (FILE *in_file, char *result)

  Calculate the MD4 sum of a file.

- rs_result rs_sig_file (FILE *old_file, FILE *sig_file, size_t, size_t)

  Generate the signature of a basis file, and write it out to another.

- rs_result rs_loadsig_file (FILE *sig_file, rs_signature_t **sumset)
  Load signatures from a signature file into memory.
- rs_result rs_file_copy_cb (void *arg, off_t pos, size_t *len, void **buf)
  Default copy implementation that retrieves a part of a stdio file.
- rs_result **rs_delta_file** (rs_signature_t *, FILE *new_file, FILE *delta_file, rs_stats_t *)
- rs_result **rs_patch_file** (FILE *basis_file, FILE *delta_file, FILE *new_file, rs_stats_t *)

Variables

- char const rs_librsync_version [ ] = (PACKAGE " " VERSION)
  Library version string.
- char const **rs_licence_string** [ ]
- int rs_inbuflen
  Buffer sizes for file IO.
- int rs_outbuflen
  Buffer sizes for file IO.

8.1.1 Detailed Description

Main public interface to librsync.

Author(s):
Martin Pool <mbp@samba.org>

Version:
librsync-0.9.1

Id:
rsync.h,v 1.10 2001/03/18 02:00:08 mbp Exp

See Introduction for an introduction to use of this library.

8.1.2 Define Documentation

8.1.2.1 #define RS_DEFAULT_STRONG_LEN 8

Default length of strong signatures, in bytes.

The MD4 checksum is truncated to this size.

8.1.2.2 #define RS_DEFAULT_BLOCK_LEN 2048

Default block length, if not determined by any other factors.

8.1.3 Typedef Documentation

8.1.3.1 typedef long rs_long_t

A long integer type that can handle the largest file offsets.

Perhaps this might have to be configured to be 'long long', 'long', or something else depending on the platform.

8.1.3.2 typedef void rs_trace_fn_t

Callback to write out log messages.

Parameters:

level a syslog level.

msg message to be logged.

8.1.3.3 typedef struct rs_stats rs_stats_t

Performance statistics from a librsync encoding or decoding operation.

See also:

rs_format_stats(), rs_log_stats()

8.1.3.4 struct rs_mdfour rs_mdfour_t

MD4 message-digest accumulator.

See also:

rs_mdfour(), rs_mdfour_begin(), rs_mdfour_update(), rs_mdfour_result()

8.1.3.5 typedef struct rs_buffers_s rs_buffers_t

Stream through which the calling application feeds data to and from the library.

See also:

struct rs_buffers_s , Buffers

8.1.3.6 typedef struct rs_job rs_job_t

Job of work to be done.

Created by functions such as rs_sig_begin(), and then iterated over by rs_job_iter().

8.1.3.7 typedef enum rs_work_options rs_work_options

Bitmask values that may be passed to the options parameter of rs_work().

8.1.3.8 typedef rs_result rs_copy_cb

Callback used to retrieve parts of the basis file.

Parameters:

pos Position where copying should begin.

len On input, the amount of data that should be retrieved. Updated to show how much is actually available.

buf On input, a buffer of at least `*len` bytes. May be updated to point to a buffer allocated by the callback if it prefers.

8.1.4 Enumeration Type Documentation

8.1.4.1 enum rs_loglevel

Log severity levels.

These are the same as syslog, at least in glibc.

See also:

rs_trace_set_level()

Enumeration values:

RS_LOG_EMERG System is unusable.

RS_LOG_ALERT Action must be taken immediately.

RS_LOG_CRIT Critical conditions.

RS_LOG_ERR Error conditions.

RS_LOG_WARNING Warning conditions.

RS_LOG_NOTICE Normal but significant condition.

RS_LOG_INFO Informational.

RS_LOG_DEBUG Debug-level messages.

8.1.4.2 enum rs_result

Return codes from nonblocking rsync operations.

Enumeration values:

RS_DONE Completed successfully.

RS_BLOCKED Blocked waiting for more data.

RS_RUNNING Not yet finished or blocked.

This value should never be returned to the caller.

RS_TEST_SKIPPED Test neither passed or failed.

RS_IO_ERROR Error in file or network IO.

RS_SYNTAX_ERROR Command line syntax error.

RS_MEM_ERROR Out of memory.

RS_INPUT_ENDED End of input file, possibly unexpected.

RS_BAD_MAGIC Bad magic number at start of stream.

Probably not a librsync file, or possibly the wrong kind of file or from an incompatible library version.

RS_UNIMPLEMENTED Author is lazy.

RS_CORRUPT Unbelievable value in stream.

RS_INTERNAL_ERROR Probably a library bug.

RS_PARAM_ERROR Bad value passed in to library, probably an application bug.

8.1.4.3 enum rs_work_options

Bitmask values that may be passed to the options parameter of rs_work().

Enumeration values:

RS_END End of input file; please finish up.

8.1.5 Function Documentation

8.1.5.1 void rs_trace_set_level (rs_loglevel *level*)

Set the least important message severity that will be output.

8.1.5.2 void rs_trace_to (rs_trace_fn_t * *new_impl*)

Set trace callback.

The callback scheme allows for use within applications that may have their own particular ways of reporting errors: log files for a web server, perhaps, and an error dialog for a browser.

Todo:

Do we really need such fine-grained control, or just yes/no tracing?

8.1.5.3 void rs_trace_stderr (int *level*, char const * *msg*)

Default trace callback that writes to stderr.

Implements rs_trace_fn_t, and may be passed to rs_trace_to().

8.1.5.4 int rs_supports_trace (void)

Check whether the library was compiled with debugging trace suport.

If this returns false, then trying to turn trace on will achieve nothing.

8.1.5.5 void rs_hexify (char * *to_buf*, void const * *from_buf*, int *from_len*)

Convert FROM_LEN bytes at FROM_BUF into a hex representation in TO_BUF, which must be twice as long plus one byte for the null terminator.

8.1.5.6 size_t rs_unbase64 (char * *s*)

Decode a base64 buffer in place.

Returns:
the number of binary bytes.

8.1.5.7 void rs_base64 (unsigned char const * *buf*, int *n*, char * *out*)

Encode a buffer as base64.

8.1.5.8 char const* rs_strerror (rs_result *r*)

Return an English description of a rs_result value.

8.1.5.9 char * rs_format_stats (rs_stats_t const * *stats*, char * *buf*, size_t *size*)

Return a human-readable representation of statistics.

The string is truncated if it does not fit. 100 characters should be sufficient space.

Parameters:
stats Statistics from an encoding or decoding operation.
buf Buffer to receive result.
size Size of buffer.

Returns:
buf

8.1.5.10 void rs_free_sumset (rs_signature_t * *psums*)

Deep deallocation of checksums.

8.1.5.11 void rs_sumset_dump (rs_signature_t const * *sums*)

Dump signatures to the log.

8.1.5.12 rs_result rs_job_iter (rs_job_t * *job*, rs_buffers_t * *buffers*)

Run a rs_job_t state machine until it blocks (RS_BLOCKED), returns an error, or completes (RS_COMPLETE).

Returns:
The rs_result that caused iteration to stop.

Parameters:
ending True if there is no more data after what's in the input buffer. The final block checksum will run across whatever's in there, without trying to accumulate anything else.

8.1.5.13 rs_result rs_job_drive (rs_job_t * *job*, rs_buffers_t * *buf*, rs_driven_cb *in_cb*, void * *in_opaque*, rs_driven_cb *out_cb*, void * *out_opaque*)

Actively process a job, by making callbacks to fill and empty the buffers until the job is done.

8.1.5.14 rs_job_t* rs_sig_begin (size_t *new_block_len*, size_t *strong_sum_len*)

Set up a new encoding job.

See also:
rs_sig_file()

8.1.5.15 rs_job_t* rs_delta_begin (rs_signature_t * *sig*)

Prepare to compute a streaming delta.

8.1.5.16 rs_job_t* rs_loadsig_begin (rs_signature_t ** *signature*)

Read a signature from a file into an rs_signature_t structure in memory.

Once there, it can be used to generate a delta to a newer version of the file.

Note:
After loading the signatures, you must call rs_build_hash_table() before you can use them.

8.1.5.17 rs_job_t* rs_patch_begin (rs_copy_cb * *copy_cb*, void * *copy_arg*)

Apply a delta to a basis to recreate the new file.

This gives you back a rs_job_t object, which can be cranked by calling rs_job_iter() and updating the stream pointers. When finished, call rs_job_finish() to dispose of it.

Parameters:

stream Contains pointers to input and output buffers, to be adjusted by caller on each iteration.

copy_cb Callback used to retrieve content from the basis file.

copy_arg Opaque environment pointer passed through to the callback.

Todo:

As output is produced, accumulate the MD4 checksum of the output. Then if we find a CHECKSUM command we can check it's contents against the output.
Implement COPY commands.

See also:

rs_patch_file()

8.1.5.18 void rs_mdfour_file (FILE * *in_file*, char * *result*)

Calculate the MD4 sum of a file.

Parameters:

result Binary (not hex) MD4 of the whole contents of the file.

8.1.5.19 rs_result rs_sig_file (FILE * *old_file*, FILE * *sig_file*, size_t *new_block_len*, size_t *strong_len*)

Generate the signature of a basis file, and write it out to another.

Parameters:

new_block_len block size for signature generation, in bytes

strong_len truncated length of strong checksums, in bytes

See also:

rs_sig_begin()

8.1.5.20 rs_result rs_loadsig_file (FILE * *sig_file*, rs_signature_t ** *sumset*)

Load signatures from a signature file into memory.

Return a pointer to the newly allocated structure in SUMSET.

See also:

rs_readsig_begin()

8.1.5.21 rs_result rs_file_copy_cb (void * *arg*, off_t *pos*, size_t * *len*, void ** *buf*)

Default copy implementation that retrieves a part of a stdio file.

8.1.6 Variable Documentation

8.1.6.1 char const rs_librsync_version[ ] = (PACKAGE " " VERSION)

Library version string.

8.1.6.2 int rs_inbuflen

Buffer sizes for file IO.

8.1.6.3 int rs_outbuflen

Buffer sizes for file IO.

9 librsync Page Documentation

10 Glossary

Author(s):
Martin Pool <mbp@samba.org>

Id:
gloss.dox,v 1.1 2001/02/26 10:19:12 mbp Exp

10.1 delta

A description of changes necessary to bring a basis file to the new state.

10.2 basis

The old version of a file, to which a delta is applied. The delta may use blocks from the basis file to reconstruct the new file.

11 rdiff network delta tool

Author(s):
Martin Pool

Version:

Id:
rdiff.dox,v 1.1 2001/02/25 03:32:06 mbp Exp

Foo.

12 Todo List

global rs_mdfour64(rs_mdfour_t *m, unsigned int *M) Recode to be fast, and to use system integer types. Perhaps if we can find an mdfour implementation already on the system (e.g. in OpenSSL) then we should use it instead of our own?

Apparently rsync 2.4 now has a fast MD4 routine. So we should copy that into here.

global rs_patch_begin(rs_copy_cb *copy_cb, void *copy_arg) As output is produced, accumulate the MD4 checksum of the output. Then if we find a CHECKSUM command we can check it's contents against the output.

Implement COPY commands.

global rs_trace_to(rs_trace_fn_t *) Do we really need such fine-grained control, or just yes/no tracing?

Index