NAME

librrd − RRD library functions

DESCRIPTION

librrd contains most of the functionality in **RRDTool**. The command line utilities and language bindings are often just wrappers around the code contained in **librrd**.

This manual page documents the **librrd** API.

NOTE: This document is a work in progress, and should be considered incomplete as long as this warning persists. For more information about the **librrd** functions, always consult the source code.

CORE FUNCTIONS

rrd_dump_cb_r(char *filename, int opt_header, rrd_output_callback_t cb, void *user)

In some situations it is necessary to get the output of `rrd_dump` without writing it to a file or the standard output. In such cases an application can ask **rrd_dump_cb_r** to call an user-defined function each time there is output to be stored somewhere. This can be used, to e.g. directly feed an XML parser with the dumped output or transfer the resulting string in memory.

The arguments for **rrd_dump_cb_r** are the same as for **rrd_dump_opt_r** except that the output filename parameter is replaced by the user-defined callback function and an additional parameter for the callback function that is passed untouched, i.e. to store information about the callback state needed for the user-defined callback to function properly.

Recent versions of **rrd_dump_opt_r** internally use this callback mechanism to write their output to the file provided by the user.

```
size_t rrd_dump_opt_cb_fileout(
    const void *data,
    size_t len,
    void *user)
{
    return fwrite(data, 1, len, (FILE *)user);
}
```

The associated call for **rrd_dump_cb_r** looks like

```
res = rrd_dump_cb_r(filename, opt_header,
    rrd_dump_opt_cb_fileout, (void *)out_file);
```

where the last parameter specifies the file handle **rrd_dump_opt_cb_fileout** should write to. There's no specific condition for the callback to detect when it is called for the first time, nor for the last time. If you require this for initialization and cleanup you should do those tasks before and after calling **rrd_dump_cr_r** respectively.

rrd_fetch_cb_register(rrd_fetch_cb_t c)

If your data does not reside in rrd files, but you would like to draw charts using the rrd_graph functionality, you can supply your own rrd_fetch function and register it using the **rrd_fetch_cb_register** function.

The argument signature and api must be the same of the callback function must be aequivalent to the on of **rrd_fetch_fn** in *rrd_fetch.c*.

To activate the callback function you can use the pseudo filename *cb//free_form_text*.

Note that rrdtool graph will not ask the same rrd for data twice. It determines this by building a key out of the values supplied to the fetch function. If the values are the same, the previous answer will be used.

UTILITY FUNCTIONS

rrd_random()

Generates random numbers just like *random()*. This further ensures that the random number generator is seeded exactly once per process.

rrd_strtodbl
an rrd aware string to double converter which sets rrd_error in if there is a problem and uses the return code exclusively for conversion status reporting.

rrd_strtod
works like normal strtod, but it is locale independent (and thread safe)

rrd_snprintf
works like normal snprintf but it is locale independent (and thread safe)

rrd_add_ptr(void ***dest, size_t *dest_size, void *src)
Dynamically resize the array pointed to by `dest`. `dest_size` is a pointer to the current size of `dest`. Upon successful *realloc()*, the `dest_size` is incremented by 1 and the `src` pointer is stored at the end of the new `dest`. Returns 1 on success, 0 on failure.

```
type **arr = NULL;
type *elem = "whatever";
size_t arr_size = 0;
if (!rrd_add_ptr(&arr, &arr_size, elem))
    handle_failure();
```

rrd_add_ptr_chunk(void ***dest, size_t *dest_size, void *src, size_t *alloc, size_t chunk)
Like `rrd_add_ptr`, except the destination is allocated in chunks of `chunk`. `alloc` points to the number of entries allocated, whereas `dest_size` points to the number of valid pointers. If more pointers are needed, `chunk` pointers are allocated and `alloc` is increased accordingly. `alloc` must be >= `dest_size`.

This method improves performance on hosts with expensive `realloc()`.

rrd_add_strdup(char ***dest, size_t *dest_size, char *src)
Like `rrd_add_ptr`, except adds a `strdup` of the source string.

```
char **arr = NULL;
size_t arr_size = NULL;
char *str  = "example text";
if (!rrd_add_strdup(&arr, &arr_size, str))
    handle_failure();
```

rrd_add_strdup_chunk(char ***dest, size_t *dest_size, char *src, size_t *alloc, size_t chunk)
Like `rrd_add_strdup`, except the destination is allocated in chunks of `chunk`. `alloc` points to the number of entries allocated, whereas `dest_size` points to the number of valid pointers. If more pointers are needed, `chunk` pointers are allocated and `alloc` is increased accordingly. `alloc` must be >= `dest_size`.

rrd_free_ptrs(void ***src, size_t *cnt)
Free an array of pointers allocated by `rrd_add_ptr` or `rrd_add_strdup`. Also frees the array pointer itself. On return, the source pointer will be NULL and the count will be zero.

```
/* created as above */
rrd_free_ptrs(&arr, &arr_size);
/* here, arr == NULL && arr_size == 0 */
```

rrd_mkdir_p(const char *pathname, mode_t mode)
Create the directory named `pathname` including all of its parent directories (similar to `mkdir -p` on the command line − see *mkdir*(1) for more information). The argument `mode` specifies the permissions to use. It is modified by the process's `umask`. See *mkdir*(2) for more details.

The function returns 0 on success, a negative value else. In case of an error, `errno` is set accordingly. Aside from the errors documented in *mkdir*(2), the function may fail with the following errors:

EINVAL
`pathname` is `NULL` or the empty string.

ENOMEM
Insufficient memory was available.

any error returned by *stat*(2)

In contrast to *mkdir*(2), the function does **not** fail if `pathname` already exists and is a directory.

rrd_scaled_duration (const char * token, unsigned long divisor, unsigned long * valuep)
Parse a token in a context where it contains a count (of seconds or PDP instances), or a duration that can be converted to a count by representing the duration in seconds and dividing by some scaling factor. For example, if a user would natively express a 3 day archive of samples collected every 2 minutes, the sample interval can be represented by `2m` instead of `120`, and the archive duration by `3d` (to be divided by 120) instead of `2160` (3*24*60*60 / 120). See more examples in "STEP, HEARTBEAT, and Rows As Durations" in rrdcreate.

`token` must be a number with an optional single-character suffix encoding the scaling factor:

`s`	indicates seconds
`m`	indicates minutes. The value is multiplied by 60.
`h`	indicates hours. The value is multiplied by 3600 (or `60m`).
`d`	indicates days. The value is multiplied by 86400 (or `24h`).
`w`	indicates weeks. The value is multiplied by 604800 (or `7d`).
`M`	indicates months. The value is multiplied by 2678400 (or `31d`). (Note this factor accommodates the maximum number of days in a month.)
`y`	indicates years. The value is multiplied by 31622400 (or `366d`). (Note this factor accommodates leap years.)

`divisor` is a positive value representing the duration in seconds of an interval that the desired result counts.

`valuep` is a pointer to where the decoded value will be stored if the conversion is successful.

The initial characters of `token` must be the base−10 representation of a positive integer, or the conversion fails.

If the remainder `token` is empty (no suffix), it is a count and no scaling is performed.

If `token` has one of the suffixes above, the count is multiplied to convert it to a duration in seconds. The resulting number of seconds is divided by `divisor` to produce a count of intervals each of duration `divisor` seconds. If division would produce a remainder (e.g., `5m` (300 seconds) divided by `90s`), the conversion is invalid.

If `token` has unrecognized trailing characters the conversion fails.

The function returns a null pointer if the conversion was successful and `valuep` has been updated to the scaled value. On failure, it returns a text diagnostic suitable for use in user error messages.

CLIENT FUNCTIONS

The following functions are used to connected to an rrdcached instance, either via a unix or inet address, and create, update, or gather statistics about a specified RRD database file.

There are two different interfaces: The **rrd_client_** family of functions operate on a user-provided client object (**rrd_client_t**) and support multiple concurrent connections to rrdcache instances. The simpler **rrdc_** family of functions handles connections transparently but can only be used for one connection at a time.

All of the following functions and data types are specified in the `rrd_client.h` header file.

rrd_client_new(const char *daemon_addr)
Create a new client connection object. If specified, connect to the daemon at `daemon_addr`. The connection can later be changed by calling **rrd_client_connect**.

rrd_client_destroy(rrd_client_t *client)
Close a client connection and destroy the object by freeing all dynamically allocated memory. After calling this function, `client` can no longer be used.

rrd_client_connect(rrd_client_t *client, const char *daemon_addr)
rrdc_connect(const char *daemon_addr)
Connect to a running rrdcached instance, specified via `daemon_addr`. Any previous connection will be closed. If `daemon_addr` is `NULL`, it defaults to the value of the `ENV_RRDCACHED_ADDRESS` environment address.

rrd_client_is_connected(rrd_client_t *client)
Return a boolean int if the client is connected to the server.

rrd_client_address(rrd_client_t *client)
Returns the server address belonging to the current connection.

rrdc_is_connected(const char *daemon_addr)
Return a boolean int to determine if the client is connected to the rrdcache daemon specified by the `daemon_addr` parameter.

rrd_client_ping(rrd_client_t *client)
rrdc_ping
Check the client connection by pinging the remote side.

rrdc_is_any_connected
Return a boolean int if any daemon connections are connected.

rrd_client_disconnect(rrd_client_t *client)
rrdc_disconnect
Disconnect gracefully from the present daemon connection.

rrd_client_update(rrd_client_t *client, const char *filename, int values_num, const char * const *values)
rrdc_update(const char *filename, int values_num, const char * const *values)
Update the RRD `filename` via the rrdcached. Where `values_num` is the number of values to update and `values` are the new values to add.

rrd_client_info(rrd_client_t *client, const char *filename)
rrdc_info(const char *filename)
Grab rrd info of the RRD `filename` from the connected cache daemon. This function returns an rrd_info_t structure of the following format:

```
typedef struct rrd_blob_t {
    unsigned long size; /* size of the blob */
    unsigned char *ptr; /* pointer */
} rrd_blob_t;

typedef enum rrd_info_type { RD_I_VAL = 0,
    RD_I_CNT,
    RD_I_STR,
    RD_I_INT,
    RD_I_BLO
} rrd_info_type_t;

typedef union rrd_infoval {
    unsigned long u_cnt;
    rrd_value_t u_val;
```

```
            char     *u_str;
            int       u_int;
            rrd_blob_t u_blo;
        } rrd_infoval_t;

        typedef struct rrd_info_t {
            char     *key;
            rrd_info_type_t type;
            rrd_infoval_t value;
            struct rrd_info_t *next;
        } rrd_info_t;
```

rrd_client_last(rrd_client_t *client, const char *filename)
rrdc_last(const char *filename)

Grab the unix epoch of the last time RRD `filename` was updated.

rrd_client_first(rrd_client_t *client, const char *filename, int rraindex)
rrdc_first(const char *filename, int rraindex)

Get the first value of the first sample of the RRD `filename`, of the `rraindex` RRA (Round Robin Archive) index number. The RRA index number can be determined by pulling the rrd_info_t off the RRD.

rrd_client_create(rrd_client_t *client, const char *filename, unsigned long pdp_step, time_t last_up, int no_overwrite, int argc, const char **argv)
rrdc_create(const char *filename, unsigned long pdp_step, time_t last_up, int no_overwrite, int argc, const char **argv)

Create RRD database of path `filename`. The RRD will have a step size of `pfp_step`, the unix epoch timestamp to start collecting data from. The number of data sources and RRAs `argc` and the definitions of the data sources and RRAs `argv`. Lastly whether or not to overwrite an existing RRD if one is found with the same filename; `no_overwrite`.

rrdc_create_r2(rrd_client_t *client, const char *filename, unsigned long pdp_step, time_t last_up, int no_overwrite, const char **sources, const char *template, int argc, const char **argv)
rrdc_create_r2(const char *filename, unsigned long pdp_step, time_t last_up, int no_overwrite, const char **sources, const char *template, int argc, const char **argv)

Create an RRD database in the daemon. **rrdc_create_r2** has the same parameters as **rrdc_create** with two added parameters of; `sources` and `template`.

where `template` is the file path to a RRD file template, with, the form defined in **rrdcreate**(1),

The `sources` parameter defines series of file paths with data defined, to prefill the RRD with. See **rrdcreate**(1) for more details.

rrd_client_flush(rrd_client_t *client, const char *filename)
rrdc_flush(const char *filename)

flush the currently RRD cached in the daemon specified via `filename`.

rrd_client_forget(rrd_client_t *client, const char *filename)
rrdc_forget(const char *filename)

Drop the cached data for the RRD file specified via `filename`.

rrdc_flush_if_daemon(const char *daemon_addr, const char *filename)

Flush the specified RRD given by `filename` only if the daemon `daemon_addr` is up and connected.

rrd_client_fetch(rrd_client_t *client, const char *filename, const char *cf, time_t *ret_start, time_t *ret_end, unsigned long *ret_step, unsigned long *ret_ds_num, char ***ret_ds_names, rrd_value_t **ret_data)

rrdc_fetch(const char *filename, const char *cf, time_t *ret_start, time_t *ret_end, unsigned long *ret_step, unsigned long *ret_ds_num, char ***ret_ds_names, rrd_value_t **ret_data)

Perform a fetch operation on the specified RRD Database given be `filename`, where `cf` is the consolidation function, `ret_start` is the start time given by unix epoch, `ret_end` is the endtime. `ret_step` is the step size in seconds, `ret_ds_num` the number of data sources in the RRD, `ret_ds_names` the names of the data sources, and a pointer to an rrd_value_t object to shlep the data.

rrdc_stats_get(rrd_client_t *client, rrdc_stats_t **ret_stats)
rrdc_stats_get(rrdc_stats_t **ret_stats)

Get stats from the connected daemon, via a linked list of the following structure:

```
struct rrdc_stats_s {
    const char *name;
    uint16_t type;
    #define RRDC_STATS_TYPE_GAUGE   0x0001
    #define RRDC_STATS_TYPE_COUNTER 0x0002
    uint16_t flags;
    union {
        uint64_t counter;
        double   gauge;
    } value;
    struct rrdc_stats_s *next;
};
typedef struct rrdc_stats_s rrdc_stats_t;
```

rrdc_stats_free(rrdc_stats_t *ret_stats)

Free the stats struct allocated via **rrdc_stats_get**.

SEE ALSO

rrcached(1) **rrdfetch**(1) **rrdinfo**(1) **rrdlast**(1) **rrdcreate**(1) **rrdupdate**(1) **rrdlast**(1)

AUTHOR

RRD Contributors <rrd−developers@lists.oetiker.ch>