



Linux Ubuntu 26.04 Manual Pages on command 'tracecmd_iterate_reset.3'

\$ man tracecmd_iterate_reset.3

LIBTRACECMD(3) libtracefs Manual LIBTRACECMD(3)

NAME

tracecmd_iterate_events, tracecmd_iterate_events_multi, tracecmd_follow_event,
tracecmd_follow_missed_events, tracecmd_filter_add, tracecmd_iterate_reset - Read events
from a trace file

SYNOPSIS

```
#include <trace-cmd.h>
```

```
int tracecmd_iterate_events(struct tracecmd_input *handle,  
                           cpu_set_t *cpus, int cpu_size,  
                           int (*callback)(struct tracecmd_input *,  
                                             struct tep_record *,  
                                             int, void *),  
                           void *callback_data);  
  
int tracecmd_iterate_events_multi(struct tracecmd_input **handles,  
                                  int nr_handles,  
                                  int (*callback)(struct tracecmd_input *,  
                                                    struct tep_record *,  
                                                    int, void *),  
                                  void *callback_data);  
  
int tracecmd_iterate_events_reverse(struct tracecmd_input *handle,
```

```

        cpu_set_t *cpus, int cpu_size,
        int (*callback)(struct tracecmd_input *,
                        struct tep_record *,
                        int, void *),
        void *callback_data, bool cont);

int tracecmd_follow_event(struct tracecmd_input *handle,
                        const char *system, const char *event_name,
                        int (*callback)(struct tracecmd_input *,
                                        struct tep_event *,
                                        struct tep_record *,
                                        int, void *),
                        void *callback_data);

int tracecmd_follow_missed_events(struct tracecmd_input *handle,
                                int (*callback)(struct tracecmd_input *,
                                                struct tep_event *,
                                                struct tep_record *,
                                                int, void *),
                                void *callback_data);

struct tracecmd_filter *tracecmd_filter_add(struct tracecmd_input handle,
                                           const char *filter_str, bool neg);

int *tracecmd_iterate_reset(struct tracecmd_input *handle);

```

DESCRIPTION

This set of APIs can be used to iterate over events after opening a trace file using one of the open functions like `tracecmd_open(3)` or `tracecmd_open_fd(3)`.

The function `tracecmd_iterate_events()` will iterate through all the events in the trace file defined by `handle`, where `handle` is returned from one of the `tracecmd_open(3)` functions. It will call the `callback()` function on the events on the CPUs defined by `cpus`. The `cpu_size` must be the size of `cpus` (see `CPU_SET(3)`). If `cpus` is `NULL`, then `cpu_size` is ignored and `callback()` will be called for all events on all CPUs in the trace file. The `callback_data` is passed to the `callback()` as its last parameter. `callback` may be `NULL`, which is useful if `tracecmd_follow_event()` is used, but note if `callback` is `NULL`, then `callback_data` is ignored

and not sent to the callback of `tracecmd_follow_event()`.

The function `tracecmd_iterate_events_multi()` is similar to `tracecmd_iterate_events()` except that it allows to iterate over more than one trace file. If `tracecmd_agent(1)` is used to get a trace file for both the host and guest, make sure that the host trace file is the first entry in `handles` and `tracecmd_iterate_events_multi()` will do the synchronization of the meta data for the guest files that come later in `handles`. `handles` is an array of trace file descriptors that were opened by `tracecmd_open(3)` and friends. Note, unlike `tracecmd_iterate_events()`, `tracecmd_iterate_events_multi()` does not filter on CPUs, as it will cause the API to become too complex in knowing which handle to filter the CPUs on. If CPU filtering is desired, then the callback should check the `record?cpu` to and return 0 if it is not the desired CPU to process. `nr_handles` denotes the number of elements in `handles`. The `callback_data` is passed to the callback as its last parameter. `callback` may be NULL, which is useful if `tracecmd_follow_event()` is used, but note if `callback` is NULL, then `callback_data` is ignored and not sent to the callback of `tracecmd_follow_event()`.

The function `tracecmd_iterate_events_reverse()` works pretty much the same way as `tracecmd_iterate_events()` works, but instead of calling the `callback()` function for each event in order of the timestamp, it will call the `callback()` function for each event in reverse order of the timestamp. If `cont` is false, it will start by calling the event with the oldest timestamp in the `trace.dat` file. If `cont` is set to true, then it will start wherever the current position of the tracing data is. For instance, if the `callback()` return something other than zero it will exit the iteration. If `tracecmd_iterate_events_reverse()` is called again with `cont` to true it will continue where it left off. If `cont` is false, it will start again at the event with the oldest timestamp. The `handle`, `cpus`, `cpu_size`, and `callback_data` act the same as `tracecmd_iterate_events()`.

The `callback()` for both `tracecmd_iterate_events()`, `tracecmd_iterate_events_reverse()` and `tracecmd_iterate_events_multi()` is of the prototype:

```
int callback()(struct tracecmd_input *handle, struct tep_record *record, int cpu, void
*data);
```

The handle is the same handle passed to `tracecmd_iterate_events()` or the current handle of handles passed to `tracecmd_iterate_events_multi()` that the record belongs to. The record is the current event record. The cpu is the current CPU being processed. Note, for `tracecmd_iterate_events_multi()` it may not be the actual CPU of the file, but the nth CPU of all the handles put together. Use `record?cpu` to get the actual CPU that the event is on.

The `tracecmd_follow_event()` function will attach to a trace file descriptor handle and call the callback when the event described by system and name matches an event in the iteration of `tracecmd_iterate_events()` or `tracecmd_iterate_events_multi()`. Note, the cpu is the nth CPU for both `tracecmd_iterate_events()` and `tracecmd_iterate_events_multi()`. If the actual CPU of the record is needed, use `record?cpu`. For `tracecmd_iterate_events_multi()`, the callback is only called if the handle matches the current trace file descriptor within handles. The `callback_data` is passed as the last parameter to the `callback()` function. Note, this `callback()` function will be called before the `callback()` function of either `tracecmd_iterate_events()` or `tracecmd_iterate_events_multi()`.

The `callback()` prototype for `*tracecmd_follow_event()` is:

```
int callback()(struct tracecmd_input *handle, struct tep_event *event, struct tep_record *_record, int cpu, void *data);
```

The `tracecmd_follow_missed_events()` function will attach to a trace file descriptor handle and call the callback when missed events are detected. The event will hold the type of event that the record is. The record will hold the information of the missed events. The cpu is the nth CPU for both `tracecmd_iterate_events()` and `tracecmd_iterate_events_multi()`. If the CPU that the missed events are for is needed, use `record?cpu`. If `record?missed_events` is a positive number, then it holds the number of missed events since the last event on its CPU, otherwise it will be negative, and that will mean that the number of missed events is unknown but missed events exist since the last event on the CPU. The callback and `callback_data` is the same format as `tracecmd_follow_event()` above. The missed events callback is called before any of the other callbacks and any filters that were added by `tracecmd_filter_add()` are ignored. If callback returns a non zero, it will stop the iterator before it calls any of the other iterator callbacks for the given record.

The `tracecmd_filter_add()` function, adds a filter to handle that affects both `tracecmd_iterate_events()` and `tracecmd_iterate_events_multi()`. The `filter_str` is a character string defining a filter in a format that is defined by `tep_filter_add_filter_str(3)`. If `neg` is true, then the events that match the filter will be skipped, otherwise the events that match will execute the `callback()` function in the iterators.

The `tracecmd_iterate_reset()` sets the handle back to start at the beginning, so that the next call to `tracecmd_iterate_events()` starts back at the first event again, instead of continuing where it left off.

RETURN VALUE

Both `tracecmd_iterate_events()`, `tracecmd_iterate_events_reverse()` and `tracecmd_iterate_events_multi()` return zero if they successfully iterated all events (handling the follow and filters appropriately). Or an error value, which can include returning a non-zero result from the `callback()` function.

`tracecmd_iterate_reset()` returns 0 on success and -1 if an error occurred. Note, if -1 is returned, a partial reset may have also happened.

EXAMPLE

```
#define _GNU_SOURCE

#include <sched.h>

#include <stdlib.h>

#include <getopt.h>

#include <trace-cmd.h>

struct private_data {
    int      cpu;
    const char *file;
};
```

```
static int print_events(struct tracecmd_input *handle, struct tep_record *record, int cpu, void *data)
```

```

{
    static struct trace_seq seq;

    struct tep_handle *tep = tracecmd_get_tep(handle);
    struct private_data *pdata = tracecmd_get_private(handle);

    /* For multi handles we need this */
    if (pdata->cpu >= 0 && pdata->cpu != record->cpu)
        return 0;

    if (!seq.buffer)
        trace_seq_init(&seq);

    trace_seq_reset(&seq);
    trace_seq_printf(&seq, "%s: ", pdata->file);
    tep_print_event(tep, &seq, record, "%6.1000d [%03d] %s-%d %s: %s\n",
                    TEP_PRINT_TIME, TEP_PRINT_CPU, TEP_PRINT_COMM, TEP_PRINT_PID,
                    TEP_PRINT_NAME, TEP_PRINT_INFO);
    trace_seq_terminate(&seq);
    trace_seq_do_printf(&seq);
    return 0;
}

```

```

static int print_event(struct tracecmd_input *handle, struct tep_event *event,
                      struct tep_record *record, int cpu, void *data)
{
    return print_events(handle, record, cpu, data);
}

```

```

static int missed_events(struct tracecmd_input *handle, struct tep_event *event,
                        struct tep_record *record, int cpu, void *data)
{
    if (record->missed_events > 0)
        printf("CPU [%03d] has %d missed events\n",

```

```

        record->cpu, record->missed_events);

    else

        printf("CPU [%03d] has missed events\n", record->cpu);

    return 0;
}

static void usage(const char *argv0)
{
    printf("usage: [-c cpu][-f filter][-e event] %s trace.dat [trace.dat ...]\n",
        argv0);
    exit(-1);
}

int main(int argc, char **argv)
{
    struct tracecmd_input **handles = NULL;
    const char *filter_str = NULL;
    const char *argv0 = argv[0];
    struct private_data *priv;
    cpu_set_t *cpuset = NULL;
    char *event = NULL;
    size_t cpusize = 0;
    int nr_handles = 0;
    int cpu = -1;
    int i;
    int c;

    while ((c = getopt(argc, argv, "c:f:e:")) >= 0) {
        switch (c) {
            case 'c':
                /* filter all trace data to this one CPU. */
                cpu = atoi(optarg);
                break;

```

```

    case 'f':
        filter_str = optarg;

        break;

    case 'e':
        event = optarg;

        break;

    default:
        usage(argv0);
    }
}

argc -= optind;
argv += optind;

if (argc == 0)
    usage(argv0);

for (i = 0; i < argc; i++) {
    handles = realloc(handles, sizeof(*handles) * (nr_handles + 1));
    if (!handles)
        exit(-1);
    handles[nr_handles] = tracecmd_open(argv[i], 0);
    if (!handles[nr_handles]) {
        perror(argv[i]);
        exit(-1);
    }
    if (filter_str) {
        if (tracecmd_filter_add(handles[nr_handles], filter_str, false) == NULL) {
            perror("adding filter");
            exit(-1);
        }
    }
}

priv = calloc(1, sizeof(*priv));
if (!priv)

```

```

        exit(-1);

priv->file = argv[i];

priv->cpu = cpu;

tracecmd_set_private(handles[nr_handles], priv);

if (event) {
    if (tracecmd_follow_event(handles[nr_handles], NULL, event, print_event, NULL) < 0) {
        printf("Could not follow event %s for file %s\n", event, argv[i]);
        exit(-1);
    }
}

tracecmd_follow_missed_events(handles[nr_handles], missed_events, NULL);

nr_handles++;
}

```

/* Shortcut */

```

if (nr_handles == 1) {
    if (cpu >= 0) {
        cpuset = CPU_ALLOC(cpu + 1);
        if (!cpuset)
            exit(-1);

        cpusize = CPU_ALLOC_SIZE(cpu + 1);
        CPU_SET_S(cpu, cpusize, cpuset);
    }

    if (event)
        tracecmd_iterate_events(handles[0], cpuset, cpusize, NULL, NULL);
    else
        tracecmd_iterate_events(handles[0], cpuset, cpusize, print_events, NULL);
} else {
    if (event)
        tracecmd_iterate_events_multi(handles, nr_handles, NULL, NULL);
    else
        tracecmd_iterate_events_multi(handles, nr_handles, print_events, NULL);
}

```

```

    for (i = 0; i < nr_handles; i++) {
        priv = tracecmd_get_private(handles[i]);
        free(priv);
        tracecmd_close(handles[i]);
    }
    free(handles);
}

```

FILES

trace-cmd.h

Header file to include in order to have access to the library APIs.

-ltracecmd

Linker switch to add when building a program that uses the library.

SEE ALSO

libtracefs(3), libtraceevent(3), trace-cmd(1) trace-cmd.dat(5)

AUTHOR

Steven Rostedt <rostedt@goodmis.org[1]>

Tzvetomir Stoyanov <tz.stoyanov@gmail.com[2]>

REPORTING BUGS

Report bugs to <linux-trace-devel@vger.kernel.org[3]>

LICENSE

libtracecmd is Free Software licensed under the GNU LGPL 2.1

RESOURCES

<https://git.kernel.org/pub/scm/utils/trace-cmd/trace-cmd.git/>

COPYING

Copyright (C) 2020 VMware, Inc. Free use of this software is granted under the terms of the

NOTES

1. rostedt@goodmis.org

<mailto:rostedt@goodmis.org>

2. tz.stoyanov@gmail.com

<mailto:tz.stoyanov@gmail.com>

3. linux-trace-devel@vger.kernel.org

<mailto:linux-trace-devel@vger.kernel.org>

libtracefs

01/22/2026

LIBTRACECMD(3)