# S.Ha.R.K. User Manual

Scuola Superiore di Studi e Perfezionamento S. Anna
ReTiS Lab

Volume V

The S.Ha.R.K. Tracer

Written by
Paolo Gai (pj@sssup.it)

Original text by
Massimiliano Giorgi (massy@gandalf.sssup.it)


RETIS Lab.
Scuola Superiore S. Anna
Via Carducci, 40 - 56100 Pisa

16th December 2004

# Contents

# Chapter 1

# The S.Ha.R.K. Tracer

The S.Ha.R.K. tracer is a small module into the kernel that allow the system to write a log of all the meaningful events that happened during the execution. Typical events that can be traced are, for example, the scheduling decision made by the kernel, the hard disk head seeks, and so on.

## 1.1 Tracer Architecture

The Architecture of the tracer is composed by three subsystems (see Figure 1.1):

1. A set of callbacks inserted into the Kernel. These callbacks are always called during kernel execution. The tracer of course only work if the tracer support is compiled and linked into the application. Otherwise, the tracer callbacks become null functions, with a negligible overhead.
2. A module that exports general functionalities of the tracer; this part redefines the callbacks with the correct behavior, replacing the null functions compiled by default.
3. A set of optional modules that implement the event handling policies. Basically, the tracer uses a set of event queues, and each queue can use a different policy. It is a user responsibility to choose the kind and how many queues to use, and which are the events that have to be sent to each queue.

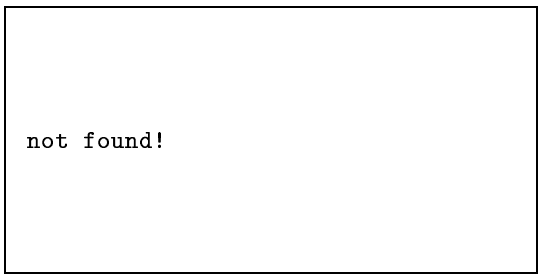


Figure 1.1: Tracer Architecture

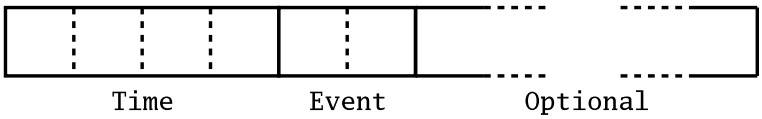


Figure 1.2: Tracer file format.

The tracer basically records events. An events is an information about something that happened at a given time. The tracer do not interpret the semantic of these events: it just records them. Every event is characterized by a time, a number, and by some additional parameters. Events can be masked (that is, whe an event fires, it is not recorded by the tracer).

For efficiency reasons events number are global to the whole system. If an application or the kernel have to increase the number of events, the file (*include/trace/types.h*) have to be modified. The numbers of the new events however have to be set between the given boundary, and event numbers have to be consecutive.

Events are divided in classes. Every class groups all the events with similar meaning. Classes are used by the tracer to simplify event management: basically all the events can be masked/unmasked with a single function call.

Currently, there are five classes:

**tracer** These are a set of events reserved by the tracer for internal use.

**system** These events are generated by the Generic Kernel or by the modules. For example, task creation, end, activation, scheduling, and so on.

**user** This event class can be used by the application. They are not generated by the Kernel.

**ll** This class only contains the low level event "interrupt", that is generated at each interrupt request.

**semaph** This class contains all the semaphore events: wait, signal, and so on.

## 1.2 Implementation

### 1.2.1 Tracer file format

In a way independent with event queue handling, the event informations are saved somewhere (at system shutdown the queues can be saved on the disk or sent through the network during execution). After that, they can be analyzed.

Figure 1.2 shows the format of every event. A trace file is simply a finite sequence of these records. Every event is characterized by three fields:

1. The time when the event was recorded (32 bit big endian, in microseconds).
2. The event number (16 bit).
3. A set of optional informations. This field has always the same size on the disk.

Please note that given the definitionof the `trc_allevents_t` structure, a tracer event has always the same size. That means also that every application that will use data files produced by the tracer will have always to consider the size of an event structure including the

**Algorithm 1** An event data structure

```
/* generics event */
typedef union TAGtrc_allevents {
  trc_tracer_event_t trc;
  trc_system_event_t sys;
  trc_user_event_t usr;
  trc_ll_event_t ll;
  trc_sem_event_t sem;
} trc_allevents_t;

/* event struct */
typedef struct TAGtrc_event_t {
  u_int32_t time;
  u_int16_t event;
  trc_allevents_t x;
} trc_event_t;
```

`include/tracer/types.h` file and using `sizeof()`. See the tracer examples (into `demos/tracer/*`) for more details.

Table 1 shows the C structure used to store an event (see `include/trace/types.h`).

**Implementing a trace analyzer**

Implementing a trace analyzer is quite simple:

1. Include the file `include/trace/types.h`, after having defined the types `u_int8_t`, `u_int16_t`, `u_int32_t` (that are unsigned integers at 8, 16, 1nd 32 bits).
2. Read the trace file storing the data into `trc_event_t` structures.
3. Use these structures to know what happened.

As an example, `demos/tracer/utils/tdump.c` and `demos/tracer/utils/jdump.c` read a trace file and print it on the standard output or to a JTracer File. the JTracer is a small Java graphical application that can be used to visualize the trace files.

These two programs use another file called `util.c`, that exports some utility functions like:

**read_trace** This function can read a trace file pissed as parameter. For each event found, a function is called. If that function return 0, the next event is processed, otherwise, read_trace terminates.

**event_class** This function find the event class of an event.

**event_name** This function find an event name and returns a meaningful (string) description.

### 1.2.2 Event queues

Currently, the following event handling policies have been implemented:

**dummy** A dummy queue simply discards all the events inserted in it. This module can be used as a template for the implementation of other policies. This module is contained into the files `include/tracer/qdummy.h` and into `kernel/modules/trcdummy.c`.

**Algorithm 2** Log functions.

```
int trc_logevent(int event, void *info);
int trc_suspend (void);
int trc_resume (void);
```

**fixed** This is a fixed queue whose dimension is defined at system startup. When the queue is full, all the events are discarded. There are two kind of fixed queues: one use the Shark filesystem, the other uses the DOSFS filesystem calls.

**circular** This is a fixed size queue handled in a circular way. In this case, data can be saved online during system execution or all the data can be saved at system shutdown. In the latter case, when the queue is full, a new event replaces the oldest one.

In the online queue, the data is saved using a server task: the events are stored into the queue; when the queue is full, events are discarded (until the queue becomes again empty); then, the server tries to write the data on a file during system execution. Of course, there is no guarantee, and events can be lost. Also note that the server task can produce an interference, specially if other tasks concurrently uses the file system.

**udp** All the events that arrives in this queue are sent using UDP to another PC. This module is still in development phase.

### 1.2.3 Expanding the tracer behavior

The tracer can be extended adding new event or extending the supported queue policies.

The tracer exports the following functions (see Table 2):

**trc_logevent** this is the main tracer function. When this function is called, an event is registered into its queue. `info` contains the optional parameters, or `NULL`.

**trc_suspend** This function suspend the tracer. it returns 0 in case of success, different from 0 otherwise. This function is called before staring system shutdown.

**trc_resume** This function resume the tracer (that was previously suspended using `trc_suspend`).

#### Creating new event classes

As introduced in section 1.1, events number are global to the whole system. To add a new event class or to modify an existing one some internal files have to be modified (the modifications are simple, don't worry).

Basically, the file `include/trace/types.h` have to be modified adding new identifiers and new data structures:

- New event identificators must be added after the existing ones. new class identifiers can be added also (solething like `TRC_CLASS_???`. Then, the constants `TRC_F_LAST`, `TRC_NUMCLASSES`, and `TRC_NUMEVENTS` have to be updated.
- If a new class is created, a new data structure whit the optional parameters should be created. An instance of that structure should be inserted into the union `trc_allevents_t`.
- The table named classtable should be modified (see the file `kernel/modules/trace.c`) to include the new class identifier.

```
typedef struct TAGtrc_queue_t {
  int type;
  trc_event_t *(*get)(void *);
  int (*post)(void *);
  void *data;
} trc_queue_t;

int trc_register_queuetype(
    int queuetype,
    int(*create)(trc_queue_t *,void *),
    int(*activate)(void *,int),
    int(*terminate)(void *)
    );
```

Table 1.1: The trc_queue_t structure.

In that way a new event class can be created. Of course, it is not guaranteed that the new trace files are compatible with the previous ones (it depends on the size of the optional part; the best thing to do is recompile all the system). Once a new event or a new event class is created, it can be used in all the system as the ones that where previously defined.

### Creating new tracer queue modules

Basically, each trace module has an init function that have to register the queue type in the system.

The function that is used to register a new queue is showed in Figure 1.1 together with the `trc_queue_t` structure. This function must be called during the initialization phase qith a number that specifies the type of the queue, ane three pointer to functions that are used by the tracer to implement the following behaviors:

**create** This is called when the user ask for the creation of a new queue of a given type. The structure passed as parameter must be filled with the correct values.

**activate** Called when the system is already in protected mode (typically, into the `__init__` task). A pointer to the queue data field is passed, together with an identification number.

**terminate** This is called before the system shuts down. A pointer to the queue data field is passed.

During the creation of a new queue, the create function is called. That function fills a descriptor structure of type `trc_queue_t`, that is passed as parameter. The field of that structure are used by the generic part of the tracer to implement its behavior. Here is a short description of these fields:

**type** Is the queue type.

**get** This function is called when the system have to handle a new event: it must retuen a pointer to a structure trc_event_t (the generic part will fill the structure with the event data). The function can return `NULL` if the queue can not handle the new event (e.g., the queue is full). A pointer to the generic data field is passed.

```
int trc_assign_event_to_queue(int event, int queue);
int trc_assign_class_to_queue(int class, int queue);
int trc_notrace_event(int event);
int trc_trace_event(int event);
int trc_notrace_class(int class);
int trc_trace_class(int class);
```

Table 1.2: Tracer utility functions.

```
int TRC_init_phase1(void);
int TRC_init_phase2(void);
int trc_create_queue(int queuetype, void *arg);
```

Table 1.3: Logging functions

**post** Thios function is called when the generic part of the tracer filled the data structure returned by the get function. This function should store the event somewhere (i.e., the data structure can be sent to another PC, stored in memory, or in a file).

**data** Pointer to a generic data structure that contains all the informations that allow to handle the particular queue behavior.

Note that the function get and post are called into the kernel at interrupt disabled. For that reason, they must be efficient, and they must not block.

## 1.3 Initialization and use

The tracer can be initialized calling a set of functions into the `__kernel_register_levels__` function and into the `__init__` task.

The functions in Table 1.2can be used to modify the behavior of event tracing:

**trc_assign_event_to_queue** This function assign an event to a particular queue. The queues can be identified by an handler returned by the function trc_create_queue. The first queue that is registered has index 0, the second 1, and so on.

**trc_assign_class_to_queue** This function assigns an event class to a queue. Calling trc_assign_event_to_queue on all the events of a class brings the same results.

**trc_notrace_event and trc_trace_event** These functions enable or disable the tracing for the events passed as parameters.

**trc_notrace_class and trc_trace_class** These functions enable and disable the tracing for all the events of an event class.

Tracer initialization is done using the functions in Table 1.3 and 1.2:

1. First, the function `TRC_init_phase1` is called during `__kernel_register_levels__`.
2. Then, the queue types used must be registered into the kernel.
3. Then, the events that must be logged must be enabled and assigned to the queues.

```
typedef struct TAGtrc_fixed_queue_args_t {
  char *filename;
  int size;
} TRC_FIXED_PARMS;

#define trc_fixed_default_parms(m)
#define trc_fixed_def_filename(m,s)
#define trc_fixed_def_size(m,s)

int trc_register_fixed_queue(void);
int trc_register_dosfs_fixed_queue(void);
```

Table 1.4: The fixed queue.

4. Finally, the function `TRC_init_phase2` is called just before `__call_main__` into the `__init__` task.

The function `trc_create_queue` is called passing the queue type that must be created , and a pointer to some specific parameters (`NULL` if the defaults can be used). The function returns a negative number in case of error, or a positive number or 0 in case of success (and that number is the handle of the queue ti be used with the functions in Table 1.2).

Here are the functions that have to be used to register a particular event queue:

### Dummy queue

The function `trc_register_dummy_queue` do not have optional parameters. It always returns `0`.

### Fixed queue

Table 1.4 shows the functions (that uses the filesystem or the DOSFS functions) and the optional data structure that can be used to initialize a fixed queue. A set of macro is also available to manipulate the optional data structure:

**trc_fixed_default_parms** Stores the default values into the parameter. The filename is set to NULL and the queue dimension is set to 8192..

**trc_fixed_def_filename** Sets the name used for the file wher the trace have to be saved. If NULL is used, the name fix followed by a number is used.

### Circular queue

As described in section1.2.3, the circular queue exists in two flavours. In the first case the queue is saved at system shutdown, in the second cades the queue is saved online during system execution.

Table 1.5 shows the data structure that can be used during queue creation; also in that case, the structure can be manipulated using the provided macros:

**trc_circular_default_parms** Set the default parameters.

**trc_circular_def_filename** Defines the file name where the trace must be saved. If `NULL`, a default filename is used.

```
typedef struct TAGtrc_circular_queue_args_t {
  char *filename; int size;
  long period;
  long slice;
  int flags;
} TRC_CIRCULAR_PARMS;

#define trc_circular_default_parms(m)
#define trc_circular_def_filename(m,s)
#define trc_circular_def_size(m,s)
#define trc_circular_def_onlinetask(m)
#define trc_circular_def_period(m,p)
#define trc_circular_def_slice(m,s)

int trc_register_circular_queue(void);
```

Table 1.5: The circular queue

```
typedef struct TAGtrc_udp_queue_args_t {
  int size;
  UDP_ADDR local,remote;
  TASK_MODEL *model;
} TRC_UDP_PARMS;

#define trc_udp_default_parms(m,local,remote)
#define trc_udp_def_size(m,size)
#define trc_udp_def_local(m,local)
#define trc_udp_def_remote(m,remote)
#define trc_udp_def_model(m,model)

int trc_register_udp_queue(void);
```

Table 1.6: The UDP queue

**trc_circular_def_size** Set the queue size (8192 by default).

**trc_circular_def_onlinetask** If this function is called, a flag is set, and the queue is saved online.

**trc_circular_def_period** This is the period of the task that is used to save the queue online. By default, it is 500 milliseconds. A soft task is used by default.

**trc_circular_def_slice** This is the mean execution time of the task. By default, it is set to 25 milliseconds.

### UDP queue

The udp queue is basically a circular queue. The queue is filled when an event arrives, and is flushed by a dedicated periodic task that sends maximum one packet every period. The packets are sent using UDP, and the network driver must be configured and running *before* calling the tracer initialization phase 2.

```
int TRC_init_phase1_standard(void);
int TRC_init_phase2_standard(void);
```

Table 1.7: Standard tracer initialization

You can use the `udpdump` demo into `demos/tracer/utils` to read the UDP packets sent by this module.

Table 1.6 shows the data structure that can be used during queue creation; also in that case, the structure can be manipulated using the provided macros:

**trc_udp_default_parms** Set the default parameters. Source and destination IP/ports must also be provided

**trc_udp_def_size** Set the queue size (8192 by default).

**trc_udp_def_local, trc_udp_def_remote** These functions can be used to set the local or remote IP.

**trc_udp_def_model** Use this function to give a custom task model to be used for the tracer task. The default model is obtained using the following initialization (the argument to the task is set by the module):

```
SOFT_TASK_MODEL model;
soft_task_default_model(model);
soft_task_def_system(model);
soft_task_def_periodic(model);
soft_task_def_period(model,250000);
soft_task_def_met(model,10000);
soft_task_def_wcet(model,10000);
```

### 1.3.1 Standard Initialization

If you get bored of all these functions, you can use a "standard" tracer initialization. The first function must be called into the `__kernel_register_levels__` (as soon as possible); the second function must be called into the `__init__` task, just before calling the `__call_main__` function.

The standard initialization creates two queues. Queue 0 is a circular queue, queue 1 is fixed. All the system events are then sent to queue 0.