RTL POSIX Trace 1.0
(a POSIX Trace System in RT-Linux)

by
Andrés Terrasa, Ana García-Fornes and Agustín Espinosa
{aterrasa,agarcia,aespinos}@dsic.upv.es

September 10, 2002

Contents

RTL POSIX Trace 1.0 (a POSIX Trace System in RT-Linux)

{aterrasa,agarcia,aespinos}@dsic.upv.es
Departamento de Sistemas Informáticos y Computación
Technical University of Valencia
Cno. de Vera s/n, 46022 Valencia (SPAIN)

Notes:

I hope the information contained in this document will be useful for those people interested in using the POSIX Trace support in RT-Linux.

RT-Linux (Real-Time Linux) is a trademark of Finite State Machine Labs, Inc. (www.fsmlabs.com). Open RT-Linux can be used free of royalties under the terms of the Open RTLinux Patent License Version 2.0 (see file PATENT in the Open RT-Linux distribution for details).

"POSIX" and "1003.1" are registered trademarks of the Institute of Electrical and Electronic Engineers, Inc.

The software described in this document, that is, the POSIX Trace system integrated into Open RT-Linux (RTL POSIX Trace 1.0), is distributed under the terms of the GNU's General Public License version 2 (www.gnu.org).

Please report any suggestion, comment or error to my e-mail address above.

A. Terrasa

Foreword

This document assumes that you already know about the POSIX Trace standard. If not, we advise you to take a look at the document entitled “A Summary of the POSIX Trace Standard” first.

Although you are most probably familiar with RT-Linux and how real-time applications are implemented on it, the following is a (very) brief description of the RT-Linux programming philosophy. We hope this will help you understand some of the design decisions adopted in the implementation of the POSIX Trace support. If you already know how to implement RT-Linux applications, you may perfectly skip it.

RT-Linux is a real-time kernel mainly focussed on implementing small hard real-time applications which need tightly bounded interrupt latencies and a fully predictable system behavior. In addition, the programmer can have full advantage of having a general-purpose, full-featured operating system as Linux available (but only running when RT-Linux allows it to do so)

From the programmer point of view, RT-Linux has been progressively taking the external form of the POSIX Minimal Realtime System Profile (MRSP) over the years. The MRSP is the most restricted profile defined in POSIX for real-time operating systems. This profile describes the facilities required to support small, embedded, hard real-time applications. The hardware requirements of this profile include only one processor, no explicit memory protection, no mass storage and, in general, simple hardware devices operated synchronously. The software requirements include only the execution of *one* process (with complete POSIX thread support), but without the need of a file system or user interaction.

Since RT-Linux has adopted the MRSP model (although the support is still incomplete), it only supports one “process” to be running at a time[1], on which many POSIX threads (real-time tasks) can be created. From the RT-Linux viewpoint, Linux and all its user processes are globally seen as an additional real-time task running in the background (at lowest priority). This means that Linux processes can still execute, but only when no real-time task wants to do so. In addition, real-time tasks can interact with Linux processes by using communication mechanisms such as shared memory and real-time “fifos”.

Overall, this model permits (and encourages) the programmer to split the real-time application into two parts: the code which actually needs hard real-time capabilities and the non-real-time code. Then, the real-time part (usually small and simple) is implemented as a set of real-time tasks at the RT-Linux level, while the non-real-time part (which can be as big and complex as needed) is implemented as one or more Linux processes.

[1] Actually, since RT-Linux supports multiprocessor (SMP) systems, the RT-Linux model assumes one running process per physical processor. However, this feature is not further considered, since our tracing system does not have multiprocessor support yet.

1 Introduction

The purpose of this document is to point out the characteristics and peculiarities of our POSIX Trace standard implementation in RT-Linux. This implementation has effectively integrated a POSIX Trace subsystem into the RT-Linux kernel. In particular, it has been carried out on the Open RT-Linux version 3.1. The trace system has been baptized "RTL POSIX Trace 1.0" (RTL-PT 1.0).

The trace system RTL-PT 1.0 entirely supports the Trace and the Trace Event Filter options defined in the standard, subject to some minor changes and limitations. Overall, this allows the programmer of a real-time application in RT-Linux to perform filtered on-line tracing of events at run time.

Conversely, RTL-PT 1.0 does not (so far) support neither the Trace Inheritance nor the Trace Log options. The reason for not supporting inheritance is straightforward, since the "target" process for RT-Linux (that is, the running real-time application) cannot *fork*. We may support trace into logs in some future version, but this is not trivial since the RT-Linux kernel does not have direct access to a standard file system in a permanent storage media (such as the computer's hard drive file systems available at the Linux level). This lack of file system access could be circumvented in a couple of ways[2], with either of them covering only *partially* the standard's requirements for the Trace Log option.

Following the RT-Linux philosophy (explained in the Foreword section), the trace system considers only *one* target process at a time (i.e., the running RT-Linux application), but potentially formed by both a set of RT-Linux tasks and any number of Linux processes. The implementation thus supports the tracing (generation) *and* retrieval of trace events from both the RT-Linux tasks and Linux processes *concurrently*. Furthermore, this target can be traced in up to `TRACE_SYS_MAX` number of streams simultaneously (this constant has currently its minimum value according to the standard, which is eight). All this is further explained in Section 2.

In a general-purpose system, one of the main benefits of having a trace system available is to test (and debug) applications from a functional point of view. That is, to check if a given application does what it is supposed to do. In a real-time system, however, it also becomes vital to know *when* the application is doing things, and *how much* these actions take to execute. This kind of temporal information should include the system overhead as well, in order to be accurate and useful. In this sense, RTL-PT 1.0 has introduced some system event types that automatically collect this type of information and make it available to analysis. A description of the system event types and its associated data can be found in Section 3. In addition, the application can optionally use a facility called "RTL kernel trace" which automatically collects and processes these new system events and generates some files with the recorded information. With one of this files, for example, a graphical trace of the application execution can be depicted by using a visor tool called Quivi[3]. The kernel trace facility is described in

[2] The first possibility is to send the trace events to a Linux process (by a RT-FIFO, for instance) and make *this process* store them into the file log. Or else, we could send the events through some port (e.g., serial, parallel, network, etc.) to another computer.

[3] Quivi is a trace displayer entirely implemented in Tcl-Tk by Agustín Espinosa. It is free software, under the terms of

Section 4.

During the implementation of the RTL-PT system, there have also been some other minor changes, adaptations and, in general, slight differences from the standard definition. All these are reported in Section 5.

Finally, Appendix A includes a full example of use, on which both a real-time task and a Linux process trace events into a stream, while another Linux process retrieves them. This example can also be found (and executed) in the directory `examples/posixtrace/rtl-and-linux`.

2 The Three Tracing Roles in RTL-PT

As described in the standard, there are three roles which are executed in any complete tracing activity: the trace controller process, the target or traced process and the trace analyzer process.

In a typical RT-Linux application, the programmer usually splits the application code in a set of real-time tasks (executed by the RT-Linux kernel) plus one or more Linux user processes (executed by the Linux kernel). These two parts will be hereafter referred to as the applications's "RT-Linux side" and "Linux side". If some trace support has to be given to the two sides, the RTL-PT system has to be present at both. Therefore, we have implemented RTL-PT as two cooperating trace subsystems, one at each level:

a) RT-Linux trace subsystem. The RTL-PT support at the RT-Linux level has been integrated into the RT-Linux scheduler (module `rtl_scheduler.o`). The trace support is then always available to the RT-Linux application, whether or not the application wants to use it. Nevertheless, its overhead in the case of not using it is practically null.

All the data structures necessary to keep the entire tracing status are created and managed inside this module. These data structure include, among other control information, the set of event types registered for the target and all the active streams with its currently stored events.

b) Linux trace subsystem. Maybe the most natural way to suport POSIX Trace at the Linux level could have been to modify the Linux kernel by adding the required facilities as new Linux system calls. However, we chose not to do this for two reasons: firstly, because then the support would have been completely coupled to a particular version of the Linux kernel, and secondly because the actual functionality to be supported (see below) did not actually require such ambitious implementation . As a result, the decision was to implement this subsystem as a *library* to be linked with any Linux process that required trace support (called `libposix_trace.a`). This library is made available in Linux when RT-Linux (with the RTL-PT system) is compiled. Internally, this library communicates with the RT-Linux scheduler (where the RT-Linux trace subsystem is) in order to make both systems work in a synchronized manner. This communication is done by several dedicated RT-FIFOs.

the the GPL, and can be found on `www.dsic.upv.es/~aespinos/quivi.html`.

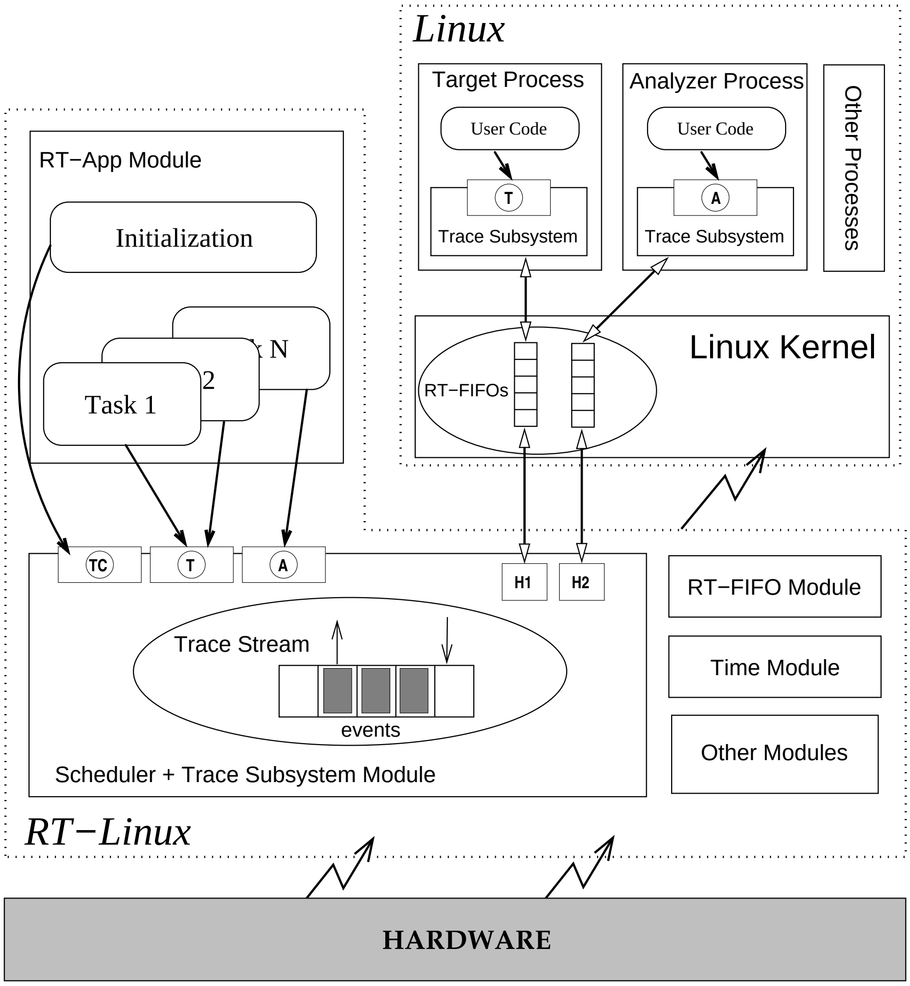

Figure 1: The RTL POSIX Trace 1.0 system.

Figure 1 depicts a complete vision of the RTL-PT system, showing both trace subsystems and their relationships with the different components of a real-time application. The figure shows all the possibilities of the RTL-PT system at once, including real-time tasks *and* Linux processes playing the target and trace analyzer roles. In this figure, the application RT-Linux side is labelled RT-App Module, while the application Linux side is formed by the Linux processes labelled Target Process and Analyzer Process. The figure is further described in the following subsections, where the three trace roles are discussed. The review of the roles is done in terms of who can play each of them in RT-Linux according to the RTL-PT system (as explained above, "who" means the real-time tasks, or a Linux process, or both).

2.1 Trace Controller Process

The process playing the trace controller process (TCP) role has the ability to create one or more trace streams in order to trace certain target process. After creating a stream, the TCP has complete control on its behavior (among other actions, it can start and stop the stream, set the filter of event types, clear the stream, shut it down, etc.).

In RTL-PT, the set up of the stream(s) on which the real-time application will be traced corresponds naturally to the RT-Linux side of the application. In particular, the actions related with the TCP will normally be executed at the *initialization* stage of the RT-Linux side (in the `init_module` function). An exception of this is the case of the stream shutting down, which will usually be done at the application's exit (inside the `cleanup_module` function). Any of the TCP functions can also be called by a running real-time task, although this should not the case for many of them[4].

Thus, if any RT-Linux application wants to trace events using the RTL-PT support, the initialization function of the module containig the application has to perform, at least, these actions:

1. A call `posix_trace_attr_init` in order to initialize a stream attribute object. If any attribute has a default value which is not appropriate for the application, then the corresponding function should be invoked to correct it.

2. A call `posix_trace_create` in order to create a stream with the stream attribute object arranged above. This call *must* provide a 0 as the function's first argument, indicating that we are going to trace the current process. If successful, a stream identifier is returned.

   If the application needs to be traced in more than one stream, then this step has to be repeated as many times as necessary. The upper limit of this is currently 7, since one stream is reserved to the kernel trace facility (see Section 4).

3. If any of the created streams requires filtering, then the application needs to create an empty (or full) event type set, add (or remove) individual event types as required, and then apply the set to the appropriate stream by calling `posix_trace_set_filter`.

4. A call `posix_trace_start` in order to make the stream to start recording events. Do this with any stream created.

The following is the full list of the functions corresponding to the TCP which are supported in RTL-PT 1.0. (this particular subset of the trace functions is represented as TC in Figure 1):

```
int  posix_trace_attr_destroy(trace_attr_t *);
int  posix_trace_attr_getclockres(const trace_attr_t *,
          struct timespec *);
int  posix_trace_attr_getcreatetime(const trace_attr_t *,
          struct timespec *);
int  posix_trace_attr_getgenversion(const trace_attr_t *, char *);
int  posix_trace_attr_getmaxdatasize(const trace_attr_t *restrict,
          size_t *restrict);
int  posix_trace_attr_getmaxsystemeventsize(const trace_attr_t *restrict,
```

[4]Some of the trace functions, for example the one that creates an stream, produces significant overhead. This overhead is not a problem when the application is being initialized, but it can be inacceptably high at run time.

```
        size_t *restrict);
int  posix_trace_attr_getmaxusereventsize(const trace_attr_t *restrict,
        size_t, size_t *restrict);
int  posix_trace_attr_getname(const trace_attr_t *, char *);
int  posix_trace_attr_getstreamfullpolicy(const trace_attr_t *restrict,
        int *restrict);
int  posix_trace_attr_getstreamsize(const trace_attr_t *restrict,
        size_t *restrict);
int  posix_trace_attr_init(trace_attr_t *);
int  posix_trace_attr_setmaxdatasize(trace_attr_t *, size_t);
int  posix_trace_attr_setname(trace_attr_t *, const char *);
int  posix_trace_attr_setstreamsize(trace_attr_t *, size_t);
int  posix_trace_attr_setstreamfullpolicy(trace_attr_t *, int);
int  posix_trace_clear(trace_id_t);
int  posix_trace_create(pid_t, const trace_attr_t *restrict,
        trace_id_t *restrict);
int  posix_trace_eventid_equal(trace_id_t, trace_event_id_t,
        trace_event_id_t);
int  posix_trace_eventid_get_name(trace_id_t, trace_event_id_t, char *);
int  posix_trace_eventset_add(trace_event_id_t, trace_event_set_t *);
int  posix_trace_eventset_del(trace_event_id_t, trace_event_set_t *);
int  posix_trace_eventset_empty(trace_event_set_t *);
int  posix_trace_eventset_fill(trace_event_set_t *, int);
int  posix_trace_eventset_ismember(trace_event_id_t,
        const trace_event_set_t *restrict, int *restrict);
int  posix_trace_eventtypelist_getnext_id(trace_id_t,
        trace_event_id_t *restrict, int *restrict);
int  posix_trace_eventtypelist_rewind(trace_id_t);
int  posix_trace_get_attr(trace_id_t, trace_attr_t *);
int  posix_trace_get_filter(trace_id_t, trace_event_set_t *);
int  posix_trace_get_status(trace_id_t,
        struct posix_trace_status_info *);
int  posix_trace_set_filter(trace_id_t, const trace_event_set_t *, int);
int  posix_trace_shutdown(trace_id_t);
int  posix_trace_start(trace_id_t);
int  posix_trace_stop(trace_id_t);
int  posix_trace_trid_eventid_open(trace_id_t, const char *restrict,
        trace_event_id_t *restrict);
```

2.2 The Traced Process

In RTL-PT 1.0, the target or traced process (TP) is always composed by a set of real-time tasks executed by the RT-Linux scheduler and, optionally, some Linux user processes. The RTL-PT system provides *both* levels with the two functions which the standard defines to this role. These functions are:

```
int  posix_trace_eventid_open(const char *restrict,
        trace_event_id_t *restrict);
void posix_trace_event(trace_event_id_t,
        const void *restrict, size_t);
```

The support for the TP at the RT-Linux level is provided by the RT-Linux scheduler module (represented as T at the upper part of the Scheduler + Trace subsystem Module box in Figure 1).

At the Linux level, the TP functions are provided by the `libposix_trace.a` library, which has to be linked with any target Linux process. The TP support provided by the library is represented as the T symbol inside the Linux Target Process in Figure 1. Inside the library, these two TP functions use a private RT-FIFO in order to communicate with the RT-Linux trace subsystem, which is the one that actually stores the information. Each time a message is written to the FIFO at the Linux side, a handler in the RT-Linux scheduler wakes up (this handler is depicted as H1 in the figure). When triggered, the handler checks the message in order to retrieve which function was called from Linux (and its arguments) and then invokes the equivalent function of the RT-Linux trace subsystem. Many Linux processes can use these functions simultaneously, effectively using the same FIFO. Obviously, this mechanism does not work if the RT-Linux scheduler module is not loaded when the Linux TP calls these functions. If the RT-Linux side has not already created the stream(s), or if it has shut them down, then calling to these two functions from Linux has no effect.

If the TP role is being executed by different *entities* (namely, the set of real-time tasks and some Linux processes), it is possible for any of them to register a new event type by calling the function `posix_trace_eventid_open`. The identifier returned by the call can be subsequently used by this entity to trace events of this type. But, as the registration is actually performed in a single place (the RT-Linux trace subsystem) for all the entities, this registration becomes *global* for all these entities forming the TP. This means that one entity can actually trace events of a type which was originally registered by another entity. For example, if the application real-time side has registered a new event type named "`evtype1`" and then a Linux process does the same, the Linux process will be returned the same internal identifier for the event type that the real-time side received. Then, both entities can actually trace events of the same type.

2.3 The Trace Analyzer Process

Since RTL-PT does not support the Trace Log option, the only way of retrieving events is to do so while the stream(s) on which the application is being traced are active. This is the role of the trace analyzer process (TAP).

In RTL-PT, the TAP support is provided by the two trace subsystems, which allows any real-time task or any Linux process to retrieve events from a given stream. At the RT-Linux level, the TAP functions are provided by the module containing the RT-Linux scheduler (symbolized as A above this module in Figure 1). In Linux, the TAP functions are provided by the `libposix_trace.a` library. In the figure, this is represented as the A symbol inside the Linux Analyzer process.

In this sense, it is worth noting that although it is technically possible to have many *entities* playing the TAP role simultaneously (e.g., many real-time tasks and/or Linux processes), this not probably a good idea unless each of them is retrieving events from a *different* stream. If, for example, two real-time tasks (or a real-time task and a Linux process) are retrieving events from the *same* stream, then each event in the stream will randomly be retrieved by one entity or the other (depending on which one asks for its retrieval first). In the end, the events retrieved by each entity will probably have to be merged, and all of them are re-ordered, in order to extract any useful information. For this reason, having only one TAP entity per trace stream is normally a better practice.

The full list of functions available for the TAP at either of the application levels is now shown. Please note that all of them, except the last three ones, are also on the list in Section 2.1; this is because these functions correspond to capabilities that both TCP and TAP roles share:

```
int  posix_trace_attr_getclockres(const trace_attr_t *,
         struct timespec *);
int  posix_trace_attr_getcreatetime(const trace_attr_t *,
         struct timespec *);
int  posix_trace_attr_getgenversion(const trace_attr_t *, char *);
int  posix_trace_attr_getmaxdatasize(const trace_attr_t *restrict,
         size_t *restrict);
int  posix_trace_attr_getmaxsystemeventsize(const trace_attr_t *restrict,
         size_t *restrict);
int  posix_trace_attr_getmaxusereventsize(const trace_attr_t *restrict,
         size_t, size_t *restrict);
int  posix_trace_attr_getname(const trace_attr_t *, char *);
int  posix_trace_attr_getstreamfullpolicy(const trace_attr_t *restrict,
         int *restrict);
int  posix_trace_attr_getstreamsize(const trace_attr_t *restrict,
         size_t *restrict);
int  posix_trace_eventid_equal(trace_id_t, trace_event_id_t,
         trace_event_id_t);
int  posix_trace_eventid_get_name(trace_id_t, trace_event_id_t, char *);
int  posix_trace_eventtypelist_getnext_id(trace_id_t,
         trace_event_id_t *restrict, int *restrict);
int  posix_trace_eventtypelist_rewind(trace_id_t);
int  posix_trace_get_attr(trace_id_t, trace_attr_t *);
int  posix_trace_get_status(trace_id_t,
         struct posix_trace_status_info *);
int  posix_trace_getnext_event(trace_id_t,
         struct posix_trace_event_info *restrict , void *restrict,
         size_t, size_t *restrict, int *restrict);
int  posix_trace_timedgetnext_event(trace_id_t,
         struct posix_trace_event_info *restrict, void *restrict,
         size_t, size_t *restrict, int *restrict,
         const struct timespec *restrict);
int  posix_trace_trygetnext_event(trace_id_t,
         struct posix_trace_event_info *restrict, void *restrict, size_t,
         size_t *restrict, int *restrict);
```

At the RT-Linux level, it is straightforward for a real-time task to retrieve an event from any given stream, just by using any of the three retrieval functions at the bottom of the list. This is because the stream identifier (which is a parameter of the three functions) is a known variable within the module containing the application RT-Linux side. According to the standard, this is the expected behavior in on-line tracing, where the TCP and the TAP are actually the same process. Therefore, the usual practice will be to create a TAP real-time task per active trace stream, so that each task is responsible of retrieving the events of that stream (or else, the same real-time task can sequentially retrieve events from different streams).

However, if we want a *Linux process* to retrieve events from a given trace stream, then there is a serious problem, because the internal identifier of *that* particular stream is not known outside the application RT-Linux module. Imagine, for example, that we want three Linux processes to retrieve and analyze events from three different streams. The programmer needs a way, at implementation time, to distinguish from these streams, although their actual stream identifiers will not be known until run time (and only at the RT-Linux level). We had to slightly change the behavior established by the standard in order to solve this. Indeed, only the expected *behavior* was changed; the interface remains intact. All this is now explained.

The way both sides of the application can agree, at implementation time, to use a particular stream is done by name. We take advantage that stream have names (it is one of the attributes in the stream attribute object) in the following way:

1. The application RT-Linux side has to give certain name to the stream when the stream is created. The header file `trace.h` provides *public* names for the eight possible streams that may be created.

   Inside the creation function (`posix_trace_create`), the RTL trace subsystem creates a private RT-FIFO which is associated *exclusively* with that stream name. All these FIFOs created for supporting TAP in Linux share a common RT-Linux *handler* (the H2 in Figure 1), which executed whenever a Linux process writes some data to one of the FIFOs.

2. A Linux process that wants to retrieve events from that stream has first to get a stream identifier that points to that particular stream. This is done by "creating" the stream again inside the Linux process with the same public name it was used in the first step.

   The `posix_trace_create` function is also used here, but with different semantics: the stream name is used to extract the FIFO name that is associated with the stream; then the function checks whether or not the FIFO has already been created by the RT-Linux side. If so, this FIFO becomes the private link between the Linux process and that particular stream. Otherwise, the function fails.

3. The internal stream identifier returned by the creation function in the previous step can then be used by the Linux process in the three alternative retrieval function. All three functions work exactly as described in the standard.

4. Since the streams are shutdown at the RT-Linux level, a mechanism is needed to inform an analyzer Linux process when this happens. This has been carried out by following the standard definition of the retrieval functions: the `[EINVAL]` error has to be returned by any of the three functions if "the trace stream identifier was invalid". In Linux, this happens when the stream has been shutdown from the RT-Linux side.

3 New System Event Types Defined in RTL-PT

The POSIX Trace standard defines, for the combination of the Trace and Trace Event Filter options, the predefined system events in the following table:

Event Type Name	Event Type Constant	Additional Data	Description
posix_trace_error	POSIX_TRACE_ERROR	int error	This event reports that an error related to the trace system occurred.
posix_trace_start	POSIX_TRACE_START	trace_event_set_t event_filter	This event is reported each time the stream starts tracing events, returning also the filter currently active for the stream.
posix_trace_stop	POSIX_TRACE_STOP	int auto	This event is reported each time the stream stops tracing events, returning also whether the stop was automatic or application-initiated.
posix_trace_filter	POSIX_TRACE_FILTER	trace_event_set_t old_event_filter, new_event_filter	This event is reported each time the application changes the filter of the stream, returning both the old and the new filters.
posix_trace_overflow	POSIX_TRACE_OVERFLOW	(none)	This event is reported at the beginning of an overflow situation (some events are overwritten in the stream, and thus they are lost).
posix_trace_resume	POSIX_TRACE_RESUME	(none)	This event is reported right before the first "valid" event after an overflow situation (that is, when the overflow situation has ended).
posix_trace_unnamed _userevent	POSIX_TRACE_UNNAMED _USEREVENT	(none)	This event is reported when the user tries to register a new event type and the per-process maximum amount of user event types (TRACE_USER_EVENT_MAX) has been reached.

Table 1: System event types predefined by the POSIX Trace standard.

In addition to these, RTL-PT 1.0 has added the following ones, mainly with the purpose of retrieving system information that can be useful for the real-time programmer. Please note that future versions of RTL-PT may modify the event types (syntactically or semantically); check file `<trace.h>` for any change. All these events are automatically recorded to any active stream which has not explicitly filtered them out:

POSIX_TRACE_CONTEXT_SWITCH :

Event type name: posix_trace_context_switch.

When it is reported: Every time a context switch is produced between real-time tasks.

Associated data:

```
typedef struct {
  void             *task;          // The new running task
```

```
  int              priority;      // Its priority
  task_state_t     state;
  bool             is_linux;      // Whether this task is Linux or not
} posix_trace_task_state_event_data;
```

POSIX_TRACE_TASK_STATE :

Event type name: posix_trace_task_state.

When it is reported: Every time a real-time task changes its state (released, ended, ready, suspended).

Associated data:

```
typedef struct {
  void             *task;         // The task the state is being reported
  int              priority;      // Its priority
  task_state_t     state;         // The task's new state
  bool             is_linux;      // Whether this task is Linux or not
} posix_trace_task_state_event_data;
```

POSIX_TRACE_KERNEL_OVERHEAD :

Event type name: posix_trace_kernel_overhead.

When it is reported: Every time the RT-Linux scheduler starts or stops running. In particular, each time the *schedule function* is invoked.

Associated data:

```
typedef struct {
  kernel_overhead_t type;         // OVERHEAD_START or OVERHEAD_END
  void             *task;         // The task that was running
  bool              is_linux;     // Whether this task is Linux or not.
} posix_trace_overhead_event_data;
```

POSIX_TRACE_SYSTEM_CALL :

Event type name: posix_trace_system_call.

When it is reported: Every time some system calls are invoked from a real-time tasks. Only some "interesting" system calls are actually instrumented (such as pthread_create, clock_nanosleep, pthread_kill, pthread_mutex_lock, etc.).

Associated data:

```
typedef struct {
  syscall_types_t  syscall;       // The invoked syscall's identifier
  void             *task;         // The invoking task
  bool             is_linux;      //  Whether this task is linux or not
} posix_trace_syscall_event_data;
```

POSIX_TRACE_MUTEX_LOCK / POSIX_TRACE_MUTEX_UNLOCK :

Event type name: posix_trace_mutex_lock, posix_trace_mutex_unlock.

When it is reported: Every time the related functions are called, including the mutex identifier and, in the case of the lock function, the result of the invocation: lock granted, lock denied (and hence blocking the task), or lock tried (if the "try" function has been invoked and the lock is not granted.)

Associated data:

```
typedef struct {
  mutex_op_t       op;             // The operation
  void             *mutex;         // The mutex id
  void             *task;          // The invoking task id
} posix_trace_mutex_event_data;
```

POSIX_TRACE_COND_WAIT / POSIX_TRACE_COND_BROADCAST :

Event type name: posix_trace_cond_wait, posix_trace_cond_broadcast.

When it is reported: Every time the related functions are called, including the identifiers of the mutex and condition variable involved, and the invoking task.

Associated data:

```
  cond_op_t        op;             // Operation: wait, signal, broadcast
  void             *task;          // The invoking task id
  void             *cond;          // The condition variable id
  void             *mutex;         // The mutex id
} posix_trace_cond_event_data;
```

4 The Kernel Trace Facility

The kernel trace facility uses some of the event types reviewed in the last section in order to generate human-friendly information of application executions.

This facility is formed by two separate components (the kernel module `rtl_ktrace.o`) and the Linux program `ktrace_analyzer` which work coordinately:

a) `rtl_ktrace.o`. When loaded, this module creates a trace stream and sets its filter so that only system events are accepted. The stream is created suspended, as usual, but the module does not start it at first. When the module is unloaded, it shuts this stream down.

   The module exports two functions, named `rlt_ktrace_start` and `rlt_ktrace_stop`, which internally starts and stops the stream, respectively. By using them, the application RT-Linux side can start and stop this automatic collection of system events whenever necessary.

   In RTL-PT 1.0, this module becomes a part of the RT-Linux module set, so it is started automatically with the other modules when the user types `rtlinux start` in a shell.

b) `ktrace_analyzer`. This program runs as a Linux process. It accesses the stream created by the `rtl_ktrace` module and retrieves the events as they are recorded. This program has to be running in the background while the real-time application is executing, since all the recorded events are lost when the `rtl_ktrace` module exits.

   The program retrieves the events of the stream above and generates two trace files on which the sequence of observed events is dumped: the format of the first file (with a `tra` extension) is appropriate to be interpreted by a graphical trace display tool called Quivi, while the format of the second file (with a `dbg` extension) is textual. Figure 2 shows a screen capture of one trace file displayed in Quivi.

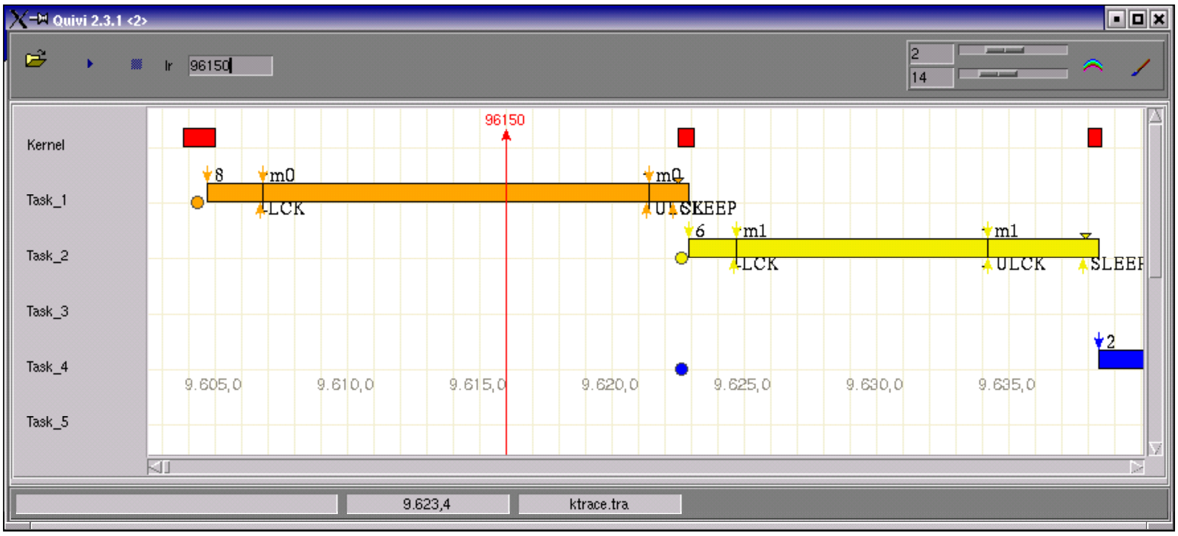

Figure 2: A snapshot of the Quivi tool displaying a trace generated by the Kernel Trace facility.

Given that the RTL-PT system automatically traces all system events and stores them in all the active trace streams on which they are not explicitly filtered out, the extra overhead of the kernel trace facility is very low. Nevertheless, if the application does not want to use this facility, it may either not loading the `rlt_ktrace` module or simply not invoking the `rlt_ktrace_start` function from the application RT-Linux side module.

5 Some Other Minor Changes

This section lists the differences between the standard definition and the implementation of the RTL-PT which have not been reported in other sections of this document.

5.1 Header Files

Some constants and basic types are defined in header files which are not the ones established by the standard. In particular: all the limit constants (such as `TRACE_SYS_MAX` or `TRACE_NAME_MAX`) should be placed in header file `limits.h` and all the private (opaque) data types (as `trace_id_t` or `trace_event_id_t`) should be placed in `sys/types.h`.

In both cases, all these types and constants have been defined in `trace.h`. The reason for this is that in RT-Linux many header files are duplicated (one version for RT-Linux and another for Linux) in such a way that some header files are identical in both sides and some other are different. This makes the entire header system so complex that we thought that, for the sake of clarity, this shortcoming could be reasonable. As a result, any RT-Linux application or Linux program that wants to use the trace support of RTL-PT has to include this global header file only.

5.2 Full Event Sets

When the application wants to have a set full of event types, it uses `posix_trace_eventset_full`, indicating one of these options: `POSIX_TRACE_WOPID_EVENTS`, `POSIX_TRACE_SYSTEM_EVENTS` or `POSIX_TRACE_ALL_EVENTS`. The first one is defined as the set of “all the process-independent, implementation-defined system trace event types”, the second is the set formed by “all the implementation-defined system trace event types” and the last one is the set formed by “all the trace event types”.

RTL-PT 1.0 considers that all system events are directly or indirectly related to *the* target process (and probably also to some of its tasks) and therefore the `POSIX_TRACE_WOPID_EVENTS` event set is considered void.

A An Example

The example this section is describing is one of the sample applications that RTL-PT 1.0 provides in the `examples/posixtrace` directory. Any of these examples can be executed by typing "`make test`" in a shell located in the example's directory.

In particular, this example features a "complex" application formed by several real-time tasks and a Linux process (called `target_process`). Both the real-time tasks and the target Linux process define new user event types and then trace events to a single stream, which is set to filter out all system events. The events are retrieved by an additional Linux process called `linux_analyzer`. This process just retrieves the events and write them to the standard output. The real-time module and both Linux processes are now explained in detail, showing excerpts of the actual C files.

A.1 The `rtl_tasks.c` Module

The application RT-Linux side is placed in this module called `rtl_tasks.c`. Inside this module, the initialization stage contains the creation of a stream attribute object, its modification, the creation of a stream and its filtering set up, and finally, the creation of four periodic real-time tasks. Among these four, only one actually traces events, while the other just consume CPU and uses some system resources (a mutex). This is how this function looks like:

```
int init_module(void) {

  trace_attr_t      attr;
  pthread_attr_t    thattr;
  trace_event_set_t set;
  int               error;

  // Start the automatic tracing of kernel events:
  rtl_ktrace_start();

  // Create and set the trace attribute:
  posix_trace_attr_init(&attr);
  posix_trace_attr_setstreamfullpolicy (&attr, POSIX_TRACE_UNTIL_FULL);
  posix_trace_attr_setname(&attr, TRACE_STREAM1_NAME);

  // Create the stream:
  error = posix_trace_create(0, &attr, &trid);
  if (error) {
    return -1;
  }

  // Set the stream filter to only record user events:
  posix_trace_eventset_fill(&set, POSIX_TRACE_SYSTEM_EVENTS);
  posix_trace_set_filter(trid, (const trace_event_set_t *) &set,
                              POSIX_TRACE_SET_EVENTSET);
```

```
  // Start tracing:
  posix_trace_start(trid);

  // Create the task which traces user events (awakes each 100 msec):
  pthread_attr_init (&thattr);
  pthread_create (&thr1, &thattr, writer, 0);
  pthread_make_periodic_np(thr1, 0, (hrtime_t) 100000000);

  // And other tasks which just consume cpu

  // This one awakes each 20 msec:
  pthread_create (&thr2, &thattr, just_execute, (void *) 100000);
  pthread_make_periodic_np(thr2, 0, (hrtime_t) 20000000);

  // This one awakes each 25 msec:
  pthread_create (&thr3, &thattr, just_execute, (void *) 300000);
  pthread_make_periodic_np(thr3, 0, (hrtime_t) 25000000);

  // This one awakes each 50 msec:
  pthread_create (&thr4, &thattr, just_execute, (void *) 200000);
  pthread_make_periodic_np(thr4, 0, (hrtime_t) 50000000);

  return 0;
}
```

The real-time task executing the `writer` function first registers three new event types (which are called “user_event_string”, “user_event_char” and “user_event_int”) and then enters into a loop, which executes one iteration each time the task is periodically released. In each loop, the task traces an event of each of these user event types. Two of the three events are traced each time with different associated data, while the third always traces the same data. The `writer` function is now shown:

```
void *writer(void *dummy) {
  int   i;
  int   j;
  char  s[32];
  char  c;
  int   k;
  void *data;

  // Create some new event types:
  posix_trace_eventid_open ("user event string", &ev_string);
  posix_trace_eventid_open ("user event char",  &ev_char);
  posix_trace_eventid_open ("user event int",   &ev_int);

  // Initialize data to be recorded along with the events
  c = 'A';
  k = 0;
  //s = "RT-Linux says: Hello world!";
  strcpy(s, "RT-Linux says: Hello world!");
```

```
  for (i=0; i<50; i++) {

    pthread_mutex_lock(&mutex);

    // Trace the "user event char" event and consume some CPU:
    data = (void *) & c;
    posix_trace_event(ev_char, data, sizeof(char));
    c += 1;
    for (j=0; j<40000; j++) ;

    // Trace the "user event int" event and consume some CPU:
    data = (void *) & k;
    posix_trace_event(ev_int, data, sizeof(int));
    k += 1;
    for (j=0; j<40000; j++) ;

    // Trace the "user event string" event and consume some CPU:
    posix_trace_event(ev_string, s, sizeof(s));
    for (j=0; j<40000; j++) ;

    pthread_mutex_unlock(&mutex);

    // Go to sleep until next period:
    pthread_wait_np();
  }
  return (void *) 0;
}
```

Finally, the module's cleanup function shuts the stream down, destroy the tasks and exits:

```
void cleanup_module(void) {

  rtl_printf("rtl_tasks: CLEANUP!!!\n");

  // Stop and shutdown the stream:
  posix_trace_stop(trid);
  posix_trace_shutdown(trid);

  // Delete the tasks:
  pthread_delete_np(thr1);
  pthread_delete_np(thr2);
  pthread_delete_np(thr3);
  pthread_delete_np(thr4);

  // Stop the automatic tracing of kernel events:
  rtl_ktrace_stop();
}
```

A.2 The `linux_target.c` Program

This program is supposed to form part of the application, and thus also traces event into the stream(s) which the RT-Linux might have created. In particular, this program register two event types called "user_event_linux" and "user_event_string". The former is unknown to the trace system and then it is registered as a new event type. However, the latter has already been created by the RT-Linux side so in this case, the register function in Linux returns the already mapped identifier for that event. This shows how RT-Linux and Linux can effectively trace events which belong to the same event type.

After registering the two event types, the program just enters in a loop in which traces the two events and then sleeps for a while.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <trace.h>

#define NTIMES 30

void timer_handler(int dummy) {
  return;
}


int main(void) {
  int               ntimes;
  trace_event_id_t  ev_linux;
  trace_event_id_t  ev_string;
  void              *data;
  char              s[32];
  struct itimerval  timeout;
  struct sigaction  action;

  // Create a new type of event:
  posix_trace_eventid_open ("user event linux", &ev_linux);

  // Register an event type which has been created in RT-Linux
  posix_trace_eventid_open ("user event string", &ev_string);
  strcpy(s, "Linux says: Hello world!");

  // Initialize the timeout value (oneshot, 500 ms):
  timeout.it_interval.tv_sec  = 0;
  timeout.it_interval.tv_usec = 0;
  timeout.it_value.tv_sec     = 0;
  timeout.it_value.tv_usec    = 500000;

  sigfillset(&(action.sa_mask));
  sigdelset(&(action.sa_mask), SIGALRM);
  action.sa_handler = timer_handler;
  sigaction(SIGALRM,&action,NULL);
```

```
  // Enter the main loop: trace this event and sleep for a timeout:
  for (ntimes = 0; ntimes < NTIMES; ntimes ++) {

    data = (void *) & ntimes;
    posix_trace_event(ev_linux, data, sizeof(int));
    posix_trace_event(ev_string, s, sizeof(s));

    setitimer(ITIMER_REAL,&timeout,NULL);
    pause();
  }
  return 0;
}
```

A.3 The linux_analyzer.c Program

This program is the trace analyzer process for the application. It basically extracts all the possible information out of the stream created at the RT-Linux side. As explained above (Section 2.3) the way a Linux analyzer process can be attached to a particular stream is by "re-creating" it using the public name of the stream (defined in "trace.h).

The program "creates" the stream, extracts and displays the stream attributes, the set of registered event types and the stream status, and then enters a loop in which waits for an event to be available in the stream. When available, it displays all its information, including its associated traced data, and then waits again for the next event.

```
#include <stdio.h>
#include <string.h>
#include <trace.h>

#define MAX_DATA_SIZE 256

int main(void) {
  char         data[MAX_DATA_SIZE];
  char         str[TRACE_NAME_MAX];
  char         s[64];
  size_t       data_len;
  size_t       size;
  int          unavailable;
  int          pol;
  int          error;
  trace_event_id_t ev;
  trace_attr_t trace_attr;
  trace_id_t   trace_id;
  struct posix_trace_event_info event;
  struct posix_trace_status_info status;
  struct timespec  time;
  int   *i;
  char  *c;
```

```
  // Create (open) **the same** trace stream already created in
// rtl_tasks.c
posix_trace_attr_init(&trace_attr);
posix_trace_attr_setname(&trace_attr, TRACE_STREAM1_NAME);
error = posix_trace_create(0,&trace_attr,&trace_id);

if(error) {
  fprintf(stderr,"ERROR %d while creating stream %s\n", error,
  TRACE_STREAM1_NAME);
  exit(1);
}

// Retrieve the attributes of this stream and print all the info:
error = posix_trace_get_attr(trace_id, &trace_attr);
fprintf(stderr, "get attr (%d)\n", error);

error = posix_trace_attr_getgenversion(&trace_attr, str);
fprintf(stderr, "get genversion (%d): %s\n", error, str);

error = posix_trace_attr_getname(&trace_attr, str);
fprintf(stderr, "get name (%d): %s\n", error, str);

error = posix_trace_attr_getcreatetime(&trace_attr, &time);
fprintf(stderr, "get create time (%d): %ld.%ld\n", error,
time.tv_sec, time.tv_nsec);

error = posix_trace_attr_getclockres(&trace_attr, &time);
fprintf(stderr, "get clock res (%d): %ld.%ld\n", error,
time.tv_sec, time.tv_nsec);

error = posix_trace_attr_getstreamfullpolicy(&trace_attr, &pol);
fprintf(stderr, "get full policy (%d): %d\n", error, pol);

error = posix_trace_attr_getmaxusereventsize(&trace_attr, 0, &size);
fprintf(stderr, "get max user event size (%d): %d\n", error, size);

error = posix_trace_attr_getmaxsystemeventsize(&trace_attr, &size);
fprintf(stderr, "get max system event size (%d): %d\n", error, size);

error = posix_trace_attr_getmaxdatasize(&trace_attr, &size);
fprintf(stderr, "get max data size (%d): %d\n", error, size);

error = posix_trace_attr_getstreamsize(&trace_attr, &size);
fprintf(stderr, "get stream size (%d): %d\n", error, size);

// Print the full list of event types available in the stream:
error = posix_trace_eventtypelist_rewind(trace_id);
fprintf(stderr, "event list rewind (%d)\n", error);

error = posix_trace_eventtypelist_getnext_id (trace_id, &ev, &unavailable);
while( ! unavailable && ! error) {
```

```
    error = posix_trace_eventid_get_name (trace_id, ev, str);

    if (! error) {
      fprintf(stderr, "event getname (%d): %s\n", error, str);
    }
    // Get the next one:
    error = posix_trace_eventtypelist_getnext_id (trace_id, &ev,
  &unavailable);
  }

  // Print the current status of the stream :
  error = posix_trace_get_status(trace_id, &status);
  fprintf(stderr, "event get status (%d): %d %d %d \n", error,
  status.posix_stream_status, status.posix_stream_full_status,
  status.posix_stream_overrun_status);

   // Start retrieving events while the stream is open:
  fprintf(stderr, "\n\nNow retrieving events: \n");

  while(1) {

    error = posix_trace_getnext_event(trace_id, &event, &data, sizeof(data),
      &data_len, &unavailable);
    if(error) {
      fprintf(stderr,"No more events (%d). Exiting\n", error);
      exit(0);

    } else if (unavailable) {
      fprintf(stderr, "   Event unavailable\n");

    } else {
      fprintf(stderr, " ** New event ----------------------------------\n");
      fprintf(stderr, " Event identifier = %d\n", event.posix_event_id);
      fprintf(stderr, " Process ID       = %d\n", event.posix_pid);
      fprintf(stderr, " Address          = %x\n", (int) event.posix_prog_address);
      fprintf(stderr, " Truncation status= %d\n", event.posix_truncation_status);
      fprintf(stderr, " Timestamp        = %ld.%9ld\n", event.posix_timestamp.tv_sec,
                                                  event.posix_timestamp.tv_nsec);
      fprintf(stderr, " Thread ID        = %d\n",   (int) event.posix_thread_id);

      posix_trace_eventid_get_name (trace_id, event.posix_event_id, str);
      fprintf(stderr, " Event Name       = %s\n", str);

      // Now switch depending on the event type (name):
      if (!strcmp(str,"user event char")) {
        c = (char *) data;
        fprintf(stderr, " Data (char)      = %c\n",  *c);
      }

      else if (!strcmp(str,"user event int")) {
        i = (int *) data;
```

```
        fprintf(stderr, " Data (int)       = %d\n",  *i);
      }

      else if (!strcmp(str,"user event string")) {
        strcpy(s, data);
        fprintf(stderr, " Data (string)    = %s\n",  s);
      }

      else if (!strcmp(str,"user event linux")) {
        i = (int *) data;
        fprintf(stderr, " Data (int)       = %d\n",  *i);
      }

      else {
        fprintf(stderr, " Data unknown\n" );
      }

      fprintf(stderr, " Data length      = %d\n",  data_len);
      fprintf(stderr, " -----------------------------------------------\n\n");
    }

  }
  return 0;
}
```