Lightweight POSIX Trace

A POSIX-based trace system for Minimal Realtime systems

Andrés Terrasa, Technical University of Valencia (UPV)
Agustín Espinosa, Technical University of Valencia (UPV)
Ana García-Fornes, Technical University of Valencia (UPV)

December 2003

Initial note: This document contains the official description of the LPTRACE component developed in the OCERA Project (http://www.ocera.org). Although the LPTRACE support will be a native part of the OCERA RT-Linux kernel, you can also add it to a regular RT-Linux (3.2-pre1) distribution without any other OCERA components.

1. Summary

Name	Lightweight POSIX Trace (**LPTRACE**)
Description	This component adds (most of) the tracing support defined in the POSIX Trace standard to RTLinux. The POSIX Trace standard defines a set of portable interfaces for tracing applications.
Author/s	Andres Terrasa, Agustin Espinosa, Ana Garcia-Fornes.
Reviewer	
Layer	Low level RTLinux and Linux
Version	2.0
Status	Stable
Dependencies	RTLinux 3.2-pre1
Release Date	M2

2. Description

Our experience in implementing the POSIX Trace (Ptrace) component in the first stage of the project, as well as some feedback from users (and partners), resulted in two main conclusions: first, that tracing mechanisms are very useful to the real-time application designer, specially when debugging and tuning the application; and second, that the POSIX Trace standard is probably too big and complex for small real-time operating systems like RT-Linux.

According to these conclusions, two different alternatives could be followed: either design and implement a custom (non-standard) tracing mechanism, or to redefine the POSIX Trace standard in order to adequate it to the requirements (and restrictions) of small kernels.

The decision was made to keep the POSIX path, adapting the POSIX Trace standard not just to RT-Linux but to the POSIX Realtime profiles (in special, to the Minimal Realtime System Profile or MRSP). In this way, our results are applicable to all kernels following these profiles, including RT-Linux.

Thus, this component presents two parts: a proposal of how the POSIX Trace standard can be subdivided and optimized to be suitable for MRSP kernels, and an implementation of that proposal in RT-Linux, along with an exhaustive performance analysis of the resulting system.

2.1. The MRSP Model and its Tracing Limitations

The MRSP is the most restricted of the real-time profiles defined by POSIX. This profile is intended to specify the minimum hardware and software requirements for small, embedded, fully reliable applications with strict, hard guarantees. The hardware requirements include one processor, no explicit memory protection (that is, no memory management unit), no mass storage devices and, in general, simple hardware devices operated synchronously. The software requirements establish a simple programming model in which the real-time system is executed by only one process (with complete POSIX thread support) without the need of a file system or user interaction.

Since hardware and software requirements in MRSP systems are very strict, applications for these systems are normally developed in a host/target manner. The complete MRSP system (kernel plus application) is developed in the host machine, then uploaded in some way to the target platform and finally executed there. Further interaction between both systems is assumed to be produced only through some kind of communication link, since this is the only device type required in the target. Even in RT-Linux, which is a bit peculiar in this aspect, we can consider the "target" to be the RT-Linux executive and the "host" to be Linux, despite the fact that both systems share the same hardware.

Both the hardware/software model and the host/target development scheme present several restrictions to the POSIX Trace philosophy. These restrictions can be grouped in two categories, related to two characteristics of the MRSP model:

1. *Single process model*. In this group we find three limitations: (1) the three roles defined in the trace standard must be executed by the only process in the system, although it is likely that each role will be played by a different thread, or set of threads, inside this process; (2) the functionality related with the Trace Inheritance implementation option does not have to be supported, since there will never be many simultaneous processes to trace; and (3) since there is only one possible target process, the list of user trace events is unique for the entire process (and shared by all the different streams that may be created to trace this process).

2. *Lack of file system*. This restrictions makes difficult to fully support the Trace Log option, since the "log" is defined to be a persistent object (which naturally corresponds to the concept of a disk file). The only way to support this functionality in a pure MRSP kernel is by associating the log file to a communication device (such as a serial or parallel port, a network card, etc.) which links the system with another computer, typically the host computer.

   However, even in this case, some aspects of the standard's Trace Log option cannot be fully supported: (1) the stream's full policy named `POSIX_TRACE_FLUSH` forces the trace system to automatically initiate a flushing operation to the log before the stream becomes full. In a real-time system, this may produce long delays at unpredictable times, and hence it is undesirable. And (2) if the log "file" is actually a simple communication device, then some access limitations arise: the device cannot be tested to be full and writing to the log can only be produced sequentially, adding data at the "end of the file". As a result, the only log full policy which can be implemented is `POSIX_TRACE_APPEND`.

In addition to these restrictions, there are also performance considerations to be taken into account, in both memory space and speed. Some of the standard's requirement, such as eight simultaneous trace streams, can easily be unacceptable for systems with memory restrictions. As another example, the temporal complexity of reporting some trace events could not be worthwhile in systems where the application and the kernel are small and well known. In situations like these, the standard can be too demanding for a small kernel.

The proposal below has taken all these factors into account in order to allow the operating system developer to tailor the trace subsystem to the real needs and limitations of the system, while maintaining almost all of the original POSIX Tracing philosophy, data structures, API, etc.

2.2. Proposal of a Lightweight POSIX Trace System

(**Previous note**: for obvious space reasons, this proposal does not contain a full explanation of the POSIX Trace standard. On the contrary, the reader is assumed to be familiar with it. If this is not the case, please refer to the "Description" section of the Ptrace component, in the Milestone 2.)

Maybe the easiest way to start describing this proposal is by explaining which parts of the original standard are **not** proposed to change:

- The general trace philosophy, based on two data structures (event and stream) and three trace roles (controller, target and analyzer) is maintained.
- All the functions in the standard's API remain sintactically (and, in most of the cases, semantically) unchanged.
- All the data structures (e.g., `posix_trace_status_info`) and the predefined values of their fields (e.g., `POSIX_TRACE_RUNNING`) are left unchanged.
- The set of system events related to the trace system itself (e.g., `POSIX_TRACE_START`, `POSIX_TRACE_STOP`, etc.) are equally defined.

Now that we know the aspects of the standard which will not be modified, let us explain the proposed changes. These changes are split in two categories: removal of restrictions and definition of "units of functionality".

2.2.1. Removal of Restrictions

The first set of changes intended by this proposal has to do with the the elimination of some general restrictions that the POSIX standard imposes to any conformant system. In other words, the proposal softens the standard's requirements in order to favor the efficiency of the trace system implementation, as well as to make this system closer to the needs and restrictions of the POSIX profiles.

In particular, this is done by allowing the operating system *not to* support some trace features required by the standard. In particular, the trace subsystem **may**:

a. **not support** the creation of several simultaneous streams, but only one at a time. (POSIX imposes a minimum of eight simultaneous trace streams.)

b. **limit** the maximum size of a trace stream that the application can create.

c. **limit** the maximum size of the data attached to an event traced by the application.

d. **not to report** some (or all) of the system events related to the trace system (such as `POSIX_TRACE_START`, `POSIX_TRACE_STOP`, etc.).

e. **not to support** all the "full policies" for streams and logs which are required by the standard (e.g., `POSIX_TRACE_LOOP`, `POSIX_TRACE_UNTIL_FULL`, etc.).

The removal of the first three restrictions allows the implementation of the trace subsystem to put an upper limit to the amount of memory devoted to tracing (instead of letting the application decide so), while the removal of the last two allows for a more efficient implementation of the tracing and retrieval of events.

It is important to point out that in all the five cases, the POSIX conformance is possible, even in a small kernel (in fact, the former Ptrace component supports them all). The proposal has just identified some key points in the original standard where it may be sensible to remove functionality in order to gain performance, taking into account the characteristics of MRSP systems.

2.2.2. Units of Functionality

The concept of "unit of functionality" is not new in POSIX. In particular, it is used in the definition of the real-time profiles. Sometimes, a profile requires *part* of the functionality of a POSIX standard, which was not originally defined to provide that part isolatedly. In such cases, the original standard is subdivided into a set of "units of functionality", which partition the standard's API. Once the units are defined, they can be indivudually selected to provide the profile the exact functionality that it requires.

This way of breakinig a standard down into small units is normally applied to old, monolithic standards, which were not originally defined with different implementation options. Although this is not the case of the POSIX Trace standard (it is defined in four implementation options), our opinion is that even the most basic implementation option contains functionality which may not be required in a typical MSRP system. According to this, we have used the same concept of "unit of functionality" in order to further partition the trace API into smaller

parts. The system can thus be provided with exact functionality it needs and be relieved from the overhead of parts it does not need.

In particular, the proposal has subdivided the API of the original POSIX Trace standard in the following seven units of functionality:

A. **Trace Core**. This unit contains the common trace support required by all the other units (but the last one). It cannot be provided standalone, but with, at least, one of the following two: "on-line trace" or "trace to log". This is because these two units, and not the "trace core", actually incorporate the means to *create* trace streams (without or with a log, respectively). The benefits of this organization are, first, that each interface function belongs one unit only, and second, that the system does not need to implement superfluous functions. (For example, according to the original standard, the trace system must incorporate the creation of active streams without a log, even when it may only want to support active streams with a log.).

   This unit of functionality includes functions for: (1) trace stream attribute manipulation (only those related with the standard's *Trace Event* option), (2) trace stream manipulation for active streams (except the ones for creating a stream), (3) event list retrieval, and (4) event type manipulation.

B. **On-Line Trace**. As its name implies, this unit contains the means to perform on-line tracing of events, at least of *system* events. This unit has to be incorporated along with, at least, the "trace core" unit.

   The "on-line trace" unit incorporates three function categories: (1) the creation of active trace streams without a log, (2) the retrieval of events from such streams, and (3) the retrieval of the stream attributes and the stream status.

C. **Trace To Log**. This option incorporates the means to trace events to an active trace stream with a log, but not to *retrieve* them. In small (possibly embedded) systems, the retrieval and analysis of events will not be done in the target but in the host computer. Thus, the retrieval of information from a log has been separated in a different unit, named "log retrieval" (see the last option in this list). As in the previous unit, the "trace to log" unit has to be incorporated along with, at least, the "trace core".

   The "trace to log" unit thus incorporates the functions to specifically manipulate active streams with a log and their attribute objects.

D. **User Events**. This unit allows the application to define new (user) event types and to trace them. This option can be supported along with either the "on-line trace" or the "trace to log" units, or both.

   The unit incorporates a single category of functions, the creation and tracing of user events.

E. **Event Filter**. This unit is equivalent to the implementation option named *Trace Event Filter* in the original standard, which allows for the dynamic filtering of events while the system is tracing events. This unit can be provided only if the "on-line trace" or "trace to log" units (or both) are supported.

   The "event filter" unit incorporates functions for (1) manipulating event filter sets, (2) applying/retrieving these filters to/from active trace streams (either with or without a log) and (3) mapping trace event names with event type identifiers.

F. **Trace Inheritance**. This unit is equivalent to its homonym implementation option in the original standard. It allows for the tracing of simultaneous target processes into the same stream(s). It can be supported only if the "on-line trace" or "trace to log" units (or both) are supported. (Nevertheless, MRSP systems cannot truly incorporate any trace inheritance, since in such systems there is only one process running.)

   This unit incorporates the functions for activating/consulting the inheritance attribute feature from the trace stream atttribute object.

G. **Log Retrieval**. This unit incorporates the functionality for opening log files into pre-recorded streams and to retrieve from such streams all the stored information (event types, status, events, etc.). This unit can be provided standalone and independently from all the others. Supporting this unit involves having a file system stored in non-volatile media and, for this reason, this unit will typically be implemented in the host computer only.

   This unit supports the manipulation of pre-recorded streams and their statuses, attributes and event types,

and the retrieval of events from such streams.

To sum up, the four implementation options of the POSIX Trace standard have been subdivided into seven units of functionality. Among these, the first six are intended to provide increasing tracing capabilities to the target system, while the last one will normally be incorporated to analyze, in the host system, the logs created by the target.

2.3. Implementation in RT-Linux

The **Lptrace** component has introduced six of the seven units of functionality to RT-Linux/Linux systems. In particular, the following units are available in RT-Linux: "trace core", "on-line trace", "trace to log", "user events" and "event filter". The unit "log retrieval" has been made available to Linux processes (as a user-level library), according to the host/target philosophy introduced in the proposal above. As a RT-Linux-specific feature (non-portable to other MRSP kernels), Linux processes can also trace events (if the "user events" unit is supported) into the same stream(s) created by a RT-Linux application. This allows the tracing of events by "applications" formed by cooperating Linux processes and RT-Linux tasks.

Compared to the first milestone's *Ptrace* component, the **Lptrace** has first increased the tracing capabilities of the RT-Linux kernel by adding the original standard's "Trace Log" implementation option. As a result, RT-Linux applications can now create streams with a log and trace events into them. Then, the complete support has been subdivided into the units of functionality proposed above, which can be individually selected when configuring the RT-Linux kernel (before its compilation).

The selection of the different units of functionality, as well as some of the restrictions mentioned in Section 2.2.1, “Removal of Restrictions”, are done in several configuration menus, which are now presented in the following figures.

Figure 1. Main configuration menu of POSIX Trace support

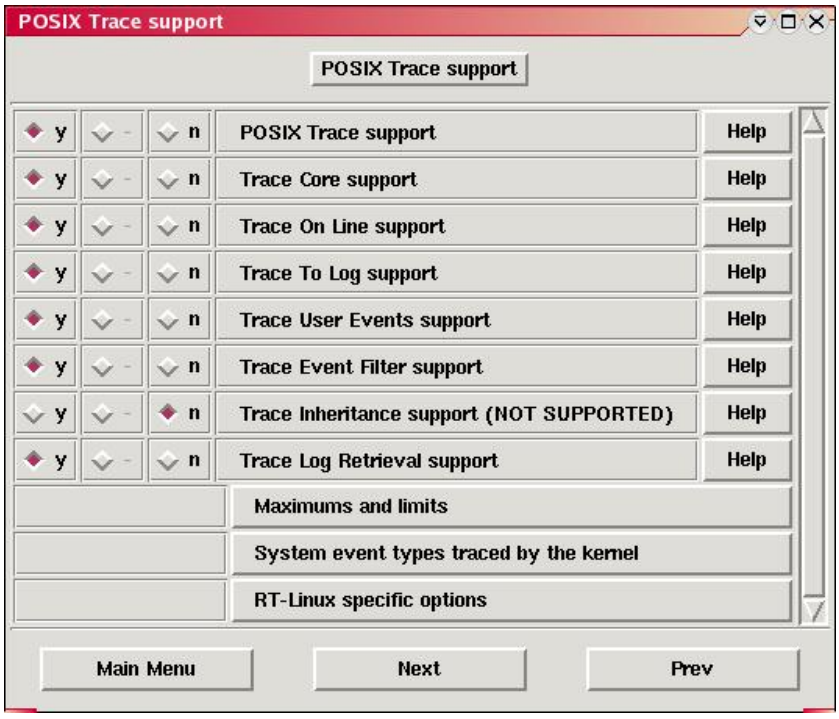


Figure 2. Maximums and limits menu

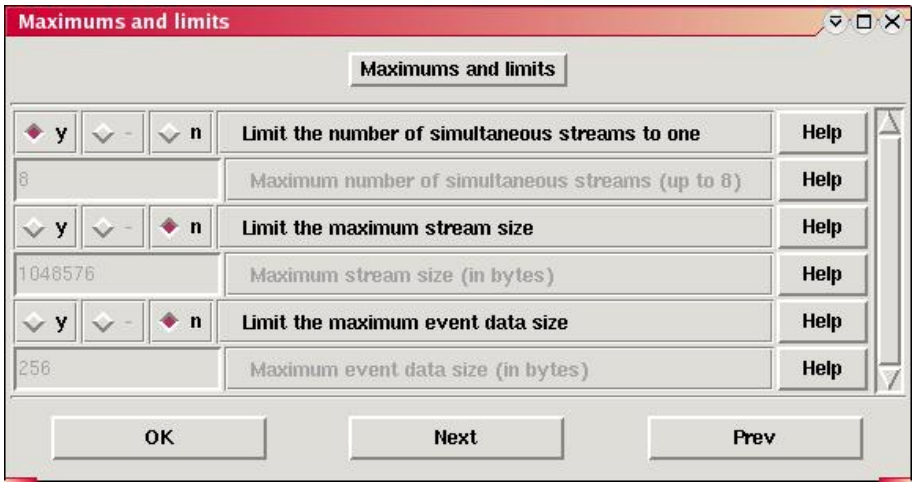


Figure 3. Selection of system event types

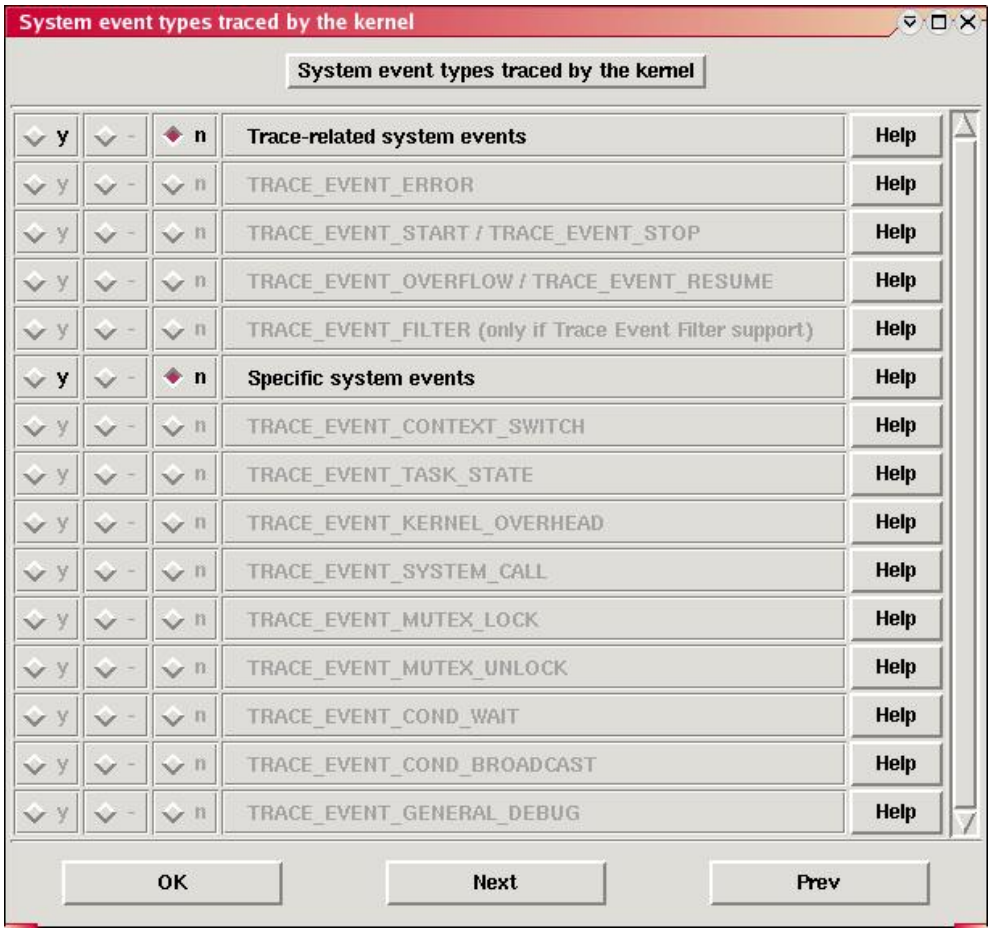
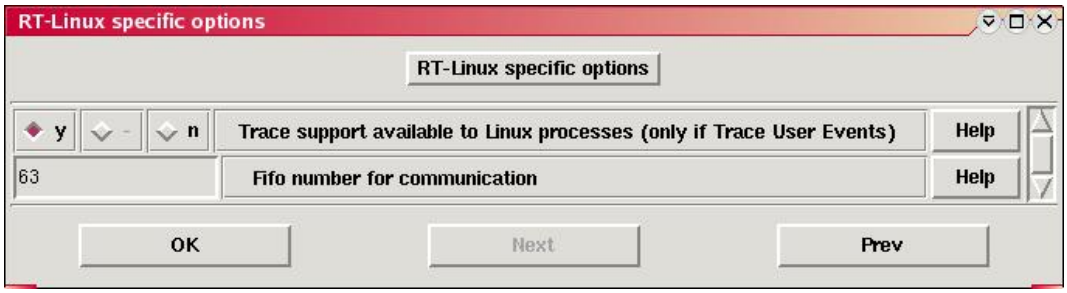


Figure 4. Specific RT-Linux options



3. API / Compatibility

This section presents the complete API of the **Lptrace**, subdivided in the units of functionality introduced above. All functions have the same syntax as defined by POSIX, and their semantics is also maintained, except the aspects related to the restrictions removed in Section 2.2.1, “Removal of Restrictions”.

A. **Trace Core**.

```
int posix_trace_attr_init(trace_attr_t *);
int posix_trace_attr_destroy(trace_attr_t *);
int posix_trace_attr_getgenversion(const trace_attr_t *, char *);
int posix_trace_attr_getname(const trace_attr_t *, char *);
int posix_trace_attr_setname(trace_attr_t *, const char *);
int posix_trace_attr_getcreatetime(const trace_attr_t *, struct timespec *);
int posix_trace_attr_getclockres(const trace_attr_t *, struct timespec *);
int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *restrict,
                                         int *restrict);
int posix_trace_attr_setstreamfullpolicy(trace_attr_t *, int);
int posix_trace_attr_getmaxuserevent size(const trace_attr_t *restrict,
                                         size_t, size_t *restrict);
int posix_trace_attr_getmaxsystemeventsize(const trace_attr_t *restrict,
                                           size_t *restrict);
int posix_trace_attr_getmaxdatasize(const trace_attr_t *restrict,
                                    size_t *restrict);
int posix_trace_attr_setmaxdatasize(trace_attr_t *, size_t);
int posix_trace_attr_getstreamsize(const trace_attr_t *restrict,
                                   size_t *restrict);
int posix_trace_attr_setstreamsize(trace_attr_t *, size_t);

int posix_trace_shutdown(trace_id_t);
int posix_trace_clear(trace_id_t);
int posix_trace_start(trace_id_t);
int posix_trace_stop(trace_id_t);

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_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 *);
```

B. **On-Line Trace**.

```
int posix_trace_create(pid_t, const trace_attr_t *restrict,
                       trace_id_t *restrict);

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);

int posix_trace_get_attr(trace_id_t, trace_attr_t *);
int posix_trace_get_status(trace_id_t, struct posix_trace_status_info *);
```

C. **Trace To Log**.

```
int posix_trace_attr_getlogfullpolicy(const trace_attr_t *restrict,
                                        int *restrict);
int posix_trace_attr_setlogfullpolicy(trace_attr_t *, int);
int posix_trace_attr_getlogsize(const trace_attr_t *restrict,
                                size_t *restrict);
int posix_trace_attr_setlogsize(trace_attr_t *, size_t);

int posix_trace_create_withlog(pid_t, const trace_attr_t *restrict,
                                int, trace_id_t *restrict);

int posix_trace_flush(trace_id_t);
```

D. **User Events**.

```
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)
```

E. **Event Filter**.

```
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_get_filter(trace_id_t, trace_event_set_t *);
int posix_trace_set_filter(trace_id_t, const trace_event_set_t *, int);
int posix_trace_trid_eventid_open(trace_id_t, const char *restrict,
                                  trace_event_id_t *restrict);
```

F. **Trace Inheritance**. Not supported.

G. **Log Retrieval**. (Note: although some of the following functions have also appeared in previous units, they are referred to pre-recorded streams instead of active streams.)

```
int posix_trace_close(trace_id_t);
int posix_trace_open(int file_desc, trace_id_t *);
```

```
int posix_trace_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_attr_getgenversion(const trace_attr_t *, char *);
int posix_trace_attr_getname(const trace_attr_t *, char *);
int posix_trace_attr_getcreatetime(const trace_attr_t *, struct timespec *);
int posix_trace_attr_getclockres(const trace_attr_t *, struct timespec *);
int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *restrict,
                                         int *restrict);
int posix_trace_attr_getmaxusereventsize(const trace_attr_t *restrict,
                                         size_t, size_t *restrict);
int posix_trace_attr_getmaxsystemeventsize(const trace_attr_t *restrict,
                                           size_t *restrict);
int posix_trace_attr_getmaxdatasize(const trace_attr_t *restrict,
                                    size_t *restrict);
int posix_trace_attr_getstreamsize(const trace_attr_t *restrict,
                                   size_t *restrict);
int posix_trace_attr_getlogfullpolicy(const trace_attr_t *restrict,
                                      int *restrict);
int posix_trace_attr_getlogsize(const trace_attr_t *restrict,
                                size_t *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_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 *);
```

4. Implementation issues

From an implementation point of view, the differences between the former *Ptrace* component and the new **Lptrace** component can be summarized in three main areas. Firstly, several bugs have been corrected and quite a few optimizations have been introduced. Secondly, the support to trace into streams with log has been added to the tracing capabilities at the RT-Linux level. And thirdly, the complete (and optimized) code has been arranged into the implementation options and general restrictions proposed above in Section 2.2, “Proposal of a Lightweight POSIX Trace System”. We will now discuss these three aspects.

There are two optimizations in the original *Ptrace* code which are worth discussing. The first one is the introduction of a *global filter* for system events. This global filter maintains the set of system events which are currently filtered in all the possible streams created by the application. The purpose of this global filter is that the instrumentation will only generate a system event when there is at least one *running* active stream which is not filtering out events of that particular type. In all other cases (that is, no streams are yet created, or none is running, or the event type is filtered out from all the running streams), the cost of the instrumentation point is reduced to an "if" sentence on which a test bit operation is performed over the global filter. This greatly reduces the overhead introduced in the kernel due to tracing system events. The second optimization is to timestamp events in TSC (TimeStamp Counter) format when tracing, and to convert it to the POSIX "timespec" struct when retrieving the event. This reduces the time in tracing an event, which is always a good feature. Obviously, the penalty here is that the cost of retrieving an event is proportionally augmented. However, this is acceptable since the tracing of events (specially of system events, generated by the kernel) is the actual time bottleneck of any tracing system.

In general, introducing tracing to streams with a log has been quite straightforward, since this type of tracing relies on having a POSIX-like input/output interface available (i.e., `open`, `close`, `read`, `write`, etc.) in order to flush the traced events into the log. In the case of an embedded system like RT-Linux the "log" is normally a synchronous device, such as a serial port (or a RT-FIFO). In such situations, a Linux process will be listening at the "other end" of the device, retrieving the events and writing them to a proper log (disk file). Following the re-

strictions stated in Section 2.1, “The MRSP Model and its Tracing Limitations”, the only log full policy implemented is `POSIX_TRACE_APPEND` and the stream policy `POSIX_TRACE_FLUSH` is not supported (that is, flushing is always asked for explicitly by the application, by calling `posix_trace_flush`). The only interesting implementation detail is related to the use of a RT-FIFO as the typical "log" device in RT-Linux. The problem with these devices are that they cannot be configured to block a RT-Linux task when full. The lack of this feature, which is considered not useful for real-time tasks, makes that when the fifo is full, the `write` operation immediatly returns with an error (maybe with the last event partially written). In order to prevent event loss during the flusing, two solutions can be used: to implement a sophisticated protocol between the flushing and the process listening at the other end, in order to include retransmissions (and ack-like confirmations) or to make sure the device has enough room when flushing. For the sake of simplicity in this (first) version of log support, the second alternative has been chosen. In this case, the flush operation is make in periodical "bursts". In particular, after a number of bytes have been written to the log, the `posix_trace_flush` function suspends the invoking task during a interval of 20ms. During that time, the Linux process listening on the fifo is supposed to retrieve all the pending events, making room for the real-time task to continue fluhsing. This mechanism, although not much sophisticated, has been proven to work and its temporal behaviour is compatible with the typical temporal analysis of hard real-time systems.

Finally, the subdivision of the trace support in the different units of functionality has been done in the typical manner, by introducing preprocessor conditional statements in the tracing code. At configuration time of the tracing support (see figures in Section 2.3, “Implementation in RT-Linux”), a label is defined for each implementation option (unit of functionality, system event, special feature, etc.) selected by the user. As a result, only the code corresponding to the selected options will be compiled, adjusting the code size of the kernel to the desired functionality. The implementation effort in this aspect has been to detect all the cross dependencies among options and to make sure that deactivating an option at configuration time really remove all its related code at compilation time.

5. Tests and validation

5.1. Validation criteria

The motivation for this component was to produce a POSIX-like tracing system which was lighter, in both memory footprint and overhead, than the original system proposed in the standard. Obviously, the validation criteria in this case is to compare the memory and time requirements of this **Lptrace** component with the requirements of the former *Ptrace*.

We have developed three different tests, one for measuring the memory requirements of the new component and two for measuring the most relevant execution costs: the tracing and retrieval of user events and the tracing of system events.

5.2. Test 1

The objective of this test is to analyse the memory requirements of the different options and units by which the original standard has been divided. In order to achive this, the trace subsystem has been compiled several times, each time with different configuration options, in order to detect the contribution in memory footprint to the size of the RT-Linux kernel. Since the combination of certain configuration options leads to the functionality of the former *Ptrace* component, the comparison between both the former and the new component is also implicit in this study.

As the number of configuration combinations is enormous, we have first studied the source code of the component in order to identify the combinations which contribute to the size of the final trace subsystem inside the kernel. The experiments have been performed in the following way:

- All the testing has been performed with a RT-Linux 3.2-pre1 version (on a Linux kernel 2.4.18) without any other OCERA component except the **Lptrace**.
- In each compilation, the other RT-Linux compilation options (these not related whith this component) are left as defined by default (except the `Enable debugging` option, which is *disabled*), since they do not affect the study. This is because we want to detect the memory usage of the implementation options inside the **Lptrace** component, and the comparison with the other options in the original RT-Linux kernel is less relevant.

(Note: the debugging option has to be deactivated since the debugging information inside the kernel modules can make them up to ten times bigger, and this is not intended for the final, optimized kernel.)

- For each combination of options, the RT-Linux kernel has been completely recompiled and then loaded into memory. As the trace support is entirely linked into the `rtl_sched.o` module, this is the module whose memory footprint, once loaded, has been collected.

- In all the experiments, the code corresponding to the instrumentation of the kernel (in places like system calls, mutex locking, context switching, etc.) has been deactivated. This is because this instrumentation *uses*, but it is not part of, the trace support.

The experiments have allowed us to isolate the contribution of the following options to the final memory footprint of the `rtl_sched` kernel module: (1) the availability of eight simultaneous streams (vs. single stream), (2) trace core unit, (3) on-line trace unit, (4) trace to log unit, (5) trace filter unit, (6) user evnets unit, and (7) ability of Linux processes to trace. The sum of all the isolate contributions, plus the generation of system events, is actually the entire support of the component.

The following figures visually summarize the results. In the first figure, the seven contributions have been compared to each other by means of a typical "pie" graphic. Then, the second figure presents a detailed view of the memory footprint of the last six contributions (the five units of functionality plus the tracing support for Linux process). In this detailed view, the footprint of each unit is further divided into two contributions: the "basic support", on which the unit is not generating the trace-related system events mandated by the standard and the generation of such events.

Figure 5. Contribution of individual options to the RT-Linux kernel memory footprint

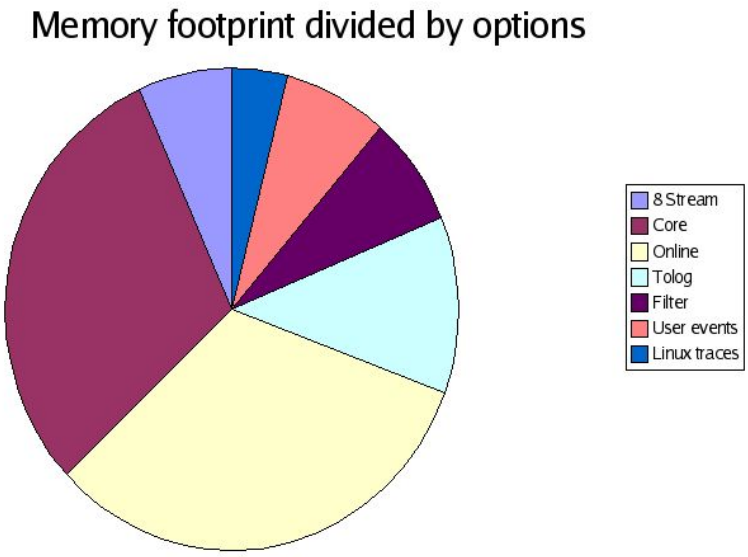
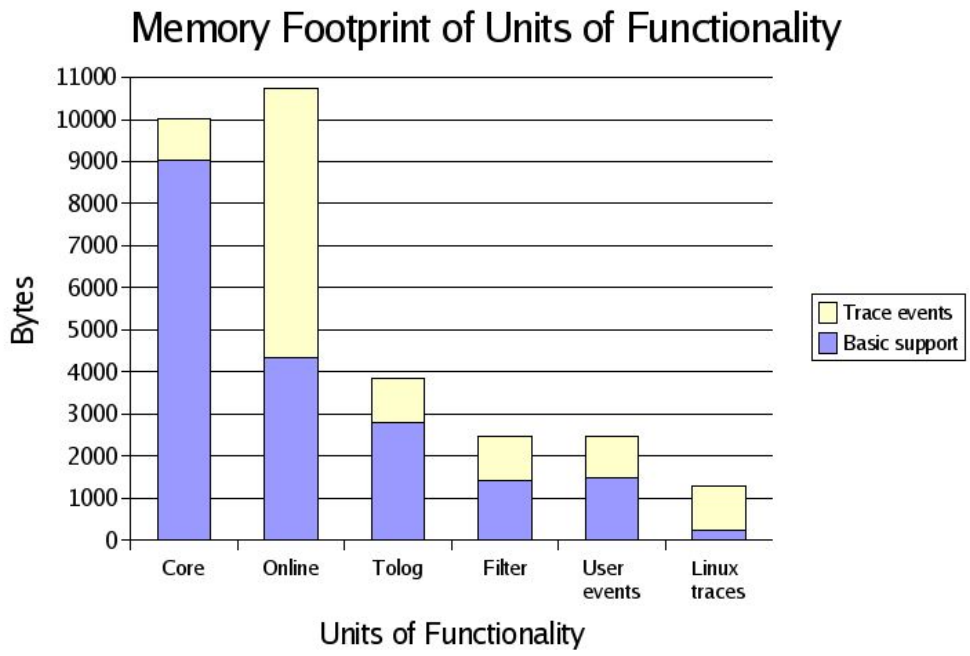


Figure 6. Memory footprint of tracing Units of Functionality



The results show that the subdivision of the original support in the implementation options proposed in this component greatly reduces the amount of memory required to partially support the POSIX trace standard, specially if only one stream is used and the tracing is either done to a log or else it is on line but without the generation of trace-related events.

5.3. Test 2

This test presents the time costs for tracing and retrieving user events. Measurements have been taken with an ad-hoc software measurement mechanism, in two different scenarios: (a) a complete, POSIX-like tracing system (with all the possible configuration options turned on) and (b) a minimal tracing system, configured with a single stream, no trace-related system events and the minimum number of units of functionality to allow on-line tracing ("trace core", "on-line trace" and "user events").

The experiments were conducted in the following conditions:

- All the tests were made on a PC computer with a 700 Mhz. Pentium-III processor, 256 kilobytes of cache memory and 256 megabytes of RAM.

- A single test corresponds to an execution of a real-time application with two periodic tasks, with a period of 10ms each: one tracing two events each time is released and another retrieving two events each time is released. The associated data of each traced event is a single integer value.

  This way of tracing makes sure that both the tracing and the retrieval of an event execute the longest section of code (there is always room for the tracing task to insert an event and there is always an event available for the retrieval task). In addition, both tasks are released with a different offset (of half a period), making sure that they do not produce any interference to each other.

- Time measurements are collected in TSC format (by executing `rdtsc`) right after and before the trace and retrieval functions are called. Then, these time values are stored in a memory area shared with a Linux process. This produces the least possible overhead in the measurement itself. After the real-time tasks have finished running, the Linux process collects all the measurements, converts them into nanoseconds, and calculates some statistics (minimum, maximum, average and variance values).

- As stated above, for each interesting function to measure (in this case, tracing and retrieving an event), the experiments take two consecutive measurements. This is done in order to detect cache effects (normally, the second measurement has many more cache hits and this is clearly shown in the measurements). The results below show the measurements separated in first and second values and also the total figures. Each experi-

ment collects a total of 20,000 values of each interesting measurement.

The next two tables show the results for two different configurations of the trace subsystem: (a) complete POSIX support and (b) minimal support, containing only the units "trace core", "on-line trace" and "user events", with a single stream and no system events.

Table 1. Results for test 1 with complete POSIX trace support

Measured Value	Minimum	Maximum	Average	Variance
Trace (1st value)	530.0000	4157.0000	710.3927	43026.7799
Trace (2nd value)	481.0000	1478.0000	590.7501	9631.2022
Trace (all values)	481.0000	4157.0000	650.5714	29907.5790
Retrieve (1st value)	411.0000	4108.0000	685.1246	28409.4437
Retrieve (2nd value)	316.0000	1169.0000	481.3689	2168.1154
Retrieve (all values)	316.0000	4108.0000	583.2468	25667.8759

Table 2. Results for test 1 with minimal trace support

Measured Value	Minimum	Maximum	Average	Variance
Trace (1st value)	497.0000	4092.0000	703.7739	96913.9476
Trace (2nd value)	469.0000	1176.0000	479.0450	423.0570
Trace (all values)	469.0000	4092.0000	591.4094	61294.2719
Retrieve (1st value)	371.0000	4537.0000	423.4935	25555.5828
Retrieve (2nd value)	368.0000	1123.0000	376.6447	626.1649
Retrieve (all value)	368.0000	4537.0000	400.0691	13639.5763

The tables show, not surprisingly, that in all cases (when considering first values, second values, or all of them) the average cost of tracing and retrieving a user event is lower when using a minimal tracing system than their corresponding in a complete, full POSIX system. In this case, the reduction in the costs is mainly due to two factors: supporting a single stream (instead of eight) and nor reporting trace-related events (such as `POSIX_TRACE_OVERFLOW` and `POSIX_TRACE_RESUME`).

5.4. Test 2

This second test presents a comparative study between the cost of tracing a system event in a complete, POSIX trace system and the same cost with a minimal trace support. In this case, the minimal support only contains the "trace core" and "on-line trace" units of functionality. The experiments have been carried out in the same computer and in analogous conditions than experiments in test 1. In particular:

- A single test corresponds to an execution of a real-time application with a single periodic task which uses the `clock_nanosleep` system call to periodicall suspend itself.
- The ad-hoc time instrumentation collects the current time (in TSC format) right after and before the tracing of a system event inside the `clock_nanosleep` function. As in test 1, these time values are stored in a memory area shared with the Linux process which will perform the statistical analysis (after the execution of the real-time task).
- Each experiment takes 20,000 values of the cost of tracing the system event inside the `clock_nanosleep` system call.

The table below summarizes the results obtained in the second test:

Table 3. Results for tracing system events with complete and minimal trace support

Measured Value	Minimum	Maximum	Average	Variance
Trace system event (complete POSIX trace support)	488.0000	1488.0000	593.0995	9651.0731
Trace system event (minimal trace support)	419.0000	1697.0000	533.6576	10625.1237

The analysis of these results is analogous to the test 1. The only aspect worth mentioning is that results for tracing a user event (in experiment 1) are not exactly comparable with results in this experiment, since the instrumentation inside the `clock_nanosleep` system call generates an event with an associated data of 48 Kbytes, instead of the 4 Kbytes (of the integer value) in the user event in test 1. Even so, results show that tracing system events is less costlier than tracing user events (this is because the trace support skips some error checking in the former case).

5.5. Results and comments

The three experiments have shown the benefits of having tailored the original POSIX trace support into the needs of small kernels like RT-Linux. It is clear from the presented results that, by following our proposed "minimal trace support", the kernel exhibits both a smaller memory footprint and a faster API while maintaining most of the POSIX trace philosophy.

Therefore, this component allows the system designer to trade off between functionality and performance when configuring the trace system.

6. Examples

This component does not present any example, since its functionality is almost the same as in the former *Ptrace* component. The examples presented for that component in the Milestone 2 are also applicable here, given that the trace subsystem is compiled with enough units of functionality to support them (on-line trace, user events, event filter, etc.).