# Monitoring and Analyzing RTAI System Behavior Using the Linux Trace Toolkit

**Karim Yaghmour**
Opersys inc.
www.opersys.com
karym@opersys.com

**Abstract**

Analyzing and understanding the behavior of real-time embedded systems is a complex task. Often, developers resort to ad-hoc methods in order to understand the behavior of their systems and isolate bugs. Answering this need, some commercial RTOS vendors have made available different types of tools to understand their product's behavior. The effectiveness of such tools vary, but they are usually quite expensive, which proves to be a barrier to entry when wanting to design any system using an RTOS.

The RTAI extensions to the Linux Trace Toolkit presented here take these barriers away and provide developers with an open-source set of tools to analyze and understand the behavior of any RTAI-based real-time system. This analysis involves presenting RTAI's behavior in a control-graph form and presenting statistics regarding the overall system behavior including all the real-time tasks that ran. Furthermore, tracing such a system involves an overhead of, at most, 1 micro-second per event, which is very acceptable behavior for most real-time systems.

## 1 Introduction

In the course of developing a real-time embedded system, there arises situations where a clear understanding of the system's different components' dynamics is needed. Using this understanding, system developers can then isolate performance bottlenecks, solve hard to catch synchronization problems or simply present outside observers with a thorough description of the system's behavior. On the hardware level, this might involve the use of different types of probes and sensors that give precise measurements of the different hardware involved. At the other end, on the application level, this might involve using performance profilers to isolate bottlenecks or symbolic debuggers to find bugs in the source code. Yet, between both extremes lies a need to understand the overall behavior of the software without modifying it's dynamics. Therefore, there are three levels of detail needed when wanting to understand the behavior and dynamics of a real-time embedded system: the hardware level, the system level and the application level.

As the current market trends show, the need for hardware level probing equipment is likely to be superseded by the need for system level probing software/equipment given the rising trend in using off the shelf equipment to build real-time embedded systems. This trend is likely to be even more important for designers planning to use a real-time Linux derivative as the basis of their system since the hardware platforms supported by Linux are more than often quite mature.

That said, contrary to many other operating systems used in an embedded real-time environnement, as will be discussed in section 6, no real-time Linux variant offers any set of tools enabling designers to understand the dynamics of the system being developed. Developers, therefore, have to resort to ad-hoc methods to "visualize" the dynamic behavior of the system they are developing. This often means inserting variants of the *printf()* call at key parts in the code, with all the penalties involved.

Hence, the need for a complete and flexible system level monitoring and analysis tool for real-time Linux. It needs to be "complete" in the sense that it has to address all the aspects of a real-time Linux system and "flexible", in order to easily accommodate the different environnements where a real-time Linux system is used. To fully convey the dynamic behavior of a real-time Linux system, the tool, or set of tools, needs to: describe the sequence of key events that occurred during the period of time when the system is observed, present the related statistics

and provide a complete detail of the events that occurred and their impact on the system.
The methods and tools discussed below are based on previous work done on the Linux Trace Toolkit which enables run-time tracing and off-line reconstruction of the dynamic behavior of the Linux kernel as described in [13].
In section 2 the details of the data collection methods used are presented. Section 3 discusses the details of the implementation of the trace toolkit RTAI support. Section 4 describes how the toolkit can be used to trace an RTAI/Linux system and provides some examples. Section 5 provides a couple of examples of the use of LTT to understand the behavior of a live RTAI system. Section 6 presents some related work. Section 7 discusses future directions.

# 2 Data collection architecture

Following the modular design philosophy adopted as the basis of the previous LTT work [13], the RTAI additions to LTT are composed of independent software modules. The interactions between these modules are as presented in figure 1. In addition to the basic modules, an RTAI trace facility has been added in order to provide the instrumented RTAI with a single entry point for tracing. Both trace facilities still use the same trace module which has been modified for this purpose. This allows the upper layer tools' interactions with the trace module to remain unmodified and ensures that there is a single system-wide repository for traces.
The arrows in figure 1 presents the flow of information through the tracing process. The initial sources of trace data are: the Linux kernel, the RTAI core and the different RTAI modules instrumented. Basically, all the primary sources of information feed the key events to the corresponding trace facility which forwards those onto the trace module. The trace daemon then reads the data collected by the trace driver and commits it to file. Note that the trace module itself is visible from user space as an entry in the */dev* directory, hence facilitating configuration and interaction. Apart from the additional trace facility and the additional event sources, the scheme used is identical to the one used for the basic LTT operation. Since the basic LTT architecture has been covered elsewhere [13], the following discussion will be limited to the additions to this architecture in order to allow RTAI to be traced. Section 2.1 discusses the RTAI trace facility. Section 2.2 discusses the RTAI instrumentation. Section 2.3 will discuss the Linux kernel trace facility. Section 2.4 discusses the instrumentation of the Linux kernel. Section 2.5 discusses the trace module. Section 2.6 will discuss the trace daemon. Finally, section 2.7 will introduce the data analysis and presentation software.

## 2.1 RTAI trace facility

The RTAI trace facility is an addition to the already existing facilities in RTAI. As with other RTAI facilities it is implemented as an independent kernel module. It's main purpose is to provide all RTAI modules with a single entry point for event tracing. Though, it does not log any events. It only forwards them onto the trace module, if it has been loaded. If the trace module hasn't been loaded, then traced events are ignored by the trace facility.
In order to fulfill it's role, the trace facility provides three main functionalities: a unified RTAI trace function, a trace module registration function and a trace module unregistration function. For the trace module to receive RTAI events, it has to register itself with the RTAI trace facility providing it with a callback function. This callback function will be called upon every time an RTAI event occurs. Typically, this is the same function that the trace module provides the Linux kernel trace facility. Hence, the unified trace repository.
Contrary to the Linux kernel trace facility, the RTAI trace facility provides no configurable option. At least, not yet. Though, it does have a very similar functionality which is to provide a link between the different RTAI modules and the trace module.

## 2.2 RTAI instrumentation

The RTAI instrumentation is a very important component of the LTT support for RTAI as it determines what information is retrieved from the different RTAI modules. As with the Linux kernel instrumentation, there are different types of events each with its set of fields to describe it. Since some events belong to the same type of functionality or to the same RTAI module, event sub-types are often used to identify an event among the group of events to which it belongs. Contrary to the Linux kernel which has a single entry point for its user services, the system call trap, RTAI has many entry points into its services. That is, to access Linux kernel services, user-space tasks all have to go through the system call trap. To access RTAI services, though, RTAI tasks call on the service's API directly, which forces the instrumentation of all entry points into RTAI services. This was simpler in Linux since only the system call trap had to be instrumented in order to identify which service was being called on. Consequently, there are far more events instrumented in RTAI than in the Linux kernel. Figure 2 presents the RTAI events and categories of events traced by LTT.

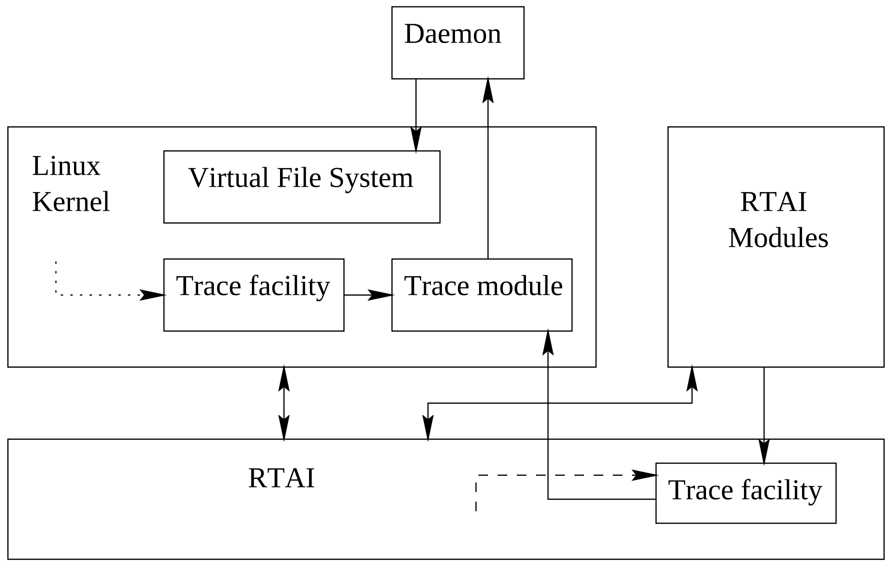


Figure 1: LTT architecture with RTAI support.

Also, as with kernel instrumentation, RTAI instrumentation can be disabled during kernel configuration by enabling or disabling it during the "Kernel tracing" configuration step.

## 2.3 Kernel trace facility

The kernel trace facility has not changed with the addition of RTAI support to LTT. Although it has seen some interesting additions such as a generalized event hooking mechanism enabling loadable kernel modules to hook on to certain events. Its main purpose, of providing a link between the trace module and the different kernel facilities, remains unchanged.

## 2.4 Kernel instrumentation

As the kernel trace facility, the kernel instrumentation has not changed with the addition of RTAI support to LTT. The key Linux kernel events traced remain the same as described in [13]. Kernel instrumentation is configurable as is RTAI instrumentation. Although the former is necessary for the later.

## 2.5 Trace module

The trace module's main purpose is to record the event descriptions into its trace buffers. This involves retrieving additional information about the event such as the ID of the CPU where the event occurred and a timestamp. In order to enable large amounts of data to be collected, a double-buffering scheme is used whereby a buffer is used to write the incoming events while the other buffer's content can be committed to file by the trace daemon. Since copying data from the buffer to user space into the daemon's memory and then to kernel space during file committing is expensive, the trace module implements the *mmap()* interface that enables the trace daemon to map the module's buffers into its own address space. Therefore, when committing, the daemon provides the file-system interface with its own pointer to the trace data without having to copy said data prior to commit.
The main modification to the trace module in order to support RTAI tracing is the use of real-time safe locking mechanisms when writing event data rather than the use of ordinary Linux kernel locking mechanisms. This is necessary to ensure that regardless of the event, its description's logging will be atomic. Although this raises the issue of ordinary Linux events having priority on RTAI events, it should be noted that recording an event's occurrence takes less that a micro-second. Also, the trace module can be configured using event masks through the trace daemon, enabling the developer to ignore non-critical or cumbersome information. Therefore, this should not be a problem when tracing a system.

## 2.6 Trace daemon

The trace daemon is the main user side tool to control tracing. It provides its user with a number of command-line options which enable him to control all the details of the tracing process. Its second purpose is to handle the data accumulated in the trace module's buffers and get it to be committed to file for

| *Event type* | *Number of subtypes* | *Event detail* |
|---|---|---|
| RTAI mount | N/A | None |
| RTAI un-mount | N/A | None |
| Global IRQ entry | N/A | IRQ ID and level flag (kernel vs. user space) |
| Global IRQ exit | N/A | None |
| CPU own IRQ entry | N/A | IRQ ID and level flag |
| CPU own IRQ exit | N/A | None |
| Trap entry | N/A | Trap ID and corresponding address |
| Trap exit | N/A | None |
| SRQ entry | N/A | SRQ ID and level flag |
| SRQ exit | N/A | None |
| Switch to Linux | N/A | CPU ID where switch occurred |
| Switch to RT | N/A | CPU ID where switch occurred |
| RT-Scheduling change | N/A | Incoming task, outgoing task and outgoing task state |
| Task | 12 | Sub-type and 3 fields |
| Timer | 5 | Sub-type and 2 fields |
| Semaphore | 6 | Sub-type and 2 fields |
| Message | 6 | Sub-type and 3 fields |
| RPC | 4 | Sub-type and 3 fields |
| Mail-box | 10 | Sub-type and 3 fields |
| FIFO | 26 | Sub-type and 2 fields |
| Shared memory | 5 | Sub-type and 3 fields |
| POSIX | 44 | Sub-type and 3 fields |
| LXRT | 8 | Sub-type and 3 fields |
| LXRT-Informed | 10 | Sub-type and 3 fields |

Figure 2: RTAI events traced.

offline post-processing. Apart from recognizing the existence of RTAI events for the setting of the event masks by the user, the trace daemon is not different from the standard LTT trace daemon as discussed in [13].

## 2.7 Data analysis and presentation software

The data analysis and presentation software is the most crucial component of the Linux Trace Toolkit as the developer's understanding of the system's behavior largely depends on its ability to provide a complete and precise description of the system's dynamics while remaining as simple as possible in its presentation of the data and the underlying analysis. Using the event sequences, the analysis software reconstructs the states in which the system was in and extracts system statistics. The system states are displayed as a control graph where the x axis is time and the y axis a software entity such as a process, a real-time task, the kernel or the RTAI core. Vertical lines therefore mark control transitions from one entity to another and horizontal lines mark time spent within code belonging to the corresponding entity. Moreover, every transition into system services or facilities, such as a system call or a hardware interrupt, is paired with a icon and text describing the event that occurred. This type of display provides the user with a clear view of the transitions that occurred and the reason of their occurrence.

Along with this graphical view, the system events can be viewed in the form of a time-ordered list where all events are detailed in full. This enables further manual trace analysis by the developer.

Summing up all the information collected, a third display provides the developer with a set of statistics regarding the system's behavior. In the case of the basic Linux kernel, the statistics displayed supersede the data presented by tools such as *ps* since the statistics displayed by LTT are not approximations but relate the exact entries and exists from and into the different software components. Moreover, some other statistics displayed are simply not available through the standard */proc* interface.

Facilitating the transition from one system view to the next, shortcut menu items are provided in case the developer needs to view more information regarding an event or requires the statistics regarding the software entity within which the event occurred.

## 3 Toolkit implementation

Understanding how key parts of the tracing system are implemented is necessary for further enhancement or expert usage. The following sections discuss each of the most important RTAI additions to LTT and the most important basic LTT code details. Section 3.1 discusses the main trace facility entry points and how they can be used. Section 3.2 discusses the trace statements and how they are used. Section 3.3 discusses the modifications brought to the trace module in order to support RTAI tracing. Section 3.4 covers the additions made to the visualization tool in order to view the behavior of an RTAI/Linux system.

### 3.1 Trace facilities entry points

As discussed in section 2.1 and 2.3, both RTAI and the standard Linux kernel have a tracing facility. While both serve the same end, they have differences and are used to record two completely different sets of events. The following covers each trace facility's services and how they are used.

#### 3.1.1 Linux kernel trace facility

The Linux kernel trace facility is part of the basic implementation of LTT. It provides the following services:

- Trace module registration:
  `register_tracer()`
  This takes a callback function as a parameter and sets it as being the function to be called upon the occurrence of an event and returns an error code. This is the function used by the trace module to register its tracing function.

- Trace module de-registration:
  `unregister_tracer()`
  Takes the already registered callback function, verifies it is the one currently used for tracing and unregisters it. Again, an error code is returned.

- Configure tracing:
  `trace_set_config()`
  Enables the trace module to setup different trace parameters such as the depth at which the caller's address should be retrieved upon a system call or the address range to which this address should belong. These settings are then used by the function that traces system call entries to fetch the desired information.

- Get configuration:
  `trace_get_config()`
  Provides the caller with the current tracing configuration as setup using the above mentioned function.

- Register event callback:
  `trace_register_callback()`
  The caller of this function provides a function callback and an event ID. Thereafter, any occurrence of the given event ID will result in the calling of the given function.

- Unregister event callback:
  `trace_unregister_callback()`
  Used to unregister a callback registered using the previously mentioned function.

- Trace event:
  `trace_event()`
  The unified kernel trace function called upon the occurrence of all kernel events. This is where the trace callback function provided by the trace module is called upon. Also, this is where registered callbacks are summoned.

Apart from the registration of arbitrary callback functions, all other functions are used in order to implement tracing of the Linux kernel.

#### 3.1.2 RTAI trace facility

The RTAI trace facility is one of the additions necessary to bring LTT support for RTAI. The following are the services provided by this facility:

- Trace module registration:
  `rt_register_tracer()`
  As the kernel registration function, this takes a callback function as a parameter and set it as being the function to be called upon the occurrence of an event and returns an error code. This is the function used by the trace module to register its tracing function. The later is the same as the one provided by trace module to the kernel trace facility.

- Trace module de-registration:
  `rt_unregister_tracer()`
  Takes the already registered callback function, verifies it is the one currently used for tracing and unregisters it. Again, an error code is returned.

- Trace event:
  `rt_trace_event()`
  The unified RTAI trace function called upon the occurrence of all RTAI events. This is where the trace callback function provided by the trace module is called upon.

All trace statements end up calling upon the *trace_event* service. Whether this ends up calling the trace module tracing function or not, the facility will serve its purpose by providing a unified interface for all trace statements.

## 3.2 Trace statements

The trace statements are at the basis of the tracing system. They are placed on the execution path of important system services code. These statements, although they can be made not to generate code, they are designed not to modify the flow of the code they are inserted to. This is an example trace statement:

```
TRACE_RTAI_SCHED_CHANGE();
```

Actually, it is a C macro and is used as a normal C function by passing it the required parameters. In this case the complete call is:

```
TRACE_RTAI_SCHED_CHANGE
(rt_current->tid, new_task->tid, rt_current->state);
```

This particular trace statement is used to monitor scheduling changes within the RTAI uniprocessor scheduler. All the other trace statements closely resemble this example. They are all C macros defined within the trace header file and are set to generate code if tracing is enabled. Otherwise, these macros generate no code at all. Here is the complete definition of the macro above, when tracing is enabled:

```
#define TRACE_RTAI_SCHED_CHANGE(OUT, IN, OUT_STATE) \
     do \
     {\
     trace_rtai_sched_change sched_event;\
     sched_event.out       = (uint32_t) OUT;\
     sched_event.in        = (uint32_t) IN;\
     sched_event.out_state = (uint32_t) OUT_STATE; \
     rt_trace_event(TRACE_RTAI_EV_SCHED_CHANGE, &sched_event);\
     } while(0);
```

When tracing is disabled, the definition is as follows:

```
#define TRACE_RTAI_SCHED_CHANGE(OUT, IN, OUT_STATE)
```

The complete list of statements and necessary definitions can be found in: `[rtai_base_directory]/include/rtai_trace.h`

## 3.3 Modifications to the trace module

The main modification to the trace module is the usage of real-time-safe locking mechanisms. Hence, rather than using the `spin_lock_irqsave()/spin_unlock_irqrestore()` pair, the `rt_spin_...` variants are used for the critical region where the event data is logged. This ensures the integrity of the data buffers. Note that when RTAI tracing is disabled, the conventional locking mechanisms are used.

## 3.4 Additions to the visualization and analysis tool

The visualization and analysis tool is the LTT component that has had the most additions in order to provide the capability of handling RTAI traces. This is mainly due to the fact that the behavior of an RTAI/Linux system can be fairly complex. Whereas with the plain Linux kernel, the system was either in kernel state or in process state, in an RTAI/Linux system there are four states in which the system can be in: RTAI core, RTAI task, Linux kernel and Linux process. Furthermore, the transitions from one state to another can sometimes be ambiguous. That is, for a same sequence of events, it is not always easy to determine which state transition took place, if any. At the time of this writing, the state machine used within the visualization tool to reconstruct the behavior of an RTAI/Linux system isn't 100% accurate, but it's fairly close. The main shortfall being the fact that scheduling changes made by the LXRT subsystem aren't recognized as state-changing events. This, though, does not constitute much of a problem since most LXRT scheduling changes occur close to RTAI scheduling changes and, as such, the state machine used reconstructs the system's behavior correctly. Although it would be interesting to discuss how the state machine is built and used in detail, the scope of such a discussion goes beyond the purpose of this writing.

Since the behavior of a normal Linux kernel without RTAI differs greatly from the behavior of an RTAI-controlled Linux kernel, the analysis made is different and so is the state machine modeling the system's behavior. That said, and in an effort to promote future additions to LTT, the data decoding and analysis layers of the visualization tool have been generalized in order to recognize different types of traces. Given a trace type, different functions and tables are used. Some modifications were also made to the display layers since an RTAI/Linux system has different characteristics from a plain Linux system and some additional information has to be displayed regarding system analysis.

Hence, the addition of the following files to the visualization tool's source code:

- `Tables.c` which contains the tables management code.
- `Tables.h` which contains the tables abstractions.
- `LinuxEvents.h` which contains the list of all the Linux event definitions.

- `LinuxTables.c` which contains the Linux specific tables.
- `LinuxTables.h` the Linux tables header file.
- `RTAIEvents.h` which contains the list of all the RTAI event definitions.
- `RTAIDB.c` which contains the code necessary to analyze a trace containing RTAI data.
- `RTAIDB.h` the RTAI database header file.
- `RTAITables.c` which contains the RTAI specific tables.
- `RTAITables.h` the RTAI tables header file.

Using the work already done to add RTAI support into LTT, adding LTT support for other systems should be a matter of providing the right analysis functions and corresponding tables. Most of the work having to be put in would regard the writing of the state machine describing the system to be supported.

## 4 Toolkit usage

This section attempts to provide a broad yet brief overview of how LTT can be used to trace and analyze RTAI/Linux systems. An emphasis is put on the actual usage of the tools, the theoretical and implementation details having already been covered. Section 4.1 discusses the details of the configuration of the RTAI-patched Linux kernel in order to support tracing. Section 4.2 discusses the loading sequence of the different RTAI modules. Section 4.3 discusses how to trace the operation of an RTAI/Linux system. Section 4.4 discusses the actual display and analysis on an RTAI trace.
It is important to note that no matter how thorough the discussion is and how efficient LTT is in providing an exact reconstruction of the system, it is expected that the designer using LTT has to acquire a certain knowledge about the internals of RTAI and Linux in order to fully appreciate the information generated. This discussion does not attempt to cover such topics, although the usage of LTT makes it easier to grasp the underlying dynamics. Also, the discussion does not attempt to reproduce the instructions provided in the LTT help files, but rather aims at enriching them.

### 4.1 Compilation pre-requisists

Having installed the Linux kernel and RTAI sources and patched each with the related LTT patch, as described in the help files found with the LTT sources and on the project's web site [1], the next step is to configure the kernel. Note that this step is fully described in the LTT help files.
During kernel configuration, to enable kernel and RTAI tracing, the "Kernel events tracing support", in the "Kernel tracing" menu, must be set to module and the "RTAI event tracing support" must be set to "yes". The reason that kernel tracing has to be selected as module is that the trace module has to be loaded after the RTAI trace facility is loaded, if RTAI tracing is to be enabled. Since the RTAI trace facility cannot be loaded prior to the kernel bootup, RTAI tracing cannot be supported if the trace module is built into the kernel.
Once this is configured and the rest of the configuration is complete, the building of the dependencies and the compilation of the kernel itself can be carried out.

### 4.2 Module loading sequence

Given the architecture outline in figure 1, the loading of the different RTAI and LTT modules has to follow a certain order. If this order is not followed, the loading of the modules will fail. The following assumes that the system is running an RTAI patched kernel configured with RTAI tracing enabled.
The first module to be loaded is the RTAI trace facility as all other RTAI modules need the services provided by this module in order to trace events. Remember that the trace statements inserted into the different execution paths of the different RTAI modules generate code which ends up calling the services RTAI trace facility, more precisely the `rt_trace_event()` function. Once it is loaded, the rest of the modules can be loaded following the order required by the architecture of RTAI.
The second module to be loaded is optional and, in a sense, it does not have to be the second module loaded. It is the trace module or trace driver (in the code, it is often refereed to as the "tracer"). If one tried to load this module prior to the loading of the RTAI trace facility, the module installation service would complain about unresolved symbols. The reason is that the trace module registration functions offered by the RTAI trace facility would be missing. Once the trace module is loaded, it registers with the Linux kernel trace facility and the RTAI trace facility. Thereafter, the trace daemon can be activated and events logged.
The third module to be loaded is the RTAI core, often compiled simply as "rtai". The rest of the modules loaded depends on the use to be made of RTAI. Usually, the next module is the RTAI scheduler often followed by the FIFO facility.

### 4.3 Tracing the system

Once all the modules have been loaded, tracing can start. In order to trace a given set of tasks compiled within a single module or a set of processes set to become hard real-time using LXRT, the sequence of operations to be conducted is the following:

1. Start the trace daemon with the appropriate arguments. Most importantly, set tracing duration, if any. During this step, output files have to be provided in order to store the traces generated.

2. Load the required task module(s) or start the required processes.

3. If the trace daemon was not started with a time limit, stop the trace daemon once the traced events have occurred.

4. If the trace is to be analyzed on the same machine as the one on which it was generated, it is recommended that the real-time tasks traced be stopped. This, since real-time tasks have priority on all other tasks as to CPU control. Therefore, running real-time tasks while trying to run normal Linux processes might hinder the latter's performance.

That done, it is now possible to view the generated traces.

### 4.4 Viewing collected traces

Given the generated trace file and system description, the visualization tool will display a window with three main thumbnails. The first contains the control graph. The second contains the statistics. The third contains the raw list of events in full detail. Note that the data displayed in graphical form can also be outputed in a text file using the visualization tool as a command line tool. The generated file can then undergo further analysis using scripts. The next section will provide examples of the screens mentioned above.

## 5 Examples

This section's purpose is to provide real life examples of how LTT can be used to understand the behavior of an RTAI/Linux system. Each of the examples chosen shows how different situations can profit from the insight LTT provides. Section 5.1 presents the usage of LTT on the "jepplin" example provided in the examples directory within the RTAI source code. Section 5.2 shows how regularity occurs within the "task timer" example. Section 5.3 discusses how LTT was used to show how RTNet [2] achieves hard real-time networking performance.

### 5.1 Jepplin

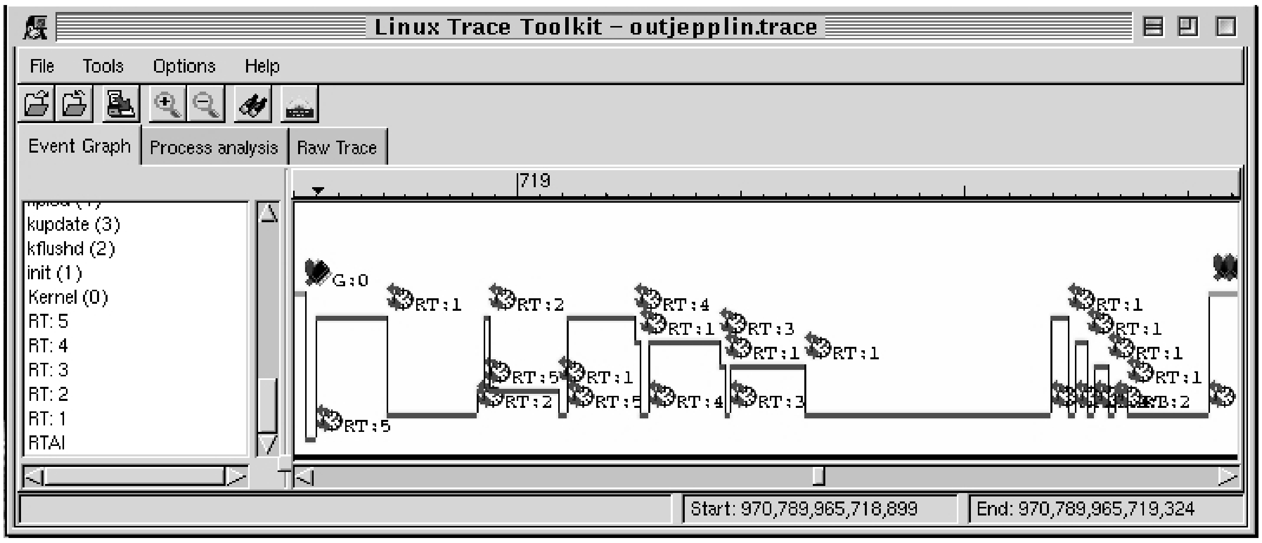
The jepplin example consists of a set of real-time tasks communicating through mailboxes and semaphores. The scheduling of the tasks depends on the messages they are waiting for. Inversely, the messages sent by a task can trigger the scheduling of another real-time task. Figure 3 displays a portion of the control-graph of the transitions due to the jepplin tasks interacting. Figure 4 displays the corresponding list of raw events. We can see the state transitions and the corresponding detail of the events and how they impact the system's dynamics. Most importantly, we can see how the communication between the tasks generates scheduling changes. If there were any synchronization problems, this type of view would help in isolating the problem

### 5.2 Task timer

The task timer example's purpose is to show that RTAI is able to "guarantee an effective processing with a contained jitter and high effectiveness", as the explanation reads. It launches three tasks during initialization, the first two are meant to be scheduled at regular intervals to do calculations and the third is used to print out information about the executions times. The point here is to test RTAI given a certain load. Figure 5 shows that the task scheduling follows a regular pattern with correct timings. Figure 6 shows the statistics for the execution of one of the two processing tasks. Here, the task is scheduled for almost 28% of the time. Also, not showed, LTT shows that the first task has been granted an equal percentage of the cpu share, 28%. The results give a good appreciation about the time taken to do a calculation loop by each task and confirm that RTAI was able to guarantee effective processing. This type of result could further be used to measure calculation loops and apply the necessary corrections in case the system does not accomplish the required processing in a "reasonable" time.

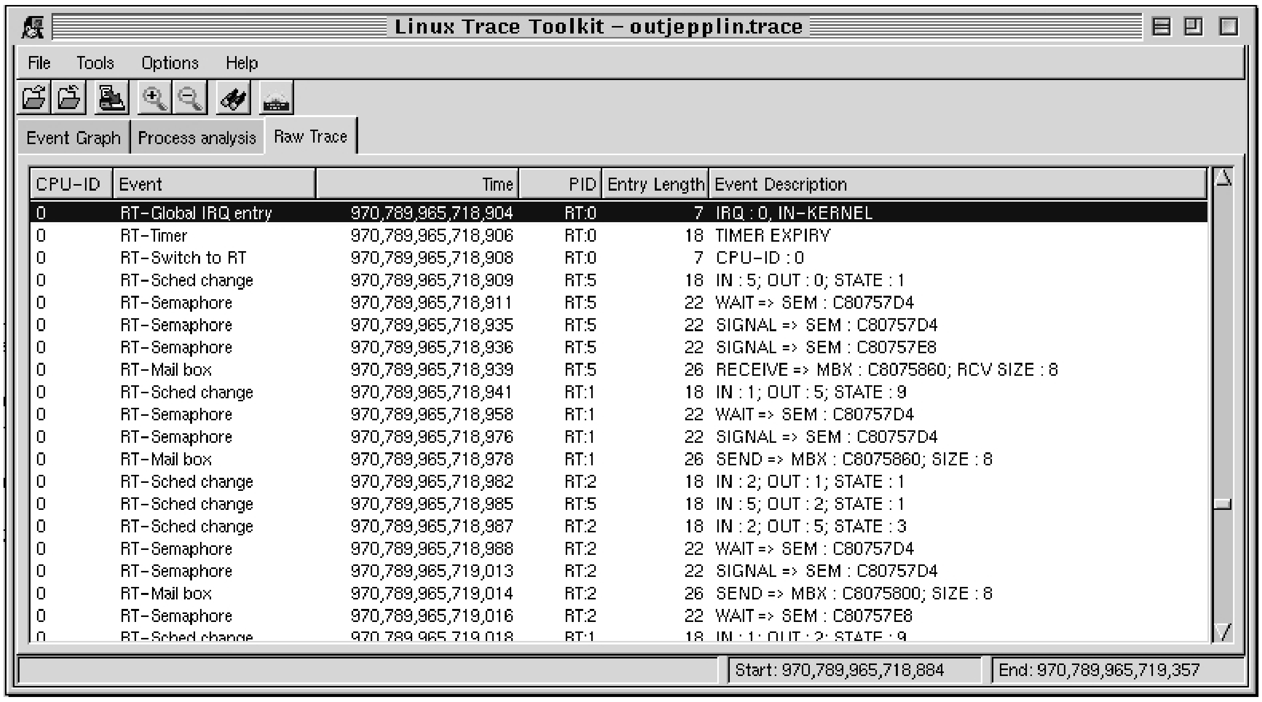
### 5.3 RT-Net

RTNet is a package written by David Schleef that enables hard real-time communication to take place using an ethernet link. The programming interface provided by RTNet is very similar to the BSD socket interface available on most Unix-like systems. In order to test RTNet's capabilities, the client/server example provided with the RTNet source code was used.



Start: 970,789,965,718,899
End: 970,789,965,719,324

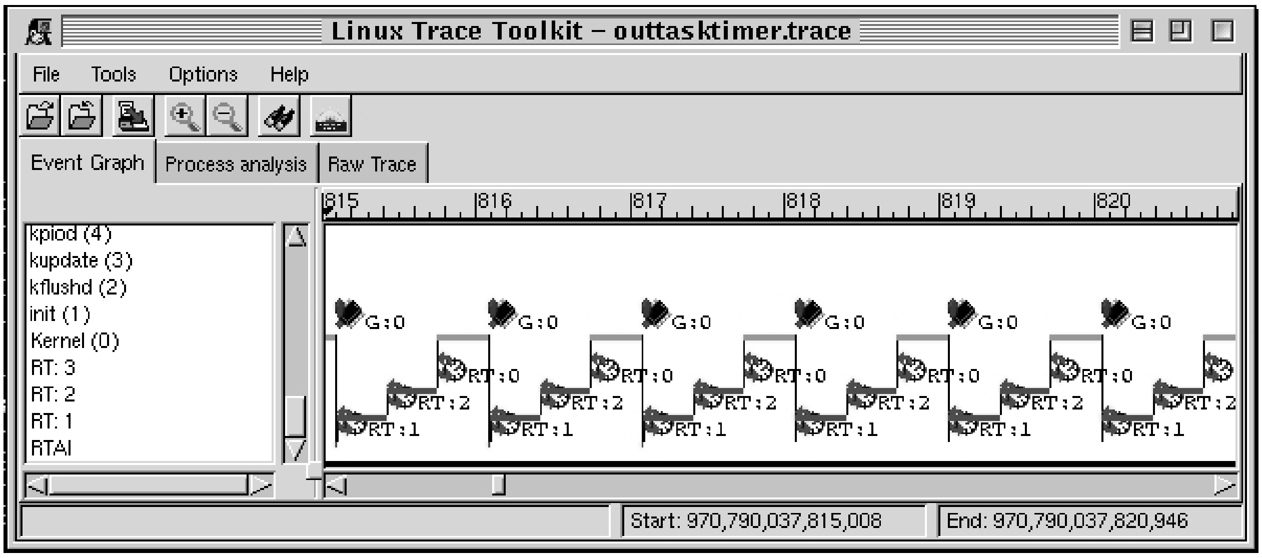
Figure 3: Jepplin communication control-graph.



| CPU-ID | Event | Time | PID | Entry Length | Event Description |
|---|---|---|---|---|---|
| 0 | RT-Global IRQ entry | 970,789,965,718,904 | RT:0 | 7 | IRQ : 0, IN-KERNEL |
| 0 | RT-Timer | 970,789,965,718,906 | RT:0 | 18 | TIMER EXPIRY |
| 0 | RT-Switch to RT | 970,789,965,718,908 | RT:0 | 7 | CPU-ID : 0 |
| 0 | RT-Sched change | 970,789,965,718,909 | RT:5 | 18 | IN : 5; OUT : 0; STATE : 1 |
| 0 | RT-Semaphore | 970,789,965,718,911 | RT:5 | 22 | WAIT => SEM : C80757D4 |
| 0 | RT-Semaphore | 970,789,965,718,935 | RT:5 | 22 | SIGNAL => SEM : C80757D4 |
| 0 | RT-Semaphore | 970,789,965,718,936 | RT:5 | 22 | SIGNAL => SEM : C80757E8 |
| 0 | RT-Mail box | 970,789,965,718,939 | RT:5 | 26 | RECEIVE => MBX : C8075860; RCV SIZE : 8 |
| 0 | RT-Sched change | 970,789,965,718,941 | RT:1 | 18 | IN : 1; OUT : 5; STATE : 9 |
| 0 | RT-Semaphore | 970,789,965,718,958 | RT:1 | 22 | WAIT => SEM : C80757D4 |
| 0 | RT-Semaphore | 970,789,965,718,976 | RT:1 | 22 | SIGNAL => SEM : C80757D4 |
| 0 | RT-Mail box | 970,789,965,718,978 | RT:1 | 26 | SEND => MBX : C8075860; SIZE : 8 |
| 0 | RT-Sched change | 970,789,965,718,982 | RT:2 | 18 | IN : 2; OUT : 1; STATE : 1 |
| 0 | RT-Sched change | 970,789,965,718,985 | RT:5 | 18 | IN : 5; OUT : 2; STATE : 1 |
| 0 | RT-Sched change | 970,789,965,718,987 | RT:2 | 18 | IN : 2; OUT : 5; STATE : 3 |
| 0 | RT-Semaphore | 970,789,965,718,988 | RT:2 | 22 | WAIT => SEM : C80757D4 |
| 0 | RT-Semaphore | 970,789,965,719,013 | RT:2 | 22 | SIGNAL => SEM : C80757D4 |
| 0 | RT-Mail box | 970,789,965,719,014 | RT:2 | 26 | SEND => MBX : C8075800; SIZE : 8 |
| 0 | RT-Semaphore | 970,789,965,719,016 | RT:2 | 22 | WAIT => SEM : C80757E8 |
| 0 | RT-Sched change | 970,789,965,719,018 | RT:1 | 18 | IN : 1; OUT : 2; STATE : 9 |

Start: 970,789,965,718,884
End: 970,789,965,719,357

Figure 4: Jepplin communication events list.

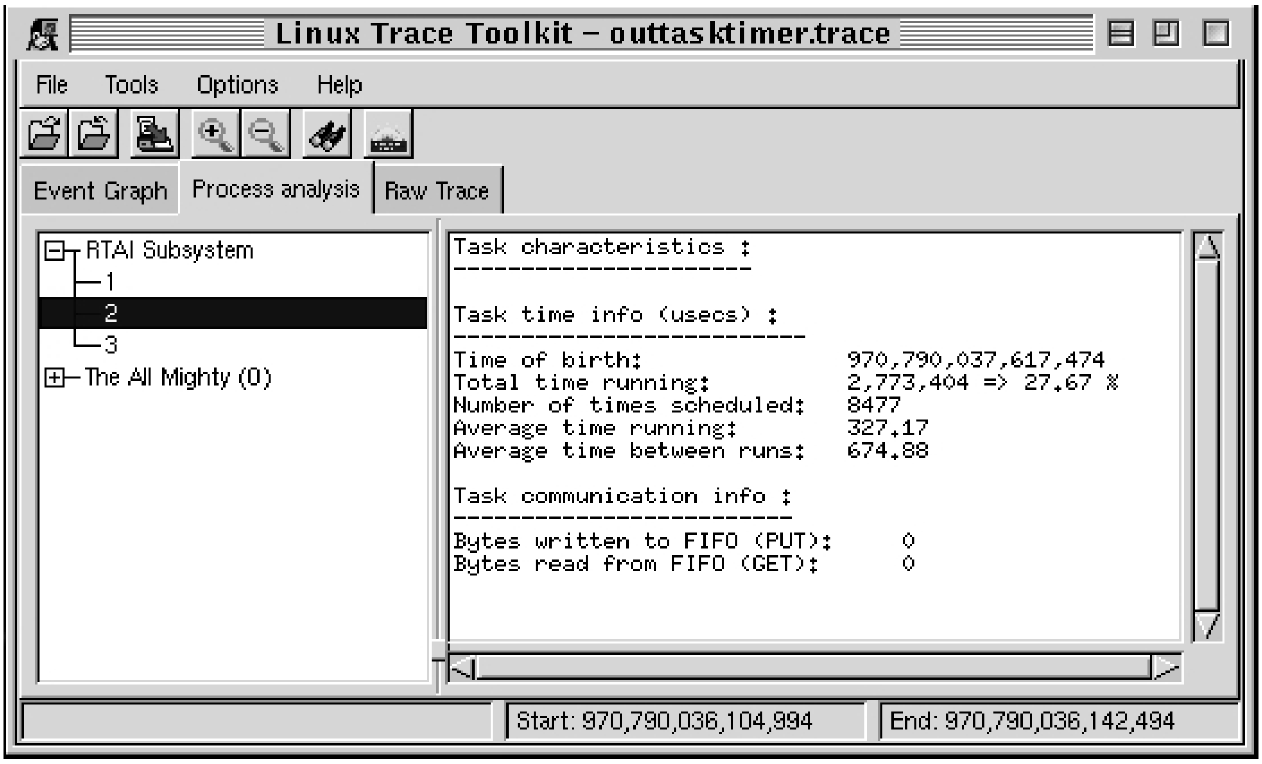


kpiod (4)
kupdate (3)
kflushd (2)
init (1)
Kernel (0)
RT: 3
RT: 2
RT: 1
RTAI

Start: 970,790,037,815,008

End: 970,790,037,820,946

Figure 5: Task timer graph of interrupt occurrences and resulting task scheduling.



```
Task characteristics :
----------------------

Task time info (usecs) :
--------------------------
Time of birth:               970,790,037,617,474
Total time running:          2,773,404 => 27.67 %
Number of times scheduled:   8477
Average time running:        327.17
Average time between runs:   674.88

Task communication info :
-------------------------
Bytes written to FIFO (PUT):     0
Bytes read from FIFO (GET):      0
```

Start: 970,790,036,104,994

End: 970,790,036,142,494

Figure 6: Task timer statistics.

Two hosts were linked using an 100Mbits Ethernet connection using a crossover cable and the client host was monitored using LTT. Figure 7 presents the control-graph of the sequence of events that occurred on the client as he sent a packet and received a reply.
On the graph, we can see the system timer firing off (IRQ 0) and waking up the client (the client was set up during it's initialization to be called once every second). The client sends something on the net. This generates an IRQ 9. RTAI then decides to schedule task 0, the Linux kernel, since the kernel timer function has to be called. This results in "klogd" (PID = 453) to be rescheduled. klogd calls on the "time" system call but while the kernel is dealing with this call, another IRQ 9 occurs. This one marks the reception of the echo from the server. Immediately, RTAI schedules the reception function of the client which does some processing and yields CPU control. RTAI then hands the CPU back to Linux which can continue to process klogd's system call.
Figure 8 shows the list of interrupt occurrences we got from the visualization tool, used as a command-line tool, to dump the trace into a file in text format by providing the correct flags to filter the interesting events.
These should always be considered in couples. The first IRQ marks the sending and the second one marks the reception. Notice that round-trip communication always stays between 63 to 65 microseconds. Hence the deterministic nature of RTNet communication.

## 6 Related work

Although LTT support for RTAI is the first trace support available for a real-time Linux variant, it isn't the first tracing system to exist. Tracing has actually been around for some time, though no tracing tool other than LTT is available under the GNU GPL [3] or any other form of open-source license. Nonetheless, it is important to stress that some of these commercial systems provide capabilities still unavailable in LTT. Also, note that prior to enabling RTAI tracing, LTT was first and foremost a toolkit to enable tracing of the Linux kernel, a capability previously unavailable.
In the realm of general purpose operating systems, we find, apart from LTT, that IBM and SUN have tracing systems. IBM has been providing for some time now a tracing facility for it's AIX system [4]. The main advantage of that system is its flexibility as it is fairly easy to add new events through the */etc/trcfmt* file. IBM does not have any tool to graphically view or analyze the event sequences although some visualization research projects do use AIX's facility to retrieve data to be graphically displayed [5]. SUN uses a format they call TNF (Trace Normal Form) on which they base their tracing system and tools [6]. Although, the tool set provided by Sun contains a graphical viewer, the information displayed is hard to follow and does not match the quality of display provided by visualization tools available for commercial RTOSes.
In the embedded/real-time world, a slew of tracing tools are available from different vendors. First and foremost, WindRiver's WindView system [7]. It is by far the most elaborate tracing and visualization tool available providing graphical configuration of the tracing process and different levels of configurability of the trace analysis. QNX, another commercial RTOS vendor, has a tracing tool called DejaView [8]. Apart from a press release, the author was unable to find further details about this tool. Other tools available for embedded/ real-time debugging include Etnus' TimeScan [9], Lauterbach's Trace32 [10], TimeSys' TimeTrace [11] and Nematron's Hyperkernel trace utility [12]. Although this is not an exhaustive list, it does show that there are many tracing tools available for the development of real-time/embedded systems.

## 7 Future directions

The Linux Trace Toolkit has been successfully used by many to understand the behavior and dynamics of the Linux kernel for some time now. Lately, RTAI support has enhanced LTT's capabilities and provided the real-time Linux community with a flexible and effective tracing tool. As usage grows and new applications for LTT are found, there is a need to extend and augment it. There are many areas of evolution where LTT can/will evolve.
First and foremost, extending support for other platforms than the i386. By this, the aim is to bring tracing support to the other architectures on which the Linux kernel can run. Parallel to this and given the abstractions built into the analysis and visualization tool, LTT will aim at adding tracing support for other operating systems. Mainly, open-source kernels such as BSD and GNU Hurd's base, Mach. This, though, does not exclude commercial vendors from instrumenting their own kernels and making available the code necessary for their traces to be analyzed by LTT under the GPL.
Given the current market trends and, consequently, users' needs, adding remote-tracing capabilities will be important. This will enable users to trace remote systems without needing to store the cumulated traces on the traced system. This is most im-

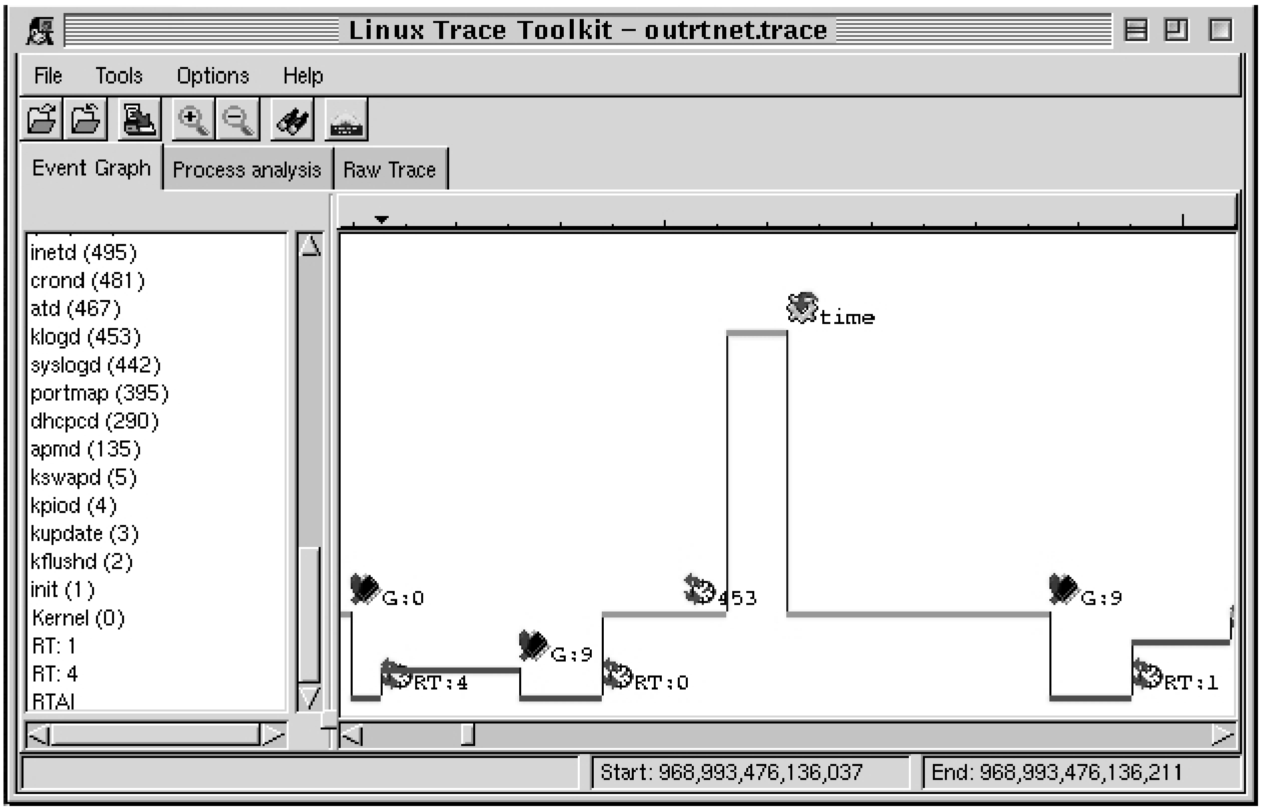


Figure 7: RNet example: Client's sending and receiving.

```
RT-Global IRQ entry     968,993,482,135,788     0       7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,482,135,853     453     7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,483,135,578     0       7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,483,135,643     453     7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,484,135,364     0       7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,484,135,429     453     7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,485,136,148     0       7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,485,136,212     453     7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,486,135,939     0       7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,486,136,004     453     7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,487,135,731     0       7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,487,135,796     453     7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,488,135,522     0       7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,488,135,584     453     7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,489,135,304     0       7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,489,135,368     453     7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,490,135,091     0       7       IRQ : 9, IN-KERNEL
RT-Global IRQ entry     968,993,490,135,153     453     7       IRQ : 9, IN-KERNEL
```

Figure 8: RTNet example: List of client network card interrupts.

portant for embedded systems as they often do not have permanent storage devices. Along with this, the capability of remotely configuring tracing will be necessary.
Since LTT might be used for custom event tracing, the addition of custom created events is a priority. This will enable developers to use the tracing capabilities to trace events that they create and that have a specific meaning according to their system.

## 8 Conclusion

This paper has introduced and discussed the tracing of the RTAI real-time Linux system using the Linux Trace Toolkit. The paper has shown that LTT is based on a modular and extensible system behavior tracing architecture. The traces generated by this system have been shown to recreate actual system behavior with great accuracy. Also, tracing an RTAI/Linux system using LTT is relatively simple, which makes it accessible to a large audience. Given the popularity of real-time Linux systems and their inherent complexity, LTT for RTAI will likely be helpful to a large number of real-time Linux enthusiasts.

## References

[1] Linux Trace Toolkit home page, http://www.opersys.com/LTT.

[2] RT-Net, see: http://opensource.lineo.com/rtai.html.

[3] GNU General Public License version 2, http://www.gnu.org/copyleft/gpl.html.

[4] IBM's AIX trace facility, see: http://anguilla.u.arizona.edu/doc_link/en_US/a_doc_lib/aixprggd/genprogc/trace_facility.htm.

[5] Program Visualization at IBM Research http://www.research.ibm.com/pvres/.

[6] SUN's Trace Normal Format tools, http://www.sun.com/smcc/solaris-migration/docs/whitepapers.html#TNF.

[7] WindRiver's WindView, http://www.windriver.com/products/html/windview2.html.

[8] QNX's DejaView, http://www.qnx.com.

[9] Etnus' TimeScan, http://www.etnus.com/products/tsproduct/index.html.

[10] Lauterbach's Trace32, http://www.lauterbach.com.

[11] TimeSys' TimeTrace, http://www.timesys.com.

[12] Nematron's HyperKernel Trace Utility, http://www.nematron.com/solutions/software/hyperkernel/hyperkernel.html.

[13] K. Yaghmour and M. Dagenais. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.