WP5 - Real-Time Scheduling Components

Deliverable D5.4_rep - Scheduling Components V2

WP5 - Real-Time Scheduling Components: Deliverable D5.4_rep - Scheduling Components V2

by Ismael Ripoll, Alfons Crespo, Patricia Balbastre, Josep Vidal, Miguel Masmano, and Alejandro Lucero

Published Febrery 2004

Table of Contents

Document presentation ..... **i**
1. Introduction **1**
2. POSIX Execution Time Timers (ExecTimers) **3**
2.1. Summary 3
2.2. Descritpion 3
2.3. Author 3
2.4. Layer 4
2.5. API / Compatibility 4
2.6. Dependencies 4
2.7. Status 4
2.8. Implementation issues 4
2.9. Validation criteria 5
2.10. Tests 5
3. Lightweight POSIX Trace (LPTrace) **7**
3.1. Summary 7
3.2. Description 7
3.2.1. The MRSP Model and its Tracing Limitations 7
3.2.2. Proposal of a Lightweight POSIX Trace System 8
3.2.2.1. Removal of Restrictions 9
3.2.2.2. Units of Functionality 9
3.2.3. Implementation in RT-Linux 11
3.3. API / Compatibility 12
3.4. Implementation issues 14
3.5. Tests and validation 15
3.5.1. Validation criteria 15
3.5.2. Test 1 15
3.5.3. Test 2 17
3.5.4. Test 2 18
3.5.5. Results and comments 19
3.6. Examples 19
4. Metrics(Metrics) **20**
4.1. Summary 20
4.2. Description 20
4.3. API / Compatibility 20
4.4. Implementation issues 23
4.5. Tests and validation 25
4.5.1. Validation criteria 25
4.5.2. Tests 26
4.6. Example 26
5. RT-Terminal (RTTerminal) **28**
5.1. Summary 28
5.2. Descritpion 28
5.3. Layer 28
5.4. API / Compatibility 28
5.5. Dependencies 29
5.6. Status 29
5.7. Implementation issues 29
5.8. Validation criteria 30
5.9. Tests 30
5.10. Example 31
6. RTLinux UDP/IP (RTLUDP) **33**
6.1. Summary 33

6.2. Description 33
6.3. Layer 34
6.4. API / Compatibility 34
6.5. Dependencies 34
6.6. Status 34
6.7. Implementation issues 34
6.8. Validation criteria 36
6.9. Tests 36
7. IDE Device Driver (RTLide) 38
7.1. Summary 38
7.2. Description 38
7.3. Layer 39
7.4. API / Compatibility 39
7.5. Dependencies 39
7.6. Status 39
7.7. Implementation issues 40
7.7.1. Configuration 40
7.7.2. Mode of operation 41
7.8. Validation Criteria 41
7.9. Tests 41
8. RTLinux Disk scheduler and file system (RTLfs) 43
8.1. Summary 43
8.2. Description 43
8.3. Layer 43
8.4. API / Compatibility 43
8.5. Dependencies 44
8.6. Status 44
8.7. File System Description 44
8.8. Available open source realtime filesystems 47
8.9. Real Time File System (RTFS) Specification 48
8.10. Features 50
8.11. Validation Criteria 51
8.12. Tests 51
8.13. APENDIX A 51
9. Stand-Alone RTLinux-GPL (saRTL) 53
9.1. Summary 53
9.2. Description 53
9.3. API / Compatibility 54
9.4. Implementation issues 54
9.5. Validation criteria 57
9.6. Tests 57
10. Stand-Alone RTLinux Memory Protection (saRTLmprot) 59
10.1. Summary 59
10.2. Description 59
10.3. API / Compatibility 60
10.4. Implementation issues 60
10.4.1. Executive protection implementation 61
10.4.2. Context Protection Implementation 62
10.5. Validation criteria 63
10.6. Tests 64
11. Stand-Alone RTLinux debugging tools (debugtools) 65
11.1. Summary 65
11.2. Description 65

11.3. API / Compatibility 65
11.4. Implementation issues 66
11.4.1. New GDB Agent funcionalities 66
11.4.1.1. Extensions to the standard GDB agent 67
11.4.2. Tracer implementation 67
11.5. Validation criteria 69
12. Stand-Alone RTLinux-GPL porting to StrongARM processor (saRTLarm) 70
12.1. Summary 70
12.2. Description 70
12.3. API / Compatibility 70
12.4. Implementation issues 71
12.5. Validation criteria 73
12.6. Tests 73
Bibliography 74

List of Tables

1. Project Co-ordinator i
2. Participant List i
3-1. Results for test 1 with complete POSIX trace support 17
3-2. Results for test 1 with minimal trace support 18
3-3. Results for tracing system events with complete and minimal trace support 18
9-1. saRTL POSIX compliant funcionalities implemented 54

List of Figures

3-1. Main configuration menu of POSIX Trace support 11
3-2. Maximums and limits menu 12
3-3. Selection of system event types 12
3-4. Specific RT-Linux options 12
3-5. Contribution of individual options to the RT-Linux kernel memory footprint 16
3-6. Memory footprint of tracing Units of Functionality 16
5-1. RT-Terminal example 32
6-1. RTLUDP component 33
7-1. Access to the hard disk. 38
7-2. Contiguos memory allocation 41
8-2. RTLFS structure 48
8-5. Super block 51
8-6. I-node structure 51
10-1. Executive memory protection layout 61
10-2. Example of two protected contexts 63
11-1. DDD Snapshot 67
11-2. Execution Trace with GTKWave 68
12-1. Processor type configuration 70
12-2. CerfPod Evaluation Board 71

Document presentation

Table 1. Project Co-ordinator

Organisation:	UPVLC
Responsible person:	Alfons Crespo
Address:	Camino Vera, 14. CP: 46022, Valencia, Spain
Phone:	+34 9877576
Fax:	+34 9877579
E-mail:	alfons@disca.upv.es

Table 2. Participant List

Role	Id.	Name	Acronym	Country
CO	1	Universidad Politécnica de Valencia	UPVLC	E
CR	2	Scuola Superiore S. Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	CEA	CEA	FR
CR	5	UNICONTROLS	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	VISUAL TOOLS S.A.	VT	E

Chapter 1. Introduction

In this second phase, our effort has focussed more on adding new functionality than on the core POSIX features. All the components designed and implemented, where directly related to RTLinux.

The components developed can be grouped into tree main categories:

Process monitoring

ExecTimers: extend the POSIX timers component to include POSIX CPU clocks.

LPTrace: A Lightweight version of the POSIX tracing standard. This component also tries to contribute with the standard presenting a revision of it that fits small embedded systems restrictions.

Metrics: Set of tools to extract useful measures from raw POSIX traces.

Basic drivers

RTTerminal: Direct console input and output. This component works in the standard RTLinux and also in the stand-alone RTLinux.

RTLUDP: Compact UDP/IP stack that uses the etherboot large set of low level Ethernet card drivers.

RTLide: Low level IDE hard disk driver.

RTLfs: A fast, simple, and robust filesystem.

Small footprint executive

saRTL: Stand Alone RTLinux. The RTLinux executive running in a i386 machine without Linux on top.

saRTLmprot: protection in Stand Alone. Two memory protection schemes with very few overhead.

debugtools: High level debugging tool for the Stand Alone RTL.

saRTLarm: Porting of the Stand Alone RTL to the StrongARM processor. saRTL ported to a completely different board and processor system.

Besides the work done on the components of this phase, the components of the first phase has been also maintained and updated. New releases of the dynamic memory allocator (DYNMEM) has been published with new improvements on the code and data structure (new implementation achieves better the cache locality: compacted code and data). Also the GNAT compiler porting to RTLinux has been updated to work with the latest version of the compiler 3.15.

Each component is described in a separate chapter. The information of each component is organised as follows:

Summary

A summary of the most important information of the component. The information provided is:

- Name of the component.
- Description
- Author/s
- Reviewer
- Layer
- Version
- Status
- Dependencies
- Release date

Description

A detailed description of the component, and a rationale about its role and interest in a real-time operating system.

Some components implement well known facilities which are available in most RTOS but not in RTLinux or Linux. These cases do not need special justification nor description and these sections may be shorter.

API / Compatibility

The new API provided by the component if applicable. Also the compatibility with the standards.

Implementation issues

Key aspects of the internal design of the component will be presented in this section.

Tests and validation criteria

All the code produced in the OCERA project must be intensively tested and validated. All the functionality must be validated both reviewing the code by external developers (other partner); and also by building a complete test suite. The tests will be implemented to validate specific criteria (overhead, etc.).

Examples

Simple examples provided by the software will help OCERA user to start its own applications.

Chapter 2. POSIX Execution Time Timers (ExecTimers)

2.1. Summary

Name

POSIX Execution Time Clocks and Timers(**ExecTimers**)

Description

Provides POSIX CPU Time clocks and timers (ADVANCED REALTIME THREADS, IEEE Std 1003.1-2001).

Author/s

Josep Vidal

Reviewer

Ismael Ripoll

Layer

Low level RTLinux.

Version

0.1

Status

Testing

Dependencies

???????

Release Date

M3

2.2. Descritpion

When performing a schedulability test, scheduling analysis relies on a known worst-case execution time (WCET). Nevertheless, estimating WCET is a difficult task due to the different execution paths within a program and today's computer architectures. Specially, in the context of concurrent programs in which cache misses are frequent after interrupt service routines or context switches.

Unfortunatelly the tasking model of most concurrent hard real-time systems, do not enforces the bound on the execution time of tasks. Without bound on execution time, a task could execute more than estimated, causing other tasks to loose its deadlines. This, on most hard real time application, may result in catastrophic consequences.

This component provides a solution to this problem, implementing execution time timers (as defined in the IEEE Std 1003.1-2001, ADVANCED REALTIME THREADS) within the task scheduler. This kind of timers may be used to detect execution time overruns in the application, and to limit their effects.

If a timer is created using a CPU-Time clock of a particular thread, and a relative expiration time is given, it can be used to notify that a certain budget of execution time has elapsed, for that thread. If the timer is armed each time a thread is activated, and the relative expiration time is set to the thread's estimated worst-case execution time (plus some small amount to take into account the limited resolution and precision of the CPU-Time clock), then the timer will only expire if the thread suffers an execution time overrun.

2.3. Author

Josep Vidal Canet jvidal@disca.upv.es

2.4. Layer

Low level RTLinux.

2.5. API / Compatibility

The execution time clocks interface defined in the proposed standard POSIX.1d is based on the POSIX.1b clocks and timers interface used for normal real time clocks. The new interface creates a only new function to access the execution time clock identifier of the desired thread: `pthread_getcpuclockid()`. In addition, it defines a new thread-creation attribute, called `cpu_clock_requirement`, which allows the application to enable or disable the use of the execution time clock of a thread, at the time of its creation. Once the thread is created, this attribute cannot be modified. To use CPU-time clocks for threads, we must set the `cpu_clock_requirement` attribute to the value `CLOCK_REQUIRED_FOR_THREAD`.

```
 #include <time.h> int
pthread_getcpuclockid(pthread_t *thread, clockid_t
*clock_id);
```

2.6. Dependencies

Depends on psignals and ptimers components.

2.7. Status

Testing.

2.8. Implementation issues

POSIX defines the execution time as the time spent executing a process or thread, including the time spent executing system services on behalf of that thread. Due to the fact that in RTLinux all threads run in kernel space, system calls are implemented as simple function calls which are fully executed in the context of the calling thread. This OS characteristic allows to implement CPU-Time execution timers in a very simple and efficient way.

The implementation of CPU-Time clocks and timers in RTLinux requires modification of the data structure that defines each thread, the thread control block, modifications of the scheduler code to include the necessary steps to update each thread's CPU-Time clock and modifications to the timers management code to operate with the CPU-Time clocks associated timers.

The information that has been added to the thread control block consists of:

- A structure with the information needed for the CPU-Time clock, including the clock identifier and the total CPU-Time consumed by that thread.
- A high resolution time member (long long) to store the time of the last activation

The main modifications required to support CPU-Time clocks and timers are focused on the scheduler API and POSIX timers API.

Modifications required to the scheduler in order to support CPU clocks consists of adding code at two places: 1) the point where a new thread becomes the running thread; and 2)

to the function that finds next preemptor thread in order to determine when the timers expire.

The point where a new thread becomes the running thread in RTLinux is located at the scheduler function (`rtl_schedule()`) just before the context switch is performed. At this point, what is done is to accumulate the execution time of the current thread and anotate the activation time of the thread that is going to take the CPU.

Finally, the find preemptor function, has been modified to handle the situation where the thread that is going to be executed has an execution time timer armed. This modification consists in programing the scheduler timer to shot at the time in which the execution time timer expires (taking into account higher priority thread timers).

2.9. Validation criteria

This OS facility allows to implement today's scheduling algorithms such as CBS and sporadic server schedulers in a more efficent and reliable way, specially at application-level with the use of POSIX-Compatible Application-defined Scheduling (available in RTLinux as an OCERA component).

As said before, execution time timers may be used to increase the fault-tolerance of critical applications due to a system timing malfunction not detected by the off-line analysis techniques.

Finally, the use execution time timers could help to estimate WCET (Worst Case Execution Time) based on statistical analysis.

As in most of the low level components, the implementation of execution time timers tries to fit two objectives: minimize the overhead introduced in RTLinux runtime and try to achieve the best efficiency. In addition to this, an acceptable timer resolution respect the available hardware should be reached.

The overhead introduced to RTLinux runtime due to the use of CPU clocks is negligible (less than 0.01 %) as we will see in test section. Execution time timers precision is near to a few microseconds, as in normal POSIX timers. This is due to the fact that the low level system timer is the same for both.

Finally, execution time timers implementation guarantees the immediacy of the timer expiration notification (similar to a context witch).

2.10. Tests

Tests have been designed to check CPU clocks and execution time timers RTLinux implementation correctness, accuracy and overhead. From the Open POSIX Test Suite project, external tests have been addapted to test CPU clocks implementation. Unfortunatelly no external tests suites have been found to test execution time timers.

- The POSIX Test Suite is an open source test suite with the goal of performing conformance, functional, and stress testing of the IEEE 1003.1-2001 System Interfaces specification in a manner that is agnostic to any given implementation.

  Among other POSIX functionalities, these suite support CPU clocks testing. We have used some of this tests slightly modified to run on RTLinux.

- To measure the overhead introduced by CPU clocks, the Baker utilization has been passed.

  This test allows to measure runtime s overhead scheduling six armonic tasks. Armonic tasks have the following periods: 1/320HZ, 2/320HZ, 4/320HZ, 8/320HZ, 16/320HZ and 32/320HZ= 100 miliseconds respectively. In each test iteration (every second) tasks load (CPU consumption) is increassed by an amount. Test finishes when a task losses its deadline. At this point, the Utilization is calculated taking tasks load from previous iteration.

From this test results, it could be stated that the overhead introduced when using CPU clocks is negligible (less than 0.01 %).

- Timers accuracy tests developed while implementing POSIX timers have been adapted to measure execution time timers precision. This tests shows that execution time timers precision is the same as normal POSIX timers. This is due to the fact that the low level system timer is the same for all clocks. This precision is near to the microsecond and depends on the available hardware.
- Basic CPU timers functionality has been tested with self built tests.

  Among other tests, there is a simple one that shows how the use of execution time timers helps to increase the fault-tolerance of applications due to timing malfunctions. In this tests a execution time timer is armed to expire after the highest priority task execution time reaches one second. At that momenent the execution time timer handler suspends the task in order to allow other tasks to take the CPU.

  On the other hand, the highest priority task is doing and endless loop wasting time. If the execution time timer handler doesn't suspends wasting time task, Linux will hang and we will loose control over the computer.

Chapter 3. Lightweight POSIX Trace (LPTrace)

3.1. Summary

Name
: Lightweight POSIX Trace

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
: Ismael Ripoll

Layer
: Low level RTLinux and Linux

Version
: 2.0

Status
: Stable

Dependencies
: RTLinux 3.2-pre1

Release Date
: M3

3.2. 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.

3.2.1. 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.

3.2.2. 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 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".

3.2.2.1. 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.

3.2.2.2. 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.

3.2.3. 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 3.2.2.1, *Removal of Restrictions*, are done in several configuration menus, which are now presented in the following figures.

Figure 3-1. Main configuration menu of POSIX Trace support

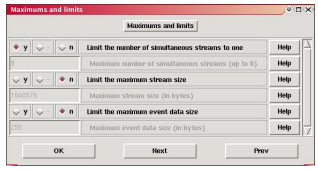

Figure 3-2. Maximums and limits menu

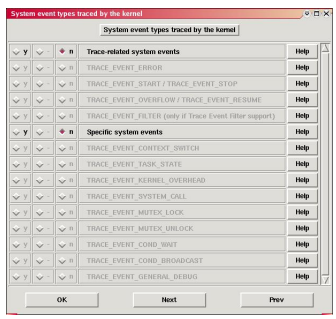

Figure 3-3. Selection of system event types

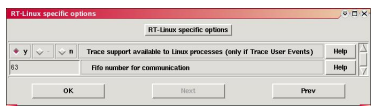

Figure 3-4. Specific RT-Linux options

3.3. 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 3.2.2.1, *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_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_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 *);
```

3.4. 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 3.2.2, *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 restrictions stated in Section 3.2.1, *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 3.2.3, *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.

3.5. Tests and validation

3.5.1. 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.

3.5.2. 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.

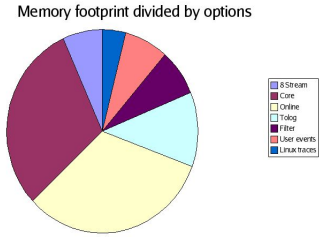

Figure 3-5. Contribution of individual options to the RT-Linux kernel memory footprint

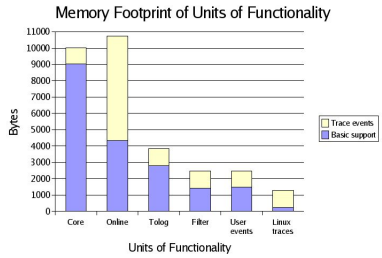

Figure 3-6. Memory footprint of tracing Units of Functionality

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.

3.5.3. 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 experiment 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 3-1. 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 3-2. 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`).

3.5.4. 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 3-3. 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

Measured Value	Minimum	Maximum	Average	Variance
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).

3.5.5. 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.

3.6. 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.).

Chapter 4. Metrics(Metrics)

4.1. Summary

Name
: Metrics

Description
: This component is a library capable of extracting measurements of system metrics from POSIX trace streams.

Author/s
: Agustin Espinosa, Andres Terrasa, Ana Garcia-Fornes.

Reviewer
:

Layer
: Library level RTLinux and Linux

Version
: 1.0

Status
: Beta

Dependencies
: Lightweight POSIX Trace

Release Date
: M3

4.2. Description

The POSIX tracing services specify a set of interfaces to allow for portable access to underlying trace management services by application programs. Programmers may use these services to get a sequence of trace events generated by the system during the execution of their application. These trace events are kept in a POSIX trace stream. The contents of a trace stream can be analyzed while the tracing activity takes place or they can be analyzed later, once the tracing activity has been completed. A trace event is generated when some action takes place in the system and this trace event may be stored in one or several trace streams. Each trace event contains data which is relative to the action that has generated it. The POSIX tracing services require that data such as event type, time stamp, process identifier and thread identifier to be associated with each trace event. Using these data, we can get time related system metrics such as the execution time of a given system call, the response time of a periodic job, etc.

Unfortunately, the interpretation of the events which are stored in trace streams may be difficult for programmers who do not know the system implementation in detail. Events stored in a trace stream represent system actions such as context switches, hardware interrupts, state changes, etc. In order to extract metrics from these events, it is necessary to know how the execution of the system generates these events, and normally this information is only known by the system designer. In order to solve this problem, this component implements a metric extraction engine and provides an application interface for using this engine. This interface allows the programmer to obtain predefined system metrics from trace streams without it being necessary for the programmer to know the system implementation.

4.3. API / Compatibility

The application interface for this component is as follows:

Metrics are used in this interface by means of metric identifiers, which are objects of the `metrics_metric_id_t` type. This library offers a fixed set of metrics and each metric has its own name, for example "M_JOB_RESPONSE_TIME". The user of this library shall provide a predefined metric name in order to get a valid metric identifier. These identifiers can be retrieved by using the following function:

```
int metrics_metric_open(const char *metric_name, metrics_metric_id_t *metric_id);
```

Metric identifiers can be grouped in metric sets, which are objects of the `metrics_metricset_t` type. A program uses these sets in order to define the metrics which shall be extracted from a trace stream. The following functions can be used in order to manipulate metric sets:

```
int metrics_metricset_empty(metrics_metricset_t *set);
int metrics_metricset_fill(metrics_metricset_t *set);
int metrics_metricset_add(metrics_metric_id_t metric_id, metrics_metricset_t *set);
int metrics_metricset_del(metrics_metric_id_t metric_id, metrics_metricset_t *set);
int metrics_metricset_ismember(metrics_metric_id_t metric_id, const metrics_metricset_t *set,
int *ismember);
```

The metric extraction engine implemented by this library shall be initialized before it can be used to extract metrics from a trace stream. This initialization is carried out by the following function:

```
int metrics_init(trace_id_t trid, const metrics_metricset_t *set);
```

This initialization action binds the metrics extraction engine with a trace stream and a metric set. The trace stream identified by the *`trid`* argument will be used later by the metric extraction engine in order to search metrics and retrieve measurements for this metrics. A pre-recorded or an active trace stream can be binded to the metric extraction engine and the corresponding identifiers shall be retrieved by using the appropriate functions available in the POSIX Tracing interface. The metric set identified by the *`set`* argument is used to select the metrics that the metric extraction engine will search in the trace stream.

Once the metric extraction engine is initialized, measurements can be retrieved from the trace stream. These measurements are objects of the *`struct metrics_measurement_t`* type and they can be retrieved by using the following functions:

```
int metrics_getnext_measurement(metrics_measurement_t *result, int *unavailable);
int metrics_trygetnext_measurement(metrics_measurement_t *result, int *unavailable);
int metrics_timedgetnext_measurement(metrics_measurement_t *result, int *unavailable, const
struct timespec *abs_timeout);
```

```
 typedef struct {
metrics_metric_id_t  metric_id;
pthread_t            thread_id;
struct timespec      duration;
struct timespec      begin;
struct timespec      end;
int                  events_count;
int                  id;

} metrics_measurement_t;
```

These functions search metrics in the trace stream and, whenever a metric is found, a measurement for the metric reported. The argument *`unavailable`* is set to zero when no more metrics can be retrieved from the trace stream. This occurs when either all the trace events in the trace stream have been retrieved (for pre-recorded trace streams) or when the trace stream is destroyed (for active trace streams).

The `metrics_getnext_measurement` function retrieves as many trace events from the trace stream as necessary in order to find a metric and returns when a metric is found. The calling process may be suspended if no trace events are stored in the trace stream and this trace stream is an active trace stream. The execution of the calling process is restarted when a new trace event is stored in the trace stream.

The `metrics_trygetnext_measurement` function retrieves only a trace event from the trace stream at once in order to find a metric. If no metric is found, then an error code is returned indicating this situation. This function is applicable to active trace streams only.

The `metrics_timedgetnext_measurement` function behaves as the `metrics_getnext_measurement` function, but if the calling process is suspended, it is restarted at the time indicated in the *`abs_timeout`* argument. In this case this function returns an error code indicating this situation. This function is applicable to active trace streams only.

Whenever a metric is found, these functions return a measurement for this metric. A measurement includes the following data:

`metric_id`	The metric associated to this measurement
`thread_id`	The thread associated to this measurement
`duration`	The amount of time in which the system has been in the state which corresponds with the metric found
`begin`	The time stamp of the first trace event which corresponds to the metric found
`end`	The time stamp of the last trace event which corresponds to the metric found
`event_count`	The number of trace events generated by the trace system while the system was in the state which corresponds with the metric found

In the current stage of the Metrics library implementation, the following metrics are currently defined:

`RUNNING_SECTION`

The amount of time elapsed since a thread is dispatched until another thread is dispatched

`CLOCK_NANOSLEEP_NO_SUSPENDS`

Kernel execution time used to serve a `clock_nanosleep` function when the calling thread is not suspended

`CLOCK_NANOSLEEP_UNTIL_SUSPENDS`

Kernel execution time used to serve a `clock_nanosleep` function until the calling thread is suspended

`CLOCK_NANOSLEEP_AFTER_SUSPENDS`

Kernel execution time used to serve a `clock_nanosleep` function since the calling thread is awakened until this function returns

`JOB_RESPONSE_TIME_CLOCK_NANOSLEEP`

Job response time for a thread which uses the `clock_nanosleep` to implement its periodic behaviour

`JOB_EXECUTION_TIME_CLOCK_NANOSLEEP`

Job execution time for a thread which uses the `clock_nanosleep` to implement its periodic behaviour

`JOB_INPUT_JITTER_CLOCK_NANOSLEEP`

Input jitter for a thread which uses the `clock_nanosleep` to implement its periodic behaviour

JOB_OUTPUT_JITTER_CLOCK_NANOSLEEP

Output jitter for a thread which uses the `clock_nanosleep` to implement its periodic behaviour

4.4. Implementation issues

Measurements are obtained by searching sequences of events stored in a trace stream that correspond to the metrics that the programmer wants to obtain. Each metric provided by the metric extraction engine is analyzed by an automaton, or, more precisely, for a group of equivalent automata, each of which is associated to a different thread identifier. These automata are internal to the implementation and therefore the users of the Metrics interface do not use them directly.

We decided to use automata because they adapt very well to the nature of the information to be processed, which is formed by a sequence of trace events. Another advantage of using automata is that they allow the implementation of new metrics in a very easy way.

These automata are defined using the TCL scripting language. Each automaton is defined as a TCL list, in which the automaton states and transitions are declared. An example of this automaton definition is shown in the next code fragment.

```
{RUNNING_SECTION
    {   ## States

         #  Name           Type

 {NOT_RUNNING      OUT }
 {RUNNING          IN  }
 {DONE             END }
    }
    {   ## Transitions

         # From        To              Label

 {NOT_RUNNING  RUNNING     {NORMAL M_CONTEXT_SWITCH  SELF_THREAD }} #1
 {RUNNING      DONE        {NORMAL M_CONTEXT_SWITCH  OTHER_THREAD}} #2
 {RUNNING      DONE        {NORMAL M_LAST_EVENT      ANY_THREAD  }} #3
 {DONE         NOT_RUNNING {LAMBDA}}                                #4
    }
}
```

This automaton example implements the metric *RUNNING_SECTION*. This metric represents the amount of time elapsed since a thread is dispatched until another thread is dispatched.

This definition declares an automaton class, and this class is instantiated at run time for any thread which is detected in the trace stream which is being scanned. In this way, if an application has ten threads, then ten automata for the metric *RUNNING_SECTION* are generated, each one of them binded to a specific thread.

This automaton class has two main states: *NOT_RUNNING* and *RUNNING*. The *NOT_RUNNING* state is the initial state of the automaton and indicates that the thread binded to the corresponding automaton instance is not running. This state is of the type *OUT*, that means that the automaton is not recognizing the metric *RUNNING_SECTION*.

When the system performs a context switch to the thread binded to the corresponding automaton instance, then the automaton switches to the *RUNNING* state. This state change is mandated for transition 1. This transition applies when the trace event *M_CONTEXT_SWITCH* is detected in the trace stream and the system selects the thread

binded (*SELF_THREAD*) to the automaton instance. The *RUNNING* state is of type *IN* that means that the automaton is now recognizing the metric *RUNNING_SECTION*.

Finally, when the automaton detects in the trace stream that the system performs a context switch to another thread or the last event of the trace stream has been retrieved, the automaton changes to the *DONE* state, which is of the type *END*. This change is mandated by transitions 2 or 3. When a state of type *END* is reached, the automaton produces a measurement. Next, due to the lambda transition 4, the automaton changes to the *NOT_RUNNING* state.

Automaton instances have two internal modes of operation: *recognizing* or *not recognizing* a metric. An automaton instance is in the not recognizing mode initially. Being the automaton in the not recognizing mode, it enters in the recognizing mode when an *IN* state is reached. An automaton instance switches to the not recognizing mode again when it reaches an *END* state, and produces a measurement with the following data:

metric_id	The metric corresponding to the automaton instance
thread_id	The thread binded to the automaton instance
duration	The sum of the length of the segments detected by the automaton while it was in the recognition mode. A segment is formed by a sequence of *IN* states and the length of a segment is the difference between the time stamps of the trace events that have determined the initial and final states of the segment
begin	The time stamp of the trace event which initiates the metric recognition mode
end	The time stamp of the trace event which makes the automaton instance to switch to the not recognition mode
event_count	The number of trace events detected while the automaton instance was in a state of the type *IN*

The previous automaton class example is quite simple since a measurement is comprised by a single segment, and so not all the functionality of these automata is shown. For example, when more complex metrics are defined, several segments are part of single measurement normally. There are two more state types also, *CANCEL* and *CANCEL_SEGMENT*. These states allow to cancel the current measurement or the current segment respectively and they are used when the automaton detects that the current situation detected in the trace stream does not really correspond to the metric which is being analyzed.

Once the automata used by the metrics extraction engine have been described, let us see how these automata are implemented.

An automaton class is represented internally as an array of state descriptors. Each one of these state descriptor holds the type of the state and a pointer to the head of a list of transition descriptors, which is formed by the output transitions of the state. A transition descriptor holds its target state, its label and a pointer to the next transition in its transition list. At compile time, a TCL script reads the definitions of the automata classes and generates C code that declares the corresponding data structures. An example of the generated C code is shown in the next code fragment, which corresponds with the automaton which scans the metric *RUNNING_SECTION* described above.

```
mtri_transition_descriptor_t tr_1 =
      {1, NORMAL, M_CONTEXT_SWITCH, SELF_THREAD, NULL};
mtri_transition_descriptor_t tr_2 =
      {2, NORMAL, M_CONTEXT_SWITCH, OTHER_THREAD, NULL};
mtri_transition_descriptor_t tr_3 =
      {2, NORMAL, M_LAST_EVENT, ANY_THREAD, &tr_2};
mtri_transition_descriptor_t tr_4 =
      {0, LAMBDA, M_ANY_EVENT, ANY_THREAD, NULL};


mtri_metric_descriptor_t mtri_metric [] =
{
```

```
{{OUT}, {IN}, {END}},
{&tr_1, &tr_3, &tr_4}
},
{
  // Other automata
};
```

As you can see, automata are declared statically in the C code. This approach has the advantage that no code is necessary at run time to initialize automata classes, making the memory size of the library smaller.

On the other hand, automaton instances are generated by using dynamic memory at run time only when they are required. Particularly, when a new thread identifier is detected in the trace stream which is being scanned, automaton instances binded to this thread are created for all the metrics selected in the metric extraction engine initialization. Each automaton instance is represented by a automaton instance descriptor. This descriptor holds information about the current state of the automaton instance, a pointer to its class automaton and accounting information related to measurement being performed by the automaton instance. All of these automaton descriptors are held in a single linked list. When an trace event is retrieved from the trace stream, this event is delivered sequentially to each one of the automaton instances in this list.

The temporal cost of retrieving a measurement is as follows. Processing a single event from a single automaton instance has a cost of *O(1)* . The `metrics_getnext_measurement` function processes as many trace event as necessary in order to get a measurement, and each trace event is delivered to all the automaton instances of the automaton instances list. The size of this list is *M x T*, being *M* the number of selected metrics and *T* the number of threads detected. The total cost for this function is *O(E x M x T)*, being *E* the number of trace events required to get a measurement. The cost of the `metrics_timedgetnext_measurement` function is the same, since it has the same definition. Finally, the cost of the `metrics_trygetnext_measurement` function is *O(M x T)*, since this function processes only a trace event at time.

4.5. Tests and validation

4.5.1. Validation criteria

The main validation criteria for this component are the following: the quality of the interface, its correctness and its efficiency, mainly when it is used for on-line metrics retrieval.

The quality of the interface includes aspects such as clarity, ease to use, compatibility with other interfaces and to be appropriate to implement it efficiently. A great effort has been done in order to design a high quality interface. This effort has been based on following the same design principles used in modern POSIX interfaces and to develop several application programs which use this interface in order to perform useful tasks, such as generating metrics reports or supervising the temporal behaviour of real-time applications.

Correctness is an obvious validation criteria. In order to meet this requirement more easily, the Metrics library has been implemented as two different parts: an automata definition system and a generic core capable to use the previously defined automata. The advantages of this approach is that the automata definition system and the generic core are small and simple programs, and so their correctness is easy to validate.

By having enough confidence about the correctness of this implementation, we can deal with the more difficult aspect of the overall correctness: to ensure that a particular automaton class corresponds with an unique path in the RT_Linux system execution

and that this execution path let us detect the metric for which the automaton class is designed. This correctness can be achieved by having a precise knowledge of the RT-Linux system and by intensive testing and examination of the trace events stored in the generated trace streams.

Respecting efficiency, in this first stage of implementation we are trying to meet this criterion by selecting the most appropriate data structures. The overhead of this Metric library will be measured in the following implementation stages, and a more fine tuning of the implementation will be done.

4.5.2. Tests

The Metrics library is currently in the testing phase, specially in respect to the implementation of new metrics. Test programs are always a set a real-time threads in which the metrics which are being tested are present. Trace streams are obtained from the execution of these programs and they are analyzed for the metrics extraction engine, in order to test if the automata corresponding to the metrics which are being tested are well defined.

4.6. Example

The following code fragment shows the expected usage of the Metrics application interface. First, a set formed by two predefined metrics is built and the metrics extraction engine is initialized. Next, all the measurements are extracted from the trace stream and processed.

```
  metrics_metricset_empty (&set);
  res = metrics_metric_open ("JOB_EXECUTION_TIME_CLOCK_NANOSLEEP",
      &JOB_EXECUTION_TIME_CLOCK_NANOSLEEP);
  res = metrics_metric_open ("JOB_RESPONSE_TIME_CLOCK_NANOSLEEP",
      &JOB_RESPONSE_TIME_CLOCK_NANOSLEEP);

  metrics_metricset_add (JOB_EXECUTION_TIME_CLOCK_NANOSLEEP, &set);
  metrics_metricset_add (JOB_RESPONSE_TIME_CLOCK_NANOSLEEP, &set);

  metrics_init (trid, &set);

  metrics_getnext_measurement (&result, &unavailable);

  while (! unavailable) {

    process_measurement (&result);

    metrics_getnext_measurement (&result, &unavailable);
  }
}
```

The above code fragment can be used for both off-line and on-line metric retrieval. For off-line retrieval, the trace stream used to initialize the metrics extraction engine should be a pre-recorded trace stream. In this case, the trace stream should be created by using the POSIX tracing `posix_trace_open` function, as it is shown in the following code fragment:

```
  fd = open ("trace.log", O_RDWR);
  posix_trace_open (fd, &trid);
```

A typical usage for off-line retrieval is the generation of a data file in which all the measurements are stored in order to generate a metrics report later. A program which generates this data file is currently available. This program generates an XML formatted data file with all the measurements found. An example of a fragment of this data file is the following:

```
<MEASUREMENT>
  <METRIC>     JOB_EXECUTION_TIME_CLOCK_NANOSLEEP   </METRIC>
  <THREAD>                        1   </THREAD>
  <DURATION>           0.003412928   </DURATION>
  <BEGIN>              1.000565376   </BEGIN>
  <END>                1.063999008   </END>
  <EV_COUNT>                   52   </EV_COUNT>
  <ID      >                   37   </ID>
</MEASUREMENT>
```

When on-line retrieval is used, the trace stream should be an active trace stream, and it should be created by using the tracing `posix_trace_create` function:

```
posix_trace_create (0, &attr, &trid);
```

By using the on-line metrics retrieval it is possible, for example, to implement a task which supervises at run time that temporal requirements, such as the deadline or the worst-case execution time, are met for the regular application tasks.

Chapter 5. RT-Terminal (RTTerminal)

5.1. Summary

Name
: RT-Terminal

Description
: Direct console access from RTLinux tasks.

Author/s
: Miguel Masmano

Reviewer
: Ismael Ripoll

Layer
: High level RTLinux.

Version
: 0.1

Status
: Beta

Dependencies
: Stand alone code. No dependencies.

Release Date
: M3

5.2. Descritpion

Currently, RTLinux human user interface capability is limited to directly print strings via functions: `conprn()` or `rtl_printf()`. Direct user input from keyboard is not possible. Complex interaction with the human user (data representation, and user input) has to be done by a non real-time Linux application communicated with the RTLinux application via shared memory or RT-FIFO's. These communication mechanisms are non-portable specific RTLinux facilities. Also, the access to the video and keyboard can not be done in real-time since it is delayed until Linux became active.

RT-Terminal is a new component that allows to RTLinux applications display data directly on the console screen and read the pressed keys directly from the keyboard.

RT-Terminal will provide direct access to the console to RTLinux applications and at the same time it will keep compatibility with the existing Linux console drivers.

RT-Terminal will also be used by the Stand-Alone RTLinux component, since standalone can not use the linux kernel to print on the screen.

5.3. Layer

Low level, between the linux kernel and the hardware (screen and keyboard).

5.4. API / Compatibility

The implemented API is basic POSIX standard (open, close, read, write) and also a subset of the Xterm (VT100) escape sequences in order to provide a compatible but fast and compact terminal control. VT100 standard has been chosen because it is a very popular terminal standard (almost all OS implements it), and because it is very easy to be used for implementing other functions based standards. For example, it is easy to wrap color functionalities with a function which prints the appropiate terminal controls.

The Xterm (VT100) subset used by the RT-Terminal is:

- < ESC >[{ROW};{COLUMN}H: Sets the cursors position where subsequent text will begin. If no row/column paramenters are provided, the cursor will move to the home position, at the upper left of the screen.
- < ESC >[{COUNT}A: Moves the cursor up by COUNT rows; the default count is 1.
- < ESC >[{COUNT}B: Moves the cursor down by COUNT rows; the default count is 1.
- < ESC >[{COUNT}C: Moves the cursor forward by COUNT columns; the default count is 1.
- < ESC >[{ROW};{COLUMN}f: Identical to the < ESC >[{ROW};{COLUMN}H.
- < ESC >[{attr1};{attr2}m: Sets multiple display attribute settings. The following lists standard attributes:
  - 0: Reset all attributes 5: Blink
  - Foreground Colors
    - 30: Black.
    - 31: Red.
    - 32: Green.
    - 33: Yellow.
    - 34: Blue.
    - 35: Magenta.
    - 36: Cyan.
    - 37: White.

  - Background Colors
    - 40: Black.
    - 41: Red.
    - 42: Green.
    - 43: Yellow.
    - 44: Blue.
    - 45: Magenta.
    - 46: Cyan.
    - 47: White.

5.5. Dependencies

RT-Terminal depends on Linux Kernel, since in order to catch the keyboard interrupts, the Linux Kernel has to be patched.

5.6. Status

Currently, RT-Terminal is in alpha status.

5.7. Implementation issues

Two different behaviors, selectable at compile time via a configuration option, will be provided:

□ Non real-time behavior: Screen and keyboard is managed by Linux (at background or slack time). Threads output buffered and only effectively printed on the screen when Linux became active.

- Real-time behavior: Read and write functions are implemented completely in RTLinux, and the output is performed immediately before the `write()` function returns to the calling thread.

Both implementations will provide the same API (POSIX + VT100 control commands), so the only difference will be when characters are printed or read from keyboard, whether in non-real-time or in real-time mode.

When the RT-Terminal is inserted in RTLinux, it is registered as a POSIX device and it is visible to RTLinux applications as a standard RTLinux device on "`/dev/tty`", so it is possible to use standard POSIX calls to `open()`, `write()`, `read()`, and `close()` the terminal.

In order to implement the RT-Terminal component, the implementation can be divided in two halves.

- Keyboard reading. It can be implemented using one of the following methods: (1) intercepting keyboard interrupts, reading the value of the keyboard chip register and if it is not a key for the RTLinux handler, then write-back the key-scancode to the keyboard and call the normal Linux handler; (2) consists of patching the kernel keyboard handler interrupt so wether the pressed key is the expected key, the handler will send it to the RT-Terminal.

  Since some PC keyboard controller are not fully compatible (scancode can not be written back to the controller), the RT-Terminal will be implemented using the second method (2).

- Screen writing. The implementation takes control of the video memory and writes directly on it. Screen writing must be a non-blocking operation. The old content of the video memory is copied in an extra buffer.

Initially RT-Terminal begins in Linux mode, all the data written by the RTLinux application is introduced into a buffer and all the pressed keys are passed to the Linux Kernel. When user press the **F10** key, the RT-Terminal mode changes to RTLinux mode, the current screen is stored in a buffer and all the buffered informations are showed on the screen, in RTLinux mode all the pressed keys are passed to the RTLinux application that reads from the RT-Terminal.

To maintain compatibility with current Linux system, RT-Terminal will intercept the Control-Alt-F9 key combination to switch the control from RTLinux console to normal Linux processing.

5.8. Validation criteria

RT-Terminal is a component that must not affect the correct behavior of a hard real time application, i.e. all the applications that had a correct time behavior before using RT-Terminal, must have the same behavior after using the component facilities.

In order to validate screen writing, it will be tested that all the data sent to the RT-Terminal by means of the `write()` function are showed correctly on the screen.

The validation criteria for keyboard reading is that the RTLinux appication must receive all the data that user is sending to it throught the keyboard.

5.9. Tests

In this componnent, two kinds of tests can be done: tests that check the correct temporal behavior of application which uses RT-Terminal and tests that check the correct VT100 interface implementation.

- Correctness of the VT100 interface implementation:

- Printing test: this test checks correction of all implemented VT100 control sequencies (color and cursor movement control sequencies). This test can not be automatised so an external user must check test's results.
- Keyboard test: this test checks the RT-Terminal `read()` POSIX function. In particular this test, read keyboard pressed keys showing them on the screen through `rtl_printf()` function.
- Printing test + Keyboard test: this test joins the two previous tests getting information from the user and printing gotten information on the screen through the RT-Terminal.

- Correct time behavior of applications that use the RT-Terminal component:
  - Printing test: This test implements several threads which write in a concurrent way using the RT-Terminal, and later using `rtl_printf()`. The test also checks that screen printer functions have a correct real-time behaviour (i.e. they are non-blocking functions) and a similar response time.
  - Keyboard test: As previously explained the reading keyboard operation, `read()`, is a blocking operation, so it must be used only by background threads. In particular this test checks this behaviour, measuring the time that the function uses to get a string from the keyboard and then print the obtained results through `rtl_printf()`.

5.10. Example

This example shows how the component works. The file screen_print.c uses the write function to print some strings on the screen.

```
#include <rtl.h>
#include <rtl_sched.h>
#include <posix/unistd.h>

static int terminal_fd;
static pthread_t pthread;

MODULE_AUTHOR("Miguel Masmano Tello <mmasmano@disca.upv.es>");
MODULE_DESCRIPTION("Example of use of the RT_Terminal");

MODULE_LICENSE("GPL");

void *main_func (void *att) {

  char str[] = "";

  char str1 [] = "This is the RT-TERMINAL\nTHIS component uses VT100 ESC
commands.\x1B[3;0H\x1B[34;47mYou\x1B[31;44mcan\x1B[36;41mchange\x1B[30;
42measily\x1B[34;47mthe\x1B[32;40mcolors\x1B[10;10H\x1B[34;43mOr you
can move by the screen";

    write (terminal_fd, str1, strlen(str1));

  return 0;
}

int init_module (void) {
  terminal_fd = open ("/dev/rt_tty", O_NONBLOCK);
  if ("terminal_fd" < 0) {
    rtl_printf ("Error opening RT_Terminal\n");
    return -1;
  }
  pthread_create (&pthread, NULL, main_func, 0);
  return 0;
}

void cleanup_module (void) {
  close (terminal_fd);
}
```

To run the example, basic rtlinux modules have to be inserted first (for example, using *rtlinux start*) and after the `rt_terminal.o` module. Once the `screen_print.o` module has been inserted, when pressing **F10** key the following appears:

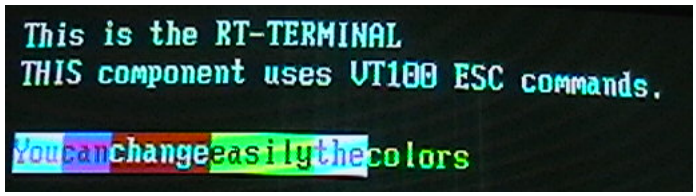

Figure 5-1. RT-Terminal example

Chapter 6. RTLinux UDP/IP (RTLUDP)

6.1. Summary

Name
: RTLinux UDP/IP

Description
: Hard real-time IP/DUP stack implementation.

Author/s
: Miguel Masmano

Reviewer
: Ismael Ripoll

Layer
: High level RTLinux.

Version
: 0.1

Status
: Testing

Dependencies
: None.

Release Date
: M3

6.2. Description

Currently, standard RTLinux/GPL has two different IP stacks: RTSOCK and LwIP (LwIP was ported to RTLinux by a college from the UPVLC, in a project sopported by the national reseach department, MCYT).

Rtsock is not a device driver for network cards. Instead, packets flow through the Linux kernel using the standard Linux drivers, up/down the standard layer 2 and layer 3 protocols, and then packets are diverted into an RTLinux task. Currently only UDP sockets are supported.

LwIP is a complete and full featured TCP/IP stack designed to be ported to embedded systems easily. The current version of RTLinux provides support of a small number of network card drivers.

This component is an **implementation of the UDP/IP stack from scratch** (not derived from BSD code as many other TCP/UDP/IP implementations). The main design criteria is efficiency, while features and compatibility are secondary. The stack will be prepared (interfaced) to work with the network device drivers developed in the project **EtherBoot**, which has a large base of supported drivers; and also be connected with the Linux networking stack (via a virtual network driver), providing a high level communication mechanism between RTLinux and Linux on the same machine.

Next figure outlines the internal structure of the proposed component: the core of the component is the UDP/IP stack, which provides the socket interface to RTLinux threads; the component will implement the required API to use all the **EtherBoot** project network device drivers; and finally, a standard Linux network driver will be provided so that Linux user application can communicate via UDP/IP with RTLinux threads.

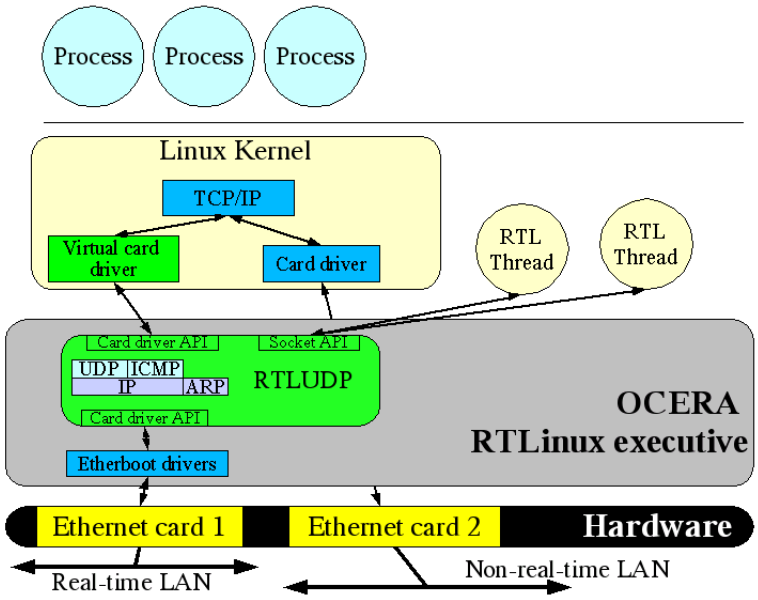

Figure 6-1. RTLUDP component

6.3. Layer

This is a Low Level RTLinux component. It also interacts with the Linux Kernel by means of a virtual network Linux driver since it allows to send ARP/IP packets via the kernel TCP/IP stack.

6.4. API / Compatibility

Standard BSD socket API: `send()` and `sendto()` for sending messages; `recv()` and `receivefrom()` to receive messages; and `connect()` to establish the connection.

The Stack will be compatible with all the **EtherBoot** network drivers.

6.5. Dependencies

RTLUDP do not have dependencies with any external components.

6.6. Status

Alpha status

6.7. Implementation issues

RTLUDP implements a light UDP stack with the following functionality:

- ARP protocol: Complete.

  The RTLUDP component implements the ARP protocol as is described in the RFC 826, titled "Ethernet Address Resolution Protocol".

  Besides the ARP protocol, RTLUDP implements a little software cache for storing Ethernet address and its IP address translation. All IP address are translated using this cache table. If an IP address is not located in the table, RTLUDP sends an ARP request only when the appropiate translation is not in the table.

  Since the ARP protocol is not an inherence real-time protocol, RTLUDP provides two non-standard functions: `insert_arp_translation()` and `flush_arp_table()`.

The first one allows user to insert into the arp cache table translations during the initialisation time and the last one allows to empty the whole arp cache table.

The use of the ARP protocol can be avoided if the physical medium used is a point-to-point medium.

- IP protocol: Complete. But packet split and recombine are not supported.

  Even using a deterministic physical medium, some IP protocol features, like the packet split, do not allow any kind of deterministic response time. Therefore in this implementation these non-deterministic features have been avoided. This IP protocol does not implement routing.

- UDP protocol: Complete.

  Avoiding the ARP protocol, since it can be statically initialised and supposing the previous described IP protocol, this UDP protocol can be considered a real-time protocol. RTLUDP only has made one simplification of this protocol: it does not calculate the UDP header checksum. However this is not an important simplification because it is covered by the RFC 768.

  The sockets abstraction has not any kind of simplification, allowing user to use any socket from 0 to 65535. RTLUDP does not reserve any port, from 0 to 1024, as conventionally done.

- ICMP protocol: Only some parts have been implemented like ECHO, DESTINATION UNREACHABLE, etc. Packages related with host or router information and management will not be implemented.

  This protocol has been implemented for debbuging porpouses so more error messages are not interesting.

  An extensive list of implemented error messages are:

  - ICMP_ECHO_REQUEST: This is a very useful message because it can be used for checking the sending and also the reception of IP messages, since it guarantees the reception of other hosts ECHO_REPLY messages.
  - ICMP_ECHO_REPLY: This message is used to response requests from other hosts. It has no sense to implement previous ECHO_REQUEST message without implementing this one.
  - DESTINATION_UNREACHABLE: This message allows to indicate that a package can not arrive to the specified destination.

- TCP protocol: TCP protocol has not been implemeted since it is not a real-time protocol.

The RTLUDP componen has been designed in two well-differenciated layer: the light UDP stack layer, previously described, and the hardware layer, which implements the drivers. RTLUDP supplies an interface layer, called hardware abstraction layer (HAL), for allowing the interaction between the UDP stack and drivers.

HAL has been designed keeping in mind two important details:

- It has to be as simple as possible, because making drivers must be an easy work.
- It must be compatible with the existing **EtherBoot** HAL, to allow to port easily existing **EtherBoot** drivers.

This HAL has been designed to be simple and also **EtherBoot** drivers compatible, thus with **EtherBoot** drivers can be used directly by the RTLUDP component.

Any implemented driver must satisfay the next interface:

- `poll()`: This function has to return "1" when a new package has arrived and "0" otherwhise.
- `reset()`: This function will be used by the stack to reset the physical device.
- `transmit()`: This function will be always called whenever some protocol wants to send a package.
- `disable()`: This function will be used by the stack to disable the physical device.
- `node_addr`: This is an optional variable which is used on the case that the physical device had an MAC address like for example the Ethernet protocol. On the case there exist a MAC address, it has to be initialised by the driver itself.
- `packet`: When the `poll()` function returns "1" means that a new packet has arrived so `packet` will contain the new packet.
- `packet_len`: When the `poll ()` function returns "1" means that a new packet has arrived so the packet variable will contain the new packet length.

Another important feature of the RTLUDP component is that it accepts dynamic insertion and extraction of the drivers, showing available drivers in the `/proc/rtl_udp` file.

Implementing drivers is a tedious task, so one solution is to use other project drivers, **EtherBoot** drivers have been choosen because it has drivers for almost all existing Ethernet cards. Due to the HAL interface is very similar to the **EtherBoot** drivers, in fact, it is the same. **EtherBoot** drivers can be used with only replacing its time functions with RTLinux time functions. This replacing is done because existing **EtherBoot** time functions use directly the hardware PIT, reprogramming it.

Besides the **EtherBoot** network driver, a new virtual Linux network driver has been implemented. This new driver allows direct network communication between the RTLUDP stack and the Linux Kernel TCP/IP stack, and it also allows to send packages through the Linux Kernel stack.

Although the first version of RTLUDP stack has been implemented using a one copy method, we will study the way to provide a real zero copy network on following versions (may be using the STREAMS interface).

6.8. Validation criteria

The validation criteria are:

- A Real Time behaviour, when the real-time protocol is used, must be achieved, being this point the main goal of the RTLUDP.
- RTLUDP stack must be completely compatible with existing UDP/IP protocol.

6.9. Tests

As is described in the Validation criteria, there are two important features to check in this component: the deterministic behaviour and the compatibility with existing protocols. There is no problem in testing the second feature, however, currently the real-time behaviour can not be tested because we have not any real-time physical device to use. Following four tests check ARP and ICMP compatibility:

- Test 1: The first test is a simple test with the ping utility. Using the virtual Linux kernel network driver, the ping utility sends several ICMP ECHO_REQUEST packets, and the RTLUDP component answers with the appropiate ICMP ECHO_REPLY package.
- Test 2: This test is similar to the previous one but this time the ICMP ECHO_REQUEST package is sent by the RTLUDP component.

- Test 3: This test checks as well as the ARP and ICMP protocol, it checks **EtherBoot** drivers. Using a different computer connected through ETHERNET without computer, it consists of sending several ICMP ECHO_REQUEST using the ping utility and waiting a correct answer.
- Test 4: This test is the same that the test 3 but this time the ICMP ECHO_REQUEST package is sent by the RTLUDP component.

EtherBoot Drivers has the same well-behavior as the Virtual Linux Kernel network driver so for following will be executed using first **EtherBoot** drivers and later the Virtual one. The following tests check the ARP and IP/UDP compatibility, (Only UDP compatibility can be tested since in the current version raw IP packages have not been implemented):

- Test 5: UDP sending and receiving compatibility. This test has been designed to prove UDP sending compatibilities with a standard TCP/IP stack. The test sends several packages, from several ports, to one open Linux port, and waits confirmation.
- Test 6: Previous tests have check if the implemented UDP sends and receives correctly packages with open ports, tests 6 checks the same but with close ports. This test checks what it happens when nobody waits a package. A Linux application sends several packages to RTLUDP closed ports and waits a response.
- Test 7: Stress test. In this test, several external applications send packages through different physical mediums to the RTLUDP, trying to get a stack overload, measuring the number of lost packages, times, and so on.

Chapter 7. IDE Device Driver (RTLide)

7.1. Summary

Name
IDE Device Driver
Description
Device driver for IDE hard disks.
Author/s
Alejandro Lucero
Reviewer
Ismael Ripoll
Layer
High level RTLinux.
Version
0.1
Status
Testing
Dependencies
Requires Linux kernel to initialise PCI and DMA devices.
Release Date
M3

7.2. Description

RTLinux does not support direct access to any kind of permanent massive storage systems, and in particular IDE hard disks. When a RTLinux thread has to storage data on the hard disk, it has to use the Linux services. The usual way of doing the transfer is by means of RTFifos: a rtl-task sends the data to Linux through an RTfifo, and then, a Linux process writes this data on the hard disk.

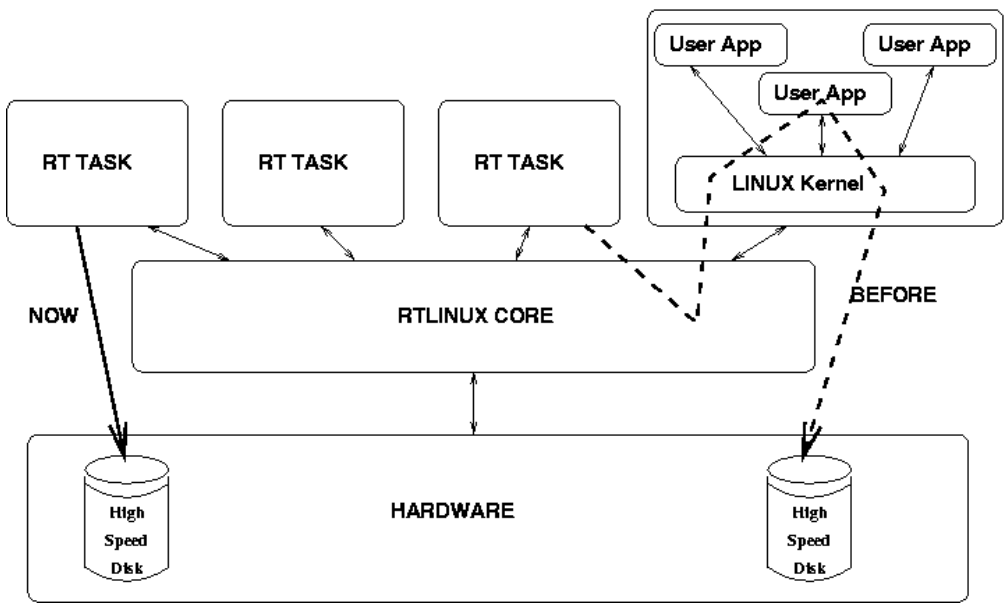

Figure 7-1. Access to the hard disk.

The exitsting method to access hard disks is slow, unpredictable and inneficient, since it is executed by the Linux (background). Moreover, in some applications, like continuous

media applications, it is required a certain degree of predictability. Therefore, for multimedia applications is more convenient that rtl-tasks have access to the hard drive, so that disk access can be done with real-time guarantees. To achieve this goal it is necessary to: 1) port the IDE driver to RTLinux, 2) implement a real-time filesystem, and 3) implement a real-time disk scheduler.

This component provides the porting of the Linux IDE driver to RTLinux. The other two blocks of the file subsystem (disk scheduler, and filesystem) are provided in a separate component (see Chapter 8, *RTLinux Disk scheduler and file system (**RTLfs**)*).

We have used the Linux device driver for IDE disks code and ported it to work in RTLinux. The part of the driver that deals with the hardware has been used as-is. Most of the porting effort was dedicated to remove the interface with Linux and the facilities and data structures provided by Linux.

7.3. Layer

Current version is a high level RTLinux component since the code do not modify RTLinux code.

7.4. API / Compatibility

Generally, IDE devices are not user oriented. Although they are implemented (inside UNIX systems) with the standar file operations as `open`, `read`, `write`, ..., users don't work with them directly, but using a file-system layer. However, some special system commands as **fdisk**, and (of course) the file system use these operations directly.

RTLinux has a standard file system interface to access devices as fifos or shared memory. For this RTLinux uses a `rtl_file_operations` structure with the `open`, `read` , `write`, `lseek` , `mmmap`, `unmap` , `close` functions. The ide driver implements these functions to achieve the POSIX approach.

7.5. Dependencies

The current version of the component depends on initialisation functions of Linux kernel for PCI, IDE buses, and DMA (Direct Memory Access) device. As the goal is to offer a high performance, is necessary to use DMA functionality which is present in modern disks. Old drivers without DMA functionality could be supported easily, but it is not clear what advantages could have.

7.6. Status

Second internal version IDE Driver is working with **DMA functionality**. The first driver version implemented at the earliest stage of this component with only had single sector operations functionality (which was used to study how Linux and RT Linux could share the same IDE disk).

Some decisions has been taken to simplify the code: First, Linux supports a wide range of IDE drivers including some weird and screwy (literal from Linux docs) ones. We support the default standard types since usually IDE disks have the same standard behaviour, but we don't support the weird and screwy devices. Second, DMA devices provides the possibility to merge several memory blocks not contiguous in memory in one single request. It's usual that file systems have a block size of 1K or 4K, and that these blocks that are contiguous on disk aren't when are located in the host memory. With this functionality disk manufacturers are aware of operating system necessities. As we explain bottom, this advanced DMA functionality is not supported in this version.

If this component gains acceptance, the weird cases and special DMA features could be implemented in next releases without excessive effort.

ATAPI devices are not supported. ATAPI is a interface originally developed to support CD-ROM devices.

In this version is not implemented the IDE sharing by Linux and RTLinux. A dedicated disk is needed for RT tasks. Therefore, the system will require at least two physical disks.

7.7. Implementation issues

Since the way the IDE device will work with RTLinux is the same as works with Linux, we could use the Linux implementation code. However, Linux IDE device driver implementation is very related to the Linux block layer which includes buffer and page cache structures. Linux requests are processed inside the IDE device driver using the `do_rw_disk()` function. All of this functionality depends on Linux buffer heads, a critical component in Linux Block Layer design.

The Linux block buffer was designed taking into account that the more frequent use exhibits: files are relatively small, read and write access are mostly sequential (spatial locality) , and data is usually accessed again in a short period of time (temporal locality). These general access characteristics are not longer valid for real-time systems: the size of the files will be much bigger (multimedia streams), spatial locality, and a few temporal locality.

Among other heuristics, the Linux block buffer reads ahead disk blocks (to improve sequential access), and keeps the disk data some time before it is finally written into the disk. Although these heuristics improves the overall performance of the system, they are not adequate for real-time systems. The block buffer has been removed in the porting.

Master DMA is a very important disk functionality. DMA is useful to avoid the CPU overhead when processing I/O requests. If DMA is not actived, the CPU must read/write data to disk using I/O instructions, with the limitation of the number of bytes a CPU can move in one single operation. On the other hand, when DMA is actived, CPU releases the bus to allow another device (IDE Disk) to do I/O requests, meanwhile the CPU do other tasks. Other advantage of using DMA is that 64K can be moved in each operation, avoiding a high number of interrupts when is CPU who takes care of the request.

Since the RTLinux file system and disk scheduler are designed to support efficiently the allocation of media streams, DMA becomes in one of the most important points in the system. Last DMA modes support 133MB/s, which is a bandwith unreachable with PIO modes (CPU Programmed Input/Output). As DMA in Linux IDE driver works with buffer heads, a new implementation of DMA routines has been necessary.

DMA is useful to avoid CPU overhead processing I/O requests. If DMA is not actived, is the CPU that must read/write data to disk using I/O instructions, with the limitation of the number of bytes a CPU can move in one single operation. On the other hand, when DMA is actived CPU only releases the bus to let another device to do I/O requests (Master DMA), then CPU can do other tasks. Other advantage of using DMA is that can move 64K in each operation avoiding a high number of interruptions that occurs when is CPU who takes care of the request.

7.7.1. Configuration

Linux makes a conservative DMA configuration: if the device is not known or is in the black list[1], DMA is not activated. Obviously the successful of our design is based on DMA functionality, so a more aggressive configuration is required. The checks to do during initialization are:

1. Is the IDE disk present?
2. Is a Master DMA and LBA capable disk?
3. Is disk into the devices black list?
4. What DMA mode is supported?

Initialization of IDE disks is done by Linux kernel at boot time. RTLinux only needs the structures that Linux has initialized, so no special initialization routines are necessary. Therefore, this component can not be directly used in stand-alone RTLinux unless buses (PCI, ISA) and devices (DMA) initialization has also been ported.

7.7.2. Mode of operation

DMA takes care of operating system necessities. Most file systems use to use block sizes of 1K-4K to manage data, also, data that has to be stored contiguously on the disk is scattered in several of these blocks. Some DMA devices can be programed with a list of non-contiguous physical blocks so that the DMA feeds the disk device as it where only a single large data block. This way, the number of disk operations is reduced. Although this is a powerful functionality for general purpose operating systems, it has not so importance in our design, since memory is managed more restrictively (at initialization time) within the RTLinux approach. When developers ask for memory at inicialization they can use the `__get_free_pages` linux kernel function that returns a pointer to a contiguous memory block. The steps to use the driver are showed in the next graphic:

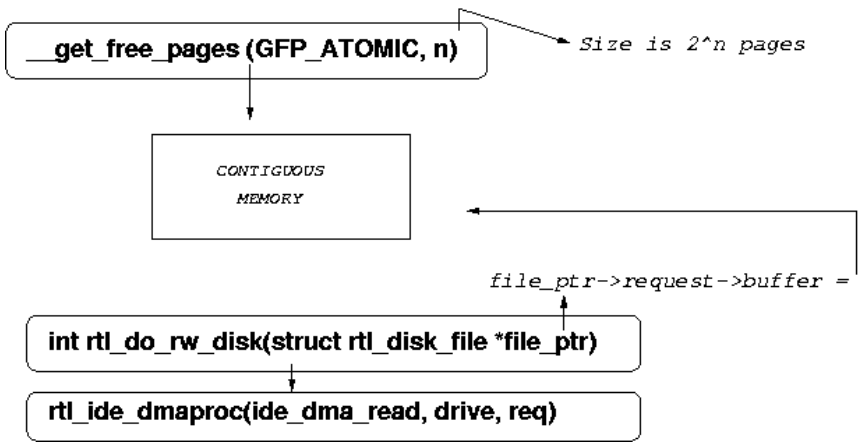

Figure 7-2. Contiguos memory allocation

The entry point to ide disk driver is the `rtl_do_rw_disk` function. A pointer to a `rtl_disk_file` structure is received, which is an object for an open file (related with one inode). The rtl_disk_file structure has a `request` field, a pointer to a `rtl_request` structure that will be used for the driver for access the disk. And one field of the rtl_request structure is a pointer to a buffer. This buffer must be allocated by users at inicialization time and must be contiguous in memory, it means the `__get_free_pages` function or similar must be used. After that, the `rtl_ide_dmaproc` will be called. This approach is distinct from the Linux one, since Linux must take care of scattered memory blocks, using aditional functions as `buid_dma_table` and `build_sg_list` . The scattered blocks funcionality makes sense in a dynamic memory enviroment, when memory is a resource very demanded for short spaces of time.

7.8. Validation Criteria

Since this component has not utility without the file system and disk scheduler, validation criteria would focus on how the complete block layer works. The interface with the RTLinux tasks is located in the file system, so validation criteria for IDE driver will be same than the validation criteria for the file system.

Another validation criteria would be how many devices has been tested with the driver. As we comment above, some special devices with a weird behaviour are not supported.

7.9. Tests

See tests section of the file system and disk scheduler component.

Notes

1. Black list: list of chip sets that are not fully compatible, or that may cause problems with the Master DMA functionality.

Chapter 8. RTLinux Disk scheduler and file system (RTLfs)

8.1. Summary

Name
: Disk scheduler and file system

Description
: Disk scheduler and hard real-time filesystem design and utilities.

Author/s
: Alejandro Lucero

Reviewer
: Ismael Ripoll

Layer
: High level RTLinux.

Version
: 0.1

Status
: Testing

Dependencies
: Relies on the RTLide component.

Release Date
: M3

8.2. Description

The aim of this component, together with the IDE Device Driver component, is to provide a full block layer for disk storage devices to RTLinux. Nowadays there is a necessity for real time tasks to storage media streams without loosing any sample or frame.

This block layer implementation will consist of three distinct elements: 1) the file system, 2) the disk scheduler and 3) the disk device driver (See the IDE driver documentation). Although we implement these elements separately to support future developments of only one of them, they are not completely independent: for example in the file system design we need to know what mechanism supports the disk device (as DMA) to take advantage of it, or the disk scheduler need information of which are the response times of the disk device to take decisions to achieve the requirements.

This component defines and implements a new file system designed to provide hard real-time (specially designed to serve multimedia applications) and robust recovery. The current disk scheduler sorts the requests by the priority of the calling thread.

Most of the effort has focussed on the file system structure, and not on the user interface.

8.3. Layer

Current version of this component do not change the RTLinux core, it only adds new functionalities. Therefore, it is a High-level RTLinux component.

8.4. API / Compatibility

The interface between the component and the system is through POSIX standar for file systems. Every read/write request from a RTL thread will use the POSIX `read` and

`write` calls, and these functions will sent the request to disk scheduler using a well-defined interface between the file system and the disk scheduler which is not in the user domain.

Accessing File system from RT tasks is through the POSIX-IO interface defined in rtl_posixio.h and implemented in rtl_posixio.c. Some changes have been done in these files: the open function has been modified to allow the use of the rtlfs file system. *Open* function searchs the RTLinux devices structure, so a new device has been added: the rtlfs device. When a user needs to open a file must use the full path *open("/dev/rtlfs/filename")*.

The file system is Linux compatible, since we have developed a Linux file-system module for this posibility, that allows Linux to mount the file system in *READ Only* mode. There is a restriction to use this funcionality: Linux can not mount the file system while it is mounted by RTLinux.

8.5. Dependencies

It is strongly related to the OCERA IDE device driver component, since only this driver has been implemented. Current implementation do not provide a clean layer between the IDE driver and the file-system. It will be provided in next release.

8.6. Status

There is a first final version of the component with a POSIX file system implemented. The disk scheduler sorts request by thread priority, being a future issue to develop a real time disk scheduler in next versions. This means that real time is not currently implemented in the block layer, since it is necessary to design and implement other kind of synchronisation mechanism to access the disk (now with mutex semaphores), and a more specific study of disks characteristics as geometry, caches or recalibration functions.

8.7. File System Description

As previously commented, the main goal of this component is to provide a file system to store media data. This point, along with the specific characteristics of a real time system leads the design. In this section we explain what are the main characteristics the file system should exhibit, and in the next section is presented the specification of the design in detail. The discussion is not only about the file system, but about the full block layer which includes global system structures.

Next are outlined the main characteristics of an embedded operating system (RTLinux) and the applications requirements:

CONCURRENCY

RTLinux is not a general purpose operating system as Linux is, where is usual to have lot of tasks working at the same time. The expected RTLinux workload will be just a few threads, possibly one or two. Obviously, mechanisms to share the file system between several tasks is a must, but it is important to fix the number of concurrent tasks supported since it is necessary several structures per task and per open file.

SIMPLICITY

The key in RTLinux is simplicity: it is not necessary to build a full real-time operating system with all kind of functionalities as a general purpose operating system. In this way, the RTLinux core is easier to maintain. If we don't want to break this approach, the file system design must be simple, avoiding complex implementations and features that are rarely used. We are not thinking in designing a file system for

all kind of requirements, only to support media streams, which have a known access pattern.

SPACE ALLOCATION

How the data is allocated in disk is one of the main functions of file systems. There are two points :

i. how data is allocated on the disk

ii. how metadata is managed.

Metadata is information about the file system: super block has global information of the file system; inodes are related with files and have information as size of the file or pointers to data blocks; free blocks list or bitmaps are used to manage free space, etc. File systems decisions about metadata (which data structures to use, and where to allocate them on the disk) are important for file system performance. For example to try allocate the inodes of a file as close as possible of their data blocks. Other decision is if metadata must be written sync or asynchronously which has a direct impact on reliability. We need a file system that can be returned to a consistent state after a crash and to avoid metadata writes can degrade performance of the file system.

The allocation policy is different depending on the feature that we want to optimise. For example, in general purpose operating systems, file systems are designed considering that most files are small, typically is a few kbytes, and that the lifetime of each file could vary from a few seconds to several months or years. The file size is important to avoid excessive fragmentation which leads to a poor usage of the disk, so general file systems use a minimal allocation unit of 1-4 Kbytes (1-8 sectors). The smaller the allocation unit is, the bigger is the metadata required to manage it, because there are more blocks (units) to handle. This implies more resources wasted and higher cost when searching through (or a complex structures to minimise the search cost).

In systems designed to collect data, the requirements are different since data will be stored for a long time (data will be processed afterwards) and will no be modified in a short space of time. Obviously, taking into account this characteristic, the approach to design the file system is different. As we focus to support the storage of media streams (large files) we can remove the complexity introduced by buffer caches and large blocks maps. Concepts as internal or external fragmentation are important for general purpose systems, but are not so critical in these kind of applications.

A critical point is how to search into the file system structures. This search must be optimized, avoiding complex data structures as AVL's used in current file systems as XFS. In some situations, as opening a file, the delay searching through the tables can be allowed (in our target), but the delay searching for free space is necessary to optimise.

We have discussed the design considering the disk access pattern, but is critical to know how disks work to improve the performance. The minimal allocation unit used by general file system has sense in the global view, but this can leads to a excessive disk head movement since physical blocks can not be consecutive for a file. We need to minimize the disk head latency as much as possible since it is critical to achieve good performance.

BUFFER CACHE

Disk latency is a bottleneck since CPU's speed began to grow as Moore's law predicts, and meanwhile disk were, and still are, restricted to a minor growth rate mainly due to the mechanical components inside. Operating systems use a tecnique to avoid this problem called Buffer Cache. This is a general memory buffer to allocate disk blocks temporally in main RAM memory that tries to avoid unnecessary disk requests.

Buffer Cache algorithms tries to exploit known disk access patterns as the short lifetime of some files (sometimes just seconds) and local and temporal references. Access patterns that are valid for general purpose systems and applications. Some of the main characteristics of buffer caches are:

1. Read ahead, based in local references: when a disk block is requested for read, then the next contiguous blocks of that file are read and stored in the buffer cache too.
2. Flexibility for disk policy: as write operations are delayed, the final disk scheduler can rearrange them to minimise disk head movements.
3. Extra copy from user buffer to system buffer cache.
4. Inconsistent state if a crash happens: data (and metadata) of the buffer cache still not written into the disk is lost when a crash happens, therefore there are more chances to lose more data.
5. Low size block to allow an easy management of the buffer: if these blocks are large a lot of resources are wasted when a few bytes are requested.

Points 3 and 4 are drawbacks and 5 is in conflict with the decision taken in the previous point about space allocation (large extents). Indeed, since other characteristics of general purpose operating systems as short life time of files or local and temporal references are not applicable for our purposes, it is not necessary in our design a buffer cache, therefore we can avoid the implementation[1].

RELIABILITY

As reliability we mean to obtain a consistent state of the file systems after a crash. Usually, file systems changes are made in structures allocated in memory which are eventually written to disk. If a crash happens before these changes are written to disk the file system state is not consistent.

Reliability is very related with the design of the file system. Log (or journal) structured file systems were designed to provide a fast way to recover data when a crash happens, which is a drawback with ext2 Linux file system, the first Linux file system implementation. But, with the log structured file system approach, reliability is achieved adding performance and resources penalty, along with a high complexity.

As one of the characteristics cited previously was simplicity, the reliability must not add excessive complexity to the design. And, of course, reliability should not achieved by loosing performance (only a minimal overhead is tolerable).

PORTABILITY

Although the indirect path (thru Linux processes) followed until now by RT tasks to read or write to/from disk was very "tricky", it has as strong point the possibility to work later with the data using Linux tools. Then, the file system used in RTLinux should can be used in Linux to work comfortably with the broad possibilities offered.

USER BUFFER ALLOCATION

RT tasks will use the file system with the standard `read` and `write` POSIX functions. These functions needs a buffer parameter, which is a pointer to a memory zone that will be used by the file system.

In the Linux approach, user buffer data is copied into the kernel buffers using buffers heads objects. This is a technique to improve performance, and works fine with general file systems due to the locality and temporal references concepts, storing data temporally in these buffer. Our expected workload will not exhibit temporal reference disk access, therefore buffers head are not necessary, avoiding to waste resources and the double data transfer, one from user buffer to kernel buffer and other from kernel buffer to disk. So, data buffer pointed by the read or write function is used directly by the device. This is possible because as we will see in the next point

read and write functions blocks the caller task, so does not exist the danger of task reusing the buffer before the end of the request.

Taking into account that the RTLinux memory allocation (OCERA DYNMEM component) do not handle DMA address ranges properly[2], the buffer allocation for read and write functions by a task must be done at initialisation time.

READ AND WRITE FUNCTIONALITY

Should a read or write call block? As we want to follow the POSIX standard for `read` and `write`, these functions must block the caller task. But, if we want to keep tasks inside the real time we need disk behaviour to be deterministic. Otherwise, to make sure a task reading continuosly from a device does not lose data samples, the task division paradigm usual in UNIX must be followed: one task reads data and other writes data to disk, sharing a buffer. We assume disk can support the bandwith required.

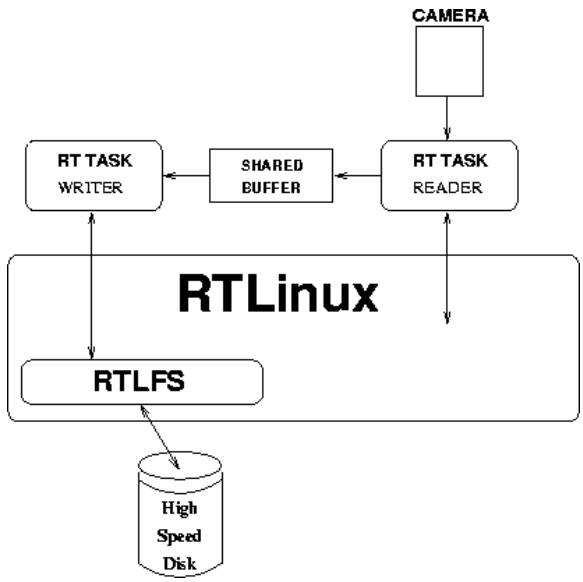

Figure 8-1. Reader and Writer division approach

TRUSTED ENVIRONMENT

Operating systems use file attributes to check access rights, along with attributes of the process accessing the file. This is a necessity in untrusted environments where the operating system has to enforce the access policy.

Since the RTLfs file system will be used in trusted environments, where all the code is written and controlled by the end user, there is no point to implement any kind of access protection. Indeed, files atributes are based on the idea of users and groups, which is not valid in RTLinux.

Another important aspect from a security point of view is how new blocks are passed to files. Assuming a trusted environment file system assigns blocks to files without deleted previous data (disk data blocks are not zeroed).

Parameters checking are very restricted in untrusted environments to avoid the bad use of the functions by a malicious process. As a trusted enviroments is assumed the checking can be more relaxed, mostly related to catch programming bugs.

8.8. Available open source realtime filesystems

As a previous step we have studied others file systems looking for how match with the characteristics defined in the previous section, and to evaluate the cost of migration of that file system to RTLinux.

We have found that the original file systems developed two decades ago fulfil the main points. It has sense as these first implementations were simple and executed in trusted environments, with a little disk capacity, and where disk latency was hidden by the lower performance of CPU's. This last point had motivated the main research of this field and the use of a general buffer cache. FAT is an example of these file systems and some of the characteristics are:

- Sectors are grouped in clusters, which can be as long as disk capacity[3].
- Linked list using an index method is used to manage clusters (groups of sectors). In each entry of the index appear the next cluster (if exists) owned by the same file or a special character if unused. The metadata management full-fills the simplicity required.

However, FAT file system are not very efficient when manages unused or spared clusters: the system must search through the index block table sequentially, what can lead to high variable search costs. This is a drawback to avoid in real time systems. On the other hand, traditional UNIX file system[4] have a more complex block allocation structures. A linked list for space management using this functionality would achieve the requirements.

The FAT file system implementation of Linux is tightly dependant on the internal Linux structures as buffer cache. This along with other complexities usual in file systems designed for general purpose operating systems as file access attributes or file access times[5], as well as legal issues (Microsoft™ is planing to request patent royalties) influenced in the decision of the implementation of a file system from scratch.

8.9. Real Time File System (RTFS) Specification

Once analysed the characteristics that the file system should provide, this section presents the proposed design and the implementation that meets these requirements. A full definition of structures appears in APPENDIX A. The next figure shows the global structure of the file system:

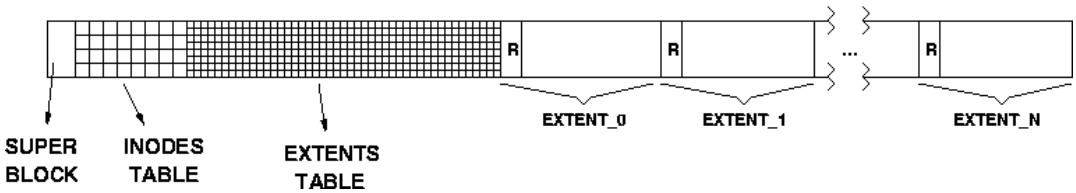

Figure 8-2. RTLFS structure

We present the file system components with the main characteristics:

- *SUPER BLOCK (SB)*

  The super block contains information describing the layout of the file system. For example, the number of sectors for inodes and extents tables are stored here, along with the extent size (an extent is a large number of contiguous sectors). In our design, the super block have two important fields to manage free space: pointers to free lists of inodes and extents. This is where our design changes with regard to how other file systems search through the linked list.

- *INODES TABLE*

Inodes are structures used to manage metadata of files: size, mode, permisions, pointers to data blocks, etc.

The decision to have a fixed number of i-nodes and extents length is to avoid complexity for data blocks allocation. In this way we can locate extents easily. The main drawback is the total amount of i-nodes the file system can have. In the current implementation the maximum number of sectors per extent is 128, what is the upper limit a DMA operation supports. With this limit the maximum number of files allowed is 1638 (inode size = 40 bytes).

Obviously this is very low number for a general purpose operating system, but we think that it is enough for embedded real time systems. It's possible that some real time applications need more files but it does not seem the normal case.

Usually in UNIX implementations directories are files which data is a list of files owned by these directories along with a inode pointer per file. In our design the file name is inside the inode structure since we are not going to support directories: only a root dirrectory for the file system.

- *EXTENTS TABLE*

  As i-node table, this is a fixed size structure created when the file system is formatted, along with the number of sectors per extent. In the current implementation the maximum number of sectors for the extent table is 128, as inode table. This sets the maximum number of extents to 16384 (an extent is a long).

  Following a simple approach, this structure is a linked list using an index and is maintained in memory to improve performance since the size is manageable thanks to the high number of sectors per extent. The next figure shows an example with a reduced extent table:

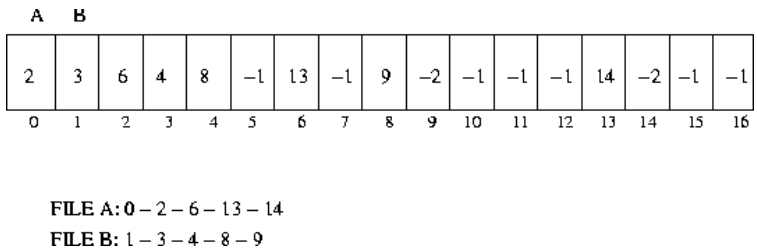

Figure 8-3. Extents table example

This approach has the advantage of the facility to found free extents when a file is deleted. As blocks owned by a file are linked, is easy to know what is the next free extent just following the links. This follows a simple algorithm to manage free extents, first found first served, althoug other algotithm could be used adding more complexity to the design. A Pinter to the head of the free list extents is stored in the super block.

- *EXTENTS RELIABILITY*

  The problem with reliability is how to maintain the file system consistent when system crashes (power failure, critical application bug, etc.), and is very related with how metadata is written to disk. File systems designers had taken different approaches to solve it: BSD file system writes metadata synchronously to disk, meanwhile Linux file systems write metadata just in buffer memory, and later aynchrously to disk.

  The metadata problem is due to the disk latency and the fact that metadata blocks may not be close to where current data is being written into disk, therefore it implies a large head movements to other disk zone. We have this problem with our design, as inodes and extents tables are fixed in the first sectors of the partition.

  We solve this using the first sector of the extents for reliability. The information written in the first sector is a *rtl_reliable_extent* structure (see APENDIX A):

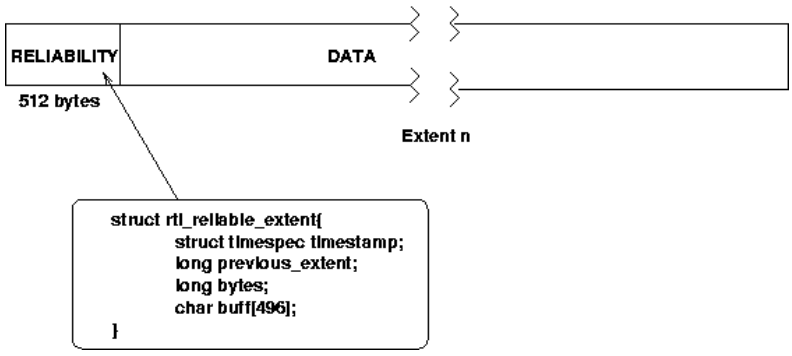

Figure 8-4. Extent header

The `timestamp` field is used to know the state of the extent related with the superblock. At recovery time, only extents with a timestamp newer than the superblock timestamp are processed for recovery. The `previous_extent` field helps to rebuild the extents list of a file. And in the `buf` field is stored the inode object of the file. With this information is possible to get a consistent state of the file system.

- *PERFORMANCE DECISIONS*

  We have explained that a general buffer cache is not necessary for our purposes, mainly due to the temporal reference characteristic is not a feature of the expected workload. However, the use of a minimal buffer cache has some advantages that could improve the performance of our file system.

  The explanation is easy to understand with an example: a process writing 500 bytes of data every 200 ms. Since the disk sector size is 512 bytes, the request does not fill a sector; and when the next request arrives, data will be placed filling the last 12 bytes of that sector and 488 bytes of the next. This behaviour implies the first sector must be read from disk before the second write, because we don't have a general buffer cache. Only in this very common situation the temporal reference is true.

  We solve this problem using a 512 bytes cache by file object that will be used when request are not sector aligned.

8.10. Features

- *PORTABILITY*: The main advantage of RTLinux is the possibility to use linux tools. As we want to maintain this, a file system module has been developed. Linux can mount the file system when is not mounted by RTLinux, but can do it in READ mode only.
- *USABILITY*: Some aspects of the file system can be managed from Linux:
  — mkfs.rtlfs works as other mkfs tools, using a parameter given to format a partition.
  — rtlfs.chk searchs the realiability data of a bad-unmounted file system building a consistent state. The file system will can not be used if it was not umounted (for example when a crash).
- *FILE SYSTEMS LIMITS*
  — The maximum size of a file is 2^{64} bytes.
  — The maximum number of extents per file is not limited.
  — Maximum number of files: 1638.
  — Maximum number of extents: 16384

8.11. Validation Criteria

Since this component has not utility without IDE device driver component, validation criteria will concern to both components, and it will be focused on the file system. The goal of the RTLinux block layer is to store media streams efficiently, then a criteria will be how many concurrent video streams can be written at the same time, and how it affects reading a video stream stored previously at the same time that other video streams are written. As one of the aims was to share the same IDE device between Linux and RTLinux, another validation criteria would be what is the performance of Linux when some RT-tasks are working with the IDE device.

8.12. Tests

We will test what is the behaviour of the system when one or several video strems are being written to disk. For this, we will make simulations of the different MPEG modes, and this will help us to limit the system possibilities. It will also be tested which is the system behaviour when a task reads a video stream stored when, at the same time, other tasks are writing data. Finally, to see how Linux is affected with this implementation, tests will be done with Linux doing several input/output oriented tasks at several levels (desktop user, server machine).

8.13. APENDIX A

```
struct rtl_disk_super_block{
      unsigned char  id;
      unsigned long  s_magic;
      unsigned long  start_sector;
      unsigned long  total_sectors;
      unsigned long  inodes_sectors;
      unsigned long  extents_sectors;
      unsigned long  extent_size;
      long           total_inodes;
      long           total_extents;
      long           current_files;
      long           current_extents;
      long           last_inode;
      long           last_extent;
      loff_t         sequence;
      unsigned short  mount_state;  /* 1 if system crashes */
}
```

Figure 8-5. Super block

```
struct rtl_inode{
      char           name[16];
      kdev_t         dev;
      int            pos_in_table;
      umode_t        i_mode;
      unsigned char  free;
      long           extent_id;
      loff_t         size;
}
```

Figure 8-6. I-node structure

Notes

1. However, as we will see in our implementation we need a 512 bytes cache per file for performance
2. Current memory allocator manages a pool of memory which is not ensured to be in accessible address range of the DMA hardware. Next revisions of DYNMEM should consider this problem.
3. Usual FAT implementations have a upper limit of 128 sectors = 64Kbytes
4. In the UNIX first version
5. If simplicity is a goal in the design we must avoid a lot of non critical features

Chapter 9. Stand-Alone RTLinux-GPL (saRTL)

9.1. Summary

Name
: Stand-Alone RTLinux

Description
: saRTL is a stand-alone core of the RTOS RTLinux.

Author
: Vicente Aurelio Esteve LLoret

Reviewer
: Ismael Ripoll

Layer
: Low level RTLinux

Version
: 2.0

Status
: BetaTesting

Dependencies
: The compilation requires Linux includes. The resulting code is directly executed on a bare machine.

Release Date
: M2

9.2. Description

saRTL is a stand-alone RTLinux core which runs in embedded systems as a complete RTOS without Linux or FreeBSD code.

One of the strongest points of RTLinux-GPL is that it is a hard real-time operating system that runs jointly with Linux. This combination forms a flexible system with all the functionality and features of a powerful desktop OS (graphic interface, complete network support, lots of hardware drivers, etc.), and a fast and predictable hard real-time system. Among the advantages of the standard RTLinux system we can mention: communication between the hard real-time tasks and the non-real-time application processes is fast; it is easy to port RTLinux to any architecture supported by Linux; fast application development, since it is not need to reboot the machine to test the application (if the system do not crash), just reload the application modules; etc.

But the standard RTLinux architecture has also several drawbacks:

- It has a large memory footprint. The Linux kernel code has to be included into the embedded system, and some intrinsic (not selectable) features are included but not used.
- RTLinux can only by ported to systems were Linux were ported previously. Due to design constrains Linux can only be ported to architectures that support memory paging, therefore a large range of small embedded processors can not be used.
- Although the RTLinux patch has been designed to virtualise the interrupt management, it is possible that some drivers or even user processes disable interrupts or lock the system for long periods of time. The problem is that Linux is still executed in privileged mode (ring level zero in ia32 architecture) so it is possible to directly execute assembler code containing these privileged instructions.

- The more code is executed in the system, the more cache and TLB misses occur. If a low priority Linux application works with a large a mount of data of if the user program have low spatial locality, then RTLinux tasks are throw out of the cache, which has a dramatic impact on the performance. This problem can be hardly solved because the cache replacement algorithm is transparently managed by the MMU processor.
- Long boot time. Linux boot and startup sequence may be long for some applications. With the Stand-alone booting is just to load the system image (which contains the application code), setup the memory and interrupt management, and jump to application code.

9.3. API / Compatibility

saRTL is POSIX 1003.13 compliant to the same extend than the Standard RTLinux. For all non-POSIX RTlinux funcionlities saRTL tries to be RTLinux compliant with the exception of all these new saRTL services.

Table 9-1. saRTL POSIX compliant funcionalities implemented.

```
pthread_self
pthread_create
pthread_attr_init
pthread_equal
pthread_kill
pthread_getschedparam
pthread_setschedparam

pthread_attr_getinheritsched
pthread_attr_setinheritsched
pthread_attr_getschedparam
pthread_attr_setschedparam

pthread_attr_getstackaddr
pthread_attr_setstackaddr
pthread_attr_getstacksize
pthread_attr_setstacksize
pthread_attr_init
pthread_attr_destroy
pthread_attr_getdetachstate
pthread_attr_setdetachstate

usleep gethrtime

sched_get_priority_max
sched_get_priority_min

pthread_spin_destroy
pthread_spin_init
pthread_spin_trylock
pthread_spin_lock
pthread_spin_unlock
```

```
sem_init
sem_wait
sem_post
sem_destroy
sem_getvalue

pthread_mutexattr_destroy
pthread_mutexattr_init
pthread_mutexattr_getprioceiling
pthread_mutexattr_setprioceiling
pthread_mutexattr_getprotocol
pthread_mutexattr_setprotocol
pthread_mutexattr_getpshared
pthread_mutexattr_setpshared
pthread_mutexattr_gettype
pthread_mutexattr_settype
pthread_mutex_destroy
pthread_mutex_init
pthread_mutex_getprioceiling
pthread_mutex_setprioceiling
pthread_mutex_lock
pthread_mutex_unlock

pthread_cond_init
pthread_cond_destroy
pthread_cond_broadcast
pthread_cond_wait
pthread_cond_signal
pthread_condattr_getpshared
pthread_condattr_setpshared
pthread_condattr_init
pthread_condattr_destroy
```

Some non-posix functions supported by saRTL are:

```
pthread_wait_np
pthread_make_periodic_np
rtl_request_global_irq
```

9.4. Implementation issues

The first attempt to do this porting was to replace the Linux booting and setup code by new developed code. But this approach showed useless due to the large amount of modifications required to detach the RTLinux code from Linux. It was not possible to run (boot) the system until all non-critical dependencies were removed. Therefore, we changed the porting strategy to do incremental code porting, that is, generate a small and naive booting image and then continue moving code from the RTLinux tree to the saRTL (tasks, context switch, scheduling, synchronisation, etc.).

Boot sequence is highly architecture dependent. Which may be quite complex as in the x86 architecture (where several processor modes has to be used to access hardware detections functions provided by the BIOS). Other processors/boards provide a clean and clever boot-loader which greatly simplifies saRTL boot sequence implementation. Compulsory requirements of boot secuence for all architectures are:

1. Stack initialization. This stack will be used by `start_kernel()` function and after that will be reused as the idle task stack.
2. Clean `.BSS` section (this section is used to store non initilized variables). If we don't clean this section, then it is difficult to trace errors, also some core relies on the fact that variables are zeroed.
3. Finally, jump to `start_kernel()` function where high level initialization is carried out.

Since the kernel size is quiet reduced it do not need to be compressed (most Linux kernels must be compressed, during boot time, due to the small memory space available in real mode. Loading an uncompressed image also speedup the booting time.

Standard RTLinux scheduler manages the Linux kernel as the lowest priority task, in other words Linux is the background task of the RTLinux scheduler. A new "Idle" task has been created to replace Linux as the background task when the system is idle. This idle task has been implemented as an infinite loop at the end of the `start_kernel()` function.

Memory allocation is still alpha code. A pointer to the end of the used memory marks the start of free space. `kmalloc()` returns the block following the used memory (page aligned) and advances the pointer of used memory. Memory is not freed. Next saRTL version will use the TLSF, which is an improved version of the DIDMA allocator also developed as part of the OCERA project.

By default, the stand-alone RTLinux implements a flat-memory memory design in the same way Linux does, where segments has base address 0 and limit 4 GB. Both, threads and executive, are executed in the higher processor privilege level (Ring0) for this reason it's only needed 2 segments: one for code and another for data. saRTL doesn't need to enable paging by default although it can be enabled if we enable memory protection (optional functionality).

Modules are not needed because thread code is compiled and linked jointly with executive in a single binary file. Function `init_tasks()` is used by users to initialise and create threads in a similar way `init_module()` does in a Linux module approach. Next is an example of `init_tasks()` use:

```
void init_tasks(void) {

#if CONFIG_OC_PBARRIERS
 pthread_barrierattr_init(&barrier_attr);
 pthread_barrier_init (&barrier, &barrier_attr, 2);
 pthread_barrierattr_destroy(&barrier_attr);
#endif

 if (BARRIERS_EXAMPLE) {
   pthread_create(&thread3,NULL,task3,0);
   pthread_create(&thread4,NULL,task4,0);
```

```
};
}
```

And then thread code can be implemented in another .c file:

```
#include <deblin/vhal.h>
#include <time.h>
#include <pthread.h>
#include <rtl_ipc.h>
#include <semaphore.h>
#include <rtl_barrier.h>
#include <rtl_conf.h>
#include <unistd.h>

static unsigned long valor3=0x0;
static unsigned long valor4=0x0;

pthread_t thread3;
pthread_t thread4;

#if CONFIG_OC_PBARRIERS
 pthread_barrier_t     barrier; // barrier synchronization object
 pthread_barrierattr_t barrier_attr;
#endif

void *task3(void *arg) {
#if CONFIG_OC_PBARRIERS
while(1){
  pthread_barrier_wait (&barrier);
  DebugString("Task3");
  valor3++;
  usleep(3000000) ;         // Sleep 3 seconds
};
#else
while(1) {};
#endif
return 0;
};


void *task4(void *arg) {
#if CONFIG_OC_PBARRIERS
while(1){
  pthread_barrier_wait (&barrier);
  DebugString("Task4");
  valor4++;
};
#else
while(1) {};
#endif

return 0;
};
```

To compile and link new code you can use a Makefile like this:

```
all: tsk00_barriers.o
        mv -f tsk00_barriers.o ../modules/

main: all

include ../rtl.mk

clean:
        find . \( -name '*~' -o -name '*.o' -o -name core \) -exec /bin/rm -r '{}' \;

.PHONY: dummy

include $(RTL_DIR)/Rules.make
```

User thread code is implemented in file `tsk00_barriers.c`. Prefix `tsk00` is used to store code in context 0 if memory protection optional functionality is enabled (prefix

names are used to allocate code on separate execution contexts when memory protection is enabled). With this makefile we get object file `tsk00_barriers.o` which is copied to module directory to link with the whole system.

9.5. Validation criteria

saRTL provides an environment fully compatible with RTLinux that fit memory requirenments for small and medium embedded systems. The same application code runs faster on a saRTL system than on standard RTLinux. There are two reasons: 1) Linux kernel and applications do not cause processor cache pollution; and 2) access to main memory is faster since no address conversions is done due to paging is not enabled. Also, the variability in the execution time of threads has been reduced.

9.6. Tests

The porting has been done following an incremental approach. Parallel to the porting, the tools needed to test and debug the ported code were developed. These tools are documented as a separated component since because the utility of these versatile debugging tools are beyond the scope of the low level kernel developer and can be easily used by applicator developers. The two tools developed are:

GDB Agent
: Allows step by step execution of thread code and executive code. Even we can insert breakpoins within interrupt handlers.

Tracer
: Non-POSIX compatible tracing utility which introduce low overhead to the embedded system.

The reader is referred to the Chapter 11, *Stand-Alone RTLinux debugging tools (**debugtools**)* for a complete description of the tools.

Following is a description of some of the tests implemented. More tests extra tests has been implemented to verify memory protection with different memory protection squemes. (See Chapter 10, *Stand-Alone RTLinux Memory Protection (**saRTLmprot**)*). The regression tests already available in the standard RTLinux code has not been used because most of them require the presence of the Linux kernel.

Test 1: Periodic Tasks
: Simple test to check timer and context switch. Two periodic tasks T1 (periodic 1 second) and T2 (periodic 2 seconds). This test has been executed with the two different timer modes: one-shot and periodic timer mode.

Test 2: Mutex
: Two threads (T1 and T2): T1 periodic with 3 seconds period and T2 non periodic. T1 locks a semaphore and sleeps until the next period. At the next period unlock semaphore and display a string on the screen. T2 is not periodic and display a message protected by the same semaphore.

Test 3: Semaphores
: Same than last test but using semaphores.

Test 4: Barriers
: 2 tasks. T1 waits 3 seconds and after that go into a barrier object. T2 just go into barrier object and when it leaves display a message. Both threads are sincronised and display a message at the same time.

Test 5: PosixIO
: Test `/dev/mem` devices which is created in PosixIO initialization. I move pointer write pointer to address `0xb8000` (video memory). Following i write in the devices displaying information in the screen.

Test 6: RTTerminal

Write test in the `/dev/rt_tty` devices and execution test of `rtl_printf()` which also use this terminal.

Test 7: Baker Test (CPU Usage)

This test measures the overhead introduced by the operating system under a know typical user load. Results will be different depending on the processor speed. The faster is the processor the better is the test.

AMD Athlon 1 GHz 99.8 usage

Pentium III 664 MHz 99.2 usage

Chapter 10. Stand-Alone RTLinux Memory Protection (saRTLmprot)

10.1. Summary

Name
: Stand-Alone RTLinux Memory Protection

Description
:

Author
: Vicente Aurelio Esteve LLoret

Reviewer
: Ismael Ripoll

Layer
: Low level stand alone RTLinux (saRTL)

Version
: Funcionality available since saRTL version 2.0.

Status
: BetaTesting

Dependencies
: saRTL for x86 architectures.

Release Date
: M2

10.2. Description

Reliability and robustness are important properties in embedded systems. Although code review and tests are necessary in the different development stages, we can never be sure to provide bug free code. One method to improve robustness is by limiting the effect of a programming bug to the task that caused the bug, which can be achieved via memory protection.

The memory protection requirements of embedded systems do not need to be as complex and powerful as that required in a multiuser system. Memory protection in a multi-task multi-user systems has to solve two problems:

1. Programming bugs do not spread beyond the scope of the faulting process. This is achieved by allowing a process to write only on its own data space.
2. Protect data from being stolen by other users (processes). This requirement forces the operating system to disable read access outside process address space.

An embedded system, where the whole application is written by a small group of well intended programmers, is not a battlefield where information is stolen. In this scenario, memory protection is used to intercept and capture programming bugs that passed unnoticed the implementation and testing phases. In fact, system wide read access is a desirable feature in an embedded system because it speeds up communication and simplifies the implementation of the RTOS (system calls that only read the status like `pthread_self()`, etc. can be implemented very efficiently).

Most processors provide memory protection associated with the virtual paging mechanisms and processor privilege level. While in user mode, only a subset of all the pages are accessible. To gain full access to the address space, the processor has to change to supervisor mode. This schema is very flexible and powerful. It is used by most operating

systems because the added protection achieved: memory protection as well as I/O and special processor control instructions (RTAI uses this mechanism in its LXRT module). The main drawback is that changing from user to supervisor mode is not a cheap operation, this operation use to require lot of processor state (registers, flags, cache, etc.) to be saved in main memory or being invalidated.

The embedded execution protection requirements are not that strong as in a general purpose OS. User application will be executed in supervisor mode, therefore we are only interested in memory protection that detect unintended programming bugs.

saRTL support 3 memory protection schemas:

- Flat memory without paging. This is the default option. This is the memory schema used by RTLinux where executive memory is accesible from whatever thread and all threads are able to access the whole memory.
- Executive memory protection (optional funcionality). This model protects the RTLinux executive against write access from application threads. Application threads are not protected among them.
- Context memory protection (optional funcionality). This is the more flexible model. Several contexts are created, each context contains one or more threads. Each thread will have total access to all threads within its context but it will have only read access to other contexts and executive.

This component implements memory protection only for the x86 architecture.

10.3. API / Compatibility

saRTL memory protection is a completely user transparent funcionality. Users must use a prefix in a object file name to define different contexts.

Faulting threads are terminated and a information related to the thread are printed.

10.4. Implementation issues

Several memory protection choices has been studied considering memory fragmentation, overhead and portability. Solutions based on segmentation were discarded due to portability problems (segments are only available in the x86 architecture) and the lack of support by compilers (gcc).

	Internal Fragmentation	**Portability**
Segmentation	Null	Low
Paging	Depend on page size	High

In Paging based schemas, the page size decision will affect not only internal fragmentation, but also to system overhead due to TLB misses.

	TLB Misses	**Internal Fragmentation**
Big Pages	Few misses	High
Small pages	A lot of misses	Low

The paging system is only used to protect memory pages, no address translation is performed. That is, the logical address generated by programs (kernel and applications) are translated to the a physical address equal to the logical address. The page size used in 4Kb.

We also use a special x86 feature to implement very fast but effective memory protection (RTLinux executive protection). The WP flag bit in the CR0 control register controls whether the page protection bits are honoured (bit set to one) or ignored (bit set o zero)

while in supervisor mode. Remember that in standard RTLinux, as well as saRTL, all code is executed in supervisor mode (ring level 0). If the WP flag is set, an attempt to access a page protected page triggers a page exception, which is handled by the kernel. Using the WP bit is very fast, it can be changed by a simple instruction and do not invalidate TLB entries. Most Uni*es (and Linux is not the exception) set this bit to one to implement the copy-on-write or to "mmap" (memory map) files. Even in some Linux versions some important data structures like IDT (Interrupt Descriptor Table) has been protected from privilege code access using read-only pages and the WP bit.

10.4.1. Executive protection implementation

When this memory protection mechanism is selected, paging is enabled at boot time. Only one single page directory is initialised so that logical pages are mapped into the same physical pages. Pages that contain the saRTL executive are marked as read-only pages (write is not allowed), and the rest of the pages are marked as read and write. Code and data are page aligned.

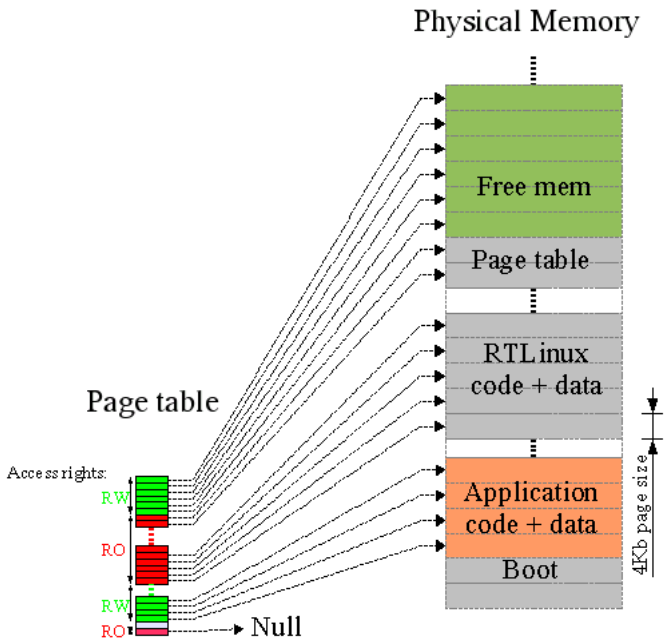

Figure 10-1. Executive memory protection layout

We can see that first lineal memory page is marked as invalid to detect null pointer access. Pages are 4 Kb long to reduce internal fragmentation. Kernel is mapped with read-only attributes and real-time threads with read/write attributes. Both executive and read-time threads are executed in supervisor mode. The only difference between executing threads code and executive code is that executive runs with WP bit cleared and threads with WP set. This WP bit modification has to be manually changed by saRTL API developers. Next is an example of how the executive changes the bit before it can access executive data:

```
int pthread_wakeup_np(pthread_t thread) {
#if CONFIG_KERNEL_MEMORYPROT
    mprot_t mprot;
#endif

    STARTKERNELCODE(mprot);
    pthread_kill(thread,RTL_SIGNAL_WAKEUP);
    ENDKERNELCODE(mprot);
    return 0;
}
```

All system calls that require write access to kernel data are surrounded STARTKERNELCODE and ENDKERNELCODE macros. The resulting new code is as follows:

```
#define STARTKERNELCODEP(v) do {      \
   asm volatile("movl %%cr0,%%eax \n" \
 "movl %%eax,%0            \n" \
 "andl $0xfffeffff,%%eax   \n" \
 "movl %%eax,%%cr0         \n":"=m" (v)::"eax","memory")} while(0)


#define ENDKERNELCODE(v) do {         \
   asm volatile("movl %%cr0,%%eax \n" \
 "andl $0xfffeffff,%%eax   \n" \
 "movl %0,%%ecx            \n" \
 "andl  $0x10000,%%ecx     \n" \
 "orl  %%ecx,%%eax         \n" \
 "movl %%eax,%%cr0         \n"::"m" (v)::"eax","ecx") } while(0)
```

As can be seen the overhead introduced in almost neglect-able. Interrupt gates based solution (change processor privilege mode using an interrupt as most OS's do) have a much higher cost. With interrupt gates, an interrupt handler has to be defined which verify service number and after that calls the correct functions (a table with the pointers of all the system calls has to be created, Linux sys_call_table). Besides calling the system service, the processor hardware has had to store in the stack several values like return address and flags, change stack pointer from user stack to superuser level stack and store user level pointer in TSS (task status segment) to restore user stack with iret assembler instruction.

This solution requires to modify most of the the executive system calls to add the corresponding macros. Once modified, the resulting code can be used transparently with protection enabled or not. User threads have direct access to executive name space (because they are compiled and linked with executive in a single binary file). There are no problem to perform executive calls because threads have read access to executive code and then in the first instruction of API function we modify WP bit to get complete memory accessibility.

It is important to note that the paging system is only used to achieve memory protections, and not a different virtual and physical address space. Lineal address generated by the programs and the resulting physical address are the same. For instance, physical address 0x6000 just can be mapped in lineal address 0x6000. This property make easy the development of tools like GDB agents and greatly simplifies the page table location problem.

Processor only works with the physical address of the page table but we need its lineal address to know where the PTD (page table directory) is. Of course we can translate this physical address to a lineal address but we need page table access to translate it. Linux systems solve this problem mapping the whole physical memory in a fixed lineal address like 0xc0000000. Other systems like Windows® NT solve this problem mapping page tables in fixed lineal memory addresses.

10.4.2. Context Protection Implementation

Context protection is an extension of the executive protection. We create several contexts where one or more threads can live. x86 implementation is based in the creation of several page table directories and active page modification changing CR3 intel register.

At boot time, the page table directory of each context is initialised. And once at run time, the only modification in all the saRTLcode is only line of code in the scheduler:

```
   if (s->rtl_current) != new_task) {
#if  CONFIG_CONTEXT_MEMORYPROT
    SET_CR3(context[new_task->contextid].cr3);
#endif
       rtl_switch_to(&s->rtl_current,new_task);
   }
```

The macro `SET_CR3` generates just two assembler instructions.

There is no need of a TSS (Task Status Segment) because these segments are only required when different privilege rings are used to provide the processor. TSS in that case is used to store user stack pointer in a supervisor level jump.

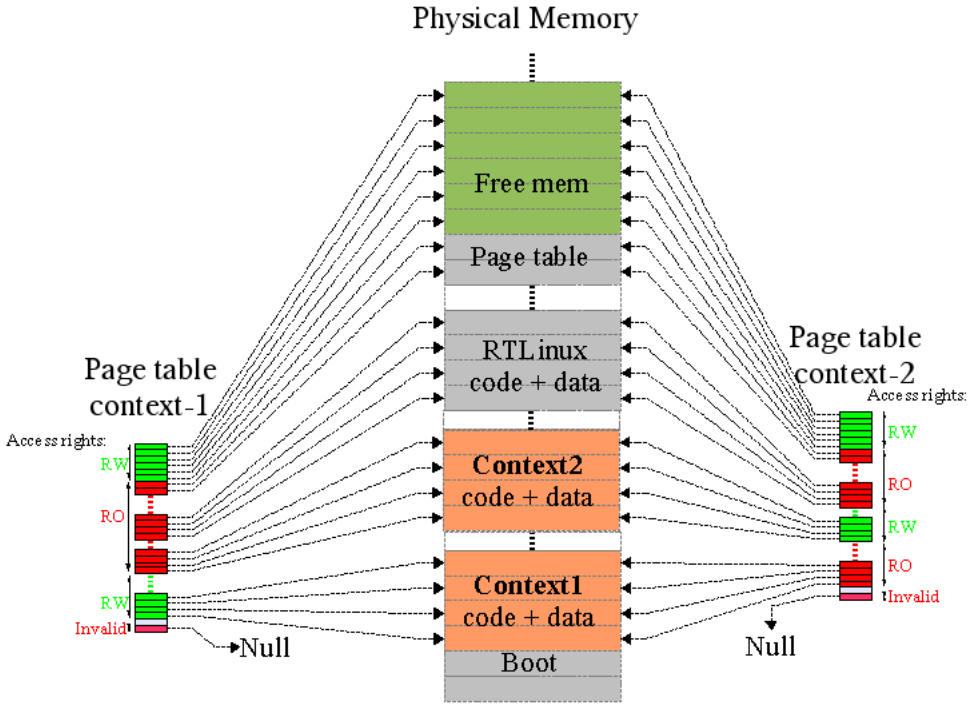

Figure 10-2. Example of two protected contexts

A thread within a context will have total access to threads within its own context and read access to the other contexts and executive.

A memory page can not be shared by code of different contexts, because these may have different protection attributes. For this reason it was necessary to use a linker script to split (into separate pages) object code of different contexts. This linker scripts creates the contexts using the name of the object files to allocate each object file into the appropriate context.

Object filenames starting with the prefix `tsk01_` are allocated in the context first context. Inside file object `tsk01_example.o` we can insert one or more threads. All the object files with the same prefix `tsk01_` will be linked into the same context. In this case all threads stored in these files will reside together in context one exclusive pages. In second context reside the object files that are prefixed by `tsk02_` and so on.

The page table directories that are initialised at boot time, use the information generated by the symbols automatically inserted by the linker script.

A very simple and naive page allocator has been implemented (allocated memory can not be freed). The memory pool used by the memory allocator is mapped at boot time. It is initialised with read-only attributes in all contexts. For this reason it is a memory suitable to be managed directly by the executive. There are two functions to request block (pages) from this pool:

- `kmalloc`. Reserve space page aligned with read-only attributes (memory only accesible by executive code, and all page table will have read-only attributes for this memory).
- `malloc`. Reserve space page aligned with read/write attributes for the active context. And only read access to the rest of the system contexts.

10.5. Validation criteria

Memory protection design has been implemented to detect programming errors with a low temporal overhead cost.

10.6. Tests

Several tests has been developed to test thread accessibility with the protection schemas:

Test 1: Executive accesibility test.

A variable located in the executive address space is read and written from a user thread.

The user thread runs correctly if we works with the default memory schema (no memory protection activated). When we enable executive protection a general protection fault is raised when the user thread tries to write in the executive variable.

Test 2: Context accesibility test.

A variable is declared in one context and it is accessed (modify) from other context. A general protection fault is raised when the variable is tried to be written.

Chapter 11. Stand-Alone RTLinux debugging tools (debugtools)

11.1. Summary

Name
: Stand-Alone RTLinux debugging tools.

Description
: Debugging tools available for the Stand-Alone RTLinux OS.

Author/s
: Vicente Aurelio Esteve LLoret

Reviewer
: Ismael Ripoll

Layer
: Low level RTLinux

Version
: Funcionality available since SA-RTL version 1.0.

Status
: BetaTesting

Dependencies
: These tools are an integral part of the Stand-Alone RTLinux for i386 architecture.

Release Date
: M2

11.2. Description

This component is a spinoff of the development tools used during the porting of the RTLinux to a bare machine (SA-RTL). The two tools implemented are:

GDB Debugging Agent
: The debugging agent: "is a small pierce of code running on the target that helps gdb to carry out requests to monitor and control the application being debugged". The implemented agent jointly with the GDB debugger opens the possibility to use all the power of the GDB debugger in SA-RTL applications. Among other it is possible to step by step execution of thread code and executive code; insert breakpoints; display and modify the variable values; etc.

  A special and very useful characteristic is that **it is possible to trace code (step by step and with breakpoints) while the processor has interrupts disabled**. This can be done because the communication between the implemented GDB agent and the host GDB is thru a serial line, and the serial line driver has been implemented using polling (not interrupt driven).

Tracer
: A non-POSIX compatible tracing utility which introduce low overhead to the embedded system.

11.3. API / Compatibility

The GDB agent implements the complete Remote Serial Protocol (RSP) functionality. Therefore all the facilities of GDB can be used.

The Tracer is not POSIX compatible looking for improvement in performance. A single function is needed:

```
RTL_TRACE(Event_Id,Event_Data);
```

11.4. Implementation issues

GDB Agent provides remote conectivity with GDB using the serial line. By means of a few basic commands like read/write registers, read/write memory and trace mode activation GDB is allowed to perform remote tracing. Serial port and register access provoke this implementation to be highly architecture dependent. The ELF file generated after compilation must contain enough information, for this reason we simply compile all files with *-g* compilation option which is automatically enable if we turn on GDB Agent in the configuration. File which we get in this way stores debug information to execute step by step all compiled modules, scheduler, OS internal functions and even interrupt handlers.

GDB agent code is completely independent of the rest of the code. It manages its own interrupts (interrupt 1, step by step execution and interrupt 3, breakpoints). RTLinux functions can not be called from the agent to void infinite loops. By default a breakpoint can be inserted in any place (except GDB agent code). The GDB Agent can not make use of the PosixIO serial port available in RTLinux. Otherwise the system may lock in an infinite loop: suppose that the GDB agent calls the posixio functions to communicate with the host gdb; then a breakpoint insertion in `write()` function will trigger an interrupt handler call. Interrupt handler call will generate a GDB Agent call and after that GDB Agent will call again PosixIO device (`write()` to send the required data to the GDB host); the system gets into a deadlock loop. GDB Agent must be as operative system stand-alone as possible with the main objective to provide not only a system user tool but a API programmers debug tool.

Interrupt handler execution is carried out completely with interrupts disable. If interrupts were enable undesirable behaviour will happen if a breakpoint is inserted in the scheduler code. Working with interrupts disable force us to make serial port readings using polling instead of interrupt based readings. Anyway interrupt based readings are possible when GDB Agent let the code run freely using the gdb "cont" command.

11.4.1. New GDB Agent funcionalities

GDB Agent allows, using the `rdtsc` assembler instruction, to measure (with very high accuracy and with almost no overhead) the time between two consecutive breakpoints. For this reason interrupt three handler (breakpoint exception) has been coded in the following way:

```
asmlinkage void breakpoint_interrupt(void); \
__asm__( \
 "\n" __ALIGN_STR"\n" \
 "breakpoint_interrupt:\n\t" \
 "pushl %eax\n\t" \
 SAVE_ALL \
 "rdtsc   \n\t" \
 "movl %eax,(breakpoint_end_time)    \n\t" \
 "movl %edx,(breakpoint_end_time+4)  \n\t" \
 "call "SYMBOL_NAME_STR(Breakpoint_Handler)"\n\t" \
 "rdtsc   \n\t" \
 "movl %eax,(breakpoint_start_time)    \n\t" \
 "movl %edx,(breakpoint_start_time+4)  \n\t" \
```

```
        RESTORE_ALL \
"iret \n\t");
```

RDTSC instruction must be executed after SAVE_ALL macro execution and before RESTORE_ALL macro to make sure register EDX and EAX are not modified by tracer code. These macros insert some undesirable time overhead to tracer measurement.

This overhead is calculated in the agent initialisation using the `calc_rdtsc_overhead()` function and compensated. Here we can see the simplicity of the procedure:

```
void calc_rdtsc_overhead(void)
{
   __asm("int $3\n\t");
   __asm("int $3\n\t");
};
```

The Difference time between two consecutive breakpoints is stored in a 64 bit long variable called `breakpoint_dif_time`. In this way It can be displayed by DDD (Data Display debugger, a GDB front end) using either printf or display commands as we can see in the following capture.

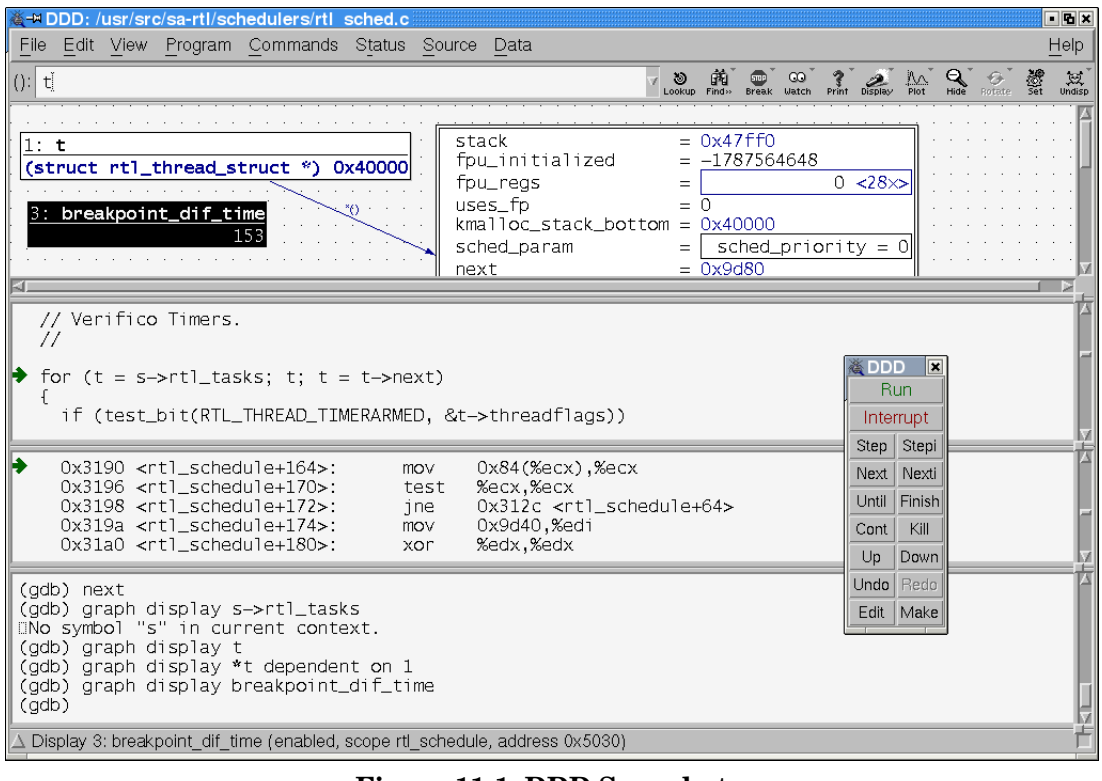

Figure 11-1. DDD Snapshot

11.4.1.1. Extensions to the standard GDB agent

News commands has been implemented using GDB serial protocol (RSP). The GDB "maintenance packet" allow us to design RSP extension improving standard GDB funcionalities. This instruction sends a command to the remote agent with the purpose to be processed by it and then the agent send back information to GDB debugger. The new commands implemented are:

- **Fcacheon**. Turn on cache in remote system.
- **Fcacheoff**. Turn off cache in the remote system.

These commands are useful to measure the worst case time.

11.4.2. Tracer implementation

A tracer is used to register several events like context switches, critical sections entry and exit, interrupt handler executions and so on. When the vents are generated, they a re first stored in memory (they are not immediately send to the host due to the communication time overhead). Standard RTLinux has advantages in the implementation of this kind of tools because communication between real time tasks and non-real time tasks is very fast, both resides in the same computer. This tracing tool tries to face the lack of efficient communication between host(Linux) and target (sa-rtl).

Events are inserted in the trace buffer. When this buffer is full a breakpoint is triggered and GDB agent stops current execution. GDB agent waits new commands and the user is allowed to dump event buffer using the GDB command **Dumptrace**. All this information is stored in a file (in the host machine) to be processed by a script which translates the trace dump in a format that can be displayed user friendly (**gtkwave**).

`RTL_TRACE` macro is used by users to get time information. Its main function is to store an event and call GDB agent when event buffer is full. Implemented like a macro and directly coded in assembler language let us to reduce debugging overhead.

```
#define RTL_TRACE(event,value)   {                    \
       asm volatile(" rdtsc                         \n"\
                    " shll $8,%%edx                 \n"\
                    " movl %0,%%ebx                 \n"\
                    " movb %%bl,%%al                \n"\
                    " movl %1,%%ebx                 \n"\
                    " movb %%bl,%%dl                \n"\
                    " movl (%2),%%edi               \n"\
                    " movl %%eax,(%%edi)            \n"\
                    " movl %%edx,4(%%edi)           \n"\
                    " addl $0x8,%%edi               \n"\
                    " movl %%edi,(%2)               \n"\
                    " cmpl %3,%%edi                 \n"\
                    " jnz  no_overflow%=            \n"\
                    "  int  $0x3                    \n"\
                    "  movl %4,(%2)                 \n"\
                    "no_overflow%=:                 \n"\
                    ::"r" (value),                     \
                      "i" (event),                     \
                      "m" (tracer_ptr),                \
                      "i" (tracerbuf[TRACERBUFSIZE]), \
                      "i" (tracerbuf[0])               \
                   : "eax","edx","ebx","edi","cc");   \
        };
#
```

`gdb2vcd.pl` is the Perl script that converts the raw trace dumped by GDB into a VCD[1] (Value Change Dump) file. This file can be displayed, among others, by the GTKwawe program available on most Linux distributions.

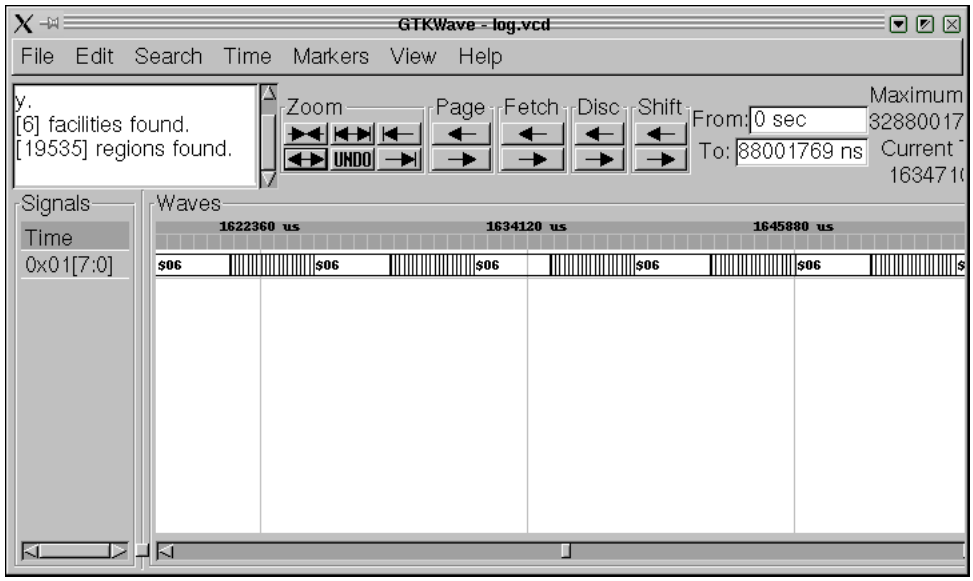

Figure 11-2. Execution Trace with GTKWave

11.5. Validation criteria

These debugging tools has been used extensively in the porting of Standard RTLinux to a bare machine. It would not be possible to do the porting without these kind of powerful tools.

Notes

1. CVD is a data format widely used electronic field (both analog and digital), but that can be also used to display program execution events

Chapter 12. Stand-Alone RTLinux-GPL porting to StrongARM processor (saRTLarm)

12.1. Summary

Name
: Stand-Alone RTLinux porting to StrongARM processors (saRTLarm)

Description
: Porting of the stand alone RTLinux executive to the StrongARM processor.

Author/s
: Vicente Aurelio Esteve LLoret

Reviewer
: Ismael Ripoll Ripoll

Layer
: Low level RTLinux

Version
: Funcionality available since saRTL version 2.1

Status
: BetaTesting

Dependencies
:

Release Date
: M2

12.2. Description

saRTLarm is a porting of the core RTLinux executive (currently developed to work on x86 processors) to the StrongARM processor.

This porting also shows that RTLinux can be ported easily to new hardware, since the amount of very specific processor code is quiet small.

The porting is not an independent saRTL tree but it has been integrated into the stand alone code. It is possible to select the processor architecture: the i386 (saRTL) or arm (saRTLarm) via the default tcl/tk config Linux and RTLinux config system.

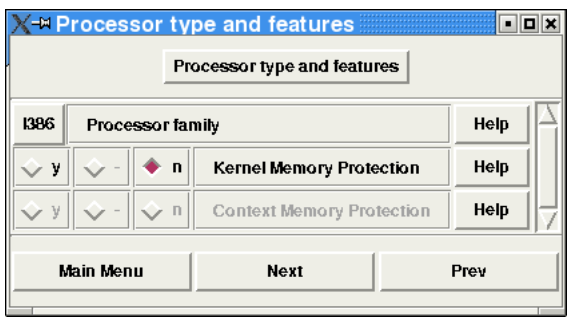

Figure 12-1. Processor type configuration

12.3. API / Compatibility

Partial port of the Stand Alone RTLinux API. Device dependent code has not been ported (except serial driver).

12.4. Implementation issues

The processor board used was the CerfPod developed by Intrinsyc, This is a full featured board based on an Intel StrongArm 1110 processor, an integrated 5'7 inches touch screen LCD display, Intel StrataFlash (16MB) and fast SDRAM (32MB). Peripheral support includes Ethernet, three serial ports, flexible digital I/O and on-board speaker. Also boundary scan bus (JTAG) is supported for board testing and Flash programming.

Figure 12-2. CerfPod Evaluation Board

Several GPL ARM gcc toolchains, some of them included in the development CD distributed by Intrinsyc along with the hardware. One of them is a customised gcc toolchain provided to compile a Cerf specific Linux version. The most used toolchains are arm-elf-gcc and arm-linux-gcc. There are only a few differences between both of them. The arm-elf-gcc is designed for operating system independent applications. For example, arm-elf-gcc will insert code for initialising global C++ constructors at the beginning of `main()`, so even no-OS platforms can work properly. As saRTLarm is a no-OS platform, the arm-elf toolchain version 2.59.3 has been used.

Also, there are several boot loaders available for this specific board and processor. The bootloader used has been i-boot-lite version 1.7 because source code availability (we need to modify interrupt table). Code of i-boot-lite is available in the software CD and it seems to have some advantages over other bootloaders.

At boot time, by default, the bootloader copies the image of the saRTLarm stored in flash memory into the SRAM memory so it can be executed. Once the image of the executive is loaded into RAM, the saRTLarm is ready to start running and the bootloader jumps to the _start function.

The bootloader provides a small set of utilities that can be used through a serial line. If at boot time, the user sends a newline character then the bootloader shows a command menu. Among other, it is possible to load new OS image in RAM by tftp (Trivial FTP) protocol using ethernet connection; copy the OS image from RAM to Flash memory; and load a new boot loader.

The very first tool required to start running and even debugging the embedded system is a printing function. The output device used is the serial line (the driver to send and receive characters was based on the bootloader code). The serial device is registered as the `/dev/sa1100_serial` special file in the RTLinux posixio device framework.

The use of standard output/input handler let us to modify rtl_printf behavior attaching different posixio drivers:

```
#if CONFIG_RT_TERMINAL
 init_rt_terminal();
 close(STDOUT);
 STDOUT = open("/dev/rt_tty",0);
#endif
```

```
#if CONFIG_SA1100_SERIAL
 init_sa1100_serial();
 close(STDOUT);
 STDOUT = open("/dev/sa1100_serial",0);
#endif
```

A large architecture dependent code, like timer programming, context switch and task stacks initialisation have been directly taken from RTLinux-GPL (Rtlinux patch for StrongARM is available in http://www.imec.be/rtlinux/). Nonetheless booting and interruption management have been implemented from the scratch.

There are some features of the standard Stand-Alone component that have not been implemented on the StrongArm porting yet, like for example the GDB agent or memory protection.

Although the StrongArm architecture supports MMU, this porting does not support it. This design criteria simplify the implementation but it also introduce some disadvantages to the porting.

Interrupt table resides in a fixed memory address like in almost all embedded systems. By default interrupt table resides in lineal address 0x00000000 (the interrupt table can also be located in the address 0xffff0000 if a flag is set in a special control register). The problem is that lineal address 0x00000000 is a FLASH memory address which access is only possible through special commands. The other possible table address (0xffff0000) is not a valid physical address. We can only enable interrupt table high address if we enable paging previously mapping a existent physical page in 0xffff0000 lineal page.

For solving this problem it has been modified all interruptions table entries, handled by the bootloader, to point directly to an accessible table located in RAM memory.

Contrarily to the i386 interrupt table, which is a table of pointers to the interrupt routines, the StrongArm processor interrupt table contains the code that is actually executed when the interrupt is attended. New code for interrupt table is (which is the modification done to the i-boot-lite bootloader):

```
        // This code is mapped into physical address 0x00000000
_start:
reset:
        // First interrupt vector
        b       run_rom
        // Undefined instruction vector
        // We are using this as an alternate entry point when running out of RAM
        // We should transition to using _start+0x20 as the offset instead.
        b       run_ram

        // Second intr. vector Software interrupt (SWI) vector
        ldr     pc,SWI_Handler

        // Prefetch abort vector
        ldr     pc,PREFETCH_Handler

        // Data abort vector
        ldr     pc,DATA_Handler

        // Reserved vector
        ldr     pc,RESERVED_Handler

        // IRQ vector
        ldr     pc,IRQ_Handler

        // FIRQ vector
        ldr     pc,FIRQ_Handler
        .word  0
        .word  0
SWI_Handler:     .word  0xc0008008
PREFETCH_Handler:.word  0xc000800c
DATA_Handler:    .word  0xc0008010
RESERVED_Handler:.word  0xc0008014
IRQ_Handler:     .word  0xc0008018
FIRQ_Handler:    .word  0xc000801c
```

Bootloader allways move saRTLarm binary from flash to `0xc0008000` address and then it jumps to this address to start OS code execution. Virtual interrupt table is stored in fixed address `0xc0008000` and is inicialized with the instruction `mov r0,r0` in order to not disturb normal saRTL execution.

```
start:
        .type        start,#function
        .rept        8               @ With this directive we can repeat instruction groups
        mov        r0, r0
        .endr
        b        1f
        .word        0x016f2818        @ Magic numbers to help the loader
user_stack:        .space        4000
end_user_stack:
1:
        adr     r2, end_user_stack
        mov     sp,r2
        mov     fp,#0
        ldr     r4,end_data
        ldr     r5,end_bss
1:      cmp     r4,r5
        strcc   fp,[r4],#4
        bcc     1b
        b       start_kernel
end_data:       .word _edata
end_bss:        .word _end
```

12.5. Validation criteria

We have implemented a saRTL porting to StrongARM processor setting the base to future portings to other architectures.

12.6. Tests

Tests which have been designed for x86 saRTL are completly compatible with StrongARM. To provide a full compatibility tests and for visualization porposes. A serial PosixIO devices has been implemented. `rtl_printf` will write in this devices instead of RT-Terminal device driver designed by Miguel Masmano in OCERA project.

Bibliography

[Aldea02] Mario Aldea-Rivas and Michael González-HArbour, 2002, 14 th Euromicro Conference on Real-Time Systems (ECRTS'02), *POSIX-Compatible Application-Defined Scheduling in MaRTE OS*.

[Abeni98] Luca Abeni and Giorgio Buttazzo, IEEE Real-Time Systems Symposium, Madrid, Spain, 1998, *Integrating Multimedia Applications in Hard Real-Time Systems*.

[Sha90] L. Sha, R. Rajkumar, and J.P. Lehoczky, 1990, IEEE Trans. on Computers, 39, 1175-1185, *Priority Inheritance Protocols: An Approach to Real-Time Synchronisation*.

[Baker91] T.P. Baker, 1991, The Journal of Real-Time Systems, 3, 67-100, *Stack-Based Scheduling of Realtime Processes*.

[Liu73] C.L. Liu and J.W Layland, 1973, Journal of the ACM, *Scheduling algorithms for multiprogramming in a hard real-time environment*.

[PTS] *Posix Test Suite*.

[BPA] *Bigphysarea*.

[Paul 95] Paul R. Winson, Mark S. Johnstone, Michael Neely, and David Boles, 1995, Proc. Int. Workshop on Memory Management, *Dynamic Storage Allocation: A Survey and Critical Review*.

[Ogasawara 95] Takeshi Ogasawara, 1995, 2nd International Workshop on Real-Time Computing Systems and Applications, *An Algorithm with Constant Execution Time for Dynamic Storage Allocation*.

[Ada95] T. Taft, R. Duff, R. Brukardt, and E. Ploedereder, 2000, Springer, Lecture Notes on Computer Science, *Consolidated Ada Reference Manual*, 3-540-43038-5.

[Baker00] T. Baker, 2000, SIGAda Conference, *Ada and Embedded Real Time Linux*.

[Lepreau02] L. Lepreau and M. Flatt, University of Utah, 2002, *The Oskit Project*.

[Aldea01] Mario Aldea-Rivas and Michael González-Harbour, 2001, Ada-Europe, *MaRTE OS: An Ada Kernel for Real-Time Embedded Applications*. .