LibeRTOS: A Configurable Single-OS Real-Time Linux Platform for Industrial Applications*

Thomas Gleixner
Linutronix
Uhldingen-Muehlhofen, Germany
tglx@linutronix.de

Douglas Niehaus
Information and Telecommunication Technology Center,
University of Kansas
Lawrence, Kansas 66045
niehaus@ittc.ku.edu

Abstract

The LibeRTOS project focuses on producing a stable, robust, and highly configurable Linux platform for industrial automation and other real-time applications. This paper provides an overview of LibeRTOS and discusses its most important capabilities which include: (1) high resolution time keeping, (2) extremely flexible and fine grain performance data collection support, (3) highly configurable scheduling policies, and (4) integration of OS computational components, including interrupt handlers, into the system scheduling model. The combination of these capabilities provides a single-OS platform appropriate for a wide range of industrial automation and other real-time applications. This paper provides an overview of the most significant and innovative aspects of the LibeRTOS system implementation and discusses recent experimental results comparing its performance to RTAI, a popular dual-OS real-time system, for both kernel based and user space computations.

1 Introduction

The LibeRTOS project focuses on producing a stable, robust, and highly configurable Linux platform for industrial automation applications. LibeRTOS has drawn extensively from the methods and experience accumulated by the authors over several years in the course of the KURT-Linux project at the University of Kansas, and of many industrial automation applications developed by Linutronix. This paper provides an overview of LibeRTOS and discusses its most important capabilities which include: (1) high resolution time keeping, (2) extremely flexible and fine grain performance data collection support, (3) highly configurable scheduling policies, and (4) integration of OS computational components, including interrupt handlers, into the system scheduling model. The combination of these capabilities provides a single-OS platform appropriate for a wide range of industrial automation applications.

Satisfying the requirements of real-time applications requires precise control over many fine grain details of system behavior to make the execution of applications as predictable as possible. One of the most difficult aspects of real-time system design and implementation is that the predictability of application behavior is largely determined by whatever aspect of the system is *least* predictable. One reason for the popularity of dual-OS approaches to real-time under Linux is that it permits the real-time portion of the system to be built from the start with an emphasis on predictability, with the Linux portion of the system executed only when the real-time OS permits. RTLinux [3] was the first system to use this approach under Linux, followed fairly quickly by RTAI [4].

*This work was supported in part by DARPA PCES contract F33615-03-C-4111 and by NSF EHS contract CCR-0311599.

The dual-OS approach is a fast and relatively simple way to create support for simple real-time computations, but it also suffers number of drawbacks. Most obviously, it strongly separates the real-time and Linux portions of the system, requiring special support for real-time computations to access Linux system services, and requiring implementation of all system services within the real-time environment. The memory footprint of the dual-OS system is significantly larger than that of the base Linux system as a result of system service duplication, and the complexity of applications with components in both the real-time and Linux domains is increased by the need to communicate across the boundary.

In contrast, the single-OS approach implements real-time system services within Linux, but faces a much more difficult problem in making the system predictable precisely because it is affected by the behavior of all system components. The KURT-Linux system [2] was the first to take this approach to real-time under Linux, and was developed at roughly the same time as the first version of RTLinux. The first version of KURT-Linux was suitable only for the most undemanding types of real-time applications because of significant event response time and scheduling jitter arising from a number of sources.

Over the last several years, continuous development of both KURT-Linux methods for supporting real-time within Linux, and in the methods used by the base Linux versions for concurrency control, have steadily improved the predictability of KURT-Linux and have improved the precision with which it can control use of system resources. The most recent set of improvements in KURT-Linux have removed essentially all the remaining performance advantages of the dual-OS approach. This prompted us to create the first version of LibeRTOS, which is intended to be a single-OS real-time Linux for industrial use.

LibeRTOS offers developers a broad set of powerful capabilities giving it significant advantages over other current approaches to real-time computing under Linux, and which are not fully matched by any other single system. As a single-OS system it has significant architectural and resource use advantages over dual-OS approaches for many applications. As a highly configurable and completely GPL system it has significant advantages over proprietary single-OS approaches that limit the user's ability to modify or replace significant portions of the system. LibeRTOS is thus an important and attractive addition to the set of current real-time Linux systems which can significantly increase the range of industrial applications using Linux. The LibeRTOS and KURT-Linux projects will continue to cooperate closely as two parts of a single effort. KURT-Linux will serve as the research and prototyping environment contributing features and methods to LibeRTOS as they mature. LibeRTOS will serve as a source of requirements and ideas for new research topics, as well as a proving ground and technology transfer platform for KURT-Linux based systems research.

2 Architecture Overview

The power and flexibility of LibeRTOS is a result of modifications to several aspects of the base Linux implementation and addition of several new capabilities. This section discusses the design of the major components of LibeRTOS, including how they help provide support for real-time and embedded systems.

2.1 High Resolution Time Keeping

The most obvious aspect of support for real-time computations is fine-grain time keeping and event scheduling. This was the basic capability addressed by the first KURT-Linux implementation [2], which divided the methods for time keeping and scheduling the interrupt representing the next scheduled event. Steady refinement of the implementation has resulted in a well defined internal LibeRTOS API for time keeping which is easily ported to a range of architectures. While the current version of LibeRTOS only runs on the x86 CPU, we have ported previous versions of KURT-Linux using this internal API to a variety of other processors including: StrongArm, XScale, and PPC CPUs.

On the x86 architecture we use the Time Stamp Counter (TSC) as the system time standard and the standard 8254 timer chip to generate interrupts at scheduled times. This approach permits us to keep track of time at the resolution of the CPU clock, and to schedule event interrupts with accuracy essentially equivalent to that provided by the current version of RTAI. More detailed performance results under various operating conditions are provided in Section 3.

It is worth noting that the integration of computation component control under the current LibeRTOS implementation has made it relatively easy to achieve significant improvement in clock calibration and clock synchronization. We have recently performed initial experiments using a 1.4 GHz Pentium machine in which we were able to calibrate the basic time keeping constant, TSC ticks per second, to within 40 parts per billion against the NTP time standard in our local network. This extremely low drift rate has made it relatively easy to synchronize the LibeRTOS system clock to within less than 10 microseconds of the NTP standard, while sending

synchronization messages every five minutes. This synchronization precision is at least an order of magnitude better than that provided by the standard NTP methods on the same system. LibeRTOS is thus capable of addressing distributed real-time applications with quite stringent clock synchronization requirements.

2.2 Performance Evaluation Support

The precision of the resource control in a real-time system must be matched by the precision of the measurement methods used to evaluate system behavior. The performance evaluation framework provided by LibeRTOS is called Data Streams, and supports collection of both kernel level and application level performance data. A wide range of events can be recorded, resulting in extremely large streams of recorded events from some experiments. Instrumentation overhead can be an issue, especially if the set of data sources for a given experiment is configured naively. Data Streams also supports the collection of aggregate data types, including counters, histograms, and application-specific "objects" capturing a snapshot of a specific set of state variables.

Sets of data from the OS and from applications can be combined and then processed after an experiment is complete to extract a wide range of information about system and application performance. Detailed visualizations are also possible, and frequently aid the developer in identifying unexpected or problematic sequences of events. The fact that the combined data sets help the developer discover how events at all levels of the system relate to one another is particularly important. We have recently been running a series of experiments in a related project which collected a complex set of events from the application, OS and ACE/TAO middleware layers.

The wealth of performance data has increased the complexity facing the developer in properly organizing the instrumentation of the relevant portions of the application and system code, in configuring the set of performance data that is collected for a given experiment, and in how the data collected can be processed to extract desired information. We have developed and are continuing to improve an extensive set of support tools for helping the developer maintain name spaces of instrumentation points, to enable and configure specific sets of instrumentation points for a given experiment, and to support a wide range of post-processing on collected performance data to extract specific kinds of information.

2.3 Fully Integrated Computation Component Control

One of the most important features of LibeRTOS is the strongly integrated control over all computation components on the system. We use the term "computation components" for two reasons. First, to emphasize that many computations of interest to users contain several concurrent components, and that LibeRTOS emphasizes control of computations as a whole, which means the group of components implementing a given computation. When people consider computations containing concurrent components, they most often focus on sets of concurrently executing processes or threads. This is the most common case under LibeRTOS as well.

The second reason we use the term "computation components" is to emphasize that the LibeRTOS scheduling model includes computation components that exist only within the Linux kernel whose execution is normally concealed from the user: hard-IRQ handlers (top halves), soft-IRQ handlers, bottom halves, and tasklets. These computations are distinguished by the fact that they *borrow context* from the currently running thread. The LibeRTOS scheduling model, called *group scheduling* treats all computation components in the same way, providing a highly configurable and extremely flexible framework within which users can describe a wide range of scheduling policies for *all* computational activities in the system.

It is important to note that while the system designer *can* explicitly specify the scheduling semantics for all computation components on the system they are *not required* to do so. The group scheduling framework uses the existing Linux scheduling policies for all computation components by default, permitting the user to specify specialized semantics only for the desired components. The unified scheduling model was demonstrated in KURT-Linux [6], and is currently being integrated into LibeRTOS. At this time it controls hard-IRQ and soft-IRQ execution, and we will add control of bottom halves and tasklets shortly. The group scheduling framework is discussed in greater detail in Section 2.4.2.

2.3.1 Integrating Concurrency Control

At this point it is important to note that unifying the control of all computation component types under the LibeRTOS group scheduling framework also required us to create a method for integrating the control of concurrency in the Linux kernel, for several reasons. The most obvious is that concurrency control and scheduling semantics often interact, since the purpose of a concurrency control model is to pre-

cisely control the conditions under which a set of computation components can execute concurrently. LibeRTOS has modified the control of all types of concurrency managed by the Linux kernel, which can be labeled: thread, interrupt, and physical. The default configuration for these modifications under LibeRTOS reproduces the original Linux semantics,

We use the term "thread concurrency" to refer to how multiple threads execute in relation to each other, which can be either on the same CPU or on different CPUs within a multi-CPU system. The key point is that these concurrency issues have to do with how the concurrent execution of two threads needs to be controlled under both the scheduler and the concurrency control model. The group scheduling framework controls the most obvious aspect of thread concurrency under LibeRTOS, while changes to the preemption support permit controlling the preemption of one thread by another under any desired semantics.

We have integrated hardware interrupt handling under the group scheduling framework by transforming the concurrency control method from hardware based interrupt enabling and disabling to a software based "big interrupt flag" model reproducing the same semantics. There are some very small regions of the kernel that still disable interrupts at the hardware level for extremely brief periods, but for all but a small number of specific issues, the interrupt handling has been placed under software control. The default configuration for hard-IRQ handling reproduces the "as fast as possible" semantics of Linux. It is important to note that specific configurations are free to modify the policy controlling the execution of interrupt handlers individually, or as a group.

The control of physical concurrency under Linux, spin-locks, have also been modified to integrate them under group scheduling. This was motivated by their involvement with the thread preemption control methods, as the semantics of physical concurrency control remain unchanged under LibeRTOS.

2.4 Configurable Scheduling Semantics

The integration of all computational components under the group scheduling framework emphasized configurability of the system semantics. The default configuration of the system provides the semantics of unmodified Linux, which is useful in simplifying potential complications during system boot. Most of the support for configurability takes the form of function pointers which call the configured routines. For example, function pointers provide access to the routines determining when and how hard-IRQ handlers are handled, when and how soft-IRQ handlers are executed, and when the group scheduling system decision function is invoked. For performance reasons, the modifications to preemption and spin-lock control are configurable only at compile time. However, we have found it relatively simple to provide implementations which support the default semantics during system boot, so this has not proved to be a significant constraint so far.

2.4.1 Configurability Example

One of the first LibeRTOS configurations we have built which significantly alters the default Linux semantics for several aspects of computation component control semantics is called the "M68K" configuration because a core component features the multi-level interrupt masking semantics made familiar to many by the Motorola 68000 family of CPU architectures. This configuration concentrates on control of threads and hard-IRQ handlers, leaving the control semantics for other computation components in the default configurations.

The M68K configuration divides all thread and hard-IRQ handlers into real-time and non-real-time classes. Within each class, each hard-IRQ is associated with a blocking level. Further, each thread also has an associated blocking level. The blocking levels of the real-time class of computation components are higher than those of the non-real-time components. The configuration provides the multi-level interrupt masking semantics by replacing the default hard-IRQ handling routine with one supporting the multi-level interrupt masking semantics. The desired thread scheduling semantics are implemented by providing a replacement for the preemption control routine which considers the blocking level associated with each thread when making a preemption decision. Interrupts which are non-real-time cannot, thus, interrupt the execution of real-time threads.

Further, this model also permits specific spin-locks to be associated with specific IRQ blocking levels, demonstrating the broad configurability of the LibeRTOS computation control framework. Implementation of this configuration required providing versions of the hard-IRQ handler control, preemption control, and spin-lock LibeRTOS components which implemented the desired semantics. It is important to note that while the total amount of code required to implement this configuration is quite modest, the modification of the system semantics is significant.

Consider an example scenario with two real-time threads; one whose execution is invoked by a periodic timer interrupt, and one invoked by asynchronous interrupts from a device interface card. In this scenario the timer interrupt has a higher blocking level than

the device, and the invoked threads have blocking levels matching that of the interrupts by which they are invoked. Under the M68K configuration, when the timer interrupt occurs, no card interrupt will be serviced until both the timer interrupt handler and the thread it invokes have completed execution, because they have a blocking level greater than that if the interface card. In contrast, when the interface card raises an interrupt, execution of its handler and the thread it invokes can be preempted by a timer interrupt.

2.4.2 Group Scheduling

A group is defined as a collection of computation components with an associated scheduling decision function (SDF) that selects among the group members when invoked. Each member of a group can have information associated with it, as required by the SDF. Groups can also be members of other groups, thus supporting hierarchical composition of more complex scheduling decision semantics, culminating in the creation of an SDF for the system as a whole; the system SDF (SSDF). A subset of the computations on a system can be placed under SSDF control because the default Linux scheduler is invoked to make a decision if the SSDF does not make a choice. Computations can be placed under exclusive control of the SSDF or joint control of the SSDF and the default Linux scheduler as the user desires. Further, the SSDF can also explicitly choose to invoke the default Linux scheduler.

Any number of SDFs may be implemented, but most systems can be implemented using selections from a standard set. We implemented a number of SDFs under the latest KURT-Linux version [6], including: static priority, dynamic priority, explicit plan, cyclic, processor share, round robin, EDF, and sequential. Most of these have now been transferred to LibeRTOS, and the rest will be done shortly. The interface for a scheduling decision function is quite simple: it takes the scheduling information describing the members of the group with which it is associated as input, and it returns a decision, which can take one of three forms: a computation component ID, *Pass* or *OS* which invokes the default OS scheduler. The group scheduling API includes the ability to modify the scheduling parameters associated with each member of the group. For example, if the SDF of a particular group is priority based, then the group API gives access to the SDF API that makes it possible to change the priority of group members.

The group scheduling framework emphasizes modularity of the SDF implementations and thus makes it relatively easy for users to implement their own SDFs if the library of available functions does not include one matching the scheduling semantics they desire. The group scheduling framework can thus easily subsume all of the popularly scheduling models by providing matching SDFs. However, it can support application specific and otherwise highly specialized scheduling semantics with equal ease,

We have, for example, recently conducted some experiments with group scheduling control of sets of pipelined computations operating on streams of messages representing streams of video frames, where each frame has a sequence number. In this scenario we compared performance under popular priority and CPU share based scheduling policies to performance under an SDF which was aware of application progress. Specifically, the "progress aware" SDF looked at information published by each processing pipeline about which frame number was last complete. The processing scenario included variable processing time for different frames within various pipeline stages. Given this information, it was reasonably simple to write an SDF that balanced the progress of the various streams much more effectively than the SDFs which were not aware of the application state.

We are currently exploring a number of ways in which the group scheduling framework can be used to improve support for various kinds of distributed real-time and embedded systems. One effort, the one using the computation load modeling video frame processing, is examining how group scheduling may be able to coordinate execution of application, middleware and OS computation components to improve execution using the ACE ORB (TAO) as the experimental platform. A second effort is considering how to implement real-time quality of service for network connections by integrating the network protocol processing soft-IRQ routines under group scheduling, and by implementing a simple time division multiplexing strategy for sharing a local network among a modest number of machines. A third effort is considering how group scheduling might be used to coordinate computation component execution in a GRID based computation.

3 Performance

This section gives a brief view of how LibeRTOS performance compares to one of the most popular dual-OS solutions, RTAI, and its support for real-time computations in user space, LXRT. The tests described here were performed on a 300 MHz Pentium machine for fairly long periods running a variety of loads using a LibeRTOS kernel based on Linux 2.4.26. These results were taken with permission of the author from a recent thesis by Jan Altenberg [1].

The measurements presented in the two tables are the maximum latencies for executing a minimal service routine associated with a test interrupt generated periodically by a custom device at 1 millisecond intervals.

	RTAI	LibeRTOS
No Load	7,8 usec	7,8 usec
ping -f	8,6 usec	9,8 usec
hackbench	12,6 usec	11,8 usec
hb + ping -f	13,6 usec	13,4 usec

TABLE 1: *OS Handler Latency*

In each table, the "No Load" line shows the maximum latency on an "idle" machine which means that it was supporting only the default Linux system processes in addition to the instrumented application. The "ping -f" or "ping flood" load is used to create a large number of interrupts from the Ethernet device, which provides competing interrupt load where the interrupt handler for the competing load is lightweight. The "hackbench" load adds a local processing load to the test machine and is disk intensive. The disk interrupt handler typically requires more processing time than the Ethernet handler, and tends to introduce more scheduling jitter than Ethernet interrupt processing. The last line in the table show the results when both the "ping flood" and "hackbench" loads were present on the system.

Table 1 presents the results for hard-IRQ handler latency where the main routine for responding to the test interrupt was implemented in the OS. Obviously, LibeRTOS and RTAI performance is extremely close under all tested load conditions.

	RTAI LXRT	LibeRTOS
No Load	18,92 usec	20,45 usec
ping -f	19,10 usec	21,14 usec
hackbench	33,10 usec	35,16 usec
hb + ping -f	34,56 usec	35,40 usec

TABLE 2: *User Space Handler Latency*

Table 2 show the maximum latency for the same basic scenario as that of Table 1, except that the handler run in response to the interrupt is in user space, rather than in the kernel. Again in this case, the performance of the RTAI and LibeRTOS platforms is extremely close, although there is a consistent advantage for RTAI of roughly 2 microseconds that was not present in the results for the OS based handler. It is clear from these results that few if any applications would find RTAI an acceptable platform but not consider LibeRTOS acceptable on the basis of response latency for user space computations. It is worth noting that in LibeRTOS development so far we have emphasized clarity and simplicity rather than performance optimization so some improvement in LibeRTOS performance is plausible.

4 Conclusions and Future Work

This paper has provided an overview of the current version of LibeRTOS. The design of the system has emphasized the creation of a stable, robust, and highly configurable Linux platform for industrial automation and other real-time applications. The key elements of the LibeRTOS system architecture are: (1) high resolution time keeping, (2) extremely flexible and fine grain performance data collection support, (3) highly configurable scheduling policies, and (4) integration of OS computational components, including interrupt handlers, into the system scheduling model. This paper has provided an overview of how the elements of the architecture, singly and in combination, implement a single-OS platform appropriate for a wide range of industrial automation applications. The recent experimental results described demonstrate that LibeRTOS performance, in terms of execution latency, is essentially the same as that of RTAI, a popular dual-OS RT system, for both kernel based and user space computations.

While the current capabilities of LibeRTOS make it suitable for a wide range of applications, a number of developments are also planned for the future. In the near future we will be experimenting with time division multiplexing on local area Ethernet which will strongly benefit from the fine grain time keeping, clock calibration, clock synchronization, and soft-IRQ computation component scheduling under LibeRTOS. Another near-term effort will consider how to extend the data stream support for performance evaluation to collection and post-processing of performance data from more than one machine in a distributed real-time system. The recent improvements in clock calibration and synchronization will significantly simplify aspects of this work.

In the middle term, we will be porting LibeRTOS to Linux 2.6 and continue development of methods for using group scheduling to create integrated control of all computational components (application, middleware, and OS) in ACE/TAO based applications [5]. In the longer term there is also work planned to provide an ACE interface to both LibeRTOS kernel based group scheduling as well as a portable middleware based implementation of group scheduling. Other work will consider how an FPGA based group scheduling implementation might be

used to help improve real-time system performance and predictability as part of a more general HW/SW co-design project [7].

References

[1] Jan Altenberg, 2004, *Adaption of a CNC on Real-Time Linux and the Comparison of Possible Scheduling Strategies of Different Linux Derivatives*, Diplomarbeit (Thesis), Fachrichtung Informationstechnik, Berufsakademie Stuttgart.

[2] B. Srinivasan, S. Pather, R. Hill, F. Ansari and D. Niehaus, 1998, *A Firm Real-Time System Implementation Using Commercial Off-The Shelf Hardware and Free Software*, Proceedings of 4^{th} Real-Time Technology and Applications Symposium.

[3] FSM Labs, *RTLinux*, http://www.fsmlabs.com

[4] DIAPM RTAI, *DIAPM RTAI — Realtime Application Interface*, http://www.aero.polimi.it/~rtai/

[5] M. Frisbie, D. Niehaus, V. Subramonian and C. Gill, 2004, *Group Scheduling in Systems Software*, Workshop on Parallel and Distributed Real-Time Systems.

[6] M. Frisbie, 2004, *A Unified Scheduling Model for Precise Computation Control*, Master's Thesis, University of Kansas.

[7] D. Niehaus and D. Andrews, 2003, *Using the Multi-Threaded Computation Model as a Unifying Framework for Hardware-Software Co-Design and Implementation*, Proceedings of Ninth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS), Capri, Italy, October 2003.