ELECTRONICS AND ELECTRICAL ENGINEERING
ISSN 1392 – 1215 *2010. No. 8(104)*
ELEKTRONIKA IR ELEKTROTECHNIKA

SYSTEM ENGINEERING, COMPUTER TECHNOLOGY
T 120
SISTEMŲ INŽINERIJA, KOMPIUTERINĖS TECHNOLOGIJOS

Hard-Soft Real-Time Performance Evaluation of Linux RTAI Based Embedded Systems

E. Dodiu, A. Graur, V. G. Gaitan
Stefan cel Mare University,
Universitatii no. 13, Suceava, Romania, phone: +40768887175, email: edodiu@usv.ro

Introduction

This article presents a few practical results obtained by testing a specific hardware architecture based on one of the most popular operating systems, Linux. We also added a real time extension to see if it suits the demands of a Hard or Soft real-time system.

Real-time systems are a bit different from the classical ones because they have to offer a certain response within a specified time period. With real-time systems, correct execution of tasks will depend not only on the correctness of the results but also on the time when they are provided. Unlike soft real-time where deadline miss is not a major problem, missing a time constraint in a hard real-time system can cause severe material damage or even human health injury [5],[10].

For this reason, real-time systems issue must be analyzed accordingly. Real-time system design is often a great engineering challenge because a lot of factors within the system can modify significantly its stability. Depending on the application, one can go for a hard real-time controller if deadlines must always be satisfied or for soft real-time if the application allows missing the time limits .

Depending on the possibility of adding new tasks at runtime, operating systems can be classified as static or dynamic.

If static operating systems are frequently associated with hard real-time, we cannot say the same for dynamic operating systems. In the first case, a predefined number of tasks are created at compile time without the possibility of adding new tasks when the system is up and running. Dynamic operating systems do allow insertion of new tasks during runtime, so there is no need of stopping, recompiling, reprogramming and restarting the devices. From this point of view, the execution speed is in inverse ratio to the scalability of task execution, and searching for the optimal solution is a main task for system designers.

The main destination of Windows or Unix like operating systems aims to Desktop computers, graphical stations, workstations, mainframes. They are not for industrial usage. We must emphasize that this huge computing power of the previous mentioned machines does not imply real-time execution as compulsory regulation. Lately, engineers made considerable efforts to classify those operating systems in the hard real-time domain but their behavior shows that they are not the best choice for industry embedded controllers.

This paper shows some practical results of tests that were conducted on a custom hardware platform to point out the system performances, jitter analysis, and whether they can be classified as hard or soft real-time. Jitter is a key element in evaluating system performance and real-time capabilities. That is why we will insist on this topic in the following pages. The paper consists of three main parts: theoretical background of real-time scheduling, architecture description and test case where practical results will be evaluated and finally, there are the conclusions.

RTAI (Real Time Application Interface) is a Free Software project developed in the Department of Aerospace Engineering of Politecnico di Milano(DIAPM) by a team of engineers coordinated by Professor Paolo Mantegazza. RTAI runs under Linux kernel space and due to its integrated scheduling policies it allows real time applications to be executed in a preemptive hard real-time environment [2]. RTAI can run both in single and multiprocessor environments [4],[8]. The RTAI Official Website provides the latest versions of software, API documentation and additional information for project development. RTAI supports several hardware architectures: x86 (with and without FPU and TSC), x86_64, PowerPC, StrongARM, ARM7, and a few chips from Cirrus Logic and Intel.

Similar testing has been performed by Eng. Peter Laurich, founder of Akamina Technologies. In his article “A comparison of hard real-time Linux alternatives”, he evaluates the performance of native Linux and Linux with RTAI for a high performance hardware platform. Most of the benchmarks and testing done until now aimed at high performance processors that go with desktop computers, workstations, servers and other expensive computing

platforms. The intention of this paper is to provide performance evaluation results for a custom hardware platform that uses one of the most used processors in multimedia, gadget and industrial applications, the ARM920T processor.

Mathematical model of a real-time task

Tasks are the main entities that are executed in a real-time operating system. They can be periodic or aperiodic and they may have time restraints [5, 6].

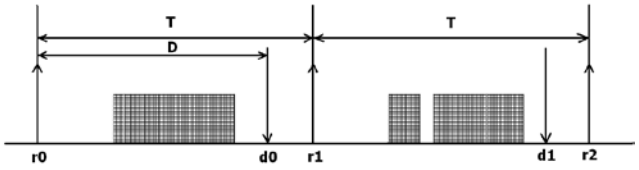

Fig. 1. Task main parameters: r – release time (periodic or event triggered); C – worst case execution time; D – task relative deadline; T – task period (specific only to periodic tasks)

If a task has a real-time constraint, then we can also talk about the absolute deadline, which is described by: $d = r + D$. Missing the deadline d determines a miss of a time constraint that is imposed for that task, which represent a major fault for real time systems. All four parameters mentioned above are present in case of periodic tasks but T is missing for the aperiodic ones. For aperiodic tasks, successive release times are of the following form: $r_k = r_0 + kT$ where r_0 is the time of the first release and r_k is the execution with sequence number $k+1$. Absolute deadlines are described by the equality $d_k = r_k + D$; if $D=T$ then the maximum admitted execution time (deadline) is equal to the period. A task is well formed if the expression $0 < C \leq D \leq T$ is true [1].

The following parameters that are derived from the above relationship are presented below:

-$u = C/T$ is the processor utilization factor for a running task and must always be equal to or less then 1

-$ch = C/D$ is the processor load factor and must always be equal to or less then 1:

- s is the start time of task execution;
- e is the finish time of task execution;
- $D(t)=d-t$ is a residual relative deadline at time t: $0 \leq D(t) \leq D$;
- $C(t)$ is the pending execution time t: $0 \leq C(t) \leq C$;
- $L=D-C$ is the nominal laxity of the task and specifies the maximum lag for the start time s when it has sole use of the processor.

Sometimes periodic requests must have fixed start times and response times. The difference between the start times of two consecutive requests, s_i and s_{i+1} is the start time jitter. The maximum jitter or absolute jitter is defined as $|s_{i+1}-(s_i+T)| \leq Gmax$. The maximum response time jitter is defined in the same manner [1].

The processor utilization factor for a set of n periodic tasks has the following expression

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}. \qquad (1)$$

The processor load factor for a set of n periodic tasks can be evaluated using the following expression

$$CH = \sum_{i=1}^{n} \frac{C_i}{D_i}. \qquad (2)$$

Because of the deadlines, neither the load factor (2) nor the processor utilization factor (1) is enough to evaluate an overload effect on timing constraints. Additionally, the processor laxity parameter $LP(t)$ was introduced and it represents the maximal time the processor may remain idle after t without causing a task to miss its deadline. Furthermore the processor idle times are the intervals where the processor laxity is strictly positive [1].

Rate monotonic algorithm

If we had to make a scheduler taxonomy of the selected software environment, we can say that it has the following characteristics: on-line centralized scheduling, preemptive environment and fixed task priority. The rate monotonic algorithm was taken in consideration for analyzing periodic tasks behavior in the test system. For a set of periodic tasks, assigning the priorities according to the RM (Rate Monotonic) algorithm means that tasks with shorter periods get higher priorities. For a RM algorithm, the worst case scenario is to have all the tasks from a task set triggered at once ($r0=r1=r2=\ldots=rn=0$).

Considering two tasks, t1 and t2 with $T1<T2$ and $D1=T1$, $D2=T2$, it is possible somehow by mistake to assign a higher priority to the task with the shorter period. In this case it is important that the inequality $C1+C2 \leq T_1$ to be satisfied [1]. If the correct assignment of priorities is done, and considering $\beta=T2/T1$ the number of periods of task t1 entirely included in the period of t2, then the following statements are correct:

$$C_1 + C_2 \leq T_1 \Rightarrow (\beta+1) \cdot C_1 + C_2 \leq T_2, \qquad (3)$$

$$C_1 + C_2 \leq T_1 \Rightarrow \beta \cdot C_1 + C_2 \leq \beta \cdot T_2. \qquad (4)$$

Statements from (3) and (4) show that if the schedule is feasible by an arbitrary priority assignment, then it is also feasible by applying the RM algorithm. Schedulability test for this algorithm implies determining the upper bound U_{max} processor utilization factor. For the same tasks mentioned above this is determined by using relation (5)

$$U_{max} = \frac{C_1}{T_1} + \frac{C_{2\max}}{T_2}. \qquad (5)$$

The generalized result for a set of n periodic tasks is done using the following relation

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \cdot (2^{1/n} - 1). \qquad (6)$$

Due to priority assignment based on task periods, the RM algorithm performs well for tasks where relative deadlines are equal to periods. The condition (6) is sufficient for processor utilization factor [1].

Performance characteristics of Linux RTAI based systems

Since Linux tasks present a variable jitter of unaccepted values, embedded systems built using this operating system are not real-time. To improve system performances and for adding real-time support RTAI extension was taken in consideration [2].

RTAI was built as additional part for Linux and can be executed both in kernel mode as well as user space. With RTAI, engineers can create tasks that run in the kernel address space with a higher priority than the kernel itself. Usually, the kernel is assigned with the lowest priority in the system and, by means of scheduling policies, the RTAI dispatcher is able to execute higher priority or greater urgency tasks.

Test equipment description

Since RTAI does not provide hardware support for the new ARM920T based chips integrated by Cirrus Logic we decided to use the Adeos Patch file for 2.4.26 kernel provided by Technologic Systems that came with the TS 7300 test board. New kernel support for the specified architecture is still in progress.

For testing we used the following software and hardware tools:

HARDWARE

- Technologic Systems TS7300 test board with the following components:
    - EP9302 Cirrus Logic Processor (ARM920T at 200MHz);
    - 32 Mbytes Samsung SDRAM;
    - Peripherals: IO ports, USB port, JTAG, CAN, USART;
    - Real time clock;
    - Altera CycloneII FPGA directly programmable from Linux;
    - SD card storage;
    - PC104 expansion slot.
- Power supply.
- 5 port Ethernet Switch.
- Compact Flash programmer.
- Tektronix TDS2024B oscilloscope.

SOFTWARE

For the development board:

- Linux Debian operating system;
- 2.4.26 Kernel;
- Adeos Patch for Technologic Systems;
- RTAI version 3.2;
- GCC 3.3.4 compiler for ARM;
- Glibc 2.3.2 libraries for ARM;

For PC development:

- Red Hat Linux 9.0;
- Cross compiler for ARM;
- 3.2.2 GCC compiler (Red Hat Linux 3.2.2-5).

Preparing the test environment

Preparing the test environment implies a series of steps that are described next. The order in which they are written is important.

Crosscompiler build. This step implies downloading and compiling of crosstool-0.28. For this *LD_LIBRARY_PATH* variable must be initialized with a null string and then from the file "demo-arm.sh" the line "cat arm.dat gcc-3.3.4-glibc-2.3.2.dat sh all.sh -notest" must be uncommented. Compilation is initiated using "./demo-arm.sh" command.

Installation of modutils-build. This is done according to the following procedures:

- The package must be decompressed using "tar –jxvfmodutils-2.4.26.tar.bz2"command.
- Installation script is configured using: "../modutils-2.4.26/configure –prefix=/home/user/… --target=arm-unknown-linux-gnu".
- Export necessary system variables "Export PATH=/opt/crosstool/arm-unknown-linux-gnu/gcc-3.3.4-glibc-2.3.2/bin:$PATH".
- Install with "make" and "make install" commands.

Apply ADEOS patch. This patch is a chunk of low level code that is used to build the hardware abstraction layer between Linux and the hardware platform on which it runs, in this case, the EP9302 processor.

The patch is applied in the following steps:

- "tar –zxvf linux24-ts8-kernelsource.tar.gz";
- "cd linux24";
- "patch –p1 <cale-catre-adeos/ts72xx_ts8_adeos.patch".

Kernel compilation. Using the new crosscompiler obtained in the previous step, the kernel is compiled for ARM architecture. First, it is extracted from the containing archive using "tar –zxvf tskernel-2.4.26-ts11-src.tar.gz"

The variable *CROSS_COMPILE* must be set to "arm-unknown-linux-gnu-" in the makefile. *Depmod* must also be set to "Export PATH=/opt/crosstool/arm-unknown-linux-gnu/gcc-3.3.4-glibc-2.3.2/bin:$PATH", and next, the kernel build is done by means of the following statements:

- "Make ts7200_config";
- "Make oldconfig";
- "Make dep";
- "Make vmlinux";
- "Make modules";
- "Make modules_install".

This last command also builds the modules that will be used further with the kernel.

Experimental results

This section contains the experimental results for the jitter and preemption latency test cases. The test programs are compiled on a desktop PC with the same compiler that was used for kernel building since using a different compiler will trigger an "*insmod*" error when trying to insert the test module into the kernel address space. For maximum efficiency, we used RTAI as a kernel module and for that reason test programs were compiled also as kernel space modules. Thus, we were able to test the real power of Linux when it comes to dynamic scheduling. Each of the test modules, including RTAI, can be inserted during runtime without the need to restart the machine which brings a major advantage as compared to other operating systems since the machine's uptime is not

affected. Anyway, writing kernel modules requires following the same strict rules of coding that are used in device drivers [9]. Time intervals were measured using a pin toggle method and because access to pin data registers is not possible using pointers, a call to "*request_region()*" was necessary. This function receives as parameter the starting address of the memory region to be used and an offset in bytes that specifies the size of the region. The time intervals are measured using a Tektronix 2024B oscilloscope synchronized on the rising edge of the signal (Fig. 2). This measuring instrument allows keeping on the screen all the variations of the signal in a specified time interval as this can be seen for the falling edge from Fig. 2. The measuring instrument also provides information about the period and the frequency of the signal between the two cursors. Each test was performed for about 30 minutes to have enough time to see the jitter variation, but even though this time interval is big enough, the values obtained herein will not show the worst case scenario.

Jitter test case 1 – idle CPU

This test shows the jitter of a 300 µs recurrent task when the processor is idle for most of the time (Fig. 2). In the picture, the main 300 µs rectangular waveform that actually defines the task execution time is represented with high intensity yellow, and the jitter is low contrast yellow near the falling edge of the signal. Since the worst case value for this test is 22 µs, the jitter is at most 7,4% from the task's main period because, even if we consider that the CPU is idle, it still runs the scheduler and some system services.

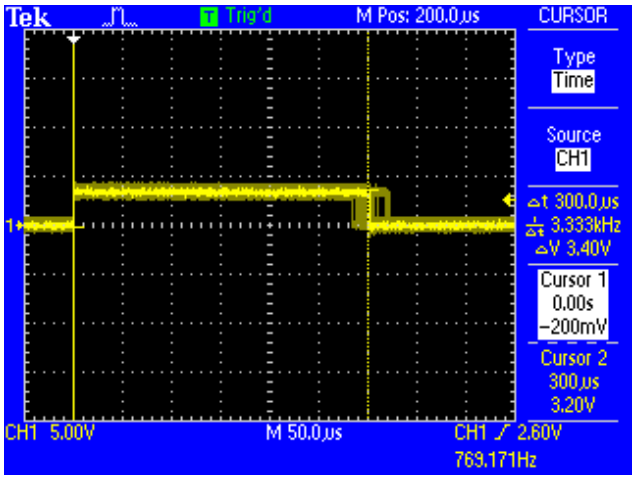

Fig. 2. Jitter for idle CPU

Jitter test case 2 – full load CPU

What is really important is to see how the system performs when high CPU consumer tasks need to be executed. Tasks with intensive DMA accesses, Ethernet traffic, memory page swapping and disk or compact flash accesses are only a few examples of high CPU consumer applications. From the above mentioned "ping –f" command performs just well by keeping the CPU in full load. Actually, the system sends ICMP packets in bursts without waiting for a response from the target interface. The results for this second test case are presented in Fig. 3. As we may observe in this situation, the jitter has a bigger value than in the previous case. The worst case jitter value obtained here is 100 µs, and this actually represents about 33% from the expected 300 µs period of the task. We will make a few comments about this result in the conclusions section of this paper as this isn't what we had expected.

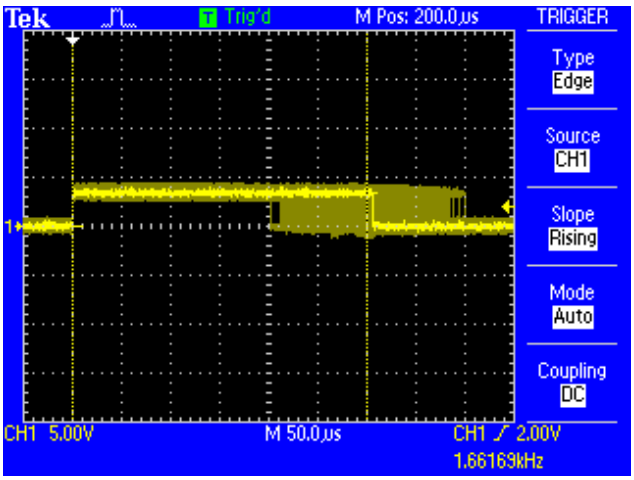

Fig. 3. Test case 2 jitter - full processor load

Using the same test cases, we performed a few measurements for different task periods. The results are shown in Table 1.

Table 1. Jitter values for different task periods

Task period (µs)	Minimum jitter value (µs)	Maximum jitter value (µs)
300	3	100
500	2	92
1000	2	80
2000	2	72
5000	2	66
10000	2	94

Preemption latency and task switch time test case

Preemptive multitasking allows embedded systems to be more reliable when it comes to catching and processing external asynchronous events. Unlike cooperative scheduling, each task can be interrupted from its activity by a higher priority task and this is a major advantage as external asynchronous events can be processed in real-time by dedicated software routines in the same processor system. In a real-time system it is very important to keep the preemption latency to minimum. All the extra delay will finally consume useful time from the task execution interval. Control is not passed instantly to the task which is responsible for exception processing, and this can be seen in Fig. 4.

Tex is the time when the external event first signals the control system. After a very short time, when the vectored interrupt controller provides the address of the service routine, control is passed to the code from ISR at time *Tisr*. The task that computes the external event request will not take control until a context switch is performed at *Tcs*. This action saves the context of the running task (internal processor registers and the program

counter) to a stack, and gives control to the new task that is elected by the scheduler.

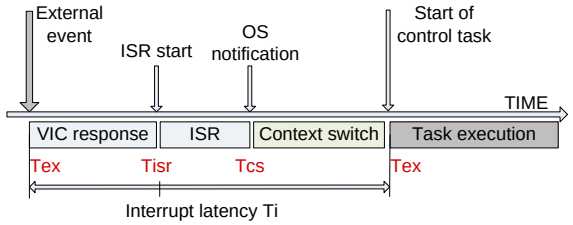

Fig. 4. Preemption latency intermediary states

EP9302 processor has 4 timers that can be used for time measurements but certain aspects determined us to use the pin toggle method. Three of them are sourced by a 508 KHz clock and the fourth by a 983.04 KHz reference, which gives us a resolution of about 1.96 µs and 1.01 µs respectively. This time resolution is good enough but because reading timers requires additional processing time, the decision was to use the oscilloscope measurement.

The test environment is built using the same hardware tools as in the previous tests, but additionally, a rectangular wave generator was added. This is used to trigger an interrupt using a 1 KHz rectangular signal that is applied on a pin with interrupt generation possibility. A task is programmed to answer the external event by toggling another IO pin from the same expansion connector of the test board. Both signals are recorded simultaneously using two probes, and the lag between the rising edges of the two rectangular pulses represents the latency we intend to measure (Fig. 5).

The last test case is performed to make a few observations on the task switch time for the proposed hardware/software architecture. As mentioned before, passing control to a higher priority tasks requires intermediary steps that have as main goal to save the context of the interrupted task to a stack.

Testing was performed in the same manner for full and minimum load processor for a range of input frequencies between 10KHz and 500Hz. The results are presented in Table 2.

Table 2. Preemption test latency for minimal and full load processor

Stimulus signal frequency (Hz)	Minimum load (µs)	Full load (µs)
500	42	42
800	40	42
1000	40	42
5000	42	44
8000	39	43
10000	41	44

Table 3 presents the results of the task switch time from a low priority to a high priority task.

Table 3. Task switch times for different processor loads

	Minimum load (µs)	Maximum load (µs)
min	28	30
max	32	38

Fig. 4 shows that interrupt latency takes a bit more time because of the hardware response and the ISR execution that happens before the context switch to the new task that will serve the external request. In this situation, the total time of the interrupt latency is *Ti=Tex+Tisr+Tcs*. On the other hand, the task switch time will only execute the context swapping (*Tcs*) and this can easily be seen by comparing results from Table 2 and Table 3. Depending on the hardware architecture, this process of context saving can spend less or more time. Practical results showed us that the ARM920T processor can produce impressive throughput with over 309 Mbytes/s at 200MHz core clock using block transfer *STMIA* instructions. That is why overall system performance depends on both the hardware architecture and the way the software is built.

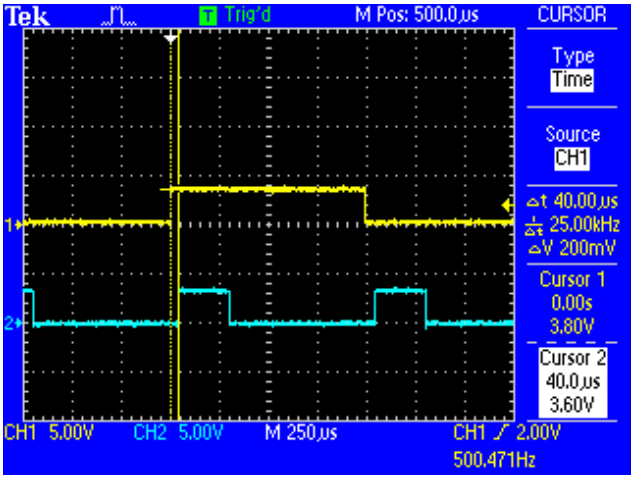

Fig. 5. Interrupt latency. The yellow is the input stimulus signal and the blue one is a small task that is triggered by the pin status change ISR. The task is triggered both on the rising and the falling edge of the input signal depending on the actual configuration of the interrupt associated with that I/O pin.

Conclusions

As mentioned in the introductory part of the paper, high level operating systems do not have the best suited performances for real time applications. Even though, we showed that the proposed hardware and software architecture can obtain good raw performance as compared to other operating systems from the same category (e.g. Windows).

Generally, a Windows Mobile based embedded device cannot go down below 1 ms resolution. Obtained results are way better than the previous mentioned value and the order of magnitude is with at last one order smaller than the Windows systems. As we see in the test case 2, the jitter at full processor load has a tendency to drop as the task period increases. For the first value in the Table 1 the result is not satisfactory since the jitter value is at most 33% from the task's main period. Thus a 300 µs task would often become 400 µs in width possibly delaying other important processes in the system. With high frequency real-time systems this is not acceptable. As for the larger period tasks, the jitter seems to be acceptable as long as it will not exceed the values imposed by system designers. Depending on the application, this system may

be used as a hard real-time controller, but it couldn't be used for controlling an anti-lock brake system because of the high jitter of the low period tasks. Since any hard real-time control system is without doubt better than a soft real-time controller, the system can be used without problems with soft real-time control. Taking in consideration the proposed hardware architecture with its 200 MHz processor the overall impression is good. By using this, one may benefit from the high level services offered by Linux, file systems, MMU, software tools, as well as the real time performance that comes with RTAI. As a future work, we will try to add some new improvements to real-time system performance using cache locking mechanisms.

References

1. **Cottet F., Delacroix J., Kaiser C., Mammeri Z.** Scheduling In Real–Time Systems. – England: John Wiley & Sons Ltd, 2002.
2. **Zhang G., Luyuan C., Yao A.** Study and Comparison of the RTHAL–Based and ADEOS–Based RTAI Real–time Solutions for Linux // First International Multi–Symposiums on Computer and Computational Sciences, 2006.
3. **Chiandone M., Cleva S., Sulligoi G.**, PC–based feedback acceleration control using Linux RTAI // 13th European Conference on Power Electronics and Applications, 2009.
4. **Balasevicius L., Dervinis G., Baranauskas V., Derviniene A.** Identification of the Unknown Parameters of an Object // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 4(100). – P. 33–36.
5. **Li Jun, Yang Fumin, Lu Yansheng**, A feasible schedulability analysis for fault–tolerant hard real–time systems // 10th IEEE International Engineering of Complex Computer Systems, 2005.
6. **Chang–Gun Lee, Joosun Hahn, Yang–Min Seo.** Enhanced analysis of cache–related preemption delay in fixed–priority preemptive scheduling // The 18th IEEE Real–Time Systems Symposium, 1997.
7. **Ho I., Jin–Cherng Lin.** A method of test cases generation for real–time systems // First International Symposium on Object–Oriented Real–time Distributed Computing, 1998.
8. **Morkūnas K., Šeinauskas R.** Circuit Reset Sequences based on Software Prototypes // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 7(103). – P. 71–76.
9. **Rubini A., Corbet J.** Linux Device Drivers, 2nd ed. – O'REILLY, 2001.
10. **Ching–Chih Han, Kwei–Jay Lin, Chao–Ju Hou.** Distance–constrained scheduling and its applications to real–time systems // IEEE Transactions on Computers, 1996. – Vol. 45. – P. 814–826.
11. **Gaitan N. C.** Real-time Acquisition of the Distributed Data by using an Intelligent System // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 8(104). – P. 13–18.

Received 2010 02 15

E. Dodiu, A. Graur, V. G. Gaitan. Hard-Soft Real-Time Performance Evaluation of Linux RTAI Based Embedded Systems // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 8(104). – P. 51–56.

A system is considered to be a hard real-time if all the computations are not only correct but their response is provided each time before a fixed deadline. Missing a deadline for such a system can possibly produce severe material damage or human health injury, therefore adequate testing and analysis should be done. This paper evaluates the suitability of a general purpose operating system like Linux with a real time extension. As it can be seen further on, Linux has overall good raw performance that can make it usable for an embedded system with industrial usage. To show the characteristics under full load stress, we have done several test cases for different scenarios that will be presented next in the paper. Ill. 5, bibl. 11, tabl. 3 (in English; abstracts in English, Russian and Lithuanian).

Э. Додю, А. Граур, В. Г. Гайтан. Исследование производительности операционной системы RTAI в реальном времени // Электроника и электротехника. – Каунас: Технология, 2010. – № 8(104). – С. 51–56.

Системы работающие в реальном времени отличаются дополнительном отказом, который может повлиять на здоровье человека. Анализируются новые системы с применением операционных устройств типа „Линух". Проведены эксперименты при полной нагрузке системы для разных случаев работы системы RTAI. Ил. 5, библ. 11, табл. 3 (на английском языке; рефераты на английском, русском и литовском яз.).

E. Dodiu, A. Graur, V. G. Gaitan. Linux RTAI operacinės sistemos našumo realiu laiku tyrimas // Elektronika ir elektrotechnika. – Kaunas: Technologija, 2010. – Nr. 8(104). – P. 51–56.

Kartais sistemoms dėl vienokių ar kitokių priežasčių sunku veikti realaus laiko režimu. Tokiu atveju skaičiavimai gali būti klaidingi ar pateikiami anksčiau nustatyto termino. Dėl to gali sugesti technika arba kilti pavojus žmogaus sveikatai. Atliekami „Linux" operacinės sistemos, kuri pasižymi dideliu našumu ir gali veikti kaip integruota sistema pramoninėse sistemose, bandymai ir analizė. Atlikti iki galo apkrautos sistemos testai skirtingais atvejais. Il. 5, bibl. 11, lent. 3 (anglų kalba; santraukos anglų, rusų ir lietuvių k.).