

Performance analyses of embedded real-time operating systems using high-precision counters

Kemal Köker, Richard Membarth, Reinhard German
Department of Computer Science, Computer Networks and Communication Systems
University of Erlangen-Nuremberg, Germany
{koeker, german}@informatik.uni-erlangen.de, richard.membarth@informatik.stud.uni-erlangen.de

Abstract

To evaluate the performance of embedded real-time operating systems, we have built a scenario of soccer playing robots according to the F180 small-size league of the Robocup with a common embedded industrial PC/104 system. Thereafter we patched a common Linux kernel with the real-time application interface RTAI and installed it on a compact flash card for using it on the robots'. We connected IR distance sensors to the on-board embedded system and performed a response time analysis of the operating system, and implemented an interrupt service routine for the board's parallel port to generate a system response for externally caused hardware interrupts, e.g. from sensors. For a faster data collection we triggered interrupts by edges using a function signal generator. To monitor the embedded systems' response time we built a monitoring system by using a high-precision histogram scaler and counter. The data for the response time has been monitored in various system loads and been analysed statistically. Our system allows an easy and low-cost way for performance analyses of embedded real-time operating systems.

Keywords: embedded Linux systems, real-time operating system, performance analysis

1 Introduction

Nowadays the predictions about a systems response time of man-machine or machine-machine architectures are often based on simulation models. A lot of tools in this area require data for the input modelling. Often these data represent the response times for an input, at which the monitoring is done, e.g. by hardware. Analysing the performance of operating systems requires monitoring the response time for interrupts which are triggered by an input (e.g. pushing a button or getting new values of sensors). Mostly this response time depends on additionally running tasks in the system and the scheduling algorithm and varies with respect to the system load. Therefore increasing the system load increases e.g. the response time for a hardware interrupt.

Real-time operating systems guarantee a maximum time for the systems response for interrupt and represent a quite interesting possibility to be used in soccer robots. Smart scheduling algorithms and other techniques are used to keep the response time as low as possible. Mostly these techniques differ in case of a one-shot and periodic or multi-mode interrupt. In case of a one-shot interrupt, the operating system has to response as quickly as possible without any precautions for further interrupts. In case of a periodic mode, precautions for the scheduling are made to keep the response time lower. Considering the overhead for a one-shot mode, there should be a significant difference to the periodic-mode when the system has to response to higher interrupt frequencies. Therefore the mean of the response time in one-shot

mode should lead to a higher value than the mean of the periodic mode.

We selected a setting in our labour assembly to confirm the upper conclusion also for embedded real-time operating systems. To keep it as easy as possible, we patched a Linux kernel with the real-time application interface RTAI [11] and installed a complete embedded Linux operating system on a compact flash card to use it in a PC/104 system [1]. The PC/104 system is a common hardware in industrial projects. We used the PC/104 module from Arbor [2] with an AMD Ultra Low Power Geode GX1-300-Mhz fan less CPU and an onboard compact flash socket. This embedded board is common for standard PC-like hardware and its interfaces like serial, parallel, LAN, USB ports.

The rest of the paper is organized as follows.

Chapter two gives a brief overview of our embedded Linux system and the components used for it.

Chapter three gives an overview about the real time application interface for Linux and explains the architecture.

Chapter four describes very shortly the preparing for the performance analysis.

Chapter five explains the measuring using an oscilloscope.

Chapter six describes the measuring using the high precision counter.

Chapter seven gives the results of the performance analysis of the used embedded Linux system.

2 Embedded Linux System

The operating embedded Linux kernel is version 2.6.9 [8]. To keep the system small and truly (!) embedded, we used BusyBox [9]. BusyBox combines tiny versions of common Linux/UNIX utilities into a single small executable (including the Web server). For this reason, it's called a multi-call binary combining many common UNIX tools. The utilities in BusyBox generally have fewer options than their full-featured GNU originals; the options that are included provide the expected functionality and behave very much like their GNU counterparts.

For the systems C library we use uClibc [10] which is an optimized C library for developing embedded systems. It is much smaller than the common GNU C library, but nearly all applications supported by glibc work without problems with uClibc. One more benefit is that uClibc even supports shared libraries and threading. All robots allow remote logins to the running system via SSH connection. With the target to keep the system small, we used the SSH-2 client/server from Dropbear [14]. For further testing we included the Linux-based streaming system Palantir [15]. Palantir is designed to transmit live data (e.g. video, audio) over an existing TCP/IP network.

The system is stored on a compact flash card with the boot loader grub. Due to the limited write-cycles of a compact flash card, the system is running in ramdisk mode by loading a compressed initrd at boot time. Table 1 gives a brief overview of the storage size for our fully functionally embedded Linux system.

Comp.	Name	Uncompr.	Compressed
Grub loader	grub	-	200 KByte
Kernel 2.6.9	bzImage	-	1,100 KByte
Root FS	initrd	4,400 KByte	2,100 KByte
		Total	3,400 KByte

Table 1: Component size of embedded Linux

3 Real-Time Application Interface on PC/104

The well-known real-time application interface RTAI is an extension and modification of a common Linux kernel. For usage, the Linux kernel has to be patched with RTAI, which enables the operating system to response in a fast and predictable way. Fast means that it has low latency, thus i.e. it responds to external, asynchronous events in a short time. Predictable means in that case that it is able to determine task's completion time with certainty. The current architecture of an RTAI system consists of the Adeos Nanokernel layer [13], the hardware abstraction layer HAL and further modules for the requested functionality.

The Adeos Nanokernel layer is inserted as a division between the lower hardware and the operating system and has the task to transmit interrupts from the lower layer to the upper layer. Using Adeos enables running of different operating systems at the same time, in which each of the operating systems registers its interrupt handler to handle the interrupt. Each operating system is assigned to a domain, and each individual domain gets a certain priority. Depending on the priority level, Adeos manages the transmission of the interrupt to the corresponding operating system, where the higher priorities are provided first and the lower priorities get the interrupt signal at last. According to the architecture, Adeos is working at least with two domains and therefore there are at least two operating systems needed to make RTAI run.

The HAL again consists of an ARTI layer (Adeos-based real-time interface) and the real-time scheduler. The HAL enables an easy access of the OS to the interrupts via ARTI. The two OS domains mentioned above represent the real-time and a non-real-time domain. In this case, the real-time part is the RTAI domain with a higher priority, and the non-real-time part is the Linux kernel with a lower priority. Therefore, each interrupt for which both OS domains are registered as handler, is first send to the RTAI domain because of the higher priority. Figure 1 shows the schematic architecture of an RTAI system with the Linux domain as the non-real-time part.

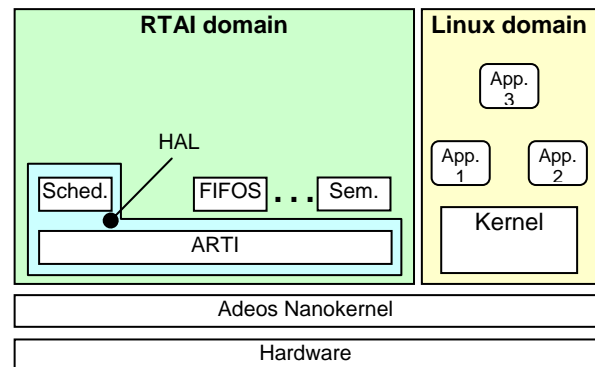


Figure 1: Schematic architecture of RTAI

4 Preparing for Performance Analyses

To evaluate the performance of a system's response time means e.g. to measure latencies for an interrupt, in which an interrupt may be generated either via software or hardware. In case of hardware, an interrupt e.g. on the parallel port is usually recognized by the system when a TTL level is changed (e.g. from 0 to +5 V) on the corresponding pin. If everything is set up correctly, the CPU is signalled to activate the interrupt handler for the corresponding port. Now the CPU stops the currently running code and jumps to the code of the interrupt handler which has to be executed und deletes the flag for the interrupt signal. After finishing the execution of the interrupt service

routine, the CPU executes the code before the interrupt was caused.

To conduct this recording we implemented an interrupt service routine ISR for the PC/104's parallel port, in which the implementation was done using RTAIs API [12]. Our ISR consists of three functions, called `xinit_module`, `handler` and `xcleanup_module`. The function `xinit_module` is called up automatically at module loading. It initializes the ISR and registers an interrupt handler for the interrupt 7 (parallel port). In case of an interrupt, a specific RTAI function is executed and a timer is started for the one-shot or periodic mode. The timer mode is depending on the mode the RTAI module was loaded at system booting. The handler function is the main function of the interrupt handler and generates just a short high signal and thereafter signals RTAI the end of service and readiness for the next interrupt. The end of the service is done by setting a low level on pin 2-9. The `xcleanup_module` is called up automatically at the time of unloading the module and cleans up the interrupt for the parallel port.

5 Monitoring the Latency Using an Oscilloscope

To verify the correct function of our ISR we performed some measurements by using a frequency generator and a digital sampling oscilloscope. To monitor the response, first the ISR has to be loaded.

We used the frequency generator GF266 from ELC [3] to generate rectangle signals at TTL level. The signal is connected to pin 10 of the parallel port and causes an interrupt. Each time a high level is on the pin, the ISR is called up and the corresponding response is served at pin 2-9. To monitor the response time for the generated signals, we connected the 4-channel, digital oscilloscope WaveSurfer 424 from LeCroy [4] to the frequency generator and to pin 4 of the parallel port. A high level generated by the frequency generator is connected parallel to channel one of the oscilloscope and on pin 10 of the parallel port (blue arrows). The response pin 4 is connected to channel 2 of the oscilloscope (pink arrows). Figure 2 displays the measurement infrastructure using the digital oscilloscope.

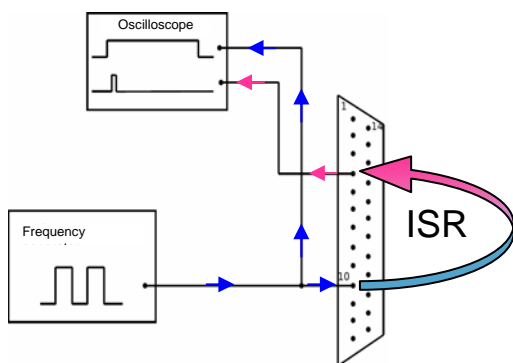


Figure 2: Monitoring via oscilloscope

The first run of the monitoring for the latency was done at 10-kHz rectangle signals of the frequency generator. The embedded RTAI Linux system was idle, so the response time for an interrupt was not influenced by any other factor than the internal task of the running operating system. Figure 3 displays a screenshot of the oscilloscope with a latency of 4.3 μ s for the response time at 10 kHz. The recording is truncated only for one signal from the frequency generator.

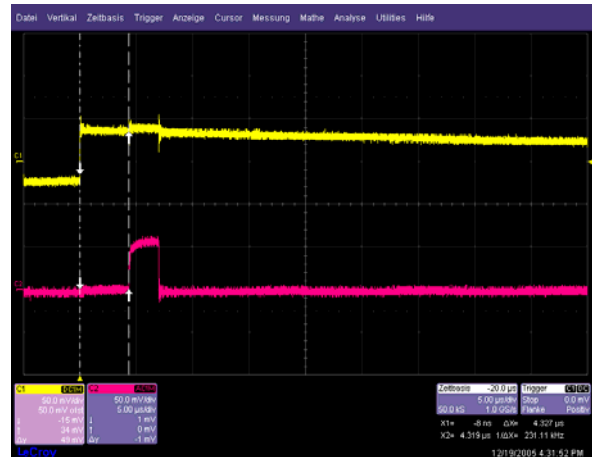


Figure 3: Screenshot of the oscilloscope – latency for response 4.3 μ s

Even though the system was idle, we monitored fluctuations for the response time. To measure and visualize this fluctuation, we used the cumulative sampling function of the oscilloscope. In this case, the sampling is recorded continuously and additionally overdrawn for the previous signal course. The number of observations of the values are now represented by colored pixels, in which a red color symbolizes a higher and the violet color for a lower appearance of the recorded value (Figure 4). Accounting the oscilloscope's recordings, we observed response times in the range of about 4–17 μ s when the system is idle.

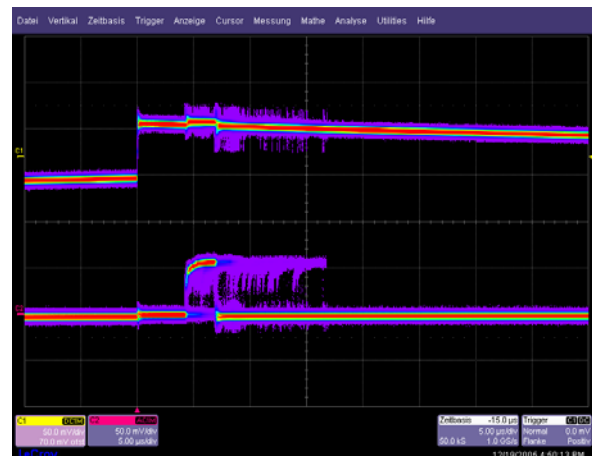


Figure 4: Cumulative recording for the response time – system: idle

We increased the frequency of the rectangle signal up to 100 kHz to make a simple check for the performance of the embedded system. The monitored data was almost equal to the first run and the increased interrupt frequency did not affect the response time.

To evaluate the system performance, the system load usually should be increased by executing parallel programs. We started a second run and executed just the ping flood to the embedded Linux system via a 100-MBit LAN connection and an interrupt frequency of 10 kHz. In this configuration, the signal course changed significantly using the cumulative recording of the system response (Figure 5). We monitored a response time in the range of 4 to 35 μ s and peeks up to 40 μ s. Even though the interrupt frequency and the system load increased, the response was “still in time”. Still in time means that we could relate each interrupt signal to its corresponding system response and vice versa. This bijectivity between interrupt and response is important for the evaluation and its statistical analyses, e.g. accounting for histograms.

We started a third run and raised the interrupt frequency slowly up to 100 kHz. The system was loaded again by an executing ping flood via LAN from an external computer. In this configuration (frequency 100 kHz) it was not possible to relate the interrupt signals to the corresponding responses. We monitored the beginning of the lagged responses at about 83.5 kHz and also some mavericks before.

All mentioned variations of the test run are not useful for statistical analyses and performance evaluation. However, our lab construction allows a first overview for the system performance of our embedded Linux system and externally caused interrupts. Therefore our lab construction allows a screenshot of the monitoring time. Detailed analyses of the performance require to record time-related data for our lab construction for a long time period. The recorded data should contain the difference of the interrupt and the corresponding response time for each generated interrupt signal.

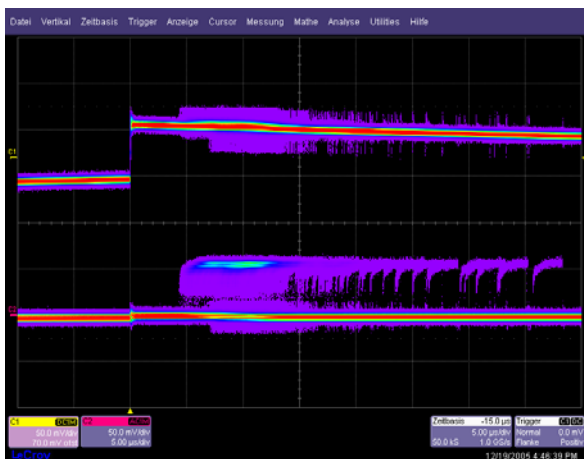


Figure 5: Signal course for the response at 10 kHz interrupt – system: load

6 MEASUREMENT INFRASTRUCTURE

We introduced a high-precision frequency counter SIS3820 [7] in the range of 50 ns to gather time-related data. The frequency counter has a start and a stop port to initiate and terminate the measuring. According to the monitoring via an oscilloscope, we connected the start port of the frequency counter with the output signal of the frequency generator and to pin 10 of the parallel port. The stop port of the counter is connected to one of the response pins of the parallel port (here it is pin 2). The counter uses an internal 50-MHz clock and therefore has a resolution of 20 ns. To measure the time between the start and stop signal, the counter is counting the ticks between the start and stop signal. Figure 6 displays the schematic for the measurement.

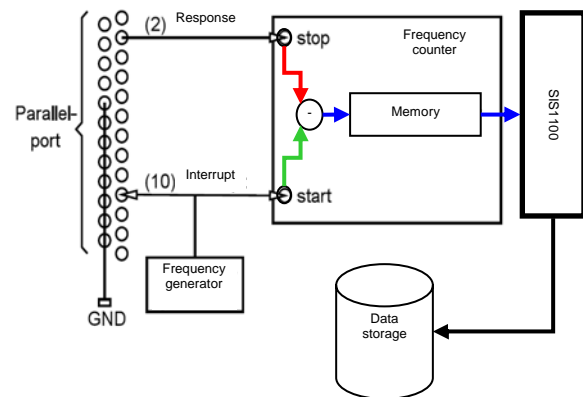


Figure 6: Schematic of the measurement via frequency counter

To perform statistical analyses we need to collect the pair of interrupt signal and time difference for that interrupt and its response. Reliable statements need a recording for a long period of time or huge amounts of interrupt signals coming from the frequency generator. The frequency counter allows to store up to more than 16 million values in a 64-MByte memory in FIFO manner. Therefore, we need a fast transmission of the collected data to an external storage like a second personal computer. The data transmission from the counter to the computer is periodically done via the SIS1100 module [6] over a fiber-distributed data interface.

7 Statistical Analyses of Example Measurements

We used our lab construction to run data collection for 3.6 million interrupts in four different test runs. The test duration lasts 6 minutes at an interrupt frequency of 10 kHz. The tests only differ for the real-time mode one-shot or periodic, and whether the system is idle or loaded. Usually, the system load is generated by different types like kernel compiling, ping flood, etc. To be able to compare the results, we again generated the system load by executing a ping

flood via LAN from an external computer and recorded the response times for the interrupts.

To evaluate the system performance based on statistical data, we took a closer look to the quartiles of the response times. The difference of the first and third quartile is defined as the distance of the quartile QD. This distance should cover 50% of all values and is used as the mass of the distribution. This mass is usually robust against runaways and is usable for statistical evaluations. Table 2 gives a brief overview of the collected data of the response time using the frequency counter.

RTAI	1. Quartile	2. Quartile	3. Quartile	QD
One-shot idle	4.20	4.30	4.40	0.2
Periodic idle	4.50	4.50	4.60	0.1
One-shot load	8.20	10.20	11.90	3.7
Periodic load	5.80	6.70	8.20	2.4

Table 2: Quartiles for the response time

Comparing the response times for the quartile distance delivers no significant difference between both modes in the case of idle system. We can see a very small difference of 0.1 μs where the one-shot mode is faster. Setting the system under load, the quartiles for the response time also increase. When comparing with the system in idle phase, we notice a significant difference for the quartile distance and a faster response time for the periodic mode. This difference may be due to the scheduler in the periodic mode, in which the scheduler is periodically executed. This means that the scheduler is prepared for further interrupts and the system can respond faster as in one-shot mode.

The results and the analyses of the quartiles are unique but do not contain detailed information for robust statements on the system performance.

Therefore we considered values like minimum, maximum, mean, jitter and standard deviation for the latency, listed in Table 3.

RTAI	Min	Max	Mean	Jitter	SD
One-shot idle	0.6	45.7	4.484	0.6318733	0.7949046
Periodic idle	0.7	43.7	4.742	0.7626195	0.8732809
One-shot load	3.7	51.6	10.24	11.53628	3.396510
Periodic load	1.4	38.3	7.586	7.754853	2.784754

Table 3: Jitter and standard deviation for the latency

The mean in the one-shot idle phase is 4.484 μs and nearly equals the mean of the periodic idle phase with 4.742 μs , and the jitter is very low between 0.631 μs and 0.762 μs . Increasing the system load almost doubles the latency in the periodic mode and more than doubles it in the one-shot mode. At the same time, the jitter increases by up to 11.536 μs for the one-shot mode and 7.754 μs for the periodic mode. Increasing the system load also increases the standard deviation SD for both modes. It is remarkable that the standard deviation for the system load in the one-shot mode amounts to more than 50% of the mean value in the idle phase. The corresponding histograms for the values in table 3 are shown in Figure 7 and are confirming our theory that the periodic scheduling mode is the better choice, independent of whether the systems status is idle or load. The y-axis of the figures is scaled logarithmically when the system is idle, and scaled linear in the load phase and is counting the number of interrupts. The x-axis is representing the latency for the system response for an interrupt. As already mentioned above, the difference between the one-shot and periodic mode in the idle phase of the system is marginal and the result is shown in Figure 8 with more detailed histograms for the range of 0-16 μs of latency time. On the other hand, when the system load increases, the histogram of the distribution for the latency in the periodic mode is slimmer in comparison to the one-shot mode.

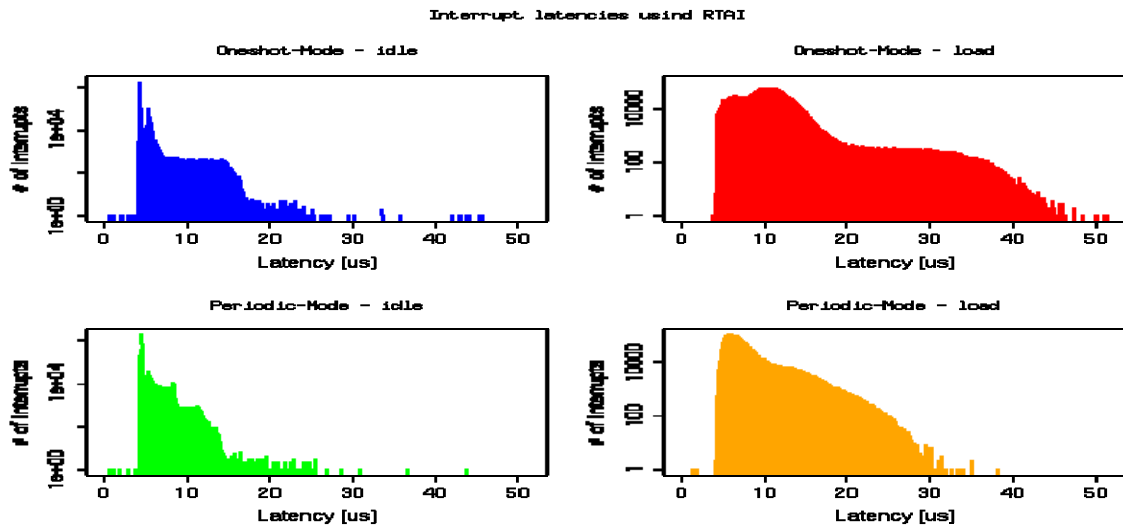


Figure 7: Histograms for the one-shot and periodic mode

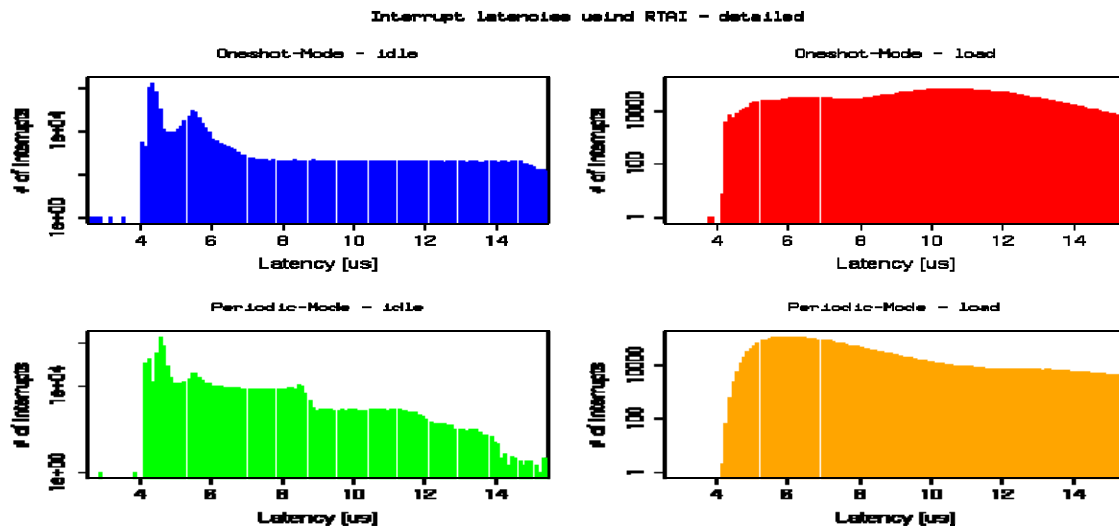


Figure 8: More detailed histograms for the one-shot and periodic mode – latency time 0-16 μ s

8 Conclusion

We patched a Linux kernel with a real-time extension to evaluate the performance of an embedded real-time operating system on an embedded industrial PC/104 board.

Thereafter we monitored the system performance for hardware interrupts using a frequency generator connected to the parallel port of the PC/104 board. To enable a response for the interrupt, we implemented a service routine for the parallel port using the real-time API. We observed the interrupt signal and the response at the same time on the oscilloscope and found out that our embedded system is fast enough to respond in time for interrupts of up to 100 kHz when it is idle. Loading the system increased the response time and disabled it for interrupts higher 83.5 kHz.

We introduced a high-precision counter to gather time-related data for the latency. The analysis of the latency in different running modes confirmed our theory that the periodic mode in a real-time system gets a better performance nearly every time. Our measurement infrastructure can be easily adapted to evaluate other embedded real-time systems.

9 References

- [1] PC/104 EMBEDDED CONSORTIUM, *PC/104-Spezifikation v2.5*, “Home Page”, <http://www.pc104.org>, visited on 12/06/2004
- [2] ARBOR INC
PC/104-module Em104-n513/VL, “HomePage”, <http://www.arbor.com.tw>, visited on 12/12/2005
- [3] ELC,
ELC Frequency generator GF 2006, “Home Page”, <http://elc.annecy.free.fr>, visited on 15/12/2005
- [4] LECROY CORPORATION,
LeCroy WaveSurfer 424, “Home Page” <http://www.lecroy.com>, visited on 23/11/2005
- [5] PHILLIPS SCIENTIFIC,
NIM LOGIC UNIT 755, “Home Page”, <http://www.phillipsscscientific.com>, visited on 9/12/2005
- [6] STRUCK INNOVATIVE SYSTEME GMBH,
SIS1100/3100-PCI/VME link/interface, “Home Page”, <http://www.struck.de>, visited on 11/12/2005
- [7] STRUCK INNOVATIVE SYSTEME GMBH,
SIS3820 multi purpose scaler, “Home Page”, <http://www.struck.de/sis3820.htm>, visited on 11/12/2005
- [8] LINUX Kernel Archives, “Home Page”, <http://www.kernel.org/>, visited on 11/11/2005
- [9] Erik Andersen, *BusyBox: The Swiss Army Knife of Embedded Linux*, “Home Page” <http://busybox.net/>, visited on 15/7/2005
- [10] Erik Andersen, *uClibc: A C library for embedded Linux*, “Home Page”, <http://www.uclibc.org/>, visited on 15/7/2005
- [11] RTAI PROJECT,
RTAI – Real-Time Application Interface, “Home Page”, <http://www.rtai.org>, visited on 4/10/2005
- [12] RTAI PROJECT, *RTAI API*, “Home Page”, <http://www.rtai.org/documentation/vesuvio/html/api/>, visited on 4/10/2005
- [13] Various Authors, *Adaptive Domain Environment for Operating Systems*, “Home Page”, <http://home.gna.org/adeos/>, visited on 4/10/2005
- [14] Dropbear, Small SSH2 Client, “Home Page”, <http://matt.ucc.asn.au/dropbear/dropbear.html>, visited on 15/7/2005
- [15] Palantir, streaming system, “Home Page”, <http://www.fastpath.it/products/palantir>, visited on 15/7/2005
- [16] Hannes Mayer, *Programmbeispiele für RTAI – Adeos – Linux*, “Home Page”, <http://www.captain.at/rtai-adeos-linux.php>, visited on 05/10/2005