Experiences with StrongARM®/Linux/RTAI combination in Mission-Critical Systems

Iztok Kobal
Iskra SISTEMI, d.d.
Stegne 21, 1000 Ljubljana, Slovenia
iztok.kobal@iskrasistemi.si
http://www.iskrasistemi.si

Davor Munda
Kopica, Davor Munda s.p.
Štihova 6, 1000 Ljubljana, Slovenia
davor.munda@kopica-sp.si
http://www.kopica-sp.si

June 06th, 2006

Abstract

Currently can combination of Linux and RTAI be run on various platforms. Yet the decision to choose one of them relies on many factors of which the stability of the microprocessor in various conditions is one but not the last of them, obviously. In this paper we want to present the development project where at start the combination Intel StrongARM/Linux/RTAI seemed very reliable and promising, at least by Google. However, as the project progresses towards the end of development cycle many unexpected and nearly unsolvable problems appeared.

1 Introduction

Protection relays in electrical power plants and electrical power Substation Automation Systems are there to prevent costly damages to the so-called primary equipment as are transformers and switchgears in case of faults within the electrical grid or *vice versa* - damages of the grid components in case of faults on primary equipment. The protection algorithms cover all kinds of possible misbehaviors in electrical systems as are over/undervoltage, over/undercurrent and various short circuit faults and irregularities as well as some more sophisticated busbar and distant protections. The mathematics behind the action becomes more and more sophisticated since both market and standards follow the progress of the microelectronics and microprocessor power. Producers are forced to meet complexity and speed which in many times are mutual exclusive, especially when the speed is also mixed with hard real-time requirements.

In the past the software for all digital protection devices has been designed in-house. No general or multipurpose operating systems were available for such devices due to their limited resource pool (EEPROM, FLASH, RAM sizes, etc.). Today, at least combinations of Linux and some hard real-time extensions (RTLinux, RTAI, ...) appear to be useful also in mission-critical environments due to the Linux stability and scalability and also serious and professional approach of the development teams of hard real-time extension packages which lately deliver stability and optimal performance. We have decided to use Linux with RTAI extension to meet real-time requirements.

We have ported Linux kernel 2.4.18 and RTAI 24.1.10 to CEP platform with SA-1110 206 MHz processor, 64 MB SDRAM and 32 MB Flash memory. Our external interrupt source is Ethernet 802.3/802.3u 10/100 connected to GPIO01 and we

use on-chip peripheral devices that are also interrupt sources: Serial Port1, Serial Port3, Operating System Timer0, OS Timer2 and OS Timer3. We use RTAI periodic timer with 1 ms period as base period for real-time tasks.

Yet, the path to successful integration of Linux + real-time extension into such system introduces some challenges which we would like to present in this article. Among others we will explain our usage of RTAI Watchdog task, the problem with lost OS Timer interrupt, the problem with current time *timeval* value, the problem with restoring interrupts on StrongARM architecture and some kernel patches to prevent user space applications crashes.

2 Why have we chosen the StrongARM/Linux/RTAI at all ?

Well, we have not been after particular hardware at first. At that time we wanted to build us software pool which could be highly reusable and scalable both engineering-wise as well as in the future development cycles. We wanted object-based elements which could offer our sales and engineering teams more than just compact devices with narrow predetermined roles in the system. We wanted to develop on PCs. We wanted all this because the effort to rewrite software every time we had to change hardware platform was the payload so high for us that we hardly afforded it.

Another reason to take a general purpose operating system were market demands which obviously headed towards extensive network exploitation both with the process data sharing as well as with the file system sharing. We have already identified big potentials of (S)FTP, HTTP etc. in Substation Automation Systems.

Another problem was that we already had some important and stable code parts written in Modula-2 and we also wanted to use various proven programming languages and methods - whatever would be at disposal and satisfy the particular task needs and/or software engineer. So we ended up with the shell scripting and also Modula-3, not mentioning of course C/C++ languages and XML technologies.

So we have decided to choose among systems which already have offered these functionalities or at least were showing to potentially cover them in the future. At our disposal were QNX, VxWorks, Windows CE and Linux. First two choices were basically too expensive for us. Windows CE had no possibility for reliable hard real-time at that time. And none of them except for Linux have covered all medium performance hardware platforms as were ARMs and low end PowerPCs. And none of them except for Linux offered us to develop & test on PC as target with all technologies listed above and only later on to recompile the software for final target hardware platform (or more of them) without portability pains. At least we thought so at that time ...

One way or another, we learned that the decision to choose Linux paid-off very soon. We are currently using services, clients and other pieces of software as are Apache, NTP, PCRE, OpenSSH with SSH(D) and SFTP(D), logrotate, cron and for sure we have forgotten some pieces which are just too natural to be used to pay attention on.

The decision to use RTAI was based on idea that we would like to develop real-time part of functionalities in user space and test them in hard real-time only after being debugged from programmer-level errors. RTAI seemed ideal for that since it had LXRT, user space soft real-time system, with nearly the same API as was for hard real-time system. The only difference for us, as a matter of fact, was only with the task start procedure. We just did not expect to use some special features of RTAI which did not appear in LXRT.

And, finally, why have we chosen Intel™ StrongARM. We have evaluated more CPU boards. Our requirements were the CPU, RAM, FLASH, Ethernet, at least 2 serial ports and expansion port. Finally we have chosen CEP module (IskraTEL Electronics, Slovenia, http://www.iskratel.si). The low power consumption, 235MIPS, Slovenian producer and relative low price prevailed against other boards.

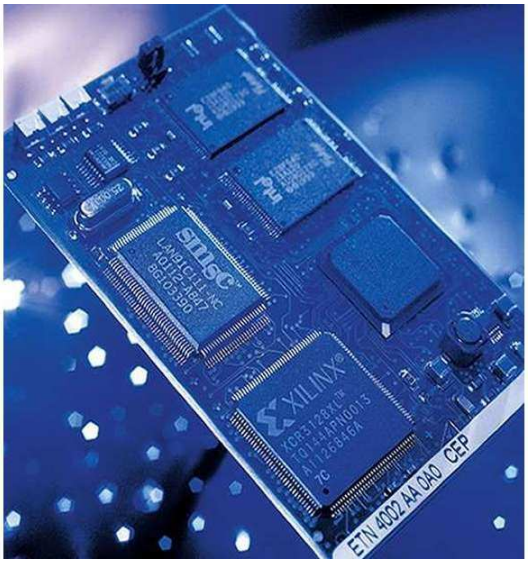

FIGURE 1: *IskraTEL CEP1000 CPU board*

StrongARM was only the first of the microprocessors that we intended to use. We have built the system which allowed us very high state of independency from the hardware platform and very short time to port the software as long as we are able to stick with Linux.

3 What software do we run ?

There are really three segments of software being run on the target:

- functional part on top of the RTAI
- data communication part
- system supervising part

Functional part

Functional part covers all autonomous protection functions. They have to be prompt and fast. Algorithms are mainly based on simple discrete differential equations so the timings should be accurate to get good results. We would like to rock'n'roll on millisecond rhythm while cover two or more protection bays which is rather courageously since nobody does this yet, as far as we know. We optimized the system by grouping the tasks on the cycle time and mutual dependency basis, thus diminishing the CPU load caused by context switching.

Some would say that the 206MHz 235MIPS microprocessor should easily cope with such load unless the amount of data was presented to them. Total amount of data needed to do the job correctly with the single 3-phase switchgear protection field is more than 500 words, most of them harmonics. Large number of measurements must be transferred from the input peripheral modules each and every millisecond while majority of them on slightly longer cycle. Still, this presents big effort to CPU and this is why we are using co-processor for data preprocessing as are fast Fourier transforms and digital debounce filters.

Data communication part

This software segment is responsible for forwarding the picture of controlled process to the supervising control level, often called SCADA level. Communication protocols in Substation Automation Systems are object oriented and very complex. Lately, the link layer should mostly be implemented as network service (DNP and IEC60870 standards). Moreover, upcoming IEC61850 standard introduces unseen data communication complexity by implementing encapsulated data services. In one of its parts even tends to replace the hard backplane data bus by distributing small amount of data over the network in real-time and will have to be implemented also within the above described functional part.

Obviously, C programming language is not enough for implementing such protocols anymore. We are using C++ and Modula-3 object languages instead.

System supervising part

For this part we are using two services:

- HTTP run by stripped-down Apache serving HTML and CGI
- proprietary service called NEO3000 SPM iServer Communication (N3KSC)

Having failed to use the XSLT technology to implement both on-line and off-line supervising, parameterizing and project management tool, we had to exploit Java on client side while on server side retained heavy XML/XSD/DOM utilization. Powerful distributed file system, parameterization, license and package management are implemented to satisfy high demands of off-line project management while on-line every single function can be controlled and every piece of diagnostics or controlled process state could be traced.

4 Problems with RTAI

We had to present the complexity of the co-operating functionality to enlighten the first problem we were confronted with at very start of RTAI usage in connection with following demands for modern protection devices:

- five nines (99.999%) availability class, six nines (99.9999%) in near future
- all-time possibility for remote device supervision
- extensive self diagnosis and auto recovery

What does this mean for the system where the RT-extender like RTAI rules in the system? Well, when one of real-time tasks gets loose we even do not get the blue screen of death, rather nothing. The system becomes not responsive. Because of the five nines rule we can not afford even restart caused by hardware watch-dog since the long Linux reboot time throws us out of the standard (it allows us 315 seconds of unavailability per year) with, let say, only three restarts caused by system or application fault. As for the near future, six nines mean 31 seconds of unavailability per year and as a consequence only one fault per 2 years is allowed.

Unlike with the industry class controllers, process dynamics in Substation Automation Systems is equal to zero most of the time. But when something bad happens everything goes crazy and devices must swallow it and react properly. We do have simulators for such cases but we really can not test the SAS for all possible combinations of process states. What we are afraid of is not that one of functions would stall. Such situation should also be considered but it is not very probable considering all those type-tests that we do before really starting to put on market the certain functional combination. It is already enough that one of functions consumes slightly too much time. In such case another function does not do its task on time or the mathematics relying on the strict cycle times starts to give invalid results. And more, in case of coding error and one of tasks stalls or goes endless loop, user space processes are not functional anymore.

It certainly helps that by standard the functions like e.g. overcurrent protection are always doubled with slightly different parameters. One checks simple effective current limits, another checks 1^{st} or higher differential, yet another checks harmonic distortions, each protection stage with slightly different reaction delay. We can be sure that even if one of functionalities misfired another would do it right. *But* only if it has its chance.

4.1 RTAI Watchdog Task Usage

While testing RTAI performances, we found out that UP Scheduler had a built-in capability to compensate for lost ticks of periodic tasks. To comply with application requirements we needed to control spent CPU time per each real-time task and remainder CPU time for Idle Linux task and, of course, lost task ticks.

First we patched RTAI UP Scheduler to contain spent CPU time within *rt_task_struct* structure. The following structure to the *rt_task_struct* structure was added:

```
typedef struct {
  RTIME  sum_uptime;   /* uP time summa */
  RTIME  sched_intime; /* sched in time */
  RTIME  max_uptime;   /* maximum uP sched time */
} RTWD_TINFO;
```

Structure members represent:
sum_uptime : spent CPU time summa
sched_intime : sched in time mark
max_uptime : max time spent within one tick.

In *rtai_sched.c* source code appropriate values to the added structure were assigned. In *init_module* and *rt_task_init* functions values were initialized, and in switch task the time spent was calculated.

Next RTAI Watchdog was patched to check CPU spent time for real-time tasks and for Idle Linux task. The data to the user space application was sent via RTFIFO device.
We started Watchdog as the highest priority task with RTAI periodic timer period (1 ms).
Watchdog policy was to suspend all real-time tasks if any task missed its period, or in case Idle Linux task used less than 20% processor time in the last second. Idle Linux task and Watchdog task stayed in operation.
User space application controlling Watchdog via RTFIFO got all the required data to start appropriate recovery action in case of tasks suspension.

Of course, we can not prevent the system crash in case of segmentation fault in kernel space unless kernel itself starts to support some kind of exceptioning model.
Nevertheless, we have achieved total control over the overrunning tasks which allows us proper diagnostics

and remote device access with manual or automatic recovery. Idle Linux task reached enough CPU time to cover non real-time application requests. And last but not least, task accounting provided important information to the engineers in the system tuning phase.

4.2 Lost Operating System Timer Interrupt

With an intervention into RTAI UP Scheduler code the switch task OS Timer0 latency was increased.
OS Timer0 interrupt is used on StrongARM architecture as system interrupt when scheduler deadline is reached.
RTAI protects StrongARM OS Timer0 with the following code:

```
if ( delay )
  next_match = OSMR0 = delay + OSCR;
else
  next_match = (OSMR0 += rt_times.periodic_tick);

while ((int)(next_match - OSCR) < LATENCY_TICKS){
  next_match = OSMR0 = OSCR + LATENCY_TICKS;
}
```

Where OSCR represents System Timer free running counter with frequency 3686400 Hz, OSMR0 represents OS Timer0 Match Register and LATENCY_TICKS is calculated Timer ticks to cover 2600 ns latency. Because latency was increased, we missed the next Timer0 match when short delay or period had been set. So we lost OS Timer0 interrupts for the whole OSCR cycle (1165 sec). This problem was solved by patching the code with the following intervention:

```
addon = 3; /* register write latency */
while ((int)(next_match - OSCR) < LATENCY_TICKS){
  OSSR = OSSR_M0;  /* Clear match on timer0 */
  next_match = OSMR0 = OSCR+LATENCY_TICKS+addon;
  addon++;
}
```

It is noteworthy that a similar solution was used in later RTAI releases.

However, in the next step of the evaluation process we noticed that OS Timer0 Interrupt could still be lost. At this point we realized that we could protect OS Timer0 Interrupt with another OS Timer Interrupt. As StrongARM has four OS Timer Interrupts implemented, we decided to use OS Timer2 Interrupt, and to use it in the following manner: RTAI UP Scheduler was patched again and the code to the *rt_timer_handler* ISR routine was added:

```
#ifdef __arm__
 hard_save_flags_and_cli(flags);
 if ( oneshot_timer ) {
   /*2 ms*/
   OSMR2 = (nano2count(1000000) << 1) + OSCR;
 }
 else {
   /*2 periods*/
   OSMR2 = (rt_times.periodic_tick<<1)+OSCR;
 }
 hard_restore_flags(flags);
#endif
```

And we implemented OS Timer2 ISR routine:

```
#ifdef __arm__
 static void buggy_timer_wakeup_isr(
   int irq,
   void *dev_id,
   struct pt_regs *regs)
 {
   unsigned long flags;

   hard_save_flags_and_cli(flags);
   OSSR = OSSR_M2;  /* Clear match on timer2 */
   hard_restore_flags(flags);

   /* Timer0_interrupt did not fire */
   rt_timer_handler();

   return;
 }
#endif /* __arm__ */
```

Naturally, the requested initialization was also implemented.
This intervention ensured the recovery of the lost OS Timer0 interrupt after one missed periodic tick. This is not what we would like to see happening. We tolerate it since it happens only when bigger number of tasks is being started nearly simultaneously which normally is the case only at system start.

4.3 Problem with Current Time

The next problem that we encountered concerns the buggy current time stamps. Linux's functions *do_gettimeofday / gettimeofday* are intended for current time access. Interface API offers microseconds accuracy. We noticed that our real-time time stamps for events did not record the correct sequence: Later events had earlier time stamps than earlier events.
Many Linux system features also use that kind of time stamps (e.g. device and network drivers).

Linux *do_gettimeofday* function calls *gettimeoffset* function whose implementation presupposes 10 ms OS Timer0 tick latch. Function *sa1100_gettimeoffset* implementation contains the code:

```
ticks_to_match = OSMR0 - OSCR;
elapsed = LATCH - ticks_to_match;
usec = (unsigned long)(elapsed*tick)/LATCH;
```

Since RTAI Scheduler sets OSMR0 to match real-time period, this implementation gives wrong results, which are fixed every 10 ms through system jiffies update. So instead of microsecond accuracy, we got 10 ms accuracy.

Here we implemented RTAI tick and RTAI tick counter to fix elapsed ticks. Again, we patched RTAI UP Scheduler to set the proper tick value and to count elapsed ticks. And we patched kernel timer functionality to properly set and update system timer *xtime* value (Linux function *update_wall_time_one_tick*). We fixed Linux function *sa1100_gettimeoffset* with the following code:

```
ticks_to_match = OSMR0 - OSCR;
elapsed = rtai_latch - ticks_to_match;
usec = (unsigned long)
       (elapsed*rtai_tick)/rtai_latch;
/* Fix usec */
usec += rtai_tick_cnt * rtai_tick;
```

In this way we managed to get correct time stamps and to fix the Linux system current time value.

4.4 Restoring Interrupts Problem

In spite of our efforts to build a reliable and available real-time system we would still come across a situation with all disabled interrupts (OS Timer interrupts were also lost), which meant losing our target device. We noticed that such a situation could take place when real-time tasks were active and we got heavy traffic on interrupt sources (e.g. serial and ethernet devices).

Because StrongARM does not support any NMI (except Watchdog Timer), we didn't have any tool to recover from such a situation without reboot device. At this point we implemented StrongARM Watchdog Timer driver. On StrongARM we could use OS Timer3 as a watchdog. After a proper initialization process, OS Timer3 served as a non maskable watchdog timer.
After time out deadline was reached, reset was applied, and boot process started.

Admittedly, such a reboot process breaks device availability. Our boot process - with application start included - takes about 3 minutes. Which is why we were still looking for a solution to avoid unrestorable disabled interrupts.

Linux *cli* and *sti* functionality uses the *MSR CPSR_c* instruction. It is known that the StrongARM processor has a bug and the first instruction following *MSR CPSR_c* is executed twice.

We patched kernel arm assembler files in manner to adding NOP (*mov r0, r0*) instructions after *MSR CPSR_c* instructions.

In the Intel™ StrongARM Developer's Manual [**?**] we found that the interrupt-enabling write to the CPSR must be separated from the interrupt disabling write to the CPSR by at least two instructions.

So we patched kernel *include/asm-arm/proc-armv/hard_system.h* file (here *cli* and *sti* functionality for StrongARM is implemented) with adding two NOPs after CPSR write instructions.

At this point we got stable kernel and RTAI release for real-time tasks.

But when we added another serial port device (interrupt source) with intensive traffic to the system, we got a situation with all disabled interrupts again.

Because we were sure that we correctly patched kernel for the StrongARM architecture, we started to examine RTAI dispatch interrupts routine.

In the RTAI 3.1 release we found Thomas Gleixner patches for StrongARM that fixed race conditions with IRQ/SRQ handling. We implemented the same solution in the RTAI 24.1.10 release. We patched RTAI core file *arch/arm/rtai.c* and we finaly got completely stable Linux real-time release.

5 Some other annoying problems which we have met during development and are not related to RTAI

5.1 ARM 32-bit alignment

Luckily, we have found that article in *DrDobb's Journal* which dealt with some issues connected to the StrongARM development and code portability. The strict 32-bit structure field alignment and 32-bit

alignment of larger-than-8-bit casts of pointer targets was just the most painfull. It only showed that we had became negligent in the past when we had mostly targeted the ix86 platforms which had not such restriction.

At the end of the day we had to search for each and every pointer dereference and check it on possible non 32-bit alignment. It resulted to extensive use of accessor/mutator functions and hiding the structure and object declarations whenever we could not avoid non 32-bit alignment of objects with size larger than single byte. This especially referred to the interpretations of the communication protocols.

5.2 GNU/G++'s -O3 and exceptions bugs

We are using gcc-2.95.3. The bugs are really not related to the GCC version since the stuff works properly for ix86 platforms. But what the bugs really mean for the running system?

-O3 introduces some time optimizations, like inlining, which we obviously can not use. It really means that the resulting code can never be as time optimal as it could be.
Even with -O2 optimization flag set, we found *memcpy* bug for the StrongARM architecture (move word to the odd address), so we had to take care of *memcpy* usage.

Exceptions basically work. But only if thrower and catcher are in the same executable. This refers, of course, also to shared objects. You can imagine that exceptions are out of question in the system like ours which is based on shared objects. Some would say that exceptions should not be used anyway. Well, depends on the point of view. Exceptions really help with auto-recovery issues if properly used. Not mentioning that in many times the source code becomes more readable if exceptions are used.

5.3 Linux 2.4.x kernel IPC semaphore bug

During user space application test process we found IPC bug that caused application crash. In Linux file *ipc/util.h* routine *ipc_checkid* is implemented with the following code:

```
if(uid/SEQ_MULTIPLIER != ipcp->seq)
    return 1;
return 0;
```

Where *ipcp* pointer can be corrupted. This results in application crash.

We patched this code with *ipcp* pointer check:

```
if ((unsigned long)ipcp < PAGE_SIZE) return 1;
if(uid/SEQ_MULTIPLIER != ipcp->seq)
    return 1;
return 0;
```

Our tests show that we have got stable Linux real-time environment for real-time tasks and for user space applications. To achieve this goal, we also had to implement some other Linux kernel patches (e.g. ethernet driver, file system driver, etc.).

6 Conclusion

Our experiences with RTAI Linux extension on StrongARM processor show that Linux with GNU Toolchain is a scalable, robust and portable system, convenient for embedded platforms. RTAI extension adds hard real-time functionality to the system. With Linux and RTAI we can cover hard real-time application requirements.

Open source policy enables interventions into the code to solve problems which are met in the development cycle. This is a big benefit, and it supports the decision to implement Linux and RTAI on embedded platforms.

Our experiences also show that developers can encounter many unexpected system problems in the development cycle. To solve the system problems some system knowledge is required. Without this knowledge Linux RTAI application implementation is likely to turn into quite a painful task.

References

[1] Allesandro Rubini, 1998, *Linux Device Drivers*, O'Reilly, ISBN: 1-56592-292-1.

[2] Craig Hollabaugh, 2002, *Embedded Linux*, Pearson Education, ISBN: 0672322269.

[3] Intel, 2001, *Intel StrongARM SA-1110 Microprocessor, Developer's Manual.*

[4] AlephOne, 2001, *ARMLinux for Developers.*

[5] DIAPM-RTAI, 2000, *A Hard Real Time support for LINUX, Manual.*

[6] IskraTEL, 2002, *CEP, Hardware version 1.0, Manual.*

Definitions, Abbreviations

Apache HTTP server (http://www.apache.org)

API Application Program Interface

CPU Central Processing Unit

EEPROM, FLASH Electrically Erasable and Programmable Read-Only Memory

HTTP HyperText Transfer Protocol

LXRT RTAI user space soft hard real-time system for Linux

NTP Network Time Protocol

PCRE Perl Compatible Regular Expressions (http://www.pcre.org)

RAM Random Access Memory

RT Real Time

RTAI Real Time Application Interface (http://www.rtai.org)

SAS Substation Automation System

SFTP Secure File Transfer Protocol

SSH Secure Shell (http://www.openssh.com)

StrongARM Member of Intel™ ARMv4 family of microprocessors