Exploiting the hard realtime features of PROFINET by a real-time capable Linux

Karl-Heinz Krause
Siemens Automation and Drives
Advanced Technologies and Standards
A&D ATS 1, Gleiwitzer Str. 555
90475 Nrnberg, Germany
Karl-Heinz.Krause@siemens.com

Abstract

A PROFINET system built out of ERTEC Switch ASICs allows to keep the nodes of a network synchronized with an accuracy below 1sec. While still allowing regular IP communication, it also guarantees the timely arrival of cyclic messages below that mark. These features are particularly required for the motion control part of automation systems.

The talk starts with browsing through the essentials of PROFINET and the ERTEC ASIC which provide the real time capabilities. When connecting such an Ethernet controller as the ERTEC to a host processor, the hardware interfaces also have a significant impact on the real time behavior. These issues are discussed. Then all the measures for providing the best possible real time response for a Linux user level program are presented. This includes the basic features of the chosen two-kernel approach as well as the functional enhancements necessary for meeting the specific requirements of drive control applications. Finally, the achieved performance data are presented.

1 Introduction

How can we apply the general definition of hard real time to communication over potentially unreliable communication lines? Is it possible to guarantee the arrival of a message within a certain time span or more precisely since we are talking about *hard* real time as required for automation systems within a time span of a few microseconds? The answer is simply no we cannot. If a message is guaranteed to arrive, then we cannot make any time guarantees. For a guaranteed time span of arrival, we cannot guarantee that a message arrives at all. That is the basic fact we have to live with and the ERTEC ASIC acknowledge this. The arrival of a real time message gets monitored, if it hasn't arrived within the expected time span it can get signaled, so that countermeasures can be taken. Such a mechanism is not only due to unreliable communication lines, it is also necessary in a distributed system in general where a node must also continue operation when cooperating nodes fail.

2 ERTEC Real Time Communication

As a consequence of the above fact, two types of communication are supported by the ERTEC switch ASIC. According to the cyclic model of operation for (digital) drive control loops the time axis of a PROFINET communication line is also partitioned into cycles. One phase of a cycle is reserved for special cyclic communication, the remaining part is free for regular (sporadic) communication and it is one of the very fundamental features of an ERTEC ASIC, that it guarantees that frames of regular communication can *never* penetrate into the phase for cyclic communication. Thus for cyclic communication for each cycle a certain data volume is guaranteed to be transferable. It should be noted that this model works only for full duplex communication lines. The cyclic communication itself has four important features:

- It is planned communication. That means that all of the frames fit into the phase and all the ERTEC-switches involved into the communica-

tion transactions guarantee that the frames are propagated within that phase or more precisely exactly at predefined points in time within that phase. .

- Received frames are not processed individually by a communication stack. The payload of a received frame gets written directly to a predefined location as specified by its frame ID. The frame of a new cycle simply overwrites the data of the previous cycle. Additional measures o provide for the discovery of missing frames. Here the user can specify the number of subsequent missing frames he is willing to tolerate. o allow for redundant transfer. A frame may get transferred redundantly over different communication lines. The update of the location at the destination is only done once per cycle.

- When all of the expected cyclic frames *should* have arrived, a specific interrupt event is set and the CPU is notified, bypassing all the communication stack based processing of regular frames. For the ARM CPU built into the ERTEC the interrupt controller can be configured such that this interrupt event can be signaled completely independent of all other communication events, providing the best possible HW-foundation for hard real time response. For a host CPU connected via PCI to the ERTEC, the independence is somewhat limited by the shared interrupt line.

- An application program processing the data of such a frame or a set of frames may work either synchronously or asynchronously to the communication cycle. For drive control applications the former one is of interest. Here supervision for timeliness is the important feature, meaning that an application needs to signal to the ERTEC that all incoming frames are processed and the data for all outgoing frames are ready. If this is not done in time, an interrupt will be generated and the outgoing cyclic frames are marked as invalid.

The functions of PROFINET communication which use the other phase of a cycle are not of interest for this talk and therefore will not be discussed. Also not touched here is – although it might be of interest – how the sub microsecond accuracy of time synchrony over a large number of switch hops is achieved.

3 Requirements of Drive Control Applications

Drive control applications employ a cyclic model, with nested periods and phase shifts over a wide range of periods. The shortest period length is usually 31,25 sec. Translated into requirements for an operating system this means that an operating system has to provide for the very efficient repetition of timer triggered activation patterns. A repetition which needs to be synchronized with some external device period, e.g. a 1 millisecond PROFINET communication cycle provided by the ERTEC switch ASIC.

As the figures of 31,25 sec minimal period length and 1 sec time accuracy indicate, this application area has the strongest performance and jitter requirements. This does not necessarily directly translate into mandatory requirements for a general purpose computer system. A very specialized subsystem may deal with the lowest 31,25 sec cycle, latches controlled by ERTEC hardware signals may provide for the 1 sec accuracy. But the more we can achieve with cost effective standard hardware the better. Besides cost effectiveness the requirement for such embedded systems to operate without forced cooling often doesnt allow to use processors with highest performance. Because of that the call for best possible exploitation of a given hardware by the operating system stays a permanent quest.

Finally as a last requirement when talking about implementation the word "application" has to be taken literally in the sense of application mode. The developers of application packages don not want to deal with low level driver and kernel development, they want to develop at the regular user level or more precisely write regular application programs based on the glibc. This leads automatically to the evaluation of the API, what parts of the API need to be used and therefore are critical with respect to performance and jitter.

4 The API

The POSIX-API as defined today and provided by the Linux kernel through the glibc accommodates various programming models. Therefore it provides for a lot of partially redundant functions and – although some parts of the POSIX API are labeled as "realtime" – is definitely not designed with real time as its focus. Fortunately real time applications, in particular the time critical part of them, need only very few functions. So when picking the needed subset, some performance/realtime biased selection can be done. Because of the time based activation pat-

terns mentioned above, the API for the notification of a user program when a timer fires is most critical for best possible real time response. Unfortunately neither the POSIX definition nor the glibc implementation are suited for that. The good news is, it can be fixed easily. The recipe in short is:

- Do not use the process wide signalling (`SIGEV_SIGNAL`) and forget about signal handlers for asynchronous events. Both together constitute one of the most indeterministic constructs of the POSIX-API.
- Instead of this, make available at the API what the Linux kernel provides, sending a queueable signal to a specified thread (`SIGEV_THREAD_ID`). All that this needs is an additional glibc function which we called `sigevent_set_notification()` which parameterizes the sigevent object accordingly[1].

Having done this, the programmer works with the regular POSIX API meaning he creates a timer with a `timer_create()` call, arms it with `timer_settime()` and waits for it to fire with one of the sigwait calls (`sigwait()`, `sigwaitinfo()`, `sigtimedwait()`).

With this cure we have not only picked the fastest and most deterministic solution available in the Linux domain; as shown later, further improvements are possible when implementing only this subset of signalling.

What about the repetition of activation patterns? Setting up a one shot timer for every action again and again is too expensive. On the other side we cannot simply set up periodic timers since all of the actions need to be resynchronized at the beginning of each external period. The best way to do this turned out to introduce a new clock type. POSIX allows for having additional clocks but doesnt specifiy how to bring it in. So an additional API call

```
clockid_t register_clock(
    int fildes, int type,
    struct itimerspec *clock_spec);
```

was added. It gets mapped to an `ioctl()` and allows to add a new clock type `CLOCK_SYNC`, where the clock event of a device indicates the beginning of a new period. The specifics for timers based on that clock are

- When setting a timer with `timer_settime()` the element `it_value` of `struct_timespec` specifies always a value relative to the beginning of a period.
- Timers dont get armed with `timer_settime()`, they get armed by a subsequent `clock_settime()` call. This allows to start a group of timers synchronously.

Again once a clock is registered, the programmer sees and uses only the standard POSIX API for timers.

Having standardized on `sigwait()` calls for responding to asynchronous events , we can make IO-events in general available through this interface. Therefore we added the call

```
int event_create(int fildes,
    struct sigevent *restrict evp,
    eventid_t eventid);
```

which tells a driver via `ioctl()` to prepare for that.

5 Kernel Implementation

For best possible performance, latency, and jitter figures we picked a two kernel approach. Thanks to the model of a multithreaded process – which is now standard for the Linux kernel and the glibc – the existence of the second kernel could be made almost completely transparent to the user. For a specially marked process the real time kernel handles the threads above a certain static priority. The trick is to have the real time kernel offering exactly the same low level system call interface as the Linux kernel. Even the glibc is not aware under which scheduler a thread is executing code. Exactly the same syscall interface does not necessarily mean that the real time kernel must comprise all the functions a Linux kernel offers. In fact the real time kernel is really tiny. For a x86 architecture it only needs 17kB of code. This is made possible by the fact that the execution of the time critical application code needs only minimal functionality. The majority of system calls is only needed during the setup phase. But this phase is not time critical. Therefore the real time kernel itself supports only a minimal set of system calls: basically timers, thread synchronization/communication and scheduling. For the execution of all the other calls e.g. for socket based communication or for file access the "real-time" thread runs temporarily under the control of the Linux scheduler.

The real time kernel can be a dynamically added module. Thanks to the transparency the same binary of a real time process runs on a system with or without real time kernel. To allow for a very fast implementation of the mentioned subset of signalling

[1] In addition it allows for supervision of timeliness, meaning detecting the situation that the timer fires and the specified thread is not already waiting for that event.

in the real time domain one restriction was put in place: There is no signalling across domain boundaries, a timer created in the real time domain can only send to a real time thread, a timer created in the Linux domain can only send to a Linux thread.

For all that, the modifications we had to do to the Linux kernel and to the Ipipe patch have been really only minor. Only the POSIX message queues experienced some more changes.

6 The ERTEC Device Driver

As mentioned earlier, after all the expected cyclic frames have been written to memory by the ERTEC an interrupt event gets created. This bypassing of all the protocol stack based communication – may it be TCP/IP or may it be PROFINET –needs to be reflected in the interrupt handling of the kernel as well. Handling of the real time events has to be done in the real time domain, protocol handling needs to be done in the Linux domain. For such dual domain drivers a `rt_request_irq()` call is provided which specifies two ISRs, a primary one for the real time part oneand a secondary one for the non real time part. The primary one has to deal with the real hardware e.g. interrupt status word of a device and to maintain a "virtual interrupt status" in memory for the non real time ISR. So after having dealt with the real time interrupt events, if the real time ISR detects that the status word still contains non real time interrupt events it will notify the non real time ISR by executing a call `execute_nonrt_handler()`.

Since the real time modules is dynamically loadable and can be brought in later, transparency has to be maintained also at the driver level. The notification function takes care for that. If the primary ISR turns out to be the regular Linux ISR, then the notification function automatically shrinks to a synchronous procedure call. For drivers which have only a real time part the `rt_request_irq()` specifies only one ISR which – depending on the presence of the real time domain – is either installed in the real time domain or the Linux domain.

It is important to mention that the hard real time capability can only be maintained if the primary ISR does the complete handling of the physical IRQ; in other words, the interrupt must be armed again when leaving the primary ISR. Therefore secondary ISR gets notified via a "virtual IRQ" and deals with virtual interrupt status words only. This also implies that real time capability cannot be achieved for shared interrupt lines.

7 Performance Figures

The first set of measurements covers two threads communicating with each other either via semaphores or via POSIX message queues. The measurement covers the three constellations

- Linux thread – Linux thread
- Linux thread – real time thread
- Real time thread – real time thread

The significant improvement seen in the real time domain for the semaphores is caused by two facts: shorter scheduling path and local futexes. For the POSIX message queues there is only a moderate improvement since both domains employ the same code path. Only the scheduling path is shorter. Waking up a Linux thread from the real time domain takes more time since first a switch to the interrupted Linux thread must be done and from there a switch to the woken up thread.

The efficiency of `CLOCK_SYNC` timers compared to regular `CLOCK_REALTIME` timers can be evaluated by running a certain test activation pattern for some specified time and have a low priority thread gathering the left CPU time. With a `CLOCK_SYNC` based solution, significantly shorter periods can be reached before the accumulated time decreases sharply.

For response times first the response to a virtual interrupt was measured with the help of the time stamp counter. For that interrupt the minimal time is shown for a couple of reasons:

- It is a good figure for comparing code path length.
- For high frequency interrupts the average time is close to the minimal time and therefore a good figure when estimating loads
- For multicore systems with dedicated processors and private caches and tlb the maximal time may approach it.

The maximum time was also measured across various loads and interrupt frequencies. Although this is the figure a real time user is ultimately interested in, it is not particularly useful for general use. It depends to much on the specific underlying HW components (graphics board, DMA, etc.).

Second the response to the cycle interrupt of an ERTEC board connected via PCI to the host computer was measured. Taken for measurement was the ERTEC counter which causes the cycle interrupt. It has a 10 nsec resolution. Although the code path length is about comparable to the virtual interrupt

scenario, the lead time to the beginning of the ISR is significantly longer. Of course we had a different load. We intentionally took a KVM patched kernel running a Windows XP allocating 512 MB of memory. This was to demonstrate the good isolation provided by the Ipipe-virtualization layer. There is no significant difference for minimal ISR time. In other words the time needed to get through the ERTEC, the PCI-bridges, the interrupt controller and executing the interruption dominates. Other things worthwhile to mention are:

- The slightly larger numbers for SMP-operation, which are basically due to the more expensive spinlock.
- The decreased sensitivity to load when one CPU is dedicated to the real time process for exclusive use.

How important an efficient connection to a host is, is highlighted by the fact, that reading out an interrupt status word from the ERTEC takes almost 1 sec, a time span in which a middle class PC can execute more than 1000 instructions. Luckily, integrated into the ERTEC ASIC is an ARM 946. In case of an additional host processor the ARM 946 may be used to bypass this bottleneck by having the integrated ARM 946 to maintain interrupt status words directly in main memory and to start interrupt notification for cycle begin in advance. This cuts lead time[2] as well as the time of useless spinning. If the real time Linux system runs directly on this integrated ARM 946, the relations change drastically. On the one side the access times to ERTEC registers are much shorter and have less jitter. On the other side instruction execution takes much longer due to a 150 MHz CPU clock and a less powerful interface to memory.

[2]To what extent an MSI-based signalling of a PCIe card can further improve the behavior still needs to be evaluated. At least the nasty shared interrupt lines will disappear.