Fraunhofer Institut
Experimentelles
Software Engineering

Harte Echtzeit unter Linux
Fallstudie RTAI vs. RT-Preempt

Autoren:
Jonas Mitschang

IESE-Report Nr. 058.07/D
Version 1.0
20. März 2007

Eine Publikation des Fraunhofer IESE

Das Fraunhofer IESE ist ein Institut der Fraunhofer-Gesellschaft.
Das Institut transferiert innovative Software-Entwicklungstechniken, -Methoden und -Werkzeuge in die industrielle Praxis. Es hilft Unternehmen, bedarfsgerechte Software-Kompetenzen aufzubauen und eine wettbewerbsfähige Marktposition zu erlangen.

Das Fraunhofer IESE steht unter der Leitung von
Prof. Dr. Dieter Rombach (geschäftsführend)
Prof. Dr. Peter Liggesmeyer
Fraunhofer-Platz 1
67663 Kaiserslautern

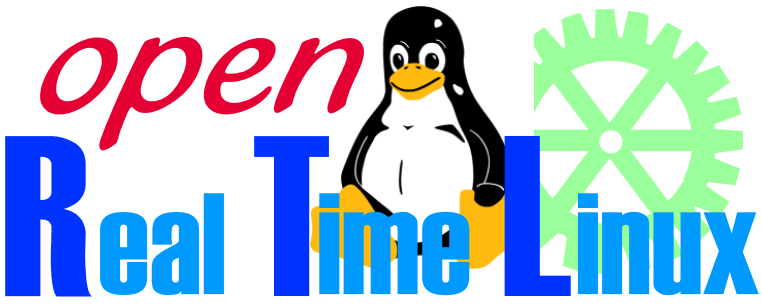
Harte Echtzeit unter Linux
Fallstudie RTAI vs. RT-Preempt

Eine Veröffentlichung des RTLOpen-Projekts
FKZ 01 IS C14

Autor:
Jonas Mitschang (IESE)

RTLOpen-Report 010/D
Version 1.0
20.03.2007

Klassifikation: öffentlich, final



Zusammenfassung

Diese Fallstudie vergleicht die beiden Linux Echtzeitsysteme RTAI und RT-Preempt. Mittels zwei einfachen Testprogrammen wird in Kapitel 3 das Echtzeitverhalten von RTAI analysiert. In Kapitel 4 wird auf dem gleichen Rechner ein RT-Preempt Kernel installiert und evaluiert.

Kapitel 5 gibt einen tabellarischen Überblick über Vor- und Nachteile der beiden Echtzeit-Erweiterungen. Generell ist zu sagen, dass RTAI zwar ein besseres Echtzeitverhalten zeigt als RT-Preempt. Allerdings ist die Installation der Systeme und die Implementierung von Software in RT-Preempt um einiges einfacher als bei RTAI.

Schlagworte: Linux, Open Source, Harte Echtzeit, RTAI, RT-Preempt, CONFIG_PREEMPT_RT

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung und Forschung unter dem Förderkennzeichen FKZ 01 IS C14 gefördert.
Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

Inhaltsverzeichnis

1 Motivation

Mit RTAI [@RTAI] gibt es unter Linux eine Möglichkeit, harte Echtzeit zu garantieren. Allerdings ist die Installation aufwändig und die Benutzerschnittstelle von RTAI relativ komplex und schreckt deshalb oft vor dem Einsatz von RTAI ab. Viel eleganter wäre es, die Posix API von Linux zu verwenden und den Linux Kernel echtzeitfähig zu machen. Diesen Ansatz verfolgt RT-Preempt. Mit RT-Preempt wird es sehr einfach, Echtzeitanwendungen zu entwickeln die vielleicht (noch) keine so gute Leistung wie RTAI bieten, aber dafür in der Implementierung wesentlich schneller zu guten Ergebnissen führen. Für viele Anwendungen sollte dabei RT-Preempt als Garant für Echtzeit hinreichend sein.

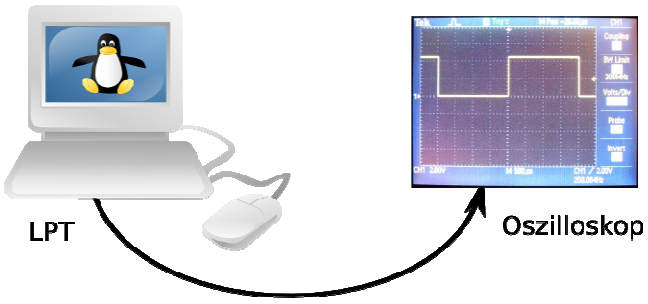
Diese Fallstudie vergleicht die beiden Echtzeiterweiterungen RTAI und RT-Preempt beschreibt und Vor- und Nachteile.

2 Ziel und Vorgehensweise der Fallstudie

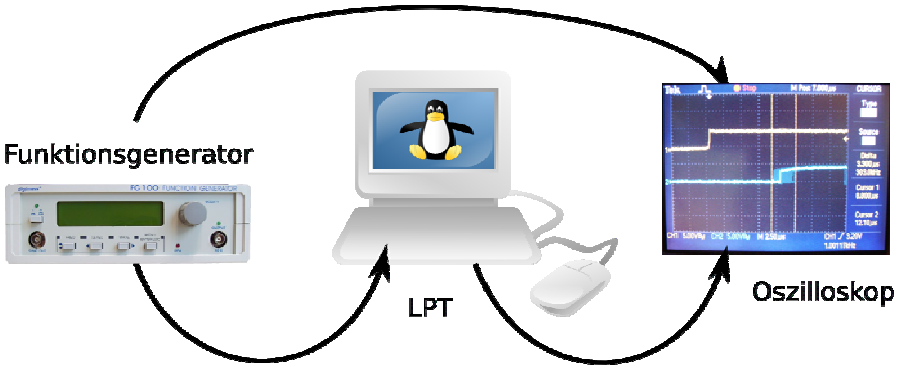
In dieser Fallstudie wird RTAI mit RT-Preempt hinsichtlich der Leistungsfähigkeit verglichen. Kerngrößen der Leistungsfähigkeit eines Echtzeitsystems sind Latenz und Jitter. Die Latenz eines Echtzeitsystems beschreibt die Zeit, die zwischen dem Auftreten eines Ereignisses und dessen Abarbeitung vergehen. Jitter ist die Schwankung, der die Latenz ausgesetzt ist. Diese beiden Eigenschaften von Echtzeitsystemen werden in dieser Fallstudie qualitativ mittels eines einfachen Testprogramms miteinander verglichen.

2.1 Testaufbau

Die Leistungsfähigkeit wird über ein Signal am LPT-Port abgegriffen und extern mit Hilfe eines Oszilloskops gemessen.



Jitter und Latenz werden auf zwei unterschiedlichen Wegen untersucht: Zum einen wird ein periodischer Thread gestartet, der mit einem festen Intervall ausgeführt wird. Die Streuung der Aufrufe dieses Threads durch den Echtzeit Scheduler ist diesem Fall der Jitter während die Latenz die Abweichung der tatsächlichen Wartezeit von der vorgegebenen Wartezeit darstellt.



Im zweiten Test wird ein periodisches Signal fester Periodendauer auf den Interrupt-Eingang des LPT Ports gegeben. Der Echtzeit Thread reagiert auf diesen Interrupt und gibt das Signal an das Oszilloskop weiter, wo Jitter und Latenz direkt abgelesen werden können. Latenz ist die Verzögerung des Signals am Interrupt Eingang zum Signal am Ausgang (beide Signale sind gleichzeitig am Oszilloskop zu sehen) und der Jitter ist die Abweichung der Latenz über die Zeit.

3 Echtzeit unter Linux mit RTAI

3.1 RTAI

RTAI (RealTime Application Interface) wird vom „Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano“ (DIAPM) in Italien und der RTAI Community entwickelt. Bei RTAI handelt es sich um einen Kernel Patch, der dem Linux Kernel eine Hardware Abstraktionsschicht (HAL) bereitstellt. Der Linux-Kernel läuft auf dieser Abstrakttionsschicht als ein Prozess. Grundlage des HAL bildet ADEOS [@Adeos]. Eine Programm-Bibliothek (API) wird ebenfalls mitgeliefert um den Zugriff auf die RTAI-spezifischen Funktionalitäten (Shared Memory, Named Pipes, Semaphoren, Timer, ...) zu bieten.

Folgende Rechnerarchitekturen werden in der aktuellen Version 3.5 unterstützt:

- x86 (mit und ohne FPU und TSC)
- x86_64 (beta)
- PowerPC
- ARM

3.2 Implementierung der Leistungsmessung unter RTAI

Um die Echtzeitfähigkeit von RTAI zu untersuchen wurde ein kleines Testprogramm geschrieben, welches mit einer definierten Periodendauer Impulse auf der parallelen Schnittstelle erzeugt. Das Programm kann zum einen für die RTAI Bibliothek übersetzt werden und zum anderen ohne RTAI Unterstützung, um das Verhalten des Vanilla-Linux Kernels zu testen.

```c
#include <sys/io.h>
#include <rtai_lxrt.h>
#define LPT_PORT 0x378          // LPT Portnummer
#define DELAY 250               // Delay in Mikrosekunden
#define RTAI                    // RTAI benutzen?

void waitPeriod() {
#ifdef RTAI
    rt_task_wait_period();
#else
    usleep(DELAY);
#endif
}
int main() {
    RT_TASK *task;
    int gray_code[] = {0,1,3,2};
    int stepper_position = 0;

    if (ioperm(LPT_PORT, 3, 1)) {
        perror("Keine Berechtigung auf den LPT Port.");
        exit(1);
    }
    outb(0, LPT_PORT + 2);

#ifdef RTAI
    // Der "periodic mode" ist am besten geeignet, da wir mit einer festen Frequenz arbeiten
    rt_set_periodic_mode();

    // Starte den Echtzeit-Timer mit 1 Mikrosekunden Intervallen
    start_rt_timer(nano2count(1E4));

    // Ordne diesen Task dem Echtzeit Scheduler zu
    if (!(task = rt_task_init(nam2num("TASK"), 15, 0, 0))) exit(1);
    rt_task_make_periodic(task, rt_get_time(), nano2count(1E3*DELAY));
#endif

    // Endlosschleife, die Impulse sendet
    while (1) {
        outb(0 , LPT_PORT);
        waitPeriod();
        outb(255 , LPT_PORT);
        waitPeriod();
    }
}
```

Abbildung 1 Implementierung des Testprogramms für periodische Threads unter RTAI

Mit einem zweiten Testprogramm soll die Latenz und der Jitter von RTAI bei Benutzung eines Hardware Interrupts untersucht werden. Aus Sicht des Echtzeit-Schedulers ist es ein großer Unterscheid, ob es sich um einen periodischen Thread oder um einen Hardwareinterrupt handelt. Während es bei einem periodischen Thread genau vorhersehbar ist, wann der nächste Kontext-Wechsel auf diesen Thread ausgeführt werden muss, verhält es sich bei Interrupt Aufforderungen von der Hardware gegenteilig: Der Scheduler kann keine Vorhersage machen, wann der nächste Interrupt ausgelöst wird und muss zu jeder Zeit bereit sein, einen Kontext-Wechsel auszuführen.

Bei dem zweiten Programm handelt es sich um ein Kernel Modul, welches bei einem Interrupt am LPT Port (IRQ 7) einen Impuls auf einem anderen Pin des Ports generiert. Der Versatz zwischen den beiden Impulsen kann auf einem Oszilloskop gemessen werden.

```c
#include <stdio.h>
#include <errno.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/io.h>

#include <rtai_lxrt.h>
#include <rtai_sem.h>
#include <rtai_usi.h>

#define LPT_PORT 0x378
#define LPT_IRQ 7

static volatile int ovr;

static void *timer_handler(void *args)
{
    int i;
    RT_TASK *handler;
    if (!(handler = rt_task_init_schmod(nam2num("HANDLR"), 0, 0, 0, SCHED_FIFO, 0xF))) {
        exit(1);
    }
    rt_allow_nonroot_hrt();
    mlockall(MCL_CURRENT | MCL_FUTURE);
    rt_make_hard_real_time();

    rt_request_irq_task(LPT_IRQ, handler, RT_IRQ_TASK, 1);
    rt_startup_irq(LPT_IRQ);
    rt_enable_irq(LPT_IRQ);

    // Endlosschleife
    while (1) {
        ovr = rt_irq_wait(LPT_IRQ);
        outb_p(0xff, LPT_PORT);
        rt_ack_irq(PARPORT_IRQ);
        rt_pend_linux_irq(LPT_IRQ);
        for (i = 0; i < 15000; i++);
        outb_p(0, LPT_PORT);
    }
}

int main() {
    RT_TASK *maint;
    if (!(maint = rt_task_init(nam2num("main"), 1, 0, 0))) {
        exit(1);
    }

    if (iopl(3)) {
        printf("iopl err\n");
        rt_task_delete(maint);
        exit(1);
    }
    outb_p(0x10, LPT_PORT + 2);

    rt_thread_create(timer_handler, NULL, 10000);
    while (1) sleep(1);
}
```

Abbildung 2 Implementierung des Testprogramms für Hardware Interrupt Latenz und Jitter

Um die Leistungsfähigkeit der Echtzeiterweiterung zu bewerten, wird während die Testprogramme ausgeführt werden, durch ein weiteres

Programm auf dem System eine hohe Last erzeugt. Dies wird über den folgenden Befehl erzielt:

nice -20 dd if=/dev/urandom of=/tmp/urandom

Der Aufruf bewirkt zum einen eine hohe Kernel Last (durch die Berechnung des Entropy Pools für urandom im Kernel) und zum anderen werden durch die Festplattenzugriffe viele Hardware Interrupts ausgelöst.

3.3 Testplattform

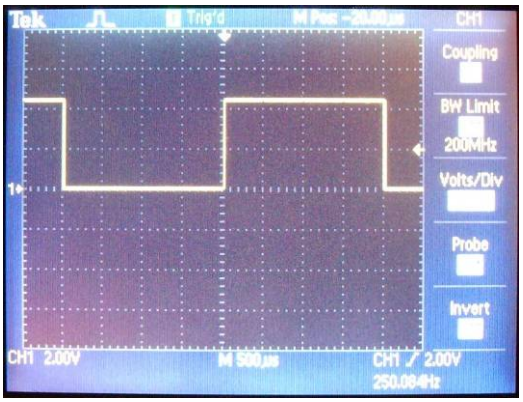
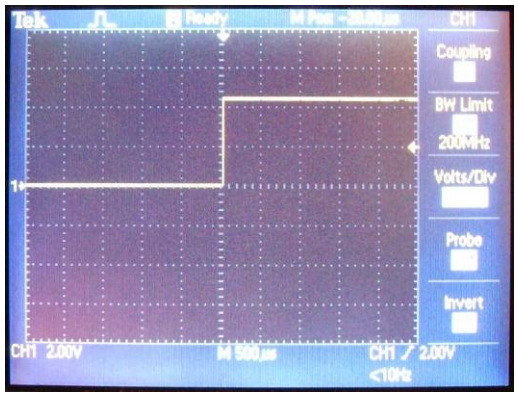
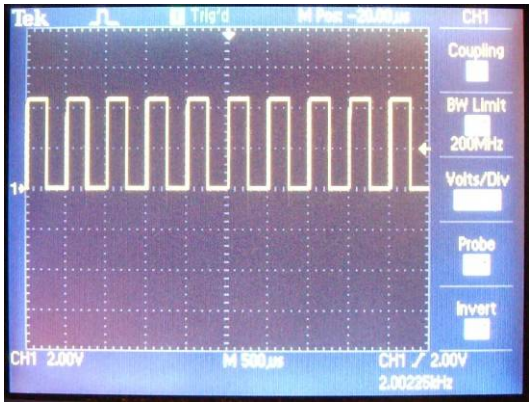
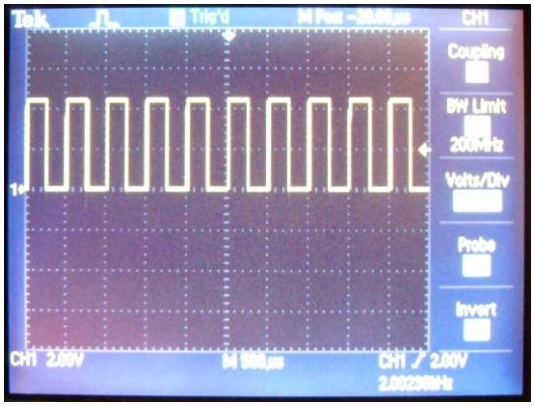
Das Testprogramm wird auf einem Pentium 4 (Taktfrequenz: 3GHz) mit Linux Kernel 2.6.12.2 und RTAI Patch – im folgenden „System A“ bezeichnet - ausgewertet.

3.4 Ergebnisse

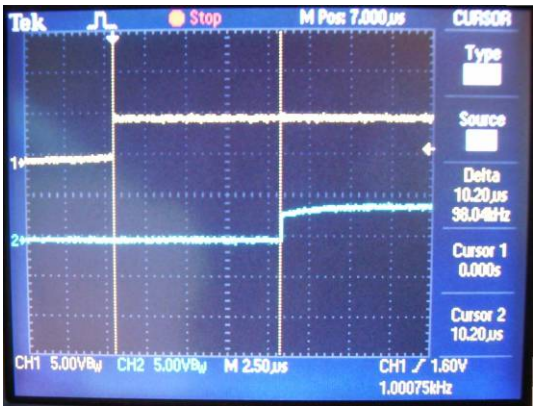
Die Tabelle zeigt die mit einem Speicheroszilloskop gemessene Spannung am LPT Port. Die Zeit-Achse ist auf 500µs pro div eingestellt. Das Testprogramm ändert alle 250µs den Zustand des gemessenen Pins des LPT Ports (die logische 0 entspricht 0 Volt; die logische 1 entspricht 4.5 Volt).

Die Tabelle zeigt das Testprogramm ohne Echtzeit und mit Echtzeit (Periodischer LXRT Thread) jeweils ohne Belastung des Systems und unter hoher Last. Ohne Echtzeit bewirkt ein „udelay“ (welches einen Prozess in den sleeping Zustand versetzt) eine minimale Verzögerung von 2ms (kann im Kernel auch auf 1ms herunter gestellt werden). Wenn das System unter Last ist, werden die Verzögerungen sehr viel größer und unregelmäßiger.

Die Messungen im Echtzeitbetrieb fallen sehr positiv auf: Unabhängig von der Last des Systems wird exakt alle 250µs ein Impuls gemessen:

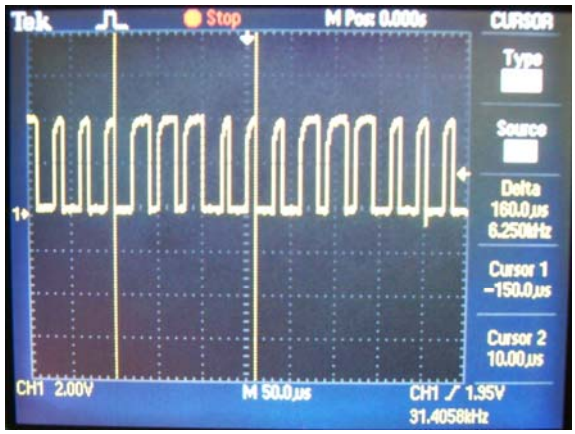
	System ohne Last	System unter Last
Ohne Echtzeit System: A		
Echtzeit (LXRT) System: A		

Die Messung der Hardware-Interrupts ergab die folgenden Bilder:

	Latenz	Jitter
System A		

Die Latenz setzt sich zusammen aus der Gatterlaufzeit des Interrupt Controllers (+CPU), der Interrupt-Service-Routine (ISR) und dem eigentlichen Context-Switch und ergibt sich beim verwendeten System A zu 10.2µs. Der Jitter, also die Abweichung der Latenz über die Zeit ergibt sich zu 1.65µs.

Wenn die Periodendauer unter RTAI verringert wird, so werden die einzelnen Perioden unterschiedlich lang (abhängig vom rt_timer) aber über die Summe ergibt sich im Durchschnitt genau die gewünschte Periodendauer. Im folgenden Bild wurde 16µs als Wartezeit gewählt und die gemessenen zehn Flanken liegen genau 160µs auseinander.



Aus System A kann man bei periodischen Threads bis zu 3µs Periodendauer herab gehen (danach bleibt das System stehen, der Thread läuft weiter).

4 Echtzeit unter Linux mit RT-Preempt

4.1 RT-Preempt

RT-Preempt (auch bekannt als CONFIG_PREEMPT_RT Patch) ist ein Kernel Patch, der dem Linux Kernel einen verbesserten Scheduler bereit stellt und ihn zu fast jeder Zeit unterbrechbar und damit echtzeitfähig macht. Er wird von RedHat Programmierer Ingo Molnar entwickelt und stützt sich auf Thomas Gleixners „Generic Clock Event Layer, High Resolution Timer“. Ziel ist es, langfristig Realtime in den Linux Kernel zu bekommen und dadurch den lästigen Kernel Patches aus dem Weg zu gehen. RT-Preempt ist auf dem besten Weg dahin (geplant ist der Merge des Realtime Core Codes für Kernel 2.6.23) und der High Resolution Timer ist bereits Teil des aktuellen Linux Kernels (seit Version 2.6.16).

Es werden die gleichen Rechner-Architekturen wie bei RTAI unterstützt.

4.2 Implementierung der Leistungsmessung unter RT-Preempt

Es wird das gleiche Testprogramm wir bei RTAI verwendet, allerdings ist die RTAI Bibliothek für den Echtzeitbetrieb nicht nötig. Es kann mit der normalen Linux API (z.B. udelay) gearbeitet werden. Das Testprogramm muss lediglich den Befehl sched_setscheduler aufrufen um den Echtzeit Round Robin Scheduler (SCHED_RR) zu verwenden. Man kann auch beliebige Prozesse aus der shell heraus auf andere Scheduler übertragen, bzw. deren Prioritäten festlegen. Unter Debian findet sich hierzu im schedutils Paket der Befehl chrt. Der Prozess mit der PID 12345 kann mit folgendem Befehl dem Round Robin Scheduler mit der Priorität 80 zugeordnet werden:

```
chrt -r -p 80 12345
```

Die Implementierung des zweiten Tests (Hardware-Interrupt Latenz und Jitter) konnte leider unter RT-Preempt nicht durchgeführt werden. Es gibt keine Möglichkeit im User-Space auf Hardware-Interrupts zu reagieren. Es existiert zwar ein Patch von Peter Chubb, der Hardware-Interrupts in den User-Space weiter gibt aber dieser Patch funktioniert nur für Kernel 2.6.11. Das Patchen des verwendeten Kernels (2.6.18) war leider nicht möglich. Das folgende Programm zeigt die prinzipielle Implementierung im Kernel-Space. Allerdings konnte die PC-Hardware nicht so programmiert werden, dass der parallele Port Interrupts für das Betriebssystem erzeugt. Beim RTAI-Test hat die Aufgabe der Programmierung der PC-Hardware das Modul parport_pc übernommen welches aber nicht gleichzeitig mit dem

Testprogramm laufen kann. Mit einer anderen Interrupt-Quelle wäre die Messung möglich gewesen, wurde aber aus Zeitgründen nicht mehr durchgeführt.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <asm/io.h>
#include <linux/interrupt.h>
#include <rtai.h>
#include <rtai_sched.h>

#define LPT_PORT 0x378
#define LPT_IRQ 7

int private_data = 0;

// Interrupt Handler
static int handler(int irq, void *dev_id, void * data) {
    int i;

    // Impuls auf dem LPT erzeugen
    outb(0xff, LPT_PORT);
    for (i = 0; i < 15000; i++);
    outb(0, LPT_PORT);
    return IRQ_HANDLED;
}

// Initialisierung
int init_module(void) {
    if (request_irq(LPT_IRQ, handler, SA_INTERRUPT |
        SA_SHIRQ, "Parallelport", &private_data)) {
        // Fehler
        printk("could not request irq %d\n", LPT_IRQ);
        return -EBUSY;
    }

    // Interrupt Modus
    outb(0x34, LPT_PORT + 0x402);
    outb(0x10, LPT_PORT + 2);
    outb(0xff, LPT_PORT);
    return 0;
}

// Cleanup
void cleanup_module(void) {
    free_irq(LPT_IRQ, &private_data);
    return;
}

MODULE_LICENSE("GPL");
```

Abbildung 3 Implementierung des Testprogramms mit Hardware Interrupts unter RT-Preempt

4.3 Testplattform

Der Rechner, auf dem die Auswertung ausgeführt wird (System B) ist derselbe wie bei der RTAI Auswertung, allerdings wird kein RTAI Kernel verwendet sondern ein 2.6.18-rt5 Kernel (mit RT-Preempt Patch).

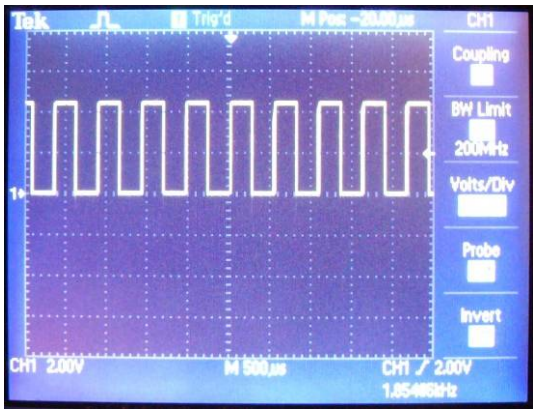
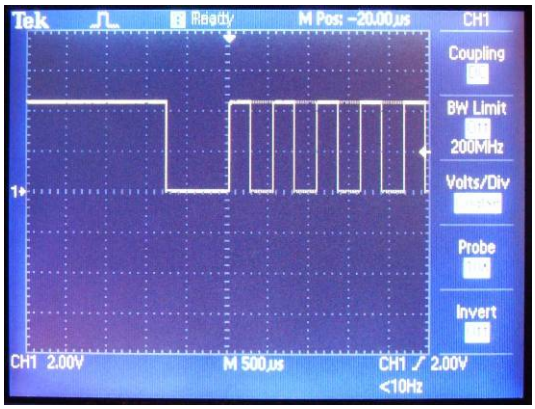
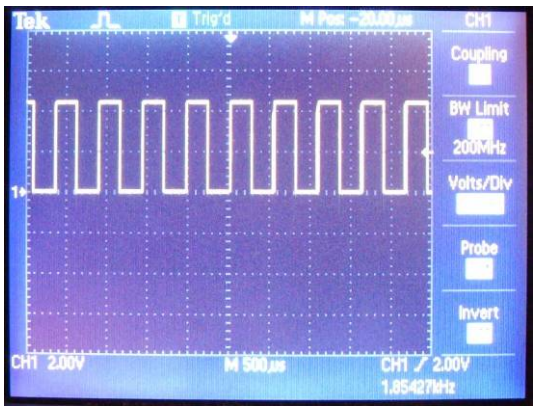
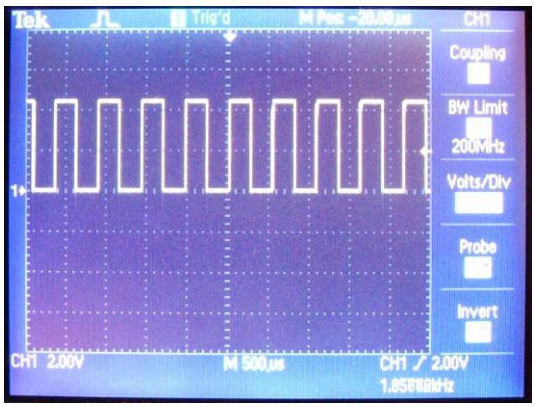
4.4 Ergebnisse

Die Tabelle zeigt, wie in der Auswertung von RTAI, die gemessene Spannung am LPT Port (Zeit-Achse ebenfalls 500µs pro div).

Dank des High Resolution Timers hat das Testprogramm ohne Echtzeit (also mit normalem Linux Scheduler: SCHED_OTHER) eine sehr höhere Auflösung als bei System A mit dem Standard-Linux auf dem RTAI-Kernel. Wenn sich das System nicht unter Last befindet können die 250µs Impulse gemessen werden. Ist das System B unter Last, verhält es sich natürlich wie beim System A: Die Impulse werden undefiniert in die Länge gezogen.

Teilt man das Programm dem Echtzeit Scheduler (SCHED_RR) zu, zeigt es sich von der Systemlast unbeeindruckt: Das Timing des Impulses ändert sich nicht mit der Last des Systems. Die Impulse sind aber allgemein etwas länger als die 250µs. Das liegt daran, dass kein periodischer Thread erzeugt wird (wie es bei RTAI der Fall ist), sondern dass jeweils nach der Änderung des Zustands des LPT-Ports der Aufruf „usleep(250);“ erfolgt. Das führt dazu, dass das Intervall des Testprogramms die Summe der Ausführungszeiten von usleep() und dem Portzugriff ist.

Bei System B konnte als kürzester Delay mit Kontextwechsel 30 µs ausgemacht werden. Wenn man die Periodendauer noch weiter herabsetzt, dann ändert sich das Bild nicht mehr.

	System ohne Last	System unter Last
Standard Scheduler ohne Echtzeit System: B		
Echtzeit Round Robin Scheduler, hohe Priorität System: B		

5 Fazit

5.1 Vor- und Nachteile

	RTAI	RT-Preempt
Installation	Relativ zeitaufwendig – man muss aufpassen welche Software-Versionen zueinander passen. So funktioniert z.B. COMEDI (Bibliothek für Analog Digital Wandlerkarten) nur bis Kernel 2.6.12.2.	Einfache Installation. Nach dem Patchen muss man lediglich im Kernel “CONFIG_PREEMPT_RT” und die High Precision Timer aktivieren und den Kernel wie gewohnt compilieren.
Implementierung	Es muss die RTAI Schnittstellen-Bibliothek verwendet werden, die teilweise ein wenig gewöhnungsbedürftig ist und nicht dem POSIX-Standard entspricht.	Nur ein Aufruf von sched_setscheduler nötig. Danach kann die normale GNU/Linux API verwendet werden.
Interrupts	Die Verwendung von Hardware-Interrupts ist sowohl im User- als auch im Kernel-Space einfach umzusetzen.	Die Verwendung im Kernel-Space ist wie gewohnt (mit request_irq) durchzuführen und im User-Space nicht möglich.
Performance	Kürzester Delay: 3µs Gemessene Latenz: 10.2µs. Gemessener Jitter: 1.65µs.	Kürzester Delay: 30µs Gemessene Latenz: n.a. Gemessener Jitter: n.a.

Die Leistungsfähigkeit von RTAI ist insgesamt höher, der Zugriff auf hardwarenahe Ressourcen ist auch aus dem User-Space möglich.
Die Einrichtung periodischer Threads unter ist unter RT-Preempt (noch) nicht einfach möglich.

5.2 Ausblick

Die Leistungsfähigkeit des Echtzeitschedulers des Vanilla-Linux-Kernels nimmt kontinuierlich zu. Die Merge-Timeline der Kernel Maintainer sieht es vor, zu jeder neuen Version einen weiteren RT-Preempt Patch in den Kernel zu übernehmen. Es dürfte also nur eine Frage der Zeit sein, bis die Performance von RTAI erreicht wird.

6 Referenzen

[@Adeos] http://home.gna.org/adeos/

[@RTAI] http://www.rtai.org

[@P_RT] http://rt.wiki.kernel.org

Dokumenten Information

Titel:	Harte Echtzeit unter Linux Fallstudie RTAI vs. RT-Preempt
Datum:	20. März 2007
Report:	IESE 058.07/D
Status:	Final
Klassifikation:	Öffentlich