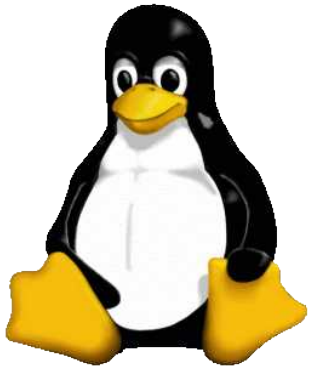
Echtzeitprogrammierung mit Linux und RTAI



Projektarbeit Datenbanken

NTA FH Isny
12. Info

Andreas Krug & David Mayr

Inhaltsverzeichnis

1 Aufgabenstellung

Echtzeitprigrammierung unter RTAI – Entwurf einer Entwicklungumgebung.

Für gängige PC's soll eine Entwicklungsumgebung geschaffen werden.

Wir sollten uns zunächst füe eine Standard Linux-Distribution entscheiden, wie zum Beispiel SUSE Linux, GENTOO oder UBUNTU.

In diese Linux-Installation soll die freie Echtzeiterweiterung RTAI integriert werden.

Im Anschluss daran ist eine CD zu erstellen, mit der man die einzelenen Installationsschritte auf den Praktikumsrechnern der FH automatisieren kann.

Wenn die bereits oben genannten Schritte gemacht sind, sollten wir zwei unter WinCE durchgeführte Pratika auf RTAI-Basis umsetzen. Die Versuche sollten möglichst einfach gestaltet sein, so dass die Praktika im Rahmen eines 90-minütigen Praktikums nachvollziebar ist.

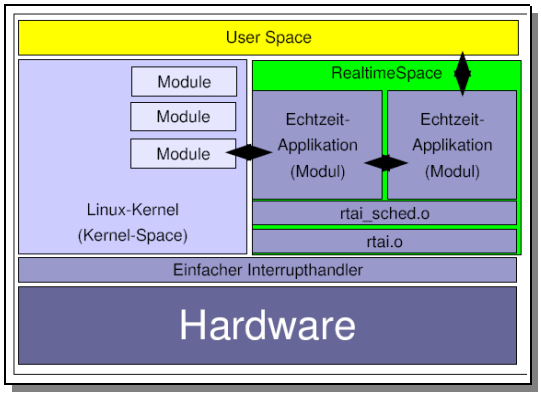
Am Ende sind noch ein paar Fragen zu klären: Welche Debug-Möglichkeiten gibt es auf dem System? Worin liegen die Unterschiede zwischen dem User-Space RTAI und dem RTAI im Kernel-Space? Wo liegen die Grenzen des Systems bezuglich des Echtzeitverhaltens?

2 Einführung

2.1 Was ist RTAI

RTAI (Real Time Application Interface) ist eine freie Erweiterung von Linux zu einem Echtzeitbetriebssystem. Gegründet wurde RTAI von Prof. Paolo Mantegazza vom *Dipartimento di Ingegneria Aerospaziale* der Universität Mailand. RTAI wurde von Beginn an als Open-Source-Projekt von einer größeren Entwicklergemeinde weiterentwickelt, wobei heute neben dem weiterhin koordinierend tätigen Prof. Mantegazza vor allem Philippe Gerum als sehr aktiver Mitarbeiter zu nennen ist.

Es gibt inzwischen auch eine Reihe diverser verwandter oder kooperierender Projekte, wie zum Beispiel Rtnet (ein Echtzeit-Netzwerk-Protokoll) und Linux Trace Toolkit.



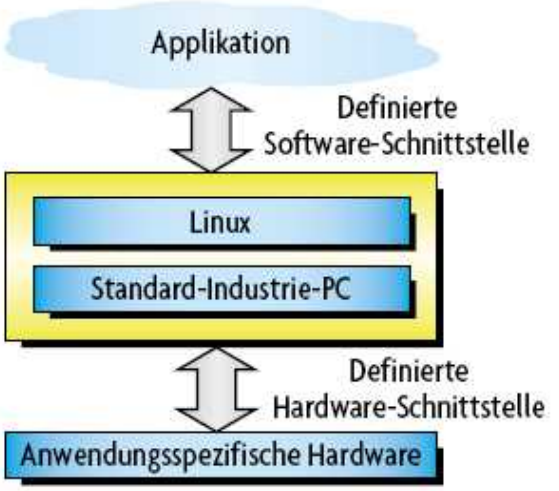
Ein großer Pluspunkt von RTAI ist, dass es mit der Variante LXRT möglich ist, Hard-Realtime-Tasks im Userspace laufen zu lassen und damit die Schutzmechanismen von Linux zu nutzen. Dies erfolgt ohne größere Einbußen im Bereich der Latenzzeit und ohne großen Overhead. Bei anderen Echtzeit-Systemen, welche ausschließlich im Kernel–Space laufen, kann sich ein Fehler im Programmablauf wesentlich verheerender auswirken.

Das Realtime Application Interface (RTAI) bindet Echtzeit-Module in den Linux-Kern ein. Langfristige Verfügbarkeit, Stabilität, harte Echtzeit und universelle Kommunikation – das erwarten Entwickler wie Strategie-Entscheider von einem modernen, industriellen Betriebssystem. Dazu noch qualifizierter Support durch den Hersteller. Und kosten darf das alles natürlich möglichst auch nichts. Utopie? Nein, die Linux- Echtzeit-Erweiterung RTAI bietet eine solide und hersteller-unabhängige Alternative, die allen Abkündigungsplänen von Zulieferern auf Dauer standhält.

RTAI wird von einer großen Zahl von Entwicklern in vielen Ländern als Basis für ihre Entwicklungen im Realtime-Bereich verwendet, hat aber ebenso wie RTLinux naturgemäß für den Standard-Büro-Desktop-Computer-Anwender keine direkte Bedeutung.

Industrie-Applikationen mit Standard-Betriebssystemen zu realisieren, hat Tradition. Viele heute existierende Lösungen wurden Anfang der 90er Jahre auf Basis von DOS entwickelt, über Jahre gepflegt und an die Bedürfnisse der Anwender angepasst. Dies war mit DOS sehr leicht möglich, da es kein Betriebssystem im modernen Sinne war, das eine vielfältige Infrastruktur für Programme bereitstellt. Vielmehr diente DOS vornehmlich dazu, die applikationsspezifische Software zu laden und auszuführen. War dies geschehen, hatte der Entwickler die Maschine vollständig unter seiner Kontrolle. Da es keine Dienste gab, die den Programmablauf unterbrechen konnten, war es ein Leichtes, schnell und deterministisch auf Hardware-Ereignisse zu reagieren und harte Echtzeit - Bedingungen zu erfüllen. Diesen Vorteil erkaufte sich der Entwickler durch den Nachteil, dass er viele Treiber für spezielle Hardware selbst entwickeln musste. Die Anforderungen an Embedded - Applikationen im Industriebereich haben sich im Laufe der letzten zehn Jahre stetig weiterentwickelt. Während zu Beginn der 90er Jahre ein serielles Interface oft noch die einzige Standard - Schnittstelle zur Außenwelt war, ist heute eine ganze Reihe weiterer Ports hinzugekommen: USB, IEEE 1394, diverse Feldbusse und nicht zuletzt Ethernet werden heute in vielen Applikationen gefordert, und vermutlich wird die Anzahl der zu unterstützenden Schnittstellen in Zukunft eher noch ansteigen. Das Entwickeln einer komplexeren Echtzeit-Anwendung unter DOS ist heute mit vertretbarem Aufwand nahezu unmöglich, sodass sich viele Entwickler nach einer

leistungsfähigeren Alternative umsehen. Die Software soll die Funktionen eines modernen 32-bit- Betriebssystems bieten, ohne dem Entwickler den Zugriff auf bestimmte Systemressourcen zu versperren. Eine solche Alternative ist das Betriebssystem Linux. Als Unix-Variante bietet es von Haus aus Hardware-Unabhängigkeit und die Unterstützung aller gängigen offenen Standards. Da es auch in Form von Quellcode vorliegt, steht es dem Kunden unabhängig von Hersteller - Entscheidungen zur Verfügung – wenn es sein muss, über Jahrzehnte. Dass Linux den Stabilitätskriterien der Industrie standhält, hat der langjährige, ausgeprochen erfolgreiche Einsatz im Server- und Kommunikationsbereich gezeigt, der ähnliche Anforderungen an Verfügbarkeit und Sicherheit der Systeme stellt. Es bleibt die Frage der Echtzeit-Fähigkeit, die für industrielle Aufgaben oft ein Muss ist.

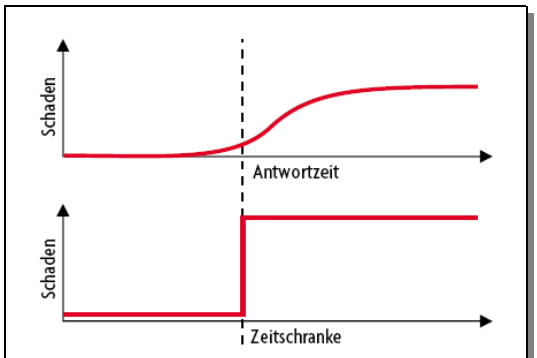


Hardware-Unabhängigkeit durch Linux. Durch Verwendung von offenen Schnittstellen bei Hardware und Software (Linux) wird die Kundenapplikation unabhängig von Prozessorgenerationen und Architekturen.

Echtzeit: Was ist „hart", was ist „weich"?

Konventionelle Betriebssysteme sind in der Regel darauf optimiert, dem Benutzer „im Mittel" eine möglichst hohe Performance zu liefern. Insbesondere bedeutet dies, dass einzelne Ereignisse der Hardware, ausgelöst durch Interrupts oder periodisch laufende Tasks, nicht die Möglichkeit haben, die Benutzer- Interaktion mit dem System nachhaltig zu blockieren. Damit eignen sich diese konventionellen Systeme nicht gut für Anwendungen, in denen ein definiertes Antwortverhal- ten auf Hardware-Ereignisse unabdingbar ist. Solche „Echtzeit"-Anforderungen sind jedoch beim Einsatz von Rechnern in industriellen Anwendungen für gewöhnlich gegeben, sodass hier meist speziell für diesen Zweck entwickelte Echtzeit-Betriebssysteme zum Einsatz kommen. In den vergangenen Jahren wurden allerdings auch einige Software- Tools entwickelt, mit deren Hilfe sich die Vorteile des Echttzeit-Einsatzes mit denen von Linux verbinden lassen. Ein solches Werkzeug ist das „Realtime Application Interface RTAI". In der Praxis stellt sich oft heraus, dass Anwender häufig zwar der Meinung sind, ein hart echtzeitfähiges Betriebssystem zu benötigen, bei genauerer Analyse der Applikation stellt sich aber in vielen Fällen heraus, dass dies nicht immer der Fall ist. Deshalb wird hier zunächst definiert, was unter „Echtzeit" zu verstehen ist. Die Frage, was eine Echtzeit-Anwendung ist und was nicht, hängt stark von den Zeitskalen der betrachteten Anwendungen ab. Während Börsenhändler bereits von Echtzeit reden, wenn Kurse innerhalb von 20 Sekunden verfügbar sind, ist es im Bereich der Motorsteuerungen unbedingt notwendig, in wenigen Mikrosekunden auf Ereignisse wie etwa das Eintreffen der Signale eines Positionsmelde- Sensors zu reagieren. Es zeigt sich, dass der Begriff „Echtzeit" letztendlich definiert, dass ein System in der Lage ist, innerhalb von vorgegebenen Zeitspannen unter allen erdenklichen Betriebszuständen auf Ereignisse zu reagieren. Die Zeitspannen selbst werden von den jeweiligen Applikationen vorgegeben. Wichtig ist die Tatsache, dass ein Echtzeit-Betriebssystem nicht nur sicherstellt, *dass* eine Aufgabe, sondern auch, dass sie *innerhalb einer* vordefinierten *Zeitspanne* erledigt wird. Es ist hingegen kein Kritierium, ob ein System besonders *schnell* ist. Neben unterschiedlichen Zeitskalen unterscheiden sich Applikationen aber auch darin,wie stark sich Verletzungen der vorgegebenen Antwortzeiten auf die zu erfüllende Aufgabe auswirken. Bei Video-Anwendungen beispielsweise ist es wichtig, dass Frames mit sehr hoher Wahrscheinlichkeit mit der richtigen Frame- Rate ausgegeben werden. Fällt aufgrund anderer Ereignisse im System die Darstellung eines einzelnen Frames aus, so ist dies prinzipiell kein

„Schaden", da der Benutzer in der Regel den Ausfallüberhaupt nicht bemerken wird. Andere Systeme, wie z.B. Ein Motorregler, können bereits bei einer einmaligen Überschreitung der Zeitschranke einen irreparablen Schaden im gesteuerten System hervorrufen.Der zu erwartende Schaden bei Nicht-Einhaltung der vorgegebenen Deadline ist damit auch das primäre Unterscheidungsmerkmal zwischen „weich" und „hart" echtzeitfähigen Systemen. Bei der Analyse der zur Aufgabenlösung notwendigen Toolsist deshalb neben den typischen Zeiten auch die „Härte" der Anforderungen ein wichtiges Designkriterium.



Schaden bei Überschreitung einer Deadline. Im harten Echtzeit-Systemen tritt der Schaden abrupt ein, wohingegen bei weicher Echtzeit gelegentliche, geringfügige Überschreitungen kaum Auswirkungen haben.

Das Echtzeit-Konzept von RTAI

Harte Echtzeit ist unter RTAI nur im so genannten Realtime-Space möglich. Echtzeit-Programme werden entweder als Kernel- Module implementiert und sind somit in ihrer Funktionalität ebenso eingeschränkt wie gewöhnliche Kernel-Treiber, oder als LXRT-Userspace Programme.

2.2 Voraussetzungen

Um RTAI und dessen Echtzeitfähigkeit nutzen zu können muss auf dem Ziel-System ein angepasster Linux-Kernel laufen. Dazu müssen die originalen Linux Kernelquellen entpackt, mit der RTAI-Erweiterung gepatcht, compiliert und zum booten installiert werden. Dafür muss natürlich ein Compiler – i.d.R. der GCC – vorhanden sein.

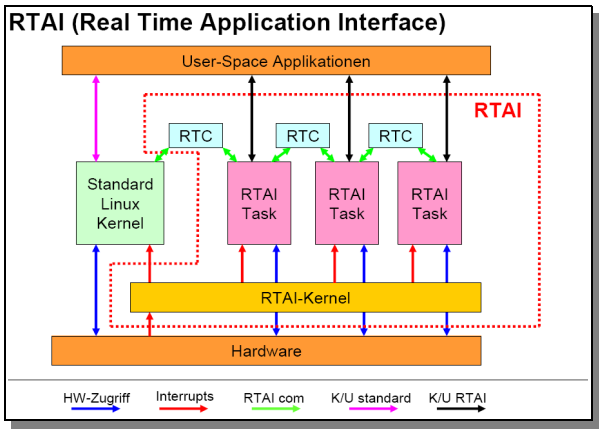
Dann muss noch zusätzlich RTAI bzw. alle mitgelieferten RTAI-Module und Tools compiliert werden.

Eine Kalibrierung des Systems mit den mitgelieferten Tools ist zwar nicht zwingend nötig, wir aber empfohlen.

2.3 Unterscheidung: User- / Kernel-RTAI

RTAI-Prozesse waren ursprünglich nur über Kernel-Module direkt im Kernelspace möglich. Dadurch hat man zwar den Vorteil niedriger Latenzen und direkter Zugriffe, aber auch das Risiko bei einem Programmierfehler das ganze System zu gefährden. Zudem ist die Kommunikation mit UserSpace-Prozessen nur über Umwege möglich und vorhandene Bibliotheks-Funktionen können in Kernel-Modulen nicht verwendet werden.

Aus diesem Grund bietet RTAI die Möglichkeit harte Echtzeitprozesse auch im UserSpace laufen zu lassen (LXRT), ohne dabei große Latenz- / Overhead-Einbußen zu erfahren wie bei anderen Systemen üblich.



Der Echzeit-Teil und die restliche Anwendung tauschen Daten über die von RTAI angebotenen Kanäle aus. Dies garantiert das Echtzeitverhalten des RTAI-Schedulers.

RTAI Echzeit-Code im Kernelspace läuft ohne den von Linux angebotenen Speicherschutz. Eine Beschränkung des Echtzeit-Codes auf das Nötigste erscheint deshalb sinnvoll.

Der einfachste Aufbau eines **Kernelmoduls** ist wie folgt:

```
// ===> testmodul/testmodul.c

#include <linux/module.h>

int xinit_module(void) {
       printk("Test-Modul geladen.\n");
       return 0;
}

void xcleanup_module(void) {
       printk("Test-Modul entladen.\n");
       return;
}

module_init(xinit_module);
module_exit(xcleanup_module);

MODULE_LICENSE("GPL");
```

Um es zu compilieren wird das Tool `make` und eine Datei mit dem Namen `Makefile` (Gross-/Kleinschreibung wichtig) benötigt, die mit folgenden Anweisungen gefüllt sein muss:

```
obj-m   := testmodul.o
KDIR    := /lib/modules/$(shell uname -r)/build
PWD     := $(shell pwd)
EXTRA_CFLAGS := -I/usr/realtime/include -I/usr/include/
default:
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
# wichtig: die vorige Zeile MUSS mit einem Tab beginnen!
```

Sind diese Dateien vorhanden, kann das Modul mit `make` kompiliert werden und liegt dann in der Datei `testmodul.ko` vor, die mit `insmod testmodul.ko` geladen werden kann.

Die Ausgabe `Test-Modul geladen` landet dabei im Protokoll vom Syslog, dessen Inhalt mit `tail -f /var/log/messages` angezeit werden kann. Alternativ können Kernelmeldungen auch mit dem Kommando `dmesg` angezeigt werden.

Um das Test-Modul zu entladen, reicht es das Kommando `rmmod testmodul` einzugeben.

Ein RTAI-Programm im **Userspace** wird wie normales Linux-Programm unter Einbindung der RTAI-Header/-Funktionen geschrieben und kann dann mit dem Kommando
`gcc -I/usr/realtime/include/  -o dateiname  dateiname.c` compiliert werden kann.

2.4 Debuggingmöglichkeiten

Das Debuggen in einem RTAI-System kann auf zwei Wege realisiert werden.

Zum einen kann man fertige Tools / IDEs wie z.B. Minute virtual Machine nutzen, welche im Hintergrund mitlaufen und wärend oder nach Beendigung der Prozessabläufe mit Hilfe einer grafischen Oberfläche die Daten und Zustände ausgewerten.

Da diese komfortableren IDEs nur kommerziell und für viel Geld zu haben sind, haben wir uns für folgendes entschieden:

Eine andere Möglichkeit die verschiedenen Vorgänge in eigenen RTAI-Programmen zu debuggen ist, sich selbst mit der RTAI-Funktion `rt_printk()` Debugging-Meldungen zu senden. Diese Meldungen werden mit einem automatisch mit einem Zeitstempel versehen und ohne das Echtzeitverhalten wesentlich zu beeinflussen in einer Protokoll-Datei unter `/var/log/messages` festgehalten. Diese Nachreichten lassen sich dann bequem in einer eigenen Konsole mit dem Befehl `tail -f /var/log/messages` mitverfolgen.

3 Realisierung

Wir haben uns nach einem kurzem Versuch auf unseren Notebooks dazu entschieden, das ganze Projekt auf einem "normalen" PC durchzuführen, da dort weniger Besonderheiten – wie z.B. dynamische CPU-Taktraten zum Stromsparen usw. - zu beachten sind. Unser Test-Gerät ist ein älterer Siemens Desktop-PC mit einer 1000 Mhz Intel Pentium III CPU, 512 MB RAM und einer 20 GB IDE-Festplatte.

3.1 Installation

Im folgenden eine Schritt für Schritt Anleitung um die Echtzeiterweiterung RTAI auf einem bestehenden Linux-System zu installieren. Prinzipiell wäre jede gängige Standard Linux-Distribution geeignet, wir haben uns hier für Ubuntu Linux 6.06 entschieden. Ubuntu ist eine junge und offene - auf Debian Linux basierende - Distribution, die in letzter Zeit sehr an Popularität gewinnt.

3.1.1 Betriebssystem-Installation

Zunächst gilt es das Betriebssystem zu installieren, wir gehen hier wie bei unserem Test-System von einer leeren Festplatte aus.

Ubuntu Linux ist frei und kann kostenlos als CD-Image von http://www.ubuntu.com heruntergeladen werden. Dieses CD-Image kann mit jedem gängigen Brennprogramm auf eine CD gebrannt werden. Anschliessend muss der Test-PC davon gebootet werden.

Bei Ubuntu ist die Installation so geregelt, dass man direkt von der CD in ein Live-System mit graphischer Oberfläche bootet. In diesem Live-System kann man schon ganz normal arbeiten ohne irgend etwas installiert zu haben, allerdings wäre die ganze Arbeit nach einem Neustart zunichte. Deshalb gibt es auf dem Desktop ein Icon mit dem Namen "Installation", der die eigentlich Installationsroutine aufruft um Ubuntu Linux auf die Festplatte zu verfrachten.

Die Intallation dauert auf unserem Test-System ca. 30 Minuten.

3.1.2 Vorbereitungen

Nachdem das System frisch installiert wurde und das erste Mal von Festplatte gestartet hat ist eine Anmeldeaufforderung zu sehen, an der man sich mit dem zuvor eingegebenen Benutzernamen und Passwort anmelden kann. Im Menü findet man dann unter *Systemwerkzeuge* das Programm *Terminal*, in dem man mit dem Befehl `sudo su` und dem Benutzerpasswort zum Adminstrator-Benutzer "root" wechselt. Dies ist nötig, da alle folgenden Kommandos root-Rechte erfordern

Als nächstes empfiehlt sich – sofern Internetverbindung per LAN / DHCP vorhanden – ein Update des Basis-Systems

```
apt-get update  &&  apt-get upgrade
```

Falls man als graphischer Oberfläche KDE dem standardmässig installierten GNOME bevorzugt, kann dies mit folgendem Befehl nachinstalliert werden:

```
apt-get install  kubuntu-desktop
```

Um übers Netzwerk per ssh auf den Test-PC zugreifen zu können, haben wir noch den ssh-Server installiert:

```
apt-get install ssh
```

Dann brauchen wir noch einen Compiler und diverse Bilbliotheken, die in der Ubuntu-Desktop Standard-Installation nicht enthalten sind:

```
apt-get install  gcc  make  libc6-dev  libc-dev  libncurses5-dev
```

3.1.3 Downloads

Als nächstes werden die Quellen des Linux Kernels sowie von RTAI benötigt. Entscheidend ist bei den Kernel-Quellen, dass eine unterstützte Version gewählt wird. Welche Versionen unterstützt sind, zeigt sich nach dem Entpacken der RTAI-Quellen am Ende dieses Abschnitts im resultierenden Verzeichnis rtai-3.3/base/arch/i386/patches/ - dort sind bei RTAI-Version 3.3 folgende Kernel-Patches enthalten:

```
hal-linux-2.4.30-i386.patch
hal-linux-2.4.31-i386.patch
hal-linux-2.4.32-i386-1.1-01.patch
hal-linux-2.6.10-i386-r12.patch
hal-linux-2.6.11-i386-r12.patch
hal-linux-2.6.12-i386-r12.patch
hal-linux-2.6.13-i386-1.0-09.patch
hal-linux-2.6.14-i386-1.1-02.patch
hal-linux-2.6.15-i386-1.1-03.patch
hal-linux-2.6.15-i386-1.2-00.patch
```

Wir haben uns für den aktuellsten Patch für Kernel-Version 2.6.15(.7) entschieden.

Linux Kernel-Quellen Download & entpacken nach /usr/src/:

```
cd /root
wget ftp://ftp.de.kernel.org/pub/linux/kernel/v2.6/linux-2.6.15.7.tar.bz2
tar -xjf linux-2.6.15.7.tar.bz2 -C /usr/src/
```

RTAI Quellen Download & entpacken:

```
wget https://www.rtai.org/RTAI/rtai-3.3.tar.bz2
tar -xjf rtai-3.3.tar.bz2
```

3.1.4 Kernel patchen & compilieren

Kernel patchen, um RTAI-Erweiterung zu integrieren (RTAI-HAL):

```
cd /usr/src/linux/
patch -p1 </root/rtai-3.3/base/arch/i386/patches/hal-linux-2.6.15-i386-1.2-00.patch
```

Kernel konfigurieren:

```
make menuconfig
```

Folgendes sollte auf alle Fälle angepasst werden, bevor die Kernel-Konfiguration verlassen wird.

```
General setup  --->
      Kernel .config support & Enable access to .config through /proc/config.gz -> AN
Loadable module support  --->
      Module versioning support --> AUS
Processor type and features  --->
      Processor family --> je nach CPU (z.B. PentiumIII, sh. "cat /proc/cpuinfo")
Device Drivers  --->
      ATA/ATAPI/MFM/RLL support  --->
            Generic PCI IDE Chipset Support ->  AN <*>
            Intel PIIXn chipsets support ->  AN <*> (UND evtl andere - sh. "lspci")
File systems  --->
      Ext3     --> AN <*>  (NICHT <M>)
      Reiserfs --> AN <*>  (NICHT <M>)
```

Zusätzlich können unbenötigte Module/Treiber abgewählt werden, um Zeit beim compilieren des Kernels zu sparen. Mit allen bzw. den meisten Modulen aktiviert compiliert unser Test-PC mit 1GHz ca. 1,5 Stunden.

Die folgende Befehlsfolge stösst den Compillierungsvorgang an:

```
      make  &&  make install  &&  make modules_install
```

3.1.5 Bootloader anpassen

Nachdem der Kernel fertig ist, sollte unter `/boot` eine neue Datei mit dem Namen `vmlinuz-2.6.15.7` erscheinen.

```
root@rtai-pc:~/rtai-3.3 #  l /boot | grep vmlinuz
-rw-r--r--  1 root root 1,4M 2006-06-14 14:15 vmlinuz-2.6.15-25-386
-rw-r--r--  1 root root 1,4M 2006-07-06 14:17 vmlinuz-2.6.15.7
```

Diese muss dann mit folgendem neuen Eintrag in der Datei `/boot/grub/menu.lst` dem Bootloader bekannt gemacht werden:

```
title UBUNTU-Linux mit RTAI-Kernel
      root    (hd0,0)
      kernel  /boot/vmlinuz-2.6.15.7  root=/dev/hda1  ro  quiet  splash
```

Zudem sollte noch die Zeile mit dem Eintrag `hiddenmenu` entfernt werden, damit das Bootmenu auch angezeigt wird.

3.1.6 Kernel booten

Anschließend steht ein Neustart bevor, um den neuen Kernel auch zu verwenden. Beim Booten muss natürlich auch darauf geachtet werden, dass der neue Eintrag aus dem Bootmanager ausgewählt wird. Nachdem der Comupter hochgefahren ist, kann die Verwendung des neuen Kernels in einem Terminal / einer Konsole wie folgt nachgeprüft werden:

```
root@rtai-pc:~/rtai-3.3 #  uname -a
Linux rtai-pc 2.6.15.7 #4 Thu Jul 6 14:04:59 CEST 2006 i686 GNU/Linux
```

3.1.7 RTAI konfigurieren & compilieren

Nun muss noch RTAI konfiguriert und compiliert werden:

```
cd /root/rtai-3.3
```

```
make menuconfig  &&  make  &&  make install
```

Das erscheinende Menü kann getrost gleich wieder beendet werden, die Standardeinstellungen sind vollkommen ausreichend. Wenn alles erfolgreich verlief, sind alle RTAI-spezifischen Dateien unter `/usr/realtime` installiert.

Zu Guter Letzt müssen noch zwei Umgebungsvariablen angepasst werden, damit Kommandos und Bibliothken aus der neuen RTAI-Installation ohne komplette Pfadangabe gefunden werden:

```
echo "export PATH=$PATH:/usr/realtime/bin"  >>/etc/profile
echo "export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/realtime/lib"   >>/etc/profile
```

Nach dem nächsten Anmelden bzw. "sudo su" in einem Terminal sind dann die Umgebungsvariablen automatisch gesetzt.

3.1.8 Erste Tests

Bei der RTAI-Standard-Installation sind einige Tests im Verzeichnis `/usr/realtime/testsuite/` untergebracht, wobei die Unterscheidung zwischen Kernel- und Userspace auch hier durch verschiedene Unterverzeichnisse deutlich wird.

Um z.B. einen allgemeinen Latenz-Test durchzuführen, muss man nur in das entsprechende Verzeichnis gehen und das dort enthalten Bash-Script `run` ausführen:

```
cd /usr/realtime/testsuite/kern/latency
./run
```

Sollte dabei eine Fehlermeldung wie `Error opening /dev/rtf3` kommen, schafft folgende Kommando-Folge Abhilfe:

```
cd /root/rtai-3.3
make devices
cd -
```

3.1.9 Optionale Kalibrierung

Um die vorhandene Hardware möglichst gut auszunutzen, wird im RTAI-Handbuch eine Kalibrierung des Systems empfohlen. Dabei werden u.a. der Scheduler auf das genaue zeitliche Verhalten von CPU und APIC eingestellt und ausserdem die max. Latenz von Interrupts ermittelt.

Um so eine Kalibrierung durchzuführen, in das entsprechende Verzeichnis zu gehen und das dort enthalten Bash-Script `run` auszuführen:

```
cd /usr/realtime/calibration
./run
```

Die Kalibrierung ermittelt einige Werte, die beim dann später beim Laden der RTAI-Module mit angegeben werden können. Alternativ könnte auch der RTAI-Quellcode angepasst und neu compiliert werden.

3.1.10 Zwei einfachere Wege zum Ziel

Um zu einem RTAI-System zu gelangen gibt es aber auch einfachere Wege:

Zum einen haben wir im Zuge dieses Projekts eine CD zusammengestellt, mit der ein frisch installiertes Ubuntu Linux einfach zu einem RTAI-System mit Hilfe von zwei Bash-Scripts umgebaut werden kann.

Zum anderen hatten wir Kontakt zum Fraunhofer Institut, das im Auftrag des Bundesministeriums für Bildung und Forschung schon seit Anfang 2004 an der Verwendung

von RTAI für die Industrie forscht. Diese Forschungsgruppe hat kurz vor Ende der Projektarbeit eine LiveCD veröffentlicht (sh. Anlagen), mit der sich beinahe jeder PC mit Standard-Hardware in nur . ca. 5 minuten in ein RTAI-System verwandeln lässt – ohne auch nur irgendwas auf der Festplatte installieren zu müssen. Dazu ist es nur nötig von der CD zu booten und zu warten bis alles geladen ist.

3.2 Versuche

3.2.1 Versuch 1: Test harter Echtzeit

Um zunächst einmal den Unterschied im zeitlichen Verhalten zwischen einem Standard Linux-Prozess und einem RTAI-Echtzeittask zu verdeutlichen, soll im folgenden auf zwei Arten ein Rechtecksignal an den Datenpins des Parallelports erzeugt werden: zum einen mit einem normalen Linux-Prozess und zum anderen mit einem RTAI Kernel-Modul.

Zunächst der einfache, normale Linux-Prozess ohne Echtzeitverhalten:

```
// ==> versuch1/normalerprozess.c

#include <stdio.h>
#include <stdlib.h>
#include <asm/io.h>
#include <signal.h>

#define BASEPORT 0x378

static int end;

static void endme(int dummy) { end=1; }

int main(int argc, char **argv) {

        // STRG-C abfangen
        signal(SIGINT, endme);

        // ParPort IO-Rechte holen (3 Bytes des IO-Bereichs, 1=Zugriff erlauben)
        if (ioperm(BASEPORT, 3, 1)) {  perror("ioperm open");  exit(1);  }

        // ParPort in Ausgabemodus versetzen
        outb(0, BASEPORT + 2);

        // Datenpins toggeln
        while (!end) {
                usleep(1);
                outb(0, BASEPORT);
                usleep(1);
                outb(255, BASEPORT);
        }

        // ParPort IO-Rechte freigeben (3 Bytes des IO-Bereichs, 0=Zugriff sperren)
        if (ioperm(BASEPORT, 3, 0)) { perror("ioperm close"); }
        exit(0);
}
```

Zum compilieren reicht ein einfaches `gcc -o normalerprozess normalerprozess.c` und das Ergebnis kann direkt mit `./normalerprozess` gestartet werden.

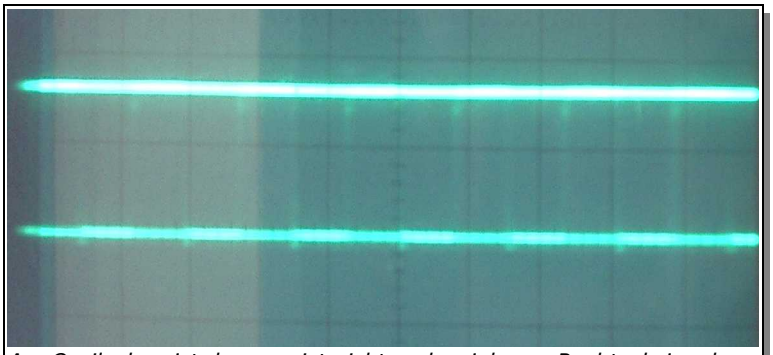
Schliesst man nun ein Oszilloscop an ein beliebiges Datenpin des Parallel-Ports, ist ein Rechtecksignal von ca. 125 Hz zu sehen (da usleep leider nicht kürzer warten kann). Wird nun zusätzlich der PC belastet (CPU, IO, ...) ist deutlich zu erkennen wie das Signal verzerrt und verschoben wird. Um für Test-Zwecke möglichst einfach eine hohe CPU-Belastung erzeugen zu

können, haben wir uns ein kleines Script erstellt:

```
#!/bin/bash
echo "Abbruch mit STRG-C ... "
until false ; do let a=99*234*3*13 ; done
```

Wenn dieses Script mit hoher Priorität z.B. -15 gestartet wird sorgt es mit seiner kleinen Rechenaufgabe für eine warme CPU und entsprechend weniger Zeit für andere Aufgaben – sodass beim Start das Rechecksignal am ParallelPort sogar bis zu mehreren Sekunden (!) ausfallen kann:

```
nice -n -15  cpu-mach-mal-was
```



Am Osziloskop ist dann meist nicht mehr viel vom Rechtecksignal zu sehen, wenn die CPU zu tun hat...

Nun zum RTAI-Modul mit Echtzeitverhalten:

```
// ==> versuch1/rtai-modul.c

#include <linux/module.h>
#include <rtai.h>
#include <rtai_sched.h>

#define TICK_PERIOD 50000
#define TASK_PRIORITY 1
#define STACK_SIZE 10000
#define BASEPORT 0x378

static RT_TASK rt_task;
static RTIME   tick_period;

static void toggle( int t ) {
        while (1) {
                outb(0, BASEPORT);
                rt_task_wait_period();
                outb(255, BASEPORT);
                rt_task_wait_period();
        }
}

int xinit_module(void) {
        outb(0, BASEPORT + 2);
        RTIME tick_period;
        rt_set_periodic_mode();
        rt_task_init(&rt_task, toggle, 1, STACK_SIZE, TASK_PRIORITY, 1, 0);
        tick_period = start_rt_timer(nano2count(TICK_PERIOD));
        rt_task_make_periodic(&rt_task, rt_get_time() + tick_period, tick_period);
```

```
        return 0;
}

void xcleanup_module(void) {
        stop_rt_timer();
        rt_task_delete(&rt_task);
        return;
}

module_init(xinit_module);
module_exit(xcleanup_module);
MODULE_LICENSE("GPL");
```

Dazu ist noch eine `Makefile` nötig:

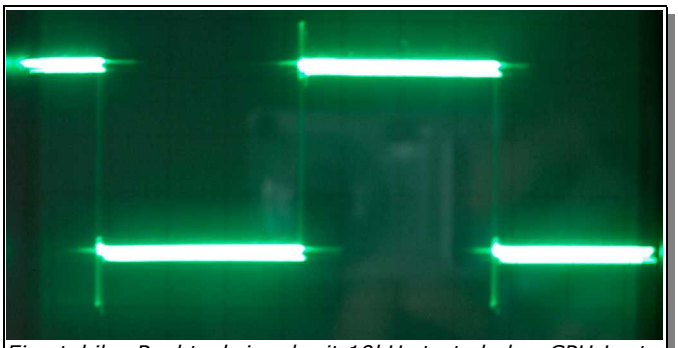
```
obj-m   := rtai-modul.o
KDIR    := /lib/modules/$(shell uname -r)/build
PWD     := $(shell pwd)
EXTRA_CFLAGS := -I/usr/realtime/include -I/usr/include/
default:
       $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```

Mit einem einfachen make-Aufruf wird der Code dann compiliert und liegt als Kernelmodul in der Datei `rtai-modul.ko` vor.

Doch bevor das Modul mit `insmod` geladen und aktiviert werden kann, müssen zuerst noch alle anderen RTAI-Module geladen werden deren Funktionalität benötigt wird.

```
moddir="/usr/realtime/modules"
insmod $moddir/rtai_hal.ko
insmod $moddir/rtai_ksched.ko
insmod ./rtai-modul.ko
```

Sollte noch ein Modul fehlen erscheint beim Laden des eigenen Moduls leider nur der irreführende Fehler `insmod: error inserting 'rtai-modul.ko': -1 Unknown symbol in module` .



Ein stabiles Rechtecksignal mit 10kHz trotz hoher CPU-Last

Am Oszilloskop ist nun eine – trotz starker IO- und CPU-Belastung - ein dauerhaft stabiles 10kHz-Rechtecksignal zu sehen.

Um den so erstellten Rechteckgenerator wieder zu stoppen, reicht es das "rtai-modul" zu entladen:

```
rmmod  rtai-modul
```

3.2.2 Versuch 2: Messung der Interrupt-Latenz

Um die Zeit zu Messen, die vom Auslösen eines Interrupts bis zu dessen Verarbeitung vergeht, haben wir ein Datenpin des Parallelports mit dem ACK-/Interrupt-Signal verbunden. Dadurch können wir per Software einen Interrupt auslösen, in dem wir an dem Datenpin kurz einen positiven Pegel anlegen. Dann muss nur noch je beim Auslösen und beim Verarbeiten ein Zeitstempel hinterlegt werden um anschliessend die beiden Werte vergleichen zu können. Die Differrenz entspricht der Latzenzzeit.

Hier das recht übersichtliche Kernel-Modul:

```
// ==> versuch2/rtai-modul.c

#include <linux/module.h>
#include <rtai.h>
#include <rtai_nam2num.h>
#include <rtai_sched.h>

#define BASEPORT 0x378
#define INTERRUPT 7

static int end;
static int startzeit;
static int endzeit;
static int zeitdiff;

// Interrupt-Handler, wird bei ausgelöstem Interrupt aufgerufen
static void irq_handler(void) {
  endzeit = rt_get_cpu_time_ns();
  zeitdiff  = endzeit  - startzeit;
  rt_printk("Interrupt empfangen. Latenz = %d ns\n", zeitdiff);
  rt_ack_irq(INTERRUPT);
}

// Erzeugt Interrupt und speichert Start-Zeit. Dazu muss ein beliebiges Datenpin
// (Pin 2-9) des ParPort mit dem ACK-Pin (Pin 10) verbunden sein.
static void generate_irq(void) {
  rt_printk("Generiere Interrupt ...\n");
  outb_p(0, BASEPORT);
  startzeit = rt_get_cpu_time_ns();
  outb_p(255, BASEPORT);
}

// INIT
int xinit_module(void) {
  int ret;
  end=0;
  rt_printk("ParPort Interrupt Latenz-Test gestartet...\n");
  ret = rt_request_global_irq(INTERRUPT, (void *)irq_handler);
  rt_enable_irq(INTERRUPT);
  outb_p(0x10, BASEPORT + 2); // IRQ im ParPort aktivieren

  generate_irq();
  return 0;
}

// EXIT
void xcleanup_module(void) {
  end=1;
  rt_disable_irq(INTERRUPT);
  rt_free_global_irq(INTERRUPT);
  rt_printk("ParPort Interrupt Latenz-Test beendet...\n");
```

```
}

////////////////////////////

module_init(xinit_module);
module_exit(xcleanup_module);
MODULE_LICENSE("GPL");
```

Die Echtzeit-Funktion `rt_printk()` schreibt über entsprechende Puffer/Fifos in die Datei `/var/log/messages`, die anschliessend recht einfach ausgewertet werden kann. Beim Laden und anschliessenden entladen des Moduls entstehen folgenden Meldungen:

```
Jul 10 21:26:34 rtai-pc kernel: ParPort Interrupt Latenz-Test gestartet...
Jul 10 21:26:34 rtai-pc kernel: Generiere Interrupt ...
Jul 10 21:26:34 rtai-pc kernel: Interrupt empfangen. Latenz = 10895 ns
Jul 10 21:26:34 rtai-pc kernel: ParPort Interrupt Latenz-Test beendet...
```

Um ein möglichst aussagekräftiges Ergebnis zu erhalten muss der Test oft wiederholt werden. Dazu muss das Modul immer wieder geladen/entladen werden. Dies erledigt hier ein kleiner Bash-Einzeiler, dem mit STRG-C wieder beendet werden kann:

```
until false ; do insmod rtai-modul.ko ; rmmod rtai-modul ; sleep 0.01 ; done
```

Schreiben der relevanten Daten (angesammelte Latenz-Werte) in eine eigene Datei:

```
cat /var/log/messages | grep "Interrupt empfangen." | cut -d\  -f10 | sort -n >liste
```

Mit einem kleinen Stück Perl-Code direkt an der Kommandozeile lässt sich recht einfach eine kleine Auswertung erstellen:

```
perl -e 'open( FH, liste); while (chomp($w=<FH>)){ $h{$w}++; } ; close FH; for $key
(keys %h) { print "$h{$key} \tX $key ns\n"; }' | sort -nr
```

Hier das gekürzte Ergebnis auf unserem Test-PC nachdem es ca. 20 min. unter Last lief. Die grösste gemessene Latenz betrug ca. 60 µs.

```
7075    X 10057 ns
6724    X 10895 ns
2715    X  9219 ns
2340    X 11733 ns
2259    X 10896 ns
1323    X 10058 ns
1275    X 11734 ns
239     X 12571 ns
204     X 12572 ns
184     X  9220 ns
110     X 13410 ns
23      X 14248 ns
14      X 14247 ns
4       X 49448 ns
3       X 36876 ns
3       X 20952 ns
2       X 54476 ns
2       X 22629 ns
2       X 21791 ns
1       X 60343 ns
1       X 57829 ns
1       X 36038 ns
1       X 20953 ns
```

3.2.3 Versuch 3: Interrupt im userspace

Zum Schluss noch ein Interrupt-Beispiel im Userspace mit LXRT – harte Echtzeit mit RTAI im Userspace.

Zunächst müssen wieder die benötigten Header eingebunden und globale Variablen angelegt werden:

```
// ===> versuch3/rtai-userspace.c

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

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

#define INTERRUPT 7
#define BASEPORT  0x378

static RT_TASK *maint, *handler;
static SEM *semaphore;
static int thread;
static volatile int maxcnt = 5;
static volatile int end = 1;
static volatile int ovr, intcnt;
```

Die Funktion irq_handler

```
static void *irq_handler(void *args) {
	if (!(handler = rt_task_init_schmod(nam2num("HANDLR"),0,0,0,SCHED_FIFO,0xF))) {
		printf("CANNOT INIT HANDLER TASK\n"); exit(1); }
	rt_allow_nonroot_hrt();
	mlockall(MCL_CURRENT | MCL_FUTURE);
	rt_make_hard_real_time();
	end=0;
	// IRQ aktivieren
	rt_request_irq_task(INTERRUPT, handler, RT_IRQ_TASK, 1);
	rt_startup_irq(INTERRUPT);
	rt_enable_irq(INTERRUPT);

	while ( !end && (ovr != RT_IRQ_TASK_ERR) ) {
		do {
			// Warte auf Interrupt:
			ovr = rt_irq_wait(INTERRUPT);
			if (ovr == RT_IRQ_TASK_ERR) break;
			if (end) break;
			if (ovr > 0) { rt_sem_signal(semaphore); }
			intcnt++;
			rt_sem_signal(semaphore); // main() benachrichtigen
			rt_ack_irq(INTERRUPT);
		} while (ovr > 0);
		rt_pend_linux_irq(INTERRUPT);
	}
	rt_release_irq_task(INTERRUPT);
	rt_make_soft_real_time();
	rt_task_delete(handler);
	intcnt = maxcnt;
	return 0;
```

```
}
```

In der main-Funktion wird nach einer Initialisierung nur noch auf die Semaphore gewartet, die bei einem Interrupt vom oberen Echtzeittask gesetzt wird.

```
int main(void) {
	// Haupt-Task erstellen
	if (!(maint = rt_task_init(nam2num("MAIN"), 1, 0, 0))) {
		printf("CANNOT INIT MAIN TASK\n"); exit(1);   }
	// Semaphore zum Benachrichtigen von main() wenn Interrupt auftritt erstellen
	if (!( semaphore  = rt_sem_init(nam2num("SEM"), 0))) {
		printf("CANNOT INIT SEMAPHORE\n"); exit(1); }
	// Berechtigung f. Zugriff auf ParPort aus Userspace holen
	if (iopl(3)) {
		printf("iopl err – keine rechte f. parport bekommen\n");
		rt_task_delete(maint); rt_sem_delete(semaphore); exit(1); }
	// ParPort in Interrupt-Modus versetzen & auf Ausgabe einstellen
	outb_p(0x10, BASEPORT + 2);
	// Thread erstellen
	thread = rt_thread_create(irq_handler, NULL, 10000);
	// warten bis Thread im harten Echtzeitmodus ist
	while (end) { usleep(100000); }
	// Bei Benachrichtigung Anzahl IRQ ausgeben - solange max. Anzahl nicht erreicht
	while (intcnt < maxcnt) {
		rt_sem_wait(semaphore); // WARTE auf Semaphore
		printf("Bisherige Anzahl IRQ: %d / Overrun: %d\n", intcnt, ovr);
	}
	end=1;
	// Finalen Interrupt erzeugen um rt_irq_wait zu befreien
	outb_p(0, BASEPORT); outb_p(255, BASEPORT); outb_p(0, BASEPORT);
	rt_release_irq_task(INTERRUPT);
	rt_thread_join(thread);
	rt_task_delete(maint);
	rt_sem_delete(semaphore);
	return 0;
}
```

Mit folgender `Makefile` im selben Verzeichnis kann der Versuch dann mit einem einfachen Aufruch von `make` compiliert werden:

```
prefix := $(shell rtai-config –prefix)
ifeq ($(prefix),)
$(error Please add <rtai-install>/bin to your PATH variable)
endif
CC = $(shell rtai-config –cc)
LXRT_CFLAGS = $(shell rtai-config –lxrt-cflags)
LXRT_LDFLAGS = $(shell rtai-config –lxrt-ldflags)
all: rtai-userspace
rtai-userspace: rtai-userspace.c
	$(CC) $(LXRT_CFLAGS) -o $@ $< $(LXRT_LDFLAGS)
```

Nun müssen noch – sofern noch nicht schon geschehen – noch die nötigen RTAI-Module geladen werden:

```
insmod /usr/realtime/modules/rtai_hal.ko
insmod /usr/realtime/modules/rtai_usi.ko
insmod /usr/realtime/modules/rtai_lxrt.ko
```

Anschliessend kann das Programm mit `./rtai-userspace` gestartet werden.

Allerdings gibt es noch keine Interrupt-Quelle, die am Parallelport-Pin 10 den Interrupt auslöst.

Dazu haben wir uns ein kleines Programm geschrieben, das die Datenpins des Parallelports kurz anschaltet und dann wieder abschaltet. Mit einem umgelöteten Druckerkabel haben wir durch Verbindung des Pin 2 (Datenpin) mit dem Pin 10 (ACK) dafür gesorgt, dass unser Programm einen Interrupt auslösen kann. Dazu reicht dann ein Aufruf von `./irq_generieren`.

```
// ===> versuch3/irq_generieren.c
// compilieren mit    gcc  -o irq_generieren  irq_generieren.c

#include <stdio.h>
#include <asm/io.h>

#define BASEPORT 0x378

int main (int argc, char **argv) {
      // 3 Bytes des IO-Port zur Verwendung freigeben
      if (ioperm (BASEPORT, 3, 1)) { perror ("ioperm open error"); exit (1); }

      // Ausgabe-Modus & Interupt im ParPort aktivieren
      outb (0x10, BASEPORT + 2);

      // Daten-Pins setzen / zuruecksetzen
      outb (0,  BASEPORT);
      outb (255,BASEPORT);
      outb (0,  BASEPORT);

      // 3 Bytes des IO-Port wieder sperren
      if (ioperm (BASEPORT, 3, 0)) { perror ("ioperm close error"); }
      exit (0);
}
```

Sofern der lxrt-Prozess noch läuft, sorgt jeder Aufruf dieses Programms auf einer eigenen Konsole dafür, dass der lxrt-Prozess eine Ausgabe in der Art wie

```
Bisherige Anzahl IRQ: 3 / Overrun: 0
```

macht.

4 Echzeitverhalten / Grenzen

Die Grenzen von RTAI und der Hardware lassen sich leicht mit den mitgelieferten Tools im Testsuite-Ordner `/usr/realtime/testsuite` ermitteln.

Es gibt drei unterschiedliche Tests – je für den Kernelspace und den Userspace (LXRT):

- latency (Latenzermittlung)
- switches (Dauer v. u.a. taskswitches)
- preempt (Jittermessung)

Im folgenden die Ausgaben aller drei Kernelspace-Tests auf unserem Test-System:

latency:

```
root@rtai-pc ~ # cd /usr/realtime/testsuite/kern/latency
root@rtai-pc /usr/realtime/testsuite/kern/latency # ./run
 [...]
RTAI Testsuite - KERNEL latency (all data in nanoseconds)
RTH|    lat min|    ovl min|    lat avg|     lat max|    ovl max|   overruns
RTD|          0|          0|       3431|       12571|       12571|          0
RTD|          0|          0|       3364|        8381|       12571|          0
RTD|          0|          0|       3361|        8381|       12571|          0
```

```
RTD|          0|          0|       3363|       8381|      12571|          0
RTD|          0|          0|       3363|       8381|      12571|          0
 [...]
```

switches:

```
root@rtai-pc ~ # cd /usr/realtime/testsuite/kern/switches/
root@rtai-pc /usr/realtime/testsuite/kern/switches # ./run
 [...]
RTAI[sched_lxrt]: Linux timer freq = 250 (Hz), CPU freq = 1193180 hz.
RTAI[sched_lxrt]: timer setup = 1676 ns, resched latency = 2514 ns.
Wait for it ...
FOR 10 TASKS: TIME 12 (ms), SUSP/RES SWITCHES 40000,
       SWITCH TIME (INCLUDING FULL FP SUPPORT) 302 (ns)
FOR 10 TASKS: TIME 13 (ms), SEM SIG/WAIT SWITCHES 40000,
       SWITCH TIME (INCLUDING FULL FP SUPPORT) 320 (ns)
```

preempt:

```
root@rtai-pc ~ # cd /usr/realtime/testsuite/kern/preempt/
root@rtai-pc /usr/realtime/testsuite/kern/preempt # ./run
 [...]
RTAI Testsuite - UP preempt (all data in nanoseconds)
RTH|     lat min|     lat avg|     lat max|    jit fast|    jit slow
RTD|        3352|        4844|        7543|       23695|       37406
RTD|        3352|        4834|        7543|       23695|       37406
RTD|        3352|        4795|        7543|       23695|       37406
RTD|        3352|        4599|        9219|       23695|       37406
RTD|        3352|        4603|       11733|       23695|       37406
RTD|        3352|        4592|       11733|       23695|       37406
RTD|        3352|        4593|       11733|       23695|       37406
RTD|        2514|        4613|       11733|       23695|       37406
RTD|        2514|        4591|       11733|       23695|       37406
```

5 Fazit

Im Gegensatz zu den kommerziellen Echtzeitsystemen ist RTAI und Linux Open Source, also kostenlos. Da wir in diesem Gebiet auf nur sehr wenig Erfahrung zurückgreifen konnten, hatten wir anfangs Startschwierigkeiten – u.a. bei der Wahl der richtigen Compilerversion.

Trotz alledem konnten wir einen Einblick in ein sehr leistungsstarkes System nehmen. Nicht ohne Grund wird das RTAI-Projekt indirekt durch das Bundesministerium für Bildung und Forschung über die "Forschungsoffensive Software Engineering 2006" gefördert. Im derzeitig geförderten Projekt handelt es sich um eine Aufgabe im Bereich Maschinen und Anlagenbau.

Zudem haben wir es wiedermal verstanden uns aus aussichtslosen Lagen durch Kooperation im Team und mit Ehrgeiz an Ziel unseres Projektes zu retten.

6 Quellen

6.1 Literatur

- Vorlesungsscript Echtzeitbetriebssysteme von Benno Gerum
- Linux Magazin 08/2006 Seiten 100f

6.2 Internet

- Ubuntu Linux: http://www.ubuntu.com
- RTAI-Projekt: http://www.aero.polimi.it/~rtai/
- RTL Open: http://www.rtlopen.de
- Captains Universe http://www.captain.at/rtai.php

7 Anlagen

- Ubuntu Linux Version 6.06 Installations-CDROM (Desktop-Variante)
- RTLopen **LiveCD** als Referenzplattform v.a. für den mittelständischen Maschinen- und Anlagenbau vom Fraunhofer Institut, gefördert durch das Bundesministerium für Bildung und Forschung. Einfach PC direkt von CD booten.
- Projekt-CDROM mit:
  - Projektarbeit (Dokumentation / Präsentation / Quellcodes)
  - zusätzlich aus dem Internet benötigte Ubuntu bzw. Debian Software-Pakete
  - Linux Kernel-Quellcode Version 2.6.15.7
  - RTAI-Quellen Version 3.3
  - einfache Installationsscripte (`prepare.sh` & `install.sh`).
    Kurzanleitung:
    - System booten, CD einlegen und mounten
    - In Mountpoint der CD wechseln, z.B. `cd /media/cdrom`
    - Scripte aufrufen mit `sh prepare.sh` bzw. sh `install.sh`