Echtzeit unter Linux mit RTAI

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.

Von Robert Schwebel

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 (*Bild 1*) 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.

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-

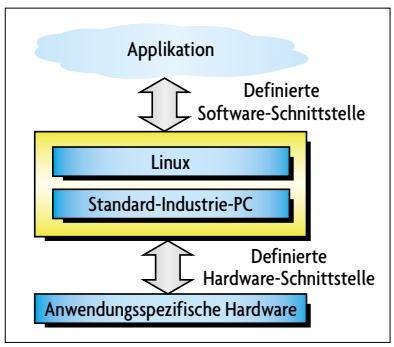

Bild 1. Hardware-Unabhängigkeit durch Linux. Durch Verwendung von offenen Schnittstellen bei Hardware (z.B. PC/104) und Software (Linux) wird die Kundenapplikation unabhängig von Prozessorgenerationen und Architekturen.

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

Bild 2. Beispiel für eine RTAI-Plattform mit wahlweise AMD-Elan-Prozessor (x86) oder PowerPC. (Bild: EuroDesign)

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" (*Bild 2*).

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 Ausfall überhaupt 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 (*Bild 3*). Bei der Analyse der zur Aufgabenlösung notwendigen Tools ist deshalb neben den typischen Zeiten auch die „Härte" der Anforderungen ein wichtiges Designkriterium.

Ereignisse, Latenz und Jitter

In einem Echtzeit-Betriebssystem muss sichergestellt sein, dass beim Eintreffen bestimmter „Ereignisse" benutzerdefinierte Programmteile ausgeführt werden. Solche Ereignisse sind:

- interne, zeitgesteuerte Unterbrechungssignale (Timer-Interrupts),
- externe Interrupts, ausgelöst von Peripherie-Komponenten.

Während bei Timer-Interrupts das Ziel ist, dass Echtzeit-Tasks in bestimmten Abständen zeitgesteuert ausgeführt werden, geht es bei externen Interrupts darum, asynchron zu den Zuständen im System auf Ereignisse zu reagieren, die über externe Leitungen von Peripherie-Komponenten gemeldet werden. Beiden Varianten ist jedoch gemeinsam, dass der entscheidene Parameter die Zeit zwischen Eintreten ei-

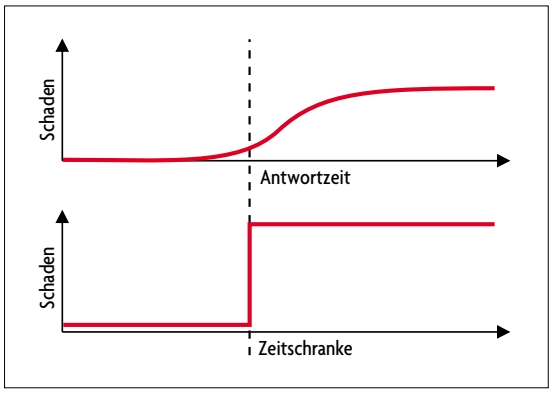

Bild 3. Schaden bei Überschreitung einer Deadline. Bei harten Echtzeit-Systemen tritt der Schaden abrupt ein, wohingegen bei weicher Echtzeit gelegentliche, geringfügige Überschreitungen kaum Auswirkungen haben.

nes „Ereignisses" (Ablaufen des internen Timers, Signal an Interrupt-Pin) und dem Start der Benutzerroutine ist. Diese Zeitspanne wird als „Latenzzeit" bezeichnet und ist ein wichtiges Designkriterium für Echtzeit-Systeme. In real existierenden Geräten mit einer gewissen Komplexität wird die Latenzzeit nicht in jedem Fall exakt gleich sein, sondern – insbesondere aufgrund

```
 1  #include <linux/module.h>
 2  #include <asm/io.h>
 3
 4  #include <rtai.h>
 5  #include <rtai_sched.h>
 6  #include <rtai_fifos.h>
 7
 8  #define TIMERTICKS 20000000   /* Timer-Tickrate in ns */
 9  #define MY_FIFO 0             /* FIFO Nr. 0           */
10
11  static RT_TASK My_Task;
12
13  static void My_Thread(int t)
14  {
15     /* Zeitstempel fuer Uebertragung in den User Space */
16     static struct { RTIME time; } msg;
17
18     static long long last_cpu_time;
19     static long long i;
20
21     while (1) {
22        i = rt_get_cpu_time_ns();
23        msg.time = i - last_cpu_time;
24        last_cpu_time = i;
25        rtf_put(MY_FIFO, &msg, sizeof(msg));
26        rt_task_wait_period();
27     }
28  }
29
30  int init_module(void)
31  {
32     RTIME tick_period; /* Periode Basistimer        */
33     RTIME now;         /* Timestamp fuer Task-Start */
34
35     /* Erzeuge FIFO */
36     rtf_create(MY_FIFO, 20000);
37
38     /* Erzeuge periodischen Task */
39     rt_task_init(&My_Task, My_Thread, 0, 2000, 0, 0, 0);
40
41     /* Timer laeuft im One Shot Mode */
42     rt_set_oneshot_mode();
43
44     /* Starte Timer und periodischen Task*/
45     tick_period = start_rt_timer(nano2count(TIMERTICKS));
46     now = rt_get_time();
47     rt_task_make_periodic(&My_Task, now + tick_period, tick_period);
48
49     return 0;
50  }
51
52  void cleanup_module(void)
53  {
54     stop_rt_timer();
55     rt_busy_sleep(10000000);
56     rtf_destroy(MY_FIFO);
57     rt_task_delete(&My_Task);
58  }
```

Listing 1. Das Beispielprogramm „Periodische Task“ läuft im Kernel-Space ab.

anderer Aktivitäten im System – gewissen Schwankungen unterworfen sein. Die statistische Schwankung der Latenzzeit wird auch als „Jitter“ bezeichnet und ist ein weiteres wichtiges Designkriterium. Entscheidend ist, dass die Latenzzeit trotz der Schwankungen vorgegebene Grenzen unter keinen Umständen überschreiten darf.

Entwickler von modernen Steuer- und Regelsystemen sehen sich mehr und mehr mit der Anforderung konfrontiert, neben den eigentlichen Echtzeit-Aufgaben auch andere, zeitunkritische Dinge erledigen zu müssen. So soll etwa ein Motorsteuergerät über eine Standard-Netzwerkschnittstelle konfiguriert oder ein Temperatur-Messsystem mit einer Druckerausgabe versehen werden.

Echtzeit und zeitunkritische Aufgaben

Es existieren derzeit zwei Ansätze, diese Anforderung zu erfüllen: Zum einen ist es möglich, ein Echtzeit-Betriebssystem Schritt für Schritt mit „artfremden“ Funktionen aufzurüsten. Der andere Ansatz besteht darin, Echtzeit-Funktionalität in ein konventionelles Betriebssystem im Nachhinein zu integrieren. Während in diesem Fall spezielle Teile des Kernels sich um die zeitkritischen Dinge kümmern, stehen wie gewohnt die „normalen“ Funktionen des Systems für zeitunkritische Aufgaben in vollem Umfang zur Verfügung. Zwischen den Echtzeit-Funktionen und konventionellen Programmen müssen Kommunikationsmechanismen geschaffen werden, mit deren Hilfe Daten die Grenze zwischen Echtzeit- und konventionellem Bereich passieren können.

Das Realtime Application Interface RTAI [1] wurde am „Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano“ von einem Team um Professor Paolo Mantegazza entwickelt, ursprünglich mit dem Gedanken, ein System für die am Institut anfallenden Echtzeit-Aufgaben aus dem Bereich der Luftfahrttechnik zur Verfügung zu haben. Das System wurde 1998 größtenteils unter der LGPL-Lizenz [2] als Freie Software im Internet zur Verfügung gestellt. Dies führte in den folgenden Jahren dazu, dass RTAI auch außerhalb der Entwicklergruppe am DIAPM von einer breiten Anzahl von Entwicklern weltweit eingesetzt und weiterentwickelt wurde und bis heute wird.

Die zentrale Kommunikation zwischen den Entwicklern findet, wie bei vielen Projekten dieser Art, über eine Mailingliste statt. Das „offizielle“ Code-Repository befindet sich nach wie vor in den Händen von Professor Mantegazza, der sich auch um die Einpflegung der von anderen Entwicklern stammenden Erweiterungen maßgeblich kümmert. Um die Kerngruppe in

Italien hat sich in der Zwischenzeit eine ganze Reihe weiterer, weltweit verteilter Core-Entwickler zusammengefunden. Dies wurde nicht zuletzt durch die sehr kooperative Bereitschaft der Kernentwicklergruppe zur Zusammenarbeit mit anderen Programmierern ermöglicht. In der heutigen Projektgruppe sind Entwickler sowohl aus dem universitären als auch dem industriellen Bereich vertreten.

Die Performance von RTAI liegt im Rahmen dessen, was andere PC-Echtzeit-Systeme ebenfalls bieten: ca. 4 µs Kontext-Switch, 20 µs Reaktionszeit auf einen Interrupt, 100 kHz für periodische Tasks und 30 kHz für One-Shot-Tasks. Diese Werte variieren je nach verwendeter Hardware-Plattform.

Das Echtzeit-Konzept von RTAI

Harte Echtzeit ist unter RTAI nur im so genannten Realtime-Space möglich. Echtzeit-Programme werden als Kernel-Module implementiert und sind somit in ihrer Funktionalität ebenso eingeschränkt wie gewöhnliche Kernel-Treiber. Insbesondere ist es nicht möglich, beliebige Bibliotheksfunktionen in Shared Libraries (dem Äquivalent zu DLLs unter Windows) aufzurufen.

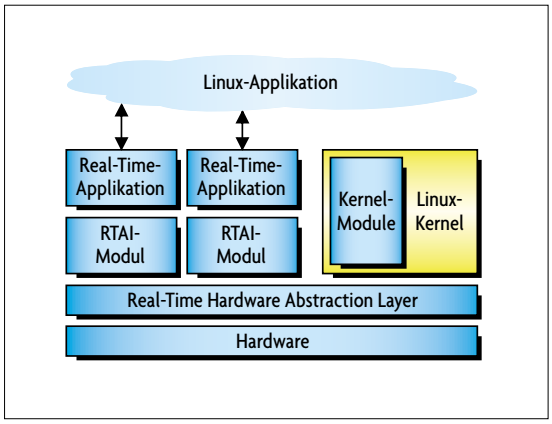

Bild 4. Aufbau von RTAI. Ein Real-Time Hardware Abstraction Layer fängt die Interrupts ab. Die Kommunikation zwischen Real-Time-Applikationen und User-Space findet über FIFOs oder Shared Memory statt.

Literaturtipps

Das Buch „Embedded Linux“ aus dem Verlag Moderne Industrie wendet sich an den Entwickler. Der Schwerpunkt des Inhalts liegt auf der Beschreibung der Linux-Architektur und auf der Einrichtung eines Entwicklungssystems zum Entwickeln von Embedded-Linux-Programmen. Der Autor beschreibt ausführlich den Boot-Vorgang sowie die Komponenten, die für einen Minimal-Kernel notwendig sind. Alle Vorgehensweisen sind mit Kommandozeilen bzw. Quelltexten dokumentiert, sodass die Beispiele anhand der Software auf der beiliegenden CD leicht nachvollzogen werden können. Ein eigenes Kapitel widmet sich der Echtzeit unter Linux und dem Echtzeit-Zusatz RTAI. Die Beispiele zur Interrupt-Behandlung, Task-Priorisierung und Interprozess-Kommunikation sind wesentlich ausführlicher, als dies im Rahmen dieses Zeitschriftenartikels möglich ist. Die Echtzeit-Erweiterung RT-Linux wird am Rande gestreift. „Embedded Linux“ bezieht sich nur auf die x86-Plattform. Echte Cross-Entwicklung sowie die Entwicklung für andere Prozessoren sind nicht Gegenstand dieses Buches.

Schwebel, Robert: Embedded Linux. Handbuch für Entwickler. MITP-Verlag, Bonn, 29,95 Euro.

Das reine Nachschlagewerk „Linux. Befehle, Begriffe, Referenz“ wendet sich laut Umschlagtext an Linux-Anwender, Anfänger sowie Umsteiger von anderen Betriebssystemen. Dass Begriffe wie RTAI oder RT-Linux nicht auftauchen, mag unter diesem Gesichtspunkt verständlich sein. Oft tauchen in Linux-Fachartikeln aber Begriffe auf, die für diese Zielgruppe unbekannt, aber wissenwert sein dürften. Was ist z.B. ein „Journaling File System“? – Aus dem Werk von Oliver Rosenbaum erfährt man das leider nicht. Dafür ist auf einer ganzen Seite ausführlich erklärt, was das Internet ist.

Auch das für Anwender und Umsteiger völlig irrelevante ISO/OSI-Schichtenmodell wird ausführlich dargestellt. Fazit: Die Suchmaschine Google ist eine preiswertere und ergiebigere Informationsquelle zu Linux-Suchbegriffen.

Rosenbaum, Oliver: Linux. Befehle, Begriffe, Referenz. VDE-Verlag, Berlin, 23,50 Euro.

Auch das Werk „Messen, Steuern, Regeln mit Linux“ konzentriert sich ausschließlich auf die x86-Architektur. Da der Autor ohne lange Umschweife auf die Besonderheiten von Embedded-Systemen zu sprechen kommt, ist das Buch frei von „Ballaststoffen“ in Form allgemeiner Erläuterungen zu Linux, zu denen es bereits genügend andere Veröffentlichungen gibt. Die ersten vier Kapitel beschäftigen sich mit der Hardware-Architektur der x86-Prozessoren, wobei ein Schwerpunkt auf dem AMD Elan SC 410 liegt sowie auf den Besonderheiten von Flash-Speicher. Die folgenden Linux-Kapitel zeigen die Bestandteile eines Minimal-Linux und die Vorgänge beim Booten – auch hier wieder mit besonderer Berücksichtigung von Flash-Speicher. Desweiteren schildert der Autor die Einrichtung eines Linux-Entwicklungssystems auf dem PC sowie die Programmierung des Embedded-Systems über die JTAG-Schnittstelle.
Im nächsten Schritt wird dem Leser gezeigt, wie Linux-Programme auf die Hardware von Ports und Speicherbereiche der x86-Architektur zugreifen können. Im Kapitel „Echtzeit“ beschäftigt sich der Autor mit der Echtzeit-Erweiterung RT-Linux. Eher anwendungsorientiert sind die Kapitel über Ethernet-Anbindung und TCP/IP-Einrichtung sowie Embedded-Webserver. Sämtliche Hardware-Beispiele des Buches beziehen sich auf den DIL/NetPC von SSV Embedded Systems. Insgesamt gibt das Buch auf über 300 Seiten eine fundierte Einführung in Linux für Embedded-x86-Systeme, wobei auch die Hardware-Aspekte nicht zu kurz kommen.

Walter, Klaus-Dieter: Messen, Steuern, Regeln mit Linux. Franzis-Verlag, Poing, 51,10 Euro.

Listing 2. Alle zeitunkritischen Aufgaben sind in das Programm „check.c“ ausgelagert, das im User-Space abläuft.

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <sched.h>
#include <signal.h>
#include <unistd.h>

static int end;

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

int main(void)
{
   int cmd0;
   struct sched_param mysched;
   struct { long long time; } msg;
   signal (SIGINT, endme);

   mysched.sched_priority = 99;

   if( sched_setscheduler( 0, SCHED_FIFO, &mysched ) == -1 ) {
      puts(„ ERROR IN SETTING UP THE SCHEDULER“);
      perror( „errno“ );
      exit( 0 );
    }

   if ((cmd0 = open(„/dev/rtf0“, O_RDONLY)) < 0) {
      fprintf(stderr, „Error opening /dev/rtf0\n“);
      exit(1);
   }
   while(!end) {
      read(cmd0, &msg, sizeof(msg));
      printf(„%f\n“, (float)msg.time/1000);
   }
   return 0;
}
```

Echtzeit-Anwendungen sollten unter Linux in der Regel so entwickelt werden, dass ein möglichst großer Anteil der Funktionalität, der nicht zeitkritisch ist, im User-Space als normales Linux-Programm implementiert ist. Bei den meisten Anwendungen ist dies relativ leicht möglich. Insbesondere Aktivitäten, die auf die Funktionalität des Betriebssystems (z.B. Massenspeicher, Parametrieren über Benutzerschnittstellen) angewiesen sind, benötigen in der Regel keine Echtzeit-Performance. Lediglich der tatsächlich zeitkritische Anteil wird im Realtime-Space als Kernel-Modul implementiert. Zwischen User- und Realtime-Space bietet RTAI eine ganze Reihe von Kommunikationsmöglichkeiten, mit deren Hilfe Daten ausgetauscht werden können (*Bild 4*).

Selbst bei aufwendigen Algorithmen, wie sie z.B. in CNC-Motorsteuersystemen vorkommen, ist es oft möglich, einen Großteil der Verarbeitung im User-Space abzuarbeiten. Dort können alle Schutzmechanismen des Betriebssystems (z.B. Speicherschutz verschiedener Programme untereinander und gegen das System) voll genutzt werden. Die Echtzeit-Module werden hingegen, ähnlich wie Gerätetreiber, direkt im Kernel-Space des Betriebssystems abgearbeitet und erfordern bei der Entwicklung besondere Sorgfalt.

Zyklische Tasks

Die Struktur eines RTAI-Programms wird im Folgenden anhand eines kurzen Beispiels verdeutlicht. Eine häufige Anwendung für ein Echtzeit-System besteht in der Ausführung periodischer Tasks. Eine solche Task könnte beispielsweise in einer Soft-SPS-Anwendung zum Einsatz kommen und dazu dienen, alle 20 ms eine Reihe digitaler und analoger Ein- und Ausgänge abzufragen oder zu setzen. In den *Listings* ist ein entsprechendes Beispielprogramm abedruckt.

Das Echtzeit-Programm rt_process.o (*Listing 1*) realisiert eine periodische Task (Zeile 13 – 28), in der jeweils direkt zu Beginn ein Zeitstempel ausgelesen wird (Zeile 22). Anschließend wird die Zeit seit dem letzten Aufruf der Tasks berechnet. Diese Differenz sollte im Idealfall die vorgegebenen 20 ms betragen. Der gemessene Wert wird in Zeile 25 in einen FIFO geschrieben, aus dem das Programm *check* (*Listing 2*) im User-Space den Wert lesen und weiterverarbeiten kann. Die Funktionen init_module() (Zeilen 30 – 50) und cleanup_module() (Zeile 52 – 58) dienen zum Initialisieren und De-Initialisieren des Kernel-Moduls bei dessen Start bzw. Beendigung.

Im User-Space erfolgt der Zugriff auf die Messdaten durch einfaches Öffnen des FIFO (Zeile 27 in Listing 2) und anschließendes Lesen (Zeile 32). Das Resultat der Messung ist in *Bild 5* grafisch dargestellt und zeigt, dass die Ab-

Die Lizenzfrage und das RT-Linux-Patent

Einige Verwirrung hat in den vergangenen Jahren die Tatsache gestiftet, dass Viktor Yodaiken, der Autor von RT-Linux, ein US-Software-Patent mit der Nummer 05995745 für das sowohl von RT-Linux als auch von RTAI verwendete Verfahren der verzögerten Interrupts (deferred interrupts) besitzt. Lange Zeit war nicht klar, was dies konkret für industrielle Anwender bedeutet – schließlich muss für eine breite Akzeptanz auch sichergestellt sein, dass proprietäre Applikationen möglich sind.
In einer gemeinsamen Anstrengung haben die Free Software Foundation [2] und die RTAI-Autoren Ende 2001 die bestehenden Unklarheiten beseitigt. Da der Kern von RTAI nun konform zur Patent-Lizenz und zur Lizenz des Kernels unter der GPL steht, ist die lizenzkostenfreie Nutzung von RTAI durch die Anwender sichergestellt. Die GPL fordert, dass der Anwender Änderungen an RTAI selbst wieder unter der GPL freigeben muss. Dies betrifft allerdings nicht die Applikationen in Form von Realtime-Modulen: Diese dürfen durchaus proprietär sein.
Eine konkrete Stellungnahme zu dieser Problematik von Eben Moglen, Rechtsprofessor an der Columbia Law School und General Counsel der Free Software Foundation liegt in [3] vor. Für Fragen im Zusammenhang mit deutschem Recht gibt das „Institut für Rechtsfragen der Freien und Open Source Software“ [4] Auskunft

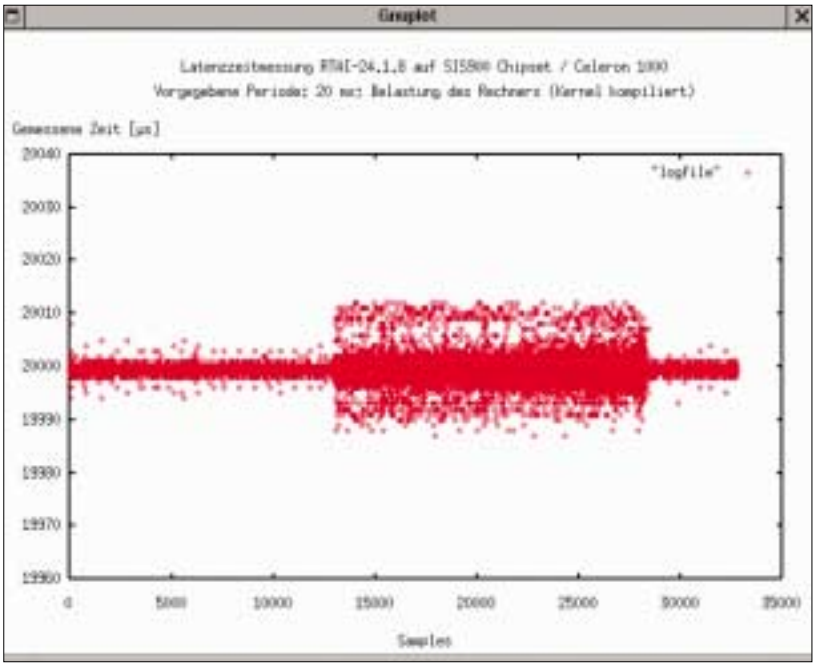

Bild 5. Plot der Zeitmessung an einer periodischen Task mit 20 ms Periodendauer. Die Abweichung ist nie größer als ±15 µs.

weichung nie größer als ±15 µs wird. Als Lasttest wurde während der Messung ein Kernel auf dem Testrechner übersetzt.

Ausblick: Echtzeit-Tasks im User-Space

Über bereits existierende Standard-Features wie periodische Tasks, externe Interrupts, Unterstützung für symmetrische Multiprozessorrechner, POSIX-API und Kommunikation über FIFOs, Mailboxen und Shared Memory hinaus ist das RTAI-Team derzeit dabei, eine Reihe interessanter Erweiterungen zu entwickeln. Eine der vielversprechendsten Neuheiten ist LXRT, mit dessen Hilfe harte Echtzeit-Tasks im User-Space laufen. Dies vereinfacht die Portierung von Software, die für andere Echtzeit-Betriebssysteme geschrieben wurde. Daneben sind ein Kompatibilitätsmodul zum MatLab/Simulink Realtime Workshop, C++-Support, Remote Procedure Call in Echtzeit und Support für die Echtzeit-Ethernet-Infrastruktur RTnet in Arbeit. *jk*

Literatur

[1] Realtime Application Interface RTAI: www.rtai.org
[2] Lesser General Public License, siehe Homepage der Free Software Foundation, www.fsf.org und www.gnu.org
[3] Informationen zur Patentfrage www.rtai.org/documentation/articles/patent.html
[4] Institut für Rechtsfragen der Freien- und Open-Source-Software www.ifross.de

Dipl.-Ing. Robert Schwebel
ist Inhaber der Firma Pengutronix (www.pengutronix.de) und beschäftigt sich mit der Entwicklung linuxbasierter Messtechnik für industrielle und wissenschaftliche Anwendungen.
E-Mail: r.schwebel@pengutronix.de