REALTIME LINUX IN DER SUB-NANO FORSCHUNG: DAS ECHTZEIT-MESS-SYSTEM ELISABET

Oliver Kühlert, Richard Schillinger,
Christian Bromberger und H. J. Jänsch
<*Oliver.Kuehlert@mpi-hd.mpg.de*>
http://www.mpi-hd.mpg.de/
http://www.physik.uni-marburg.de/
http://www.kuehlert.net/

Zusammenfassung

Die Arbeitsgruppe Oberflächenphysik der Philipps-Universität Marburg untersucht in Zusammenarbeit mit dem Max-Planck-Institut für Kernphysik in Heidelberg die Oberflächen von Festkörpern mittels β-Zerfallselektronendetektierter Kernspinresonanz (β-NMR). Im Mittelpunkt des Interesses stehen dabei lokale elektronische und strukturelle Eigenschaften von Halbleitern und Metallen. Die Experimente erfordern eine hohe Parallelität in der Experimentsteuerung (Temperatur der Oberfläche, Polarisation von Atomstrahlen, Magnetfelder etc.) und der Datenaufnahme („Zählen“, Detektieren von Zerfallselektronen).
Deshalb ist es von großem Interesse, derartige Parameter in Echtzeit mit hoher Präzision steuern zu können. Auch muss die Messwertaufnahme zu genau definierten Zeitpunkten in zur Steuerung parallelen Threads erfolgen. Zugriffe auf die Hardware (in diesem Fall einem CAMAC-Controller) müssen voreinander geschützt werden.
Der Inhalt des Vortrages besteht aus einer kleinen Einführung in die Kernel-Modulprogrammierung und wie durch deren Laden (insmod, modprobe) dem Kernel neue Funktionalität zur Laufzeit dynamisch hinzugelinkt wird. Anschließend werden dann die Funktionen unterster Ebene zum Ansprechen der Hardware im Detail vorgestellt, da für den erwähnten Controller keine Treiber existieren.
Es folgt eine kurze Definition von Echtzeit, bevor die RTLinux Erweiterung des Kernels und (auf Wunsch) dessen Installation vorgestellt wird. Am Beispiel unseres Mess-Systems werden dann die Programmiertechniken und Funktionen von RTLinux vorgestellt:

- Erzeugen, Starten und Stoppen von Threads.
- Schützen von paralleler Ausführung in kritischen Bereichen durch Mutex-locking.
- Erzeugen und Benutzen von FIFOs zur Kommunikation von RTLinux mit einem Hauptprogramm.
- Benutzung eines Handlers zur asynchronen Benachrichtigung von FIFOs.
- Suspendieren und Aufwecken von Threads.
- Zeiten und Uhren im Kernel.
- Die Thread-Ausführung anhalten und zu periodischen (oder einmaligen), genau festgelegten Zeitpunkten fortsetzen.

Die Kommunikation zwischen Hauptprogramm und RTLinux erfolgt über FIFOs. Zu deren Vereinfachung wurde eine C++ Klassenbibliothek, welche kompatibel zum Analyseprogramm „ROOT“ (Teil der CERN-library) ist, erstellt. Deren Aufbau (ebenfalls Threads benutzend) soll kurz skizziert werden.
Eine in unserem Projekt nicht benutzte Technik ist der mbuff shared-memory driver. Diese Methode soll zum Abschluss vorgestellt werden. Auf Wunsch kann noch kurz gezeigt werden, wie das anfangs besprochene CAMAC Modul in einen „echten“ Linux-Treiber verwandelt werden kann.

Kernelmodule

Linux unterscheidet streng zwischen dem so genannten *Kernel-* bzw. *User-Space*. Dabei werden im User-Space „normale“ Anwendungsprogramme, die kompiliert und gegen die entsprechenden libraries gelinkt worden sind, ausgeführt. Im Kernel-Space wird jedoch Code ausgeführt, der direkt auf die Hardware zugreift oder besondere Anforderungen hat, die Anwendungsprogrammen im User-Space nicht zugänglich sind. Dazu zählen z.B. Treiber oder auch RTLinux-Programme. Die Kommunikation zwischen Anwendungsprogrammen im User-Space und dem Code im Kernel-Space erfolgt dabei über bestimmte Schnittstellen, so genannte Gerätedateien, welche mit dem Befehl `mknod` erzeugt wurden und in der Regel im Verzeichnis `/dev` zu finden sind.

Code, der im Kernel-Space ausgeführt werden soll, muss direkt gegen den Kernel gelinkt werden. In frühen Versionen von Linux war dazu i.d.R. ein Neukompilieren des Kernels und nach entsprechenden Einträgen in den Bootloader ein Neustarten des Rechners nötig. Ein Treiber, der auf diese Weise dem Kernel hinzugefügt wird, nennt sich *monolithisch*. Ab Version 2.0.x ist es möglich, zusätzlich Objekt-Code während der Laufzeit gegen den Kernel zu linken. Derartig beschaffener Code nennt sich Kernel-Modul und muss überraschend wenigen Konventionen genügen. Kernel-Module werden i.d.R. in C geschrieben und können mit dem Gnu-C Compiler *gcc* erzeugt werden. Standardmäßig linkt gcc nach erfolgreicher Übersetzung den Objekt-Code gegen die Standard C-library. Dies wird durch den Aufruf mit dem Schalter `-c`, also z.B. `gcc -c hello.c` verhindert. Das Linken gegen den Kernel während dessen Laufzeit erfolgt dann mittels `insmod hello.o`.

Der Quellcode eines einfachen Kernel-Moduls kann so aussehen: (hello.c)

```
#define MODULE

#include <linux/module.h>

int init_module(void)
{
  printk("<1>Hallo Linux-Tag !");
}

void cleanup_module(void)
{
  printk("<1>Tschüß !");
}
```

Die Definition des Symbols `MODULE` ist notwendig, ebenso die beiden Funktionen `init_module` und `cleanup_module`. Dabei wird `init_module` beim Aufruf von `insmod` ausgeführt. Das Entfernen eines Moduls ist möglich, wenn sichergestellt ist, dass der Kernel es nicht mehr benötigt. Dies geschieht mit dem Befehl `rmmod hello`, entsprechend wird dann die Funktion `cleanup_module` des Kernel-Moduls aufgerufen.

Die `printk` Aufrufe schreiben den folgenden Text zunächst in einen Buffer, bis ein entsprechender Log-Dämon sie weiter bearbeitet. Die Zahl in den <>-Klammern bezeichnet dabei den *Loglevel*. Darauf soll nicht näher eingegangen werden, i.d.R. kann man die Ausgaben durch den Befehl `tail -f /var/log/messages` sehen. Natürlich will man außer dem Ausgeben von Text auch Funktionen des Kernels oder von anderen Modulen aufrufen. Da ja direkt gegen den Kernel gelinkt wird, sind entsprechende libraries nicht notwendig, wohl aber die Header-Dateien mit den Prototypen der Funktionen bzw. globalen Variablen-Deklarationen. Die Header-Dateien des Kernels sind in `/usr/include/linux` und `/usr/include/asm` zu finden. Um ein versehentliches Einbinden in Anwendungsprogramme zu verhindern, ist Kernel-Code dort durch ein `#ifdef __KERNEL__` geschützt. Deshalb muss im Kernel-Modul dann das Symbol `__KERNEL__` definiert werden. (`#define __KERNEL__`)

Weitere Funktionen oder Variablen, die man seinem eigenen Kernel-Modul hinzufügt, sind nach dem Laden mit `insmod` im Kernel bekannt und können auch im Kernel-Space beliebig aufgerufen werden. Der Kernel führt dazu eine Symboltabelle, die mit `less /proc/ksyms` ausgelesen werden kann. Dabei ist der Programmierer des Kernel-Moduls selbst dafür verantwortlich, seine öffentlichen Funktionen oder Variablen z.B. mit Präfixen so zu benennen, dass es keine Kollisionen mit anderen in der Tabelle befindlichen Symbolen gibt. Alternativ kann er sie durch das Schlüsselwort `static` als nicht öffentlich deklarieren, um auf diese Weise die so genannte „Namespace Pollution“ so gering wie möglich zu halten. (Außer der Verwendung von `static` gibt es auch weitere Möglichkeiten, auf die hier nicht eingegangen werden soll.)

Da das Mess-System ELISABET zur Steuerung der Experimente auf ein CAMAC-System zurückgreift, besteht die unterste Ebene aus einem Modul zum Zugriff auf die Hardware eines CAMAC-Controllers. Deshalb soll im folgenden kurz ein CAMAC-System vorgestellt werden.

CAMAC

Computer **A**utomated **M**easurement **A**nd **C**ontrol (CAMAC) ist ein System zur Datenaufnahme bzw. Experimentsteuerung und wurde hauptsächlich für Anwendungen in der Hochenergiephysik vom ESONE Comittee of European Laboratories entwickelt. Inzwischen wird es zwar immer mehr von VME-Systemen verdrängt, es ist aber trotzdem ein extrem verlässliches und robustes System. CAMAC-Crates und Module sind außerdem in einer Vielzahl an physikalischen Instituten verfügbar, sodass damit Mess-Systeme kostengünstig aufgebaut werden können.

Ein CAMAC-System besteht zunächst aus einem so genannten Crate, das ist im wesentlichen ein Rahmen, in den bis zu 24 Module eingeschoben werden können. Bei den Modulen handelt es sich um die anzusteuernden bzw. auszulesenden Geräte, wie z.B. Digital-Analog-Konverter (DAC) oder Zähler, etc. Die Module sind im Crate untereinander und mit dem Crate-Controller durch Bus-Systeme verbunden. (Adress-Bus, 24-bit Schreib- und Lesebus) Jede Einschub-Position am Rahmen hat eine eindeutige Nummer (1-24), über diese kann das jeweilige eingeschobene Modul adressiert werden. (Station-Number N). $N = 24$ steht dabei immer für den Crate-Controller. Ein Modul kann bis zu 16 Unteradressen haben ($A = 0 \ldots 15$), und für jede dieser Unteradressen existiert ein (oder auch mehrere) 24-bit Register, welche lesend oder schreibend manipuliert werden können. Um letztendlich ein solches Register anzusprechen, benötigt man noch einen Function-Code F. Dabei werden die Function-Codes $0 \ldots 15$ zum Auslesen eingesetzt, d.h. es wird ein 24-bit-Wert vom Gerät zum Controller übertragen und die Codes $15 \ldots 31$ bei einem schreibenden Zugriff. Die genaue Bedeutung der Function-Codes ist vom jeweiligen Modul abhängig. Zusammenfassend kann man sagen, jede Modulfunktion lässt sich über die Adressierung *NAF* und einen 24-bit Wert (lesend oder schreibend) ansprechen.

camac.c

CAMAC-Controller können mit vielen unterschiedlichen Computer-Typen verbunden werden. In unserem Fall handelt es sich natürlich um einen Linux-PC mit einer entsprechenden Steckkarte. Der Controller (6002) und die PC-Karte (PC004) sind von dsp Technology Inc. Leider konnten keine fertigen Treiber gefunden werden. Treiber für CAMAC-Systeme anderer Hersteller finden sich z.B. hier [1].

Die Steckkarte ist eine ISA-Modell und belegt 16 IO-Ports des Rechners. Der Bereich beginnt standardmäßig bei 0240h, dies kann aber durch Setzen von DIP-Schaltern geändert werden. Tabelle 1 zeigt einige wichtige Register.

Register	Beschreibung
0...2	24-bit Wert für CAMAC-write
3	Subaddress A
4	Function-Code F
5	Station-Number
7	Start dataway-cycle
9.....11	24-bit Wert für CAMAC-read

Tabelle 1: Wichtige Register der Steckkarte PC004.

Das folgende Kernel-Modul beinhaltet die wichtigsten Funktionen zum Ansprechen der Hardware. Beispielhaft sind die Funktionen `camac_read` und `camac_write` komplett dargestellt. Die Funktionen `camac_setinhibit` bzw. `camac_clearinhibit` setzen das so genannte inhibit-Flag für das ganze Crate. Ein gesetztes inhibit-Flag hat z.B. auf Zählmodule die Auswirkung, dass ein an den Eingängen befindliches Signal ignoriert und nicht gezählt wird.

```
#include <linux/module.h>
#include <linux/ioport.h>
#include <asm/io.h>
#include "camac.h"

int camac_port=0x240;
MODULE_PARM(camac_port,"i");

void camac_write(byte N, byte A, byte F,
                 D24WORD data){
  outb(N,camac_port+5); // load N
  outb(F,camac_port+4); // load F
  outb(A,camac_port+3); // load A

  outb( data & 0x0000ff,     camac_port+2); //low
  outb((data & 0x00ff00)>>8 ,camac_port+1); //middle
  outb((data & 0xff0000)>>16,camac_port+0); //high

  outb(100,camac_port+7); //start camac-cycle
}

void camac_read(byte N, byte A, byte F,
                D24WORD * data){
  D24WORD low,middle,high;
  outb(N,camac_port+5); // load N
  outb(F,camac_port+4); // load F
  outb(A,camac_port+3); // load A

  outb(100,camac_port+7); //start camac-cycle

  low=inb(camac_port+11);
  middle=inb(camac_port+10);
  high=inb(camac_port+9);

  *data=low | (middle<<8) | (high<<16);
}

void camac_setinhibit(){
    ... }

void camac_clearinhibit(){
    ... }

int init_module(void) {
  int back;
  back=check_region(camac_port,16);
         //checking if ports free
  if (back<0) return back; //ioports nicht allozierbar

  request_region(camac_port,16,"camac");

  ...  // Initialisieren des CAMAC-Crates

  return 0;
}

void cleanup_module(void) {
  release_region(camac_port,16);
       //ports wieder freigeben
}
```

Zunächst wird die globale Variable `camac_port` mit dem Wert 0x240 initialisiert, da standardmäßig der IO-Bereich ab 0240h beginnt. Die nächste Zeile definiert

diese Variable als Modul-Parameter, d.h. wird beim Laden des Moduls mit `insmod` dieser Parameter angegeben, kann der Wert überschrieben werden. Also `insmod camac.o camac_port=0x200`, wenn auf der ISA-Karte ein IO-Bereich von 0200h eingestellt wurde.

Die Funktion `init_module` dient nun zum Registrieren der IO-Bereiche für den Kernel und dem Initialisieren der Hardware. Der Linux-Kernel verwaltet die benutzten IO-Ports und stellt sicher, dass keine Überschneidungen auftreten können. Welche Ports von welchem Modul benutzt werden, lässt sich von der Kommandozeile mit `less /proc/ioports` anzeigen. Dazu sollte das Modul mit `check_region` zunächst feststellen, ob der gewünschte Bereich überhaupt frei ist und im Fehlerfall `init_module` mit der entsprechenden Fehlernummer abbrechen. Ist der Bereich verfügbar, kann er anschließend unter Angabe des Modulnamens mit `request_region` reserviert werden. Die Initialisierung der Hardware ist aus Platzgründen weggelassen worden.

In der Funktion `cleanup_module` wird lediglich der benutzte IO-Bereich mit `release_region` wieder freigegeben. Man beachte, dass es sich bei dem Modul `camac.c` nicht um einen kompletten Treiber handelt, da es keinerlei Schnittstellen zum User-Space gibt, auf die Anwendungsprogramme zugreifen könnten. Es existieren nach dem Laden des Moduls lediglich im Kernel vier neue Funktionen, die von anderen Modulen benutzt werden können. In unserem Fall handelt es sich dabei natürlich um RTLinux-Module.

Echtzeit

Ein Betriebsystem bezeichnet man als echtzeitfähig, wenn es innerhalb eines garantierten Zeitraumes auf eine Anforderung reagieren kann. Dabei wird die zwischen der Anforderung und der Reaktion des Systems verstreichende Zeit als Latenzzeit bezeichnet. Bei so genannter „weicher“ Echtzeit ist diese Zeit um einen, meist angegebenen Mittelwert verteilt. Es gibt aber keine obere Schranke und es ist auch möglich, dass es gar keine Reaktion geben kann. Solche Systeme eignen sich nicht für viele Steuerungs- oder Messaufgaben. Bei „harter“ Echtzeit gibt es eine maximale Latenzzeit, innerhalb der das System reagiert haben muss. Für RTLinux ist eine maximale Latenzzeit von 15μs angegeben.

Installation von RTLinux

Der normale Linux-Kernel ist von sich aus nicht echtzeitfähig, es sind daher einige nicht unerhebliche Veränderungen, hauptsächlich am Scheduler, notwendig. Dies geschieht durch entsprechende Patches. In der aktuellen Version (RTLinux 3.1) werden die Kernel 2.4.4 und 2.2.19 unterstützt. RTLinux kann man hier [2] herunterladen und den gewünschten Kernel bekommt man hier [3].

Da die Dokumentation inzwischen eine sehr gute Installationsanleitung enthält, soll der Vorgang hier nur kurz skizziert werden:

1. Ein Verzeichnis `/usr/src/rtlinux` erzeugen.
2. Dort die beiden Pakete auspacken, man erhält zwei Unterverzeichnisse: `rtlinux-3.1` und `linux`.
3. Den Kernel patchen:
   ```
   cd linux
   patch -p1 < ../rtlinux-3.1/kernel_patch-2.4.4
   ```
4. Den Kernel ganz normal konfigurieren (`make menuconfig`), installieren und damit den Rechner neu starten.
5. In `/usr/src/rtlinux/rtlinux-3.1` einen symbolischen Link auf das neue Kernel-Verzeichnis erstellen. Die RTLinux-Erweiterungen werden ebenfalls mit `make menuconfig` konfiguriert und mit `make` und `make devices` übersetzt.

Nach der Installation können die RTLinux-Module mit dem Script `scripts/insrtl` geladen und somit RTLinux gestartet werden. Zum Test empfiehlt es sich, die Beispielprogramme im `examples`-Ordner auszuprobieren.

Threads

Im normalen, ungepatchten Linux unterbricht der Scheduler niemals laufenden Kernel-Code. Alle Kernel- und Modulfunktionen werden in einem einzigen Thread, dem Kernel-Thread, ausgeführt. Daher sollten Kernel-Funktionen keine langen Schleifen bearbeiten und möglichst schnell zurückkehren, oder sich schlafen legen oder explizit den Scheduler aufrufen. Die Zeit, die vergeht, bis der Scheduler wieder die Kontrolle bekommt, ist jedoch unbestimmt und somit das System für Echtzeit untauglich.

RTLinux löst dieses Problem, indem auf unterster Ebene Threads einführt werden. RTLinux-Threads können unterschiedliche Prioritäten haben und ein Thread einer höheren Priorität kann jederzeit einen mit einer niedrigeren unterbrechen. Dem Kernel-Thread und somit auch allen Anwendungen im User-Space ist dabei die niedrigste Priorität zugeteilt, er kann also jederzeit durch RTLinux-Threads unterbrochen werden. Er ist auch dadurch ausgezeichnet, dass bestimmte Operationen, wie z.B. das Erzeugen neuer Threads, nur während seiner Ausführung möglich sind. Im Folgenden sollen RTLinux-Threads

einfach als Threads bezeichnet werden. Soll auf die Besonderheiten des Kernel-Threads eingegangen werden, werde ich explizit diese Bezeichnung verwenden.

Wie tief unten im System der neue Scheduler angesiedelt ist, zeigte sich in einer frühen Phase des Projektes ELISABET. Die Kontrolle über laufende Messungen, insbesondere das Stoppen, war noch nicht besonders ausgereift. Versehentlich wurde der Mess-Rechner während einer laufenden Messung heruntergefahren. Die Messung lief jedoch weiter (dies konnte an dem Blinken von Kontroll- Leuchtdioden an einem CAMAC-Testeinschub beobachtet werden) und konnte nur durch Ausschalten des Rechners gestoppt werden. Im Grunde handelt es sich um einen kleinen Kernel, der dem Linux-Kernel vorgeschaltet wird.

Neue Threads dürfen nur durch den Kernel-Thread erzeugt werden (z.B. in `init_module`). Dies geschieht durch die `pthread_create` Funktion:

```
#include <pthread.h>
int pthread_create(pthread_t * thread,
                   pthread_attr_t * attr,
                   void *(*start_routine)(void *),
                   void * arg);
```

Die ID des Threads wird in der Variablen, auf die der Zeiger `thread` zeigt, gespeichert. Die Variable muss nicht initialisiert werden, es muss nur sichergestellt sein, dass sie existiert und der Zeiger darauf zeigt. `attr` zeigt auf eine Struktur, die vorher mit den Attributen des Threads gefüllt werden muss. Man kann auch `NULL` angeben, es werden dann Default-Attribute verwendet. `start_routine` ist ein Zeiger auf die Start-Funktion des Threads. Sie muss als Parameterliste ein `void *` erwarten und gibt auch ein `void *` zurück.

Ein Thread kann gelöscht werden durch:

```
#include <pthread.h>
int pthread_delete_np(pthread_t thread);
```

`thread` bezeichnet die dabei oben eingeführte Thread-Variable, in der die entsprechende ID gespeichert ist.

Wurde beim Erzeugen des Threads in dem `attr`-Argument nicht dessen Priorität gesetzt, kann dies im Thread selbst nachgeholt werden. Dies geschieht durch:

```
#include <pthread.h>
int pthread_setschedparam(pthread_t thread,
                          int policy,
           const struct sched_param *param);
```

`thread` bezeichnet wieder die Thread-Variable, auf sie kann vom Thread selbst auch einfach mit `pthread_self()` zugegriffen werden. Das Argument `policy` wird zur Zeit noch nicht von RTLinux ausgewertet, aus Kompatibilitätsgründen zu späteren Versionen sollte jedoch `SCHED_FIFO` verwendet werden. `param` ist ein Zeiger auf die Struktur `sched_param`. Bei der der oben erwähnten policy muss hier lediglich die gewünschte Priorität gesetzt werden:

```
struct sched_param p;
p.sched_priority = 1;
pthread_setschedparam(pthread_self(),
                      SCHED_FIFO,&p);
```

setzt die Priorität des Threads, in welchem der Aufruf gemacht wird, auf 1. (Je höher die Priorität, desto höher auch diese Zahl.) Bei gleicher Priorität ist unbestimmt, welcher Thread bevorzugt wird.

Mutex-Locking

Die durch RTLinux gewonnene Parallelität der Ausführung im Kernel kann zu erheblichen Problemen, insbesondere bei Zugriffen auf die Hardware, führen. Betrachten wir den Fall, dass zwei laufende Threads das oben beschriebene CAMAC-Modul benutzen. Derjenige mit der niedrigeren Priorität führt z.B. gerade einen Zugriff auf die Hardware in `camac_write` aus, hat dort gerade die Adress- und Datenregister beschrieben und steht unmittelbar vor der Ausführung des CAMAC-cycles, als er von dem Thread mit höherer Priorität unterbrochen wird. Dieser tut nun alles, was er so tun möchte, u.a. auch einen Zugriff mittels `camac_write`, wobei er natürlich auch die entsprechenden Register verändert. Irgendwann gibt er die Ausführung an den Thread niedriger Priorität wieder ab und dieser führt den CAMAC-cycle durch. Leider stehen in den Registern inzwischen die falschen Werte...

Offensichtlich benötigt man einen Mechanismus, um bestimmte kritische Bereiche im Code vor paralleler Ausführung zu schützen. Dafür gibt es so genannte Mutex-Objekte (mutual-exclusive). Zu Beginn eines kritischen Bereiches im Code wird der dazu gehörende Mutex gesperrt und dahinter wieder entsperrt. Betrachtet man obiges Szenario wieder, so hat der Thread niedriger Priorität beim Beginn des Beschreibens der Register den Mutex gesperrt. Er wird nun wieder vom anderen Thread unterbrochen. Dieser tut nun alle notwendigen Dinge und kommt irgendwann an die Stelle, an der er für den Zugriff auf die Register selbst den Mutex sperren möchte. Bei dem Versuch wird er sofort geblockt und der Scheduler gibt die Kontrolle wieder an den anderen Thread ab. Dieser führt seinen kritischen Bereich aus und bekommt nach dem Entsperren des Mutex sofort die Kontrolle wieder abgenommen. Der Thread mit der höheren Priorität kann nun seinen kritischen Bereich ausführen.

Bei der Verwendung von Mutex-Objekten sollte man vor allem darauf achten, dass der kritische Bereich wirklich

nur sehr kurz ist und so gut wie keine Rechenzeit beansprucht. Schließlich kann hier ein Thread niedriger Priorität einen mit höherer aufhalten! Ein schönes Analogon ist ein langsames Auto, das kurz vor einem schnelleren in einen Tunnel fährt, in dem Überholen nicht möglich ist. Man sollte sich also überlegen, wie man den Tunnel möglichst kurz baut. Vielleicht ist es auch möglich, aus einem langen Tunnel zwei kurze zu konstruieren, zwischen denen ein Überholen möglich ist? Gerade bei Schleifen lohnt sich die Überlegung, ob es nicht doch möglich ist, pro Durchlauf den Mutex zu sperren und wieder zu entsperren, statt dies vor bzw. nach dem kompletten Schleifenkörper zu tun.

Um nicht vor jedem Aufruf der CAMAC-Funktionen das entsprechende Mutex-Objekt zu sperren und danach zu entsperren, haben wir die Mutex-Kontrolle direkt in die CAMAC-Befehle eingebaut. Das Modul `camac.c` wurde entsprechend angepasst. Ähnlich wie bei den Threads gibt es für jedes Mutex-Objekt eine Mutex-Variable. Sie ist vom Typ `pthread_mutex_t`. Vor der Benutzung muss der Mutex initialisiert werden. Dies geschieht natürlich in `init_module` mit der Funktion:

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

`attr` bezeichnet wieder entsprechende Attribute, i.d.R. reichen die Default-Werte, die man durch Angabe von `NULL` einstellt. `pthread_mutex_destroy` ist das entsprechende Gegenstück, das in `cleanup_module` aufgerufen wird.

Die Funktionen zum Sperren bzw. Entsperren des kritischen Bereiches lauten:

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Vor jeden Hardwarezugriff in `camac.c` wurde der lock- und danach der unlock-Befehl gesetzt, z.B. in `camac_write`:

```
void camac_write(byte N, byte A,
           byte F,D24WORD data)
{
  pthread_mutex_lock(&camac_mutex);

    outb(N,camac_port+5); // load N
    outb(F,camac_port+4); // load F
    outb(A,camac_port+3); // load A

    outb( data&0x0000ff,     camac_port+2);
    outb((data&0x00ff00)>>8 ,camac_port+1);
    outb((data&0xff0000)>>16,camac_port+0);
    outb(100,camac_port+7); //start camac-cycle

  pthread_mutex_unlock(&camac_mutex);
}
```

Da es auch zwischen den unterschiedlichen CAMAC-Funktionen keine Überschneidungen geben darf, wurde für alle kritischen Bereiche derselbe Mutex benutzt, d.h. ein Thread, der `camac_write` aufruft, wird blockiert, wenn sich ein anderer in einem kritischen Bereich von `camac_read` befindet.

FIFOS

Für die Kommunikation der RTLinux-Threads mit Anwendungsprogrammen stehen 64 FIFO-Buffer (**F**irst-**I**n-**F**irst-**O**ut) zur Verfügung. Sie sind uni-direktional, zum bi-direktionalen Datenaustausch benötigt man also deren zwei. Vor ihrer Benutzung müssen sie erzeugt und später wieder gelöscht werden. Dies ist wieder nur im Kernel-Thread erlaubt.

```
#include <rtl_fifo.h>
int rtf_create(unsigned int fifo, int size);
int rtf_destroy(unsigned int fifo);
```

`fifo` bezeichnet dabei die Nummer des FIFOs (0-63) und `size` dessen Größe. Bei der Wahl der Größe sollte berücksichtigt werden, wieviel Daten geschickt werden sollen und wie oft das Anwendungsprogramm Rechenzeit bekommt. Solange der Thread nämlich, ohne zwischendurch die Kontrolle über die Ausführung abzugeben, Daten in den FIFO schreibt, wächst der Buffer, weil die Anwendung keine Zeit bekommt, diese abzuholen. Zum Schreiben bzw. Lesen in die FIFOS existieren die Funktionen

```
#include <rtl_fifo.h>
int rtf_put(unsigned int fifo, char * buf,
            int count);
int rtf_get(unsigned int fifo, char * buf,
            int count);
```

`buf` bezeichnet einen Speicherbereich aus dem in den FIFO geschrieben bzw. in den gelesen wird und `count` die Anzahl der Bytes. Dem Rückgabewert entspricht die Anzahl der tatsächlich in/aus dem FIFO transferierten Bytes, bzw. ein negativer Wert im Fehlerfall. Beide Funktionen kehren in jedem Fall zurück, blockieren also nicht.

Auf der Anwendungsseite erscheinen die FIFOS als Gerätedateien (`/dev/rtf0`...`/dev/rtf63`) und können ganz normal mit `open` geöffnet bzw. `close` geschlossen werden. Das Schreiben bzw. Lesen geschieht über die entsprechenden `write` und `read` Funktionen:

```
... // FIFO öffen zum Schreiben:
int fdo=open("/dev/rtf1",O_WRONLY|O_NONBLOCK)
... write(fdo,buf,anzahl);
... close(fdo);
... // FIFO öffen zum Lesen:
int fdi=open("/dev/rtf0",O_RDONLY|O_NONBLOCK)
... read(fdi,buf,anzahl);
... close(fdi); ...
```

In vielen Fällen möchte man, dass eine bestimmte Aktion in den RTLinux-Threads (Kernel-Space) auf das Beschreiben der FIFOS vom Anwendungsprogramm (User-Space) erfolgt. Denkbar ist z.B. ein Kommando mit folgendem Datenblock, das vom RTLinux-Programm über den FIFO entgegengenommen, interpretiert und entsprechend ausgeführt wird. Eine Methode um festzustellen, ob die Anwendung etwas geschickt hat, besteht sicher darin, den FIFO in regelmäßigen Zeitabständen abzufragen (Polling).

Eine wesentlich elegantere Methode (die auch mit der Rechenzeit sparsamer umgeht) ist die Installation eines Handlers. In diesem Fall bewirkt das Beschreiben eines FIFOs vom Anwendungsprogramm her den Aufruf einer vorher bestimmten Funktion im Kernel-Space. Die Priorität ist dabei zunächst die niedrigste (Kernel-Thread), der Handler kann dann aber einen Thread in der gewünschten Priorität erzeugen oder einen bereits erzeugten aufwecken. (Mehr dazu im nächsten Abschnitt.) In der Regel wird der Handler aber zunächst nachschauen, was für Daten (Kommandos) im FIFO stehen und dann entsprechende Schritte einleiten. Die Funktion zur Installation eines Handler sollte ebenfalls nur vom Kernel-Thread aus benutzt werden.

```
#include <rtl_fifo.h>
int rtf_create_handler(unsigned int fifo,
     int (* handler)(unsigned int fifo));
```

`fifo` steht für den FIFO, dessen Beschreiben zum Aufrufen des Handlers `handler` führt. Beim Löschen des FIFOs mittels `rtf_destroy` wird auch der zugehörige Handler deinstalliert.

Suspendieren und Aufwecken von Threads

Für einen neu erzeugten Thread gibt es nicht immer sofort etwas zu tun oder er wartet nach getaner Arbeit auf die nächste Aufgabe. In solchen Fällen kann man ihn schlafen legen. Dies geschieht durch:

```
#include <rtl_sched.h>
int pthread_suspend_np(pthread_t thread);
```

In den meisten Fällen führt er das Kommando selbst aus, also wird man `pthread_suspend_np(pthread_self());` aufrufen. Wieder aufwecken kann ihn z.B. der oben besprochene Handler:

```
#include <rtl_sched.h>
int pthread_wakeup_np(pthread_t thread);
```

Zeiten und Uhren im RTLinux-Kernel

```
#include <rtl_time.h>
hrtime_t clock_gethrtime(clockid_t clock);
```

liefert die Zeit der gewählten Uhr `clock`. Die Zeiteinheit ist dabei ns und die Auflösung von der Hardware abhängig, aber besser als $1\mu s$. Tabelle 2 zeigt eine Übersicht über die in RTLinux verfügbaren Uhren.

clockid_t	Beschreibung
CLOCK_REALTIME	Die Standard POSIX realtime-Uhr. Sie liefert die seit der Epoche vergangene Zeit und kann mit mit `clock_settime` gestellt werden.
CLOCK_MONOTONIC	Diese Uhr läuft stetig weiter und wird niemals gestellt oder auf Null gesetzt.
CLOCK_RTL_SCHED	Die vom Scheduler benutzte Uhr. Sie liefert auch die für `pthread_make_periodic_np` maßgebliche Zeit.

Tabelle 2: Verfügbare Uhren in RTLinux

Threads blockieren und zu bestimmten Zeiten periodisch oder einmalig weiterlaufen lassen

Bei Mess-Systemen möchte man in vielen Fällen eine bestimmte Größe in Abhängigkeit der Zeit aufnehmen. Dabei sind meist feste Zeitabstände gewünscht. Dies lässt sich zum Beispiel bewerkstelligen, indem man zwischen dem Einlesen der Datenpunkte eine feste Zeitspanne wartet. Diese Methode hat jedoch einen Nachteil: Die wirkliche Wartezeit ist kleinen Schwankungen unterlegen. Außerdem ist der absolute Zeitpunkt, an dem der Wert tatsächlich aufgenommen wird, zusätzlich leicht gegen den Soll-Zeitpunkt verschoben, weil Code ausgeführt werden muss, um die Hardware anzusprechen. Am Anfang mag dieser Effekt sehr klein sein, da aber das Warten immer relativ zum aktuellen Zeitpunkt ist, addieren sich diese Fehler im Laufe der Zeit.

Daher ist es wünschenswert, den Thread nicht eine bestimmte Zeitspanne schlafen zu lassen, sondern eher zu ruhen, bis ein bestimmter **absoluter** Zeitpunkt erreicht ist. In einem ersten Ansatz könnte man die Uhr laufend

abfragen, bis die gewünschte Zeit erreicht ist. Das erinnert an Polling und ist nicht besonders elegant. Wesentlich besser wäre es, wenn der Thread einen Zeitpunkt in der Zukunft bestimmt, an dem er vom Scheduler aufgeweckt werden möchte und sich dann schlafen legt. Die ganze Vorrede wäre natürlich sinnlos, wenn RTLinux nicht für diese Zwecke entsprechende Befehle hätte:

```
#include <rtl_sched.h>
int pthread_make_periodic_np(pthread_t thread,
                             hrtime_t start_time,
                             hrtime_t period);
int pthread_wait_np(void);
```

Mit `pthread_make_periodic_np` wird dem Scheduler mitgeteilt, dass er den Thread `thread` zum Zeitpunkt `start_time` aufwecken soll. Ist `period` $\neq 0$, so wird der Scheduler dies nicht nur einmal, sondern periodisch zu den absoluten Zeitpunkten `start_time` + $n \times$ `period`, $n = 0 \ldots \infty$ tun. In der Regel bestimmt der Thread dies selbst, d.h. für das Argument `thread` nimmt man meist `pthread_self()`. Anschließend legt der Thread sich schlafen mit `pthread_wait_np`.

. . . und in der Praxis: TDS

Die **T**hermische-**D**esorptions-**S**pektroskopie ist eine in der Oberflächenphysik sehr wichtige und einfach zu verstehende Analysemethode. Sie dient der Bestimmung der Anzahl, Art und Bindungsstärke von Teilchen, die eine Festkörperoberfläche bedecken. Dabei befindet sich der zu untersuchende Kristall in einer UHV-Kammer (**U**ltra-**H**igh-**V**acuum). In unserem Fall handelt es sich dabei um Ruthenium- oder Silizium-Festkörper.

Die Aufnahme von TDS-Spektren besteht im Wesentlichen darin, die Temperatur der Oberfläche mit einer konstanten Rate (bei uns in der Größenordnung $10\,^{\circ}\mathrm{C}$ pro Sekunde) zu erhöhen und währenddessen die die Oberfläche verlassenden (desorbierenden) Teilchen zu zählen. Das Zählen der Teilchen wird getrennt für verschiedene Teilchensorten mit einem Quadrupol-Massenspektrometer (QMS) durchgeführt. Die Trennung erfolgt über die Teilchenmasse, von der man mehr oder weniger eindeutig auf das entsprechende Atom bzw. Molekül rückschließen kann. Das QMS kann dabei zwischen den gewünschten Massen so schnell umschalten, dass sich die Temperatur in dieser Zeit nicht wesentlich ändert. Trägt man nun die Desorptionsraten der verschiedenen Teilchensorten gegen die Temperatur auf, so erhält man Kurven, aus denen man wichtige Informationen über die Bedeckung ziehen kann.

Zur Durchführung eines TDS-Experimentes benötigt man also zwei parallel verlaufende Prozesse:

1. Die Temperatur des Kristalls muss mit einer konstanten Rate erhöht werden. Wir benutzen dabei einen externen Regler. Dieser verlangt eine Spannung, die der gewünschten Soll-Temperatur proportional ist. Diese wird durch einen DAC (**D**igital-**A**nalog-**C**onverter) des CAMAC-Crates erzeugt. Ein RTLinux-Thread muss also periodisch mit hoher zeitlicher Präzision das entsprechende Register im CAMAC-System um einen bestimmten Wert erhöhen.

2. Parallel dazu erfolgt die eigentliche Datenaufnahme. Ein entsprechender Thread muss also regelmäßig aufwachen, die gewünschten Massen am QMS einstellen, für jede dieser Massen die Desorptionsrate messen und diese über einen FIFO an das Anwendungsprogramm schicken, das die Daten speichert oder gleich anzeigt. Zusätzlich wird die zu diesem Zeitpunkt herrschende Ist-Temperatur gemessen. Da im Zweifelsfall die Messung wichtiger ist, hat dieser Prozess eine höhere Priorität gegenüber Prozess 1.

Beginnen wir mit dem ersten Prozess. Im Mess-System ELISABET unterscheiden wir zwischen Experimenten und Kommandos. Da eine Temperatur-Rampe eher universellen Charakter hat und auch bei der Präparation der Oberflächen in verschiedenen Kombinationen mit anderen Maßnahmen auftreten kann, befindet sie sich im „Kommando-Modul“ `rt_command.c`. Genau genommen handelt es sich um eine Spannungsrampe und kann somit auch für ganz andere Dinge eingesetzt werden.

Die `init_module`-Routine des Kommandomoduls erzeugt zunächst den FIFO 3, um Kommandos zu empfangen und FIFO 2, um dem Anwendungsprogramm zu antworten. Für den FIFO 3 wird ein Handler installiert. Anschließend wird ein Thread erzeugt, der später die Rampe fahren soll. Dieser legt sich jedoch sofort nach dem Setzen seiner Priorität mit `pthread_suspend_np` zur Ruhe, da noch nichts zu tun ist.

Schreibt nun irgendwann die Linux-Anwendung in den FIFO 3, wird sofort im Kernel-Thread der Handler aufgerufen. Dieser nimmt das Kommando inklusive Parameter mit `rtf_get` entgegen, interpretiert es und schreibt die Parameter in eine globale Struktur. Auf die Details des Protokolls soll nicht eingegangen werden. Manche Kommandos, wie z.B. das einmalige Setzen eines Registers im CAMAC-Crate kann der Kernel-Thread-Handler sofort erledigen. Bei einer Rampe ist er jedoch auf den wartenden RTLinux-Thread angewiesen. Also schickt er diesem ein `thread_wakeup_np`. Der RT-Thread macht sich sofort an die Arbeit. Die Parameter findet er in der oben erwähnten globalen Struktur (`theramp`).

```
delta_t=theramp.delta_t*1000000; // ms -> ns
start_t=clock_gethrtime(CLOCK_RTL_SCHED)+delta_t;
for (dac=theramp.startdac;
     (dac<=theramp.enddac) && (ramp_running);
      dac+=theramp.step){
```

```
  pthread_make_periodic_np (pthread_self(),
                            start_t,0);
  pthread_wait_np ();

  camac_write(theramp.N, theramp.A,16,dac);

  start_t+=delta_t;
}
```

delta_t bezeichnet die Zeit, die zwischen zwei Punkten vergeht, an denen die Temperatur gesetzt wird und wird vom Anwendungsprogramm in ms übergeben. Da RTLinux jedoch mit ns arbeitet, wird sie entsprechend umgerechnet. Die Startzeit start_t berechnet sich relativ zur aktuellen Zeit. In der Schleife wird der Sollwert in Schritten von step beginnend bei startdac bis enddac hochgezählt und die Startzeit jeweils durch Addition von delta_t aktualisiert. Die Schleife kann auch durch Rücksetzen des Flags ramp_running vorzeitig abgebrochen werden. Dies geschieht wieder durch den Kommando-Handler, ausgelöst durch einen entsprechenden Befehl des Anwendungsprogramms. Die Befehle in ihrem Inneren bestehen darin, die Zeit der Fortführung des Threads festzulegen, den dazugehörigen Wartebefehl zu aktivieren und schließlich den entsprechenden CAMAC-Befehl auszuführen.

Der zweite Thread ist im Modul rt_task.c, wo sich der Code für die einzelnen Experimente befindet. Die Datenübergabe ist ähnlich wie oben und es werden die FIFOS 0 und 1 benutzt, darauf soll aber ebenfalls nicht weiter eingegangen werden. Während die Soll-Temperatur oben einfach über das Setzen eines DACs erfolgte, geschieht das Einlesen nicht etwa über einen ADC (Analog-Digital-Konverter), sondern über eine Frequenzmessung. Dies liegt daran, dass in unserem Labor auch Experimente mit hoher radioaktiver Strahlung durchgeführt werden und es vom Kontrollraum durch Betonmauern getrennt ist. Daher ist das CAMAC-Crate über lange — durch Kabelbäume geführte — BNC-Kabel mit dem QMS verbunden. Die Signale können auf diesem Weg stark verrauscht werden. Um dem zu begegnen, könnte man einfach mehrmals messen und den Mittelwert bilden. Eleganter ist es jedoch, das Ausgangssignal des QMS von einer Spannung in eine Frequenz zu wandeln (0-10 V entsprechen bei uns 100 kHz-1100 kHz) und diese über das Kabel zu schicken. Auf der anderen Seite muss dann entsprechend eine Frequenz gemessen werden. Wir haben uns nach einigen Tests entschieden, auf eine weitere Elektronik zu verzichten und stattdessen einen CAMAC-Zähler, gesteuert von RTLinux, zu verwenden und als Zeitbasis die PC-interne Uhr zu nehmen. Das ist sicher keine Präzisionsmessung, aber weit besser als das Rauschen des QMS-Signals, schon **vor** dem BNC-Kabel.

Der folgende Code beginnt wieder kurz nachdem er vom Handler aufgeweckt worden ist. Die nötigen Parameter sind in der Struktur exp_block gespeichert.

```
for (zaehler=0;
    (zaehler<datapoints) && (!rtf_get(1,buf,1));
    zaehler++)
{
  pthread_make_periodic_np(
                    pthread_self(),start_t,0);
  pthread_wait_np ();

  switch (experiment_type)
  {
    case EXP_TDS:
     tpd_time=start_t;
     for (i=0;i<exp_block.tpd_number;i++)
     {
       camac_write(exp_block.tpd_setmassdac_N,
          exp_block.tpd_setmassdac_A,16,
          exp_block.tpd_mass_information[i].dac);
                                //Masse ans QMS

       switch(
         exp_block.tpd_mass_information[i].range)
       {
         // Hier werden je nach gewünschter
         // Vorverstärkung entsprechende
         // Kommandos ans das QMS geschickt.
       }


       switch(
         exp_block.tpd_mass_information[i].gain)
       {
         // Hier werden je nach gewünschter
         // Nachverstärkung entsprechende
         // Kommandos ans das QMS geschickt.
       }

       tpd_scan_t=
         exp_block.tpd_mass_information[i].
           scan_t*1000000; // ms->ns


       pthread_make_periodic_np(pthread_self(),
          tpd_time+tpd_wait_t,0);//Warten,
       pthread_wait_np ();    //Bis QMS bereit

       camac_read(exp_block.tpd_qmssignal_N,
         exp_block.tpd_qmssignal_A,9,&camac_in);
           //Zähler löschen
       camac_clearinhibit(); //Zählen erlauben
       scan_start=clock_gethrtime(CLOCK_RTL_SCHED);

       pthread_make_periodic_np(pthread_self(),
         tpd_time+tpd_wait_t+tpd_scan_t,0);
           //scan Zeit warten
       pthread_wait_np();

       camac_setinhibit(); //Zählen stoppen
       camac_read(exp_block.tpd_qmssignal_N,
        exp_block.tpd_qmssignal_A,0,&camac_in);
          //Zähler lesen
       scan_stop=clock_gethrtime(CLOCK_RTL_SCHED);

       tpd_time+=(tpd_wait_t+tpd_scan_t); //Zeit
         //für neuen Schleifendurchlauf anpassen

       time_diff=scan_stop-scan_start;
       rtf_put(0,&camac_in,sizeof(camac_in));
       rtf_put(0,&time_diff,sizeof(time_diff));

     }  //end of Teilchen-Massen-Schleife

     camac_read(exp_block.tpd_temp_in_N,
       exp_block.tpd_temp_in_A,9,&camac_in);
         //Zähler löschen
     camac_clearinhibit(); //Zählen erlauben
     scan_start=clock_gethrtime(CLOCK_RTL_SCHED);
```

```
    pthread_make_periodic_np(pthread_self(),
      tpd_time+tpd_temp_scan_t,0);
       //scan Zeit warten
    pthread_wait_np();


    camac_setinhibit();  //Zählen stoppen
    camac_read(exp_block.tpd_temp_in_N,
      exp_block.tpd_temp_in_A,0,&camac_in);
        //Zähler lesen
    scan_stop=clock_gethrtime(CLOCK_RTL_SCHED);

    time_diff=scan_stop-scan_start;
    rtf_put(0,&camac_in,sizeof(camac_in));
    rtf_put(0,&time_diff,sizeof(time_diff));
    break;  // TDS

    //TDS ENDE

    default:
      rtl_printf("experiment not defined !!!");

  }// Ende switch-Befehl
  start_t +=delta_t;
} //Ende äußere for-Schleife
```

Betrachten wir zunächst die äußere for-Schleife. Sie wird `datapoints`-mal durch laufen. Zusätzlich wird abgebrochen, wenn sich Zeichen im FIFO 0 befinden. So kann man ein Experiment schnell beenden, indem man irgendeinen String an `/dev/rtf0` schickt, selbst wenn das Anwendungsprogramm abgestürzt ist. Das Innere der Schleife ist genauso aufgebaut, wie das vorhergehende Beispiel. Statt des `camac_write` Befehls steht hier eine größere switch-Konstruktion. Diese unterscheidet zwischen den verschiedenen Experimenttypen. Allen ist nämlich die selbe äußere Schleife gemeinsam, die Unterscheidung findet erst hier statt und zwar anhand der Variablen `experiment_type`. Um Platz zu sparen, wird hier nur der Code für ein TDS gezeigt.

Innerhalb des TDS-spezifischen Codes soll der Thread ebenfalls gehalten und zu definierten Zeitpunkten wieder gestartet werden. Dies soll ebenfalls nicht relativ zur aktuellen Zeit, sondern zu absoluten Zeitpunkten geschehen. Der letzte absolute Zeitpunkt steht noch in der Variablen `start_t`, sie darf nicht verändert werden, da sie für die äußere Schleife gebraucht wird, also wird eine Kopie in `tpd_time` angelegt. Es folgt eine Schleife, die alle gewünschten Teilchen-Massen durchgeht. Die Anzahl steht in dem Feld `tpd_number`. Dort wird zunächst mittels `camac_write` die gewünschte Masse am QMS eingestellt. Die zwei folgenden switch-Konstruktionen sind etwas länger und nicht dargestellt, sie stellen die Vor- und Nachverstärkung für die entsprechende Masse ein und sind für das weitere Verständnis nicht von Bedeutung. Danach ist das QMS nicht sofort bereit, es braucht einige ms, bis es auf die neuen Einstellungen reagiert hat. Dies wird durch die Wartezeit `tpd_wait_t` berücksichtigt. Der Thread stellt sich den Scheduler zum Wecken auf `tpd_time+tpd_wait_t` und legt sich zur Ruhe. Nach dem Aufwachen beginnt endlich die Frequenzmessung des gewandelten Signals vom QMS. Dazu wird mittels eines `camac_write` an den gewünschten Zähler dieser gelöscht (Function-Code $F = 9$) und durch ein `camac_clearinhibit` das Zählen gestartet. Der tatsächliche Zeitpunkt des Starten des Zählers wird in der Variablen `scan_start` gespeichert. Während des Zählens kann sich der Thread wieder schlafen legen. Die Dauer beträgt `tpd_scan_t`. Danach wird der Zähler gestoppt (`camac_setinhibit`), mit `camac_read` in die Variable `camac_in` ausgelesen und die aktuelle Zeit in `scan_stop` geschrieben. Für die nächste Masse im folgenden Schleifendurchlauf wird die Zeit-Variable `tpd_time` auf den aktuellen Stand gebracht.

Die für diese Masse gemessene Desorptionsrate soll nun an das Anwendungsprogramm übertragen werden. Dazu reicht es eigentlich, den Wert von `camac_in` mit `rtf_put` über den FIFO zu schicken. Es hat sich aber gezeigt, dass `time_diff=scan_stop-scan_start` sich leicht von dem Soll-Wert `tpd_scan_t` unterscheidet und dabei auch Schwankungen (eben der Latenzzeit von RTLinux) unterlegen ist. Messungen mit einem Frequenzgenerator haben ergeben, dass man diese Schwankungen verringern kann, indem man die Frequenz durch `camac_in/time_diff` statt `camac_in/tpd_scan_t` ausdrückt.

Eine weitere Einschränkung von RTLinux ist, dass man möglichst in Threads keine Gleitkomma-Berechnungen ausführen sollte. Prinzipiell ist es zwar möglich, mittels `pthread_attr_setfp_np` einem Thread die Berechtigung zu solchen Operationen zu geben, dies verschlechtert aber die Performance von RTLinux. Standardmäßig ist es daher auch abgeschaltet.

Deshalb wird die Frequenz nicht ausgerechnet, sondern einfach das Wertepaar an das Anwendungsprogramm übertragen.

Damit ist die Messung der Desorptionsrate für alle Massen abgeschlossen. Es folgt noch die Messung der Temperatur. Dies geschieht nach dem gleichen Schema mit der Ausnahme, dass die Wartezeit `tpd_wait_t` entfällt.

Bei der Wahl von `delta_t` für die äußere Schleife ist zu berücksichtigen, dass diese Zeit mindestens so lang ist, wie ein Durchlauf für ein TDS dauert, und somit

$$\begin{aligned} \text{delta\_t} &> \text{tpd\_temp\_scan\_t} \\ &+ \text{tpd\_number} \times \text{tpd\_wait\_t} \\ &+ \sum_{m=1}^{tpd\_number} \text{tpd\_scan\_t}_m \end{aligned}$$

Ein C++ API

Das Mess-System ELISABET besteht im Wesentlichen aus drei Teilen:

- Der Echtzeit-Komponente und dem CAMAC-Modul.
- Ansteuerung von Hardware, die über andere Schnittstellen (u.a. RS-232) angesprochen wird und nicht in einer Echtzeit-Umgebung benötigt wird.
- Einem Anwendungsprogramm und einigen zusätzlichen Tools, die die Datenaufnahme steuern und die Ergebnisse grafisch darstellen bzw. auswerten können. Dazu wird die library des hervorragenden Daten-Analyse-Systems *ROOT* [4] benutzt.

Das Projekt wird von ebenso vielen Personen durchgeführt. Gerade der letzte Teil stellt einen enormen Arbeitsaufwand dar. Um die Arbeit an dieser Komponente etwas zu erleichtern, wurde eine Schnittstelle zu dem CAMAC-Modul und den Echtzeit-Funktionen geschrieben. Dadurch können auch Änderungen an dem Protokoll, welches über die die FIFOs betrieben wird, durchgeführt werden, ohne dass Änderungen an dem Anwendungsprogramm notwendig sind.

Die Klassendefinition für den Kommando-Teil sieht folgendermaßen aus:

```
class commandModule
{
public:
  static void SetDAC(byte N, byte A, byte F,
                     D24WORD dac);
  ...
  static void doRamp(byte N, byte A,
                     D24WORD startdac,
                     D24WORD enddac,
                     D24WORD step,
                     unsigned long long delta_t,
                     int hold=0);

  static void stopRamp();
};
```

Alle Funktionen sind statisch, werden also z.B. durch `commandModule::SetDAC...` aufgerufen. Aus Platzgründen sind nicht alle Methoden dargestellt.

Für jedes Experiment wurde eine eigene Klasse geschrieben. Sie stammen alle von der abstrakten Klasse `CBaseExperiment` ab:

```
class CBaseExperiment
{
friend class CExperimentThread;
public:
 CBaseExperiment(byte experiment_type,
           unsigned long datapoints,
           unsigned long long delta_t);

 virtual void start();
 virtual void stop();

 int save(const char *path);

 char * getFileName();

 virtual ~CBaseExperiment();

protected:
  virtual void processData(int fdi)=0;
  virtual void getType(char * buf)=0;
  virtual void printHeader(FILE *f)=0;
  virtual void printData(FILE *f)=0;
  exp_data exp_block;
  TMutex mutex;
  int stopFlag;

private:
  virtual void Run();
  char fileName[256];
 };
```

Die Methode `start` startet die Messung, indem über den FIFO 1 ein entsprechendes Kommando geschickt wird. Die *ROOT*-library unterstützt ebenfalls Threads. Diese laufen aber im User-Space und dürfen nicht mit den Threads von RTLinux verwechselt werden. Daher wird in der Methode `start` ein Thread gestartet, der nichts anderes tut, als die Daten entgegen zu nehmen, die über den FIFO 0 kommen. Danach kehrt `start` an seinen Aufrufer zurück. Der Thread beginnt seine Ausführung mit der Methode `Run`. Da jedoch die von RTLinux kommenden Daten abhängig vom jeweiligen Experiment sind, kann `CBaseExperiment` nicht wissen, wie diese zu behandeln sind. Deshalb ruft `Run` regelmäßig die abstrakte Methode `processData` auf. Die jeweiligen Experimente müssen diese dementsprechend überladen. Die anderen abstrakten Methoden werden für das Sichern der Daten benutzt und von `save` aufgerufen. Darauf soll nicht weiter eingegangen werden.

Es folgt beispielhaft die Klassendefinition des TDS-Experiments. Sie benötigt die Hilfsstrukturen `struct tdsdata` sowie `class CTDSMasses`, diese sind aus Platzgründen nicht dargestellt.

```
class CTDSExperiment:public CBaseExperiment
{
public:
  CTDSExperiment(byte ramp_N, byte ramp_A,
                 D24WORD ramp_start,
                 D24WORD ramp_end,
                 D24WORD ramp_step,
                 unsigned long long time_diff,
                 unsigned long datapoints,
                 unsigned long long delta_t,
                 unsigned long long wait_t,
                 CTDSMasses massinfo,
                 byte set_mass_N, byte set_mass_A,
                 byte qmssignal_N, byte qmssignal_A,

                 unsigned long long temp_scan_t,
                 byte temp_in_N, byte temp_in_A,

                 byte range0_N,byte range0_A,
                 byte range1_N,byte range1_A,

                 byte gain0_N,byte gain0_A,
                 byte gain1_N,byte gain1_A);
```

```
  virtual ~CTDSExperiment();

  virtual void start();
  virtual void stop();

  unsigned long getMaxData();

  tdsdata operator[](unsigned long j);

protected:
  virtual void processData(int fdi);
  virtual void getType(char *buf);
  virtual void printHeader(FILE *f);
  virtual void printData(FILE *f);

private:
  byte r_N;
  byte r_A;
  D24WORD r_start;
  D24WORD r_end;
  D24WORD r_off;
  unsigned long long r_t_diff;

  unsigned long counter;
  tdsdata *daten;
};
```

Fazit

Das Mess-System ELISABET wird seit ca. einem Jahr erfolgreich zur Oberflächenpräparation und Analyse eingesetzt. Es ersetzt das alte System LISBET, das unter *VMS* auf einer *DEC-microvax* lief.

Aufgrund des Alters der Hardware kam es jedoch öfter zu Ausfällen. Zur Reparatur nötige Ersatzteile waren nur noch in Restbeständen zu finden und sind inzwischen gar nicht mehr verfügbar. Aus diesen Gründen entschlossen wir uns zu einer Neuentwicklung auf der Basis eines modernen echtzeitfähigen und wohl dokumentierten Betriebsystems.

Es hat sich gezeigt, dass sich mit RTLinux sogar Frequenzen messen lassen. Zwar nicht mit extremer Präzision, in aber einer für die entsprechenden Messungen mehr als ausreichenden Genauigkeit. Der durch die Latenzzeiten entstehende Fehler konnte zudem durch Messung der tatsächlichen Zeit weitgehend ausgeglichen werden.

Die für RTLinux angegebene maximale Latenzzeit beträgt $15\ \mu$s. Abbildung 1 zeigt die Verteilung der Latenzzeiten, wie sie mit dem Testprogramm `measurements` aus dem Ordner `examples` der RTLinux 3.1 Version gemessen wurde. Testrechner war dabei ein 1 GHz Athlon. Kaum sichtbar ist der kleine Balken der Größe 1 bei 20500 ns. Sieht man die Definition von Latenzzeit wirklich streng, würde dies der maximalen Latenzzeit entsprechen.

Unter zu hilfenahme zusätzlicher Elektronik und einem weiteren PC sollen in Zukunft auch β-NMR Messungen sowie Messungen der Halbwertszeit von ^{8}Li mit dem

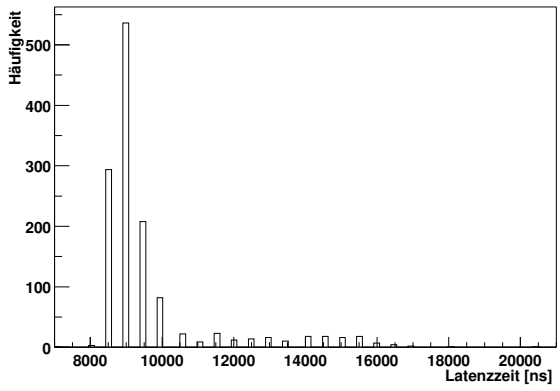

Abbildung 1: Verteilung der Latenzzeiten. In den meisten Fällen ist sie nicht größer als 10 μs. Bei 18000 ns und 20500 ns treten jedoch (kaum sichtbar) noch zwei „Ausreisser“ auf. Gemessen mit einem 1 GHz Athlon unter RTLinux 3.1

neuen System durchgeführt werden. Dafür ist eine sehr exakte Zeitbasis notwendig, für die die PC-interne Uhr nicht ausreicht und somit von einem externen Zeitgeber generiert werden muss. Das System ist in einer Testphase wird in Kürze in Betrieb genommen.

Shared Memory

In der RTLinux-Distribution befindet sich auch der mbuff driver von Tomasz Motylewski. Voraussetzung zu dessen Benutzung ist, dass das Modul mbuff.o geladen ist und die Gerätedatei `/dev/mbuff` existiert. (Sie kann mit `mknod /dev/mbuff c 10 254` erzeugt werden.) Mit diesem Treiber ist es möglich, Speicherbereiche, die gemeinsam vom User- und Kernel-Space benutzt werden, einzurichten. Die Befehle zum Allozieren und Freigeben des Speichers lauten:

```
#include <mbuff.h>
void * mbuff_alloc(const char *name, int size);
void mbuff_free(const char *name, void * mbuf);
```

Dabei bezeichnet `name` einen Namen, unter dem der gewünschte Bereich der Größe `size` verwaltet wird. Beim ersten Aufruf von `mbuff_alloc` mit einem neuen Namen wird der Speicher belegt, ein Zeiger darauf zurückgegeben und ein interner Referenzzähler auf 1 gesetzt. Weitere Aufrufe dieser Funktion unter dem selben Namen, i.d.R. von anderen Prozessen, egal ob aus dem User- oder Kernel-Space, geben einen Zeiger auf den selben Speicherbereich zurück und erhöhen den Zähler um 1. Im Fehlerfall erhält man immer `NULL` zurück.

Aufrufe von `mbuff_free` erniedrigen den Zähler. Ist der Referenzzähler schließlich 0, wird der Speicherbereich wieder freigegeben. Das Argument `mbuf` sollte der von `mbuff_alloc` erhaltene Zeiger sein.

Die Funktionen sind inline in der Datei `mbuff.h` ausprogrammiert. Daher sind sie auch für Anwendungsprogramme ohne zusätzliches Linken von Objekt-Code oder libraries direkt verfügbar.

Linux Gerätetreiber

Das am Anfang beschriebene Modul `camac.c` stellt keinen Treiber dar, weil es vom User-Space her nicht benutzbar ist. Erst zusammen mit dem Kommando-Modul unter einem RTLinux-Kernel kann man auf die Funktionen zugreifen. Unter Umständen möchte man vielleicht auch von einem normalen Kernel aus das CAMAC-Crate steuern. In so einem Fall macht natürlich die erweiterte Version mit den Mutex-Objekten keinen Sinn. Deshalb gehen wir nun wieder von der ursprünglichen Datei aus.

An dieser Stelle kann nur eine recht oberflächliche und einfache Darstellung des Themas Linux-Gerätetreiber erfolgen. Es gibt darüber ein hervorragendes Buch [5]. Es behandelt zwar nur die 2.0-er Kernel und gibt einen Ausblick auf die 2.1-Versionen, in Kürze soll aber eine Neuauflage erscheinen [6].

Außerdem sei auch nicht verschwiegen, dass wir nur das absolut notwendigste für unsere Projekt programmiert haben, DMA-Transfer und Interrupts (ausgelöst durch so genannte Look-At-Me Signale der CAMAC-Einschübe) sind bei CAMAC-Controllern möglich.

Unter Linux gibt es Netzwerk, Block und Zeichentreiber. Blocktreiber werden z.B. für Geräte benutzt, auf denen Filesysteme angelegt werden können; Zeichentreiber für alle anderen Fälle. Vom User-Space wird auf einen Zeichentreiber über eine Gerätedatei zugegriffen. Diese befinden sich meist im Verzeichnis `/dev` und werden wie normale Dateien mit `open` geöffnet, mit `read` und `write` benutzt und schließlich wieder mit `close` geschlossen. Erzeugen kann man eine Gerätedatei durch das Kommando `mknod` */dev/name* `c` *major minor* von der Kommandozeile aus. Dabei bezeichnet der Parameter *major* die so genannte Major-Nummer. Dies ist eine Zahl, unter der sich der Treiber im Kernel registriert hat.

Treiber in Modulform registrieren sich beim Kernel durch einen Aufruf von `register_chrdev`. Dies geschieht von `init_module` aus.

```
#include <linux/fs.h>
int register_chrdev(unsigned int major,
                    const char *name,
                    struct file_operations *fops);
int unregister_chrdev(unsigned int major,
                      const char *name);
```

`major` steht für die oben beschriebene Zahl und `name` für den Namen des Gerätes. Alle Gerätenamen mit ihrer Major-Nummer findet man in `/proc/devices`. Im Fehlerfall liefert die Funktion eine Zahl < 0 zurück.

Das entsprechende Gegenstück `unregister_chrdev` gibt die Major-Nummer wieder frei und sollte von `cleanup_module` aus aufgerufen werden.

Die Struktur `file_operations` ist eine Sprungtabelle. In ihr stehen Zeiger auf Funktionen, die aufgerufen werden, wenn durch den Treiber eine Operation erfolgen soll. Sie ist häufig Änderungen unterworfen, im Kernel 2.4.4 sieht sie folgendermaßen aus:

```
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file * filp,
                   char * buf , size_t count, loff_t *);
  ssize_t (*write)(struct file * filp, const char * buf,
                    size_t count, loff_t *);
  int (*readdir) (struct file *, void *,
                  filldir_t);
  unsigned int (*poll) (struct file *,
                        struct poll_table_struct *);
  int (*ioctl) (struct inode *, struct file *,
                unsigned int, unsigned long);
  int (*mmap) (struct file *,
               struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, struct dentry *,
                int datasync);
  int (*fasync) (int, struct file *, int);
  int (*lock) (struct file *, int, struct file_lock *);
  ssize_t (*readv) (struct file *,
           const struct iovec *,
           unsigned long, loff_t *);
  ssize_t (*writev) (struct file *, const struct iovec *,
           unsigned long, loff_t *);
};
```

Es müssen nicht alle Funktionen programmiert werden, in diesem Fall sollte der Zeiger in der Tabelle `NULL` sein. Für den CAMAC-Treiber könnte diese Struktur z.B. so gefüllt werden.

```
static struct file_operations camac_fops =
{
read:    f_camac_read,
write:   f_camac_write,
open:    f_camac_open,
release: f_camac_release,
};
```

Öffnet ein Anwendungsprogramm die Gerätedatei, wird also vom Kernel die Funktion aufgerufen, auf die `open` zeigt, also `f_camac_open`. Für `read` und `write` gilt entsprechendes, ein Schließen entspricht der Methode `f_camac_release`. In `f_camac_open` sollte der Verwendungszähler durch Aufruf des Makros `MOD_INC_USE_COUNT` hochgezählt werden. Dadurch wird ein versehentliches Entladen des Treibers mit `rmmod` verhindert, wenn ein Anwendungsprogramm noch darauf zugreift. Entsprechend sollte er in

f_camac_release durch MOD_DEC_USE_COUNT wieder erniedrigt werden.

Betrachten wir kurz die wichtigsten Parameter der Funktionen read und write aus der Sprungtabelle. buf ist ein Speicherbereich, in dem die an oder vom Treiber zu übertragenden Daten stehen. count steht für deren Größe in Byte. Dabei ist zu beachten, dass dies ein Zeiger auf den User-Space ist, auf den nicht direkt zugegriffen werden darf! Um Daten vom Kernel-Space in den User-Space zu übertragen oder in die andere Richtung, gibt es die folgenden Funktionen. Ihre Parameter sollten selbst erklärend sein:

```
#include <asm/uaccess.h>
#include <asm/segement.h>
unsigned long copy_from_user(unsigned long to,
                             unsigned long from,
                             unsigned long len);
unsigned long copy_to_user(unsigned long to,
                           unsigned long from,
                           unsigned long len);
```

In den f_camac_read bzw. In den f_camac_write Methoden werden dann die tatsächlichen Hardwarezugriffe camac_read und camac_write ausgeführt und deren Daten in den User-Space transferiert. Etwas schwierig ist allerdings die Adressierung. Für die einzelnen Einschübe könnte man im Verzeichnis /dev Gerätedateien mit laufender Minor-Nummer erzeugen. In den Aufrufen von f_camac_read oder f_camac_write kann man diese dann folgendermaßen aus dem Parameter filp extrahieren.

```
struct inode* inode = filp->f_dentry->d_inode;
unsigned int minor = MINOR(inode->i_rdev);
```

Dann fehlen aber immer noch Unteradresse A und der Function-Code F. Diese kann man jedoch den Daten beifügen, die der write-Befehl der Anwendung an den Treiber schickt. Ein solcher String könnte dann z.B. so aussehen:

```
char command[]="read 3 0";
write(CAMAC_DRIVER,command,strlen(command));
```

Statt read als Kommando wären z.B. auch write, setinhibit und clearinhbt denkbar. Anschließend würde die Anwendung über die Gerätedatei dann die Antwort auslesen.

Auf der Kernelseite muss dann f_camac_write den String wieder zerlegen und das Kommando ausführen. Der nächste f_camac_read Aufruf liefert die Antwort des CAMAC-Crates über die Gerätedatei an die Anwendung zurück.

Danksagung

Wir danken Prof. D. Fick (Marburg) und dem Max-Planck-Institut für Kernphysik in Heidelberg für die fortlaufende ideelle und materielle Unterstützung, sowie allen Entwicklern und Förderern der Open-Source-Community, aus deren Arbeit dieses Projekt zusammengesetzt wurde.

References

[1] http://www.ifh.de/~ole/camac/

[2] http://fsmlabs.com/community/

[3] http://www.kernel.org/

[4] http://root.cern.ch/

[5] A. Rubini, *Linux Gerätetreiber*, O' Reilly Verlag, 1998, ISBN 3-89721-122-x.

[6] A. Rubini, *Linux Gerätetreiber*, O' Reilly Verlag, 2002, ISBN 3-89721-138-6.

About the author:

Oliver Kühlert, geb. am 1.1.1972, hat während des Studiums der Physik in Marburg an Treibern und Diagnoseprogrammen für verschiedene Firmen gearbeitet. Zunächst für Windows NT, später Linux. Momentan ist er Doktorand der Philipps-Universität Marburg und u.a. verantwortlich für die Programmierung der Echtzeit-Komponente des Mess-Systems ELISABET am MPI für Kernphysik in Heidelberg.