Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für verteilte Systeme

Laborpraktikumsbericht

Entwicklung eines echtzeitfähigen CAN-Treibers für „Publisher-Subscriber“-Middleware unter Linux

Verfasser:

Sergiy Dzhantimirov

13 September 2006

Betreuer:

Prof. Dr. rer. nat. Jörg Kaiser
Dipl. Inform. Michael Schulze

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D-39106 Magdeburg
Germany

Inhaltsverzeichnis

Einführung

Diese Arbeit beschäftigt sich mit der Entwicklung eines Echtzeitfähigen CAN-Treibers, der in einer „Publisher-Subscriber“-Kommunikationssoftware für Linux zukünftig benutzt werden soll. Nach einer kurzen Aufgabenbeschreibung wird ein kleiner Überblick über einige vorhandene RT-Linux Erweiterungen gegeben, die als Plattform für den RT-Treiber benutzt werden können. Die ausgewählte Variante wird detaillierter behandelt. Nach einer kompakten Beschreibung des ausgewählten CAN-Controllers folgt eine ausführliche Treiberbeschreibung mit Installations-, Architektur- und API-Übersicht. Zum Schluss werden einige Anwendungsbeispiele und Testergebnisse präsentiert. Die Arbeit wird mit einer Zusammenfassung abgerundet.

Entwicklungsgrund eines echtzeitfähigen CAN-Treibers für Linux.

Die Entwicklung eines echtzeitfähigen CAN-Treibers für Linux wurde vom COSMIC Projekt hervorgerufen. Die ereignisbasierte, echtzeitfähige COSMIC (COoperating SMart devICes) Middleware für CAN-Bus unterstützt ein verteiltes System von kooperierenden autonomen Geräten (Aktoren und Sensoren). Das von COSMIC angebotene Applikationsinterface ermöglicht eine applikationsgerichtete, abstrakte Formulierung von zeitlichen und qualitativen Kommunikationsattributen. Zu diesem Zweck werden Ereigniskanäle mit unterschiedlichen Zeit- und Zuverlässigkeits-Eigenschaften angeboten. Hard Real-Time-Ereigniskanäle garantieren eine sichere, planmäßige Nachrichtenzustellung in festgelegten zeitlichen Grenzen unter speziellen Ausfallannahmen. Soft Real-Time-Kanäle werden nach Deadlines bearbeitet, bieten aber keine Zustellgarantie für Nachrichten bei Überlastung. Letztendlich werden die Non-Real-Time-Ereigniskanäle für Nachrichten ohne spezielle zeitliche Anforderungen benutzt, die nach „best-effort“ Prinzip bearbeitet werden dürfen. Alle Kommunikationsteilnehmer, die nicht nach COSMIC-Regeln „spielen“, werden automatisch als Non-Real-Time Teilnehmer behandelt.

Die erweiterte Beschreibung von COSMIC lässt sich in entsprechenden Publikationen finden [1]. Für uns sind vor allem die Hard Real-Time Anforderungen von COSMIC interessant. Da die Middleware unter Linux implementiert werden soll, werden auch entsprechende Echtzeitanforderungen an das Betriebssystem selbst gestellt. Linux ist ein „general purpose“ Betriebssystem und kann im besten Fall nur als soft Real-Time System benutzt werden. Deswegen hat sich die Frage nach einer passenden Real-Time Linux-Erweiterung gestellt.

Real-Time Linux-Erweiterungen

RT-Linux

RTLinux ist eine Erweiterung von Linux zu einem Echtzeitbetriebssystem, die ursprünglich von Professor Victor Yodaiken zusammen mit seinem Studenten Michael Barabanov an der Universität von New Mexico entwickelt wurde. Die Echtzeittasks werden von einem (von Linux unabhängigen) erweiterbaren Minikernel verwaltet. Der eigentliche Linux-Betriebssystemkern ist aus Sicht dieses Minikernels ebenfalls nur eine Echtzeittask, allerdings mit der niedrigsten Priorität im System (Idle-Task). Durch diese Architektur gewinnt eine Echtzeitanwendung einerseits vollständige Kontrolle über die Hardware, kann aber andererseits nur sehr bedingt Linux-

Systemaufrufe nutzen. Neben preemptiven Priority-Scheduling für die RT-Taskverwaltung sind auch Module für Earliest-Deadline-First Scheduling und Kommunikationsmechanismen zwischen RT-Tasks über Queues und Semaphoren erhältlich. Der Datenaustausch zwischen RT-Tasks und Applikationen die im Linux-Userspace laufen, erfolgt für gewöhnlich über spezielle RT-FIFOs - daneben sind aber auch Mechanismen für Shared-Memory vorhanden. Die Philosophie der Entwickler von RT-Linux lautet: so wenig wie möglich in den bestehenden Linux-Kernel eingreifen, um dessen Stabilität nicht zu gefährden. Die nachfolgende Abbildung soll einen zusätzlichen Einblick in die Architektur von RTLinux gewähren.

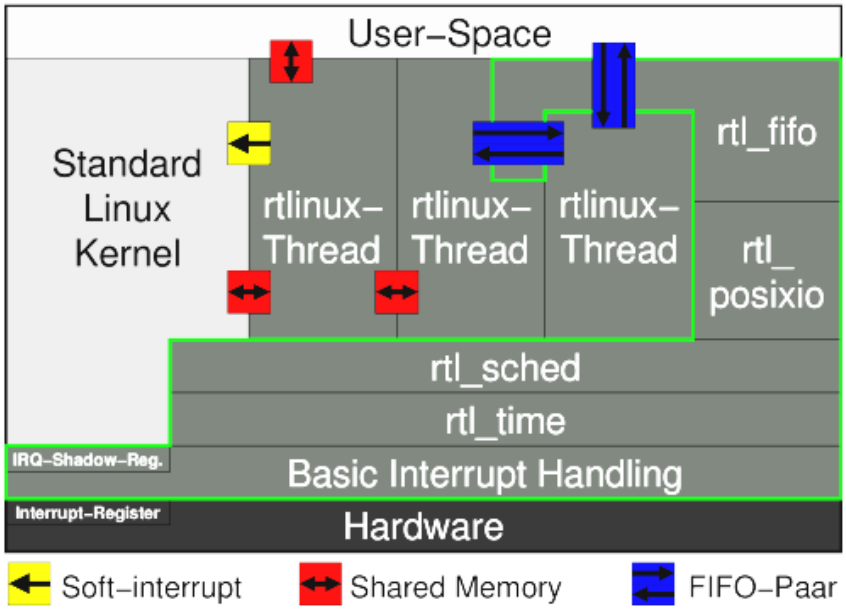

Abb. 1: RTLinux Architektur

RTLinux wird heute in einer freien und kommerziellen Version von der Firma FSMLabs vertrieben. Die Entwicklung wurde allerdings bei Version „3.1“ (Mai-2001) angehalten. Die „2.6“ Kernel Generation wird nicht unterstützt.

RTAI-classic

RTAI (Real Time Application Interface) ist eine Erweiterung von Linux zu einem Echtzeitbetriebssystem. Begründet wurde RTAI von Prof. Paolo Mantegazza vom Dipartimento di Ingegneria Aerospaziale der Universität Mailand. RTAI wurde vom Beginn an als Open-Source-Projekt von einer größeren Entwicklergemeinde weiterentwickelt.

Ein großer Pluspunkt von RTAI ist, dass es mit der Variante LXRT möglich ist, Hard-Real-Time-Tasks im Userspace laufen zu lassen und damit die Schutzmechanismen von Linux zu nutzen. Dies erfolgt ohne größere Einbußen im Bereich der Latenzzeiten und ohne großen Overhead. Bei

anderen Echtzeit-Systemen, welche ausschließlich im Kernelspace laufen, kann sich ein Fehler im Programmablauf verheerend auswirken.

Das RTAI Projekt befindet sich in aktiver Entwicklung und bietet ständig neue Verbesserungen und Möglichkeiten.

Xenomai (RTAI-fusion)

Das Xenomai Projekt wurde im August 2001 gestartet. 2003 wurde er mit dem RTAI Projekt vereint mit dem Ziel eine industriereife freie Real-Time Softwareplattform für GNU/Linux auf der Basis von Xenomai zur Verfügung gestellten abstrakten RTOS Kernels zu entwickeln. Das Projekt hat den Namen RTAI-fusion bekommen und wurde als experimenteller extra-Zweig parallel zur RTAI-classic entwickelt. 2005 ist RTAI-fusion von RTAI unabhängig geworden und ist jetzt unter dem Namen Xenomai bekannt.

Die Xenomai Real-Time Erweiterung (genauso wie parallel entwickelte RTAI-fusion) vererbt die meisten Eigenschaften von RTAI-classic. Das wichtigste Merkmal von RTAI – Userspace Real-Time Tasks, wird beibehalten. Es kommen aber auch sehr attraktive neue Möglichkeiten hinzu. Eine, wohl die interessanteste, Neuheit bei Xenomai besteht in der Möglichkeit unterschiedliche APIs (POSIX, VRTX, VxVorks, pSOS+, RTAI, ulTRON) dank der „Skin“-Technologie zu unterstützen, wobei diese APIs gleichzeitig benutzt werden können. Dieses Konzept erleichtert die Erstellung von neuen als auch die Portierung von existierenden Programmen für andere RTOS auf die Xenomai Plattform. Darüber hinaus wird, genauso wie von RTAI, auch eine relativ breite Palette von Hardwareplattformen unterstützt, was den Einsatz von Xenomai auch in manchen eingebetteten Geräten ermöglicht. Die ausführlichere Beschreibung dieser Plattform lässt sich auf der Entwicklerseite (http://www.xenomai.org) und auf unzähligen Technisch-basierten Foren und Internetseiten finden.

Anhand der beschriebenen Eigenschaften, einer sehr aktiven Entwicklung und breiter Unterstützung in der Web-Community wurde die Wahl zugunsten der Xenomai Real-Time Erweiterung getroffen.

Xenomai Installation

Xenomai ist als Paket in der Aktuellen Version 2.2.1 von http://www.xenomai.org/ herunterladbar. Nach dem Entpacken finden sich die Kernel Patches im Unterverzeichnis ./xenomai-2.2.1/ksrc/arch/, wo man die Auswahl zwischen den unterstützen Architekturen findet. Im jeweiligen Unterverzeichnis findet man diverse Patch-Files für die verschiedensten Kernel-Versionen. Die passende Kernel-Version ist unter http://www.kernel.org/pub/linux/kernel/v2.x

herunterladbar. Damit hat man das Handwerkzeug zusammen, um ein Xenomai-System aufzusetzen. Da der Installationsablauf sehr genau in der Installationsanleitung (./README.INSTALL) beschrieben ist, wird er an dieser Stelle nicht wiederholt. Die Datei ./TROUBLESHOUTING behandelt mögliche Probleme und ihre Lösungen, die vor der Installation zu berücksichtigen sind.

Trotzdem müssen noch ein Paar Wörter über einige Probleme gesagt werden, die während der Xenomai-Installation auf manchen Rechnern aufgetaucht sind.

Für den stabilen und präzisen Betrieb des Xenomai-gepatchten Linux Kernels ist es wichtig alle nicht maskierbare Interruptquellen zu eliminieren. Xenomai ist nicht immer in der Lage solche Quellen zu detektieren und richtig zu behandeln. Deswegen ist es z.B. empfehlenswert grundsätzlich auf das Power-Management zu verzichten. Ein weiteres Problem wurde rein zufällig entdeckt und ließ sich nicht softwaremäßig eliminieren. Wenn ein USB-Memory-Stick angeschlossen wurde, dann hat das zum periodischen (ca. 2 s) Jitter im System geführt, welcher eventuell über bestimmte nicht maskierbare Interrupts verursacht wurde. Deswegen soll darauf geachtet werden, dass im Laufenden Betrieb keine unnötige Peripherie physisch angeschlossen bleibt, ausser wenn sie als mögliche Problemquelle sicher ausgeschlossen werden kann. Beim parametrieren des Linux-Kernels soll eventuell auf Prozessorspezifische Optimierungen verzichtet werden. Die Athlon-Optimierung des Kernels hat z.B. zu den Laufzeitproblemen (nicht vorhersagbares Systemverhalten, Crash) beim Einsatz der Real-Time Erweiterung geführt.

Unten wird exemplarisch eine funktionierende Software-Konfiguration für einen IBM ThinkPad Laptop mit einem Pentium 3 Prozessor angegeben:

Softwareversionen		
Kernel	2.6.17.11	http://kernel.org
Xenomai	2.2.0	http://www.xenomai.org
Adeos Kernel Patch	1.3-08	./xenomai-2.2.0/ksrc/arch/i386/patches oder http://www.gna.org/projects/adeos/
Kernelparameter		
SMP	aus	
APIC	aus	Weil kein APIC wirklich vorhanden ist. Lese ./TROUBLESHOUTING von Xenomai für extra Info.
ACPI und APM	aus	
Prozessor	Pentium-III/...	Bei AMD K7-Derivaten darf keine K7 Optimierung benutzt werden!

Generic x86 support	ein	
Preemption Model	Low-Latency	
Preempt The Big Kernel Lock	ein	
Interrupt Pipeline	ein	Erscheint nur beim Xenomai-gepatchten Kernel
Real-time sub-system --> Machine --> SMI workaround --> Globally disable SMI	ein	

Tabelle 1: Exemplarische Softwarekonfiguration für IBM ThinkPad T23 Laptop (PIII)

CAN Hardware

Als Hardwareplattform für die Entwicklung eines Real-Time CAN-Treibers hat der LPT CAN-Dongle der Firma PEAK-System (http://www.peak-system.com) gedient (Abb. 2).

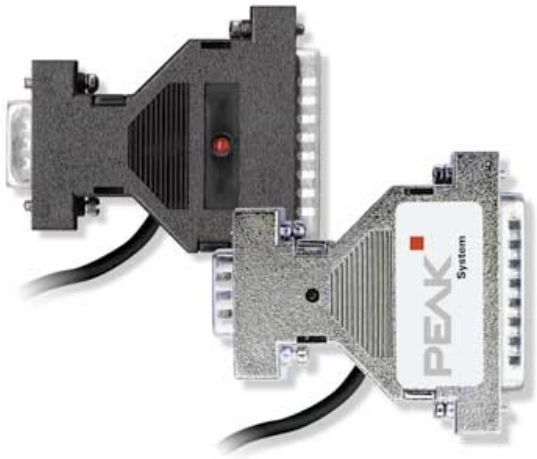

Abb. 2: PC-Parallel Port zu CAN Interface

Dieser Dongle ist für die einfache und kostengünstige Anbindung an CAN-Netzwerke der Spezifikation 2.0A und 2.0B konzipiert. Der in einem Steckergehäuse untergebrachte Parallel-zu-CAN-Umsetzer ist durch seine geringe Größe optimal für den Einsatz an Laptops oder Notebooks geeignet. Er kann in Abhängigkeit von der vorhandenen parallelen Schnittstelle in einem „Multiplex-“ oder in einem „Enhanced Parallel Port“-Modus betrieben werden. Die Stromversorgung des PCAN-Dongle erfolgt über einen speziellen Adapter, der am Tastaturausgang des Rechners angeschlossen wird.

Technische Daten

- Baudrateneinstellung bis zu 1 MBit/s
- PHILIPS CAN-Controller SJA1000, 16 MHz Taktfrequenz
- CAN-Transceiver 82C251
- Erfüllt die CAN-Spezifikationen 2.0A (11-Bit ID) und 2.0B (29-Bit ID)
- Hardware-Reset des SJA1000 per Software möglich
- Logikkontrolle mittels integriertem CPLD
- In platzsparendem Portadaptergehäuse DIN 25-pol. (LPT) auf DIN 9-pol. (CAN)
- Anschluss an CAN-Bus über D-SUB, 9-polig (nach CiA DS102-1)
- Softwaremäßiges Umschalten von „Multiplex-“ auf „EPP (Enhanced Parallel Port)“-Betrieb

Die Gerätewahl wurde nicht von der technischen Seite sondern eher von der Verfügbarkeit des Dongles diktiert, trotz grundlegender Mängel, die schon allein durch das Parallel-Port Interface bedingt sind. Da der Parallel-Port eines normalen PCs mit maximal 2MBit/s funktioniert, führt es automatisch zu relativ hohen Latenzzeiten bei der CAN-Nachrichtenübertragung. Das werden später auch die entsprechenden Messungen mit dem funktionierenden Real-Time-Treiber zeigen.

Beim Anschluss des Dongles an einen PC sollen bestimmte Konfigurationsdetails berücksichtigt werden. Um die Latenzzeiten so gering wie möglich zu halten, soll der Parallel-Port des Computers im ECP-Modus betrieben werden. Genaue Bedingungen und Einstellungen für den richtigen Betrieb können aus der mit dem Dongle mitgelieferten Dokumentation entnommen werden [4].

Der Gerätehersteller bietet Treiberunterstützung für Windows und Linux Betriebssysteme an, wobei der Linux-Treiber unter GPL Lizenz verbreitet wird. Die Verfügbarkeit des Open-Source Linux Treibers hat sich als sehr nützlich erwiesen, weil der Original-Treiber als Grundlage für die Real-Time-Umsetzung benutzt werden konnte.

Real-Time CAN-Treiber

Treibereigenschaften

Obwohl die Real-Time-Version des CAN-Treibers von der Originalversion der Firma „Peak System“ abgeleitet wurde, unterscheiden sich beide Varianten in der API ziemlich stark, was im wesentlichen durch die Nutzung von Xenomai-Erweiterung bedingt ist. Neben der einigen

Begrenzungen hat Xenomai auch bestimmte Erweiterungen beigebracht, die sich im RT-Treiber abgebildet haben.

Der „rtpcan.ko“ Treiber unterstützt momentan nur die „PCAN-Dongle“ Hardware. Die Unterstützung anderer Geräte auf der sja1000 Basis ist durch die modulare Treiberarchitektur (wird unten detaillierter beschrieben) nicht schwierig. Potenziell können insgesamt bis zu 8 Geräte unterschiedlicher Typen unterstützt werden. Für den PCAN-Dongle sind die Basisadresse und die Interruptnummer als Kommandoparameter einstellbar.

Die Kommunikation zwischen dem Treiber und dem Benutzerprogramm erfolgt über zwei Mechanismen: Real-Time-Events und Real-Time-Ioctl. Bestimmte Treiberaktivitäten oder Ereignisse auf dem CAN-Bus werden dem Benutzerprogramm vom RT-Treiber über entsprechende Events mitgeteilt. Dazu zählen zum Beispiel solche Ereignisse wie „Bereit zum Lesen“, „Bereit zum Schreiben“, „Arbitrierungsverlust“ usw. Das Programm kann dann entsprechende Aktionen über das Real-Time-Ioctl Interface durchführen.

Der Treiber kann von mehr als einem Programm gleichzeitig benutzt werden. Wenn mehrere Programme das gleiche physikalische Interface benutzen, dann laufen sie im Prinzip im „Konkurrenzmodus“, weil der gleiche physikalische Ein- und Ausgang zusammen benutzt wird.

Das Interface zum Treiber ist in der Datei „rtpcan.h“ definiert.

Manuelle Treiberinstallation

Die Aktuelle Treiberversion (2006-08-25) unterstützt nur die 2.6 Kernelgeneration. Die ausführlichen Tests wurden mit der 2.6.17.x Kernelserie durchgeführt. Voraussetzung für die erfolgreiche Treiberinstallation ist das Laufende System mit dem Xenomai-gepatchten Linux-Kernel und dem vollständig installierten Xenomai-Packet (siehe „Xenomai Installation“ oben).

Nach dem entpacken der „rtpcan.tar.gz“ Datei in einen optionalen Verzeichnis (z.B. /home/klaus/tmp) entsteht ein Dateienbaum:

/home/klaus/tmp/rtpcan/

```
|-- common
    |-- include
        |-- rtpcan.h                    Treiber-API
|-- driver                              Treiberquellen
    |-- include
        |-- rtpcan_common.h
        |-- rtpcan_device.h
        |-- rtpcan_fops.h
        |-- rtpcan_lpt.h
        |-- rtpcan_main.h
        |-- rtpcan_queue.h
```

```
        |-- rtpcan_sja1000.h
        |-- rtqueue.h
    |-- src
        |-- rtpcan_device.c
        |-- rtpcan_fops.c
        |-- rtpcan_lpt.c
        |-- rtpcan_main.c
        |-- rtpcan_sja1000.c
        |-- rtqueue.c
    |-- Makefile                        Makefile des Treibers
    |-- start.sh                        Beispielscript zum Treiberstart
    |-- stop.sh                         Beispielscript zum Treiberstop
|-- test                                Testprogramme
    |-- src
        |-- t1_rx.c
        |-- t1_tx.c
        |-- t2_rx.c
        |-- t2_tx.c
        |-- t3_master.c
        |-- t3_slave.c
    |-- Makefile                        Makefile für Testprogramme
```

Nach dem erfolgreichen Erstellen des Treibers (cd ..../driver/ -> make) entsteht ein Kernelmodul „rtpcan.ko“ (Kernel 2.6), welches vom root-Benutzer aus der Kommandozeile gestartet werden kann. Dabei können die zusätzlichen Parameter spezifiziert werden:

```
/sbin/insmod ./rtpcan.ko  itype=epp ctype=sja1000 io=0x278 irq=5
/sbin/insmod ./rtpcan.ko  itype=epp,sp ctype=sja1000,sja1000 \
                          io=0x278,0x378 irq=5,7
```

Momentan werden folgende Kommandoparameter vom Treiber unterstützt:

- itype: – Spezifiziert das verwendete Interface
  - epp – Das Gerät wird über den EPP Parallel-Port betrieben
  - sp – Das Gerät wird über den SP Parallel-Port betrieben
- ctype – Spezifiziert das verwendete CAN-Chip
  - sja1000 – das Gerät enthält den sja1000 CAN-Chip von Philips
- io – Basisadresse des Parallel-Ports/Gerätes (0x278, 0x378, 0x12467234, ...)
- irq – Interrupt des Parallel-Ports /Gerätes(5, 7, ...)

Nach dem Treiberstart werden die Geräteinstanzen im Xenomai-Registry unter den Namen „rtpcanX“ (rtpcan0, rtpcan1, usw.) registriert. Über diese Namen wird ein gezielter Gerätezugriff ermöglicht. Die Xenomai Registry-Namen ersetzen für RT-Programme die traditionellen Gerätenamen im /dev/ Verzeichnis.

Eine kompakte Übersicht über alle vom Treiber initialisierten Geräte gibt der `/proc/rtpcan` Eintrag (Tabelle 2):

```
                                /proc/rtpcan

        ------------------------------------------------------------
        -                                                          -
        -     PEAK-Systems CAN Realtime interfaces (xenomai-based) -
        -        [See /proc/xenomai/rtdm/... for more details]     -
        -                                                          -
        ------------------------------------------------------------
        - device | itype | ctype   |  base  | irq | irqs          -
        ------------------------------------------------------------
          rtpcan0   epp    sja1000   0x278     5       0
          rtpcan5   sp     sja1000   0x378     7       0
```

Tabelle 2: Übersicht über verfügbare Geräte

Zusätzliche Laufzeit-Information zum RT-Treiber und zum Xenomai System insgesamt lassen sich im Verzeichnis `/proc/xenomai/rtdm/` finden.

Treiberarchitektur

Der Original-Treiber von „Peak System“ hatte wesentlichen Einfluss auf die Entwicklung der Real-Time Variante. Die ersten RT-Testversionen waren im Prinzip nichts anderes als adaptierte Varianten des Original-Treibers für die Real-Time Umgebung. Die aktuelle Real-Time Version hat wesentliche Unterschiede in der API, behält aber trotzdem die wichtigste Eigenschaft seines Vorgängers – die Modularität auf der Quellcode-Ebene, welche eine leichte Integration von neuen Geräten mit dem sja1000 CAN-Controller (und eventuell mit einem anderen) und unterschiedlichen physikalischen Anschlüssen (USB, PCI, PCMCIA, usw.) ermöglicht.

Jedes angeschlossene Gerät wird im Treiber in einer individuellen Instanz der „t_RtPcanDeviceContext“ Struktur abgebildet. Diese Struktur kann als eine Art Geräte-Klasse (im Sinne C++) betrachtet werden. Die zwei weiteren „privaten“ Strukturen, „chip“ und „interface“, die sie enthält, bilden eine geräteabhängige Abstraktionsschicht mit der einheitlichen internen API. Die „chip“ Struktur ermöglicht einen CAN-Chip-Zugriff über ein einheitliches Interface (Funktionszeiger). Der Hardware-Zugriff zum CAN-Chip erfolgt über eine physikalische Schnittstelle (LPT, USB usw.), die über die „interface“ Struktur angesprochen wird. Das „interface“ stellt eine Soft-Brücke bei der Kommunikation zwischen dem PC und dem CAN-Chip dar. Das .heißt: alle „chip“-Funktion können nur über bestimmte „interface“-Methoden den realen CAN-Chip ansprechen. Sollte die Unterstützung für ein neues physikalisches Interface oder einen CAN-

Chip hinzugefügt werden, dann müssen lediglich die Funktionen neu geschrieben werden, welche die „interface“- oder die „chip“-Schnittstelle implementieren.

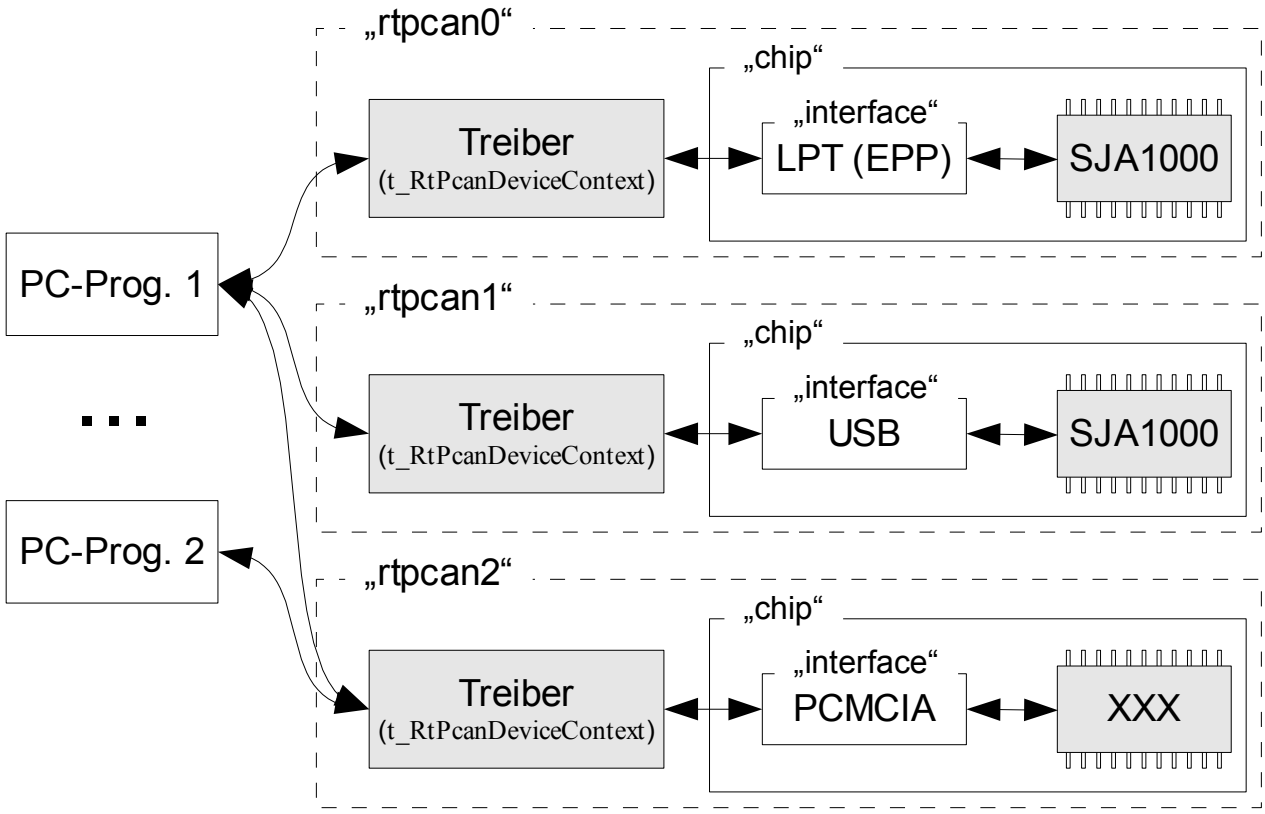

Abb. 3: Kommunikationsstruktur des Treibers

Die „interface“ Struktur bietet eine einheitliche Schnittstelle zum physikalischen Interface. In der aktuellen Implementierung werden folgende Methoden angeboten:

- create() bereitet die benötigten Software-Ressourcen vor.
- destroy() bereinigt die allozierten Software-Ressourcen.
- open() öffnet die Schnittstelle (macht die Hardware betriebsbereit)
- release() schließt die Schnittstelle (schaltet sie „offline“)
- readreg() liest von einem Chip-Register
- writereg() schreibt in ein Chip-Register

Die „chip“ Struktur stellt eine einheitliche Schnittstelle zum integrierten in das Gerät CAN-Controller dar.

- create() bereitet die benötigten Software-Ressourcen vor.
- destroy() bereinigt die allozierten Software-Ressourcen.
- open() öffnet den CAN-Controller (macht die Hardware betriebsbereit)
- release() schließt den CAN-Controller (schaltet ihn „offline“)
- probe() testet, ob der CAN-Controller im Gerät verfügbar und ansprechbar ist
- read() liest vom CAN-Controller die empfangene CAN-Nachricht

- write() schreibt in den CAN-Controller eine neue zu übertragende Nachricht
- txabort() bricht die aktuelle Nachrichtenübertragung ab
- irqhandler() Interrupt-Behandlungsroutine – fängt alle CAN-Controller-Interrupts ab

Beim Hinzufügen neuer Hardware sollen die entsprechenden „chip“- oder „interface“-Methoden einfach an die Instanz der „t_RtPcanDeviceContext“-Struktur „angehängt“ werden (Abb. 4).

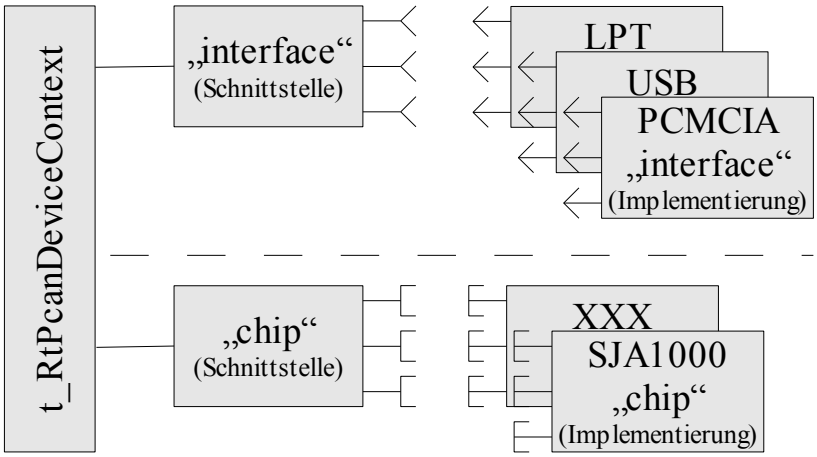

Abb. 4: Modulare Treiberstruktur.

Die Treibermodularität wirkt sich auch entsprechend auf der Dateienebene aus. Sie lässt sich leicht auf dem früher vorgeführten Quellcode-Verzeichnis/Dateien-Baum erkennen. Die Bedeutung einiger Quellcode-Dateien wird hier etwas detaillierter beschrieben.

- `rtqueue.h`

  deklariert die API Strukturen, gemeinsame Typen und Konstanten. Die Datei wird für den Treiber und für alle Benutzerprogramme benötigt.
- `rtpcan_lpt.c`

  implementiert die „interface“-Schnittstelle für den Parallel-Port (EPP/SP)
- `rtpcan_sja1000.c`

  implementiert die „chip“-Schnittstelle für den SJA1000 CAN-Controller
- `rtpcan_device.c`

  implementiert die allgemeinen Gerätemethoden (create, destroy, open, release)
- `rtpcan_fops.c`

  implementiert die API-Treiberfunktionen, die von den Benutzerprogrammen angesprochen werden: rtopen, rtclose, rtioctl (write, read, txabort, status, init).
- `rtpcan_main.c`

  enthält die allgemeinen Methoden für die Initialisierung, Laufzeitbeobachtung und Bereinigung des Treibers.

Treiberinterface

Wie schon oben erwähnt wurde, erfolgt die Kommunikation zwischen dem Real-Time Treiber (Kernel-Space) und den Benutzerprogrammen (User/Kernel-Space) über zwei Mechanismen. Die Ereignisse auf der Treiber-Ebene werden den Benutzerprogrammen über vordefinierte Xenomai-Events mitgeteilt. Der nachfolgende Datenaustausch zwischen den Benutzerprogrammen und dem Treiber erfolgt über das Xenomai-ioctl (rtioctl) Interface.

Die Events (technisch gesehen Event-Bits) werden in zwei Gruppen aufgeteilt: automatische und manuelle.

Die automatischen Events müssen vom Benutzerprogramm nur abgefangen werden. Weitere Aktionen bezüglich dieser Events sind vom Benutzerprogramm nicht nötig. Die manuellen Events benötigen manuelles Zurücksetzen (Löschen) im Benutzerprogramm, nachdem sie abgefangen wurden. Sonst wird das nächste Treiber-Ereignis nicht richtig auf der Benutzerebene erkannt/interpretiert.

In der aktuellen Treiberversion sind folgende Event-Bits deklariert:

Die „automatische" Event-Gruppe (RTPCAN_EVGROUP_AUTO):

- RTPCAN_EV_READ
  Eine oder mehrere CAN-Nachrichten befinden sich im RX-Buffer des Treibers und können vom Benutzerprogramm über das rtioctl Xenomai-Interface gelesen werden. Dieses Event-Bit wird automatisch nach dem Auslesen der letzten Nachricht zurückgesetzt.
- RTPCAN_EV_WRITE
  Der TX-Buffer des Treibers ist nicht voll und kann die nächste zu sendende CAN-Nachricht empfangen (über rtioctl). Sobald der Buffer voll ist, wird dieses Event-Bit zurückgesetzt.
- RTPCAN_EV_TXCOMPLETE
  Die physikalische Übertragung einer zuletzt übertragenen CAN-Nachricht wurde vollständig und erfolgreich abgeschlossen. Das Event-Bit wird beim nächsten Schreibvorgang automatisch gelöscht.
- RTPCAN_EV_TXABORT
  Die Übertragung einer im TX-Buffer des CAN-Controllers wartenden CAN-Nachricht wurde erfolgreich abgebrochen (nach der manuellen Abbruch-Anforderung) und sie wurde aus dem Buffer verworfen. Dieses Event-Bit wird automatisch beim nächsten Schreibvorgang zurückgesetzt.

Die „manuelle" Event-Gruppe (RTPCAN_EVGROUP_MANUAL):

- RTPCAN_EV_ERROR_WARN
  Der Bus- oder Fehler-Zustand hat sich geändert. Genauere Information lässt sich in der Anleitung zum SJA1000 CAN-Controller finden (bei der Beschreibung des Interrupt- und Status-Registers im PELICAN Modus)
- RTPCAN_EV_DATA_OVERRUN
  Es wurde eine ankommende CAN-Nachricht wegen der Überfüllung des Controller- oder Treiber-RX-Buffers verloren.
- RTPCAN_EV_WAKE_UP
  Der CAN-Controller wurde aus dem Schlafmodus wegen einer Bus-Aktivität geweckt.
- RTPCAN_EV_ERROR_PASSIVE
  Eine Änderung des fehlerpassiven Zustandes hat stattgefunden.
- RTPCAN_EV_ARBIT_LOST
  Der CAN-Controller hat die Arbitrierung verloren.
- RTPCAN_EV_BUS_ERROR
  Auf dem CAN-Bus wurde ein Fehler vom CAN-Controller detektiert.
- RTPCAN_EV_TX
  „Tx“-Event-Bit wird sofort nach dem platzieren der ausgehenden CAN-Nachricht in den TX-Buffer des CAN-Controllers gesetzt.
- RTPCAN_EV_RX
  Eine oder mehrere angekommene CAN-Nachrichten befinden sich im RX-Buffer des CAN-Controllers.

Die Events werden im Benutzerprogramm mit dem Xenomai-Aufruf „rt_event_wait(...)“ abgefangen. Weitere Details werden später bei den Anwendungsbeispielen behandelt.

Der Datenaustausch zwischen dem Treiber und dem Benutzerprogramm erfolgt grundsätzlich über den zweiten Xenomai-Mechanismus – rtioctl, die Real-Time Alternative zum nativen Linux-ioctl Systemaufruf. Dabei stehen folgende Kommandos zur Verfügung:

- RTPCAN_IOCTL_WRITE_MSG
  schreibt eine zu sendende CAN-Nachricht in den TX-Buffer des Treibers.
- RTPCAN_IOCTL_READ_MSG
  liest eine CAN-Nachricht aus dem RX-Buffer des Treibers.
- RTPCAN_IOCTL_GET_STATUS

liest die Statusinformation von einem ausgewählten CAN-Controller (Siehe die „t_RtPcanStatus“ Struktur).

- RTPCAN_IOCTL_INIT
  initialisiert einen ausgewählten CAN-Controller erneut (Siehe die „t_RtPcanInit“ Struktur)
- RTPCAN_IOCTL_TXABORT
  fordert den Übertragungsabbruch für die aktuelle ausgehende CAN-Nachricht.

Der Datenaustausch erfolgt über die Instanzen von in der Datei „rtpcan.h“ definierten Strukturen:

- t_RtPcanMsg
  wird mit dem Rtioctl-Befehl _WRITE_MSG und _READ_MSG für den beidseitigen CAN-Nachrichtentransfer benutzt.

```
typedef struct
{
  union                    // 11/29 bit code
  {
    DWORD id;              // as double word
    BYTE  id_byte[4];      // as 4-bytes array
  };

  BYTE  type;              // MSGTYPE_ (STANDARD, EXTENDED, RTR, TXONCE)
  BYTE  len;               // count of data bytes (0..8)

  union                    // data bytes, up to 8
  {
    BYTE  data[8];         // as 8-bytes array
    QWORD data_u64;        // as quad word
  };

  QWORD timestamp;         // a timestamp in nanosec (valid only for rx)

} t_RtPcanMsg;
```

- t_RtPcanStatus
  wird für die Statusabfrage mit dem Rtioctl-Befehl RTPCAN_IOCTL_GET_STATUS verwendet.

```
typedef struct
{
  WORD  itype;           // the interface type (see INTERFACE_TYPE_...)
  WORD  ctype;           // the chip type (see CHIP_TYPE_...)
  DWORD io;              // the base address or port of this device
  WORD  irq;             // the irq level of this device
  DWORD bitrate;         // current bitrate (bits/s)
  DWORD irq_counter;     // counts all interrupts
  int   open_counter;    // number of „open“ calls for this device
} t_RtPcanStatus;        // for RTCAN_IOCTL_GET_STATUS
```

- t_RtPcanInit

  wird beim reinitialisieren des CAN-Controllers verwendet (RTPCAN_IOCTL_INIT)

```
typedef struct
{
  DWORD bitrate;          // [out] bitrate (bits/s)
  BYTE  extended;         // [out] 0, no extended frames are accepted
  BYTE  listenonly;       // 1/0 (true/false)
} t_RtPcanInit;           // for RTCAN_IOCTL_INIT
```

Wie man sehen kann, ist die Benutzer-API ziemlich kompakt. Möglicherweise ist es noch nicht ausgereift und kann deswegen in zukünftigen Versionen geändert werden. Aber die wichtigsten Kommunikationsmechanismen (rtioctl und events) werden beibehalten. Der Zugriff auf die Quellcodedateien kann einen detaillierteren Einblick in die Treiberstruktur verleihen.

Anwendungsbeispiele

Die Treiberbeschreibung währe ohne einiger Beispiele nicht vollständig. Hier werden einige Quellcode-Stücke vorgeführt, welche die Benutzung der Xenomai-Erweiterung und des Real-Time-Treibers im User-Space demonstrieren.

- Beispiel 1 – einfache Real-Time Anwendung:

```
#include <native/task.h>
#include <sys/mman.h>
#include <stdio.h>

void main()
{
  RT_TASK       task;   // Task Descriptor

  // Sperre  das  Paging  für alle Seiten, die in den Adressraum des
  // aufrufenden Prozesses eingebunden sind.

  mlockall(MCL_CURRENT | MCL_FUTURE);

  // Wandle den aktuellen Task in einen nativen Xenomai-Task
  // (Real-Time Task) um (RT-Priorität 51, FPU erlaubt).

  ret = rt_task_shadow(&task, NULL, 51, T_FPU);

  if (ret == 0 || ret == -EBUSY)
  {
    // Schalte in den primären (Real-Time) Modus um.
    // Ab jetzt kann ich von Linux nicht unterbrochen werden
    // (nur von Xenomai)!

    rt_task_set_mode(0, T_PRIMARY, NULL);

    // Initiiere einen nativen Linux Systemaufruf und verlasse somit
    // (möglicherweise unerwünscht) die primäre Domäne (Real-Time Modus).

    printf(„Hello, Real-Time World!\n“);
```

```
    // Jetzt sind wir in der sekundären (Non-Real-Time) Domäne
    // und werden von Linux gescheduled. Die nativen Linux-Systemaufrufe
    // sollen im Real-Time Modus vermieden werden, weil sie die Anwendung
    // zwingend in die sekundäre Domäne rausschmeißen. Die Rückkehr in
    // die Primäre Domäne ist nicht nur mit dem „rt_task_set_mode“
    // möglich. Auch manche Xenomai-Aufrufe zum RT-Treiber (wie z.B.
    // „rt_dev_ioctl“) können die Umschaltung in den Real-Time Modus
    // hervorrufen (hängt von der Treiberkonfiguration).
    // Jetzt schalten wir wieder in den Real-Time Modus.

    rt_task_set_mode(0, T_PRIMARY, NULL);

    ...
    // Lege die task für 1 Sekunde schlafen
    rt_task_sleep(1000000000L)

    ...
  }
}
```

- Beispiel 2 – CAN-Nachrichten Sender:

```
#include <native/task.h>
#include <native/event.h>
#include <stdio.h>
#include <sys/mman.h>
#include <rtpcan.h>

#define DEVICE  "rtpcan0"

int main()
{
  RT_TASK       task;     // Task Descriptor
  RT_EVENT      event;    // Event Flag Group Descriptor
  int           fd = 0;   // „File“ descriptor
  t_RtPcanMsg   msg;      // CAN-Nachricht
  int           ret;
  RTIME         time;
  unsigned long evmask;
  unsigned long evmask_r;
  int           i;

  // Sperre  das  Paging  für alle Seiten, die in den Adressraum des
  // aufrufenden Prozesses eingebunden sind.

  mlockall(MCL_CURRENT | MCL_FUTURE);

  // Wandle den aktuellen Task in einen nativen Xenomai-Task
  // (Real-Time Task) um (RT-Priorität 51, FPU erlaubt).

  ret = rt_task_shadow(&task, NULL, 51, T_FPU);

  if (ret == 0 || ret == -EBUSY)
  {
    // Schalte den Timer in aperiodischen/oneshot Modus um.
    // (in diesem Fall ist dieser Aufruf überflüssig)

    ret = rt_timer_set_mode(TM_ONESHOT);
```

```
if (ret == 0)
{
  // Öffne das CAN-Gerät

  if ((fd = rt_dev_open(DEVICE, 0)) >= 0)
  {
    // Binde „event“ mit der Event-Quelle

    if ((ret = rt_event_bind(&event, DEVICE, TM_NONBLOCK)) == 0)
    {
      // Speichere die Start-Zeit

      time = rt_timer_read();

      // Versuche zyklisch 100 CAN-Nachrichten zu senden

      for (i = 0; i < 100; i++)
      {
        // Prüfe die Schreiberlaubnis vom Treiber (nicht blockierend)

        evmask = RTPCAN_EV_WRITE;
        ret = rt_event_wait(&event,
                            evmask,
                            &evmask_r,
                            EV_ANY,
                            TM_NONBLOCK);

        if (ret == 0)
        {
          // Schreiberlaubnis erteilt.

          // Parametriere neue CAN-Nachricht

          msg.id       = 0;
          msg.type     = MSGTYPE_EXTENDED | MSGTYPE_TXONCE;
          msg.len      = 8;
          msg.data_u64 = i;

          // Sende die CAN-Nachricht

          ret = rt_dev_ioctl(fd, RTPCAN_IOCTL_WRITE_MSG, &msg);

          if (ret != 0)
            break;

          // Lösche die "manuellen" Event-Bits
          // (überflüssig in diesem Fall)

          rt_event_clear(&event, RTPCAN_EVGROUP_MANUAL, NULL);
        }

        // Lege den Task bis zum nächsten Zyklusschritt schlafen
        // (schlafe die restliche Periodenzeit (1s) aus)

        rt_task_sleep_until(time += 1000000000L);
      }

      // Trenne mich von der Event-Quelle

      rt_event_unbind(&event);
    }
```

```
        // Schlafe eine Weile vor dem Schließen

        rt_task_sleep(1000000000L);
      }
    }
  }

  // Schließe das geöffnete Gerät (falls geöffnet)

  if (fd >= 0)
  {
    if ((ret = rt_dev_close(fd)) < 0)
      printf("main: error while rt_dev_close -> %d\n");
  }

  printf("main: done.\n");
}
```

- Beispiel 3 – CAN-Nachrichten Empfänger:

```
#include <native/task.h>
#include <native/event.h>
#include <stdio.h>
#include <sys/mman.h>
#include <rtpcan.h>

#define DEVICE  "rtpcan0"

int main()
{
  RT_TASK       task;     // Task Descriptor
  RT_EVENT      event;    // Event Flag Group Descriptor
  int           fd = 0;   // „File“ descriptor
  t_RtPcanMsg   msg;      // CAN-Nachricht
  int           ret;
  RTIME         time;
  unsigned long evmask;
  unsigned long evmask_r;
  int           i;

  // Sperre  das  Paging  für alle Seiten, die in den Adressraum des
  // aufrufenden Prozesses eingebunden sind.

  mlockall(MCL_CURRENT | MCL_FUTURE);

  // Wandle den aktuellen Task in einen nativen Xenomai-Task
  // (Real-Time Task) um (RT-Priorität 51, FPU erlaubt).

  ret = rt_task_shadow(&task, NULL, 51, T_FPU);

  if (ret != 0 && ret != -EBUSY)
  {
    // Schalte den Timer in aperiodischen/oneshot Modus um.
    // (in diesem Fall ist dieser Aufruf überflüssig)

    ret = rt_timer_set_mode(TM_ONESHOT);

    if (ret != 0)
    {
      // Öffne das CAN-Gerät
```

```
if ((fd = rt_dev_open(DEVICE, 0)) < 0)
{
  // Binde „event“ mit der Event-Quelle

  if ((ret = rt_event_bind(&event, DEVICE, TM_NONBLOCK)) == 0)
  {
    // Speichere die Start-Zeit

    time = rt_timer_read();

    // Versuche zyklisch bis zu 1000 CAN-Nachrichten zu lesen

    for (i = 0; i < 1000; i++)
    {
      // Prüfe die Leseerlaubnis vom Treiber (nicht blockierend)

      evmask = RTPCAN_EV_READ;
      ret = rt_event_wait(&event,
                          evmask,
                          &evmask_r,
                          EV_ANY,
                          TM_NONBLOCK);

      if (ret == 0)
      {
        // Leseerlaubnis erteilt.
        // Lese die angekommenen CAN-Nachricht (und kehre
        // automatisch in den primären (RT) Modus zurück)

        ret = rt_dev_ioctl(fd, RTPCAN_IOCTL_READ_MSG, &msg);

        // !!! Nur für Beispielzwecke !!!
        // Zeige die empfangene CAN-Nachricht an und
        // verlasse somit den primären Modus!

        if (ret == 0)
        {
          printf("main: 0x%02x (%s) 0x%04x %d [%08x]\n",
                 msg.type,
                 (msg.type == MSGTYPE_STANDARD)? "std": "ext",
                 msg.id,
                 msg.len,
                 msg.data_u64);
        }

        // Lösche die "manuellen" Event-Bits
        // (überflüssig in diesem Fall)

        rt_event_clear(&event, RTPCAN_EVGROUP_MANUAL, NULL);
      }

      // Lege den Task bis zum nächsten Zyklusschritt schlafen
      // (schlafe die restliche Periodenzeit (0.1s) aus)

      rt_task_sleep_until(time += 100000000L);
    }

    // Trenne mich von der Event-Quelle

    rt_event_unbind(&event);
  }
}
```

```
    }
  }

  // Schließe das geöffnete Gerät (falls geöffnet)

  if (fd >= 0)
  {
    if ((ret = rt_dev_close(fd)) < 0)
      printf("main: error while rt_dev_close -> %d\n");
  }

  printf("main: done.\n");
}
```

Zusätzliche lauffähige Beispiele lassen sich im „test“-Verzeichnis finden.

Laufzeit-Test

Nach der Treiberimplementierung wurden einige Tests durchgeführt, um das ganze Konzept zu überprüfen und die Treibereigenschaften zu ermitteln. Es wurde ein „Master – Slave“ Test speziell für die Ermittlung von Latenzzeiten erstellt (die Dateien „`t3_master.c`“ und „`t3_slave.c`“ im „test“ Verzeichnis). Der „Master“ sendet eine CAN-Nachricht und misst dabei eigene Sendelatenz (die Zeit *m_dt_tx*, die zwischen der „write“-Operation und dem Absenden der Nachricht vergeht). Der Slave empfängt diese Nachricht und misst sofort eigene Empfangslatenz *s_dt_rx* (die Zeit, die zwischen dem Eintreffen der Nachricht im internen RX-Buffer des CAN-Controllers und dem Zeitpunkt sofort nach der „read“-Operation liegt). Slave antwortet mit zwei CAN-Nachrichten. Mit der ersten wird die Slave-Empfangslatenz (*s_dt_rx*) abgesendet und gleichzeitig eigene Sendelatenz (*s_dt_tx*) gemessen, welche sofort mit der zweiten Nachricht abgeschickt wird. Der Master empfängt beide Nachrichten und misst gleichzeitig eigene Empfangslatenz (*m_dt_rx*). Somit kennt der Master nach einer Runde alle Latenzzeiten und die aktive Rundenzeit (*m_dt_tx, m_dt_rx, s_dt_tx, s_dt_rx, t_round*) (Abb. 5).

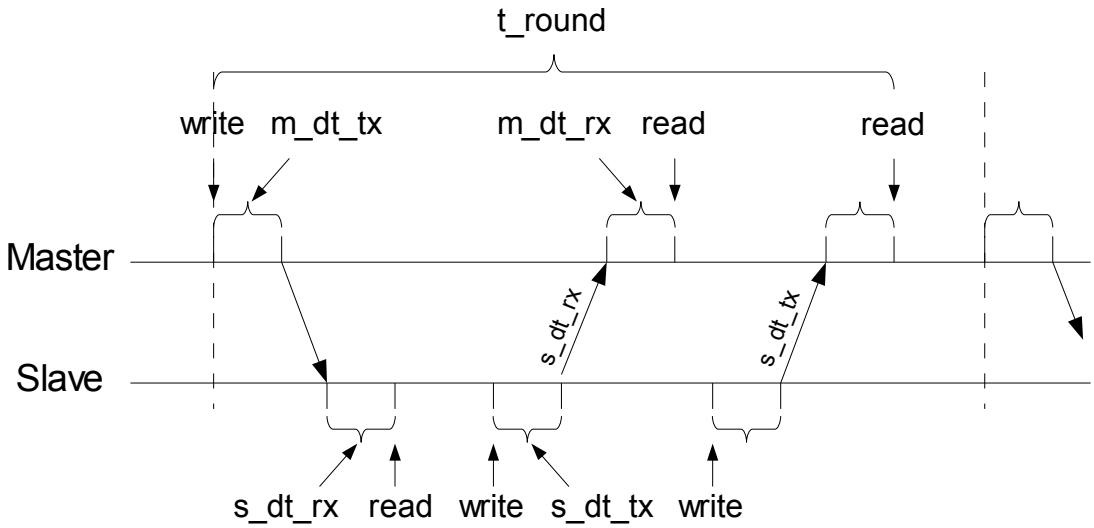

Abb. 5: "Master-Slave" Latenztest

Die Tests wurden bei unterschiedlichen Rechner-Belastungskombination durchgeführt, um den Einfluss einer Non-Real-Time-Last zu zeigen. Die CPU wurde mit der Komprimierung eines Verzeichnisses belastet. Somit konnte sowohl die CPU als auch das I/O Subsystem belastet werden. Zur besseren Übersicht werden alle Kurven im gleichen Maßstab dargestellt.

- Master-Slave Latenzzeiten (CPU unbelastet)

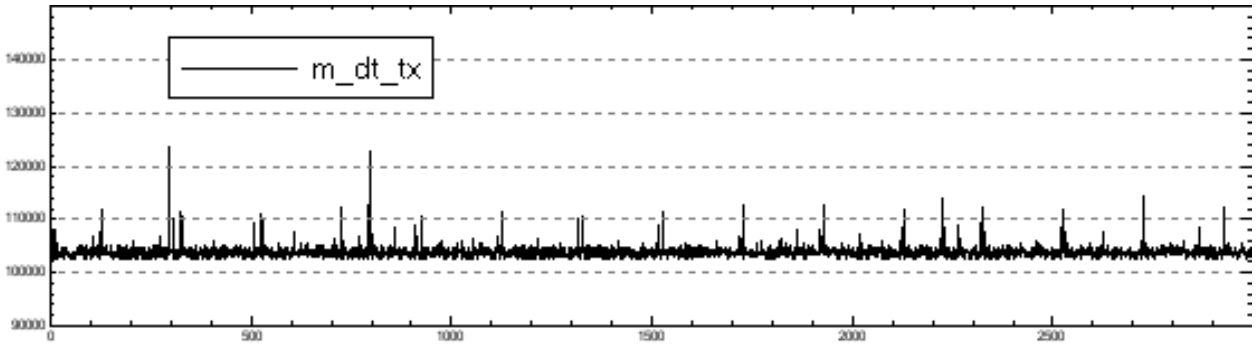

Abb. 6: Master TX-Latenzzeit (CPU unbelastet). Y-Skala 90...150 us

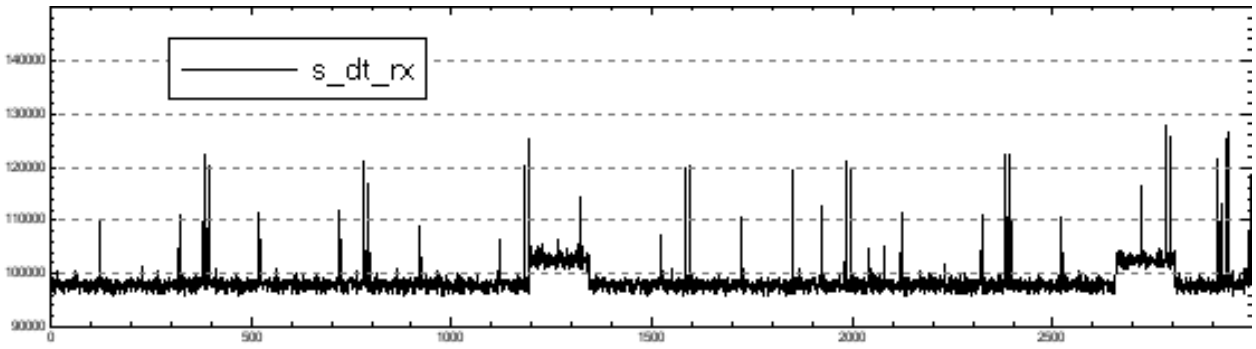

Abb. 7: Slave RX-Latenzzeit (CPU unbelastet). Y-Skala 90...150 us

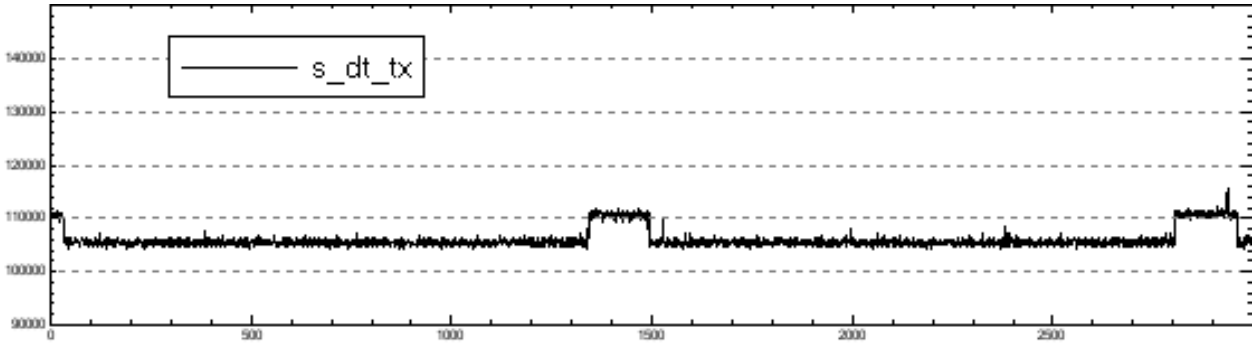

Abb. 8: Slave TX-Latenzzeit (CPU unbelastet). Y-Skala 90...150 us

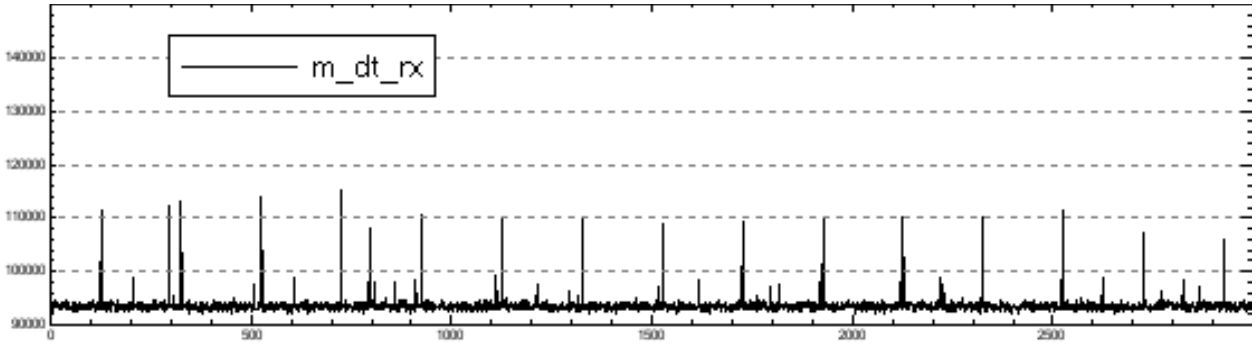

Abb. 9: Master RX-Latenzzeit (CPU unbelastet). Y-Skala 90...150 us

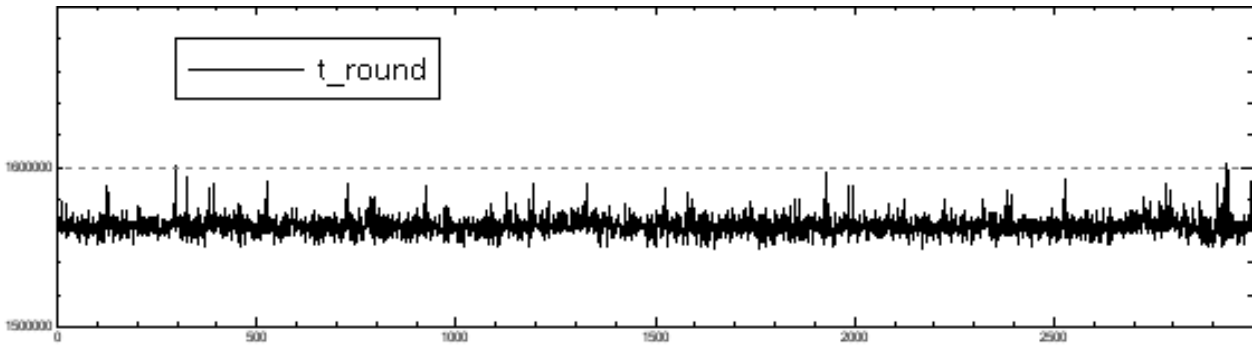

Abb. 10: Master-Slave Rundenzeit (CPU unbelastet). Y-Skala 1500...1700 us

- Master-Slave Latenzzeiten (CPU 100% belastet)

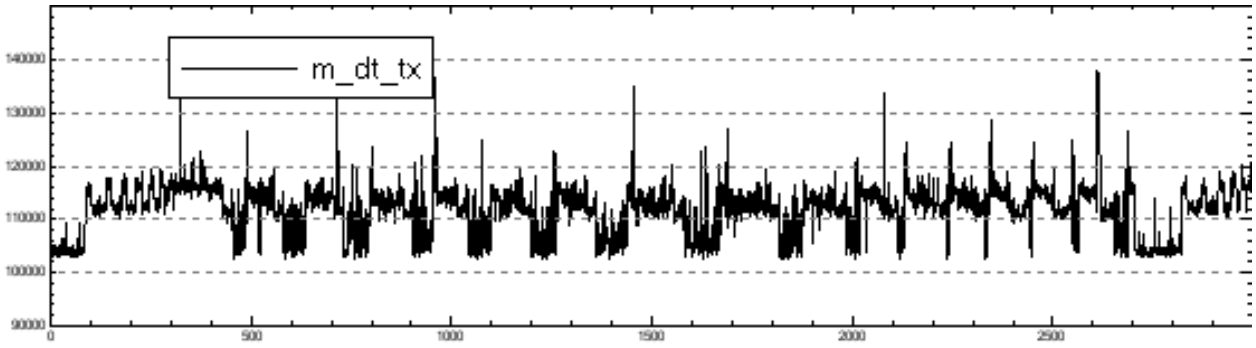

Abb. 11: Master TX-Latenzzeit (CPU 100% belastet). Y-Skala 90...150 us

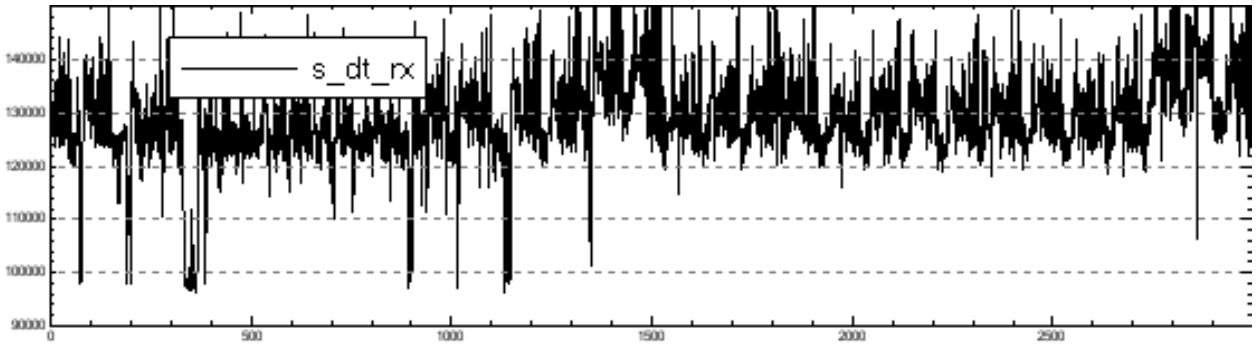

Abb. 12: Slave RX-Latenzzeit (CPU 100% belastet). Y-Skala 90...150 us

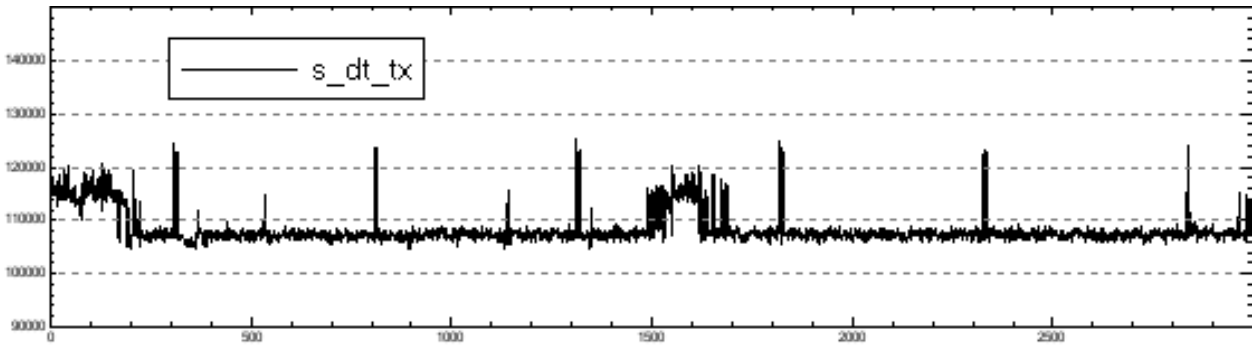

Abb. 13: Slave TX-Latenzzeit (CPU 100% belastet). Y-Skala 90...150 us

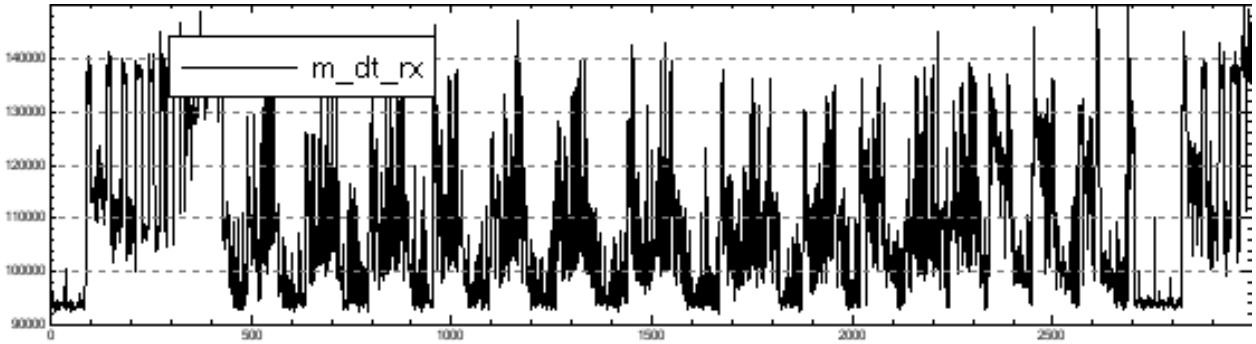

Abb. 14: Master RX-Latenzzeit (CPU 100% belastet). Y-Skala 90...150 us

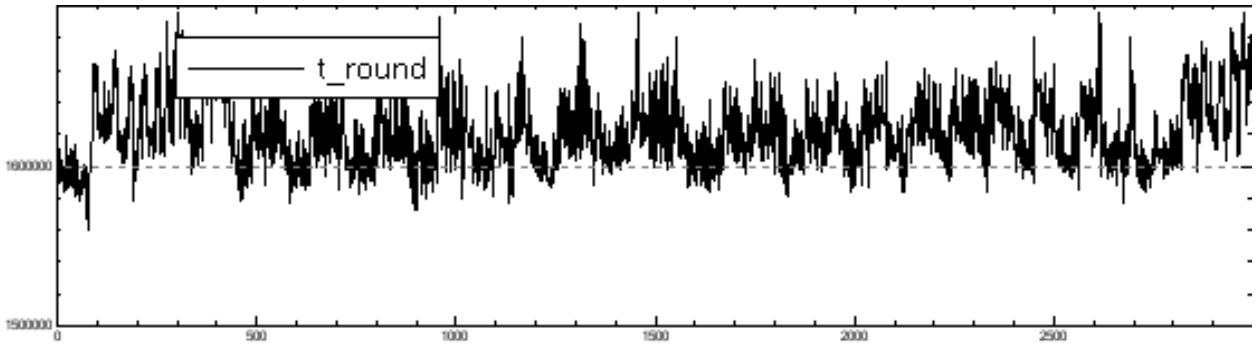

Abb. 15: Master-Slave Rundenzeit (CPU 100% belastet). Y-Skala 1500...1700 us

Es ist deutlich zu sehen, dass alle TX/RX-Latenzzeiten in beiden Fällen im Bereich von 90...150 us liegen, was im wesentlichsten von der Geschwindigkeit der parallelen Schnittstelle abhängig ist. Die CAN-Dongles wurden im ECP Modus getestet. Das Umschalten in SP-Modus würde die Latenzzeiten fast verdoppeln (hier nicht gezeigt).

Im unbelasteten Zustand sehen die Latenzzeiten relativ stabil aus, obwohl schon einige

sporadische und periodische Störungen sichtbar sind. Möglicherweise ist es dadurch zu erklären, dass die Xenomai-Erweiterung nicht in der Lage ist manche Linux-Prozesse zu unterbrechen und manche Interrupts abzudecken. Die Störungen steigen weiter bei der Vollbelastung der CPU mit einem nativen Linux-Prozess. Möglicherweise werden diese Störungen minimiert, wenn das I/O Subsystem (z.B. Festplatten- und Grafik-Aktivität) ausgeschlossen wird. Trotz der Störungen bleiben die Latenzzeiten immer noch in einem akzeptablen relativ schmalen Bereich von etwa 60 us.

Diese Tests zeigen eine akzeptable Treiberperfomance, lassen aber vermuten, dass eventuell noch einige Treiber-Verbesserungen im Bezug auf die Stabilität von Latenzzeiten vorgenommen werden können.

Zusammenfassung

Das Ziel dieser Arbeit bestand in der Entwicklung eines echtzeitfähigen CAN-Treibers für die Linux-Variante der COSMIC-Middleware – einer CAN-Kommunikationssoftware, die nach dem „Publisher-Subscriber“ Prinzip funktioniert. Dementsprechend entstand die Frage nach einer passenden Echtzeit-Erweiterung für Linux. Nach dem Betrachten einiger Varianten wurde die Wahl zugunsten der Xenomai-Erweiterung getroffen. Als Hardwareplattform wurde der CAN-Dongle der Firma „Peak System“ genommen. Die ersten RT-Treiberversionen ließen sich von der Non-Real-Time Version des nativen Linux-Treibers für dieses Gerät ableiten und haben von ihm die wesentlichen Merkmale geerbt. In weiteren Versionen wurde die Treiber-API in Richtung Xenomai-spezifischer Mechanismen verschoben. Der RT-Treiber wurde an einer Reihe von Tests ausprobiert und hat eine ziemlich gute Performance gezeigt. Es wurden natürlich auch einige Problemstellen aufgedeckt, sowohl in der Software, als auch in der bestehenden Hardware. Die Xenomai-Erweiterung war nicht in der Lage die totale Kontrolle über den Rechner zu erhalten. Einige Interruptquellen konnten nicht abgedeckt werden. Das Problem liegt sowohl in der auf dem Rechner benutzten Software, die in manchen Fällen einige Interrupts blockieren kann, als auch in der Rechner-Hardware. Dieses Problem gilt möglicherweise für alle existierenden RT-Linux-Erweiterungen, weil bestimmte Verzögerungen nur von der Rechner-Hardware verursacht werden und sich nicht softwaremäßig umgehen lassen. Das Problem lässt sich in manchen Fällen vollständig vermeiden, indem die Störquellen lokalisiert und deaktiviert werden. Der Treiber selbst hat einige Instabilitäten bei den Sende- und Empfangs-Latenzzeiten gezeigt, die bei steigender CPU-Last seitens nativer Linux-Prozesse gestiegen sind. Die Software-Probleme werden sich möglicherweise durch zukünftige Weiterentwicklungen abschwächen lassen. Der ausgewählte CAN-Dongle hat sich als das „schwächste“ Mitglied in der Performance-Kette gezeigt. Die hohen Sende- und Empfangs-Latenzzeiten, bedingt durch die parallele Schnittstelle, stellen die Frage der Einsatzmöglichkeit dieses Gerätes in der COSMIC-Middleware. Die Lösung würde eine andere CAN-Controller-Variante anbieten, die über eine schnellere Schnittstelle (z.B. über USB oder PCMCIA) mit dem Rechner verbunden ist.

Allgemein lässt sich die Arbeit als gelungen betrachten, weil der implementierte RT-Treiber mit dem vorhandenen CAN-Dongle die Konzeptprüfung ermöglicht und einige Schwachstellen aufgedeckt hat. Die Xenomai-Erweiterung hat sich dabei als nützliche und komfortable Real-Time Ergänzung für Linux gezeigt.

Quellen

[1] COSMIC: A real/time event-based middleware for the CAN-bus.
Jörg Kaiser, Christiano Brudna, Carlos Mitidieri.
Department of Computer Structures, University of Ulm, Ulm, Germany

[2] Linux-Treiber entwickeln.
Jürgen Quade. Dpunkt Verlag. ISBN: 3898642380

[3] Linux Device Drivers, Third Edition
http://lwn.net/Kernel/LDD3/

[4] SJA1000: Stand-alone CAN controller. Philips Semiconductors. 2000.01.04
http://coecsl.ece.uiuc.edu/ge420/datasheets/SJA1000_3.pdf

[5] http://www.xenomai.org

[6] http://www.captain.at/xenomai.php

[7] http://www.peak-system.com/index_de.html