

Sprach-Mapping von PEARL auf die Linux-Systemschnittstelle

Holger Kölle

Hochschule Furtwangen, 78120 Furtwangen
Holger.Koelle@koelle-ohg.de

Zusammenfassung. Dieser Artikel beschreibt die Machbarkeit einer Sprachabbildung, auch Sprach-Mapping genannt, der Echtzeitprogrammiersprache PEARL auf die Programmiersprache C, mit Linux als Laufzeitumgebung. Untersucht wird, ob und falls ja, wie die wichtigsten Laufzeitkonstrukte von PEARL (Scheduling, Taskkonzept, Semaphoren und Interrupts) auf die Programmiersprache C abbildbar sind. Analysiert werden diese auf einem Standard-Linux, einem Standard-Linux mit preemptive priority scheduling und einem Linux mit Echtzeitkernel (Xenomai). Grundsätzlich ist die Sprachabbildung von PEARL auf die Linux Systemschnittstelle möglich. Die Mindestvoraussetzung dafür ist ein Linux mit preemptive priority scheduling. Allerdings müssen dann sowohl Zeit- als auch Interruptsteuerung selbst implementiert werden. Auch eine Abbildung auf Xenomai ist denkbar. Allerdings erfordert diese deutlich mehr Aufwand, sowohl bei der Implementierung als auch bei der späteren Wartung.

1 Einleitung

Die Programmiersprache PEARL wurde im Jahr 1981 erstmals von der DIN genormt. Im späteren Verlauf wurde die Sprachnorm wesentlich erweitert und nicht praxisrelevante Teile entfernt. Diese Arbeit basiert auf der aktuellen Sprachnorm mit der Bezeichnung „PEARL 90“.

Der Name PEARL steht für „**P**rocess and **E**xperiment **A**utomation **R**ealtime **L**anguage“. Zielsetzung der Sprache ist eine leichte Erlernbarkeit und eine komfortable Programmierung von Echtzeit Multitasking Systemen [1].

Da PEARL für Multitasking konzipiert wurde, verfügt es natürlich auch über entsprechende sprachliche Mittel, wie beispielsweise Semaphoren oder Tasks.

In folgender Abbildung ist dargestellt, wie einfach es mit PEARL ist, komplexere Anwendungen zu realisieren. Die beiden Produzententasks können nur in den Puffer schreiben, wenn der Ausgabetask die entsprechende Semaphore freigegeben hat. Andererseits kann der Ausgabetask auch nur etwas aus dem Puffer ausgeben, wenn zuvor von den Produzenten hineingeschrieben wurde.

Ein einführendes Beispiel der sprachlichen Mittel von PEARL :

PROBLEM;

DCL (In_den_Puffer,Aus_dem_Puffer) **SEMA**

Ausgabe: **TASK;**

```

    RELEASE In_den_Puffer;
    ACTIVATE Produzent_1;
    ACTIVATE Produzent_2;
    REPEAT
        REQUEST Aus_dem_Puffer;
        !Ausgabe auf den Protokolldrucker
        RELEASE In_den_Puffer;
    END; !Schleife
END; !Ausgabe

```

Produzent_1: **TASK ;**

```

    REPEAT
        !Aufbereiten der Nachricht
        REQUEST In_den_Puffer;
        !Puffern
        RELEASE Aus_dem_Puffer;
    END; !Schleife
END; !Produzent_1

```

Produzent_2: **TASK ;**

```

    REPEAT
        !Aufbereiten der Nachricht
        REQUEST In_den_Puffer;
        !Puffern
        RELEASE Aus_dem_Puffer;
    END; !Schleife
END; !Produzent_2

```

Abb. 1. Einführungsbeispiel, entnommen aus [2]

2 Zielsetzung

In dieser Arbeit soll analysiert werden, ob eine Sprachabbildung der wichtigsten Laufzeitkonstrukte der Programmiersprache PEARL auf die Systemschnittstelle von Linux mittels der Programmiersprache C möglich ist.

Desweiteren können die Ergebnisse dieser Arbeit als Entscheidungsgrundlage dienen, ob es sinnvoll ist, einen Compiler für diesen Zweck zu entwickeln.

Zusätzlich wird aufgezeigt, welche Mindeststandards erfüllt sein müssen, falls die Sprachabbildung möglich ist. Generell wird unter drei Linux Systemschnittstellen unterschieden, auf denen die Machbarkeit analysiert wird:

- Linux mit Standardscheduling
- Linux mit preemptive priority scheduling
- Linux mit einem Echtzeitkernel (Xenomai¹)

Unter einem Linux mit Standardscheduling (Standard-Linux) wird in dieser Arbeit eine Linuxdistribution verstanden, wie sie direkt nach der Installation aus Benutzersicht ohne Modifikation ist. Dabei wird im Benutzermodus der „Completely Fair Scheduler“ benutzt.

Linux mit preemptive priority scheduling bezeichnet hier ein Standard-Linux mit dem Unterschied, dass die Prozesse bzw. Threads nach einem Echtzeitverfahren (preemptive priority) dem Prozessor zugeteilt werden. Sie werden nicht nach dem Completely Fair Scheduling-Prinzip abgearbeitet. PEARL setzt im wesentlichen harte Echtzeit voraus. In dieser Arbeit liegt der Schwerpunkt allerdings auf dem Zweck der Lehre, daher wird hier nicht zwingend harte Echtzeit vorausgesetzt.

Die zu analysierenden Sprachkonstrukte sind:

- Scheduling
- Taskkonzept
- Zeitverhalten und zeitliche Steuerung von Tasks
- Semaphoren
- Interrupts

Andere Konstrukte werden hier nicht betrachtet.

3 Analyse

3.1 Scheduling

Da PEARL für harte Echtzeit konzipiert wurde, arbeitet der PEARL Scheduler entsprechend nach dem preemptiv priority-Prinzip. Dabei werden den einzelnen Tasks Prioritäten zugeordnet, nach denen sie abgearbeitet werden. Tasks mit niedriger Priorität werden dabei von Tasks mit höherer Priorität unterbrochen, bis diese blockieren, sich schlafen legen oder abgearbeitet sind [2].

¹ <http://www.xenomai.org>

Das Linux mit Standardscheduling arbeitet nur mit dem „Completely Fair Scheduler“. Dieser arbeitet nicht mit Prioritäten, sondern versucht, alle Prozesse möglichst gleich zu behandeln. Aus diesem Grund ist er nicht für Echtzeitanwendungen geeignet. Daher wird das Linux mit Standardscheduling in dieser Arbeit nicht weiter verfolgt [5].

Beim Linux mit preemptive priority scheduling (nachfolgend „Linux“ genannt) besteht die Möglichkeit, Prozesse und Threads nach verschiedenen Echtzeitverfahren abzuarbeiten.

Speziell die Option „`SCHED_FIFO`“ ist hier interessant, weil sie sehr gut auf das Verhalten des PEARL-Schedulers passt.

Beim Linux mit Echtzeiterweiterung (nachfolgend als „Xenomai“ bezeichnet) besteht ebenfalls wie beim Linux die Möglichkeit, zwischen mehreren Echtzeitverfahren zu wählen. Standardmäßig kann hier auch präemptives Prioritätsscheduling eingesetzt werden. Daher bereitet auch bei Xenomai die Sprachabbildung des Scheduling keine Probleme [3].

3.2 Taskkonzept

PEARL bietet sehr vielfältige Optionen um Tasks zu steuern. Sie können sich nach ihrer Erzeugung gegenseitig

- Starten mit Prioritätsangabe.
- Beenden.
- Anhalten/suspendieren.
- Fortsetzen (evtl. mit Prioritätsänderung).
- Verzögern.

Grundsätzlich bieten sich POSIX-Threads für die Abbildung des Taskkonzepts am ehesten an. Denn PEARL-Tasks können genau wie Threads auf alle Variablen innerhalb ihres Moduls zugreifen. Wollte man es auf Prozesse abbilden, müsste man zusätzlich eine Art der Interprozesskommunikation realisieren, was zusätzlichen Aufwand bedeuten würde [2, 5].

Bei Xenomai hingegen bieten sich die Xenomai-Echtzeittasks an. Auch sie können untereinander über gemeinsame Variablen kommunizieren [3].

3.3 Semaphoren

PEARL bietet nur zählende Semaphoren. Binäre Semaphoren (Mutexe) werden einfach durch die Initialisierung auf 1 der zählenden Semaphore realisiert. Die in PEARL verfügbaren BOLT-Variablen wurden hier nicht untersucht [2]. Beim Linux bieten sich dafür drei Alternativen, benannte Semaphoren, unbenannte Semaphoren und binäre Semaphoren [5]. Da eine zählende Semaphore nicht auf eine binäre Semaphore abbildbar sein kann, wird diese nicht weiter untersucht. Von den beiden verbliebenen Varianten ist die unbenannte Semaphore die bessere Wahl. Sie wird beim Beenden des Prozesses vom System zerstört. Die Benannte dagegen wird erst durch einen expliziten Aufruf oder beim Herunterfahren des

Systems zerstört [5]. Da später die Threads, die PEARL-Tasks nachbilden, alle in einem Prozess laufen, genügt die Unbenannte voll und ganz. Sie unterstützt alle Funktionen, die die PEARL-Semaphore auch bietet.

Xenomai unterstützt ebenfalls Mutexe und zählende Semaphoren. Hier sind die zählenden Semaphoren interessant, denn auch sie bieten die benötigten Funktionen. Allerdings gibt es bei PEARL eine Spezialität. Ein PEARL-Task kann gleichzeitig auf mehrere Semaphoren warten. Er wird solange blockiert, bis alle Semaphoren, auf die er wartet, frei sind. Beim Linux funktioniert das auch, allerdings mit einer Einschränkung. Es können pro Semaphorensatz maximal 32 Semaphoren gleichzeitig inkrementiert bzw. dekrementiert werden. Allerdings kann dies verändert werden durch eine Kernelvariable [5]. Bei Xenomai ist das schon ein erster Rückschlag. Das „Mehrfachwarten“ kann hier nicht durch Xenomai-Funktionen realisiert werden. Man muss es selbst implementieren.

3.4 Zeitliche Steuerung von Tasks

Linux bietet bei weitem nicht so ausgefeilte Optionen zur Steuerung von Threads mit und ohne zeitliche Abhängigkeit. PEARL dagegen bietet außer den oben genannten Möglichkeiten zur Steuerung von Tasks auch viele zeitliche Steuerungsmöglichkeiten. Die Tasks können verzögert werden bis zu einem Zeitpunkt oder für eine Zeitdauer. Desweiteren können sie periodisch ausgeführt werden und optional kann als Endbedingung auch wieder ein Zeitpunkt oder eine Zeitdauer angegeben werden. Die Start-, Perioden- und Endbedingungen können beliebig miteinander verknüpft werden. Sie gelten allerdings nur für das Starten eines beendeten Tasks. Desweiteren können Tasks sich selbst für eine Zeitdauer oder bis zu einem Zeitpunkt suspendieren. Zu guter Letzt können Tasks auch andere suspendierte Tasks nach einer Zeitdauer oder zu einem Zeitpunkt aufwecken, gegebenenfalls mit einer anderen Priorität [2]. Weil Linux das schlichtweg nicht kann, muss hier nachgeholfen werden. Es bietet sich an, einen Thread einzuführen, der die Zeitsteuerung für alle anderen Threads übernimmt (nachfolgend als „Verwaltungsthread“ bezeichnet). Als Timer können die POSIX-Timer verwendet werden. Sie bauen seit Kernelversion 2.6.16 [6] auf dem High-Resolution-Timer-Framework auf. Sie unterstützen Auflösungen bis zu einer Nanosekunde und periodische Ausführung.

Auch Xenomai bietet nicht alle Möglichkeiten, die benötigt werden, zur Realisierung der Sprachabbildung. Deshalb bietet sich auch hier oben erwähnter Verwaltungsthread (bzw. bei Xenomai Verwaltungstask) an.

3.5 Interrupts

PEARL unterstützt auch die Tasksteuerung in Abhängigkeit von Hardware- und Softwareinterrupts [2]. Moderne Betriebssysteme gestatten aber in der Regel keinen direkten Zugriff auf die Hardware-Interrupts (IRQ-Leitungen). Daher muss auch hier ein anderer Weg gegangen werden. Da für die Zeitsteuerung sowieso ein Verwaltungsthread eingeführt werden musste, kann dieser praktischerweise auch die Interruptsteuerung übernehmen. Er kann sich z.B. vom Kernel informieren

lassen, falls auf einem der Gerätetreiber Daten vorhanden sind z.B. „SELECT“-Aufruf [5]. Sind Daten auf einer der gewünschten Schnittstellen vorhanden, kann er diese auslesen und entsprechende Aktionen einleiten.

Bei Xenomai bietet sich ebenfalls der Verwaltungstask an. Allerdings besteht hier die Option, auch eigene Echtzeittreiber zu implementieren. Allerdings ist das ein hoher Aufwand. Im Hinblick darauf, dass die „normalen“ Linuxkernel ständig in ihrer Echtzeitfähigkeit verbessert wurden und in der Zukunft sicher noch werden, stellt sich die Frage, ob Xenomai in Zukunft überhaupt noch nötig sein wird. Daher ist es wahrscheinlich nicht empfehlenswert, für Xenomai extra einen anderen Weg zu gehen als beim Linux. Zumindest für Projekte wie dieses, bei denen es um den Lehrzweck geht.

3.6 Fazit

Das Linux mit Standardscheduling schied schon sehr früh aus, da es schon an der Abbildung des Taskkonzepts scheiterte.

Ein Teil der essentiellen PEARL Sprachkonstrukte (Scheduling, Semaphoren, Taskkonzept) lässt sich problemlos auf die Systemschnittstelle eines Linux mit preemptive priority scheduling und auf Xenomai abbilden. Die Task-, Interrupt- und Zeitsteuerung ist aber auf keines von beiden ohne Hilfsmittel direkt abbildbar. Deswegen entstand die Idee, einen Verwaltungsthread einzuführen, der die Verwaltung von Interrupt- und Zeitsteuerung zentral für die anderen Threads übernimmt. Für Xenomai bietet sich für die Zeitsteuerung zusätzlich die Möglichkeit, sie teilweise Xenomaispezifisch zu implementieren und teilweise in den Verwaltungsthread auszulagern. Zumindest müssen die Optionen, die Xenomai nicht direkt unterstützt, ausgelagert werden. Allerdings sind bei dieser Variante die Bedenken deutlich schwerer als der Nutzen, daher wurde auf diese Variante nicht weiter eingegangen. Konkret bedeutet das, dass sowohl die Interrupt- als auch die Zeitsteuerung bei beiden Systemen komplett auf einen Verwaltungsthread ausgelagert werden.

4 Design

Ein einziger Verwaltungsthread, wie im Analysekapitel eingeführt, wird für die komplette Interruptsteuerung nicht ausreichen. Er müsste auf allen Schnittstellen, für die Interrupts registriert werden, zyklisch nach Daten fragen, diese auswerten, gegebenenfalls entsprechende Aktionen einleiten und noch die Zeitsteuerung übernehmen. Das alles wäre sehr viel für einen Thread, was das ganze unstrukturiert und unübersichtlich machen würde.

Daher ist die Idee, für jede Schnittstelle, für die ein Interrupt registriert wird, statisch einen eigenen Thread zu erzeugen. Dieser holt die Daten von seiner ihm zugeordneten Schnittstelle ab und schreibt sie in eine Message Queue. Der Verwaltungsthread selbst braucht dann „nur“ noch die Daten aus der Message Queue zu lesen und hat die Interrupts dann auch gleich in der Reihenfolge wie sie aufgetreten sind. Das Modell für Linux ist in Abb. 2 dargestellt.

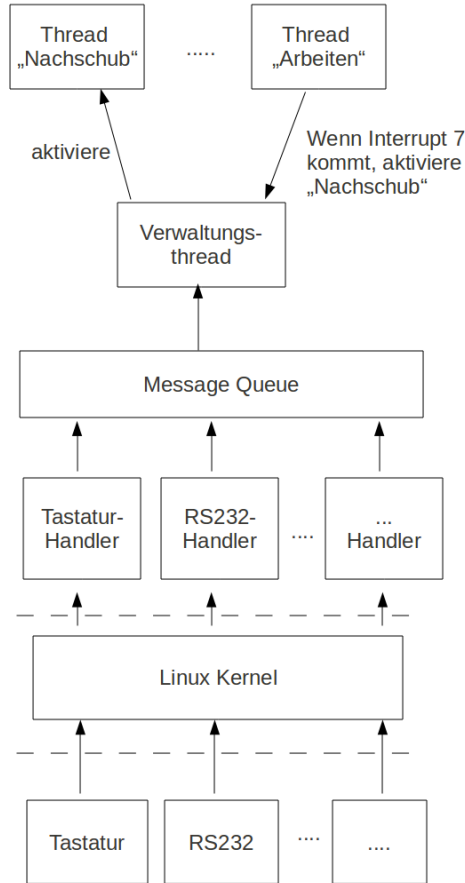


Abb. 2. Blockdiagramm Interrupt- und Zeitsteuerung Linux

Für Xenomai unterscheidet sich das Blockdiagramm in wenigen Punkten. Die Message Queue wird ersetzt durch eine Xenomai spezifische Real-Time Message Queue. Das ist nötig, denn die Handlerthreads laufen auf dem Linuxkernel, der Verwaltungs- und die Arbeitsthreads hingegen auf dem Xenomai Echtzeitkernel. Arbeitsthreads sind Threads, die aus den PEARL-Tasks erzeugt werden. Die Handlerthreads können nur über eine Real-Time Message Queue mit dem Verwaltungsthread kommunizieren. Außerdem garantiert diese, dass die Interrupts, die von den Handlern in die RT-Message Queue geschrieben werden, in chronologischer Reihenfolge beim Verwaltungsthread ankommen.

Das Verwaltungsthreadmodell für Xenomai ist in Abb. 3 dargestellt. Die Module, die auf den Echtzeitkernel verschoben wurden, sind grau hinterlegt.

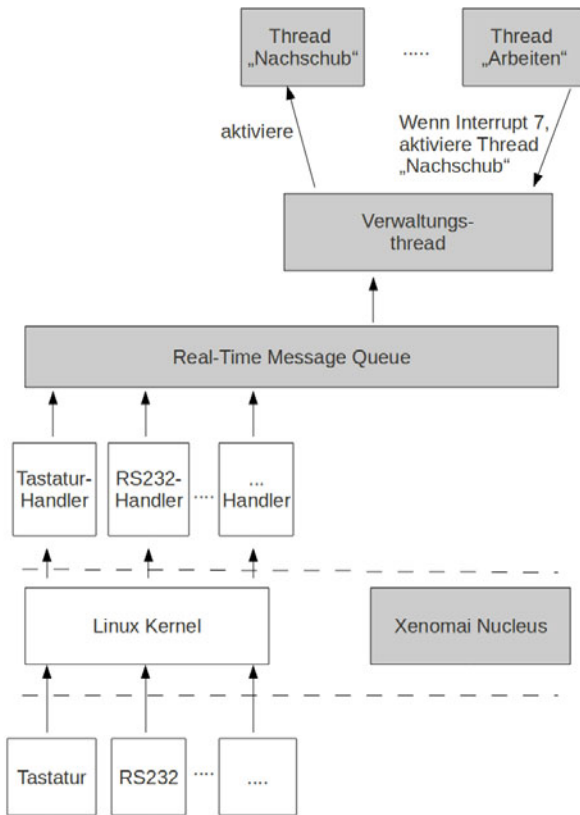


Abb. 3. Blockdiagramm Interrupt- und Zeitsteuerung Xenomai

5 Implementierung eines Testprogramms (Linux)

Um nun die Aussagen von Abschnitt 3 zu testen, wurde ein kleines Testprogramm für Linux entwickelt. Das Testsystem ist ein Hewlett Packard Modell dc5850 mit einem Ubuntu 10.04 als Betriebssystem. Die Kernelversion ist 2.6.32-38. Alle verwendeten Systemfunktionen sind bis auf wenige Ausnahmen nach dem POSIX-Standard.

Die Software führt im wesentlichen drei Tests durch. Getestet wird das Prioritätsscheduling anhand von vier Threads. Im Anschluss wird geprüft, ob die Semaphorwarteschlange nach Chronologie oder nach Prioritäten geordnet ist. Das ist wichtig zu wissen, da die Semaphorwarteschlange von PEARL strikt nach Prioritäten sortiert ist. Zu guter Letzt wird das Mehrfachwarten anhand des Philosophenproblems durchgeführt.

In der Initialisierungsphase wird die Anzahl der CPUs, die den Prozess bearbeiten dürfen, auf 1 gesetzt. Damit wird erreicht, dass die Threads strikt nacheinander in einen Puffer schreiben. Jeder der Threads schreibt seine Kennung (0 bis 3) jeweils zehn mal in den Puffer. Dieser wird nach den Schreibvorgängen mit einem Referenzarray abgeglichen. Wenn das Prioritätsscheduling funktioniert, stimmt die Reihenfolge des Puffers und des Referenzarrays überein. Damit gilt der erste Test als bestanden. Der zweite Test ist eine Erweiterung des Ersten. Hier sind die Schreibvorgänge mit einer Semaphore abgesichert. Diese Semaphore ist die ersten 20 Sekunden gesperrt. Sie wird erst dann von einem niederpriorisierten Thread freigegeben. Damit sollen die vier schreibenden Threads genug Zeit haben, sich in die Semaphorwarteschlange einzureihen. Nach der Semaphorfreigabe ist der restliche Ablauf gleich dem des ersten Tests. Die letzte Prüfung ist dagegen nicht ganz so einfach. Anhand des Philosophenproblems wird hier das Warten auf mehrere Semaphoren implementiert. Dabei nimmt jeder Philosoph immer zwei Gabeln (Semaphoren). Daher dürfte es sich nie verklemmen. Um allerdings zu testen, dass sich der Test nie verklemmt, müsste er bis in alle Ewigkeit laufen. Deshalb muss diese Prüfung nach Sicht beurteilt werden. Es obliegt der Verantwortung des Nutzers, wann er die Software abbricht.

Die Testsoftware soll es erleichtern, sich einen Überblick über die Gegebenheiten eines Systems zu verschaffen. Anhand der Ergebnisse kann entschieden werden, ob es sich lohnt, sich intensiver mit einem System zu beschäftigen.

6 Implementierung einer Zeitsteuerung (Linux)

6.1 Grundlegende Tasksteuerung

Um zu testen, ob eine Zeitsteuerung nach dem vorgestellten Design funktionieren kann, wurde diese implementiert. Im ersten Teil wurden die generellen Tasksteuerungsoptionen ohne Zeitsteuerung gebaut, d.h. Threads anhalten, fortsetzen, aktivieren und abbrechen. Aktivieren entspricht einem Aufruf von „pthread_create(..)“ und abbrechen einem „pthread_cancel(..)“. Für das Anhalten und Fortsetzen hingegen musste getrickst werden. Jeder der Threads besteht aus einer Struktur mit verschiedenen Attributen, eines davon ist ein Mutex. Dieser wird bei der Initialisierung belegt. Zusätzlich bekommt er einen Signalhandler registriert für das Signal „SIGUSR1“. Will ein anderer Thread ihn nun suspendieren, schickt er ihm einfach das Signal. Damit springt er in seinen Signalhandler, wo er versucht, seinen Mutex nochmals zu belegen und somit blockiert. Soll der blockierte Thread fortgesetzt werden, wird einfach sein Mutex einmal freigegeben.

6.2 Zeitsteuerung

Für die Zeitsteuerung mussten die Threadstrukturen um ein paar Attribute erweitert werden. Es kamen drei Einplanungsstrukturen pro Thread hinzu, jeweils für aktivieren, pausieren und verzögertes fortsetzen. Jedes dieser Einplanungsmodule enthält die nötigen Strukturen für Zeiten und Dauern (Start, Periode

und Ende) und einen entsprechenden POSIX-Timer. Die Funktionen für zeitbedingtes pausieren und verzögertes warten sind schnell zu implementieren, da sie nur Startbedingungen haben. Das bedeutet, dass nach einmaligem Auslösen des Timers die entsprechende Funktion der generellen Tasksteuerung aufgerufen und der Timer rückgesetzt wird. Die Timer selbst lösen bei Ablauf das Signal „SIGRTMIN+1“, „SIGRTMIN+2“ oder „SIGRTMIN+3“ aus, je nach Art des Timers. Diese drei Signale werden nur vom Verwaltungsthread empfangen, welcher dann in den entsprechenden Signalhandler wechselt und die weitere Bearbeitung übernimmt. Bei den Aktivierungstimer ist die Behandlung am kompliziertesten, falls eine Perioden- und Endbedingung gesetzt wurde. Hier muss dann entsprechend die Systemzeit ausgelesen und anhand einer Berechnung festgestellt werden, ob die Endbedingung erreicht ist oder nicht. Anhand dessen wird dann entschieden, ob der Task aktiviert oder der Timer deaktiviert wird.

7 Fazit

Die Mindestvoraussetzung für eine Sprachabbildung von PEARL auf die Linux Systemschnittstelle ist ein Linux mit preemptive priority scheduling. Alles bis auf die Zeit- und Interruptsteuerung ließ sich problemlos abbilden. Die Zeitsteuerung kann aber selbst nachgebildet werden. Leider reichte die Zeit nicht aus, um die Interruptsteuerung ebenfalls nachzubilden. Allerdings ist anhand des Designs zu erwarten, dass dies ebenfalls keine unlösbaren Probleme verursacht. Um harte Echtzeit zu erreichen kommt die Abbildung auf Xenomai ebenfalls in Betracht. Jedoch stellt sich dabei die Frage, ob es im Hinblick auf Aufwand und Nutzenverhältnis sinnvoll ist. Vor allem vor dem Hintergrund der stetig steigenden Echtzeitfähigkeit des Linuxkernels.

Literaturverzeichnis

1. <http://www.irt.uni-hannover.de/pearl/pearlein.html>, Zugriff 14.03.2012
2. PEARL 90 Sprachreport, Version 2.0, Januar 1995
3. <http://www.xenomai.org/documentation/trunk/html/api/>, Zugriff 18.4.2012
4. Artikel: hrtimer: Hochauflösende Timer in Linux von Andreas Klingler
<http://www.elektronikpraxis.vogel.de/themen/embeddedsoftwareengineering/implementierung/articles/264868/>, Zugriff 30.5.2012
5. Linux Manual Pages
6. http://kernelnewbies.org/Linux_2_6_16, Stand 23.3.2012

Kommunikation unter Echtzeitbedingungen

Echtzeit 2012

Halang, W.A. (Hrsg.)

2013, VIII, 142 S. 61 Abb., Softcover

ISBN: 978-3-642-33706-2