Entwicklung einer OSEK/VDX-kompatiblen Systemschnittstelle für Linux

Johannes Bauer

JohannesBauer@gmx.de

Abstract: Dieses Paper fasst die Studienarbeit von Johannes Bauer zusammen, die sich mit der Entwicklung einer Systemschnittstelle für das im KFZ-Bereich häufig anzutreffende Echtzeitbetriebssystem OSEK/VDX beschäftigt.

1 Einführung

Für eingebettete Systeme, die Echtzeitbedingungen erfüllen müssen, eignen sich Betriebssysteme mit deterministischem, statischen Schedulingverhalten ausgezeichnet. Ein solcher Betriebssystemstandard ist OSEK/VDX [OSE05]. Da dieses vornehmlich auf tief eingebetteten Systemen zum Einsatz kommt, sind die Möglichkeiten des Testens funktionaler Komponenten und des Debuggings eher beschränkt, wenn überhaupt vorhanden. Eine Simulation des Laufzeitverhaltens auf einer mächtigeren Plattform ist hier also wünschenswert. Dies ist genau das Ziel der Studienarbeit. Durch die darin vorgestellten Werkzeuge wird es möglich, Applikationen, die gegen das OSEK-Betriebssysteminterface programmiert sind, auf einem herkömmlichen x86-32 oder x86-64 auszuführen, sofern auf diesem ein UNIX-artiges Betriebssystem läuft. Es gibt bereits zwei ähnliche Projekte, *openOSEK* [ope07] und *Trampoline* [BBFT06]. *openOSEK* befindet sich momentan noch im Entwicklungsstadium und ist nicht lauffähig, *Trampoline* hingegen hat bereits eine solide Codebasis. *Trampoline* verfolgt den Ansatz, durch Virtualisierung der Zielhardware möglichst portablen OSEK-Code zu erzeugen. In der vorliegenden Arbeit ist der Schwerpunkt ein anderer: Hier wird versucht, eine möglichst hardwarenahe Abbildung auf das darunterliegende UNIX-artige Betriebssystem zu erhalten. Dies geschieht zum Einen, um es zu ermöglichen, das OSEK-Betriebssystem auch direkt auf eingebettete Hardware zu übersetzen, zum Anderen um möglichst „echt“ zu Lehrzwecken zu zeigen, wie ein OSEK-Betriebssystem in der Praxis gebaut ist.

2 Zielstellung und Aspekte der UNIX-Implementierung

Die Studienarbeit beschäftigt sich mit der Erstellung eines Codegenerators, genannt *josek*, welcher für eine gegebene OSEK-Betriebssystemkonfiguration C-Code erzeugt. In

der Konfigurationsdatei, dem sogenannten *OIL*[1] [OSE05, OSE04] wird die Konstitution des zu erzeugenden Betriebssystems beschrieben. Dies schließt also unter anderem die Anzahl der Tasks und Ressourcen sowie deren Beziehungen miteinander ein. *josek* parst die OIL-Datei und erzeugt dann mit Hilfe verschiedener situationsabhängiger Templates dynamisch den Betriebssystemcode. Hier kommt der Vorteil eines Codegenerators im Gegensatz zu vorgefertigtem Code zum Tragen: Spezielle Optimierungen können vom Codegenerator berücksichtigt werden und dadurch effizienter Code erzeugt werden. Dies ist zum Beispiel der Fall, wenn zur Generierungszeit ermittelt werden kann, dass zwei Tasks zur Laufzeit nie gleichzeitig ausgeführt werden. Zwei unterschiedliche Stackbereiche für die beiden Tasks sind dann nicht mehr nötig; sie können zu einem Stackbereich verschmolzen werden und so RAM sparen. Auch sind diverse Möglichkeiten vorgesehen, den Code durch Debuggingfeatures zu ergänzen. So kann, sofern das OSEK-OS auf einem Linux ausgeführt wird, automatisch Code zum Speicherschutz der Stackbereiche erzeugt werden. Sollte ein Task zur Laufzeit seinen Stackbereich überschreiten, führt dies zu einem synchronen Programmabbruch, der leicht zu debuggen ist. Neben den mannigfaltigen Optimierungs- und Spezialisierungsmöglichkeiten stellt sich noch ein dritter Vorteil von generiertem Code automatisch mit ein: das Endprodukt ist gut lesbar und kann so zu Lehrzwecken eingesetzt werden. Der generierte Code kann dann übersetzt und gegen die eigentliche Applikation gelinkt werden. Obwohl auf den Plattformen, auf denen UNIX-artige Betriebssysteme lauffähig sind, Ressourcen wie Plattenspeicher und RAM-Größe im Überfluss vorhanden sind, verfolgt *josek* dennoch eine duale Zielstellung: Praktische Anwendungen, auf der OSEK-konforme Betriebssysteme zum Einsatz kommen, sind in ihren Ressourcen möglicherweise stark limitiert. Deshalb wurde *josek* derartig konzipiert, dass es praktisch möglich ist, auch eine Abbildung auf direkte Hardware durchzuführen – hierbei wurde die 8-Bit Mikrocontrollerfamilie AVR anvisiert.

3 Implementierung

Soll nun aber – wie es der Hauptzweck der Arbeit war – eine Betriebssysteminstanz auf einem UNIX-artigen System simuliert werden, sind einige Abbildungen nötig, da direkter Hardwarezugriff weder einfach möglich noch wünschenswert ist. Deshalb wird eine komplette Betriebssystem-Instanz auf einen UNIX-Prozess abgebildet. Der Stack, den die verschiedenen Tasks benutzen, wird in diesem UNIX-Prozess in den Heap gelegt. Dies wird dadurch möglich, dass zur Generierungszeit des Betriebssystems die Größen für die einzelnen Stacks bereits feststehen müssen statt dynamisch zur Laufzeit zu wachsen. Letztendlich werden die Tasks selbst auf User-Level Threads abgebildet.

Um einem UNIX-Prozess eine asynchrone Programmunterbrechung zu senden, kann *josek* bei Bedarf eine ISR mit einem POSIX-Signal verknüpfen. Ein Signalhandler wird automatisch installiert und springt die ISR-Routine an, sobald der Prozess das Signal empfängt. Hierdurch wird die Interruptsemantik erfüllt und nachgebildet, allerdings mit einem Haken: Die ISRs werden zeitverzögert zugestellt, weil Signalhandler in Linux nur dann angesprungen werden, wenn ein Rescheduling-Point erreicht wird. Das heißt der ISR wird

[1] OSEK Implementation Language

erst dann ausgeführt, wenn der OSEK-OS-Prozess durch Ablaufen seiner Zeitscheibe verdrängt wird oder selbst einen Systemaufruf durchführt. Dies stellt allerdings kein Problem dar, da die OSEK-Spezifikation bei der Definition der Interrupts nicht wählerisch ist. Insbesondere wird kein maximaler Jitter gefordert.

Die Interaktion eines UNIX-Prozesses mit der Außenwelt, also beispielsweise einer Device-Node die einen CAN-Bus-Controller repräsentiert, ist nicht ganz einfach: Sollte der Prozess nämlich einen blockierenden `read` oder `write` Systemaufruf auf diese Device-Node ausführen, blockiert nicht nur der momentan laufende Task, sondern das ganze Betriebssystem. Dies kann natürlich durch nicht-blockierende Systemaufrufe und Polling unterbunden werden, ist aber unschön. Auf realer Hardware sähe die Realisierung schließlich so aus: die Peripherie signalisiert der CPU durch einen Interrupt, dass auf dem Gerät Daten anliegen oder dass das Gerät zum Schreiben bereit ist. *josek* unterstützt auch diese Art von IO-Interrupts: Hierzu kann der laufende Tasks eine Device-Node öffnen und *josek* durch eine Systemschnittstellenerweiterung den erhaltenen Filedeskriptor mitteilen. Dies geschieht mit der zusätzlichen Anforderung, dass Interrupts ausgelöst werden sollen, sobald der übergebene Deskriptor zum Lesen oder Schreiben bereit ist. Erhält das OSEK-Betriebssystem eine solche Anforderung, wird ein zweiter asynchroner Aktivitätsträger durch Zuhilfenahme der *pthreads*-Bibliothek erzeugt. Dieser neu entstandene Thread führt auf den überwachten Filedeskriptoren den POSIX-Systemaufruf `poll` aus, der solange blockiert, bis eine der Wartebedingungen erfüllt ist. Sobald dies geschieht, sendet der `poll`-Thread dem gesamten Prozess das Signal `SIGIO`. Im Hauptkontrollfluss ist vorher ein Signalhandler installiert worden, der dann die entsprechenden IO-Interrupts delegiert.

Werden von der Applikation, die auf dem OSEK-OS laufen soll, Alarms benutzt, so werden diese alle auf den Systemtimer, der jedem UNIX-Prozess zur Verfügung steht, abgebildet. Mittels `setitimer` wird hier also ein Systemalarm definiert, der dann wiederum in der Callback-Funktion – dem Signalhandler für `SIGALRM` – an die OSEK-Alarms entsprechend dispatcht.

4 Zusammenfassung

Insgesamt stellt der in der Arbeit entwickelte Codegenerator *josek* eine nützliche Möglichkeit dar, Programme, die gegen die OSEK-Schnittstelle programmiert sind, auf einem UNIX-System auszuführen und funktionale Komponenten so zu testen. Dies ist insbesondere deshalb sinnvoll, weil höhere Betriebssystemschichten, wie beispielsweise das an der Universität Erlangen entwickelte KESO [WSSP07], die relativ schlanke OSEK-Schnittstelle als Hardwareabstaktionsschicht voraussetzen. Zusätzlich zeigt die Arbeit in ihrem dualen Charakter, dass es möglich ist einen Codegenerator derart zu entwickeln, dass mit geringen Anpassungen auch Programmcode erzeugt werden kann, der direkt auf Hardware aufsetzt, welche stark ressourcenbeschränkt ist.

Literatur

[BBFT06] Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou und Yvon Trinquet. Trampoline – An OpenSource Implementation of the OSEK/VDX RTOS Specification. 2006. http://trampoline.rts-software.org/IMG/pdf/trampoline.pdf.

[ope07] openOSEK.org. *The most common questions answered*. March 2007. http://www.openosek.org/tikiwiki/tiki-index.php.

[OSE04] The OSEK Group. *OSEK/VDK Binding Specification 1.4.2*. July 2004. http://portal.osek-vdx.org/files/pdf/specs/binding142.pdf.

[OSE05] The OSEK Group. *OSEK/VDK Operating System Specification 2.2.3*. February 2005. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf.

[WSSP07] Christian Wawersich, Michael Stilkerich und Wolfgang Schröder-Preikschat. An OSEK/VDX-based Multi-JVM for Automotive Appliances. In *Embedded System Design: Topics, Techniques and Trends*, IFIP International Federation for Information Processing, Seiten 85–96, Boston, 2007.