Massgeschneiderte VM-Architektur

Teil 1: Winziger virtueller CPU-Kern unterstützt Multithreading

Für die Realisierung einer winzigen programmierbaren Embedded-Steuerung, welche eine Hand voll Ein- und Ausgänge aufweist und auf kleinstem Adressraum Multithreading unterstützen soll, wurde eine massgeschneiderte VM-Architektur entworfen und angewandt.

Zumindest seit der Verbreitung der Java-Technologie [1], ist die virtuelle Maschine (kurz VM) in der Softwarewelt ein allgemein bekannter Begriff. Dabei handelt es sich um einen Prozessor, welcher nicht wie üblich als Mikrochip in Silizium “gegossen” ist, sondern eine reine, "virtuelle" Softwarerealisierung. Also ein Softwareprogramm, welches das Verhalten eines Prozessors besitzt, Code ausführt, verarbeitet, rechnet und steuert. Dieser virtuelle Prozessor wird dann schlussendlich doch von einem reellen Prozessor ausgeführt, da er ja selber in Software realisiert ist. Also was soll das Ganze? Wieso verwendet man virtuelle Maschinen, diese zusätzliche Schicht, wenn dieser doch dasselbe wie ein reeller Prozessor macht, und der reelle Prozessor ist ja sowieso schon vorhanden?

Die naheliegenste Antwort auf diese Frage lautet: um z.B. eine Applikation auf verschiedenen Hardwareplatformen ausführen zu können, ohne es für die jeweilige Platform einzeln anpassen, verteilen und verwalten zu müssen. Aslo um mehrere Platformen zu unterstützen, ohne dabei die Entwicklungskosten der Applikation zu erhöhen. Und die Rechnerwelt besteht nun mal aus verschiedenen Platformen. Genau hier kommt die virtuelle Maschine ins Spiel. Diese hat eine und nur eine Architektur. Sie abstrahiert die reelle Hardwarearchitektur. Aus Sicht der Applikation ist es, als würde nur eine Platform existieren. Eben diese virtuelle Platform.

Es können auch andere Gründe für die Existenzberechtigung einer virtuellen Maschine aufgezählt werden. Einige in diesem Artikel weiter verfolgte Argumente sind:

- **hohe Flexibilität** in ein System einfügen: typischerweise wird die Flexibilität eines Systems mit Konfigurationsdaten erhöht. Dieses Verfahren ist jedoch relativ beschränkt, da nicht alle Szenarien im voraus einbezogen werden können, und die Konfigurationsmöglichkeiten werden sehr umfangreich und unübersichtlich. Ein wesentlich leichtgewichtiger und flexibler Ansatz besteht darin, bestimmte Aspekte des Systems mit Skripten zu erweitern. Diese Skripte können frei programmiert und geladen werden, ohne das eigentliche System zu verändern. Solche Skripte können z.B. von einer virtuellen Maschine ausgeführt werden.
- **höhere Abstraktion** des Programms einführen, um die Business-Bedürfnisse treffender und direkter im Programm beschreiben zu können. Und somit auch um das Programm kürzer und einfacher zu gestalten. Dazu wird die Architektur der virtuellen Maschine Business-Spezialitäten aufweisen müssen, z.B. mit Hilfe von Spezial-Instruktionen.
- **Programm-Einschränkungen**: Eine massgeschneiderte Architektur erlaubt Programme mit erhöhten Schutzmechanismen und Funktionseinschränkungen auszuführen, welche sonst üblicherweise nicht oder umständlicher gewährleistet wären, z.B. in einer Sandbox.

Die virtuelle Maschine hat sehr viele Gemeinsamkeiten mit einer reellen CPU: typische Architektur mit Adressraum, Instruktionssatz, sequentieller Ablauf, von Neumann oder Harvard, Stack... usw. Die Grundkonzepte sind dieselben. Verglichen mit einer reellen CPU, hat die virtuelle Maschine vielleicht den Vorteil, besser skalierbar zu sein. Anzahl oder Grösse von funktionellen Einheiten sind nur durch den reellen Speicher begrenzt, deren Kombination ist frei verfügbar. Bei reellen CPUs sind hier die Grenzen durch die Schaltung gegeben, wobei eine gegebene Schaltung für eine bestimmte Aufgabe entworfen wurde. Diese ist meist nicht frei umkonfigurierbar auf andere Funktionalität. Die virtuelle Maschine hat den Nachteil, eine vergleichsweise kleinere Rechenkapazität aufzuweisen, und zusätzliche Resourcen für die virtuelle Maschine selber in Anspruch zu nehmen.

Wahl der Architektur

Nun wie wählt man eine geeignete Rechner-Architektur? Diese Frage ist nicht leicht zu beantworten, man hat nämlich die Qual der Wahl. Es ist sicher empfehlenswert, sich an bestehende Architekturen stützen, und nicht eine völlig neue Welt zu erfinden. Bleiben noch so ganz grundlegende Fragen wie:

- Soll die zu verarbeitende Datenwortbreite, 8bit, 16bit, 32bit... sein?
- Integer-Arithmetik oder auch Floatingpoint ?
- Soll der Instruktionssatz rein Stack-basiert oder Register-basiert sein?
- Soll die Speicherorganisation nach von Neumann oder Harvard erfolgen?
- Für welches Umfeld soll der Instruktionssatz spezialisiert werden?
- Welchen Instruktionsumfang, RISC oder CISC ?

Wenn die Grundzüge der Archtektur definiert sind, folgen dann noch tausend weitere Fragen, welche die Architektur noch im Detail beeinflussen werden. Viele Aspekte der Architektur werden vom Instruktionssatz

beeinflusst, und umgekehrt. Der Instruktionssatz muss auch auf Komplettheit überprüft werden. Somit wird man den Instruktionssatz sehr sorgfältig entwerfen.

Nicht zu vernachlässigen ist der Aufwand für die Entwicklung eines für diese Architektur angepassten Assemblers und anderen Entwicklungswerkzeugen, welche diese Architektur unterstützen werden. Diese Tools sind ein essentieller, unentbehrlicher Bestandteil des Gesamtsystems. Auf diesen Aspekt werden wir in einem folgenden Artikel dieser Serie eingehen.

Fish automate II

Kommen wir zurück zu unserem ganz konkreten Fall. Das in diesem Artikel vorgestellte Projekt ist eine kleine low-cost Echtzeit-Embedded-Steuerung für den Home-, Industrie- und Robotikbereich [2]. Diese erlaubt das Erfassen von digitalen oder analogen Eingangssignalen, wie z.B. Bedientasten, Endschalter, Lichtschranken, Temperaturfühler... usw., und das Stellen von einfachen digitalen und analogen Aktoren wie z.B. Lampen, Magnetspulen und DC-Motoren... usw. Diese Steuerung soll durch den Bediener frei programmierbar sein, indem er selber geschriebene Programme auf die Steuerung herunterladen kann. Diese Programme, welche Eingänge erfassen und Ausgänge steuern, implementieren beliebige Automatismen und Verhalten des Embedded Systems. Idealerweise sollte die Programmierung auf einer hohen abstrakten Ebene erfolgen, in der Form von mehreren parallel laufender Statemachines. Der Name dieser Steuerung is "fish automate II", es ist bereits die zweite Generation, darum die Zwei.

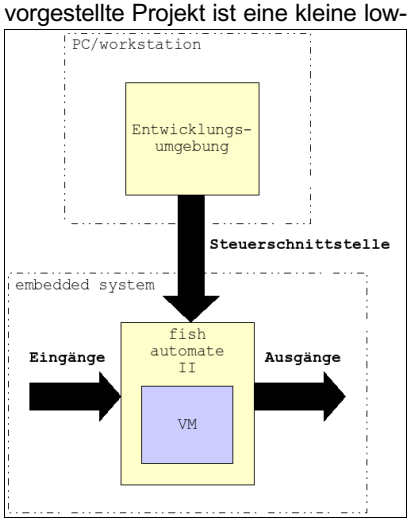


Bild 1: Übersicht des fish automate II in seiner Umgebung

Die gesamte Steuerung soll mit den knappen Resourcen eines low-cost single-chip 8bit Mikrokontrollers auskommen. Die Referenzimplementation bedient sich eines Atmel AVR ATmega16 Mikrokontrollers [3], welcher 16KB Flash, 1KB SRAM, 0.5KB EEPROM aufweist. Die Hardware-Architektur soll zu jeder Zeit geändert werden können, ohne das Konzept zu gefährden. Der fish automate soll jedoch immer mit dieser Grössenordnung von Resourcen auskommen können.

Diese in der Programmiersprache C realisierte Embedded-Steuerung weist unter anderem eine virtuelle Maschine auf. Das vom Benutzer heruntergeladene Programm wird auf der virtuellen Maschine ausgeführt. Diese virtuelle Maschine soll folgende Grundeigenschaften aufweisen:

- die Hardware-Platform abstrahieren, um unabhängig von dieser zu sein
- die Abstraktion des Programs erhöhen, typische Komponenten dieser Embedded-Welt beinhalten, wie z.B. Registerbereich für die Seuerung der speziellen Peripheriekomponenten, wie PWM-Ausgang oder DC-Motor-Treiber
- Multithreading und Thread-Synchronisation vollständig unterstützen
- muss Echtzeitfähigkeit aufweisen
- Optimierung auf möglichst kleine Programmgrösse, und somit Library-Funktionalität möglichst vom ladbaren Programm in die VM-Architektur zu verlagern

Das Projekt weist Aspekte auf, welche weniger kritisch sind, und somit vernachlässigt werden dürfen:

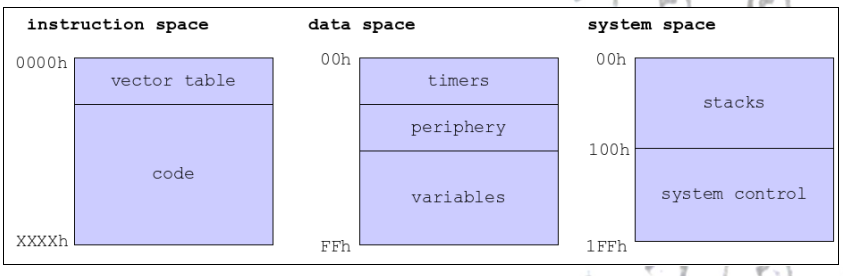
- Datendurchsatz ist unkritisch, kann gering sein
- Datenwortbreite und Rechenleistung können gering sein. Die Programme sind mehr auf das Steuern ausgelegt, als aufs Rechnen
- Das System ist aus Programm-Sicht eigenständig und autonom, die Kommunikationsströme mit der Aussenwelt können vernachlässigt werden

Die gewählte Architektur hat folgende Merkmale:

- 8bit Datenwortbreite
- basierend auf Harvard-Architektur, mit getrennten Adressbereichen für Code (instruction space), Daten (data space) und System (system space, in welchem die Stacks und die Register für die globale Verwaltung der Steuerung liegen)
- Multi-Kontexte und einen eingebauten Scheduler für die Unterstützung von Multithreading. Bis zu 16 statische Kontexte können gleichzeitig laufen, welche jeweils eigene Instruktionspointer, Zustandsverwaltung und Stack aufweisen. Somit laufen mehrere Threads parallel in ein und demselben Code ab
- rein Stack-basierte Architektur. Die Instruktionen holen sich ihre Parameter vom Stack, und legen die Resultate wieder in den Stack. Dies reduziert durch das klare Konzept die Speicherverwaltung im Code, und fördert Multithreading indirekt durch Reentrance. Jeder Kontext hat einen eigenen Stack. Alle Stacks befinden sich im system-space. Die Aufteilung der Stacks ist flexibel konfigurierbar.

Der **instruction-space** ist persistent,

Bild 2: Die Adressbereiche für Code, Daten und Systemzugriffe sind vollständig voneinander getrennt und werden separat adressiert, jeweils bei null beginnend. Der Grund für diese Trennung liegt vorallem bei der Zugriffsrechtverwaltung. Das Programm hat quasi nur auf den data space Zugriff.



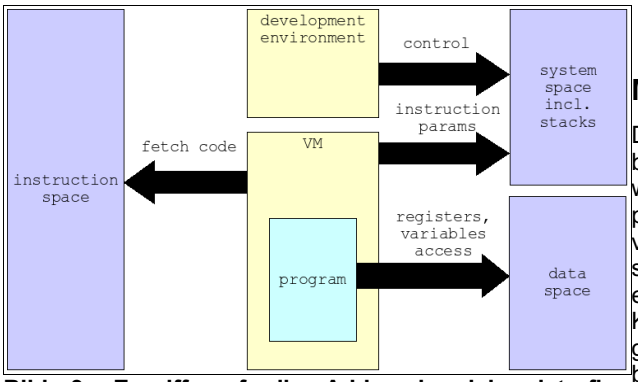


Bild 3: Zugriff auf die Addressbereiche ist fix organisiert: Während das Programm nur auf den data space direkten Zugriff hat, benutzen einige Instruktionen den Stack, welcher sich im system space befindet. Ausserdem werden die Instruktionen vom instruction space geladen. Die Entwicklungsumgebung hat auch Zugriff auf den gesamten system space, um die VM zu steuern.

8bit-weise organisiert, die Instruktionen sind von variabler Länge, um Codegrösse zu optimieren. Der unterste Bereich beinhaltet die vector table, welche je nach Anzahl benutzter Kontexte, eine variable Länge aufweist. Jeder Eintrag ist ein Sprung zum Code des entsprechenden Threads. Die Grösse des instruction-spaces kann maximal 64KB sein (16bit-adressierbar), wobei diese in Abhängigkeit der benutzten Hardware-Platform auch kleiner sein darf. Dieser Adressbereich kann nicht direkt vom Programm zugegriffen werden, es kann nur aus diesem ausgeführt werden. Dank der Persistenz startet das Program nach dem Powerup automatisch.

Der **data-space** ist 8bit-weise organisiert, random-access, und ist 256 Bytes gross (8bit-Adresse ist ausreichend). Dieser beinhaltet Register für die Kontrolle der spezialisierten Peripherie (wie Eingänge, Ausgänge, DC-Motor-Treiber... usw.) und programmierbare Timer. Der Rest dieses Adressbereiches kann frei für Variablen verwendet werden. Dieser Bereich kann also vom Programm aus lesend und schreibend zugegriffen werden.

Der **system-space** beinhaltet spezielle Systemobjekte, welche vom Programm aus nicht direkt zugreifbar sind. Dieser ist 8bit-weise organisiert und 512 Bytes gross. Die unteren 256 Bytes müssen sich die bis zu 16 Stacks teilen, somit sind die Stack-Pointers 8bit-adressierbar. Die oberen 256 Bytes beinhalten die globalen Steuerregister des fish automates, welche nur über die Steuerschnittstelle zugreifbar sind. Diese Steuerschnittstelle wird für das Herunterladen eines Programms und für das Debuggen dieses benutzt.

Multi-Kontexte

Die VM unterstützt bis zu 16 Kontexte, welches ein pseudo-paralleles Ausführen von 16 Threads im selben Codebereich erlaubt. Wieviele Kontexte ein Programm nun konkret benutzt, kann es mit der Spezialinstruktion `context,n` selber angeben. Diese Instruktion muss ausserdem immer auf der Adresse 0x0000 liegen. Die Grösse der folgenden vector-table hängt von dieser ersten Instruktion ab. Je kleiner die vector-table, desto mehr Raum bleibt für den eigentlichen Applikationscode.

Ein in der VM fest eingebauter Scheduler erledigt den Kontext-Switch. Dieser ist ziemlich effizient, da keine Daten herumkopiert werden müssen, jeder Kontext hat ja seine eigenen Register. Es ist quasi nur ein Registerbank-Wechsel. Kontext-Switch erfolgt in folgenden Situationen:

- kooperativ, wenn ein Thread die Instruktion `yield` ausführt
- preemptiv, wenn ein Thread eine bestimmte maximale Anzahl Instruktionen ausgeführt hat, ohne den Kontext abzugeben

Da in dieser Architektur keine Thread-Prioritäten unterstützt werden, erfolgt das Scheduling nach dem round-robin Verfahren, also immer ein Thread nach dem anderen.

Jeder Kontext hat seinen eigenen Stack. Dieser wird durch den individuellen Stack-Pointer zugegriffen. Jeder Thread muss seinen Stack selber initialisieren, dies macht er mit der speziellen Instruktion `stack,l`, wobei die

```
;fish automate II assembler example: gated blinky
;(c) 2005 oli4, switzerland
;
;description:
;this is a little program which drives a flashing lamp at output 6
;as long as input 1 is high. when input 1 is low, then the lamp
;finishes its last flash-sequence and keeps dark.
;in addition, analog input 2 drives pwm output 2 proportionally

include fish.inc

;vector table
0x0000: context 3
        jmpa      task0
        jmpa      task1
        jmpa      task2


task0: stack 10                  ;initialize stack
       movc .dout67.enable .true ;enable digital-output 6/7
       movc .dout6 .false        ;switch lamp off
pulse: wait
       call pulseLamp            ;pulse lamp two times
       call pulseLamp
       movc .timer0 10           ;wait 1000ms
read:  load .timer0
       yield
       branchz
       jmpa read
       jmpa pulse                ;loop back


task1: stack 10          ;initialize stack
scan:  load .din1        ;read input
       yield
       branchz           ;signal other task i
...
```

Bild 4: Dies ist ein Ausschnitt des Assemblers, welcher den Startupcode zeigt. Man erkennt den Label 0x0000, welches den Assembler darauf hinweist, dass diese Instruktion auf die Adresse 0 platziert wird, und somit den Ursprung ist. Es werden die Anzahl Kontexte definiert. Danach folgt die 3 Einträge grosse vector table, welche alle einen Sprung zu ihren entsprechenden Code beinhaltet, hier mit den Labels task0, task1 und task2 benannt.

maximale Stackgrösse angegeben werden muss. Die Platzierung des Stacks im system-space verwaltet die VM automatisch. Der Stack besitzt einen Speicherschutz-Mechanismus, Stack Über- und Unterlauf werden von der VM detektiert, und bei Auftreten

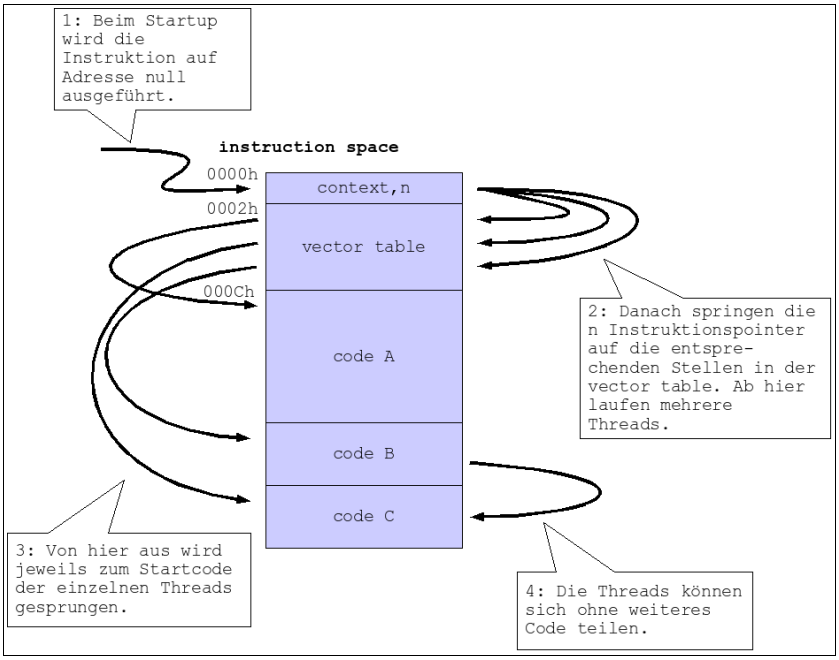


Bild 5: Bootprozess, die Geburt der Threads, Schritt für Schritt erklärt

wird der entsprechende Thread gekillt. Bevor der Stack initialisiert wurde, darf keine Instruktion ausgeführt werden, welche den Stack verwendet. Denn jeder Zugriff würde einen Über- oder Unterlauf verursachen.

Der Bootprozess erfolgt folgendermassen: falls in den ersten Sekunden die Steuerschnittstelle inaktiv bleibt (also kein Programmdownload oder Debuggingzugriff von aussen gestartet wird), wird die erste Instruktion des Programmcodes ausgeführt. Dieses teilt der VM die Anzahl aktiven Kontexte mit. Falls sich dort eine andere Instruktion befindet, wird das gesamte Programm abgebrochen. Aber im Normalfall werden danach die Instruktionspointer auf die benutzten Vektoren gesetzt. Jeder Eintrag der vector table kann entweder eine Jump-Instruktion zum Startpunkt der Threads beinhalten, oder eine Kill-Instruktion, um den entsprechenden Thread von Anfang an zu stoppen. Ab diesem Zeitpunkt laufen mehrere Kontexte mit deren Instruktionspointer pseudo-parallel ab. Bevor aber ein Kontext eine Instruktion ausführt, welche seinen Stack benutzt, muss dieser Stack initialisiert werden. Nun sind die Threads voll betriebsbereit.

Jeder Thread kann einer der folgenden Zustände aufweisen:

- `run`: der Thread läuft zur Zeit oder ist bereit und wartet auf den Scheduler, weil ein anderer Thread gerade aktiv ist
- `halt`: der Thread ist angehalten, und wartet auf ein Signal von einem anderen Thread oder Timer
- `step`: wie run, hält aber nach der Ausführung einer Instruktion an. Dies wird fürs Steppen mit dem Debugger benutzt
- `slice`: wie run, hält aber nach vollendetem Scheduler-Zeitschlitz oder kooperativem yield an, auch hier für das Debuggen gedacht
- `killed`: der Thread ist beendet. Ein beendeter Thread, kann nicht mehr gestartet werden, bis zur nächsten Initialisierung des Programms. Bei Speicherzugriffsverletzung wird ein Thread auch gekillt. Ein Thread kann sich aber auch selber beenden, mit der `kill` Instruktion

Spezielle Instruktionen

Bei unserer VM handelt es sich um eine massgeschneiderte Variante, welche mehr Funktionalität aufweist, als bei üblichen CPUs. Eine der Spezialitäten hier ist die Unterstützung für Multithreading und Threadsynchronisation. Hier werden einige dieser speziellen Instruktionen vorgestellt:

- `kill`: beendet den aktuellen Thread
- `yield`: der aktuelle Thread startet freiwillig einen kooperativen Kontext-Switch. Sein Zeitschlitz wird hier frühzeitig beendet
- `stack,l`: initialisiert den Stack des aktuellen Threads. Die Stackgrösse l wird mitgeliefert
- `wait`: der aktuelle Thread geht in den Wartezustand. Somit wird implizit ein Kontext-Switch ausgelöst
- `signal,c`: sendet ein Signal dem angegebenen wartenden Thread c. Im Zusammenhang mit der wait-Instruktion ist somit Thread-Synchronisation möglich
- `sleep,t`: setzt den aktuellen Thread für eine gegebene Zeit t in den Wartezustand. Nach abgelaufener Zeit wird der Thread von alleine wieder starten
- context,n: definiert die Anzahl zu startende Kontexte. Es können 1 bis 16 Kontexte gestartet werden. Diese Instruktion muss auf der Adresse 0 liegen.

Steuerschnittstelle

Der fish automate weist eine serielle Schnittstelle RS-232 auf. Die Steuerschnittstelle erlaubt das Herunterladen und das Debuggen des Programms, und die direkte Ansteuerung der Ein- und Ausgänge des fish automates. Über diese Schnittstelle können alle Speicherbereiche zugegriffen werden:

- instruction-space: das Programm kann geschrieben und gelesen werden. Der Programm-Download bedient sich dieses Zugriffes
- data-space: dient zum Debuggen, und das direkte Ansteuern der Peripherie
- system-space: der Debugger kann die Stack-Inhalte lesen (und bei Bedarf verändern). Über die Systemregister kann ein Einblick in die Kontexte erfolgen. z.B. kann der Zustand der Kontexte gelesen werden. Nebst den oben erwähnten Zustände des Kontextes, kann im Falle eines gekillten Threads die genaue Ursache nachgelesen werden. Wurde dieser vom Programm aus

gekillt? Oder wurde versucht, eine illegale Instruktion auszuführen? Oder gab es eine Stack Unterlauf oder Überlauf? Oder wurde zuviel Stackspeicher alloziert? Diese Information zusammen mit dem aktuellen Stand des Instruktionspointers können die Ursache ziemlich genau eingrenzen.

Fazit

Anhand des konkreten Projektes einer winzigen programmierbaren Embedded Steuerung wird gezeigt, dass der Entwurf einer virtuellen Maschine mit massgeschneiderten Architektur und mit spezialisierten Instruktionen sinnvoll sein kann. Die Anforderung, mit möglichst geringen Programmspeicher auszukommen aber trotzdem Konzepte anzuwenden, welche normalerweise für grössere Systeme reserviert sind, wie in unserem Fall Multithreading, zwingt uns, möglichst viel in die "virtuelle Hardware" zu verlagern, was sonst üblicherweise in Code-Libraries im Programmcode verteilt wird.

In einem weiteren Artikel dieser Serie werden wir die Entwicklungsumgebung einer solchen VM näher betrachten.

Literatur und Links

[1] java.sun.com

[2] www.oli4-soft.ch

[3] www.atmel.com

Autor

Olivier Gäumann, dipl.-Ing. HTL, hat an der Ingenierschule Fribourg Elektrotechnik studiert. Er ist hauptsächlich in den Bereichen Embedded Systems, Mechatronik und DSP tätig, als Softwarearchitekt und Entwickler, im Umfeld C/C++ und java.