65. Jahrgang Elektronik reader's choice August 2016

Elektronik

elektroniknet.de

August 2016 € 12,80

reader's choice

Fachmedium für ausgewählte Elektronik-Themen

Internet der Dinge in der Automatisierungstechnik:

Funk statt Kabel

>> Seite 6

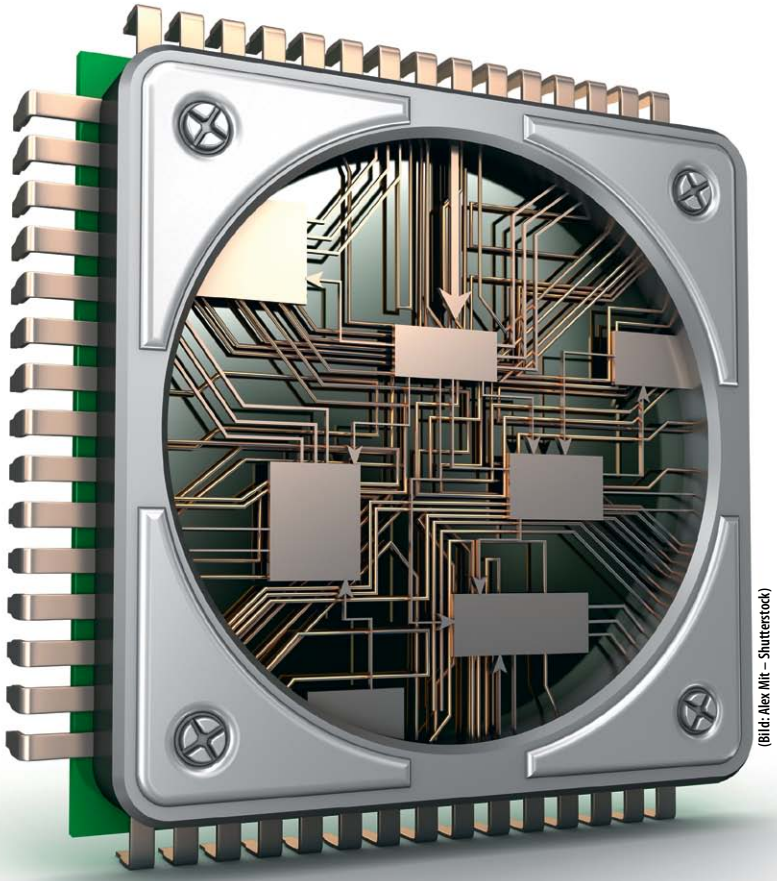
HMS

Asymmetrisches Multiprocessing:

Echtzeit mit eigenem CPU-Kern

Für Echtzeit unter Linux gibt es neben Preempt-RT und Xenomai auch die Möglichkeit, einen einzelnen CPU-Kern eines Multicore-Prozessors mehr oder weniger abgeschottet mit einer Echtzeit-Aufgabe zu betrauen. Eine Implementierung zeigt Vor- wie auch Nachteile.

Von Thomas Brinker



(Bild: Alex Mit – Shutterstock)

Interfaces der Bare-Metal-Echtzeit-Software

Für die echtzeitfähige Bare Metal Software sind folgende Schnittstellen und CPU-Bestandteile zu initialisieren und durch Treiber zu bedienen:

- CPU
- MMU
- Interrupt Controller
- C-Laufzeitumgebung
- Interfaces (hier: CAN)
- gegebenenfalls auch:
  - private Timer,
  - FPU/NEON,
  - Watchdog

Für die Entwicklung von langlebigen Industrieprodukten auf Basis von Linux sind Mehrkern-Prozessoren mittlerweile keine Exoten mehr. Die Tatsache, dass viele dieser Derivate „Drop-in Replacements" ihrer Ein-Kern-Brüder sind, vereinfacht den Einsatz von zwei, vier oder sogar noch mehr Kernen selbst in kleineren und kostensensitiven Produkten. In dieser Konsequenz bieten viele Hersteller ihre Computermodule und Industrie-PCs in Ein- oder Mehrkernvarianten an. Für Linux ist der Betrieb von mehreren CPU-Kernen seit Jahren Stand der Technik. Dabei werden alle Prozesse und Threads dynamisch entsprechend der Auslastung auf alle Rechenkerne verteilt. Das funktioniert auf x86, ARM Cortex-A9 und weiteren Architekturen.

Eine andere, statische Zuweisung von CPU-Kernen zu bestimmten Threads ergibt sich mit asymmetrischem Multiprocessing (AMP). Hierbei werden die Kerne nicht vollständig von einer Betriebssysteminstanz verwaltet und den dort laufenden Prozessen und Threads zugeordnet, sondern vielmehr werden einzelne Kerne exklusiv für eine bestimmte Software komplett außerhalb des Betriebssystems genutzt. Auf diesen separierten Kernen kann ohne den Ballast eines Betriebssystems die volle Rechenleistung genutzt werden. Der Entwickler hat die volle Kontrolle über die Ereignisse auf „seinem" CPU-Kern. Er kann frei bestimmen, wann welcher Interrupt kommt und wann nicht. Entsprechend hohe Leistungen können für Echtzeitsysteme erwartet werden. Die Programmierung des Thread auf der separaten CPU erfolgt direkt auf der CPU, also „bare-metal" in Assembler, C oder auch C++.

Emlix hat auf Grundlage dieser Architektur beispielsweise eine Regelung implementiert, die Schwingungscharakteristika per CAN von Sensoren erfasst und per CAN Steuerbefehle ausgibt. Diese Regelung umfasst umfangreiche Filterungen, die den Einsatz der ARM NEON Engine erfordern. Der Filter- und Regelalgorithmus wurde in C verfasst und läuft bare-metal auf Kern drei des NXP i.MX6 Quad, siehe **Bild 1**. Der Funktionsumfang der Bare-Metal Software kann meist sehr gering gehalten werden. Daher ist die für den Entwickler benötigte Lernkurve überschaubar. Zudem kann er sich auf erprobte Templates stützen und sich damit schnell auf seine eigentliche Kernkompetenz im Kontext der jeweiligen Applikation konzentrieren.

Alle Interfaces und Prozesse, die nicht für die unmittelbare Echtzeitregelung von Relevanz sind, also z.B. USB, Ethernet und Grafik, bleiben im Linux-System. Multitasking war verzichtbar,

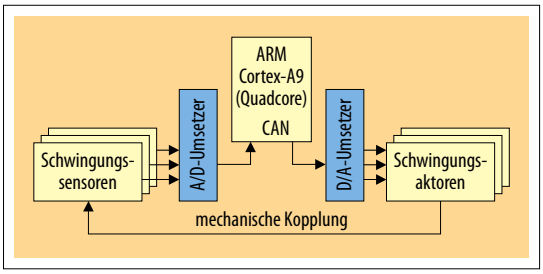


Bild 1. Klassische Regelungsschleife zur Messung mechanischer Schwingungen, harte Echtzeitregelungs-Software auf Cortex-A9.

da wie bei vielen Applikationen nur ein einziger Echtzeit-Thread benötigt wurde. Es reicht also eine Initialisierung nur für die benötigten Schnittstellen – in diesem Falle der CAN Controller und eine serielle Schnittstelle zum Debugging – und der Laufzeitumgebung, um einen Thread in C programmiert laufen zu lassen. Alle Hardware-Schnittstellen, die im Echtzeitsystem verwendet werden sollen, müssen von Linux unberührt bleiben.

Kommunikation über On-Chip-Speicher

Als Speicher verwendet die Echtzeit-Bare-Metal-Anwendung das On-Chip RAM (OC-RAM) des i.MX6. Dieser ist 64 bit breit, am AXI-Bus angeschlossen und 256 KB groß. Der Zugriff ist entsprechend schnell, siehe **Bild 2**. Reicht der On-Chip-Speicher für die Daten des Echtzeit-Thread nicht, so könnten Teile des DDR-RAM ergänzend genutzt werden. Die jeweiligen Caches erlauben auch darauf eine schnelle und effiziente Ausführung. Jedoch muss die Synchronisation bedacht werden, wenn Datenpakete von einem System ins andere transferiert werden.

Die Kommunikation zwischen den beiden Systemen erfolgt über Shared-Memory-Bereiche im On-Chip-RAM und über Interrupts. Mit hoher Performance informiert das Echtzeit-System das Linux-System über das aktuell detektierte Schwingungsverhalten. Dazu werden definierte Message-Strukturen fortlaufend in einen Ringpuffer im Shared Memory geschrieben. Anhand bestimmter atomarer Markierungen kann die Linux-Software erkennen, welche Messages im Ringpuffer gültig sind und welche gerade vom Echtzeit-Thread bearbeitet werden und somit noch nicht gelesen werden dürfen. Im Falle von drohenden Pufferüberläufen wird das Linux-System per Interrupt informiert. Kommt es danach dennoch zum Überlauf des Puffers, gehen zwar Daten für die Visualisierung und Archivierung verloren, aber ohne Auswirkungen für den Regelungsalgorithmus.

In der umgekehrten Datenrichtung werden Konfigurationsparameter der Regelung von Linux in den Echtzeit-Prozess übertragen. Dazu wird von Linux ein Flag gesetzt, welches die Regelung regelmäßig abfragt („pollt“) und dann zu geeigneter Zeit in den Regelungsalgorithmus übernimmt. Eine Unterbrechung des Echtzeit-Systems durch Interrupts ist nicht vorgesehen.

Toolchain für die Entwicklung

Durch die Symmetrie der Prozessorkerne ist keine spezielle Toolchain erforderlich. Die gleichen Compiler, wie sie auch für normale Linux-Programme genutzt werden, also beispielsweise der GCC, sind auch in der Lage, Bare-Metal-ausführbare Programme zu erzeugen. Lediglich ein paar zusätzliche Flags sind erforderlich, damit nicht wie sonst üblich gegen die Standard-C-Library gelinkt wird. Die steht in der Echtzeit-Software selbstverständlich nicht zur Verfügung. Standard-Funktionen, die man üblicherweise dieser Library entnimmt, müssen daher komplett neu implementiert werden. Das **Listing** zeigt den Compile eines Bare-Metal-Programms. Im GCC wird ein solches Programm als Free-Standing bezeichnet.

Einige Schritte der Initialisierung und zum Beispiel die ARM-Vektor-Tabelle lassen sich nicht in C ausdrücken und müssen in eigenen Assembler-Dateien implementiert werden. Auch der Code zur Initialisierung der MMU ist zwar in C verfasst, enthält aber etliche Inline-Assembler-Anweisungen. Der Initialisierungscode enthält auch die notwendige Initialisierung der FPU und der NEON Engine. Diese werden für die sehr umfangreichen Filter-Berechnungen der Schwingungsregelung mit Gleitkommazahlen benötigt.

Damit der Linker die korrekten Adressen einsetzt, welche zum Ausführungsort des Realtime-Programms passen, ergänzt ein Linker Script den Bauvorgang. Speicheradressen und -größen müssen im Linker Script, in der MMU-Initialisierung und im Linux-Kernel korrespondieren. Fehler an diesen Stellen bedeuten oftmals überlappende Speicher, die von keiner Software-Komponente erkannt werden. Der Entwickler erhält somit keine Warnung und ist mit einem unvorhersehbaren Verhalten des Komplettsystems konfrontiert. Das Debugging dieser Situation gestaltet sich entsprechend schwierig.

Starten des separierten CPU-Kerns

Das resultierende Binary aus dem gezeigten Compile im **Listing** ist ein Executable and Loadable File (ELF). Diese Datei muss im Verzeichnisbaum des Linux-Systems vorhanden sein und wird von dort aus in den reservierten OC-

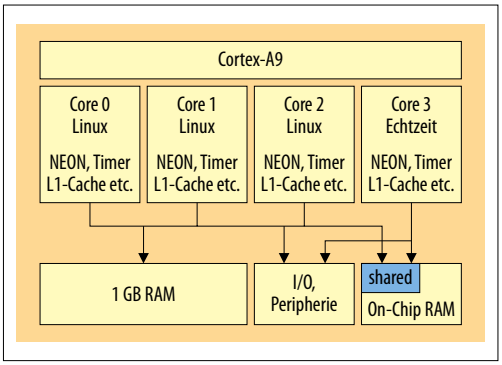


Bild 2. Aufteilung der Kerne mit getrennter und gemeinsamer Nutzung des Adressraumes.

```
arm-unknown-linux-gnueabi-gcc -c -o startup.o startup.S
arm-unknown-linux-gnueabi-gcc -c -o vectors.o vectors.S
arm-unknown-linux-gnueabi-gcc -ffreestanding -O2 -g -Wall -std=gnu99 -c -o mmu.o mmu.c
...
arm-unknown-linux-gnueabi-gcc -ffreestanding -O2 -g -Wall -std=gnu99 -c -o main.o main.c
arm-unknown-linux-gnueabi-ld -T linkscript.lds startup.o vectors.o mmu.o main.o [...]/libgcc.a -o bmapp
```

Compile eines Bare-Metal-Programms. Im GCC wird ein solches Programm als Free-Standing bezeichnet.

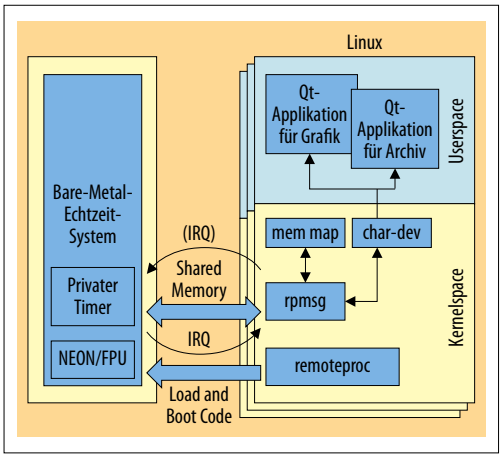


Bild 3. Aufteilung der vier ARM-Cortex-A9-Kerne auf die Software-Komponenten und deren Kommunikation.

RAM-Speicher geladen. Das Linux-System selber läuft auf drei CPU-Kernen. Der vierte Kern ist beim Boot nicht gestartet worden; auf ihm wird die Echtzeit-Software gestartet.

Diese Aufgaben übernimmt das Linux-Treiber-Interface *remoteproc*. Remoteproc liest die ELF-Datei mit dem in Linux gut etablierten Firmware-Interface aus dem Verzeichnis /lib/firmware/<name> ein, interpretiert die ELF-Struktur und lädt die Inhalte an die erforderlichen Speicheradressen. Danach wird der Program Counter (PC) des noch stillstehenden CPU-Kerns auf den Anfang des Programms gesetzt und der Reset gelöst. Die Software startet mit ihrer Initialisierung und beginnt die Verarbeitung.

Die Bare Metal Software kann auf die kompletten Hardware-Ressourcen des SoC zugreifen. Es ist jedoch penibel auf eine exakte Trennung zwischen den zwei Software-Systemen zu achten. Hardware-Ressourcen, die im Echtzeit-Programm verwendet werden, können von Linux nicht verwendet werden und umgekehrt. Das Starten der Bare Metal Software geschieht durch den Befehl

```
modprobe mx6_remoteproc
```

Es wird lediglich ein Linux-Kernel-Modul geladen, welches gleichzeitig auch die Linux-Seite der Kommunikation mit der Echtzeit-Software implementiert.

Sparsam mit Interrupts

Für die Kommunikation zwischen Linux und der Echtzeit-Bare-Metal-Software stehen ein gemeinsam genutzter Speicherbereich von 100 KB als Shared Memory und Inter-Processor Interrupts (IPI) zur Verfügung. Die Kerne können dem jeweils anderen System Interrupts zusenden, die der Empfänger dann ausführt oder auch mittels Maskieren ignoriert, wenn die Behandlung für die Echtzeitverarbeitung hinderlich wäre.

Die Schwingungssteuerung reduziert den Einsatz von Interrupts zwischen den Kernen auf ein Minimum. Für den Anwendungszweck ist ein auf Polling basierendes Kommunikationsschema völlig ausreichend. Die Linux-Applikation liest alle 0,25 Sekunden die aktuellen Messwerte ein und visualisiert deren Darstellung im Display. Dieses sorgt für eine ausreichend flüssige grafische Darstellung. Der Echtzeit-Prozess prüft nach jedem Zyklus, ob die Parameterstruktur im Shared Memory von der Linux-Applikation geändert wurde. Ist das der Fall, werden die neuen Parameter für den folgenden Zyklus übernommen. Folgender Ablauf bietet sich für ein solches Schema an:

- Boot Linux,
- Boot der Echtzeit-Anwendung mittels remoteproc, Einrichten des Shared Memory mit Rpmsg,
- Echtzeit-Anwendung pollt Go-Bit im Shared Memory.
- Linux startet den Echtzeit-Prozess mit Setzen des Go-Bits und pollt danach selber auf Finished-Bit.
- Die Echtzeit-Software führt nun den Messprozess durch und setzt das Finished-Bit zum Abschluss.
- Linux erkennt das gesetzte Finished-Bit und wertet den beendeten Messprozess aus.
- Durch erneutes Setzen des Go-Bits kann Linux einen zweiten Messvorgang starten.

Während der Messung werden regelmäßig Messdatenausschnitte in Form definierter Pakete in einen Ringpuffer geschrieben. Ein Markierungsbit in der Paketstruktur ermöglicht die Erkennung gültiger Pakete. Gelesene Pakete werden durch das Löschen des Bits wieder freigegeben.

Zur Vereinfachung und Generalisierung der Kommunikation in einem AMP-System wurde das Interface *rpmsg* geschaffen. Es erlaubt die flexible Verwaltung von Shared Memory mit korrekter Einbindung ins Linux Memory Management und die wechselseitige Zuweisung von Interrupts. rpmsg ist dabei Bestandteil des Linux-Kernels.

Das rpmsg-Modul der Schwingungsmessung stellt das Shared Memory zur Kommunikation durch Memory Mapping zur Verfügung. Das erlaubt jeder Linux-Applikation, den Speicher einzublenden und direkt lesend und schreibend Zugriff darauf zu nehmen. Genutzt wird dieses für Debugging-Zwecke. Im

Anwendungsprojekt für AMP

In einem Teststand für passive elektronische Baugruppen werden von einem Digital-Analog-Umsetzer Prüfstrom und -spannung eingestellt und durch einen Prüfling geleitet. Resultierende Ströme und Spannungen werden mit schnellen 12-bit-Analog-Digital-Umsetzern aufgenommen und zur weiteren Regelung der Eingangsspannung und des Eingangsstroms genutzt. Ein vorgegebener Testalgorithmus verifiziert dann das Bauteilverhalten bei bestimmten Strom- und Spannungswerten sowie bei entsprechenden Spitzen. Um valide Messergebnisse zu erhalten und zum Schutz von Prüfling und Tester ist ein Regelzyklus von 120 kHz (hart) erforderlich.

Die Ansteuerung der A/D- und D/A-Umsetzer über das I^2S-Modul des i.MX6 erfolgt aus der Echtzeit-Software im abgetrennten CPU-Kern; der Regelalgorithmus greift auf die Funktionen der NEON-Engine zu. Testergebnis und ermittelte Bauteilparameter werden von Linux-Programmen auf den anderen Kernen angezeigt und archiviert. Die Bedienung erfolgt über die HTML5-basierte grafische Oberfläche.

Echtzeit-Ethernet

Eine denkbare Erweiterung des Konzeptes besteht in der Nutzung des Ethernet Controller durch den Echtzeit-Kern. Echtzeit-relevante Ethernet-Pakete würden dort bearbeitet. TCP/IP und ähnliche Pakete würden über die Shared-Memory-Schnittstelle an Linux und den dortigen Netzwerk-Stack gereicht. Inwieweit dieses Konzept eine Benutzung von Profinet, EtherCat etc. erlaubt, müsste in einem Proof-of-Concept überprüft werden. Immerhin liegt die schnellste Profinet-Zykluszeit bei 31,25 µs und damit deutlich über den ermittelten Werten, sodass die Lösung zumindest nicht ausgeschlossen zu sein scheint.

	Standard IRQ			ARM FIQ		
	Ø	Min.	Max.	Ø	Min.	Max.
Low-Pegel	11,062 µs	6,514 µs	19,968 µs	10,022 µs	4,999 µs	15,059 µs
High-Pegel	11,012 µs	6,637 µs	19,577 µs	9,9841 µs	4,38 µs	15,047 µs

Tabelle 1. Während die CPU von einer Stresstest-App maximal ausgelastet wurde, erzeugte der private Timer der Echtzeit-Software ein Rechtecksignal aus 10 µs high und 10 µs low.

	Standard IRQ				ARM FIQ			
	Ø	Min.	Max.	Anzahl	Ø	Min.	Max.	Anzahl
High-Pegel	17,578 µs	17,00 µs	24,94 µs	287.400	17,27 µs	17,00 µs	18,773 µs	54

Tabelle 2. Messung über einen Zeitraum von 16 Stunden mit dem Benchmark „Stressapptest". Aufgezeichnet wurden nur Trigger-Ereignisse mit High-Pegel länger als 17 µs. Bei den Fast Interrupts (FIQs) kommt es nur zu 54 Überschreitungen. Die längste davon ist 18,773 µs lang.

Normalbetrieb wird ein Service des Linux-Kernel genutzt, um fertige Messdatenpakete aus dem Ringpuffer in verschiedene FIFOs zu kopieren. Ein FIFO wird mit GUI-relevanten Messdaten befüllt, der andere nur mit Delta-Messdaten zur Archivierung. Beide FIFOs sind als Standard Linux Character Devices zugänglich. Die Linux-Grafik-Applikation entnimmt diese Daten aus ihrem FIFO und stellt die Daten mittels Qt Framework auf dem Display dar. Die Archivierungsapplikation entnimmt die Delta-Messdaten und speichert diese auf einer SSD. **Bild 3** gibt einen Überblick.

Das Debugging der Echtzeit-Applikation kann mittels JTAG, entsprechendem Adapter und Tooling erfolgen. Um den Einrichtungsaufwand zu minimieren, bietet sich alternativ das Trace Debugging über eine serielle Schnittstelle an. Dass diese Möglichkeit weniger mächtig und komfortabel ist, wird unter Abwägung des Einrichtungsaufwands in Kauf genommen. Der Funktionsumfang des Trace Debugging ist in der Regel jedoch vollkommen ausreichend.

Echzeit-Software im Stresstest

Die Messungen des Echtzeitverhaltens wurden in einem vereinfachten Aufbau an einem GPIO-Port des NXP i.MX6 Quad auf einem armStone-Board von F&S aufgenommen. Die Echtzeit-Software erzeugt mittels Standard-Interrupt oder ARM-FIQs (Fast Interrupt Request) aus dem privaten Timer ein Rechtecksignal aus 10 µs high und 10 µs low. Der Aufbau ist in **Bild 4** schematisch dargestellt. Die Messwerte in **Tabelle 1** wurden über acht Stunden per Oszilloskop aufgenommen. Die CPU wurde mit dem Lasttest-Tool „stressapptest" (https://github.com/stressapptest/stressapptest) von Google unter Linux maximal ausgelastet. Mit dem Befehl

```
./stressapptest -s <Laufzeit>
-M 930 -m 16 -W
```

wird mit 16 Threads parallel auf 930 MB Speicher nichtsequenziell lesend und schreibend zugegriffen. Dieses Tool versucht insbesondere die Speicherbusse maximal zu belasten. Die Inanspruchnahme ist damit deutlich höher als die durchschnittliche Belastung im Normalbetrieb.

Die Messung in **Tabelle 2** zeigt einen Zeitraum von 16 Stunden unter Last mit Benchmark „stressapptest". Dabei wurden nur Trigger-Ereignisse aufgezeichnet mit High-Pegel länger als 17 µs. Unter FIQs kommt es nur zu 54 Überschreitungen während 16 h. Die Längste davon ist 18,773 µs lang.

Vergleichsmessungen mit dem PreemptRT-Patch auf der gleichen Hardware zeigten Jitter von 100 µs und mehr. Es fallen also mehrere komplette Zyklen aus. Nur bei Zykluszeiten von unter 10 kHz konnten vergleichbar stabile Verhältnisse wie mit AMP beobachtet werden. Auf eine Belastung mit stressapptest wurde verzichtet.

Wenig Overhead

Neben PreemptRT, das den Linux-Kernel selbst einer Echtzeitfähigkeit sehr gut annähert, und Xenomai, das einen eigenen Hypervisor umfasst, ist AMP eine weitere, sehr hochperformante Architektur, um die Linux-Interface-Vielfalt und Anforderungen hinsichtlich harter Echtzeitfähigkeit „unter einen Hut" (Chip) zu bekommen. AMP-Lösungen bieten durch ihre wesentlich dünneren Software-Schichten die beste Performance. Dieses Ergebnis überrascht daher wenig. Andererseits hängt AMP von der Verfügbarkeit einer MultiCore-CPU ab; Single-Core CPUs sind logischerweise für Systemlösungen dieser Art nicht geeignet. Nachteilig wirkt sich aus, dass Teile der Entwicklung auf Bare Metal umgesetzt werden müssen, was den Aufwand erhöhen kann. Dem gegenüber steht wiederum, dass durch die strikte Software-Trennung Dokumentation, Tests und Validierung zertifizierungspflichtiger Lösungen vereinfacht werden können. Die Auswahl der richtigen Echtzeitarchitektur bleibt also auch weiterhin eine produkt- und applikationsspezifische Designentscheidung.

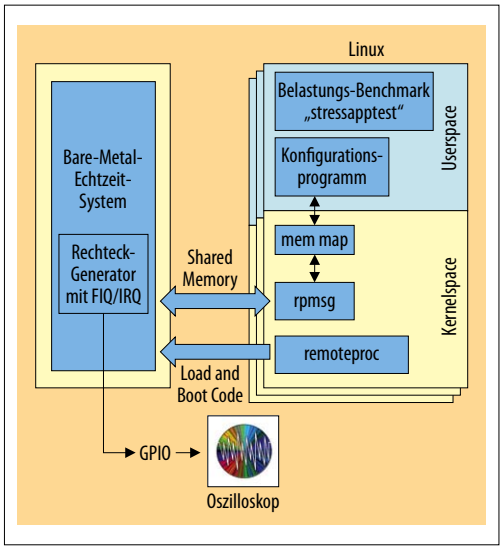


Bild 4. Messaufbau zur Erzeugung eines 50-kHz-Rechtecksignals mit maximaler Stressbelastung und statistischer Messung.ware-Komponenten und deren Kommunikation.

Als Ausblick auf weitere Möglichkeiten mit AMP sei auf ultraschnelles Booten verwiesen. Offen bleibt ebenso die Frage, ob die sehr dünne Software-Schicht in der abgetrennten CPU einer höheren Zertifizierung nach funktionaler Sicherheit (SIL4) standhalten kann. *jk*

Thomas Brinker
ist Senior System Engineer und Project Manager im Berliner Büro der emlix GmbH. Er beschäftigt sich mit Architektur und Design von Embedded-Linux-Geräten in Konsum- und industriellen Geräten sowie deren Vernetzung.
thomas.brinker@emlix.com