Debugging in virtuellen Welten: *Den Hypervisor „durchschauen“*

Um Kosten zu sparen, werden die Funktionen mehrerer Elektronik-Geräte auf einer gemeinsamen Hardware konsolidiert. Für die softwareseitige Trennung der Funktionen sorgt ein Hypervisor. Dadurch wird das Debugging anspruchsvoller, aber keineswegs unmöglich.

Hypervisor – dieses Wort begegnet Entwicklern von Embedded-Software zur Zeit ständig. Es scheint geradezu einen Hype um diese Technologie zu geben (man beachte das Wortspiel). Beispielsweise in den Bereichen Automotive, Luft- und Raumfahrt, sowie Medizintechnik wird gerade viel darüber geredet. Aber wie wirkt sich das auf den Entwicklungszyklus aus, besonders auf die Fehlersuche – also auf das Debugging?

Mehrere Maschinen auf einer Hardware

Insbesondere Debugging-Werkzeuge, die auf die Hardware zugreifen (z.B. JTAG-Debugger), müssen eine Menge berücksichtigen, wenn ein Hypervisor auf dem Zielsystem eingesetzt wird. Nicht zuletzt will natürlich der Entwickler auch ein Werkzeug zur Hand haben, das ihm den Zustand des Embedded-Systems vollständig zeigt, inklusive aller Komponenten wie Hypervisor, Gast-Betriebssysteme und Prozesse in den Gästen.

„Hypervisoren erlauben den simultanen Betrieb mehrerer Gastsysteme auf einem Hostsystem“, lautet die Erklärung in der Wikipedia. Sie werden verwendet, um verschiedene Aufgaben auf einer einzigen Hardware zu parallelisieren. Die Aufgaben sind dabei derartig verschieden, dass für deren Implementierung unterschiedliche Betriebssysteme verwendet werden. Der Hypervisor ist dafür zuständig, diese verschiedenen Betriebssysteme auf einem Computer laufen zu lassen – entweder indem die CPU im Zeitscheiben-Verfahren auf die Betriebssysteme aufgeteilt wird, oder indem in einer Multicore-Umgebung die einzelnen Cores dynamisch unterschiedlichen Gästen zugeordnet werden. Auf dem Desktop-Computer kennt jeder solche Hypervisoren, wie z.B. VMWare oder VirtualBox. Diese erlauben es z.B. eine (oder mehrere) komplette Linux-Distribution(en) unter Windows laufen zu lassen. Andere Beispiele, die auch in Embedded-Systemen zur Anwendung kommen, sind Xen, KVM, Jailhouse und QEMU.

Ein konkreter Anwendungsfall aus dem Bereich der Embedded-Systeme könnte so aussehen: In einem Auto soll ein Armaturenbrett mit einer industriellen Linux-Distribution arbeiten, das Infotainment-System soll unter Android laufen, die Klimatisierung unter FreeRTOS und die Motorsteuerung mit einem AUTOSAR-Stack. Während man früher hierfür tatsächlich vier (und mehr) verschiedene Hardware-Plattformen verwendet hat, werden heute alle diese Funktionen in einem System integriert, möglichst sogar auf einer CPU.

Warum? Zunächst einmal aus Kostengründen. Embedded-Systeme sind heute so leistungsfähig, dass ein einziges System alle diese Aufgaben bewältigen kann. Und natürlich ist es günstiger, ein integriertes Hardware-Modul zu fertigen und zu verbauen, als vier verschiedene. Da gerade in der Automobilindustrie jeder Cent zählt, ist dies wohl die Hauptmotivation. Als „Beigabe“ erhält man mit einem Hypervisor Ausfallsicherheit. Der Hypervisor kann alle Gäste überwachen, und bei einem Problem entsprechend handeln, z.B. indem er einen Gast neu startet. Der Schutz der Gäste gegeneinander muss ebenso gewährleistet sein. Eine technische Voraussetzung dafür ist, dass alle Gäste hardwaremäßig über eine eigene MMU voneinander getrennt werden **(Bild 1)**. Gerade dieses Feature wird uns beim Debugging wieder begegnen.

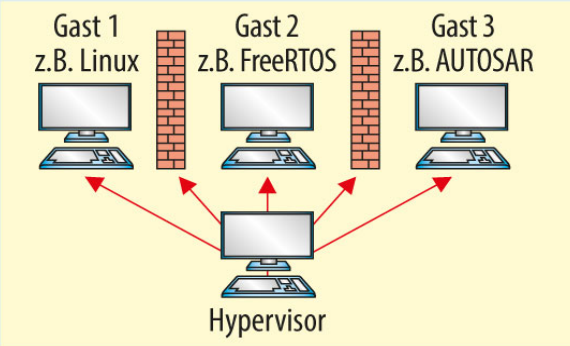

Bild 1. Ein Hypervisor koordiniert den Betrieb mehrerer virtueller Maschinen auf einer realen Maschine. Dabei sorgt er für die strikte Trennung der virtuellen Maschinen.

Lauterbach GmbH • Altlaufstraße 40 • D-85635 Höhenkirchen-Siegertsbrunn • Tel. +49 8102 9876-0 • sales@lauterbach.com • www.lauterbach.com

Funktionsweise eines Hypervisors

Die einzelnen Gäste lassen sich hardware-technisch voneinander trennen, wenn die CPU eine vollständige Hardware-Abstraktion bietet.

Dazu müssen im Wesentlichen drei Dinge virtualisiert werden: der Speicher, die Peripherie, und die CPU selbst **(Bild 2)**. Das Betriebssystem im Gast soll ja gar nicht wissen, dass es in einer virtualisierten Maschine läuft. D.h. das Betriebssystem verwaltet seine eigene MMU („Stage 1 MMU“) und seinen eigenen „physikalischen“ Speicher („guest physical“ = „intermediate“). Allerdings ist dieser nicht wirklich physikalisch, sondern wird dann in einer zweiten MMU („Stage 2 MMU“) des Hypervisors in einen realen physikalischen Adressraum übersetzt. Damit jeder Gast mit der Umgebung interagieren kann, wird auch die Peripherie virtualisiert („virtual I/O“). Der Hypervisor entscheidet dabei, welcher Gast auf welche Peripherie zugreifen darf und auf welche Interrupts der Gast reagiert. Und schließlich erhält jeder Gast eine oder mehrere virtuelle CPUs, die über einen Scheduler auf die tatsächlichen Cores abgebildet werden. Dabei kann die Anzahl der virtuellen CPUs eines Gastes kleiner oder auch größer der Anzahl der realen Cores sein.

Eine Ereigniskette am Beispiel des vorher genannten Automobilsystems würde dann so ablaufen:

- → Ein Temperatursensor erkennt einen Abfall auf unter 3° C und löst einen Hardware-Interrupt aus. Dieser Interrupt wird vom Hypervisor empfangen und bearbeitet. Der Hypervisor leitet den Interrupt als virtuellen Interrupt an den Gast für das Armaturenbrett weiter.
- → Das Gastsystem empfängt den virtuellen Interrupt (bzw. ein Treiber im Gast) und sendet ein Signal an den Prozess innerhalb des Gastes, der für das Warnsignal zuständig ist und „Gefahr: Eisglätte“ anzeigt.

Ein anderes Beispiel soll die Kommunikation zwischen den Gästen aufzeigen: Der Fahrer hat erkannt, dass es draußen kalt ist und drückt am Armaturenbrett die Taste „Heizung ein“. Der Gast, der für das Armaturenbrett zuständig ist, sendet daraufhin – über den Hypervisor – ein Signal an den Gast für die Klimatisierung, dass dieser doch bitte die Heizung einschaltet.

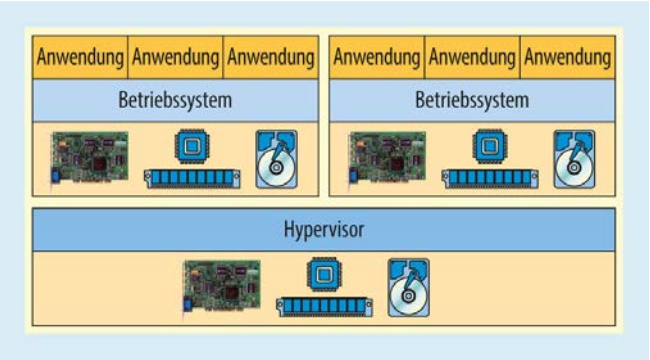

Bild 2 – Der Hypervisor sorgt für eine volle Virtualisierung der vorhandenen Hardware.

Einfluss von Hypervisoren auf Debugger

Soweit, so gut – was aber, wenn während der Entwicklungsphase das System nicht so reagiert wie gewünscht? Wenn etwa der Prozess für das Warnsignal gar nicht aufgerufen wird, oder die Klimatisierung nichts vom Wunsch des Fahrers mitbekommt? Um den Fehler zu finden muss die Software mit einem Debugger untersucht werden.

Prinzipiell gibt es zwei Methoden des Debuggings: das software-gesteuerte Run-Mode Debugging und das hardware-gesteuerte Stop-Mode Debugging.

Beim Run-Mode Debugging wird eine zusätzliche Debug-Software ins System geladen (z.B. „gdbserver“ für Linux-Prozesse), die das eigentliche Debugging bewerkstelligt. Einzelschrittbetrieb, Breakpoints, etc. wird alles von dieser Software (auch „Debug Agent“ genannt) verwaltet. Ein Debugger auf dem Entwicklungsrechner kommuniziert dazu mit dem Agent z.B. über eine serielle Schnittstelle oder Ethernet. Damit das funktioniert, wird nur die zu debuggende Komponente angehalten, z.B. ein Linux-Prozess. Der Rest des Systems läuft weiter (daher „Run-Mode“). Das System muss auch weiterlaufen, damit die Kommunikation mit dem Debugger weiter aufrechterhalten wird.

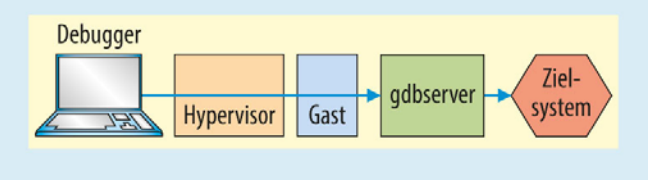

Bild 3. „Run-Mode“ Debugging mit einem gdbserver

Lauterbach GmbH • Altlaufstraße 40 • D-85635 Höhenkirchen-Siegertsbrunn • Tel. +49 8102 9876-0 • sales@lauterbach.com • www.lauterbach.com

Eine solche Debug-Session benötigt lediglich einen entsprechenden Kommunikationskanal. Im Falle eines darunterliegenden Hypervisors wird der Kanal einfach durch diesen hindurch geroutet **(Bild 3)**. Wenn diese Route einmal steht, wissen weder der Debugger, noch der Agent, dass ein Hypervisor dazwischenliegt – d.h. das Debugging ist „Hypervisor agnostisch“. Diese Methode ist perfekt, wenn das System während des Debuggings weiterlaufen muss, z.B. weil Protokolle bedient werden müssen. Zum Debugging von Funktionen innerhalb eines Prozesses oder in Prozessen innerhalb einer Maschine ist dies völlig ausreichend. Sobald aber Treiber (Linux-Module) involviert sind, kommt diese Methode an ihre Grenzen. Erst recht, wenn die VM verlassen wird und andere Gäste oder der Hypervisor mit betroffen sind. Sollte also bei der Ereigniskette oben ein Fehler außerhalb des Prozesses für das Warnsignal liegen, ist eine andere Debugging-Methode erforderlich.

Stop-Mode Debugging: Alles auf Halt

Beim hardware-gesteuerten Debugging wird der Debugger über spezielle Pins direkt an die CPU angeschlossen **(Bild 4)**. Über diese Pins – typischerweise JTAG – kann der Debugger dann die CPU selbst steuern, z.B. anhalten, einzelne Programmschritte auslösen, Register oder Speicher lesen. Das heißt aber auch, bei einem Breakpoint wird das komplette System gestoppt, inklusive aller Prozesse, Gäste und natürlich auch der Hypervisor. Es werden dann keine Interrupts mehr bedient, es laufen keine Kommunikationsprotokolle, und es finden keine VM-, Prozess-, oder Taskwechsel statt. Die CPU ist sozusagen „eingefroren“, daher auch „Stop-Mode“.

Die CPU „sieht“ in diesem Zustand nur die Komponenten, die gerade über die MMU freigegeben sind, also nur einen Gast (den, der gerade auf der CPU läuft) und nur einen Prozess (den, der gerade in dem Gast aktiv ist). Alle Register und Speicherzugriffe sind auf diesen Kontext bezogen. Die CPU hat keinen Zugriff auf andere VMs oder Prozesse. Ein Hardware-Debugger greift ebenfalls über die CPU auf das System zu und unterliegt damit zunächst den *gleichen Beschränkungen: Er „sieht“ nur die aktuelle Situation. Allerdings kann er ein wenig mehr: Durch minimale temporäre Manipulation der MMU Register kann er auch direkt den physikalischen Adressraum und auf den aktuellen „intermediate“ (= „Gast physikalisch“) Adressraum lesen. Allerdings sind alle Debug-Symbole der Prozesse und Gäste auf virtuelle Adressen gelegt, so dass diese zusätzliche Sicht erst mal nicht viel bringt. Trotzdem möchte ein Entwickler alles sehen: den Hypervisor, alle Gäste und auch alle Prozesse aller Gäste, alles auf einmal und alles gleichzeitig! Aus den genannten Gründen ist das im Run-Mode aber prinzipiell nicht möglich. Im Stop-Mode jedoch schon. Das ist seine große Stärke.*

Damit der Debugger über den aktuellen Zustand hinaus alles sehen kann, muss er über das System Bescheid wissen, also eine „Awareness“ haben. Er benötigt eine „Hypervisor Awareness“, eine „OS Awareness“ für jeden Gast und eine „MMU Awareness“ sowohl für den Hypervisor als auch für jeden Gast, die durchaus unterschiedlich ausfallen kann. Da der Debugger das Layout des Systems nun kennt, kann er die Liste der Gäste und Prozesse und deren MMU-Tabellen aus dem System auslesen. Mit diesem Wissen führt der Debugger den notwendigen MMU Table-Walk (Umsetzung der virtuellen in physikalische Adressen) für jede virtuelle Adresse eines Gastes oder Prozesses dann selbst durch, sozusagen an der Hardware-MMU vorbei, und liest die entsprechenden Daten direkt aus dem physikalischen Speicher. Auf diese Weise greift der Debugger auf alle Adressen aller Gäste und aller Prozesse zu, egal ob virtuell, intermediate, oder physikalisch. Und das alles gleichzeitig!

Bild 4. JTAG Debugger mit Zielsystem: So wird ein Hardware-Debugger an das Zielsystem angeschlossen.

Lauterbach GmbH • Altlaufstraße 40 • D-85635 Höhenkirchen-Siegertsbrunn • Tel. +49 8102 9876-0 • sales@lauterbach.com • www.lauterbach.com

Debugger muss „Awareness“ haben

Der Debugger greift also auf ein Zielsystem zu, das aus einem Hypervisor und mehreren Gast-Betriebssystemen besteht. Jede Maschine hat ihren eigenen Satz an Registern, MMU-Umsetzungen, Prozessen, Tasks, Symbolen, Breakpoints, etc. Der Debugger muss mit jeder dieser Maschinen arbeiten können. Sowohl mit der „realen“ Maschine (also dem Hypervisor), als auch mit allen virtualisierten Gast-Systemen, als wären sie reale Maschinen. Selbstverständlich mit allen Maschinen gleichzeitig.

Dafür wird die schon oben erwähnte „Awareness“ eingesetzt. Eine Hypervisor Awareness muss dediziert auf jeden Hypervisor angepasst sein und ermittelt die Liste der virtuellen Maschinen, deren IDs, virtuelle CPUs und die MMU-Einstellungen dazu. Um die notwendige Information aus dem System zu lesen, nutzt die Awareness die Debug-Symbol-Information des Hypervisors (ELF/DWARF). Die Darstellung der virtuellen Maschinen und ihrer Ressourcen gibt schon einen guten Überblick über das System **(Bild 5)**. Die Hypervisor-Awareness ist auch dafür zuständig, das Layout der Stage-2-MMU-Umsetzung zu verwalten, so dass der Debugger Zugriff auf alle VMs hat.

Um den Inhalt eines Gast-Betriebssystems zu analysieren, wird dann eine „OS Awareness“ benötigt – und zwar für jeden Gast eine eigene. Auch hier wird die Awareness zu jedem OS dediziert entwickelt. Diese ermittelt dann die Prozesse des Betriebssystems und die MMU-Einstellungen innerhalb der VM, sowie das Layout der MMU-Tabelle (Stage 1 Translation). Die Awareness nutzt hierzu dann die Debug-Symbol-Information des jeweiligen Betriebssystems, bei Linux z.B. die Datei „vmlinux“.

Damit können die Prozesse, Threads und andere Ressourcen dargestellt werden.

Mit diesen Awarenesses kennt der Debugger schließlich die Maschinen, die Betriebssysteme und die Prozesse, die im Zielsystem laufen. Der von Lauterbach entwickelte Debugger TRACE32 kann damit einen Hierarchiebaum des gesamten Systems darstellen. Diverse Kommandos und Fenster können spezifisch auf eine bestimmte Maschine oder einen bestimmten Prozess angewendet werden. So lassen sich beispielsweise gleichzeitig der Prozess einer Linux-Maschine und die Task eines FreeRTOS-Gastes anzeigen **(Bild 6)**. Die Symbolverwaltung von TRACE32 wurde entsprechend abgeändert, so dass der Entwickler die geladenen Symbole einer bestimmten Maschine, oder einem bestimmten Prozess zuordnen kann. Damit auch der Entwickler jederzeit auf jede virtuelle Adresse zugreifen kann, wurde diese um eine Maschinen-ID und eine Prozess-ID erweitert. So ist jede virtuelle Adresse eindeutig.

Läuft die Software auf einen Breakpoint, so wird, wie oben beschrieben, das komplette System angehalten. Der Debugger schaltet dann automatisch auf den (realen) Core und zeigt die Maschine und den Prozess an, der auf den Breakpoint aufgelaufen ist. Man sieht damit sofort die Umstände, die zu diesem Break geführt haben. Die aktuelle VM ist dann die „Current Machine“. Natürlich kann man manuell auf andere Cores und deren „Current Machines“ umschalten.

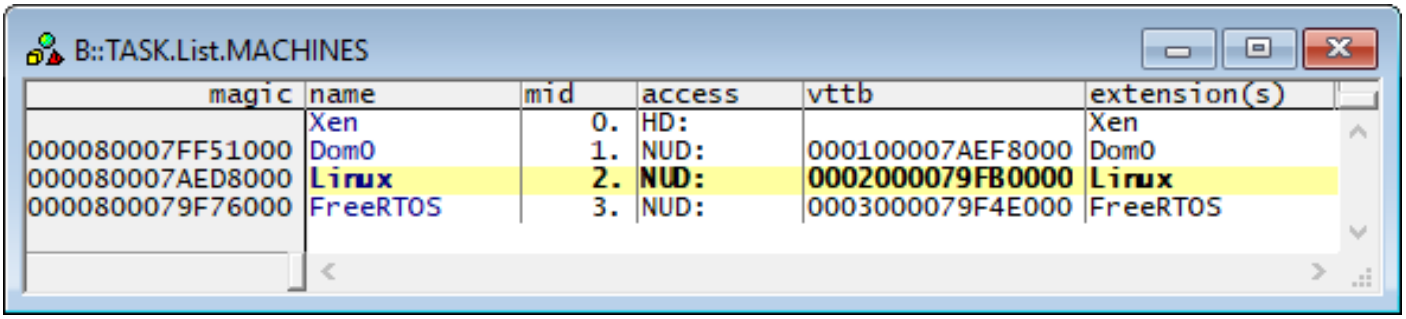

Bild 5. Der Debugger „kennt“ den Hypervisor und die Gast-Maschinen.

Lauterbach GmbH • Altlaufstraße 40 • D-85635 Höhenkirchen-Siegertsbrunn • Tel. +49 8102 9876-0 • sales@lauterbach.com • www.lauterbach.com

Zugriff auf inaktive Gastsysteme

Man kann mit diesem Konzept die Sichtweise aber nicht nur auf andere Hardware-Cores, sondern auch auf andere, zurzeit nicht aktive Gast-Systeme umschalten. So ist jederzeit ein symbolischer Zugriff auf alle Funktionen und Variablen anderer Maschinen möglich. Die Symbole wurden ja für eine bestimmte Maschine geladen. Der Debugger übersetzt die virtuelle Adresse der Symbole in eine physikalische Adresse (er macht den MMU Table-Walk selbst) und liest z.B. den Wert einer Variablen dann aus dem physikalischen RAM. Es ist dabei wichtig, dass zu keiner Zeit der Zustand der CPU verändert wird, sondern alles innerhalb des Debuggers abläuft. Durch den Zugriff auf die Symbole aller Maschinen können auch jederzeit Breakpoints auf jedwede Funktionen irgendeiner Maschine gesetzt werden. Natürlich kann der Debugger auch auf den Registersatz einer bestimmten Maschine oder eines Prozesses umschalten. Wenn die Register gerade nicht in einem realen Core geladen sind, liest der Debugger die Werte dazu aus dem Speicher des Hypervisors oder des Gast-Systems. Aus diesen Werten ermittelt der Debugger den aktuellen Stack Frame um so z.B. die aktuelle Aufrufhierarchie der Funktionen einer Task anzuzeigen. Der Entwickler sieht auf Anhieb, an welcher Stelle die Task zurzeit steht, und warum sie eventuell wartet.

Wie lassen sich diese Funktionen nun zum Debuggen für den Anfangs erwähnten Anwendungsfall nutzen? Zur Erinnerung: Ein Hardware-Interrupt kam nicht beim Prozess im Gast an. Tatsächlich ist es jetzt einfach, dieses Problem zu analysieren. Mit dem Hardware-Debugger setzt man einen Breakpoint direkt auf den Interrupt-Vektor.

Sobald der Interrupt ausgelöst wird, hält das System an. Da der Debugger über alle Komponenten Bescheid weiß, kann der Entwickler nun die Ereigniskette verfolgen, also den Lauf vom Interrupt durch den Hypervisor, über das Gast-Betriebssystem bis zum Prozess im Einzelnen, entweder im Einzelschrittmodus oder durch Breakpoints in den jeweiligen Stufen. Ein Fehler, der sich irgendwo zwischendrin eingeschlichen hat, ist dadurch schnell auffindbar. Schwieriger zu finden sind Deadlocks, wenn sich beispielsweise zwei Kommunikations-Prozesse gegenseitig blockieren. Hier hilft die Systemsicht, in der die Zustände aller beteiligten Komponenten nebeneinander dargestellt werden können. So kann man leicht erkennen welche Tasks welches Gastes – oder auch des Hypervisors – sich in einem nicht erlaubten Zustand befinden, oder vielleicht auf gemeinsame Ressourcen warten. Nicht zu unterschätzen ist auch die Möglichkeit der Post-Mortem-Analyse, wenn das System als Ganzes in einen Zustand läuft, in dem es nicht mehr reagiert. Dadurch dass ein Hardware-Debugger keine aktive Software auf dem Zielsystem benötigt, kann er nun den Zustand jeder beliebigen Komponente untersuchen. Da TRACE32 von Lauterbach auch einen Instruction-Set-Simulator beinhaltet, kann der Entwickler aus einem solchen System einen kompletten Speicher-Abzug ziehen, und diesen bequem zu einem späteren Zeitpunkt ohne echte Ziel-Hardware analysieren, ähnlich einer Coredump-Analyse. Diesmal aber über das gesamte System, inklusive Hypervisor, allen Gästen und allen Prozessen.

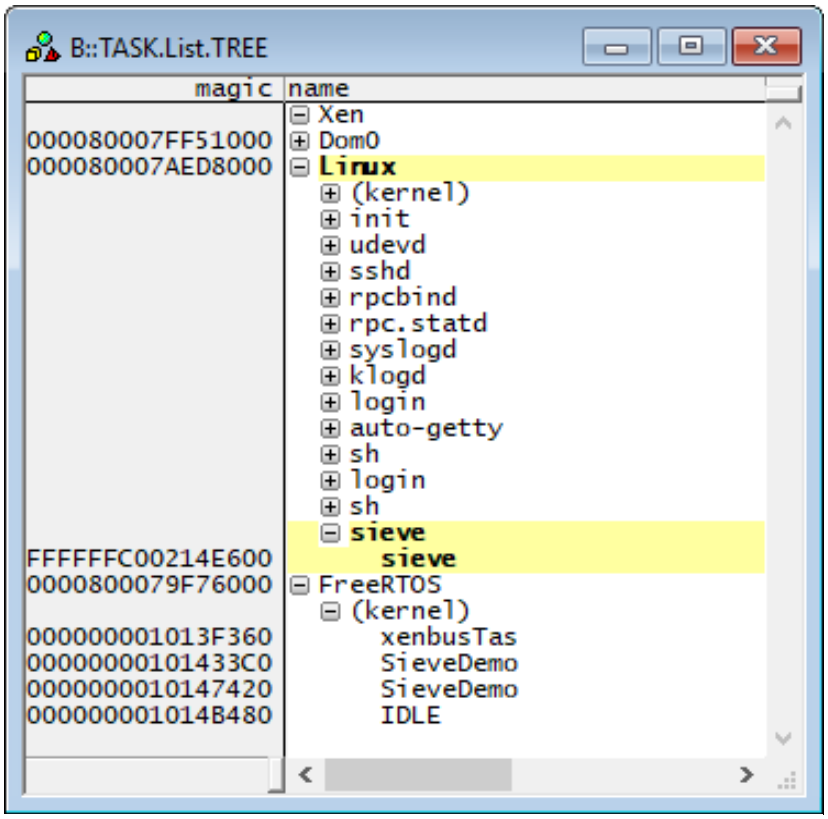

Bild 6. Eine Baumstruktur zeigt das Layout des Zielsystems.

Lauterbach GmbH • Altlaufstraße 40 • D-85635 Höhenkirchen-Siegertsbrunn • Tel. +49 8102 9876-0 • sales@lauterbach.com • www.lauterbach.com

Geringere Kosten, mehr Komplexität

Hypervisoren werden zunehmend auch in der Embedded-Welt eingesetzt. Vorteile wie Kostenersparnis und Laufzeit-Überwachung sprechen klar dafür. Allerdings werden diese durch eine höhere System-Komplexität erkauft.

Die Hardware muss eine Virtualisierung über eine zweistufige MMU-Hierarchie bereitstellen, die vom Hypervisor verwaltet werden muss. Ein Hardware-gestützter Debugger (z.B. über JTAG) benötigt das Wissen um den Hypervisor und um die Gast-Systeme, um dem Entwickler Einsicht in die Software zu geben. Hierzu wird eine dem jeweiligen Hypervisor und dem jeweiligen Gast-Betriebssystem angepasste „Awareness“ in den Debugger geladen, die die notwendige Information aus dem Zielsystem liest.

Lauterbach hat eine Referenzimplementierung mit dem Hypervisor Xen und den Gästen Linux und FreeRTOS auf einem Hikey Board erstellt, die die Funktionalität demonstriert. Der im Debugger TRACE32 implementierte MMU Support und eine Erweiterung der Adressverwaltung auf virtualisierte Systeme erlauben den Zugriff auf alle Komponenten zu jeder Zeit. Das ermöglicht ein Debugging des Hypervisor, aller Gast-Betriebssysteme und aller Prozesse in den Gästen. Auch eine nachträgliche Analyse eines Speicherabbildes ist damit ohne Probleme möglich. Alles in allem wird dem Entwickler ein Werkzeug in die Hand gegeben, mit dem er auch solch komplexe Systeme mühelos debuggen kann.

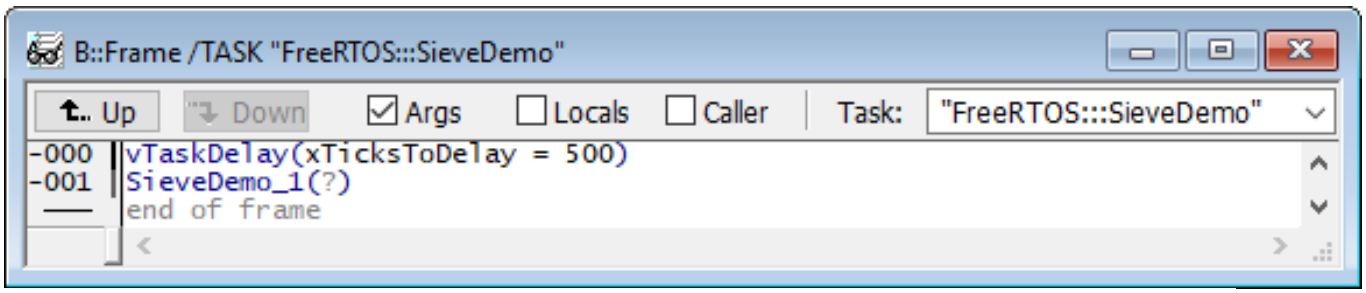

Bild 7. Aufrufhierarchie einer inaktiven Task innerhalb eines inaktiven Gastes.

Lauterbach GmbH
Rudolf Dienstbeck
rudolf.dienstbeck@lauterbach.com

Veröffentlichung in Heft Nr. 20 der Elektronik
vom 04. Oktober 2017