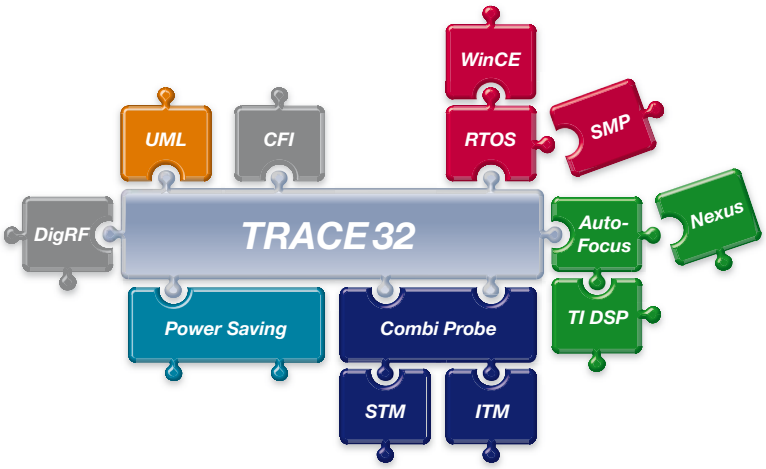
NEWS

Neuheiten zur Embedded World 2008

Im Verborgenen zum Erfolg

Embedded Systeme haben in den letzten 15 Jahren eine rasante Entwicklung genommen. Waren sie früher eigentlich nur in technisch aufwendigen Produkten zu finden, die man nicht in seiner häuslichen Umgebung verwendet, so sind sie heute allgegenwärtig. Von der breiten Öffentlichkeit wird das



vielleicht nicht so wahrgenommen, denn sie sind ja „embedded“ und arbeiten im Verborgenen. Doch wir aus der „Embedded Community“ wissen, jeder normale Bürger kommt inzwischen täglich mit ihnen in Berührung. Im Auto, im Mobiltelefon, im MP3-Player, in der Heimkinoanlage, in der Waschmaschine, im Herd – in fast allen technischen Geräten des täglichen Gebrauchs findet man embedded Systeme mit einem bzw. mehreren Mikrocontrollern, mit digitalen Signalprozessoren und natürlich der dazugehörigen Software. Und wir, die „Embedded Community“ müssen dafür sorgen, dass alles ohne Aufsehen reibungslos funktioniert.

TRACE32 – ein Premiumprodukt

Lauterbach ist vor 29 Jahren in der Pionierzeit der embedded Systeme gegründet worden und konnte die Herausforderungen dieses Marktes mit dem richtigen Know-how und großem Engagement in entsprechende Erfolge – technologische und wirtschaftliche – umsetzen. So wie die embedded Systeme in den Produkten des täglichen Gebrauchs nicht mehr wegzudenken sind, so kommt ein Entwickler in der embedded Branche heute nicht mehr ohne seinen „TRACE32“ aus. Ob in Europa, USA oder Asien, TRACE32 ist als Premiumprodukt anerkannt und weltweit die bekannteste Marke unter den Mikroprozessor-Entwicklungssystemen.

Dies lässt sich auch ganz eindrucksvoll mit Zahlen belegen: Alleine im Jahr 2007 wurden weltweit über 7000 neue Arbeitsplätze mit TRACE32 ausgerüstet!

Die Marke TRACE32 steht für Kontinuität, aber vor allem für Fortschritt und Erfindergeist. Je nach Industriezweig steht das eine oder das andere im Vordergrund. Damit Sie sich alle auch in Zukunft auf Lauterbach als innovativen Partner verlassen können, werden wir im kommenden Jahr intensiv die Weiterentwicklung unserer Produkte vorantreiben und in solides Wachstum investieren.

Natürlich sind wir 2007 nicht stehen geblieben. Was sich im vergangenen Jahr technologisch getan hat, das möchten wir Ihnen gerne in unseren News 2008 näher bringen. Wir wollen Sie vor allem ermutigen, das eine oder andere neue Feature für die Entwicklung zu nutzen, um so Ihre Effizienz zu steigern. Viele dieser Neuerungen präsentieren wir auf der *Embedded World 2008* in Nürnberg, zu der wir Sie, wie jedes Jahr, wieder recht herzlich einladen möchten.

Inhaltsverzeichnis

LAUTERBACH

TRACE32 für Virtual Prototypes

Virtual Prototypes

CoWare®	CoWare Virtual Platform	ARM
Synopsys®	Virtual Platform (ehemals Virtio)	ARM XScale
VaST™	VaST Virtual System Prototype	ARM OAK PowerPC/eTPU SH2A StarCore TeakLite TriCore/PCP V850

Schnelle und effiziente Software-Entwicklung ist heute vielfach der bestimmende Faktor für die Markteinführung neuer Produkte. Um mit dem Start der Entwicklung nicht auf die ersten Hardware-Prototypen warten zu müssen, wird zunehmend mit Software-Modellen der Hardware, so genannten *Virtual Prototypes*, gearbeitet. Sobald ein *Virtual Prototype* für die Zielhardware zur Verfügung steht, kann mit dem Debugging der Treiber, des Betriebssystems und der Anwendung begonnen werden.

Um unseren Kunden bereits in dieser Phase ihres Projekts die professionelle Entwicklungsumgebung TRACE32 anzubieten, unterstützt Lauterbach seit 2007 das Debugging von Software-Modellen. Als Debug-Schnittstelle dient dabei die Debug-API des *Virtual Prototype*.

TRACE32 als GDB Front-End

GDB Front-End

ARM	verfügbar
I386	verfügbar
MIPS32	verfügbar
PowerPC	verfügbar
SH4	geplant
XScale	verfügbar

Seit Jahresbeginn 2007 kann die TRACE32-GUI auch als Bedienoberfläche für das Prozess-Debugging über GDB eingesetzt werden. Die unterstützten Kommunikationsschnittstellen zur Zielhardware sind Ethernet oder RS-232.

Debugging von Anwendungsprozessen

Um bei Bedarf mehrere Anwendungsprozesse gleichzeitig debuggen zu können, stellt Lauterbach seinen eigenen *Debug Agent* – t32server – zur Verfügung. Der t32server wird zunächst über das *Linux Terminal Window* gestartet. Danach können über die TRACE32-GUI auf dem Host mehrere GDB-Server für das Testen einzelner Anwendungsprozesse gestartet werden. Der Dienst t32server übernimmt dabei die Datenvermittlung zwischen der TRACE32-GUI und den einzelnen GDB-Servern (siehe Bild 1).

Weitere Neuerungen

Nachdem Lauterbach mit den News 2007 bereits das „Integrierte Run & Stop Mode Debugging für Embedded Linux“ vorgestellt hat, stellen wir dieses Jahr ein neues Konzept für das Debugging von SMP-Linux vor. Siehe dazu Seite 13.

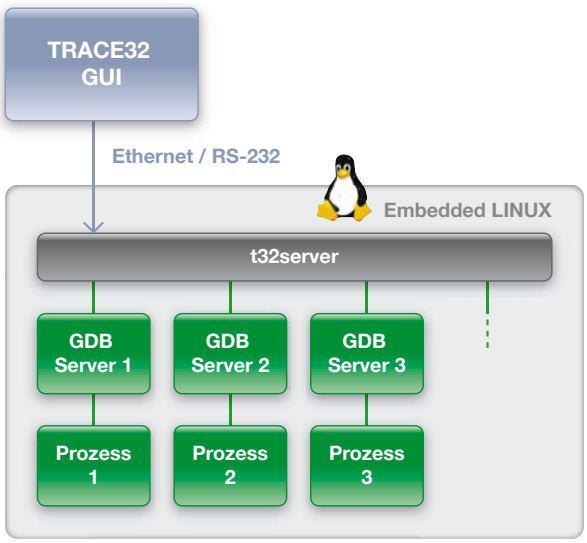


Bild 1: Der Dienst t32server erlaubt das gleichzeitige Debugging mehrerer Anwendungsprozesse.

LAUTERBACH

Integration, Test und Debugging von C-Code gemeinsam mit UML-generierten Sourcen

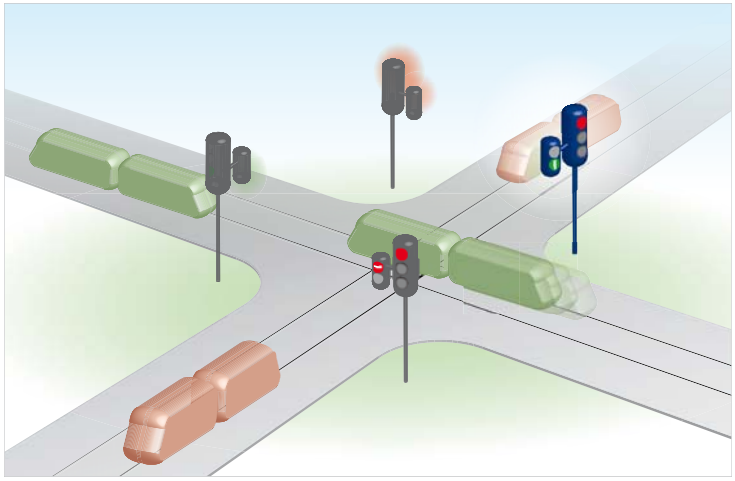


Bild 2: Eine Kreuzung für den Kfz-Verkehr soll um Straßenbahnampeln erweitert werden.

Einführung

Die wachsende Komplexität von Software wird in Zukunft ohne CASE-Tools nicht mehr zu bewältigen sein. Gerade UML ermöglicht ein durchgängiges Design von modularen Softwarekomponenten mit automatisierter Codegenerierung. Typische Codierungsfehler werden auf diese Weise von vornherein ausgeschlossen.

Viele Unternehmen scheuen jedoch den vermeintlichen Aufwand der Umstellung auf diese neuen Techniken. Sie haben eine seit Jahrzehnten gewachsene Codebasis, bei der die Software zwar ständig erweitert und verbessert wurde, jedoch auch immer mehr „Spaghetti-Code“ entstanden ist. Es müssten nun Millionen funktionierende Codezeilen analysiert, umprogrammiert und schließlich erneut getestet werden.

Diese verständliche Angst ist meist unbegründet. Vorhandener C-Code kann (fast) ohne Änderungen in ein UML-Modell-Element portiert werden. Zukünftige Erweiterungen können dann als UML-Modell geschrieben, generiert und mit dem „alten“ C-Code integriert werden. Somit ist ein einfacher und stufenweiser Umstieg von C nach UML möglich. Im Folgenden wird anhand eines Beispiels vorgeführt:

1. Eine Überführung von „altem“ C-Code in UML.
2. Die Integration einer neuen, in UML codierten Funktionalität.
3. Das durchgängige Testen und Debuggen in UML, C++ und C.

Alter C-Code

Die Ausgangslage ist ein fertiges Projekt, das in C vorliegt. Das hier verwendete Beispiel implementiert eine Ampelschaltung (ein unter Software-Designern gern verwendetes, weil einfaches Beispiel). Wir haben eine Standardkreuzung mit vier Ampeln. Die jeweils gegenüberliegenden Ampeln sind gleich gesteuert. Die quer dazu laufenden Ampeln sind – natürlich – entgegengesetzt gesteuert. Die Phasen der Ampeln sind: rot – rotgelb – grün – gelb – rot, wobei in der Rot-Phase auf die andere Richtung umgeschaltet wird. Es gibt also eine kurze Zeitspanne, in der die Ampeln für alle Richtungen rot sind. Wir werden später sehen, dass dies wichtig ist.

Diese Schaltung ist in C codiert, als einfache Zustandssteuerung (switch-case) innerhalb einer Endlosschleife (while (1)). Die Endlosschleife befindet sich direkt in der „main()“ Routine der Applikation und wird gleich nach der Initialisierung aufgerufen (siehe Bild 3).

```
...
int main (void)
{
   horizontal = vertical = red;
   state = closed_to_h;

   while (1)
   {
      switch (state)
      {
          case closed_to_h:
             wait (1);
             horizontal = yellowred;
             state = yellowred_h;
             break;

          case yellowred_h:
             wait (3);
             horizontal = green
             state = green_h;
             break;
          ...
```

Bild 3: Bestehender C-Code für die Ampelschaltung; in der „main()“ Routine befindet sich eine Endlosschleife, welche die Schaltungslogik als Switch-Case-Konstrukt enthält.

Neue Anforderung

Die Straßenplaner haben entschieden, dass durch diese Kreuzung Straßenbahnen laufen müssen (Bild 2), und zwar in jeder Richtung (horizontal ⟩⟩

und vertikal). Für die Straßenbahnen ist die Kreuzung zunächst durch eine eigene Ampel gesperrt. Nach Anforderung durch den Zugführer wird die Kreuzung in der nächsten Rot-Phase für die Autos gesperrt und die anfordernde Straßenbahn erhält das Durchfahrtsrecht.

UML-Wrapper für C-Code

Natürlich könnten wir diese Anforderung in unseren C-Code „einflicken“. Aber auf diese Weise wird die Funktion immer größer und unübersichtlicher, was ja vermieden werden soll. Stattdessen gehen wir das Problem jetzt modular an und wollen es mit Hilfe der UML lösen.

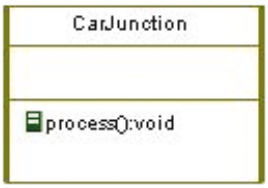


Bild 4: Die Klasse „CarJunction“ enthält die komplette „alte“ Applikation.

Zunächst müssen wir die vorhandene Applikation in unser Modell „überführen“. Das geht einfacher als gedacht: Wir erzeugen eine Klasse „CarJunction“, welche die komplette „alte“ Applikation enthält (siehe Bild 4).

Das einzige, was angepasst werden muss, sind die entsprechenden Schnittstellen. Die sonst übliche Endlosschleife in „main()“ wird schon im Framework bereitgestellt und muss somit entfallen. „main()“ wird in „processJunction()“ umbenannt und bildet den Eintrittspunkt. Als Austrittspunkt der Funktion wird der Zustand gewählt, in dem alle Ampeln auf rot sind. Nun können in der Rot-Phase, falls gewünscht, Aktionen durchgeführt werden (siehe Bild 5). — Das war's.

```
void processJunction (void)
{
   horizontal = vertical = red;
   state = closed_to_h;

   do
   {
      switch (state)
      {
          case closed_to_h:
             wait (1);
             horizontal = yellowred
             state = yellowred_h;
             break;
  ........
          case yellow_v:
             wait (3);
             vertical = red;
             state = closed_to_h;
             break;

      } // switch (state)
   } while((state != closed_to_h)&&(state != closed_to_v));
} // main()
```

Bild 5: Modifizierter C-Code, der in dieser Form in das UML-Werkzeug eingebunden werden kann.

UML-Design für neue Anforderung

Jetzt können wir uns schon um das Design der neuen Anforderung kümmern. Wir bauen uns eine Klasse „Junction“, welche die alte Ampelsteuerung als „CarJunction“ enthält, sowie die beiden neuen Straßenbahnampeln (Bild 6). Damit ist das Design fertig, und wir kümmern uns nun um das Verhalten der erweiterten Kreuzung. Hierzu wird ein Zustandsdiagramm verwendet.

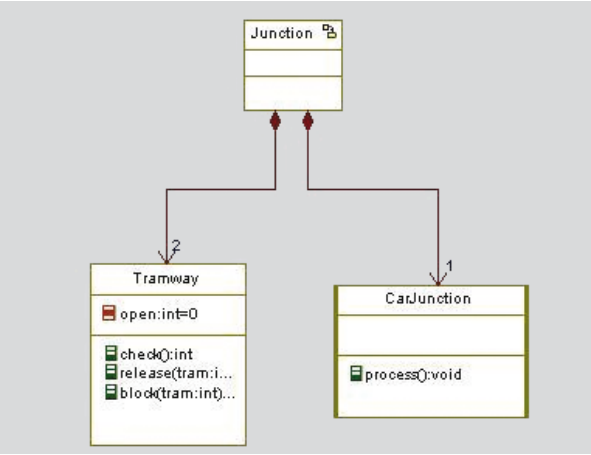


Bild 6: Die neue Klasse „Junction“ integriert die „alte“ Ampelschaltung und die beiden neuen Straßenbahnampeln.

Auch hier wird die alte Applikation als Ausgangsbasis genommen und das komplette Verhalten dieser Applikation als ein einziger Zustand der neuen Kreuzung modelliert (Zustand: cars). Siehe Bild 7 auf der gegenüberliegenden Seite. Wenn das neue Modell sich in diesem Zustand befindet, verhält es sich genauso wie die alte Applikation. Erst wenn das neue Feature angefordert wird – der Zugführer verlangt das Durchfahrtsrecht – wird der alte Code verlassen und die neue Modellierung startet. Im Zustandsdiagramm werden nun die Zustände und die Übergänge der Zustände für beide Straßenbahnlinien spezifiziert: warten – fahren – warten – gesperrt (eventuell mit Wechsel der Straßenbahnlinie). Damit ist die Modellierung der neuen Anforderung erledigt.

Code-Generierung

Nun erzeugen wir die lauffähige Applikation. ⟩⟩

Guten UML-Tools reicht die oben genannte Modellierung aus, um eine fertige Anwendung zu generieren. Jetzt ist es allerdings an der Zeit, einige Punkte zur Integration zu beachten: Die Ausgangs-Applikation ist in C geschrieben und bleibt dies auch. UML-Tools generieren typischerweise C++ Code. D.h. „CarJunction“ ist eine Wrapper-Klasse, die in den C-Code verweist. Dabei sind die üblichen C/C++ Übergänge zu beachten.

Meist wird zur Verwaltung des Modells ein RTOS verwendet, das entsprechend mit eingebunden werden muss. Ist dies alles einmal erledigt, reicht ein Knopfdruck – und aus dem Modell wird die fertige Applikation erzeugt.

Run in Target

Im embedded Bereich unterscheidet sich die Zielhardware vom Entwicklungsrechner. Das bedeutet, wir müssen jetzt die erzeugte Applikation in das so genannte „Target“ laden und dort laufen lassen. Dies kann mit jedem gängigen externen Tool durchgeführt werden, das Code auf ein Target laden kann.

Besonders einfach funktioniert dies, wenn das UML-Tool und der Debugger über eine gemeinsame (Software-)Schnittstelle kommunizieren. Zwischen dem UML-Tool Rhapsody von Telelogic und dem TRACE32-Debugger von Lauterbach besteht eine solche Integration. Diese ermöglicht, dass auf Knopfdruck im Modellierungswerkzeug die Applikation über die Debug-Schnittstelle ins Target geladen wird und – wenn gewünscht – sofort startet. Der komplette Zyklus Design – Modellierung – Generierung – Run ist damit aus einer Oberfläche heraus möglich.

Test in C

Es wurde eingangs schon erwähnt, dass das Testen und Debuggen in einer solchen heterogenen Umgebung bestimmte Anforderungen stellt. Es muss schließlich gewährleistet sein, dass die komplette Applikation in ihrer jeweiligen Funktion und Implementierung getestet werden kann.

Wir gehen zwar davon aus, dass wir eine schon funktionierende Applikation in C hatten. Trotzdem darf die Möglichkeit nicht fehlen, diese zu debuggen. Schließlich muss der Übergang der neuen Features zum „alten“ Bestand getestet und eventuell auch noch die alte Source betreut werden. Zumindest für diesen Teil der Applikation kann jeder handelsübliche Debugger verwendet werden, da das Debugging von C-Code im embedded Bereich heute Standard ist.

Test in C++

Interessanter wird es, wenn wir die neuen Features auf Source-Ebene testen wollen. Die meisten UML-Tools erzeugen C++ Code mit allen dazugehörigen Eigenschaften, wie Templates, Polymorphismus, Exception Handling, etc. Hier ist zu beachten, dass ⟩⟩

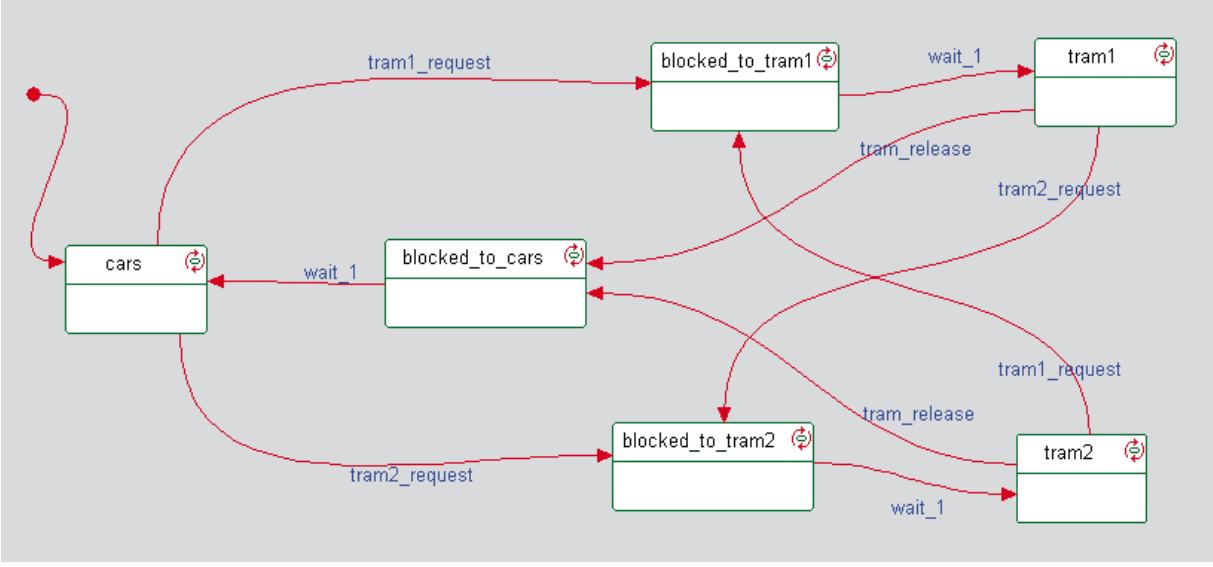


Bild 7: Das Zustandsdiagramm der neuen Ampelschaltung.

der eingesetzte Debugger den jeweiligen C++ Dialekt des Compilers voll beherrscht. Der TRACE32-Debugger von Lauterbach arbeitet mit allen gängigen C++ Compilern zusammen und gewährleistet so ein komfortables Debugging im objektorientierten C++ Code.

Test in UML

Aber eigentlich haben wir die Applikation ja gar nicht in C++ geschrieben, sondern in UML. Was liegt da näher, als auch auf der Modellierungsebene zu debuggen?

Rhapsody von Telelogic beispielsweise bietet hierzu verschiedene Möglichkeiten an.

Wenn das Verhalten der Applikation komplett in UML modelliert ist, kann aus dem Modell heraus der Ablauf bereits simuliert werden. Es ist somit für die reine Verhaltensanalyse keine aktuelle Zielhardware notwendig. Natürlich wird sich das Target zeitlich anders verhalten als die Simulation, daher kann die Applikation auch als so genannte „Animation“ auf dem echten Target laufen. Über einen Kommunikationskanal (seriell oder Ethernet) kann dabei das UML-Tool den Ablauf im Target steuern und visualisieren. So kann man beispielsweise „Single Steps“ durch *State Charts* durchführen oder Events einschleusen.

In Kombination mit dem TRACE32-Debugger kann diese Kommunikation über die Debug-Leitung erfolgen, wenn z. B. keine anderen freien Schnittstellen verfügbar sind. Über „Animation“ wird ein direktes Debugging im UML-Modell ermöglicht.

Integriertes Debugging UML -> C++ -> C

In dieser Situation arbeiten wir auf drei Ebenen: UML, C++ und C. Besonders im C++ Code möchte man, wenn man einen Fehler findet, diesen im Modell beheben und nicht im generierten C++ Code.

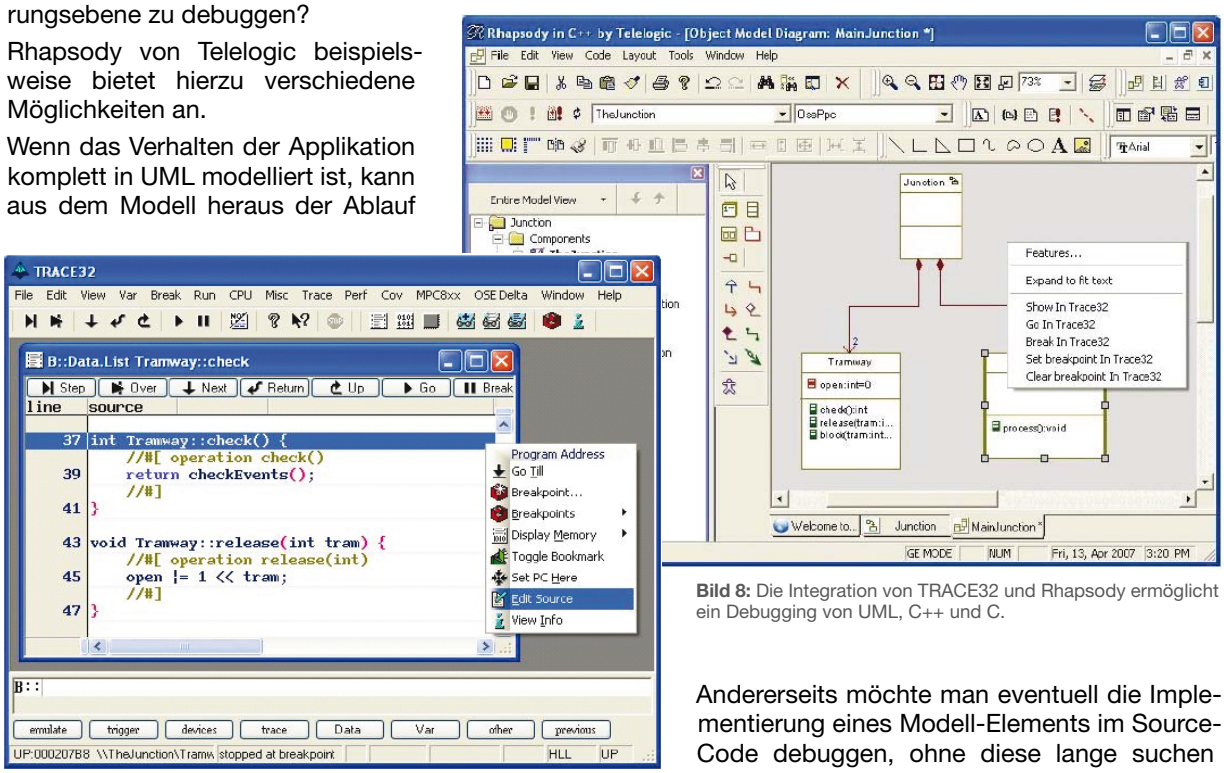


Bild 8: Die Integration von TRACE32 und Rhapsody ermöglicht ein Debugging von UML, C++ und C.

Andererseits möchte man eventuell die Implementierung eines Modell-Elements im Source-Code debuggen, ohne diese lange suchen zu müssen.

Die Integration von TRACE32 und Rhapsody bietet daher weitreichende Funktionalitäten, um diese Ebenen zu verbinden (Bild 8). So ist eine einfache wechselseitige Navigation implementiert. Ein Mausklick auf ein Modell-Element in Rhapsody genügt, um in TRACE32 den dazugehörigen Source-Code anzuzeigen. Es kann dann eine weitergehende Untersuchung von Variablen, Funktionsaufrufen etc. folgen. Umgekehrt ist es auch möglich, im Debugger auf eine Source-Zeile zu klicken und automatisch im UML-Tool das zugehörige Modell-Element zu markieren. Das ist besonders hilfreich, wenn man im ⟩⟩

Debugger einen Fehler findet, den man dann im Modell beheben will. Ein langwieriges Suchen und Navigieren zum entsprechenden Element entfällt so.

Es können im UML-Modell auch Debug-Breakpoints gesetzt und das Target in den Go- oder Stop-Status versetzt werden. Somit kann man einen Breakpoint auf eine Klasse setzen und das Target starten, alles innerhalb des UML-Tools. Der Debugger hält dann die Applikation an, sobald das zu untersuchende Element angesprungen wird.

Fazit

Es muss nicht immer ein komplettes Redesign sein. Gerade bei Projekten, die aus Zeit- oder Organisationsgründen kein komplettes Re-Engineering erlauben, ist oft ein stufenweiser Ansatz richtig. Die Umstellungsmaßnahmen halten sich dabei in Grenzen. Dafür eröffnet sich die Möglichkeit, die bestehende Software schrittweise in ein modernes und zukunftsweisendes CASE-Tool zu portieren.

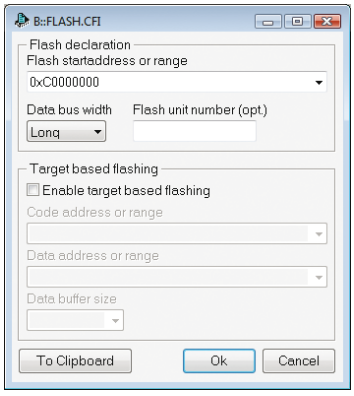
Voraussetzung ist natürlich, dass die dabei eingesetzten Tools diesen Schritt unterstützen anstatt ihn zu behindern. Die Auswahl der Werkzeuge ist deshalb ein wichtiger Punkt. Schließlich müssen diese einen solchen Ansatz nicht nur zulassen, sondern auch fördern.

Lauterbach GmbH und Telelogic haben sich mit der Integration ihrer Tools TRACE32 und Rhapsody genau diesem Anforderungsprofil angenommen. Beide Firmen wollen ihren Kunden dabei helfen, die Qualität der Software trotz steigender Komplexität und immer mehr Anforderungen auf hohem Niveau zu halten. Einer Modernisierung alter Code-Basen in Richtung UML steht so nichts mehr im Wege.

Aktuelles kurz notiert

CFI Flash-Programmierung

Seit September 2007 wird eine automatische Generierung der Flash-Deklaration über das *Common Flash Memory Interface* – kurz CFI – unterstützt.



Um ein Flash zu programmieren, waren bisher umfassende Kenntnisse über den genauen Typ und die Organisation des Bausteins notwendig. Durch den speziellen Query-Modus CFI-konformer Flashbausteine kann TRACE32 nun die für die Programmierung benötigten Parameter selbstständig ermitteln und auf Basis dieser Daten automatisch eine Flash-Deklaration erstellen.

Serielles Flash / NAND-Flash

Seit September 2007 unterstützt TRACE32 die Programmierung von NAND-Flashbausteinen sowie von seriellen Flashbausteinen mit einem eigenen Kommando (*FLASHFILE*).

Code Overlays

Seit Dezember 2007 unterstützt TRACE32 die Symbol-verwaltung für *Code Overlays* in ELF-Files mit dem Kommando *sYmbol.OVERLAY*. Aktuell werden die ARC-, die ARM- und die PowerPC-Architekturen unterstützt. Weitere Architekturen sind für 2008 geplant.

PCP-Debugger

Seit November 2007 muss für das Debugging des *Peripheral Control Processors* – kurz PCP – des TriCore-Prozessors von Infineon eine eigene TRACE32-Instanz gestartet werden. Diese Neuerung war notwendig, um das Laden und Testen von C-Code für den PCP zu ermöglichen. Selbstverständlich wird nun auch die Konfiguration des PCP-Trace und die Auswertung der Traceinformation über diese Instanz durchgeführt. Die Lizenz für das PCP-Debugging und -Tracing ist bereits im Lieferumfang des TRACE32-Debuggers für den TriCore enthalten.

Neu unterstützte Prozessoren

Neue Architekturen

P.A. Semi	PWRficient™	Q2/2008
Renesas	H8SX	verfügbar
Tensilica	Xtensa Processors	verfügbar
Texas Instruments	TMS320™C28xx	verfügbar

PWRficient

Der Dual-Core Prozessor PA6T-1682M wird der erste 64-Bit PowerPC der PWRficient-Architektur sein, für den ein Lauterbach Debugger eingesetzt werden kann.

TMS320C28xx

Mit dem Debugger für die TMS320C28xx DSPs erweitert Lauterbach seine Unterstützung für die *TMS320 Processor Platform* von Texas Instruments.

Neue Derivate

AMCC	**PPC405** - PPC405EX/EXR **PPC44x** - PPC460EX
ARC	**ARC** - ARC® 700 Core - ARCtangent-A4 - ARCtangent-A5
Freescale™	**ColdFire** - MCF52110/MCF52100 - MCF5221x - MCF523x/MCF532x - MCF5372/MCF5373 **MPC5500** - MPC551x - MPC56xx **MPC5200** - MPC512x mit AXE Audio Prozessor **PowerQUICC II** - MPC8377 **PowerQUICC III** - MPC8572

Freescale™ (Forts.)	**ARM11/StarCore** - MXC91321
Infineon	**C166S** - XC22xx/XC23xx/XC27xx - XE166 **TriCore** - TC1736 - TC1767/TC1767ED - TC1797/TC1797ED
Microchip	**MIPS32** - PIC32™
NXP	**ARM9** - LPC3000 **MIPS32** - PNX83xx/PNX85xx
STMicro-electronics	**MPC5500** - SPC563Mxx - SPC560Bxx/Pxx/Sxx **Cortex-M** - STM32
Texas Instruments	**Cortex-A/DSP** - OMAP3430

LAUTERBACH

Debugging mit Stromsparmodi für ARM-Cores

Wegen der stetig steigenden Anforderungen nach sparsamem Energieverbrauch und langen Standby-Zeiten verfügen moderne Cores oft über mehrere Energiespar-Mechanismen. Diese erlauben der Software, wenn immer möglich, entweder die Core-Frequenz herabzusetzen oder den Core gleich ganz abzuschalten. Diese so genannten Stromsparmodi können jedoch die Kommunikation zwischen Debugger und Core beeinträchtigen. Im Folgenden werden, am Beispiel der ARM-Architektur, einige Varianten für ein problemfreieres Debugging von Programmen, die Stromsparmodi verwenden, aufgezeigt.

Stromsparmodi ignorieren

Der erste und einfachste Weg ist natürlich, den Core anzuweisen, während des Debuggens die Stromsparmodi zu ignorieren. Der ARM11 verfügt beispielsweise über ein so genanntes *Powerdown Disable Bit*. Durch das Setzen dieses Bits aktiviert der Core eine Leitung, die das externe Power- und Clock-Management über das Debugging informiert. Versucht das Programm nun einen Stromsparmodus zu aktivieren, kann das Power- und Clockmanagement diesen einfach ignorieren, um Probleme während des Debuggings zu vermeiden. Das Setzen des *Powerdown Disable Bits* gehört zu den Grundeinstellungen des TRACE32-Debuggers für den ARM11, Cortex-A und Cortex-M.

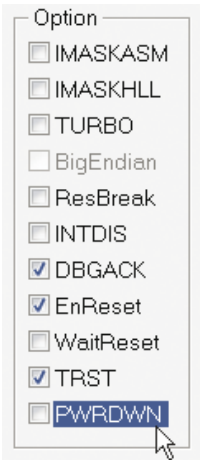


Bild 9: Die Option PWRDWN regelt, ob die Stromsparmodi beim Debugging ignoriert oder beachtet werden.

Für den Fall, dass der Entwickler Tests durchführen möchte, die das Verhalten seines Designs im Zusammenhang mit den Stromsparmodi überprüfen, kann er das *Powerdown Disable Bit* durch eine Debugger-Option selbstverständlich zurücksetzen (*SYStem.Option PWRDWN ON*), siehe Bild 9.

Es gibt aber auch viele ARM-basierte Chips, die nicht über ein *Powerdown Disable Bit* verfügen. In diesem Fall könnte man eine spezielle Debug-Version der Software erstellen, die keine Anweisungen für die Aktivierung von Stromsparmodi enthält. Dieses Vorgehen hat jedoch den Nachteil, dass sich die Debug-Version der Software von der offiziellen Version unterscheiden würde.

Aber warum kann es zu Problemen beim Debugging von Stromsparmodi kommen? Im ersten Teil dieses Artikels betrachten wir zunächst *Single Core Chips:* Wie wirkt sich das Abschalten der Spannung bzw. des Clocks auf das Debugging aus? Anschließend untersuchen wir *Multi Core Chips*. Aus der Vielzahl möglicher Herausforderungen konzentrieren wir uns auf die Frage, welche Auswirkungen das Abschalten des Clocks eines einzelnen Cores auf das Debugging des gesamten Chips hat.

Abschalten der Spannung

Über den VTREF-Pin (target voltage reference) des JTAG-Steckers kann der Debugger das Abschalten der Spannung erkennen. Konfiguriert der Anwender den Debugger für die Verwendung dieser Stromspartechnik, kann dieser, statt eine Fehlermeldung auszugeben, auf das Wieder-Einschalten der Spannung warten (StandBy-Mode).

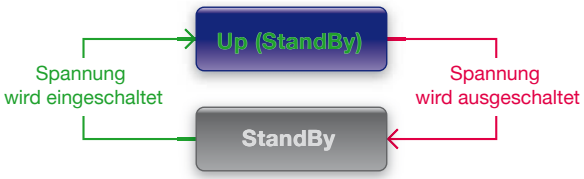


Bild 10: Statt eine Fehlermeldung auszugeben, wartet der Debugger nach dem Abschalten der Spannung auf ihr Wieder-Einschalten.

Praktisch gesehen kann sich der Debugger in einem der beiden folgenden Zustände befinden:

- **Up (StandBy):** Der Prozessor hat Spannung und es besteht eine Kommunikation zwischen Debugger und Prozessor.
- **StandBy:** Die Spannung ist abgeschaltet und der Debugger hält den ARM-Prozessor über die SRST-Leitung im Reset. Der Debugger wartet auf das Wieder-Einschalten der Spannung (siehe Bild 10).

Beim einem Wieder-Einschalten der Spannung gibt der Debugger die Reset-Leitung frei und der Prozessor läuft wie nach einem *Power On Reset* los.

Durch den *Power On* des ARM-Prozessors wurde jedoch auch die Debug-Logik, die ETM/ETB-Logik sowie die Freischaltung für die ETM-Pins zurückgesetzt. Die Einstellungen, die vor dem Abschalten der Spannung aktiv waren, sind damit gelöscht. ⟩⟩

Ein nahtloses Fortsetzen des Debuggings ist jedoch nur dann möglich, wenn es dem Debugger gelingt, diese Einstellungen zu restaurieren, bevor der ARM-Prozessor mit der Bearbeitung des Programms beginnt.

Damit der Debugger beim Wieder-Einschalten der Spannung die notwendigen Restaurierungen durchführen kann, müssen von der Zielhardware einige Voraussetzungen erfüllt sein.

Der Idealfall

Vom Idealfall sprechen wir dann, wenn folgende Anforderungen erfüllt sind:

1. Der Debugger kann den ARM-Prozessor über die Leitung SRST im Reset halten.
2. Der Prozessor-Reset SRST und der Reset für die Debug-Logik TRST sind getrennte Leitungen.
3. Über den VTREF-Pin der JTAG-Schnittstelle kann der Debugger schnell erkennen, dass die Spannung abgeschaltet wurde.

Sind diese Anforderungen erfüllt, lässt der Debugger zunächst nur den Reset für die Debug-Logik los, sobald er das Wieder-Einschalten der Spannung erkennt. Die Einstellungen der Debug-Logik und die ETM/ETB-Konfiguration wird nun restauriert. Anschließend wird dann der Prozessor-Reset freigegeben und das Debugging kann, wenn auch zeitlich verzögert, nahtlos fortgesetzt werden (siehe Bild 11).

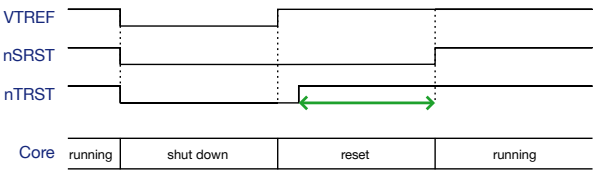
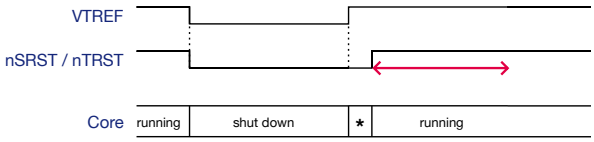


Bild 11: Nach dem Wieder-Einschalten der Spannung wird zunächst die Debug-Logik und die ETM/ETB-Konfiguration restauriert (grüner Pfeil). Dann erst beginnt der ARM-Prozessor die Abarbeitung des Programms.

Da die Konfiguration der I/O-Pins über *Memory Mapped Registers* erfolgt, können die Pins für den ETM-Traceport nicht restauriert werden, bevor das Programm losläuft. Das Programm muss also selbst möglichst schnell den ETM-Port aktivieren.

Sonstige Fälle

Sind die Voraussetzungen 1 und/oder 2 nicht erfüllt, läuft der ARM-Prozessor bereits beim Wieder-Einschalten der Spannung los (siehe Bild 12), während



(* = reset)

Bild 12: Läuft der Prozessor bereits los, während der Debugger noch die Debug-Logik und die ETM/ETB-Konfiguration restauriert (roter Pfeil), kann es zu unerwünschten Seiteneffekten kommen.

der Debugger gleichzeitig die Debug-Logik und die ETM/ETB-Konfiguration restauriert. Dies kann zu unerwünschten Seiteneffekten führen. Gleichzeitig sind natürlich die *On-Chip Breakpoints* und die Traceaufzeichnung in der Restaurationsphase inaktiv.

Abschalten des Clocks

Das Abschalten des Prozessor-Clocks stellt den Debugger vor eine ganz andere Herausforderung. Hier geht es um die Kommunikation zwischen Debugger und Prozessor, oder genauer gesagt, um die Kommunikation zwischen Debugger und On-Chip Debug-Logik. Diese Kommunikation läuft praktisch immer, auch dann, wenn der ARM-Prozessor das Programm abarbeitet.

Solange die Debug-Logik unabhängig vom Prozessor-Clock arbeitet, funktioniert die Kommunikation, auch wenn der Prozessor-Clock abgeschaltet wird. Zur Taktung der eigentlich asynchron standardisierten Debug-Logik wird jedoch häufig ein heruntergeteilter und verzögerter Prozessor-Clock verwendet. Als Anwender erkennt man dies leicht daran, dass bei der Konfiguration des Debuggers das Kommando *SYStem.JtagClock RTCK* verwendet werden muss.

Das Abschalten des Prozessor-Clocks kann die Kommunikation zwischen Debugger und On-Chip Debug-Logik beeinträchtigen, wenn die beiden folgenden Voraussetzungen erfüllt sind:

1. Die Taktung der Debug-Logik hängt vom Prozessor-Clock ab.
2. Der Prozessor-Clock wird mitten in der Kommunikation zwischen Debugger und On-Chip Debug-Logik abgeschaltet.

Beim Abschalten des Prozessor-Clocks passiert nun nämlich folgendes:

1. Der ARM-Prozessor friert seinen aktuellen Zustand ein.
2. Die Debug-Logik friert ebenfalls ihren aktuellen Zustand ein, da ihre Taktung ja vom Prozessor-Clock abhängt.

››

3. Der Debugger bricht die laufende Kommunikation mit der On-Chip Debug-Logik nach einem *Time Out* ab, da diese nicht mehr antwortet.

Beim Wieder-Einschalten des Prozessor-Clocks setzt die Debug-Logik nun die Kommunikation mit dem Debugger exakt an der Abbruchstelle fort, während der Debugger die Kommunikation seinerseits längst erneut aufgesetzt hat, da er von der Debug-Logik keine Antwort erhielt.

Da die Debug-Logik des ARM-Prozessors vom Debugger gesteuert wird, müsste dieser die Debug-Logik nun zunächst zurücksetzen, um die Kommunikation wieder aufzusynchronisieren. Damit würden aber, zumindest für einen kurzen Zeitraum während des Programmlaufs, auch alle *On-Chip Breakpoints* gelöscht. Somit ist dieser Weg nicht praktikabel.

Die folgenden Lösungsansätze haben sich dagegen beim Kunden bewährt:

1. Prozessor-Clock überprüfen

Wir empfehlen, den Debugger so zu konfigurieren, dass er vor Beginn der Kommunikation mit der On-Chip Debug-Logik überprüft, ob der Prozessor-Clock noch läuft. Dazu versetzt der Debugger die Debug-Logik am Ende jeder Kommunikation in den *Run Test/Idle Mode*. In diesem Mode lässt sich der Prozessor-Clock vor einer erneuten Kommunikation einfach testen.

2. Reduktion der Statusabfragen

Während der ARM-Core das Programm abarbeitet, sendet der Debugger 10-mal pro Sekunde Statusabfragen an die Debug-Logik. Damit überprüft er, ob das Programm immer noch läuft oder inzwischen angehalten wurde. Ein nächster Schritt ist, die Häufigkeit dieser Statusabfragen zu reduzieren. Empfehlenswert ist, diese Abfrage nur noch alle 2-3 Sekunden durchzuführen (*SETUP.URATE 3.s*).

3. Time Out verlängern

Wenn das Programm sehr oft, aber immer nur sehr kurz den Prozessor-Clock abschaltet, kann der Debugger so konfiguriert werden, dass er einfach länger wartet, bevor er die Kommunikation mit der Debug-Logik abbricht (*SYStem.POLLING SLOW*). Er würde so unter Umständen ein Abschalten des Prozessor-Clocks gar nicht bemerken.

Multi Core Chips

Multi Core Chips, die zum Debuggen mehrerer verketteter Cores nur eine JTAG-Schnittstelle verwenden, erfordern zusätzliche Maßnahmen, wenn für einzelne Cores Stromsparmodi verwendet werden. Betrachten wir beispielhaft die folgende Situation: Die Taktung der Debug-Logik hängt für Core 2 vom Core-Clock ab (RTCK) und genau für diesen Core wird nun der Clock abgeschaltet (Bild 13).

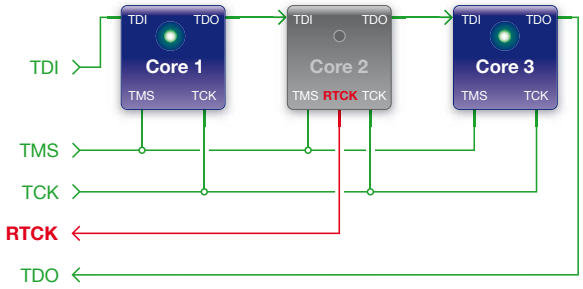


Bild 13: Das Abschalten des Clocks für Core 2 blockiert das Debugging für den gesamten Chip.

Wie bereits im Abschnitt „Abschalten des Clocks“ beschrieben, wird in diesem Fall die Debug-Logik des Core 2 eingefroren. Da durch die Verkettung der Cores Debug-Kommandos für den Core 1/Core 3 den Core 2 durchlaufen müssen, ist nun das Debugging für den gesamten Chip blockiert.

Um beim Wieder-Einschalten des Clocks für Core 2 das Debugging komplikationsfrei fortsetzen zu können, empfiehlt sich, die Debugger für die einzelnen Cores so zu konfigurieren, dass vor Beginn der Kommunikation mit der Debug-Logik überprüft wird, ob der Core-Clock noch läuft.

Damit erhält man jedoch noch keine Lösung für das Problem, dass das Debugging für den gesamten Chip blockiert ist, sobald die Debug-Logik eines einzelnen Cores eingefroren ist. Für diese Thematik gibt es heute bereits einige gute Lösungsansätze. Die einzelnen Chiphersteller betrachten ihre Lösungen jedoch als vertraulich, so dass diese hier nicht vorgestellt werden können.

Wir beschränken uns deshalb im Folgenden darauf zu erläutern, wie die CoreSight DAP-Technologie dieses Problem auflöst.

Bei *Multi Core Chips,* deren Debug- und Trace-Funktionalität auf der CoreSight-Technologie ⟩⟩

basiert, kommuniziert der TRACE32-Debugger nicht mehr direkt mit der Debug-Logik der einzelnen Cores, sondern zunächst mit einem so genannten DAP (*Debug Access Port*). Die Aufgabe des DAPs ist es, die Debug-Kommandos an die einzelnen Cores zu verteilen.

Für den Fall, dass man ein Debugging mit Stromsparmodi durchführen möchte, ist die wichtigste Neuerung der DAP-Technologie wohl, dass der DAP über seine eigene Power- und Clock-Domain verfügt. Damit bleibt der DAP von den Stromsparmodi der einzelnen Cores unberührt. Gleichzeitig sind unter CoreSight die Cores nicht mehr miteinander verkettet, sondern ihre Debug-Logik wird:

- über Buszugriffe kontrolliert, wenn die Debug-Logik bereits über *Memory Mapped Debug Register* verfügt.
- über einen *JTAG Access Port* gesteuert, wenn der Core noch mit der traditionellen Debug-Logik arbeitet.

Diese neuen Ansteuermechanismen garantieren, dass der DAP die einzelnen Cores immer direkt ansprechen kann und keine Debug-Kommandos mehr durch eventuell eingefrorene Cores blockiert werden (siehe Bild 14).

Grundsätzlich gilt für *Multi Core Chips*: Wird in der Entwicklung besonderer Wert auf das Debugging von Stromsparmodi gelegt, empfiehlt es sich diese Thematik bereits bei der Chipauswahl mit dem Halbleiterhersteller zu diskutieren. Gerne beraten die Experten des Lauterbach-Teams Sie zu diesem Thema.

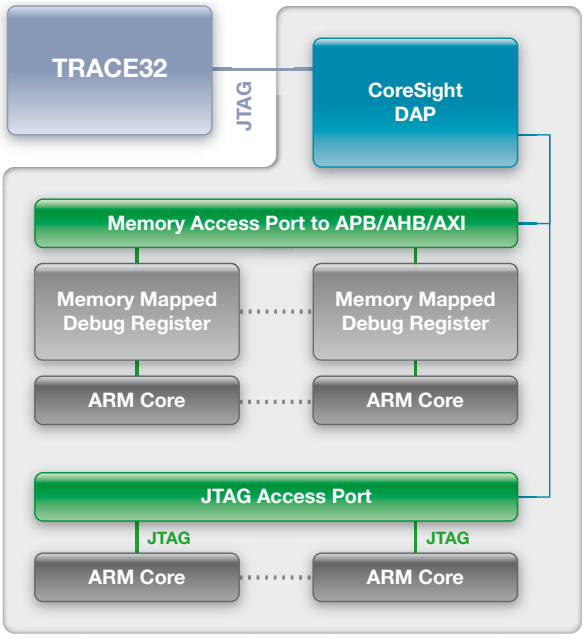


Bild 14: Der DAP kann die Debug-Logik der einzelnen Cores immer direkt ansprechen. Damit ist es nicht mehr möglich, dass das Abschalten des Clocks für einen einzelnen Core das Debugging des gesamten Chips blockiert.

Analyse des Energieverbrauchs

Für alle ARM-basierten Chips mit ETM und einem Off-Chip Traceport bietet Lauterbach die Möglichkeit zu analysieren, ob die verwendeten Stromsparmodi wirklich zu einer maximalen Reduktion des Energieverbrauchs für das Design geführt haben. Um diese Frage zu beantworten, ist eine Messanordnung notwendig, die den Strom- und Spannungsverlauf an ausgewählten Messpunkten aufzeichnet und zu den am Traceport ausgegebenen Programmfluss-Informationen in Bezug setzt.

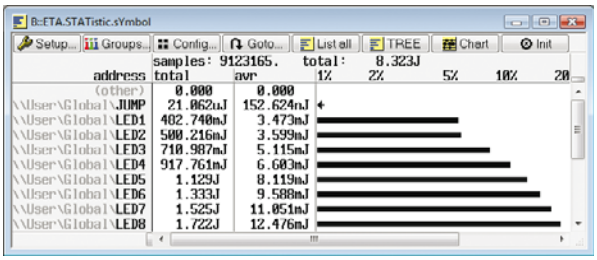


Bild 15: Der Zusammenhang zwischen dem Programm und der Strom-/Leistungsaufnahme der Hardware lässt sich graphisch darstellen.

Seit 2007 bietet Lauterbach eine solche Messanordnung an. Dazu muss der TRACE32-Debugger um Folgendes ergänzt werden:

- einen Echtzeit-Trace für die ETM
- eine *Analog Probe* und einen Lauterbach *Logic Analyzer*

Als *Logic Analyzer* kann entweder der *IProbe Logic Analyzer* im PowerTrace II oder der TRACE32-PowerIntegrator verwendet werden. Da in dieser Messanordnung sowohl der Strom- und Spannungsverlauf als auch der Programmfluss von der TRACE32-Software mit einem synchronen Zeitstempel markiert werden, lassen sich die Zusammenhänge zwischen Steuersoftware und Strom-/Leistungsaufnahme einfach graphisch darstellen (siehe Bild 15). Die wichtigen Kenndaten Energieverbrauch und Standby-Zeiten lassen sich so verifizieren.

Neues Debug-Konzept für Symmetrisches Multiprocessing (SMP)

Lauterbach, der weltweit führende Hersteller hochwertiger Debugger und Echtzeit-Trace-Tools, stellt auf der Embedded World 2008 sein neues Debug-Konzept für SMP-Systeme vor. SMP-Systeme setzen ein Betriebssystem ein, um Prozesse dynamisch auf mehrere Cores bzw. Hardware Threads zu verteilen. Als Live-Demonstration wird auf der Messe SMP-Linux auf einem MIPS 34K präsentiert.

Hardwareseitige Parallelisierung

Um eine höhere Rechenleistung für komplexe Aufgaben zu erreichen und Strom zu sparen, wird heute zunehmend auf die Parallelisierung von Abläufen gesetzt. Am weitesten verbreitet ist der Ansatz, mehrere identische Ausführungseinheiten bereitzustellen, die alle die gleichen Aufgaben bearbeiten können.

Bei dem Begriff „identische Ausführungseinheiten“ denken die meisten Leser sicher zunächst an symmetrische Multi-Core Prozessoren. Man kann sich leicht vorstellen, dass der Kernel eines Betriebssystems fest auf einem Core läuft und dafür sorgt, dass die Anwendungsprozesse gleichmäßig auf alle Cores verteilt werden (siehe Bild 16).

Für die Parallelisierung von Abläufen benötigt man jedoch nicht zwangsläufig einen Multi-Core Prozessor. Hardwareseitiges Multithreading beispielsweise ist ein Ansatz, der eine Parallelisierung auch für Single-Core Prozessoren ermöglicht. Hierbei wird bei

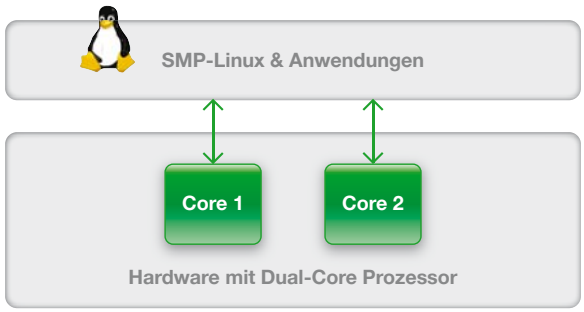


Bild 16: : SMP-Linux auf einem Dual-Core Prozessor.

einem Grundproblem von Cores mit Pipeline-Architektur angesetzt: *Cache Misses* oder Datenabhängigkeiten zwischen den einzelnen Befehlen führen hier nämlich immer wieder dazu, dass die Pipeline-Verarbeitung der Befehle angehalten werden muss, um auf die Verfügbarkeit der passenden Daten zu warten. Je größer die Differenz zwischen Befehlsausführzeit und Speicherzugriffszeit ist, umso mehr Rechenleistung geht verloren.

Hardwareseitiges Multithreading versucht dieses Problem dadurch zu umgehen, dass der Core mehrere, voneinander unabhängige Aufgaben quasi gleichzeitig bearbeitet. Im Falle eines Betriebssystems sind mit Aufgaben Prozesse gemeint.

Im Prinzip funktioniert dies wie folgt: Sobald ein Prozess nicht mehr weiter bearbeitet werden kann, weil auf die passenden Daten gewartet werden muss, wird einfach die Bearbeitung eines anderen Prozesses fortgesetzt.

Bild 17 zeigt zunächst die Abarbeitung dreier Prozesse auf 3 Cores mit einfacher Pipeline-Architektur und stellt dieser dann die Ausführung auf einem multithreaded Core gegenüber.

Damit hardwareseitiges Multithreading funktioniert, muss ein schnelles Umschalten zwischen den einzelnen Prozessen möglich sein. Dies lässt sich einfach dadurch realisieren, dass jeder Prozess für seinen Kontext einen eigenen Registersatz erhält. Aus dieser Vorgehensweise kann man leicht ableiten, dass die Anzahl der vom Core zur Verfügung gestellten Registersätze hardwareseitig festlegt, wie viele Prozesse parallel bearbeitet werden können. Beim ⟩⟩

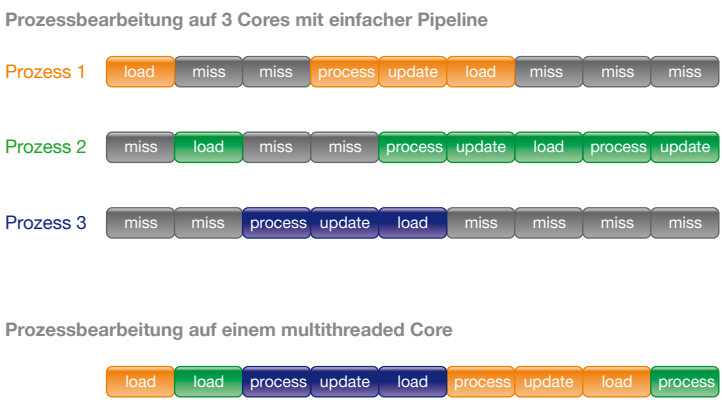


Bild 17: Durch die quasi gleichzeitige Bearbeitung mehrerer Prozesse entstehen keine Wartezeiten mehr in der Pipeline.

MIPS 34K sind dies fünf. Pro Registersatz entsteht so quasi eine „Ausführungseinheit“, ein so genannter *Hardware Thread* (siehe Bild 18).

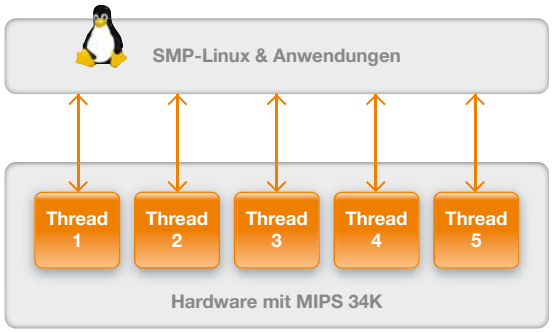


Bild 18: SMP-Linux auf einem multithreaded Core – hier MIPS 34K.

SMP-Betriebssysteme

Auf Hardware-Ebene wird die Parallelisierung von Abläufen durch symmetrische Multi-Core Prozessoren oder multithreaded Cores realisiert. Nun benötigt man noch Software, welche die Parallelisierung organisiert. Für diese Aufgabe wird in der Regel ein Betriebssystem eingesetzt.

Eine Variante, vor allem für Hardware mit identischen Ausführungseinheiten sehr geeignet, sind Betriebssysteme, die ein symmetrisches Multiprocessing – kurz SMP – realisieren. Hauptmerkmal von SMP-Betriebssystemen ist die dynamische Verteilung der Prozesse auf die verfügbaren Cores bzw. *Hardware Threads* zur Programmlaufzeit.

Weitere Merkmale sind:

- Eine Instanz des Betriebssystems betreibt alle Cores bzw. *Hardware Threads*.
- Jeder Anwendungsprozess kann auf jedem Core bzw. *Hardware Thread* laufen. Nur der Kernel ist in der Regel fest einem Core bzw. *Hardware Thread* zugeordnet.
- Alle Cores bzw. *Hardware Threads* können gleichberechtigt Ressourcen anfordern und benutzen (z. B. Speicher, externe Schnittstellen, externe Geräte).
- Das Betriebssystem stellt die Funktionen zur Verteilung der Ressourcen an die Cores bzw. *Hardware Threads* bereit.

Neues Debug-Konzept

Aktuell ermöglicht das TRACE32-Konzept mit einer Instanz des Debuggers genau einen Core zu debuggen (*Core View*). Bisher funktionierte dieses Konzept auch für Multi-Core Prozessoren reibungslos, da diese Prozessoren als asymmetrische Multiprocessing Systeme – abgekürzt AMP – betrieben wurden. Asymmetrisches Multiprocessing bedeutet: Auf jedem Core läuft eine unabhängige Instanz eines Betriebssystems. Für AMP-Systeme ist dadurch immer statisch festgelegt, auf welchem Core welcher Prozess läuft. Damit können auch die Debug-Informationen eindeutig dem entsprechenden Core zugewiesen werden.

Im Gegensatz dazu findet bei SMP-Systemen die Zuordnung der Prozesse auf die Cores bzw. *Hardware Threads* erst zur Laufzeit statt. Dadurch ist es nicht mehr sinnvoll, für das Debugging eines ausgewählten Cores bzw. *Hardware Threads* gezielt eine Instanz des Debuggers zu starten.

System View

Als Alternative zum *Core View*, der für AMP-Systeme hervorragend funktioniert, gibt es nun für SMP-Systeme den *System View*.

Mit dem *System View* wird für das Debugging aller Cores bzw. aller *Hardware Threads* nur eine Instanz des TRACE32-Debuggers gestartet (siehe Bild 19). Den Ausgangspunkt für die Informationsdarstellung bildet auch hier ein Core bzw. ein *Hardware* ⟩⟩

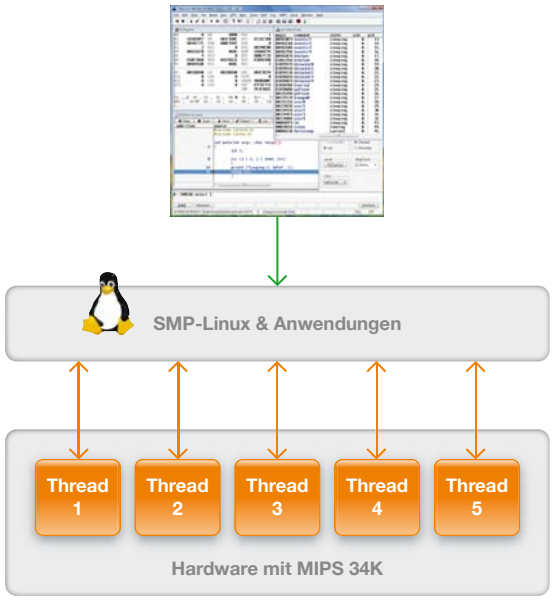


Bild 19: Für das Debugging aller Cores bzw. Hardware Threads wird nur eine Instanz des TRACE32-Debuggers verwendet.

Thread. Neu ist nun aber, dass die Informationsdarstellung per Kommando auf einen anderen Core bzw. *Hardware Thread* umgeschaltet werden kann.

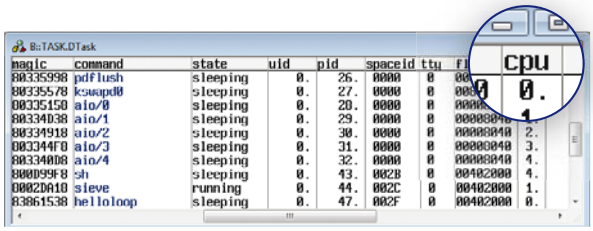


Bild 20: Aus dem TASK.DTASK Fenster lässt sich entnehmen, welchem Core bzw. Hardware Thread SMP-Linux einen Prozess zugewiesen hat.

Das Prozess-Debugging kann nun wie folgt durchgeführt werden:

1. Aus der aktuellen Auflistung aller laufenden Prozesse lässt sich entnehmen, auf welchem Core bzw. *Hardware Thread* ein Prozess läuft (Bild 20). Linux verwendet dabei den Begriff CPU, um vom Core bzw. *Hardware Thread* zu abstrahieren.
2. Durch ein Kommando kann man die Informationsdarstellung im Debugger auf den gewünschten Core bzw. *Hardware Thread* umschalten (Bild 21).

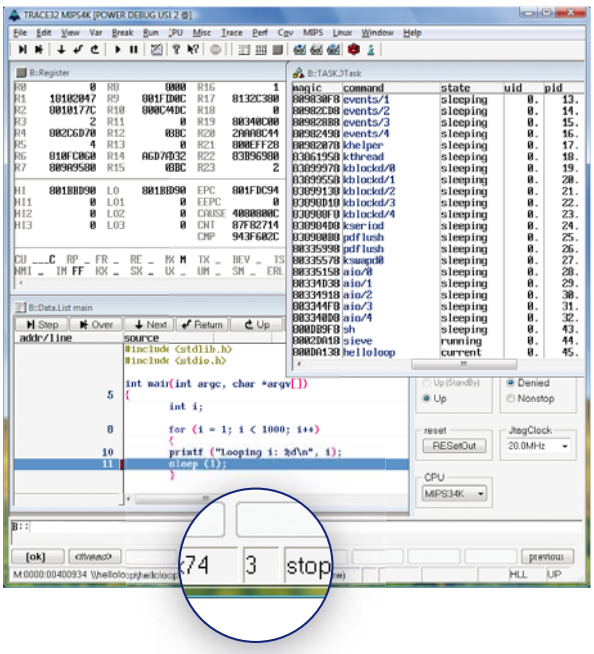


Bild 21: Im TRACE32-Fenster wird immer nur der Kontext eines Cores bzw. Hardware Threads dargestellt. Die Nummer des Cores bzw. Hardware Threads wird in der Statuszeile angezeigt.

Gemeinsame Breakpoints

Die Tatsache, dass ein SMP-Betriebssystem die Anwendungsprozesse erst zur Laufzeit einem Core bzw. *Hardware Thread* zuweist, hat natürlich auch Konsequenzen für das Setzen von *On-Chip Breakpoints*.

Zur Veranschaulichung ein einfaches Beispiel: Der Prozess „sieve“ soll gestoppt werden, sobald er auf die Variable „xyz“ schreibt. Damit der Debugger diese Anforderung umsetzen kann, muss er die Break-Logik aller Cores bzw. *Hardware Threads* für diese Abbruchbedingung programmieren, da ja erst zur Laufzeit festgelegt wird, auf welchem Core bzw. *Hardware Thread* der Anwendungsprozess „sieve“ laufen wird. Das heißt, selbst wenn jeder Core bzw. *Hardware Thread* über eine eigene Break-Logik verfügt, programmiert der Debugger die Breakpoints für den Anwender so, als gäbe es nur eine einzige, allen gemeinsame Break-Logik.

Beim Setzen von *Software Breakpoints*, für deren Realisierung der Originalbefehl im Speicher durch einen Abbruchbefehl temporär überschrieben wird, gibt es keine Änderungen gegenüber den AMP-Systemen. Betriebssysteme schirmen die Adressräume der einzelnen Prozesse ja gegeneinander ab. Wird der gleiche Anwendungsprozess jedoch mehrmals gestartet, lädt z. B. Linux den Programmcode nur einmal. Jede Instanz des Anwendungsprozesses sieht so den gleichen Programmspeicher sowie die dort gesetzten *Software Breakpoints*. Um sicher zu stellen, dass die Programmausführung nur im gewünschten Anwendungsprozess gestoppt wird, ermöglicht der TRACE32-Debugger das Setzen von prozess-spezifischen *Software Breakpoints*.

Zusammenfassung

Durch die gezielte Erweiterung des TRACE32-Konzepts bietet Lauterbach seinen Kunden ein komfortables Debugging von embedded Designs, die ein SMP-Betriebssystem zur Steuerung mehrerer Cores bzw. *Hardware Threads* einsetzen. Um dieses neue Konzept für das Debugging von SMP-Betriebssystemen nutzen zu können, genügt ein TRACE32-Debugger, dessen Debug-Kabel neben der Lizenz für die Prozessorarchitektur eine zweite, z. B. eine Multi-Core-Lizenz enthält.

Nach der Unterstützung für den MIPS 34K ist für das Frühjahr 2008 das Debugging von SMP-Systemen auch für die ARM- und die PowerPC-Architekturen geplant.

Weitere Neuheiten zum Thema RTOS

Windows CE 6.0 für ARM und SH4

TRACE32 unterstützt nun auch Windows Embedded CE 6.0 für die ARM- und die SH4-Architektur.

Seit Ende 2006 können Entwickler die neue Version von *Windows CE* für ihre Projekte einsetzen. In der Kernel-Architektur wurden mit der Version 6.0 einige grundlegende Veränderungen eingeführt. *Windows CE* unterstützt nun beispielsweise Prozess-Adressräume von 2 GByte. Für die Version 5.0 waren diese noch auf 32 MByte beschränkt.

Seit Herbst 2007 steht eine aktualisierte Version der *TRACE32 Windows CE Awareness* zur Verfügung, damit können mit einem TRACE32-Debugger alle Prozesse, Threads und Bibliotheken gleichzeitig überwacht und getestet werden. In Kombination mit einem Echtzeit-Trace-Tool ist zudem eine Laufzeit-Analyse von *Windows CE Threads* möglich. Die Konfiguration der *Windows CE Awareness* wurde vereinfacht:

- Der Debugger erkennt die *Kernel Objects* nun automatisch. Damit muss die Symbolinformation für den Kernel nicht mehr explizit geladen werden.
- Der *Windows CE Autoloader* des Debuggers vereinfacht die Symbolverwaltung für die verschiedenen Prozesse und Bibliotheken. Durch die entsprechende Konfiguration des Autoloaders (*sYmbol.AutoLOAD.CHECKWINCE*) wird die passende Symbolinformation bei Bedarf automatisch in den Debugger nachgeladen.

Integration in den Windows CE Platform Builder

Seit Anfang 2007 können Lauterbach Debugger im Windows CE Platform Builder als Hardware-Debug-Back-End verwendet werden.

Für das Kernel- und Anwendungsprozess-Debugging von *Windows CE* bietet Lauterbach einen eXDI2-Treiber, der es erlaubt, über den internen Debugger des *Windows CE Platform Builders* einen Lauterbach Debugger zu steuern.

Mittels der *Target Device Connectivity* wird zunächst der TRACE32 eXDI2-Treiber in den *Platform Builder* – kurz PB – eingebunden. Durch diese Integration werden nun alle Debug-Kommandos des PB-Debuggers im Hintergrund in TRACE32-Kommandos übersetzt und an den Lauterbach Debugger weitergereicht.

Der eXDI2-Treiber ist für *Windows CE* 5.0 und 6.0 sowie für *Windows Mobile* Version 5 und 6 verfügbar.

Erweiterungen

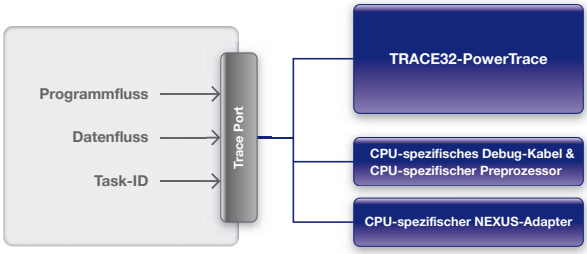
- Linux – Support für *Xenomai Threads*
- Linux – Symbol Autoloader
- OSE – Support für 5.2 Load Modules
- PXROS – Redesign für C167

Neu unterstützte RTOS

DSP/BIOS für TMS320C28xx	verfügbar
eCOS für MIPS	verfügbar
FreeRTOS für ARM und MicroBlaze	verfügbar
HI7000 für SH4	verfügbar
Linux für ARC und MicroBlaze	geplant
LynxOS-SE für PowerPC	geplant
NetBSD für ARM und PowerPC	verfügbar
OS/9 für ARM und PowerPC	verfügbar

RX4000 für MIPS	verfügbar
ThreadX für ARC und Blackfin	verfügbar
ThreadX für Xtensa	geplant
Windows CE 6.0 für ARM und SH4	verfügbar
µClinux für Blackfin	geplant
µClinux für MicroBlaze	verfügbar
µC/OS-II für V850	verfügbar
µC/OS-II für Xtensa	geplant

Neue Preprozessoren/NEXUS-Adapter



Viele SoC (System-On-Chip) verfügen heute über einen Traceport, der dazu dient Laufzeit-Informationen Off-Chip sichtbar zu machen. Typische Informationen sind die vom Programm ausgeführten Befehle sowie die Datentransfers. Beim Einsatz von Betriebssystemen kann eine Ausgabe der aktuellen Task-ID hilfreich sein. Als die bekanntesten Traceports gelten sicher die ETM für die ARM-Architekturen sowie NEXUS für die PowerPC-Architektur.

Für die Aufzeichnung der Laufzeit-Informationen bietet Lauterbach das Echtzeit-Trace-Tool PowerTrace an. Auf Hardware-Ebene arbeitet dieses Tool wie folgt: Ein architektur-spezifischer Preprozessor bzw. NEXUS-Adapter greift die Daten am Traceport ab und überträgt diese dann in den bis zu 4 GByte großen Tracespeicher der universellen Lauterbach-Hardware PowerTrace. Die so aufgezeichneten Informationen dienen dem Nutzer dann als Basis für eine effiziente Fehlersuche sowie für umfassende Laufzeitanalysen seines Designs.

Lauterbach bietet bereits heute Preprozessoren/NEXUS-Adapter für mehr als 30 Prozessorarchitekturen an. Auch 2008 wird diese Unterstützung weiter ausgebaut.

Neuer NEXUS-Adapter für die MPC5500-Architektur

Ab März 2008 wird der NEXUS-Adapter für die MPC5500-Architektur in einer neuen Technologie ausgeliefert. Um flexibel und schnell auf neue Varianten des NEXUS-Ports reagieren zu können, unterstützt dieser Adapter eine Portbreite von bis zu 16-Bit MDO und einen Spannungsbereich von 1.0 bis 5.0 Volt.

Für die Anpassung an den speziellen NEXUS-Port des einzelnen Prozessors bietet Lauterbach die passenden Konverter an.

Um auch bei hohen Traceport-Datenraten eine optimale Abtastung der Tracesignale zu garantieren, enthält der neue NEXUS-Adapter auch FPGAs zur automatischen Einstellung des optimalen Abtastzeitpunkts (AUTOFOCUS-Technologie).

NEXUS-Adapter MPC5500 AF

- NEXUS-Portbreite bis zu 16-Bit MDO, 2-Bit MSEO, EVTI, EVTO
- Datenrate bis 200 MBit/s
- 200 MHz bei SDR bzw. 100 MHz bei DDR
- Spannungsbereich 1.0 V bis 5.0 V
- Adapter für die unterschiedlichen Varianten des NEXUS-Ports
- AUTOFOCUS-Technologie

Neue AUTOFOCUS II Preprozessoren

Damit Preprozessoren auch Traceport-Frequenzen von über 500 MHz weiterverarbeiten können, wurde 2006 die AUTOFOCUS II Technologie entwickelt. Das wichtigste Merkmal dieser Technologie ist die automatische Einstellung des optimalen Abtastzeitpunkts für die Tracedaten. Basierend auf dieser Technik werden nun weitere Preprozessoren für eine Reihe von DSPs angeboten.

Neue AUTOFOCUS II Preprozessoren

Preprozessor für Ceva-X	Q1/2008
Preprozessor für StarCore	Q3/2008
Preprozessor für TMS320C55x und TMS320C64x	Q1/2008

MicroBlaze

Seit November 2007 ist nun auch ein Preprozessor für die MicroBlaze-Architektur lieferbar. Über einen 22-Pin breiten Traceport macht der so genannte *MicroBlaze Trace Core* den Programmfluss sowie wahlweise auch den Datenfluss nach außen sichtbar. Traceport-Frequenzen von über 100 MHz können vom Preprozessor weiterverarbeitet werden.

CombiProbe

Um schnell wichtige Systeminformationen zur Programmlaufzeit auszugeben, verfügen erste Cores mittlerweile über einfache Traceports. Diese erlauben es einem Anwendungsprogramm beliebige Daten Off-Chip sichtbar zu machen. Für die Aufzeichnung und Weiterverarbeitung solcher Daten bietet Lauterbach eine neue Tool-option – die CombiProbe – an. Im Folgenden soll die Hardware sowie die Bedienung dieses neuen Produkts vorgestellt werden.

Instrumentierung

Die meisten Entwickler kennen Testszenarien, für die ein einfaches printf() die effizienteste Realisierung ist. Typische Beispiele sind die Ausgabe von Diagnose-Informationen bzw. das Mitprotokollieren wichtiger Systemereignisse. Gemeint sind also Testszenarien, bei denen die Anwendung bereits rund läuft und nun Betriebstests gefahren werden müssen. Traditionell wird bei solchen Tests mit Code-Instrumentierung gearbeitet.

Wenn wichtige Systeminformation mittels Instrumentierung zur Programmlaufzeit sichtbar gemacht werden soll, sind zwei Schritte notwendig:

1. Zunächst werden in das Anwendungsprogramm Befehle eingefügt, welche die gewünschte Information bereitstellen.
2. Anschließend muss ein Weg gefunden werden, diese Information nach außen sichtbar zu machen. Typischerweise wird dazu eine externe Kommunikationsschnittstelle wie RS-232 oder Ethernet verwendet.

Instrumentierung hat den großen Vorteil, dass genau die Informationen sichtbar gemacht werden, die für den Tester von Interesse sind. Gleichzeitig entsteht aber ein großer Kommunikations-Overhead, da in der Regel über das Betriebssystem eine weitere Kommunikationsschnittstelle bedient werden muss. Bei einer langsamen Schnittstelle kann es zudem zu erheblichen Laufzeitbeeinträchtigungen für die eigentliche Anwendung kommen.

System Trace

Der Kommunikations-Overhead, der bei der Instrumentierung entsteht, lässt sich erheblich reduzieren, wenn der Core einen eigenen Traceport für die Ausgabe der vom Anwendungsprogramm generierten Systeminformation anbietet. Als IP-Provider bietet ARM im Rahmen seiner CoreSight-Technologie eine solche Lösung an.

Die CoreSight *Instrumentation Trace Macrocell* (ITM) arbeitet vereinfacht wie folgt:

1. Das Anwendungsprogramm schreibt die Informationen, die nach außen sichtbar gemacht werden sollen, auf ein 32-Bit *Memory Mapped Register*, das der ITM zugeordnet ist.
2. Die ITM macht die Informationen entweder direkt über den *Serial Wire Output* (siehe Bild 22) oder zusammen mit anderen Tracedaten über die CoreSight *Trace Port Interface Unit (TPIU)* nach außen sichtbar.

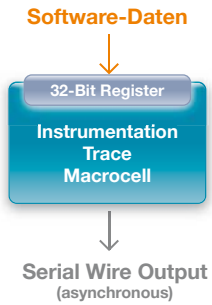


Bild 22: Der CoreSight Instrumentation Trace mit Ausgabe über den Serial Wire Output.

Da bereits viele Mobilfunk-Hersteller mit ähnlichen proprietären Lösungen für ihre Chips arbeiten, jedoch aus Kostengründen an einem einheitlichen Standard für einen solchen Traceport interessiert sind, hat die „Test und Debug“ Arbeitsgruppe der *MIPI Alliance* im Mai 2007 unter dem Namen „System Trace“ einen Standard für ein 4-Bit Traceprotokoll sowie einen 34-Pin Debug- und Tracestecker spezifiziert.

Mit der Standardisierung wurde auch gleich die Funktionalität des Traceports erweitert. Neben der vom Anwendungsprogramm generierten Systeminformation können mittels des *System Trace* nun auch Hardware-Informationen ausgegeben werden (siehe Bild 23). Beispielsweise kann ein programmierbarer *Bus Watcher* bestimmte Buszyklen erkennen und ausgeben. Oder ein Signalmonitor liefert den Status ausgewählter chip-interner Signale.

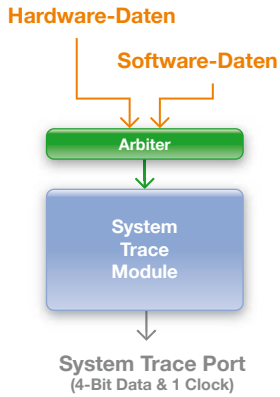


Bild 23: Der von der MIPI Alliance spezifizierte System Trace.

CombiProbe Hardware

Als Mitglied der „Test und Debug“ Arbeitsgruppe konnte Lauterbach aktiv die Spezifikation des *System Trace* mitgestalten und parallel an der ⟩⟩

Entwicklung des passenden Debug- und Trace-Tools – genannt CombiProbe – arbeiten. Seit Oktober 2007 liefert Lauterbach dieses neue Produkt nun in Serienreife.

Bild 24: POWER DEBUG INTERFACE / USB 2 mit einer CombiProbe.

Die CombiProbe (siehe Bild 24) ist eine Kombination aus einem speziellen Debug-Kabel und einem 128 MByte Tracespeicher. Sie kann beispielsweise an die universelle Lauterbach Basis-Hardware POWER DEBUG INTERFACE / USB 2 angesteckt werden. Als Verbindung zum Zielsystem wird der 34-Pin Debug- und Tracestecker aus dem MIPI Standard verwendet, der Anschlüsse für einen JTAG-Port und einen 4-Bit *System Trace Port* umfasst. Selbstverständlich werden auch Adapter zur Anpassung an andere Zielsystemstecker angeboten.

Bedienkonzept

Für das Arbeiten mit der CombiProbe gibt es neben den klassischen Debug-Funktionen einige neue Konfigurationskommandos sowie Kommandos zur Darstellung und Auswertung der aufgezeichneten Traceinformationen:

Zunächst die Konfigurationskommandos:

- *ITM.<subcommand>* zur Konfiguration der *Instrumentation Trace Macrocell* der CoreSight-Technologie (siehe Bild 25).
- *STM.<subcommand>* zur Konfiguration eines *System Trace Modules*.
- *SystemTrace.<subcommand>* zur Konfiguration der CombiProbe.

Für die Analyse und Weiterverarbeitung der aufgezeichneten Tracedaten wird ebenfalls das *SystemTrace*-Kommando verwendet.

Klassisch können die am Traceport sichtbaren Systeminformationen im Tracespeicher der Combi-Probe aufgezeichnet und mit TRACE32-Kommandos ausgewertet werden. Eine intuitive Auswertung der Systeminformationen wird dadurch unterstützt, dass die Trace-Rohdaten mit Hilfe der so genannten *TRACE32 Protocol API* anwendungsspezifisch aufbereitet werden können.

Diese Vorgehensweise hat jedoch den Nachteil, dass die Größe des Tracespeichers in der CombiProbe die Aufzeichnungszeit limitiert. ⟩⟩

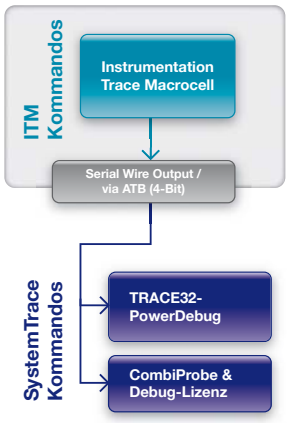


Bild 25: Die ITM-Kommandos erlauben eine Konfiguration der Instrumentation Trace Macrocell. Die SystemTrace-Kommandos ermöglichen eine Darstellung und Analyse der aufgezeichneten Traceinformation.

Kenndaten CombiProbe

- Debug-Kabel und 128 MByte Tracespeicher
- JTAG-Unterstützung für ARM-Cores
  - Standard JTAG
  - Serial Wire Debug Port von ARM
  - cJTAG (IEEE P1149.7), geplant
- Standard-JTAG für andere Prozessorarchitekturen möglich
- Traceport-Unterstützung für
  - MIPI System Trace
  - ITM über Serial Wire Output
  - ITM über 4-Bit CoreSight TPIU
  - 4-Bit ETMv3 im Continuous Mode, geplant
- Bandbreite von 200 MBit/s pro Tracekanal bei bis zu 4 Tracekanälen
- Spannungsbereich 0.3 V bis 3.3 V, 5 V tolerant
- 34-Pin Half-Size Stecker zur Zielhardware
- Adapter auch für 10- und 20-Pin Half-Size Stecker

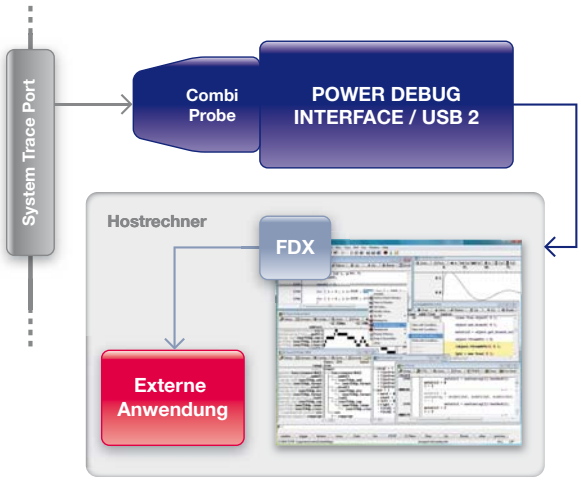


Bild 26: Die CombiProbe lässt sich so konfigurieren, dass eine externe Software die Traceinformation bereits zur Aufzeichnungszeit weiterverarbeiten kann.

Um längere Aufzeichnungszeiten zu realisieren wird für die CombiProbe ein *PIPE Mode* angeboten. Im *PIPE Mode* werden die Tracedaten gleich auf den Hostrechner übertragen. Der Tracespeicher in der CombiProbe dient hier lediglich als Zwischenpuffer.

Hostseitig lässt sich TRACE32 so konfigurieren, dass die Tracedaten direkt in eine Datei abgelegt werden. Die Aufbereitung und Auswertung der Systeminformation kann dann nach Abschluss der Aufzeichnung auch von einer externen Anwendung durchgeführt werden.

Alternativ ist es möglich, eine externe Auswertungs-Software so aufzusetzen, dass diese die Tracedaten über die so genannte *TRACE32 FDX API* abholt und bereits zur Aufzeichnungszeit weiterverarbeitet (siehe Bild 26).

Zusammenfassung

Mit der CombiProbe erweitert Lauterbach seine Produktpalette um eine weitere Tooloption zur Aufzeichnung von Laufzeitinformationen. Für komplexe Testfälle ist es möglich, die CombiProbe auch zusammen mit dem PowerTrace einzusetzen. Da alle Lauterbach Tools über eine gemeinsame Zeitbasis verfügen, können dann wichtige Systeminformationen in einen direkten zeitlichen Zusammenhang mit dem Programm- bzw. Datenfluss des Cores gesetzt werden.

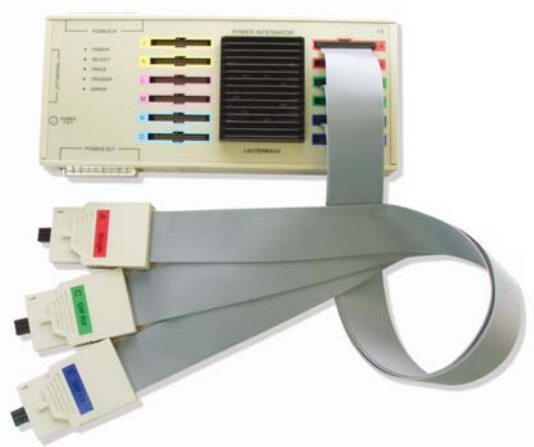
3G/DigRF-Protokoll

Seit Sommer 2007 bietet Lauterbach ein neues Set von Probes für den PowerIntegrator an, das es erlaubt, den Datenverkehr auf einer 3G/DigRF-Schnittstelle aufzuzeichnen. Für die Aufzeichnung werden folgende Abtastraten benutzt:

- 2 GigaSample/s für die RX/TX-Leitungen
- 250 MegaSample/s für die SysClkEnable-Leitung

Abhängig vom Datenverkehr auf den Leitungen ist eine Aufzeichnungsdauer von 25,156 ms bei voller Last bis hin zu mehreren Stunden möglich. Dabei garantiert die hohe Empfindlichkeit der Probes und die sehr geringe Belastung der Eingangssignale eine fehlerfreie Aufzeichnung.

Eine Auswertung und Analyse kann selbstverständlich auf Protokoll-Ebene sowie zeitkorreliert zum Programmfluss durchgeführt werden.



Benachrichtigen Sie uns:

Falls wir Sie aus unserer Mailing-Liste streichen sollen, schicken Sie bitte eine E-Mail an: info@lauterbach.com