Integriertes Run & Stop Mode Debugging für Embedded Linux

Zur Entwicklung von Embedded-Linux-Anwendungen werden heute oft zwei unterschiedliche Debugger verwendet. Für die Inbetriebnahme der Zielhardware wird zunächst ein JTAG-Debugger eingesetzt. Sobald Embedded Linux auf der Zielhardware in Grundzügen läuft, beginnt das Prozess-Debugging mit GDB.

Zur Embedded World 2007 stellt Lauterbach nun einen integrierten Linux-Debugger vor, der beide Debug-Konzepte zusammenführt. Da so die Stärken beider Methoden in einer einheitlichen Bedienoberfläche nutzbar sind, lassen sich die Entwicklungszeiten für Embedded-Linux-Anwendungen erheblich verkürzen.

Im Folgenden werden die Konzepte des integrierten Linux-Debuggers am Beispiel der ARM-Architektur vorgestellt.

Stop Mode Debugging

Ein JTAG-Debugger arbeitet mit dem so genannten *Stop Mode Debugging*: An einem Breakpoint wird der Prozessor und damit das Gesamtsystem gestoppt. Informationen über den Zustand des Prozessors bzw. der Zielhardware können nun über die JTAG-Schnittstelle ausgelesen werden (siehe dazu auch Bild 1).

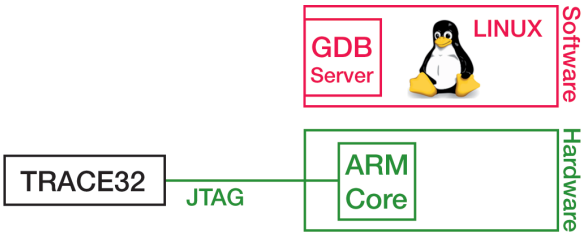

Bild 1: Beim Stop Mode Debugging wird der Prozessor und damit das Gesamtsystem über die JTAG-Schnittstelle angehalten.

Stop Mode Debugging hat unter anderem folgende Vorteile:

- Die einzige Voraussetzung für das *Stop Mode Debugging* ist eine funktionierende JTAG-Schnittstelle. Damit ist Debuggen ab dem Reset-Vektor möglich.
- Mit einem Debugger, der sowohl Linux- als auch MMU-Unterstützung bietet, ist ein Debuggen des Kernels und über Prozessgrenzen hinweg möglich.

Integriertes Run & Stop Mode Debugging

Embedded Linux mit GDB als Debug Agent

ARM
Run Mode Debugging via DCC verfügbar

ARM
Run Mode Debugging via Ethernet
geplant für Q2/2007

PowerPC
Run Mode Debugging via Ethernet
geplant für Q2/2007

Symbian OS mit TRK als Debug Agent

ARM
Run Mode Debugging via DCC verfügbar

- Für den Fall, dass die Software nicht mehr reagiert, kann der Prozessor angehalten werden, um herauszufinden, an welcher Codestelle das Programm hängt.
- Bei angehaltenem Prozessor können keine störenden Effekte auftreten, die durch den Kernel oder einen anderen Prozess verursacht werden.

Stop Mode Debugging hat jedoch einen gravierenden Nachteil:

Sobald der Prozessor gestoppt ist, werden auch alle Kommunikationsschnittstellen angehalten. Dies führt in der Regel dazu, dass externe Geräte, die über Ethernet, Bluetooth bzw. CAN mit der Linux-Anwendung kommunizieren, die Verbindung abbauen, da die Anwendung nicht mehr antwortet. Durch das Stoppen an einem Breakpoint ändert sich also der Zustand des Gesamtsystems. Eine Fortsetzung des Debuggens ist so unter Umständen nicht mehr sinnvoll.

Run Mode Debugging

GDB arbeitet im so genannten *Run Mode Debugging*: An einem Breakpoint wird nur der ausgewählte Prozess angehalten, der Kernel sowie alle

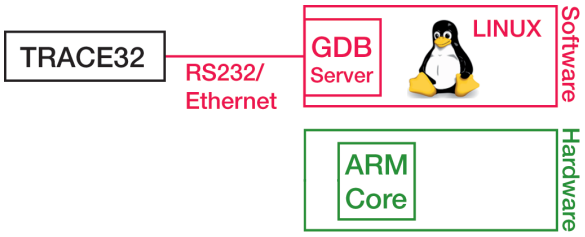

Bild 2: Beim Run Mode Debugging wird der ausgewählte Prozess angehalten, während das Gesamtsystem weiterläuft.

anderen Prozesse laufen einfach weiter.

GDB ist jedoch ein reiner Software-Debugger. Zum Debuggen wird Folgendes benötigt:

- ein *GDB Server* als Linux-Prozess auf der Zielhardware
- eine Debugger-Software – hier TRACE32 – auf dem Host (siehe auch Bild 2)

TRACE32 kommuniziert über eine RS232- oder Ethernet-Schnittstelle mit dem *GDB Server*, um Informationen über den aktuell angehaltenen Prozess abzufragen.

Run Mode Debugging ist immer dann ideal:

- wenn die Inbetriebnahme der Zielhardware abgeschlossen ist.
- wenn der *GDB Server* immer aktiviert werden kann, also die Kommunikationsschnittstelle einwandfrei läuft und der Prozessor nicht fehlerhaft an einer Codestelle festhängt.

Beide Debug-Methoden haben also ganz offensichtlich ihre Stärken und Schwächen. Aus diesem Grund bietet Lauterbach einen Debugger an, der beide Methoden so zusammenführt, dass sich ihre Stärken voll entfalten, ihre Schwächen jedoch verschwinden.

Integriertes Run & Stop Mode Debugging

Der TRACE32-Debugger mit *Integriertem Run & Stop Mode Debugging* für Embedded Linux arbeitet wie folgt (siehe dazu auch Bild 3):

1. Der TRACE32-Debugger wird zunächst über die JTAG-Schnittstelle im *Stop Mode Debugging* gestartet. In einem ersten Schritt müssen nun die Zielhardware und das *Run Mode Debugging* (GDB) konfiguriert werden.
2. Liegt das Augenmerk auf der Inbetriebnahme der Zielhardware, kann weiterhin mit *Stop Mode Debugging* (JTAG) gearbeitet werden.
3. Nach Abschluss der Hardware-Inbetriebnahme lässt sich TRACE32 für das Anwendungs-Debugging auf *Run Mode Debugging* (GDB) umschalten. Einzelne Prozesse können nun getestet werden, während das Gesamtsystem weiterläuft.

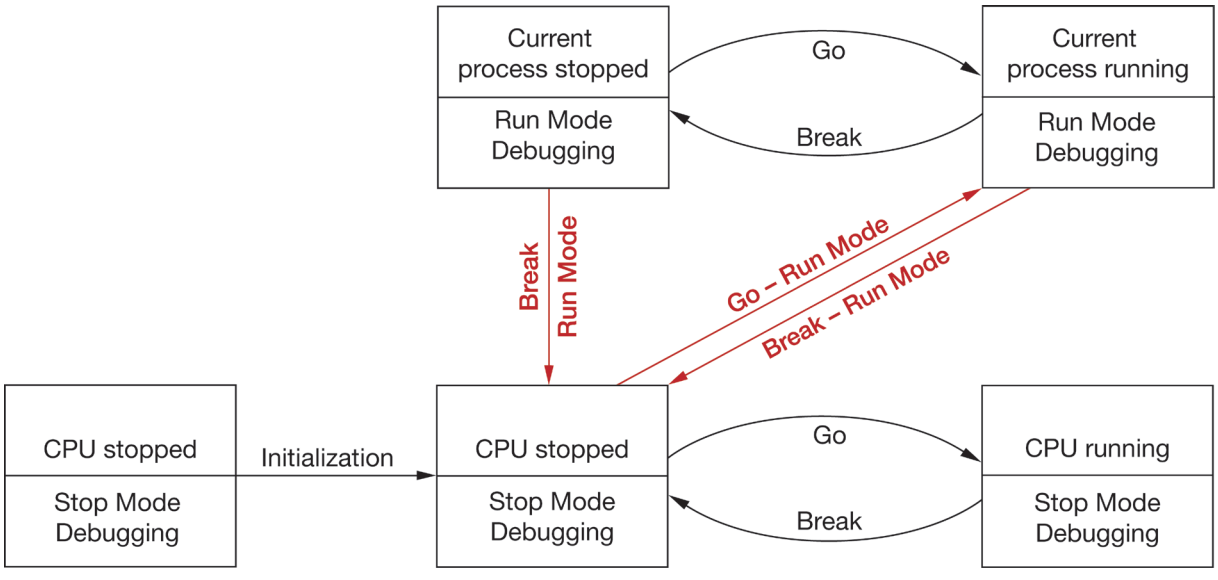

Bild 3: Der Anwender kann je nach Bedarf mit Run Mode oder Stop Mode Debugging testen.

4. Für den Fall, dass beim *Run Mode Debugging* die Verbindung zum *GDB Server* abgebrochen wird, kann jederzeit wieder auf *Stop Mode Debugging* umgeschaltet werden, um die Ursache des Problems zu ermitteln.

Gleichzeitig mit der Implementierung des *Integrierten Run & Stop Mode Debuggings* wurde das *Run Mode Debugging* noch um folgende Funktionen erweitert:

- Für die ARM-Architektur kann neben Ethernet und RS232 auch der *Debug Communications Channel* – DCC – als Kommunikationsschnittstelle benutzt werden. Damit kommt das *Run & Stop Mode Debugging* mit JTAG als alleinige Schnittstelle aus (headless target).
- Bei Bedarf ist gleichzeitiges Debuggen mehrerer Prozesse möglich.

DCC als Kommunikationsschnittstelle

Die JTAG-Schnittstelle für die ARM-Architektur beinhaltet einen so genannten *Debug Communications Channel* – kurz DCC. Prinzipiell soll über DCC ein Informationsaustausch zwischen

- einer Debugger Software auf dem Host (TRACE32) und
- einer beliebigen Anwendung auf dem Zielsystem – hier dem *GDB Server*

möglich sein, während die Anwendung auf dem Prozessor läuft. Verwendet TRACE32 also die DCC-Funktion der JTAG-Schnittstelle, um vom *GDB Server* Informationen über den aktuell angehaltenen Prozess abzufragen, wird für das *Run Mode Debugging* keine externe Kommunikationsschnittstelle mehr benötigt (siehe dazu auch Bild 4).

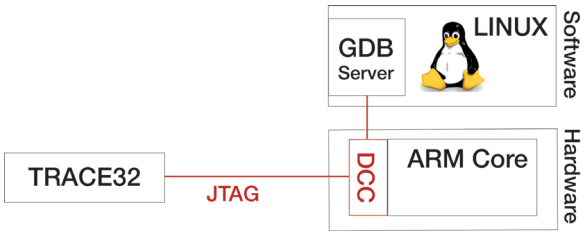

Bild 4: Statt einer externen Kommunikationsschnittstelle kann die DCC-Funktion der JTAG-Schnittstelle als Kommunikationskanal zum GDB Server verwendet werden.

Gleichzeitiges Debuggen mehrerer Prozesse

In manchen Fällen ist es notwendig, mehrere Prozesse gleichzeitig zu debuggen. Um dieses Feature anzubieten, stellt Lauterbach den *T32Server* für das *Run Mode Debugging* zur Verfügung.

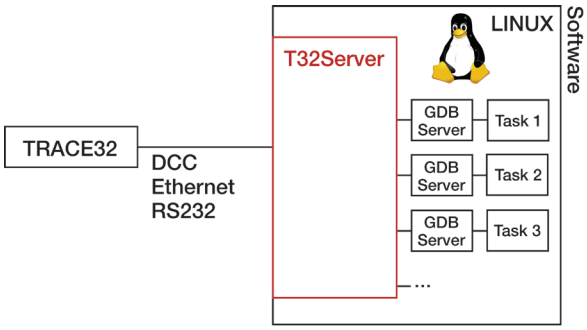

Bild 5: Mit Hilfe des T32Servers können mehreren Prozessen eigene GDB Server zugeordnet werden. Dadurch lassen sich mehrere Prozesse gleichzeitig debuggen.

Nachdem der *T32Server* einmal als Linux-Prozess über das Terminal-Window gestartet wurde, ist Folgendes direkt über TRACE32-Kommandos möglich:

- das Starten von Prozessen (TASK.RUN)
- das Verbinden (Attach) zu laufenden Prozessen (TASK.SELect)
- das Beenden von Prozessen (TASK.KILL)

Beim Starten/Verbinden eines Prozesses wird jedem Prozess vom *T32Server* ein eigener *GDB Server* zugewiesen (siehe dazu auch Bild 5).

Bild 6 auf der nächsten Seite zeigt das TRACE32 *Run Mode Debugging* am Beispiel eines *TASK.List*-Fensters. Neben den neuen Kommandos zum Starten, Verbinden und Beenden von Prozessen kann das Debuggen ansonsten wie gewohnt mit der TRACE32-GUI durchgeführt werden.

Zusammenfassung

Das *Integrierte Run & Stop Mode Debugging* bietet die optimale Basis für eine effiziente Entwicklung von Embedded-Linux-Anwendungen, da mit einem einzigen Entwicklungswerkzeug und einer durchgängigen Bedienoberfläche sowohl komplexe Hardware- als auch Software-Fehler schnell gefunden

werden können. Dabei muss weder die Anwendung noch Linux selbst modifiziert werden.

Integriertes Run & Stop Mode Debugging wird für die ARM-Architektur seit November 2006 unterstützt und ist ohne Aufpreis mit jedem TRACE32 JTAG-Debugger für ARM-Prozessoren nutzbar.

Eine Implementierung für die PowerPC-Architekturen ist bis Mai 2007 geplant.

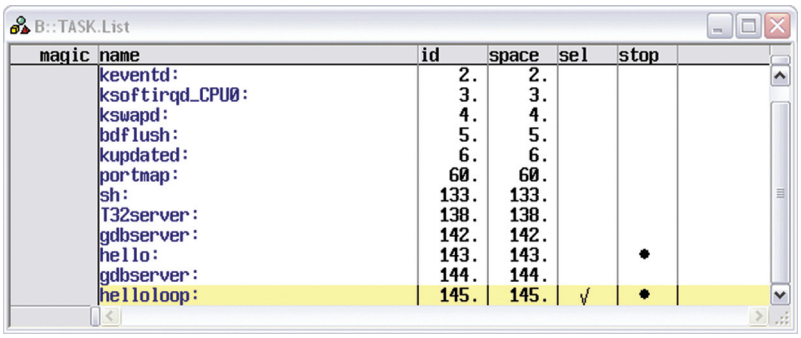

Bild 6: Die Prozesse „hello“ und „helloloop“ sind angehalten. Der Prozess „helloloop“ ist der aktuell für das Debuggen ausgewählte Prozess.