Mit asymmetrischem Multiprocessing:

Schnellstart für Linux-Systeme

Beim Starten von Linux-Systemen lässt die „Kern-Reaktion“ mitunter lange auf sich warten. Bis der komplette Kernel oder sogar die Applikation geladen ist, vergeht viel Zeit. Mit einer „Early Action Application“, der ein eigener Prozessorkern zugewiesen wird, kann das System schon kurz nach dem Einschalten auf wichtige Ereignisse reagieren.

Von Thomas Brinker

Mehrkern-Prozessoren sind heute etablierter Stand der Technik und zusammen mit Linux als Betriebssystem in zahlreichen Projekten aller Branchen zu finden. Typischerweise sind es 2, 4 oder 8 Kerne, die die Leistungsfähigkeit gegenüber nur einem einzigen Kern erheblich erhöhen. Voraussetzung ist jedoch, dass dem Betriebssystem mehrere Threads oder Prozesse zur Verfügung stehen, die es auch gleichzeitig auf die Kerne verteilen kann. Ist die Algorithmik single-threaded und damit nur auf einem Kern ausführbar, so verpufft die Leistung der anderen Kerne wirkungslos. Der Einsatz moderner Frameworks und verteilter Architekturen, sogenannter Microservices, mit verschiedenen Client-/Server-Applikationen kommt dem Einsatz mehrerer Kerne dabei sehr entgegen.

Der Artikel [1] zeigt, wie man einzelne Kerne aus dem Verbund einer Mehrkern-CPU herauslösen kann und für Echtzeit-Anwendungen nutzbar macht, indem ein CPU-Kern exklusiv außerhalb des Betriebssystems hoch deterministisch arbeitet, während die verbleibenden Kerne vom Betriebssystem verwaltet werden. Wird diese Architektur nun unmittelbar nach dem Power-up in Betrieb gesetzt, so können weitere interessante Anwendungsfälle abgedeckt werden, wie etwa das sehr schnelle Booten.

Es ist möglich, auf dem abgetrennten Kern sehr schnell anwendungsspezifische Aufgaben durchzuführen, ohne auf die vollständige Initialisierung eines Linux-Systems zu warten. **Bild 1** zeigt den zeitlichen Ablauf. Der Bootloader startet unmittelbar nach Anlaufen der CPU eine Early-Action-Anwendung

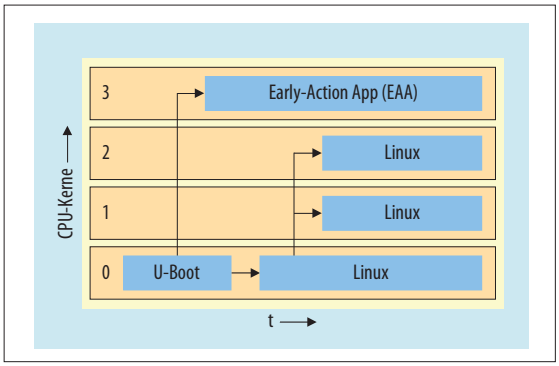

Bild 1. Zeitlicher Ablauf eines Boot mit abgetrennt betriebenem CPU-Kern.

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

Listing 1. Übersetzung eines Bare-Metal-Programms, das für Early-Action-Anwendungen genutzt werden kann.

```
U-Boot > icache off; dcache off; fatload mmc 0 0x10000000 app; cpu 3 release 0x10000000
```

Listing 2. U-Boot-Befehl zum Aufruf einer Early-Action-Applikation.

(EAA) auf der abgetrennt betriebenen CPU. Diese EAA läuft direkt auf der CPU ohne Nutzung eines Betriebssystems oder umfangreicher Libraries, also Bare Metal. Dieser unmittelbare Betrieb erlaubt eine minimale Initialisierungsphase, sodass die Funktionen dieser Anwendung innerhalb weniger Millisekunden bereitstehen und ihre Aufgaben übernehmen können.

Währenddessen wird von dem Bootloader das eigentliche Linux-Betriebssystem gestartet. Es initialisiert die verbleibenden Prozessor-Kerne, alle erforderlichen Schnittstellen und weitere Software-Infrastrukturen. Die zwei getrennten Systeme können nach dem Boot über das bewährte Interface „rpmsg“ und damit über Shared Memory und Inter-Processor Interrupts (IPI) Informationen austauschen.

Übliche Entwicklungswerkzeuge reichen aus

Durch die Symmetrie der Prozessorkerne ist keine spezielle Toolchain für die Compilierung der EAA erforderlich. **Listing 1** zeigt den Quelltext eines Bare-Metal-Programms, welches für Early-Action-Anwendungen genutzt werden kann. Im GCC wird ein solches Programm als Free-Standing bezeichnet, daher der etwas exotischere Aufrufparameter „-ffreestanding“.

Einige Schritte der sehr zeitsparsamen Initialisierung lassen sich nicht in C ausdrücken und müssen in wenigen Assembler-Dateien ausgelagert werden. Speicheradressen und -größen müssen im Linker-Script, in der MMU-Initialisierung und im Linux-Kernel korrespondieren. Fehler an diesen Stellen bedeuten oftmals überlappende Speicher, welche erst im Boot-Vorgang des Linux-Systems auffallen – typischerweise in Form von unvorhersehbarem Verhalten, zufälligen Kernel-Panics oder Einfrieren des Systems ohne jede weitere Meldung. Das Debugging solcher Situationen gestaltet sich entsprechend schwierig. Die verringerten Debug-Funktionen während des Boot-Vorgangs und der eingesetzten Bootloader erschweren das Auflösen solcher Fehlerzustände. Entsprechende Vorsicht und gute theoretische Vorplanung der Speicherzuordnung bilden eine gute Grundlage für die fehlerfreie und zügige Inbetriebnahme.

Boot-Vorgang: Getrennte Bereiche einrichten

Das Starten des Systems erfolgt ganz gewöhnlich per U-Boot-Bootloader, welcher von einem persistenten Speicher (z.B. eMMC oder NAND-Flash) geladen wird. Mit Hilfe des in **Listing 2** gezeigten Codes wird eine Early-Action-Applikation (EAA) von einer eMMC geladen und auf Core 3 gestartet. Die U-Boot-Kommandozeile kehrt nach diesem Aufruf zurück und erwartet weitere Befehle. Die EAA, welche in der Datei „app“ gespeichert ist, läuft zeitgleich an und bleibt aktiv.

Der U-Boot-Bootloader kann nun auf konventionellem Weg einen Linux-Kernel starten. Zuvor wird jedoch das Device Tree Binary (dtb) geladen, welches die durch Linux zu nutzende Hardware beschreibt. Hierin müssen Schnittstellen, Speicher und CPU-Kern der EAA ausgespart werden, damit es nicht zu einem Ressourcenkonflikt kommt. Eine besondere Herausforderung stellt dabei die korrekte Verwaltung der Taktsignale dar. Da diese aus einer einzigen PLL-Quelle erzeugt werden, ergeben sich hier automatisch Konflikte, die in beiden Welten genauestens aufzulösen sind.

Während das Linux-System sich selbst und alle erforderlichen Schnittstellen initialisiert, kann die EAA bereits umfangreich mit der Außenwelt kommunizieren. Es sind mehrere Szenarien dabei denkbar.

Datenpuffer

Alle Daten, die über die Schnittstellen der EAA kommen, werden in einen eigenen Puffer gespeichert und können später von Linux-Applikationen zur Bearbeitung abgeholt werden. Ist der Pufferspeicher hinreichend groß dimensioniert, so können alle empfangenen Nachrichten später bearbeitet werden.

Kommunikation

Wird die EAA erweitert, um bestimmte Aufgaben selbstständig zu übernehmen und umfangreich mit der Außenwelt zu kommunizieren, so können komplette Kommunikationsprotokolle oder auch nur zeitkritische Untermengen davon implementiert werden. Die volle Kommunikation wird dann später von der Linux-Applikation übernommen.

Datenpuffer + Kommunikation

Werden Datenpufferung und eigenständige Kommunikation der EAA kombiniert, entsteht eine weitere Variante, in der je nach Funktionsweise der Kommunikationsschnittstellen Linux-Applikation und die EAA zwei getrennte Teilnehmer auf einem Bus sein können. Das wird so zum Beispiel in der im Fol-

Anwendung: grafische Animation

In der Early-Action-Anwendung (EAA) können auch grafische Funktionen implementiert werden, die auf einem Display dargestellt werden. Dazu sind lediglich die Image Processing Unit (IPU) oder äquivalente Subsysteme zu initialisieren. Der Pixelpuffer wird dann im Shared Memory zwischen Linux und EAA platziert. Ein Hand-over zwischen Linux-Treibern und EAA mittels Interprozessor-Interrupt (IPI) stellt die kontinuierliche und unterbrechungsfreie Nutzung der IPU und des Pixelpuffers sicher. Mittels Mehrfach-Pufferung im Pixelpuffer können nun kleine Animations-Slideshows in der EAA implementiert werden.

genden beschriebenen CAN-Bus-Anwendung implementiert.

Asynchrones Ausgeben/animiertes Boot-Logo

Eine weitere mögliche Nutzung besteht in der direkten Ausgabe von Daten durch die EAA ohne weitere Interaktion mit anderen Teilnehmern. So können auf bestimmte Schnittstellen beispielsweise Wartepakete gesendet werden, mit denen die baldige Bereitschaft angekündigt wird. In einer Variante hiervon können auf einem Display zum Beispiel einfache Animationen dargestellt werden, die bis zur Umschaltung auf die Linux-Treiber stabil sichtbar sind. Gerade bei der Implementierung eines animierten Boot-Logos sind viele verschiedene Ausbaustufen mit mehr oder weniger Aufwand denkbar. Einfache Animationen, die effektiv aus einer Slideshow bestehen, sind noch am einfachsten ohne Betriebssystem umzusetzen. Dekodierungen von Video-Streams oder gar 3D-Animationen auf OpenGL-Basis stellen das Maximum dar, was in einigen Fällen auch wirtschaftlich umsetzbar sein sollte.

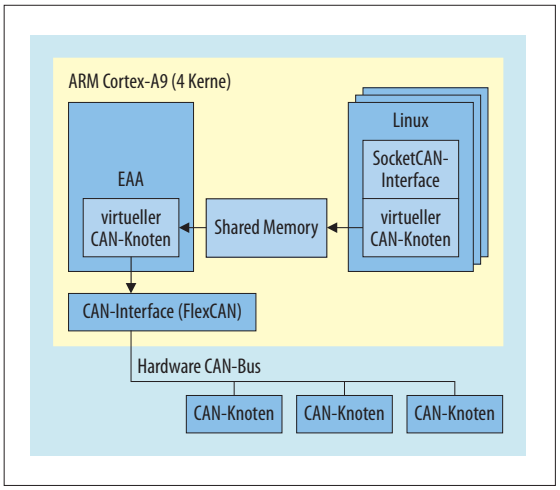

Bild 2. Hinter der CAN-Hardware des SoC verbergen sich zwei virtuelle Devices: ein früh verfügbares in der EAA und ein in Linux integriertes.

Legacy Controller Software

Etwas aus dem bisherigen Rahmen fällt die Nutzung einer bereits existierenden (Legacy-)Software als EAA, welche ursprünglich für den Betrieb auf einem Controller entwickelt wurde. Bei geeigneter Hardware-Abstraktion können solche Programme als EAA auf einen ARM Cortex-A9 portiert werden. Dort sind diese dann wieder in der gewohnten Umgebung verfügbar. Neue, auf dem Legacy-System ursprünglich nicht vorhandene Schnittstellen wie z.B. USB, WiFi oder Display werden dann später unter Linux verfügbar.

Erfahrung mit Kundenprojekt

Die dritte Variante (Datenpuffer + Kommunikation) mit eigenem Handling und gleichzeitigem Datenpuffer wurde bei emlix in einem Kundenprojekt mit CAN-Bus umgesetzt (**Bild 2**). Der physisch am i.MX6 angeschlossene CAN-Bus wird um zwei logische Devices erweitert – eines in einer Linux-Applikation und eines in der EAA. Zwischen Linux und EAA werden die CAN-Pakete über Shared Memory transportiert. Alle Pakete, die die Linux-Applikation per Linux-SocketCAN-API erhält oder sendet, werden über den gemeinsamen Speicher an die EAA gegeben. Diese behandelt das Paket wie ein normales CAN-Paket und reagiert ggfs. mit dem Versand eines Antwortpaketes. In jedem Fall aber wird das Paket auf den CAN-Bus gesendet. In umgekehrter Richtung werden alle Pakete, die per CAN-Bus das SoC erreichen, sowohl von der EAA empfangen und bearbeitet als auch von der Linux-Applikation.

Die CAN-Pakete werden von der EAA in einen Puffer geschrieben, bis diese von der Linux-Seite abgeholt werden. Die Größe ist so bemessen, dass alle relevanten Nachrichten, welche das System während des Linux-Boot erreichen, gespeichert werden können.

Die EAA beantwortet bestimmte CAN-Pakete aus dem Netzwerk mit sogenannten „NOT-YET"-Paketen bereits wenige Millisekunden nach Power-on. Dieses war in der historisch gewachsenen Netzwerkarchitektur der Controller so spezifiziert. Jeder Teilnehmer im CAN-Netzwerk hatte sich innerhalb von weniger als einer Sekunde nach dem Power-on zu melden. Unterbleibt dieses, so werden Kommandos von dieser Komponente später nicht mehr von anderen Teilnehmern akzeptiert. Die direkte Optimierung eines Linux-Systems hin zur spezifizierten Reaktionszeit hätte Modifikationen nach sich gezogen, die einer nachhaltigen, wirtschaftlichen Wartung im Wege stünden.

Wann ist der Boot-Vorgang beendet?

Zur Messung und zum Vergleich von Boot-Zeiten bedarf es einer kurzen Definition verschiedener Zeitpunkte und damit Milestones, welche während des Hochlaufens erreicht werden müssen.

- ➜ Power On: Die Hauptversorgungsspannung wird an das Board angelegt (typischerweise 5...12 V).
- ➜ Power Good: Die Spannungswandler auf dem Board sind eingeschwungen oder das Power Sequencing ist abgeschlossen. Die CPU kann nun anlau-

Reintegration der CPU nach Boot

Die Reintegration eines nicht mehr benötigten Prozessorkerns nach dem Boot-Vorgang könnte eine mögliche Erweiterung des Konzeptes darstellen. Nachdem das System seinen vollen Betriebszustand erreicht hat, ist die EAA eigentlich nicht mehr erforderlich. Es wäre wünschenswert, den Kern und alle damit verbundenen Speicherabschnitte in das Linux-System zu integrieren, sodass auch diese Komponenten zum Betrieb genutzt werden können. Dies wäre insbesondere bei Zwei-Kern-Prozessoren von Interesse, da diese sonst nur mit einem einzigen Kern betrieben werden können. Der Anpassungsaufwand hierfür sollte im Linux-Kernel mit seinen Speicher- und Prozessor-Hotplug-Funktionen überschaubar sein, aber auf hohem Niveau liegen.

Boot-Zeit-Optimierung im Kernel

Zur direkten Optimierung der Boot-Zeit eines Linux-Systems müssen viele Konfigurationsparameter des Bootloader, des Kernel und des Init-Systems optimiert werden. Idealerweise arbeitet man sich bei der Optimierung von der größten Verzögerung zu den kleineren vor. Dabei fallen Modifikationen am Linux-System an, die auch in einer späteren Wartungsphase mit gepflegt werden müssen und gegebenenfalls in neue Versionen von Kernel und Co. einzupflegen sind. Im dargestellten Ansatz bleibt das Linux-System nahezu unverändert und Updates sind erheblich vereinfacht.

fen. Dieser Zustand kann von außen per Messung an POR oder vergleichbaren Pins gemessen werden.

- Linux-Shell: Linux ist gebootet und Befehle auf der Shell können per serieller Schnittstelle eingegeben werden. Dieser Zustand ist bei jedem Linux-System erreichbar und erlaubt damit einen allgemeinen Vergleich.
- Bereit (Linux&Applikation): Die nachfolgenden Schritte der Initialisierung eines Linux-Systems, wie etwa automatisches Starten einer oder mehrere Applikationen, sind zwar für die Gerätefunktion obligatorisch, können aber höchst individuell und damit nicht allgemein vergleichbar ausfallen. So können sehr kleine Headless-Kommunikationsknoten in nur wenigen Sekunden starten, während aufwendige Grafikapplikationen mit vielen Threads durchaus bis zu Minuten benötigen können.
- Bereit (EAA): In diesem Zustand kann die EAA mit ihren beschränkten Ressourcen und Schnittstellen gerätespezifische Funktionen übernehmen.

Die Zeit von Power On bis Power Good kann von der Software nicht beeinflusst werden. Hier sind die Hardware-Designer gefragt, um die Spannungen etc. schnellstmöglich bereitzustellen. Auf marktgängigen Systems-on-Module (SOMs) wurden Zeiten bis 800 ms zwischen Power On und Power Good gemessen.

Ab Power Good beginnt nun die Software in ROM- und Flash-Bausteinen zu arbeiten. Zu einem möglichst frühen Zeitpunkt wird die Initialisierung in zwei logische Threads aufgeteilt, siehe **Bild 3**: den langsameren Thread mit mehr Schnittstellen, der in ein vollwertiges Linux-System mündet, und den schnellen, der ein kleines leichtgewichtiges Bare-Metal-System umfasst, das EAA. Der Zustand *Bereit* des EAA wird 25 ms nach Power Good auf einem i.MX6 mit vier Kernen erreicht. Der Zustand *Linux Shell* wird nach etwa 7 Sekunden erreicht. Danach würde nun die Initialisierung der spezifischen Linux-Applikationen beginnen. Die zeitliche Dauer dieser Initialisierung ist hochgradig von der Architektur der Software abhängig. Je mehr Skripte, Prozesse und Threads durchlaufen und gestartet werden müssen, desto länger wird dieser Prozess dauern. In den meisten Fällen müssen auch Konfigurationsdateien eingelesen und Hardware-Komponenten geprüft und erkannt werden.

In modernen Cloud-Anwendungen müssen sogar erst Verbindungen zum Internet per WiFi oder Mobilfunk aufgebaut werden, bevor ein Embedded-Gerät voll funktionstüchtig ist. Der Startvorgang eines solchen Embedded-Systems kann also bis an die Minuten-Grenze und darüber hinaus dauern. Erste Aktivitäten können aber schon nach 25 ms durch die EAA erfolgen.

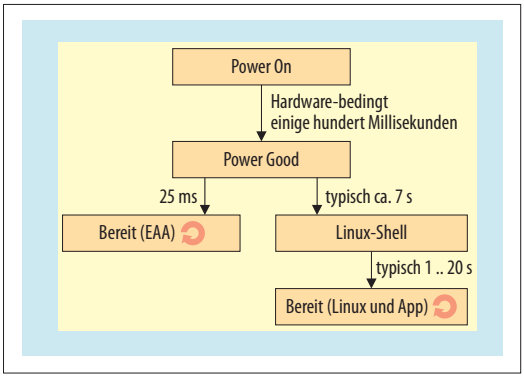

Bild 3. Sobald der Zustand *Power Good* erreicht ist, werden zwei unterschiedlich schnelle Initialisierungspfade beschritten.

Mit oder ohne Betriebssystem? – Sowohl als auch!

Die Boot-Zeit ist wohl eines der wichtigsten Argumente, welche gegen ein Betriebssystem auf Embedded-Geräten ins Feld geführt wird. Gerade bei der Umstellung eines existierenden Gerätes von Controller-Programmierung in Bare Metal auf eine Linux-Architektur wird die deutlich längere Boot-Zeit zuweilen negativ wahrgenommen. War diese vorher bei nahe Null, beträgt sie nun einige Sekunden.

Die beschriebene asymmetrische Multiprocessing-Architektur, die hier zur Boot-Beschleunigung genutzt wird, erlaubt, Teile der Funktion in klassischer Weise unabhängig vom Betriebssystem und damit sehr schnell bereitzustellen. Parallel dazu werden weitere Schnittstellen in Linux initialisiert. Mit den dortigen leistungsfähigen Software Stacks für USB, Ethernet, WiFi etc. werden diese zu einem späteren Zeitpunkt vollumfänglich nutzbar. Tiefgreifende Modifikationen des Linux-Systems zur Boot-Optimierung können entfallen und erlauben somit eine wirtschaftliche Wartung des Systems, da Modifikationen von Original-Komponenten aus der Open Source Community zum Zwecke der Boot-Zeit-Optimierung entfallen.

Der Entwicklungsaufwand und die -risiken sowie die daraus resultierenden Wartungsvorgänge der Software müssen bei jedem Projekt individuell gegen die technischen Möglichkeiten abgewogen werden. *jk*

Literatur

[1] ***Brinker, T.*: Echtzeit mit eigenem CPU-Kern. *Elektronik* Reader's Choice, S. 34, August 2016.**

Dipl.-Ing. Thomas Brinker
ist Senior System Engineer und Project Manager der emlix GmbH. Er beschäftigt sich mit Architektur und Design von Embedded-Linux-basierten Systemen in der Medizintechnik, im Konsumgüter- und im industriellen Bereich sowie mit deren Netzwerkintegration.
thomas.brinker@emlix.com