Individuelle Linux-Systeme mit Open-Source-Software

IT-Klinger — Andreas Klinger
ak@it-klinger.de

11. März 2012

Ein erfolgreicher Projektabschluß setzt eine zielführende Planung und Durchführung voraus. Dies gilt besonders bei Open-Source-Projekten. Hier hat der Verantwortliche eine unüberschaubar große Auswahl an Software-Paketen und damit auch an möglichen Optionen zur Verfügung. In diesem Artikel wird eine Konzeptionshilfe für die Erstellung eines Embedded-Linux-Systems gegeben. Der Focus liegt darauf aufzuzeigen, welche Komponenten benötigt werden und wie diese zielführend eingesetzt werden können. Damit bietet sich ein „Roter Faden“ durch das Embedded-Linux-Projekt, der an individuelle Bedürfnisse angepasst werden kann.

1 Ist Embedded-Linux ein anderes Linux?

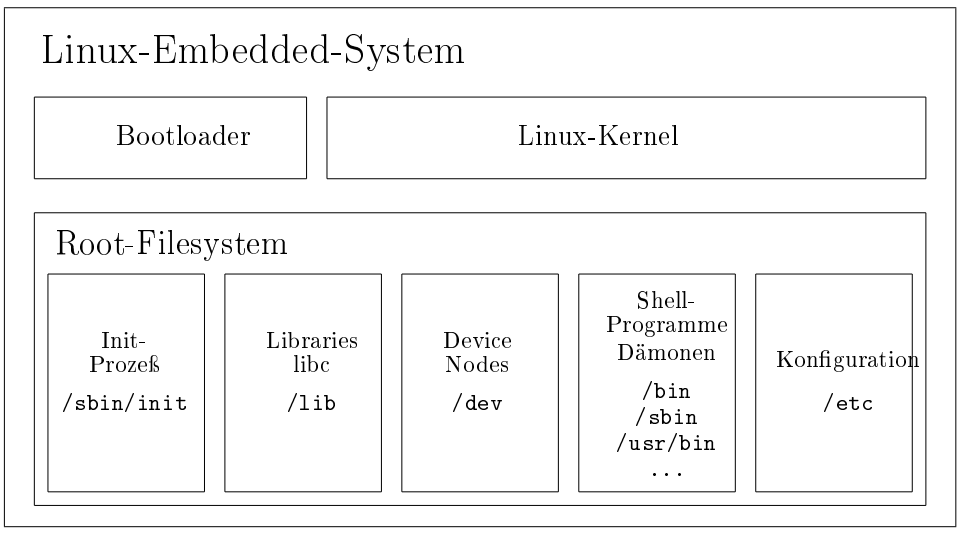

Abbildung 1: Embedded-Linux-System

Beide haben Vieles gemeinsam, insbesondere die gleichen Kernel-Quellen, das Rootfilesystem und die notwendigen Dämonen. Allerdings bestehen bei Embedded-Systemen spezielle Limitierungen, insbesondere hinsichtlich der Hardware-Ausstattung. Es sind

Grenzen gesetzt bei der Rechenleistung, der Speicherausstattung und der verfügbaren Peripherie. Dies erfordert, das Betriebssystem darauf zu optimieren, die verfügbare Hardware bestmöglich zu nutzen. Bei Embedded-Systemen werden höhere Anforderungen hinsichtlich Zuverlässigkeit bis hin zu Echtzeitanforderungen gestellt. Auch im autonomen Betrieb wird ein stabiles Systemverhalten vorausgesetzt.

2 Komponenten eines Embedded-Linux-Systems

2.1 Cross-Development-Toolchain

Die Cross-Development-Toolchain wird auf dem Hostsystem ausgeführt und erstellt für das Zielsystem alle benötigten Komponenten, wie Bootloader, Kernel und das Root-Filesystem. Selbst wenn die Architekturen vom Host- und Target-System übereinstimmen empfiehlt es sich, eine eigene Cross-Development-Toolchain zurechtzulegen. Dies hat den Vorteil, dass der Erstellvorgang immer gleich abläuft, unabhängig von zusätzlich installierten Programmen auf dem Hostrechner. So kommt es vor, dass bei der Installation von Anwendungen einzelne Bibliotheken oder Include-Dateien ausgetauscht werden und diese dann zu einem anderen Build-Prozess für das Target führen. Diesem Problem kann man aus dem Weg gehen, indem man immer, unabhängig von der zu erstellenden Architektur eine eigene Buildumgebung benutzt.

Bei der verwendeten Version der Entwicklungsumgebung ist insbesondere darauf zu achten, dass die von den zu verwendenden Komponenten geforderten Versionen eingehalten werden. Sowohl Bootloader, als auch der Kernel sind in dieser Hinsicht teilweise auf spezielle Versionen angewiesen. Ansonsten kann es vorkommen, dass ein nicht funktionierendes System entsteht.

In allen Fällen sollte soweit möglich auf bereits vorhandene Toolchains zurückgegriffen werden. Die Erstellung einer individuellen Entwicklungsumgebung aus den GNU-Sourcen bedeutet unter Umständen einen sehr grossen Aufwand.

2.2 Bootloader

Der Bootloader hat die Aufgabe, Hardware zu initialisieren und boardspezifische Einstellungen vorzunehmen, wie beispielsweise die Belegung der CPU-Pins. Er lädt eine RAM-Disk in den Hauptspeicher, legt Kernel-Parameter ab, lädt den Kernel und startet ihn. Für diese Aufgaben haben sich zwei Bootloader etabliert: grub und u-boot.

Der grub wird vorzugsweise im Intel-Umfeld eingesetzt. Dort ist er einfach zu konfigurieren, passt sich den unterschiedlichen Massenspeichern wie etwa CF-Card oder USB-Speicher hervorragend an und kann mit einer ganzen Reihe von Dateisystemen arbeiten. Er ist in der Lage seine eigene Konfiguration, sowie die zu ladenden Images (Kernel, RAM-Disk, ...) direkt aus dem Filesystem zu verwenden.

Mit dem u-boot hat man gute Karten bei nicht Intel-basierenden Mikrocontrollern. Er eignet sich hervorragend, um unterschiedlichste Boards zu initialisieren. Dazu wird der Bootloader um die entsprechende Funktionalität erweitert. Das erfordert zwar etwas Programmierung, funktioniert aber anhand der vielen bereits integrierten Boards

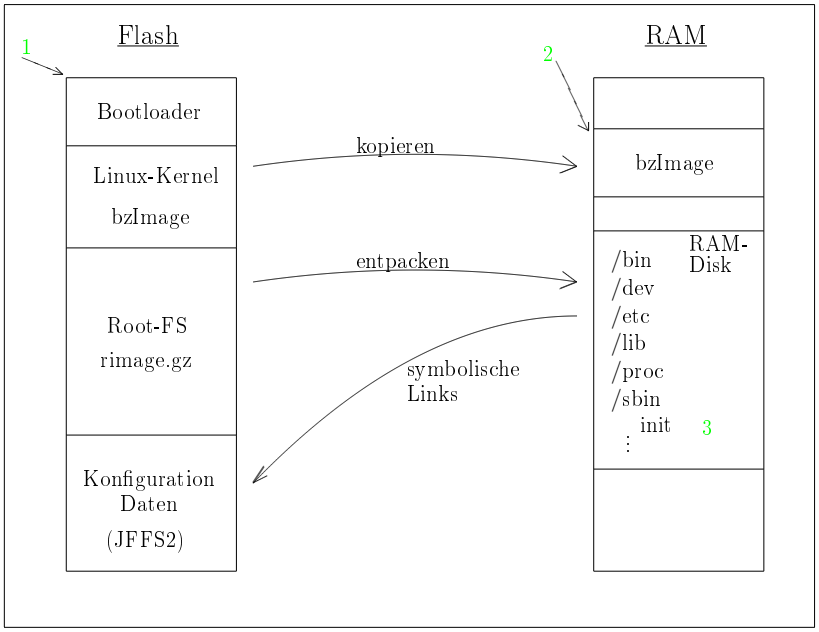

Abbildung 2: Bootvorgang eines Embedded-Linux-Systems

anhand von Beispielen sehr gut.

Dabei arbeitet der u-boot mit einem eigenem Image, welches den Kernel und das Root-Filesystem enthält. Dieses Image kann vom Bootloader genutzt werden, um das Flash zu beschreiben. Das ist mit ein paar wenigen Änderungen auch im Feld möglich, wenn das System bereits produktiv läuft. Nachträgliche Software-Updates sind damit realisierbar.

2.3 Linux-Kernel

Beim Linux-Kernel empfiehlt sich auf jeden Fall einen Mainstream-Kernel einzusetzen, um auch nachträglich in den Genuss von Verbesserungen und Weiterentwicklungen zu kommen. Gegebenfalls sind darauf aufbauende Patches für die eingesetzte Architektur oder eine Echtzeiterweiterung wie den RT-Preemption-Patch notwendig.

Es ist zu beachten, daß beim Einsatz von mehreren Patches die Gefahr von Überschneidungen besteht. Das bedeutet, daß mindestens zwei Patches die gleichen Sourcen ändern. Genauso kann es vorkommen, daß ein notwendiger Architektur-Patch auf einer anderen Kernelversion aufsetzt als der ebenfalls notwendige Echtzeit-Patch. In solchen Fällen müssen beide Patches zusammengeführt werden. Durch die Integration einer zunehmenden Zahl an Architekturen und Boards in den Kernel-Mainstream und die sukzessive Integration des RT-Preemption-Patches wird das geschilderte Szenario zunehmend weniger dramatisch.

Bei der Kernel-Konfiguration beginnt man am zweckmässigsten mit einer minimalen Konfiguration, welche die verwendeten Bussysteme, Massenspeicher und Netzwerkkarte

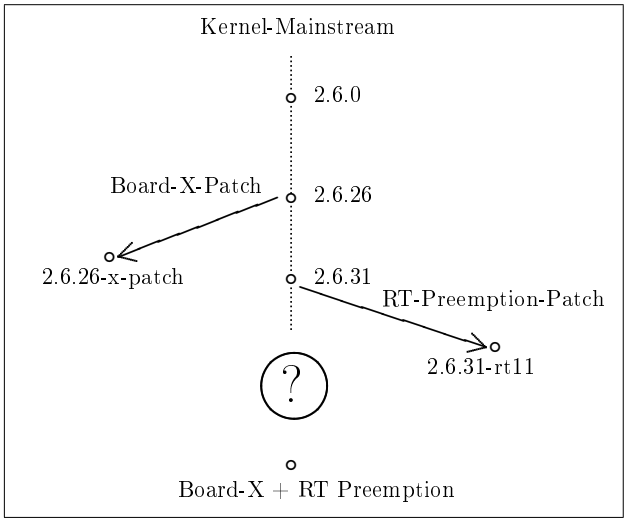

Abbildung 3: Szenario bei der Verwendung zweier unabhängiger Patches

enthält. Darauf aufbauend kann das Betriebssystem erweitert und ergänzt werden.

2.4 Root-Filesystem

Das Root-Filesystem enthält alles, was zu einem lebensfähigen Linux-System dazugehört. Das sind Shell-Programme, Device-Nodes, Konfigurationseinstellungen und Bibliotheken. Am leichtesten ist es, wenn man die Busybox zusammen mit der uClibc einsetzt und so bereits ein komplettes für schlange Embedded-Systeme geeignete Auswahl erhält.

Bei der Gestaltung des Root-Filesystems existieren mehrere Variationen. Es kann sich als Filesystem auf einem Flash oder anderen Massenspeicher befinden und vom System direkt verwendet werden. Genauso gut kann es gepackt auf dem Flash sein, beim Booten entpackt und in den Hauptspeicher kopiert als RAM-Disk verwendet werden. Für welche Variante man sich entscheidet hängt davon ab, in welche Richtung das System hinsichtlich der Resourcennutzung optimiert wird.

Häufig entscheided man sich dafür, die nicht notwendigerweise beschreibbaren Teile in einem Image abzulegen und die beschreibbaren Teile wie etwa die Systemkonfiguration in einer eigenen Partition unterbringen. Im Betriebszustand wird dann von der RAM-Disk aus auf die Konfiguration mittels symbolischen Links verwiesen.

3 Verfügbare Tools für die Erstellung

Neben der busybox gibt es inzwischen schon einige gut funktionierende Build-Systeme, welche den Entwickler dabei unterstützen, die Erstellung des Embedded-Systems zu vereinfachen und Routineaufgaben zu automatisieren. An dieser Stelle soll buildroot vorgestellt werden. Dieses Build-System beinhaltet neben der busybox und uClibc auch

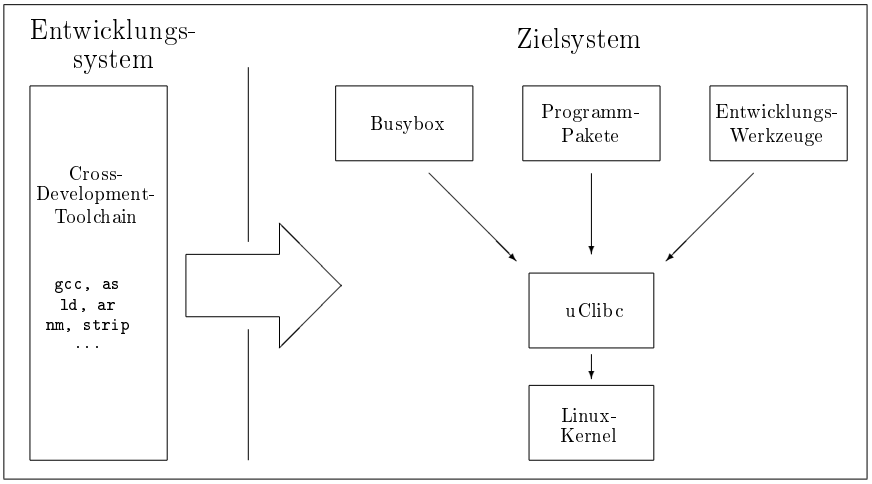

Abbildung 4: Bildprozess eines Embedded-Linux

Cross-Development-Tools für viele Architekturen, unzählige Programm-Pakete sowie Bootloader und Linux-Kernel.

Dieses System erstellt ausgehend von der gewählten Konfiguration zuerst eine Cross-Development-Toolchain. Mit dieser werden die restlichen Komponenten direkt für das Target erstellt. Rauskommen tut dabei ein komplettes Root-Filesystem mit allem, was für das Target benötigt wird. Dabei wird alles aus den Original-Sourcen erstellt und selbst Patches werden mit berücksichtigt. Es können auch Komponenten von ausserhalb in das Buildsystem aufgenommen werden.

Bereits die anfangs erstellte Cross-Toolchain macht das System wertvoll, da diese für die Erstellung der eigenen Anwendungen und Treiber eingesetzt werden kann.

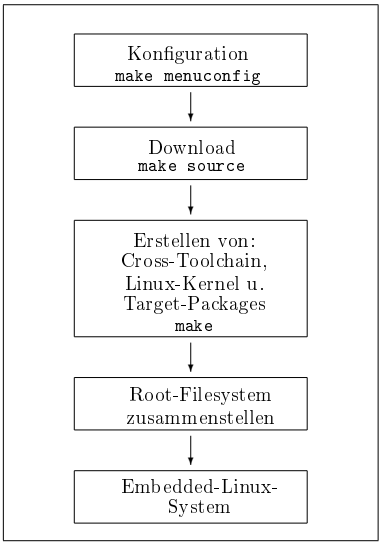

Abbildung 5: Ablaufsequenz bei der Erstellung mit buildroot

Die häufig vorhandene Anforderung nach einem reproduzierbaren Erstellvorgang kann mit diesem System abgedeckt werden. Dazu werden die Entwicklungswerkzeuge in einem separaten Verzeichnissystem abgelegt und mittels eines Change-Roots (chroot) in dieses Verzeichnissystem gewechselt. Bei diesem Verfahren ist sichergestellt, daß der Erstellvorgang immer exakt gleich abläuft.

4 Schlussfolgerungen

Es ist die Absicht dieses Artikels aufzuzeigen, daß der Erstellvorgang eines Embedded-Linux-Systems weitgehend automatisiert werden kann. Auf der anderen Seite kann das Target-System auch ganz ohne die Ziel-Hardware zusammengebaut werden. Probleme und Unverträglichkeiten zwischen unterschiedlichen Paketen oder Versionskonflikte werden dadurch frühzeitig erkannt.

Dem Entscheider bietet sich dadurch eine Grundlage für Abschätzungen bezüglich der erforderlichen Hard- und Software des Zielsystems. Um es anders auszudrücken, die Planungs- und Designphase kann durch Tests im Vorfeld unterstützt und verifiziert werden.

Der Aufwand für derartige „Proof-Of-Concepts“ ist im Vergleich zu etwaigen Redesigns aufgrund der breit angelegten und stabilen Softwarebasis sehr gering.