Grafikstacks für Embedded Linuxe

Kleine Maler

Systembuilder helfen dem Entwickler beim Bau eingebetteter Linux-Systeme, nicht jedoch bei der Auswahl eines für die eigene Anwendung und die Zielhardware optimalen Grafikstacks. Hier setzt der folgende Beitrag an und erklärt auch, warum Distributionen von der Stange bei kleiner Hardware keine gute Wahl sind. Alexander Hagg


© yarruta, 123RF

Vorgefertigte Distributionen für Embedded-Systeme enthalten umfangreiche grafische Fähigkeiten mit Videodecodierung, Audiocodecs und andere Komponenten, die viele Anwendungsfälle gar nicht benötigen, und somit Speicher und Performance unnötig belasten. Denn manche Anwendung braucht keine grafische Ausgabe oder kann ihre Funktionalität auf einem kleinen Touchdisplay per GUI entfalten.

Es liegt also nahe, beim Systemdesign Überlegungen zu den in Frage kommenden Grafikstacks anzustellen und sich mit dem Gedanken vertraut zu machen, eben keine Distribution von der Stange zu benutzen. Zum Glück hält die Open-Source-Tool-Landschaft ausreichend Hilfsmittel bereit.

Statt also ein aufgeblähtes, fertiges System zu verwenden, stellen Systembuilder wie Buildroot [1] und Yocto [2] minimalistische Linux-Systeme nach Entwickler-Wunsch zusammen. Die Werkzeuge erschaffen nicht nur das Userland, sondern, sofern gewünscht und nicht vorhanden, auch eine Cross-Compile-Toolchain und einen Kernel.

Während Yocto den Systemen eine Paketverwaltung spendieren kann, verzichtet Buildroot sogar darauf und ist damit besonders für minimale, überwiegend statische Systeme geeignet. Beim Zusammenstellen des Linux-Systems wählt der Entwickler neben den Grundkomponenten, für die es meist mitgelieferte Minimalkonfigurationen gibt, auch die grafischen Komponenten aus (siehe Kasten „Systembuilder").

Das Ganze am Beispiel eines Raspberry Pi

Für diesen Artikel benutzte der Autor einen Raspberry Pi 3 Model B mit Embedded Linux und Buildroot 2017.08.1. Die Buildkonfiguration modifizierte er – ausgehend von der mitgelieferten Konfigurationsdatei »raspberrypi3_defconfig« – auf ein dynamisches Management der Gerätedateien hin, um den Grafikbibliotheken einen Zugriff auf die Grafikressourcen zu bieten.

Um die verschiedenen Grafikstacks testen und eigene Anwendungen schreiben zu können, fügte er das Anwendungstoolkit Qt in Version 5 hinzu. Alternativ stehen in Buildroot auch GTK+, FLTK und weitere Werkzeugsammlungen bereit. Die noch anstehende Wahl der Grafikbibliothek dokumentiert dieser Artikel.

Grafikschichten zwischen Hardware und Anwendung

Anwendungen, die ohne Pixelbildschirm auskommen und ihre Informationen über LEDs oder ein Textdisplay liefern, sprechen ihre Zielhardware über Kernelkomponenten direkt an. Ist Grafikhardware vorhanden, lässt sich Text im Text-Mode in den Framebuffer projizieren. Das Linux Framebuffer Device, auch als Fbdev bekannt, ist eine grafische Abstraktionsschicht, um Hardware-unabhängig Grafiken anzuzeigen – beispielsweise das Tux-Logo, das manche Distributionen beim Booten (auf das Device »/dev/fb*«) auf die Konsole projizieren.

Während der Framebuffer ein Stück Speicher der Grafikeinheit ist, ermöglicht das Framebuffer-Device per Kernelmodul den Zugriff darauf, ohne den Overhead des X Window System in Kauf nehmen zu müssen. Dem Entwickler vereinfacht das Framebuffer-Device seine Arbeit jedoch nur wenig, da er alles pixelweise programmieren muss, was in der Regel auch die Performance beeinträchtigt. Zudem verarbeitet das Framebuffer-Device keine Eingabeereignisse.

Soll eine Anwendung komplexere grafische Oberflächen anbieten, schiebt man wie in Abbildung 2 besser zusätzliche Ebenen zwischen Kernel (2) und Anwendungen (5): Grafiktreiber (3) und Anzeigeserver (4).

Grafiktreiber sind typischerweise an die vorliegende Hardware angepasst und bieten nach oben hin von der Hardware abstrahierte Schnittstellen. Anzeigeserver und Fenstermanager vereinfachen die Ressourcenverwaltung zwischen mehreren Anwendungen und schaffen eine einheitliche grafische Plattform.

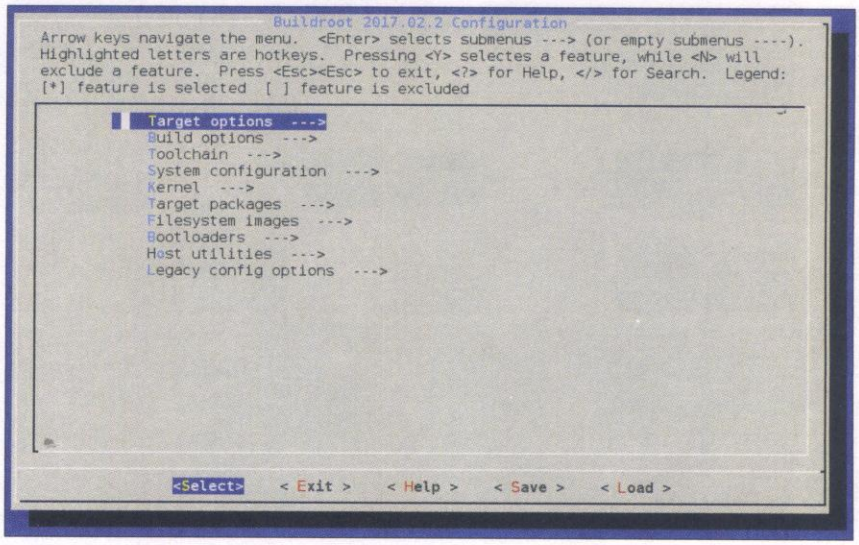

Abbildung 1: In der Menükonfiguration von Buildroot lassen sich Zielhardware, Systemeigenschaften, Anwendungen und Imageformat einfach anpassen.

Klein und schnell: Direct FB

Direct FB ist eine kleine Bibliothek, die direkt auf dem Linux-Framebuffer-Device aufbaut und eine Schnittstelle für einen vereinfachten Zugriff auf die Grafikhardware bietet. Neben der Abstraktion der Grafikausgabe arbeitet Direct FB mit der Hardwarebeschleunigung zusammen, kann verschiedene Eingabegeräte anbinden und am Ende einen Fenstermanager darüberstülpen. Direct FB stellt also ganz ähnlich wie ein normaler Anzeigeserver einen Grafiktreiber – erweitert um Eingabeverarbeitung und eine Fensterverwaltung – zur Verfügung.

Speziell für den Einsatz in eingebetteten Systemen entwickelt, hält Direct FB den Funktionsumfang zugunsten der Ressourcennutzung gering. Das Projekt ist 2015 verwaist, Buildroot offeriert die letzte stabile Veröffentlichung 1.7.7 auch

Systembuilder

Wer ein komplettes (Embedded)-Linux-System von Grund auf neu bauen will, benötigt eine Toolchain und Konfigurationsdateien, um Kernel und Treiber passend zur Hardware sowie Anwendungen passend zu Kernel und Treibern zu kompilieren und abzustimmen. Das Zusammensuchen der Makefiles und Konfigurationen ist aber aufwändig und langwierig.

In der Praxis haben sich deshalb Systembuilder durchgesetzt – Sammlungen von Makefiles, Meta-Informationen und Konfigurationsdateien, die aus den Quellen den Kernel, das Root-Filesystem, eine Toolchain und das Systemabbild eines individuellen eingebetteten Linux-Systems erzeugen [3].

Heruntergeladen und entpackt stellen die vielseitigen Werkzeugkisten Basiskonfigurationen für eine Vielzahl von Hardware-Architekturen bereit. Nicht mitgelieferte Konfigurationen finden sich oft online oder lassen sich selbst mit wenig Aufwand erstellen. Die Grundkonfiguration für einen Raspberry Pi 3 liegt für den Systembuilder Buildroot beispielsweise in der Konfigurationsdatei »buildroot/configs/raspberrypi3_defconfig«.

Die Systembuilder Open Embedded und Yocto bedient der Entwickler über die Kommandozeile oder – mit einem Zusatzprogramm – mit einem GUI. Die Auswahl der Vorkonfiguration von Buildroot erfolgt über die Kommandozeile, danach kann der Entwickler das System mit einem übersichtlichen, textbasierten UI im Konsolenfenster feintunen.

Nur Klicken ist einfacher

Möchte ein Entwickler beispielsweise ein Embedded Linux für einen Raspberry Pi 3 mit Buildroot entwickeln, wechselt er auf seinem Entwicklungsrechner in das Verzeichnis des entpackten Buildroot und tippt:

```
make raspberrypi3_defconfig
make menuconfig
```

Das erste Kommando kopiert die vorgefertigte Basiskonfiguration mit Informationen zur Hardware-Architektur von der Konfigurationssammlung in das Arbeitsverzeichnis von Buildroot. Im zweiten Aufruf erzeugt und startet das Make-Skript das Konfigurations-UI. Der Entwickler modifiziert dann in der einfach zu bedienenden Menüoberfläche die Eigenschaften des Zielsystems, dessen Treiber und Anwendungen sowie das Format und die Eigenheiten der auszugebenden Image-Dateien. Abbildung 1 zeigt das Buildroot-Menü direkt nach Start.

Nach Auswahl der Systemkonfiguration ruft der Entwickler in der Kommandozeile den Befehl »make« auf, um das Zielsystem zu übersetzen. Die Makefiles überprüfen die Konfigurationen nun auf Ungereimtheiten, laden die benötigten Quellcodes aus dem Internet, lösen Abhängigkeiten auf und kompilieren die einzelnen Komponenten in der korrekten Reihenfolge. Am Ende landen Cross-Compile-Toolchain und Images im Ordner »buildroot/output«.

Eigene Anwendungen und die Qual der Wahl

Die vom Systembuilder erzeugte Cross-Compile-Toolchain hilft Entwicklern, für das Linux-System passende eigene Anwendungen zu kompilieren. Die entstandenen Binaries übertragen sie durch Mounten des erzeugten Filesystems (auch automatisiert mit »postbuild«-Skripten für Buildroot) oder nach Systemstart mit SSH, Sftp, Rsync oder anderen Protokollen vom Entwicklungsrechner auf das Zielsystem – wenn auch dies die Protokolle integriert hat.

Mit Open Embedded und dessen Erweiterung Yocto gelingt es schnell, per Kommandozeile und GUI einfache Systeme aufzubauen und mit der Paketverwaltung »iPKG« zu erweitern. Yocto eignet sich besonders für Systeme, die dynamisch mit ihren Aufgaben wachsen.

Die Stärke von Buildroot sind dagegen statische Systeme, die ohne umfangreiche Anpassung direkt nach der Kompilierung in den Produktiveinsatz gehen sollen. Mit Hilfe des übersichtlichen UI kann der Buildroot-Benutzer seine Zielsysteme feingranularer konfigurieren als Yocto-Anwender.

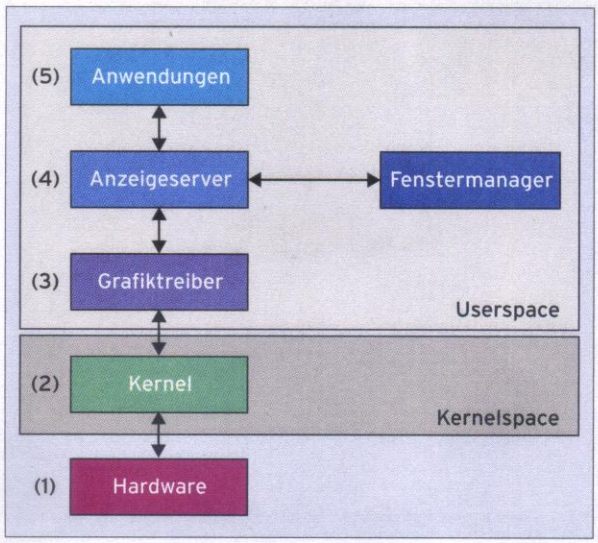

Abbildung 2: Das Schichtenmodell für einen Linux-Grafikstack.

für die neuesten Systembuilds. Yocto bietet nur die ältere Version 1.4.6 an.

Der Klassiker: X11

Das geschichtsträchtige X Window System (auch als X11 oder X bekannt) ist ein Anzeigeserver für Unix-basierte Computer. Es vereint den architekturunabhängigen Zugriff auf die Hardware mit einem Framework zum Darstellen und Verwalten von Anwendungen.

Das X Window System mit seiner Implementierung X.org lässt sich als Komplettpaket von grafischen Komponenten verstehen. Es enthält eine vollständige Ressourcenverwaltung und 2-D-Grafiktreiber sowie Bibliotheken, die Elemente auf grafischen Oberflächen zeichnen und verschieben und mit Eingabegeräten interagieren.

Ursprünglich für den Netzwerkeinsatz konzipiert ist X11 sehr vielseitig und deckt schon in einfacher Konfiguration eine breite Aufgabenfülle ab. Die Kommunikationsprotokolle sorgen für die Darstellung der Oberfläche auf einem Computersystem mit X-Server, die eigentliche Anwendung, der X-Client, läuft auf einem anderen Rechner im selben Netzwerk. X.org genießt breite Unterstützung und lässt sich unter Buildroot und Yocto umfassend konfigurieren und mit Treibern für diverse Grafik-Backends und Hardwarekonfigurationen ausstatten.

Der Neue: Wayland

Als moderne Alternative zum X Window System wird das 2008 erstmals erschienene Wayland-Protokoll immer beliebter [4]. Im Gegensatz zu X Window will Wayland kein Komplettpaket sein, der Fokus liegt auf der Beschreibung neuer Kommunikationsregeln sowie der Reduzierung auf die minimal notwendigen Komponenten.

Der Anzeigeserver ist hier keine Anwendung, nur eine Bibliothek. Bei Wayland ist jeder Fenstermanager zugleich ein Anzeigeserver, die Kombination trägt den Namen Compositor. Anwendungen wie auch andere Anzeigeserver (auch X-Server) können als Clients auf die Ressourcen eines Compositors zugreifen.

Während beim X Window System eine Vielzahl von Grafik- und anderen Treibern zum Paket gehört, muss sie der Wayland-Benutzer separat hinzufügen. Buildroot und Yocto stellen die Referenzimplementierung Weston mit Grafik-Backends für X11 und KMS zur Verfügung. Für Weston integriert Buildroot auch schon das Framebuffer-Device (Fbdev) sowie Open GL (Abbildung 3) und die Anwendungstoolkits Qt und GTK+ [5]. Wer sich für Open GL entscheidet, kann den Compositor beim Rendern teilweise umgehen.

Kann auch klein: Open GL

Entwickler, die komplexe grafische Oberflächen darstellen, setzen gern die 3-D-Grafikbibliothek Open GL ein. Während auf den meisten Desktoprechnern zusätzlich ein Anzeigeserver läuft, erhalten Embedded Linuxe ihre Grafikstacks auch ohne ihn bereitgestellt.

Um der schwächeren Hardware eines eingebetteten Systems bei komplexer Grafik zu helfen, benutzen Entwickler das Ressourcen-sparende Open GL ES mit der Erweiterung EGL. Letztere ist eine Programmierschnittstelle, die ohne zusätzlichen Fenstermanager ein einzelnes Fenster darzustellen in der Lage ist. Open GL ES arbeitet also alleinstehend oder mit Schnittstellen als Grafikbackend für X11 oder Wayland.

Wegen der knappen Hardwareleistung bleibt die Performance vieler Open-GL-Anwendungen auf Embedded-Systemen im Vergleich zu den anderen vorgestellten Backends zurück. Open GL ohne Anzeigeserver out of the Box zu benutzen, gelingt zudem selten.

Werkzeugkisten für Anwendungen

Wer eine native Anwendung von Grund auf zu schreiben beabsichtigt, tut gut daran, sich mit einem Anwendungstoolkit (auch GUI-Toolkit oder Anwendungsframework genannt) die Arbeit zu erleichtern. Denn ein solches Tookit vereinfacht die Anpassung an verschiedene grafische Backends der Zielsysteme und stellt geeignete Bibliotheken, Module und Funktionen bereit.

Tabelle 1: Grafik-Backends im Überblick

Eigenschaft	Direct FB	Open GL ES/EGL	X.org	Weston	Qt Web Engine
Bildwiederholrate	>120 fps	<15 fps	>120 fps	>100 fps	6 bis 22 fps
Arbeitsspeicher	++	++	+	+	-
CPU-Nutzung	++	++	+	+	-
Vollbildanwendungen	ja	ja	ja	ja	ja
Fenstermanager	optional	nein	ja	ja	nein
Anpassbarkeit	beschränkt	keine	hoch	hoch	keine
Funktionsumfang	gering	groß	groß, erweiterbar	groß, erweiterbar	groß

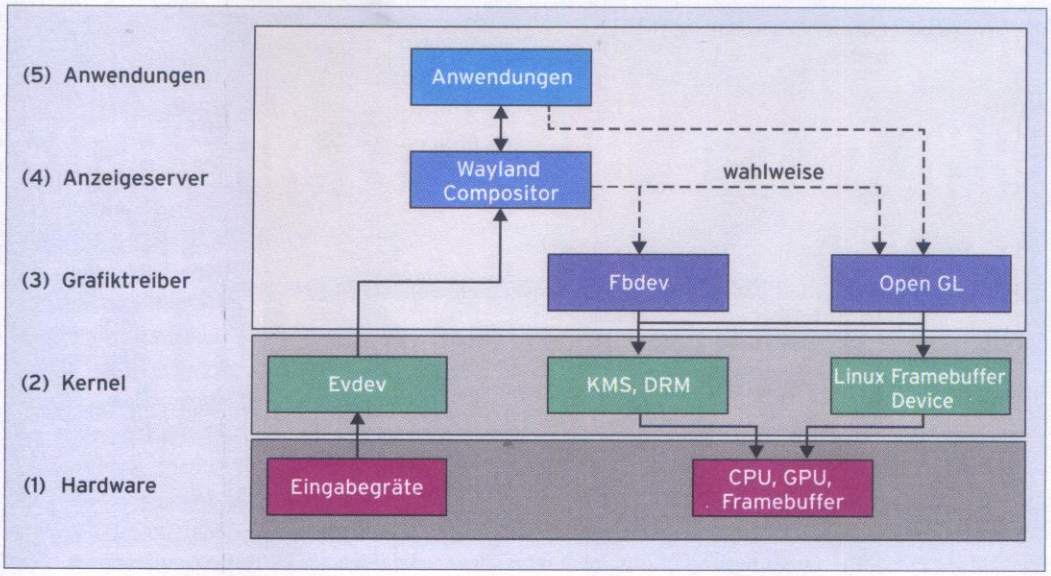

Abbildung 3: Der Wayland-Grafikstack mit Weston und Grafiktreibern für das Fbdev oder alternativ Open GL.

Die bekanntesten Anwendungstoolkits, GTK+, Qt und FLTK [6], beinhalten Programmierwerkzeuge, Steuerelement-Bibliotheken und Schnittstellen für die Anwendungsentwicklung unter diversen Betriebssystemen und Grafik-Backends wie Direct FB, Open GL und X11. Während GTK+ und Qt auch Wayland- und HTML-5-Anwendungen bedienen, bietet nur Qt eine stabile und vielfältige Unterstützung für mobile Plattformen, eingebettete Systeme sowie Open GL ES mit EGL.

X11-Desktop-Entwickler, die eher mit Toolkits wie Wx Widgets [7] und Tk vertraut sind, müssen bei Embedded umlernen. Wer seinen Fokus dagegen schon länger auf neuere Technologien wie Wayland oder Webapps gelegt hat, kann damit weitermachen. In der Praxis hat sich für Embedded- und mobile Anwendungen Qt ganz besonders bewährt, GTK+ so mittelprächtig.

Cute: Qt

Das quelloffene Qt-Framework abstrahiert Betriebssystem und Grafik-Backend so stark, dass Anwendungsentwickler nach dem Prinzip „Code once, deploy everywhere“ arbeiten können. Bei einer einheitlichen Codebasis für alle Plattformen wählen sie zur Compilezeit das Zielsystem und, sofern gewünscht, erst beim Start der Anwendung das Grafik-Backend aus.

Eine statische Verlinkung zwischen der Anwendung und Bibliotheken mit dem Ziel, das Qt-Framework auf dem Zielsystem zu sparen, gelingt Qt-Enwicklern am einfachsten mit der Entwicklerlizenz, die auch eine Nutzung der Qt-Anwendungen in kommerziellen Umgebungen ermöglicht [8]. Für nicht-kommerzielle Nutzung stehen der Qt-Kern, viele zusätzliche Bibliotheken und eine funktionsstarke C++-IDE frei zur Verfügung.

Mit oder ohne Browser: HTML 5

Wer das Glück hat, für eine Embedded-untypisch starke Hardware wie den Raspberry Pi zu entwickeln, für den rücken auch Webanwendungen in den Bereich

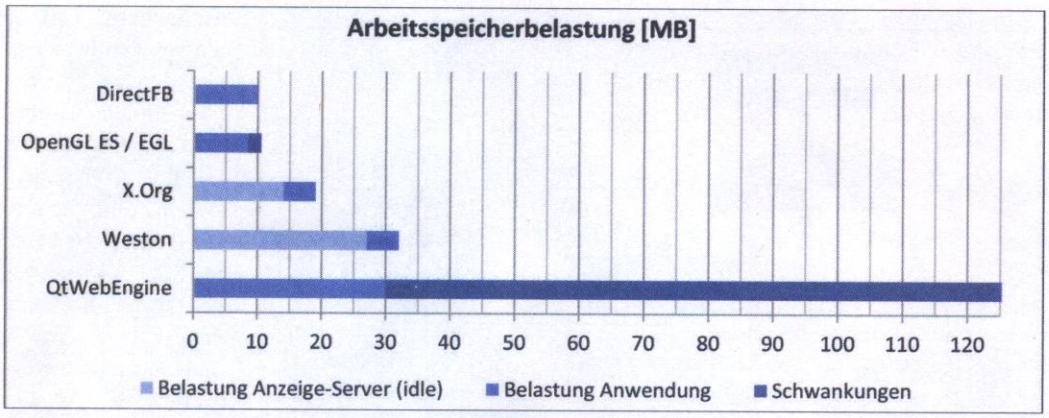

Abbildung 4: Die Arbeitsspeicherbelastung der Grafik-Backends im Einsatz. Native Anwendungen erweisen sich als genügsam, der Browser bringt das System an seine Grenzen.

des Möglichen. Der große Vorteil von Webapps ist die Plattformunabhängigkeit – die Gegenwart eines Open-GL-fähigen Browsers vorausgesetzt. Yocto und Buildroot bauen bei einem Open-GL-Backend den Browser Midori ein.

Mit Qt oder GTK+ lässt sich eine HTML-5-Anwendung sogar ohne vollwertigen Browser direkt auf einer HTML-Rendering-Engine ausführen. GTK+ hält dafür den HTML-Renderer Gecko parat, also den Kern von Mozilla Firefox.

Qt hat im Jahr 2014 von Apples Browserengine Webkit zu Blink gewechselt, der Engine des Browsers Chromium. Der Blink-Wrapper mit dem Namen Qt Web Engine [9] ist im Open-Source-Sinne um die Google-Komponenten bereinigt.

Die Leistung von Browsern auf eingebetteten Systemen ist wie zu erwarten niedrig. Die auf Desktops bekannten Browser-Probleme mit der großzügigen Arbeitsspeicher-Allokierung gewinnen bei den knappen Ressourcen von Embedded-Systemen mehr Bedeutung. Auf komplexe Webanwendungen mit Animationen sollte man besser nicht hoffen, der Betrieb einfacher Applikationen, die nur wenig Render-Arbeit verlangen, sollte hingegen gelingen.

Benchmarking

Das für diesen Artikel verwendete Raspberry-Pi-Testsystem kam auch für Geschwindigkeitsvergleiche zum Einsatz. Als Benchmark diente eine simple Grafikanwendung, die einen bewegten Schriftzug anzeigt, den ein Direct-FB-Backend über 125-mal pro Sekunde zu rendern schafft (erste Zeile in Tabelle 1). Einem X11-Backend, speziell der Referenzimplementierung X.org, gelangen ähnliche Bildwiederholraten, während ein Wayland-Backend nur knapp über 100 Frames per Second (fps) schaffte.

Zum Vergleich liefen anfänglich rudimentäre Tests mit einem reinen Open-GL-ES/EGL-Backend. Hier war die Animation mit rund 14 Bildern pro Sekunde nicht mehr als fließend zu erkennen. Auch eine vergleichende Webanwendung mit Animation war auf einem Minimalbrowser unter der Qt Web Engine je nach Animationskomplexität mit 6 bis 22 fps nur mit Geduld nutzbar.

Beim Ressourcenkonsum stach die Qt Web Engine wenig überraschend negativ heraus, die anderen Grafik-Backends belegten vergleichsweise wenig Arbeitsspeicher und CPU-Leistung. Die CPU-Belastung lag bei Direct FB, X11 und Wayland zwischen 15 und 30 Prozent. Die RAM-Auslastung dokumentiert Abbildung 4, hier wird die Speicheroptimierung der Anzeige-Server deutlich.

Durch die Auslagerung diverser Komponenten der Ressourcen- und Anwendungsverwaltung belegten der X.org-Server und der Weston Compositor auch im Leerlauf einen relativ großen Bereich des Arbeitsspeichers. Beim Start einer Anwendung waren daher viele grafische Komponenten bereits geladen, sodass nur wenig Speicher zusätzlich benötigt wurde.

Ohne den Overhead des Anzeigeservers kann das Direct-FB-Backend nicht durch Pre-Loading sparen, braucht allerdings insgesamt weniger Speicher. Der virtuelle Speicher der Qt Web Engine lag bei über 1 GByte, hier stieß auch im Test der Raspberry Pi an seine Grenzen.

Fazit

Anwendungsentwickler für minimalistische eingebettete Systeme erreichen mit Systembuildern wie Buildroot und Yocto schnell gute Ergebnisse. Dazu passen Anwendungen, die sich auf Anwendungstoolkits (insbesondere Qt) stützen. Die Wahl des dazwischen liegenden Grafik-Backends hängt vom Anwendungsgebiet ab: Soll das System eine einzelne native Vollbildanwendung anzeigen, bietet sich das schlanke und hochperformante Direct FB an. Für einen Desktop-Look mit Anwendungswechsel und Erweiterbarkeit liegt man mit dem guten alten X Window System sicher nicht falsch. Ein Blick auf das modernere Wayland lohnt aber allemal.

Wessen Hardware genug Kraft hat, kann vielleicht von der Plattform-Neutralität einer Webanwendung profitieren. Qt Web Engine und ein Open-GL-Backend stellen hier die Eckpfeiler. Die Praxis zeigt: In Sachen Geschwindigkeit bleiben hier Wunder trotzdem aus. *(jk)* ■

Infos

[1] Buildroot: [https://buildroot.org]
[2] Yocto: [https://www.yoctoproject.org]
[3] Quade, „Embedded Linux lernen mit dem Raspberry Pi": Dpunkt-Verlag, 1. Auflage, 2014
[4] Tim Schürmann, „Die neuen Displayserver": Linux-Magazin 01/17, S. 28
[5] Martin Gräßlin, „Qt-Anwendungen unter Wayland": Linux-Magazin 01/17, S. 36
[6] Fast Light Toolkit: [http://www.fltk.org]
[7] Wx Widgets: [http://www.wxwidgets.org]
[8] Qt-Lizenzen: [https://www1.qt.io/licensing/]
[9] Qt Web Engine: [https://wiki.qt.io/QtWebEngine]

Der Autor

Alexander Hagg hat den Master of Science in Informatik an der Hochschule Niederrhein gemacht. Seine Masterarbeit über Embedded Linux hat sein Interesse für freie Software noch gesteigert.