

Mastering Embedded Linux Programming

Third Edition

Create fast and reliable embedded solutions with
Linux 5.4 and the Yocto Project 3.1 (Dunfell)

Frank Vasquez

Chris Simmonds

Packt

BIRMINGHAM—MUMBAI

Mastering Embedded Linux Programming

Third Edition

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Wilson Dsouza

Publishing Product Manager: Sankalp Khattri

Senior Editor: Rahul Dsouza

Content Development Editor: Sayali Pingale

Technical Editor: Sarvesh Jaywant

Copy Editor: Safis Editing

Project Coordinator: Neil Dmello

Proofreader: Safis Editing

Indexer: Tejal Soni

Production Designer: Nilesh Mohite

First published: December 2015

Second edition: June 2017

Third edition: March 2021

Production reference: 0140421

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78953-038-4

www.packt.com

*To the open source software community (especially the Yocto Project)
for welcoming me in wholeheartedly. And to my wife Deborah for putting
up with the late-night hardware hacking. The world runs on Linux.*

– Frank Vasquez

Contributors

About the authors

Frank Vasquez is an independent software consultant specializing in consumer electronics. He has over a decade of experience designing and building embedded Linux systems. During that time, he has shipped numerous devices including a rackmount DSP audio server, a diver-held sonar camcorder, and a consumer IoT hotspot. Before his career as an embedded Linux engineer, Frank was a database kernel developer at IBM, where he worked on DB2. He lives in Silicon Valley.

Chris Simmonds is a software consultant and trainer living in southern England. He has almost two decades of experience in designing and building open source embedded systems. He is the founder and chief consultant at 2net Ltd, which provides professional training and mentoring services in embedded Linux, Linux device drivers, and Android platform development. He has trained engineers at many of the biggest companies in the embedded world, including ARM, Qualcomm, Intel, Ericsson, and General Dynamics. He is a frequent presenter at open source and embedded conferences, including the Embedded Linux Conference and Embedded World.

About the reviewers

Ned Konz is an autodidact who believes in Sturgeon's law and tries to work on the other 10% of everything. His work experience over the last 45 years includes software and electronics design for industrial machines, and consumer and medical devices, as well as doing user interface research with Alan Kay's team at HP Labs. He has embedded Linux in devices including high-end SONAR systems, inspection cameras, and the Glowforge laser cutter. As a senior embedded systems programmer at Product Creation Studio in Seattle, he designs software and electronics for a variety of client products. In his spare time, he builds electronics gadgets and plays bass in a rock band. He has also done two solo bicycle tours of over 4,500 miles each.

I'd like to thank my wife Nancy for her support, and Frank Vasquez for recommending me as a technical reviewer.

Khem Raj holds a bachelor's degree (Hons) in electronics and communications engineering. In his career spanning 20 years in software systems, he has worked with organizations ranging from start-ups to Fortune 500 companies. During this time, he has worked on developing operating systems, compilers, computer programming languages, scalable build systems, and system software development and optimization. He is passionate about open source and is a prolific open source contributor, maintaining popular open source projects such as the Yocto Project. He is a frequent speaker at open source conferences. He is an avid reader and a lifelong learner.

Table of Contents

Preface

Section 1: Elements of Embedded Linux

1

Starting Out

Choosing Linux	4	Selecting hardware for embedded Linux	10
When not to choose Linux	5	Obtaining the hardware for this book	12
Meeting the players	6	The Raspberry Pi 4	12
Moving through the project life cycle	7	The BeagleBone Black	13
The four elements of embedded Linux	8	QEMU	14
Navigating open source	8	Provisioning your development environment	15
Licenses	9	Summary	16

2

Learning about Toolchains

Technical requirements	18	Building a toolchain using crosstool-NG	26
Introducing toolchains	18	Installing crosstool-NG	26
Types of toolchains	20	Building a toolchain for BeagleBone Black	27
CPU architectures	21	Building a toolchain for QEMU	28
Choosing the C library	23		
Finding a toolchain	25		

Anatomy of a toolchain	29	Shared libraries	37
Finding out about your cross compiler	30	The art of cross-compiling	39
The sysroot, library, and header files	32	Simple makefiles	40
Other tools in the toolchain	33	Autotools	40
Looking at the components of the C library	34	Package configuration	45
		Problems with cross-compiling	46
Linking with libraries – static and dynamic linking	35	CMake	47
Static libraries	36	Summary	49
		Further reading	50

3

All about Bootloaders

Technical requirements	52	Labels and interrupts	60
What does a bootloader do?	52	Device tree include files	61
The boot sequence	53	Compiling a device tree	63
Phase 1 – ROM code	54	U-Boot	64
Phase 2 – secondary program loader	55	Building U-Boot	64
Phase 3 – TPL	56	Installing U-Boot	66
		Using U-Boot	69
Moving from the bootloader to a kernel	57	Booting Linux	74
Introducing device trees	58	Porting U-Boot to a new board	75
Device tree basics	58	Building and testing	81
The reg property	59	Falcon mode	82
		Summary	83

4

Configuring and Building the Kernel

Technical requirements	86	Building the kernel	91
What does the kernel do?	86	Getting the source	91
Choosing a kernel	88	Understanding kernel configuration – Kconfig	93
Kernel development cycle	88	Using LOCALVERSION to identify your kernel	98
Stable and long-term support releases	89	When to use kernel modules	99

Compiling – Kbuild	99	Booting the Raspberry Pi 4	108
Finding out which kernel target to build	100	Booting the BeagleBone Black	109
Build artifacts	101	Booting QEMU	110
Compiling device trees	103	Kernel panic	110
Compiling modules	103	Early user space	110
Cleaning kernel sources	104	Kernel messages	111
Building a 64-bit kernel for the Raspberry Pi 4	104	The kernel command line	112
Building a kernel for the BeagleBone Black	107	Porting Linux to a new board	113
Building a kernel for QEMU	107	A new device tree	114
Booting the kernel	108	Setting the board's compatible property	115
		Summary	118
		Additional reading	119

5

Building a Root Filesystem

Technical requirements	122	Booting with QEMU	140
What should be in the root filesystem?	123	Booting the BeagleBone Black	140
The directory layout	124	Building an initramfs into the kernel image	141
The staging directory	125	Building an initramfs using a device table	142
POSIX file access permissions	126	The old initrd format	143
File ownership permissions in the staging directory	127	The init program	144
Programs for the root filesystem	128	Starting a daemon process	145
Libraries for the root filesystem	132	Configuring user accounts	145
Device nodes	134	Adding user accounts to the root filesystem	147
The proc and sysfs filesystems	136	A better way of managing device nodes	148
Kernel modules	137	An example using devtmpfs	148
Transferring the root filesystem to the target	138	An example using mdev	149
Creating a boot initramfs	139	Are static device nodes so bad after all?	150
Standalone initramfs	139		
Booting the initramfs	140		

Configuring the network	150	Testing with QEMU	155
Network components for glibc	151	Testing with the BeagleBone Black	156
		Problems with file permissions	157
Creating filesystem images with device tables	152	Using TFTP to load the kernel	157
Booting the BeagleBone Black	153	Summary	158
		Further reading	159
Mounting the root filesystem using NFS	154		

6

Selecting a Build System

Technical requirements	162	Introducing the Yocto Project	179
Comparing build systems	162	Background	180
Distributing binaries	164	Stable releases and supports	181
Introducing Buildroot	165	Installing the Yocto Project	182
Background	165	Configuring	182
Stable releases and long-term support	165	Building	183
Installing	165	Running the QEMU target	184
Configuring	166	Layers	185
Running	167	Customizing images via local.conf	191
Targeting real hardware	169	Writing an image recipe	192
Creating a custom BSP	169	Creating an SDK	193
Adding your own code	176	The license audit	195
License compliance	179	Summary	196
		Further reading	196

7

Developing with Yocto

Technical requirements	198	Capturing changes with devtool	216
Building on top of an existing BSP	199	Development workflows	216
Building an existing BSP	199	Creating a new recipe	218
Controlling Wi-Fi	206	Modifying the source built by a recipe	220
Controlling Bluetooth	209	Upgrading a recipe to a newer version	222
Adding a custom layer	213		

Building your own distro	225	Runtime package management	227
When and when not to	225	Provisioning a remote package server	229
Creating a new distro layer	225	Summary	231
Configuring your distro	226	Further reading	232
Adding more recipes to your distro	227		

8

Yocto Under the Hood

Technical requirements	234	Adding more logging	245
Decomposing Yocto's architecture and workflow	234	Running commands from devshell	246
Metadata	236	Graphing dependencies	246
Build tasks	238	Understanding BitBake syntax and semantics	248
Image generation	239	Tasks	248
Separating metadata into layers	240	Dependencies	249
Troubleshooting build failures	243	Variables	250
Isolating errors	243	Functions	254
Dumping the environment	244	RDEPENDS revisited	257
Reading the task log	245	Summary	258
		Further reading	258

Section 2: System Architecture and Design Decisions

9

Creating a Storage Strategy

Technical requirements	262	Accessing flash memory from the bootloader	268
Storage options	263	U-Boot and NOR flash	268
NOR flash	263	U-Boot and NAND flash	269
NAND flash	264	U-Boot and MMC, SD, and eMMC	269

Accessing flash memory from Linux	269	Discard and TRIM	289
Memory technology devices	270	Ext4	291
The MMC block driver	276	F2FS	292
		FAT16/32	292
Filesystems for flash memory	277	Read-only compressed filesystems	293
Flash translation layers	277	SquashFS	293
Filesystems for NOR and NAND flash memory	278	Temporary filesystems	294
JFFS2	278	Making the root filesystem read-only	295
YAFFS2	281	Filesystem choices	296
UBI and UBIFS	283	Summary	297
Filesystems for managed flash	288	Further reading	297
Flashbench	288		

10

Updating Software in the Field

Technical requirements	300	Asymmetric image update	309
From where do updates originate?	300	Atomic file updates	310
What to update	301	OTA updates	312
Bootloader	302	Using Mender for local updates	313
Kernel	302	Building the Mender client	313
Root filesystem	302	Installing an update	316
System applications	303	Using Mender for OTA updates	319
Device-specific data	303	Using balena for local updates	327
Components that need to be updated	303	Creating an account	327
The basics of software updates	303	Creating an application	328
Making updates robust	304	Adding a device	329
Making updates fail-safe	305	Installing the CLI	332
Making updates secure	307	Pushing a project	334
Types of update mechanism	308	Summary	337
Symmetric image update	308		

11

Interfacing with Device Drivers

Technical requirements	340	SPI	362
The role of device drivers	341	Writing a kernel device driver	362
Character devices	342	Designing a character driver interface	363
Block devices	344	The anatomy of a device driver	364
Network devices	346	Compiling kernel modules	368
Finding out about drivers at runtime	347	Loading kernel modules	369
Getting information from sysfs	350	Discovering the hardware configuration	370
Finding the right device driver	353	Device trees	370
Device drivers in user space	354	The platform data	371
GPIO	354	Linking hardware with device drivers	372
LEDs	358	Summary	374
I2C	359	Further reading	375

12

Prototyping with Breakout Boards

Technical requirements	378	Closing the SPI jumper	403
Mapping schematics to the device tree's source	379	Attaching the GNSS antenna	404
Reading schematics and data sheets	380	Attaching the SPI header	404
Installing Debian on the BeagleBone Black	384	Connecting the SPI jumper wires	406
Enabling spidev	385	Probing SPI signals with a logic analyzer	410
Customizing the device tree	392	Receiving NMEA messages over SPI	419
Prototyping with breakout boards	401	Summary	423
		Further reading	423

13

Starting Up – The init Program

Technical requirements	426	systemd	438
After the kernel has booted	426	Building systemd with the Yocto Project and Buildroot	438
Introducing the init programs	428	Introducing targets, services, and units	439
BusyBox init	428	How systemd boots the system	442
Buildroot init scripts	430	Adding your own service	442
System V init	430	Adding a watchdog	444
inittab	432	Implications for embedded Linux	445
The init.d scripts	435	Summary	446
Adding a new daemon	436	Further reading	446
Starting and stopping services	437		

14

Starting with BusyBox runit

Technical requirements	448	Custom start dependencies	471
Getting BusyBox runit	449	Putting it all together	472
Creating service directories and files	455	Dedicated service logging	472
Service directory layout	455	How does it work?	473
Service configuration	457	Adding dedicated logging to a service	473
Service supervision	465	Log rotation	475
Controlling services	467	Signaling a service	477
Depending on other services	469	Summary	478
Start dependencies	470	Further reading	479

15

Managing Power

Technical requirements	482	Scaling the clock frequency	487
Measuring power usage	483	The CPUFreq driver	488
		Using CPUFreq	489

Selecting the best idle state	492	Power states	499
The CPUIdle driver	493	Wakeup events	501
Tickless operation	496	Timed wakeups from the real-time clock	502
Powering down peripherals	497	Summary	504
Putting the system to sleep	499	Further reading	504

Section 3: Writing Embedded Applications

16

Packaging Python

Technical requirements	508	Installing precompiled binaries with conda	523
Getting Docker	509	Environment management	524
Retracing the origins of Python packaging	509	Package management	526
distutils	510	Deploying Python applications with Docker	529
setuptools	510	The anatomy of a Dockerfile	529
setup.py	511	Building a Docker image	532
Installing Python packages with pip	514	Running a Docker image	533
requirements.txt	516	Fetching a Docker image	534
Managing Python virtual environments with venv	520	Publishing a Docker image	535
		Cleaning up	536
		Summary	537
		Further reading	538

17

Learning about Processes and Threads

Technical requirements	540	Terminating a process	544
Process or thread?	540	Running a different program	545
Processes	542	Daemons	548
Creating a new process	543	Inter-process communication	549

Threads	555	Messaging between processes	563
Creating a new thread	555	Messaging within processes	565
Terminating a thread	556	Scheduling	567
Compiling a program with threads	557	Fairness versus determinism	567
Inter-thread communication	557	Time-shared policies	568
Mutual exclusion	558	Real-time policies	570
Changing conditions	558	Choosing a policy	571
Partitioning the problem	560	Choosing a real-time priority	571
ZeroMQ	562	Summary	572
Getting pyzmq	562	Further reading	572

18

Managing Memory

Technical requirements	574	Using mmap to access device memory	586
Virtual memory basics	574	How much memory does my application use?	586
Kernel space memory layout	576	Per-process memory usage	587
How much memory does the kernel use?	577	Using top and ps	588
User space memory layout	579	Using smem	589
The process memory map	581	Other tools to consider	591
Swapping	583	Identifying memory leaks	591
Swapping to compressed memory (zram)	583	mtrace	592
Mapping memory with mmap	584	Valgrind	593
Using mmap to allocate private memory	585	Running out of memory	595
Using mmap to share memory	585	Summary	596
		Further reading	597

Section 4: Debugging and Optimizing Performance

19

Debugging with GDB

Technical requirements	602	GDB user interfaces	622
The GNU debugger	602	Terminal User Interface	623
Preparing to debug	603	Data Display Debugger	623
Debugging applications	604	Visual Studio Code	625
Remote debugging using gdbserver	604	Debugging kernel code	633
Setting up the Yocto Project for remote debugging	605	Debugging kernel code with kgdb	633
Setting up Buildroot for remote debugging	606	A sample debug session	634
Starting to debug	607	Debugging early code	636
Native debugging	617	Debugging modules	637
Just-in-time debugging	619	Debugging kernel code with kdb	638
Debugging forks and threads	619	Looking at an Oops	639
Core files	620	Preserving the Oops	643
Using GDB to look at core files	621	Summary	644
		Further reading	645

20

Profiling and Tracing

Technical requirements	648	Building perf with Buildroot	654
The observer effect	649	Profiling with perf	655
Symbol tables and compile flags	649	Call graphs	657
Beginning to profile	650	perf annotate	658
Profiling with top	651	Tracing events	660
The poor man's profiler	652	Introducing Ftrace	660
Introducing perf	653	Preparing to use Ftrace	661
Configuring the kernel for perf	654	Using Ftrace	661
Building perf with the Yocto Project	654	Dynamic Ftrace and trace filters	664
		Trace events	665

Using LTTng	667	Using BPF tracing tools	675
LTTng and the Yocto Project	667	Using Valgrind	678
LTTng and Buildroot	667	Callgrind	678
Using LTTng for kernel tracing	668	Helgrind	679
Using BPF	670	Using strace	680
Configuring the kernel for BPF	671	Summary	683
Building a BCC toolkit with Buildroot	674	Further reading	683

21

Real-Time Programming

Technical requirements	686	The Yocto Project and PREEMPT_RT	697
What is real time?	687	High-resolution timers	697
Identifying sources of non-determinism	689	Avoiding page faults	698
Understanding scheduling latency	690	Interrupt shielding	699
Kernel preemption	691	Measuring scheduling latencies	700
The real-time Linux kernel (PREEMPT_RT)	692	cyclicttest	700
Threaded interrupt handlers	692	Using Ftrace	704
		Combining cyclicttest and Ftrace	706
Preemptible kernel locks	695	Summary	707
Getting the PREEMPT_RT patches	696	Further reading	708
		Why subscribe?	709

Other Books You May Enjoy

Index
