PRACTICAL GPL COMPLIANCE

A GUIDE FOR STARTUPS, SMALL BUSINESSES AND ENGINEERS

ARMIJN HEMEL, MSC AND SHANE COUGHLAN

Copyright © 2017 Linux Foundation
All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review and certain other noncommercial uses permitted by copyright law.

Printed in the United States of America

First Edition, 2017

ISBN: 978-0-9989078-0-2

1 Letterman Drive
Building D
Suite D4700
San Francisco CA 94129
Phone/Fax: +1 415 723 9709
https://linuxfoundation.org

About the Authors

Shane Coughlan

Shane Coughlan is an expert in communication, security, and business development. His professional accomplishments include spearheading the licensing team that elevated Open Invention Network into the largest patent non-aggression community in history, establishing the leading professional network of open source legal experts, and aligning stakeholders to launch both the first law journal and the first law book dedicated to open source. He currently leads the OpenChain community as Program Manager.

Shane has extensive knowledge of open source governance, internal process development, supply chain management, and community building. His experience includes engagement with the enterprise, embedded, mobile, and automotive industries.

Armijn Hemel

Armijn Hemel is the owner of Tjaldur Software Governance Solutions. He is an active researcher of and internationally recognized expert in open source license compliance and supply chain management. He studied computer science at Utrecht University in The Netherlands, where he pioneered reproducible builds with NixOS. In the past he served on the board of NLUUG and was a member of the coreteam of gpl-violations.org. Currently he is a board member at NixOS Foundation.

To Kate, for tirelessly driving the compliance community forward

ARMIJN HEMEL

To Lana, who has a sense of curiosity that spans the whole world

SHANE COUGHLAN

“Don’t Panic.”

DOUGLAS ADAMS

Introduction

Practical GPL Compliance is a compliance guide for startups, small businesses, and engineers, particularly focused on complying with the versions of the GNU General Public License (GPL). It is designed for engineers shipping products with GPL-licensed software included (e.g., consumer electronics, drones, IoT devices). Its goal is to provide practical information to quickly address common issues. It is intended to be useful for solving real-world challenges rather than providing perfection in an imperfect world and to serve as the basis for empowering a compliance engineer or compliance team to get their job done as efficiently as possible. Hopefully, the practices laid out in this guide will assist you in complying with other open source license terms as well.

This book begins by introducing the tools used to practice GPL compliance. It then expands on the goals of our approach, and follows with an explanation of how to accomplish those goals. It continues with a "checklist" of pitfalls frequently encountered by compliance engineers and a list of steps that can be applied in common situations when releasing a product or product family. It ends with some handy flowcharts to visualize key approaches or best practices.

If you have a physical copy of this book, it should end up as a battered, dog-eared reference text lurking beside your keyboard. If you have a digital copy, it should be something that appears in your favorites list or your desktop. Compliance engineering is not something easily memorized and finished. It is a process — an approach backed by tools and knowledge of best practices — and we do best in this field when we continually refresh and hone our skills.

Thankfully, compliance engineering is no longer a "black box" mastered by only a few. Today, with texts like this or *Open Source*

Compliance in the Enterprise[1], every engineer can support the excellent use of third-party code. Licenses like the GPL, once regarded as challenging to fully adhere to, can become understandable and addressable by large and small entities alike.

Practical GPL Compliance and Open Source Compliance in the Enterprise work together to help engineers, startups, small companies, and enterprises master open source license compliance. However, they do not exist in isolation, and you can find more materials via the Linux Foundation Open Compliance Program at **https://compliance.linuxfoundation.org**. To get you started, Appendix 1 contains an overview of other publications available and a list of useful compliance templates.

You may also be interested in Appendix 2: Compliance Standards, Appendix 3: Professional Networks, and Appendix 4: Tools and Infrastructure. There is a wealth of free resources available via The Linux Foundation and from other parties to ensure that knowledge of best practices and processes is readily available to you. Or…, just go to the next page and get started right away with our cheat sheet.

1. **http://go.linuxfoundation.org/open-source-compliance-ebook**

The GPL Compliance Engineer Task-Based Cheat Sheet

No time to read a book? Welcome to our world. We suggest copying this page, pinning it to your desk, and using it as a shortcut for getting things done.

What you need to do 2

The tools you can use 6

How to deal with binary code 8

Find problem binaries 10

Rebuilding a binary 14

Finding incorrectly licensed code 19

Common pitfalls 22

Releasing software 37

Buying software 42

Building a FOSS code center 48

Get checklists to help 52

Get flowcharts to help 54

Table of Contents

Chapter 1: Approach 1

Context 2

Compliance Requirements 3

Compliance Goals 3

Toolbox 6

Analysis of Binary Files 8

Source Code Analysis and Rebuild 10

Chapter 2: Common Pitfalls 22

Pitfall #1: Toolchain 23

Pitfall #2: Android 24

Pitfall #3: “Out of tree” Linux Kernel Modules 27

Pitfall #4: Rescue Mode/Install Mode Systems 30

Pitfall #5: Bootloader 31

Pitfall #6: Missing Build System 31

Pitfall #7: Incorrect or Missing BusyBox Configuration Files 32

Pitfall #8: Incorrect or Missing Linux Kernel Configuration Files 33

Pitfall #9: Not Including the Version Number in Firmware and Source Code Archive Names 35

Chapter 3: Scenarios for Releasing Software 37

Scenario #1: Software On A Device/Offline Distribution 38

Scenario #2: Providing A Manual Download From Website 39

Scenario #3: Providing An Automatic Download/ Over The Air 40

Scenario #4: Field Engineer Applied Updates 41

Chapter 4: Scenarios for Buying Software 42

Context 43

Scenario #1: SupplyChain Solutions For SoC Vendors 43

Scenario #2: Supply Chain Solutions For ODMs 45

Scenario #3: Supply Chain Solutions For Others 46

Chapter 5: Building a FOSS Code Center 48

Context 49

"FOSS Code Center" As A Requirement 49

Keep Firmware And Source Code Together 50

Chapter 6: Tracking Tasks and Processes 51

Checklists 52

Flowcharts 54

Appendices 63

Appendix 1: The Open Compliance Program 64

Appendix 2: Compliance Standards 68

Appendix 3: Professional Networks 69

Appendix 4: Tools & Infrastraucture 70

CHAPTER 1:

Approach

“In theory there is no difference between theory and practice. In practice there is.”

YOGI BERRA

Context

Compliance engineering consists of a good approach, a good toolbox, and a good process. First things first: What are our goals in compliance engineering?

This book focuses on GPL Compliance Engineering. The GPL is a copyright license with terms that trigger when we distribute the code. The most famous and widely used version of the license is the GPL Version 2 (GPLv2). Software such as the Linux kernel or many commonly-used versions of GNU userland tools fall under this license. There are other licenses, such as the Library or Lesser GPL (LGPL) or Affero GPL (AGPL), which are regarded as being in the same "family" but which have different terms. We address key aspects of the LGPL in Flowchart #5[2] later in this text, but we do not specifically address the terms of the AGPL in this book.

The focus in this book will be on GPL Version 2. This is formally referred to as the GPLv2. However, throughout this book we will refer to it as the GPL to keep things simple. This means that unless explicitly stated otherwise, we mean "GPL Version 2" when we say "GPL." You can review this license and all the other GPL family licenses on the Free Software Foundation website.[3]

GPL compliance engineers are concerned about software products that are electronically or physically distributed. Sometimes this means scanning source code to confirm it has the expected licensing. Sometimes this means scanning binary code or cross-checking binary code with source code to confirm that they match. In practice, most compliance work is focused on physical products sent to market or firmware downloads.

2. See page 62

3. See **https://www.gnu.org/licenses/licenses.html**

This book explains the process of confirming whether the binary code on a physical device contains GPL code and then taking action to ensure compliance if it does.

Compliance Requirements

The GPLv2 (hereafter the GPL) has a couple of important requirements. One is ensuring that a copy of the license is provided along with the distributed binary or source code. The other is providing access to source code when distributing binary code either as a stand-alone software application or as part of a physical product.

The GPL describes two ways to comply with the source code access requirement:

1. Accompany a product with the "complete and corresponding source code" (section 3a)

2. Include a written offer to supply the "complete and corresponding source code" (section 3b)

These two methods differ in **how** the source code is delivered (either with the binary code or later if requested), but they do not differ in **what** needs to be delivered: complete and corresponding source code.

Compliance Goals

Now that we understand the GPL compliance requirements, we can understand our overarching compliance goals:

1. Make sure a copy of the license accompanies the distributed binary or source code .

2. Make sure that the "complete and corresponding" source code is available.

Copy of the License

When you distribute GPL code as binary or source code, you need to ensure it is accompanied by a copy of the GPL license. This is the easiest and quickest part of any GPL compliance engineering process.

You can include a copy of the GPL license as physical or digital media along with a product. Some examples are:

- Smart televisions that come with a copy of the GPL physically printed at the back of the instruction manual along with other legal notices.
- Smartphones that come with a copy of the GPL under the Settings > Legal menu or similar location.

The important objective is to ensure that the license is easily discoverable by an interested party.

"Complete and Corresponding" Source Code

This is a challenging part of GPL compliance and the situation that the majority of this book helps to explore, explain, and solve. As mentioned above, our focus is primarily on physical products distributed with GPL software contained inside, because this is the most common and most problematic use case for compliance engineers. Typical examples include firmware flashed onto a device or firmware updates downloadable from a website.

Two Key Considerations

There are two primary checks you need to focus on to ensure compliance:

1. The source code does not contain any license violations.
2. The source code is "complete and corresponding" for the binary code distributed.

The Ideal Situation

In an ideal world, there is a binary package or a collection of binary packages, such as a firmware and a corresponding source code archive. The corresponding source code archive contains only open source components and perhaps some object files to relink binaries that contain Library or Lessor GPL (LGPL) software, along with instructions for rebuilding the binary or firmware. After performing the rebuild, the original binary file is an exact match to the rebuilt binary file.

The Reality

Source code is often not “complete and corresponding.” When you rebuild a binary, you may find different file sizes or even completely different versions or types of binary code compared to the expected original. This is the first major challenge we face when seeking to ensure GPL compliance.

Another challenge comes as a consequence of the above. The source code for a device often contains license violations that are unrelated to the binary code it is meant to support. For example, source code archives sometimes contain prebuilt binaries that have no relevance to being “complete and corresponding,” and instead serve only to put a company out of compliance when it makes source code available in good faith.

This is where the bulk of GPL compliance engineering takes place. Compliance engineers live on the intersection between source code and binary code. Our challenge is to ensure that a physical product ships with the expected code, with a copy of the correct licenses, and with the “complete and corresponding” source code or a written offer to provide that “complete and corresponding” source code on demand.

Toolbox

Now that our challenge is clear, it is time to talk about the type of tools we need to get things done. This discussion is not intended to be exhaustive, but rather to provide a starting point. If you have access to the tools we talk about below, you can do everything contained in this guide. In turn, that will allow you to address the vast majority of GPL compliance engineering challenges out there.

Default Tools

The primary tool for any open source compliance engineer is Linux. This means that every active engineer needs to download, install, and set up a standard Linux distribution such as Fedora, CentOS, openSUSE, Debian, or Ubuntu. They all come with default tools pre-installed that act as the backbone of our work. Examples include *file, readelf, find, xargs, grep, dd,* and *modinfo.*

Here are free installations you can get, and where you can get them:

- Fedora: **https://getfedora.org/**
- openSUSE: **http://opensuse.org/**
- Debian: **https://www.debian.org/**
- Ubuntu: **https://www.ubuntu.com/**
- CentOS: **https://www.centos.org/**

Armijn Hemel's default installation for compliance engineering is Fedora. However, the choice of distribution matters less than the engineer being comfortable with that choice.

Binary Analysis

There are some specialized tools to help with analysis of binaries.

We mainly use the Binary Analysis Tool (BAT), but you have options, and can select the one you're most comfortable using.

The Binary Analysis Tool (BAT)

The Binary Analysis Tool makes it easy to look inside binary code, find compliance issues, and reduce uncertainty when deploying Free and Open Source Software. It is a modular framework that assists compliance and due diligence activities by using the same type of approach applied by copyright holders to discover issues in consumer electronics. BAT can open more than 30 types of compressed files, file systems, and media files; search for Linux kernel and BusyBox issues; identify dynamically linked libraries; and scan arbitrary ELF, Android Dalvik, and Java class files using a database with information extracted from source code to find out what software is inside. BAT is available for free under the Apache license so that everyone can use, study, share, and improve it.

- BAT direct download: **https://github.com/armijnhemel/binaryanalysis**
- BAT user guide: **https://github.com/armijnhemel/binaryanalysis/blob/master/doc/bat-manual.pdf**

binwalk

Another tool for analysing firmware is binwalk — an easy to use tool for analyzing, reverse engineering, and extracting firmware images.

- binwalk download: **https://github.com/devttys0/binwalk**
- binwalk user guide: **https://github.com/devttys0/binwalk/wiki**

Source Code Analysis

Our focus is on how to address binary code and the distribution of physical devices. This does not mean that tools to address

source code are unavailable. Indeed, every good compliance engineer has at least one such tool on hand to assist with confirming that source code licenses are what they expect. The best place to get started is usually with FOSSology, a free license scanner that examines source code archives and lets you know what licenses they appear to be under.

FOSSology

FOSSology (**https://www.fossology.org**) is both a compliance software system and a toolkit. As a toolkit, it allows you to run license, copyright, and export control scans from the command line. As a system, it provides a database and web UI to give you a compliance workflow. With one click, you can generate an SPDX file or a README with the copyright notices from your software. FOSSology deduplication means that you can scan an entire distro, submit a new version, and only the changed files will get rescanned. This can be a huge timesaving tool for large projects.

- Learn more about the project here: **https://www.fossology.org**
- Find a simple 'Get Started' guide to FOSSology here: **https://www.fossology.org/get-started**

Analysis of Binary Files

A Word about Binaries

In this book, the word "binary" can mean different things. Sometimes it means a single executable, sometimes it means an object file, sometimes it means an archive of binaries, sometimes a firmware; other times it means an unknown blob of data.

Here is what these different types of binary uses all have in common:

1. They are not source code.
2. They could be built from open source code.
3. They should be analysed.

Tools for Analyzing Binaries

General Approach

Analysis of a binary can be performed using a number of methods and tools. It is generally recommended to use the Binary Analysis Tool[4] or *binwalk*[5] where possible, to quickly and simply look inside binary code without reverse analysis. You can also manually dissect a binary file, blob, or package such as firmware, but this tends to take more time, and provides little advantage in exchange for the increased complexity.

Limitations

Not every binary can be unpacked for analysis. For example, firmware might have been obfuscated through file system modifications, or it may have been encrypted. Sometimes it is possible to identify the file system modifications or to reverse the encryption, but in other cases, it is impossible.

Advanced Methods

Advanced methods of getting around obfuscation include grabbing code from a "live" or running device through soldering connections or breaking into it over a network. These techniques are out of scope of this book. In the real world, most of the time, you will not face these challenges.

4. **http://www.binaryanalysis.org**
5. **https://github.com/devttys0/binwalk**

Source Code Analysis and Rebuild

A source code archive should be inspected to do the following tasks:

1. Find possibly problematic binaries in source code archives
2. Perform a rebuild of the source code and comparing it to the original binary
3. Find incorrectly licensed code

Finding Possibly Problematic Binaries

Source code archives from chipset manufacturers and ODMs (original design manufacturer) often contain more than just source code. Inside these archives you will often find:

- Object files leftover from a previous build
- “Out of tree” Linux kernel modules in binary form
- Libraries/executables such as a root file system from a previous build
- File system images
- Linux kernel images with initial embedded ramdisks/initramfs file systems
- Other firmware images

Each of these files will be explored in more detail later.

You can easily find possibly problematic files using the *“find”* command in combination with the *“file”* and *“xargs”* commands. One easy way to do this is to run *“file”* for every file and redirect the output to a result file. You can then inspect this result file at your convenience. Here is an example command to get you started:

```
$ find /path/to/source/code -type f -print0 |
xargs --null file > /path/to/result/file
```

You should search for:

- **ELF files**
  Pay attention to the architecture. If an unexpected architecture is shown, such as MIPS, but your deployed device is ARM, then the binary can probably be removed.

- **PE32 and PE32+ files**
  These are Windows binaries and usually have no place in a source code release related to an embedded Linux system. The exception is if they are related to ActiveX plugins.

- **Linux kernel boot images**
  If these are present either as a compressed file or as part of a U-Boot boot image, they can almost always be removed, because they were almost certainly compiled using a different configuration file. Including them would lead to potentially introducing licensing requirements unrelated to the target device. If such images cannot be removed because they are needed by the build process, it indicates that the source code is not complete. The shortcut is to search for “vmlinux,” “vmlinuz,” or similar files.

- **MacOS X files**
  Like Windows files, these have no place in a source code release of an embedded Linux system, except if they are related to software that would be served to Apple MacOS X machines by the device. The shortcut to search for is “Mach-O.” These files are often present in the prebuilt toolchain sources from Android as distributed by Google.

Object Files
Object files (extension “.o”) can frequently be found in source code releases. They are usually not problematic from a compliance

perspective, because the corresponding source code is typically also present. However, they make analysis more complex, because they mean more files to look at. You can address this quickly and cleanly by checking if the corresponding extension ".c" or ".cc" files are present and, if so, using "make clean" to remove the object files.

If there is no source code, you may have a compliance challenge. There are situations where the object files are needed to complete the build process and should not be removed. One example is object files that are part of a proprietary program that is statically linked with LGPL licensed code and which need to be relinked. Another example is when object files are part of a proprietary program that is statically linked with GPL licensed code and where compliance may be difficult or impossible depending on your legal jurisdiction and your legal counsel's interpretation of the GPL license.

"Out of tree" Linux Kernel Modules

Some devices contain components unsupported by the default Linux kernel. These need extra drivers to function correctly, and such drivers are often implemented as Linux kernel modules. A few examples include WiFi drivers, camera drivers, or firewalling modules. These extensions provide two common challenges:

1. Many drivers are provided prebuilt by vendors to ease integration issues. The license of these drivers should be carefully checked to ensure they are compatible with the license of the Linux kernel source.
2. When extra driver source code is available, it may not be integrated correctly in the source code tree. Incorrect integration in the build system often leads to missing source code in the final delivery.

Libraries/Executables

There are often library or executable binaries in the source code tree. This occurs when:

1. The source code tree was not properly cleaned up after building. This often occurs because someone forgot to run "make clean" after building binaries from the source code.
2. The binaries are in a "template" or "skeleton" directory used as the blueprint to build a firmware. The directory structure and the binaries in the template directory are copied first, and other files are added to it during the build.

If these binaries contain any open source licensed code and do not match the source code prepared for distribution, they should be removed. A wrong version number, a different configuration, or any other alteration from the source code could introduce unintended violations.

The key question when considering this matter is: "Is the source code 'complete and corresponding' to the built binary code and free of any extra binary or source code elements?"

File System Images

Sometimes entire file system images are included in the source code tree. For example, many Android source code trees contain file system images called "boot.img" and "system.img." If these contain any open source software without corresponding source code, this could lead to unwanted violations. These images can typically be removed without any problem.

Other Firmware Images

There have been instances where complete firmware for devices unrelated to the device being brought to market have been found in the source code archive. These often contain different software versions, packages, and sometimes even architectures to the current device. As such, these firmware are an unwanted and unnecessary source of license violations, and can typically be removed without any problem.

Performing a Rebuild

The most effective way to see if the source code is complete and corresponding is to rebuild the software and compare it to the original (binary) software. If they are identical — or nearly identical — then it is a good indication the source code is likely correct.

Perfect(ish) Rebuilds

In some cases, paths and time stamps are incorporated into a binary file. This makes it very difficult, if not impossible, to do an exact rebuild. Therefore, “close enough” means that the only differences should be in timestamps, filename paths recorded in the binary, and similar items.

Requirements

For a rebuild, it is important to have the following information:

1. A description of the build environment.
2. Full build instructions.

Goals

A rebuild has two key goals:

1. Verify that the build works.
2. Verify the results.

Describing the Build Environment

To successfully compare rebuilt binaries with original binaries, the build environment has to be described as accurately as possible. This description should include:

- The name and version of the Linux distribution or operating system that needs to be installed (example: Fedora 7, 32 bit, or Ubuntu LTS 12.04, 64 bit).

- The name and version of any packages to install if they are not installed in a default installation.
- Any modifications that need to be made to the default system, such as:
    - Symbolic links that need to be created.
    - Directories that need to be created.
    - Permissions that need to be changed.[6]
    - Files that need to exist.
    - Specific users that need to be created.
    - Environment variables that need to be set.[7]

If the build environment is different from the original, even to the extent of using a different compiler or different compiler options, it could have a big impact on the generated code. This in turn makes it a lot more difficult to compare the binary files to verify whether the source code appears to be "complete and corresponding" to the original binary.

Supplier/Client Roles

The requirement for accurate re-creation of build environments leads to a simple dynamic for providing information.

- If you are a supplier needing to provide build environment information to a client, you should be as detailed as possible.
- If you are a client needing to have build environment information, you should ask your supplier to be as detailed as possible.

6. These include items like executable bits, read/write permission, and ownership permissions.
7. These include PATH, CLASSPATH, and similar.

Rebuild Instructions

The build instructions should clearly explain the exact steps taken to rebuild a binary. This includes:

- The exact commands needed to rebuild the binary or firmware.
- The expected results such as, for example, where binaries can be found after a rebuild.

In an ideal situation, you could give the instructions to a random engineer, who would then be able to perform a nearly perfect rebuild without any problems.

Verifying the Instructions

In the real world, you might expect that people doing a rebuild for enforcement purposes will stop at the first hurdle they encounter. You should not assume that people can (or want to) fix these issues; therefore, the rebuild instructions should be complete, tested, and foolproof.

It is best to perform a rebuild on a clean physical or virtual machine, using the exact instructions that were provided. This is because undocumented modifications frequently exist on development machines such as the one on which the original build was completed. If possible, task another engineer — one without extensive knowledge of the project — to do the rebuild and to document any problems encountered. Adjust the instructions as necessary to ensure clarity and build success.

Verifying the Results

After the rebuild, you need to verify the results. Make sure that the right results are being examined — nothing cached from a previous build. Two methods for accomplishing this are:

1. The checksum of the binaries.
2. The content of the binaries.

The Checksum of the Binaries

A cryptographic checksum or hash can be computed for the contents of the file. If the file that was rebuilt has the same hash as the original binary, then the files are identical. The tools for this are *"md5sum"* for MD5 hashes and *"sha256sum"* for SHA256 hashes. The following commands will compute the hashes for two binaries and print the results. If the results are the same except for the path, then the files are identical:

```
$ md5sum /path/to/original/binary /path/to/new/binary
```

```
$ sha256sum /path/to/original/binary /path/to/new/binary
```

It should be noted that it is best to run these commands on the individual binaries in a firmware (like *"smbd"* or *"iptables"*) and not on the whole freshly built firmware. This is because the checksums for firmware might never be the same. Some binaries like BusyBox or the Linux kernel will return a different checksum every time, because by default they include timestamps internally.

The Content of the Binaries

In many cases, the checksums of an original binary and a rebuilt binary will not be identical because the paths of source code files and timestamps are included. These often differ with each build, even if an environment is carefully set up.

Not to worry. The following steps can be taken to see if a rebuilt binary is close enough to the original binary:

1. Check the file size.
2. Compare the contents of the file.

Checking the File Size

The file size of the rebuilt binary should be very close to the original binary. If there is a big difference, first check if one of the binaries is "stripped" (no debugging symbols present) and the other is not stripped. If this is the case, strip the other binary too, using the "strip" command (this tool should be included in the toolchain). If the difference is still significant, then the binaries are likely not the same.

Comparing the Contents of the File

The "strings" command can be used to extract human readable strings from a binary. This can provide a lot of useful information.

The most important point is that the differences between binaries that were built identically will be very minor. For the most part, you can expect differences to be constrained to timestamps, filenames, and directory names.

This means that you can compare the contents of a file with a simple three-step process:

1. Rebuild the binary.
2. Extract contents using "strings."
3. Compare the results to the original binary.

The following commands will help you extract the contents of the files:

```
$ strings /path/to/old/binary >
/tmp/strings.old

$ strings /path/to/new/binary >
/tmp/strings.new

$ diff -u /tmp/strings.old
/tmp/strings.new | less
```

If the differences uncovered are limited to timestamps and path names, then it is almost certain that the two binaries are in fact identical.

Finding Incorrectly Licensed Code

Modifications to open source licensed programs may not be licensed correctly. One classic example is when changes to the Linux kernel made by a chipset manufacturer either lack license statements or contain a statement that is not compatible with the GPL.

There have been enforcement cases focused on relicensing Linux kernel driver code added by chipset manufacturers. The following steps should be taken to avoid such situations:

1. Find incorrectly licensed files.
2. Find out who introduced the incorrectly licensed files.
3. Find out if the files are actually needed.
4. Find a version under an acceptable license.
5. Seek permission to change the licenses.
6. Rewrite the software.

Finding Incorrectly Licensed Files

Using license scanners, it is possible to find out the license of source code files. There are many license scanners. Some of them are proprietary (e.g., Black Duck Protex, Palamida, Whitesource, Protecode, FOSSID, and FOSSA); others are open source (FOSSology, Scancode). None of the license scanners do a perfect job, because license scanning is difficult to do correctly, and there are many source code files that use non-standard license headers, or have no license text in a header at all.

For code you wrote yourself, there are ways to make license identification easier for license scanners. One example is to use SPDX short identifiers. SPDX is a simple, standard way of describing package contents and has seen adoption across the industry for its management of licensing descriptions. You can learn more about SPDX at **https://spdx.org/**.

Finding Who Introduced Incorrectly Licensed Files

After discovering incorrectly licensed files, it is important to understand who introduced them. It usually boils down to one of two sources:

1. The upstream project.
2. The ODM/chipset manufacturer.

If you identify a situation where open source code does not have a correct license statement, many upstream projects will appreciate having this pointed out. Fixing the issues at the source provides better information for everyone. Please note that in the case of the Linux kernel, there are many files that do not have a correct license statement. However, existing files have been in the Linux kernel for many years and are not considered a problem. Your concern is to find new rather than known files. A method to quickly filter the known files is described here: **https://lwn.net/Articles/552758**

Sometimes people or companies have a list of "trusted" upstream projects. They regard all code originating from that project to be ready for use without further review. What projects will be trusted and for what reasons is a decision that will differ per person or organization. An illustrative rule of thumb is that the Linux kernel obtained directly from kernel.org may be trusted but a random kernel fork on GitHub may not.

Finding Out If Files Are Actually Needed

If the problematic files were not introduced by the upstream project (whether the upstream project is to be trusted is a separate topic for discussion) then it might be wise to remove the files. As an example, if a file for a Microsoft Windows driver is present in the Linux kernel source code tree, it can be safely removed.

Similarly, if a file is not used in the final binary or during the build process, it can be safely removed. One way to check is by removing it from the source code tree and rebuilding the binary again (after cleaning up build artifacts from previous builds, of course, and

doing a fully clean build). If the resulting binary is identical (or "close enough"), then the file can safely be removed.

Finding a Version of a Driver Under an Acceptable License

In the embedded Linux industry, many mistakes have been made in the past regarding incorrectly licensed source code files, particularly for drivers. Some vendors have already relicensed newer versions of the driver under acceptable licenses. However, ODMs frequently still ship old driver versions, either because they are unaware of the updates or because they don't want to test if the new driver works properly with their components.

Seeking Permission to Change the Licenses

Changing a license is usually the most difficult solution, but sometimes the only option left, apart from rewriting the software. If a file is used and does not have the correct license, you should ask the copyright owner to relicense the file. Some copyright owners might not be willing to, but other manufacturers or developers might not object (see the previous section).

CHAPTER 2:

Common Pitfalls

“We demand rigidly defined areas of doubt and uncertainty!”

DOUGLAS ADAMS

Pitfall #1: Toolchain

One component that is essential for doing a rebuild is the toolchain, which consists of the compiler, assembler/linker/et cetera, and a C library. For embedded Linux systems, the compiler is almost always GCC (although LLVM is starting to be used), the assembler/linker comes from GNU binutils, and the C library is glibc or uClibc (both LGPL licensed) or musl (MIT licensed) on "regular" embedded Linux, and bionic on Android systems (although glibc is frequently used on Android too for add on programs). Although LLVM and musl are being used increasingly in embedded systems, they are still the exception.

The toolchain is often found to be not compliant. A common scenario is that a toolchain with GCC and GNU binutils is provided in binary-only form, without the source code or the offer for the source code. Although it is possible to use the provided binary toolchain to rebuild the binaries, it is not the correct approach. The GCC compiler and GNU binutils are released under GPL v2 or v3, depending on the version. Their source code, or a written offer for the source code, should be included with the binary. When glibc or uClibc is used, there is an additional reason: Parts of the (prebuilt) toolchain (from the C library) are sometimes copied from the toolchain into a firmware image. This means that the sources and configuration to rebuild the C library need to be provided too (as per the LGPL license conditions). The quickest way to fulfill the requirements is by having the complete toolchain sources.

Another consideration is that for embedded Linux, the toolchain is a necessary component in rebuilding the binary. Embedded Linux devices use different CPUs than regular PCs. While regular PCs use Intel or AMD chips based on the x86 or x86-64 architectures, the embedded devices are often built using ARM, MIPS, or PowerPC chips (although other architectures can be found too). The binaries

for these platforms are generated by a so-called “cross-compiler” that runs on a regular PC but outputs code for a different platform such as MIPS or ARM. Building a working cross-compiler is a non-trivial task; without the sources and the exact description how to rebuild the cross-compiler (either using a script or the manual instructions), it will be very difficult to recreate the correct setup to perform a rebuild.

Pitfall #2: Android and Embedded Devices

Systems that are either running Android or that borrow heavily from Android, may have a few common pitfalls.

Android prebuilt tools

The standard Android software development kit as shipped by Google comes with a large number of tools that are prebuilt for various platforms, such as Linux, Darwin, and Microsoft Windows, and even Linux kernel images for QEMU. Many of these tools are licensed under GPL or LGPL, such as GCC and binutils, cmake, gdb, and many others. These files can easily be identified by looking for directories that contain “prebuilt”:

```
$ find -d /path/to/android/sdk | grep prebuilt
```

These directories often contain a variety of prebuilt tools or even Linux kernel images that may be without obviously placed corresponding source code or written offer. Frequently there is a file called “PREBUILT” in the directory that also contains the binaries. This file points to source code and sometimes also contains more detailed build instructions. As an example (from an earlier version of Android, for the ccache tool):

> The objects in this prebuilt directory can be rebuilt using the source archive
>
> ccache-2.4-android-20070905.tar.gz
>
> hosted at <http://android.kernel.org/pub/>.

It should be noted that these particular instructions may not be good enough to result in GPL compliance for chipset manufacturers, ODMs, and their downstream recipients, for a number of reasons:

1. This method does not produce a valid written offer, according to GPLv2 section 3b. While they arguably cover the originator of the code by the "equivalent access" clause in GPLv2 section 3 (because they distribute the source code only online), the instructions do not extend to the chipset manufacturers, the ODMs, and their downstream recipients.
2. As of the publication date of this book, it should be noted that one commonly referred to location for Android source, called android.kernel.org, has been offline since September 2011. This means that the relevant source code can no longer be found at this particular location, though it may be found at other URLs.

Having these prebuilt components in the source code archive without the corresponding source code can present a compliance risk.

One solution is to also include the source code for these prebuilt components. Another solution is to remove the components if they are not needed to do a rebuild (e.g., in most cases, it makes sense to remove the binaries for MS Windows and Darwin), or to replace the prebuilt components, if they are actually needed for the build, with instructions on how to fetch the prebuilt components from Android's Git server. You should make sure that the exact same version as the prebuilt versions is fetched; otherwise the build might fail or it might be difficult to compare binaries (see "Performing a Rebuild"). It should be noted that for some components, such as the toolchain, it might

still be necessary to provide sources, in case glibc or uClibc has been used and shipped on the device or in the firmware.

Missing/Incorrect License Files

Android's build system generates a NOTICES.html.gz file that is displayed by default in the "legal" tab on a phone or tablet. This file is generated by a script that looks for files that indicate the license status, called "NOTICE."

For some tools and programs in Android, you may find missing license identifiers or have used the wrong license text (in case of the Linux kernel). These omissions were fixed in the most recent versions of Android (Android 6, possibly later versions of 5), but many older versions (including several versions of Android 5) do not have these fixes. Chipset manufacturers and ODMs typically have not applied the existing fixes because they were not informed by Google. The omission of these texts has been raised in enforcement cases.

The most common omissions and errors are with:

- iproute2 — missing license reference in the NOTICES file
- iptables — missing license reference in the NOTICES file
- Linux kernel — sometimes wrong version of the license in the NOTICES file (Linux kernel 2.6-specific)

Fixing these issues is not hard at all, and patches are readily available, as described below.

iptables

The notices and license files for iptables are missing in older versions of Android. Google fixed the bug in the following Git commit:

https://android.googlesource.com/platform/external/iptables/+/b6da12d1a9020e2819f3c449244801a285659f81

iproute2
The notices and license files for iproute2 are missing in older versions of Android. Google fixed the bug in the following Git commit: **https://android.googlesource.com/platform/external/iproute2/+/5aa4845c8ef3ea0371955a2ba5f7baf7ed4e2df4**

Linux kernel
The Linux kernel license file is sometimes wrong, because Google used the license text of a prebuilt Linux kernel (2.6.x), which has a slightly different license text than later versions. This was not a problem for Android versions using the 2.6.x kernel, but when the Linux kernel moved to 3.x and later 4.x, the license text was not entirely correct. Google fixed this in late 2015:

https://android.googlesource.com/platform/build/+/b463fcde80f5615b3fe6891b8b78c010ec8cd37b

Pitfall #3: “Out of tree” Linux Kernel Modules

Many vendors ship Linux kernel modules that add functionality that is not provided by the standard Linux kernel, or that is not yet present in the version shipped for the device, such as support for certain hardware, firewalling modules, new security features, etc. Linux kernel modules for the 2.6 and later releases have the “.ko” extension. Kernel modules for the 2.4 and older kernel often have the extension “.o” (but that could also be used for regular object files).

For these so-called “out of tree” kernel modules, it is important to find out which license they are under and if there is complete and corresponding source code.

Linux kernel modules can contain several fields that detail things such as the author and a description, but also a license field. An

example from Linux kernel 4.5. (file “drivers/clk/clk-pwm.c”) looks like this:

```
MODULE_AUTHOR(“Philipp Zabel
<p.zabel@pengutronix.de>”);

MODULE_DESCRIPTION(“PWM clock driver”);

MODULE_LICENSE(“GPL”);
```

These fields are then included in the kernel module binary when it is built. They can later be extracted from the binary either by using the “modinfo” tool (preferred) or manually (as recent versions of “modinfo” no longer support the format for Linux kernel 2.4 or older). The important fields to look at are the author field and the license field. The author field usually indicates the copyright holders of the specific code being reviewed. The license field could indicate the possible license of a file. This field is quite important, as certain pieces of functionality in the Linux kernel can only be used by modules that have explicitly declared that they are GPL-licensed.

It also happens that kernel modules are distributed in a firmware or source code archive, but they are not used, because they are never loaded by the operating system, either because there are no programs to load them, or because the operating system does not allow it (it may be a different version or even a completely different architecture). Finding out if a module is used is outside of the scope of this book.

Extracting License and Author Fields from a Kernel Module

The license field can be extracted from a Linux kernel module using the modinfo tool:

```
$ modinfo -l /path/to/kernel/module
```

Similarly, the author field can be extracted using:

```
$ modinfo -a /path/to/kernel/module
```

Note: Recent versions of the modinfo program no longer have support for kernel modules for Linux kernel 2.4.X and earlier (using the ".o" extension). For those modules, you can use the "strings" command instead:

```
$ strings /path/to/kernel/module | grep -i license
```

Extracting Version and Architecture Fields from a Kernel Module

Similarly to the license field, the version and architecture information can easily be retrieved from a Linux kernel module:

```
$ modinfo /path/to/kernel/module | grep ^vermagic
```

For 2.4.X and earlier, the version can be extracted as follows (because the modinfo tool on recent Linux distributions no longer can process modules for 2.4 or earlier):

```
$ strings /path/to/kernel/module | grep kernel_
version
```

The architecture can be retrieved using different means, such as the "file" command:

```
$ file /path/to/kernel/module
```

Pitfall #4: Rescue Mode/Install Mode Systems

Quite a few embedded Linux devices have a special mode that is used only for system recovery (rescue mode) or when installing a new firmware. This is done by booting a different Linux kernel from a different partition on the flash memory. These rescue partitions are often not updated when a new firmware is released and are simply forgotten. However, for compliance, it is very important to have the complete and corresponding source code for all of the different Linux systems that are used on a device or while updating a firmware.

These rescue partitions tend to have different contents than other partitions. It is very common to see that both the rescue partition and the normal one have a copy of BusyBox, but with a different size and set of tools integrated. This means that they were built with different configurations. It is also not uncommon to see that a different Linux kernel (older version, known to work) has been used, but that the source code releases have the source code only for the Linux kernel that is booted in normal operation. There have also been instances where the C library was different (uClibc in the rescue partition, glibc in the normal partition, and so on).

It also happens that a separate version of Linux is booted only to perform the installation of a new firmware, and that version is embedded in the firmware update itself and is not on the device. Or, it could be that there are three different instances of Linux involved in one single firmware update: a temporary Linux booted when performing the update, a different version when writing a rescue partition, as well as a third version for the regular partition. It is important to look at everything that is installed or used at installation time: The device and the firmware update are both important.

Pitfall #5: Bootloader

One overlooked component in compliance engineering is the bootloader. A few commonly used bootloaders on embedded Linux systems (e.g., U-Boot and Redboot) are GPL-licensed. The reason they are overlooked is because they come preflashed on the boards or chips, and ODMs frequently do not touch this component at all. Many times the bootloader is also not included in a firmware update, but the firmware update overwrites only parts of the flash chip in a device and leaves the bootloader alone. However, if the bootloader is GPL-licensed, source code for the bootloader should be delivered as well.

If possible, perform your analysis on the firmware update as shipped to customers (see “Pitfall #4”) as well as on a dump of the flash contents of the actual device, unless the firmware update is actually the same as the flash dump.

If the bootloader is not included, it might be necessary to extract the contents of the bootloader from the device. This is outside of the scope of this book.

Pitfall #6: Missing Build System

There are build systems that separate the sources of packages and the description of how to build them. Some build systems have a directory called “download” or “dl” that contains the sources, while the Makefiles, configurations, and patches are in a separate directory. Some companies will publish the contents of the directory only with the sources but not the build system.

This is wrong for a few reasons:

1. Any patches that might have been applied are now not included, meaning that the source code is incomplete.

2. Makefiles and other build scripts often contain configuration options (environment variables, compiler options, etc.) that influence how a package is built. Without this information, the binary cannot be rebuilt successfully, or at least not (near-)identically.

Pitfall #7: Incorrect or Missing BusyBox Configuration Files

An often-encountered problem is that BusyBox cannot be rebuilt in such a way that it corresponds to the binary or binaries in a firmware. The BusyBox program is very modular; functionality can be added or removed by editing a configuration file (usually using a special configuration program). This configuration file is read during build time and determines which functionalities (called “applets”) will be included in the BusyBox binary. The configuration file is therefore a very necessary part of the “complete and corresponding source code,” and a missing or incorrect configuration file for BusyBox has been enforced many times.

Missing BusyBox configuration file

Source code release archives often contain only the source code for BusyBox, but not the configuration, because the build system is not included (see Pitfall #6).

An easy check for this is to look for a file called “.config” in the top-level source code file of the BusyBox source code tree. If it cannot be found, it might be in a separate directory in the build system, if present.

Incorrect BusyBox configuration file

Another problem encountered at times is that the BusyBox

configuration file or files are incorrect: The original binary and the rebuilt binary have different sets of applets. In many of these cases, the chipset manufacturer or ODM cannot find or recreate the correct configuration file.

It is possible (with tools contained in the Binary Analysis Tool) to re-create a BusyBox configuration file that could be used as the basis of re-creating the real configuration file. However, this solution should be used only as a last resort.

Multiple different BusyBox binaries, one configuration file

Quite often there are multiple BusyBox binaries included in a firmware, each with a different configuration. A common example is a rescue system (see Pitfall #4) that contains a minimal version of BusyBox, with the full system containing a BusyBox instance with much more functionality. The source code archive should include the configurations for all Busybox instances that are in use on the device, but frequently, the configuration for only one of the instances of BusyBox is present.

Pitfall #8: Incorrect or Missing Linux Kernel Configuration Files

Very similarly to Pitfall #7, the configuration file for the Linux kernel is often missing or incorrect.

Finding which Linux kernel configuration was used

Finding out what configuration was used to build a Linux kernel binary is not always trivial, and sometimes a rebuild and comparison (as described earlier in this book) will be necessary. Sometimes the kernel configuration will be included in the Linux kernel binary

as a bzip2 compressed file. This happens if the “CONFIG_IKCONFIG” option was enabled during the kernel build. In that case, it is easy to find the kernel configuration that was actually used (for example, by unpacking the binary with the Binary Analysis Tool and then looking for the configuration). If the configuration was not stored in the Linux kernel image, then your only option to verify whether a kernel configuration is 100% correct is a rebuild and compare.

Missing Linux kernel configuration file

Source code release archives often contain only the source code for the Linux kernel, but not the configuration. Depending on the setup, the Linux kernel configuration could be in various locations. One common location is a file “.config” in the root of the Linux kernel source tree (generated by the Linux kernel configuration commands like “make config” or “make menuconfig”). Another location is in the “arch” subdirectory. By default, the Linux kernel source code tree contains many configuration files, and vendors tend to put the configurations there. For example, “arch/arm/configs/bcm2835_defconfig” contains the configuration for a particular Broadcom board. Which configuration file to use is set by the build scripts. A third option is that the configuration file is kept outside of the Linux kernel archive, with the build scripts, and is first copied to the Linux kernel source code tree during the build. If the build system is missing (see Pitfall #6) and the configuration file is not included in the Linux kernel archive, then the Linux kernel source code will not be complete and corresponding.

Multiple Linux kernel binaries, one configuration file

Quite often there are multiple Linux kernel binaries included in a firmware, each with a different configuration. A common example is a rescue system (see Pitfall #4) which contains a minimal version of the Linux kernel, with the full system containing a Linux kernel

instance with much more functionality. The source code archive, however, may have the configuration for only one of the two versions (or three, or even more).

Incorrect Linux kernel configuration file

It also happens that the Linux kernel configuration file may simply be not correct and that the appropriate Linux kernel binary cannot be compiled because functionality has been added or removed in the configuration.

Pitfall #9: Not Including the Version Number in Firmware and Source Code Archive Filenames

One very common mistake is that firmwares and corresponding source code archives often do not have the version name (and the device name) in the filename, but use a generic name, such as "GPL.zip," for various devices and versions of source code. This makes it very easy to make mistakes and deliver the wrong files, which might lead to the impression that you are out of compliance.

The solution is to use, or demand that suppliers use, a naming convention that would include:

- Device name/model number (or multiple, if the files are identical)
- Firmware revision number
- Revision level

For example: A device called AB-123 with firmware 1.2.3.4 would have a firmware filename "FW_AB-123_1.2.3.4.bin" and a source code archive filename "GPL_AB-123_1.2.3.4-0.bin."

Using naming conventions like these will make it a lot easier to locate the right files, avoid making mistakes, and spot errors on download sites.

CHAPTER 3:

Scenarios for Releasing Software

"'Did I do anything wrong today,' he said, 'or has the world always been like this and I've been too wrapped up in myself to notice?'"

DOUGLAS ADAMS

Scenario #1: Software On A Device/Offline Distribution

Software is distributed onto a device, for example, flashed onto a device. There are two ways to comply with the license:

1. Deliver the complete and corresponding source code with the device, for example on a CD, DVD, or other medium. In some cases it might also be possible to include it on the device itself (given enough flash or storage space), but what is important is that the user should be able to retrieve the source code as well.
2. Add a written offer, valid for at least three years (for GPLv2; for GPLv3 this period might be longer, depending on how long the device is supported), to any third party for the source code.

The benefit of the first method is that there are no further obligations, as all the necessary information (license texts, copyright information, and so on) are included in the source code archive, as long as it is complete and corresponding. The drawback is that often the source is not complete and corresponding, and it is difficult to correct any mistakes discovered after creating the CD, DVD, or image. One situation that often arises is that just before shipping, a new firmware is flashed onto the device, but the CD/DVD distributed alongside it contains the GPL source code for a prior firmware revision. This happens because there was not enough time to create a new CD/DVD. Issues like this can potentially be addressed with a good Over-The-Air update process. See Flowchart #4 on page 61 for one high-level example.

With the written offer it is easier to correct any mistakes later on, but there are a few drawbacks: It is necessary to keep an

infrastructure to fulfill the written offer (someone has to be responsible to create a source code CD/DVD and ship it) and there are also some legal gray areas pertaining to whether or not license statements and copyright statements should be delivered with the binary. Not everyone agrees on this, but there are increasingly more copyright holders who insist on having the copyright notices from the source code delivered with the binary. As previously stated, when delivering source code with the product this requirement has already been fulfilled, but when shipping a written offer, this is not the case. Extracting copyright notices from source code is awkward. FOSSology is one tool that can help, but additional checking may be required.

It might be best to choose shipping the source code with the device, with an extra optional written offer with information on how to ask for the source code. This way, it can be argued (if there is an error with the source code shipped) that the preferred way always has been the written offer and that the source code was mainly to provide license texts and copyright statements. If the source code is not complete and corresponding it might be incorrect, but not materially incorrect.

Please note that this scenario is independent of how firmware on a device is updated, which the next scenarios will dig into.

Scenario #2: Manual Download From Website

When providing a firmware update on a website (independent of how the compliance for an initial device delivery is done — see the previous scenario for that) there are often three possible choices to make:

1. Provide the source code with the firmware update in a single archive. This would fulfill the requirements of GPLv2, section 3a.[8]
2. Include a written offer, as per GPLv2, section 3b.[9]
3. Provide a separate download next to the firmware, which would count as "equivalent access."

The same benefits and drawbacks apply as in Scenario #1, although it is much easier to correct mistakes (in choices 1 and 3) than when shipping a physical device, as it is easier to change the download.

Keep in mind that when offering source code online for download, there is always the possibility that a new website will be created later. If the web team is not aware that source code downloads were a requirement, a company may fall out of compliance after a new website is rolled out. It is important to note this download-requirement in the website change procedures and process.

Scenario #3: Automatic/Over The Air Updates

For automatic updates and "over the air" (OTA) updates, shipping source code is almost impossible. This scenario for delivering binaries was likely not available when the GPLv2 was drafted in 1991. Automatic updates surgically update a few programs or files, and the size of the source code delivered to a device would dwarf the size of the executable (for example, the Linux kernel). Also, on many devices that can receive OTA updates it would be hard to access the downloaded source code or do anything with

8. See **https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html**

9. Ibid.

it. Another problem is that there might simply not be enough space available on the device, and distribution of the source code might fail.

For OTA updates, the practical solution is to include a written offer. Please note that this is independent of how the device was originally shipped (source code, or written offer).

How and where copyright notices and license texts should be delivered can be a real challenge, especially if the device does not offer any way to interact with the user (display, web interface, and so on). If that is the case it might be best to point the user to a website and offer the information there instead of on the device (which would be futile as the user has no access to it) and point out that the license text is from 1991 and that OTA delivery models were not available then.

Scenario #4: Field Engineer Applied Updates

In some cases (for example, industrial automation) it is common that a firmware update is applied by a field engineer of the manufacturer, or by a support firm. It is recommended to have the field engineer bring a CD/DVD with the source code and hand it over after the firmware update has been completed.

CHAPTER 4:

Scenarios for Buying Software

“Everything starts somewhere, although many physicists disagree.”

TERRY PRATCHETT

Context

When procuring devices or components from a third party to sell as part of your product, it is important to understand that the company offering the product in the market is responsible for the license compliance of the product. Often when a compliance problem arises, there is a conflict about who is responsible and who should bear the costs. This can lead to suppliers and downstream OEMs pointing fingers and blaming the other party. However, in the end, the company delivering the product to the market will likely be found responsible for ensuring compliance with the license.

It is also true that *fixing* compliance issues will be more expensive than *preventing* compliance issues. If the whole supply chain works together from the start to prevent these issues, costs for fixing issues can generally be kept low.

That being said, changing how current supply chains work is a multi-year effort. In this section, we will explore a few solutions that look into lowering risks for all parties in the supply chain.

Scenario #1: Supply Chain Solutions For SoC Vendors

In most supply chains, the System on Chip (SoC) vendors have the biggest impact: They choose or build the software development kit and build a reference implementation that ODMs subsequently modify. If the SoC vendors get compliance right, then it is much easier for downstream recipients to comply with the license as well. Here are a few solutions that are worth considering:

1. Pick a standard SDK instead of building your own SDK.

2. Actively participate in the upstream software projects, and try to use as much "vanilla" software as possible.
3. Have a third party check/audit license compliance.

Pick a Standard SDK

It is highly recommended that SoC vendors use a standard SDK instead of creating their own. From a license compliance perspective, the most compelling reason for this is that these standard SDKs often have mechanisms for easier license compliance built directly into the system. Plus, they have been reviewed by many people already and are supported by a much larger group of people than an in-house developed solution would be. Examples of standard SDKs are:

- OpenWrt (and its offshoot LEDE)
- Yocto
- buildroot
- Android

The additional benefit is that the so-called "scripts to control compilation and installation" are all available in the SDK.

Actively Participate in Upstream Projects

Many people are triggered to complain about license compliance status if they cannot make things work using standard software. Actively participating in upstream projects and supporting hardware in "vanilla" projects has two benefits:

1. People will be less likely to complain, since they can simply work around any issues. Of course, this is not an excuse to not take any other measures for license compliance.
2. People will be more inclined to send a "friendly ping" to point out any issues than to complain loudly. Of course, participating

engineers should know what to answer, and how to interact with communities at large, and be in sync with other departments and players (for example, the legal department).

Some organizations are more than willing to help SoC vendors to merge their code upstream and help them become effective open source citizens. For the Linux kernel, there is, for example, the Long Term Support Initiative (LTSI) run by The Linux Foundation. For vendors in the ARM ecosystem, there is also Linaro.

Third Party Audits

It might be useful to let a third party check for any issues in new SDKs before they are shipped downstream, and to incorporate any findings into the work process. The OpenChain Project (**http://openchainproject.org**) is a good start. Another hands-on, practical approach to consider is the OSADL License Compliance Audit (**http://www.osadl.org**).

Scenario #2: Supply Chain Solutions for ODMs

The Original Design Manufacturers (ODMs) are often in between the System on Chip (SoC) vendors and the companies that deliver products into the market. They are also the first point of contact for companies if something is wrong. As referenced earlier in this book, the technical choices made by the SoC vendor has one of the biggest impacts on any open source compliance situation. An ODM can help pick the right SoC vendor or make sure that any (possible) mistakes from SoC vendors do not impact them. Examples of actions that an ODM can take are:

1. Work with SoC vendors that use standard SDKs or whose chipsets are well supported in standard SDKs (such as

OpenWrt) so the SDK you use is a standard SDK and not a SoC-specific SDK.

2. Work with SoC vendors that use a SDK that has been certified by the OSADL license compliance audit or similar.
3. Use contracts to push compliance damages to the SoC vendor.
4. Actively participate in upstream projects.
5. Let a third party audit a reference design.

Scenario #3: Supply Chain Solutions for Others

When procuring devices, there are a few things that can be done to reduce compliance risk.

First and foremost, you can engage with other open source stakeholders who are addressing similar challenges. A great place to start is the OpenChain Project. OpenChain focuses on identifying common best practices in compliance programs that should be applied across a supply chain for efficient and effective compliance with open source licenses. It provides (free of charge) comprehensive specification, conformance, and curriculum material suitable for small, medium, and large enterprises. You can learn more at **www.openchainproject.org.**

Here are some direct measures you may wish to consider:

1. Select only ODMs that use a solution from a chipset vendor that uses a standard SDK or have replaced the SoC SDK with a standard SDK.
2. Explicitly ask for use of a certified/audited SDK.

3. Use contracts to push damages upstream.
4. Audit a sample before purchasing, or contract a third party to do this.

Doing an audit before a purchase is an interesting strategy to see how much risk a certain device will bring with it. This is, however, easier said than done. Very few ODMs are prepared to hand out samples; even when there are firm orders for tens of thousands of devices there are usually only a few samples (one or two) available for testing. Before a firm order has been placed, it will be very unlikely that an ODM will give out samples, fearing that the device will be taken to a cheaper competitor to be cloned.

One solution to this would be to let a third party do an audit and send a status report only to the purchaser, who can then use it to calculate its risk.

CHAPTER 5:

Building a FOSS Code Center

"A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools."

DOUGLAS ADAMS

Context

It is common for companies to have a “FOSS code center” or “open source download center” on their website, where users can download the (open source) source code for products.

While this delivery system is a great convenience to users, it should be noted that the GPLv2 license was drafted in a time when broadband connections were extremely rare, and therefore the focus of the license is mostly on delivering source code with the product or on a physical medium like a CD. In 2017, the situation has rapidly changed: Broadband is widely available (although it should be noted that there are still many places where broadband is rare, or even non-existent) and FOSS code centers fulfill all practical needs (namely, access to source code) for most people.

Although a FOSS code center could fulfill the requirements of the GPLv2 license in certain cases (like firmware updates, where providing source code on the same location could then fall under “equivalent access” from GPLv2, section 3), it should not be seen as a replacement for the traditional compliance methods, but instead as a convenience to users.

There are no strict rules when it comes to designing a FOSS code center, as every website is different. However, there are a few best practices which are mostly non-technical, that are worth following.

“FOSS Code Center” as a Requirement

Many companies have a “FOSS code center” on their website. Some of the companies have added this as an afterthought, and not as a requirement for the design of the website. And whenever there is a new or updated website, the designers often are not

aware of the requirement for a FOSS code center, and it vanishes, with the company possibly falling out of compliance.

Keep Firmware and Source Code Together

Sometimes FOSS code centers are not in the same place as firmware downloads. It might be useful to keep them together so it is clear which source code belongs to which firmware. It is recommended to at least make a reference to the right place in a (separate) FOSS code center from the firmware download page; otherwise users might not know where to find the corresponding source code and falsely assume that there is no source code, which would, of course, cause frustration.

CHAPTER 6:

Tracking Tasks and Processes

"Engineers like to solve problems. If there are no problems handily available, they will create their own problems."

SCOTT ADAMS

Checklists

A quotation worth bearing in mind with regard to all aspects of compliance engineering comes from C. Northcote Parkinson:

"Work expands so as to fill the time available for its completion."

When used properly, checklists can be a useful way to manage your GPL compliance tasks in a quick, consistent, and effective manner. When used incorrectly, they can be too general to cover the work at hand or become overwhelming catalog requiring endless review. A happy medium is the goal. What constitutes a happy medium really depends on your organizational size.

For a small organization, a general compliance checklist could be as simple as the following:

General Compliance Checklist

Step #1: Ongoing Compliance Tasks

- ☐ Discover all FOSS early in the procurement/development cycle.
- ☐ Review and approve all FOSS packages used.
- ☐ Verify the information necessary to satisfy FOSS obligations.
- ☐ Review and approve any outbound contributions to FOSS projects.

Step #2: Support Requirements

- ☐ Ensure adequate compliance staffing and designate clear lines of responsibility.
- ☐ Adapt existing business processes to support the FOSS compliance program.
- ☐ Make training on the organization's FOSS policy available to everyone.

☐ Track progress of all compliance activities.

This checklist is from the *OpenChain Curriculum slides*,[10] which is based, in turn, on the Linux Foundation Open Compliance Program *Self-Assessment Compliance Checklist.*[11]

You might elect to have more specific checklists to address specific compliance goals. For example, the concept of addressing the "complete and corresponding" source code for distribution is arguably the first and most useful area for which to have a specific checklist. One way of approaching this would be to create an exhaustive list of all the steps possible and necessary. Another way would be to cover the "core" of the issue and leave details to trained personnel or sub-checklists as needed. What follows is an example of the latter.

Checklist For Rebuilding Product X

This checklist is part of the check-system for ensuring that "complete and corresponding" source code is available when distributing products containing GPL code.

Step #1
☐ Is a complete description of the build environment provided?[12]

Step #2
☐ Is a list of rebuild steps provided?

Step #3
☐ Has a rebuild been successfully completed on a clean machine?

10. **https://www.openchainproject.org/curriculum**
11. **https://www.linuxfoundation.org/projects/opencompliance/self-assessment-compliance-checklist**
12. This should include package versions and any similar information critical to ensuring compliance.

Step #4
☐ Have the rebuild results been verified?

Step #5
☐ Have any uncertainties been escalated to the Open Source support team?

Plenty of options exist for more comprehensive checklists. A great place to start is the Open Compliance Program Self-Assessment Compliance Checklist. This comprehensive list, which may be required for a larger organization, is, like the material above, free of charge and freely available, so you can explore what is best to meet your requirements.

Flowcharts

One great way to approach the challenge of managing your GPL compliance process is to use flowcharts. Like checklists, flowcharts can be modest or complex. For various reasons, larger organizations tend towards weighty versions of both. You do not need to. C. Northcote Parkinson once again has a quote that is applicable in our context:

"Expansion means complexity and complexity decay."

Bearing this in mind, let us take the first steps towards introducing simple, clear flowcharts for GPL compliance.

Professionals in open source governance have been sharing suggestions for various flowcharts for years. For example, this photo shows Jeremiah Foster of the GENIVI Alliance adding thoughts to a distribution flowchart at an interactive session led by Armijn Hemel at the Open Compliance Summit 2015 in Japan.

In this section, we explore some example flowcharts that could support practical GPL compliance with minimum fuss and complexity.

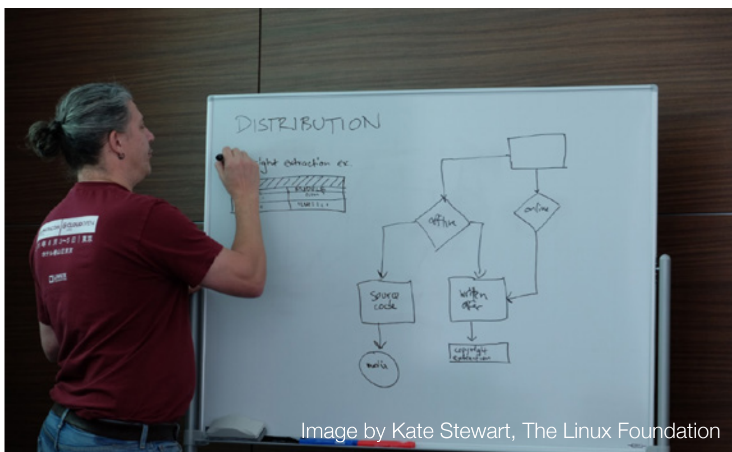


Image by Kate Stewart, The Linux Foundation

To get the ball rolling, we will highlight an older flowchart loosely based on work by Arnoud Engelfriet from his tenure as the open source counsel at Royal Philips Electronics. He kindly shared his approach with Free Software Foundation Europe (FSFE) e.V., a charitable organization focused on promoting Free Software/Open Source, which then released a modified version to the world through inclusion in public presentations.

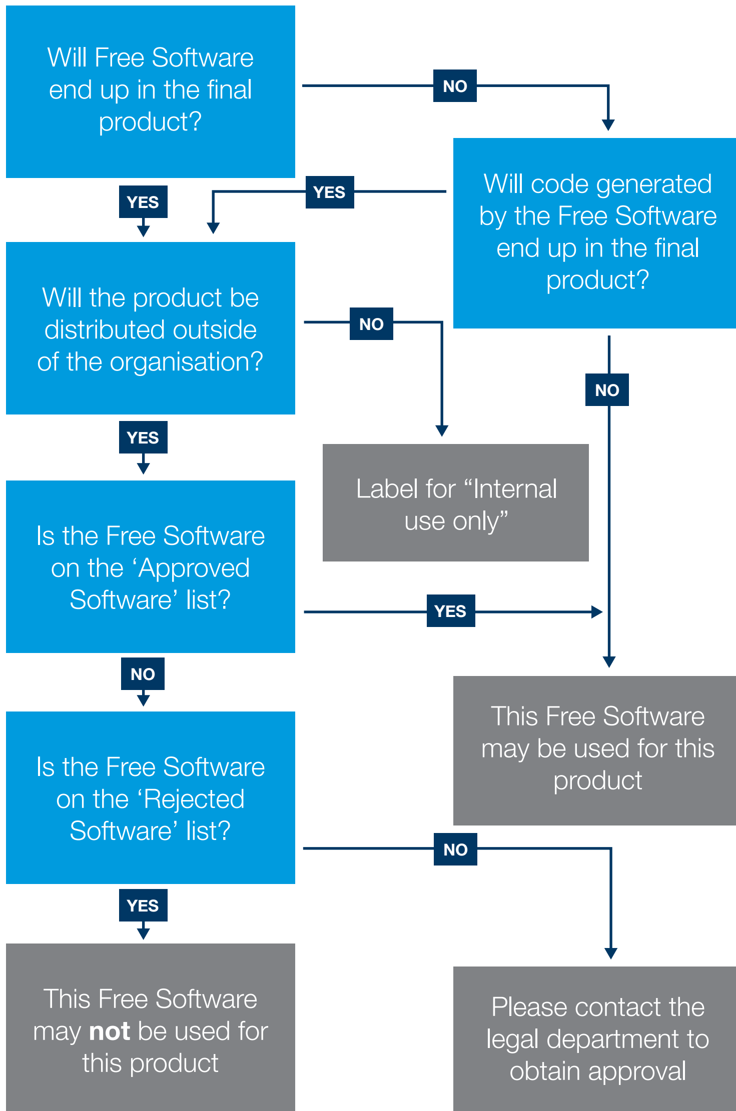
Flowchart #0 is a great example of what we need to assist our work as compliance engineers. It is short, clear, and applicable to physically distributed devices. It pre-assumes relatively light and responsive infrastructure to support it:

1. An "approved" list of open source licenses.
2. A "rejected" list of open source licenses.
3. A contact in the legal department to deal with any edge cases.

Flowchart #0 was released into the public domain in January 2008 via FSFE's legal department — at that time run by Shane Coughlan — with the intention of helping to spread knowledge to companies of all sizes wondering how best to approach process management around open source.

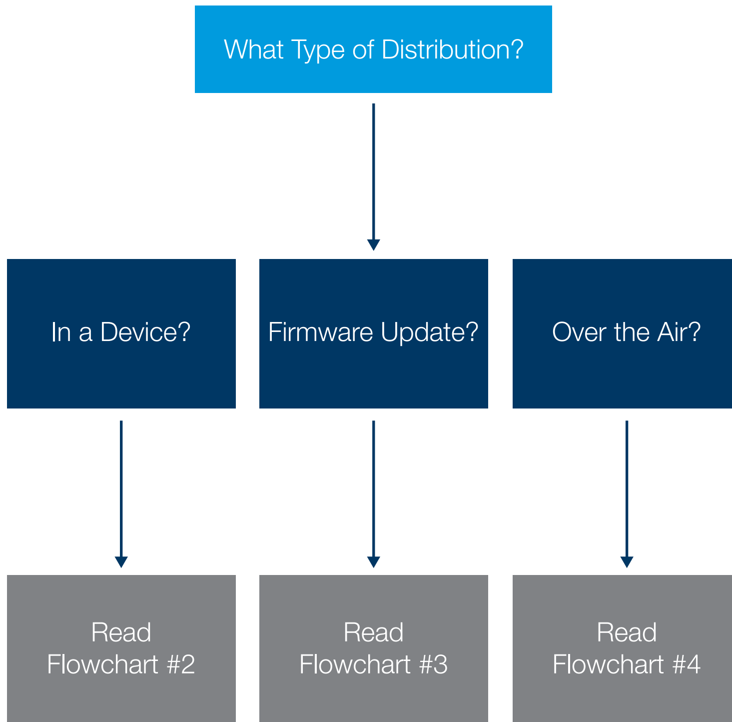
After reviewing **Flowchart #0**, you will find a series of interconnected flowcharts prepared by Armijn Hemel to show a more detailed process for managing compliance matters. These are not intended to be a panacea for open source compliance flowcharting, but they do provide a substantial starting point that can be easily adopted, adapted, and deployed by organizations of all sizes.

Flowchart #0: General Approval Flowchart

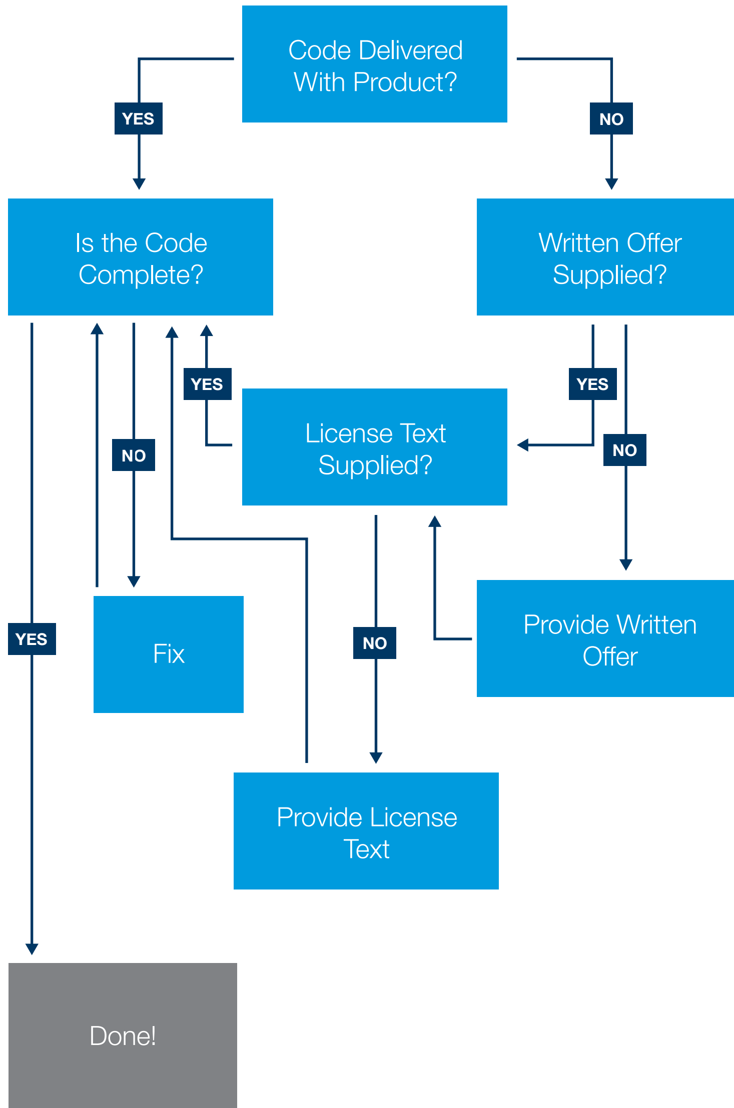


With thanks to Royal Philips Electronics

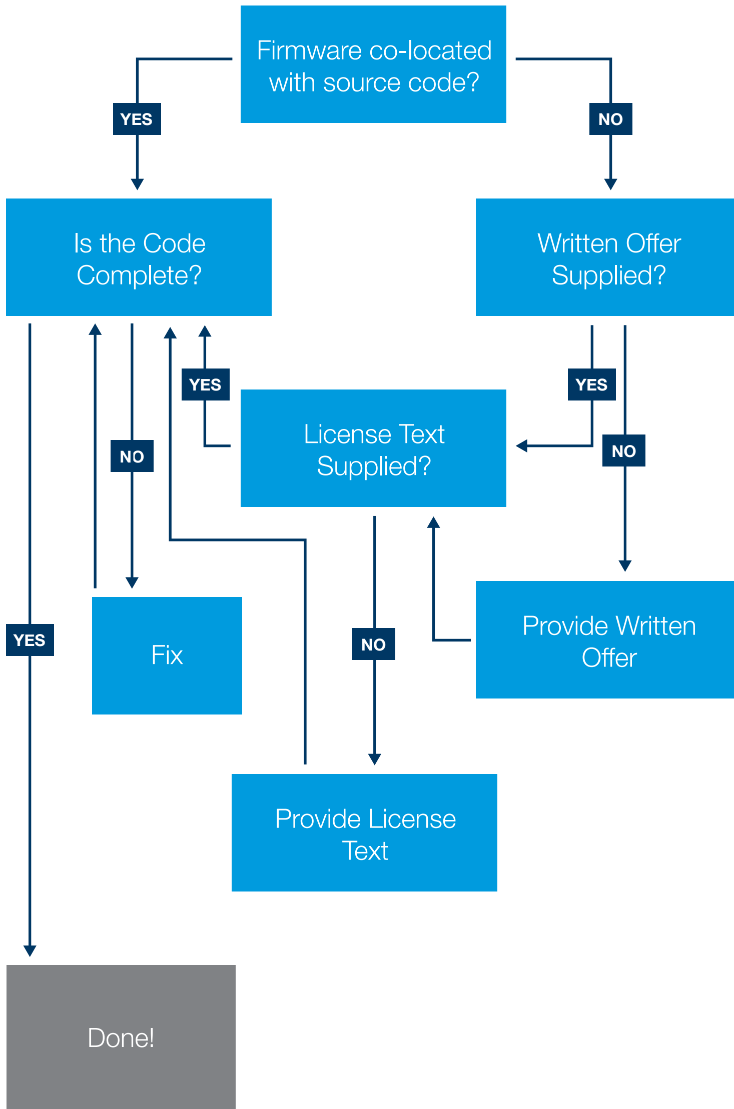
Flowchart #1: How Do I Distribute?



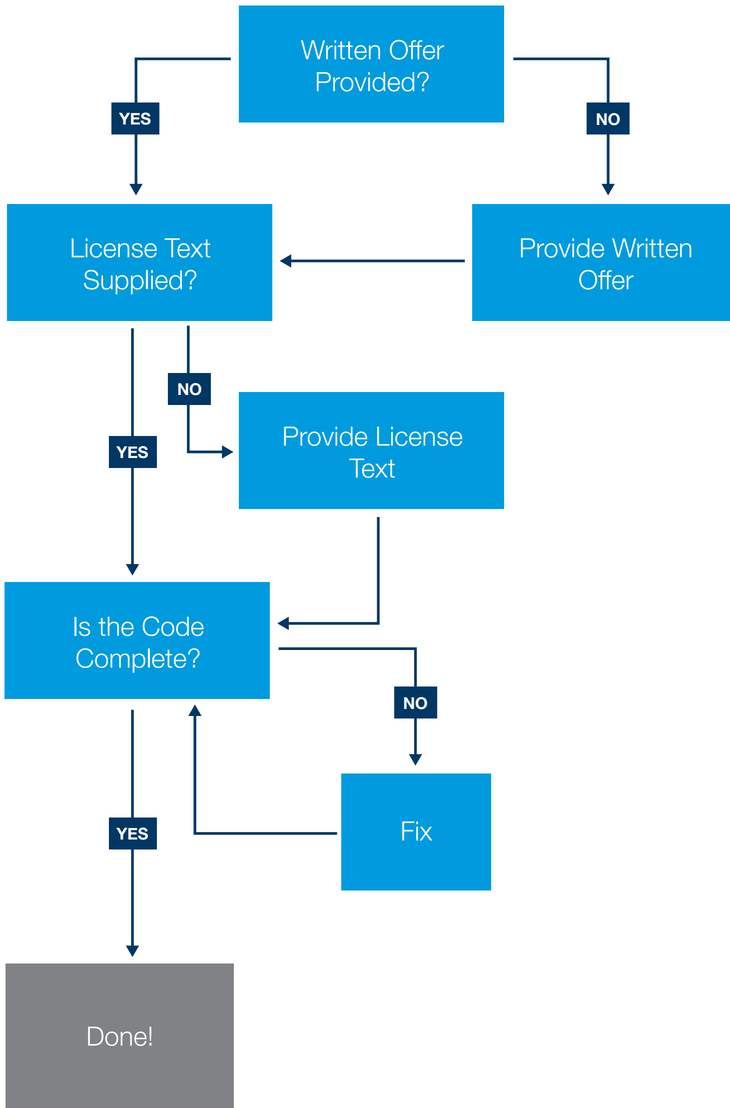
Flowchart #2: Offline Distribution



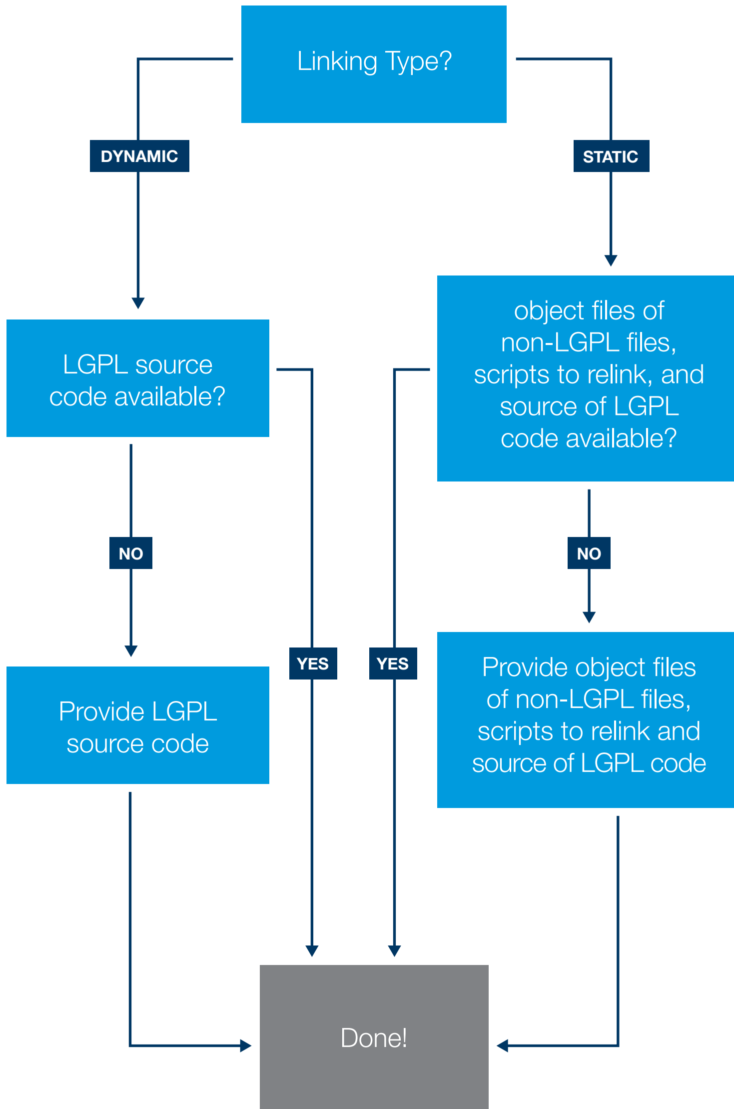
Flowchart #3: Firmware Updates



Flowchart #4: Over The Air



Flowchart #5: LGPL Code



Appendices

Appendix 1: The Open Compliance Program

Other Open Source Compliance Publications

Free and Open Source Software Compliance: The Basics You Must Know[13]
This paper provides basic discussion on the changing business environment moving to a multi-source development model, the objectives of compliance, the benefits resulting from having a successful compliance program, and much more.

Author: Ibrahim Haddad (Ph.D.), The Linux Foundation

Free and Open Source Software Compliance: Who Does What[14]
Ever since companies started integrating FOSS in their products, there has been the need to ensure compliance with applicable FOSS licenses. Different companies have used various ways to structure their teams responsible for fulfilling this function. Other companies have opted for cross-functional teams that include a dedicated Open Source Compliance Officer who has access to various individuals and teams that contribute to the compliance effort without being part of a centralized team. In this paper, we examine the latter model of a FOSS compliance team and discuss the roles and responsibilities of individuals and teams involved in the compliance process.

Author: Ibrahim Haddad (Ph.D.), The Linux Foundation

13. **https://www.linux.com/publications/free-and-open-source-software-compliance-basics-you-must-know**
14. **https://www.linux.com/publications/free-and-open-source-software-compliance-who-does-what**

Establishing Free and Open Source Software Compliance Programs: Challenges and Solutions[15]
This white paper focuses on the practical aspects of ensuring free and open source software (FOSS) compliance in the enterprise.

Author: Ibrahim Haddad (Ph.D.), The Linux Foundation

Keys to Managing a FOSS Compliance Program[16]
This paper examines the managerial practices needed to plan, coordinate, and control a successful compliance program.

Author: Philip Koltun (Ph.D.), The Linux Foundation

A Five-Step Compliance Process for FOSS Identification and Review[17]
This white paper focuses on the various practical aspects of ensuring free and open source software (FOSS) compliance in the enterprise. This paper provides a sample five-step compliance process for FOSS identification and review. The paper focuses on using and integrating FOSS with proprietary and third party source code in a commercial product.

Author: Ibrahim Haddad (Ph.D.), The Linux Foundation

Achieving FOSS Compliance in the Enterprise[18]
This white paper focuses on the various practical aspects of ensuring free and open source software (FOSS) compliance in

15. **https://www.linux.com/publications/establishing-free-and-open-source-software-compliance-programs-challenges-and-solutions**
16. **https://www.linux.com/publications/keys-managing-foss-compliance-program**
17. **https://www.linux.com/publications/five-step-compliance-process-foss-identification-and-review**
18. **https://www.linux.com/publications/achieving-foss-compliance-enterprise**

the enterprise. This paper examines a sample end-to-end compliance process.

Author: Ibrahim Haddad (Ph.D.), The Linux Foundation

FOSS Compliance Practices for Supplied Software[19]
This white paper examines compliance practices necessary when software supplied by a third party vendor is brought into the code baseline of a product to be distributed externally. The paper discusses requirements a company should impose upon its suppliers to disclose FOSS in their deliverables and to provide what's needed to achieve compliance. It also discusses steps a company should take to review and validate the FOSS disclosures made by its suppliers. In addition to those topics, the white paper addresses measures a company can undertake to assess its suppliers' compliance capabilities.

Author: Philip Koltun (Ph.D.), The Linux Foundation

Compliance Templates

Self-Assessment Checklist
The Linux Foundation has compiled this extensive checklist of compliance practices found in industry-leading compliance programs. Companies can use this checklist as a confidential internal tool to assess their progress in implementing a rigorous compliance process and to help them prioritize process-improvement efforts. The Self-Assessment Checklist is constructed using at least two concepts from well-established models of process maturity, such as the Software Engineering Institute's Capability Maturity Model:

- A distinction should be made between process goals and the practices implemented to achieve those goals. The

19. **https://www.linux.com/publications/foss-compliance-practices-supplied-software**

compliance checklist explicitly recognizes valid alternative practices that may be used to achieve a particular goal.

- Process adoption progresses from initial process definition through institutionalization to a state of controlled process management. The goal of a compliance process, as with any process, is to achieve consistent and expected business results from its use. A checklist of recommended practices should prompt companies to assess the extent to which they've institutionalized compliance actions and to which those actions produce needed business results.

Compliance practices included in the checklist will improve the effectiveness of compliance programs as well as deliver tangible benefits relative to the cost of those practices. A process failure modes effects analysis (FMEA) approach has been used to identify the ways in which a compliance process can fail and practices to prevent those process failures.

Author: The Linux Foundation

Generic FOSS Policy

Companies using FOSS often create a company-wide policy to ensure that all staff is informed of how to use FOSS (especially in products), to maximize the impact and benefit of using FOSS, and to ensure that any technical, legal, or business risks resulting from that usage are properly mitigated. This document is a new free resource available from the Linux Foundation under the Open Compliance Program. It offers a generic FOSS policy that companies can use as starting point in creating their own FOSS policy. It provides a template policy that focuses on governing FOSS usage in externally distributed products and that can be customized to the company's specific needs.

Author: The Linux Foundation

Template for an Approval Request Form for the Use of Free and Open Source Software
This document is one of the free resources made available by The Linux Foundation Open Compliance Program. It offers a template for the Approval Request Form used by developers to request approval to use Free and Open Source Software (FOSS) in a commercial product. The company's Open Source Review Board (OSRB) then reviews the submission and determines approval. In most cases, the submission, review, and approval of such requests is managed via an online submission system that is part of the company's FOSS compliance management process.

Author: The Linux Foundation

Appendix 2: Compliance Standards

Education and training build a base of knowledgeable resources to guide your open source journey. However, these tools alone will not solve efficiency issues if everyone implements compliance processes differently. The Linux Foundation projects enable the industry to develop compliance standards for companies and entire supply chains to exchange compliance data in a consistent way.

- OpenChain identifies common best practices in open source compliance that should be applied as a standard across a supply chain.[20]
- SPDX Specifications enable projects and organizations to communicate accurate summaries of the licensing and copyright information in software deliverables.[21]
- SPDX License List is a curated list of commonly found licenses that can be referenced by the use of a standardized

20. **https://www.openchainproject.org**
21. **https://spdx.org/specifications**

short identifier per license. For each short identifier, the list contains the full name for each license, vetted license text, other basic information, and a canonical permanent URL for each license and exception.[22]

- SPDX Meta Tags enable the use of the standardized short identifier in source code to efficiently refer to a license without having to redundantly reproduce the full license.[23]

Appendix 3: Professional Networks

You're not alone in your open source compliance journey. Many of our members have found it beneficial to participate in the projects we host, simply to access the network of experts participating in the projects. In addition, The Linux Foundation hosts professional networks to help compliance professionals find each other and collaborate on ways to improve compliance practices, tooling, and processes.

- TODO Group[24]
- Compliance Directory[25]
- OpenChain[26]

22. **https://spdx.org/license-list**
23. **http://wiki.spdx.org/view/Technical_Team/SPDX_Meta_Tags**
24. **http://todogroup.org/**
25. **https://compliance.linuxfoundation.org/references/open-compliance-directory**
26. **https://www.openchainproject.org**

Appendix 4: Tools and Infrastructure

To achieve higher levels of scale and reduce the overhead costs of compliance, companies have contributed to creating open source tools and infrastructure to achieve compliance at a lower cost, increasing not only cross-organization efficiency but also integration of compliance with product development.

- **FOSSology** scans codebases, identifies licenses in use, creates machine readable license lists, and enables automatic notice-file creation.[27]
- **FOSS Bar Code Tracker** simplifies the way FOSS components are tracked and reported in commercial products. The tool allows companies to easily generate custom QR codes for each product containing FOSS. The QR codes contain important information about the FOSS stack contained in a product, such as component names, version numbers, license information, and links to download the source code, among other details.[28]
- **SPDX Tools** are tools for validating, transforming, reading, and writing SPDX format files. SPDX Tools also provides links to community-maintained and commercially available tools that support SPDX.[29]
- **Dependency Checker** is capable of identifying code combinations at the dynamic and static link level. The tool also offers a license policy framework that enables FOSS compliance officers to define combinations of licenses and linkage methods that are to be flagged if found as a result of running the tool.[30]

27. **https://www.fossology.org/**
28. **http://git.linuxfoundation.org/?p=foss-barcode.git;a=summary**
29. **http://spdx.org/tools**
30. **http://git.linuxfoundation.org/dep-checker.git**

- **The Code Janitor** provides linguistic review capabilities to make sure developers did not leave comments in the source code.[31]

31. **http://git.linuxfoundation.org/janitor.git**

“So long, and thanks for all the fish.”

DOUGLAS ADAMS

Practical GPL Compliance is a guide for startups, small businesses, and engineers tasked with shipping products that contain GNU General Public License Version 2 (GPLv2) code. It is directly applicable to consumer electronics, drones, IoT, or automotive devices based on generic Linux or Android code-bases. It provides simple instructions, checklists, and flowcharts that empower compliance teams to work with open source as efficiently as possible.

The goal of this guide is to demystify GPL Compliance Engineering and to make it possible for every stakeholder in open source to quickly address common issues. The focus is on solving real world challenges in a manner that lays the foundation for addressing open source compliance more broadly in organizations of all sizes.

"Today it takes more than simple legal advice to be compliant with the requirements of open source licenses. This guide is based on the long-term experience of the authors and represents a unique collection of practical information about how to solve the technical challenges compliance engineers are facing."
- Dr. Till Jaeger, Partner at JBB Rechtsanwälte

"Shane Coughlan and Armijn Hemel have created a wonderful resource for anyone working with open source software being shipped in a device. They describe in detail how to ensure compliance with the various software licenses and communities in which we all rely on."
- Greg Kroah-Hartman, Linux Kernel Maintainer and Fellow at The Linux Foundation

"Armijn Hemel is the world's foremost expert on the technical aspects of GPL compliance. This book is an indispensable starting point for all compliance engineers."
- Professor Eben Moglen, President and Executive Director at Software Freedom Law Center

"A practical guide with hands-on advice that condenses the significant experience of pioneers in GPL compliance who actually made some of the tools of the trade. Given their longstanding and recognized experience in compliance this is one of the most keenly awaited texts by compliance advisors and in-house officers. I am glad we now have it and that it exceeds my high expectations."
- Carlo Piana, Founder at Array Law and General Counsel at Free Software Foundation Europe e.V.

THE LINUX FOUNDATION