# RAUC Update & Device Management Manual Mickledore

**PHYTEC Messtechnik GmbH**

**2026 年 02 月 13 日**

# Contents

| RAUC Update & Device Management Manual | |
|---|---|
| 文档标题 | RAUC Update & Device Management Manual Mickledore |
| 文档类型 | RAUC 升级和设备管理手册 |
| Last Modified | 2025-01-14 |
| 母文档 | RAUC Update & Device Management Manual |

| 适用 BSP | BSP 发布类型 | BSP 发布日期 | BSP 状态 |
|---|---|---|---|
| BSP-Yocto-NXP-i.MX93-PD24.1.0 | 大版本 | 05.02.2024 | 已发布 |
| BSP-Yocto-NXP-i.MX93-PD24.1.1 | 小更新 | 08.05.2024 | 已发布 |

本手册适用 Yocto 版本 Mickledore

PHYTEC 的 Yocto 发行版 Ampliphy(前身是 Yogurt) 支持 RAUC 机制。RAUC 管理设备的固件升级过程，包括升级 kernel，设备树和根文件系统。对 emmc 设备而言，它也包括升级 bootloader。

本手册描述了如何在 PHYTEC 平台上实现以及使用 RAUC 机制。需要注意的是，不同核心板使用不同的 bootloader 和烧写存储设备，这导致了不同平台 RAUC 处理过程的差异性。请确保阅读对应硬件平台的本手册相关部分。

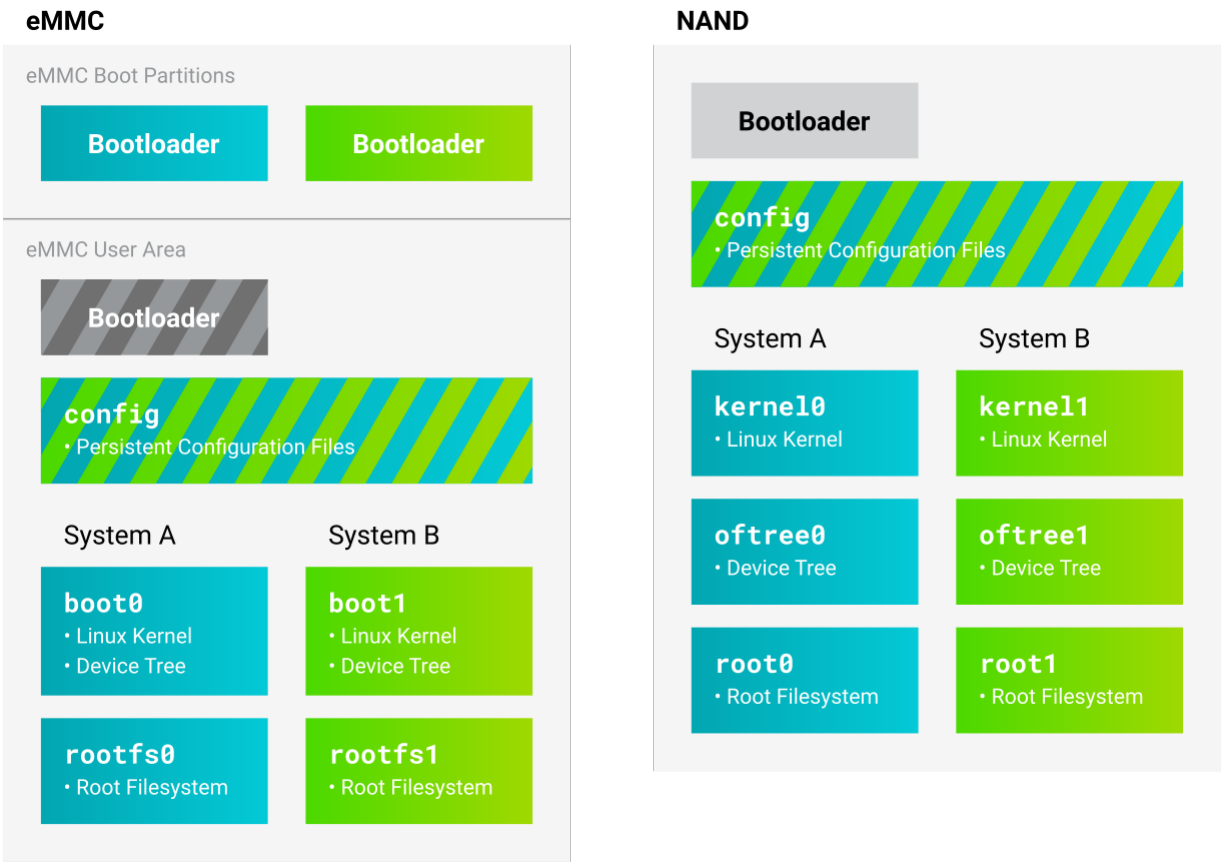
**备注**

本手册使用了一些特定 machine 的路径和变量。请在执行任何命令之前确保您使用了正确的 machine 和设备名称

CHAPTER 1

# 系统配置

RAUC 机制可以用在 eMMC 和 NAND flash 存储的设备上，使用 distro `ampliphy-rauc` 或者 `ampliphy-vendor-rauc`，这两个 distro 上 RAUC 机制默认被使能，无需额外配置。RAUC 可以应用在不同的升级场景下。例如：我们需要 BSP 去实现系统镜像的 A/B 备份（包括 eMMC 上的 bootloader）。请注意：RAUC 会产生一个 config 分区存储永久性的配置数据，这些数据不会被系统升级所覆盖



## 1.1 RAUC 配置范例

分区布局在 `/etc/rauc/system.conf` 文件中定义。例如：下面是配置 eMMC 烧写设备的 i.MX 8M Mini 的分区布局配置：

列表 1: /etc/rauc/system.conf

```
[system]
compatible=phyboard-polis-imx8mm-4
bootloader=uboot
mountprefix=/mnt/rauc

[handlers]
pre-install=/usr/lib/rauc/rauc-pre-install.sh
post-install=/usr/lib/rauc/rauc-post-install.sh

[keyring]
path=mainca-rsa.crt.pem

[slot.bootloader.0]
device=/dev/mmcblk2
type=boot-emmc

# System A
[slot.rootfs.0]
device=/dev/mmcblk2p5
type=ext4
bootname=system0

[slot.boot.0]
device=/dev/mmcblk2p1
type=vfat
parent=rootfs.0

# System B
[slot.rootfs.1]
device=/dev/mmcblk2p6
type=ext4
bootname=system1

[slot.boot.1]
device=/dev/mmcblk2p2
type=vfat
parent=rootfs.1
```

请注意，slots 中 device 随不同的 machine 配置而有所不同

**警告**

RAUC 升级使用 OPENSSL 证书验证镜像的有效性。BSP 中包含一个可以在开发过程中使用的临时证书。在生产系统中，强烈建议使用您重新创建的私钥和证书。如果您需要修改设备上存储的密钥，请参考*切换 RAUC 密钥* 以获取更多信息

CHAPTER 2

# 设计注意事项

为了避免系统被锁住，建议您使用一个硬件看门狗。如果 kernel 不启动或者是系统发生了其他灾难性的事件导致系统无法正常运行，硬件看门狗可以重启系统。默认情况下，看门狗是不使能的。如果需要使能硬件看门狗，请参考对应核心板适用的 BSP 参考手册。

其他重要的设计注意事项以及 checklist，可以在官方 RAUC 文档: https://rauc.readthedocs.io/en/latest/checklist.html 中找到。

CHAPTER 3

# 初始设置

为了使用 RAUC，烧写设备需要烧写入完整的 Linux 系统和 bootloader。推荐使用烧写工具：partup.

## 3.1 烧写存储

要给设备烧写正确的分区/卷，请使用 `ampliphy-rauc` 或者 `ampliphy-vendor-rauc` 发行版编译出的 partup 包。您可以直接使用 BSP release 中预编译的 partup 包，也可以使用 Yocto 自主编译。修改 `local.conf` 文件去修改 Distro，这样不同的 distro，会产生不同的 build 文件夹，存储编译过程中生成的对应软件包和镜像。使用 OE init 脚本初始化 build 目录：

```
host:~$ TEMPLATECONF=../meta-phytec/conf/templates/default source sources/poky/oe-init-build-env
```

把 distro 变量配置为 `ampliphy-rauc` (i.MX6, AM6x，i.MX8 mainline BSP) 或者 `ampliphy-vendor-rauc` (i.MX8,i.MX9 vendor BSP):

列表 1: build/conf/local.conf

```
DISTRO ?= "ampliphy-rauc"
```

使用该 distro 编译出的任何镜像都包含一个完整的 A/B 系统，按如下方法编译镜像：

```
host:~$ bitbake phytec-headless-image
```

生成的 partup 包存储在 `deploy-ampliphy-vendor-rauc` 目录下，例如:

```
deploy-ampliphy-vendor-rauc/images/phyboard-segin-imx93-2/phytec-headless-image-phyboard-segin-imx93-2.
↪partup
```

该 partup 包包含烧写 eMMC 所必须的所有数据和配置。Partup 可以从它的 发布页面 获取。查看其中的 README 文件以获取详细的 安装指导。Partup 工具已经安装在我们的 Ampliphy 镜像中，例如 `phytec-headless-image` ，我们可以直接使用它。

**备注**

要烧写初始的 RAUC 系统，需要先在另外一个烧写设备上烧写并启动一个非 RAUC 系统。例如，你可以从 SD 卡启动一个常规的 `ampliphy` 发行版的 `phytec-headless-image` 镜像。

### 3.1.1 eMMC

在目标设备上，从 SD 卡启动非 RAUC 系统后，拷贝使用 distro `ampliphy-rauc` 或者 `ampliphy-vendor-rauc` 编译出的 `.partup` 包到运行系统上：

```
host:~$ scp phytec-headless-image-phyboard-segin-imx93-2.partup 192.168.3.11:/root
```

然后将 partup 包安装到 eMMC 中：

```
target:~$ partup install phytec-headless-image-phyboard-segin-imx93-2.partup /dev/mmcblk0
```

现在目标设备可以启动烧写好的 A/B 系统了

### 3.1.2 NAND

**备注**

在之前的 barebox 中提供了一些脚本去初始化 NAND flash 上的 A/B 双系统：`rauc_init_nand`, `rauc_flash_nand_from_tftp` 和 `rauc_flash_nand_from_mmc`。这些脚本已经被弃用。当前建议使用 *Ampliphy* 发行系统中，Linux 环境下的 `rauc-flash-nand` 脚本

在裸 NAND flash 上，kernel，设备树和根文件系统分别单独被烧写进 flash 上。在目标设备的 Linux 环境下初始化 NAND flash 以创建正确的卷：

```
target:~$ rauc-flash-nand -k /path/to/zImage -d /path/to/oftree -r /path/to/root.ubifs
```

初始化脚本自动使用 NAND flash 上所有可用空间。NAND 设备也会通过在/proc/mtd 路径下查找 root 设备来确定。

在 i.MX6 和 i.MX6UL 设备上，使用 bbu (barebox update) 去烧写 bootloader

```
target:~$ bbu.sh -f /path/to/barebox.bin
```

A/B 备份系统现在可以从 NAND Flash 启动了

## 3.2 Bootloader

### 3.2.1 默认启动 A/B 系统

自 Yocto 版本 *hardknott* 后，启动 A/B 系统的工作大部分由 bootloader 自动完成。配备 eMMC 烧写的设备，在 BSP 编译过程中，相关设置会被写入 bootloader 环境变量中。具体来说，如果使用了发行版 `ampliphy-rauc` 或者 `ampliphy-vendor-rauc` 系统（如前所述），bootloader 会自动启动 A/B 系统并且设置相应的环境变量。

自动设定的过程也可以通过在 bootloader 中设置一个变量来实现手动设定。该过程在随后的 U-boot、barebox 章节中会被详细描述。

### 3.2.2 U-Boot

在成功启动进入 Linux 环境后，可以使用下面的命令来查看可用参数：

```
target:~$ fw_printenv
```

我们将看到该参数和其他参数一起出现在控制台输出中：

```
doraucboot=1
```

为了手动使能/关闭带 RAUC 机制 A/B 系统的自启动，将该变量设置为 `0` 或者 `1`:

```
target:~$ fw_setenv doraucboot 1
```

该参数也可以在 U-boot 中被修改。重启设备，按任意键停止自动启动，进入 bootloader 环境。查看 bootloader 环境变量。

```
u-boot=> printenv
```

并且设置它们：

```
u-boot=> setenv doraucboot 1
u-boot=> saveenv
```

### 3.2.3 Barebox

在 barebox 中，可以通过名称选定要启动的系统。要启动 A/B 系统，包括 RAUC，需要使用 `bootchooser` 。例如：要从 SD 卡启动不带 RAUC 的系统，使用 `mmc` ，如果是 NAND 设备，使用 `nand` ：

```
barebox$ nv boot.default=bootchooser
```

CHAPTER 4

# 创建 RAUC 升级包

要升级带 RAUC 的系统，需要创建 RAUC 升级包（`.raucb`）。它包含所有升级必要的脚本和镜像，以及一个 RAUC `manifest.raucm` 文件描述 RAUC 升级包的内容。BSP 中包含了可以生成 RAUC 升级包的 recipe。

要使用 Yocto 创建 RAUC 升级包，在 distro `ampliphy-rauc` or `ampliphy-vendor-rauc` 初始化的 build 目录下执行下列命令：

```
host:~$ bitbake phytec-headless-bundle
```

该命令在 “deploy/images/<MACHINE>/“ 下生成 `.raucb` 升级包文件。无需手动创建 `manifest.raucm` ，在 build 过程中会自动生成。作为参考，生成的 manifest 文件会有以下类似内容：

列表 1: manifest.raucm

```
[update]
compatible=phyboard-polis-imx8mm-3
version=r0
description=PHYTEC rauc bundle based on BSP-Yocto-FSL-i.MX8MM-PD20.1.0
build=20200624074335

[image.rootfs]
sha256=cc3f65cd1c1993951d7a39bdb7b7d723617ac46460f8b640cd8d1622ad6e4c17
size=99942000
filename=phytec-headless-image-phyboard-polis-imx8mm-3.tar.gz

[image.boot]
sha256=bafe46679af8c6292dba22b9d402e3119ef78c6f8b458bcb6993326060de3aa4
size=12410534
filename=boot.tar.gz.img
```

关于 manifest 格式的更多信息，请参考 https://rauc.readthedocs.io/en/latest/reference.html#manifest。

CHAPTER 5

# RAUC 升级

要使用 RAUC 升级系统，上述过程中生成 RAUC 升级包需要先被拷贝到核心板内存，或者在 Linux 中挂载的存储设备上。拷贝文件的一种方式是使用 `scp` ，它要求核心板文件系统中有足够的剩余空间。首先启动设备到 Linux 环境，然后通过以太网连接到 host PC.

在主机上运行：

```
host:~$ scp phytec-headless-bundle-phyboard-polis-imx8mm-3.raucb root@192.168.3.11:/tmp/
```

在设备上，rauc 升级包可以被读取：

```
target:~$ rauc info /tmp/phytec-headless-bundle-phyboard-polis-imx8mm-3.raucb
```

输出会类似于：

```
rauc-Message: 12:52:49.821: Reading bundle: /phytec-headless-bundle-phyboard-polis-imx8mm-3.raucb
rauc-Message: 12:52:49.830: Verifying bundle...
Compatible:     'phyboard-polis-imx8mm-3'
Version:        'r0'
Description:    'PHYTEC rauc bundle based on BSP-Yocto-FSL-i.MX8MM-PD20.1.0'
Build:          '20200624073212'
Hooks:          ''
2 Images:
(1)     phytec-headless-image-phyboard-polis-imx8mm-3.tar.gz
        Slotclass: rootfs
        Checksum:  342f67f7678d7af3f77710e1b68979f638c7f4d20393f6ffd0c36beff2789070
        Size:      180407809
        Hooks:
(2)     boot.tar.gz.img
        Slotclass: boot
        Checksum:  8c84465b4715cc142eca2785fea09804bd970755142c9ff57e08c791e2b71f28
        Size:      12411786
        Hooks:
0 Files
```

(续下页)

(接上页)

```
Certificate Chain:
 0 Subject: /O=PHYTEC Messtechnik GmbH/CN=PHYTEC Messtechnik GmbH Development-1
   Issuer: /O=PHYTEC Messtechnik GmbH/CN=PHYTEC Messtechnik GmbH PHYTEC BSP CA Development
   SPKI sha256:␣
↪E2:47:5F:32:05:37:04:D4:8C:48:8D:A6:74:A8:21:2E:97:41:EE:88:74:B5:F4:65:75:97:76:1D:FF:1D:7B:EE
   Not Before: Jan  1 00:00:00 1970 GMT
   Not After:  Dec 31 23:59:59 9999 GMT
 1 Subject: /O=PHYTEC Messtechnik GmbH/CN=PHYTEC Messtechnik GmbH PHYTEC BSP CA Development
   Issuer: /O=PHYTEC Messtechnik GmbH/CN=PHYTEC Messtechnik GmbH PHYTEC BSP CA Development
   SPKI sha256:␣
↪AB:5C:DB:C6:0A:ED:A4:48:B9:40:AC:B1:48:06:AA:BA:92:09:83:8C:DC:6F:E1:5F:B6:FB:0C:39:3C:3B:E6:A2
   Not Before: Jan  1 00:00:00 1970 GMT
   Not After:  Dec 31 23:59:59 9999 GMT
```

要检查系统的当前状态，运行:

```
target:~$ rauc status
```

得到类似如下输出:

```
=== System Info ===
Compatible:  phyboard-segin-imx6ul-6
Variant:
Booted from: rootfs.0 (system0)

=== Bootloader ===
Activated: rootfs.0 (system0)

=== Slot States ===
o [rootfs.1] (/dev/ubi0_6, ubifs, inactive)
        bootname: system1
        boot status: good
    [dtb.1] (/dev/ubi0_3, ubivol, inactive)
    [kernel.1] (/dev/ubi0_2, ubivol, inactive)

x [rootfs.0] (/dev/ubi0_5, ubifs, booted)
        bootname: system0
        boot status: good
    [kernel.0] (/dev/ubi0_0, ubivol, active)
    [dtb.0] (/dev/ubi0_1, ubivol, active)
```

要使用下载的升级包升级当前系统，运行:

```
target:~$ rauc install /tmp/phytec-headless-bundle-phyboard-polis-imx8mm-3.raucb
```

然后重启:

```
target:~$ reboot
```

如果升级成功，RAUC 会自动切换到新升级的系统。在重启过程中，RAUC 统计 kernel 的尝试启动次数，如果在启动次数多于系统设定值，RAUC 会切换到旧系统，并且将新系统标记为 bad。如果成功启动到 kernel，新系统标记为 good，然后使用同样的指令进行升级另外一个旧系统。在两轮成功的 `rauc install` and `reboot` 之后，两个系统都被升级了

**小技巧**

当你从 USB 存储设备升级，请确保在成功升级之后移除 USB 存储设备。如果 USB 存储设备没有被移除，每次重启后都将会从 USB 存储设备自动升级系统。其中缘由在下面的章节*Automatic Update from USB Flash Drive with RAUC* 中作出解释。

## 5.1 修改优先的启动 Slot

我们可以手动切换优先系统：

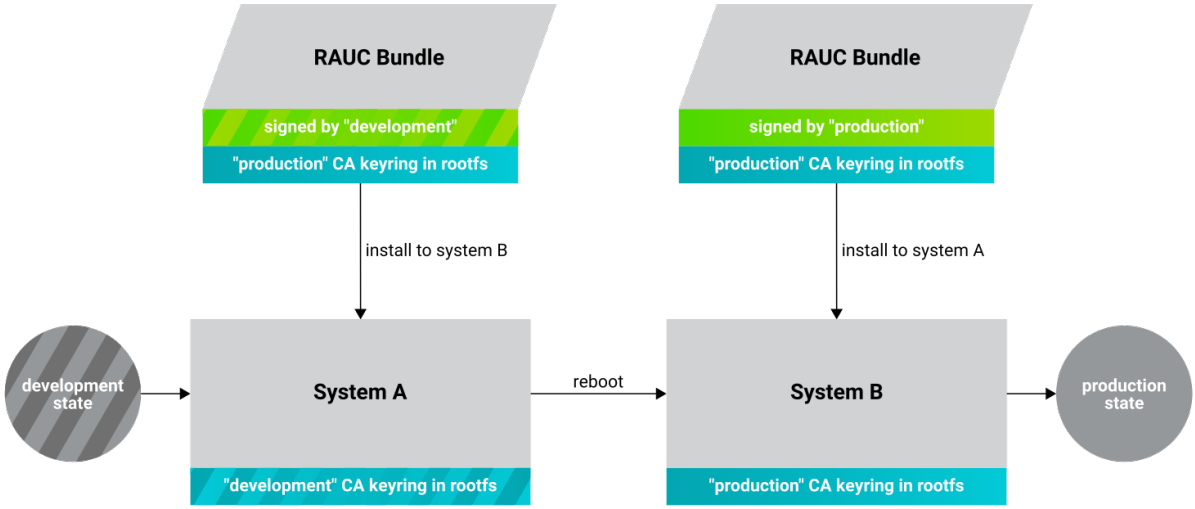
```
target:~$ rauc status mark-active other
```

重启后，设备会从另外一个系统启动。

CHAPTER 6

# 切换 RAUC 密钥

PHYTEC 的发行版包含只用于开发阶段和演示用途的密钥和证书。要在设备发布后更新到新的 PKI, RAUC 的密钥必须要修改。本章描述从开发到量产的完整处理流程。必须要记住，发布您的产品设备时使用新的量产密钥永远比过度依赖开发阶段的旧密钥要安全的多。下图展示了切换 RAUC 密钥的通用处理过程：




## 6.1 密钥切换

给您自己的 PKI 创建新的证书和密钥。请查阅我们的安全手册，里面详细描述了如何创建一个自定义 PKI。在本手册中，我们把这个新创建的 PKI 定义为” 量产”，以和现存的” 开发” 密钥区别开来。

将生成的密钥和证书放到 Yocto 工程根目录下，和 `build/` 以及 `sources/` 目录同一级。

**警告**

存储私钥的时候必须要十分小心！千万不要泄露。例如：不要将私钥存储在公共 git 仓库。否则，一些未授权的组织或者个人可能会用这写密钥创建出可以在您的设备上安装并启动的 RAUC 升级包！

现在，我们可以创建出一个在根文件系统中含有新的” 量产” 证书密钥的 RAUC 升级包，但是该升级包仍然以” 开发” 证书签名。这样升级后系统将会从一个” 开发” 系统升级成一个” 量产” 系统。首先，需要将 Yocto sources 目录下面 RAUC recipe 中安装的 `ca.cert.pem` 文件替换。在您自己的 Yocto layer 中创建一个 `rauc_%.bbappend` 文件。

列表 1: recipes-core/rauc/rauc_%.bbappend

```
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"

RAUC_KEYRING_FILE = "${CERT_PATH}/rauc-customer/ca.cert.pem"
```

然后像之前一样编译出 RAUC 升级包，只是这一次我们使用了替换的密钥:

```
host:~$ TEMPLATECONF=../meta-phytec/conf/templates/default source source/poky/oe-init-build-env
host:~$ bitbake phytec-headless-bundle  # Build the desired RAUC bundle
```

安装生成的 RAUC 升级包。目标设备现在已经将包含有” 量产” 密钥的镜像安装在与非当前运行系统的另外一个 Slot 中 (即上图中的”System B” )。我们需要重启去启动新的系统。

所有未来给” 量产” 系统的 RAUC 升级包必须用” 量产” 证书所签名。在 bundle recipe 中将密钥和证书修改为您新生成的” 量产” 密钥:

列表 2: recipes-images/bundles/customer-headless-bundle.bb

```
require phytec-base-bundle.inc

RAUC_SLOT_rootfs ?= "phytec-headless-image"

RAUC_KEY_FILE = "${CERT_PATH}/rauc-customer/private/production-1.key.pem"
RAUC_CERT_FILE = "${CERT_PATH/rauc-customer/production-1.cert.pem"

RAUC_INTERMEDIATE_CERT_FILE = ""
```

重编译 RAUC 升级包:

```
host:~$ bitbake customer-headless-bundle
```

现在 RAUC 升级包已经准备好，可以安装到您的” 量产” 目标系统中，并且已经完全从” 开发” 系统迁移。这也意味着只有使用” 量产” 证书签名的升级包才能被安装在您的目标设备上 (例如像” 开发” 升级包将无法安装到目标设备)

CHAPTER 7

# 使用举例

## 7.1 使用 RAUC 从 USB Flash 自动升级

RAUC 最为人熟知的一个用例是从 USB flash 自动升级系统。BSP 中包含了一个此用例的参考实现。我们使用标准 Linux 机制去配合 RAUC 机制。当 USB 插入的时候 kernel 通知 *udev* , 使用一个自定义的 *udev* 规则，在 USB 事件发生时触发 systemd 服务。

列表 1: 10-update-usb.rules

```
KERNEL!="sd[a-z][0-9]", GOTO="media_by_label_auto_mount_end"

# Trigger systemd service
ACTION=="add", TAG+="systemd", ENV{SYSTEMD_WANTS}="update-usb@%k.service"

# Exit
LABEL="media_by_label_auto_mount_end"
```

该服务自动挂载 USB flash 设备然后通知上层应用

列表 2: update-usb@.service

```
[Unit]
Description=usb media RAUC service
After=multi-user.target
Requires=rauc.service

[Service]
Type=oneshot
Environment=DBUS_SESSION_BUS_ADDRESS=unix:path=/run/dbus/system_bus_socket
ExecStartPre=/bin/mkdir -p /media/%I
ExecStartPre=/bin/mount -t auto /dev/%I /media/%I
ExecStart=/usr/bin/update_usb.sh %I
ExecStop=/bin/umount -l /media/%i
ExecStopPost=-/bin/rmdir /media/%I
```

在参考实现中，我们使用了 shell 脚本来简单实现应用逻辑

列表 3: update_usb.sh

```
#!/bin/sh

MOUNT=/media/$1

NUMRAUCM=$(find ${MOUNT}/*.raucb -maxdepth 0 | wc -l)

[ "$NUMRAUCM" -eq 0 ] && echo "${MOUNT}*.raucb not found" && exit
[ "$NUMRAUCM" -ne 1 ] && echo "more than one ${MOUNT}/*.raucb" && exit

rauc install $MOUNT/*.raucb
if [ "$?" -ne 0 ]; then
    echo "Failed to install RAUC bundle."
else
    echo "Update successful."
fi
exit $?
```

升级逻辑可以用 *systemd D-Bus API* 集成到应用中，无需使用命令行接口调用 RAUC。

**小技巧**

RAUC 支持 D-Bus API 接口 (详见 https://rauc.readthedocs.io/en/latest/using.html#using-the-d-bus-api).

## 7.2 安全措施：降级屏障

在第二个参考示例中，我们将会实现一个安全机制：降级屏障。当你在系统中检测到安全漏洞时，您将会修复并升级系统。这样带有新软件的系统又重新变为安全系统。如果黑客拥有旧软件的升级包，并且它有有效签名，他们仍然可能会安装旧的软件包，然后利用之前已经被修复的安全漏洞。为了防止这样的事情发生，您可以在每次升级时吊销旧的升级证书，然后创建一个新的证书。依赖于具体的环境，这种方式可能有点不容易实现。更简单的方式是用版本检测的方式只允许单向升级。

列表 4: rauc_downgrade_barrier.sh

```
#!/bin/sh

VERSION_FILE=/etc/rauc/downgrade_barrier_version
MANIFEST_FILE=${RAUC_UPDATE_SOURCE}/manifest.raucm

[ ! -f ${VERSION_FILE} ] && exit 1
[ ! -f ${MANIFEST_FILE} ] && exit 2

VERSION=`cat ${VERSION_FILE} | cut -d 'r' -f 2`
BUNDLE_VERSION=`grep "version" -rI ${MANIFEST_FILE} | cut -d 'r' -f 3`

# check from empty or unset variables
[ -z "${VERSION}" ] && exit 3
[ -z "${BUNDLE_VERSION}" ] && exit 4

# developer mode, allow all updates if version is r0
#[ ${VERSION} -eq 0 ] && exit 0
```

(续下页)

(接上页)

```
# downgrade barrier
if [ ${VERSION} -gt ${BUNDLE_VERSION} ]; then
        echo "Downgrade barrier blocked rauc update! CODE5\n"
else
        exit 0
fi
exit 5
```

该脚本被安装在目标设备上，但是功能并没有被激活。您需要将脚本中 developer mode 的代码行去掉来激活降级屏障

## 7.3 通过 HTTP 流传输升级包

将升级包放置到设备上，除了简单拷贝的方式，还可以利用 HTTP 的流传输，这种方法的优势在于不需要目标设备上的本地存储空间。一种简单的实现方式是在 Docker 容器中运行 NGINX 服务。下面的例子展示了如何通过使能 HTTP Range Request 特性来实现一个最小下载服务器。

创建一个有下列内容的 Dockerfile

列表 5: Dockerfile

```
FROM nginx

COPY bundles /bundles
COPY nginx.conf /etc/nginx/nginx.conf
```

配置 NGINX，使能 HTTP range request，并且将它指向升级文件。

列表 6: nginx.conf

```
events {}
http {
    server {
        proxy_force_ranges on;

        location / {
            root /bundles;
        }
    }
}
```

将升级包放置在 bundles 子目录。在创建所有配置文件后，文件夹结构如下：

```
user@host:rauc-bundle-streaming$ find
.
./bundles
./bundles/phytec-headless-bundle-phyboard-polis-imx8mn-1.raucb
./nginx.conf
./Dockerfile
```

在主机系统上编译并且运行 docker 容器：

```
host:~$ sudo docker build -t rauc-bundle-streaming .
host:~$ sudo docker run --name bundles -p 80:80 -d rauc-bundle-streaming
```

在当前非活跃分区上安装升级包：

```
target:~$ rauc install http://192.168.3.10/phytec-headless-bundle-phyboard-polis-imx8mn-1.raucb
```

**备注**

升级完成后，设备可能会显示如下错误，但是对升级结果无影响：

```
[ 7416.336609] block nbd0: NBD_DISCONNECT
[ 7416.340413] block nbd0: Send disconnect failed -32
```

CHAPTER 8

# 参考

## 8.1 启动逻辑实现

**小技巧**

本章中描述的实现细节仅作参考。支持 RAUC 的 PHYTEC BSP 默认包含这些实现，并且稍作改动。

### 8.1.1 U-Boot 环境变量

对 U-Boot 来说，选择正确启动分区的启动逻辑是在其环境中实现的。作为参考，下面这些是 U-Boot 环境变量中和 RAUC A/B 双备份系统相关性最强的变量：

| 变量名称 | 功能 |
|---|---|
| BOOT_ORDER | 包含以空格分隔的按启动优先级排列的启动设备列表。该参数由 RAUC 自动设置。 |
| BOOT_<slot>_LEFT | 包含每个 slot 的剩余可尝试启动次数。该参数被 RAUC 自动设置 |
| `raucinit` | 包含设置分区的启动逻辑，以加载正确的系统 |
| `doraucboot` | 如果设置为 1，则使能启动 A/B 系统，如果设置为 0 则相反。 |
| `raucargs` | 设置 Kernel 启动参数，例如控制台，根文件系统，RAUC slot |
| `raucrootpart` | 设置设备的根文件系统分区 |
| `raucbootpart` | 设置设备的启动分区 |

这些环境变量在 U-boot 代码 `include/environment/phytec/rauc.env` 中定义

**备注**

分区布局的改变，例如：当需要使用一个额外的数据分区时，可能需要修改变量 `raucrootpart` and `raucbootpart` 。在您修改变量后，请确保使用新的 bootloader 环境重编译镜像。

### 8.1.2 Barebox Bootchooser 框架

对 barebox 来说，它使用 bootchooser 和 state 框架来实现选择正确镜像的启动逻辑。查阅 barebox 文档以获取详细信息：Barebox Bootchooser Framework, Barebox State Framework.

首先，需要在设备树中配置 state 框架。查阅 Yocto 参考手册中的 Customizing the BSP 章节。针对支持的 SoC，我们的 BSP 中已经包含 state 框架配置，可以直接添加到主 barebox 设备树中。例如针对 i.MX6 系列核心板:

```
#include "imx6qdl-phytec-state.dtsi"
```

之后重编译镜像，并且烧写新的 bootloader。

**警告**

注意，添加 state 框架之后，EEPROM 起始的 160 字节被占用，不可以被用户用于其他用途

下面的设备树片段展示了 BSP 中使用的 state 框架配置示例。可以看到，EEPROM 用于存储 state 信息。

```
/ {
    aliases {
        state = &state;
    };

    state: imx6qdl_phytec_boot_state {
        magic = <0x883b86a6>;
        compatible = "barebox,state";
        backend-type = "raw";
        backend = <&backend_update_eeprom>;
        backend-stridesize = <54>;

        #address-cells = <1>;
        #size-cells = <1>;
        bootstate {
            #address-cells = <1>;
            #size-cells = <1>;
            last_chosen {
                reg = <0x0 0x4>;
                type = "uint32";
            };
            system0 {
                #address-cells = <1>;
                #size-cells = <1>;
                remaining_attempts {
                    reg = <0x4 0x4>;
                    type = "uint32";
                    default = <3>;
                };
                priority {
                    reg = <0x8 0x4>;
                    type = "uint32";
                    default = <21>;
                };
                ok {
                    reg = <0xc 0x4>;
                    type = "uint32";
```

(续下页)

(接上页)

```
                    default = <0>;
                };
            };
            system1 {
                #address-cells = <1>;
                #size-cells = <1>;
                remaining_attempts {
                    reg = <0x10 0x4>;
                    type = "uint32";
                    default = <3>;
                };
                priority {
                    reg = <0x14 0x4>;
                    type = "uint32";
                    default = <20>;
                };
                ok {
                    reg = <0x18 0x4>;
                    type = "uint32";
                    default = <0>;
                };
            };
        };
    };
};

&eeprom {
    status = "okay";
    partitions {
        compatible = "fixed-partitions";
        #size-cells = <1>;
        #address-cells = <1>;
        backend_update_eeprom: state@0 {
            reg = <0x0 0x100>;
            label = "update-eeprom";
        };
    };
};
```

要从两个系统中选择一个来启动，bootchooser 需要了解 state 框架配置。对于每个系统，我们都需要一个启动脚本。对于 NAND flash 系统，第一个系统的启动脚本会类似如下：

列表 1: /env/boot/system0

```
#!/bin/sh

[ -e /env/config-expansions ] && /env/config-expansions

[ ! -e /dev/nand0.root.ubi ] && ubiattach /dev/nand0.root

global.bootm.image="/dev/nand0.root.ubi.kernel0"
global.bootm.oftree="/dev/nand0.root.ubi.oftree0"
global.linux.bootargs.dyn.root="root=ubi0:root0 ubi.mtd=root rootfstype=ubifs"
```

第二个启动脚本和第一个结构相同，但是使用包含第二个系统的分区。配备 eMMC flash 的 Machine 使用相似的启动脚本，但是挂载和启动参数会有差异。

运行下面的命令创建必需的 bootchooser 非易失性环境变量：

```
barebox$ nv bootchooser.state_prefix=state.bootstate
barebox$ nv bootchooser.system0.boot=system0
barebox$ nv bootchooser.system1.boot=system1
barebox$ nv bootchooser.targets="system0 system1"
```

## 8.2 eMMC Boot 分区

使用 eMMC flash 存储，我们可以使用 Boot 分区来实现备份 bootloader

默认情况下，使用我们的 BSP 编译出的升级包（例如 `phytec-headless-bundle`）包含用于升级 eMMC boot 分区的 bootloader

注意，U-boot 环境变量仍然存放在第一个分区前的 user 区域。user 区域也包含镜像初始化过程中首次加载的 bootloader。

要手动将 bootloader 写入到 eMMC boot 分区，首先关闭写保护：

```
target:~$ echo 0 > /sys/block/mmcblk2boot0/force_ro
target:~$ echo 0 > /sys/block/mmcblk2boot1/force_ro
```

将 bootloader 写入到 eMMC boot 分区：

```
target:~$ dd if=imx-boot of=/dev/mmcblk2boot0 bs=1k seek=33
target:~$ dd if=imx-boot of=/dev/mmcblk2boot1 bs=1k seek=33
```

该示例针对 i.MX 8M Mini SoC。注意，其他 SoC 可能会有不同的 bootloader 文件以及不同的 bootloader 加载偏移值。该偏移值通过 seek 参数给定。查看下表获取不同 SoC 所要求的偏移值：

| SoC | User 区域偏移值 | Boot 分区偏移值 | eMMC 设备 | Bootloader |
|---|---|---|---|---|
| i.MX 6 | 1 kiB | 0 kiB | /dev/mmcblk | barebox.bin |
| i.MX 6UL | 1 kiB | 0 kiB | /dev/mmcblk | barebox.bin |
| i.MX 8M | 33 kiB | 33 kiB | /dev/mmcblk | imx-boot |
| i.MX 8M Mini | 33 kiB | 33 kiB | /dev/mmcblk | imx-boot |
| i.MX 8M Nano | 32 kiB | 0 kiB | /dev/mmcblk | imx-boot |
| i.MX 8M Plus | 32 kiB | 0 kiB | /dev/mmcblk | imx-boot |
| i.MX 93 | 32 kiB | 0 kiB | /dev/mmcblk | imx-boot |
| AM62x AM62Ax AM64x | N/A | 0 kiB 512 kiB 2560 kiB | /dev/mmcblk | tiboot3.bin tispl.bin u-boot.img |

### 8.2.1 Bootloader 偏移

注意，即使是同一 SoC，偏移值也会随 bootloader 烧写位置在 user 区域还是 boot 分区而不同。

在 bootloader 写入 eMMC boot 分区后，从 boot 分区启动需要使用以下命令使能：

```
target:~$ mmc bootpart enable 1 0 /dev/mmcblk2
```

这也意味着只有写到 eMMC boot 分区的 bootloader 才会被使用。user 区域的 bootloader 不再被使用。在使用升级包升级目标系统时，这些步骤也会在 RAUC 内部逻辑中执行。

要关闭从 eMMC boot 分区启动，只要输入下面的命令：

```
target:~$ mmc bootpart enable 0 0 /dev/mmcblk2
```

此命令执行后，eMMC user 分区提供系统的 bootloader

使用 U-Boot 时，在 bootloader 中也可以使用相似的命令：

```
u-boot=> mmc partconf 2 0 0 0  # disable
u-boot=> mmc partconf 2 0 1 0  # enable
```