rxos Documentation

Release 1.0rc1

Outernet Inc

May 14, 2017

Contents

rxOS is a Linux-based operating system (and also the system image) that is used in Outernet's Lantern and Lighthosue Mk 2 products, both based on Raspberry Pi 3 and CHIP.

rxOS is built on top of Buildroot and constists of two parts:

- Linux kernel image with early userspace
- rootfs image

CHAPTER 1

Documentation

Getting stated with rxOS

This chapter is aimed towards DIY users that wish to deploy rxOS on the supported boards. If you purchased a rxOS-powered device from Outernet, you may wish to skip ahead to *How rxOS works*.

A DIY build based on Raspbian is not covered in this guide. For instructions on how to set up an Outernet L-band receiver using Raspbian, you should look at the Outernet L-band on Raspberry Pi guide.

Raspberry Pi

In order to build a Raspberry Pi receiver, you will need the following:

- Raspberry Pi 3
- 4GB (or larger) microSD card (8GB and larger is recommended)
- RTL-SDR USB dongle
- LNA
- patch antenna

The RTL-SDR radio dongle, LNA, and antenna, can be purchased through Outernet either individually or as a kit.

You will also need a Raspberry Pi image, which can be downloaded from archive.outernet.is/images/rxOS-Raspberry-Pi.

Flashing the image

In order to use this image you will need to flash it to an SD card.

Windows

Obtain Win32 Disk Imager. Open the program (it will ask for administrative privileges), select the image file and destination drive, and click on "Write".

> **Warning:** Be careful not to select your system drive as the destination drive.

Linux

Insert the card into the card reader. Find out the what your SD card's device node is by using the `dmesg | tail` command. Let's say the device node is `/dev/sdb1`.

> **Warning:** Be absolutely sure it's the correct device node. `dd` will not ask any questions, and will happily overwrite anything you give it. If you are unusure, it's best to ask on a Linux forum about how to find out whether you have the right device node.

Make sure the SD card is not mounted if you have an automouter.:

```
$ sudo umount -f /dev/sdb1
```

To write the image to the card:

```
$ sudo dd if=sdcard.img of=/dev/sdb1 bs=16m
```

Mac OSX

Insert the card into the card reader. Find out what your SD card's device node is by using the `diskutil list` command. Let's say the device node is `/dev/disk4`.

> **Warning:** Be absolutely sure it's the correct device node. `dd` will not ask any questions, and will happily overwrite anything you give it. If you are unusre, it's best to ask on an OSX forum about how to find out whether you have the right device node.

You need to unmount the disk:

```
$ diskutil unmountDisk /dev/disk4
```

Finally you can write the image:

```
$ dd if=sdcard.img of=/dev/disk4 bs=16m
```

Connecting the hardware

Once the image is done, put the SD card into the SD card slot. Connect the antenna to the LNA, and then connect the LNA to the RTL-SDR dongle. Finally, plug the RTL-SDR dongle into one of the available ports on the Raspberry Pi.

Power the Raspberry Pi on and continue to *The first steps*.

C.H.I.P

In order to build a CHIP-based receiver, you will need the following:

- CHIP
- RTL-SDR USB dongle
- LNA
- patch antenna
- microUSB cable with data connection
- (optional) USB hub, if you wish to connect both the RTL-SDR dongle and some other device, like USB storage
- (optional) UARD USB adapter, if you wish to monitor the boot loader output
- (optional) USB Ethernet adapter for wired network connection

The RTL-SDR radio dongle, LNA, and antenna, can be purchased through Outernet either individually or as a kit.

You will also need a CHIP image, which can be downloaded from archive.outernet.is/images/rxOS-Raspberry-Pi.

Flashing the image

At this time, CHIP can only be flashed from a Linux machine, either a virtual machine or a native install.

Serial console

It helps to have access to a serial console. You will need an UART-USB or similar adapter. The UART pins are located on the U14 pin header on the inside, near the microUSB connector, marked as UART-TX and UART-RX. If the board is powered from a source other than the PC to which the UART is connected, remember to also plug in the ground lead to one of the pins marked as GND on the CHIP (one is conveniently provided right next to the UART pins).

Preparing the system

To access CHIP as a normal user, you will need to set up the udev rules so that the device has appropriate permissions when plugged in. Download the `99-chip.rules` file. Edit the file with your favorite editor, and change all referneces to `plugdev` to your username. Alternatively, you can add your user to the `plugdev` group.

Reload the udev rules with this command:

```
$ sudo udevadm control --reload-rules
```

Next we need to set up the software. Using your systme's package manager, install the following software (possible package names are specified in parenthesis):

- build tools (see note below)
- git (git)
- dd (coreutils)
- lsusb (usbutils)
- fel (sunxi-tools)
- mkimage (uboot-tools)

- fastboot (android-tools or android-tools-fastboot)
- img2simg (android-tools, simg2img, or android-tools-fsutils)

Note: Here are some commands for installing build tools on different distros:

Ubuntu/Debian and derivatives:

```
$ sudo apt-get install build-essential
```

Fedora:

```
$ sudo yum groupinstall "Development Tools" "Development Libraries"
```

Arch Linux:

```
$ sudo pacman -Sy base-devel
```

Opensuse:

```
$ sudo zypper install --type pattern devel_basis
```

We will need to clone a few Git repositories which contain non-standard tools that are not commonly available as packages (yet):

```
$ git clone https://github.com/Outernet-Project/CHIP-tools.git
$ git clone https://github.com/Outernet-Project/sunxi-tools.git
```

We need to compile these tools so run a few more commands:

```
$ cd CHIP-tools
$ make
$ CHIP_TOOLS=$(pwd)
$ cd ../sunxi-tools
$ make
$ FEL=$(pwd)
```

Don't close the terminal just yet.

Flashing

CHIP must be put into FEL mode before it can be flashed. To do this you will need a (relatively thin) paper clip, or a twist tie with stripped ends (exposed wire). With the paper clip or twist tie, connect the pins marked as FEL and GND. The FEL pin is located at the 4th position in the inside row towards the microUSB port. GND pin is located at the last position in the same row as the FEL pin. Once the pins are connected, plug the CHIP into a USB port.

You should see a device node /dev/usb-chip. If you don't see it, your udev rules may need a refresh, or you may need to check the rules file for typos.

In the terminal where you set the software up, navigate to where you unpacked the flash files. Start the flash script:

```
$ cd path/to/firmware/directory
$ PATH="$PATH:$CHIP_TOOLS:$FEL" bash chip-flash.sh
[  0.01 ] ===> Preparing the payloads
[  0.01 ] .... Preparing the SPL binary
[  0.01 ] .... Preparing the U-Boot binary
```

```
[  0.30 ] .... Preparing sparse UBI image
[  0.41 ] ===> Creating U-Boot script
[  0.42 ] .... Writing script source
[  0.42 ] .... Writing script image
[  0.42 ] ===> Uploading payloads
[  0.43 ] .... Waiting for CHIP in FEL mode...OK
[  0.44 ] .... Executing SPL
[  1.97 ] .... Uploading SPL
[  8.86 ] .... Uploading U-Boot
[ 16.15 ] .... Uploading U-Boot script
[ 16.16 ] ===> Executing flash
[ 16.17 ] .... Waiting for fastboot.......OK
target reported max download size of 314572800 bytes
sending 'UBI' (204800 KB)...
OKAY [ 16.949s ]
writing 'UBI'...
OKAY [ 44.200s ]
finished. total time: 61.149s
resuming boot...
OKAY [  0.000s ]
finished. total time: 0.000s
[101.84] ===> Cleaning up
[101.86] ===> Done

!!! DO NOT DISCONNECT JUST YET. !!!

Your CHIP is now flashed. It will now boot and prepare the system.
Status LED will start blinking when it's ready.
```

As the message says, within about 2 minutes, the status LED will start blinking. At that point, you will be able to start using your newly flashed CHIP.

Note on fastboot and virtual machines

If you are using a virtualmachine (e.g., VirtualBox or VMware), you should be aware that, during the flashing, when the "Waiting for fastboot" message appears, the CHIP will change its USB ID. This means that the USB ID you originally set up while it was in FEL mode will no longer apply, and the guest OS will loose the connection to CHIP. This results in fastboot timeout.

Once fastboot times out, you should reconfigure your virtual machine manager to make the new USB ID available to the guest.

Connecting the hardware

Connect the antenna to the LNA, and then connect the LNA to the RTL-SDR dongle. Finally, plug the RTL-SDR dongle into the USB port on the CHIP.

Power the CHIP on and continue to *The first steps*.

The first steps

Once your receiver is all flashed and running, here are a few things you can do.

Access the web-based interface

rxOS ships with a web-based user interface called Librarian. This interface is available on port 80 at the receiver's IP address.

In the most common scenario, you will find a new WiFi access point named 'Outernet'. If you connect to it, you will be able to reach the web interface at 10.10.10.10 or librarian.outernet.

Note: If your browser interferes with 'librarian.outernet' domain name and takes you to search instead, you can either prefix the 'http://' bit or use any domain name like example.com to reach the receiver's web interface.

You can get to the receiver a few different ways, which is documented in the *Remote shell access* section.

Plug in the external storage

rxOS-basd receivers support external storage devices to expand the storage available on the SD card or built-in NAND flash. More about that can be read in the *Using external storage devices*.

Point the antenna

If you haven't already, you will want to adjust the antenna's pointing. Precise pointing is a topic that is outside the scope of this guide.

What rxOS is and what it isn't

This section is intended for people who have some familiarity with the Linux operating system. If you are not familiar with Linux, you may have some trouble following along, but the information found in this section is not essential for using an rxOS-based device so feel free to skip it.

Probably the most important thing to note is that rxOS is not a Linux distribution. This means that, among other things:

- it has no writable root filesystem
- it has no packages or package manager
- it has no build tools
- it has no system administration tools

Read-only root filesystem

From time to time, we are asked about customizing the rxOS-based receivers. While some of the configuration persists across reboots, one should not expect to permanently modify the system outside of the short list of persistent files.

No packages

The rxOS image is built as a unit. It is one monolithic system and there is no notion of packages (nor there ever will be). There are sets of add-on files that are added on top of the image via the root filesystem overlays (see *Root filesystem overlays* for an in-depth treatment of the subject), but those are not packages that you install with package managers such as `apt` and `pacman`.

No build tools

As mentioned under the previous heading, rxOS is built as a unit. It is built on a standard Intel PC and then flashed to a device. Because of this there are no build tools *on* the device. Adding your own programs to the image is done by rebuilding the image the same way it was originally built (a short guide on that topic can be found in the *Building and customizing the image* chapter).

No sysadmin tools

Well, this isn't entirely true. There are some tools for working with the system. What is missing, though, are things that would normally alter the system configuration. E.g., there is no support for changing the init script order and state, there are no system configuration tools, etc. The rxOS system is mainly intended to be left as is as much as possible, and modification to the system are done at built time.

What good is all this?

While all of this may sound a bit limited, it has some advantages when it comes to use on embedded devices.

Writable file systems are susceptible to corruption. If a device is writing during power loss, the filesystem may be left in a state that makes it impossible to boot up.

You can write files to the root filesystem when the system is up and running. This is done by overlaying a fake filesystem in the system memory (RAM) on top of the otherwise read-only root filesystem. If there is a bug in the software, this may sometimes cause the system to exhibit weird behavior. Having a read-only root filesystem allows the user to power-cycle the device and go back to the known good state. Not having packages, build tools, and so on, are design decisions along the same lines: the less modification you can make to the system, the better the chance of being able to get back to the known good state.

Another reason rxOS is quite stripped down is that the image needs to be updated over-the-air (OTA). This puts a restriction on the amount of bandwidth the image can use, so virtually all non-essential items have been stripped out.

How rxOS works

This section provides a high-level overview of how rxOS works.

The boot process

The boot process is divided into 3 stages:

- Hardware boot
- Early userspace init
- Userspace init

Storage contents

The are two types of storage devices used in rxOS, based on the target device. On Raspberry Pi, SD cards are used, while on CHIP, the built-in NAND flash storage is used.

SD card boot partition contents

The first partition of the SD card is a 400MB FAT32 partition with the following contents:

- Raspberry Pi stage 2 bootloader (`bootcode.bin`)
- Raspberry Pi stage 3 bootloader (`start.elf`)
- Raspberry Pi fixup data (`fixup.dat`)
- device tree blob (`bcm2710-rpi-3-b.dtb`)
- kernel image with early userspace (`kernel.img`)
- main root filesystem image (`root.sqfs`)
- backup root filesystem image (`backup.sqfs`)
- original factory root filesystem image (`factory.sqfs`)

NAND contents

The NAND contains the following:

- SPL (secondary program loader) partition
- SPL backup partition
- U-Boot bootloader partition
- U-Boot bootloader environment partition
- **boot partition**
  - kernel image (`zImage`)
  - device tree blob (`sun5i-r8-chip.dtb`)
- root filesystem partition
- backup root filesystem partition
- data partitions

Hardware boot

Before any of the rxOS-specific software is executed, there is a phase in which the base hardware is initialized and the hardware-specific bootloaders are run.

Raspberry Pi boot

The first stage is the same for all Raspberry Pi devices, but it will be described here for completeness.

The ROM on the Raspberry Pi board contains the stage 1 bootloader (S1). When power is applied, VideoCore GPU is activated, and S1 is executed on a small RISC core that is present on the board. The S1 bootloader mounts the SD card, and loads S2, `bootcode.bin`, into the GPU's L2 cache and executes it on the GPU.

S2 activates the SDRAM. It also understands ELF binaries, and loads S3, `start.elf`, from the SD card. S3 is also known as GPU firmware, and this is what causes the rainbow splash (4 pixels that are blown up to full-scren) to be displayed.

Note: If there is a rainbow splash on the screen but no further activity, it means that S3 has been loaded successfully but kernel image did not boot.

S3 reads the firmware configuration file `config.txt` (if any), and then proceeds to split the RAM between GPU and ARM CPU. S3 also loads a file called `cmdline.txt` if it exists, and will pass its contents as kernel command line. S3 finally enables the ARM CPU and loads the kernel image (`kernel.img` by default, configurable via `config.txt`), which is executed on the ARM CPU.

Note: `start.elf` is actually a complete proprietary operating system known as VCOS (VideoCore OS).

The kernel image contains a minimal early userspace and its `init` script is executed.

CHIP boot

When the board receives power, code in the boot rom (BROM) is executed. This code will activate a small amount of memory and load SPL from either the first NAND partition or its backup image on the second partition.

SPL's job is to activate all of DRAM and load the U-Boot bootloader from the third partition.

When U-Boot is activated it reads the boot parameters from the U-Boot environment partition. By default, this will cause U-Boot to mount the boot partition (labelled 'linux'), and load the kernel image and the DTB from it.

Finally, the kernel image is executed.

The early userspace

Early userspace initialization happens within the `init` script in the root of the kernel's initramfs. We will refer to this script as EI for brevity (early init).

EI is generated from a template found in `rxos/initramfs/init.*.in.sh` file. The asterix in the name can be either 'nand' or 'sdcard' depending on the target platform. The sources are thoroughly documented, so if you need to know more than what's presented here, you are welcome to peruse the sources.

It first mounts devtmpfs to `/dev` so that device nodes are accessible. It then mounts the boot partition in order to access root filesystem images/partitions.

On Raspberry Pi, additional data partitions are created. For NAND-based boot, this step is skipped because the extra partitions are programmed along with the base system when the NAND is flashed at the factory.

On Raspberry Pi, there are three possible candidates for the final userspace, and those are `root.sqfs`, `backup.sqfs`, and `factory.sqfs`. On CHIP, there are two possible candidates, `ubi0:root` and `ubi0:root-backup`.

A RAM disk with size configurable at build-time (default is 80 MiB) is created to serve as a write-enabled overlay over the read-only root filesystem. The mount points for the SD card and devtmpfs are moved to `/boot` and `/dev` in the target rootfs, respectively.

If overlay SquashFS images are found (named `overlay-<name>.sqfs`), they are laid over the root filesystem to provide device-specific extension.

For each candidate root filesystem, EI mounts the image, and creates a write overlay using OverlayFS and the previously configured RAM disk. It then attempts to switch to the new root filesystem using BusyBox's `switch_root` command which executes the `/sbin/init` binary in the target root filesystem.

If the switch is successful, early userspace initialization is complete and the userspace proper takes over.

If the switch is not successful, the next candidate is tried until no root filesystem candidates are left. If none of the root filesystem images/partitions can be booted, EI starts an emergency shell where troubleshooting can be performed.

Note: Even if switch is successful, it does not mean the boot will succeed. Minimal checking is performed to ensure that the root filesystem contains a path `/sbin/init` which an executable file or a symlink pointing to one, but nothing beyond that is done. If the executable fails or does something that terminates the init process, kernel **will** panic and boot will fail. In general, however, this is not quite realistic as long as valid images built for rxOS are used.

The userspace

Userspace initialization happens in the rootfs and is carried out by the init scripts in `/etc/init.d`. The init scripts are executed synchronously in the lexical file name order.

The userspace will start the WiFi hotspot, web and database servers, and Outernet applications. Any attached external storage devices will also be mounted during this stage.

The early userspace

The early userspace consists of an init script and a minimal set of libraries and programs necessary to prepare the system for the proper userspace initialization.

The early userspace is built by enabling the "User-provided options > System configuration > Create early userspace cpio archive" option. This option is defined by a local package called `ramfsinit`. The package contains the file list for the early userspace and the init script template.

During the early userspace initialization, the SD card's boot partition is mounted. If the card contains any one-time boot hooks, those are executed and immediately removed. The root filesystem images are found, and mounted, and the init script finally switches to the mounted root filesystem (userspace proper). If no mountable root filesystem images are found, init script drops into an emergency shell.

Boot hooks

Boot hooks are scripts in the early userspace image that perform pre-boot setup. One example of such a hook is the `selfpartition` hook (provided by `boothook-selfpartition` local package) which prepares the data persistence partitions on the Raspberry Pi SD card.

Hooks are looked up by `hook-*.sh` name pattern and are executed in a subshell every time rxOS boots. It is the hook script's job to determine wither they need to take action or not.

You can read more about boot hooks in *Boot hooks*.

Mounting the rootfs

On the SD card there are three root filesystem images. The `rootfs.sqfs` is the default root filesystem image. `backup.sqfs` is the same image as the default one, and serves as a backup in case the first image is corrupted or OTA update fails. Lastly, the `factory.sqfs` serves as a fallback image. The latter remains the same throughout the device's useful life (factory state).

Root filesystem images are loop-mounted read-only and are overlaid with a RAM disk that provides the write layer. This provides protection against corruption resulting from partial writes and allows the user to reset the device to a known state by power-cycling it.

The situation is similar for NAND flash, but the root filesystem is stored as UBI volumes, and there are only two copies.

Emergency shell

Emergency shell is a minimal shell that can be used when troubleshooting boot issues. This is not a fully featured environment such as the one typically found in desktop Linux distributions, so many familiar commands are missing (e.g., no `lspci`, `lsusb`, `htop`, etc). All commands are available through the `busybox` binary at the root of the early userspace RAM disk. Busybox documentation describes these commands (called 'applets'), but one should have in mind that not all Busybox features are enabled.

Remote shell access

rxOS only supports the following methods for remotely accessing the system:

```
- SSH over wireless hotspot (Pi3 and CHIP)
- SSH over Ethernet (Pi3)
- SSH over USB Ethernet (CHIP)
- Serial console over USB (CHIP)
- Serial console over UART (CHIP)
```

IP addresses

The following addresses are assigned to on-board network interfaces:

interface	address
WiFi	10.0.0.1
USB Ethernet	10.10.10.10
Ethernet	dynamic (DHCP)

Connecting to networks

rxOS-based receivers have several ways to network with other devices and networks.

Wireless hotspot

When a receiver is started, a wireless network is created automatically. The default network name (SSID) is "Outernet" and is not password-protected.

USB Ethernet (CHIP only)

USB Ethernet connection is establish when the receiver is plugged in to a computer using the microUSB cable. Assignment of IP address to the connected computer is automatic.

Ethernet connection (Raspberry Pi3 only)

The receiver can be connected to a router, and it will automatically obtain an IP address. Since this IP address is dynamic, you will need to consult your router's administrative interface, or use the wireless hotspot on the receiver to find the IP address that was assigned.

SSH access

For SSH access, use port 22 and username `outernet`. The default password for the `outernet` user is `outernet`.

Note: Once you go through the setup wizard in the web inteface, the SSH user will change to the superuser credentials that you set up during the wizard. Please adjust accordingly.

Root login is not enabled, but `sudo` can be used to gain full root access:

```
[rxOS][outernet@rxos ~]$ sudo su
Password: ********
[rxOS][root@rxos /home/outernet]# _
```

USB serial console access (CHIP only)

USB serial console becomes available when the receiver is connected to a computer using a microUSB cable. Software such as PuTTY, screen, or minicom, can be used to access this console. This console is log-in only and does not show boot messages. For a more comprehensive console access, UART console is recommended.

UART console (CHIP only)

USB-UART adapter (not included) can be plugged into the UART pin headers on the CHIP to provide access to UART console. Depending on the receiver design, the case may need to be removed to gain access the these pins. Software such as PuTTY, screen, or minicom, can be used to access this console.

The UART console provides access to boot messages, and allows the user to interact with not just rxOS, but also the U-Boot bootloader.

Changing the user password

User password can be changed using the `passwd` command:

```
[rxOS][outernet@rxos ~]$ passwd
Current password: *******
Enter new password: ******
Retype new password: ******
```

Using external storage devices

Over time, the storage on the SD card may become full. rxOS supports using external storage devices via the USB ports. Only one external storage device can be used at a time.

The following table shows the supported disk formats and their characteristics.

format	integrity check	large disks	power failure recovery
FAT32	Yes	No	No
NTFS	No	Yes	Yes
ext2[1]	Yes	Yes	No
ext3[1]	Yes	Yes	Yes
ext4[1]	Yes	Yes	Yes

[1] Creation of disks in this format is only supported on Linux operating systems

FAT32 is the most common disk format for USB sticks and external hard drives. Although exFAT is becoming more common with large-capacity devices, it is not supported by rxOS, so such devices should be reformatted using the NTFS format (FAT32 formatting on Windows does not support large disks).

Linux users may use ext2, ext3 or ext4 format in addition to the other two.

Integrity check is prerformed on the disk where supported and an attempt is made to fix any problems with the on-disk data. While FAT32 supports integrity checking, it is not as complete as running disk checks on Windows machines.

Connecting an extrnal storage device

External storage devices can be simply plugged into one of the available USB ports.

> **Warning:** Some external storage devices (and especially mechanical USB hard disks) draw a lot of electrical current from the USB port and may cause Raspberry Pi to shut down. In such cases, a powered USB hub will be required.

When the storage device is plugged in, it is checked for format. If the device uses an unsupported format, it is not used and may be safely unplugged.

rxOS then mounts the disk to a temporary location to test that it can be mounted. If the test mount fails, the disk is ignored and not used as external storage.

When the test mount is successful, rxOS will mount the disk as external storage and will redirect downloads to it.

Once the disk is mounted, reindexing of the content is started, and any content that already existed on the attached disk becomes available after a while.

Note: Although it is possible to attach more than one storage device, only the last-attached device is used as external storage. The other storage device is unmounted and may be unplugged.

Disconnecting the external storage device

External storage devices may be simply disconnected. Unmounting is done after the fact, unless the user interface provides support for unmounting.

When a disk is disconnected, downloads are redirected to the internal storage on the SD card, and reindexing is started. Any files that were present on the external storage device become unavailable afer a while.

Status LED indication

During storage hot-plug events, the status LED (green) indicates the status of the hot-plugging.

LED blinking slowly (0.5s interval) indicates that the storage mounting has started.

LED blinking fast (0.1s interval) indicates that the integrity check has started.

LED solid indicates that the storage device was mounted.

LED off indicates that the storage device was not mounted due to an error.

Storage layout

rxOS storage is highly compartmentalized to allow different pieces of the system and application code to utilize different portions of the storage without stepping on each other. There are two types of storage used by rxOS:

- SD card (on Raspberry Pi)
- NAND flash (on CHIP)

The following table shows different portions of the storage on SD card and how they are used:

Storage device		Size	Format	Usage
1	Primary partition	200M	FAT32	*Boot files*
2	Extended partition			*Receiver state*
5	conf	24M	ext4	Persistent system configuration
6	cache	600M	ext4	Download cache
7	data	2G	ext4	Application data
8	downloads	rest	ext4	Downloaded files

The following table shows different portions of the NAND flash storage and how they are used:

Storage device		Size	Format	Usage
1	spl	4M	raw	Secondary program loader
2	spl-backup	4M	raw	SPL backup
3	uboot	4M	raw	U-Boot Bootloader
4	env	4M	raw	Bootloader settings
5	swap	400M	raw	(reserved for future use)
6	UBI			Kernel and *Receiver state*
1	linux	64M	ubifs	*Boot files*
2	root	200M	ubifs	Root filesytem
3	root-backup	200M	ubifs	Backup root filesystem
4	conf	64M	ubifs	Persistent configuration
5	cache	600M	ubifs	Download cache
6	appdata	1G	ubifs	Application data
7	data	rest	ubifs	Downloaded files

Boot files

Stores the boot files.

On Raspberry Pi, it contains the following files:

- Raspberry Pi 3 binary device tree
- Stage 2 and 3 bootloader
- `start.elf`
- Kernel image (`kernel.img`)
- Main and fallback rootfs SquashFS images (`root.sqfs`, `backup.sqfs`)
- Factory default SquashFS image (`factory.sqfs`)

On CHIP, it contains the following files:

- CHIP binary device tree
- Kernel image (`zImage`)

These files are read-only (except when updating the system).

Receiver state

The receiver state extended partition is split into 4 logical partitions. These partitions contain:

- persistent system configuration
- download cache
- application data
- downloads

Persistent system configuration

The init script that sets up configuration overrides maintains a list of configuration files that should be overridden based on the contents of the config partition. The files from the config partition are symlinked to appropriate locations in the rootfs.

Download cache

The download cache is a storage area for partial downloads. ONDD download cache is stored in a separate partition to maximize control over the storage capacity.

Application data

Application data partition stores application configuration, databases, and other application state.

Downloaded files

Remaining space on the SD card is used for permanent storage of downloaded files.

OTA updates

rxOS supports secure automatic over-the-air updates (OTA updates).

When updated firmware is delivered into `/updates/<platform name>/` directory, it is passed to `pkgtool` which is an update package verfication tool. The update package contains the payload files, executable script (installer script) that controls the update process, and a signature. The signature is checked by the `pkgtool` to ensure the authenticity of the update package. Once the package is verified, the installer script is executed.

The installer script may contain arbitrary code, so udpate is just one of the many things it can be configured to do. In this section, we will discuss a typical update process.

Normally, the payload for an update pakage consits of the root filesystem image and one or more of the optional extras:

- the pre-install script
- the post-install script
- kernel image
- DTB file
- firmware
- bootloader

- device-specific overlays

The update starts with a version check. If the version of the receiver's firmware is the same or newer than the version of the payload, installer aborts the update.

If pre-install script is present, it is executed first.

Next the payload is copied or flashed into appropriate location.

On CHIP, the root filesystem is flashed to its backup UBI volume and then the names of the backup and main root filesystem volumes are swapped, so that the new root filesystem will get mounted on boot.

On Raspberry Pi 3, the root filesystem is copied into the boot partition just like the kernel image.

Once the installation is finished, the post-install script, if present, is invoked. This script performs any one-time adjustments to the system before reboot.

The system is finally rebooted to complete the update.

This entire process happens automatically without user intervention.

Setting up for remote support

rxOS allows remote support channel to be set up by adding a simple text file on an USB stick and plugging it into the receiver.

Note: Because of the security implicatins, remote support channel is only opened with full cooperation from Outernet, and only trusted individuals and organizations (e.g., partners, clients) will are allowed to use this feature.

Configuration file

The remote access configuration file is named REMOTE (all-caps). The file contains configuration parameters in `NAME='value'` format, exactly one parameter per line. The following table contains all the possible parameters and their purpose. All parameters are optional except the `KEY`.

Parameter	Example	Meaning
PORT	9978	This value should be a port number issued by Outernet staff. If omitted, a port between 20000 and 30000 will be randomly selected.
HOST	support.outernet.is	Domain name of the support server as specified by the Outernet staff. If omitted, sub.outernet.is is used.
NAME	john-lantern	Name of the receiver. This helps the support staff identify the receiver. If omitted, the default name 'rxos' is used.
SSID	mywifi	SSID (access point name) of the access point which should be used to connect to Internet. If this is left blank, wireless connection is not used, and instead, it is assumed that the receiver will gain access to Internet by some other means. PASSCODE parameter is required when using SSID.
PASSCODE	some secret	Passcode (password) of the wireless access point that should be used to connect to Internet.
KEY	(gibberish)	Access key that is used to establish a connection with the server and provided by the Outernet staff.

Warning: Any and all values must be quoted using single quotes. Failure to do this may result in unexpected behavior.

Warning: NEVER SHARE THE ACCESS KEY WITH ANYONE. You must ensure that the access key does not fall into the wrong hands. If the key is misused by a malicious user, any receivers connected to the support server or the networks they are on may get compromised.

Activating the support connection

To activate the support connection:

- power down the receiver
- put the `REMOTE` file onto USB storage device
- plug the USB storage device into the receiver
- power the reciever up

Deactivating the support connection

To deactivate the support connection:

- power down the receiver
- remove the USB storage device from the reciever (if you need to use the storage device again, delete the `REMOTE` file from it)
- power the receiver up

How it works

When rxOS boots, if the appropriate configuration file and access key are found on the USB storage device, the networking is reconfigured (when using the `SSID` parameter) to access the Internet, and a SSH connection is established with the remote support server allowing a tunnel from the specified `PORT` back to the SSH port on the receiver. The Outernet staff then accesses the receiver using SSH through this tunnel (reverse SSH).

Building and customizing the image

This section provides information about building and customizing the rxOS image.

Build requirements

In order to build the rxOS firmware, you will need a Linux (virtual) machine.

For virtual machines, hardware virtualization (Hyper-V, etc) is highly recommended, as well as as much RAM as you can spare.

You will need to have the following packages installed:

- Build tools[1]
- bc
- git
- rsync
- unzip
- cpio
- wget
- mercurial (hg)

Note: Here are some commands for installing build tools on different distros:

Ubuntu/Debian and derivatives:

```
$ sudo apt-get install build-essential
```

Fedora:

```
$ sudo yum groupinstall "Development Tools" "Development Libraries"
```

Arch Linux:

```
$ sudo pacman -Sy base-devel
```

Opensuse:

```
$ sudo zypper install --type pattern devel_basis
```

Building the firmware image

The build is broadly divided into three parts that are carried out in one sequence, driven by Buildroot.

- Linux kernel compilation
- Rootfs compilation
- Early userspace and kernel recompilation

During Linux kernel compilation, a kernel image is compiled with a dummy initramfs image (just an empty file). Following the kernel, the root filesystem contents are compiled. Finally, the cpio archive for the early userspace is created and linked into the kernel image.

In order to build the firmware, you should be familiar with Buildroot.

Before you build

Before you build, you need to clone the rxOS git repository:

[1] The build tools include compilers (e.g., gcc), standard libraries, and build automation tools (make, automake, autoconf, etc). In some Linux distributions, there are packages that bundle these tools together (e.g., 'build-essential' on Ubuntu, or 'base-devel' on Arch Linux). If you are unsure, try googling "build-essential equivalent for <distro name>".

```
$ git clone --recursive https://github.com/Outernet-Project/rxOS.git
```

The `--recursive` flag causes git to init and update any submodules. If you forgot it, you need two additional steps:

```
$ git submodule init
$ git submodule update
```

Selecting the board

rxOS build supports a few different target boards. These boards are selected using the `BOARD` variable in the makefile, like this:

```
$ make BOARD=<boardname> [TARGET]
```

The boardname can be one of the following:

- `rpi3`: Raspberry Pi3
- `chip`: NTC C.H.I.P.

Shortcut scripts for making for specific targets are available at the root of the source tree. These scripts are named after the respective target boards. For example, to bring up the configuration menu for Raspberry Pi 3:

```
$ ./rpi3 menuconfig
```

Note: In the rest of the documentation, you can replace `make` with the board script (e.g., `./rpi3`) depending on the board you wish to build for.

Starting the build

The build is initated by invoking `make` or `make build`.

To completely clean up the build and restart it from scratch, use `make clean build`. It is generally not needed to do this, though. In most cases, a more efficient alternative is to call `make rebuild-everything`.

Customizing the build

To customize the build use `make menuconfig`, which brings up the Buildroot's configuration menu.

Updating the existing build

To update your local repository clone:

```
$ cd path/to/rxOS
$ git pull
$ git submodule update
```

Apply possibly updated build configuration:

```
$ make config
```

Rebuild starting from the linux kernel:

```
$ make rebuild-with-linux
```

Linux kernel compilation

The kernel is compiled using Buildroot's build scripts. The kernel compilation can be controlled to some degree using Buildroot's Kernel menu. Any patches applied to the kernel can be found in `rxos/patches/linux` directory.

The kernel configuration is found in the `rxos/configs/rxos_kernel_defconfig` file.

The following files are generated during the build targeting the Raspberry Pi board:

zImage	kernel image with linked initramfs
kernel.img	kernel image with Raspberry-Pi-specific trailer

The `kernel.img` file is created by a post-image hook called `rxos/scripts/add_trailer.sh`. The script's source includes more information on what it does and why.

The following files are generated during the build targeting the CHIP board:

zImage	kernel image with linked initramfs

To restart the build from the kernel image compilation you can use the `rebuild-with-linux` target.

U-Boot compilation

The CHIP build will also generate a bootloader image, `u-boot-dtb.bin` and its derivative SPL images (secondary program loader), `sunxi-spl.bin` and `sunxi-spl-with-ecc.bin`.

To recompile U-Boot, the `uboot-dirclean` target is used to clean the U-Boot build, and `uboot-rebuild` target to rebuild it.

Rootfs compilation

The root filesystem image is created from a collection of packages, both from the Buildroot's own package collection found in `buildroot/package` and the additional Outernet-provided packages found in `rxos/package`. The Buildroot's packages are selected using Buildroot's Target packages configuration section. Outernet-provided packages are selected in various sections within the User-provided options section of the Buildroot's configuration menu.

The rootfs format is SquashFS compressed using LZ4 algorithm. This can be changed in the Buildroot's Target filesystems menu.

To apply small changes to the root filesystem (e.g., a new package was added or an existing package was updated), run `make rebuild`. Keep in mind that Buildroot does not track what files belong to what package. Because of this, when removing packages, or when updating packages to version that no longer contain some of the files that they used to contain, you may end up with stray files from the previous builds. If this happens, `make clean build` should be used instead.

Early userspace

The early userspace is built last as it is built from pieces of the root filesystem. This is facilitated by the Outernet-specific patches applied to the version of Buildroot used by rxOS.

The initial RAM disk (initramfs) image is build as a compressed cpio archive, and the list of files that end up in the final initramfs image is controlled by several different packages, including the `ramfsinit` local package. The packages each provide a template that points the kernel's `gen_init_cpio` script to appropriate files in the root filesystem.

Known issues

The build scripts are still under development. In some cases, `rebuild*` targets may fail. If a rebuild target fails, try falling back on another one (e.g., if `rebuild` fails, try `rebuild-with-linux`), and finally do a `make clean build`.

Also, be sure to report any build issues so we can address them.

Boot hooks

Boot hooks are early userspace shell scripts (part of the initial RAM filesystem image) that are invoked by the `init` script on each boot.

Boot hook local packages

Boot hooks are provided by local packages that are named `boothook-*`. A typical boot hook consists of the following:

- initial RAM filesystem file list (to be added to the default set of files)
- hook script

Additionally, the package may select other packages that would become part of the root filesystem and used as the source for the initial RAM filesystem file list, or provide configuration options for the hook script.

The initial RAM filesystem file list is mandatory and it must be installed in the `$(BINARIES_DIR)/initramfs` directory. The file must contain at least a reference to the hook script and place the hook script into the root of the initramfs.

There are no rules as to the naming of the initramfs list file, but a convention is to use the package name without the `boothook-` prefix, and append `.cpio.in` extension. The file must also be declared as the initramfs list extension by appending its name (just the name, not the full path) to `INIT_CPIO_LISTS` variable in the makefile. More information about the initramfs list format can be found on landley.net.

By convention, we install the hook script into the same directory as the initramfs list, so that it can be accessed quickly when needed for debugging and similar purposes, but this is not required.

Finally, the package is registered under the "System configuration" section in the rxOS's master `Config.in`.

Simple boot hook example

For our example, we will create a simple hook script that shows a greeting with configurable name. Let's call this hook 'greeter'.

We first need to create a directory for the hook.

```
$ mkdir -p rxos/local/boothook-greeter/src
```

Note that we have created not only the hook package directory, but also a `src` directory inside it. This is a typical setup for local packages, where the `src` directory provides the source code.

Now let's create the `Config.in` file for the hook package (do not copy this example as it is _not_ correctly formatted).

```
config BR2_PACKAGE_BUILDHOOK_GREETER
    bool "Greet the user"
    help
      Greet the user using a configurable message.

if BR2_PACKAGE_BUILDHOOK_GREETER

config BR2_PACKAGE_BUILDHOOK_GREETER_MESSAGE
    string "Greeting message"
    default "Welcome to rxOS!"
    help
      Greeting message printed to the user during
      boot.

endif # BR2_PACKAGE_BUILDHOOK_GREETER
```

We will also register this configuration. We edit the `rxOS/Config.in` file and modify the "System configuration" menu.

```
menu "System configuration"
    ...
    source "$BR2_EXTERNAL/local/boothook-greeter/Config.in"
    ...
endmenu
```

Before we can write the matching makefile, we need the hook script itself. In the `src` directory, we create a file called `greeter.sh` that looks like this:

```
#!/bin/sh
MESSAGE="%MSG%"
echo "
**********************************
$MESSSAGE
**********************************
"
```

The `%MSG%` is a placeholder that will be populated later in the makefile.

Note: The `%NAME%` syntax is just a convention. You can use whatever you like for your placeholders, and also any technique for populating them. This is just an example of how we do it.

We also need a initramfs list file. We will assume that, as per convention, the hook script will be installed in `$(BINARIES_DIR)/initramfs`. The list will be saved as `src/init.cpio.in`.

```
file /hook-greeter.sh %BINDIR%/initramfs/greeter.sh
```

Now we have all the files we need, so we can proceed to write the makefile.

Now let's create the makefile. The makefile is called `buildhook-greeter.mk` and should be saved in the package directory.

```
################################################################
#
# buildhook-greeter
#
################################################################
```

```
BUILDHOOK_GREETER_VERSION = 1.0
BUILDHOOK_GREETER_LICENSE = GPLv3+
BUILDHOOK_GREETER_SITE = $(BR2_EXTERNAL)/local/buildhook-greeter/src
BUILDHOOK_GREETER_SITE_METHOD = local

### (More stuff here soon...) ###

$(eval $(generic-package))
```

Thus far, it's a typical Buildroot generic package makefile. (Assume that we will keep the code above and below the 'More stuff' comment in the snippets that follow.)

We first need to prepare the user-specified message. When a value is specified in the Buildroot's menuconfig, it comes with double-quotes around them so those need to be stripped out.

```
BUILDHOOK_GREETER_MSG = $(call qstrip,$(BR2_BUILDHOOK_GREETER_MESSAGE))
```

Now we define the install commands that will install the initramfs list and the hook script in appropriate places and replace the placeholders with appropriate values.

```
define BUILDHOOK_GREETER_INSTALL_TARGET_CMDS
    install -Dm644 $(@D)/init.cpio.in \
        $(BINARIES_DIR)/initramfs/greeter.cpio.in
    sed -i 's|%BINDIR%|$(BINARIES_DIR)' \
        $(BINARIES_DIR)/initramfs/greeter.cpio.in
    install -Dm644 $(@D)/greeter.sh \
        $(BINARIES_DIR)/initramfs/greeter.sh
    sed -i 's|%MSG%|$(BUILDHOOK_GREETER_MSG)' \
        $(BINARIES_DIR)/initramfs/greeter.sh
endef
```

Finally, we need to register the cpio list file:

```
INIT_CPIO_LISTS += greeter.cpio.in
```

This concludes our makefile. Now we can test whether it all works.

```
$ make boothook-greeter-rebuild
```

When we're done, we can take a look at `out/images/initramfs` directory and inspect the results. If everything is alright, we can include the hook in initramfs by enabling "User-provided options > System configuration > Greet the user" option and setting the message.

Adding files to the initial RAM filesystem

If the hook script requires files that are not present in the stock initial RAM filesystem image, additional files can be added via the cpio list file.

Modifying the hooks

When the hook is modified, and it's time to rebuild the image, we need to do it like so:

```
$ make boothook-<hookname>-rebuild rebuild
```

Simply doing `make rebuild` will not update the hook as the package is marked as already installed.

Root filesystem overlays

rxOS supports customizing the base image using root filesystem overlays (henceforth we'll refer to them simply as just 'overlays'). Overlays are SquashFS images that contain a set of files and directories that either augment or override the base root filesystem contents.

The benefit of using overlays as opposed to modified root filesystem images are:

- ability to receive OTA updates meant for the base root filesystem only while retaining the customization intact
- simpler build process as the full build environment is not needed for most simple overlays
- OTA update for the overlay itself does not consume too much bandwidth as overlays are typically small

There is no limit to the number of overlays that can be added to the system, though one should be mindful about RAM usage as each overlay is loop-mounted.

Creating an overlay image

To create an overlay image, we first prepare a directory with the image contents, and then convert it using squashfs-tools. Only LZ4-compressed images are supported at the moment.

As an example, we will prepare an overlay that contains a single text file.

First prepare the work directory:

```
$ mkdir overlay && cd overlay
```

Next we create the directory where we will keep our text file and the text file itself:

```
$ mkdir -p opt/doc
$ echo 'Hello world!' >> opt/doc/hello.txt
```

Finally, we adjust the ownership of the directories:

```
$ sudo chown -R 0:0 opt/doc
```

Note: We are using numeric IDs for the user and group. Since user and group names on your system may not map to the same IDs on rxOS, you should stick to using 0 for root, and 1000 for the outernet user.

We are now ready to create the SquashFS image:

```
$ cd ..  # Exit overlay directory
$ mksquashfs overlay/ overlay-hello-1.0.sqfs -comp lz4 -Xhc
```

The output file name must be named `overlay-<name>-<version>.sqfs` because that is the pattern that the init script looks for. The `name` should not contain any dashes or spaces. The `version` can be in the X.Y or X.Y.Z format and the following suffixes are supported:

- aN - alpha version N
- bN - beta version N
- rcN - release candidate N

Here are some examples of valid version numbers:

- 1.0a2 - second alpha version for 1.0 release

- 2.0b1 - first beta version for 2.0 release
- 3.1 - third major, fist minor release
- 1.3.002 - first major, third minor, second patch release
- 5.1rc2 - second release candidate for 5.1 release

Adding binary executables to overlays

Binary executables have to be compiled for the rxOS target platform(s). While you can create binaries any way you prefer, the simplest approach is to use the rxOS build itself to generate the binary files.

The advantage of this method is that the build-related tooling is already set up. Disadvantages are that it is time consuming as it requires a complete rxOS build and that you are limited to packages that are in the build (or you have to create new ones for 3rd party packages that are not available in the build).

Warning: This method uses the buildroot package infrastructure to create the binaries. Because buildroot packages do not maintain a list of files that belong to them, if a package you wish to compile overwrites a file from another package, the overwritten file will be competely removed from the output directory. If this happens, your build will be left in an inconsistent state, and you will need to rebuild the original package to which the overwritten file belongs, or, if you are not sure which package you need to rebuild, do a clean rebuild of the entire project.

First complete a rxOS build itself using the git tag for the version you wish to target. Next, build only the package you are interested in putting in your overlay. In this example, we will add htop:

```
$ make -s htop
>>> htop 1.0.3 Extracting
>>> htop 1.0.3 Patching
>>> htop 1.0.3 Updating config.sub and config.guess
>>> htop 1.0.3 Configuring
>>> htop 1.0.3 Autoreconfiguring
libtoolize: putting auxiliary files in '.'.
libtoolize: copying file './ltmain.sh'
....
>>> htop 1.0.3 Patching libtool
...
>>> htop 1.0.3 Building
...
>>> htop 1.0.3 Installing to target
 /usr/bin/mkdir -p '/home/hajime/code/rxos/out/target/usr/bin'
 /usr/bin/mkdir -p '/home/hajime/code/rxos/out/target/usr/share/applications'
 /usr/bin/mkdir -p '/home/hajime/code/rxos/out/target/usr/share/pixmaps'
 /usr/bin/mkdir -p '/home/hajime/code/rxos/out/target/usr/share/man/man1'
...
```

Note: You don't have to (and you should not) select the package in the menu. Making the target that matches the package name will build that package even if it's not selected.

Once the package has finished building, we will collect the new files. To do this we will use the `newfiles.sh` script:

```
$ tools/newfiles.sh path/to/overlay
usr/bin/htop
```

```
usr/share/pixmaps/htop.png
usr/share/applications/htop.desktop
usr/share/man/man1/htop.1
```

Now the package files are moved to the overlay directory. The list of files shown in the output is the list of files that are copied to the overlay. Some of these files are stripped afterwards: in particular, the pixmaps, applications, and man directories will be stripped.

Finally, we need to dirclean the package to reset it to unbuilt state:

```
$ make -s htop-dirclean
```

The last step is optional, but we do it just in case we change our minds later and decide to make the package part of the build (otherwise buildroot will think the package is already built and won't rebuild it).

Booting with an overlay

To create an image that includes overlays, put them in `out/images/sdcard-extras` directory for SD card builds (e.g., Raspberry Pi), and `out/imags/overlays` for NAND builds (e.g., CHIP).

To install an overlay to a running system, upload the overlay to the receiver, and then:

```
$ sudo chbootfsmode
$ sudo mv <path/to/overlay> /boot
$ sudo chbootfsmode
$ sudo reboot
```

Creating update packages for overlays

The `tools` directory contains a script called `mkoverlaypkg.sh`. This script will create an OTA update `.pkg` file for any overlay images found in `out/images/overlays`. Run the script with `-h` flag to see the options it supports.

The generated overlay files have the following naming convention:

```
rxos-<platform version>-overlay-<name>-<version>-<timestamp><suffix>.pkg
```

- `platform version` can be a version of a rxOS release (e.g., v1.0) or `any`. If the version is specified, the update package can only be installed on that particular version of rxOS.
- `name` is the overlay name, and only overlays that have the same name that are *already installed* are going to be updated by the generated upate package
- `version` is the overlay version, and if version check is enabled (see `suffix` below), only overlays that are newer than the already installed overlay are upgraded
- `timestamp` is a timestamp in local time, when overlay package was created
- `suffix` can be either a blank string or `nv`, for non-version-checking package

Setup scripts

If the "User-provided options > System configuration > Perform initial system setup" option is enabled, a `S00setup` init script is installed in the target system. This script runs the scripts found in `/etc/setup.d` during init, allowing the developer to perform arbitrary tasks during init.

The setup scripts must have an exectuable flag and must have a `.sh` extension (even if it's not a shell script!). When a script is run, a log file is created that is named so as to match the script name: `/var/log/setup/<scriptname>.sh.log`. Any messages from the setup script are logged in that file. The log file is kept only if the script's exit code is not 0. No output from the script is echoed to console.

One example of a setup script is the `persist.sh` which is installed by enabling "User-provided options > System configuration > Persistent system configuration" option.

Troubleshoting

This section contains the information related to finding and fixing problems with the rxOS build as well as the running rxOS operating system.

External storage issues

This section describes possible issues related to external storage support, and also provides general troubleshooting tips.

Storage does not mount

If storage simply does not mount, ensure that it is one of the supported formats (see *Using external storage devices*) and that the files are accessible on a computer. If your USB stick or memory card is factory-formatted, check that it is FAT32 and not exFAT (exFAT is common with large capacity sticks and cards, larger than 64GB).

For NTFS filesystem, there is no integrity check on mount, so ensure that the disk is checked on a Windows computer prior to use.

FSCK*.REC files appearing on storage after use

The integrity check for FAT32 system may create `FSCK*.REC` files (where `*` is a sequence of four digits). These files represent recovered orphan file data. It is usually safe to remove them.

External storage files are not present in the file list

Files not appearing shortly after plugging in an external storage is normal. External storage devices have to be scanned for new files, and this scan may take up to several minutes. The time it takes to scan the disk depends on such factors as storage device perofrmance and the number of files (not their size).

If the new files do not appear even after an hour of waiting, try restarting the receiver.

Warning: Dot not remove and reattach the disk multiple times in in quick succession as this will cause multiple scanning to be triggered and will take even longer to index the disk.

Wrong partition mounted

Currently, only one partition can be used from multi-partition disk, and this is the last partition with supported partition format (see *Using external storage devices*).

General troubleshooting

This subsection contains instructions useful for general troubleshooting of storage-related issues.

Testing the hotplug script

The hotplug script, although a shell script, is not meant to be run from the shell. It is executed by udev, and expects to see some of the udev environment variables. However, it is still possible to run the script manually by simulating the udev environment.

The following environment variables are expected:

- `ACTION`: 'add' or 'remove', tells the script whether the device was attached or detatched
- `DEVNAME`: device node (e.g., /dev/sda1)
- `ID_FS_TYPE`: disk format (vfat, ntfs, ext2, ext3, or ext4)
- `ID_BUS`: must be 'usb'

Here is an example simulating a hot-plug event for `/dev/sdb1` device with `ntfs` disk format:

```
$ sudo su
Password: ********
# ACTION=add DEVNAME=/dev/sdb1 ID_FS_TYPE=ntfs ID_BUS=usb \
    /usr/sbin/hotplug.storage
```

Troubleshooting storage issues

If storage does not appear to be used after plugging in, it is recommended that integrity check is performed on a computer.

Restarting the receiver may also help in some cases.

The system logs at `/var/log/messages` contain messages with more information about the nature of failure. To get the storage hotplug logs, log in using SSH (see *Remote shell access*) and execute the following command:

```
$ grep hotplug.sd /var/log/messages
Jan  1 00:21:21 rxos user.notice hotplug.sda: Handling hotplug even for /dev/sda
Jan  1 00:21:21 rxos user.notice hotplug.sda: Attempting to use iso9660 disk /dev/sda
Jan  1 00:21:21 rxos user.notice hotplug.sda: iso9660 is not a supported filesystem.
Jan  1 00:21:21 rxos user.notice hotplug.sda1: Handling hotplug even for /dev/sda1
Jan  1 00:21:21 rxos user.notice hotplug.sda1: Attempting to use vfat disk /dev/sda1
Jan  1 00:21:22 rxos user.notice hotplug.sda1: Checking disk integrity
Jan  1 00:21:22 rxos user.notice hotplug.sda1: Mounting with options: 'utf8'
Jan  1 00:21:22 rxos user.notice hotplug.sda1: Doing a trial mount on /mnt/sda1
Jan  1 00:21:22 rxos user.notice hotplug.sda1: Final mount to /mnt/external
Jan  1 00:21:22 rxos user.notice hotplug.sda1: Redirecting ONDD to external storage
Jan  1 00:21:22 rxos user.notice hotplug.sda1: Refreshing file index
```

Appendices

This section contains information that does not fit into other sections.

Appendix: Source tree layout

- `buildroot/`: Buildroot submodule
- `docs/`: Documentation
- **`rxos/`: External board directory**
  - **`configs/`: Build configuration**
    - `busybox.config`: Busybox default configuration
    - `config.txt`: Raspberry Pi bootloader configuration
    - `rxos_defconfig`: rxOS build defaults
    - `rxos_kernel_defconfig`: Linux kernel defaults
    - `users`: User/group list
  - `installer/`: Files used to create the update package's installer
  - `local/`: Packages with local source code (mostly system configs)
  - `misc/`: Miscellaneous files (d'oh!)
  - `package/`: Custom software packages submodule
  - `patches/`: Custom patches
  - `rootfs/`: Rootfs overlay
  - `scripts/`: Build hook scripts
  - `support/`: Various build hooks
  - `Config.in`: External board config
  - `external.mk`: External boad makefile
  - `local.mk`: Board-specific overrides
- `Makefile`: Main makefile
- `README.rst`: README file

Appendix: Make targets

The following table describes the available make targets:[1]

[1] Targets described here are custom targets for rxOS build. Full set of Buildroot's own `make` targets are also available.

target	function
version	Get current version
build[2]	Build both SD card image and update package
menuconfig	Bring up Buildroot configuration menu
linuxconfig	Bring up Linux kernel configuration menu
busyboxconfig	Bring up Busybox configuration menu
saveconfig	Save all configuration
config	Load default configuration (overwrites any modifications)
rebuild	Rebuild the rootfs and recompile linux with initramfs
rebuild-with-linux	Completely rebuild the linux kernel, DTB, and rootfs
rebuild-everything	Rebuild everything except host tools/libs
clean-rootfs	Partial cleanup (useful when trying to apply small modifications)
clean-linux	Partial cleanup (useful when linux configuration changes)
clean-deep	Clean everything except host tools/libs
clean	Complete cleanup (also removes any unsaved modifications)
print-post-script-args	Print the arguments that would be passed to the post-build and post-image hooks.
manual	Build the HTML manual and output it to `docs/build/html`.

Appendix: Updating rxOS manually

rxOS firmware can be updated manually. There are three ways to update the rxOS firmware:

- Create a new SD card
- Copy the updated images to the SD card
- Update using shell access

When updating individual files, commonly the following three files are updated:

- `kernel.img` - kernel image
- `root.sqfs` - root filesystem image (applications)
- `backup.sqfs` - a backup copy of the root filesystem image (usually updated at the same time the root filesystem image itself is updated)

Here we will discuss the latter two options.

Copy updated images to SD card

The kernel image (`kernel.img`) and the root filesystem image (`root.sqfs`) can be updated by opening the SD card on a computer and replacing the matching files on the card.

When replacing `root.sqfs`, please note that a copy of the file is named `backup.sqfs`. Ideally we want to replace `backup.sqfs` with a copy of `root.sqfs` as well.

> **Warning:** Make sure the card is safely removed (unmounted) from the computer. Failing to do so may result in partial writes and factory image booting instead of the updated one.

[2] Default target when invoking `make`

Update using shell access

If direct access to the receiver's SD card is impossible, remote shell access can be used as an alternative method of updating. To update remotely, first use scp to transfer the files to the receiver. Note that only around 50MB of files can be stored at a time, so we may need to update files one by one. (For more information about remote shell access in general, see *Remote shell access*.)

Note: To check the available space, the following command can be used: `df -h | grep overlay | awk {print $4}`

The boot partition is read-only by default. To make it read-write, we first run this command:

```
$ sudo chbootfsmode
Password: ********
Changing /boot to read-write mode.
```

Next we scp the files to the receiver.

Note: rxOS only supports scp and not sftp, therefore programs like FileZilla cannot be used. PuTTY users should use the `-scp` option when using `pscp.exe`.

Using PuTTY as an example:

```
C:\> pscp -scp root.sqfs outernet@<IP address>:
outernet@<IP address>'s password: ********
root.sqfs       | 48548kb | 754.2 kB/s | ETA 00:00:00 | 100%
```

Now we can copy the file to the boot directory:

```
$ sudo mv root.sqfs /boot/
Password: ********
mv: can't preserve ownership of '/boot/root.sqfs': Operation not permitted
```

The error message in the example is harmless and can be ignored.

We repeat the last two steps with any other files we may want to update. Finally:

```
$ sync && sudo reboot
```

Wait for the receiver to reboot.

Appendix: Creating a chrooted build environment under Arch Linux

Because the rxOS build depends on a large number of complex software packages, and Arch Linux is a rolling release distribution, discrepancies between the package versions installed on developers' machines may lead to random build failure for some. To ensure maximum compatibility, we can set up a build environment in a chroot and fix the package versions to known good ones.

A chroot is more or less a full Linux distro within a distro (sans the kernel, init scripts, bootloader, and things you do not need for bootstrapping). It is created in an arbitrary directory within our install, and can use different versions of software as if they were installed on the host system.

Arch Linux makes it easy to create the chrooted environment by making the scripts normally used during installation available in form of packages.

Requirements

To build the chrooted environment, you will need a working Arch Linux install, and the following packages:

- `arch-install-scripts`
- `devtools`

Creating the chroot

We will first pick a directory where we will keep our chroot(s). Let's call this directory `chroots` for simplicity:

```
$ mkdir chroots
```

Once we have the directory, we create the actual chroot directory within it:

```
$ mkarchroot chroots/buildroot base
```

The `base` argument is the name of the package group we want installed in the chroot. Although we can install any number of packages, we won't do it at this stage because we want to downgrade all installed packages to a known good version. At this step, we merely want to install packages that will allow us to do so later.

Next we need to edit the `mirrorlist` file *within the chroot* to facilitate the downgrade.

```
$ echo 'Server = https://archive.archlinux.org/repos/2016/02/19/$repo/os/$arch' \
    > chroots/buildroot/etc/pacmand.d/mirrorlist
```

Once the mirrorlist is edited, we can enter the chroot, remove unnecessary packages and downgrade the packages we need:

```
$ sudo arch-chroot chroots/buildroot /bin/bash
# pacman -Rncs linux linux-firmware systemd
# pacman -Syyuu
```

Note: If you use a terminal emulator that isn't quite xterm-compatible, you may need to install terminfo files for your terminal emulator within the chroot. For urxvt, for example, you need to install `rxvt-unicode-terminfo` package. Neglecting to do so will result in weird terminal behavior such a Backspace not echoing correctly.

Next we install the build prerequisites:

```
# pacman -S base-devel python2 git mercurial bc unzip rsync wget cpio
```

Once everything is installed, we can remove the package cache to recover disk space:

```
# pacman -Scc
```

Creating the unprivileged user

There is no need, nor it is desirable, to perform the builds as root. Therefore, we need a normal user account to use while building. Inside the chroot we run the following command:

```
$ useradd -Umk /etc/skel <USERNAME>
```

Making development files available to the chroot

While inside the chrooted environment, we cannot access any files on the host system. Since it is wasteful to clone the code inside the chroot, install all the development tools, and otherwise bloat the chroot, we will make the development files (local git repository) available within the chroot. This allows us to user our normal environment to work on the files (edit, etc), while using the chroot for the actual build.

Since symlinking to location outside the chroot does not work, we will use a bind mount instead. From the host system:

```
$ sudo mount --bind /path/to/local/repo \
    chroots/buildroot/home/<USERNAME>/rxos
```

Building

Now whenever we want to build, we enter the chroot and build as the unprivileged user:

```
$ sudo arch-chroot chroots/buildroot /bin/bash
# su <USERNAME>
$ cd ~/rxos
... build commands ...
```

Appendix: C.H.I.P. NAND storage characteristics

C.H.I.P. has built-in NAND flash storage mounted on the top side of the board. It's an 8GB SK-hynix-branded SLC NAND chip.

NAND characteristics

The following table contains information returned by executing `nand info` in the U-Boot shell.

Capacity	8589934592 B (8 GiB)	0x200000000
Erase block size	4194304 B (4 MiB)	0x400000
Page size	16384 B (16 KiB)	0x4000
Subpage size	16384 B (16 KiB)	0x4000
OOB	1664 B	0x680
Options		0x40003200
BBT Options		0x00110000

Information returned by `mtdinfo` is as follows:

```
Type:                           mlc-nand
Eraseblock size:                4194304 bytes, 4.0 MiB
Amount of eraseblocks:          2044 (8573157376 bytes, 8.0 GiB)
Minimum input/output unit size: 16384 bytes
Sub-page size:                  16384 bytes
OOB size:                       1664 bytes
Character device major/minor:   90:8
Bad blocks are allowed:         true
Device is writable:             true
Default UBI VID header offset:  16384
Default UBI data offset:        32768
Default UBI LEB size:           4161536 bytes, 4.0 MiB
Maximum UBI volumes count:      128
```

Additional notes

Normally about a dozen or more bad sections will be found on first boot.

Appendix: Tools for working with compute boards

This appendix lists some of the tools that could come in handy when working with the compute boards (Raspberry Pi and CHIP).

Raspberry Pi tools

- HDMI-HDMI cable for connecting to an HDMI monitor or TV
- DVI-HDMI cable for connecting to a DVI monitor
- USB keyboard
- 5V/2A power adapter with microUSB cable
- spare SD cards (8GB+)
- Cat 5 cable (LAN cable) for connecting the board to a router
- USB stick

CHIP tools

- USB-UART adapter
- male-male jumper (dupont) wire for grounding the FEL pin
- male-male or male-female jumper (dupont) wires (3x) for attaching USB UART adapters (connector type depends on the UART adapter)
- 5V/2A power adapter with stripped connector exposing voltage and ground wires for powering the board via CHG-IN pin
- USB cable with stripped connector for powering the board via CHG-IN pin
- precision tweezers for removing broken-off pins from the pin header
- passive or powered USB hub for connecting multiple external devices
- USB stick