UBports Documentation

Marius Gripsgard

Jan 07, 2022

ABOUT

Welcome to the official documentation of the UBports project!

UBports develops the mobile phone operating system Ubuntu Touch. Ubuntu Touch is a mobile operating system focused on ease of use, privacy, and convergence.

On this website you find guides to *install Ubuntu Touch on your mobile phone*, *use Ubuntu Touch*, *develop Ubuntu Touch apps*, *port Ubuntu Touch to an Android handset* and *learn more about system components*. If this is your first time here, please consider reading our *introduction*.

If you want to help improving this documentation, *the Documentation contribute page* will get you started.

You may view this documentation in the following languages:

- English
- Català
- Français
- Deutsch
- Italiano
- Română
-
- Türkçe
- Español
- Simplified Chinese

CHAPTER

ONE

INTRODUCTION

Our goal is to create a copylefted libre mobile operating system. One you can use, study, change and share; with all.

1.1 About UBports

The project was founded by Marius Gripsgard in 2015 and in its infancy a place where developers could share ideas and educate each other in hopes of bringing the Ubuntu Touch platform to more mobile devices.

After Canonical suddenly announced plans to terminate support for Ubuntu Touch in April of 2017, UBports and its sister projects began work on the source code; maintaining and expanding its possibilities for the future. Today, UBports is a volunteer group, formalised as a charitable foundation.

1.2 About the Documentation

Changes to the documentation are made by the UBports community. It is written in reStructuredText and converted into this readable form by Sphinx, recommonmark, and Read the Docs. Start contributing by checking out the *Documentation contribution document*.

1.3 Attribution

This documentation was heavily modeled after the Godot Engine's Documentation, by Juan Linietsky, Ariel Manzur and the Godot community.

CHAPTER

TWO

PROCESSES

This section of the documentation details standardized processes for different teams.

Note: The process definitions are still a work in progress and need to be completed by the respective teams.

2.1 Issue-Tracking Guidelines

This document describes the standard process of dealing with new issues in UBports projects. (Not to be confused with the *guide on writing a good bugreport*.)

2.1.1 Where are bugs tracked?

Since quality assurance depends heavily on community effort, issues are tracked where users expect them, instead of separated by repository. This means issues of almost all distributed components (as with the system-image) are tracked in the Ubuntu Touch tracker. An exception to this are click-apps, which can be updated independently through the OpenStore.

Most other repositories track issues locally. You will find out whether a repository uses its own tracker or not in its README.md file. Repositories that don't track issues locally have their bugtracker turned off.

This page is mainly about the Ubuntu Touch tracker, but most principles apply to other projects as well.

Note: Practical exceptions to purity are to be described in the project's README.md file.

2.1.2 GitHub projects

In the interest of transparency and communication, GitHub projects (Kanban-Boards) are used wherever practical. In case of github.com/ubports/ubuntu-touch, a single project is used for all issues. Projects support filtering by labels, so that only issues belonging to a specific team or ones affecting a specific device can be viewed.

These are the standard columns:

- **None (awaiting triage)**: Issue approved by a member of the QA team awaiting review from the responsible development team. If a bug, instructions to reproduce are included in the issue description. If a feature request, it has passed a primary sanity check by the QA team, but not yet been accepted by the responsible development-team.

- **Accepted**: Issue accepted by the responsible development-team. If a bugreport, the team has decided it should be fixable and accept responsibility. If a feature request, the team thinks it should be implemented as described.
- **In Development**: A patch in development. Usually means a developer is assigned to the issue.
- **Quality Assurance**: A completed patch passing initial testing. The QA team will review it and provide feedback. If problems are found, the issue is moved back to "Accepted".
- **Release Candidate**: A patch passing QA, ready for release. In case of DEB packages included in the system-image, the patch will be included in the next over-the-air update on the *rc* channel, and (provided everything goes well) in the next release of the *stable* channel.
- **None (removed from the project)**: Open issue labeled "help wanted". Community contributions are required to resolve it. If it's closed, either a patch has been released on the stable channel (a comment on the issue should link to the patch) or the issue is rejected (labeled "wontfix").

2.1.3 Labels

All issues — even closed ones — should be labeled to allow use of GitHub's global filtering. For example, these are all of the issues labeled 'enhancement' inside @ubports. Consult the GitHub help pages to learn more about searching and filtering.

List of labels normally used by all repositories:

- **needs confirmation**: The bug needs confirmation and / or further detailing by affected users.
- **bug**: This issue is a confirmed bug. If it's reproducible, reproduction steps are described.
- **opinion**: This issue needs further discussion.
- **enhancement**: This issue is a feature request.
- **question**: This issue is a support request or general question.
- **invalid**: This issue can not be confirmed or was reported in the wrong tracker.
- **duplicate**: This has already been reported elsewhere. Please provide a link and close.
- **help wanted**: This issue is ready to be picked up by a community developer.
- **good first issue**: The report contains instructions or hints required to fix it. It is an excellent place for someone new to learn about the project by fixing a real issue.
- **wontfix**: A bug it does not make sense to fix, since it will probably resolve itself, be too much work, isn't fixable, or an underlying component will soon change.

Additional special labels can be defined. As an example, these are the labels used in the Ubuntu Touch tracker:

- **critical (devel)**: Critical issue only occuring on the *devel* channel is blocking the release of the next *rc* image.
- **critical (rc)**: Critical issue only occuring on the *devel* and *rc* channel is blocking the release of the next stable release. Usually, issues that can not simply be moved to a different release and have the power to postpone the release are labeled this way.
- **device: [DEVICE CODENAME]**: Issue affecting only the specified device(s).
- **team: [TEAM NAME]**: Issue falls under the responsibility of a specific team (HAL, middleware, UI).

Note: If a repository tracking issues locally defines it's own labels, they should be documented in the README.md.

2.1.4 Milestones

Milestones are used for stable OTA releases only. In general, milestones for the work-in-progress OTA and the next OTA are created. The ETA is set once the work on the release starts (that is 6 weeks from start date), but can be adjusted afterwards. Learn more in *release-schedule*.

2.1.5 Assignees

To make it transparent who's working on an issue, the developer should be assigned. This also allows the use of GitHub's global filtering as a type of TODO list. For example, this is everything assigned to mariogrip in @ubports.

Developers are encouraged to keep their list short and update the status of their issues.

2.1.6 Examples

Bug Lifecycle

Note: The same principle applies to feature requests, only they are labeled **enhancement** instead of **bug**. **needs confirmation** is not applicable for feature requests.

- A *user* files a new bug using the issue-template.
- The *QA-Team* labels it **needs confirmation** and tries to work with the user to confirm the bug and add potentially missing info to the report.
- Once the report is complete a **team-label** is added to the issue, the issue will be put on the **awaiting-triage-list** of the project and the label needs confirmation will be replaced with **bug**.
- The affected *Team* triages the issue and either rejects (label **wontfix**, closes and removes from the project) or accepts the issue.
- The team decides whether to fix the issue in-house (move to "Accepted" and assign a team member) or wait for a community developer to pick it up (by labeling it **help wanted**, removing it from the project board and providing hints on how to resolve the issue and further details on how the fix should be implemented if necessary). For non-critical issues trivial to fix, the label **good first issue** can be added as well.
- Once a *developer* is assigned and starts working on the issue, it is moved to "In Development".
- As soon as there is something to show for, the issue is closed and automatically moved to "Quality Assurance" for feedback from the QA team. If necessary, the developer provides hints on how to test the patch in a comment on the issue.
- The *QA-Team* tests the fix on all devices and provides feedback to the developer. If problems are found, the issue is re-opened and goes back to "Accepted", otherwise it is moved to "Release Candidate" for inclusion in the next release.
- If not done already, the issue is added to the next milestone, which once released removes the issue from the project board.

2.2 Release Schedule

OTA updates usually follow this rhythm:

- devel: daily builds
- rc: weekly if no critical issue exists in the devel channel
- stable: every six through eight weeks, if no critical issue exists in the rc channel

This is not a definitive cycle. Stable releases are ready when they're ready and should not introduce new bugs or ship very incomplete features. Since the OTA update can not be released before all critical issues are closed, the ETA might have to be moved by making an educated guess for when all the issues can be handled. If there are too many issues added to a milestone, they are either removed or added to the next milestone.

2.3 repo.ubports.com

This is the package archive system for UBports projects. It hosts various PPAs containing all the DEB-components of Ubuntu Touch.

2.3.1 Repository naming convention

Native packages

Native packages (like https://github.com/ubports/system-settings) are repositories that contain a `debian/` folder **with** the source used to create the DEB source package (`.dsc`).

The name of the Debian source package generated from the repository and the name of the Git-repository should be the same.

Packaging repositories

Packaging repositories (like https://github.com/ubports/pulseaudio-packaging) contain a `debian` folder **without** the source used to create the Debian source package (`.dsc`). They also contain instructions to tell `debhelper` how to get the sources used to create the source package.

The repository should have the name of the source package it generates with `-packaging` appended to the end. For example, a packaging repository that generates a source package called `sandwich` would be called `sandwich-packaging`.

2.3.2 Creating new PPAs

New PPAs can be created dynamically by the CI server using a special *git-branch naming convention*. The name of the branch translates literally to the name of the PPA: `https://repo.ubports.com/dists/[branch name]`

Non-standard PPAs (i.e. not `vivid`, `xenial`, or `bionic`) are kept for three months. If they need to be kept for longer, a file with the name `ubports.keep` containing a date in the form of in the form of `YYYY-MM-dd` can be used to decide how long to keep the PPA in the repository for. If this file is empty, the PPA will be kept for two years after the last build.

2.4 Branch-naming convention

Our branch-naming convention ensures software can be built by our CI and tested easily by other developers.

Every Git repository's README file should state which branch-naming convention is used and any deviations from the norm.

2.4.1 Click-Packages

Software exclusively distributed as a click-package (and not also as a DEB) only uses one `master` branch that is protected. Separate temporary development branches with arbitrary descriptive names can be created and merged into master when the time comes. Ideally Git tags or GitHub releases should be used to mark and archive milestones in the development history.

2.4.2 DEB Packages

To make most efficient use of our CI system, a special naming convention for Git branches is used.

For pre-installed Ubuntu Touch components, DEB-packages are used wherever possible. This includes Core apps, since they can still be independently updated using click-package downloads from the OpenStore. This policy allows making use of the powerful Debian build-system to resolve dependencies.

Every repository using this convention will have branches for the actively supported Ubuntu releases referenced by their codenames (`bionic`, `xenial`, `vivid`, etc.). These are the branches built directly into the corresponding images and published on *repo.ubports.com*. If no separate versions for the different Ubuntu bases are needed, the repository will have one `master` branch and the CI system will still build versions for all actively supported releases and resolve dependencies accordingly.

Branch-extensions

To build and publish packages based on another repository, an extension in the form of `xenial_-_some-descriptive_extension` can be used. The CI system will then resolve all dependencies using the `xenial_-_some-descriptive_extension` branch of other repositories or fall back to using the normal `xenial` dependencies, if it doesn't exist. These special dependencies are not built into the image, but still pushed to *repo.ubports.com*.

Multiple branch extensions can be chained together in the form of `xenial_-_dependency-1_-_dependency-2_-_dependency-3`. This means the CI system will look for dependencies in the following repositories:

```
xenial
xenial_-_dependency-1
xenial_-_dependency-1_-_dependency-2
xenial_-_dependency-1_-_dependency-2_-_dependency-3
```

Note: There is no prioritization, so the build system will always use the package with the highest version number, or the newest build if the version is equal.

Dependency-file

For complex or non-linear dependencies, a `ubports.depends` file can be created in the root of the repository to specify additional dependencies. The branch name will be ignored if this file exists.

```
xenial
xenial_-_dependency-1_-_dependency-2_-_dependency-3
xenial_-_something-else
```

Note: The `ubports.depends` file is an **exclusive list**, so the build system will not resolve dependencies linearly like it does in a branch name! Every dependency has to be listed. You will almost always want to include your base release (i.e. `xenial`).

CHAPTER

THREE

INSTALL UBUNTU TOUCH

Installing Ubuntu Touch is easy, and a lot of work has gone in to making the installation process less intimidating to the average user. The UBports Installer is a nice graphical tool that you can use to install Ubuntu Touch on a supported device from your Linux, Mac or Windows computer. For more experienced users, we also have manual installation instructions for every device on the devices page.

Warning: If you're switching your device over from Android, you will not be able to keep any data that is currently on the device. Create an external backup if you want to keep it.

You can Download the UBports Installer from GitHub. The following packages are available:

- Windows: `ubports-installer-<version-number>.exe`
- macOS: `ubports-installer-<version-number>.dmg`
- Ubuntu or Debian: `ubports-installer-<version-number>.deb`
- Other Linux distributions (snap): `ubports-installer-<version-number>.snap`
- Other Linux distributions (AppImage): `ubports-installer-<version-number>.AppImage`

Start the installer and follow the on-screen instructions that will walk you through the installation process. That's it! Have fun exploring Ubuntu Touch!

If you're an experienced android developer and want to help us bring Ubuntu Touch to more devices, visit the *porting section*.

Note: Please do **not** start the installer with `sudo`. It is meant to be run as a normal user, not as root. Using the installer with sudo will mess up permissions in the installer's cached files and lead to problems down the road. If the installer does not recognize your device, check the troubleshooting information below.

3.1 Troubleshooting

If you need help, you can join UBports' support channels on telegram or matrix or ask a question in the forum or on askubuntu. If you believe that the UBports Installer is not working correctly, you can also file a new issue to help us solve the problem.

3.1.1 Connection lost during installation

Try a different USB cable and a different USB port on your computer. Cheap and old cables tend to lose connection during the installation.

3.1.2 Windows Defender prevents Installer from starting

We have contacted Microsoft about this problem, but they seem to think it's enjoyable. To dismiss the warning, click on "More Information", then select "Run anyway".

3.1.3 Device not detected

Virtual Machine

If you're running the installer in a virtual machine, make sure the virtual machine is allowed to access the USB port.

Windows ADB drivers

Install the universal adb driver and reboot your computer.

Fairphone 2 Vendor-ID

Run `echo 0x2ae5 >> ~/.android/adb_usb.ini` in the terminal and restart the installer. This does not affect the snap package.

Missing udev-rules

If the device is not detected on Linux and you are not using the snap package, then you might be missing some **udev-rules**.

1. See if `/etc/udev/rules.d/51-android.rules` exists and contains the rules below. If not, add them to the file and run `sudo service udev restart` or `sudo udevadm control --reload-rules && udevadm trigger`.

```
SUBSYSTEM=="usb", ATTRS{idVendor}=="0e79", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0502", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0b05", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="413c", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0489", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="091e", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="18d1", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0bb4", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="12d1", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="24e3", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="2116", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0482", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="17ef", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="1004", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="22b8", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0409", MODE="0666", GROUP="plugdev"
```

(continues on next page)

(continued from previous page)

```
SUBSYSTEM=="usb", ATTRS{idVendor}=="2080", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0955", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="2257", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="10a9", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="1d4d", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0471", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="04da", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="05c6", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="1f53", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="04e8", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="04dd", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0fce", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0930", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="19d2", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="2ae5", MODE="0666", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="2a45", MODE="0666", GROUP="plugdev"
```

3.1.4 Other issues

If the troubleshooting tips do not work, you might also try following the manual installation instructions for your device on the devices page.

3.2 Install on legacy Android devices

While the installation process is fairly simple on most devices, some legacy Bq and Meizu devices require special steps. This part of the guide does not apply to other devices.

Note: This is more or less uncharted territory. If your device's manufacturer does not want you to install an alternative operating system, there's not a lot we can do about it. The instructions below should only be followed by experienced users. While we appreciate that lots of people want to use our OS, flashing a device with OEM tools shouldn't be done without a bit of know-how and plenty of research.

Meizu devices are pretty much stuck on Flyme. While the MX4 can be flashed successfully in some cases, the Pro5 is Exynos-based and has its own headaches.

Warning: BE VERY CAREFUL! You are responsible for your own actions!

1. Disconnect all devices and non-essential peripherals from your PC. Charge your device on a wall-charger (not your PC) to at least 40 percent.
2. Download the Ubuntu Touch ROM for your device and extract the zip files:
   - Bq E4.5 (*krillin*)
   - Bq E5 HD (*vegetahd*)
   - Bq M10 HD (*cooler*)
   - Bq M10 FHD (*frieza*)
   - Meizu MX4 (*arale*)

3. Download the latest version of SP flash tool (*aka MTK flash tool*).
4. Extract the zip files
5. Open a Terminal, enter the SP flash tool directory and run it with `sudo ./flash_tool.sh`.

Note: On Ubuntu 17.10, there are issues with flash_tool loading the shared library 'libpng12', so this can be used as a workaround:

```
wget -q -O /tmp/libpng12.deb http://mirrors.kernel.org/ubuntu/pool/main/libp/libpng/
↪libpng12-0_1.2.54-1ubuntu1_amd64.deb \
&& sudo dpkg -i /tmp/libpng12.deb \
&& rm /tmp/libpng12.deb
```

---

6. In the "Scatter-loading File" section, press the "choose" button and select the `*Android_scatter.txt` file from the archive you downloaded in the second step.
7. Choose "Firmware Upgrade" from the drop-down menu.

Warning: If you select DOWNLOAD ONLY rather than FIRMWARE UPGRADE, you will end up with a useless brick rather than a fancy Ubuntu Touch device. Be sure to select FIRMWARE UPGRADE.

Also, after selecting the `*Android_scatter.txt`, the drop-down menu choice is reset to DOWNLOAD ONLY. Be sure to select it after selecting the file.

8. Turn your device completely off, but do not connect it yet.
9. Press the button labeled "Download".
10. Perform a final sanity-check that you selected the "Firmware Upgrade" option, not "Download Only".
11. Make sure your device is off and connect it to your PC. Don't use a USB 3.0 port, since that's known to cause communication issues with your device.
12. Magic happens. Wait till all the progress bars ended.

Note: If the installation fails with a 100% red bar and hangs there (and logs says something like `COM port is open. Trying to sync with the target...`), remove the `modemmanager` package as the one installed is not compatible with SP flash tool, close it and try again from step 4.

---

13. Congratulations! Your device will now boot into a super old version of Ubuntu Touch. You can now use the UBports Installer to install an updated version.

CHAPTER

FOUR

DAILY USE

This section of the documentation details common tasks users may want to do on their Ubuntu Touch device.

4.1 Using location services

Location services in Ubuntu Touch allow apps to access your current location. Using this data, apps can offer turn-by-turn navigation, track your exercise paths, locate public transport near you, and more.

The location services built into Ubuntu Touch are designed to protect your privacy — you choose to permit or deny apps access to this data. Apps may only access your location data when the device screen is on and the app is in use.

4.1.1 Checking if location services are on

The Location Indicator is displayed whenever location services are on. Normally, this indicator will appear to be dark.

Fig. 1: The Location Indicator, showing location on but not in use.

If an app is accessing location data, the Location Indicator brightens.

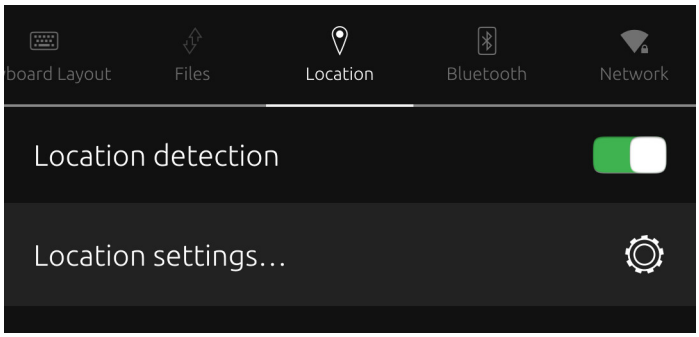
Fig. 2: The Location Indicator, showing location on and in use.

4.1.2 Accessing location services settings

There are several ways to access settings related to location services in Ubuntu Touch.

Open location quick settings

The Location Indicator contains quick settings related to location services, including a switch to turn services on or off.

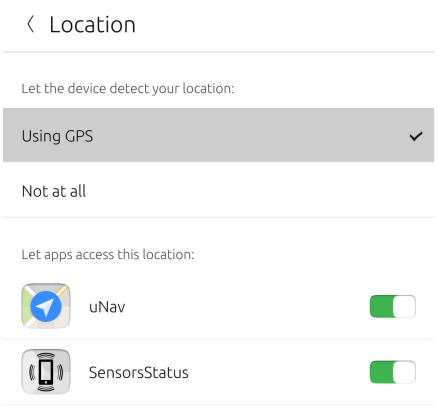


To access these settings, press the Location Indicator and pull down toward the bottom of the screen.

If location services are off, you can access the location quick settings by pressing on any other indicator and pulling down. Then, scroll through the icons at the top of the screen to find Location. Select "Location to open the quick settings.

Open location settings in the settings app

The system settings app can be used to control location services. There are more controls in the system settings app than in the Location Indicator.



There are two ways to access location services controls in the system settings app.

Quick settings

Open location quick settings, then select `Location settings...`.

System settings app

Launch the `System Settings` app. Select `Security & Privacy`, then select `Location`.

4.1.3 Enabling or disabling location services

If location services are on, apps you have granted access to your location data may request your current location. Location services are on by default on Ubuntu Touch.

If location services are off, apps will not be able to access your current location.

There are several ways to turn on or off location services in Ubuntu Touch.

Quick settings

Open location quick settings. Select the `Location detection` switch to turn on or off location services.

System settings app

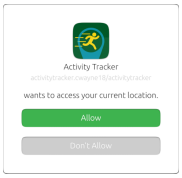
Open location settings in the settings app. Under `Let the device detect your location:`, select the appropriate option:

- `Using GPS` turns on location services.
- `Not at all` turns off location services.

4.1.4 Allowing apps access to location data

Ubuntu Touch allows you to allow or deny apps access to your location data. This decision is presented the first time the app tries to access your current location. You can change your decision later.

An app will show the location permission request the first time it tries to access your current location.



Select `Allow` to give the app access to your current location. Select `Don't Allow` to deny the app access to your current location. Some app features may not work correctly if you select `Don't Allow`.

Changing your permissions

You can change your decision to allow an app access to your location data. This is useful if you have granted an app access to your location but would now like to revoke it, or if you denied access but would like to grant access again.

There are several ways to find these controls.

Location settings

Open location settings in the settings app. All apps that have requested access to your location are listed under `Let apps access this location:`. Toggle the switch next to an app "on" to allow it to access your location data. Toggle the switch "off" to deny access.

App permission settings

Launch the `System Settings` app. Select `Security & Privacy`, then select `App permissions`, followed by `Location`. All of the apps that have requested access to your location are listed. Toggle the switch next to an app "on" to allow it to access your location data. Toggle the switch "off" to deny access.

4.1.5 Using apps with location services

Apps will be able to access your current location after they are granted access to location data. However, there are some things to note while using apps with location services.

Time to first fix

A "fix" is a data point generated by GPS hardware that contains its current location and the expected accuracy of that location. GPS hardware is rated by how long it takes to return one of these data points after the device is activated. This rating is called "Time To First Fix," or TTFF.

A device with a mobile data connection should take one to four minutes to receive its first fix. If location services have not been used for a long time *and* the device does not have a mobile data connection, the first fix can take up to an hour. To prevent this from causing problems for you, follow these instructions:

> **Warning:** Do not leave your device in direct sunlight. (The heat can damage the device.)

1. Turn off `Lock when idle` in the Battery settings
2. Download an app which uses location services and use it to start accessing location data.
3. Leave your device near a window, but out of direct sunlight.

(Turn the screen brightness down to reduce energy consumption.)

After 20-60 minutes, your device should display your current location. You can now close the app and turn on `Lock when idle`. Future attempts to aquire your location should now take 1-4 minutes.

App suspension

Apps are suspended whenever not in the foreground, or when the device is locked. When an app is suspended, it cannot receive location data. For this reason, apps will not be able to track your location whenever they are not in use or the device is locked.

4.2 Run desktop applications

Libertine allows you to use standard desktop applications in Ubuntu Touch.

To install applications you can use the *Settings* interface on your Ubuntu Touch device, or the command-line as described further below.

4.2.1 Manage containers

Create a container

The first step is to create a container where applications can be installed:

Open Settings and scroll down to *Libertine* under the *System* sub-heading.

Tap *Libertine* to get to the *Manage Libertine Containers* sub-menu. Already installed containers will be visible here. Tap the + sign at the upper right to add a container.

The *Container Options* dialog box will open asking you to enter a container name and optionally a password. Confirm your entries by tapping "OK", and the container will be created.

To do this via the command-line:

```
libertine-container-manager create -i CONTAINER-IDENTIFIER
```

You can add extra options such as:

- `-n name` name is a more user-friendly name of the container
- `-t type` type can be either `chroot` or `lxc`. The default is `chroot` , which is compatible with every device.

If the kernel of your device supports it, `lxc` is suggested.

Creation can take a while due to the size of the container (some hundred megabytes).

Note: The `create` command shown above cannot be run directly in the terminal app, due to AppArmor restrictions. You can run it from another device using either an ADB or SSH connection. Alternatively, you can run it from the terminal app using a loopback SSH connection by running this command: `ssh localhost`.

List containers

Use the *Settings Manage Libertine Containers* sub-menu to list all the containers you have created on the device.

You can also use the command-line. Run:

```
libertine-container-manager list
```

Destroy a container

```
libertine-container-manager destroy -i CONTAINER-IDENTIFIER
```

4.2.2 Manage applications

Once a container is set up, the installed applications are available in the app drawer. You can also tap the name of the container in the *Manage Libertine Containers* menu.

You can also list them via the command-line:

```
libertine-container-manager list-apps
```

Install a package

From the apps list in the *Manage Libertine Containers* menu you can tap the + sign to get a dialog box that lets you add a package.

Enter the package name if you know it. Otherwise you can search the archives for a package. You can make this process easier by making the *Lubuntu Software Center* your first package install. Being a desktop app, you will need to use the *Libertine Tweak Tool* in the *Open Store* to make the text large enough to read on a mobile device.

Packages may also be installed via the command-line:

```
libertine-container-manager install-package -p PACKAGE-NAME
```

Remove a package

Remove a package by swiping it to the right from its entry in the package list (in *System Settings* > *Libertine* > *Manage Libertine Containers* > *CONTAINER NAME*). An option to remove it will be revealed.

Or you can remove the package via the command-line:

```
libertine-container-manager remove-package -p PACKAGE-NAME
```

Note: If you have more than one container, you can use the option `-i CONTAINER-IDENTIFIER` to specify which container you want to perform an operation on.

4.2.3 Run desktop application

Run it from the app menu like any other app. To start a graphical app from the command-line, for example `gedit`, run this in a terminal:

```
# ubuntu-app-launch <CONTAINER_ID>_<desktop_file_name>_0.0
ubuntu-app-launch xenial_gedit_0.0
```

4.2.4 Files

Libertine applications have access to these folders:

- Documents
- Music
- Pictures
- Downloads
- Videos

4.2.5 Tips

Locations

For every container you create, two directories will be created:

- A root directory `~/.cache/libertine-container/CONTAINER-IDENTIFIER/rootfs/` and
- a user directory `~/.local/share/libertine-container/user-data/CONTAINER-IDENTIFIER/`

Shell access

There are two options for executing commands inside the container.

The first option is based on `libertine-container-manager exec`. It lets you run your commands as root. The drawback is that the container is not completely set up. So far we know that the folders mentioned above (Documents, Music, ...) are not mounted i.e., the `/home/phablet/` directory is empty. Likewise the directory referenced in `TMPDIR` is not available, which may lead to problems with software trying to create temporary files or directories. You may use this option to installing packages.

To execute a command you can use the following pattern:

```
libertine-container-manager exec -i CONTAINER-IDENTIFIER -c "COMMAND-LINE"
```

For example, run:

```
libertine-container-manager exec -i CONTAINER-IDENTIFIER -c "apt-get --help"
```

To get a shell into your container as `root`, run:

```
libertine-container-manager exec -i CONTAINER-IDENTIFIER -c "/bin/bash"
```

The second option is based on `libertine-launch`. It will execute your commands as user phablet in a completely set up container. You may use this option to modify your files using installed packages.

To execute a command you can use the following pattern:

```
libertine-launch -i CONTAINER-IDENTIFIER COMMAND-LINE
```

For example, run:

```
libertine-launch -i CONTAINER-IDENTIFIER ls -a
```

To get a shell as the user `phablet`, run:

```
DISPLAY= libertine-launch -i CONTAINER-IDENTIFIER /bin/bash
```

Note: When you launch Bash in this way you will not get any specific feedback to confirm being *inside* the container. You can check `ls /` to confirm for yourself you are actually inside the container. The listing of `ls /` will be different inside and outside the container.

Accessing SD card

To access your SD-card or any other part of the regular filesystem from inside your Libertine container you must create a bind mount.

To add a bind mount, use:

```
libertine-container-manager configure -i CONTAINER-IDENTIFIER -b add -p /media/phablet/
↪ID-OF-SD
```

You can also make deep links if you only want parts of your SD-card available in the container. In this case just add the entire path to the directory you want to bind mount:

```
libertine-container-manager configure -i CONTAINER-IDENTIFIER -b add -p /media/phablet/
↪ID-OF-SD/directory/you/want
```

This will not give the container access to any of the directories earlier in the path for anything other than accessing your mounted directory.

In order to use the SD card as extra space for your container, make sure to first format it using ext4 or similar. There is a mis-feature in UDisks2 that mounts SD-cards (showexec), ensuring only files ending in .bat, .exe or .com can be executed from the drive if it is (V)FAT formatted. This has been changed in other distributions, allowing any file to have execute priviliges, but not in Ubuntu. The recommended workaround is to add a udev rule to control how to mount a card with a given ID, but since the udev rules are on the read-only port on Ubuntu Touch, this is not possible.

Shortcuts

If you want, you can add aliases for command-line tools. Add lines like the following ones to your `~/.bash_aliases`:

```
alias git='libertine-launch -i CONTAINER-IDENTIFIER git'
alias screenfetch='libertine-launch -i CONTAINER-IDENTIFIER screenfetch'
```

4.2.6 Background

A display server coordinates input and output of an operating system. Ubuntu Touch does not use X, but a new display server called Mir. This means standard X applications are not directly compatible with Ubuntu Touch. A compatibility layer called XMir resolves this. Libertine relies on XMir to display desktop applications.

Another challenge is that Ubuntu Touch system updates are released as OTA images. As a consequence of this the root filesystem is read-only. Libertine provides a container with a read-write filesystem to allow installation of regular Linux desktop applications.

4.3 Android apps

Anbox is a minimal Android container and compatibility layer to run Android apps on GNU/Linux operating systems such as Ubuntu Touch.

Note: "Computer" refers to another device you connect your Ubuntu Touch device to (via USB here). Your USB-attached computer must have `adb` and `fastboot` installed.

Note: You will need to execute commands on your Ubuntu Touch device (and/or attached computer) to install Anbox and APKs. You can do that by using the terminal application, but it is easier to use `adb shell` or *set up SSH* to access your Ubuntu Touch device from your computer.

4.3.1 Supported devices

Make sure your device is supported:

- Meizu PRO 5
- Fairphone 2
- OnePlus One
- Nexus 5
- BQ Aquaris M10 HD
- BQ Aquaris M10 FHD

4.3.2 How to install

Warning: Installing Anbox is only recommended for experienced users.

Install Anbox kernel

Some devices require you to install a custom Linux kernel to use Anbox. These devices are:

- Meizu PRO 5 (codename: `turbo`, name of the boot partition: `bootimg`)
- BQ Aquaris M10 HD (codename: `cooler`, name of the boot partition: `boot`)
- BQ Aquaris M10 FHD (codename: `frieza`, name of the boot partition: `boot`)

If your device is not in this list, the Anbox kernel was automatically installed when you installed Ubuntu Touch. Please skip ahead to *Run the Anbox installer*.

You will need to repeat these steps after each Ubuntu Touch update. Not doing so can put your Ubuntu Touch device into an unstable state. Only update your Ubuntu Touch device when you have a computer with you to re-flash the modified kernel image.

1. Be sure to have a backup of your device.
2. Open a terminal on your computer and set some device specific variables by running `export CODENAME="turbo" && export PARTITIONNAME="bootimg"`, but replace the part between the quotes respectively with the codename and name of the boot partition for your device. See the above list.
3. Activate *Developer mode* on your device.
4. Connect the device to your computer and run the following commands from your computer (the same terminal you ran the `export` command):

```
wget https://cdimage.ubports.com/anbox-images/anbox-boot-$CODENAME.img
adb shell # connect from your computer to your Ubuntu Touch device
sudo reboot -f bootloader # 'adb shell' will exit after this command, the prompt␣
↪will be back on your computer
sudo fastboot flash $PARTITIONNAME anbox-boot-$CODENAME.img
sudo fastboot reboot
rm anbox-boot-$CODENAME.img
exit
```

Run the Anbox installer

Once your device has the Anbox kernel installed, you can use the Anbox Tool to install the Anbox container.

1. Run `adb shell` from your computer to get a shell on your Ubuntu Touch device.
2. Run the following command on your Ubuntu Touch device: `anbox-tool install`.
3. Follow the on-screen instructions.

Now you're done! You might have to refresh the app drawer (pull down from the center of the screen and release) for the new Android apps to show up.

Note: You now have an ADB server running on your Ubuntu Touch device.

This guide asks you to run some ADB commands, sometimes on your computer, other times on the device itself. Carefully check which device you are on!

You can check that the ADB server is running on your Ubuntu Touch device by entering `adb devices` in its terminal app. You should see something like:

```
phablet@ubuntu-phablet:~$ adb devices
List of devices attached
emulator-5558       device
```

4.3.3 How to install new APKs

- Copy the APK to `/home/phablet/Downloads`. Then run the following from your computer:

```
adb shell # connect from your computer to your Ubuntu Touch device
adb install /home/phablet/Downloads/my-app.apk # This is the adb of your device,␣
↪not your computer
exit
```

- Done! You might have to refresh the apps scope (pull down from the center of the screen and release) for the new Android apps to show up.

4.3.4 Keep your apps up to date

- To keep your apps up to date you can use of F-Droid or ApkTrack. If you want to install any of the above apps you can find them here:
- F-Droid: https://f-droid.org/
- ApkTrack: https://f-droid.org/packages/fr.kwiatkowski.ApkTrack/

4.3.5 How to uninstall apps

- To uninstall apps from the Ubuntu Touch device, run `adb uninstall [APP_ID]` from your computer:

```
adb shell # connect from your computer to your Ubuntu Touch device
sudo mount -o rw,remount /
adb uninstall [APP_ID] # This is the adb of your device, not your computer
exit
```

- Done! You might have to pull down from the app list for the new Android apps to show up.

4.3.6 Access Android storage

The Android storage is located at `/home/phablet/anbox-data/data/media/0`.

- Links to android libraries can be made in their respective XDG counterparts:

```
ln -s ~/anbox-data/data/media/0/Documents ~/Documents/android
ln -s ~/anbox-data/data/media/0/Pictures ~/Pictures/android
ln -s ~/anbox-data/data/media/0/Music ~/Music/android
ln -s ~/anbox-data/data/media/0/Movies ~/Videos/android
```

4.3.7 Troubleshooting

- When you want to install an APK, but get the error `Invalid APK file` that error could also mean "file not found"
  - Check that you typed the filename correctly.
  - If the APK does not reside in the folder you are in when you run adb, you have to specify the full path, e.g. `/home/phablet/Downloads/my-app.apk` instead of just `my-app.apk`

4.3.8 Reporting bugs

Please *report any bugs* you come across. Bugs concerning Ubuntu Touch are reported in the normal Ubuntu Touch tracker and issues with Anbox are reported on our downstream fork. Thank you!

CHAPTER

FIVE

ADVANCED USE

In this section you will find advanced tasks power users can perform on their Ubuntu Touch devices.

Note: Some of these guides involve making your system image writable, which may break OTA updates. The guides may also reduce the overall security of your Ubuntu Touch device. Please consider ramifications beforehand.

5.1 Shell access via ADB

You can put your UBports device into developer mode and access a Bash shell from your PC. This is useful for debugging or more advanced shell usage.

5.1.1 Install ADB

First, you'll need ADB installed on your computer.

On Ubuntu:

```
sudo apt install android-tools-adb
```

On Fedora:

```
sudo dnf install android-tools
```

And on macOS with Homebrew:

```
brew install android-platform-tools
```

For Windows, grab the command-line tools only package from developer.android.com.

5.1.2 Enable developer mode

Next, you'll need to turn on Developer Mode.

1. Reboot your device
2. Place your device into developer mode (Settings - About - Developer Mode - check the box to turn it on)
3. Plug the device into a computer with ADB installed
4. Open a terminal and run `adb devices`.

Note: When you're done using the shell, it's a good idea to turn Developer Mode off again.

If there's a device in the list here (The command doesn't print "List of devices attached" and a blank line), you are able to use ADB. If not, continue to the next section.

ADB shell commands:

```
adb shell - Gives you access to the Linux command-line shell on your device.
adb shell [command] - Runs the specified shell command on your device.
```

Additional Adb commands:

```
adb push [source] [destination] - Pushes a file from your computer to your device.
adb pull [destination] [source] - Pulls a file from your device to your computer.
```

For more ADB commands, refer to the official documentation.

5.1.3 Add hardware IDs

ADB doesn't always know what devices on your computer it should or should not talk to. You can manually add the devices that it does not know how to talk to.

Just run the command for your selected device if it's below. Then, run `adb kill-server` followed by the command you were initially trying to run.

Fairphone 2:

```
printf "0x2ae5 \n" >> ~/.android/adb_usb.ini
```

OnePlus One:

```
printf "0x9d17 \n" >> ~/.android/adb_usb.ini
```

5.2 Shell access via SSH

You can use SSH to access a shell from your PC. This is useful for debugging or more advanced shell usage.

You need a SSH key-pair for this. Logging in via password is disabled by default.

5.2.1 Create your public key

If not already created, create your public key. Default choices should be fine for LAN. You can leave empty password if you don't want to deal with it each time:

```
ssh-keygen
```

5.2.2 Copy the public key to your device

You need then to transfer your public key to your device. There are multiple ways to do this. These are a few options:

- Connect the UBports device and the PC with a USB cable. Then copy the file using your file-manager.
- Or, transfer the key via the Internet by e-mailing it to yourself, or uploading it to your own cloud storage, web-server, etc.
- You can also connect via *adb* and use the following command to copy it (Some devices cant support ADB, check the devices page to learn more.):

```
adb push ~/.ssh/id_rsa.pub /home/phablet/
```

5.2.3 Configure your device

Now you have the public key on the UBports device. Let's assume it's stored as `/home/phablet/id_rsa.pub`. Use the terminal app or an ADB connection to perform the following steps on your device.

```
mkdir /home/phablet/.ssh
chmod 700 /home/phablet/.ssh
cat /home/phablet/id_rsa.pub >> /home/phablet/.ssh/authorized_keys
chmod 600 /home/phablet/.ssh/authorized_keys
chown -R phablet:phablet /home/phablet/.ssh
```

Now start the SSH server. If you are using an Android-based device:

```
sudo android-gadget-service enable ssh
```

If you are using a Linux-based device (like the PinePhone):

```
sudo service ssh start
```

5.2.4 Connect

Now everything is set up and you can use `ssh`

```
ssh phablet@ubuntu-phablet
```

or, if that does not work

```
ssh phablet@<ip-address>
```

To identify the IP-address of your UBports device, open the Terminal app on your device and run the following command:

```
hostname -I
```

The output is a list of IP addresses separated by spaces. Use the IP address that matches your subnet. On your PC or laptop:

```
debian2:~/$ hostname -I
192.168.42.41 2001:982:89e9:1:bc6b:758:7ba2:c190
```

On the phone:

```
phablet@ubuntu-phablet:~$ hostname -I
10.55.74.177 192.168.42.52 2001:982:89e9:1:ef68:5f7c:3db4:c0d3
```

In this case you use the second IP address

Of course you can now also use `scp` or `sshfs` to transfer files.

5.2.5 References

- askubuntu.com: How can I access my Ubuntu phone over SSH?
- gurucubano: BQ Aquaris E 4.5 Ubuntu phone: How to get SSH access to the Ubuntu phone via Wi-Fi

5.3 Switch release channels

You can change the release channel using the system settings. The 3 existing channels are *stable*, *rc* (release candidate) and *devel* (nightly builds). More info in *the release schedule*.

5.4 Screencasting your Ubuntu touch device to your computer

The bundled `mirscreencast` command-line utility dumps screen-frames to a file. Use it to stream your Ubuntu Touch display to a computer over the network (or directly through ADB) to watch it live or record it to a file.

5.4.1 Using ADB

You can catch output directly from the `adb exec-out` command and forward it to MPlayer:

```
adb exec-out timeout 120 mirscreencast -m /run/mir_socket --stdout --cap-interval 2 -␣
↪384 640 | mplayer -demuxer rawvideo -rawvideo w=384 h=640:format=rgba -
```

`timeout` above is used to kill the process in a proper manner on the Ubuntu Touch device (120 seconds here). (Otherwise the process continues even if killed on the computer.) Reduce or increase the number of frames per second with``–cap-interval`` (1 = 60fps, 2=30fps, ...) and the size of frames `384 640` means a width of 384 px and a height of 640 px.

5.4.2 Via the network

On the receiver

For real-time casting:

Prepare your computer to listen to a TCP port (1234 here) and forward the raw stream to a video player (MPlayer here) with a framesize of 384x640:

```
nc -l -p 1234 | gzip -dc | mplayer -demuxer rawvideo -rawvideo␣
↪ =384 h=640 format=rgba -
```

For stream recording:

Prepare your computer to listen to a TCP port (1234 here), unpack and forward the raw stream to a video encoder (MEncoder here):

```
nc -l -p 1234 | gzip -dc | mencoder -demuxer rawvideo -rawvideo␣
↪ fps=60 w=384 h=640 format=rgba -ovc x264 -o out.avi -
```

On the Ubuntu Touch device

Forward and gzip the stream with 60 FPS (–cap-interval 1) and a framesize of 384x640 to the computer at 10.42.0.209 on port 1234:

```
mirscreencast -m /run/mir_socket --stdout --cap-interval 1 -s 384 640 | gzip -c | nc 10.
↪42.0.209 1234
```

Example script

Run this on a computer (with MPlayer installed and SSH access to the Ubuntu Touch device) to screencast a remote Ubuntu Touch device to it.:

```
#!/bin/bash
SCREEN_WIDTH=384
SCREEN_HEIGHT=640
PORT=1234
FORMAT=rgba


if [[ $# -eq 0 ]] ; then
    echo 'usage: ./mircast.sh UT_IP_ADDRESS , e.g: ./mircast.sh 192.168.1.68'
    exit 1
fi

IP=$1

LOCAL_COMMAND='nc -l -p $PORT | gzip -dc | mplayer -demuxer rawvideo -rawvideo w=$SCREEN_
↪WIDTH:h=$SCREEN_HEIGHT:format=$FORMAT -'
REMOTE_COMMAND="mirscreencast -m /run/mir_socket --stdout --cap-interval 1 -s $SCREEN_
↪WIDTH $SCREEN_HEIGHT | gzip -c | nc \$SSH_CLIENT $PORT"
ssh -f phablet@$IP "$REMOTE_COMMAND"
eval $LOCAL_COMMAND
```

You can download it here: `files/mircast.sh`

5.4.3 References

- Initial source: https://wiki.ubuntu.com/Touch/ScreenRecording
- Demo: https://www.youtube.com/watch?v=HYm4RUwwo5Q

5.5 Reverse tethering

This tutorial helps you to connect your Ubuntu Touch device to your computer to access the Internet on it. This is useful if there is no available Wi-Fi connection or you don't have a data subscription on your Ubuntu Touch device.

Prerequisite: A Ubuntu Touch device with "Developer mode" on, attached via USB to the computer connected to the Internet.

5.5.1 Steps

1. On the Ubuntu Touch device: `android-gadget-service enable rndis`
2. On your computer: get your RNDIS IP address e.g:`hostname -I`
3. On the Ubuntu Touch device:

   add a gateway: `sudo route add default gw YOUR_COMPUTER_RNDIS_IP`

   add nameservers: `echo "nameserver 8.8.8.8" | sudo tee /etc/resolv.conf`

   refresh the DNS table: `resolvconf -u`
4. On computer:

   Turn on IP forwarding: `echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward`

   Apply NAT: `sudo iptables -t nat -A POSTROUTING -s 10.0.0.0/8 -o eth0 -j MASQUERADE`

5.5.2 References

- Ask Ubuntu: https://askubuntu.com/questions/655321/ubuntu-touch-reverse-tethering-and-click-apps-updates

5.6 CalDAV and CardDAV syncing

CalDAV and CardDAV are protocols to sync calendars and contacts with a remote server. Many e-mail hosters provide a CalDAV and CardDAV interface.

Note: CalDAV sync can also be set up by using the calendar app. Open it, click on the little calendar icon in the top right corner and select "Add internet calendar > Generic CalDAV". Enter your calendar URL as well as your username and password to complete the process.

At the moment, the Ubuntu Touch graphical user-interface lacks a CardDAV implementation, but you can do it by using SyncEvolution + cron. There is a script you can run in the terminal, or via a phablet SSH connection. These instructions work for CalDAV as well.

1) Follow this guide to activate Developer Mode and a connection either through ADB or SSH.
2) Download this script (let's call it dav.sh) and edit the following variables:

- Server side: CAL_URL, **CONTACTS_URL**, USERNAME, PASSWORD (of your ownCloud/nextCloud/baikal/SOGO/... server)
- CONTACT and CALENDAR _ NAME / VISUAL_NAME / CONFIG_NAME (it's more cosmetic)
- CRON_FREQUENCY (for the frequency of synchronization)
- Line 61: write `sudo sh -c "echo '$COMMAND_LINE' > /sbin/sogosync"` , instead of `sudo echo "$COMMAND_LINE" > /sbin/sogosync`, to avoid a "Permission denied" error.

3) Move the file to your Ubuntu Touch device, either by way of a file manager or through the use of ADB:

```
adb push dav.sh /home/phablet
```

4) Connect with the phablet shell (using `adb shell`) or type the following directly into the Ubuntu Touch device terminal app:

```
chmod +x dav.sh
./dav.sh
```

5.6.1 Sources:

- https://askubuntu.com/questions/616081/ubuntu-touch-add-contact-list-and-calendars/664834#664834
- https://gist.github.com/boTux/069b53d8e06bdb9b9c97
- https://gist.github.com/tcarrondo
- https://gist.github.com/bastos77
- https://askubuntu.com/questions/601910/ssh-ubuntu-touch

CHAPTER

SIX

CONTRIBUTING TO UBPORTS

Welcome! You're probably here because you want to contribute to UBports. The pages you'll find below here will help you do this in a way that's helpful to both the project and yourself.

If you're just getting started, we always need help with *thorough bug reporting*. If you are multilingual, *translations* are also a great place to start.

If those aren't enough for you, see our contribute page for an introduction of our focus groups.

6.1 Bug reporting

This page contains information to help you help us by reporting an actionable bug for Ubuntu Touch. It does NOT contain information on reporting bugs in apps, most of the time their entry in the OpenStore will specify where and how to do that.

6.1.1 Get the latest Ubuntu Touch

This might seem obvious, but it's easy to miss. Go to (Settings - Updates) and make sure that your device doesn't have any Ubuntu updates available. If not, continue through this guide. If so, update your device and try to reproduce the bug. If it still occurs, continue through this guide. If not, do a little dance! The bug has already been fixed and you can continue using Ubuntu Touch.

6.1.2 Check if the bug is already reported

Open up the bug tracker for ubuntu-touch.

Note: Pinephone users should check for and report bugs here

First, you'll need to make sure that the bug you're trying to report hasn't been reported before. Search through the bugs reported. When searching, use a few words that describe what you're seeing. For example, "Lock screen transparent" or "Lock screen shows activities".

If you find that a bug report already exists, select the "Add your Reaction" button (it looks like a smiley face) and select the +1 (thumbs up) reaction. This shows that you are also experiencing the bug.

If the report is missing any of the information specified later in this document, please add it yourself to help the developers fix the bug.

6.1.3 Reproduce the issue you've found

Next, find out exactly how to recreate the bug that you've found. Document the exact steps that you took to find the problem in detail. Then, reboot your phone and perform those steps again. If the problem still occurs, continue on to the next step. If not…

6.1.4 Making the bug report

Now it's time for what you've been waiting for, the bug report itself! Bug reports need to be filed in English.

First, pull up the bug tracker and click "New Issue". Log in to GitHub if you haven't yet.

Next, you'll need to name your bug. Pick a name that says what's happening, but don't be too wordy. Four to eight words should be enough.

Now, write your bug report. A good bug report includes the following:

- What happened: A synopsis of the erroneous behavior
- What I expected to happen: A synopsis of what should have happened, if there wasn't an error
- Steps to reproduce: You wrote these down earlier, right?
- Software Version: Go to (Settings - About) and list what appears on the "OS" line of this screen. Also include the release channel that you used when you installed Ubuntu on this phone.

Once you're finished with that, post the report. You can't add labels yourself, so please don't forget to state the device you're experiencing the issue on in the description so a moderator can easily add the correct tags later.

A developer or QA-team member will confirm and triage your bug, then work can begin on it. If you are missing any information, you will be asked for it, so make sure to check in often!

6.1.5 Getting Logs

Here we'll discuss the general steps to get logs, files that contain important debugging messages, on Ubuntu Touch. You may be asked for logs after submitting your issue. This section will help you to retrieve these logs from the device.

> **Warning:** Log files may contain information you prefer to keep private. Please be sure to skim through them to ensure this is not the case before posting them. We can coordinate to get them through e-mail or a similarly less public medium.

To get ready, download the Logviewer app from the OpenStore. This app will help us find, view, and send the logs we need.

Now, open the Logviewer app. You'll see a list of all of the log files available in your user log folder, `/home/phablet/.cache/upstart/`. This folder contains many, but not all, of the logs for software running on your device. You can tap on any log to view it, then tap the "send" icon in the top right corner to share it using an online service.

If you don't see the log that you've been asked to send in Logviewer, it may be in a different folder or the application may not have created any logs yet. Tell this to the person who asked you for the logs. They should be able to get you the information you need to get back on track.

6.2 Code changes

This page helps you get your changes to Ubuntu Touch merged. We describe the things we look for in code, commits, and changesets. We also take a look at what to expect in the code review process.

6.2.1 Assumptions

This page does not include information about how to develop applications or changes to Ubuntu Touch. To learn how to develop Ubuntu Touch components, see the *System software development* section. To learn how to develop applications, see the *App development* or *Preinstalled apps* sections.

This page assumes you know how to write your changes, use Git to commit them, push them to GitHub or Gitlab as needed, fork a repository, and make a Pull Request (PR) on GitHub or Merge Request (MR) on GitLab.

6.2.2 Definitions

Since we use GitHub and GitLab for our code hosting depending on the component, we use some terms interchangeably in this document.

changeset A GitHub Pull Request (PR) or GitLab Merge Request (MR).

issue report A GitHub or GitLab Issue.

6.2.3 Reasoning

Many users use Ubuntu Touch on many different devices. Breaking functionality that these users rely on can be catastrophic. Our review process helps us avoid broken functionality by ensuring a few properties of the changeset.

In particular, we want to make sure the change:

- meets its own goals, such as adding a feature or fixing a bug
- does not cause any unexpected behavior
- makes it easy for someone to make further changes to the software in the future

The rules on this page explain how you should write and document your changes to Ubuntu Touch. Following these rules makes the review process easier for you and us.

These are not absolute; we won't automatically reject your change based on whether you follow the rules. If you break a rule, make sure you can explain why.

6.2.4 Code change checklist

Keep this checklist in mind while writing your code and as you prepare to commit your changes.

Do not commit commented code

As developers, we often comment out code that is only there to help us verify assumptions (like console logging messages). Make sure this **never** makes it into your commits. If you must disobey this rule, make sure you add a comment explaining why the commented code still exists in the source. If you feel like you must make a `// TODO:` or `// FIXME:` comment, consider whether filing an issue report referencing the broken code or TODO item would be better.

Write tests for your changes

If a component has a test suite, use it. Make sure that you know how to run the tests for the component and understand the output of those tests. The component's README file should contain this information. If not and you don't know how to find it, ask us. We're happy to help when someone is trying to learn about a component for the first time. However, be prepared to get a pile of links to documentation for a build system or test framework rather than a hand-crafted list of commands for you to run. That may be all we have time to provide. If you're still confused after checking out the materials you've received, feel free to ask again with your new, specific concerns. We would **really** appreciate it if you added your newfound knowledge to the component's README. The best time to write documentation is after doing something for the first time.

Once you know how to run the test suite, you can add your tests. Make sure you test as many aspects of your new functionality as you feel are reasonable. Lean on the side of more tests: we rarely ask you to remove tests, but we may ask you to add more.

If you notice that a component doesn't have a test suite, it's an excellent time to file a bug on that component! We would appreciate your help adding a test suite, but we do not require that of you.

Follow any existing style guides

If a repository has a style guide, use it. If there is no style guide, try to follow the existing style in the file you're changing as closely as possible.

6.2.5 Commit checklist

Long after writing code, we often need to understand how and why it introduces a bug. Commit messages are our first step in understanding these complex questions. This checklist helps future viewers of your code understand why it is the way it is.

For an example of a series of commits that follows these guidelines very well, see the three commits at the top of the git log leading to 840777f in Lomiri. Not every change requires such an in-depth description as these examples. Lean on the side of more detail; someday, *you* are the one looking back at your commits and wondering why you made them.

Answer crucial questions in commit messages

We ask questions like, "what?" "why?" and "how?" to gain context of a situation quickly.

Strive to answer these questions in your commit metadata or content. Doing so helps reviewers and future readers understand the context of a change. In order of importance, answer:

What? Answer this question with the first few lines of your commit message. "What does your commit do?" "Optimize wallpapers for load times and memory use".

Why? Always state the reasoning behind a change: the bugs it fixes, the features it builds up to, and especially the limitations that cause it to look more complicated than a viewer expects. Provide permanent links to issue reports or documentation if they give context.

Who? Git embeds your name and email address as the committer of your changes. If someone else has helped you write your changes, you should add them with a `Co-Authored-By: Their Name <their-email@example.com>` line in your commit message.

It is possible to provide some of this context in code comments rather than in commit messages. Comments can assist people who are only reading the code without using tools like `git blame`. Bear in mind that comments can quickly become out-of-date compared to the code that they are near. A commit message is linked only to the code you wrote.

In closing, make sure you anticipate any questions you expect yourself or others to have about your code in the future and answer them.

Make commits as small as possible, but no smaller

Some changes require multiple logical steps to complete. If this is the case, split these steps into separate commits. Each commit should follow the entire commit checklist. It should not require any of the commits that come after it to build or pass tests.

Let's say you are implementing a new way to search for phone numbers in the Dialer app. Your change requires three distinct steps:

1. Fix a bug in the current phone number search
2. Add a new API to support your new number search
3. Add the UI elements to use your new search

If you do all of these changes in a single commit and there is a problem in step three, we reject the entire changeset. If you split the changes into separate commits instead, the bug fix and new API could be added to the mainline software while you work on redesigning your UI.

Rebase your changes during the review, if possible

It's common for a commit log on GitHub to go something like this:

- Fix bug in phone number search
- Add new API call for new phone number search
- Add UI for new phone number search
- Fix UI
- Fix bug fix
- Review changes
- Fix API after comments from reviewers

If you have to come back to this commit log in the future, you'll be confused in no time at all. What did the reviewers say, what was wrong with the UI and the bug fix?

Don't add a new commit when you receive a correction that affects the first commit in your series. Instead, edit the first commit. If doing this changes the information in your commit message, update that too.

Editing commits requires learning new tools in Git. Edit a series of commits with `git-rebase`. Push the resulting changes with a force push, or create a new branch and open a new changeset.

Luckily, there are graphical tools that make this job more manageable. Unluckily, if you don't understand what they are doing on the Git command line, you might mess up and have to start over from your old series. Even seasoned Git pros mess up this process sometimes, don't worry. There is almost always a way back to your old series.

Each commit must continue to build, run, and pass tests after your changes.

Format your commit message correctly

Keep the first line of your commit message (the summary) to 50 characters or less. Every other line in the commit message should be 72 characters or less. It's OK if you have to break the rules — some changes can't be summarized in 50 characters, and some links are longer than 72.

6.2.6 Changeset description checklist

Ensure that your changeset answers the following questions in its description.

If you've followed the guidelines for commit messages, you can probably copy the relevant information from the commit messages into your changeset description. GitHub and GitLab do this for you in single-commit changesets. A description that says, "this is complicated to explain; please read the commit messages," is also acceptable. We trust you to strike a balance.

What issue does your changeset resolve?

Link changesets to the issue report that they resolve, whether you add a feature or fix a problem. Specify the devices the issue occurs on or if it applies to all devices. Provide enough information so that someone can look at it and know how to reproduce the problem or test the feature. Add this information to the issue report as well if it is missing.

For example:

> Fixes https://github.com/ubports/ubuntu-touch/issues/1

How did you test that the change was successful?

All changes require testing to ensure they resolve the issue report they're referencing. List the environment you tested your changes in, including the operating system under test, its version, and the devices you tested on. Explain your testing process. If you are not sure how to test a feature related to your changes, mention it.

For example:

> I've tested this dialer change on the Nexus 5. It should work on all devices since it's not modifying anything device-specific, but more testing would be appreciated. To fix a bug, I had to touch a bit of code in the phone number search area (#53). I'm not entirely sure how to test that for regressions.

Do any changes need to be merged before or after this changeset?

Some changes depend on one or more changes before they work correctly. If this is the case, you should document the dependencies in your changeset descriptions. Document all sets that must be merged *before* and *after* the current one in a series, if needed.

For example, say you've filed three Merge Requests on GitLab, `ubports/core/docs!1`, `ubports/core/code!2`, and `ubports/core/infrastructure!3`. They must be merged in that order. In that case, your MR descriptions would have something similar to this included:

- In `ubports/core/docs!1`:

  > ubports/core/code!2 and ubports/core/infrastructure!3 depend on this MR.

- In `ubports/core/code!2`:

  > ubports/docs!1 must be merged before this MR. ubports/core/infrastructure!3 depends on this MR.

- In `ubports/core/infrastructure!3`:

ubports/docs!1 and ubports/core/infrastructure!3 must be merged before this MR.

What does the interface look like before and after the change?

If your changes also change the look of the user interface, include screenshots to illustrate these changes.

6.2.7 Submission and review

Now that you've checked your changeset, it's time to submit it for the review! Thank you in advance for contributing to Ubuntu Touch. Here is what you can expect from us as we review your changes.

We respect you

You chose to use your time to contribute to Ubuntu Touch. That decision is never made lightly. A reviewer must treat you with respect and work with you toward your changeset becoming a part of Ubuntu Touch.

Respect is a two-way street. Ubuntu Touch is a large project; there are never enough hands to do all the work needed. It may take a while for your changeset to see any attention, and after a long wait, you might come back to find a stern request for changes. Read a stern-looking message as someone trying to work as quickly as possible, not as an attempt to be rude to you. We extend the same grace to you.

We ask many questions

Reviewers better understand the code you're changing after asking you questions about how it works. They may know what your code does already, but they'll still ask. If you can explain how your code works, it's more likely that you have done the proper testing to ensure it works.

If you are not able to articulate why something works, it is a red flag. The change is likely working around a different bug that can come back to haunt us in the future. Fixing the real bug instead of making workarounds is a better use of time.

We ask for changes

Another person looking at your code may see problems with it that you have missed. Whether those problems are inefficiencies, style issues, or new bugs, they are more likely to be found by your reviewers than by you. Don't worry if you get a book of requested changes back from your reviewer. Nobody is perfect. Finding and fixing these potential issues leads to a faster and more stable Ubuntu Touch. It is not an insult to your skills as a developer.

We may ask for massive changes

Sometimes reviewers look at a change and realize that a much deeper issue exists in the component. Sometimes you start out developing a feature with the wrong assumptions in mind and end up with an unwieldy mess of spaghetti code. Either way: this is not the correct path to the destination. Whatever the reason, your reviewer points out the potential problems with your change and constructively suggests a new direction for you to take.

You can tell us that we are wrong

You might be sure that an underlying bug is not easily fixable or that the path you've taken is the best. You might not have enough time to do a fix correctly as asked. If you think that our rejection of your changeset is wrong for any reason, let us know. Reviewers try to work with you to reach a solution that's best for our users.

6.2.8 Merge and maintenance

After ensuring your changeset meets all of our requirements, it becomes a part of Ubuntu Touch. Thank you! You are a member of a small group of people contributing to a beautiful community. However, this is not the end of the work.

Sometimes your changes break a piece of Ubuntu Touch functionality despite our best efforts. You might be called upon to help investigate the source of the issue and prepare a fix if this happens. Your change could be reverted if we find that your change was the direct cause of a bug and we cannot contact you.

New contributors might notice that you have recently worked on a component they want to work with. We would be grateful if you helped teach them what you learned during this process.

If nothing else, stick around for the people thanking you for the change you've made. The Ubuntu Touch community is incredibly supportive and thankful. It would be a shame for you to miss your share of the good vibes.

6.3 Quality Assurance

This page explains how to help the UBports QA team, both as an official member or a new contributor. Please also read the *issue tracking* and *bugreporting* guides to better understand the workflow. For real-time communication, you can join our telegram group.

6.3.1 Smoke testing

To test the core functionality of the operating system, we have compiled a set of standardized tests. Run these tests on your device to *find and report bugs and regressions*. It's usually run on all devices before a new release to make sure no new issues were introduced.

6.3.2 Confirming bug reports

Unconfirmed bugreports are labeld **needs confirmation** to enable global filtering. Browse through the list, read the bugreports and try to reproduce the issues that are described. If necessary, add *missing information or logs, or improve the quality of the report by other means*. Leave a comment stating your device, channel, build number and whether or not you were able to reproduce the issue.

If you have write-access to the repository, you can replace the **needs confirmation** label with **bug** (to mark it confirmed) or **invalid** (if the issue is definitely not reproducible). In that case it should be closed.

If you find two issues describing the same problem, leave a comment and try to find their differences. If they are in fact identical, close the newer one and label it **duplicate**.

6.3.3 Testing patches

Pull-requests can be tested using the QA scripts. Run `ubports-qa -h` for usage information.

Once the pull-request has been merged, the issue it fixes is moved to the quality assurance column of the GitHub project. Please check if the issues in this column are still present in the latest update on the devel channel, then see if anything else has broken in the update. Check if the developer mentioned specific things to look out for when testing and leave a comment detailing your experience. If you have write-access to the repository, you can move the issue back to **In Development** (and reopen it) or forward to **Release Candidate** as specified by the *issue tracking guidelines*.

6.3.4 Initial triaging of issues

Initial triaging of new issues is done by QA-team members with write-access to the repository. If a new issue is filed, read the report and add the correct labels as specified by the *issue tracking guidelines*. You can also immediately start confirming the bugreport.

If the new issue has already been reported elsewhere, label it **duplicate** and close it.

6.4 Documentation

Tip: Documentation on this site is written in ReStructuredText, or RST for short. Please check the RST Primer if you are not familiar with RST.

This page will guide you through writing great documentation for the UBports project that can be featured on this site.

6.4.1 Documentation guidelines

These rules govern how you should write your documentation to avoid problems with style, format, or linking.

Title

All pages must have a document title that will be shown in the table of contents (left sidebar) and at the top of the page.

Titles should be "sentence cased" rather than "Title Cased". For example:

```
Incorrect casing:
    Writing A Good Bug Report
Correct casing:
    Writing a good bug report
Correct casing when proper nouns are involved:
    Installing Ubuntu Touch on your phone
```

There isn't a single definition of title casing that everyone follows, but sentence casing is easy. This helps keep capitalization in the table of contents consistent.

Page titles are underlined with equals signs. For example, the markup for *Bug reporting* includes the following title:

```
Bug reporting
=============
```

Note that:

1. The title is sentence cased
2. The title is underlined with equals signs
3. The underline spans the title completely without going over

Incorrect examples of titles include:

- Incorrect casing

```
Bug Reporting
=============
```

- Underline too short

```
Bug reporting
======
```

- Underline too long

```
Bug reporting
=================
```

Headings

There are several levels of headings that you may place on your page. These levels are shown here in order:

```
Page title
==========

Level one
---------

Level two
^^^^^^^^^

Level three
"""""""""""
```

Each heading level creates a sub-section in the global table of contents tree available when the documentation is built. In the primary (web) version of the documentation, this only shows four levels deep from the top level of the documentation. Please refrain from using more heading levels than will show in this tree as it makes navigating your document difficult. If you must use this many heading levels, it is a good sign that your document should be split up into multiple pages.

Table of contents

People can't navigate to your new page if they can't find it. Neither can Sphinx. That's why you need to add new pages to Sphinx's table of contents.

You can do this by adding the page to the `index.rst` file in the same directory that you created it. For example, if you create a file called "newpage.rst", you would add the line marked with a chevron (>) in the nearest index:

```
.. toctree::
   :maxdepth: 1
   :name: example-toc

   oldpage
   anotheroldpage
>  newpage
```

The order matters. If you would like your page to appear in a certain place in the table of contents, place it there. In the previous example, newpage would be added to the end of this table of contents.

Moving pages

Sometimes it becomes necessary to move a page from one place in the documentation to another. Generally this is to improve document flow: For example, it makes more sense for the page to come after a page you've just added in a different section.

However, people link to our documentation from many sources that we do not control. Blogs, websites, and other documentation sites can direct people here using links that they may never update. It is a terrible experience to follow a link from a different site and land on a 404 page, left to your own devices to find your way in restructured documentation.

We use a tool called Rediraffe to avoid this bad experience. Rediraffe creates redirect pages which can send a user from an old, invalid link to a new, useful link. Please create a redirect link when changing a page's name or moving a page within the documentation's directory structure. Redirect links are created by placing the filename of the old document and the filename of the new document, relative to the documentation's root, in the redirects.txt file.

We use Rediraffe's `checkdiff` builder to ensure that pages are not deleted from the documentation without a redirect in place. This builder is run as part of the `build.sh` script in the repository and as part of our automated build once you submit a Pull Request.

What follows are some examples of situations where you should create redirects.

You are moving `systemdev/calendars.rst` to `appdev/calendars.rst`. Add the following to the `redirects.txt` file:

```
"systemdev/calendars.txt" "appdev/calendars.txt"
```

You are moving `appdev/clickable.rst` into several pages in `appdev/clickable/` to give significantly more information about the tool than there was previously. You have created an introduction page, `appdev/clickable/introduction.rst`. In this case, it would be a good idea to redirect the old page to the new introduction page. This can be done by adding the following to `redirects.txt`:

```
"appdev/clickable.rst" "appdev/clickable/introduction.rst"
```

Warnings

Your edits must not introduce any warnings into the documentation build. If any warnings occur, the build will fail and your pull request will be marked with a red 'X'. Please ensure that your RST is valid and correct before you create a pull request. This is done automatically (via sphinx-build crashing with your error) if you follow *our build instructions* below.

Line length

There is no restriction on line length in this repository. Please do not break lines at an arbitrary line length. Instead, turn on word wrap in your text editor.

6.4.2 Contribution workflow

The following steps will help you to make a contribution to this documentation after you have written a document.

Note: You will need a GitHub account to complete these steps. If you do not have one, click here to begin the process of making an account.

Forking the repository

You can make more advanced edits to our documentation by forking ubports/docs.ubports.com on GitHub. If you're not sure how to do this, check out the excellent GitHub guide on forking projects.

Building the documentation

If you'd like to build this documentation *before* sending a PR (which you should), follow these instructions on your *local copy* of your fork of the repository.

The documentation can be built by running `./build.sh` in the root of this repository. The script will also create a virtual build environment in `~/ubportsdocsenv` if none is present.

If all went well, you can enter the `_build/html` directory and open `index.html` to view the UBports documentation.

If you have trouble building the docs, the first thing to try is deleting the build environment. Run `rm -r ~/ubportsdocsenv` and try the build again. Depending on when you first used the build script, you may need to run the `rm` command with `sudo`.

Final check of your contribution

After you have created your PR on github, the CI (continuous integration) system will make a test build of your contribution. Please double check whether this builds successfully and whether the result looks as you intended it to:

- Scroll to the bottom of the "Conversation" tab of your PR on github, here you will see the checks (You may have to click on "Show all checks")
- It can have a yellow dot, i.e., "pending" then wait a few more seconds.
- Or it may have a red X, i.e., it failed. In this case please check why it failed
- If it shows a green check mark, it means the PR could be built successfully
- Now please click on "Details",
- then "Artifacts" on the top right,
- then "_build/html/..index.html",
- and finally on "Go to start page".

Now you can browse a complete build of the UBports docs site with your contribution included. Double check whether your changes look ok.

6.4.3 Alternative methods to contribute

Translations

You may find the components of this document to translate at its project in UBports Weblate.

Writing documents not in RST format

If you would like to write documents for UBports but are not comfortable writing ReStructuredText, please write it without formatting and post it on the UBports Forum in the relevant section (likely General). Someone will be able to help you revise your draft and write the required ReStructuredText.

Uncomfortable with Git

If you've written a complete document in ReStructuredText but aren't comfortable using Git or GitHub, please post it on the UBports Forum in the relevant section (likely General). Someone will be able to help you revise your draft and submit it to this documentation.

6.4.4 Current TODOs

This section lists the TODOs that have been included in this documentation. If you know how to fix one, please send us a Pull Request to make it better!

To create a todo, add this markup to your page:

```
.. todo::

   My todo text
```

Todo: There is also another way to create somewhat more featureful webapps, sometimes referred to as webapp+ or alternative container. This needst to be properly documented. It's a simple qml app that can be easily configured. Creation is almost as simple as 'classic' webapp, but result is more powerfull with the a nice navigation feature. A rather advanced example of this is the YouTube app from Mateo Salta which has quite some modifications on top of the template.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/doniks-docsubportscom/checkouts/latest/appdev/webapp/index.rst, line 19.)

Todo: Document the process for Nexus 4 (mako)

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/doniks-docsubportscom/checkouts/latest/systemdev/kernel-hal.rst, line 57.)

6.5 Translations

Although English is the official base language for all UBports projects we believe you have the right to use it in any language you want.

We are working hard to meet that goal, and you can help as well. You decide how much time you can put into translation. From minutes to hours, everything counts.

6.5.1 How does a text get translated (technical background)

Most apps running on Ubuntu Touch use GNU gettext for translations. A *.pot* template file holds an apps strings. From this template for every language an individual *.po* file is derived. Those *.po* files do contain the actual translations. To add a new language a new *.po* file needs to be created. To translate a string the *.po* file needs to be edited. Next to the original string (english in most cases) the appropriate translation needs to be entered following the *.po* gettext file syntax. Some apps do use QtLinguist for translation. It does follow the same priciples only using *.ts* files for translations with another syntax. As template one *.ts* file is used.

Within an app strings are marked either this way `i18n.tr("string")` when using gettext or like this `tr("string")` when using QtLinguist. When the app is built those translatable strings are extracted and written into the *.pot* or *.ts* template files. The *.po* and *.ts* language files then need to be updated from their template.

Once strings have been translated, they are made available for users with a new release of an app.

6.5.2 Methods of Translation

There are three commonly used ways to translate an app:

- Weblate: A web based translation tool. UBports apps using Weblate can be translated here. There is an automatic transfer of new strings from an apps repo to weblate. Translated strings get back as commits to the repo from weblate.
- Editor: Translation is done by changing *.po* files with the editor of your choice, and a GitHub/GitLab account. The *.po*/*.ts* files for each project are in their repository on our GitLab organization or on our GitHub organization. After translating the translator commits new translations to an apps repo generally by opening a merge request/pull request.
- Team translation: We also have a Translation Forum to discuss translating Ubuntu Touch and its core apps. Some projects are using Telegram or Matrix groups too, and some teams are still using the Ubuntu Launchpad framework.

Core apps and some community apps like dekko or TELEports are generally using weblate for translation. This is the preferred way to translate and it does not need any programming skills. Please have a look at the UBports translation projects. When using weblate do not edit *.po* files manually. Commiting edited *.po* files can break the automated process of translation.

Most apps maintained by individual developers are mainly using *.pot*/*.po* files for translation. Those generally need to be edited manually and commited to the apps repo.

6.5.3 How-To

UBports Weblate

You can go to UBports Weblate, click on "Dashboard" button, go to a project, and start making anonymous suggestions without being registered. If you want to save your translations, you must be logged in.

For that, go to UBports Weblate and click on the "Register" button. Once in the "Registration" page, you'll find two options:

- Register using a valid email address, a username, and your full name. You'll need to resolve an easy control question too.
- Register using a third party registration. Currently the system supports accounts from openSUSE, GitHub, Fedora, and Ubuntu.

Once you're logged in, the site is self-explanatory and there you'll find all the options and customization you can do.

Now, get on with it. The first step is to search if your language already exists in the project of your choice. If your language is not available for a specific project, you can add it yourself.

.po/.ts File Editor

If you want to work with *.po*/*.ts* files directly you need to know what you're doing for sure. The first thing you should always remember is:

> **Warning:** Do **never** commit *.po* files to projects that are translated using weblate. This could break the whole translation exchange process with weblate.

As was said up above, you need a file editor of your choice and a GitHub/GitLab account to translate *.po*/*.ts* files directly. Idealy you do have clickable set up and know how to build an app.

There are online gettext .po editors and those you can install on your computer. You can choose whatever editor you want, but we prefer to work with free software only. There are too many plain text editors and tools to help you translate *.po*/*.ts* files to list here.

Note: For high quality translations it is recommended to go through all of the steps given below. Although if you are not familiar with building apps using clickable, skip the steps 4. to 6. Although, you may not be able to translate the latest strings and you will not know if the strings fit into the apps layout.

How to proceed:

1. Fork the apps repo into your GitHub or GitLab account
2. Clone the repo locally or use GitLab's online editor
3. (optional) Create a new branch for the translation
4. Build the app using clickable to get up-to-date *.pot*/*.ts* template files
5. Update the desired *.po*/*.ts* language file(s) from the template
6. Translate all strings
7. Build and install the app on your device and test the translations
8. (optional) Repeat steps 6. and 7. until you are satisfied with your work

9. Push the translated *.po*/*.ts* file(s) to your forked repo [do NOT commit *.pot*/*.ts* template files]
10. Open a merge/pull request from your fork to the apps repo

Translation Team Communication

You may talk to others regarding translations by using the forum category that UBports provides for this task. To use it you need to register, or login if you're registered already.

The only requirement is to start your post putting down your language in brackets in the "Enter your topic title here" field. For example, `[Spanish] How to translate whatever?`

In your interactions with your team you'll find the best way to coordinate your translations.

6.5.4 License

All the translation projects, and all your contributions to this project, are under a Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) license that you explicitly accept by contributing to the project.

Go to that link to learn what this exactly means.

6.6 Preinstalled apps

This page will help you get started with developing the apps which are included with Ubuntu Touch.

6.6.1 Core apps

Core apps are applications which are included with UBports distributions of Ubuntu Touch **and** placed in the OpenStore for updates. Core apps are a good "first development" experience within UBports. Many are built with Clickable, an easy to use build and test system for Ubuntu Touch apps. Most core apps can even be built and run without an Ubuntu Touch device!

Which core apps currently exist?

A full list of the Ubuntu Touch core apps follows.

Application Name	Store page	Code repository
Calculator	Calculator on OpenStore	calculator-app on GitLab
Calendar	Calendar on OpenStore	calendar-app on GitLab
Clock	Clock on OpenStore	clock-app on GitLab
File Manager	File Manager on OpenStore	filemanager-app on GitLab
Gallery	Gallery on OpenStore	gallery-app on GitLab
Music	Music on OpenStore	music-app on GitLab
Notes	Notes on OpenStore	notes-app on GitLab
OpenStore	OpenStore… on OpenStore	openstore-app on GitLab
Terminal	Terminal on OpenStore	terminal-app on GitLab
UBports Welcome	UBports Welcome on OpenStore	ubports-app on GitLab
Weather	Weather on OpenStore	weather-app on GitLab

6.6.2 Other preinstalled apps

The following applications are preinstalled in Ubuntu Touch but are not considered core apps. Most of the time these projects must be updated with the system because they use many system services which do not necessarily have a stable API.

These apps may be more difficult to work with, but their repository should contain a document stating how to build and run them on a device.

- Browser (morph-browser on GitHub)
- Contacts (address-book-app on GitHub)
- Camera (camera-app on GitLab)
- External Drives (ciborium on GitHub)
- Media Player (mediaplayer-app on GitHub)
- Messaging (messaging-app on GitHub)
- Phone (dialer-app on GitHub)
- System Settings (system-settings on GitHub)

Instructions for contributing to these applications can be found in their respective repositories.

6.7 Monetary support

You can help us keep the lights on at UBports by becoming a patron on Liberapay or Patreon!

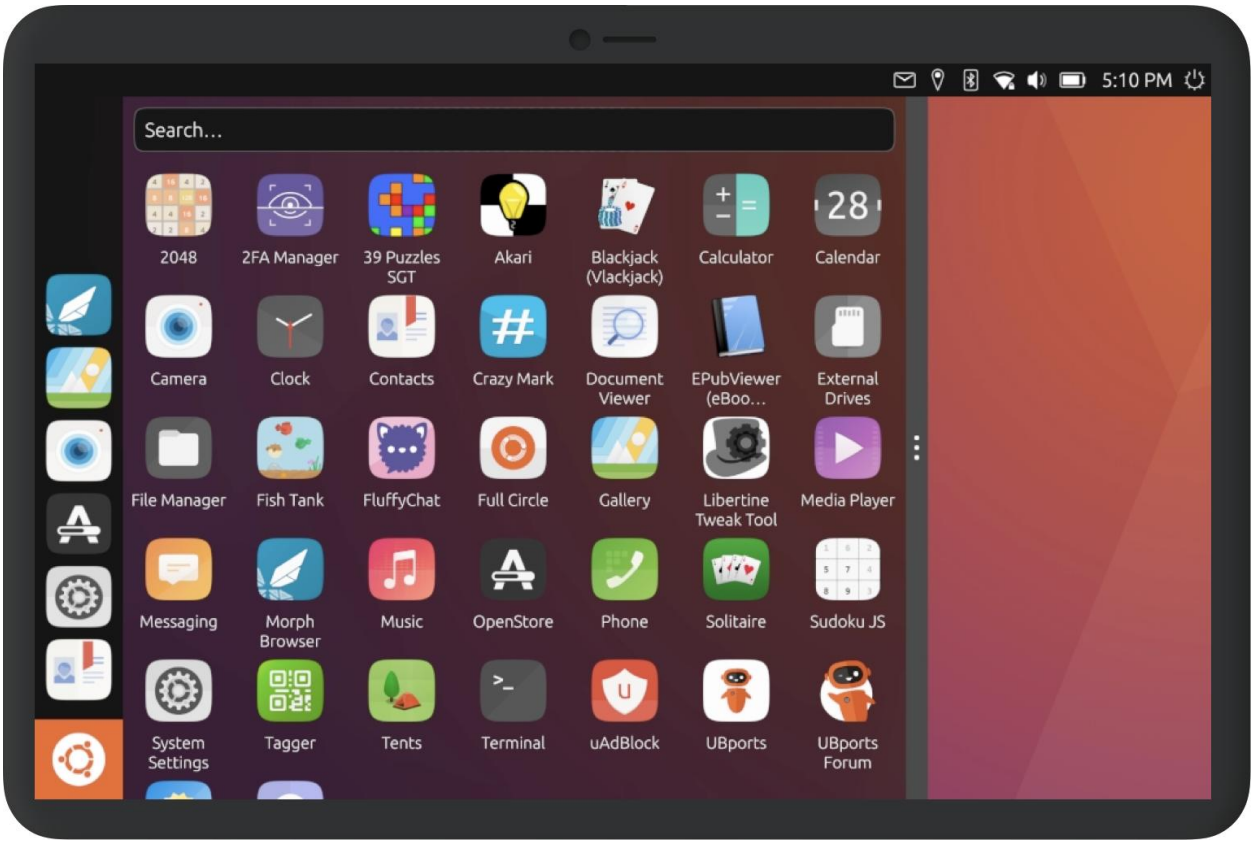
Your contribution finances our server infrastructure and debug services, and helps covering additional costs.

CHAPTER

SEVEN

APP DEVELOPMENT

You'd like to develop an app for Ubuntu Touch? Great!



7.1 Quick start

Building your first app should be as simple as:

- Install clickable
- Plug in your Ubuntu Touch device
- Enable developer mode in System Settings
- Run `clickable create`

- Choose “QML Only” from the list of app templates
- Fill in the requested information for this new app
- When the app has been generated, enter the newly created directory
- Run `clickable`

Voilà you have your first app running on Ubuntu Touch!

7.2 Overview

Ubuntu Touch supports two kinds of apps: Firstly, *native apps* have their user interface written in QML or HTML and their business logic in JavaScript, C++, Python, Rust, or Go. Secondly, *web apps* are special containers to run websites in.

7.2.1 Native apps

The recommended way to build a native application on Ubuntu Touch is to use the Qt framework (current version since OTA-16 is Qt 5.12.9). You do not have to stick with the traditional C++<->QML Qt model, you can use also Python, Go, Rust or even Javascript for your backend.

Ubuntu Touch provides a subset of Qt QML components to ease the development of your app : QML API and Icons: Suru Icons, but you can also ship your cross platform app using standard Qt QML QtQuick2 components.

You can use Clickable to facilitate Packaging, deploying, publishing and tests .

Check out informations about the platform and confinement model in *Platform* section

7.2.2 Web apps

Ubuntu Webapps can be a nice way to deliver online web applications into Ubuntu.

The Ubuntu platform provides an advanced web engine container to run online applications on the Ubuntu client devices.

Web applications are hosted online. They can be as simple as a website, like an online news site, or they can distribute content like videos. They can also have a rich user interface or use the WebGL extension to deliver games online.

Note: Ubuntu webapps and Ubuntu HTML5 apps are similar but not identical. The main difference is that the content of a webapp is provided through a URL, whereas HTML5 apps install their content (and usually provide an Ubuntu HTML5 GUI). Webapps also have restricted access to platform APIs.

Note: A different approach to webapps published in the open store is to simply create shortcuts directly yourself with webber app.

Todo: There is also another way to create somewhat more featureful webapps, sometimes referred to as webapp+ or alternative container. This needst to be properly documented. It’s a simple qml app that can be easily configured. Creation is almost as simple as ‘classic’ webapp, but result is more powerfull with the a nice navigation feature. A rather advanced example of this is the YouTube app from Mateo Salta which has quite some modifications on top of the template.

Guide

Webapp guide

How the webapp fits into the shell

A web app displays in a webview inside a webapp-container that runs as an Ubuntu app in the Ubuntu/Unity shell.

Taking a closer look:

> At the innermost level, there is a website that the developer identifies by URL. The website is rendered and runs in an Oxide webview. Oxide is a Blink/Chrome webview that is customized for Ubuntu. The Oxide webview runs and displays in the webapp-container. The webapp-container is the executable app runtime that is integrated with the Ubuntu/unity shell.

Launching

You can launch a webapp from the terminal with::

```
webapp-container URL
```

For example::

```
webapp-container http://www.ubuntu.com
```

This simple form works, but almost every webapp also uses other features, such as URL containment with URL patterns as described below.

User interface

A webapp generally fills the entire app screen space, without the need of the UI controls generally found on standard browsers.

In some cases some navigation controls are appropriate, such as Back and Forward buttons, or a URL address bar. These are added as command line arguments:

- `--enable-back-forward` Enable the display of the back and forward buttons in the toolbar (at the bottom of the webapp window)
- `--enable-addressbar` Enable the display of the address bar (at the bottom of the webapp window)

URL patterns

Webapp authors often want to contain browsing to the target website. That is, the developer wants to control the URLs that can be opened in the webapp (all other URLs are opened in the browser). This is done with URL patterns as part of the webapp command line.

However, many web apps use pages that are hosted over multiple sites or that use external resources and pages.

However, both containment and access to specified external URLs are implemented with URL patterns provided as arguments at launch time. Let's take a closer look.

Uncontained by default

By default, there is no URL containment. Suppose you launch a webapp without any patters and only a starting URL like this::

```
webapp-container http://www.ubuntu.com
```

The user can navigate to any URL without limitation. For example, if they click the Developer button at the top, they navigate to developer.ubuntu.com, and it displays in the webapp.

Tip: You can see the URL of the current page by enabling the address bar with `--enable-addressbar`.

Simple containment to the site

One often wants to contain users to the site itself. That is, if the website is www.ubuntu.com, it may be useful to contain webapp users only to subpages of www.ubuntu.com. This is done by adding a wildcard URL pattern to the launch command, as follows::

```
webapp-container --webappUrlPatterns=http://www.ubuntu.com/* http://www.ubuntu.com
```

`--webappUrlPatterns=` indicates a pattern is next http://www.ubuntu.com/* is the pattern The asterisk is a wildcard that matches any valid sequence of trailing (right-most) characters in a URL

With this launch command and URL pattern, the user can navigate to and open in the webapp any URL that starts with http://www.ubuntu.com/. For example, they can click on the Phone button (http://www.ubuntu.com/phone) in the banner and it opens in the webapp, or the Tablet button (http://www.ubuntu.com/tablet). But, clicking Developer opens the corresponding URL in the browser.

Tip: Make sure to fully specify the subdomain in your starting URL, that is, use http://www.ubuntu.com instead of www.ubuntu.com. Not specifying the subdomain would create an ambiguous URL and thus introduces security concerns.

More complex wildcard patterns

You might want to limit access to only some subpages of your site from within the webapp. This is easy with wildcard patterns. (Links to other subpages are opened in the browser.) For example, the following allows access to www.ubuntu.com/desktop/features and www.ubuntu.com/phone/features while not allowing access to www.ubuntu.com/desktop or www.ubuntu.com/phone:

```
webapp-container --webappUrlPatterns=http://www.ubuntu.com/*/features http://www.ubuntu.
↪com
```

Multiple patterns

You can use multiple patterns by separating them with a comma. For example, the following allows access only to www.ubuntu.com/desktop/features and www.ubuntu.com/phone/features::

```
webapp-container --webappUrlPatterns=http://www.ubuntu.com/desktop/features,http://www.
↪ubuntu.com/phone/features http://www.ubuntu.com
```

Tip: Multiple patterns are often necessary to achieve the intended containment behavior.

Adding a specific subdomain

Many URLs have one or more subdomains. (For example, in the following, "developer" is the subdomain: developer.ubuntu.com.) You can allow access to a single subdomain (and all of its subpages) with a pattern like this::

```
--webappUrlPatterns=http://developer.ubuntu.com/*
```

However, one usually wants the user to be able to navigate back to the starting URL (and its subpages). So, if the starting URL is http://www.ubuntu.com, a second pattern is needed::

```
--webappUrlPatterns=http://developer.ubuntu.com/*,http://www.ubuntu.com/*
```

Putting these together, here's an example that starts on http://www.ubuntu.com and allows navigation to http://developer.ubuntu.com and subpages and back to http://www.ubuntu.com and subpages::

```
webapp-container --webappUrlPatterns=http://developer.ubuntu.com/*,http://www.ubuntu.com/
↪*  http://www.ubuntu.com
```

Adding subdomains with a wildcard

Some URLs have multiple subdomains. For example, www.ubuntu.com has design.ubuntu.com, developer.ubuntu.com and more. You can add access to all subdomains with a wildcard in the subdomain position::

```
webapp-container --webappUrlPatterns=http://*.ubuntu.com/*  http://www.ubuntu.com
```

Note: An asterisk in the subdomain position matches any valid single subdomain. This single pattern is sufficient to enable browsing to any subdomain and subpages, including back to the starting URL (http://www.ubuntu.com) and its subpages.

Adding https

Sometimes a site uses https for some of its URLs. Here is an example that allows https and https as access within the webapp to www.launchpad.net (and all subpages due to the wildcard)::

```
webapp-container --webappUrlPatterns=https?://http://www.launchpad.net/* http://www.
↪launchpad.net
```

Note: the question mark in https?. This means the preceding character (the 's') is optional. If https is always required, omit the question mark.

Command line arguments

The webapp-container accepts many options to fine tune how it hosts various web applications.

See all help with::

```
webapp-container --help
```

Note: Only the following options apply to converged Ubuntu.:

```
--fullscreen Display full screen
--inspector[=PORT] Run a remote inspector on a specified port or 9221 as the default port
--app-id=APP_ID Run the application with a specific APP_ID
--name=NAME Display name of the webapp, shown in the splash screen
--icon=PATH Icon to be shown in the splash screen. PATH can be an absolute or path␣
↪relative to CWD
--webappUrlPatterns=URL_PATTERNS List of comma-separated url patterns (wildcard based)␣
↪that the webapp is allowed to navigate to
--accountProvider=PROVIDER_NAME Online account provider for the application if the␣
↪application is to reuse a local account.
--accountSwitcher Enable switching between different Online Accounts identities
--store-session-cookies Store session cookies on disk
--enable-media-hub-audio Enable media-hub for audio playback
--user-agent-string=USER_AGENT Overrides the default User Agent with the provided one.
```

Chrome options (if none specified, no chrome is shown by default)::

```
--enable-back-forward Enable the display of the back and forward buttons (implies --
↪enable-addressbar)
--enable-addressbar Enable the display of a minimal chrome (favicon and title)
```

Note: The other available options are specific to desktop webapps. It is recommended to not use them anymore.

User-Agent string override

Some websites check specific portions of the web engine identity, aka the User-Agent string, to adjust their presentation or enable certain features. While not a recommended practice, it is sometimes necessary to change the default string sent by the webapp container.

To change the string from the command line, use the following option::

```
--user-agent-string='<string>' Replaces the default user-agent string by the string␣
↪specified as a parameter
```

Browser data containment

The webapp experience is contained and isolated from the browser data point of view. That is webapps do not access data from any other installed browser, such as history, cookies and so on. Other browser on the system do not access the webapp's data.

Storage

W3C allows apps to use local storage, and Oxide/Webapp-container supports the main standards here: LocalStorage, IndexedDB, WebSQL.

Quick start

There are several tools to help you package and deploy your webapp to your device:

- Webapp creator application available from the openstore
- Clickable CLI

Debugging your webapp

This guide give you some tips to help you debug your webapp.

Debug webapps

Most web-devs will probably want do most of their coding and debugging in the usual browser environment. The Ubuntu Touch browser is compliant with modern web standards, and most webapps will just work without further changes.

For those (hopefully) rare cases where further debugging is needed, there are two ways to gain further information on the failure.

Watch the logs

If you are comfortable in a CLI environment, most Javascript errors will leave an entry in the app log file:

```
.cache/upstart/application-click-[YOUR_APP_NAME.AUTHOR_NAME..].log
```

You can check the log file in the terminal or use the LogViewer app.

Note: Enable developer mode for debugging to keep logs until the next reboot. Without developer mode enabled, logs will get deleted after you close the app.

Debugging in the browser

The default Ubuntu Touch browser is based on the Blink technology that is also used in Chrome/Chromium. By starting the browser in a special mode, you have access to the regular Chrome-style debugger.

On your phone, start the browser in inspector mode::

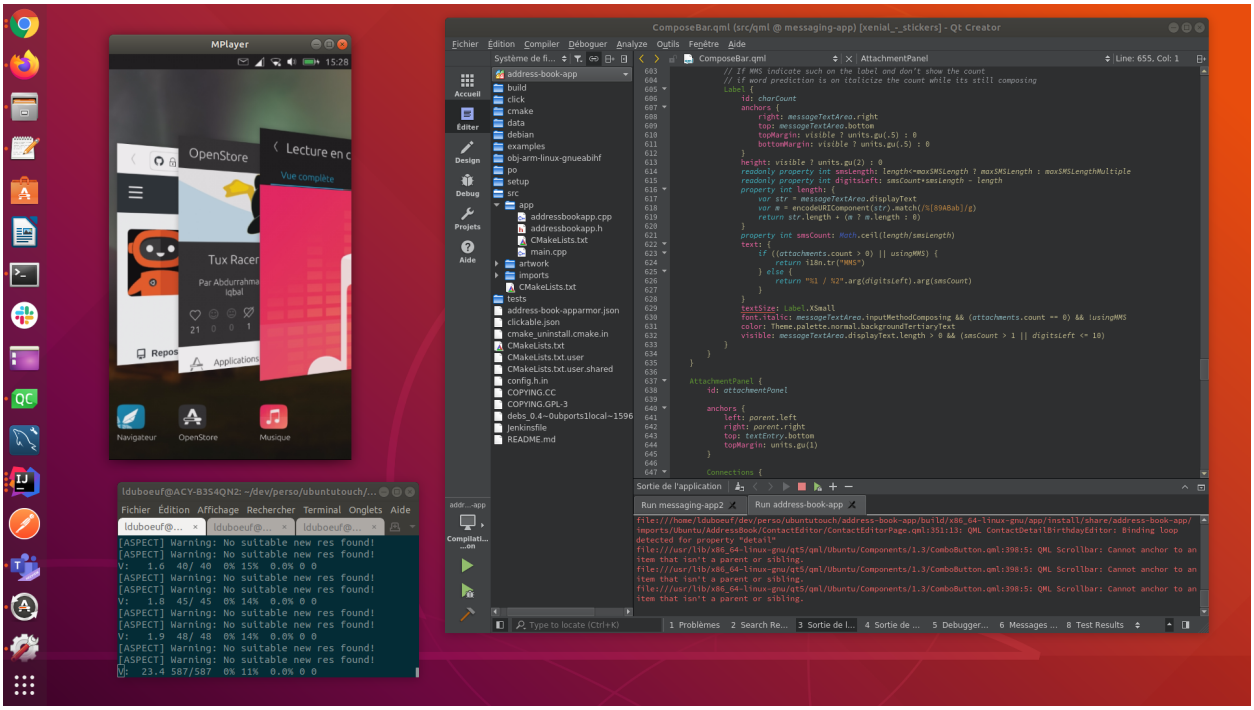
```
export QTWEBENGINE_REMOTE_DEBUGGING=0.0.0.0 9221
morph-browser --desktop_file_hint=/usr/share/applications/morph-browser.
↪desktop [web_site_url]
```

Now on your computer, launch Chrome/Chromium browser, and point address to `http://YOUR_UT_IP_ADDRESS:9221`

The following sections provide guides and API documentation.

Clickable is the main tool to build apps on Ubuntu Touch. It is an easy to use command line tool with a lot of great features. Clickable allows you to compile, build, test, and publish your app. It also provides various templates to get you started with app development.

You can use any code editor or IDE to work on the code. Then you build your app from the command line with Clickable.



Applications are shipped as a *click* package and can be published on the official app store OpenStore. Note also that most apps in the OpenStore are open source and can serve as references and inspiration.

7.3 References and guides

7.3.1 Code editor integrations

- Atom

  You can use clickable with the Atom Editor by installing the atom-clickable-plugin. Or you can simply run `clickable ide atom` in your project directory to launch an Atom editor instance on top of your docker image. This way you will have all Atom features - including the clickable plugin - available for you project.

- QtCreator

  Available on top of clickable docker image, `clickable ide qtcreator` command within your project directory will launch a QtCreator instance and auto setup the project for you, you will have code completion and navigation for Ubuntu Touch components as well as run/debug facilities.

- Ubuntu SDK (Unmaintained)

  Alternatively there is the old Ubuntu SDK IDE. Be aware that it is no longer supported by Canonical, and UBports has chosen to not support it either due to lack of manpower.

  You can still install the SDK IDE in Ubuntu 16.04, but it is not guaranteed to work correctly. You can use the following commands to install:

```
sudo add-apt-repository ppa:ubuntu-sdk-team/ppa
sudo apt update && sudo apt dist-upgrade
sudo apt install ubuntu-sdk
sudo reboot # or logout/login
```

7.3.2 Developer guides

To get started with your first app check out the following developer guides:

- Qt programming course by mimecar for Ubuntu Touch
- Application Templates by fulvio
- Python examples by pavelprosto
- Python + UT apps quickstart guide by AaronTheIssueGuy
- QML templates by Joan CiberSheep.
- First steps with QML.
- Qt examples and tutorials.

If you are interested in helping to create developer guides check out our GitLab Project.

To get familiar with some more advanced concepts see the following sections:

The content hub - tips and tricks

On Ubuntu Touch the apps are confined. The way of sharing files between them is through the Content Hub, a part of the system that takes care of file import, export and sharing.

Different ways of sharing the content

As we can see in the Content Hub documentation, there are several ways of handling the file to be shared:

- `ContentHandler.Source` (The selected app will provide a file to be imported)
- `ContentHandler.Destination` (The selected app will be the destination for the exported file)
- `ContentHandler.Share` (The selected app will be the destination for the exported file, which will then be shared externally)

Importing

Looking into the code of Webapp Creator, we'll find the code to import an image to be used as icon. Tapping on the place holder will open the Content Hub that will let us choose from where to import the image (see the Webapp Creator source code)

```
ContentPeerPicker {
    anchors { fill: parent; topMargin: picker.header.height }
    visible: parent.visible
    showTitle: false
    contentType: picker.contentType //ContentType.Pictures
    handler: picker.handler //ContentHandler.Source
```

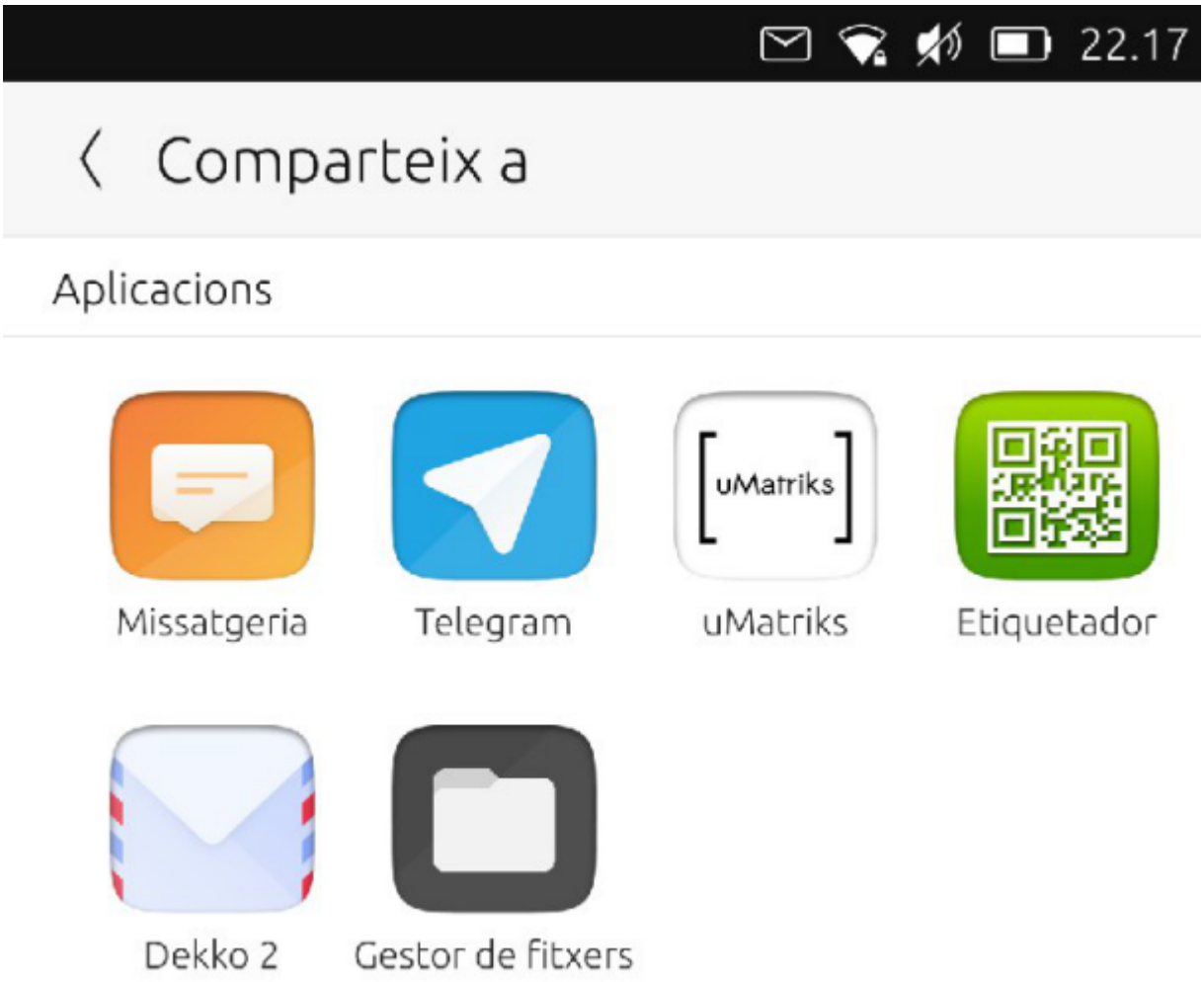


Fig. 1: Content Hub Share Page

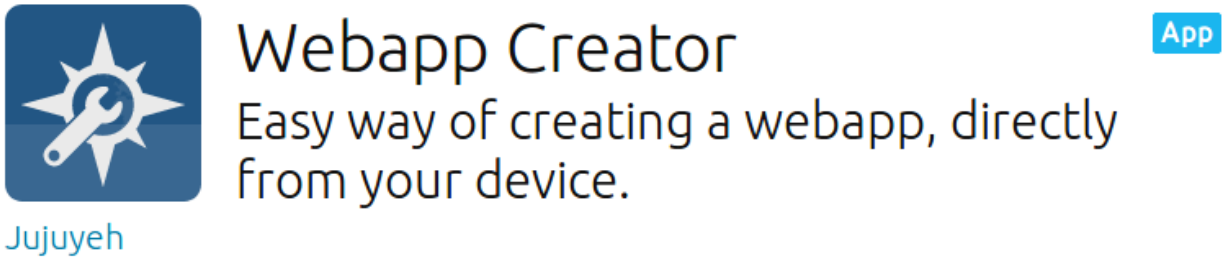


Fig. 2: Webapp Creator on the OpenStore

`ContentPeerPicker` is the element that shows the apps.

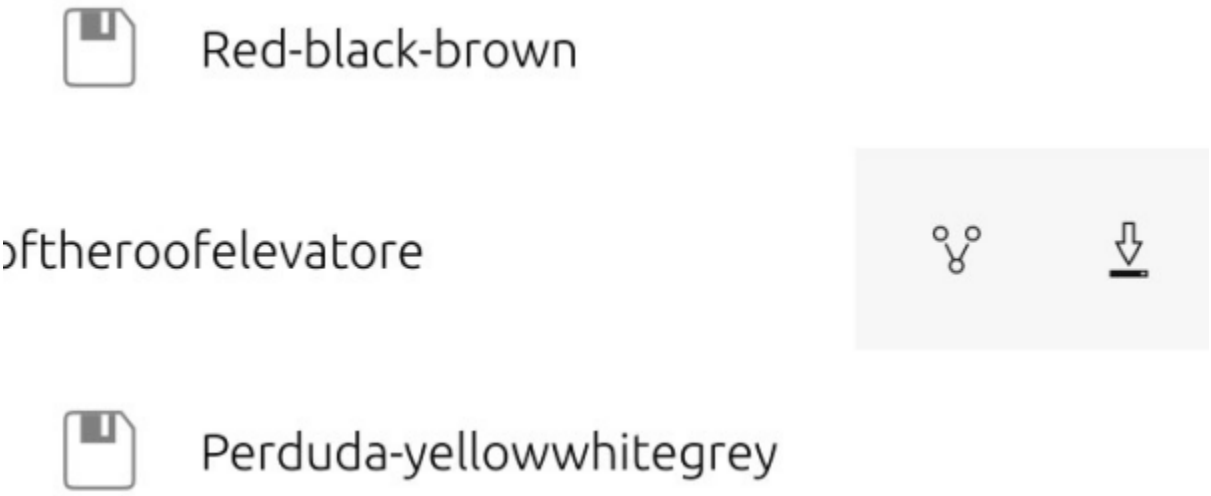
```
var importPage = mainPageStack.push(Qt.resolvedUrl("ImportPage.qml"),{"contentType":␣
↪ContentType.Pictures, "handler": ContentHandler.Source})
```

`contentType` is passed in Main.qml as ContentType.Pictures. So, we will only see apps from which we only can import images. `handler` is passed in the same line as ContentHandler.Source. As we want to import an image from the app selected in the Content Hub.

Exporting

Fig. 3: Gelek in the OpenStore

In Gelek, we are going to end with some saved games that we want to save in our device or share with ourselves (in Telegram and then save them to our computer).



Tapping on the download icon we will get a Content Hub to *save* the game file (which is actually an export).

The game file is a file of type `glksave`. We will tell Content Hub that we are sending a file of type `All` (see the Install Page code).

```
ContentPeerPicker {
    anchors { fill: parent; topMargin: picker.header.height }
    visible: parent.visible
    showTitle: false
    contentType: ContentType.All
```

(continues on next page)

(continued from previous page)

```
    handler: ContentHandler Destination

    onPeerSelected: {
```

`contentType` is `ContentType.All`, so we will only see apps which are able to receive unmarked file types. `handler` is `ContentHandler.Destination`, so the app selected should store the saved game.

Tapping on the File Manager we will save the saved game in the folder we choose.

Sharing

Similarly, tapping on the share icon will allow us to send the saved game through Telegram to ourselves (see the Webapp Creator Import Page source code). Sharing is similar to exporting, except the destination app may share the content externally (for example, over Telegram or text message).

```
ContentPeerPicker {
    anchors { fill: parent; topMargin: picker header height }
    visible: parent visible
    showTitle: false
    contentType: picker contentType //ContentType.Pictures
    handler: picker handler //ContentHandler.Source

    onPeerSelected: {
```

The only difference between this and the previous code is that `handler` is `ContentHandler.Share`.

Wait a minute. Why the different apps?

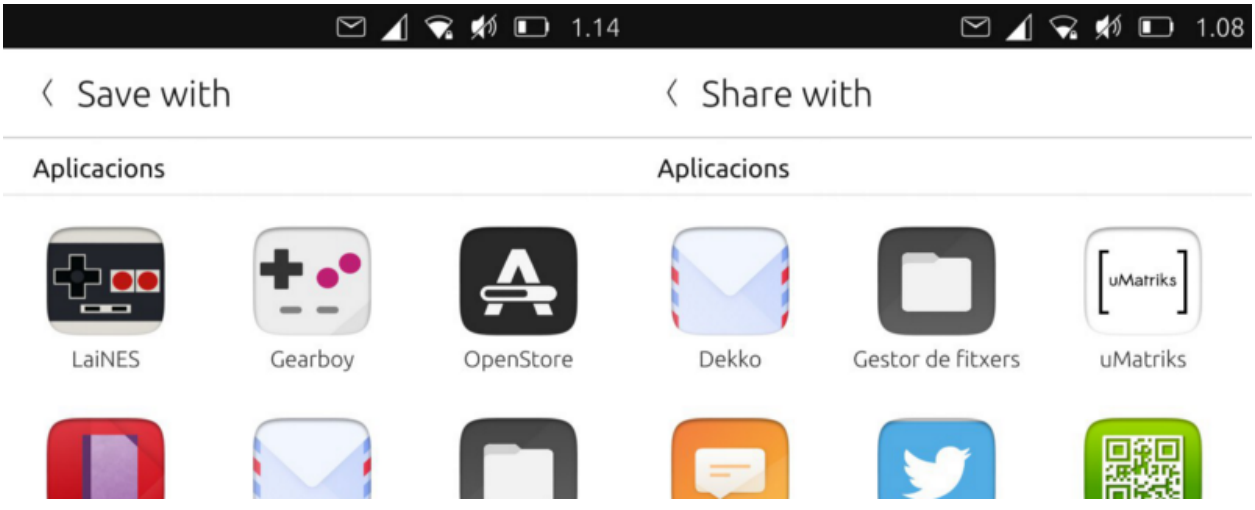


Fig. 4: Content Hub: Export vs Share

Each developer can decide the rules each app would follow in relation to the Content Hub. Why the OpenStore is shown as the destination of an export?

Let's check its manifest.json

```
"hooks": {
    "openstore": {
        "apparmor": "openstore/openstore.apparmor",
        "desktop": "openstore/openstore.desktop",
        "urls": "openstore/openstore.url-dispatcher",
        "content-hub": "openstore/openstore-contenthub.json"
    }
},
```

The above code defines that the hooks for the app named `"openstore"` in relation to the `"content-hub"` should follow the rules defined in `openstore-contenthub.json`

```
{
    "destination": [
        "all"
    ]
}
```

This means, the OpenStore will be the destination for *all* `ContentTypes`.

What about uMatriks? Let's see its `content-hub.json`

```
{
    "destination": [
        "pictures",
        "documents",
        "videos",
        "contacts",
        "music"
    ],
    "share": [
        "pictures",
        "documents",
        "videos",
        "contacts",
        "music"
    ],
    "source": [
        "pictures",
        "documents",
        "videos",
        "contacts",
        "music"
    ]
}
```

So, with this example, uMatriks will be able to be the destination, source and share app for all kinds of `ContentType`. What about the other hooks in the `manifest.json`? That is discussed in the next guide.

Importing from Content Hub and URLdispatcher

```
"hooks": {
    "openstore": {
        "apparmor": "openstore/openstore.apparmor",
        "desktop": "openstore/openstore.desktop",
        "urls": "openstore/openstore.url-dispatcher",
        "content-hub": "openstore/openstore-contenthub.json"
    }
},
```

In the previous guide we have seen a little bit about how Content Hub works. In this guide we will see how `URLdispatcher` works and how to handle imported data from the Content Hub.

Handle data from the Content Hub

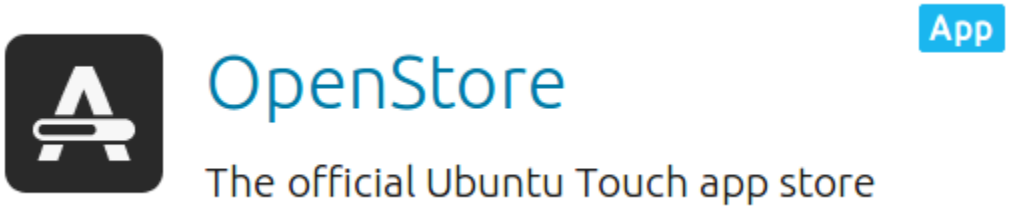
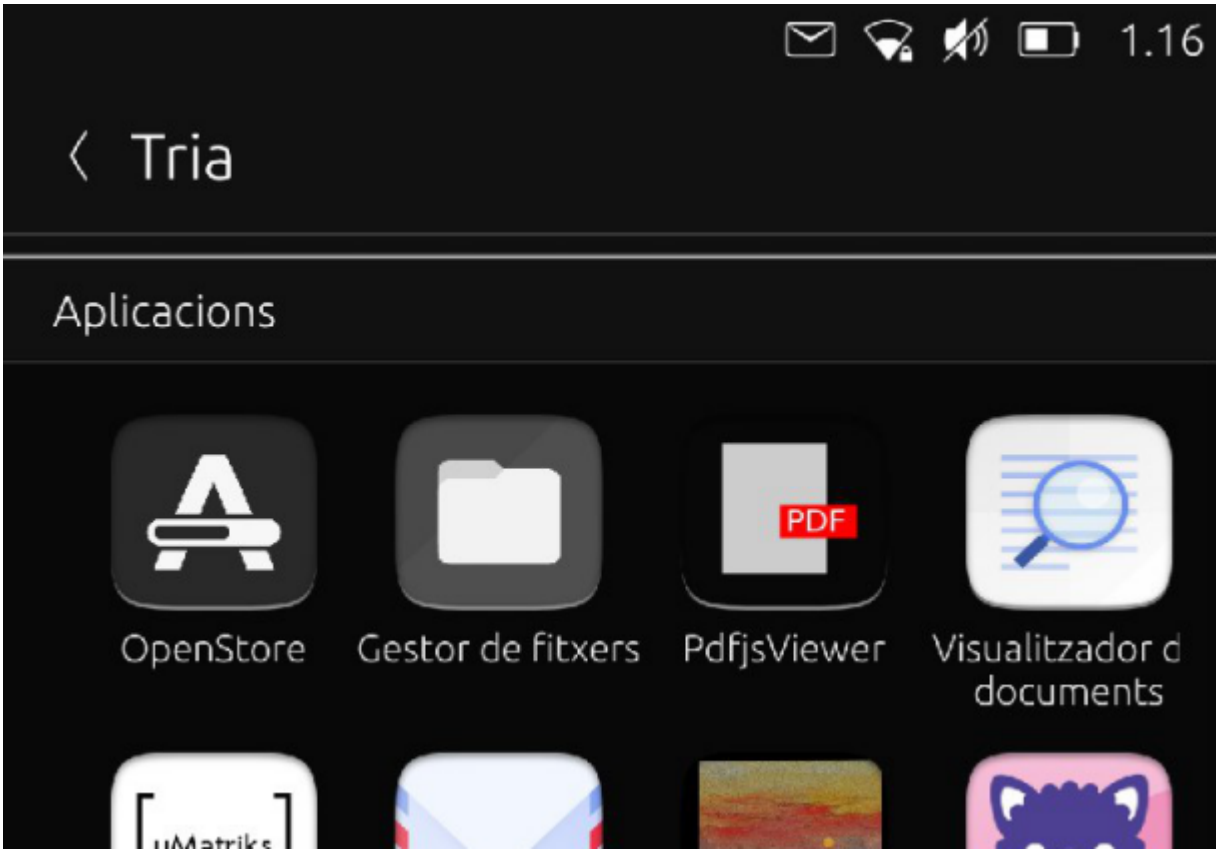


Fig. 5: OpenStore app from open-store.io

One of the easiest ways of testing an app, is to send a test click to yourself on Telegram, and opening that click file with the OpenStore through the Content Hub:



If we tap on the OpenStore app, it will be opened and it will ask if we want to install the click file. Let's take a look into the Main.qml code of the app to see how it is done:

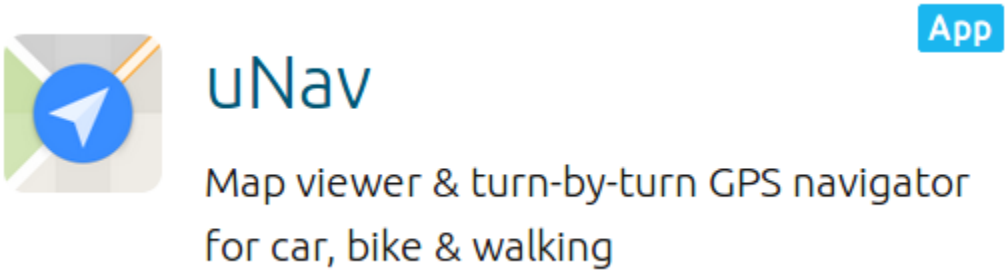
```
Connections {
        target: ContentHub

        onImportRequested: {
            var filePath = String(transfer.items[0].url).replace('file://', '')
            print("Should import file", filePath)
            var fileName = filePath.split("/").pop();
            var popup = PopupUtils.open(installQuestion, root, {fileName: fileName});
            popup.accepted.connect(function() {
                contentHubInstallInProgress = true;
                PlatformIntegration.clickInstaller.installPackage(filePath)
            })
        }
}
```

Do you see that Connections element that targets the ContentHub? When it receives the signal onImportRequested, it will take the url of the data sent from the Content Hub (`transfer.items[0].url` is the url of the first data sent) and open a `PopUp` element to let the user install the click.

What about the URLdispatcher?

The URL dispatcher is a piece of software, similar to the Content Hub, that makes our life easier trying to choose the correct app for a certain protocol. For example: We probably want to open the web browser when tapping on an http protocol. If we tap on a map link it is handy to open it with uNav or to open a twitter link in the Twitter app! How does that work?

The `URLdispatcher` selects which app (according to their `manifest.json`) will open a certain link.



Let's see how our navigation app looks inside. uNav's manifest shows special hooks for the `URLdispatcher` in its manifest.json code:

```
1  [
2      {
3          "protocol": "http",
4          "domain-suffix": "map.unav.me"
5      },
6      {
7          "protocol": "http",
8          "domain-suffix": "unav-go.github.io"
9      },
10     {
11         "protocol": "geo"
12     },
13     {
14         "protocol": "http",
15         "domain-suffix": "www.openstreetmap.org"
16     },
17     {
18         "protocol": "http",
19         "domain-suffix": "www.opencyclemap.org"
20     },
21     {
22         "protocol": "https",
23         "domain-suffix": "maps.google.com"
24     }
25 ]
```

This means that a link that looks like http://map.unav.me/xxxxx,xxxxx will be opened in uNav. And that's defined in lines 2 and 3, where it looks for protocol http followed by map.unav.me.

Also, a URI formatted geo:xxx,xxx should open in uNav, as it's defined in line 11.

And how do we manage the received URL?

After the `URLdispatcher` sends the link to the correspondent app, we need to handle that URL or URI in the targeted app. Let's see how to do that:

In the main qml file, we need to add some code to know what to do with the dispatched URL. Let's check how Linphone app manages this adding a connection to the URI Handler with a Connections element setting `UriHandler` as a target.

```
Connections {
    target: UriHandler

    onOpened: {
        console.log('Open from UriHandler')

        if (uris.length > 0) {
            console.log('Incoming call from UriHandler ' + uris[0]);
            showIncomingCall(uris[0]);
        }
    }
}
```

This code will manage an URI in the form `linphone://sip:xxx@xxx.xx` when the app is opened. But what do we need to do to handle this link when the app is closed?

We need to add a little bit extra code that will cover two cases: 1) We receive one URL 2) We receive more than one URL

Let's check if `Qt.application.arguments` is not empty and if not, if any argument matches our URI format.

```
Component.onCompleted: {
    //Check if opened the app because we have an incoming call
    if (Qt.application.arguments && Qt.application.arguments.length > 0) {

        for (var i = 0; i < Qt.application.arguments.length; i++) {
            if (Qt.application.arguments[i].match(/^linphone/)) {
                console.log("Incoming Call on Closed App")
                showIncomingCall(Qt.application.arguments[i]);
            }
        }
    }

    //Start timer for Registering Status
    checkStatus.start()
}
```

Remember to check that %u (to receive 1 URL) or %U (to receive 1 or more URLs) is present under the `Exec` line in the .desktop file of the app.

Tools

From command line, `url-dispatcher-dump` command will give you the full list of registered protocol schemes and their corresponding app.

Another usefull tool, but not installed by default on devices is `url-dispatcher-tools`, it allows you to simulate a call from a third party app. e.g: `url-dispatcher https://youtu.be/CIX-a-i6B1w` will launch youtube webapp.

To install, it make your partition writable (`sudo mount -o rw,remount /`) and install it via `sudo apt install url-dispatcher-tools`

What happens if more than one app has the same URL type defined?

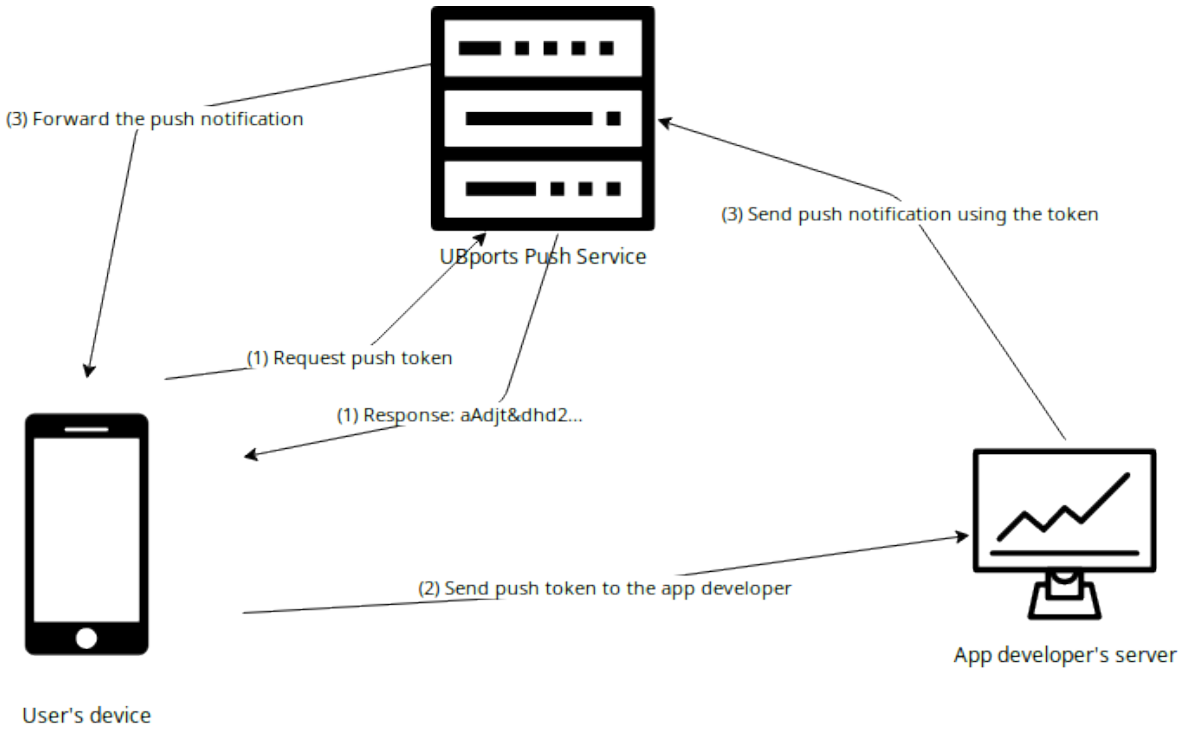
A very good question. What happens if we tap on a Twitter link? How is such a URL handled by the `URLdispatcher` as protocol `http` or the protocol `http://twitter`?

What happens if two apps have the same defined protocol?

Now it's time to do some tests and share the results in the next guide.

Push notifications

Let's assume that you have an app created with Clickable and published on the OpenStore. But now you want to be able to send Push Notifications to your users. First of all, you need to understand how this is working:



1. Every app which uses push notifications has got a unique token. This token identifies the user, the device and the app itself. The token is generated by the UBports Push Service.
2. You will need the token to send a push notification. So the app sends its token to the app developer's server.
3. With the token you can send a HTTP request to the UBports Push Server which will forward the notification the user's device.

Let's practice this step-by-step.

Note: In the following example we will not implement a server. Also the communication between your app and your server is up to you. Please inform the user about the communication with your server by providing a privacy policy!

Make the app ready for push notifications

Implementing the PushClient

First we need to add the policy group "push-notification-client". Your apparmor file could look like this:

```
    {
        "policy_groups": [
            "networking",
            "push-notification-client"
        ],
        "policy_version": 16.04
    }
```

In the next step we need to modify the Qml parts. We need to add a pushclient component:

```
//...

import Ubuntu PushNotifications 0.1

//...

PushClient {
        id: pushClient
        appId: "myapp.yourname_hookname"
        onTokenChanged: console.log("", pushClient.token)
}
```

You need to set the correct appId! If the app name in your manifest file is myapp.yourname and the name of the main hook (the one which handles the .desktop file) is hookname, then the appId is: myapp.yourname_hookname. When we now start the app, it will get a token and print this token in the logs. With clickable logs we will be able to copy this token out of the terminal. But the app is not yet ready to receive a push notification. For this we need something called a pushhelper!

Implementing the pushhelper

The pushhelper is a part of the app which will receive all push notifications and process them before sending them to the system notification center. It will receive a json-file and must output another json-file in the correct format. The pushhelper is seperated from the app. So we need a new hook in the manifest. It could look like this:

```
    {
        //...

        "title": "myapp",
        "hooks": {
```

(continues on next page)

(continued from previous page)

```
            "myapp": {
                "apparmor": "myapp.apparmor",
                "desktop":  "myapp.desktop"
            },
            "push": {
                "apparmor": "push-apparmor.json",
                "push-helper": "push.json"
            }
        },

        //...
    }
```

It should be clear that we now need a different apparmor file and a different executable file. The **push-apparmor.json** file must only contain the policy group push-notification-client and should look like this:

```
{
    "template": "ubuntu-push-helper",
    "policy_groups": [
        "push-notification-client"
    ],
    "policy_version": 16.04
}
```

The **push.json** is for redirecting to the executable file:

```
{
    "exec": "pushexec"
}
```

In our tutorial we will use python to create a executable named **pushexec** which will forward the notification without changing anything:

```
#!/usr/bin/python3

import sys

f1, f2 = sys.argv[1 3]

open(f2, "w").write(open(f1).read())
```

We also need to add this new files to the **CMakeLists.txt** and make the pushexec executable:

```
[...]

install(FILES pushexec PERMISSIONS OWNER_EXECUTE OWNER_WRITE OWNER_READ DESTINATION $
↪{DATA_DIR})
install(FILES push.json DESTINATION ${DATA_DIR})
install(FILES push-apparmor.json DESTINATION ${DATA_DIR})

[...]
```

Now the app is ready to receive and process push notifications!

Using the Push Service API

So now you have the token and the app is ready to receive and process push notifications. To send a notification, you need to send a HTTP request to this address: https://push.ubports.com/notify The content-type must be application/json and it must fit in the correct format. A example in javascript could look like this:

```
var req = new XMLHttpRequest();
req.open("post", "https://push.ubports.com/notify", true);
req.setRequestHeader("Content-type", "application/json");
req.onreadystatechange = function() {
        if (req.readyState === XMLHttpRequest.DONE) {
                        console.log(" Answer:", req.responseText);
        }
}
var approxExpire = new Date();
approxExpire.setUTCMinutes(approxExpire.getUTCMinutes()+10);
req.send(JSON.stringify({
        "appid" : "appname.yourname_hookname",
        "expire_on": approxExpire.toISOString(),
        "token": "aAnqwiFn§DF%2",
        "data": {
                "notification": {
                        "card": {
                                "icon": "notification",
                                "summary": "Push Notification",
                                "body": "Hello world",
                                "popup": true,
                                "persist": true
                        },
                "vibrate": true,
                "sound": true
                }
        }
}));
```

Push Notification Object

Parameter	Type	Description
appid	string	Required. ID of the application that will receive the notification, as described in the client side documentation.
expire_on	string	Required. Expiration date/time for this message, in ISO8601 Extendendformat.
token	string	Required. The token identifying the user+device to which the message is directed, as described in the client side documentation.
clear_pending	bool	Discards all previous pending notifications. Usually in response to getting a "too-many-pending" error. Defaults to false.
replace_tag	string	If there's a pending notification with the same tag, delete it before queuing this new one.
data	Data	A JSON object. The contents of the data field are arbitrary. We can use it to send any data to the app.

Data

Parameter	Type	Description
notification	Notification	A JSON object which defines how this notification will be presented.
message	object	A JSON object that is passed as-is to the application via PopAll.

Notification

Parameter	Type	Description
tag	string	The tag of the push notification.
sound	bool or string	This is either a boolean (play a predetermined sound) or the path to a sound file. The user can disable it, so don't rely on it exclusively. Defaults to empty (no sound). The path is relative, and will be looked up in (a) the application's `.local/share/<pkgname>`, and (b) standard xdg dirs.
vibrate	bool or Vibrate	The notification can contain a vibrate field, causing haptic feedback, which can be either a boolean (if true, vibrate a predetermined way) or an Vibrate object.
emblem-counter	Emblem-counter	A JSON object, which defines how to display the emblem counter.
card	Card	A JSON object with information about the notification card.

Card

Parameter	Type	Description
summary	string	Required. A title. The card will not be presented if this is missing.
body	string	Longer text, defaults to empty.
actions	array	If empty (the default), a bubble notification is non-clickable. If you add a URL, then bubble notifications are clickable and launch that URL. One use for this is using a URL like `appid://com.ubuntu.developer.ralsina.hello` which will switch to the app or launch it.
icon	string	An icon relating to the event being notified. Defaults to empty (no icon); a secondary icon relating to the application will be shown as well, regardless of this field.
timestamp	integer	Seconds since the unix epoch, only used for persist for now. If zero or unset, defaults to current timestamp.
persist	bool	Whether to show in notification centre; defaults to false.
popup	bool	Whether to show in a bubble. Users can disable this, and can easily miss them, so don't rely on it exclusively. Defaults to false.

Vibrate

Parameter	Type	Description
pattern	array	A list of integers describing a vibration pattern (duration of alternating vibration/no vibration times, in milliseconds).
repeat	integer	Number of times the pattern has to be repeated (defaults to 1, 0 is the same as 1).

Emblem-Counter

Parameter	Type	Description
count	integer	A number to be displayed over the application's icon in the launcher.
visible	bool	Set to true to show the counter, or false to hide it.

User Metrics

What are User Metrics?

If you look on the lock screen, you will see a circle. Inside the circle is text. Look closer, and you'll notice that the text contains data regarding the user's activity. Double tap on the middle of the circle, and you will see more "metrics" about the user.

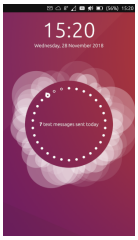


Fig. 6: This shows "7 text messages sent today." How did it know?

For the most part, these messages are quite clearly state what they are counting, and which app is related. But where do these metrics come from?

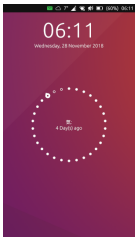


Fig. 7: This is from a 3rd-party application (nCounter) that makes use of the User Metrics feature.

How can I use User Metrics in my application?

All of the following information will be based on the code for nCounter.

Your app's apparmor file must include `usermetrics` in the policy:

```
{
    "policy_groups": [
        "usermetrics"
    ],
    "policy_version": 16.04
}
```

Next, you will need to import the module in the QML file that will handle the User Metrics:

```
import UserMetrics 0.1
```

(There may be updated versions of this above 0.1)

Next, the specific Metric must be defined in the code as an object:

```
Metric { // Define the Metric object.
    property string circleMetric // Create a string-type variable called "circleMetric".␣
↪This is so you can update it later from somewhere else.
    id: metric // A name to reference the metric elsewhere in the code. i.e. when␣
↪updating format values below.
    name: "nCounter" // This is a unique ID for storing the user metric data
    format: circleMetric // This is the metric/message that will display "today". Again␣
↪it uses the string variable that we defined above
    emptyFormat: i18n.tr("Check nCounter") // This is the metric/message for tomorrow.␣
↪It will "activate" once the day roles over and replaces "format". Here I have use a␣
↪simple translatable string instead of a variable because I didn't need it to change.
    domain: "ncounter.joe" // This is the appname, based on what you have in your app␣
↪settings. Presumably this is how the system lists/ranks the metrics to show on the␣
↪lock screen.
}
```

Now that the metric is created, we can update the "format" or "emptyFormat" when an event takes place by referencing the variables in the Metric object.

```
onButtonPressed: {
    metric.circleMetric = "New Metric Message"
    metric.update(0)
```

(continues on next page)

(continued from previous page)

```
    console.log("Metric updated")
}
```

Here we assign a new value to the circleMetric string variable that's inside the Metric object:

(Remember that circleMetric is the variable value assigned to format)

Metric Id [dot] Variable Name [equals] New variable information

```
metric.circleMetric = "New Metric Message"
```

We then tell the lock screen to update the metric.

Metric ID [dot] update (specific amount to set if included in the format)

```
metric.update(0)
```

(Note: In this example, `0` is arbitrary since the metric value doesn't include a counter)

We have now updated the metric for today. When the time rolls over to tomorrow, the metric will be reset to whatever is in `emptyFormat`.

For most apps, this defaults to 0 counts for messages, calls, etc.

How do User Metrics work?

User Metrics are made up of two "formats":

- metrics/messages for today (`format`)
- metrics/messages for tomorrow (`emptyFormat`)

The value of `emptyFormat` is what displays on the lock screen when no value has been stored in `format`. In order to display a new value of `format` the metric must be updated.

There are two options for updating the metric:

- Set the metric to a specific amount:

`metricID.update(x)` (where x is a number of type *double* to set for a counter value). `metricID` is the `id:` specified in the Metric item. The counter value can be included in the `format` setting by using `%1`. e.g. `format: "%1 buttons pressed today"`

- Increment the metric:

`metricID.increment(x)` (where x is the amount to add to the current counter)

The metric will reset back to the value stored in `emptyFormat` each day.

Applications make use of User Metrics by setting and updating the "formats" whenever a certain event takes place. e.g. When you press send in Telegram, or when you receive a phone call. The application may store the data for manipulation, but generally the data is stored in the system (/var/lib/usermetrics).

(See this blog post for a simple example)

Limitations and wonders

Once a metric is registered, it remains on the lock screen even after the app has been uninstalled. A database file (db) is stored in */var/lib/usermetrics*, which can be deleted by root (but not with sudo). Deleting this file and rebooting will remove all stored metrics. Presumably, the db file could be edited instead of deleted.

Based on how the "formats" are set up, it seems that it is difficult to maintain a running tally beyond one day (though not impossible. See FluffyChat).

In the case of the nCounter app. I wanted to count the number of days, but since the metric "resets" each day, that presents a problem. I created a workaround that updates the metric every time the application is opened. Thus, the `emptyFormat` (default) tells the user to open the application. This, however, nearly defeats the purpose of the metric entirely, other than having a neat stat reminder for the day.

There must be a way for a process to run independently in the background (e.g. cron) to retrieve data from a specific app code. One lead is the Indicator Weather app. This runs a process every X minutes to update the weather indicator automatically without having to open the app.

Writable directories

App confinement is part of the Ubuntu Touch security concept. Data can be exchanged between apps only according to the AppArmor policies, mainly using the *ContentHub*. This being said, apps can only read and write files that are located in one of three app specific directories explained in this guide.

Standard Paths

Besides the write access to the app directories explained below, the app can write debug messages into the app log file located at `/home/phablet/.cache/upstart/application-click-<fullappname>_<appname>_<version>.log`. To append messages to the log file, use the Qt debug functions.

Config

Path: `/home/phablet/.config/<fullappname>/`

This is the place to store configurations files to. The music app for example stores its configuration to `/home/phablet/.config/com.ubuntu.music/com.ubuntu.music.conf`.

Cache

Path: `/home/phablet/.cache/<fullappname>/`

This is the place to cache data for later use. The cache is also used by the Content Hub. Files that have been shared with the music app for example can be found in `/home/phablet/.cache/com.ubuntu.music/HubIncoming/`.

App data

Path: `/home/phablet/.local/share/<fullappname>/`

This is where your app stores any data. The music app for example stores its data bases to `/home/phablet/.local/share/com.ubuntu.music/Databases/`.

Using Standard Paths in C++

The Qt header `QStandardPaths` provides the app's writable locations in C++:

```
#include <QStandardPaths>
...
QString configPath = QStandardPaths::writableLocation(QStandardPaths::AppConfigLocation);
QString cachePath = QStandardPaths::writableLocation(QStandardPaths::CacheLocation);
QString appDataPath = QStandardPaths::writableLocation(QStandardPaths::AppDataLocation);
...
```

Since the value of the QStandardPaths strings are decided by the Qt application name, this needs to be the same as the clickable application name.

Using Standard Paths in QML

The Qt module Qt.labs.platform provides the app's writable locations in QML:

```
import Qt.labs.platform 1.0
...
Label
{
    text: StandardPaths.writableLocation(StandardPaths.AppConfigLocation)
}
Label
{
    text: StandardPaths.writableLocation(StandardPaths.CacheLocation)
}
Label
{
    text: StandardPaths.writableLocation(StandardPaths.AppDataLocation)
}
...
```

Notice that `QStandardPaths` returns paths (`'/phablet/home/...'`), and QML `StandardPaths` returns urls (`'file:///phablet/home/...'`). This must be considered specially if the app is sharing location strings between QML and C++.

Handle C++ dependencies with Clickable

If an app depends on a library that is not pre-installed in Ubuntu Touch, the app needs to ship it inside the click package. This guide shows how this can be done with Clickable.

Building

Sometimes libraries are available pre-built, allowing you to skip the build step. Compiling your dependency is recommended though, as it gives you more control over the whole process and may lead to better results (e.g. better performance or smaller package sizes).

Compilation

Put the library's source code at `libs/LIBNAME` (replacing `LIBNAME` by the library's name), because this is where clickable will look for it by default. If the dependency source code is available as a git repository, it is a good idea to add it as a git submodule. Otherwise add a script to download the sources.

Add a libraries section to your clickable.json, like this:

```
{
  "builder": "cmake",
  "libraries": {
    "LIBNAME": {
      "builder": "cmake"
    }
  }
}
```

If the library does not contain a CMake configuration, you need to use the *qmake* or *custom* builder instead.

Optionally, configure the compilation by adding `build_args`, which may look like this:

```
{
  "builder": "cmake",
  "libraries": {
    "LIBNAME": {
      "builder": "cmake",
      "build_args": [
        "-DBUILD_EXAMPLES=OFF",
        "-DBUILD_DOCS=OFF",
        "-DBUILD_TESTS=OFF",
        "-DBUILD_SHARED_LIBS=OFF"
      ]
    }
  }
}
```

Most build arguments are project specific. Therefore, study the library's build instructions and also look for `option` settings in its `CMakeLists.txt` .

To actually build the library for all architectures run `clickable build-libs --arch armhf`, `clickable build-libs --arch arm64` and `clickable build-libs --arch amd64`. Don't forget to mention this step in your README, so that others can reproduce the build process.

See how Teleports clickable.json uses the libraries feature to build its dependency tdlib.

Pre-built

If you compiled your library as described above, skip this step.

A pre-built library is usually only available as a shared object that needs to be linked dynamically. Furthermore, it may contain components that you don't need, resulting in a bloated app. It may even miss something that you could achieve by compiling it yourself. Sometimes, a library is available in the Ubuntu Repositories, but is not installable for the architecture you need (likely `armhf` or `arm64`). In this case you have to compile the library as described above.

If the library is available in the Ubuntu Repositories, you can add it to the dependencies list, like this:

```
{
  "builder": "cmake",
  "dependencies_target": [
    "libsomething-dev"
  ]
}
```

Clickable will install the specified package automatically for the target architecture inside the build container. An example can be found in Guitar Tools' clickable.json.

If the library can be found in a PPA, you can add the PPA to the clickable.json, too. For example:

```
{
  "builder": "cmake",
  "dependencies_ppa": [
    "ppa:someone/libsomething"
  ],
  "dependencies_target": [
    "libsomething-dev"
  ]
}
```

Otherwise add a script to download the pre-built library.

Usage

First, you need to specify the include directory where the headers can be found that you include into the app's source code. Second, you need to link the library's binary against your app's binary.

In case the library contains a CMake configuration file, you can use the find_package command, which provides you with a target to link against or variables pointing to the library's binary and include directories. The additional lines on your CMakeLists.txt then may look like:

```
find_package(SOMELIBRARY REQUIRED)

include_directories(${SOMELIBRARY_INCLUDE_DIRS})
target_link_libraries(mytarget ${SOMELIBRARY_LIBS})
```

The `find_package` command usually defines the path to the include directory as `SOMELIBRARY_INCLUDE_DIRS` and the library's binaries as `SOMELIBRARY_LIBS` (check on the library's documentation on what their CMake configuration provides exactly). Use them with the `include_directories` and `target_link_libraries` commands. See the Camera Scanner ImageProcessing CMakeLists.txt for a real world example.

At this point you should already be able to test with Clickable's desktop mode running `clickable desktop`.

Deployment

If you linked the library statically, you can skip this step, as the library's binary is already inside your app's binary.

Find out which shared object files (`*.so`) you need to ship. You can do so by starting the app on your device via `clickable && clickable logs`. You should see an error message telling you which shared object file was missing.

Find the path to the shared object files. For libraries built via clickable, they are located somewhere in the library's install dir, which is located inside the library's build dir by default (e.g. `build/arm-linux-gnueabihf/opencv/install`). For pre-built libraries run `clickable run "find / -name 'libSomething.so'"` (replacing `libSomething.so` by the file your are looking for). This should print the path to the file (along with some error messages you can ignore). In general, `/usr/lib` is a good bet when looking for the shared object files.

To get the files into the click package, add the install_lib key to your clickable.json:

```
{
  "builder": "cmake",
  "libraries": {
    "LIBNAME": {
      "builder": "cmake",
      "install_lib": [
        "$LIBNAME_LIB_INSTALL_DIR/usr/lib/$ARCHITECTURE_TRIPLET/libqmapboxgl.so*"
        "/usr/lib/$ARCHITECTURE_TRIPLET/libSoundTouch.so.*"
      ]
    }
  }
}
```

The lines above contain two examples. The first one installing a library built with Clickable. The asterisk in `.so*` helps to catch symbolic links along with the actual library which are used to point to the current version.

Depending on the library, you might need to ship some additional data. In that case consult the library's readme or install instructions.

You should be set up to build and install your click package on a device connected to your computer now by running `clickable`.

Playground

In a completely free and open source community, it is natural to have community members exploring the limits of the platform in many many directions. In this section you will find links to external resources that do exactly that: Explore. The purpose of this list is to show the unlimited possibilities of an open platform like Ubuntu Touch.

Note: The resources listed here do not necessarily represent the officially endorsed way of developing applications for Ubuntu Touch, but are interesting experiments.

- Free Pascal development for Ubuntu Touch
- Lazarus development for Ubuntu Touch
- Geany on Ubuntu Touch device as text editor, source code editor, debugger and compiler for multiple languages
- Deploy your existing Qt QML app to Ubuntu Touch using QMake (Video)
- Deploy your existing Qt QML app to Ubuntu Touch using CMake (Video)
- Getting started with Clickable (Video)

- Content Hub overview (Video)
- Import files to your app with Content Hub (Video)

7.3.3 API docs

In this section you find the various APIs you can use to develop Ubuntu Touch apps.

Firstly there are the QML and HTML APIs that you can use for the front end:

- QML API
- All 5.12 QML types
- Cordova HTML5 API

Secondly, the platform documentation describes how to integrate your app into the system:

Platform

Here are Ubuntu Touch platforms key topics when you want to extend your app with Ubuntu Touch ecosystem:

Content Hub Each application can expose content outside its sandbox, giving the user precise control over what can be imported, exported or shared with the world and other apps.

Push notifications By using a push server and a companion client, instantly serve users with the latest information from their network and apps.

URL dispatcher Help users navigate between your apps and drive their journey with the URL dispatcher.

Online accounts Simplify user access to online services by integrating with the online accounts API. Accounts added by the user on the device are registered in a centralized hub, allowing other apps to re-use them.

Read the docs

Packaging your app

Here you will get some informations about the confinment model and the packaging system:

Click package

Every `click` application package must embed at least 3 files:

`manifest.json` file Contains application declarations such as application name, description, author, framework sdk target, and version.

Example `manifest.json` file:

```
{
    "name": "myapp.author",
    "title": "App Title",
    "version": "0.1"
    "description": "Description of the app",
    "framework": "ubuntu-sdk-16.04",
    "maintainer": "xxxx <xxx@xxxx>",
    "hooks": {
```

(continues on next page)

(continued from previous page)

```
        "myapp": {
            "apparmor": "apparmor.json",
            "desktop": "app.desktop"
        }
    }
}
```

AppArmor profile policy file Contains which policy the app needs to work properly. See *Security and app isolation* below for more information on this file.

`.desktop` file The launcher file will tell UT how to launch the app, which name and icon to display on the home screen, and some other properties.

Example of `app.desktop`:

```
[Desktop Entry]
Name=Application title
Exec=qmlscene qml/Main.qml
Icon=assets/logo.svg
Terminal=false
Type=Application
X-Ubuntu-Touch=true
```

Non exhaustive list of properties:

- Name: Application title has shown in the dash
- Exec: Path to the executable file
- Icon: Path to the icon to display
- Terminal: `false` if will not run in terminal window
- Type: Specifies the type of the launcher file. The type can be Application, Link or Directory.
- X-Ubuntu-Touch: `true` to make the app visible
- X-Ubuntu-XMir-Enable: `true` if your app is built for X
- X-Ubuntu-Supported-Orientations: `landscape` or `portrait` to force your app start in landscape mode and portrait mode respectively.

Security and app isolation

All Ubuntu apps are confined respecting AppArmor access control mechanism (see Application Confinement) , meaning they only have access to their own resources and are isolated from other apps and parts of the system. The developer must declare which policy groups are needed for the app or scope to function properly with an apparmor `.json` file.

Example `apparmor.json` file:

```
{
    "policy_version": 16.04,
    "policy_groups": [
        "networking",
        "webview",
        "content_exchange"
    ]
```

(continues on next page)

(continued from previous page)

```
}
```

For a full list of available policy groups, see *AppArmor Policy Groups*.

AppArmor Policy Groups

This document contains a full list of Ubuntu Touch's available policy groups and a description of what they give your app permission to access.

Each entry follows this format

```
Title
-----

Description: Description from apparmor file

Usage: How common it is to use this policy (from apparmor file)

Optional longer description
```

Policy usage affects whether your app will be accepted by the OpenStore. Apps containing policies with common usage are generally accepted immediately, while reserved usage policies will need to be manually reviewed.

Note: Coding tip: Everytime you change your apparmor policy file you need to update your app's version for this to be taken into account.

accounts

Description: Can use Online Accounts.

Usage: common

The accounts policy gives your app the permissions it needs to access the Online Accounts API.

audio

Description: Can play audio (allows playing remote content via media-hub)

Usage: common

The audio policy is needed for your app to play audio via pulseaudio or media-hub. The permission also gives it the ability to send album art to the thumbnailer service, which is then shown on the sound indicator.

bluetooth

Description: Use bluetooth (bluez5) as an administrator.

Usage: reserved

This policy grants unrestricted access to Bluetooth devices. It is provided for administration of bluetooth and as a stepping stone towards developing a safe bluetooth API all apps can access.

calendar

Description: Can access the calendar.

Usage: reserved

Calendar grants access to the Evolution dataserver's calendar and alarms APIs. It also grants access to sync-monitor.

This policy is reserved since it grants free access to all calendars on the device at any time. The legacy bug about this situation is LP #1227824 .

camera

Description: Can access the camera(s)

Usage: common

The camera policy grants access to device cameras.

connectivity

Description: Can access coarse network connectivity information

Usage: common

The connectivity policy allows apps to determine rough information about the device's connectivity. This includes whether the device is connected to the Internet and whether it is connected via a Wi-Fi or mobile data connection.

contacts

Description: Can access contacts.

Usage: reserved

The contacts policy allows apps to access the device user's contacts list. It is marked as reserved because it allows access to sync-monitor and unfettered access to the address book.

content_exchange

Description: Can request/import data from other applications

Usage: common

Using the content_exchange policy allows your app to be a consumer of content on content-hub.

content_exchange_source

Description: Can provide/export data to other applications

Usage: common

The content_exchange_source policy allows your app to provide content on content-hub.

debug

Description: Use special debugging tools. This should only be used in development and not for production packages. Note: use of this policy group provides significantly different confinement than normal and is not considered secure. You should never run untrusted programs using this policy group.

Usage: reserved

document_files

Description: Can read and write to document files. This policy group is reserved for certain applications, such as document viewers. Developers should typically use the content_exchange policy group and API to access document files instead.

Usage: reserved

This policy allows apps to read and write to the "Documents" folders in the user's home directory and external media.

document_files_read

Description: Can read all document files. This policy group is reserved for certain applications, such as document viewers. Developers should typically use the content_exchange policy group and API to access document files instead.

Usage: reserved

This policy allows apps to read the "Documents" folders in the user's home directory and external media.

history

Description: Can access the history-service. This policy group is reserved for vetted applications only in this version of the policy. A future version of the policy may move this out of reserved status.

Usage: reserved

keep-display-on

Description: Can request keeping the screen on

Usage: common

location

Description: Can access Location

Usage: common

Allows an app to request access to the device's current location.

microphone

Description: Can access the microphone

Usage: common

music_files

Description: Can read and write to music files. This policy group is reserved for certain applications, such as music players. Developers should typically use the content_exchange policy group and API to access music files instead.

Usage: reserved

The music_files policy group allows an app to read or write to the Music directories in the user's home folder or on external media.

music_files_read

Description: Can read all music files. This policy group is reserved for certain applications, such as music players. Developers should typically use the content_exchange policy group and API to access music files instead.

Usage: reserved

The music_files_read policy group allows an app to read the Music directories in the user's home folder or on external media.

networking

Description: Can access the network

Usage: common

The networking policy group allows an app to contact network devices and use the download manager.

nfc

Description: Can access the NFC functionality

Usage: common

The nfc policy group allows an app to read and write NFC tags via NDEF data as well as establishing a peer-to-peer connection between two devices.

picture_files

Description: Can read and write to picture files. This policy group is reserved for certain applications, such as gallery applications. Developers should typically use the content_exchange policy group and API to access picture files instead.

Usage: reserved

The picture_files policy group allows an app to read and write to the Pictures directories in the user's home folder or on external media.

picture_files_read

Description: Can read all picture files. This policy group is reserved for certain applications, such as gallery applications. Developers should typically use the content_exchange policy group and API to access picture files instead.

Usage: reserved

The picture_files_read policy group allows an app to read the Pictures directories in the user's home folder or on external media.

push-notification-client

Description: Can use push notifications as a client

Usage: common

sensors

Description: Can access the sensors

Usage: common

Allows apps to access device sensors

usermetrics

Description: Can use UserMetrics to update the InfoGraphic

Usage: common

Allows an app to write metrics to the UserMetrics service so they can be displayed on the InfoGraphic.

video

Description: Can play video (allows playing remote content via media-hub)

Usage: common

video_files

Description: Can read and write to video files. This policy group is reserved for certain applications, such as gallery applications. Developers should typically use the content_exchange policy group and API to access video files instead.

Usage: reserved

The video_files policy group allows an app to read and write to the Videos directories in the user's home folder or on external media.

video_files_read

Description: Can read all video files. This policy group is reserved for certain applications, such as gallery applications. Developers should typically use the content_exchange policy group and API to access video files instead.

Usage: reserved

The video_files_read policy group allows an app to read the Videos directories in the user's home folder or on external media.

webview

Description: Can use the UbuntuWebview

Usage: common

The webview policy group allows apps to embed a web browser view.

Thirdly, it is possible to develop cross-platform apps that don't depend on Ubuntu specific APIs, although this is not the official way of developing apps for Ubuntu Touch. If running on other operating systems is a requirement for your app, you can refer to these APIs instead:

- Kirigami API
- Qt Quick Controls 2 API

This is a helpful and welcoming community and everything works better in teamwork! Looking for other developers that might want to collaborate with you on your app? Have questions that aren't answered in the docs or want to chat with other Ubuntu Touch developers? Join our UBports UT App Dev Telegram group or chat with us on the UBports App Development forum! You can also check out our list of *preinstalled apps*. Contributions to those are greatly appreciated and directly reach a larger audience.

7.4 Maintainers wanted

All apps love contributions. But some core apps are lacking maintainers. Please check the list of core apps and see if you want to step up as new maintainer for one of our core apps. The maintainer reviews MR's, triages and investigates issue reports and helps to develop the app.

The best way to start is by making some contributions and let the reviewers know that you are interested. Or join our matrix group at #ubcd:matrix.org and introduce yourself.

CHAPTER

EIGHT

HUMAN INTERFACE GUIDELINES

Ubuntu Touch follows a clean, beautiful, modern and (above all) convergent design. We aim to keep developing the design started by Canonical.

8.1 Design concepts

As designed by Canonical, we use the following system palette (from Michał Pręd otka and Ubuntu documentation):

8.1.1 Operating system and app palette

- #FFFFFF White

Recommended for background on Ambiance theme (light theme) or regular text on Suru Dark theme.

- #111111 Jet

Recommended for regular text on Ambiance theme or background in Suru Dark.

- #3B3B3B Inkstone

Recommended for foreground colours in dark themes.

- #5D5D5D Slate

Recommended for text and action icons.

- #666666 Graphite

Recommended for coloring dark themes' background.

- #888888 Ash

Recommended for subtitles and other tertiary content.

- #CDCDCD Silk

Recommended for neutral action buttons and secondary text.

- #F7F7F7 Porcelain

Recommended for foregrounds.

- #335280 Blue

Recommended for progress bars, selected areas, text selection and text cursor. Also neutral actions (Ambiance).

- #19B6EE Light Blue

Recommended for progress bars, selected areas, text selection and text cursor. Also neutral actions (SuruDark).

- #0E8420 Green

Recommended for positive action buttons (Ambiance).

- #3EB34F Light Green

Recommended for positive action buttons (SuruDark).

- #C7162B Red

Recommended for negative and irreversible action buttons, errors and alerts (Ambiance).

- #ED3146 Light Red

Recommended for negative and irreversible action buttons, errors and alerts (SuruDark).

- #E95420 Orange

Orange is recommended for branded elements, focus and intensity.

8.1.2 Units

Resolution Independence Approach

The objective of resolution independence is to make it easy for graphical user interfaces in Ubuntu Touch to scale to all the form factors that Ubuntu Touch targets: phones, tablets, laptops and desktops. The approach taken combines simplicity for the designers and developers with visual fidelity, quality and usability.

Measurement Units

Grid Unit

A measurement unit called the *grid unit* is defined, abbreviated *gu*. 1 grid unit translates to a given number of pixels depending on the type of screen that the user interface is displayed on. For example, on a laptop computer 1 grid unit will typically translate to 8 pixels. The number of pixels per grid unit is chosen in order to preserve the perceived visual size of UI elements and therefore depends on the density of the display and the distance the user is to the screen. We also ensure that 1 grid unit is always an integer number of pixel.

Device	Conversion
Most laptops	1 gu = 8 px
High DPI laptops	1 gu = 16 px
Phone with 4 inch screen at HD resolution (around 720x1,280 pixels)	1 gu = 18 px
Tablet with 10 inch screen at HD resolution (around 720x1,280 pixels)	1 gu = 10 px

Density Unit

Exceptionally, in order to accommodate for the rare cases where measurements of less than 1 gu are needed another unit is available: the *density independent pixel,* abbreviated dp. 1 dp typically translates to 1 pixel on laptops and low density mobile phones and tablets.

Use

```
Item {
  width:  units.gu(2)
  height: units.dp(1)
}
```

Device Sizes

Device Grid Units	width	height
E5 (portrait)	42	72
M10 HD (landscape)	160	97
M10 FHD (landscape)	148	89
N5 (portrait)	47	80

8.1.3 Typography

Fonts

Our default font is Ubuntu [2] installed in the system.

For the font sizes to be consistent across devices, instead of setting the font size as a number of pixels or points, it is imperative to define the font size in terms of size names:

Font Size Name	gu	Desktop	Phone	Tablet
			4” HD Screen	10” HD screen
x-small	1.25	10 px	22 px	12 px
small	1.5	12 px	27 px	15 px
medium	1.75	14 px	31 px	18 px
large	2.5	20 px	45 px	25 px
x-large	3.5	34 px	76 px	42 px

8.1.4 App icon design

The whole system and apps follow the Suru design principles started by Canonical and Samuel Hewitt.

Design Philosophy and Suru Meaning

- Suru alludes to Japanese culture.
- The design of Suru is inspired by origami.
- Paper can be used in two and three dimensional forms, because it is transferable and diverse.

Suru brings a precise yet organic structure to the Ubuntu Touch interface. The sharp lines and varying levels of transparency evoke the edges and texture of paper.

Design Tips for an App Icon

Descriptive design

The icon you choose for your app should describe what your app is about, what it does, or what service it is there for in a recognizable way.

Distinctive shapes

Depending on the object you choose to show in the icon, on the perspective, lighting and other factors, it might be hard to recognize what you are trying to depict. A TV might look like a box, or a computer screen, a remote control like a phone, etc.

Details

A detailed icon is very nice to look at as it shows quality and your eye for the small things. It might add confusion though, especially in smaller sizes or for people who have impaired eyesight. Show what's necessary, pay attention to detail, but don't overload the icon.

Colours

Use distinctive colours, make your icon stand out a bit and give the shapes enough contrast to be visible.

Gradients

If you were to add a gradient to your icon, only use colours that are slightly different from each other so that the change is not too dramatic. For example:

Composition

Suru icons are composed of simple geometric shapes. The background is usually a coloured surface with the pictogram, also composed of flat shapes, "floating" above it.

The Fold Effect

In keeping with the origami motif, some Suru icons have an implied fold. In many icons this is a single horizontal or vertical line but sometimes the fold line(s) correspond or align with elements of the pictogram.

The fold is drawn by creating an overlay of a white or black polygon of very low opacity (usually between 1%~10%).

Grids

Using a grid layout ensures consistency across all icons and will force you to reserve area for the icon background for even padding around the pictogram.

Note that the circular elements are slightly larger than rectangular elements of the grid, this overshoot is needed to compensate for the optical illusion where circular objects appear smaller than rectangular objects of the same dimensions.

Unit Grid

Designing with an overall pixel grid in mind is crucial to having crisp, clean icons. Since most desktop icons have dimensions that are a factor of four (16x16, 24x24, 32x32, 48x48, 256x256) using a pixel grid with lines every 4 pixels and drawing to that grid is the best practice.

Shadows

Often a Suru icon is drawn with two distinct sets of shadows, one for the pictogram to create a drop shadow effect, and a second below the overall icon shape.

The drop shadow effect on the central pictogram is a combination of two shadow that are identical in shape to the pictogram:

- a shape that is rgba(0,0,0,0.1) with vertical offset of 2 pixels and a blur of 1%
- a shape that is rgba(0,0,0,0.4) with vertical offset of 4 pixels and a blur of 10%

If the fold effect is present, the second shadow is drawn using a linear gradient with three stops whose positions correspond to the location of the background fold.

Highlights

Pictograms have an ever~so~slight (1 pixel) white highlight along the top edge of them. To do this, make 2 solid white copies of the main pictogram shape and move one copy 1 pixel down and 'difference' the two so what's left is an 'edge' that you can align to the top of the pictogram. Repeat as needed.

Depending on the colour of your pictogram element, you can vary the opacity of your highlight. For instance, if it is using bright, primary colours, there isn't any need for a highlight.

Grids and other resources can be found in this gitlab repository.

8.1.5 In-app icons design

Interface Icons

Interface icons communicate either status or action displays, such as battery level or an alarm, whilst also enabling the user to perform actions like opening a menu, sharing content or print a document. The interface icons are located in the top bar or in-app icons, such as in the Header or the menus.

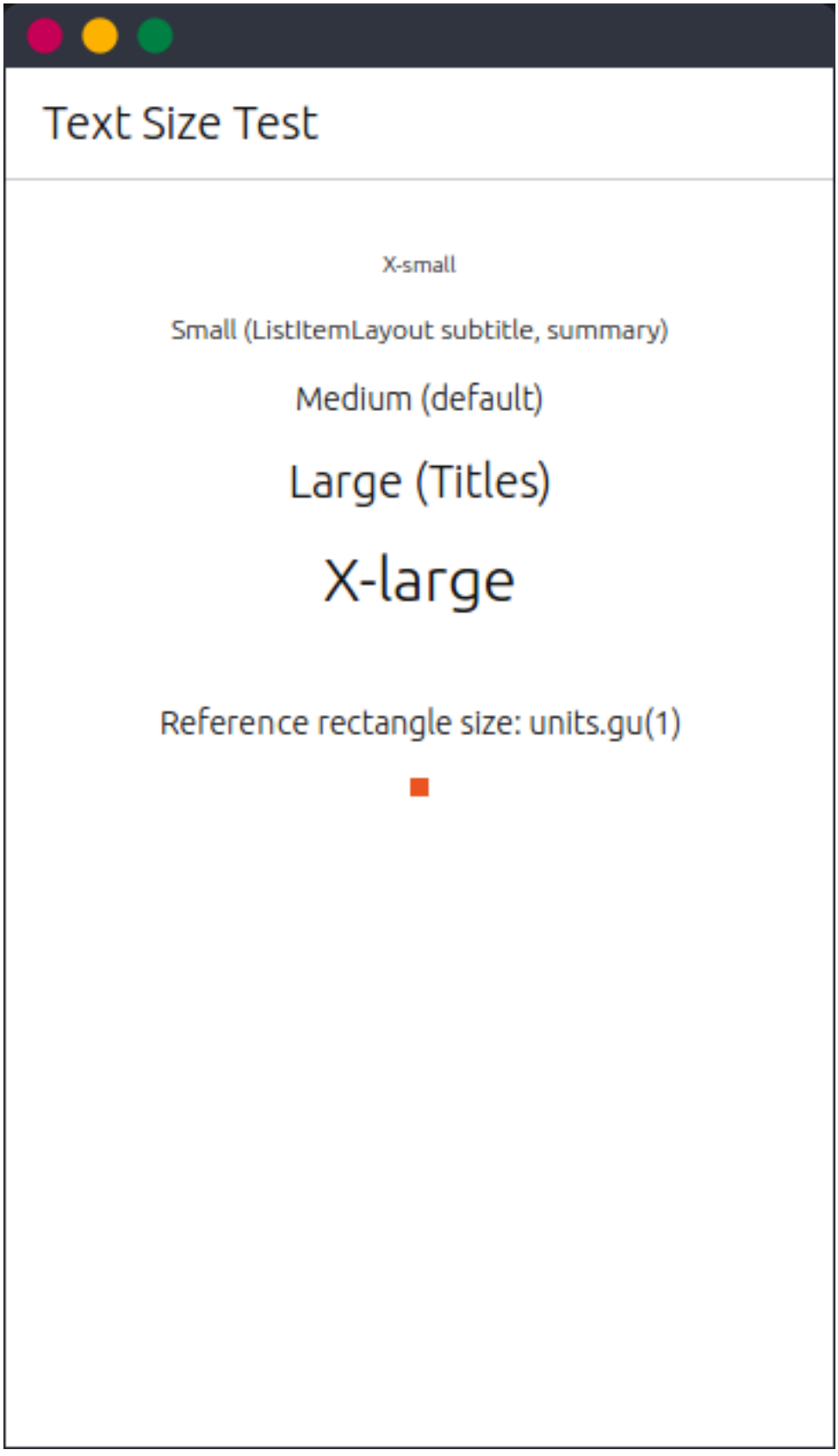
See a list of the icons. available in the system.

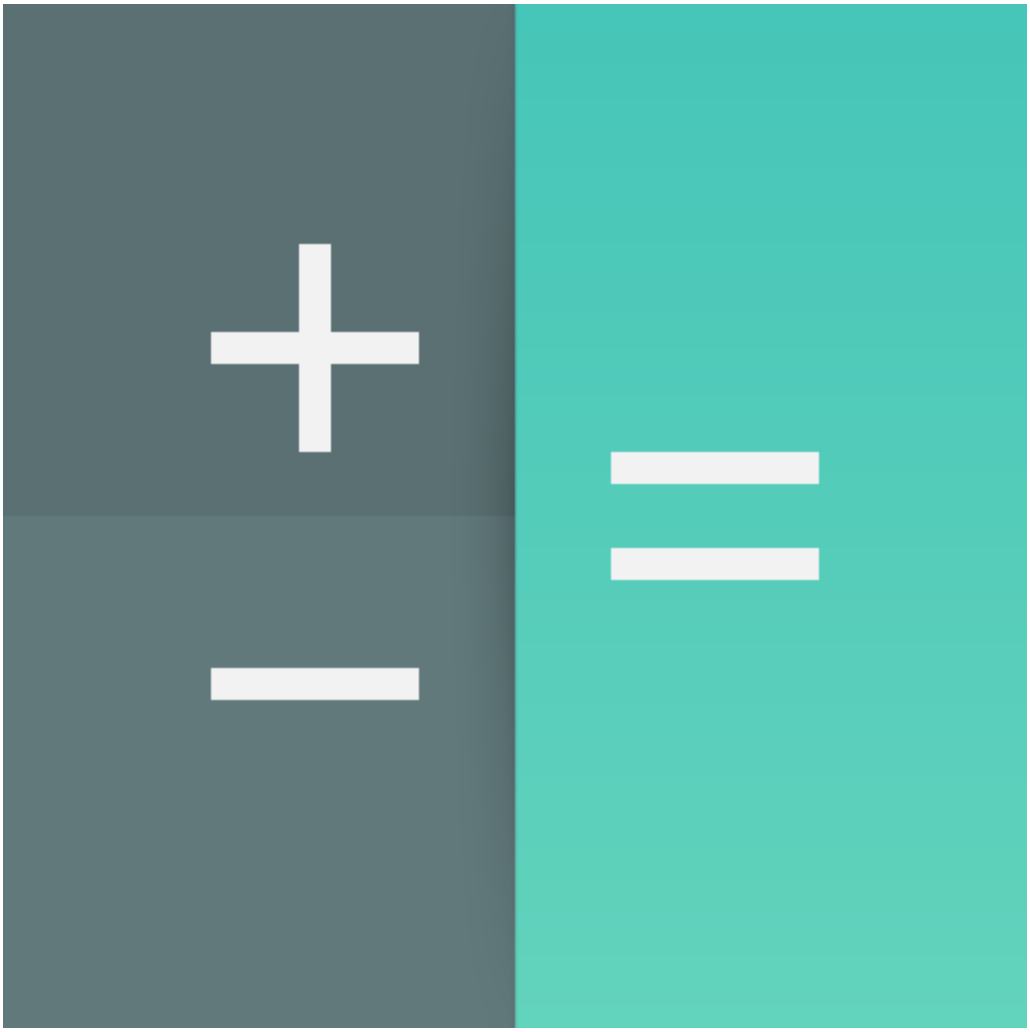
Color

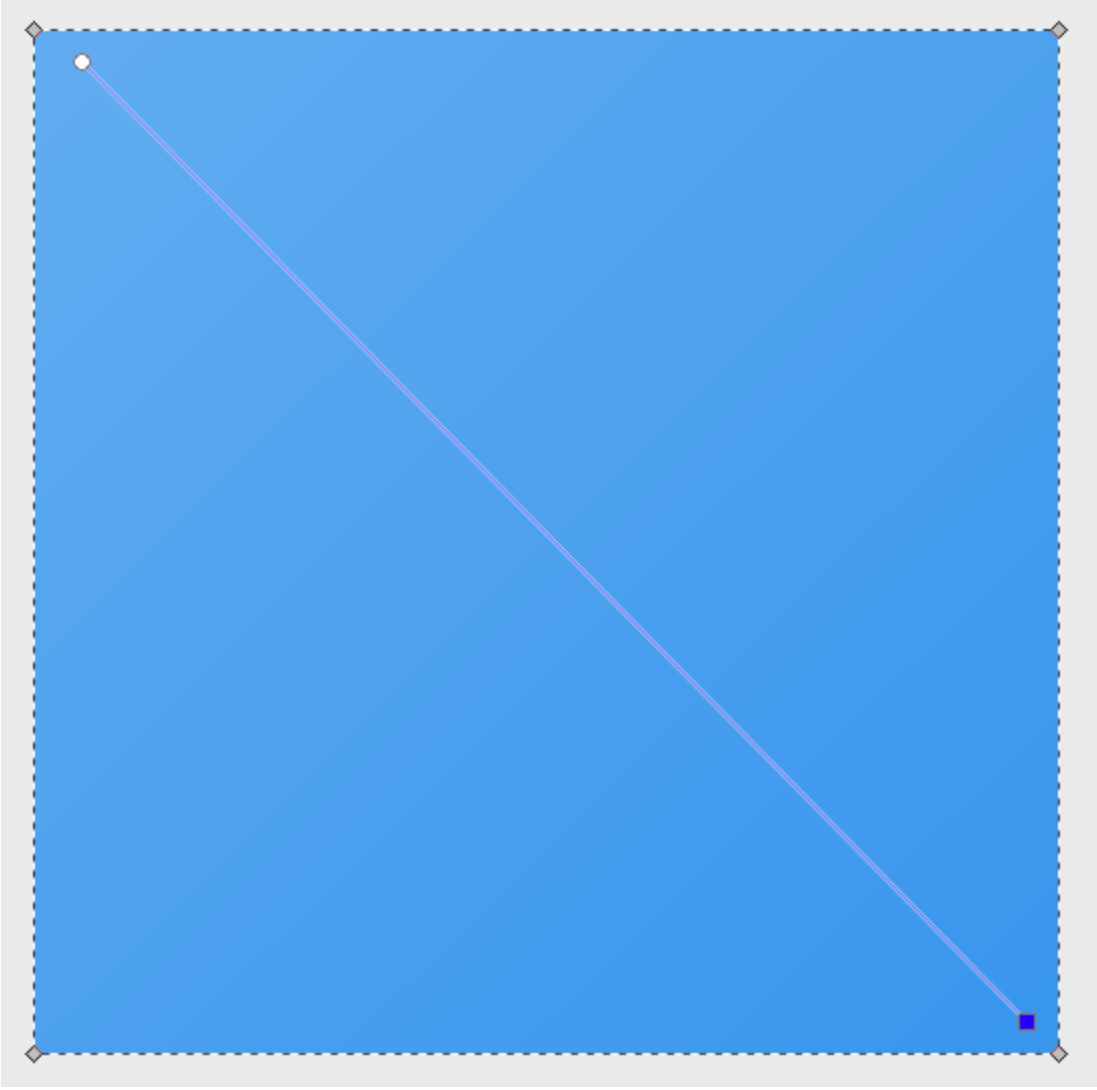
Interface icons can be colourized in qml therefore are designed in #808080 color.

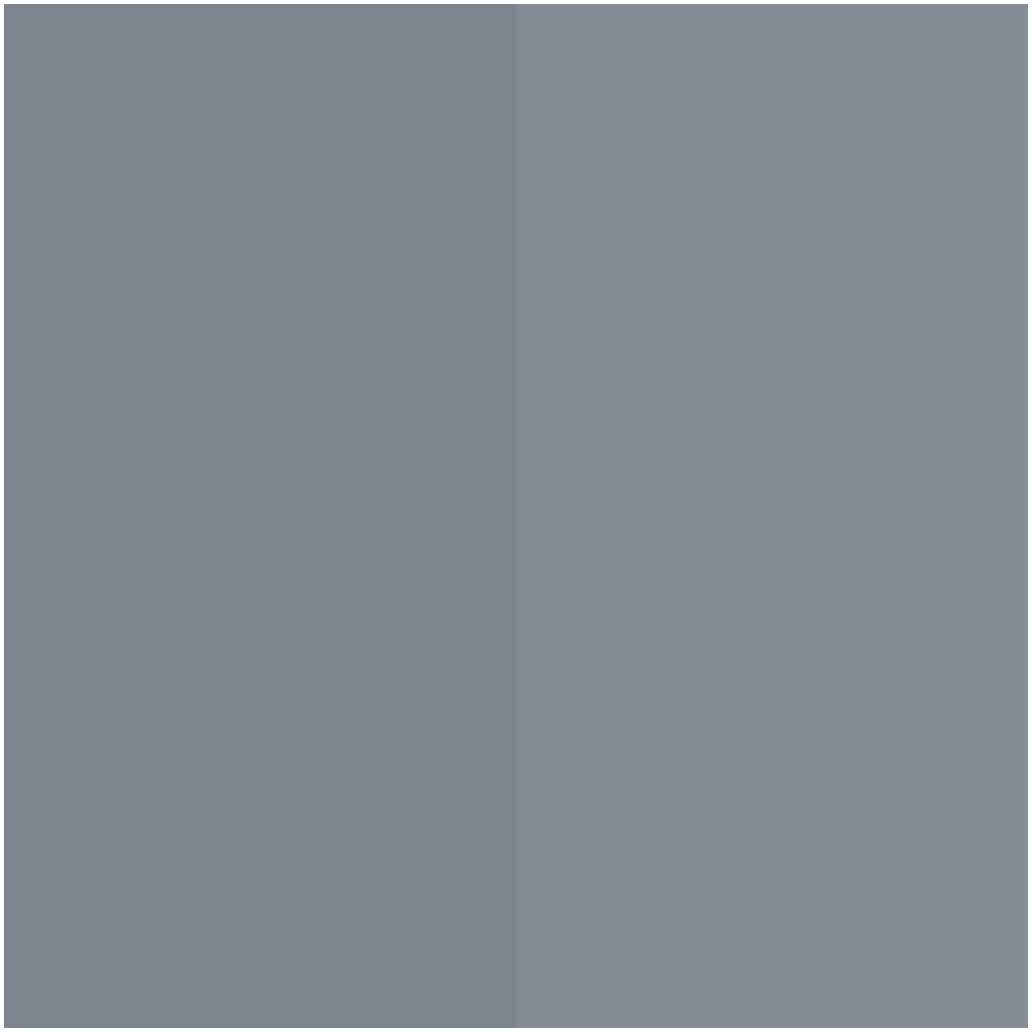
Font

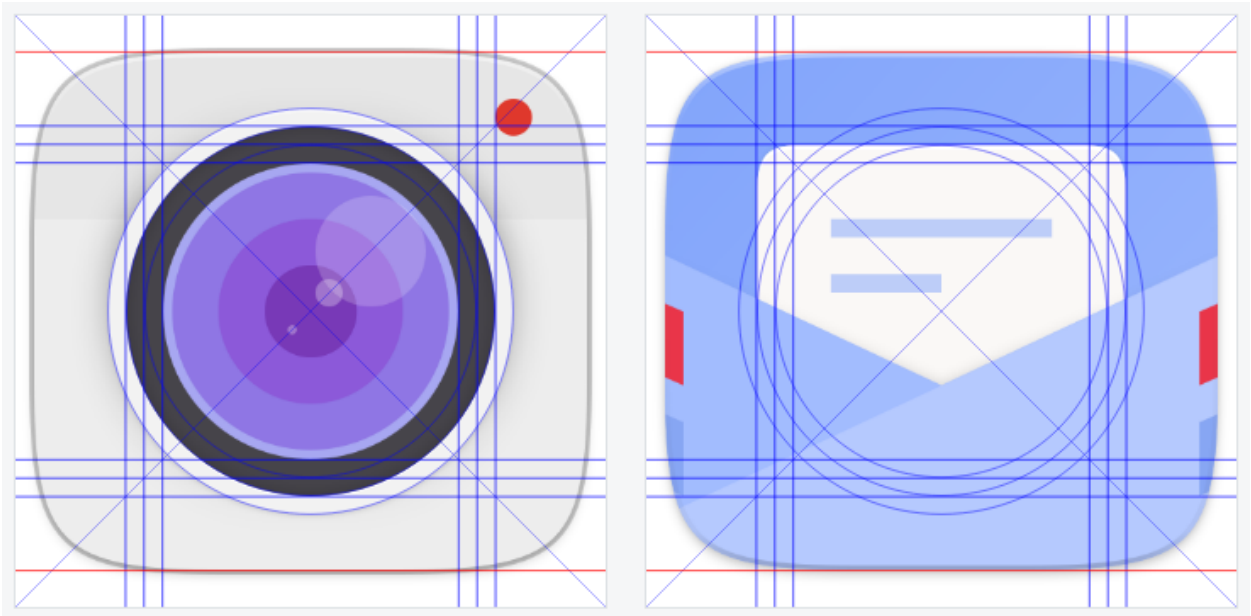
To provide a consistent look to the UI and to make the Suru icons distinctive, the design is based on the same patterns as the Ubuntu font. The font patterns can applied to the symbol to define its contours.

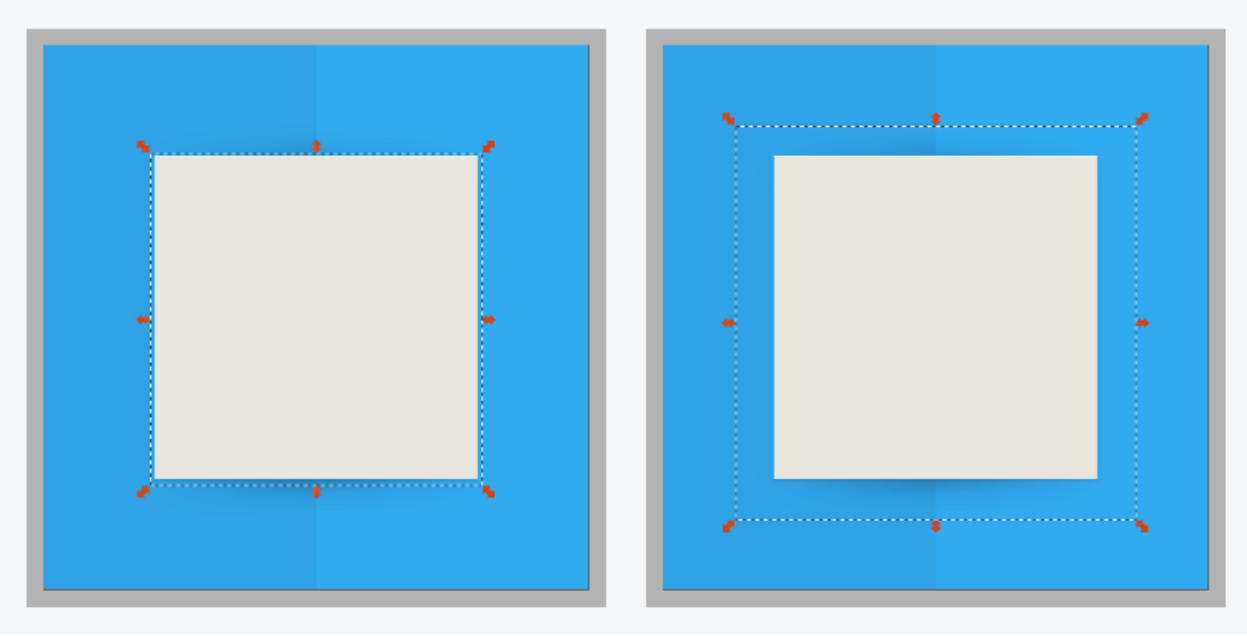

Text Size Test
X-small
Small (ListItemLayout subtitle, summary)
Medium (default)
Large (Titles)
X-large
Reference rectangle size: units.gu(1)











Strokes

Stroke ends are squared but the upper end of vertical strokes is slightly oblique with a +10º angle.

Rotation

When they are not horizontal or vertical, objects are oriented along a +45º axis.

Corners

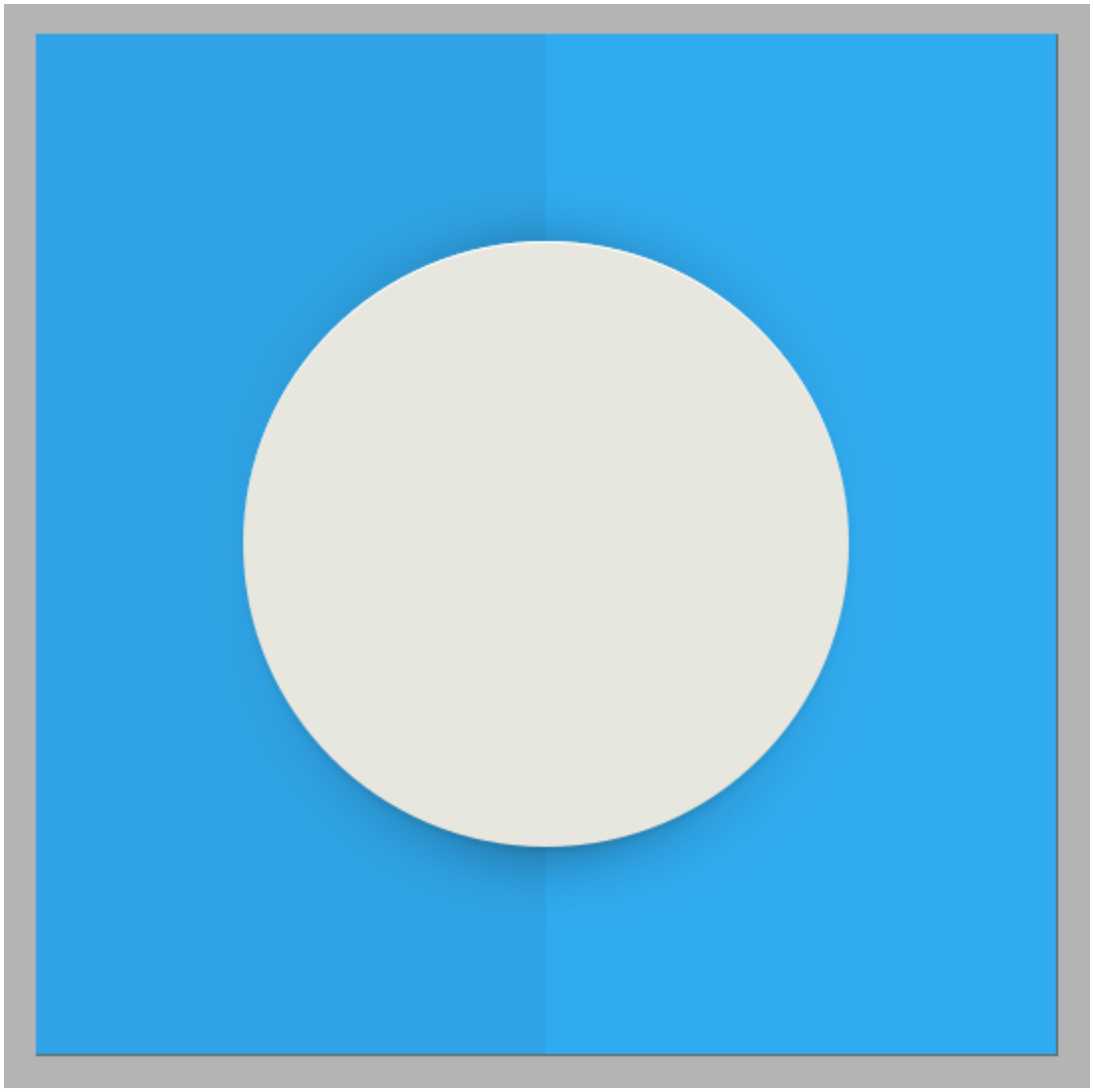
Curved corners are preferred if possible. The following picture shows a zoom on the default outer and inner curvatures, with nodes and Bezier handle. The default curvature can be scaled down when the shape is smaller.

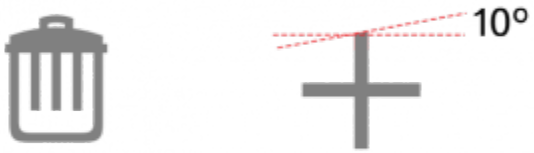
Arrows

A chevron symbol is used for navigation arrows. The arrowhead is an isosceles triangle where the two equal sides are slightly curved (following the pattern of the Y character).

Opacity

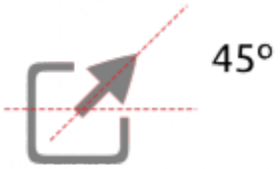
50% opacity is used to reflect the status of an indicator. In case of a double space (enabled and disabled, for example), the opacity of the whole icon changes. When the status allows a range of values, the opacity of a single portion of icon can be altered.

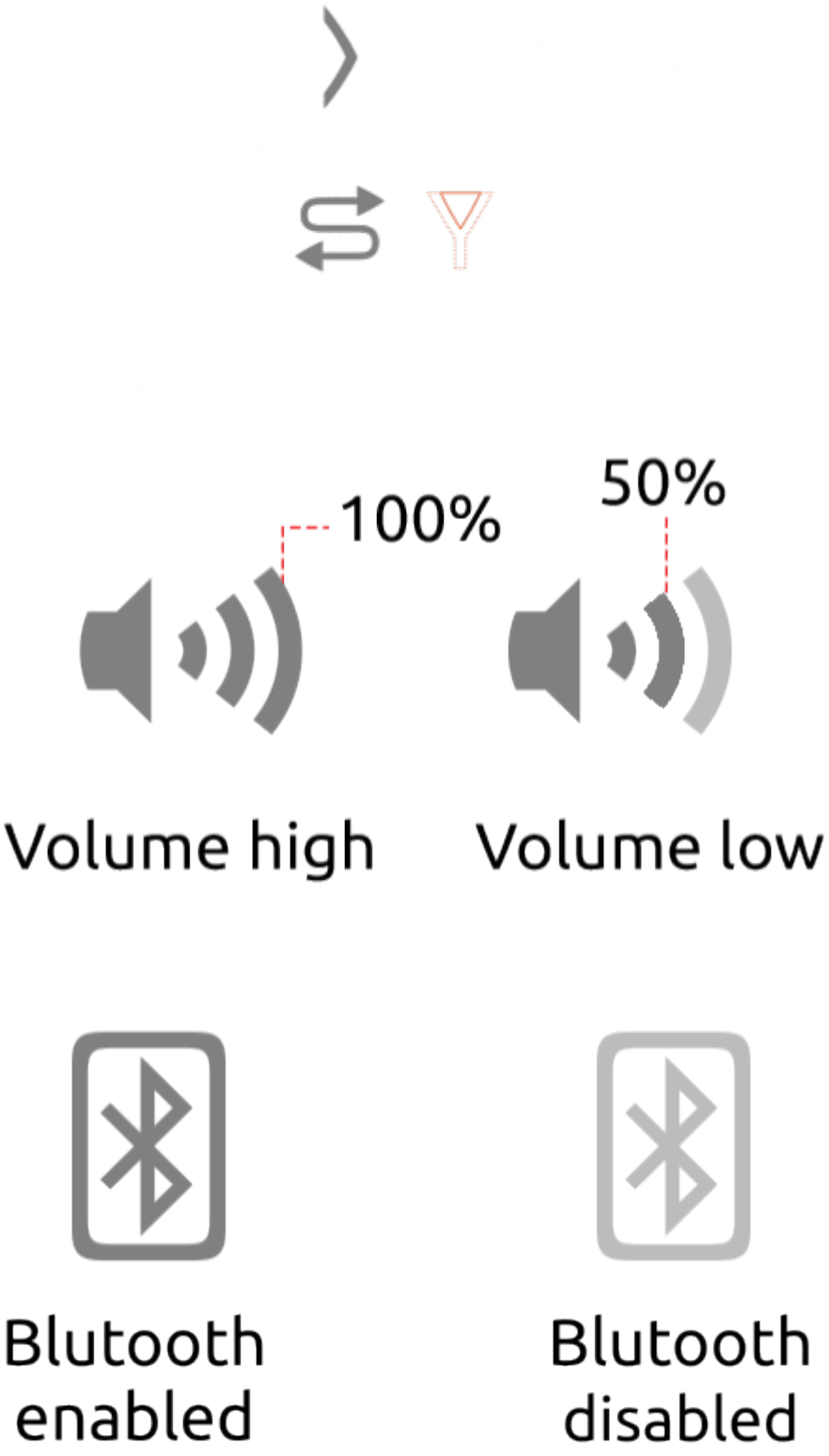




10°

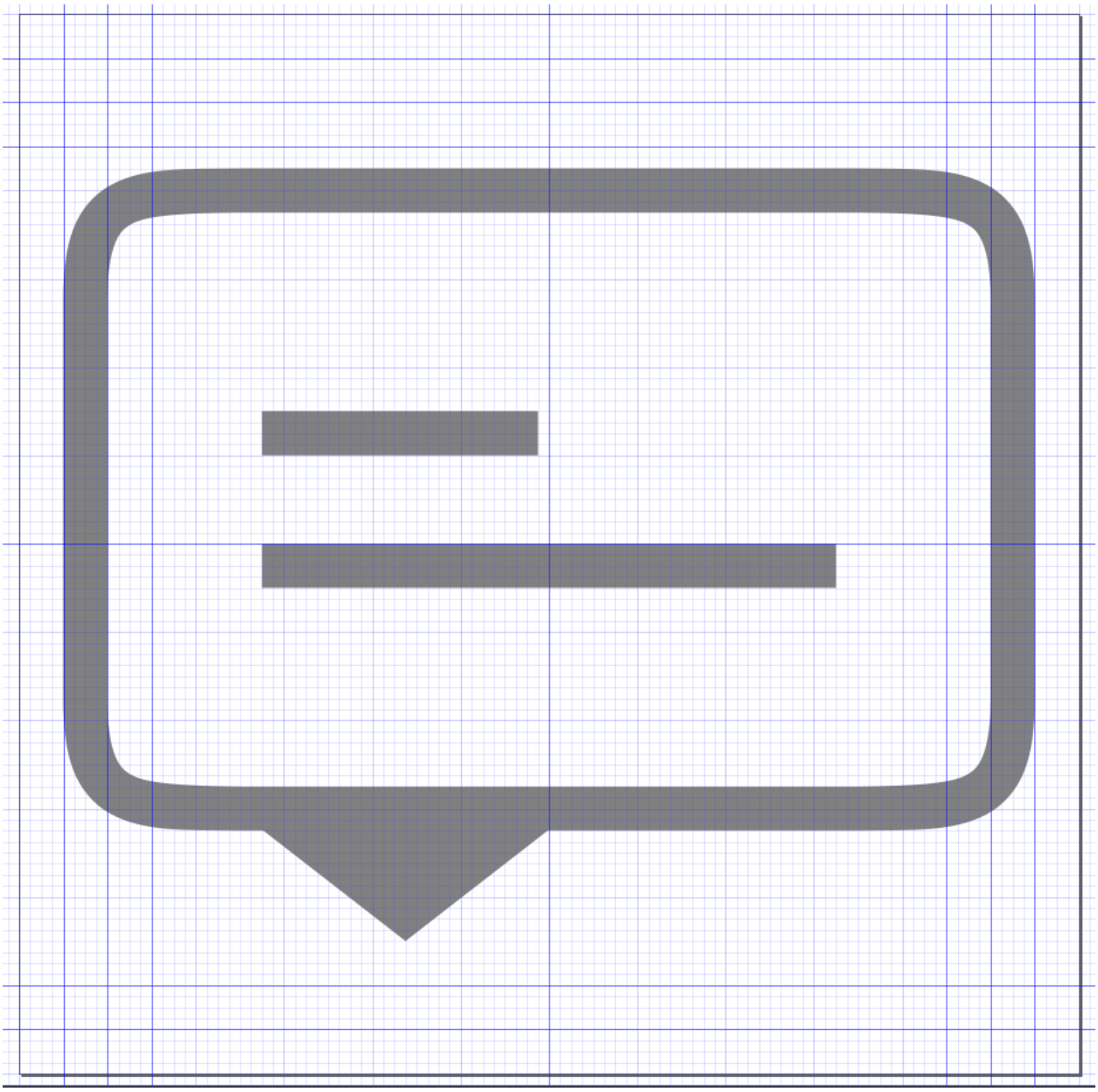

45°


45°


100%
50%
Volume high
Volume low
Blutooth enabled
Blutooth disabled

Grid

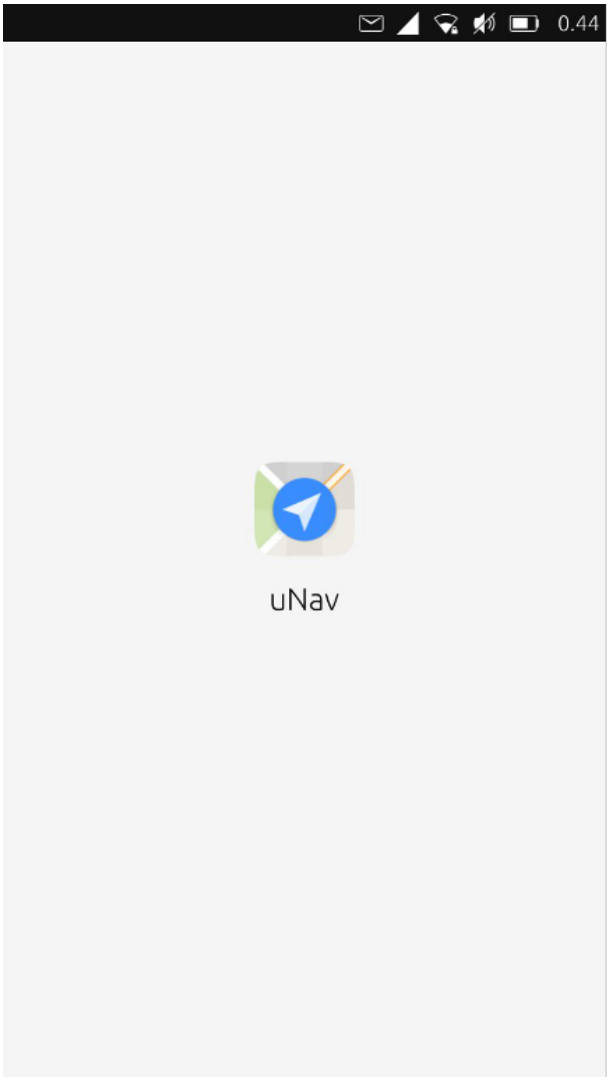
Interface icons are designed for 96x96 pixels, with at least 4 pixels of margin on every side.

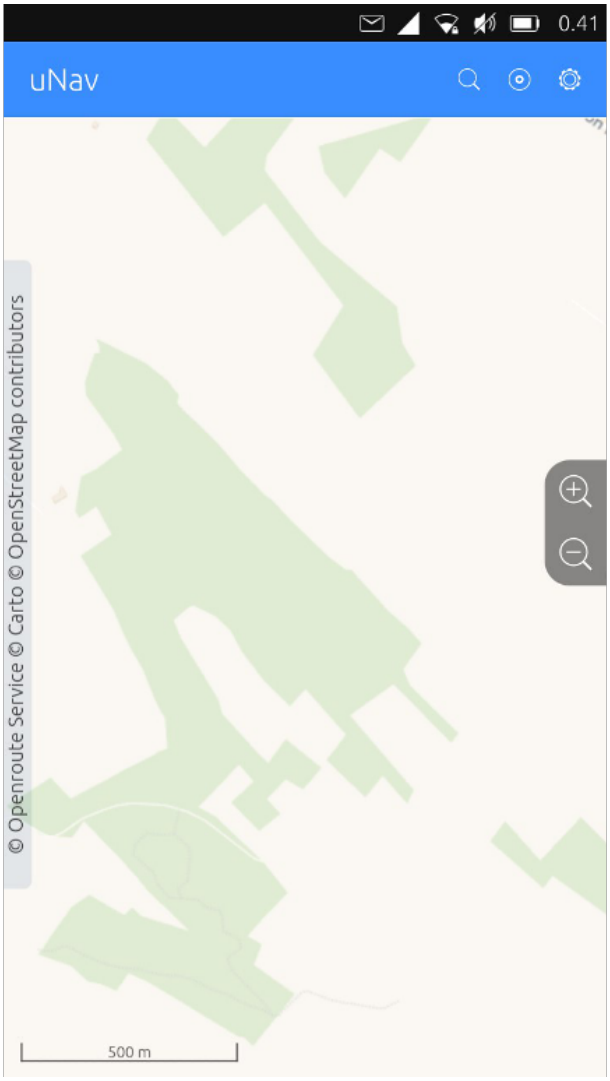


8.2 App layout

8.2.1 App main view

When an app starts, shows a generic splash screen (as in the image) or a custom one, then the main view.





The standard parts of a main view or any other page may be:

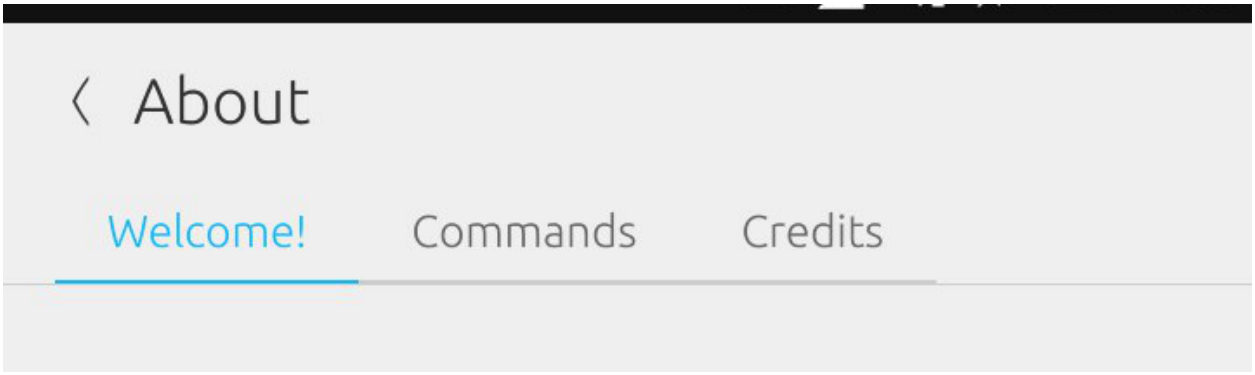
Header

Display app name, leading area and trailing area icons.

In this case, showing title and trailing actions.

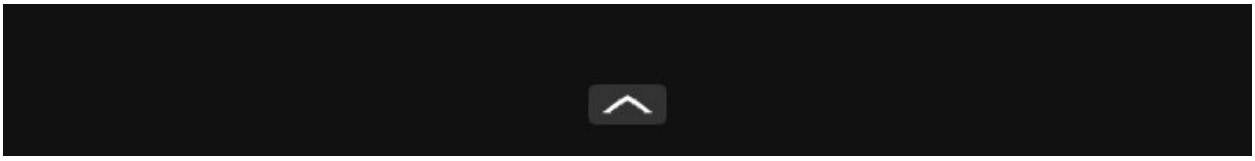
Tabbed content

To organize information under tabs.



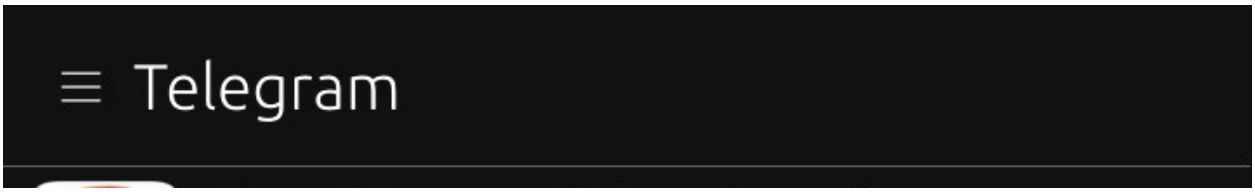
Bottom edge menu

For quick actions. See Quick actions.



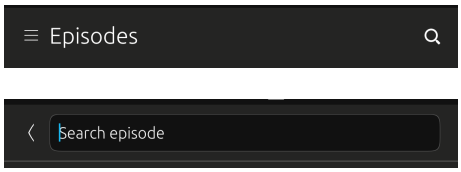
8.2.2 Header uses

App headers can be used to show app name, app section or page, search box, or other globally accessible functions.



Some special apps doesn't show any header, like the Camera app or some games.

The header takes care of showing the back button when a page stack is used.



8.2.3 Main actions

Main actions are displayed in the corresponding header of the Page. The header should show only relevant actions to the current page.

Main page of Contacts app shows:

When viewing a single contact, only relevant actions are shown:

Slots

The header contains a number of slots that can hold actions or navigational options. Depending on the surface or window size, additional slots can be added to show the actions otherwise hidden in drawers.

Action Drawer

Actions will be placed into an action drawer when there are no available slots to house them. However, when your app is on a larger surface, like on a desktop, then actions will appear in the slots.

Responsive layout

As the header gains width across screen sizes, additional slots become visible and actions in the drawer will appear automatically.

3 slot layout

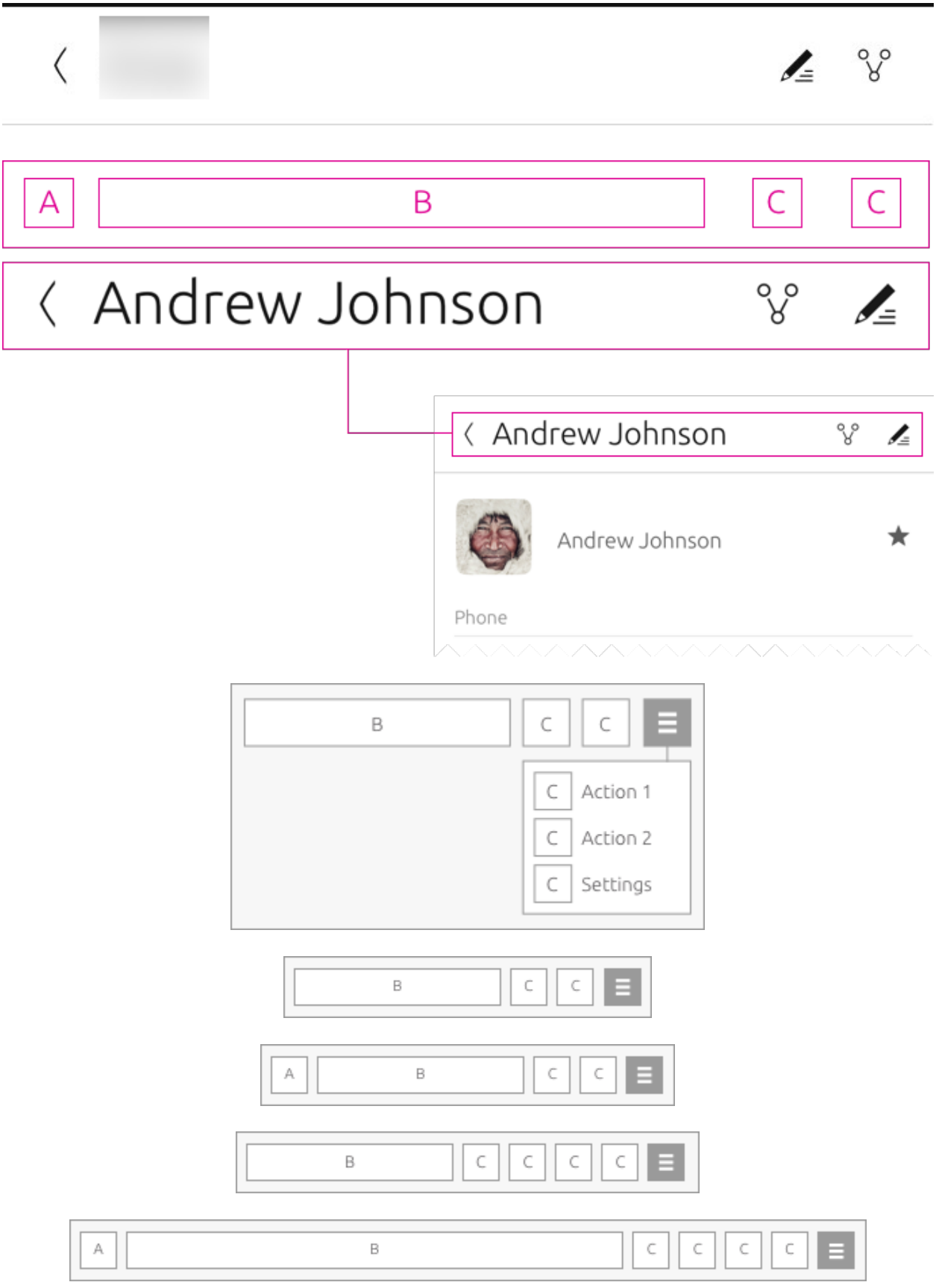
4 slot layout

5 slot layout

6 slot layout

Medium to large screens

The maximum number of visible action slots in a convergent environment is 6. If this is exceeded then additional actions will migrate to the action drawer.


A
B
C
C
Andrew Johnson
Andrew Johnson
Andrew Johnson
Phone
B
C
C
C
Action 1
C
Action 2
C
Settings
B
C
C
A
B
C
C
B
C
C
C
C
A
B
C
C
C
C

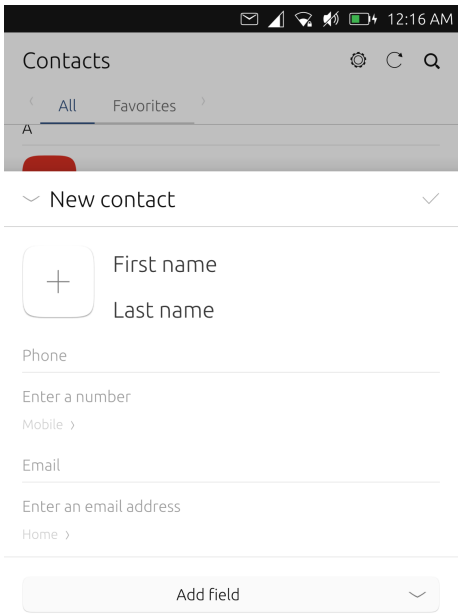
Simple

Keep in mind to keep a clean design. The simpler, the better.

8.2.4 Quick actions

The most common action for an app, the one that needs to be accessed quickly might be added as a bottom edge action, accessed with a swipe from outside of the screen upwards.

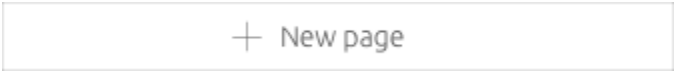
You can add a new contact in the Contacts app swiping up:



Hint

The hint is used to let the user know that there is something worth trying at the bottom of the screen.

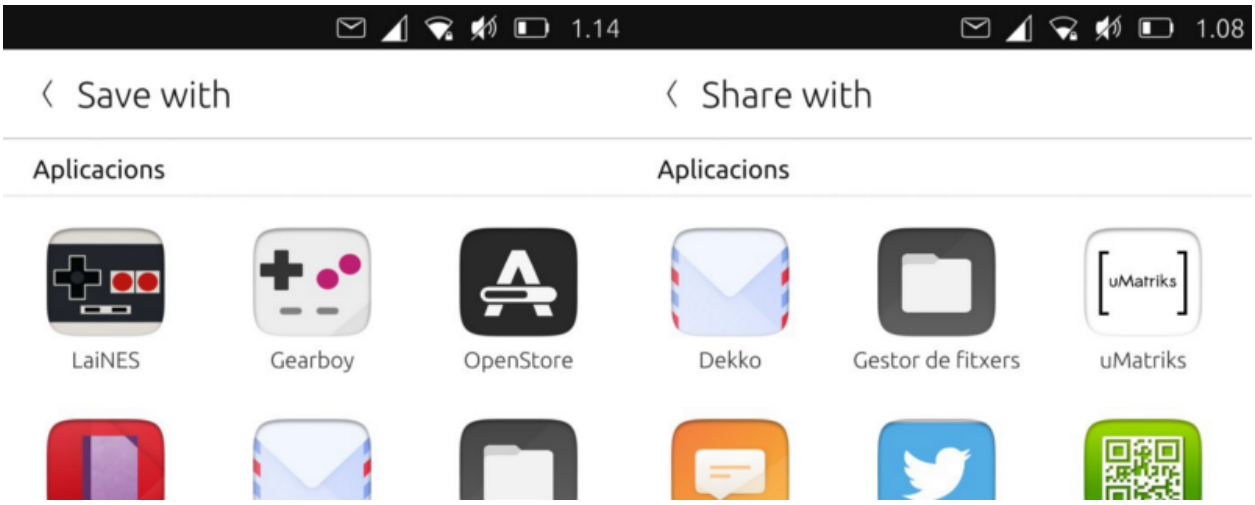
After opening an app, a hint with a text and/or an icon is shown:



After after a few seconds it hides and a floating icon is shown

8.2.5 Importing Data

Interchange of data between apps is done through Content Hub. This improves isolation between apps.



Apps with the appropriate apparmor permissions can import, export or share data between them.

For more information see Importing from Content Hub and URLdispatcher.

8.2.6 Presenting data

List Items

List items can be used to make up a list of ordered scrollable items that are related to each other.

When images or icons are presented without text or actions, it would make more sense to show them inside a grid rather than a list; like in a photo gallery.

Items in a list can have actions that can be placed in a context menu. The context menu can be accessed in two ways: by swiping or right-clicking the list item.

The actions are placed within two categories: leading for negative actions and trailing for positive actions.

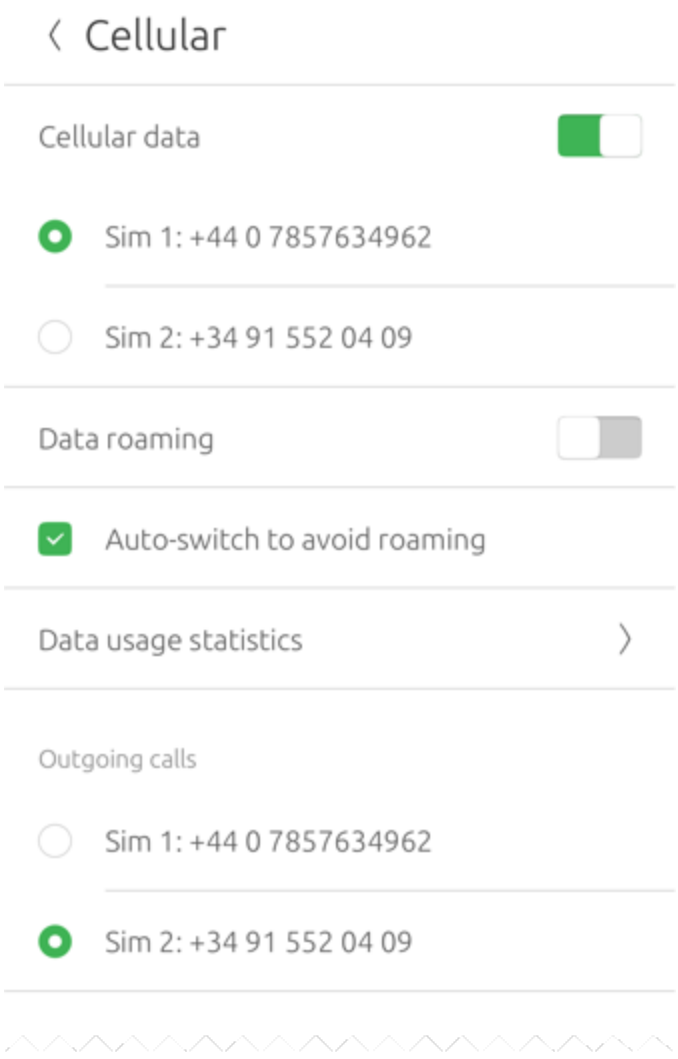
Touch swiping

Mouse right click

Keyboard navigation

Lists in edit mode

Edit mode allows users to modify a particular item or multiple items at once.


Cellular
Cellular data
Sim 1: +44 0 7857634962
Sim 2: +34 91 552 04 09
Data roaming
Auto-switch to avoid roaming
Data usage statistics
Outgoing calls
Sim 1: +44 0 7857634962
Sim 2: +34 91 552 04 09

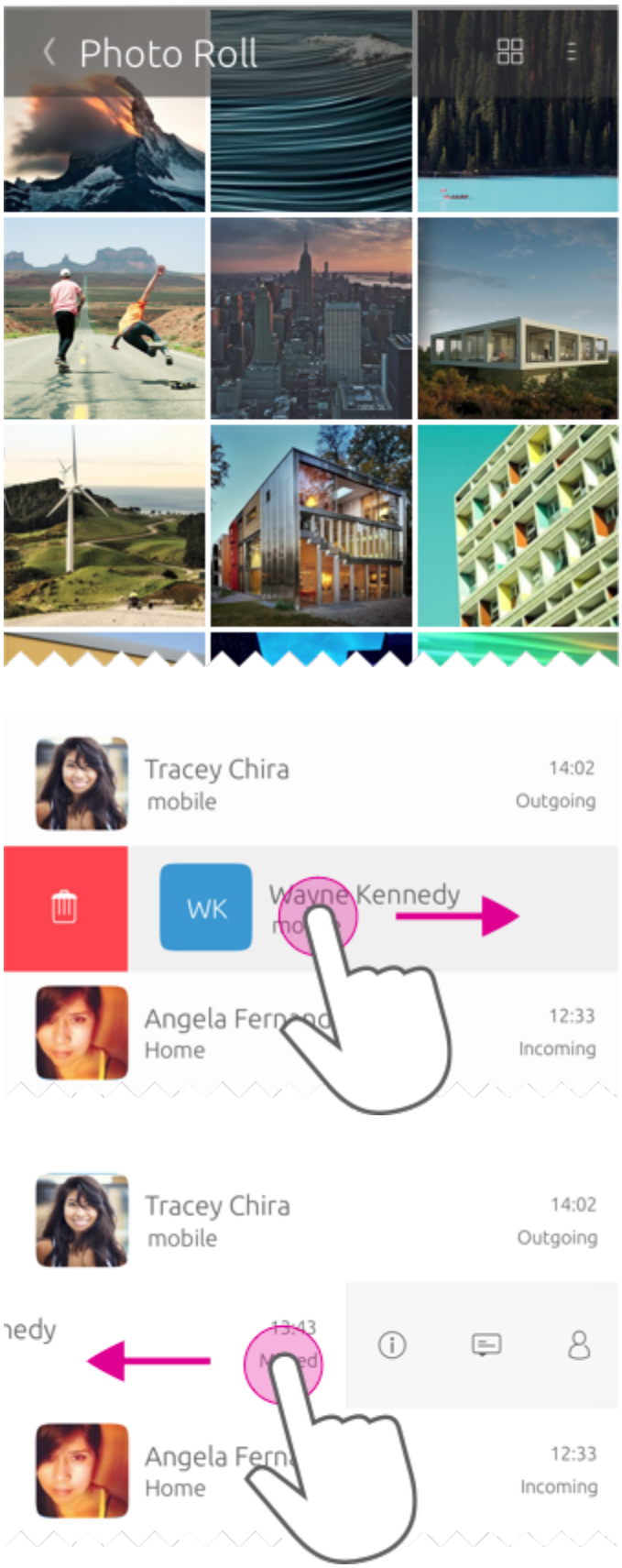
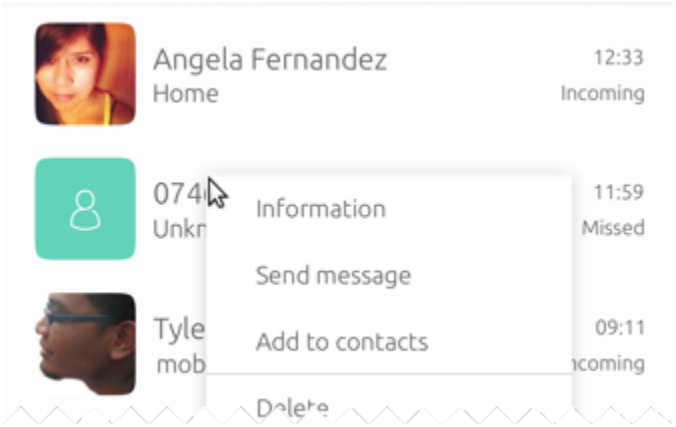
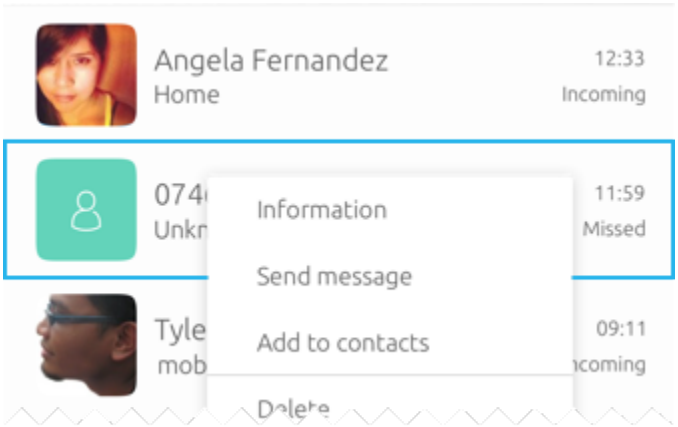
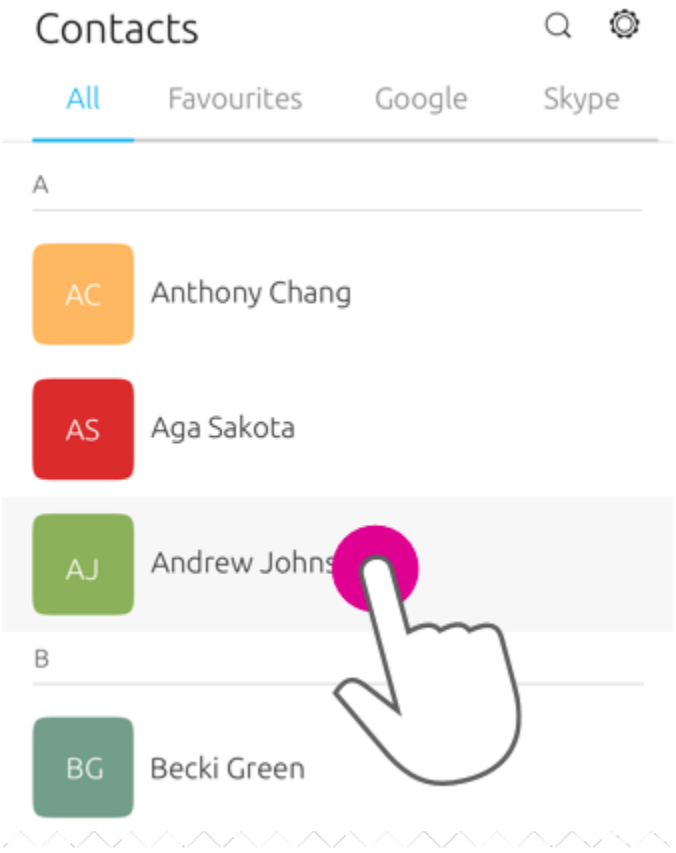


Photo Roll
Tracey Chira
mobile
14:02
Outgoing
WK
Wayne Kennedy
Angela Fernand
Home
12:33
Incoming
Tracey Chira
mobile
14:02
Outgoing
Angela Fern
Home
12:33
Incoming


Angela Fernandez
Home
12:33
Incoming
074
Unkn
Information
Send message
Add to contacts
Delete
11:59
Missed
Tyle
mob
09:11
coming


Angela Fernandez
Home
12:33
Incoming
074
Unkn
Information
Send message
Add to contacts
Delete
11:59
Missed
Tyle
mob
09:11
coming

You can use edit mode to allow users to multi-select, rearrange or delete items inside a list. When edit mode is entered the whole screen becomes an edit state and the header will show associated editing actions for the content. Alternatively, if the user long presses an item a context menu will show the associated editing actions too.



Layout of detailed list items

1. 1 line – Title
2. 1 line – Subtitle
3. 2 lines – Summary

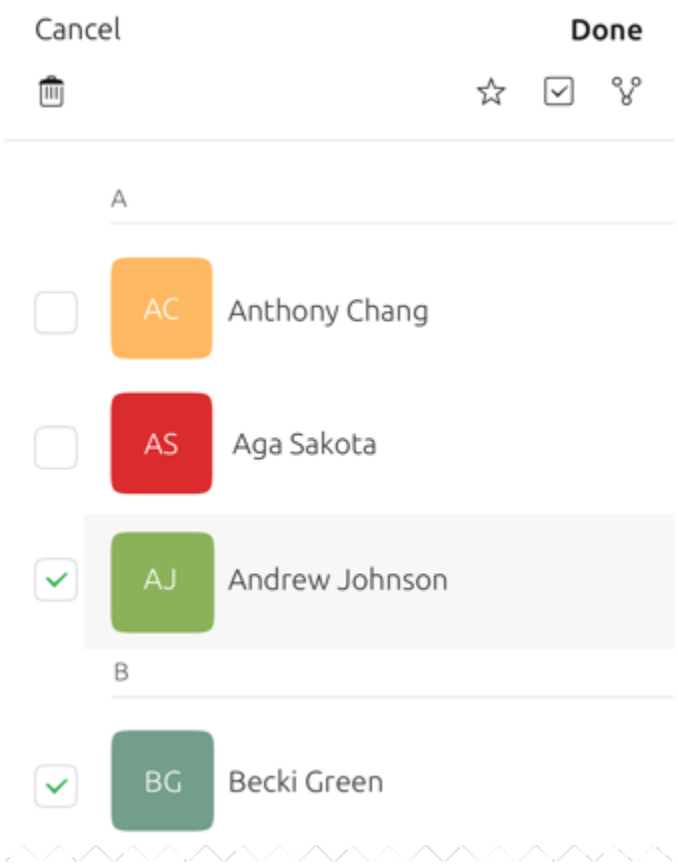
8.2.7 Dialogs

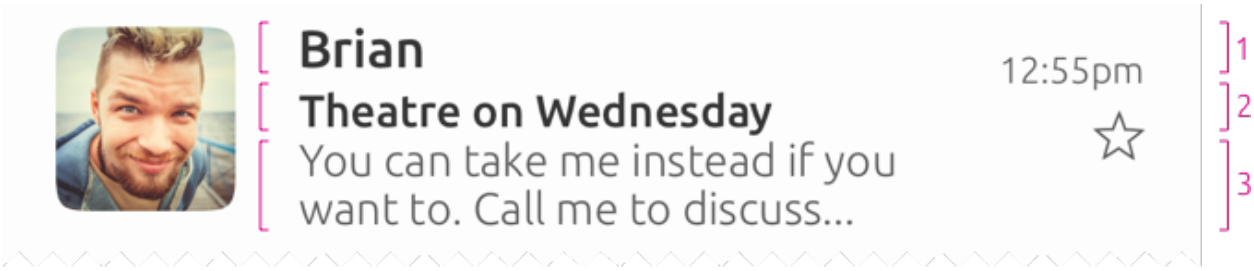
A dialog is a way of informing the user about important notices or letting them complete a single action.

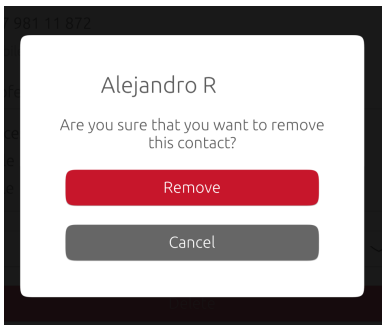
Dialogs should be unambiguous and concise in their wording. Only the most common, or main, action should be highlighted with color. The main action should be placed topmost in vertical dialogs or rightmost in horizontal dialogs.

Main actions may be positive, negative, or neutral. Positive actions add something to their subjects. Negative actions remove, delete, or otherwise make changes that can't be easily undone. Neutral actions do not add or remove from the subject, but still make a change.

The main action should be highlighted using `theme.palette.normal.negative` if it is destructive, `theme.palette.normal.positive` if it is positive, or `theme.palette.normal.focus` if it is neutral. The remaining options should be gray, without defining a color theme property.

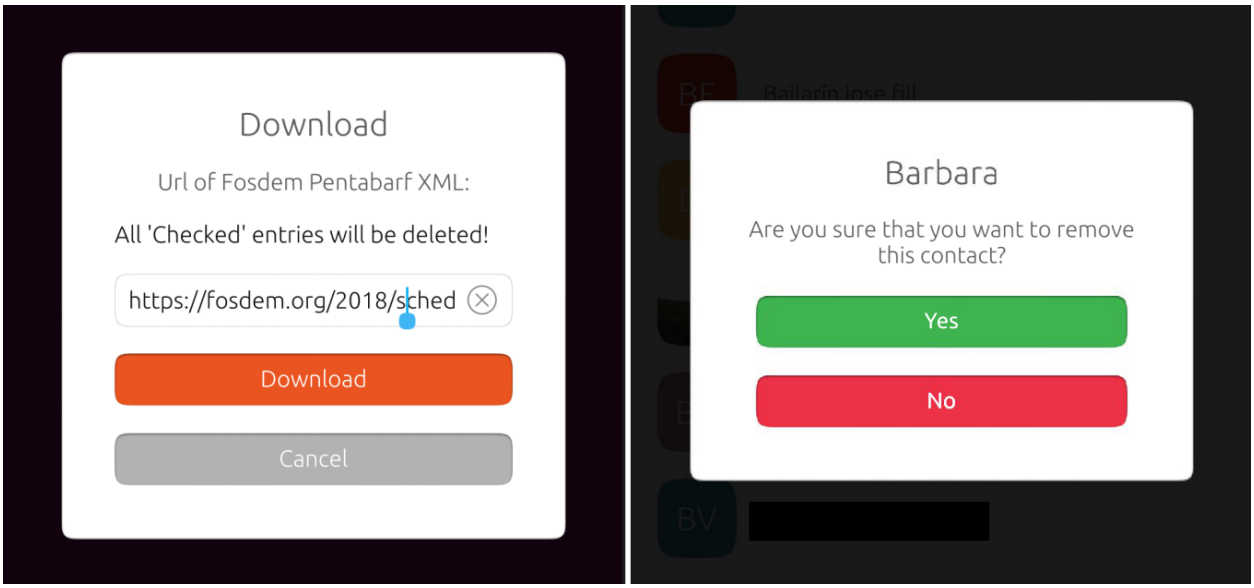

Cancel
Done
A
AC
Anthony Chang
AS
Aga Sakota
AJ
Andrew Johnson
B
BG
Becki Green


Brian
Theatre on Wednesday
You can take me instead if you want to. Call me to discuss...
12:55pm
1
2
3


Alejandro R
Are you sure that you want to remove this contact?
Remove
Cancel

Dialogs should resume actions, not ask a yes or no question. For example, the user has selected an option to delete a contact. Your dialog should ask, “Are you sure you would like to delete this contact?” The options should then be “Delete”, a negative action colored in red, and “Cancel”, colored in gray.

Examples of designs to avoid

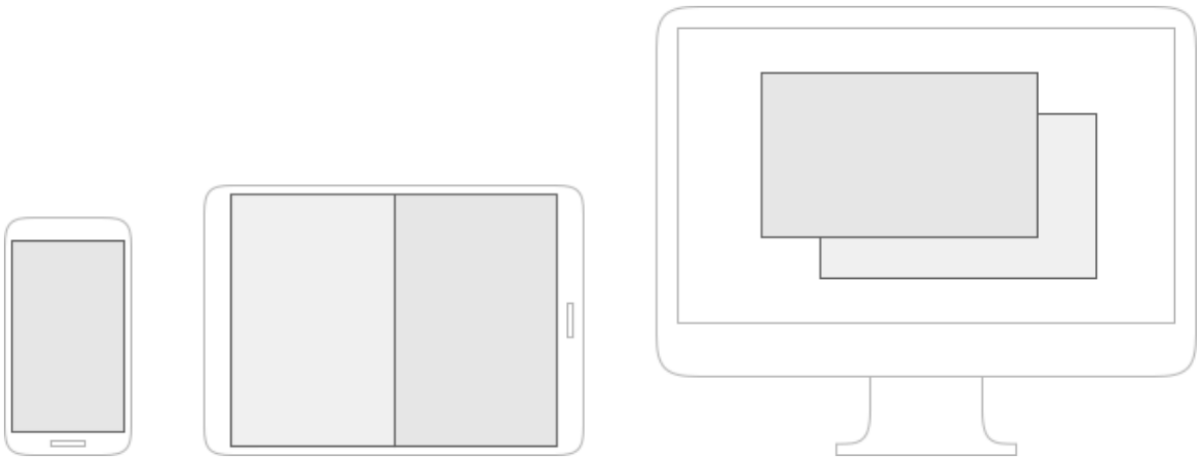


Left: Download button uses old focus color, Cancel button (secondary action) uses old grey color.

Right: Dialog uses generic answers, both buttons colored as main actions with confusing color meaning: “Will ‘No’ cause an irreversible action, and ‘Yes’ cause an additive action?”

8.3 Other design considerations

8.3.1 Convergence



Convergence is one of the milestones Ubuntu Touch wants to achieve. One app available across all screen sizes and input methods, such as mobile handsets, tablets, and laptops.

Convergence refers to the ability to use the same code base across multiple hardware form factors. Multiple devices using the same code base allows for features to be developed once, then shipped on multiple device types. This reducing development time and cost. The goal is for Ubuntu Touch to provide a consistent experience, whether you are using it on a big screen with mouse and keyboard or on a small touchscreen.

Screen size

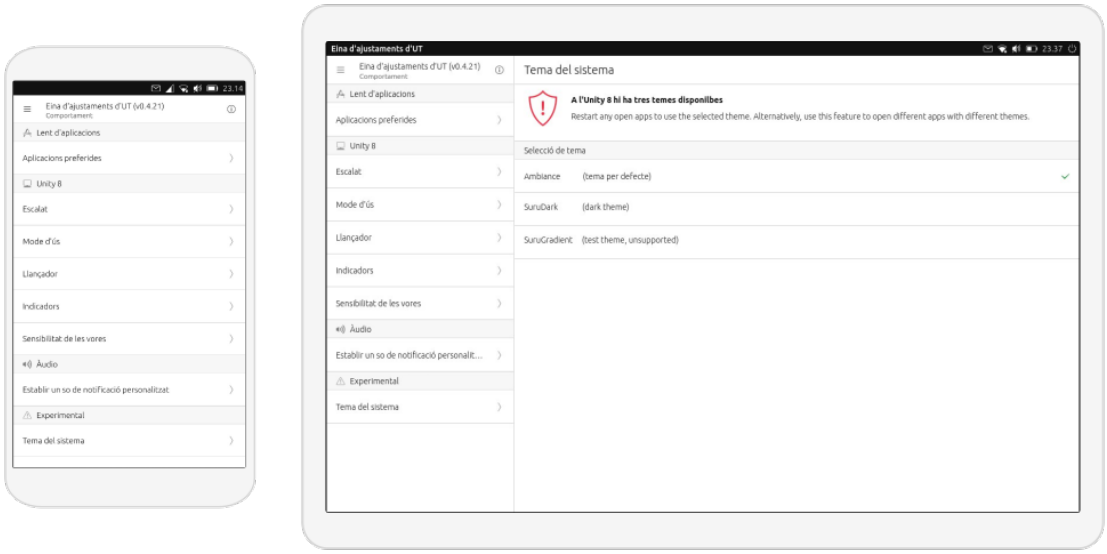
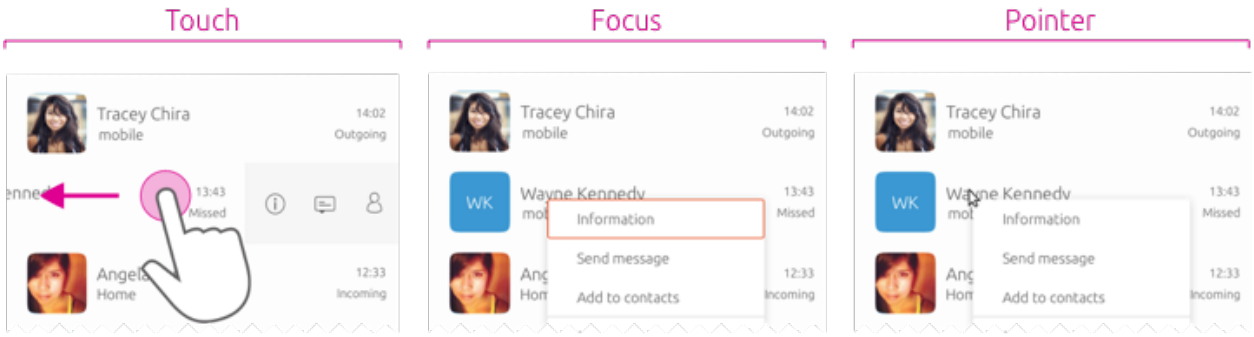


Fig. 1: Ubuntu Touch Tweak Tool app

Main page stays as the main page on phones or on the left on bigger screen devices. Additional pages on the phone, can be displayed at variable sizes on the right on bigger screens.

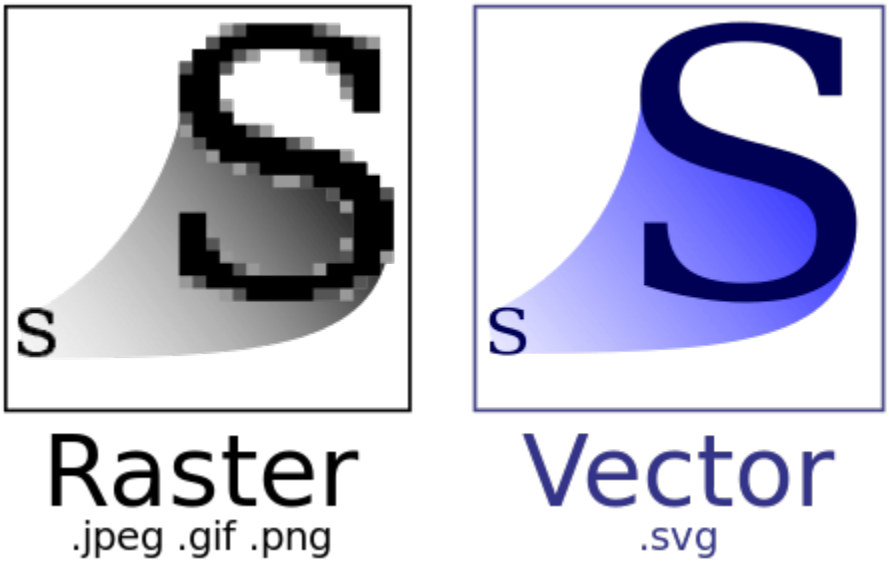
Input method

Using touch, a user can swipe or long-press on a list item to reveal a contextual menu. Using a pointer (mouse or trackpad) a user can right-click the item to reveal the contextual menu. Using a keyboard a user can focus the desired item and press the MENU key to open the context menu.



8.3.2 Scaling images

Scaling images is important in a convergent app. App icons and images can be created in scalable formats to improve their ability to scale.



One of the most popular scalable formats is SVG. SVG files can be created using software like Inkscape.

One particularity of qml, though, is that SVGs are rendered to use less memory, losing the advantages of scalability. When using a SVG file in an `Image` element we can use the property called `sourceSize`. This will define the rendered size of the SVG image. A SVG image with a `sourceSize` width and height of 252 pixels, will be rendered as if the scalable image would be a png of 252 per 252 pixels.

It is not advisable to define `sourceSize` as a relative size to avoid performance issues. A good practice is defining `sourceSize` based on a set list of thresholds. In the example below the image.svg will be rendered 60 per 60 grid units if the main view is bigger that 70 grid units but as an image of 40 per 40 grid units in the other cases. By defining these thresholds, we get scalable images without a huge performance cost.

```
Image {
  source: "image.svg"
  sourceSize.width: mainView.width > units.gu(70) ? units.gu(60) : units.gu(40)
  sourceSize.height: sourceSize.width
}
```

Read more about image performance in the Qt wiki.

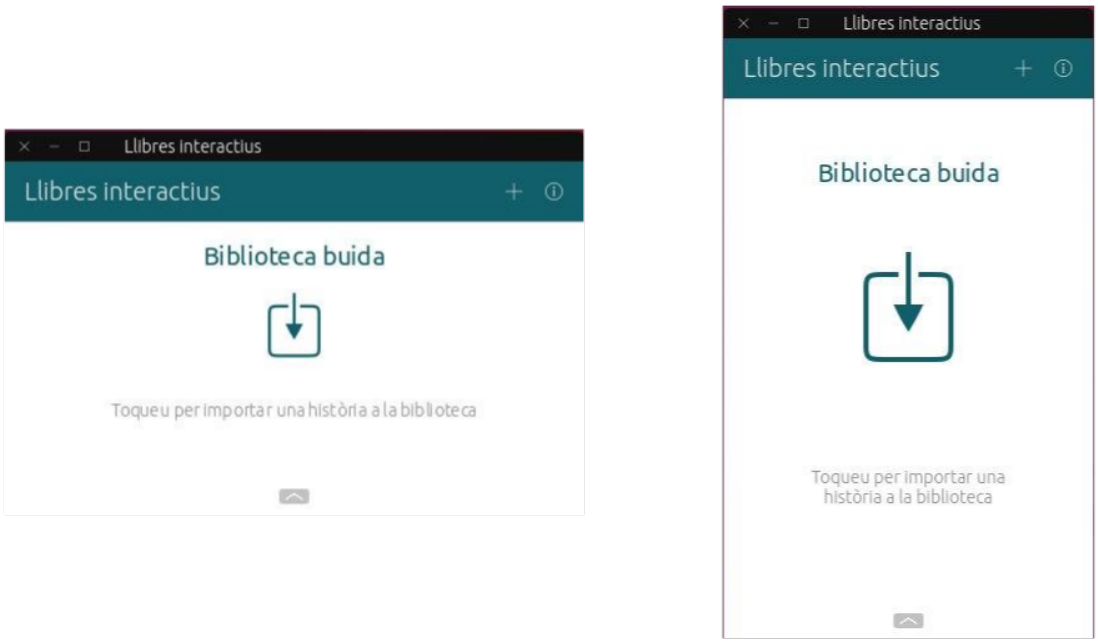


Fig. 2: Different rendered sizes of the same svg image

8.3.3 Accessibility

Accessibility is important to make all users to be able to use the operating system.

Inclusive palette and theme

We would like to create and implement color palettes helpful to people who have sight impairment:

- Additional palette proposal.
- A high contrast theme.
- Adaptable font size theme.

Input methods

Apps should be accessible using any combination of mouse, touch, and keyboard.

8.3.4 Translations

Adding translations to your app makes it more accessible to international users. Clickable will handle this process automatically, extracting new translatable strings into a pot file (the source strings of your app, to be translated).

Marking a string to be translatable can be achieved like this i18n.tr(string). For example:

```
Label {
    text: i18n.tr("My Label")
}
```

More information and options can be found in this a bit outdated tutorial on localization an app.

- Suru Icons

CHAPTER

NINE

SYSTEM SOFTWARE DEVELOPMENT

This section has various documents which will teach you how to work with the system level of Ubuntu Touch. This includes the Kernel, Android HAL, Ubuntu UI Toolkit, Unity8, and all of the other software that makes Ubuntu Touch what it is.

This section does not cover most of the *applications* preinstalled on Ubuntu Touch. See *Preinstalled apps* for more information on those.

9.1 Kernel and hardware abstraction

This page documents the resources and processes to build the kernel and hardware abstraction for UT devices. This document is useful if you would like to work on:

- A hardware related issue (camera, sensors, radios)
- A Linux kernel related issue
- An issue related to the system-image upgrade process

This document is not useful if you would like to modify *Preinstalled apps* or *System software*. See the respective documentation for each.

There are a few different groups of Ubuntu Touch devices with respect to how the kernel and hardware abstraction is implemented:

- Android 5.1 based ports
- Halium based ports
- Linux kernel

9.1.1 hammerhead, bacon and FP2

This page documents how to build the Kernel and the Android image for the LG Nexus 5 (hammerhead), OnePlus One (bacon), or Fairphone 2 (FP2).

Note: We highly suggest that you follow the *Halium porting guide* if you want to bring up a new device. The steps in here are only relevant for hammerhead, bacon and FP2.

This document assumes you already have knowledge of building Android or Halium. It also assumes that your device has Ubuntu Touch installed.

Getting set up

ubp-5.1 ports must be built using Ubuntu 16.04. A container or virtual machine based on 16.04 is recommended for this purpose.

Let's get started by installing some build dependencies:

```
sudo dpkg --add-architecture i386 && sudo apt update
sudo apt install schedtool gcc g++ g++-multilib zlib1g-dev:i386 \
    zip libxml2-utils bc python-launchpadlib phablet-tools
```

Create a directory for your ubp-5.1 source:

```
mkdir ~/ubp-5.1
cd ~/ubp-5.1
```

Next, we'll initialize the repository:

```
repo init -u https://github.com/ubports/android -b ubp-5.1-allthefixings --depth=1
```

Note: The `allthefixings` branch is provided for convenience. It adds all of the current UT device ports to the tree at the expense of a bit more downloaded data. If you are download-sensitive, initialize using `-b ubp-5.1` and use the manifest in build-scripts to pick the repos you like.

Finally, we'll download the source:

```
repo sync -j10 -c
```

Set up and build

With the sources downloaded, we need to set up our environment and build the images. Make sure you're in your ubp-5.1 directory to continue through these steps.

First, bring in the default Android build environment:

```
source build/envsetup.sh
```

Run `lunch` and pick the appropriate combo for your device. The name of the combination should start with `cm_`, followed by the device name and ending with `-userdebug`:

```
lunch
```

With that done, the build can be started:

```
mka
```

Install the new image

Now that the build is complete, we can flash it to the device. Note that all of these commands should be run from a terminal which has been set up with `source build/envsetup.sh` and `lunch` to ensure the needed tools are in your PATH.

We'll begin with the boot and recovery images. Boot your device into fastboot mode and run the following commands:

```
cout
fastboot flash boot boot.img
fastboot flash recovery recovery.img
```

Now boot your device to ensure your kernel build is sane. You may also want to boot into recovery to ensure it is working as well.

To install your new build of the system image, use the replace-android-system script. It can be run as follows with your device attached:

```
./replace-android-system system.img
```

9.1.2 PinePhone and PineTab kernel

This document describes how to build and install the PinePhone or PineTab kernel. First the recovery system is explained. Then the actual building and installation. At the bottom there are some references.

Recovery

The Ubuntu Touch image contains a recovery boot mode based on "jumpdrive". To get into the recovery press and hold the power and volume-up buttons until the led turns on.

The recovery presents the PinePhone as a network device over USB and runs a telnet server. When you connect the PinePhone to your desktop via USB, you will see the rndis device show up in your desktop's `dmesg`:

```
usb 1-7.2: Product: PinePhone
usb 1-7.2: Manufacturer: Pine64
usb 1-7.2: SerialNumber: Jumpdrive
rndis_host 1-7.2 1.0 usb0: register 'rndis_host' at usb-0000 00 14.0-7.2, RNDIS device
```

You can log in to the recovery system with `telnet 172.16.42.1`.

The recovery also exposes the eMMC and the microSD card over USB. Assuming you have already installed Ubuntu Touch on your microSD card you will see something like this in your desktop's `dmesg`:

```
usb-storage 1-7.2 1.2: USB Mass Storage device detected
scsi host3: usb-storage 1-7.2 1.2
scsi 2 0 0 0: Direct-Access     JumpDriv e eMMC                 PQ: 0 ANSI: 2
scsi 2 0 0 1: Direct-Access     JumpDriv e microSD              PQ: 0 ANSI: 2
sd 2 0 0 0: Attached scsi generic sg1 type 0
scsi 2 0 0 1: Attached scsi generic sg2 type 0
sd 2 0 0 0: [sdb] 30785536 512-byte logical blocks: (15.8 GB/14.7 GiB)
sd 2 0 0 1: [sdc] 124735488 512-byte logical blocks: (63.9 GB/59.5 GiB)
 sdb: sdb1 sdb2
sd 2 0 0 0: [sdb] Attached SCSI removable disk
 sdc: sdc1 sdc2 sdc3 sdc4 sdc5 sdc6 sdc7 sdc8 sdc9 sdc10
```

The eMMC with jumpdrive contains two partitions: pmOS_boot and pmOS_root.

The microSD card with Ubuntu Touch contains 10 partitions: loader, scr, persist, boot_a, boot_b, recovery_a, recovery_b, cache, system and userdata. To update the kernel you want to mount boot_a. Check which device is the SDcard in your OS and mount boot_a. In the example above this is sdc4. Inside that partition you'll see the kernel vmlinuz and related files: `config-5.6.0-pine64` `dtb` `initrd.img` `modules/` `System.map-5.6.0-pine64` `vmlinuz`.

Building the kernel

To install dependencies, get the source code, configure and build the kernel, run the following:

```
sudo apt install build-essential flex bison gcc-aarch64-linux-gnu libssl-dev
git clone -b pine64-kernel-ubports https://gitlab.com/pine64-org/linux.git
cd linux
ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- KBUILD_VERSION=arm64 LOCALVERSION=-pine64␣
↪make pine64_defconfig
ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- KBUILD_VERSION=arm64 LOCALVERSION=-pine64␣
↪make -j18 bindeb-pkg
```

Install the kernel

Boot into recovery and mount boot_a. Make a backup if needed. Then copy over your newly built kernel and if needed modules:

```
cp -v linux/debian/linux-image/boot/vmlinuz-5.6.0-pine64 [MOUNT POINT BOOT_A]/vmlinuz
cp -v linux/debian/linux-image/boot/config-5.6.0-pine64 [MOUNT POINT BOOT_A]
cp -v linux/debian/linux-image/boot/System.map-5.6.0-pine64 [MOUNT POINT BOOT_A]
sudo rm -rf [MOUNT POINT BOOT_A]/modules
sudo cp -rv linux/debian/linux-image/lib/modules [MOUNT POINT BOOT_A]
```

Afterwards be sure to unmount the partition so it is cleanly written. Now you can reboot the PinePhone. Once booted, you can verify that the kernel has been successfully updated by checking the build date with `uname -a` on the device.

References

- Pine64 wiki has general HW info, pointers to other PinePhone OS projects, HW revisions, serial UART over headphone jack (also in their store), spec sheets, known HW issues, etc
- Main UBports repository has instructions for how to install UT images on the PinePhone
- UBPorts issue tracker
- PinePhone forum from Pine64 is where many other communities gather that work with the PinePhone
- sunxi community is where much of the mainlining effort for Allwinner devices including the PinePhone takes place. Note their build instructions.
- Crust firmware implements a deep sleep state and runs on a dedicated System Control Processor outside the main CPU and DRAM.

9.1.3 Android 5.1 based ports

Android 5.1 based ports of Ubuntu Touch consist of the Linux kernel for this device plus a minimal Android system that is used to enable all the hardware. Every device has it's own fork of the Linux kernel. They are all heavily modified for the specific hardware used in that device. These forks are also based off quite old Linux kernel versions.

For some devices continuous integration (CI) has been set up to build both the Kernel as well as the Android system. Other devices have to be built manually from their repositories. For a third group of devices, we do not unfortunately, have the full source code available. The Bq and Meizu devices have kernel sources, but the "Android device tree" is not publicly available. The manufacturers of these devices provided binary builds instead.

- **With CI**
  - Nexus 5 (hammerhead)
  - OnePlus One (bacon)
  - Fairphone 2 (FP2)
- **Manual builds**
  - Nexus 4 (mako)
  - Nexus 7 2013 Wifi (flo)
- **(Partial) Binary builds**
  - Bq Aquaris E4.5 (krillin)
  - Bq Aquaris E5 (vegetahd)
  - Bq Aquaris M10 HD (cooler)
  - Bq Aquaris M10 FHD (frieza)
  - Meizu MX4 (arale)
  - Meizu Pro 5 (turbo)
  - Nexus 7 2013 GSM (deb)

Detailed steps for the three CI enabled devices can be found under *HAL for Nexus 5, OnePlus One, and Fairphone 2*.

For the Nexus 7 2013 Wifi (flo) head over to the Community Ports repository for flo.

The port for the Nexus 7 2013 GSM (deb) was created by a community member. Unfortunately those repositories have disappeared over time. So this build also falls into the group of prebuilt binaries.

Todo: Document the process for Nexus 4 (mako)

9.1.4 Halium based ports

Newer ports to Android devices are based on Halium. In some aspects Halium is similar to the 5.1 based ports:

- It also uses the device specific fork of the Linux kernel provided by the manufacturer.
- It also uses a minimal Android system to enable some hardware.

However, Halium permits a more generic way of porting to Android devices. This allows the work to be shared between multiple projects that bring different flavours of Linux systems to Android devices. Halium ports are also based on newer Android versions 7.1 and above.

Some examples of Halium ports are those for Sony Xperia X and Oneplus 3. Basically all devices that are listed on devices.ubuntu-touch.io, expect those explicitly mentioned above as 5.1, or below as Linux based ports.

All new ports of UT to Android devices should follow the Halium process. Further details can be found under *Halium porting*

9.1.5 Linux based ports

Linux based port refers to devices where a Linux kernel is used without any Android parts. The following devices are in this group:

- Desktop PC (x86)
- Librem 5 (librem5)
- Pinebook (pinebook)
- *Pinephone (pinephone)*
- *Pinetab (pinetab)*
- Raspberry Pi (rpi) (see also this blog post)

9.2 System Software guides

These guides will give you general instructions on building and testing your own changes to Ubuntu Touch system software. They are not not an exhaustive reference on everything you will come across during development, but they are a great starting point.

Note: If you get stuck at any point while going through this documentation, please contact us for help via the UBports Forum or your preferred communication medium.

9.2.1 Making changes and testing locally

On this page you'll find information on how to build Ubuntu Touch system software for your device. Most of the software preinstalled on your Ubuntu Touch device is shipped in the device image in the form of a Debian package. This format is used by several Linux distributions, such as Debian, Ubuntu, and Linux Mint. Plenty of documentation on deb packages is available, so we won't be covering it here. Besides, in most cases you'll find yourself in need of modifying existing software rather than developing new packages from scratch. For this reason, this guide is mostly about recompiling an existing Ubuntu Touch package.

There are essentially two ways of developing Ubuntu Touch system software locally:

- *Cross-building with crossbuilder*
- *Building on the device itself*

We'll examine both methods, using address-book-app (the Contacts application) as an example.

We only recommend developing packages using a device with Ubuntu Touch installed from the devel channel. This ensures that you are testing your changes against the most current state of the Ubuntu Touch code.

Warning: Installing packages has a risk of damaging the software on your device, rendering it unusable. If this happens, you can *reinstall Ubuntu Touch*.

Cross-building with crossbuilder

Crossbuilder is a script which automates the setup and use of a crossbuild environment for Debian packages. It is suitable for developers with any device since the code compilation occurs on your desktop PC rather than the target device. This makes Crossbuilder the recommended way to develop non-trivial changes to Ubuntu Touch.

Note: Crossbuilder requires a Linux distribution with `lxd` installed and the unprivileged commandset available. In other words, you must be able to run the `lxc` command. If you are running Ubuntu on your host, Crossbuilder will set up `lxd` for you.

Start by installing Crossbuilder on your host:

```
cd ~
git clone https://github.com/ubports/crossbuilder.git
```

Crossbuilder is a shell script, so you don't need to build it. Instead, you will need to add its directory to your `PATH` environment variable, so that you can execute it from any directory:

```
echo 'export PATH="$HOME/crossbuilder:$PATH"' >> ~/.bashrc
# and add it to your current session:
source ~/.bashrc
```

Now that Crossbuilder is installed, we can use it to set up LXD:

```
crossbuilder setup-lxd
```

If this is the first time you have used LXD, you might need to reboot your host once everything has completed.

After LXD has been set up, move to the directory where the source code of your project is located (for example, `cd ~/src/git/address-book-app`) and launch Crossbuilder:

```
crossbuilder
```

Crossbuilder will create the LXD container, download the development image, install all your package build dependencies, and perform the package build. It will also copy the packages over to your target device and install them if it is connected (see *Shell access via ADB* to learn more about connecting your device). The first two steps (creating the LXD image and getting the dependencies) can take a few minutes, but will be executed only the first time you launch crossbuilder for a new package.

Now, whenever you change the source code in your git repository, the same changes will be available inside the container. The next time you type the `crossbuilder` command, only the changed files will be rebuilt.

Unit tests

By default crossbuilder does not run unit tests; that's both for speed reasons, and because the container created by crossbuilder is not meant to run native (target) executables: the development tools (qmake/cmake, make, gcc, etc.) are all run in the host architecture, with no emulation (again, for speed reasons). However, qemu emulation is available inside the container, so it should be possible to run unit tests. You can do that by getting a shell inside the container:

```
crossbuilder shell
```

Then find the unit tests and execute them. Be aware that the emulation is not perfect, so there's a very good chance that the tests will fail even when they'd otherwise succeed when run in a proper environment. For that reason, it's probably wiser not to worry about unit tests when working with crossbuilder, and run them only when not cross-compiling.

Building on the device itself

This is the fastest and simplest method to develop small changes and test them in nearly real-time. Depending on your device resources, however, it might not be possible to follow this path: if you don't have enough free space in your root filesystem you won't be able to install all the package build dependencies; you may also run out of RAM while compiling.

> **Warning:** This method is limited. Many devices do not have enough free image space to install the packages required to build components of Ubuntu Touch.

In this example, we'll build and install the address-book-app. All commands shown here must be run on your Ubuntu Touch device over a remote shell.

You can gain a shell on the device using *Shell access via ADB* or *Shell access via SSH*. Remount the root filesystem read-write to begin:

```
sudo mount / -o remount,rw
```

Next, install all the packages needed to rebuild the component you want to modify (the Contacts app, in this example):

```
sudo apt update
sudo apt build-dep address-book-app
sudo apt install fakeroot
```

Additionally, you probably want to install `git` in order to get your app's source code on the device and later push your changes back into the repository:

```
sudo apt install git
```

Once you're finished, you can retrieve the source for an app (in our example, the address book) and move into its directory:

```
git clone https://github.com/ubports/address-book-app.git
cd address-book-app
```

Now, you are ready to build the package:

```
DEB_BUILD_OPTIONS="parallel=2 debug" dpkg-buildpackage -rfakeroot -b
```

The `dpkg-buildpackage` command will print out the names of generated packages. Install those packages with `dpkg`:

```
sudo dpkg -i ../<package>.deb [../<package2>.deb ...]
```

Note, however, that you might not need to install all the packages: generally, you can skip all packages whose names end with `-doc` or `dev`, since they don't contain code used by the device.

Next steps

Now that you've successfully made changes and tested them locally, you're ready to upload them to GitHub. Move on to the next page to learn about using the UBports CI to build and provide development packages!

9.2.2 Uploading and testing with `ubports-qa`

The UBports build service is capable of building Ubuntu Touch packages and deploying them to the UBports repository. This capability is offered to any developer who wishes to take advantage of it.

This guide assumes that you have a cursory understanding of using Git and making Pull Requests on GitHub.

To use the UBports build service, make sure you understand our *branch naming convention*. It is required that you follow the convention for deb-packages for CI to build your package correctly.

Fork the repository

The first step to make a change to any repository you don't have write access to is to fork it. Open your desired repository on GitHub and click the "Fork" button in the upper right corner. If offered, select an appropriate account to fork the repository to. Then, clone your fork to your computer.

Now you're ready to make changes!

Make and commit changes

Now that you have the package source downloaded, you can make your desired changes.

Before changing anything, make sure you have checked out the branch you want to work from (probably `xenial`, assuming you are making changes for the phone images). Then, create a new branch abiding by the *branch naming convention*.

After making your changes, commit them with a descriptive commit message stating what is wrong and why your changes fix that problem.

You have successfully created and committed your changes. Before pushing your changes, we'll want to make sure your device will install them.

Update the `debian/changelog` file

Generally, `apt` will not install a new package from any repository if it has a lower (or the same) version number as the package it replaces. Users may also want to see the changes that are included in a new version of a package. For that reason, we will need to update the package changelog to add a new version.

Note: This is not an exhaustive reference of the `debian/changelog` format. See deb-changelog(5) for more information.

Determine a new version number

To start, figure out what the current version numbering for the package is:

```
head debian/changelog
```

This will return a few lines, but the first is the most important to us:

```
morph-browser (0.24+ubports2) xenial; urgency=medium
```

The part inside the parentheses `(0.24+ubports2)` is our version number. It consists of several parts:

1. The `0.24` is the *upstream version number*, the version that the original project maintainers give to the release we are using. For most UBports projects, the repository you'll be working on is the original project code. This makes UBports the "upstream" of that project.

If you are making large changes to the repository and UBports is the upstream, you should increment the first part of the version number before the plus (+) and reset the distribution suffix. In our example above, you would make this new version number:

```
0.25+ubports0
```

If you are making changes only to the package build (files in the `debian/` folder), it is best to only increment the version suffix:

```
0.24+ubports3
```

Note: If you find a package which does not seem to follow the above versioning format, please contact us to ask how to proceed.

Write the changelog entry

Now it is time to write your changelog entry! Start with the following template:

```
PACKAGE-NAME (VERSION) DISTRIBUTION; urgency=medium

  * CHANGES

-- NAME <EMAIL>  DATETIME
```

If you open the `debian/changelog` file, you'll find that every entry follows this format. This helps everyone (including computers) read and understand the contents. This is used, for example, to name the output package correctly for every package version.

Let's assume I, John Doe, am making a packaging change to the `morph-browser` package for Ubuntu Touch. I'll replace the different all-caps placeholders above in the following way:

- PACKAGE-NAME is replaced with `morph-browser`
- VERSION is replaced with `0.24+ubports3` (which we determined above)
- DISTRIBUTION is replaced with `xenial`

- CHANGES is replaced with the changes I made in this release. This will include summarized information from my commit messages along with the bugs fixed by those changes. If I've fixed multiple bugs, I'll create multiple bullet points.
- NAME is replaced with my name, `John Doe`
- EMAIL is replaced with my e-mail, `john.doe@example.com`.

> **Note:** You should not use a "noreply" e-mail as your EMAIL for package changelog entries.

- DATETIME is replaced with the date and time I made this changelog entry in RFC2822/RFC5322 format. The easiest way to retrieve this is by running the command `date -R` in a terminal.

Note that no line in your changelog entry should exceed 80 characters in length.

With that, my new changelog entry follows:

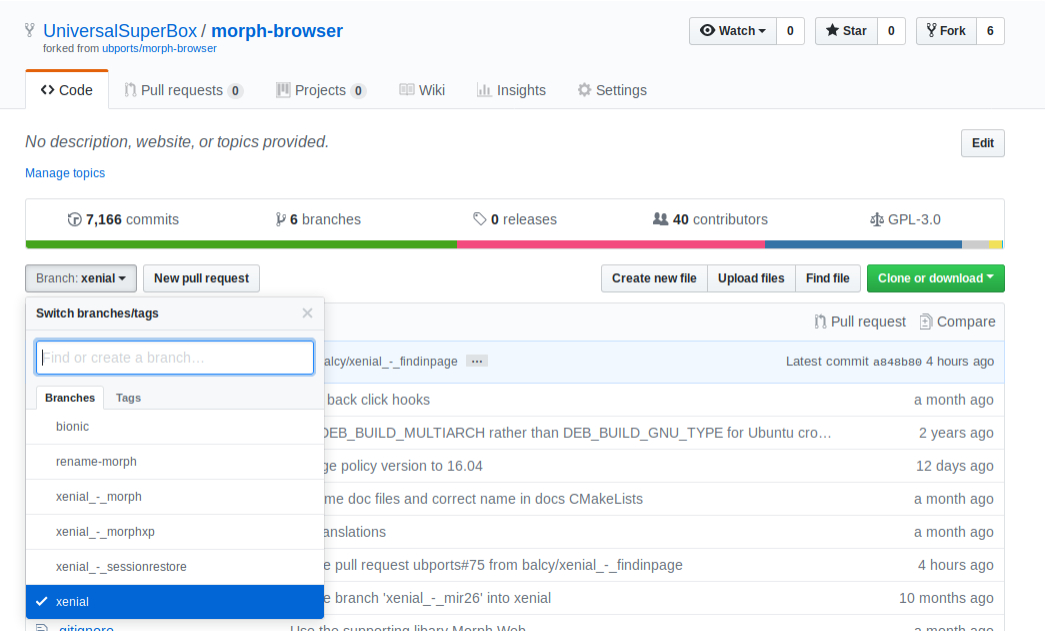
```
morph-browser (0.24+ubports3) xenial; urgency=medium

  * Add the new "Hello world" script to the package. Fixes
    https://github.com/ubports/morph-browser/issues/404.
  * Fix whitespace and formatting in the format.qml file

-- John Doe <john.doe@example.com>  Mon, 29 Oct 2018 12:53:08 -0500
```

Add your new changelog entry to the top of the `debian/changelog` file and commit it with the message "Update changelog". Push your changes. Now you're ready to make your Pull Request!

Create your pull request

A pull request asks UBports maintainers to review your code changes and add them to the official repository. We'll create one now.

Open your fork of the repository on GitHub. Navigate to the branch that you just pushed to using the "Branch" selector:



Once you've opened your desired branch, click the "New pull request" button to start your pull request. You'll be taken to a page where you can review your changes and create a pull request.

Give your pull request a descriptive title and description (include links to reference bugs or other material). Ensure that the "base" branch is the one you want your changes to be applied to (likely `xenial`), then click "Create pull request".

With your pull request created, we can move on to testing your changes using the UBports build service!

Test your changes

Once your pull request is built (a green check mark appears next to your last commit), you are ready to test your changes on your device.

Note: If a red "X" appears next to your last commit, your pull request has failed to build. Click the red "X" to view the build log. Until your build errors are resolved, your pull request cannot be installed or accepted.

We'll use `ubports-qa` to install your changes. Take note of your pull request's ID (noted as `#number` after the title of the pull request) and follow these steps to install your changes:

1. Ensure your device is running the newest version of Ubuntu Touch from the `devel` channel.
2. Get shell access to your device using *Shell access via ADB* or *Shell access via SSH*.
3. Run `sudo ubports-qa install REPOSITORY PR`, replacing `REPOSITORY` with the name of the repository you have submitted a PR to and `PR` with the number of your pull request without the `#`. For example run `sudo ubports-qa morph-browser 123` to install the PR number 123 to morph-browser repo.

`ubports-qa` will automatically add the repository containing your changed software and start the installation for you. All you will need to do is check the packages it asks you to install and say "yes" if they are correct.

If `ubports-qa` fails to install your packages, run it again with the `-v` flag (for example, `ubports-qa -v install ...`). If it still fails, submit the entire log (starting from the `$` before the `ubports-qa` command) to Ubuntu Pastebin and contact us for help.

Once `ubports-qa` is finished, test your changes to ensure they have fixed the original bug. Add the `ubports-qa` command to your pull request, then send the link to the pull request to other developers and testers so they may also test your changes.

When getting feedback from your testers, be sure to add the information to the pull request (or ask them to do it for you) so that everyone is updated on the status of your code.

Every time you make a change and push it to GitHub, it will trigger a new build. You can run `sudo ubports-qa update` to get the freshest changes every time this happens.

Celebrate!

If you and your testers are satisfied with the results of your pull request, it will be merged. Following the merge, the UBports build service will build your code and deploy it to Ubuntu Touch users worldwide.

Thank you for your contribution to Ubuntu Touch!

9.3 System Software reference

This section includes reference guides on how different parts of the Ubuntu Touch system interact to create the user experience.

There's not much here yet, but maybe you'd like to add some reference material? Check out *our guide to contributing to documentation* to learn more.

9.3.1 Working on the Calendar feature

The Calendar feature is provided by several components:

- The Calendar App, providing the user interface
- Evolution Data Server (often shortened as "EDS"), which is the backend where your calendars are stored
- sync-monitor, the service responsible for managing the synchronisation with remote calendars
- SyncEvolution, the service responsible for performing the synchronisation to a WebDAV/CalDAV remote calendar

Debugging

The most convenient way to run commands on the device and collect logs is by opening a remote shell. This can be done by using *Shell access via ADB* or *Shell access via SSH*. In the following sections, it's assumed that you've got access to a terminal console to the device.

Scheduling and account issues

If you are not sure whether the calendar is being synchronised, or whether the operation is successful, `sync-monitor` is the component that needs to be examined. This service should always be running in the background, and its logs can be found in `~/.cache/upstart/sync-monitor.log`. In order to see them printed in real-time as you operate on the device, you can run this command:

```
tail -f .cache/upstart/sync-monitor.log
```

These logs should be enough to give you an idea on whether your calendar accounts are recognised, and whether a synchronisation is scheduled, happening, and completing successfully.

Calendar data synchronisation issues

Follow these steps one you are confident that a synchronisation of your account is at least attempted, and want to investigate a synchronisation failure or issues with missing or duplicate items. On the device, it's possible to run `syncevolution` in debug mode. To do so, kill any existing processes:

```
pkill sync-evo
```

Then, start the `syncevolution` process with the proper environment variable:

```
SYNCEVOLUTION_DEBUG=1 /usr/lib/arm-linux-gnueabihf/syncevolution/syncevo-dbus-server
```

At this point, open the Calendar App (if it was not open already) and manually trigger a synchronisation (note that the Synchronisation action is only available if your device is connected to the internet): all the logs will appear in the terminal.

In some cases, the output from `syncevolution` might not be enough: for example, the raw HTTP data is usually not printed. Should you need to see that as well, then you'll have to modify a configuration file as well. `syncevolution`'s configuration files are located under `~/.config/syncevolution/`, in a subdirectory whose name takes the form `<provider-name>-<account-id>`. There might be stale directories as well, referring to old accounts which have been since deleted. To find out what accounts are still valid, you can invoke the `account-console` tool like this:

```
account-console list
```

This will print the list of the current valid accounts. Once you've figured out what is the account you are interested in, open the file `~/.config/syncevolution/<account>/peers/target-config/config.ini` and set the `loglevel` variable to a higher value (`11` seems enough to print all the HTTP traffic):

```
# level of detail for log messages:
# - 0 (or unset) = INFO messages without log file, DEBUG with log file
# - 1 = only ERROR messages
# - 2 = also INFO messages
# - 3 = also DEBUG messages
# > 3 = increasing amounts of debug messages for developers
loglevel = 11
```

Note that in order for these changes to take effect, you'll need to restart the `syncevolution` process again, as explained above.

9.3.2 Working on the Online Accounts subsystem

The Online Accounts subsystem consists of the following components:

- The Accounts UI: this provides the user facing components and the base classes for the whole Online Accounts feature. More in detail, you will find these subdirectories:
  - `click-hooks`: the programs run when a *Click application* making use of the Online Accounts functionality is installed or removed
  - `client`: the library that client applications can use to request access to an account
  - `online-accounts-service`: the main service, implementing the logic to process client requests; it has no UI, but is able to attach the `online-accounts-ui` on top of an application's UI
  - `online-accounts-ui`: the user interface which pops up when applications interact with online accounts (for example, the dialog that appears when an application requests access to an account)
  - `plugins`: the base classes and QML elements that account plugins can use in their implementation
  - `system-settings-plugin`: the user interface for the Accounts panel in the System Settings application
- The Account plugins define the available account providers and implement the user interface and logic for creating the accounts
- A set of components from the Accounts&SSO project: while development happens in the upstream projects, the UBports forks add the Debian packaging. These projects are:
  - libaccounts-glib: account management API for C/GLib applications
  - libsignon-glib: authentication API for C/GLib applications
  - libaccounts-qt: account management API for Qt applications

  - signond: authentication daemon, exposing a D-Bus API. It includes the `libsignon-qt` library, which offers an authentication API for Qt applications
  - signon-plugin-oauth2: OAuth (1.0 and 2.0) plugin for signond
  - accounts-qml-module authentication and account management API, for QML applications (can be used when implementing account plugins, too)
- The account-polld service, which runs in the background and checks every 5 minutes for new account activity (the interval is decided by the Ubuntu Push service)
- The account-polld-plugins-go, a repository of plugins for the `account-polld` service. The plugins in this repository are written in Go, but plugins can be written in whatever language and can reside in different repositories

Debugging

Account creation issues

It may happen that the account creation fails, either due to communication issues with the remote server, or to some bug in the account plugin itself (this can easily happen while developing a new plugin). In order to debug such situations, you can open a terminal (it's easier if done from a remote shell – you can gain a shell on the device using *Shell access via ADB* or *Shell access via SSH*) and start the `online-accounts-service` in debug mode:

```
pkill online-accounts-service
OAU_LOGGING_LEVEL=2 OAU_DAEMON_TIMEOUT=9999 online-accounts-service
```

If you believe that the issue might be caused by some errors in the authentication phase, you can also enabled extensive logging by `signond` like this:

```
pkill signond
export SSO_LOGGING_OUTPUT="stdout"    # signond logs to the syslog by default
SSO_LOGGING_LEVEL=2 SSO_DAEMON_TIMEOUT=9999 signond
```

At this point, repeat the operation that was failing, and you'll get all debugging output printed on the console.

9.3.3 QtMir and QtUbuntu

QtMir and QtUbuntu are Qt Platform Abstractions (QPAs) for the Ubuntu Touch platform. QtMir can be thought of as a server for Unity8 while QtUbuntu is used for client applications. Each of them allows their respective domains to use Qt's high-level abstractions of surfaces, windows, controls, and more, without worrying about the underlying operating system.

QtMir

The QtMir QPA allows Unity8, the shell of Ubuntu Touch, to be written in QML and Qt C++ while operating on Mir Surfaces and Windows. It also allows the desktop to be represented and reasoned about as a Qt Scene Graph.

You'll find most of QtMir's logging in Unity8's logs, normally stored at `/home/phablet/.cache/upstart/unity8` on Ubuntu Touch. Different logging categories, like `qtmir.sessions` and `qtmir.surfaces`, relate directly to different modules within QtMir's codebase.

While Unity8 shows almost all QtMir logging by default, you can enable even more verbose logging by placing the following content in the file `/usr/share/upstart/sessions/unity8.override`:

```
env QT_LOGGING_RULES='qtmir.*=true'
```

Then restart Unity8:

```
restart unity8
```

QtUbuntu

The QtUbuntu QPA uses the Mir client API and Ubuntu Platform API to provide Ubuntu Touch apps with a stable hardware compatibility API.

Since QtUbuntu is used directly by apps, any logging output from it will be located in an app's log file. Most of the time, QtUbuntu gives no logging output. However, if you would like to receive more output, you can set the `QT_LOGGING_RULES` environment variable appropriately. Since apps on Ubuntu Touch are started using Upstart's user session, you can set this for all apps until you restart your device with the following command:

```
initctl set-env QT_LOGGING_RULES='qt.qpa.mirclient.*=true'
```

We plan to replace QtUbuntu with QtWayland in the future.

Components

QtUbuntu has a number of sub-components to provide other features in the Ubuntu Touch platform. In theory, these could be used on other platforms to provide similar features. In practice, this has never happened.

QtUbuntu-Camera provides the `aalCamera` (Android Abstraction Layer Camera) plugin to QtMultimedia. This plugin allows apps to access Android device cameras through the QtMultimedia standard API. We are trying to replace this component with the gst-droid plugin for GStreamer for all new Android device ports.

QtUbuntu-Sensors provides Android haptic feedback, GPS, orientation, and accelerometer sensors to QtSensors. We are trying to replace this component with sensorfw, a single daemon capable of providing these functions using Android or standard Linux kernel drivers.

QtUbuntu-Media provides hardware encoding and decoding of audio/video content on Android devices to QtMultimedia. It is tightly integrated with media-hub.

9.3.4 MMS infrastructure components

oFono - responsible for providing the data context used to transfer MMS data (image/music). It also propagates wap push notifications to upper layers.

nuntium - daemon that listens to wap push notifications and activates the MMS data context on ofono on demand to send/receive MMS's. It provides a local store.

telepathy-ofono - talks to nuntium through dbus and is used to both relay messages from phone-app to nuntium and inject into the telepathy infrastructure MMS's received by nuntium as multi-part messages. This component also marks messages as read and delete successfully received messages from nuntium.

history-service - this component watches the telepathy communication and store the messages received by telepathy-ofono or sent by messaging-app.

telephony-service-approver - this component is in charge of adding SMS's to the messaging menu and to display incoming text notifications.

messaging-app - It renders MMS's in the conversation view and also provides a way to attach media files to a message in order to send an MMS.

General Description

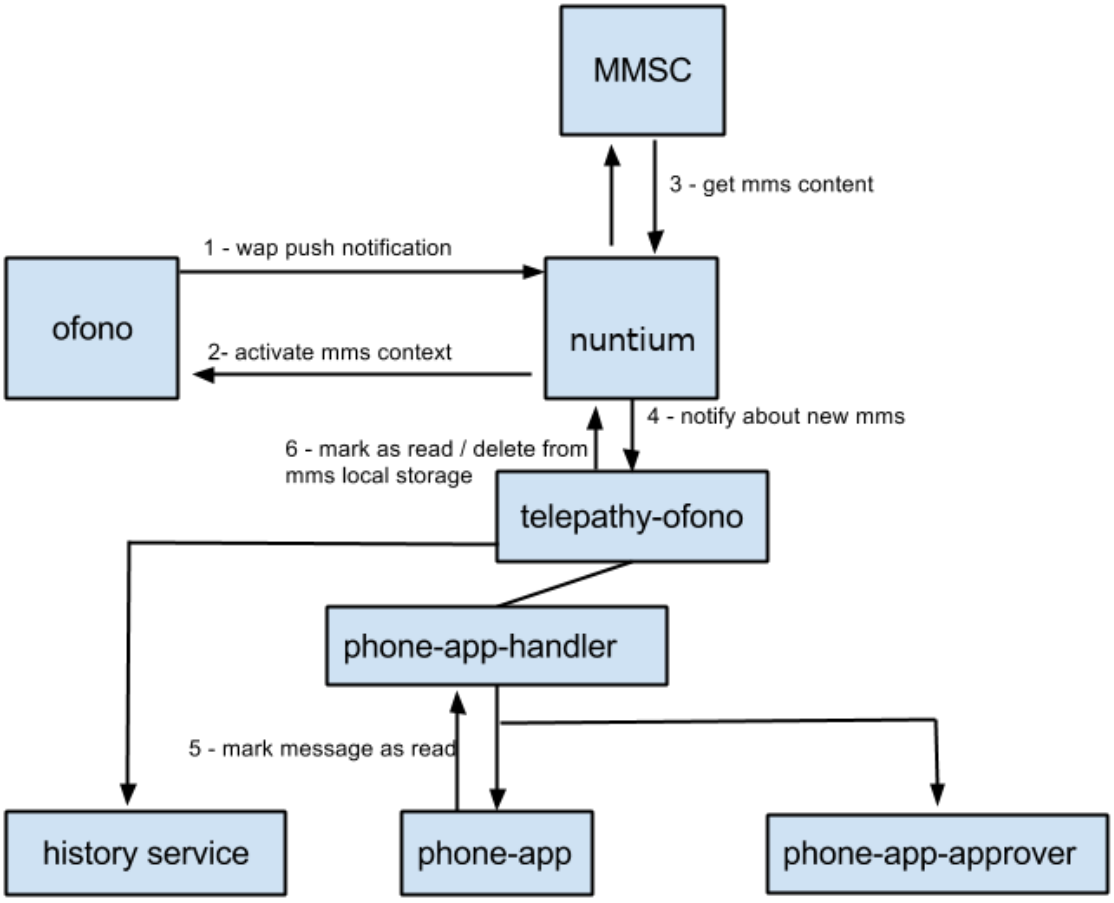
Incoming MMS

- An MMS WAP push notification is received by ofono and propagated to nuntium
- nuntium enables the MMS context in ofono, retrieves the content from MMSC and propagates the new MMS to upper layers.
- A new MMS arrives at telepathy-ofono.
- If automatic retrieval is not enabled or not available at the moment, a control message is sent to the app
- messaging-app request the retrieval of the message to telepathy-ofono.
- After the message is retrieved from MMSC, it is delivered to the apps via normal text channel in a multi-part message (one text/html part, one text/plain part and one part per attachment). If the control message was previously delivered, this real message have a flag telling it is a replacement for the previous message.
- The new message history service saves the attachment parts to disk, and replaces the paths on text/html to point to the new attachment locations. It also saves a flag and the content type.

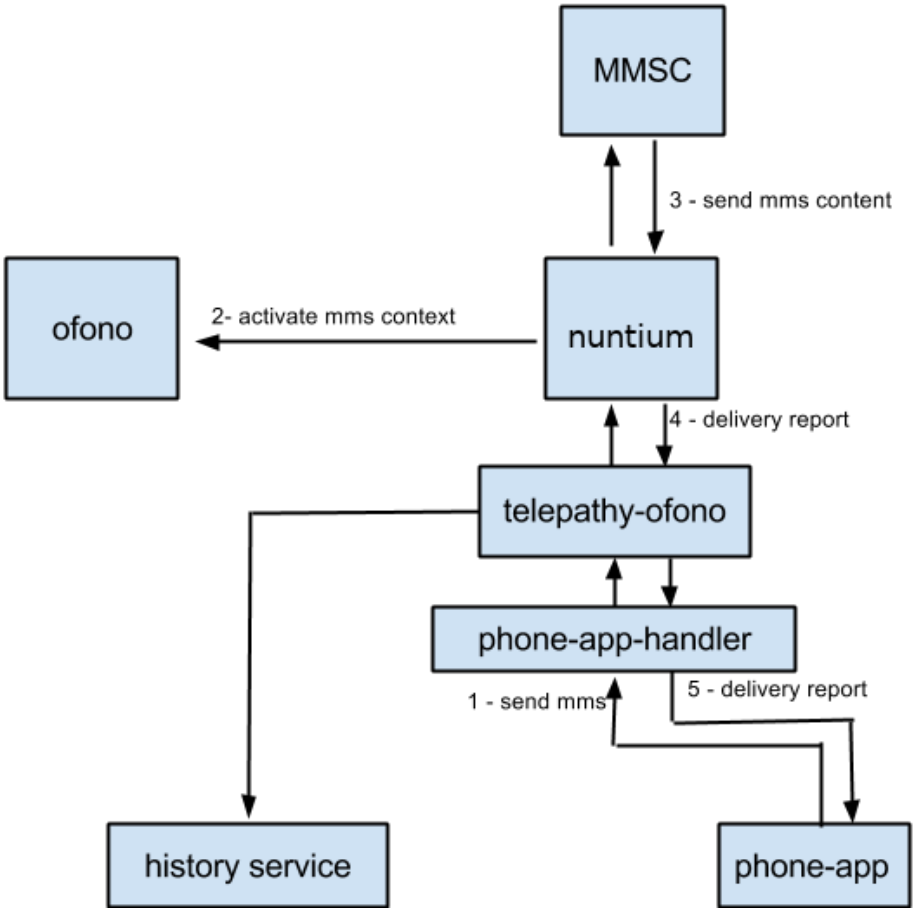
Outgoing MMS

- messaging-app creates a new text channel in telepathy-ofono, or uses an existing channel.
- messaging-app sends the MMS as a multi-part message to telepathy-ofono
- nuntium enables the MMS context in ofono, sends the MMS through MMSC and signals the upper layers about the current status (sent, draft).
- telepathy-ofono sends signals to applications with delivery reports.

Incoming MMS diagram



Outgoing MMS diagram



References

- initial source: https://wiki.ubuntu.com/Touch/Specs/MMSInfrastructure

CHAPTER

TEN

INTRODUCTION

Porting is the process of making Ubuntu Touch run on new hardware, i.e. on devices that have not previously been able to run Ubuntu Touch.

On the vast majority of commercially available devices crucial parts of their systems are not open source. Furthermore, these proprietary parts are specific, not only to the manufacturer, but usually also to the individual model.

Porting Ubuntu Touch involves building it in such a way as to integrate it with these proprietary components on a device so as to enable it to "talk to" the device hardware.

Before attempting to port Ubuntu Touch to a new device, there are a few things to understand and consider, some research to be done, and a suitable build environment needs to be set up. This section addresses these points.

10.1 Introduction to porting

This first section will introduce you to the specifics of porting Ubuntu Touch to an Android device. Note that it is written with the general public in mind, not primarily the experienced porting developer. In subsequent sections we have made an effort to differentiate by keeping the main text more concise in order to meet the needs of the more experienced reader, while providing links to supplementary reading for the less experienced.

The guide as a whole is written as a sequence of steps describing how an ideal port might proceed. However, porting is unpredictable and messy. Every device is different and in reality you will likely iterate and revisit some steps, skip over other steps and uncover new and undocumented challenges.

10.1.1 What is Ubuntu Touch?

Ubuntu Touch is an open source operating system for mobile devices. It can be ported to devices that originally shipped with Android OS. Alas, the majority of these devices are dependent to some degree on proprietary software.

To be specific, device vendors tend to keep the code that speaks directly to the device hardware (the low level device drivers) proprietary. These components are commonly called the vendor 'blobs' (Binary Large OBjects). The vendor blobs need to be incorporated into an Ubuntu Touch port. Note that these components are specific not only to each device, but also to each Android version. It is therefore necessary to secure the correct version of these components when building a port.

This is why Ubuntu Touch cannot be built completely from source code for most commercial devices. Instead, porting the system to these devices involves integrating the previously mentioned vendor blobs into the rest of the system, which can be built from source.

The next component of Ubuntu Touch is a pre-compiled root filesystem which needs to be installed on the device. This component does not communicate directly with the device hardware. Instead, this communication is mediated by a Hardware Abstraction Layer (HAL) which needs to be built for each specific device, because each device has its

specific hardware architecture. This component is called Halium and is available in different versions (5.1 which is largely obsolete, 7.1, 9 and 10 as of writing) corresponding to different Android versions.

The Halium project enables Linux systems to run on Android hardware. It is a joint effort by multiple mobile operating systems, notably Lune OS and UBports.

UBports porting builds on top of Halium porting. Consequently, you will be using both the Halium porting guide and the UBports porting guide. At times it may also be helpful to test with one of the other operating systems to debug a problem from different angles.

Halium is an indispensible part of an Ubuntu Touch port and is available in the form of open source software. Developing a new version of Halium is a very considerable task which is why only a few versions of Halium are available. Each port of Ubuntu Touch has to be based on one of the available Halium versions and vendor blobs from the corresponding Android version. See the first two columns of the table below for details.

Thus an Ubuntu Touch port is composed of the these components:

- The Ubuntu Touch (UT) root filesystem (rootfs)
- Halium (contained in the boot and system images)
- The vendor blobs

You, the porter, need to build Halium (in part or in whole, depending on *porting method*) and install this together with the Ubuntu Touch rootfs in order to create a functioning Ubuntu Touch port.

10.1.2 Android and Halium versions

Halium is built using source code for a modified version of the Android operating system called LineageOS (see the LineageOS website and wiki). The required source code is available online and needs to be downloaded and configured to build the correct Halium version for each individual device port. The table below shows which versions are required for the different Halium versions.

Android version	Halium version	Lineage OS (LOS)
7.1	7.1	14.1
9.0	9.0	16.0
10.0	10.0	17.0

10.1.3 Generic System Image

Starting with Android version 9.0, a significant change of architecture was introduced. The device-specific vendor blobs now reside on a separate partition instead of sharing a partition with the rest of the system image. This separation of device-specific code from generic code made possible what is known as the **Generic System Image (GSI)**.

A GSI is a system image that is built to be able to function with a wide range of devices. Android devices, as of version 9.0, use a GSI. For more information, see the Android Developer pages

The development of the Android GSI architecture also cleared the way for the now available generic Halium 9.0 arm64 system image (hereafter referred to as *the Halium GSI*, or simply *the GSI*) which is used for Ubuntu Touch. This, however, is somewhat different from the Android GSI. For a more detailed explanation of the Halium GSI, please refer to the wiki page on Gitlab CI builds of the generic Halium system image.

What does this mean for the porting process?

Since the GSI is a prebuilt, device-independent component, it effectively simplifies the task of building a viable port by removing much of the meticulous and time consuming task of getting the hardware-specific vendor blobs compiled into the system image and configured to function properly.

10.1.4 Porting methods

This guide documents three different porting methods, which we call: **Full system image method**, **Halium-boot method**, and **Standalone kernel method**. When porting based on Halium 7.1 the Full system image method is the only available method to follow. For Halium 9.0 all three methods are possible.

Full system image method

This porting method requires building both the boot image (halium-boot.img) and the full device specific system image (system.img) from source and installing these together with the UBports root file system (rootfs). For Halium 7.1 ports this is the only possible method (Consequently, this method is sometimes referred to as *the Halium 7.1 method*). For Halium 9.0 it is also possible to use this method, however for Halium 9.0 the other two methods below are probably easier.

Halium-boot method

For this porting method it is sufficent to build the halium-boot.img and install this together with the Halium GSI and the UBports rootfs. This method can be used for Halium 9.0 ports.

Standalone kernel method

This porting method only requires building the kernel and installing this together with the Halium ramdisk, the Halium GSI and the UBports rootfs. This method can be used for Halium 9.0 ports.

All methods share some common steps. However, there are also significant differences that must not be missed. Therefore, the methods will be treated separately in the subsequent sections where needed.

The remainder of this section gives some words of advice to new porters. If you already have porting experience or ROM building experience, you can likely skip straight to *Preparations*.

10.1.5 The challenges of the porting process

Building the necessary components and getting them to work together properly always involves an amount of code modifications, configuring and testing, but considerably moreso when doing full system image builds, compared to builds using the GSI (see *porting methods*).

Luckily, there is a community of porters out there who are eager to see Ubuntu Touch ported to new devices. When you run into trouble, you should search the sources below (*Getting community help*) to see if others before you have solved the issue. There are online Telegram chat groups you can join to ask for help, but please bear in mind that those participating are doing so in their spare time.

10.1.6 Prior knowledge and skills

Porters come in all sizes and shapes, so to speak. Therefore, this guide does not presuppose extensive knowledge or skills in any particular field. You should, however, as a bare minimum be familiar with some common shell commands and be comfortable working from the terminal on you host PC. Furthermore, the guide is based on a host PC running Linux. If you have some knowledge of programming, this will come in handy at some point, especially if you are familiar with C / C++. Also, you should familiarize yourself with git and set up a Github or Gitlab account to keep track of your code changes. It is wise to start documenting your steps from the very beginning.

We have attempted to give a certain amount of explanation along the way. However, this guide is not an in-depth reference into the architecture and inner workings of Ubuntu Touch, and gaining a deeper understanding will consequently require an amount of research on your part.

10.1.7 Getting community help

When you run into trouble, and you will, refer to one or more of the sources below:

- Telegram: @halium
- Telegram: @ubports_porting
- The UBports Forum
- Matrix: #halium:matrix.org

10.1.8 General advice

The more rigorous you are at making notes and documenting your steps, the less time you will spend backtracking your steps and guessing your way along. When dealing with issues that arise along the way, it is wise to work on them one at a time. If you try to correct several things at once, you risk ending up trying to guess which changes solved a given issue, which easily leads to breaking the functionality in question once more at some later stage.

If you are not discouraged after reading this, we welcome your efforts and wish you the best of luck!

The next section presents a key to the rest of this guide.

10.2 Preparations

Not all devices can be made to run Ubuntu Touch. Research your target device before you start.

To determine your device's specifications, search for it on GSM Arena and/or Device Specification. Refer to Pick an Android target device in the Halium Porting Guide for further information on requirements and how to check if your device qualifies.

10.2.1 Locate relevant guides and other information

If your desired target device complies to the above, the next thing to do is locate the available guides and other documentation. This step will help you later on when you run into issues developing your port. For now, the main thing to look for is how to unlock your bootloader. (See below).

Head over to the LineageOS Wiki. Look up your device and read and bookmark the guides that are listed there. Try a web search for additional information, specifying both its retail name and the code name you found on the LineageOS site. Be careful to check that the information you gather applies to your specific device, keeping in mind that many devices are sold in a number of different variants with different hardware specifications.

Another rich source of information is the XDA Developers Forum.

10.2.2 Unlock the bootloader

Vendors usually provide their devices in a locked bootloader state. This is a kind of software "seal" intended to prevent modifications directly to the operating system and system software. If you unlock the bootloader, you will be able to make such modifications, but in this state your device's warranty might be void. The choice is yours, but this step is mandatory if you wish to install Ubuntu Touch on the device.

Devices differ and there is no general method that covers all makes and models. Therefore, you need to check the aforementioned guides for instructions on how to unlock the bootloader of your particular device.

10.2.3 Install TWRP recovery

You need to install a custom recovery image on your device in order to handle formatting and flashing image files onto it. Head over to the Team Win Recovery Project and locate the image file for your device. Follow the installation instructions provided on the website.

10.2.4 Recommendations for the host/build PC

The remainder of this guide presumes you are using a build PC running Linux. Although high performance always is nice, stability and sufficient RAM and harddisk space are the main concerns here. You do not need the latest and the best hardware, nor do you need the latest distribution release. In fact, it is not uncommon to run into issues when choosing the very newest release, as some of the software needed may not yet have been built for it. For example, as of writing the latest release of Ubuntu Linux is 20.04, but many still recommend using the previous long term support release, 18.04.

Having completed the steps above, you are now ready to set up your build environment and start the porting process per se (next section).

10.3 Setting up the build environment

Your host PC needs a number of tools installed before you can begin to port. This section describes the necessary preparations.

10.3.1 Prerequisites

When setting up the build environment you need to have Python 3.6 or newer installed on your system. This can be installed via your system's package management system. Significant changes in syntax were introduced from Python 2 to Python 3, and some stages of the porting process may require Python 2 instead of Python 3. To check which version is active on your system, type:

```
python -V
```

Note: Any Linux distribution can easily be set up to switch between Python versions. Consult the documentation for your distribution to find out how this can be done.

Debian (Stretch or newer) / Ubuntu (16.04 or 18.04)

If your host PC has a 64-bit architecture (amd64), enable the usage of the i386 architecture:

```
sudo dpkg --add-architecture i386
```

Update your package lists to take advantage of the new architecture:

```
sudo apt update
```

Install the required dependencies:

```
sudo apt install git gnupg flex bison gperf build-essential \
zip bzr curl libc6-dev libncurses5-dev:i386 x11proto-core-dev \
libx11-dev:i386 libreadline6-dev:i386 libgl1-mesa-glx:i386 \
libgl1-mesa-dev g++-multilib mingw-w64-i686-dev tofrodos \
python-markdown libxml2-utils xsltproc zlib1g-dev:i386 schedtool \
repo liblz4-tool bc lzop imagemagick libncurses5 rsync
```

Ubuntu (20.04 or newer)

If your host PC has a 64-bit architecture (amd64), enable the usage of the i386 architecture:

```
sudo dpkg --add-architecture i386
```

Update your package lists to take advantage of the new architecture:

```
sudo apt update
```

Install the required dependencies:

```
sudo apt install git gnupg flex bison gperf build-essential \
zip bzr curl libc6-dev libncurses5-dev:i386 x11proto-core-dev \
libx11-dev:i386 libreadline6-dev:i386 libgl1-mesa-glx:i386 \
libgl1-mesa-dev g++-multilib mingw-w64-i686-dev tofrodos \
python3-markdown libxml2-utils xsltproc zlib1g-dev:i386 schedtool \
liblz4-tool bc lzop imagemagick libncurses5 rsync \
python-is-python3
```

Create a directory named ‘bin’ in your home directory, and include it in your path:

```
mkdir -p ~/bin
echo export PATH=\$PATH:\$HOME/bin >> ~/.bashrc
source ~/.bashrc
```

Download the repo script and make it executable:

```
curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
chmod a+rx ~/bin/repo
```

Arch

If your host PC has a 64-bit architecture, you need to add the [multilib] repository to your /etc/pacman.conf . This will allow you to install and run i686 packages. Please refer to 'Official Repositories/multilib' on the Arch Wiki.

Make sure you have the base-devel package installed.

Install the required dependencies from AUR:

```
git clone https://aur.archlinux.org/halium-devel.git && cd halium-devel && makepkg -si
```

CHAPTER

ELEVEN

BUILDING AND BOOTING

Now it's time to download and configure the necessary source code. Then we'll build it and get it to boot. Firstly follow the **Building** steps for your Halium version and porting method. Secondly, proceed with the final **Install and boot** section.

11.1 Building

11.1.1 Full system image method or Halium-boot method

If you are using either of the **Full system image** method (for Halium 7.1 or 9.0), or the **Halium-boot** method (for Halium 9.0), then please pick the two steps below matching your Halium version.

Halium-7.1 - Setting up the sources

The sources necessary for the **full system image build** for Halium-7.1. Halium 7 requires a **full system image build**.

Setting up the sources for Halium-7.1 builds

Create a directory for your Halium source tree:

```
mkdir halium && cd halium
```

This directory will be referred to as BUILDDIR throughout the remainder of this guide.

First, initialize your source to the correct version of Halium:

```
repo init -u https://github.com/Halium/android -b halium-7.1 --depth=1
```

Then download the code by issuing the command:

```
repo sync -c -j 16
```

The download will take some time as it counts several gigabytes. If you have a fast internet connection, you may set an extra JOBS=[number] environment variable at the beginning of the command to open more parallel downloading jobs. Generally, 12 is recommended, which is the default. When it completes, your BUILDDIR will contain a copy of the Halium source tree, but important parts are still missing.

Adding your device-specific source

The next step is to add the device-specific sources that need to be integrated into the source tree before you can attempt to build. The missing sources are those required to build the kernel as well as a host of other hardware-specific components of the port.

The necessary sources need to be located and specified by creating a device manifest file (or editing an existing one) in the directory `BUILDDIR/halium/devices/manifests`.

Note: A correct device manifest is crucial to the success of your port.

Locating the sources

Locate your device repository on LineageOS's GitHub organization. This is done by typing your device's codename into the search box. The device repository follows the naming convention: android_device_[manufacturer]_[device]. Make a note of this name.

Open the device repository on Github. It will contain a `lineage.dependencies` (or `cm.dependencies`) file which specifies all other repositories that your device is reliant upon.

Creating the device manifest file

Now create (or edit) the file `BUILDDIR/halium/devices/manifests/[manufacturer]_[device].xml`. (The description below presupposes that you are creating the file from scratch.)

Paste the following into the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>

</manifest>
```

The device repository

Your sources must now be entered into the manifest. Start with your device repository. Between the <manifest> and </manifest> tags, create the line below, making sure to replace the information inside the square brackets with your own:

```
<project path="device/[manufacturer]/[device]" name="[repository name]" remote="[remote]
↪" revision="[revision]" />
```

'path' specifies the target path (subdirectory of your BUILDDIR) where the source code from repository 'name', which is found at 'remote', will be placed. The repository may contain a number of branches and it is important to secure the correct branch with 'revision='

Note: The revision attribute may be omitted if the default revision for this remote is the one that you wish to use.

If you are unsure of your remote, refer to *Remotes*.

Dependencies

Now create more lines like the previous one, using the `lineage.dependencies` (or `cm.dependencies`) file you found earlier in your device repository. This file lists all the remaining repositories needed to build for your selected device. Create one line for each entry in this file, using the following template:

```
<project path="[target_path]" name="[repository]" remote="[remote]" revision="[revision]
↪" />
```

The target path is found in the repository's name. The preceding "android" or "proprietary" is omitted and underscores are replaced with slashes. For example, `android_device_lge_bullhead` goes in `device/lge/bullhead`.

Vendor blobs

Vendor blobs go in the `vendor/` folder of your `BUILDDIR`.

The first place to search for your device's vendor blobs is in 'TheMuppets' GitHub organization. Enter your device's codename into the search field and see if you get a match. If you are unable to find your device in TheMuppets, you will need to search elsewhere.

It may also be possible to extract the vendor blobs from the device itself by following instructions for your device on the LineageOS wiki as applicable.

Remotes

A remote entry specifies the name, location (fetch) prefix, code review server, and default revision (branch/tag) for the source.

You can create a remote by adding a remote tag to your manifest as shown in the following example:

```
<remote  name="aosp"
    fetch="https://android.googlesource.com"
    review="android-review.googlesource.com"
    revision="refs/tags/android-7.1.1_r25" />
```

Only the name, fetch, and revision attributes are required. The review attribute specifies a Gerrit Code Review server, which probably won't be useful for initial porting purposes.

For example, let's say that you have a bunch of repositories at https://github.com/SomeUserName/ and your desired branch name is los-14.1 in each. You would create a remote as follows and place it into your local manifest:

```
<remote name="sun"
    fetch="https://github.com/SomeUserName"
    revision="los-14.1" />
```

There are also some remotes available to you by default, though they differ between Halium versions. The following tables will help you identify these. See more information on these remotes can be found at the top of the file `BUILDDIR/halium/.repo/manifests/default.xml`.

The following remotes are available by default in Halium 7.1:

Remote Name	Remote Description, URL
aosp	Android Open Source Project, https://android.googlesource.com
los	LineageOS, http://github.com/LineageOS
hal	Halium (link to GitHub root for legacy reasons), http://github.com
them	TheMuppets, http://github.com/TheMuppets
them2	TheMuppets (for some xiaomi vendor repos) https://gitlab.com/the-muppets

Halium-7.1 - Building

For Halium-7.1 porting it is necessary to build both halium-boot.img and system.img.

Initializing the build environment

First you need to initialize the environment using the envsetup.sh tool included in your source tree. Enter your BUILDDIR and issue the command:

```
source build/envsetup.sh
```

The output will look like this:

```
including device/lge/bullhead/vendorsetup.sh
including vendor/cm/vendorsetup.sh
including sdk/bash_completion/adb.bash
including vendor/cm/bash_completion/git.bash
including vendor/cm/bash_completion/repo.bash
```

The `breakfast` command is used to set up the build environment for a specific device. From the root of your BUILDDIR run the following, replacing [CODENAME] with your device's specific codename:

```
breakfast [CODENAME]
```

Breakfast will attempt to find your device, set up all of the environment variables needed for building, and give you a summary at the end.

Modify the kernel configuration

The default LineageOS/Halium kernel configuration file needs modifications as Ubuntu Touch requires a slightly different kernel config than Halium, including enabling Apparmor. A script that does this job is provided in your Halium source tree: `BUILDDIR/halium/halium-boot/check-kernel-config`.

Locate your configuration file. It should be at `arch/arm/configs/<CONFIG>` or `arch/arm64/configs/<CONFIG>` depending on the architecture of your device. If you have trouble finding it, run `grep "TARGET_KERNEL_CONFIG" device/<VENDOR>/<CODENAME>/BoardConfig.mk` to determine the location.

Then, from the root of your BUILDDIR, run:

```
./halium/halium-boot/check-kernel-config path/to/my/defconfig -w
```

You may have to do this twice. It will likely fix things both times. Then, run the script without the `-w` flag to see if there are any more errors. If there are, fix them manually. Once finished, run the script without the `-w` flag one more time to make sure everything is correct.

Ubuntu Touch requires setting console=tty0

The halium-boot initramfs expects `/dev/console` to be a console device and will not start init if it is not available. This is commonly the case on recent devices, because they either have UART disabled or `console=` is not specified (null) by default. This can be fixed by supplying `console=tty0` as the last argument in the kernel cmdline. To achieve this, proceed as follows:

It should be done in the makefile named `BoardConfig.mk` (or `BoardConfigCommon.mk`) located in the root directory of your device tree, e.g. `~/halium/device/<vendor>/<model_codename>/BoardConfig.mk`

Add the following line:

```
BOARD_KERNEL_CMDLINE += console=tty0
```

If your makefile already includes a line beginning with `BOARD_KERNEL_CMDLINE`, you may add it just below that to keep things tidy.

Note: The above method, although the preferred one, may not work for some Samsung devices. The result will be that you cannot get access to the device through ssh after boot, and Unity 8 will not be able to start. If you run into this problem, you can specify the setting in your device's kernel config file instead. Add the following lines:

```
CONFIG_CMDLINE="console=tty0"
CONFIG_CMDLINE_EXTEND=y
```

Note: In rare cases the bootloader overwrites the kernel command line argument, rendering the setting above useless. This is the case for the Google Pixel 3a (sargo). To deal with this issue, replicate this commit.

Build

Halium will use the mkbootimg tool for creating the boot image. In most cases it is not on the local harddisk, so it can be built by issuing:

```
mka mkbootimg
```

Now build the `halium-boot.img` using the commands:

```
export USE_HOST_LEX=yes
mka halium-boot
```

Note: If you prefer `make` instead of `mka` you should set `-j[num]` for parallel building, which reduces build time. Replace `[num]` with the number of threads in your system plus 2.

Build errors

There are a number of known build errors which you might encounter, depending first of all upon how rigorous you have been in following the steps outlined, but you may also run into unforeseen issues. If your build fails at first, make sure you have done exactly as described, then go through the list of known errors in the Halium guide.

I your particular error is not listed, you will need to do some research of your own. If you end up here, know that there is a community of porters, developers and enthusiasts who might be able to help you. Refer to *Getting community help*.

Building the system image (system.img)

Once you have successfully built halium-boot.img you can proceed to directly to building system.img:

```
mka systemimage
```

Likelier than not, you will run into one or more errors along the way when building the system image. A number of possible errors are documented in the Halium guide. If yours is not listed, *seek community help*.

Halium-9.0 - Setting up the sources

Initializing and downloading the Halium source

Create a directory for your Halium source tree:

```
mkdir halium && cd halium
```

This directory will be referred to as BUILDDIR throughout the remainder of this guide.

First, initialize your source to the correct version of Halium, depending on your device. If in doubt, refer to *Android and Halium versions*.

For Halium 9.0:

```
repo init -u https://github.com/Halium/android -b halium-9.0 --depth=1
```

With the Halium tree initialized you are ready to download the code by issuing the command:

```
repo sync -c -j 16
```

The download will take some time as it counts several gigabytes. If you have a fast internet connection, you may set an extra JOBS=[number] environment variable at the beginning of the command to open more parallel downloading jobs. Generally, 12 is recommended, which is the default. When it completes, your BUILDDIR will contain a copy of the Halium source tree, but important parts are still missing.

Adding your device-specific source

The next step is to add the device-specific sources that need to be integrated into the source tree before you can attempt to build. The missing sources are those required to build the kernel as well as a host of other hardware-specific components of the port.

The necessary sources need to be located and specified by creating a device manifest file (or editing an existing one) in the directory `BUILDDIR/halium/devices/manifests`.

Note: A correct device manifest is crucial to the success of your port.

Locating the sources

Locate your device repository on LineageOS's GitHub organization. This is done by typing your device's codename into the search box. The device repository follows the naming convention: android_device_[manufacturer]_[device]. Make a note of this name.

Open the device repository on Github. It will contain a `lineage.dependencies` (or `cm.dependencies`) file which specifies all other repositories that your device is reliant upon.

Creating the device manifest file

Now create (or edit) the file `BUILDDIR/halium/devices/manifests/[manufacturer]_[device].xml`. (The description below presupposes that you are creating the file from scratch.)

Paste the following into the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>

</manifest>
```

The device repository

Your sources must now be entered into the manifest. Start with your device repository. Between the `<manifest>` and `</manifest>` tags, create the line below, making sure to replace the information inside the square brackets with your own:

```
<project path="device/[manufacturer]/[device]" name="[repository name]" remote="[remote]
↪" revision="[revision]" />
```

'path' specifies the target path (subdirectory of your BUILDDIR) where the source code from repository 'name', which is found at 'remote', will be placed. The repository may contain a number of branches and it is important to secure the correct branch with 'revision='

Note: The revision attribute may be omitted if the default revision for this remote is the one that you wish to use.

If you are unsure of your remote, refer to *Remotes*.

Dependencies

Now create more lines like the previous one, using the `lineage.dependencies` (or `cm.dependencies`) file you found earlier in your device repository. This file lists all the remaining repositories needed to build for your selected device. Create one line for each entry in this file, using the following template:

```
<project path="[target_path]" name="[repository]" remote="[remote]" revision="[revision]
↪" />
```

The target path is found in the repository's name. The preceding "android" or "proprietary" is omitted and underscores are replaced with slashes. For example, android_device_lge_bullhead goes in device/lge/bullhead.

Vendor blobs

Vendor blobs go in the vendor/ folder of your BUILDDIR.

The first place to search for your device's vendor blobs is in 'TheMuppets' GitHub organization. Enter your device's codename into the search field and see if you get a match. If you are unable to find your device in TheMuppets, you will need to search elsewhere.

It may also be possible to extract the vendor blobs from the device itself by following instructions for your device on the LineageOS wiki as applicable.

Remotes

A remote entry specifies the name, location (fetch) prefix, code review server, and default revision (branch/tag) for the source.

You can create a remote by adding a remote tag to your manifest:

```
<remote  name="aosp"
    fetch="https://android.googlesource.com"
    review="android-review.googlesource.com"
    revision="refs/tags/android-7.1.1_r25" />
```

(Note: The above is merely an example and cannot be used as such for Halium-9.0)

Only the name, fetch, and revision attributes are required. The review attribute specifies a Gerrit Code Review server, which probably won't be useful for initial porting purposes.

For example, let's say that you have a bunch of repositories at https://github.com/SomeUserName/ and your desired branch name is los-16.0 in each. You would create a remote as follows and place it into your local manifest:

```
<remote name="sun"
    fetch="https://github.com/SomeUserName"
    revision="los-16.0" />
```

There are also some remotes available to you by default, though they differ between Halium versions. The following tables will help you identify these. See more information on these remotes can be found at the top of the file BUILDDIR/halium/.repo/manifests/default.xml.

The following remotes are available by default in Halium 9.0:

Remote Name	Remote Description, URL
github	LineageOS, https://github.com/
gitlab	Gitlab, https://about.gitlab.com/
aosp	Android Open Source Project, https://android.googlesource.com

If you do not specify a remote, aosp is assumed.

Sync and apply hybris patches

With your manifest now complete, go to the root of your `BUILDDIR` and issue the following command (replace DEVICE with your device's codename) to bring the device-specific source code into your source tree:

```
./halium/devices/setup DEVICE
```

This will first link your manifest from Halium devices to `.repo/local_manifests/device.xml` and then sync all repositories.

Once complete you can now run a script to apply hybris patches. These are mainly used for building the system image and can be omitted if your goal is to only build `halium-boot.img`. However, if you run into build errors, it might be worthwhile to try applying these patches all the same.

From the root of your BUILDDIR, run the following command:

```
hybris-patches/apply-patches.sh --mb
```

Document

After completing the relevant steps above (according to your chosen, create a pull request containing your device manifest on the Halium devices repository. Also, create a device page on the UBports page under devices. You should link the manifest on Halium devices in your UBports device page.

Halium-9.0 - Building

When doing a Halium 9.0 port, you first need to build `halium-boot.img`. This boot image can then either be combined with the GSI, or you can build your own, device-specific `system.img` as well.

Building the boot image (halium-boot.img)

Initializing the build environment

First you need to initialize the environment using the `envsetup.sh` tool included in your source tree. Enter your `BUILDDIR` and issue the command:

```
source build/envsetup.sh
```

The output should resemble this:

```
including device/lge/bullhead/vendorsetup.sh
including vendor/cm/vendorsetup.sh
including sdk/bash_completion/adb.bash
including vendor/cm/bash_completion/git.bash
including vendor/cm/bash_completion/repo.bash
```

The `breakfast` command is used to set up the build environment for a specific device. From the root of your `BUILDDIR` run the following command, replacing [CODENAME] with your device's specific codename:

```
breakfast [CODENAME]
```

Breakfast will attempt to find your device, set up all of the environment variables needed for building, and give you a summary at the end.

Modify the kernel configuration

The default LineageOS/Halium kernel configuration file needs modifications as Ubuntu Touch requires a slightly different kernel config than Halium, including enabling Apparmor. A script that does this job is provided in your Halium source tree: `BUILDDIR/halium/halium-boot/check-kernel-config`.

Locate your configuration file. It should be at `arch/arm/configs/<CONFIG>` or `arch/arm64/configs/<CONFIG>` depending on the architecture of your device. If you have trouble finding it, run `grep "TARGET_KERNEL_CONFIG" device/<VENDOR>/<CODENAME>/BoardConfig.mk` to determine the location.

Then, from the root of your `BUILDDIR`, run:

```
./halium/halium-boot/check-kernel-config path/to/my/defconfig -w
```

You may have to do this twice. It will likely fix things both times. Then, run the script without the `-w` flag to see if there are any more errors. If there are, fix them manually. Once finished, run the script without the `-w` flag one more time to make sure everything is correct.

Ubuntu Touch requires setting console=tty0

The halium-boot initramfs expects `/dev/console` to be a console device and will not start init if it is not available. This is commonly the case on recent devices, because they either have UART disabled or `console=` is not specified (null) by default. This can be fixed by supplying `console=tty0` as the last argument in the kernel cmdline. To achieve this, proceed as follows:

It should be done in the makefile named `BoardConfig.mk` (or `BoardConfigCommon.mk`) located in the root directory of your device tree, e.g. `~/halium/device/<vendor>/<model_codename>/BoardConfig.mk`

Add the following line:

```
BOARD_KERNEL_CMDLINE += console=tty0
```

If your makefile already includes a line beginning with `BOARD_KERNEL_CMDLINE`, you may add it just below that to keep things tidy.

Note: The above method, although the preferred one, may not work for some Samsung devices. The result will be that you cannot get access to the device through ssh after boot, and Unity 8 will not be able to start. If you run into this problem, you can specify the setting in your device's kernel config file instead. Add the following lines:

```
CONFIG_CMDLINE="console=tty0"
CONFIG_CMDLINE_EXTEND=y
```

Note: In rare cases the bootloader overwrites the kernel command line argument, rendering the setting above useless. This is the case for the Google Pixel 3a (sargo). To deal with this issue, replicate this commit.

Build

Halium will use the `mkbootimg` tool for creating the boot image. In most cases it is not on the local harddisk, so it can be built by issuing:

```
mka mkbootimg
```

Now build the `halium-boot.img` using the commands:

```
export USE_HOST_LEX=yes
mka halium-boot
```

Note: If you prefer `make` instead of `mka` you should set `-j[num]` for parallel building, which reduces build time. Replace `[num]` with the number of threads in your system plus 2.

Build errors

There are a number of known build errors which you might encounter, depending first of all upon how rigorous you have been in following the steps outlined, but you may also run into unforeseen issues. If your build fails at first, make sure you have done exactly as described, then go through the list of known errors in the Halium guide.

I your particular error is not listed, you will need to do some research of your own. If you end up here, know that there is a community of porters, developers and enthusiasts who might be able to help you. Refer to *Getting community help*.

Building the system image (system.img)

If you are using the **Halium-boot** method, you can skip this step. If you are following the **Full system image** method, this step is required.

To build the system image:

```
mka systemimage
```

It's likely that you will run into one or more errors when building the system image. A number of possible errors and known solutions are documented in the Halium guide. If yours is not listed, *seek community help*.

11.1.2 Standalone kernel method

If you are using the **Standalone kernel** method, then please pick the step below.

Halium-9.0 - Standalone kernel method

For some devices, Halium-9.0 (and later) ports can be built based on just the kernel source code.

This method makes use of scripts that download and prepare the kernel source, build the boot image and combine this together with other necessary components, thus creating the files needed which can subsequently be flashed to the device.

Setting up the build environment

Assuming you have already installed the tools specified in the section on *Setting up the build environment*, there are just a few more to add before your build environment is complete. Run this commmand to add them:

```
sudo apt install android-tools-mkbootimg bc bison build-essential \
ca-certificates cpio curl flex git kmod libssl-dev libtinfo5 python2 \
sudo unzip wget xz-utils img2simg jq
```

Obtaining kernel source

Locate the kernel source code for your device. Fork this repo to your own Github account so that you can easily make any necessary adjustments as your work progresses. Make a note of the link to your repo. It needs to be specified in a configuration script for the build (see *Configure, build and install*).

Clone a Gitlab CI repo to base your build on

Ideally you should base your build on a previous build for a device with similar platform/SoC. However, if unsure, use the repo for Xiaomi-violet

Configure, build and install

Go through the file `deviceinfo` and make all necessary adjustments so that it conforms to your device. The first lines should be self-explanatory.

To complete the `deviceinfo_flash_offset_` lines, you will need to obtain and examine an existing `boot.img` for your particular device, e.g. from a LineageOS ROM. Lineage provides a Python script for reading the information in question. Download this script and run it on the boot image you have obtained:

```
<path-to-unpack_bootimg.py> --boot_img <path-to-boot.img>
```

This tool will not supply the base offset, only the others, and it gives the absolute value. In other words, the value returned by the script for the kernel offset is in fact the (base offset + kernel offset). In the same manner, the other values are also the sum of the base offset and the offset value in question. Note that offset values for devices that share the same SoC will be the same.

Using this information, you can calculate the correct values for the offsets to be specified in the file `deviceinfo`. If in doubt, *seek help*.

When you have finished editing `deviceinfo`, carefully read and follow the instructions on this page to build and install, remembering to replace 'violet' with your device's codename.

11.2 Install and boot

Afterwards, proceed with these steps for any porting method.

11.2.1 Installing Halium-based builds

With the necessary compontents built, you are now ready to install them on your device and check whether they function as intended. There are only minor differences between Halium versions when it comes to the installation process. In all cases, the system image and rootfs are installed together on the userdata partition of the device. This is done for practical reasons, as it makes the installation process simpler to manage during the first phases of the porting process. Once the port reaches the level of maturity required for it to be offered through the UBports installer (see *Finalizing the port*), this must be changed in order to reserve a maximum of possible space for userdata.

In brief:

- **Halium-7.1** requires installing the boot image and system image you have built, together with the UBports rootfs, which you need to download.
- **Halium-9.0 and newer** require installing the boot image you have built together with the GSI and the UBports rootfs. Both the GSI and the rootfs are available for download. Note: If you chose to build your own system.img, then you must use this instead of the GSI when installing.

In order to install Ubuntu Touch, you need a recovery image with Busybox, such as TWRP, installed on your phone. If you have not done so yet, refer to section *Install TWRP recovery* and install it now.

You will also need to ensure the `/data` partition is formatted with `ext4` and is not encrypted. Boot into recovery, check and adjust as necessary.

Flashing halium-boot.img

To install halium-boot, reboot your phone into fastboot mode. Then do the following from the root of your `BUILDDIR`:

```
cout
fastboot flash boot halium-boot.img
```

If you have trouble accessing your device in fastboot mode, but are able to access it in TWRP recovery mode using adb, then boot into recovery mode and try the following alternative method:

```
cout
adb push halium-boot.img /tmp/
```

On your device, choose *Install* in the TWRP menu, navigate to the `/tmp` directory, choose *Image*, select your image file, select *Boot* and swipe to flash.

Note: Samsung devices: Flashing halium-boot.img on Samsung devices is done using the Heimdall flashing utility (on Linux) or the Odin utility (on Windows) after first bringing the device into 'download mode'. See more information on these utilities here. Follow the instructions for the utility you choose, including the specific flashing command for flashing the boot partition. Install system.img and rootfs (below) with the device in recovery mode.

Installing system image and rootfs

Read carefully and perform only the steps which apply to your Halium version and the files you will be flashing!

Download the appropriate rootfs

Start by downloading the appropriate rootfs for your device. You need a rootfs corresponding to your device's architecture and the Halium version you have built. At the moment there are two available versions for Halium 7.1, an armhf (32 bit) version and an arm64 (64 bit) version. For Halium 9.0 there is only an arm64 rootfs available. Follow the appropriate link below to download the one you need.

- Halium 7.1, armhf (32 bit): ubports-touch.rootfs-xenial-armhf.tar.gz
- Halium 7.1, arm64 (64 bit): ubuntu-touch-hybris-xenial-arm64-rootfs.tar.gz
- Halium 9.0, armhf (32 bit): ubuntu-touch-android9-armhf.tar.gz
- Halium 9.0, arm64 (64 bit): ubuntu-touch-android9-arm64.tar.gz

Halium-9.0: Download the GSI

If you have built your own system.img, skip to the next step.

Download the Halium 9 LXC container image (GSI).

Extract the downloaded file and locate the file `android-rootfs.img` in the directory `system/var/lib/lxc/android`. This is the GSI file you will be transferring to the device.

Download the halium-install script

Clone or download the halium-install repository. This repository contains tools that can be used to install a Halium system image and distribution rootfs. Reboot your device to recovery (e.g. TWRP), to get adb access again. Now use the `halium-install` script to install Ubuntu Touch on your device.

Perform the installation

For **Halium 7.1 ports** use the following command:

```
path/to/halium-install -p ut path/to/rootfs.tar.gz path/to/system.img
```

For **Halium 9.0 ports**, use this command:

```
path/to/halium-install -p ut -s path/to/ubuntu-touch-android9-arm64.tar.gz path/to/
↪[SYSTEM_IMAGE]
```

where [SYSTEM_IMAGE] will be the file `android-rootfs.img` you downloaded and extracted, or alternatively the file `system.img` you built yourself.

The script will copy and extract the files to their proper places, then allow you to set the user password for your device (the *phablet* user's password).

Note: halium-install might fail due to unconfigured bintfmt-support for qemu throwing errors such as 'chroot: failed to run command 'passwd': Exec format error'. To fix this, `/proc/sys/fs/binfmt_misc/register` should be updated. binfmt-manager by @mikkeloscar can provide quickfix.

11.2.2 Booting

In the previous sections you completed building and installing Ubuntu Touch on your device. The next step is to boot the device, gain remote access and debug as necessary.

What is a successful boot?

When you boot your build for the first time you will normally not get anything on the screen. This is not necessarily an indication of something gone wrong. Your system may well have booted successfully and this can be ascertained by gaining remote access.

If your system booted successfully, you will be able to connect to your device from your host using ssh and continue debugging and configuring from there.

If something went wrong, you should normally be offered a telnet connection through which you can proceed with debugging. If you don't get either of these, the problem might be more difficult to determine. If you get stuck don't hesitate to seek help.

Gaining remote access

When your device boots it will likely stay at the bootloader screen. However, you should also get a new network connection on the computer you have it plugged in to. To determine if this is the case, and what type of connection you have been offered, proceed as described below.

ssh connection

To confirm that your device has booted correctly, on your host computer, run dmesg -w and watch for "GNU/Linux device" in the output. This confirms that you can establish an ssh connection to the device (see below). If you instead get something similar to "Halium initrd Failed to boot", skip to *telnet connection*.

Establish an ssh connection to your device as follows:

Your computer should now have a newly connected RNDIS interface. Check the output of `ip link show` for the device name. The device name will most likely resemble `enp0s29u1u1`. Assign this device a fake MAC address (if the default one is all 00s) and an IP of 10.15.19.100:

```
ip link set <devicename> address 02 11 22 33 44 55
ip address add 10.15.19.100/24 dev <devicename>
ip link set <devicename> up
```

Once finished, you should be able to login with the following:

```
ssh phablet@10.15.19.82
```

The password will be the one you set when you ran the halium-install script.

telnet connection

If you have been offered a telnet connection the rootfs and system image have likely not been found and mounted at boot time. This may indicate that one or more partitions have not been mounted as they should. Connect to your device with the following command:

```
telnet 192.168.2.15
```

From here you can start debugging to find out what went wrong. Please *seek help* to find out how to proceed.

CHAPTER

TWELVE

CONFIGURING, TESTING AND FIXING

Now that you have successfully booted your build, you are ready to bring up the graphical UI, merge AppArmor into your port and see to any remaining functionality that needs configuring. This section explains how to perform these tasks.

If you are doing a GSI port, much, if not all device features, should already work 'out of the box' once you have brought up Lomiri and added AppArmor. For this reason, Halium-9/GSI porting as well as porting based on standalone kernel builds should normally require only udev rules, AppArmor patches and only limited, if any, configuring using the overlay file method. If, despite performing these tasks, your GSI-based build still does not function properly, you may have to resort to building a device-specific `system.img` in order to reach your goal.

On the other hand, Halium 7.1 porting and Halium-9.0 porting (when building the full device-specific `system.img`) will require the most work.

Start by bringing up Lomiri, adjusting the display as required, and adding AppArmor. The remaining steps can be performed in arbitrary order and some may not apply to your particular port.

As you go along, it can be helpful to have a list of device functions to go by when checking what works and what needs debugging and fixing. This list, along with instructions on how to add your device to the list of devices that can run Ubuntu Touch, can be found in the informative README.md file linked here.

12.1 Lomiri - the graphical UI

Now that you have gained access to your newly booted device, there remain a number of things to be configured before Ubuntu Touch will be fully functional on your device. The first is to add udev rules which are required for the graphical UI to be able to launch.

12.1.1 What are udev rules?

On your running system there is a continuously running udev daemon which governs how the system handles events on peripheral devices, *e.g.* connecting the device to a PC via USB cable. This daemon needs a set of rules, *the udev rules*, to tell it what to do for each possible event. These rules must observe a specific format and they are stored in a file which needs to be generated for each specific device.

12.1.2 Add udev rules

The correct way to do this is by storing these settings in overlay files (*Overlay file method*), and not by making the root directory writeable, as described below. However, in order to quickly set up udev rules which are necessary to complete subsequent steps, you can use the method detailed here for first time setup.

Make / (root) writable

Before you make any changes to the rootfs (which will be required for the next steps), you'll need to remount your root directory (/) with write permissions. Do this by running the following command:

```
sudo mount -o remount,rw /
```

Create and add udev rules

You must create some udev rules to allow Ubuntu Touch software to access your hardware.

If you are building a Halium-7.1 based port, run the following command, replacing [CODENAME] with your device's codename:

```
sudo -i # And enter your password
cat /var/lib/lxc/android/rootfs/ueventd*.rc|grep ^/dev|sed -e 's/^\/dev\///'|awk '
↪{printf "ACTION==\"add\", KERNEL==\"%s\", OWNER=\"%s\", GROUP=\"%s\", MODE=\"%s\"\n",
↪$1,$3,$4,$2}' | sed -e 's/\r//' >/usr/lib/lxc-android-config/70-[CODENAME].rules
```

For a Halium-9.0 based port you should use the commands below, again replacing [CODENAME] with your device's codename:

```
sudo -i # And enter your password
DEVICE=[CODENAME]
cat /var/lib/lxc/android/rootfs/ueventd*.rc /vendor/ueventd*.rc | grep ^/dev | sed -e 's/
↪^\/dev\///' | awk '{printf "ACTION==\"add\", KERNEL==\"%s\", OWNER=\"%s\", GROUP=\"%s\
↪", MODE=\"%s\"\n",$1,$3,$4,$2}' | sed -e 's/\r//' >/etc/udev/rules.d/70-$DEVICE.rules
```

Note: If you are building a Halium-9.0 based port for a non-treble device, i.e. a device without a separate vendor partition, the command above will give an error. Simply edit and remove the following string from the command: `/vendor/ueventd*.rc`.

Now, reboot the device. If all has gone well, you will eventually see the Ubuntu Touch spinner followed by Unity 8. Your lock password is the same as you set for SSH.

When Unity 8 is first brought up on your device, you will probably notice that everything is very small. The *Display settings* section describes how to deal with this.

12.2 Overlay file method

The UBports rootfs comes with a set of standard configuration files for a number of features such as display scaling, sound, bluetooth and more. These files may not tailored to the needs of your specific device and must therefore to be replaced in order for the feature in question to function as it should. This is done with overlay files, i.e. files that the original files get overwritten with. In other words, you need to rewrite the files in question, making the necessary adjustments for your device, and then see to it that these are incorporated into the build so that they will replace the originals.

If your build is based on the downloaded GSI and rootfs, installed using the halium-install script, then further adjustment of the build will not be possible. This is because the halium-script does not include any method for injecting overlay files. If, on the other hand, your build is based on Gitlab CI scripts, there is a way to further tweak your port with overlay files. This method is described at the end of this section.

12.2.1 Configuring features with overlay files

For full `system.img`-based builds, irrespective of Halium version, the method described below applies if the file you need to edit can be found in the /etc directory (or a subdirectory of this) on your device. In other words, the file or files you wish to overlay must actually exist and be located where specified above. You should not attempt to overwrite files located elsewhere using the method described here.

If your port is based on Gitlab CI scripts, it is actually possible to not only overlay existing files, but also introduce files that were not originally present. The method for Gitlab CI script based builds is detailed at the end of this section. If this is the method you will be using, you should still read both sections to gain a better understanding of how this method actually works.

Overlay method for full system.img builds

In your device directory, create a subdirectory named 'ubuntu'. Collect the files you wish to inject into your build in this directory.

Relevant files are for example (but this list is incomplete):

- 70-android.rules
- android.conf (for display scaling, see below)
- touch.pa (for pulseaudio sound configuration/initialization, see below)

These files are then injected by adding a code block to the file `device.mk` in your device directory. For the three files above add the following code:

```
### Ubuntu Touch ###
PRODUCT_COPY_FILES += \
    $(LOCAL_PATH)/ubuntu/70-android.rules:system/halium/lib/udev/rules.d/70-android.
↪rules \
    $(LOCAL_PATH)/ubuntu/android.conf:system/halium/etc/ubuntu-touch-session.d/android.
↪conf \
    $(LOCAL_PATH)/ubuntu/touch.pa:system/halium/etc/pulse/touch.pa
### End Ubuntu Touch ###
```

The first of these three files, 70-android.rules, is the one you created when bringing up Lomiri. You can extract this file from the device using `adb pull` or `scp` and copy it to the 'ubuntu' directory of your device source tree, making sure to add the corresponding line to your `device.mk` file, as described above.

Explanation:

The string before the colon '$(LOCAL_PATH)/ubuntu/70-android.rules' specifies the path to the source file to be injected. The string after the colon 'system/halium/lib/udev/rules.d/70-android.rules' specifies the target location on your device.

The general steps to follow are thus:

1. Copy the file you wish to modify to the 'ubuntu' directory you have created in your device source tree.
2. Edit the file as needed.
3. Add a line to the PRODUCT_COPY_FILES section of your device.mk file as shown above.
4. Rebuild your system.img and reflash together with the ubports rootfs.

Note: The target paths for the files mentioned above are *not* randomly chosen. You must use the specified paths.

Note: When you specify target path 'system/halium/etc/myfilename' your file 'myfilename' will end up in the '/etc' directory of your device (i.e. without the leading 'system/halium')

Rebuild system.img

For Halium-7.1 ports and Halium-9.0 ports using a device specific system.img in place of the GSI, the system.img now needs to be rebuilt.

When you have made the adjustments you need and prepared your source as described above, rebuild the system image: `mka systemimage`.

When rebuilding the system image after small changes like these, you need not `mka clean` first. However, changes to PRODUCT_PROPERTY_OVERRIDES might not get detected by the build system. To ensure this, go to the directory `BUILDDIR/out/target/product/CODENAME/system` and delete the file `build.prop` before rebuilding.

Overlay method for Gitlab CI script-based builds

Builds in this category can also be adapted to a particular device using overlay files, and when building by this method (see *Halium-9.0 - Standalone kernel method*) it is possible not only to replace existing files, but also to introduce new ones.

When you have prepared the files to replace or introduce, and determined the exact locations where they need to go on your device, these locations need to be replicated under the directory `overlay/system/` before running the Gitlab CI scripts according to the instructions included.

For example:

The udev rules file `70-android.rules` needs to go into `system/halium/lib/udev/rules.d/`. This is accomplished by first creating the directory in the Gitlab scripts build tree. Standing in the root of this tree, first create the directory and then copy the file to this location:

```
mkdir /halium/lib/udev/rules.d
cp <path-to-70-android.rules> /halium/lib/udev/rules.d/70-android.rules
```

Now run the build scripts and the file will get incorporated into your build. Flash the files as per the instructions.

12.3 Display settings

There are two variables that set the content scaling for Lomiri and Ubuntu Touch applications: `GRID_UNIT_PX` and `QTWEBKIT_DPR`.

There are also other options available that may be useful for you depending on your device's form factor. These are mentioned below and explained in depth *in the section on display settings*.

All of these settings are guessed by Unity 8 if none are set. There are many cases, however, where the guess is wrong (for example, very high resolution phone displays will be identified as desktop computers). To manually set a value for these variables, simply edit the file at `etc/ubuntu-touch-session.d/android.conf` specifying them. For example, this is the file for the Nexus 7 tablet:

```
$ cat /etc/ubuntu-touch-session.d/flo.conf
GRID_UNIT_PX=18
QTWEBKIT_DPR=2.0
NATIVE_ORIENTATION=landscape
FORM_FACTOR=tablet
```

The method for deriving values for these variables *is explained below.*

Once you have adjusted the `android.conf` file to the display settings needed for your device, this file should be incorporated into your build. Follow *the overlay file method* corresponding to your Halium version.

12.3.1 Determining the correct display settings

Display scaling

`GRID_UNIT_PX` (Pixels per Grid Unit or Px/GU) is specific to each device. Its goal is to make the user interface of the system and its applications the same *perceived* size regardless of the device they are displayed on. It is primarily dependent on the pixel density of the device's screen and the distance to the screen the user is at. The latter value cannot be automatically detected and is based on heuristics. We assume that tablets and laptops are the same distance and that they are held at 1.235 times the distance phones tend to be held at.

`QTWEBKIT_DPR` sets the display scaling for the Oxide web engine, so changes to this value will affect the scale of the browser and webapps.

A reference device has been chosen from which we derive the values for all other devices. The reference device is a laptop with a 120ppi screen. However, there is no exact formula since these options are set for *perceived* size rather than *physical* size. Here are some values for other devices so you may derive the correct one for yours:

Device	Resolution	Display Size	PPI	Px/GU	QtWebKit DPR
'Normal' density laptop	N/A	N/A	96-150	8	1.0
ASUS Nexus 7	1280x800	7"	216	12	2.0
'High' density laptop	N/A	N/A	150-250	16	1.5
Samsung Galaxy Nexus	1280x720	4.65"	316	18	2.0
LG Nexus 4	1280x768	4.7"	320	18	2.0
Samsung Nexus 10	2560x1600	10.1"	299	20	2.0
Fairphone 2	1080x1920	5"	440	23	2.5
LG Nexus 5	1080x1920	4.95"	445	23	2.5

Experiment with a few values to find one that feels good when compared to the Ubuntu Touch experience on other devices. If you are unsure of which is the best, share some pictures (including some object for scale) along with the device specs with us.

There are two other settings that may be of interest to you:

Form factor

`FORM_FACTOR` specifies the device's form factor. This value is set as the device's Chassis, which you can find by running `hostnamectl`. The acceptable values are `handset`, `tablet`, `laptop` and `desktop`. Apps such as the gallery use this information to change their functionality. For more information on the Chassis, see the freedesktop.org hostnamed specification.

Native orientation

`NATIVE_ORIENTATION` sets the display orientation for the device's built-in screen. This value is used whenever autorotation isn't working correctly or when an app wishes to be locked to the device's native orientation. Acceptable values are `landscape`, which is normally used for tablets, laptops, and desktops; and `portrait`, which is usually used for phone handsets.

12.4 AppArmor

Without AppArmor a number of device features will not function properly and most apps will crash when launched. For more information on AppArmor, refer to the Ubuntu Wiki.

AppArmor is added through a combination of integrating the necessary code into the kernel source tree and setting necessary kernel configuration. The process is slightly different on Halium-7.1, compared to newer Halium versions.

12.4.1 AppArmor for Halium-7.1 ports

Start by downloading the backported AppArmor patch corresponding to your device's kernel version. Your kernel version is specified at the beginning of your kernel defconfig file, i.e. the one you edited in section *Modify the kernel configuration* above. (For more information on backporting, see the *bluetooth* section below.)

You now need to delete your entire BUILDDIR/kernel/VENDOR/MODEL/security/apparmor subdirectory and replace it with the one you downloaded. Then rebuild `halium-boot.img`.

If you get errors when building, resolve them one at a time, modifying your source code as needed. Note that you should only modify the code in the AppArmor subdirectory if at all possible. Modifying code elsewhere will more than likely just compound your problems.

Seek help as needed from one of the sources mentioned in the subsection on *getting community help*.

Once you have successfully rebuilt `halium-boot.img`, flash it to your device. If keyboard vibration works, this is a good indication that AppArmor has been successfully applied. Also, check if apps launch normally.

12.4.2 AppArmor for Halium-9.0 ports

When doing Halium-9.0 (and later) ports, it is generally sufficient to cherry pick certain commits to the kernel source rather than replacing the whole `apparmor` directory in the kernel source tree. Follow the link below which corresponds to your device's kernel version and select the most recent block of commits (i.e. all the commits that were added on the same and most recent date):

- 3.18 AppArmor patches
- 4.4 AppArmor patches

- 4.9 AppArmor patches
- 4.14 AppArmor patches

Make sure your kernel defconfig has the setting:

```
CONFIG_DEFAULT_SECURITY="apparmor"
```

Now rebuild the boot image following your chosen build method, and flash it onto your device. Check if keyboard vibration works. This is a good indication that AppArmor has been successfully applied. Also, check if apps launch normally.

12.5 Wifi

For the time being, refer to the Halium porting guide.

12.6 Sound

Ubuntu Touch uses Pulseaudio as sound server. Documentation can be found at freedesktop.org and in the Ubuntu manpages

The default configuration file used on Ubuntu Touch is `touch.pa`. This file is located in the `/etc/pulse` directory on your device and it will need adjustment in order for sound to function properly. Extract the file and copy it to the `ubuntu` directory you created in your device repo (see *Overlay file method*).

Locate the line:

```
load-module module-droid-discover voice_virtual_stream=true
```

and replace it with this:

```
load-module module-droid-discover rate=48000 quirks=+unload_call_exit
```

At the end of the file, append this:

```
### Automatically load the audioflinger glue
.ifexists module-droid-glue-24.so
load-module module-droid-glue-24
.endif
```

Your modified `touch.pa` file now needs to be included in your build. Follow *the overlay file method* corresponding to your Halium version.

12.7 Bluetooth

Halium-7.1 porting to devices with kernels predating 4.2 will or may require backporting drivers from a newer kernel version, whereas Halium-9.0 based ports can skip this step.

12.7.1 Bluetooth backporting to older kernel versions

When porting to devices running older kernel versions (mainly version 3.x found in Android 7/ Halium-7.1 devices), it is necessary to replace the kernel bluetooth stack with a newer one. This is because the newer bluetooth hardware in today's bluetooth peripheral devices often has trouble talking to the older bluetooth drivers. This can be fixed by bringing in driver code from newer Linux kernel versions. The process is called *backporting*.

Backporting has been greatly facilitated by the Linux Backports Project which has existed for some time. This project is aimed at mainline Linux kernels and the tools (scripts) therein are not specifically tailored to Ubuntu Touch. They will consequently abort at some point during the process. However, they are the best option available, and can provide significant help all the same. The method below is based on the use of a version of these scripts which has been specially prepared by Canonical.

Note: Although there are other kernel versions besides v4.2 available, the backports script is specifically tailored to backporting from version 4.2 and thus effectively limits you to this option.

Bluetooth backporting steps

The steps are as follows:

1. Record bluetooth driver and settings
2. Download backports script
3. Download 4.2 kernel source
4. Run backports script and fix errors
5. Apply security patch
6. Apply new settings
7. Build and flash halium-boot.img
8. Build and flash system.img

These steps will bring in bluetooth driver source from the mainline 4.2 kernel and place it in a directory named `backports` in your device's kernel source tree. It will also modify Makefiles and Kconfigs as necessary, thereby disabling the original `drivers/bluetooth` directory of your kernel source. The mainline kernel may not contain all bluetooth drivers required for the device being ported. For this reason it is important to make sure to first record all necessary drivers, as any ones missing in the mainline kernel will have to be migrated from their original location (`drivers/bluetooth`) into the `backports/drivers/bluetooth` directory as described *later in this section* before rebuilding halium-boot.img.

Record bluetooth driver and settings

By the time you reach this point in the porting process, you will have completed building halium-boot (probably a number of times). Your kernel defconfig will contain bluetooth settings including one that designates the driver used by your device. These must be recorded before proceeding.

The experienced developer will likely be able to determine the relevant settings manually by searching through the defconfig file. Many of them will appear next to each other in one place in the file. Some may be spread elsewhere making them difficult to locate. When searching manually, help can be found by consulting the *Kconfig* files in relevant subdirectories of your kernel source tree.

If you do not have extensive experience, use the `menuconfig` tool instead, taking care to use it ONLY for reference, *i.e.* without making any changes.

Important: Modifications done with `menuconfig` will not affect your kernel defconfig file, but may still corrupt your build.

After completing a build of halium-boot.img:

```
cd out/target/product/[DEVICE]/obj/KERN_OBJ
ARCH=arm64 make menuconfig
```

(If your device is armhf, use `ARCH=arm` instead.)

Navigate to the bluetooth drivers submenu and note down all activated settings and what they do. Also note which other settings they depend on (found under Help).

Example:

> For the Samsung Galaxy S7 (herolte) the original defconfig file contains a number of CONFIG_BT settings, none of which actually designate the bluetooth driver used by this device. The setting for the driver itself is CONFIG_BCM4359=y. This was not one of the drivers brought in by the backporting steps below. It therefore had to be *added afterwards*.

Download backports script

Clone the backports scripts into a directory outside your halium source tree by issuing this command from your home (~) directory:

```
git clone https://github.com/ubuntu-phonedations/backports.git -b for-ubuntu backport-
↪scripts
```

This downloads the backports scripts prepared by Canonical based on the Backports Project mentioned above, and places them in the directory `~/backport-scripts`. The scripts are specifically written to backport from kernel version 4.2.

Download 4.2 kernel sources

Create a directory (outside your halium source tree) for the kernel source from which you will pull the newer drivers:

```
mkdir ~/kernel-backports
```

Now clone the kernel source for v4.2:

```
cd ~/kernel-backports
git clone https://kernel.googlesource.com/pub/scm/linux/kernel/git/next/linux-next -b v4.
↪2
```

Run backports script and fix errors

Navigate to your backports scripts directory and issue the command below (using Python2 as shown):

```
python2 ./gentree.py --copy-list ./copy-list --integrate --clean --git-revision v4.2 ~/
↪kernel-backports/linux-next ~/halium/kernel/[VENDOR]/[MODEL_NAME]
```

It is to be expected that there are errors during this step. You will then have to determine the cause, fix it and retry. The last error message concerns the Makefile and includes info about having generated a file named `Makefile.rej`, this means you will find information in this file about changes that did not complete successfully, but which you can apply yourself. These need to be completed before proceeding with the build.

Apply security patch

An additional generic security patch needs to be applied.

Apply new settings

Your kernel config file (defconfig) needs to be modified in order for the backported driver and protocol code to be activated.

Start by locating all lines beginning with `CONFIG_BT_` and move these to the end of the file. Collecting them there makes the subsequent steps somewhat easier by helping to keep track of the changes you make.

Next, deactivate all that are activated, *i.e.* do not have a leading `#`, by inserting this leading `#`. At the same time, for each one, add a corresponding one beginning with `CONFIG_BACKPORT_BT_`, *e.g.*:

```
CONFIG_BT=y
```

becomes:

```
#CONFIG_BT=y
```

and then insert the corresponding line for backports:

```
CONFIG_BACKPORT_BT=y
```

Now add these settings:

```
#Depending options for new stuff from backports
#CONFIG_CRC16=y
CONFIG_CRYPTO=y
CONFIG_CRYPTO_BLKCIPHER=y
CONFIG_CRYPTO_AES=y
CONFIG_CRYPTO_CMAC=y
CONFIG_CRYPTO_HMAC=y
CONFIG_CRYPTO_ECB=y
CONFIG_CRYPTO_SHA256=y
CONFIG_CRYPTO_USER_API=y
CONFIG_CRYPTO_USER_API_HASH=y
CONFIG_CRYPTO_USER_API_SKCIPHER=y
#CONFIG_TTY=y
```

At this point, check for any remaining settings you *recorded from your original defconfig*, which were dependent upon `CONFIG_BT=y` and have not been replaced by a corresponding `CONFIG_BACKPORT_BT_XXXX=y` setting, making sure not to forget your device's bluetooth driver. Such settings will no longer have any effect and must be pulled into the build in the following manner:

The corresponding source file(s) will have to be migrated from their original location to the corresponding location under `backports/drivers/bluetooth/`. The files `Makefile` and `Kconfig` need to be edited to include this missing setting or else they will not be built. Check the corresponding files in the original location for the necessary settings.

Once the above is complete, add the following lines and edit as necessary, following the directions below:

```
CONFIG_BACKPORT_DIR="backports/"
CONFIG_BACKPORT_INTEGRATE=y
# CONFIG_BACKPORT_KERNEL_3_5=y #disable for kernel > 3.4
# CONFIG_BACKPORT_KERNEL_3_6=y #disable for kernel > 3.4
# CONFIG_BACKPORT_KERNEL_3_7=y #disable for kernel > 3.4
# CONFIG_BACKPORT_KERNEL_3_8=y #disable for kernel > 3.4
# CONFIG_BACKPORT_KERNEL_3_9=y #disable for kernel > 3.4
# CONFIG_BACKPORT_KERNEL_3_10=y #disable for kernel > 3.10
# CONFIG_BACKPORT_KERNEL_3_11=y #disable for kernel > 3.10
# CONFIG_BACKPORT_KERNEL_3_12=y #disable for kernel > 3.10
# CONFIG_BACKPORT_KERNEL_3_13=y #disable for kernel > 3.10
# CONFIG_BACKPORT_KERNEL_3_14=y #disable for kernel > 3.10
# CONFIG_BACKPORT_KERNEL_3_15=y #disable for kernel > 3.10
# CONFIG_BACKPORT_KERNEL_3_16=y #disable for kernel > 3.10
# CONFIG_BACKPORT_KERNEL_3_17=y #disable for kernel > 3.10
# CONFIG_BACKPORT_KERNEL_3_18=y #disable for kernel = 3.18
CONFIG_BACKPORT_KERNEL_3_19=y
CONFIG_BACKPORT_KERNEL_4_0=y
CONFIG_BACKPORT_KERNEL_4_1=y
CONFIG_BACKPORT_KERNEL_4_2=y
CONFIG_BACKPORT_KERNEL_NAME="Linux"
CONFIG_BACKPORT_KERNEL_VERSION="v4.2"
CONFIG_BACKPORT_LINUX=y
CONFIG_BACKPORT_VERSION="v4.2"
CONFIG_BACKPORT_BPAUTO_USERSEL_BUILD_ALL=y
```

As an example, the lines above have been edited to conform with backporting from kernel 4.2 to a device with kernel version 3.18. For devices running lower kernel versions enable each line specifying a version above the device's kernel version by removing the leading `#` on these lines.

You are now ready to build.

Build and flash halium-boot.img

Return to the root of your BUILDDIR and build:

```
mka halium-boot
```

Build errors may occur and will vary depending on device. Handle them one at a time, *seeking help* as necessary.

After building and flashing halium-boot, check the output of `dmesg` on the device to see that bluetooth has been enabled:

```
dmesg | grep tooth
```

Your output should resemble the following (from the Samsung Galaxy S7):

```
phablet@ubuntu-phablet:~$ dmesg | grep tooth
[    2.219667] lucky-audio sound: moon-aif3 <-> lucky-ext bluetooth sco mapping ok
[    2.252591] Bluetooth: RFCOMM TTY layer initialized
[    2.252601] Bluetooth: RFCOMM socket layer initialized
[    2.252613] Bluetooth: RFCOMM ver 1.11
[    2.252626] Bluetooth: BNEP (Ethernet Emulation) ver 1.3
[    2.252631] Bluetooth: BNEP filters: protocol multicast
[    2.252639] Bluetooth: BNEP socket layer initialized
[    2.252646] Bluetooth: HIDP (Human Interface Emulation) ver 1.2
[    2.252654] Bluetooth: HIDP socket layer initialized
[    2.252661] Bluetooth: Virtual HCI driver ver 1.5
[    2.252736] Bluetooth: HCI UART driver ver 2.3
[    2.252743] Bluetooth: HCI UART protocol H4 registered
[    2.252749] Bluetooth: HCI UART protocol BCSP registered
[    2.252754] Bluetooth: HCI UART protocol LL registered
[    2.252760] Bluetooth: HCI UART protocol ATH3K registered
[    2.252765] Bluetooth: HCI UART protocol Three-wire (H5) registered
[    2.252771] Bluetooth: HCI UART protocol BCM registered
[    2.252876] Bluetooth: Generic Bluetooth SDIO driver ver 0.1
[    2.253388] [BT] bcm4359_bluetooth_probe.
[    2.253630] [BT] bcm4359_bluetooth_probe End
[    5.376110] [BT] Bluetooth Power On.
[    7.499943] [BT] Bluetooth Power On.
[    8.051620] [BT] Bluetooth Power On.
```

If you do not get similar output, something has gone wrong. Check that you completed all steps above as described and seek help as needed.

You have now rebuilt your halium-boot.img to include updated bluetooth drivers and only one final step remains.

Build and flash system.img

The system image needs to be rebuilt with a configuration script for bluetooth adapted to your device. On the completed build, this file is located at:

```
/etc/init/bluetooth-touch-android.conf
```

An example script can be found here. Make sure to adapt as necessary.

Place this script in your `device/[VENDOR]/[DEVICE]/ubuntu` directory and inject it using the *overlay file method*.

Rebuild and flash your `system.img`.

CHAPTER

THIRTEEN

FINALIZING THE PORT

Once most frequently used features are working on your port you can start thinking about finalizing it, i.e. building recovery and preparing the installer so that it is easier for people to install and test it. This can give valuable feedback that will potentially help you to find and diagnose remaining issues faster than you could have done on your own.

Previously, your port has had the rootfs and system image coexisting on the userdata partition. These need to be moved to the system partition in order to ensure a maximum of available space for user data. This is done by modifying the code, building `recovery.img` and subsequently rebuilding the boot image.

With the necessary components prepared, these should first be installed in their correct places manually for testing purposes. Once it has been confirmed that the port boots and works as it should, the final step is to prepare and test an installer config file that will permit the automation of the whole process.

13.1 Building UBports recovery

For the time being, Halium 7.1 porters should refer to the UBports porting notes on the UBPorts installer and System image.

For Halium-9.0, exact steps are not available at this time. Please *get in touch with the community for help.*

13.2 Configuring the UBports installer

For the time being, Halium-7.1 porters should refer to the UBports porting notes on the UBPorts installer and System image.

For Halium-9.0, exact steps are not available at this time. Please *get in touch with the community for help.*