UBports Documentation

Release 1.0

Marius Gripsgard

May 18, 2018

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 instructions how to install Ubuntu Touch on your mobile phone, user guides and detailed documentation on all 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

CHAPTER 1

Introduction

This is the documentation for the UBports project. Our goal is to create an open-source (GPL if possible) mobile operating system that converges and respects your freedom.

1.1 About UBports

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

After Canonical suddenly announced their plans to terminate support of Ubuntu Touch in April of 2017, UBports and its sister projects began work on the open-source code, maintaining and expanding its possiblilities for the future.

1.2 About the Documentation

This documentation is always improving thanks to the members of the UBports community. It is written in ReStructuredText and converted into this readable form by Sphinx, ReCommonMark, and Read the Docs. You can start contributing by checking out the *Documentation intro*.

1.3 Attribution

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

CHAPTER 2

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 our quality assurance depends heavily on the community, we try to track issues where the user would expect them, instead of seperated by repository. This means, that issues of almost all components that are distributed as with the system-image are tracked in the Ubuntu Touch tracker. An exception of this are click-apps that can be updated independently through the OpenStore.

Most other repositories track issues locally. If you're unsure whether a repository uses its own tracker or not, consult the README.md file. Repositories that don't track issues locally have their bugtracker disabled.

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

Note: Practicality beats purity! Exceptions might apply and should be described in the projects README.md file.

2.1.2 GitHub projects

To increase transparency and communication, GitHub projects (Kanban-Boards) are used whereever 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 that belong to a specific team or affect a specific device can be viewed.

These are the standard columns:

- **None (awaiting triage)**: The issue has been approved by a member of the qa team and is awaiting review from the responsible development team. if the issue is a bug, instructions to reproduce are included in the issue description. if the issue is a feature request, it has passed a primary sanity check by the qa-team but has not yet been accepted by the responsible development-team.
- **Accepted**: The issue has been accepted by the responsible development-team. If the issue is a bugreport, the team has decided that it should be fixable and accepts the responsibility. If the issue is a featrue request, the team thinks it should be implemented as described.
- **In Development**: A patch is in development. Usually means that a developer is assigned to the issue.
- **Quality Assurance**: The patch is completed and has passed initial testing. The QA team will now review it and provide feedback. If problems are found, the issue is moved back to "Accepted".
- **Release Candidate**: The patch has passed QA and is ready for release. In case of deb-packages that are included in the system-image, the patch will be included in the next over-the-air update on the rc channel, and, if everything goes well, in the next release of the stable channel.
- **None (removed from the project)**: If the issue is open and labeled "help wanted", community contributions are required to resolve the issue. If it's closed, that means that either a patch has been released on the stable channel (a comment on the issue should link to the patch) or the issue has been rejected (labeled "wontfix").

2.1.3 Labels

All issues - even closed ones - should be labeled to allow the use of GitHub's global filtering. For example, these are all of the issues labeled 'enhancement' inside @ubports. Consult the GitHub help pages for more information on search and filtering.

Here's a list of labels that are normally used by all repositories.

- **needs confirmation**: The bug needs confirmation and / or further information from affected users
- **bug**: This issue is a confirmed bug. If it's reproducable, 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 somewhere else. Please provide a link and close.
- **help wanted**: This issue is ready to be picked up by a community developer.
- **good first issue**: This issue is not critical and trivial to fix. It is reserved for new contributors as a place to start.
- **wontfix**: It does not make sense to fix this bug, since it will probably resolve itself, it will be too much work to fix it, it's not fixable, or an underlying component will soon change.

Additional special labels can be defined. As an example, here's the labels used in the Ubuntu Touch tracker:

- **critical (devel)**: This critical issue that only occurs on the devel channel is blocking the release of the next rc image.
- **critical (rc)**: This critical issue that only occurs 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.

- **device: [DEVICE CODENAME]**: This issue affects only the specified device(s).
- **team: [TEAM NAME]**: This issue is falls under the responsibility of a specific team (hal, middleware, UI).

Note: If a repository that tracks 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. See the *release-schedule* for more info.

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. The only difference is, that instead of the label **bug**, the label **enhancement** is used. The **needs confirmation** label is not applicable for feature requests.

- A *User* files a new bug using the issue-template.
- The *QA-Team* adds the label **needs confirmation** and tries to work with the user to confirm the bug and add potentially missing information to the report. Once the report is complete a **team-label** will be 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* will triage the issue and either reject (label **wontfix**, close and remove from the project) or accept the issue. The team decides if it they will fix the issue in-house (move to "Accepted" and assign a team member) or wait for a community developer to pick it up (Label **help wanted**, remove from the project board and provide hints on how to resolve the issue and further details on the how the fix should be implemented if necessary). For non-critical issues that are 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 they have something to show for, the issue is closed and automatically moved to "Quality Assurance" for feedback from the QA team. If neccessary, the developer will provide hints on how to test his 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", else it's moved to "Release Candidate" to be included in the next release.
- If not done already, the issue is added to the next milestone. Once the milestone is released, the issue is removed 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 deb-components of Ubuntu Touch. 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: `http://repo.ubports.com/dists/[branch name]`

Non-standard PPAs (i.e. not `xenial`, `vivid` or `bionic`) are kept for three months. In case they need to be kept for a longer time, a file with the name `ubports.keep` can be created in the root of the repository, containing the date until which the PPA shall be kept open in the form of **YYYY-MM-dd**. 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 conventions ensure that 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 possible deviations from the norm.

2.4.1 Click-Packages

Software that is 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. For marking and archiving milestones in development history, ideally Git tags or GitHub releases should be used.

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 us to make use of the powerful Debian build system to resolve dependencies.

Every repository that uses this convention will have branches for the actively supported Ubuntu releases referenced by their codenames (`bionic`, `xenial`, `vivid`, etc.). These are the branches that are 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 just 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 on the normal `xenial` dependencies, if that doesn't exist. These special dependencies are not built into the image but still pushed to on *repo.ubports.com*.

Multiple branch extensions can be chained together in the form of `xenial_-_dependency-1_-_dependency-2_-_dependen`
This means that 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 3

Install Ubuntu Touch

Installing Ubuntu Touch is easy, and a lot of work has gone in to making the installation process less intimidating to less technical users. 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.

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.

Go to the download page and download the version of the installer for your operating system:

- Windows: `ubports-installer-<version-number>.exe`
- macOS: `ubports-installer-<version-number>.dmg`
- Ubuntu or Debian: `ubports-installer-<version-number>.deb`
- Other Linux distribtuions: `ubports-installer-<version-number>.snap` or `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*.

3.1 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:
   - Bq E4.5 (*krillin*)
   - Bq E5 HD (*vegetahd*)
   - Bq M10 HD (*cooler*)
   - Bq M10 FHD (*frieza*)
   - Meizu MX4 (*arale*)
3. Download SP flash tool for Linux.

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
```

You will also need to use the latest version of the tool.

4. Extract the zip files
5. Start the tool with `sudo`
6. Select the `*Android_scatter.txt` file from the archive you downloaded in the first step as the scatter-loading file
7. Choose "Firmware Upgrade"
8. Double-check you chose "Firmware Upgrade" and not "Download" or "Format All"

> **Warning:** If you select DOWNLOAD 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.

9. Turn your device completely off, but do not connect it yet
10. Press the button labeled "Download"
11. Perform a final sanity-check that you selected the "Firmware Upgrade" option, not "Download"
12. 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.
13. Magic happens. Your device will boot into a super old version of Ubuntu Touch.

14. Congratulations! Your device will now boot into a very old version of Ubuntu Touch. You can now use the UBports Installer to proceed.

CHAPTER 4

Daily use

This section of the documentation details common tasks that users may want to perform while using their Ubuntu Touch device.

4.1 Run desktop applications

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

To display and launch applications you need the *Desktop Apps Scope* which is available in the Open Store. To install applications you need to use the commandline as described below.

4.1.1 Manage containers

Create a container

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

```
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`. Default is `chroot` and is compatible with every device. If the kernel of your device supports it then `lxc` is suggested.

The creating process can take some time, due to the size of the container (some hundred of megabytes).

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

List containers

To list all containers created run: `libertine-container-manager list`

Destroy a container

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

4.1.2 Manage applications

Once a container is set up, you can list the installed applications:

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

Install a package:

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

Remove a package:

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

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

4.1.3 Files

Libertine applications do have access to these folders:

- Documents
- Music
- Pictures
- Downloads
- Videos

4.1.4 Tipps

Locations

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

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

Shell access

To execute any arbitrary command as root inside the container run:

```
libertine-container-manager exec -c COMMAND
```

For example, to get a shell into your container you can run:

```
libertine-container-manager exec -c /bin/bash
```

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

To get a shell as user `phablet` run:

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

4.1.5 Background

A display server coordinates input and output of an operating system. Most Linux distributions today use the X server. Ubuntu Touch does not use X, but a new display server called Mir. This means that 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. A consequence of this is that the root filesystem is read only. Libertine provides a container with a read-write filesystem to allow the installation of regular Linux desktop applications.

4.2 Run android applications

Anbox is a minimal android container and compatibility layer that allows you to run android applications on GNU/Linux operating systems.

Note:

- Anbox is in early development
- Anbox for Ubuntu Touch is in even more early development
- Anbox only works on the 16.04 version of Ubuntu Touch, which is also in early development
- Only selected devices are supported for the moment, more will be added in the future.

4.2.1 Supported devices

Make sure your device is supported:

Meizu Pro 5 (codename: *turbo*, name of the boot partition: *bootimg*) BQ M10 HD (codename: *cooler*, name of the boot partition: *boot*) BQ M10 FHD (codename: *frieza*, name of the boot partition: *boot*)

You will need the device codename and the name of your boot partition for the installation.

4.2.2 How to install

Warning: Because this feature is in such an early stage of development, the installation is only recommended for experienced users.

- *Install* the 16.04/devel channel on your supported device
- Open a terminal and run `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.
- Activate developer mode on your device.
- Connect the device to your computer computer and run the following commands:

```
adb shell sudo reboot -f bootloader
wget http://cdimage.ubports.com/anbox-images/anbox-boot-$CODENAME.img
sudo fastboot flash $PARTITIONNAME anbox-boot-$CODENAME.img
sudo fastboot reboot
rm anbox-boot-$CODENAME.img
```

- wait for the device to reboot, then run:

```
adb shell
sudo mount -o rw,remount /
sudo apt update
sudo apt install anbox-ubuntu-touch
anbox-setup
exit
```

- Done! Select “Anbox” in the apps scope to use android applications. You might have to refresh the apps scope (pull down from the center of the screen and release) for the app to show up.

4.2.3 Reporting bugs

Please *report any bugs* you come accross. Bugs with Ubuntu Touch 16.04 are reported in the normal Ubuntu Touch tracker and issues with anbox are reported on our downstream fork. Thank you!

CHAPTER 5

Advanced use

This section of the documentation details advanced tasks that power users may want to perform on their Ubuntu Touch device.

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 these ramifications before hacking on your device too much!

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 here.

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 successfully. If not, continue to the next section.

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. For example:

- Connect the ubports device and the PC with a USB cable. Then copy the file using your filemanager.

- Or transfer the key via the internet by mailing it to yourself, or uploading it to your own cloud storage, or webserver, etc.
- You can also connect via *adb* and use the following command to copy it:

```
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 and adb connection to perform the following steps on your phone.

```
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:

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

5.2.4 Connect

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

```
ssh phablet@<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 Wifi

5.3 Switch release channels

5.4 Screen Casting your UT device to your computer

Ubuntu Touch comes with a command line utility called `mirscreencast` which dumps screen frames to a file. The idea here is to stream UT display to a listening computer over the network or directy trough adb so that we can either watch it live or record it to a file.

5.4.1 Using adb

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

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

NB: `timeout` here is used in order to kill process properly on device (here 120 seconds). Otherwise process still continuing even if killed on computer. You can reduce or increase frame per second with‘‘–cap-interval‘‘ (1 = 60fps, 2=30fps, . . .) and size of frames `384 640` means width=384 height=640

5.4.2 Via network

On receiver

For real time casting:

Prepare your computer to listen to a tcp port(here 1234) and forward raw stream to a video player (here mplayer) with a frame size of 384x640:

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

For stream recording:

Prepare your computer to listen to a tcp port(here 1234), ungzip and forward raw stream to a video encoder (here mencoder):

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

On device

Forward and gzip stream with 60fps (–cap-interval 1) and frame size of 384x640 to computer 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

This script allows you to screen cast remote UT device to your local PC (must have ssh access to UT and mplayer installed on PC), run it on your computer:

```
#!/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 -'
```

(continues on next page)

(continued from previous page)

```
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

Some users may not have an available wifi connection for their phone nor a data subscription from their mobile provider. This short tutorial will help you to connect your Ubuntu Touch to your computer to access internet.

Prerequisite: Phone is connected to the computer with usb and developer mode enabled.

5.5.1 Steps

1. On phone: `android-gadget-service enable rndis`
2. On computer: get your rndis ip address e.g:`hostname -I`
3. On phone:

   add gateway: `sudo route add default gw YOUR_COMPUTER_RNDIS_IP`

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

   refresh dns table: `resolvconf -u`
4. On computer:

   enable 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

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

5.6 CalDAV and CardDAV synchronization

CalDAV and CardDAV are protocols to syncronize calendars and contacts with a remote server. Many email-hosters provide a CalDAV and CardDAV interface.

Note: CalDAV Sync can also be set up in using the calendar app. Open the app, 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, there is no carddav implemention directly accessible from the Ubuntu Touch graphical user-interface, so the only way to sync carddav is by using syncevolution + cron. However, there is a simple way to do that with a script that you can run in the terminal or via phablet SSH connection. These instructions work for caldav as well.

1. Follow this guide to activate Developer Mode and ADB (or SSH) connection.
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 synchronisation)
- Line 61: write `sudo sh -c "echo '$COMMAND_LINE' > /sbin/sogosync"` , instead of `sudo echo "$COMMAND_LINE" > /sbin/sogosync`, to avoid permission denied error

3. Move the file to your UbuntuTouch device, either by file manager or with adb:

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

4. Connect with the phablet shell (`adb shell`) or directly on the phone Terminal app and type the following:

```
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 6

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.

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 Getting Logs

We appreciate as many good logs as we can get when you report a bug. In general, /var/log/dmesg and the output of /android/system/bin/logcat are helpful when resolving an issue. I'll show you how to get these logs.

To get set ready, follow the steps to *set up ADB*.

Now, you can get the two most important logs.

dmesg

The **dmesg** (*display message* or *driver message*) command displays debug messages from the kernel.

1. Using the steps you documented earlier, reproduce the issue you're reporting
2. cd to a folder where you're able to write the log
3. Run the command: *adb shell dmesg > UTdmesg.txt*

This log should now be located at UTdmesg.txt under your working directory, ready for uploading later.

logcat

The **logcat** (*log concatenator*) command displays debug information from various parts of the underlying android system.

1. cd to a folder where you're able to write the log
2. Run the command: *adb shell /android/system/bin/logcat -d > UTlogcat.txt*
3. Using the steps you documented earlier, reproduce the issue you're reporting

This log will be located at UTlogcat.txt in your current working directory, so you'll be able to upload it later.

6.1.5 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?
- Logs: Attach your logs by clicking and dragging them into your GitHub issue.
- 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 bug. 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!

6.2 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.

6.2.1 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.

6.2.2 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 neccessary, 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 reproducable). 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**.

6.2.3 Testing patches

Once a developer finished working on an issue, it's moved to the quality assurance column of the GitHub project. Check if the issue is still present in the latest update on the devel channel and try to find any problems it is causing. 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** or forward to **Release Candidate** as specified by the *issue tracking guidelines*.

6.2.4 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.

6.3 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.

6.3.1 Documentation guidelines

These rules govern *how* you should write your documentation to avoid problems with style, format, or linking. If you don't follow these guidelines, we will not accept your document.

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. This title is underlined with equals signs.

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.

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.

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.

Update translations

You must update the translation files to match your changes before you commit them. To do this, run `update-translations.sh` in this repository.

6.3.2 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`.

Remember to *update translations*, then commit your changes and submit a pull request.

6.3.3 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.

6.3.4 Current TODOs

This page 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!

Documentation TODOs

Todo: This is a workaround for ubports/ubports-boot #1 and ubports/ubports-boot #2. It should not be considered a permanent fix. Be prepared to revert this later when these bugs have been fixed.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/docsubportscom/checkouts/latest/porting/building-ubports-boot.rst, line 47.)

Todo: Change the rootstock link to point to UBports once the actuallyfixit PR is merged.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/docsubportscom/checkouts/latest/porting/installing-16-04.rst, line 29.)

Todo: This should be a little heavier on "What to do when something goes wrong" content.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/docsubportscom/checkouts/latest/porting/installing-16-04.rst, line 78.)

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

```
.. todo:

   My todo text
```

6.4 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.

There are two levels for this:

- A casual approach, as a translator volunteer.
- A fully committed approach as a UBports Member, filling in this application.

6.4.1 Tools for Translation

For everyone: A web based translation tool called Weblate. This is the recommended way.

- For advanced users: Working directly on .po files with the editor of your choice, and a GitHub account. The .po files for each project are in their repository on our GitHub organization.

A Translation Forum to discuss on translating Ubuntu Touch and its core apps.

6.4.2 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 you'll find there 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.

You decide how much time you can put into translation. From minutes to hours, everything counts.

.po file editor

As was said up above, you need a file editor of your choice and a GitHub account to translate .po files directly.

There are online gettext .po editors and those you can install in 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 files to put down a list here.

If you want to work with .po files directly you know what you're doing for sure.

6.4.3 Translation Team Communication

The straightforward and recommended way is to use 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?`

Just for your information, some projects are using Telegram groups too, and some teams are still using the Ubuntu Launchpad framework.

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

6.4.4 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.

6.5 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 7

App development

Welcome to an open source and free platform under constant scrutiny and improvement by a vibrant global community, whose energy, connectedness, talent and commitment is unmatched. Ubuntu is also the third most deployed desktop OS in the world.

7.1 Introduction

Ubuntu Touch has three types of applications: *Web applications* (WebApps), Scopes (deprecated) and *Native applications*. Applications are packaged, distributed and deployed using a format called click packaging. App UIs can be created using QML or HTML5 and behavior can be created using JS, Qt, C++, Python, or Go.

7.1.1 Click package overview

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-15.04",
    "maintainer": "xxxx <xxx@xxxx>",
    "hooks": {
        "myapp": {
        "apparmor": "apparmor.json",
        "desktop": "app.desktop"
        }
```

(continues on next page)

(continued from previous page)

```
    },


}
```

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

7.1.2 Security and app isolation

All Ubuntu apps and scopes 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": 1.3,
    "policy_groups": [
        "networking",
        "webview",
        "content_exchange"
    ]

}
```

Non exhaustive policy groups:

- accounts: Can use Online Accounts

- audio: Can play audio
- camera: Can access the camera(s)
- connectivity: Can access coarse network connectivity information
- content_exchange: Can request/import data from other applications
- content_exchange_source: Can provide/export data to other applications
- keep-display-on: Can request keeping the screen on (available since 15.04, OTA 5)
- location: Can access Location
- microphone: Can access the microphone
- networking: Can access the network
- push-notification-client: Can use push notifications as a client
- sensors: Can access the sensors
- usermetrics: Can use UserMetrics to update the InfoGraphic
- video: Can play video
- webview: Can use the UbuntuWebview

7.1.3 Ubuntu Touch platform

Platform key notes:

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

7.2 Getting started

Clickable is a meta-build system for Ubuntu Touch applications that allows you to compile, build, test and publish *click* packages and provides various templates to get you started with UT app developpment. It is currently the easiest and most convenient way of building click packages for Ubuntu Touch. Also, looking at existing published app source code is a good way to learn: see openstore

7.3 Ubuntu UI-Toolkit

Here you can find the API documentation for the Ubuntu UI toolkit.

- QML API
- Cordova HTML5 API

- Click packages

7.4 Ubuntu SDK IDE (unsupported)

The Ubuntu SDK IDE is no longer supported by Canonical, and UBports does not currently have the the time and manpower to get it to a working state.

It can still be installed in Ubuntu 16.04, but issues are expected.

```
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
```

7.5 Cookbook

A collection of external resources

7.5.1 App development cookbook

Unofficial resources

In this section you will find links to external resources about creating applications for Ubuntu Touch.

- Ubuntu Touch programming book

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 neccessarily 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

7.6 System Software

Working on system components

7.6.1 System Software

In this page you'll find information on how to develop system software, including how to build it, cross-compile it and make it available to other users. Most of the software preinstalled in your Ubports 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, Mint) and plenty of documentation is available out there on how to work with it, 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 Ubports package.

There are essentially three ways of developing Ubports system software: building it directly on device, cross-compiling it or uploading the source code to a Launchpad PPA and have it built by Launchpad. We'll examine all of the three methods, and use address-book-app (the Contacts application) as an example.

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; or, your device's RAM might not be enough for the compiler. Assuming that you are lucky enough not to run into these restrictions, and that you don't mind reflashing your device afterwards (to clear it from all the development packages you installed), please read on.

You can ssh to the device (via `phablet-shell`, for example) and then install all the packages needed to rebuild the component you want to modify (the Contacts app, in this example):

```
sudo apt-get build-dep address-book-app
sudo apt-get install fakeroot
```

This will install a bunch of packages into your device's rootfs. Additionally, you probably want to install `git` in order to get your app's source code in the device and later push your changes back into the repository:

```
sudo apt-get install git
git clone git@github.com:ubports/address-book-app.git   # or your own clone
cd address-book-app
```

Now, you are ready to build the package:

```
DEB_BUILD_OPTIONS="parallel=2 debug" dpkg-buildpackage -rfakeroot -b
```

and finally, install it. The `dpkg-buildpackage` command will print out the list of generated packages, and it's those filenames you will need to pass to the next command:

```
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 name ends with `-doc` or `dev`, since they don't contain code used by the device.

Cross-building with crossbuilder

This is the recommended way to develop non trivial changes, and it's suitable for all devices since the build happens on your desktop PC (will call it "host" from now on) and not on the target device. It's also extremely fast and easy to use.

Start with installing `crossbuilder` in your host:

```
git clone git@github.com:ubports/crossbuilder.git
```

It's 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 own session:
export PATH="$HOME/crossbuilder:$PATH"
```

Then, you need to setup LXD; luckily, crossbuilder has a command which does everything for you; you just need to carefully follow its instructions:

```
crossbuilder setup-lxd
```

If this is the first time you used LXD, you might need to reboot your host once everything has completed. After LXD has been setup, using crossbuilder is as easy as it can get: just move to the directory where the source code of your project is (for example, `~/src/git/address-book-app`) and launch it like this:

```
cd ~/src/git/address-book-app
crossbuilder --ubuntu=15.04
```

Note: if your device is connected to the PC, you don't need to specify the ''–ubuntu=15.04'' parameter because crossbuilder will figure out the proper Ubuntu version by itself. If you don't specify any parameter and have no device connected to your PC, crossbuilder will assume ''16.04'' (Xenial).

Crossbuilder will do everything for you: it will create the LXD container, download the development image, install all your package build dependencies, perform the build and finally, if your device is connected to your host, it will copy the packages over to the target and install them. 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, and the next time you'll type the `crossbuilder` command, only the changed files will be rebuilt. This makes iterative development blazing fast.

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 inside the container. You can do that by getting a shell inside the container with

```
crossbuilder --ubuntu=15.04 shell
```

and 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 into 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.

Developing in the host architecture, deploying via PPA

Another way to develop system software is to develop it locally on your desktop machine, and then push the source code to a Launchpad PPA and have it built there for the armhf architecture. Depending on whether the feature you

are developing can be reasonably tested in your local machine, and whether you can wait for Launchpad's builders to start working on your package (this could take some hours), this might or might not be a suitable way of device development.

Note: as of late 2017, Launchpad has dropped support for vivid-based PPAs. So, if you target the current 15.04 Ubports devices, this method is not suitable for you.

Start by getting VirtualBox and an Ubuntu image matching the base image of your device. You can get the Ubuntu image here:

- `16.04 (Xenial)`: http://releases.ubuntu.com/16.04/ubuntu-16.04.3-desktop-amd64.iso

Boot your VirtualBox machine with the Ubuntu image you downloaded, and once the installation is completed and you get to a terminal, add the Stable Phone Overlay PPA like this:

```
sudo add-apt-repository ppa:ci-train-ppa-service/stable-phone-overlay
sudo apt-get update
sudo apt-get dist-upgrade
```

You can then install the development tools you need, as well as the build dependencies of the component you want to work on:

```
sudo apt-get install vim git devscripts
sudo apt-get build-dep address-book-app
```

and then build the package locally:

```
DEB_BUILD_OPTIONS="parallel=4 debug" dpkg-buildpackage -rfakeroot -b
```

Change the `parallel` option according to how many processor cores you've made available to VirtualBox in order to amximize the build speed. The command above will build your package and also run all unit tests associated with it, so it's an easy (though not sufficient!) way to check that your changes won't break existing functionality. You can now develop your changes and test them locally (though, if your component needs access to phone hardware, that will obviously not work), until you are satisfied with the result.

Once you get to a state where you believe that your changes should work, you can push them into a PPA, so that they'll be built for your Ubports device and you (and other users) will be able to test them. First, create a PPA by visiting https://launchpad.net/~/+activate-ppa; enter a name and a description, then push the Create button, and on the next page pick the "Change details" link near the upper right corner. You can then enable your phone's architecture (with most likelihood, it's "ARM ARMv7 Hard Float"), disable all the architectures you don't care about, and click on "Save". Supposing that your Launchpad username is "ubdeveloper" and the PPA is called "myppa", then the commands to push your changes to the PPA will be as follows:

```
debuild -S
dput ppa:ubdeveloper/myppa ../address-book-app_*_source.changes
```

where the exact filename of the `.changes` file will be printed by the debuild command near the end of its output. Note that in order for the upload to succeed you will need to have a valid GPG key setup, and it must be added to Launchpad. If you are new to this stuff, it's recommended that you carefully read the documentation in Launchpad.

After the package has been uploaded, you should receive an e-mail by launchpad telling you whether the upload was accepted; if it was, then it means that Launchpad will try to build the source package for all the architectures supported by your PPA and, if successful, will finally publish the resulting package(s) in it. Now all what is left to do is to install the packages in your phone: to accomplish that, you can use `phablet-shell` to get access to your phone, and from there type the following commands:

```
sudo add-apt-repository ppa:ubdeveloper/myppa
sudo apt-get update
sudo apt-get install <your new package(s)>
```

You can also give the same installation instructions to other community members, if you want them to test your changes before submitting them upstream for review.

7.7 Web applications

Ubuntu webapps are web-hosted sites displayed inside an Ubuntu app container. They are true apps that users install, see, launch and use. But their content is provided through URLs.

7.7.1 Web applications

Ubuntu Webapps are a great 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. Webapps for converged Ubuntu

Guide

This guide targets webapps for converged Ubuntu, that is, Ubuntu for Devices (phones and tablets). The Ubuntu Desktop has additional webapps support not covered here. Support for webapps on converged Ubuntu will continue to grow, and of course the future of Ubuntu is convergence, so stay tuned.

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.

HoweverBoth 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 asterix 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 web application

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
```

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::

```
ubuntu-app-launch webbrowser-app --inspector
```

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

7.8 Native applications

7.8.1 Native applications

CHAPTER 8

Porting information

Note: If you are looking for information on installing Ubuntu Touch on a supported device, or if you would like to check if your device is supported, please see this page.

This section will introduce you to some of the specifics of porting Ubuntu Touch to an Android device.

Before you begin, you'll want to head over to the Halium porting guide and get your `systemimage` built without errors. Once you're at the point of installing a rootfs, you can come back here.

Start at *Building ubports-boot* if you'd like to install the UBports Ubuntu Touch 16.04 rootfs.

8.1 Building ubports-boot

Ubuntu Touch uses Upstart rather than Systemd for its init daemon. Because of this, it is not fully Halium-compatible and is not able to use the vanilla hybris-boot that Halium produces. For this reason, we need to build ubports-boot.

8.1.1 Add source to Halium tree

The first thing that needs to be done is adding the ubports-boot source to your Halium tree. You may choose to do this by adding it to your local manifests (recommended) or simply cloning it in place.

Add to local manifest

Create a new file in `.repo/local_manifests` called `ubports-boot.xml`. Add the following to the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
    <project name="ubports/ubports-boot" path="halium/ubports-boot" remote="hal"␣
↪revision="master" />
</manifest>
```

Now, just do a `repo sync` to download the new source.

Manual clone

You may also choose to clone the ubports-boot repository into your tree manually. It should be placed into `BUILDDIR/halium/ubports-boot`.

If you share your code and build instructions, please note that you've done this.

8.1.2 Include in build system

The android build system won't know where to find ubports-boot normally. To fix this, open the file `BUILDDIR/build/core/main.mk` in a text editor. Find the comment `# Specific projects for Halium` and add `halium/ubports-boot` to the list of `subdirs`

```
subdirs += \
        halium/hybris-boot \
        halium/ubports-boot
```

8.1.3 Fix mounts

Todo: This is a workaround for ubports/ubports-boot #1 and ubports/ubports-boot #2. It should not be considered a permanent fix. Be prepared to revert this later when these bugs have been fixed.

The initramfs of the ubports-boot image takes a commandline parameter, `datapart=`, which points it to the block device that is normally mounted at `/data` in Android. This parameter is automatically embedded into the image at build time by finding `/data` in your device's default fstab and using its source. Unfortunately, this source is generally a node in `/dev/block/bootdevice/by-name` or something else strange, which is only available in an Android initramfs. To fix this, Halium uses the fixup-mountpoints script, which you are probably familiar with by now.

The ubports-boot initramfs does not use the fixup-mountpoints script, nor does it put block devices in the proper place for mounting without modification. For this reason, we are going to have to edit your device's fstab.

The first step to this process is figuring out where your fstab actually is. For most, this is inside `BUILDDIR/device/MANUFACTUER/CODENAME/rootdir/etc` and it is named either `fstab.qcom` or `fstab.devicename`. Open the file for editing.

With the file open, change the `src`, or first attribute, of the `/data` mountpoint to what you put in fixup-mountpoints for `/data`, but without the `/block` folder. For example, on the Moto G5 Plus, the block device is `/dev/block/mmcblk0p54`, so I put `/dev/mmcblk0p54` in this place. Also change the type to ext4, if it is not already.

Now, remove all `context=` options from all block devices in the file. Save and exit.

For an example of this, see this commit on universalsuperbox/android_device_motorola_potter. Removing the `wait` flag is not required and was an accident.

8.1.4 Edit init.rc

Some Android services conflict with ofono in 16.04 and will cause your device to reboot without warning, about 30-60 seconds after it boots. We will need to disable these services until the issue is resolved.

To do this, open up your device's default `init.rc` (this is likely init.qcom.rc or init.[codename].rc), comment out any `import` statements, and add `disabled` to services like rild, qti, and others that interface with the

radio. Most of them have a `user radio` line. For an example, see commit 7875b48b on UniversalSuperBox/android_device_motorola_potter

8.1.5 Edit kernel config

Ubuntu Touch requires a slightly different kernel config than Halium, including enabling Apparmor. Luckily, we have a nice script for this purpose, `check-kernel-config`. It's included in the ubports-boot repository. Simply run it on your config as follows:

```
./halium/ubports-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.

8.1.6 Build the image

Once ubports-boot is in place, you can build it quite simply. You will also need to rebuild system.img due to our changes.

1. `cd` to your Halium BUILDDIR
2. `source build/envsetup.sh`
3. Run `breakfast` or `lunch`, whichever you use for your device
4. `mka ubports-boot`
5. `mka systemimage`

8.1.7 Continue on

Now that you have ubports-boot built, you can move on to *Installing Ubuntu Touch 16.04 images on Halium*.

8.2 Installing Ubuntu Touch 16.04 images on Halium

Warning: These steps will wipe **all** of the data on your device. If there is anything that you would like to keep, ensure it is backed up and copied off of the device before continuing.

Now that you've *built ubports-boot*, we'll use a script called `rootstock-touch-install` to install an Ubuntu Touch rootfs on your device.

In order to install Ubuntu Touch, you will need a recovery with Busybox, such as TWRP, installed on your phone. You will also need to ensure the /data partition is formatted with ext4 and does not have any encryption on it.

8.2.1 Install ubports-boot

We'll need to install the ubports-boot image before installing an image. Reboot your phone into fastboot mode, then do the following from your Halium tree:

```
cout
fastboot flash boot ubports-boot.img
```

8.2.2 Download the rootfs

Next we'll need to download the rootfs (root filesystem) that's appropriate for your device. Right now, we only have one available. Simply download `ubports-touch.rootfs-xenial-armhf.tar.gz` from our CI server. If you have a 64-bit ARM (aarch64) device, this same rootfs should work for you. If you have an x86 device, let us know. We do not have a rootfs available for these yet.

8.2.3 Install system.img and rootfs

Todo: Change the rootstock link to point to UBports once the actuallyfixit PR is merged.

Download the rootstock-touch-install script from universalsuperbox/rootstock-ng. Boot your device into recovery and run the script as follows:

```
rootstock-touch-install path/to/rootfs.tar.gz path/to/system.img
```

The script will copy and extract the files to their proper places, then allow you to set the phablet user's password. If it gets all the way to `rebooting device` and doesn't seem to produce any errors, it's time to continue to the next step. If something goes wrong, please get in touch with us. If your device doesn't reboot automatically, reboot it using your recovery's interface.

If you get errors similar to `broken pipe` or `out of memory`, use the `-b` option to push the busybox or toybox build that came from your tree. These may have fewer bugs than your recovery's busybox. More information about this option is available in the script's README.

8.2.4 Get SSH 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. We will use this to debug the system.

To confirm that your device has booted correctly, run `dmesg -w` and watch for "GNU/Linux device" in the output. If you instead get something similar to "ubports initrd i hit a nail", please get in contact with us so we can find out why. You may also choose to run `watch ip link` and look for changes in network devices.

Similar to the Halium reference rootfs, you should set your computer's IP on the newly connected RNDIS interface to `10.15.19.100` if you don't get one automatically. Then, run the following to access your device:

```
ssh phablet@10.15.19.82
```

The password will be the one that you set while running rootstock.

8.2.5 Make / writeable

Before we make any changes to the rootfs (which will be required for the next step), you'll need to remount it with write permissions. To do that, run the following command:

```
sudo mount -o remount,rw /
```

8.2.6 Add udev rules

Now that you're logged in, you must create some udev rules to allow Ubuntu Touch software to access your hardware. 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
```

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.

8.2.7 Continue on

Congratulations! Ubuntu Touch has now booted on your device. Move on to *Running Ubuntu Touch* to learn about more specific steps you will need to take for a complete port.

Todo: This should be a little heavier on "What to do when something goes wrong" content.

8.3 Running Ubuntu Touch

8.3.1 Display settings

When the device boots, you'll probably notice that everything is very small. There are two variables that set the content scaling for Unity 8 and Ubuntu Touch applications: `GRID_UNIT_PX` and `QTWEBKIT_DPR`.

There are also some other options available that may be useful for you depending on your device's form factor. These are discussed below.

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 create a file at `/etc/ubuntu-session.d/[codename].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
```

Methods for deriving values for these variables are below.

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.

Form factor

There are two other settings that may be of interest to you.

`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` 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.