Elektrobit

# Getting started with EB corbos Linux – built on Ubuntu

Publication Date: 2024-04-10

## Contents

# 1 Introduction

EB corbos Linux – built on Ubuntu is an open-source operating system for high-performance controllers, leveraging the rich functionality of Linux while meeting security and industry regulations.

The open source nature of EB corbos Linux makes it simple to use, understand, and develop, while receiving security updates and issue resolutions during the complete life cycle of the automotive ECU.

This document provides guidance in topics such as application development, image provisioning, and software packaging.

## 1.1 Key features

EB corbos Linux – built on Ubuntu (from now on referred to as EB corbos Linux) comes with the following key features:

**Easy start**

EB corbos Linux is a package-based Linux distribution, leveraging a variety of available open source software. You can provision your targets in a matter of minutes from Elektrobit's online package repositories without recompiling and rebuilding everything.

**Ready to run**

EB corbos Linux includes automotive features such as event management, a logging system, configurable root file system initialization, as well as tooling for development and image provisioning. These features are available as optional software packages together with ready-to-run preset examples of images for targets such as QEMU or Raspberry Pi.

**Simple customization**

EB corbos Linux tooling is designed for local modification and testing as well as for publishing those modifications as new custom packages. It enables precise tailoring of the system to specific needs and specifications.

**Seamless integration**

Containerized workloads that enable software compilation, building, or packaging can be seamlessly integrated into any existing CI/CD pipeline.

**Security**

EB corbos Linux comes with regular security updates and seamless integration with prevalent security management tools to safeguard against data breaches and cyberattacks.

**Scalability**

EB corbos Linux can be adapted to various embedded projects without having to create a unique distribution for each individual project.

# 2 How to use EB corbos Linux

This section guides you through the first steps with EB corbos Linux.

All tools used by EB corbos Linux are open-source, and detailed documentation is available online for most of them. This manual refers to the open source documentation wherever appropriate.

To work with EB corbos Linux, we recommend a native Ubuntu 22.04 environment, but as the EB corbos Linux SDK is based on a Docker container, any other environment that supports the execution of Docker containers should work.

This section guides you through the first steps with EB corbos Linux.

All tools used by EB corbos Linux are open-source, and detailed documentation is available online for most of them. This manual refers to the open source documentation wherever appropriate.

## 2.1 The EB corbos Linux SDK

The EB corbos Linux SDK is an Amazon Machine Image (AMI) that can be used via the VS Code Remote Development extension, and provides all tools to provision EB corbos Linux images and to (cross-)compile C/C++ applications for the usage in EB corbos Linux images.

### 2.1.1 Getting the EB corbos Linux SDK

Our recommended way of working with the EB corbos Linux SDK is using it as a VS Code dev container.

As prerequisites, you need to have the following tools set up:

- VS Code (https://code.visualstudio.com/) configured with the Remote Development extension

To set up the workspace, follow these steps:

1. Subscribe to the EB corbos Linux SDK product in the AWS Marketplace (https://aws.amazon.com/marketplace/pp/prodview-x3jf75g6hcqts)
2. Launch an EC2 instance from the subscribed EB corbos Linux SDK product. In case you create a key pair in this process, note down the location of the private key file. It will be required later.
3. Retrieve the EC2's public IPv4 address from the AWS console. It will be required later.
4. On your local PC create a `config` file under your SSH `home` (e.g. `C:\users\<your_user>\.ssh` for Windows, or `/home/<your_user>/.ssh`) and put the following contents:

```
Host <your_machine_name>
    HostName <IP address of the EC2 machine>
    User ubuntu
    IdentityFile <path to your ssh key for the EC2 machine>
```

1. Start VS Code and connect to the remote machine by pressing **Ctrl** + **Shift** + **P** and writing **Remote-SSH:Connect to Host**. You should be able to select <**your_machine_name**> from the configuration file above.

Note

The public IP address of the EC2 instance will change on every start/reboot. So the **config** file has to be adapted as soon as the public IP changes.

### 2.1.2 Structure of the EB corbos Linux SDK workspace

The EB corbos Linux SDK workspace has the following structure:

**apps**

The source code of our example applications

**apt**

The workspace specific local APT repository

**bin**

The workspace specific scripts and binaries

**gpg-keys**

Tooling and GPG keys for signing packages

**images**

Example image descriptions

**result/app**

Application build result

**result/image**

Created images

**sysroot_aarch64**

Sysroot for building the application for an aarch64 SoC

**sysroot_x86_64**

Sysroot for building the application for an x86_64 SoC

Note that some folders might only be populated after you run commands in the EB corbos Linux SDK container.

### 2.1.3 Commands provided in the EB corbos Linux SDK container

The EB corbos Linux SDK container provides the following commands, which can be used in VS Code tasks:

**box_build_image.sh <path to appliance XML>**

Use Berrymill to integrate an x86_64 image for the provided image description.

**cross_build_image.sh <path to appliance XML>**

Use Berrymill to integrate an aarch64 image for the provided image description.

**box_build_sysroot.sh <path to appliance XML>**

Use Berrymill to generate the x86_64 sysroot for the provided image description.

**cross_build_sysroot.sh <path to appliance XML>**

Use Berrymill to generate the aarch64 sysroot for the provided image description.

**cmake_x86_64.sh**

Use cmake to compile the application for the x86_64 sysroot environment.

**cmake_aarch64.sh**

Use cmake to cross-compile the application for the aarch64 sysroot environment.

**gen_sign_key.sh**

Generate a GPG key for Debian package and apt repository metadata signing. Please customize the data in the VS Code workspace folder `gpg-keys/env.sh` before running this command.

**prepare_deb_metadata.sh**

Generate the Debian metadata for the application.

**build_package.sh**

Build and package the application as an x86_64 Debian package.

**cross_build_package.sh**

Build and package the application as an aarch64 Debian package.

**build_config_package.sh**

Package configuration files as x86_64 and aarch64 Debian package.

**prepare_repo_config.sh**

Generate and sign the apt repository metadata for the Debian packages contained in the VS Code workspace folder *result/app*.

**serve_packages.sh**

Serve a local apt repository for local building.

**qemu_efi_aarch64.sh**

Run an aarch64 qcow2 image with enabled EFI boot in QEMU.

**qemu_efi_x86_64.sh**

Run an x86_64 qcow2 image with enabled EFI boot in QEMU.

### 2.1.4 EB corbos Linux SDK reference images

EB corbos Linux provides a choice of two init managers, *systemd* and *crinit*. The systemd init manager serves as the default choice for most server and desktop distributions, and it is commonly adopted in embedded Linux systems. Its widespread use is attributed to its robust feature

set and its ability to parallelize the system service startup process. The crinit init manager provides an embedded Linux relevant subset of systemd features such as parallel startup, but with a much smaller resource footprint. crinit is an Elektrobit-initiated open source project that is available as a GitHub repository (https://github.com/Elektrobit/crinit).

The EB corbos Linux SDK workspace contains the following example images:

**QEMU crinit**
: This image description is a Linux image that uses the crinit init manager. It is intended as a starting point for your own EB corbos Linux experiments.

**QEMU systemd**
: This image description is a Linux image that uses the systemd init manager.

**Raspberry Pi crinit**
: This image description is a Raspberry Pi 4 Model B image that uses the crinit init manager. It is intended as a starting point for your own EB corbos Linux experiments on hardware.

**Raspberry Pi systemd**
: This image description is a Raspberry Pi 4 Model B image that uses the systemd init manager.

**RDB2 crinit**
: This image description is a NXP RDB2 image that uses the crinit init manager. It is intended as a starting point for your own EB corbos Linux experiments on automotive grade hardware.

**RDB2 systemd**
: This image description is a NXP RDB2 image that uses the systemd init manager.

All example image descriptions contain network and APT repository configuration.

## 2.2 EB corbos Linux prebuilt images

We provide three prebuilt binary EB corbos Linux images, one for the Raspberry Pi 4 Model B and one for QEMU x86_64. These images are available for download at https://linux.elektrobit.com/images.

**QEMU image**
: This binary image was built using the QEMU crinit image description. Its contained in `qemu_image.zip`.

**Raspberry Pi image**

This binary image was built using the Raspberry Pi crinit image description. Its contained in `rpi4_image.zip`.

**RDB2 image**

This binary image was built using the RDB2 crinit image description. Its contained in `rdb2_image.zip`.

### 2.2.1 Running the QEMU image

To run the QEMU image using the EB corbos Linux SDK, follow these steps:

1. Download the archive from the Elektrobit server at https://linux.elektrobit.com/images/qemu_image.zip and extract the archive.
2. To run the image, you need QEMU, which is part of the EB corbos Linux SDK container. Extract the image and open the folder bind-mounted in the EB corbos Linux SDK container using the following command:

```
docker run --rm -it \
    -v <path to extracted image>:/build/img \
    linux.elektrobit.com/ebcl/sdk:latest
```

3. In the container, run QEMU using the commands:

```
cd img
./start_qemu_x86_64.sh
```

4. After a few seconds, a Linux root prompt appears. If it is hidden because it was scrolled away by logs of the system boot, press **Enter**. All example images use the username *root* and the password *linux*.

You can take a closer look at the demo.

To shut down QEMU, use the following command:

```
$ crinit-ctl poweroff
```

### 2.2.2 Running the Raspberry Pi 4 image

1. Download the archive from the Elektrobit server at https://linux.elektrobit.com/images/rpi4_image.zip, and extract the archive. In the extracted folder, the file named `raspberry_pi_crinit.aarch64-1.1.0-0.raw` is the Raspberry Pi 4 SD-card image.

2. Flash the image onto an SD-card. You can use a GUI tool such as balena Etcher (https://etcher.balena.io/), or you can use the `dd` command. If your SD-card is available as device `/dev/sdd`, you can use the following `dd` command to flash the image:

   ```
   $ sudo dd if=raspberry_pi_crinit.aarch64-1.1.0-0.raw of=/dev/sdd bs=64k oflag=dsync
    status=progress
   ```

3. Put the SD-card into a Raspberry Pi 4 to run EB corbos Linux. All example images use the username *root* and the password *linux*.

To shut down QEMU, use the following command:

```
$ crinit-ctl poweroff
```

### 2.2.3 Running the NXP RDB2 image

1. Download the archive from the Elektrobit server at https://linux.elektrobit.com/images/rdb2_image.zip, and extract the archive. In the extracted folder, the file named `rdb2_crinit.aarch64-1.1.0-0.raw` is the NXP RDB2 SD-card image.

2. Flash the image onto an SD-card. You can use a GUI tool such as balena Etcher (https://etcher.balena.io/), or you can use the `dd` command. If your SD-card is available as device `/dev/sdd`, you can use the following `dd` command to flash the image:

   ```
   $ sudo dd if=rdb2_crinit.aarch64-1.1.0-0.raw of=/dev/sdd bs=64k oflag=dsync
    status=progress
   ```

3. Configure the RDB2 board to boot from SD card, and put the SD-card into the slot to run EB corbos Linux. All example images use the username *root* and the password *linux*.

To shut down QEMU, use the following command:

```
$ crinit-ctl poweroff
```

## 2.3 Build your own images

For building images, we use Berrymill (https://github.com/isbm/berrymill/), an appliance generator of root file systems for embedded devices. Berrymill is a wrapper around Kiwi NG (https://osinside.github.io/kiwi), which is a flexible image builder that is also used by major Linux distributions.

Kiwi NG and Berrymill only support building images for the architecture of the build hosts. Kiwi NG utilizes host tooling for some steps during the build process, which makes the build results dependent on your host environment. To avoid this risk, we recommend to use the `box build` commands. These commands run the build process on a QEMU virtual machine that is provided by Elektrobit, which ensures reliable build results.

Kiwi NG is intended as an appliance builder for servers and desktops, while Berrymill enriches Kiwi NG with a structured way to provision images for x86_64 or aarch64 architectures. It also adds an inheritance mechanism to easily derive images. This provides a scalable approach for defining and maintaining image variations. For further details, refer to the Berrymill manpage (https://github.com/isbm/berrymill/blob/main/doc/manpages/berrymill.8.md).

### 2.3.1 Building a QEMU image

To build an image for QEMU from a reference description, follow these steps:

1. Open the EB corbos Linux SDK workspace and select a QEMU image reference description. You can build these descriptions for x86_64 or aarch64.
2. To build the image, open the build task menu (**Ctrl** + **Shift** + **B**) and select the `Image` build task for your chosen image description.

When the build is completed, you can find the result in the `result/image` folder of the VS Code workspace.

The build tasks make use of the `box_build_image.sh` command, which builds images for x86_64, and the `cross_build_image.sh` command, which builds images for aarch64.

To build your own image, you can either add a new task to `.vscode/tasks.json` providing the path to your appliance file, or open a terminal and run `box_build_image.sh <relative path to appliance XML in /build/img folder>` manually.

When the build task is finished, you can run the image by either selecting the **Run QEMU** task for the image description, or by using the terminal command `qemu_efi_aarch64.sh` or `qemu_efi_x86_64.sh`.

### 2.3.2 Building a Raspberry Pi 4 image

To build your own image for Raspberry Pi 4 Model B follow these steps:

1. Open the EB corbos Linux SDK workspace and select a Raspberry Pi 4 reference image description or your own image description file that also provides compatible kernel and firmware packages.
2. Run `cross_build_image.sh` with your image description provided as an argument.

After the build, you can find the result in the `result/image` folder.

When the build is finished, you can run the image as described in section *Section 2.2.2, "Running the Raspberry Pi 4 image"*.

### 2.3.3 Berrymill image creation

Berrymill is executing the following high-level steps to build an image:

1. Parse configuration options
2. Initialize repositories
3. Resolve image configuration from base image and derived images
4. Start Kiwi NG with appropriate arguments to build the image

For more details on the step of starting Kiwi NG, you can find more details in the next section. For the step of resolving the image configuration, see *Section 2.3.4, "The image description"*.

The apt repositories for downloading the packages during the Kiwi NG build are defined in the Berrymill configuration contained in `/etc/berrymill/berrymill.conf`. Repositories defined in the image descriptions are ignored. The EB corbos Linux SDK is configured to use the Elektrobit apt repositories.

#### 2.3.3.1 Kiwi NG image creation

For image provisioning, Kiwi NG uses the following high-level build steps:

1. Populate the root file system with selected packages
2. Apply the overlay tree

3. Run the configuration scripts
4. Pack the image

For populating the root filesystem, Kiwi NG uses the appliance description XML file. This file describes most aspects of the image and includes a list of packages that are added to the image. You can add additional packages or remove installed packages by modifying this file.

The step of applying the overlay tree copies all files from the `root` folder, next to the appliance description XML, to the image. Using this folder is the easiest way to get your own configuration files and applications into the image.

You can do advanced configuration using the `config.sh` script, which is also next to the appliance description XML. This script is executed in a chroot environment containing the unpacked image.

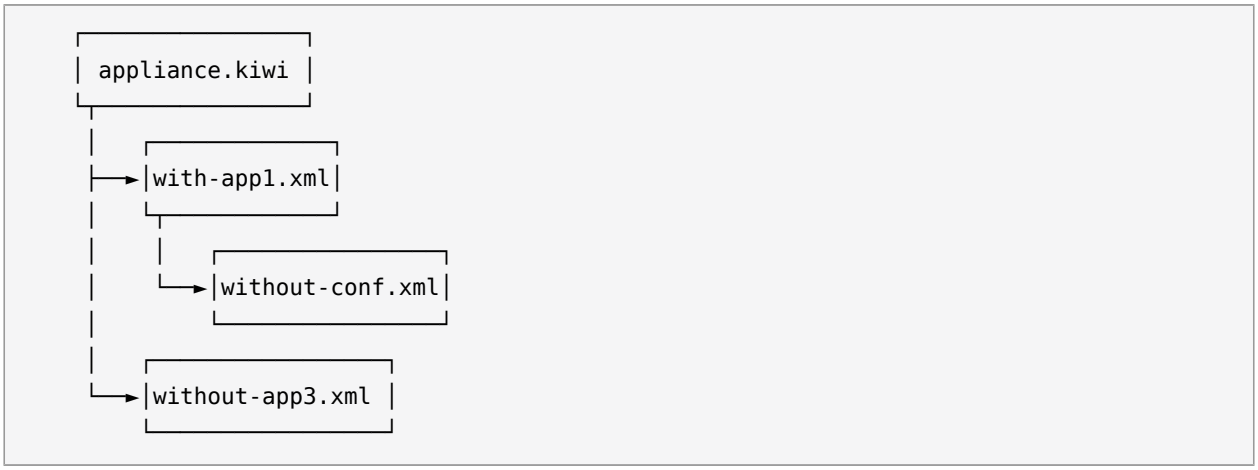
The `images` folder of the EB corbos Linux SDK workspace contains an example as reference.

For more details on image provisioning with Kiwi NG, refer to build workflow in Kiwi NG (https://osinside.github.io/kiwi/concept_and_workflow.html).

### 2.3.4 The image description

Berrymill implements the concept of derived images to extend Kiwi NG image descriptions with derivations like adding or removing packages, or changing the size or the type of a file system.

An example setup might be:




The `appliance.kiwi` file describes the base image, which can also be built on its own. It has the following basic structure:

```
<image schemaversion="7.4" name="image_name"> ❶
```

```
    <description type="system"> ❷
        <author>name</author>
        <contact>contact</contact>
        <specification>text</specification>
    </description>
    <preferences arch="arch"> ❸
        <version>major.minor.release</version> ❹
        <packagemanager name="packagemanager"/> ❺
        <type image="image_type"/> ❻
    </preferences>

    <packages type="image"/>
        <package name="name"/> ❼
        ...
    </packages>
</image>
```

❶ `image_name`: Name of the image

❷ `description`: Information about author, contact, license etc.

❸ `arch`: Architecture of the image, e.g. aarch64

❹ `version`: Version of the image

❺ `packagemanager`: The package manager used for building the image. In case of EB corbos Linux, this is always apt.

❻ `image`: Type of the image to be created. For embedded usage this is usually `oem`. For more details, refer to the Kiwi NG documentation about image types and results (https://osinside.github.io/kiwi/image_types_and_results.html).

❼ `packages`: A list of packages to be included in the image. Dependencies are automatically added using the package manager.

You can dive deeper by reading more about the image descriptions (https://osinside.github.io/kiwi/image_description/elements.html) on the Kiwi NG website.

A derived image can only inherit one base image description, but can be further derived by other descriptions. To inherit the `appliance.xml` description in `with-app1.xml`, xml inherits must be used:

```
<image schemaversion="6.0" name="app1_image">
 <inherit path="/path/to/appliance.xml"/>
</image>
```

In the derived image description, the following aggregates can be used to modify the base image description:

```
<add/>
<remove/>
<remove_any/>
<merge/>
<replace/>
<set/>
```

For further documentation, refer to the manpage of Berrymill (https://github.com/isbm/berrymill/blob/main/doc/berrymill.8.md).

### 2.3.5 Adding an existing package to a derived image

Since we now have an idea how image provisioning works and what an image description is, we can start defining our own derived images for example, by adding additional packages.

Let's start by adding vim to one of the reference images:

1. Get a reference image you want to use as a base and create a file `derived_image.xml` next to the basic description.
2. Add packages according to your needs. You can use the following content to add vim to the derived image:

```
<?xml version="1.0" encoding="utf-8"?>

<image schemaversion="6.8" name="Ubuntu-22.04_appliance">
    <inherit path="appliance.kiwi"/>

    <add>
        <!-- Add package to image !-->
        <packages type="image">
            <package name="vim"/>
        </packages>
    </add>

</image>
```

If you only want to use available packages, you are ready to build your own images.. If you also want to add your own application, read the next section.

## 2.4 Building and adding your own application

If you build an application for an embedded environment, you need to make sure that the build process is using the right versions of the headers and shared libraries. This is usually done by providing a sysroot to the compiler that contains the headers and libraries for the embedded environment. This is also true for EB corbos Linux, and the EB corbos Linux SDK provides commands to generate image-specific sysroot folders.

For Debian, there are often two packages for a given library. One package provides only the library binary file, and the other package provides the development assets such as headers. Since embedded environments are usually size-constrained, the development assets must not be part of the image but of the sysroot that is provided to the compiler.

An example for this split is *libjsoncpp*, which is used by the demo app that is included in the EB corbos Linux SDK workspace. To be able to use the library at runtime, we need to add the package `libjsoncpp25` to the embedded image. To be able to build an application using this library, we also need the package `libjsoncpp-dev`, which contains the development assets.

For EB corbos Linux we solve this by using the derived image feature of Berrymill. We add the line `<package name="libjsoncpp25"/>` to the image description and create the derived image `appliance_sysroot.kiwi` that also includes the development package:

```
<?xml version="1.0" encoding="utf-8"?>

<image schemaversion="6.8" name="appliance_dev">
    <inherit path="/tmp/img/appliance.kiwi"/>
    <!--
        Add libjsoncpp headers.
    -->
    <add>
        <packages type="image">
            <package name="libjsoncpp-dev"/>
        </packages>
    </add>
</image>
```

This allows us to run the prepare commands for `appliance_sysroot.kiwi`, which results in a sysroot that contains all files to compile applications using `libjsoncpp`.

### 2.4.1 Preparing an image-specific sysroot

The EB corbos Linux SDK provides the commands `box_build_sysroot.sh <path to appliance XML>` and `cross_build_sysroot.sh <path to appliance XML>` to build the sysroot. The path can be an absolute path in the EB corbos Linux SDK container or a path relative to `/build/img`, which is `workspace/images` in the host environment. The command `box_build_sysroot.sh` generates the sysroot for x86_64 in the folder `sysroot_x86_64`, and the command `cross_build_sysroot.sh` generates the sysroot for aarch64 in the folder `sysroot_aarch64`.

To initiate the build for the x86_64 sysroot using `appliance_sysroot.kiwi` in `workspace/images/qemu-crinit-x86_64`, you can either run the command `box_build_sysroot.sh qemu-crinit-x86_64/appliance_sysroot.kiwi` or the corresponding VS Code build task.

After the sysroot build, the folder `workspace/sysroot_x86_64` contains the sysroot for the selected image. This sysroot is used when compiling the application for x86_64.

### 2.4.2 Compiling the application

To compile a cmake application that is located in `apps/my-json-app` for an x86_64 environment, you can use the task **EBcL: App (x86_64)**, which executes the following commands:

```
cmake --toolchain /build/cmake/toolchain-x86_64.cmake -B /build/result_app/<build folder>
 -S /build/apps/my-json-app
cd /build/result_app/<build folder>
make
```

The cmake is initiated using the EB corbos Linux SDK cmake toolchain at `/build/cmake/toolchain-x86_64.cmake`, which ensures that the right compiler and sysroot is used. As build folder `/build/result_app` is used, which is `workspace/result/app` in the host environment. As last step make is invoked. The result of this sequence is a binary file for the x86_64 image environment described by `workspace/images/qemu-crinit-x86_64/appliance.kiwi`

Similar steps are needed to compile the application for an aarch64 image environment.

### 2.4.3 Adding the application to the image

To test the compiled binary file, we need to add it to the image. This can be done using the `root` folder of the image description. Copy the binary to `workspace/images/default/root/usr/bin`, then build the image as described before, and run the resulting image using QEMU. Now you can call your application and see if it works as expected.

For productive usage, we recommend to package the application as a Debian package, and provide it using an apt repository.