Kria SOM App Store Applications

Developer Deployment Guide for Ubuntu

UG1630 (v1.0) August 11, 2023

Table of Contents

Chapter 1

Overview

This guide provides an overview of the AMD Kria™ App Store for developing Ubuntu applications for the Kria Starter Kits. It describes the development requirements, over-the-air deployment packaging, and application release. It is not intended to provide detailed application development steps. The scope is to ensure partner developed applications for the Kria App Store have a seamless user experience for end customers, and that partner application developers understand the technical requirements for packaging and deployment. This document includes definitions of Linux infrastructure (kernel version and dependent libraries), shared libraries, over-the-air deployment mechanisms, and verification requirements.

This particular document outlines application deployment on Ubuntu for Kria SOMs over either native Debian packaging or a Docker image. Refer to the Kria K26 SOM Wiki for a list of comprehensive documentation related to the Kria SOM.

Application Deployment Steps

This guide outlines the key steps for an Kria App Store partner application developers to follow when preparing a Kria SOM accelerated application. The flow ensures that the application works within the shared Kria SOM Starter Kit software framework, can be hosted in the shared Kria Starter Kit deployment framework, and delivers the intended out-of-box customer experience. The document provides the shared infrastructure components information, application base operating system alignment deployment guidance, and formal release processes. The overall application deployment preparation through public release is outlined in the following five steps:

- **Step 1**: Aligning application implementation to Kria SOM Starter Kit Linux and associated infrastructure
- **Step 2**: Application packaging
  - Package generation overview
  - Identify open-source and third-party dependencies
  - Aligning application packages with naming convention
- **Step 3**: Application deployment
  - Application delivery and deployment
  - Moving Linux deployment packages to selected host
  - Regressions, build, and documentation

- **Chapter 5: Application Release Criteria and Functional Verification (Step 4):** Application release criteria and functional verification
- **Chapter 6: Application Documentation Requirements (Step 5):** Kria App Store partner application page requirements

RECOMMENDED: *Potential application developers should fill out the Partner Registration form to receive communication in advance of development. The Kria App Store team welcomes direct communication from the ever-expanding Kria SOM ecosystem and can potentially provide additional market and business insight.*

Starter Kit Linux Application Infrastructure (Step 1)

AMD Kria™ Starter Kit applications are released to operate on a targeted prebuilt Linux image. These prebuilt Linux images are generated against a Ubuntu release version. Applications targeting a given prebuilt Linux image must align on key points of coupling in the Starter Kit Linux including kernel version, Ubuntu version, and dependent AMD library version. This section of the document defines the intended application infrastructure for the Kria Starter Kit applications. To ensure a robust validation process and curated customer experience, key points of infrastructure must be aligned and maintained over the life cycle of a Starter Kit application.

Linux Kernel Alignment

The Linux kernel used by the application must be aligned with the targeted Ubuntu version. Ubuntu releases are asynchronous to AMD tool releases. Refer to the Ubuntu Xilinx site to determine which versions of Ubuntu are supported on the Kria SOM.

Package and Deployment Tools

The Kria SOM Starter Kit Ubuntu focuses on Debian as the package technology and APT as the Linux package manager. It also supports Docker to package and deploy the application. To leverage the Starter Kit standard Kria SOM Ubuntu image, the application being developed must align with Debian and APT and/or Docker based deployment.

Note: A known restriction since the Ubuntu 22.04 and dfx-mgr 22.1 release is that the hardware management functions can only be used on base OS and not Docker. Therefore, the PL firmware for an application must be packaged as a Debian image in addition to its corresponding Docker image, and the Debian image should be loaded prior to running the docker container. Despite this limitation, developers can consider Docker containers as a packaging method when, for an example, the application depends on a library version that is different than the library version required by the base OS.

Starter Kit Ubuntu rootfs, PPA, and Packages

The Kria SOM Starter Kit Ubuntu has a `rootfs` that provides a common Linux OS boot and enables flexibility for a series of applications to dynamically bring in their dependencies. In the released Ubuntu image, the `oem-archive` is already included in the personal package archive (PPA) lists.

- **OEM:** Index of /updates (canonical.com)

There are three AMD library-specific PPAs that need to be added to the platform sources list. These are:

- **SDK:** Index of /ubuntu-xilinx/sdk/ubuntu (launchpadcontent.net)
- **Gstreamer:** Index of /ubuntu-xilinx/gstreamer/ubuntu (launchpadcontent.net)
- **AMD example apps:** https://ppa.launchpadcontent.net/xilinx-apps/ppa/ubuntu/

These PPAs are added by the following commands on target:

- `sudo add-apt-repository ppa:xilinx-apps`
- `sudo add-apt-repository ppa:ubuntu-xilinx/sdk`
- `sudo add-apt-repository ppa:ubuntu-xilinx/gstreamer`

Alternatively, snap can be used to add the PPAs:

```
sudo snap install xlnx-config --classic --channel=2.x
```

For more complete commands, refer to the github.io Kria SOM Starter Kit Linux Boot page.

Table 1: **Starter Kit Ubuntu Packages**

Package	Archive
bootgen-xlnx	OEM
fpga-manager-xlnx	OEM
xlnx-firmware	OEM
u-boot-xlnx	OEM
flash-kernel	OEM
linux-xilinx-zynqmp	OEM
linux-meta-xilinx-zynqmp	OEM
OEM-limerick-kria-meta	OEM
OEM-limerick-meta	OEM
OEM-limerick-zynqmp-meta	OEM
libegl-mali-xlnx	OEM
xf86-video-armsoc-endlessm	OEM

Table 1: **Starter Kit Ubuntu Packages** *(cont'd)*

Package	Archive
xorg-server	OEM
dfx-mgr	OEM
fru-print	OEM
libdfx	OEM
xmutil	OEM
kria-dashboard	OEM
bokeh	OEM
xlnx-platformstats	OEM
xlnx-default-bitstreams	OEM
xilinx-vcu-ctrl	sdk
xilinx-vcu-omx	sdk
linux-firmware-xilinx-vcu	sdk
xilinx-runtime	sdk
vitis-ai	sdk
jansson	sdk
vvas	gstreamer
gst-perf	gstreamer
gst-xilinx-omx	gstreamer
v4l-utils	gstreamer
gst-plugins-bad1.0	gstreamer
gst-plugins-base1.0	gstreamer
gst-rtsp-server1.0	gstreamer
gstreamer1.0	gstreamer
gst-plugins-good1.0	gstreamer
gst-python1.0	gstreamer
libdrm	gstreamer
simd	gstreamer
xlnx-app-kr260-mv-defect-detect	Xilinx-apps
xlnx-app-kr260-pmod-rs485-test	Xilinx-apps
xlnx-app-kr260-tsn-examples	Xilinx-apps
xlnx-firmware-kv260-aibox-reid	Xilinx-apps
xlnx-firmware-kv260-benchmark-b4096	Xilinx-apps
xlnx-firmware-kv260-defect-detect	Xilinx-apps
xlnx-firmware-kv260-nlp-smartvision	Xilinx-apps
xlnx-firmware-kv260-smartcam	Xilinx-apps

On-target Utilities and Firmware

The Kria Starter Kit Ubuntu includes a set of utilities to help manage dynamic deployment and loading of accelerated applications. The `xmutil` application is a front-end wrapper that provides a common user-experience for interacting with the different sub-utilities. The `dfx-mgrd` utility is used to identify, load, and unload application firmware. It can be called directly or by `xmutil`.

xmutil

`xmutil` is a Python-based utility wrapper that provides a front-end to other standard Linux utilities (e.g., `dnf`) or AMD platform specific sub-utilities (e.g., platformstats). In the context of dynamic deployment, `xmutil` provides the functionality to read the package feeds defined by the on-target `*.repo` file and down select the package-groups based on the hardware platform name read from the SOM and carrier card (CC) EEPROM contents. For example, when calling `xmutil getpkgs` on the KV260 starter kit, the utility will query the package feed and then only present the package-groups that include the string `kv260` in them. This is intended to help quickly identify the accelerated applications related packages for the active platform. Developers can also use standard `dnf` calls to interact with the package feed or `apt` calls to interact with Debian PPAs.

dfx-mgrd

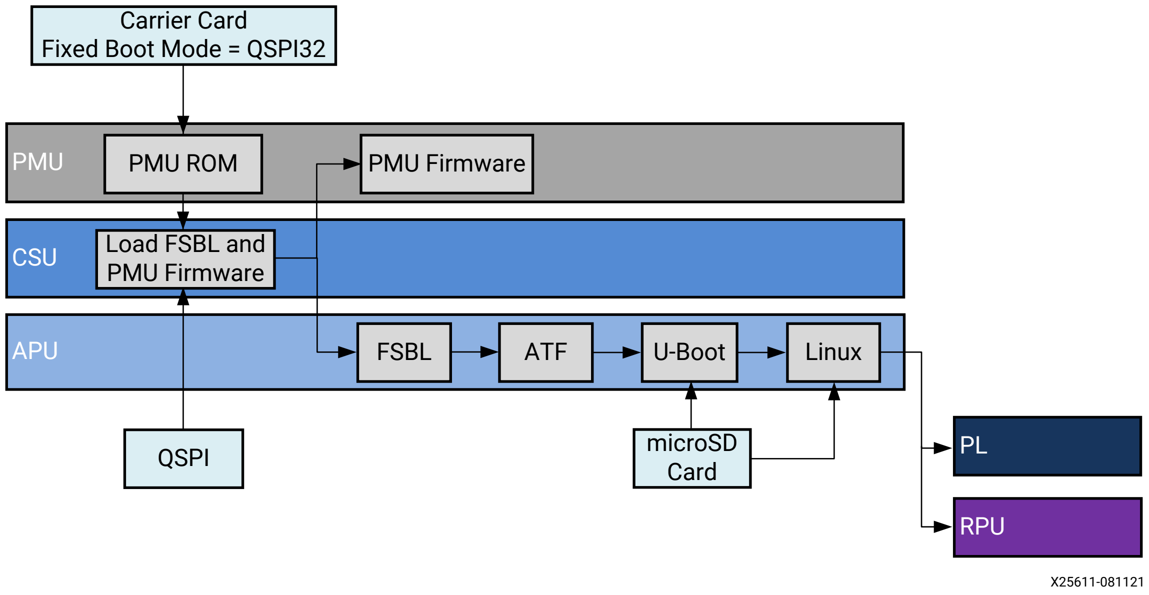
The `dfx-mgrd` is an AMD library that implements an on-target daemon for managing a data model of applications and their corresponding bitstream. The daemon maintains a data model of relevant application bitstreams, bitstream types, and active bitstreams. The `xmutil` application is calling `dfx-mgrd` when making the calls `loadapp`, `unloadapp`, and `listapps`. The `dfx-mgrd` requires that the application bitstreams and required files be loaded in `/lib/firmware/xilinx/<app_name>`. The `dfx-mgrd` uses i-notify to identify when new applications are brought into the system, and requires that the files required for an application have the same `<app_name>`. The `app_name` directory must contain:

- Application bitstream converted to `*.bit.bin` format
- Application bitstream device tree overlay `*.dtbo`
- A `shell.json` file containing information about the PL design, required 2021.1 or later
- A metadata file for PL design `*.xclbin` (optional, only required for XRT-based designs)

Starter Kit Firmware

The Starter Kit firmware, consisting of the first-stage boot loader (FSBL), platform management unit (PMU), Arm® trusted firmware (ATF), and U-Boot, are built by AMD, write protected, and shipped pre-loaded into the starter kit primary boot device. This boot firmware is locked to support a carrier card and an application agnostic boot process, which requires all application specific content to be managed in the Linux domain as shown in the following figure. A Kria App Store based application should not require any of the boot firmware components managed in the quad-SPI (QSPI) device to be edited, updated, or changed. AMD occasionally updates the user boot partitions and it is recommended to use the latest BOOT.BIN on Kria Wiki when developing and testing applications.

Figure 1: **Starter Kit Boot Process**



The EEPROM contents for both the SOM and carrier card are also pre-loaded before shipping and are write-protected. The contents of the EEPROMs are not to be edited or changed by any application developer. Content definitions are available in the *Kria K26 SOM Data Sheet* (DS987).

Application Packaging (Step 2)

This step covers requirements around the AMD Kria™ SOM Starter Kit Ubuntu application, packaging technology, and dependent libraries. The initial Starter Kit application deployment focuses on Linux packages and application deployment technologies. Two technologies can be deployed on the Kria SOM Starter Kit: Ubuntu–Debian and Docker. Both methods are well documented outside of this user guide. Overview information on Linux package and package management technologies are available at the following references:

- **Debian:** Debian Packaging Tutorial
- **Docker:**
  - https://docs.github.com/en/packages/working-with-a-github-packages-registry
  - Docker Extension for Visual Studio Code
  - Docker Documentation | Docker Documentation
  - Reference documentation | Docker Documentation

Ubuntu Debian Packaging

All Debian based Linux distribution like Ubuntu supports the package installation through binary Debian files. The Debian files (.deb) can be installed locally or through remote archives using apt family of programs (apt, apt-get, apt-cache, aptitude). Apart from regular Debian software package repositories, Ubuntu also provides Personal Package Archive (PPA) based user package hosting. Refer to the Ubuntu Packaging Guide for details on an Ubuntu development flow. The remote hosting provides easy way to maintain and upgrade the Debian packages. Before working with Debian packages for Ubuntu, it is important to understand the distinction between source and binary packages.

Binary packages (DEB) are archive files (ar) containing all the files that the package will install (binaries, libraries, configurations, documentation, etc.) along with the metadata required by the packaging system (apt) to install and maintain those packages. These packages are usually maintained by the OS vendor in a package repository and installed/updated on the system by the user with a package manager such as apt.

Source packages (files `*.orig.tar.gz` (sources), `*.debian.tar.gz` (changes for package build), and .dsc (contains the name of the package and other information)) contain all the sources and metadata required to build a working binary package. The source package is uploaded and compiled by a build server, and is then made available through the PPA.

Depending on the constraints of the chosen Debian package host, the developer either needs to upload binary packages (.deb files) or the source version of the package. This guide highlights both the basic Debian source package creation as well as the basic Debian binary packaging process. Refer to the Debian packaging wiki for details on the source package generation tutorial. This Practical Guide is a good Debian binary package generation guide.

Build Requirements

Building packages for the Kria SOM requires access to either a native build environment (Linux running on Arm64) or a chroot/cross-compile setup (Linux with sbuild or pbuilder on AMD64). Of the two options, a native setup is significantly easier to work with and allows the installation and testing of the resulting packages. The following steps outline the process for getting Ubuntu set up and configured for packaging on a Kria SOM board.

***Note*:** It is possible to use an emulated Ubuntu on a native Arm64 machine, such as an M1 powered MacBook, but that is out of scope of this document.

***Note*:** It is possible to build the source packages on a standard x86 machine and rely on Launchpad to do the building. For details see https://launchpad.net/launchpad-buildd.

The developer is encouraged to first test the build and installation of the source locally on the target, to ensure it can be built and is functional. For example, `git clone` the source and then run `make` and `sudo make install` to see everything is behaving as expected.

Setup Package Build Environment

The build environment expects certain tools to build the packages. These tools also provide a helper utility to generate the some of the Debian files required for the build process.

Install following tools before starting the build process:

```
sudo apt install gnupg dput dh-make devscripts lintian pbuilder fakeroot
```

Setup the email and user name to be used for the package. Consider keeping this in the `.bashrc`.

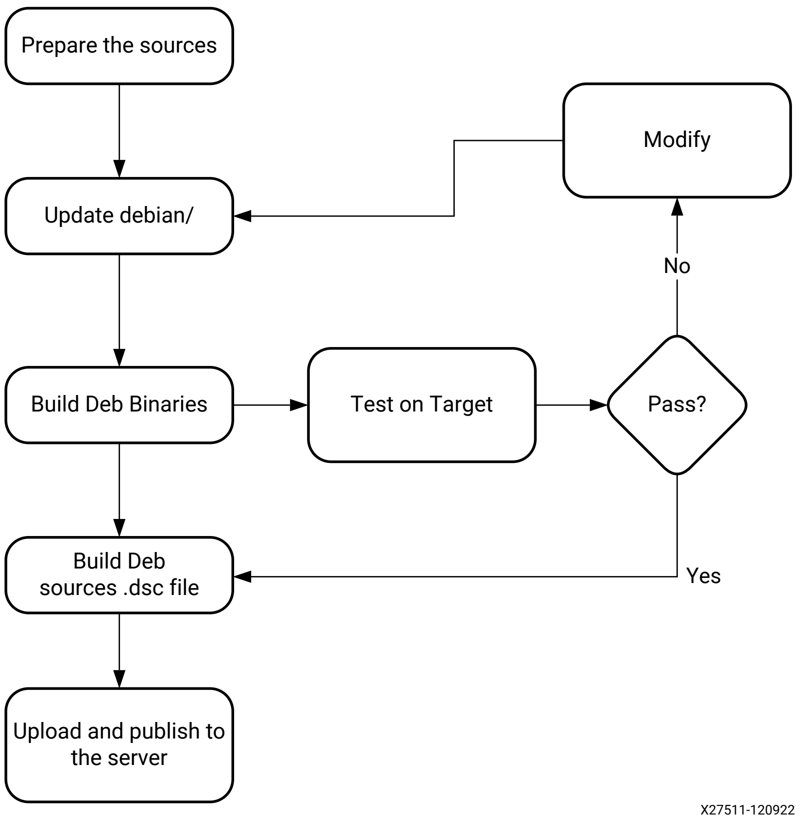
```
DEBEMAIL="your.email.address@example.org"
DEBFULLNAME="Firstname Lastname"
export DEBEMAIL DEBFULLNAME
```

Build Flow for Debian Source Packaging

A typical build flow for Debian source packaging involves the following steps:

1. Prepare sources
2. Prepare build metadata (create/update `debian/` directory)
3. Build DEB binary and test
4. Build sources DSC file
5. Upload the source DSC file

Figure 2: **Ubuntu Development Flow**



Prepare Sources

The build is performed from inside the source directory. All the dependency packages must be installed for the source to build. The source working directory is prepared as described here for new sources or existing packages.

New Package

For a new package, choose the correct package name that represent the application. Refer to *DEB Naming Convention* for the naming convention to follow.

The upstream source packages can be fetched using `wget` or equivalent Linux commands. Normally the downloaded tarball is in the form of `<package>-<version>.tar.gz`. These usually contain a directory called `package-version` with all the sources inside. If the latest version of the source is available through a version control system (VCS) such as Git, Subversion, or CVS, you need to get it with `git clone`, `svn co`, or `cvs co` and repack it into `tar+gzip` format by using the `--exclude-vcs` option.

The source must include the correct license used for the software distribution. Extract the package and change directory to move inside the package source.

Existing Package

For the existing Debian packages, the latest sources can be fetched using the `apt source <package>[=<version>]` command. For this to work, make sure the `deb-src` entry for the repository hosting the package is added to the `/etc/apt/sources.list` file. For example:

```
deb-src https://ppa.launchpadcontent.net/xilinx-apps/ppa/ubuntu/ jammy
        main
```

The `apt source <package>` command downloads the following files for the package:

- Debian source package control file: `<package>_<version>-<BuildVersion>.dsc`
- Debian modification (`debian/` directory): `<package>_<version>-<BuildVersion>.debain.tar.xz`
- Original sources with the Debian directory: `<package>_<version>.orig.tar.gz`

By default, the `apt source` command also extracts the package source and adds the Debian directory from the tar ball.

If the new source package is available (as an update), then rename it to `<package>_<new_version>.orig.tar.gz` and keep it at the same level as `<package>_<version>.orig.tar.gz`. Next, issue the `uupdate` command to automatically generate the new update package, and change the directory to the following new source:

```
cd <package>-<version>
uupdate ../<package>_<new_version>.orig.tar.gz -v <new_version>
cd ../<package>-<new_version>
```

Update Debian Metadata

The information needed to build a Debian package is inside the `debian/` directory under the source code directory. For example: `<package>-<version>/debian`.

Before examining the content of the files inside the `debian/` directory, make sure the directory exists in the working source package. For an existing package, it will already be there if the `uupdate` command is being used. Make sure to also update the change-log for the existing package. For a new package, the relevant files (with default template) can be generated using the `dh-make` command. To run:

```
cd <package>-<version>/
dh_make -p <package>_<version> --single --createorig --copyright  <license
type>
```

Generates the `<package>-<version>/debian` directory.

Contents of the Debian Directory

All the packaging work should be made by modifying files in the `debian/` directory. The Debian documentation in Chapter 4: Application Deployment, Release, and Maintenance (Step 3) includes a basic explanation of the files under this directory. The minimum necessary (required) main files under the Debian directory include:

- **Control:** Metadata about the package (dependencies, package name, etc.). The fields of this file are described in the *Debian Policy Manual, 5: Control files and their fields*.
- **Rules:** Specifies how to build the package. The main Makefile triggers the build/install/clean system for the source package).
- **Copyright:** Copyright information for the package. Use the `dch` command to automatically fill the fields for the next update.
- **Changelog:** This is the history of the Debian package. The format is described in the *Debian Policy Manual, 4.4 Debian changelog*.
- **Source/format:** The format of the source package. Add a single line `"3.0 (quilt)"` to this file.

For the basic packaging, delete rest of the files in the `debian/` directory. For more advance settings and an explanation of the each of the files, refer to the Chapter 5: Application Release Criteria and Functional Verification (Step 4) section. Other files under the Debian directory.

Build Debian Binaries and Sources DSC File

In the package sources directory, use the `debuild` command:

1. Install the dependencies of the package specified by the `debian/control` file.

   ```
   cd <package>-<new_version>/
   sudo apt build-dep <package>
   ```

2. Build the Debian binary (only).

   ```
   debuild -b -us -uc
   ```

3. Inspect the contents of the binary.
    a. The new `*.deb` file is now available in the parent directory
    b. To see the content of file:

    ```
    dpkg -c ../*.deb
    ```

4. Install the binary on the target and verify functionality.

    ```
    dpkg -i ../*.deb
    ```

Upload the Package

Use the `debuild` command again using the `-S` option (`debuild -S`) for a full upstream source build. Use the `generate source.change` file in the parent directory to upload the package to the required personal package archive (PPA).

```
debuild -S cd .. dput <PPA to upload> <package>_<new_version>-
<build_version>_source.change
```

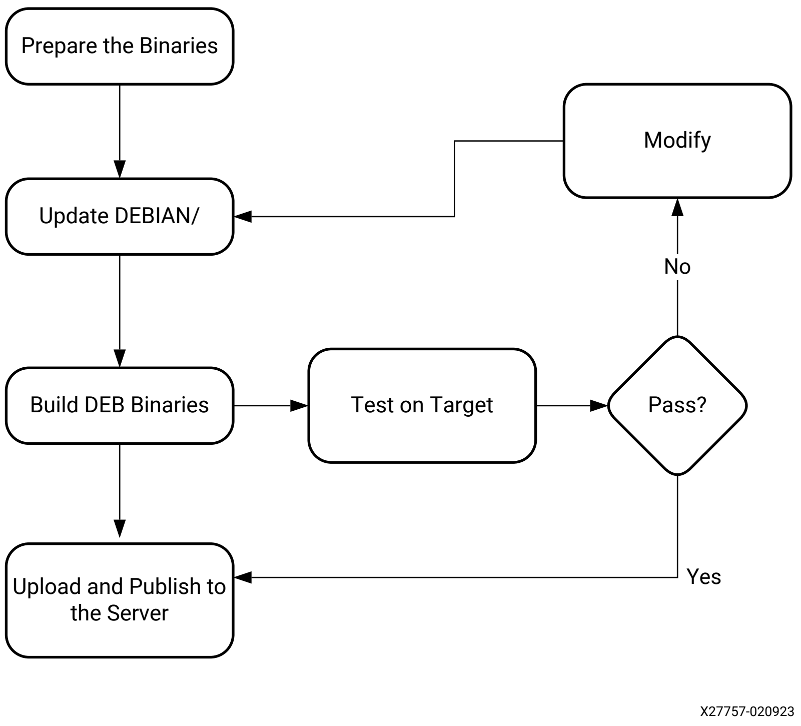
Build Flow for Debian Binary Package

A typical build flow for a Debian binary package involves following steps:

- Prepare binaries
- Prepare build metadata (create/update `DEBIAN/` directory)
- Build DEB binary and test
- Upload the Debian binary DEB file

Note: In the appendix, there is a simple Hello World Example for Debian binary packaging.

Figure 3: **Debian Binary Package Flow**



Prepare Binaries

The first step is to choose the correct package name that represent the application. For the naming convention to follow, refer to the DEB Naming Convention. Create a temporary working directory for the package following the naming convention. In the working directory, put your program files where they should be installed onto the SOM target. For example, if the files should be in the `/opt/company/app/` folder on target, then place them in the `<working directory>/opt/company/app/`.

Update Debian Metadata

The information needed to build a Debian package should be placed inside a `DEBIAN/` directory under the source code directory (`DEBIAN` is all capitalized). Create a `<working directory>/DEBIAN/`, and then create two files:

- The control file contains the metadata about the package. The fields of this file are described in the Debian Policy Manual, 5.3 "Binary Package Control files".
- The md5sums are generated to specify the md5sum of the package.

Build Debian Binary File

To build the DEB file, `cd` above the `<working directory>` and build the DEB file:

```
dpkg-deb --build --root-owner-group <working directory>
```

This generates a DEB file. Move the file onto the target, and install and test on the target. If it works as expected, the DEB file is ready to be uploaded and published.

Naming Convention for Debian Packages

Application developers should follow the recommended naming convention when hosting packages on their server to avoid duplication and include necessary information.

```
<vendor>-<package_name>[-firmware]_<upstream_version>-<debian_revision>
+<dist_codename>_<Architecture>.deb
```

Examples:

```
mycorp-my-app_1.0.1-1+focal_amd64.deb
```

```
mycorp-my-app-firmware_1.0.1-1+focal_amd64.deb
```

- **Vendor:** Name of company generating the package. It has no special character, no space, only lower case: *[a-z0-9]+*.
- **package_name:** `<app_name>-<platform>` Name describing the packaged software. It is important to include platform name especially for firmware packages as bitstreams are tightly coupled with target device. A string without special character or "." can include "-": *[a-z0-9-]+*.
- **Firmware:** A suffix to apply only in case the app is for the PL part.
- **upstream_version:** Version of the packaged software: Lowercase string without special except "." and "-" *[a-z0-9.-]+*.
- **debian_revision:** Package revision, to increment (if required) to republish a package with the same `upstream_version`. 1 on the first package version, numbers only: *[0-9]+*.
- **dist_codename:** The Linux distribution codename: example: `1.0.1-1+focal` for ubuntu v20.04 or `1.0.1-1+bullseye` for Debian 11. Lowercase letters only: `[a-z]+`.
- **Architecture:** The package architecture. Example: `"amd64", "all",` ....

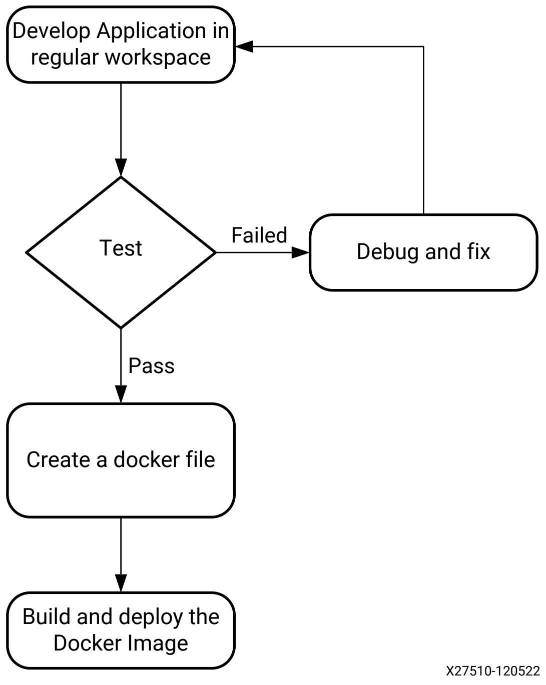
Ubuntu Docker Packaging

Docker Image Creation

The application can be packaged as a docker image. Refer to the Docker container documentation (https://www.docker.com/resources/what-container/) for its capabilities. This tutorial focuses on packaging the Docker image on the target. Containerizing an application (creating a docker image for the application) consists of the following top level steps:

1. Create/fetch and test the application
2. Create the `dockerfile`
3. Build the docker image using the `dockerfile`
4. Push the docker image to the `dockerhub` or an equivalent registry

Figure 4: **Creating a Docker File**


Develop Application in regular workspace
Test
Failed
Debug and fix
Pass
Create a docker file
Build and deploy the Docker Image
X27510-120522

Test Application

The first step to creating a docker image is to have the application tested in a working environment (e.g., `ubuntu + gstreamer-xilinx1.0-pulseaudio + python3.10` and so on). It is possible to develop and test inside another base docker container (a sandbox running process of a docker image). This allows specific versions of dependencies that are needed for the application without disturbing the original workstations environment (e.g., having python 3.11 in the docker workspace versus 3.10 on a local working environment). A separate working environment is not mandatory, you can develop and test the application on a regular machine, provided it can install all the dependencies and set the required environment.

AMD also provides two example docker images used to rebuild example Docker containers.

1. The `kria-developer` Docker image can be found on the Docker hub. It contains all the development libraries you need to build against.
2. The `kria-runtime` Docker image is a subset of the `kria-developer` Docker image, and can also be found on the Docker hub. Because it only contains the runtime library dependencies, it is, therefore, a smaller footprint base image that can be used to install a pre-built app.

Docker Image

A docker container runs on an isolated file system. This custom file system is provided by a docker image. Because the image includes the container's file system, it must have everything needed to run an application, all the dependencies, configurations, scripts, binaries, etc. The image also includes other configuration elements for the container, such as environment variables, a default command to run, and other metadata. A docker file must be built to create a docker container image. Additionally, the docker itself must be installed on the system before building the docker image.

Dockerfile

In addition to the application source code, a `dockerfile` is needed to describe the containerization of the application. Generally, the `dockerfile` is place in the top-level directory of the source code. This file contains list of instructions. The first must be the `FROM <ImageName>` instructions, that specify the parent image (base image) to be used for this image. A detailed `dockerfile` format is documented in Dockerfile Reference. A normal docker file could contain instructions to:

- Use the `FROM` reference to the base/parent image using `FROM ImageName`.
- Set the working directory in the `rootfs` using the `WORKDIR` for any RUN, CMD, ENTRYPOINT, COPY and ADD using the `WORKDIR /path/to/workdir`.
- Install the required dependencies using RUN to run regular commands in the base image (for example: cd, apt install, etc.) using the `RUN <command>`.

- Setup environment variables using ENV with the `ENV <key>=<value> ....`
- Copy the required files to the `rootfs` from local or another image using COPY or ADD COPY

  ```
  [--chown=<user>:<group>] <src>... <dest>
  ADD [--chown=<user>:<group>] [--checksum=<checksum>] <src>... <dest>
  ```

- Expose the network ports (docker network) using:

  ```
  EXPOSE <port> [<port>/<protocol>...]
  ```

- Run the actual application using CMD. Each image should only have one CMD: `CMD ["executable","param1","param2"]`.

The following is an example `dockerfile`:

```
# Choose the base parent image for build
FROM ubuntu:latest as build

RUN apt-get update && \
apt-get install -y build-essential git cmake autoconf libtool pkg-config

WORKDIR /src

COPY CMakeLists.txt main.cpp ./

RUN mkdir build && cd build

RUN cmake .. && make




# Choose the base parent image for deployment
FROM ubuntu:latest

WORKDIR /opt/xilinx/test

COPU -from=build /src/build/test-app ./

CMD [“/.test-app”]
```

Other example docker files for the Kria SOM project are available in GitHub (https://github.com/Xilinx/kria-docker). If using a similar build environment for multiple applications development, it might make sense to create a build-image docker image, with all the tools installed, and use that docker image to build the application image. An example is the kria-developer docker image in the AMD docker hub. The base developer and application runtime docker images are available there. Optionally they can be leveraged by an app developer as a basis for a docker-based application image. See the docker documentation for information on multi-stage builds.

Build Docker Image

To build the docker image named `test-image`, go to the top-level directory of the source (which contains the `dockerfile`) and run the following docker build command:

```
docker build -t test-image
```

This generated image is now available to use and test on the current workstation using the `docker run test-image` command. See the docker guide to learn how to configure and run the docker images on a target.

Application Dependent Libraries

When not included in the default Kria Ubuntu Linux image, application developers must identify all dependencies and install them as part of their application. If the corresponding library is already available as a standard Ubuntu library, the dependencies can be specified when generating the application package.

Third-party and Open-Source Libraries

Application developers must ensure that the packages they create and deploy third party libraries in a manner that is compliant to the libraries open-source license. The preferred deployment is to include third-party libraries as part of the core OS library offered through the application and library deployment.

AMD-Xilinx Dependent Libraries

Application developers must account for version alignment changes of any other AMD libraries required for their application. The following lists the commonly used AMD libraries for some Kria SOM applications. These libraries are managed in the previously captured AMD SDC and Gstreamer PPAs. Multiple versions of these libraries are available over the lifetime of the device, thus an application developer should call out the version dependencies (as appropriate) when generating their application package.

- Xilinx Runtime (XRT)
- AMD Vitis™ AI
- Vitis video analytics software developer's kit (VVAS)

Note: The AMD developed libraries are expected to be delivered as packages in the AMD PPA, and as an application developer you can use these pre-built libraries when creating your applications.

Package File System Unpacking

The Linux community defines best practices on managing the Linux file system for dynamically deployed applications (https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.html#optAddonApplicationSoftwarePackages). Many open-source libraries also make assumptions around the file system structure, including the AMD on-target management utilities (e.g., `dfx-mgrd`). Application developers must ensure that they create package definitions that install the packaged files using the following file system structures:

- Load the application into `/opt/<company_name>/<application>/bin`
- Load libraries into `/opt/<company_name>/<application>/lib`
- Load firmware into `/lib/firmware/xilinx/<application>`
- Supporting media:
  - `/opt/<company_name>/<application>/share/movies`
  - `/opt/<company_name>/<application>/share/notebooks`
  - `/opt/<company_name>/<application>/share/images`

In all instances, developers should verify names to eliminate any conflicts with existing packages in the Docker registry.

The following example shows how the package must be unpacked to the file system for the `xmutil` and `dfx-mgrd` to run in the file structure for an AMD accelerated application.

- Application location: `/opt/uncanny/uncanny-kv260-smartcamera-alpr`
- Contains executable application
- Contains associated application helper scripts
- Application bitstream: `/lib/firmware/xilinx/uncanny-kv260-smartcamera-alpr`
- Contains `*.bit.bin` or `*.bit`
- Contains `*.dtbo`
- Contains `*.xclbin` (optional, only required for XRT-based designs)
- Contains `shell.json`

Application Deployment, Release, and Maintenance (Step 3)

AMD Kria™ SOM starter kit applications are expected to be deployed to target customer systems over-the-air. Once an application is created for a particular starter kit Linux image, it is the responsibility of the partner application developer to implement any migration over time to support future starter kit Linux images.

Application Deployment

Kria SOM applications are intended for dynamic deployment to a target that is running the Kria SOM Starter Kit Ubuntu. The applications can be deployed as Debian packages hosted by partners or Docker containers hosted on a partner's Docker hub.

Partner Application Delivery–Debian

Third-party application developers are responsible for generating a package that is compatible with the targeted starter kit Linux version. The package name must incorporate the target platform name (e.g., KV260).

Debian packages can also be hosted on a developer's server. In addition, Canonical can be engaged to support developing on Ubuntu as well as hosting Debian source packages. An application document should include information on how to add the Debian PPA into the target platform.

Partner Application Delivery–Docker

To share the Docker image, push or publish the image to a container library like `dockerhub`, `linuxserver.io`, `githubpackages`, or an equivalent. When using `dockerhub`, the Kria SOM starter kit Linux requires no additional setup. However, when using other docker registries, application documentation needs to include steps to add the docker registry to the docker configuration.

After a registry is setup, create an account on the chosen repository. Use the following command to push the docker image to the repository:

```
docker tag <name of docker image> <docker registry>/<name of docker
image>:<TAG>
docker push <docker registry>/<name of docker image>:<TAG>
```

Source and Build Infrastructure

The third-party application hosting discussed in this document focuses on the distribution of pre-built binary content. The application developer is responsible for the capture and maintenance of source and its associated build infrastructure. While not an AMD requirement, the recommendation is to make the application or customizable aspects of the application source available (e.g., GitHub) and to support the end-customer's ability to build from source. The hosting of the source repositories is also the responsibility of the third-party application developer.

Out-of-box and Regression Testing

As part of the application in development, create a simple example application that allows the end customer to exercise the application with a minimal knowledge of the system. Another recommended practice is to include a series of regression tests for release testing and validation. Because example and regression tests are application dependent, it is the application developer's responsibility to ensure that your end customer creates a series of tests with test coverage that ensures customer-facing functionality.

Chapter 5

Application Release Criteria and Functional Verification (Step 4)

Before releasing an application to the public, the AMD team must perform a functional verification in the form of a testing bash. A bash makes sure that the demonstration and getting started flow work as expected.

Bash

To initiate a bash on your application, contact AMD at least six (6) weeks prior to launch. To get started, email kria_appstore@xilinx.com. The bash verification of the application includes:

- Performance criteria conformance
- Stability testing: No crashes, no unexpected outcomes when running through the getting started steps, and demonstration bring up issues
- Application package tests `getpkgs` and `dnf install`
- Validation of the licensing mechanism
- Feedback collected from the bash team and incorporated into the getting started guide
- Fixing all critical bugs

Bash Documentation

The next section provides a detailed list of documentation required as part of the application release criteria. However, a subset of this list is required for running the bash. For this subset, GitHub documentation is preferred and recommended.

- Getting started guide: Step-by-step instructions on how to bring up the application demonstration on the KV260 starter kit.
- Architecture document: Include details for the team to understand the solution and any information you plan to make public post launch.
- Known issues: A list with a description on how to fix them.

Note: All GitHub documentation is hosted on the partner GitHub. AMD does not host the documentation pages for your application. Links are included to your external web page from the Kria App Store.

Chapter 6

Application Documentation Requirements (Step 5)

Application developers are expected to provide the following list of content, documents, and graphics for AMD to create and host a dedicated page for your application:

- Content for the application page:
  - Title
  - Description of application
  - Top five features list
  - Hardware needed, including accessories (with URLs to purchase accessories)
  - Steps to access packages, including how to setup access to required package feed/Debian package server/Docker registry
  - Top five FAQs
  - Link to your web page (with URL)
  - Information on AMD Kria™ SOMs and the accelerated application
- Documentation:
  - Solution brief
  - Figure of merit/benchmarking
  - Customer-facing presentation
  - Field training presentation with pointer for sales
  - Standard customer-facing documentation (GitHub)
  - Getting started guide (GitHub)
- Graphics:
  - High-level block diagram
  - High-resolution use case images
  - Solution video with source file and consent to host
  - High resolution logo
  - Light and dark background

When this collateral is ready, the next step is to create a Kria App Store page. Email kria_appstore@amd.com to start the process of creating a mock up. AMD strongly recommends initiating this request at least four weeks prior to the go-live date planned for the application.

Hello World Examples

This appendix outlines two examples for, creating on target, testing on target, where a binary package is created and can be uploaded to a private Debian PPA or a Docker hub.

Both examples use a simple `hello.c` file with the following content:

```
#include <stdio.h>
int main()
{
        printf("Hello World!\n");
}
```

Hello World Example for Debian

This example generates a Hello World binary Debian package. This example is created entirely on a Kria SOM target.

First, create a folder. This example uses `amd-hello-kria_1.0.0-1+jammy_arm64/`. Customize the example to a specific project: `<vendor>-<package_name>-<target>_<upstream_version>-<debian_revision>+<dist_codename>_<Architecture>`.

In the folder, create the following folders and add the perspective files. In compliance with the Package File System Unpacking file system structures, the files are put into a `/opt/amd/hello/` folder:

```
amd-hello-kria_1.0.0-1+jammy_arm64/opt/amd/hello/src/hello.c
```

```
amd-hello-kria_1.0.0-1+jammy_arm64/opt/amd/hello/src/makefile
```

This is the `makefile`:

```
CC      = gcc
CFLAGS  = -g
RM      = rm -f

default: all
all: hello
hello: hello.c
                $(CC) $(CFLAGS) -o hello hello.c

install: hello
```

```
        install -d $(DESTDIR)/opt/xilinx/hello/bin/
        install -m 755 hello $(DESTDIR)/opt/xilinx/hello/bin/

clean veryclean:
        $(RM) hello
```

In the `amd-hello-kria_1.0.0-1+jammy_arm64/opt/amd/hello/src/` folder, execute the following to copy the executable to the `bin/` folder:

```
make
mkdir ../bin
mv hello ../bin
```

Now you have:

`amd-hello-kria_1.0.0-1+jammy_arm64/opt/amd/hello/bin/hello`

Next, create the `amd-hello-kria_1.0.0-1+jammy_arm64/DEBIAN/control` file. Modify the fields accordingly, making sure the source, vendor, version, and architecture fields match.

```
Source: amd-hello-kria_1.0.0-1+jammy_arm64
Section: unknown
Priority: optional
Maintainer: <your name> <your email>
Standards-Version: 4.6.0
Homepage: <your home page>
Rules-Requires-Root: no
Package: amd-hello-kria
Version: 1.0.0-1+jammy
Vendor: amd
Architecture: arm64
Description: hello world test
```

In the `amd-hello-kria_1.0.0-1+jammy_arm64/` folder, execute this to generate `amd-hello-kria_1.0.0-1+jammy_arm64/DEBIAN/md5sums`.

```
md5sum $(find * -type f -not -path 'DEBIAN/*') > DEBIAN/md5sums
```

The following files should be created:

`amd-hello-kria_1.0.0-1+jammy_arm64/opt/amd/hello/bin/hello`

`amd-hello-kria_1.0.0-1+jammy_arm64/opt/amd/hello/src/hello.c`

`amd-hello-kria_1.0.0-1+jammy_arm64/opt/amd/hello/src/makefile`

`amd-hello-kria_1.0.0-1+jammy_arm64/DEBIAN/control`

`amd-hello-kria_1.0.0-1+jammy_arm64/DEBIAN/md5sums`

In the top-level folder, find `amd-hello-kria_1.0.0-1+jammy_arm64/`, and generate a DEB file:

```
dpkg-deb --build --root-owner-group amd-hello-kria_1.0.0-1+jammy_arm64
```

This generates an `amd-hello-kria_1.0.0-1+jammy_arm64.deb` file. Find and test the package using the following commands, and inspect the `/opt/amd/hello/` folder after installing the Debian package.

```
dpkg -c  amd-hello-kria_1.0.0-1+jammy_arm64.deb      # look at the content
of .deb
sudo dpkg -i  amd-hello-kria_1.0.0-1+jammy_arm64.deb # install the Debian
package
sudo dpkg --remove amd-hello-kria                    # uninstall Debian
package
```

Next, upload the DEB file into a private PPA, and add the PPA to the PPA list on the Kria SOM target. Once the PPA is visible on the target, use the following to install the package:

```
sudo apt install -y amd-hello-kria
```

After that, you should see a binary in the `/opt/Xilinx/hello/bin/hello` folder and be able to execute it.

Hello World Example for Docker

To create a Docker image, first create a `makefile` to compile `hello.c`:

```
CC       = gcc
CFLAGS   = -g
RM       = rm -f

default: all
all: hello
hello: hello.c
    $(CC) $(CFLAGS) -o hello hello.c
clean veryclean:
    $(RM) hello
```

To create a `Dockerfile` using the following. The intention is to print *Hello World* from `hello.c` when running the Docker image.

```
# Choose the base parent image for build
FROM ubuntu:latest as build
RUN apt-get update && \
apt-get install -y build-essential git cmake autoconf libtool pkg-config
WORKDIR /workspace
COPY makefile hello.c ./
RUN make
```

```
# Choose the base parent image for deployment
FROM ubuntu:latest
WORKDIR /opt/xilinx/test
COPY --from=build /workspace/hello  ./
CMD ["/.hell"]
```

On the target, put the three (3) files in the same folder. In this example, the files are in `amd_hello_kria/`. Customize the example to a specific project: < `vendorName_appName_targetName` > and build and test on the target:

```
docker build -t amd_hello_kria .     # build the docker image

docker run amd_hello_kria // test docker image
```

Now you can upload to your repository for testing. Create a user name on www.docker.com, remember your `<username>` and `<password>`. Then, upload the docker image into your private repository for staging:

```
sudo docker login --username <username> --password <your password> # login
into dockerhub
docker build -t <username>/
amd_hello_kria .                                                   # build
for private repo
sudo docker push <username>/
amd_hello_kria:latest                                     # push to private
repo
```

Test using the following commands:

```
docker pull <username>/amd_hello_kria:latest                              #
pull image
docker run <username>/amd_hello_kria:latest                              #
run image
```

Appendix B

Additional Resources and Legal Notices

Finding Additional Documentation

Documentation Portal

The AMD Adaptive Computing Documentation Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Documentation Portal, go to https://docs.xilinx.com.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the AMD Vivado™ IDE, select **Help → Documentation and Tutorials**.
- On Windows, click the **Start** button and select **Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

***Note*:** For more information on DocNav, refer to the *Documentation Navigator User Guide* (UG968).

Design Hubs

AMD Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- Go to the Design Hubs web page.

Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Support.

References

These documents provide supplemental material useful with this guide:

1. *Kria KV260 Vision AI Starter Kit Data Sheet* (DS986)
2. *Kria K26 SOM Data Sheet* (DS987)
3. *Kria KV260 Vision AI Starter Kit User Guide* (UG1089)
4. *Debian Packaging Tutorial*

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
8/11/2023 Version 1.0	
Initial release.	N/A

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright