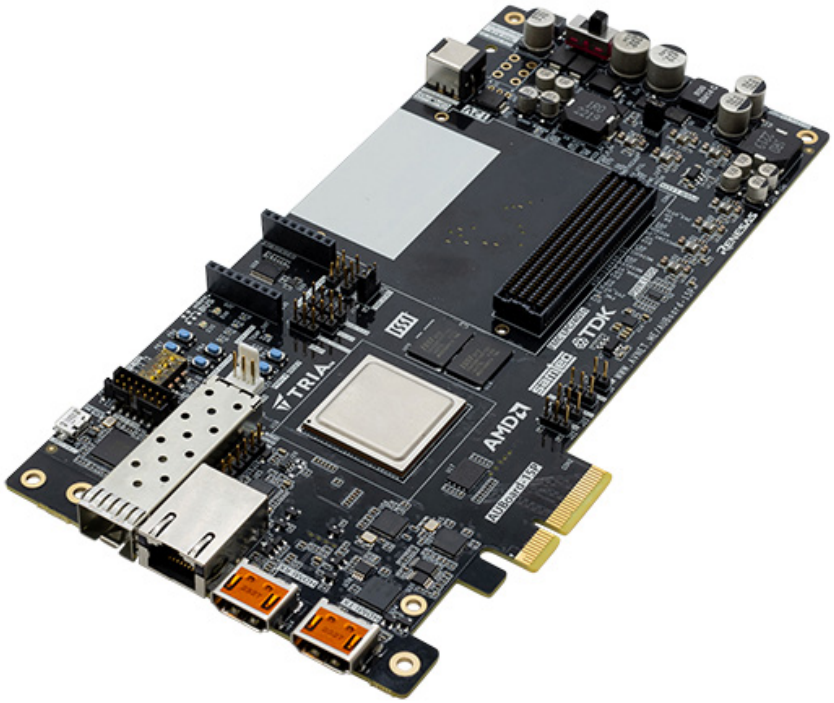
TRIA
AN AVNET COMPANY



AUBoard 15P Development Kit

Setup guide

doc version: v1.6.2

CONTENTS

1 Introduction **5**

2 Yocto for microblaze **5**
2.1 Poky structure . . . 5
2.1.1 Some definitions . . . 5
2.1.2 Some usefull variables . . . 6
2.2 Installation . . . 7
2.3 Poky Langdale Xilinx: Initial Setup for the AUBoard . . . 8
2.3.1 Custom "meta-aag-auboard" layer . . . 8
2.3.2 Creating the Machine File for AUBoard 15P Development Kit . . . 9
2.3.3 Using a Custom XSA Generated with Vivado (AUBoard 15P Development Kit) . . . 9
2.3.4 The external-hdf.bbappend . . . 10
2.3.5 The bitstream-extraction Recipe . . . 10
2.3.6 mysource.tar.gz . . . 11
2.4 Compilation . . . 11
2.4.1 Setting the Number of Threads for Compilation (optional) . . . 11
2.4.2 Saving Disk Space (optional) . . . 11
2.4.3 AUBoard 15P Development Kit board target . . . 11
2.5 Compilation output . . . 12
2.6 Conclusion . . . 12

3 Custom Layers **13**
3.1 Custom images targets . . . 13
3.1.1 Set default root password . . . 13
3.2 Openssh patch . . . 13
3.3 Setting UTF8 locale encoding . . . 14
3.4 Patching recipes with devtool command . . . 14
3.4.1 The udmabuf driver (out-of-tree) . . . 15

4 AUBoard **17**
4.1 Board Settings and Structure . . . 17
4.1.1 QSPI . . . 17
4.2 The meta-aag-auboard layer . . . 17
4.3 Configurations folder . . . 18
4.4 The recipes-bsp recipe . . . 18
4.4.1 U-Boot . . . 18
4.4.2 The device-tree . . . 19
4.4.3 The external_hdf recipe . . . 20
4.5 The recipes-core group . . . 20
4.5.1 The fs-version and base-files recipes . . . 20
4.5.2 The images recipes target . . . 20
4.6 The recipes-kernel group . . . 21
4.6.1 Kernel Driver Configurations . . . 22
4.6.2 The kernel-module-hdmi Out-of-Tree Driver . . . 23
4.6.3 IDT8T49N240 Issue and Kernel Patch . . . 24
4.6.4 HDMI demo script . . . 24
4.6.5 HDMI-TX Driver RGB Color Issue . . . 24
4.7 The recipes-demo group . . . 26
4.8 Compilation . . . 27
4.8.1 Deployment . . . 27
4.8.2 Generating the mcs file . . . 27
4.8.3 Programming the Flash Memory with Vivado . . . 28
4.9 Vivado project . . . 31
4.9.1 Required Vivado Licenses . . . 35

5 Conclusion **36**
5.1 HDMI Screenshot Demo . . . 36
5.2 Bonus Section: Dump Image from the PCIe . . . 36
5.2.1 Identifying the PCIe Device . . . 37
5.2.2 Enable IOMMU Support *(One-Time Setup)* . . . 37
5.2.3 Bind the PCIe Device to VFIO . . . 37
5.2.4 Determine IOMMU Group . . . 38
5.2.5 Enumerate BAR Regions . . . 38
5.2.6 Dump BAR1 Memory . . . 38
5.2.7 Verify Output . . . 41
5.2.8 Brief Recap . . . 41
5.2.9 Doc source code . . . 41

References **42**

Document Control

Document Title:	**AUBoard 15P Development Kit – System Documentation**
Version:	1.6.2
Company:	Tria Technologies
Author:	Jacopo Fessia
Approved by:	-
Creation Date:	2025/03/03
Last Modified:	2025/06/06
Status:	Draft

Revision History

Rev.	Date	Description
1.0.0	2025/03/03	Initial release of the system documentation
1.1.0	2025/03/27	HDMI color format fix and idt8t49n24x patch
1.2.0	2025/03/28	Compilation, deploy, demo and conclusion
1.2.1	2025/03/28	Review
1.3.0	2025/04/01	Vivado project description and minor changes
1.4.0	2025/04/11	Dump image from AUBoard
1.5.0	2025/04/17	Instructions for PCIe dump
1.6.0	2025/05/22	Minor fix, Vivado ELF instructions
1.6.1	2025/05/30	HDMI Vivado IP license instructions
1.6.2	2025/06/06	Instruction to Program the AUBoard flash memory

1 INTRODUCTION

"Yocto Project®" is an open source collaborative project aimed at creating custom Linux-based operating systems, independent of the product's hardware architecture. The key components of the Yocto project are BitBake, OpenEmbedded-Core, and Poky.

- The "bitbake" is the engine of the OpenEmbedded build system. It analyzes metadata, generates task lists, and executes them.
- The "OpenEmbedded-Core" is a set of metadata, namely *recipes*, classes, and associated files, used by systems derived from OpenEmbedded.
- Poky, as often misunderstood, is not a Linux-based operating system but rather the application of the Yocto Project that contains a set of tools and metadata enabling its creation and customization. Poky is platform-independent and performs cross-compiling, using the BitBake Tool, OpenEmbedded Core, and a default set of metadata. The main objective of Poky is to provide all the features an embedded developer needs.

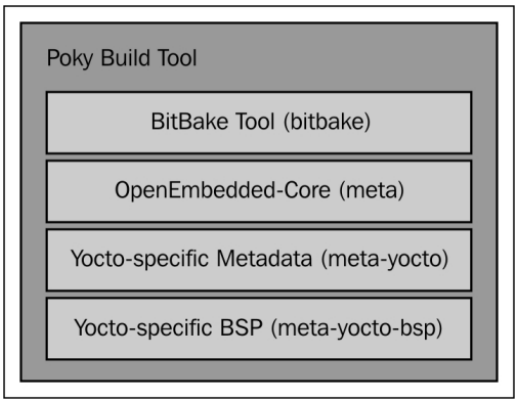


Figure 1: Poky Build Tool

Other terms used in this environment include:

- Recipe: the most common form of metadata. The recipe contains a list of settings and tasks used to create a binary image file. The recipe describes where to get the code and what (code) patch needs to be applied. At the same time, it also describes dependencies on other recipes or libraries, as well as configuration and build options. The recipe is stored within the *Layers*.
- Layer: a collection of related recipes. A *layer* is a repository containing relevant metadata that instructs the OpenEmbedded build system how to generate the target. The layering model of the Yocto Project promotes collaboration, sharing, customization, and reuse within the Yocto Project development environment. For example, it is possible to have a kind of "inheritance" of properties by reusing existing *layers* and *recipes*. Layers logically separate your project's information. The layer also contains recipe files, patches, and additional files provided by the user.

2 YOCTO FOR MICROBLAZE

2.1 Poky structure

This project has been developed to use the ARM processor of the **AUBoard 15P Development Kit** board. For this reason, the version of Poky used is the fork of the Langdale project (4.1.4) modified by Xilinx itself.

2.1.1 Some definitions

For this project, all *layers* that have been added to the project beside the default ones are grouped under the folder "`mysources/`". The custom layers are grouped under the folder "`mysources/meta-tria-technologies/`".

This structure is designed to be used across different projects, all of which share a common layer with general settings named "meta-tria-technologies/meta-tria-technologies-bsp". For this reason, this layer will always be included in the "bblayers.conf" 2.2 file.

However, to personalize each individual project, there will be an exclusive custom layer for each of them.

2.1.2 Some usefull variables

The poky tool provide a lot of environment variables used inside all the recipes. This is a brief list of some usefull ones:

- ${B}: The build directory for the recipe. It's where the build process takes place, and compiled files, object files, and other build artifacts are stored.
- ${D}: The destination directory for installing files during the build process. It's where files are installed before packaging.
- ${WORKDIR}: The working directory for the recipe during the build process. It typically holds source code, patches, and other build-related files.
- ${S}: The source directory for the recipe. It's where the source files for the recipe are unpacked.
- ${PN}: The name of the recipe package being built.
- ${PV}: The version of the recipe package being built.
- ${D}${bindir}: The directory where executable files are installed. It's typically "${D}/usr/bin" for the target system.
- ${D}${libdir}: The directory where libraries are installed. It's typically "${D}/usr/lib" for the target system.
- ${D}${includedir}: The directory where header files are installed. It's typically "${D}/usr/include" for the target system.

2.2 Installation

Is recommend using Docker with Ubuntu 18.04, due to specific package dependencies required by the Yocto Langdale release. Without it, some recipes may fail to build properly.

So, before installing Poky on the Linux machine, it is necessary to have docker installed (https://docs.docker.com/engine/install/ubuntu/).

The first step to install Poky is to clone the project on the host machine. The "repo" tool was used (available in the official repositories of Ubuntu 22.04 LTS or in the official site [1]). The Xilinx repository uses a versioning system different from the classic Yocto.

For this project, tag version "xlnx-rel-v2023.1" compatible with the "Kirkstone" and "Langdale" versions of Yocto was used [6] [5].

So be sure to have the following packages installed:

```
sudo apt install docker docker-compose git

# Optional
sudo apt install repo
```

Then do:

```
# Fetch the manifest and checkout the target release version
repo init -u https://github.com/Xilinx/yocto-manifests.git -b "refs/tags/xlnx-rel-v2023.1"

# Fetch all the source from the repositories in the manifest
repo sync

# OPTIONAL: Create a development branch on each repo
repo start <name-brach> --all
```

Once the repository has finished create a docker-compose.yml:

```
# docker-compose.yml
version: '3'
services:
  langdale_xilinx:
    build:
      context: .
      dockerfile: Dockerfile
    volumes:
      - "./:${PWD}"
    working_dir: "${PWD}"
    stdin_open: true
    tty: true
    container_name: langdalexilinx-langdale_xilinx-1
```

and the Dockerfile:

```
# Dockerfile
# Use an existing base image
FROM crops/poky:ubuntu-18.04

# Switch to root user
USER root

# Install ffmpeg and sudo
RUN apt-get update && \
    apt-get install -y ffmpeg sudo

RUN useradd -r -u 1000 pokyuser

USER pokyuser

# Set the working directory
WORKDIR .

ENTRYPOINT ["/bin/bash"]
```

The first time you need to build the container with the follow "`docker-compose`" command:

```
docker-compose up --build
```

When finished to build, ctrl-c to stop the container.

Then create a script:

```
# To run the container
docker-compose up -d

# To attach the console to the new container
docker attach  langdalexilinx-langdale_xilinx-1

# Pressing ctrl-c the above command will end
# and this will shut down the container
docker-compose down
```

Is important to run the script above **inside** the folder you will use.

Upon initial execution, this script will download the Docker image "crops/poky:ubuntu-18.04". Following the download, the script will start the container and attach the console to it.

To stop the container, press Ctrl-C. This will detach the console and shut down the container.

Once the repository download is complete, you can navigate to the poky folder and type the follow command:

```
source setupsdk
```

This command, equivalent to the "`source oe-init-build-env`" command in the original Poky, initializes the Poky development environment. If this command is run for the first time, it generates the Build Directory Tree.

The "bblayers.conf" file configure all the *layers* in use during the compilation, including all the custom layers created.

Furthermore, the "`build/tmp/deploy/`" folder, at the end of the compilation, will contain all the generated files. Every time the container is launched, it is necessary to run it.

2.3 Poky Langdale Xilinx: Initial Setup for the AUBoard

In this section, the minimum settings required to compile a working system for the microblaze board are described.

2.3.1 Custom "meta-aag-auboard" layer

Firstly, generate a new custom *layer* (e.g. meta-aag-auboard) under the "`mysources/meta-tria-technologies/`" folder:

```
mkdir -p mysource/meta-tria-technologies

cd mysource/meta-tria-technologies

bitbake-layers create-layer meta-aag-auboard
```

And then add it to the "bbappend.conf" file manually or with the follow command:

```
bitbake-layers add-layer meta-aag-auboard
```

After that, the file "bblayers.conf" will look like this:

```
BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
    /home/avnet/YoctoPoky/LangdaleXilinx/sources/poky/meta \
    /home/avnet/YoctoPoky/LangdaleXilinx/sources/poky/meta-poky \
    /home/avnet/YoctoPoky/LangdaleXilinx/sources/poky/../meta-xilinx/meta-microblaze \
    /home/avnet/YoctoPoky/LangdaleXilinx/sources/poky/../meta-xilinx/meta-xilinx-core \
    /home/avnet/YoctoPoky/LangdaleXilinx/sources/poky/../meta-xilinx/meta-xilinx-standalone \
    /home/avnet/YoctoPoky/LangdaleXilinx/sources/poky/../meta-xilinx/meta-xilinx-bsp \
    /home/avnet/YoctoPoky/LangdaleXilinx/sources/poky/../meta-xilinx/meta-xilinx-vendor \
    [...]
    ${TOPDIR}/../mysources/meta-tria-technologies/meta-aag-auboard \
"
```

During the creation of the new *layer* it will create a new example *recipe* that can be deleted.

However, these steps are not yet sufficient for compiling a system supported by the AUBoard. Firstly we must create a machines file for the board 2.3.2.

2.3.2 Creating the Machine File for AUBoard 15P Development Kit

To create the machine file for the microblaze, we used the "microblaze-generic.conf" machine file from the Xilinx layer under "meta-xilinx/meta-xilinx-core/conf/machine" as a base and modified it according to our requirements.

We changed the following configuration:

```
KERNEL_IMAGETYPES = ""
```

to:

```
KERNEL_IMAGETYPES += "linux.bin fitImage"
```

Then, we removed the following lines:

```
YAML_DT_BOARD_FLAGS ?= "{BOARD kcu105}"
```

```
KERNEL_EXTRA_ARGS += "UIMAGE_LOADADDR=\textdollar{UBOOT_ENTRYPOINT}"
```

Lastly, we modified:

```
HDF_MACHINE = "kcu105-microblazeel"
```

to:

```
HDF_MACHINE = "microblaze-auboard-15p"
```

2.3.3 Using a Custom XSA Generated with Vivado (AUBoard 15P Development Kit)

By default, if a system is generated for the "AUBoard 15P Development Kit" board using only the *layer* provided by Xilinx, it will fail to load the FPGA.

To compile a working system, it is necessary to use a custom XSA generated by Vivado:

- Create a new project inside Vivado, targeting the AUBoard 15P Development Kit.
- Customize the Vivado project as you need.
- Export the "XSA" (the new "HDF" format) including the bitstream.

The Vivado version must be aligned with the Poky version, so is used the 2023.1 Vivado version.

2.3.4 The external-hdf.bbappend

Once created and exported from "Vivado" in "xsa" format, it is necessary to create a new custom *recipe* for this purpose.

The recipe responsible to compiling the XSA of the AUBoard is named "`recipe-bsp/hdf/external-hdf.bb`" and is contained inside the "meta-xilinx-tools" *layer*.

To modify the *recipe* without altering the original *layer*, you need to recreate within our custom layer the original path of the recipe including the name, changing its extension from bb to bbappend. So the new "bbappend" would be like this:

"`meta-tria-technologies/meta-tria-technologies/recipe-bsp/hdf/external-hdf.bbappend`"

So let's create the new recipe:

```
cd meta-aag-auboard

# Create the tree folder
mkdir -p recipes-bsp/hdf
cd recipes-bsp

# Create the folder that will contain the Vivado generated xsa for the AUBoard
# and put the file inside it (microblaze-auboard-p15\_vX.X.X.xsa)
mkdir -p hdf/files
```

Now create the actual file of the *recipe* that will modify the already existing *recipe*:

```
touch external-hdf.bbappend
```

Our file recipe tree should be like the follow:

```
poky_xilinx/
└── mysource/
    └── meta-tria-technologies/
        └── meta-aag-auboard/
            └── recipes-bsp/
                └── hdf/
                    ├── files/
                    │   └── microblaze-auboard-p15_vX.X.X.xsa
                    └── external-hdf.bbappend
```

Lastly insert the AUBoard XSA inside the "`files/`" folder and write the following text in the external-hdf.bbappend file:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"

HDF_BASE:microblaze-auboard-15p = "file://"
HDF_PATH:microblaze-auboard-15p = "microblaze_auboard/microblaze-auboard-p15_vX.X.X.xsa"
```

2.3.5 The bitstream-extraction Recipe

A bbappend file has been created to modify the existing bitstream-extraction recipe. This modification deploys the `fpga.bin` file into the final output folder:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"

SRC_URI = " \
    file://fpga-bit-to-bin.py \
"

do_deploy:append() {
    if [ -e ${XSCTH_WS}/${XSCTH_PROJ}_hwproj/*.bit ]; then
        ${WORKDIR}/fpga-bit-to-bin.py ${DEPLOYDIR}/${BITSTREAM_BASE_NAME}.bit \
        ${DEPLOYDIR}/${BITSTREAM_BASE_NAME}.bin
        ln -sf ${BITSTREAM_BASE_NAME}.bin ${DEPLOYDIR}/fpga-hdf.bin
    fi
}
```

With these changes, at the end of the compilation process, the `fpga-hdf.bin` file will be present in the deploy folder, ready for use in further steps.

2.3.6 mysource.tar.gz

All layers are provided as a compressed archive containing the `mylayer` directory. Inside this directory, there is also a "configuration" folder that includes the `bblayers.conf` and `local.conf` files specifically configured for the AUBoard project.

To apply these configurations, copy both files into the following path, replacing any existing files:

```
/<poky-global-path>/build/conf/
```

WARNING

This zip contains some linux soft-link and must be extracted from a linux system, otherwise the link will be broken.

2.4 Compilation

Now the minimal configuration to make the compiled system work on AUBoard 15P Development Kit are done.

2.4.1 Setting the Number of Threads for Compilation (optional)

To set the number of threads to be used during *bitbake*, add the following lines to the local.conf file:

```
[...]
BB_NUMBER_THREADS = "8"
PARALLEL_MAKE = "-j 8"
[...]
```

2.4.2 Saving Disk Space (optional)

The "rm_work" class supports the deletion of temporary workspace, which can lighten the demands on the hard disk during compilations. To enable this option, add the following lines to the local.conf file:

```
[...]
INHERIT += "rm_work"
[...]
```

If you are modifying and compiling source code from the working directory for a *recipe*, enabling this option may result in the loss of changes made to the source code. To exclude some recipes from the deletion of work directories, you can add the names of the recipes you are working on to the `RM_WORK_EXCLUDE` variable as follows:

```
[...]
# To exclude recipes from rm_work option
RM_WORK_EXCLUDE += "busybox glibc"
[...]
```

2.4.3 AUBoard 15P Development Kit board target

To compile for a board using Poky, you need the machine configuration files. These files are essential components of Yocto Project or Poky-based build systems, providing detailed specifications and settings tailored to specific embedded hardware platforms.

The machine configuration files serve as blueprints for generating customized target images optimized for the target hardware. They encapsulate various parameters and settings, including CPU architecture, peripheral devices, memory configurations, and supported features, ensuring that the resulting image is compatible and optimized for the target board.

So to compile for the AUBoard, is necessary select the target machine in the "local.conf" file changing the argument of the `MACHINE` variable [1].

```
[...]
MACHINE ?= "microblaze-auboard-15p"
[...]
```

Now, the compilation can begin. From within the Docker container (see Section 2.2), execute the following commands:

```
source setupsdk

bitbake core-image-nano
```

The example above builds the custom image `core-image-nano`, specifically tailored for the AUBoard 15P project (see Section 4.5.2).

The compilation can take several hours, depending on factors such as network speed, the number of CPU cores assigned to the build process, and the overall performance of the host PC.

2.5 Compilation output

Once the compilation has terminated, the folder "`poky_xilinx/build/tmp/deploy/microblaze-auboard-15p`" will be populated by all the compiled files. Inside this folder look for these files:

- core-image-minimal-microblaze-auboard-15p.cpio.xz
- fitImage-microblaze-auboard-15p.bin
- fitImage-only-fs
- fpga-hdf.bin
- u-boot.bin

These file must be flashed with Vivado on the AUBoard qspi flash.

2.6 Conclusion

These steps provide the minimal configuration required to compile a basic and functional Linux system for the AUBoard 15P Development Kit.

However, additional configurations may be required depending on the specific use case of the system. In particular, setting up the HDMI-RX and HDMI-TX subsystems is essential for display functionality. For more details on configuring these components, refer to sections 4.

This setup serves as a solid foundation, allowing further customizations and optimizations based on project requirements.

[1] This machine configuration file is provided by the xilinx layer under the folder `meta-xilinx/meta-xilinx-vendor/conf/machine/microblaze-auboard-15p.conf`

3 CUSTOM LAYERS

The previous chapter outlined the basic configuration required to compile a system for the AUBoard 15P Development Kit. This chapter focuses on the custom layers used in the project.

Within the Yocto Project setup, two distinct custom layers have been defined:

- "meta-tria-technologies-bsp":

  A generic layer that contains common configurations and components shared across multiple projects.

- "meta-aag-auboard":

  A project-specific layer designed for the AUBoard 15P Development Kit. This layer includes the XSA recipe and the machine configuration for the target board as previously seen in 2.3.1.

The following section describes the structure and configuration of the "meta-tria-technologies-bsp" layer.

3.1 Custom images targets

As seen in the 2.4.3 section, to create a complete Linux filesystem we need to use an image target such as "`core-image-minimal`". By default there are a lot of different target images. This images are essentially recipes and can be modified by bbappend.

Is also possible to create new target images recipes inheriting the already existing ones. Follow an example:

```
# core-image-minimal-custom.bb

require recipes-core/images/core-image-minimal.bb

IMAGE_INSTALL += "\
    openssh openssh-sshd openssh-sftp openssh-sftp-server \
    libstdc++ glibc gdb packagegroup-core-tools-debug\
    packagegroup-core-buildessential \
    cmake \
"
```

So this recipes can be used for the compilation using the core-image-minimal as starting point:

```
bitbake core-image-minimal-custom
```

3.1.1 Set default root password

From the recipe of the image target is possible to setup the default password as follow:

```
# core-image-minimal-custom.bb
[...]
inherit extrausers

EXTRA_USERS_PARAMS = " \
    usermod -p '$(openssl passwd 'newpassword')' root; \
"
[...]
```

3.2 Openssh patch

When using a ramdisk system, the use of *openssh* significantly increases the boot time of the board. This is because, being a system based on volatile filesystems, encryption keys must be generated every time the system is powered on.

By default, during boot, OpenSSH generates RSA, ECDSA, and EC25519 keys.

- The RSA key is the most commonly used key and is supported by various systems, providing good security. However, it requires several seconds for key generation.
- The ECDSA key is less commonly used and has some vulnerabilities.

- The EC25519 key, on the other hand, provides good security, and key generation is much faster than RSA.

These keys are generated by the "sshd_check_keys" script within the "openssh_9.0p1.bb" recipe located at "poky/meta/recipes-connectivity/openssh/openssh/".

Therefore, it was decided to keep only the EC25519 key. To implement this change, create a *bbappend* recipe named "openssh_9.0p1.bbappend". This recipe overrides the original file with a local copy of the file in which the key generation commands for RSA and ECDSA keys have been removed.

With the following recipe, only one EC25519 key will be retained:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"

SRC_URI:append = " \
    file://my_sshd_check_keys \
"

do_install:append() {
    install -D -m 0755 ${WORKDIR}/my_sshd_check_keys ${D}${libexecdir}/${BPN}/sshd_check_keys
}

FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"
```

folder tree is the follow:

```
poky_xilinx/
└── mysources/
    └── meta-tria-technologies/
        └── meta-tria-technologies-bsp/
            └── recipes-connectivity/
                └── openssh/
                    ├── openssh_9.0p1.bbappend
                    └── openssh/
                        └── sshd_check_keys
```

3.3 Setting UTF8 locale encoding

To set the correct encoding, it is necessary to insert the following line in "local.conf" or, as in our case, inside the configuration file of a recipe.

`./poky_xilinx/mysources/meta-tria-technologies/meta-tria-technologies-bsp/conf/layer.conf`

```
IMAGE_LINGUAS:append = " en-us en-gb "
```

3.4 Patching recipes with devtool command

Sometimes it is necessary to modify the source code of a *recipes* to fix or add some functionalities. When this need occur, the command "`devtool`" can be used.

- First we must unpack the *recipe* we want to modify with the follow command:

```
devtool modify <recipe-name>

```

The command "devtool modify <recipe-name>" creates a temporary recipe folder in "build/workspace". This command also automatically inserts this new recipe into the "bblayers.conf" file.

In this workspace directory, "devtool" recreates a local copy of the specified recipe's source code, facilitating modifications, testing, and debugging without altering the original sources maintained within the Yocto build system.

The generated recipe folder, located at "build/workspace/sources/<recipe-name>", contains the source code of the recipe. This setup provides an isolated environment to make and track changes to the recipe's source code, ensuring that the original source files remain intact.

- Now the *recipes* can be modified directly inside the folder generated. The changes can be tested compiling the *recipe*:

```
devtool build <recipe-name>

```

- If all the modifications are done we can must create a commit inside the *recipes* folder.

```
git add .
git commit -m "recipe patch 1"

```

- Finally, with devtool finalise the operation:

```
devtool finish <recipe-name> /path/to/your/custom/meta-layer

```

This command will create a file ".patch" inside your custom *layer*.

This last command also remove the source code from the source folder and move it under "`build/workspace/attic`". This is only a backup if you need to review changes and **It can be deleted safely**.

Once finished the recipe patch you can also **remove safely** the workspace recipe from the "bblayers.conf" config file config file config file.

3.4.1 The udmabuf driver (out-of-tree)

For the udmabuf driver, instead of modify the kernel source, it was created a *recipe* to build a out-of-tree module [3] to include inside the filesystem in use.

udmabuf-mod_v4.5.3.bb:

```
SUMMARY= "A Linux device driver that allocates contiguous memory blocks in the kernel space as DMA buffers"

LICENSE="BSD-2-Clause"
LIC_FILES_CHKSUM = "file://LICENSE;md5=bebf0492502927bef0741aa04d1f35f5"


inherit module

SRC_URI = " \
    file://LICENSE \
    file://Kconfig \
    file://Makefile \
    file://u-dma-buf.c \
"

S = "${WORKDIR}"

# The inherit of module.bbclass will automatically name module packages with
# "kernel-module-" prefix as required by the oe-core build environment.

RPROVIDES_${PN} += "kernel-module-udmabuf"
```

Follow the directory tree:

```
poky_xilinx/
└── mysource/
    └── meta-tria-technologies/
        └── meta-tria-technologies-bsp/
            └── recipes-kernel/
                └── linux-xlnx/
                    └── udmabuf-mod/
                        ├── udmabuf-mod_v4.5.3.bb
                        └── files/
                            ├── LICENSE
                            ├── Kconfig
                            ├── Makefile
                            └── u-dma-buf.c
```

4 AUBOARD

This chapter will focus on the project, including the **meta-aag-auboard** layer structure, board settings, and the Yocto layer.

4.1 Board Settings and Structure

The AUBoard board features only a 64MB QSPI flash.

The primary boot method used in production involves loading the `boot.bin`, FPGA configuration, filesystem, and kernel from the QSPI.

4.1.1 QSPI

The QSPI is 64 MB and is partitioned as follows:

Address	Description	Filename	Size	Size kB
0x4000000	—	—	—	—
0x0C00000	Ramdisk	core-image-nano-microblaze-auboard-15p.cpio.xz.u-boot	0x03400000	53 248 kB
0x0700000	Linux Kernel [2]	fitImage-microblaze-auboard-15p.bin	0x00500000	5120 kB
0x0600000	Bootloader	boot.bin	0x00100000	1024 kB
0x0000000	FPGA Bitstream	fpga-hdf.bin	0x00600000	6144 kB

Table 1: QSPI Memory Mapping for Partitions

4.2 The meta-aag-auboard layer

As mentioned in section 3.1, each project will include the generic "meta-tria-technologies-bsp" layer and its specific custom layer. This custom layer is located in the "`mysource/meta-tria-technologies/`" folder.

Thus, the "meta-aag-auboard" is created.

Follow the folder structure of the layer:

[2] The device-tree is inside the generated fitimage along with the kernel image

```
meta-tria-technologies
└── meta-aag-auboard
    ├── auboard-system
    ├── conf/
    │   ├── layer.conf
    │   └── machine/
    │       └── microblaze-auboard-15p.conf
    ├── recipes-bsp/
    │   ├── device-tree/
    │   ├── hdf/
    │   └── u-boot/
    ├── recipes-core/
    │   ├── base-files/
    │   ├── fs-version/
    │   └── images/
    ├── recipes-devtools/
    │   └── gcc/
    ├── recipes-kernel/
    │   ├── hdmi/
    │   ├── hdmi_tx_solo_working.patch
    │   ├── linux/
    │   └── modprobe/
    ├── recipes-multimedia/ .4 custom-edid/
    │   └── gstreamer/
    ├── recipes-qt/
    │   ├── microblaze-qt-test/
    │   └── qt5/
    └── recipes-utils/
        ├── get-system-version/
        └── scripts-utils/
```

4.3 Configurations folder

Inside the `meta-tria-technologies/configurations` there are the `local.conf` and the `bblayers.conf` files for the current project.

4.4 The recipes-bsp recipe

The "recipes-bsp" folder contains all the recipes and metadata necessary to build images for a specific hardware platform or board (BSP).

In this project, there are tree recipes:

- u-boot
- device-tree
- hdf

4.4.1 U-Boot

The U-Boot recipe allows modifications to its configurations, enabling customization of boot parameters and hardware support.

The U-Boot 'bbappend' recipe directory structure is as follows:

```
recipes-bsp/
└── u-boot/
    └── u-boot-xlnx/
        └── eth.cfg
```

```
    │   └── mb_env_offsets.cfg
    └── u-boot-xlnx_%.bbappend
```

The file "u-boot-xlnx_%.bbappend" applies additional configurations during the compilation process. For instance, it enables support for the KSZ8091 Ethernet PHY chip.

Additionally, the file "mb_env_offsets.cfg" modifies specific register values for boot commands. Below is an example of the boot configuration:

u-boot-xlnx.bbappend:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"

SRC_URI:append = " \
    file://mb_env_offsets \
    file://eth.cfg \
"
# RAM offsets
RAM_KERNEL_OFFSET ?= "0x82000000"
RAM_RAMDISK_OFFSET ?= "0x82500000"

# QSPI offsets
QSPI_KERNEL_OFFSET ?= "0x700000"
QSPI_RAMDISK_OFFSET ?= "0xC00000"

# QSPI read size
QSPI_KERNEL_READ_SIZE ?= "0x500000"
QSPI_RAMDISK_READ_SIZE ?= "0x1200000"

# Boot delay in seconds
CONFIG_BOOTDELAY ?= "3"

do_compile:prepend() {
    sed -e "s:@@RAM_KERNEL_OFFSET@@:${RAM_KERNEL_OFFSET}:g" \
        -e "s:@@RAM_RAMDISK_OFFSET@@:${RAM_RAMDISK_OFFSET}:g" \
        -e "s:@@QSPI_KERNEL_OFFSET@@:${QSPI_KERNEL_OFFSET}:g" \
        -e "s:@@QSPI_RAMDISK_OFFSET@@:${QSPI_RAMDISK_OFFSET}:g" \
        -e "s:@@QSPI_KERNEL_READ_SIZE@@:${QSPI_KERNEL_READ_SIZE}:g" \
        -e "s:@@QSPI_RAMDISK_READ_SIZE@@:${QSPI_RAMDISK_READ_SIZE}:g" \
        -e "s:@@CONFIG_BOOTDELAY@@:${CONFIG_BOOTDELAY}:g" \
        "${WORKDIR}/mb_env_offsets" >> "${S}/../microblaze-generic.cfg"
}
```

4.4.2 The device-tree

The device tree is typically generated from two types of files: the DTS (Device Tree Source) and the DTSI (Device Tree Source Include). The former is the generic device tree source code and can be compiled as is. The latter is also a device tree source file but is meant to be included by other DTS files and cannot be compiled as is [3].

This recipe contains the following DTSI file `system-user.dtsi`:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"

SYSTEM_USER_DTSI ?= "system-user.dtsi"

SRC_URI:append:microblaze-auboard-15p = " \
    file://${SYSTEM_USER_DTSI} \
"

do_configure:append:microblaze-auboard-15p() {
    # User defined device tree
    cp ${WORKDIR}/${SYSTEM_USER_DTSI} ${B}/device-tree
    echo "/include/ \"${SYSTEM_USER_DTSI}\"" >> ${B}/device-tree/system-top.dts
}
```

In summary, this configuration ensures that the user's custom device tree configurations are properly integrated into the build process, resulting in a customized "system-top.dts" device tree for the AUBoard. These changes are made by the configuration to the "system-top.dts" device tree:

```
    [...]
```

[3] They are device tree fragments that contain reusable sections of device tree source code shared among multiple devices or system configurations. Including common configurations in separate DTSI files helps in modularizing and organizing the device tree source code.

```
};
/include/ "microblaze-auboard-15p.dtsi"
/include/ "system-user.dtsi"
```

4.4.3 The external_hdf recipe

As described in the Section 2.3.4, we create a bbappend recipe "hdf/external_hdf.bbappend" to use our custom "xsa" for the microblaze-auboard-15p:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"

HDF_BASE:microblaze-auboard-15p = "file://"
HDF_PATH:microblaze-auboard-15p = "microblaze_auboard/microblaze-auboard-p15_vX.X.X.xsa"
```

The complete path for the xsa file is:

/path/to/mysources/meta-tria-technologies/meta-aag-auboard/recipes-bsp/hdf/files/microblaze-auboard-p15_vX.X.X.xsa

4.5 The recipes-core group

The recipes-core directory contains core recipes essential for building the basic system. These recipes define fundamental components required to generate a minimal and functional base image.

```
[...]
└── recipes-core/
    ├── base-files/
    ├── fs-version/
    └── images/
```

4.5.1 The fs-version and base-files recipes

fs-version:

A simple recipe designed to embed the release version number and changelog into the compiled filesystem.

base-files:

A 'bbappend' recipe that configures the hostname of the target filesystem, setting it to "microblaze-auboard-15p".

4.5.2 The images recipes target

These recipes define the packages, libraries, and configurations that will be included in the final target image.

Within **meta-tria-technologies-bsp**, we have created a custom lightweight Linux image, "core-image-nano.bb", designed to be as small as possible.

```
DESCRIPTION = "A custom ultra-minimal image"
LICENSE = "MIT"

inherit core-image
inherit extrausers

# Set root password securely
EXTRA_USERS_PARAMS = " \
    usermod -p '$(openssl passwd 'root')' root; \
"

# Set basic functionality
#IMAGE_INSTALL:append = " busybox tiny-init"
IMAGE_INSTALL:append = " bootchart2 grep rsync dropbear findutils "

# Distribution and image features
DISTRO_FEATURES:remove = "largefile nls opengl ptest multiarch glibc-gconv glibc-locale sysvinit-inittab"
EXTRA_IMAGE_FEATURES = ""

```

```
# Disable RPM post-install scripts
#IMAGE_FEATURES:remove = "package-management"

# Exclude unnecessary packages to reduce size
IMAGE_INSTALL:remove = "packagegroup-base-extended"
PACKAGE_EXCLUDE:append = " rng-tools git python3 gdb cmake glibc"
#udev

# Fonts
IMAGE_INSTALL:append = " cantarell-fonts"

# Install minimal requirements
IMAGE_INSTALL:append = " util-linux devmem2"

IMAGE_FEATURES:remove = " tools-debug tools-profile tools-sdk debug-tweaks"

# Disable locale generation
IMAGE_LINGUAS = ""
GLIBC_GENERATE_LOCALES = ""

# Define filesystem formats and exclude wic
IMAGE_FSTYPES = "cpio.gz cpio.xz cpio.gz.u-boot tar.gz"
IMAGE_FSTYPES:append = " tar.xz ext4.xz cpio.xz.u-boot"
IMAGE_FSTYPES:remove = " wic wic.gz"

# Compression settings for a smaller root filesystem
IMAGE_COMPRESSION_LEVEL = "9"

# Remove /boot content to save space
ROOTFS_POSTPROCESS_COMMAND:append = " remove_boot_folder; "

remove_boot_folder() {
    if [ -d ${IMAGE_ROOTFS}/boot ]; then
        rm -rf ${IMAGE_ROOTFS}/boot/*
        find "${IMAGE_ROOTFS}/boot" -type f ! -name '*.dtb' -delete
        echo "" > ${IMAGE_ROOTFS}/boot/uImage
    fi
}

# Ensure locale-archive is removed from the final image
ROOTFS_POSTPROCESS_COMMAND:append = " remove_locale_archive; "

remove_locale_archive() {
    rm -f ${IMAGE_ROOTFS}/usr/lib/locale/locale-archive
    rm -rf ${IMAGE_ROOTFS}/etc/rpm-postinsts
}
```

And then, we modify it specifically for the AUBoard by using a "bbappend" recipe inside the **meta-aag-auboard** layer. This ensures that board-specific configurations, package selections, and optimizations are applied while maintaining compatibility with the base image defined in "meta-tria-technologies-bsp".

4.6 The recipes-kernel group

This "bbappend" recipe is responsible for enabling all the necessary kernel drivers required for the project, as well as applying patches to the source kernel code. It includes a set of ".cfg" files that define the kernel configurations, ensuring that the required features and modules are correctly enabled.

This is the recipe directory tree:

```
recipes-kernel/
├── hdmi/
│   ├── kernel-module-hdmi/
│   │   └── 0001-Added-support-for-microblaze-32bit.patch
│   └── kernel-module-hdmi_%.bbappend
├── hdmi_tx_solo_working.patch
├── linux/
│   ├── linux-xlnx/
│   │   ├── 0004-fix-for-xilinx-hdmi-tx-with-idt8t49n24x.patch
│   │   ├── boot-options.cfg
│   │   ├── bump_version.sh
│   │   ├── cripto.cfg
│   │   ├── debug.cfg
│   │   ├── default-minimal.cfg
│   │   ├── ethlite.cfg
│   │   ├── filesystem.cfg
│   │   ├── fitImage.its.in
│   │   ├── fpga-xlnx.cfg
│   │   ├── hdmi_tx.cfg
│   │   ├── hid.cfg
│   │   ├── i2c.cfg
│   │   ├── idt8t49n240.cfg
│   │   ├── krnl_version.cfg
│   │   ├── leds.cfg
│   │   ├── mtd_ubi.cfg
│   │   ├── net.cfg
│   │   ├── pci_express.cfg
│   │   ├── test.cfg
│   │   └── uartlite.cfg
│   └── linux-xlnx_%.bbappend
└── modprobe/
    ├── files/
    │   └── blacklist-hdmi.conf
    └── modprobe-blacklist.bb
```

4.6.1 Kernel Driver Configurations

This section contains the Description of all the necessary ".cfg" files for this project, used to configure the kernel during compilation.

```
inherit kernel-fitimage
KERNEL_IMAGETYPE = "fitImage"
BOOT_IMAGE_TYPE = "image.ub"

FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"

#file://fitImage.its.in
SRC_URI += " \
    file://krnl_version.cfg \
    file://boot-options.cfg \
    file://fpga-xlnx.cfg \
    file://mtd_ubi.cfg \
    file://net.cfg \
    file://i2c.cfg \
    file://uartlite.cfg \
    file://cripto.cfg \
    file://ethlite.cfg \
    file://filesystem.cfg \
    file://hid.cfg \
    file://leds.cfg \
    file://idt8t49n240.cfg \
    file://hdmi_tx.cfg \
    file://pci_express.cfg \
    file://debug.cfg \
"
```

Each .cfg file included in the "SRC_URI" specifies a set of kernel configuration options that are necessary for the corresponding hardware components or features. The "FILESEXTRAPATHS" variable is used to ensure that the build system can locate these configuration files within the recipe directory structure.

4.6.2 The kernel-module-hdmi Out-of-Tree Driver

In this project, we included the out-of-tree module recipe "kernel-module-hdmi" inside the **core-image-nano.bbappend**. However, this driver was originally designed for **Versal** and **ZynqMP** boards. As a result, when compiling this driver with **microblaze-auboard-p15** as the target board, the build process failed due to incompatibilities.

64-bit Division Issue

We found the out-of-tree Xilinx HDMI module for Yocto, but it was only compatible with **Versal** and **ZynqMP** architectures. During the build process, the compilation failed because **MicroBlaze (32-bit) lacks support for 64-bit division operations**. The following errors were encountered:

```
ERROR: modpost: "__umoddi3" [/<poky-path>/build/workspace/sources/kernel-module-hdmi/hdmi/xilinx-vphy.ko] undefined!
ERROR: modpost: "__udivdi3" [/<poky-path>/build/workspace/sources/kernel-module-hdmi/hdmi/xilinx-vphy.ko] undefined!
ERROR: modpost: "__udivdi3" [/<poky-path>/build/workspace/sources/kernel-module-hdmi/hdmi/xilinx-hdmi-rx.ko] undefined!
ERROR: modpost: "__udivdi3" [/<poky-path>/build/workspace/sources/kernel-module-hdmi/hdmi/xilinx-hdmi-tx.ko] undefined!
```

To resolve the 64-bit division issue, we used an implementation from an external repository [2].

By incorporating this solution, we were able to **bypass the 64-bit division limitation**, allowing the module to compile successfully.

Recipe Patch Creation

To patch this recipe, we used the **devtool** commands to modify the driver's source code:

```
devtool modify kernel-module-hdmi
```

A new temporary layer is created under "/<yocto-poky-path/build/workspace/" and appended at the bottom of the **bbappend** file inside the **bblayers** configuration:

```
BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
    [...]
    ${TOPDIR}/../mysources/meta-qt5-extra \
    ${TOPDIR}/../mysources/meta-tria-technologies/meta-tria-technologies-bsp \
    ${TOPDIR}/../mysources/meta-tria-technologies/meta-aag-auboard \
    /home/avnet/YoctoPoky/LangdaleXilinx/build/workspace \
"
```

Within this new layer, the source code of the kernel module recipe is extracted under the `workspace/sources/kernel-module-hdmi` and used during compilation until the patching process is complete.

Next, we included the modified source file inside the driver and finalized the patch using the following command:

```
git add .
git commit -m "Added support for microblaze 32bit"
devtool finish kernel-module-hdmi /path/to/yocto/poky/mysource/meta-tria-technologies/meta-aag-auboard
```

This modifies the existing **kernel-module-hdmi_%.bbappend** file as follows:

```
[...]
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"

SRC_URI += " \
    [...]
    file://0001-Added-support-for-microblaze-32bit.patch \
"
```

Finally, we also add the compatibility with the **microblaze** in the same file as before:

```
COMPATIBLE_MACHINE:microblaze-auboard-15p = "microblaze-auboard-15p"
[...]
```

4.6.3 IDT8T49N240 Issue and Kernel Patch

While trying to get the HDMI-RX and HDMI-TX blocks working, we discovered that the **IDT8T49N240** clock chip requires a specific register configuration. During the board's initial boot sequence, the **0x70** register must be set to **0x05**. After the HDMI-RX and HDMI-TX drivers are probed, we must then revert **0x70** back to **0x00**. This sequence ensures that both the clock chip and the HDMI drivers are properly configured.

Similarly to the previous modifications, we needed to patch the Linux kernel to accommodate this behavior. We again used the **devtool** command to modify the **linux-xlnx** kernel:

```
devtool modify linux-xlnx
```

This command extracts the kernel source code into "/path/to/yocto/build/workspace/sources/linux-xlnx/". Within this source, we added a new function in the **IDT8T49N240** driver that writes a single register value. We then **exported** this function so the HDMI-TX driver can call it when switching **0x70** from **0x05** to **0x00** at the correct time.

After completing our modifications, we committed the changes:

```
git add .
git commit -m "Fix for Xilinx HDMI-TX with IDT8T49N24x"
```

Finally, we ran the following command to integrate our patch back into the relevant layer:

```
devtool finish linux-xlnx ../mysources/meta-tria-technologies/meta-aag-auboard
```

With this patch, the IDT8T49N240 clock chip is correctly initialized, and the HDMI-RX and HDMI-TX drivers can reliably operate without timing or configuration conflicts.

This modifies the existing **linux-xlnx_%.bbappend** file as follows:

```
[...]
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"

SRC_URI += " \
    [...]
    file://NNNN-fix-for-xilinx-hdmi-tx-with-idt8t49n24x.patch \
"
```

4.6.4 HDMI demo script

4.6.5 HDMI-TX Driver RGB Color Issue

During the tests of the system, we discovered a format compatibility issue between the HDMI-RX and HDMI-TX IP blocks regarding framebuffer color formats. Specifically, the HDMI-RX block only accepts the `bgr888` format in the device tree, whereas the HDMI-TX block only accepts `rgb888`.

By default, the Xilinx framebuffer driver (`frmbuf`) includes a `rgb888` entry that internally sets the hardware to a **BGR** color pipeline. This behavior introduces a mismatch when one IP expects RGB-formatted pixel data while the other expects **BGR**.

Why the Mismatch?

- **HDMI-RX block**: Accepts only `bgr888` in the device tree.
- **HDMI-TX block**: Accepts only `rgb888` in the device tree.

Role of XILINX_FRMBUF_FMT_*

Each format entry in the driver includes an id field such as XILINX_FRMBUF_FMT_RGB8 or XILINX_FRMBUF_FMT_BGR8. This identifier is the actual value written to the framebuffer IP's register, directly controlling how the IP interprets incoming pixel data. Therefore, customizing this value is crucial for aligning color formats across different IP blocks.

Creating a Custom Format

To resolve this format conflict, we added a new custom format entry named bgr888_auboard to xilinx_frmbuf.c. This entry is a modified copy of the default rgb888 definition, but with the internal format ID switched from XILINX_FRMBUF_FMT_BGR8 to XILINX_FRMBUF_FMT_RGB8, ensuring genuine **RGB** behavior in hardware.

Existing (Driver-Provided) rgb888 Entry:

```
{
    .dts_name   = "rgb888",
    .id         = XILINX_FRMBUF_FMT_BGR8,    /* Note: BGR8 here! */
    .bpw        = 24,
    .ppw        = 1,
    .num_planes = 1,
    .drm_fmt    = DRM_FORMAT_RGB888,
    .v4l2_fmt   = V4L2_PIX_FMT_BGR24,
    .fmt_bitmask = BIT(18),
},
```

New bgr888_auboard Entry:

```
{
    .dts_name   = "bgr888_auboard",
    .id         = XILINX_FRMBUF_FMT_RGB8,    /* Switched to RGB8 */
    .bpw        = 24,
    .ppw        = 1,
    .num_planes = 1,
    .drm_fmt    = DRM_FORMAT_RGB888,
    .v4l2_fmt   = V4L2_PIX_FMT_BGR24,
    .fmt_bitmask = BIT(18),
},
```

Applying the Patch with devtool

First, we used devtool modify to extract the kernel source into a temporary workspace:

```
devtool modify linux-xlnx
```

After editing xilinx_frmbuf.c to add our custom format entry, we committed these changes locally:

```
git add .
git commit -m "Add bgr888_auboard format for HDMI color alignment"
```

Finally, we ran devtool finish to integrate our modifications back into the relevant layer:

```
devtool finish linux-xlnx ../mysources/meta-tria-technologies/meta-aag-auboard
```

Device Tree Update

With the new format defined, we updated the device tree to apply it where the framebuffer read (v_frmbuf_rd) is instantiated:

```
hdmi_output_v_frmbuf_rd_0: v_frmbuf_rd@44a40000 {
    status = "okay";
    clocks = <&clk_cpu>;
    clock-names = "ap_clk";
    reset-gpios = <&gpio2 2 1>; // GPIO_ACTIVE_LOW=1; GPIO_ACTIVE_HIGH=0
    xlnx,vid-formats = "bgr888_auboard";
};
```

This ensures that both the HDMI-RX and HDMI-TX IP blocks receive their expected color format, eliminating any channel-swapping or color inversion and resulting in correct video output.

4.7 The recipes-demo group

As part of our custom layer `meta-aag-auboard`, we created a `recipes-demo` directory to host demonstration scripts and utilities. One of these demonstrations is the `hdmi-screenshot-demo-script` recipe, which showcases HDMI capture and display functionality, along with a utility to convert raw framebuffer data to a bitmap image.

The directory structure for this demo is as follows:

```
hdmi-screenshot-demo-script/
└── hdmi-screenshot-demo-script/
    ├── hdmi_demo.sh
    ├── hdmi-screenshot-demo-service
    └── raw2bmp.c
└── hdmi-screenshot-demo-script.bb
```

The `hdmi-screenshot-demo-script.bb` recipe is shown below:

```
DESCRIPTION = "AUBoard HDMI demo script"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/files/common-licenses/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
FILES:${PN} += " ${sysconfdir}/init.d/hdmi-screenshot-demo-service ${sbindir}/raw2bmp "

PR = "r0"

S = "${WORKDIR}"

SRC_URI = " \
    file://hdmi-screenshot-demo-service \
    file://hdmi_demo.sh \
    file://raw2bmp.c \
"

do_compile() {
    ${CC} ${CFLAGS} ${LDFLAGS} raw2bmp.c -o raw2bmp
}

do_install() {
    install -d ${D}${sysconfdir}/init.d
    install -d ${D}${sysconfdir}/rcS.d
    install -d ${D}${sbindir}/

    install -m 0755 raw2bmp ${D}${sbindir}/raw2bmp
    install -m 0755 ${WORKDIR}/hdmi_demo.sh ${D}${sbindir}/
    install -m 0755 ${WORKDIR}/hdmi-screenshot-demo-service ${D}${sysconfdir}/init.d/
    ln -sf ../init.d/hdmi-screenshot-demo-service ${D}${sysconfdir}/rcS.d/S99hdmi-screenshot-demo-service
}
```

Utility Files Overview

- **`raw2bmp.c`**: A small C program that reads raw 24-bit RGB data and outputs it as a 32-bit BMP (with an unused alpha channel). Although standard BMP files do support 24-bit color, certain decoders or pipelines can mistakenly interpret the pixel ordering, leading to inverted red and blue channels. Adding the extra 8 bits for alpha avoids this confusion, ensuring correct color alignment in the final image.
- **`hdmi-screenshot-demo-service`**: A SysV-style init script that automatically launches the main HDMI demo script on system startup.
- **`hdmi_demo.sh`**: The primary demo script that captures HDMI input frames, writes them to the local framebuffer, and converts the raw data into a BMP file. Additionally, it copies this BMP data into `/dev/udmabuf0`, making it accessible to a host PC over PCIe for further analysis or archiving.

Demo Script Explanation

This script configures GPIO to detect a button press, sets the video capture format (`RGB3`) via `v4l2-ctl`, grabs one frame from the HDMI input, and displays it on the local framebuffer. It then converts the raw capture to a

BMP file and places that file in /dev/udmabuf0 for further hardware use.

When the script detects a button press on GPIO pin 506, it captures a frame, places it onto the local display (/dev/fb0), and creates a BMP snapshot for further processing or storage. By copying the BMP file into /dev/udmabuf0, the captured image is also accessible to a host PC through the PCIe interface, ensuring easy integration and analysis with external systems.

4.8 Compilation

To begin the build process (refer to Section 2.4.3 for board configuration details), run:

```
bitbake core-image-nano
```

4.8.1 Deployment

Once the compilation completes, all generated files are placed under the directory:
"/<poky-global-path>/build/tmp/deploy/images/AAG_auboard/microblaze-auboard-15p/"

Some items in this directory are symbolic links.

Below are the main artifacts located in the deploy folder:

1. boot.bin
2. uImage
3. core-image-nano-microblaze-auboard-15p.cpio.tar.xz.u-boot
4. fpga-hdf.bin

These files include the bootloader (boot.bin), kernel (uImage), root filesystem (core-image-nano-microblaze-auboard-15p.cpio.tar.xz.u-boot), and the FPGA bitstream (fpga-hdf.bin), providing everything needed to deploy and run your custom image on the target board.

This output path is defined by the DEPLOY_DIR_IMAGE variable within the configuration file of the meta-aag-auboard layer:

```
/<poky-global-path>/mysources/meta-tria-technologies/meta-aag-auboard/conf/layer.conf
```

The relevant line is:

```
DEPLOY_DIR_IMAGE = "${TOPDIR}/tmp/deploy/images/AAG_auboard/${MACHINE}"
```

If you wish to change the output location, simply modify this variable inside the layer.conf file accordingly.

4.8.2 Generating the mcs file

Now to write into the auboard flash, you need to create a msc file with all of these. To do so we use the "**srec_cat**" tool.

If this tool isn't installed yet, install with:

```
sudo apt install srecord
```

Then create the msc file:

```
srec_cat \
  fpga-hdf.bin -binary -offset 0x0000000 \
  u-boot.bin -binary -offset 0x0600000 \
  fitImage-microblaze-auboard-15p.bin -binary -offset 0x0700000 \
  core-image-nano-microblaze-auboard-15p.cpio.xz.u-boot -binary -offset 0x0C00000 \
  -o final_image.mcs -intel
```

Finally you need to flash into the auboard QSPI using Vivado.

4.8.3 Programming the Flash Memory with Vivado

Once the final `.mcs` image has been generated (for example, `final_image.mcs`), it can be written to the AUBoard's SPI flash using Vivado's Hardware Manager interface. The procedure is detailed below.

1. Open Vivado and select **Open Hardware Manager** from the main interface:

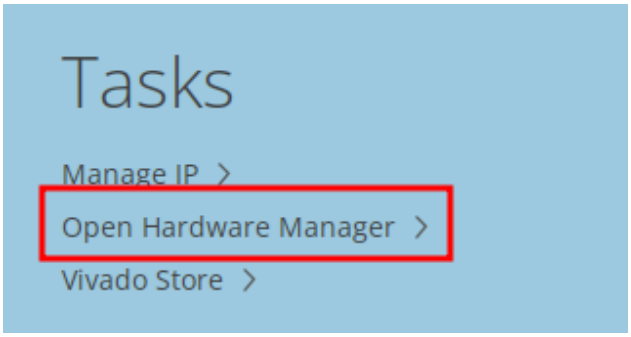


Figure 2: Vivado — Open Hardware Manager

2. In the Hardware Manager, click on **Open Target** (in the top bar), then choose **Auto Connect** from the drop-down menu:

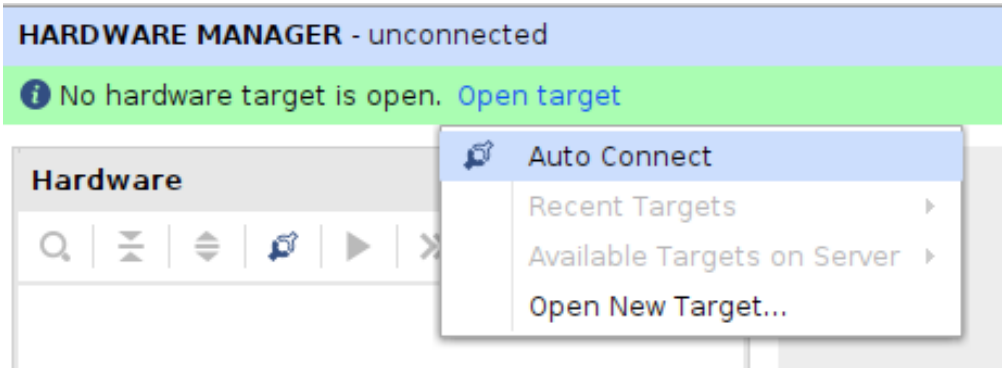


Figure 3: Vivado — Auto Connect to Target

3. Once the target is connected, right-click on `xcau15p_0 (2)` and select **Add Configuration Memory Device...**:

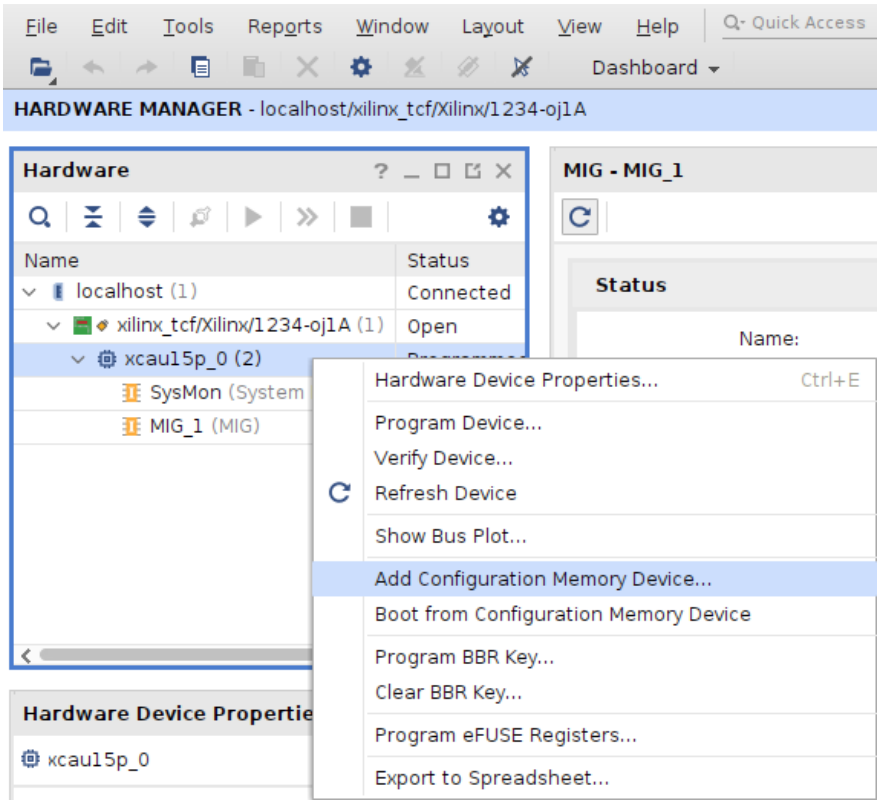


Figure 4: Vivado — Add Configuration Memory Device

4. A new window will appear. Search for the memory part number `is25wp512m-spi-x1_x2_x4`, select it, and click **OK**:

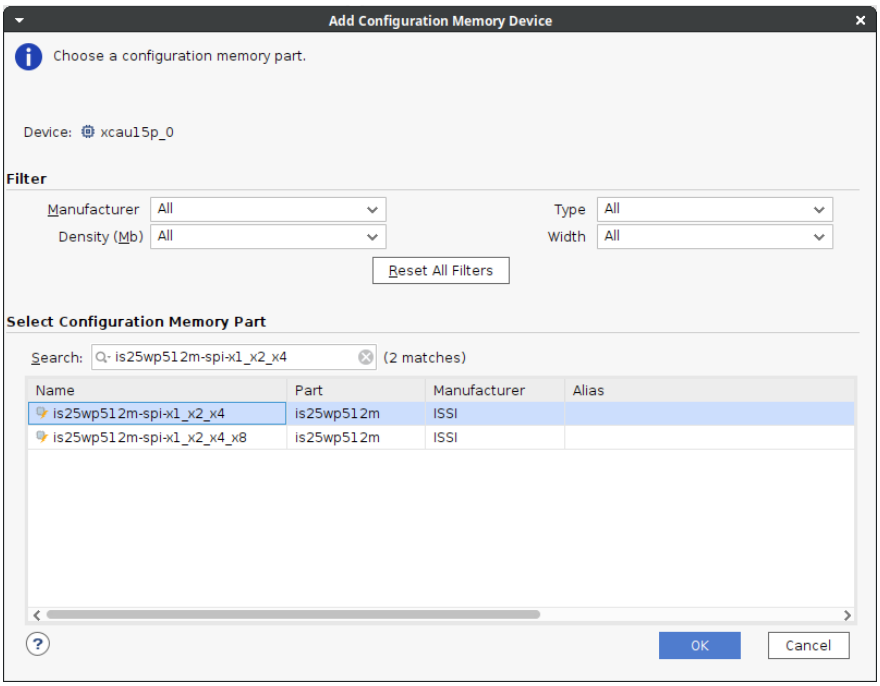


Figure 5: Vivado — Select Flash Memory Part

5. Right-click on `is25wp512m-spi-x1_x2_x4` and choose **Program Configuration Memory Device...**:

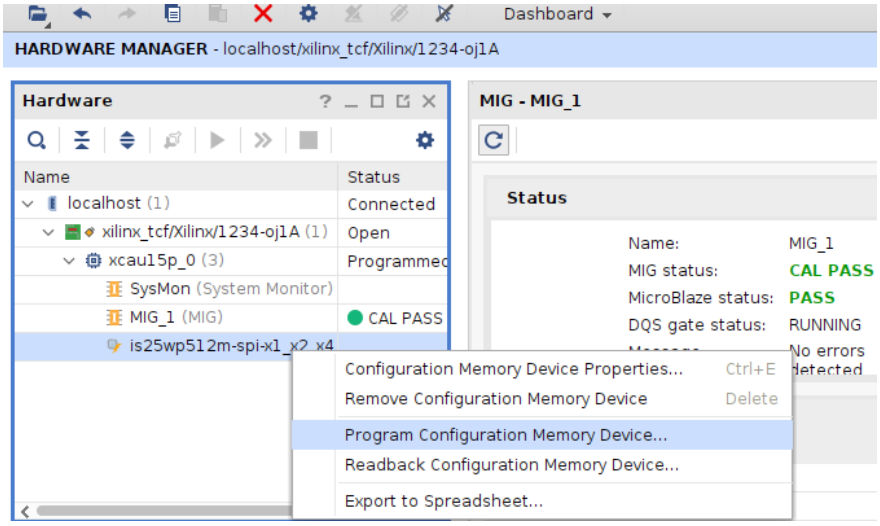


Figure 6: Vivado — Program Flash Memory

6. In the dialog that opens, select the `final_image.mcs` file as the image to be flashed and press **OK**.

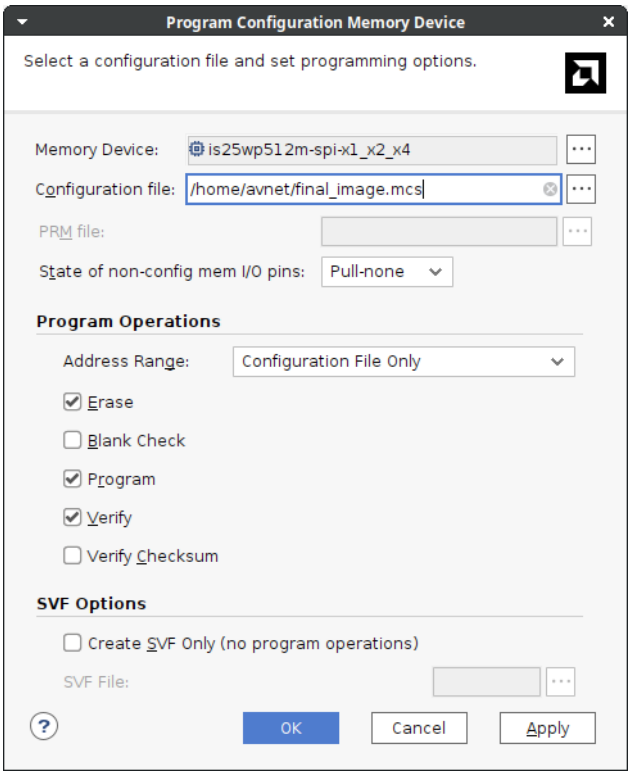


Figure 7: Vivado — Select the msc file

At this point, the programming process will begin. Wait until the process completes successfully before disconnecting the board or closing Vivado.

4.9 Vivado project

The Vivado project contains the hardware layer supporting the HDMI and PCIe functionalities. It has been carried out using Vivado 2023.1.

The block design includes the MicroBlaze 32-bit soft processor, which supports an instruction set optimized for embedded applications with many user-configurable options. In addition to other features, it provides the AXI interface and Memory Management Unit.

The reference design includes the IPs required to support the HDMI demo, as shown in figure 8 (see the Paragraph 4.9.1 for HDMI IP license). The chain consists in:

- A video frame buffer `v_frmbuf_rd` providing high bandwidth access to handle video frame read from memory;
- The HDMI Transmitter Subsystem IP `v_hdmi_tx_ss`, incorporating all the necessary logic to interface with PHY layers and provide HDMI™ encoding functionality properly. The subsystem takes incoming video and provides transfer to the PHY layer;
- The Video PHY Controller IP `vid_phy_controller`, to enable plug-and-play connectivity with Video MAC transmit or receive subsystems;
- The HDMI Receiver subsystem `v_hdmi_rx_ss`, that receives the captured data from the PHY layer. It then extracts and converts the video streams from the HDMI stream.
- A video frame buffer `v_frmbuf_wr` providing high bandwidth access to handle video frame write to memory;

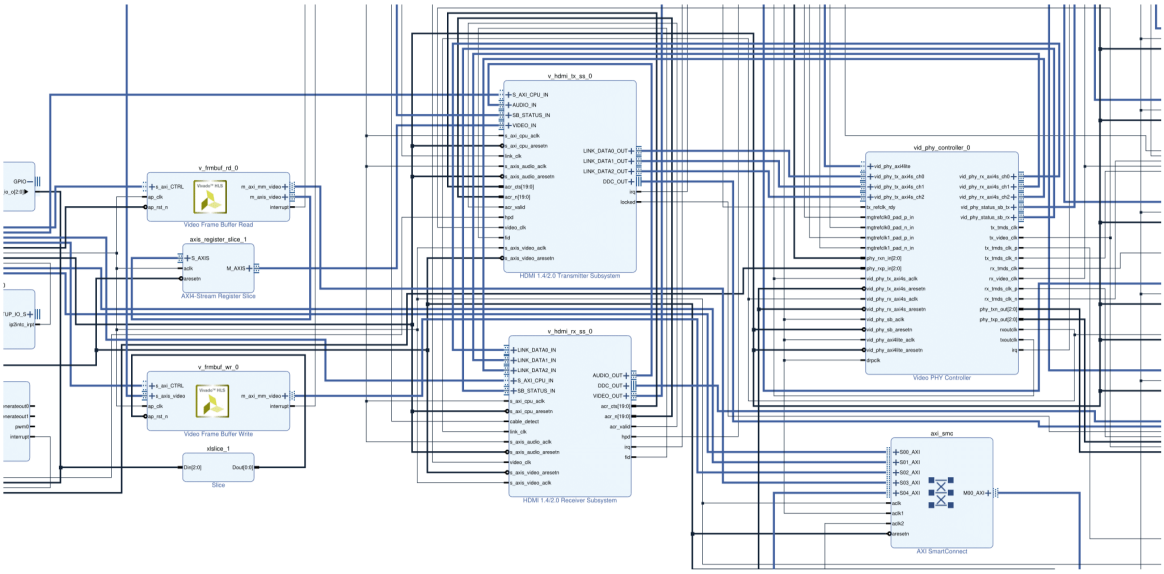


Figure 8: HDMI receiver/transmitter system in block design

The block design also includes the support for PCIe Express (figure 9). This feature is provided by the DMA/Bridge Subsystem IP for PCIe `xdma`, which has been configured as a high-performance direct memory access (DMA) data mover. In this way, the host can directly access the user logic through the AXI4-Lite Master Configuration port or the AXI Memory Mapped Master Bypass port.

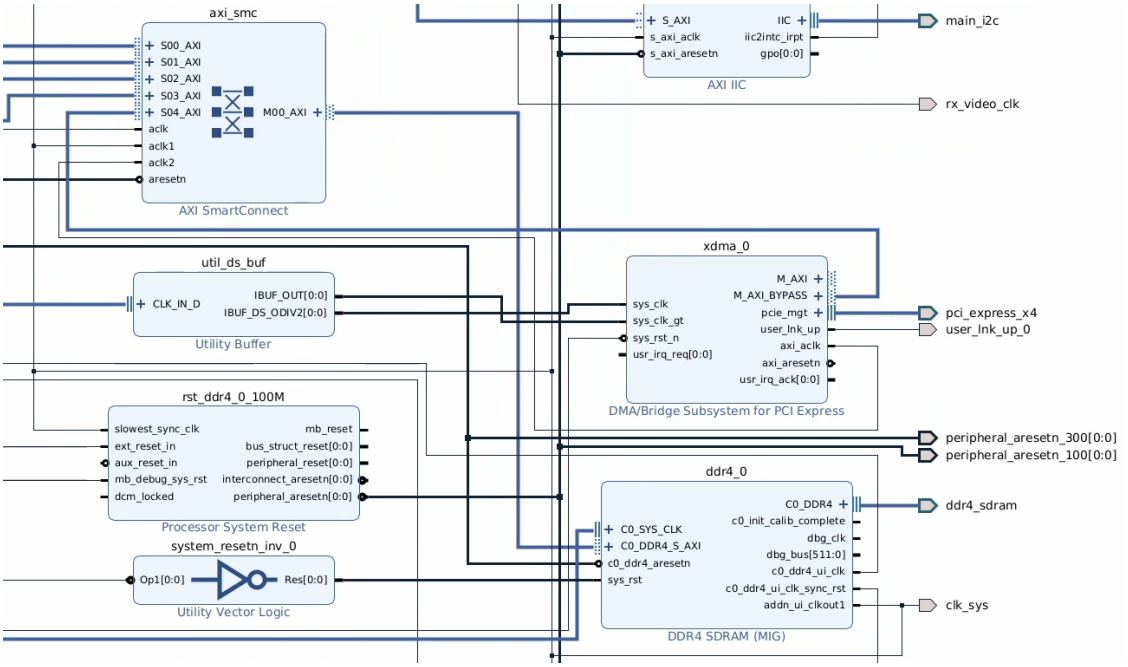


Figure 9: xdma for PCIe support

By double-clicking on the xdma IP, it is possible to customize the subsystem. For the reference design, the IP basic setup is DMA data mover, using X4 lane width, at maximum link speed of 5 GT/s. The selected DMA interface is AXI4 Memory Mapped. However, to access user memory, the AXI Memory Mapped CQ Bypass port is enabled in **PCIe: BARS** tab, providing high-bandwidth transfer. This setup is shown in figure 10.

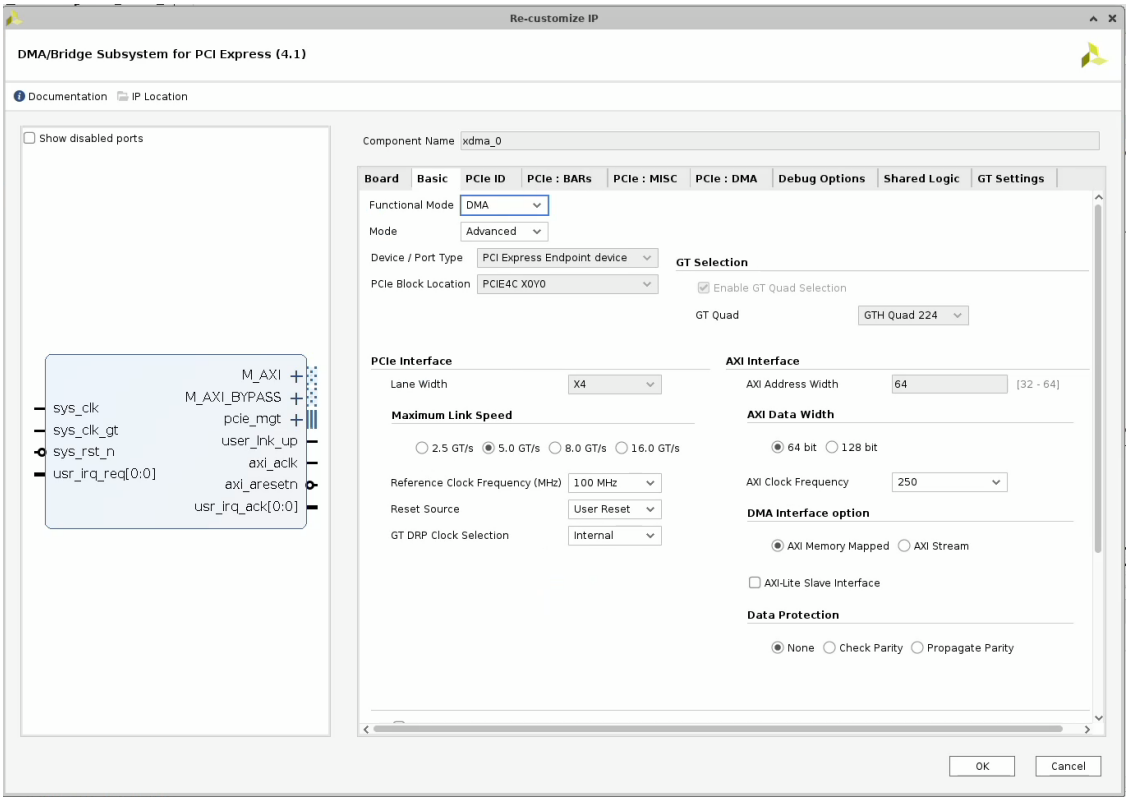


Figure 10: xdma IP - basic tab

Important Note on xdma IP setup

It is fundamental to set correctly the option "Disable GT Channel LOC Constraint" to **true** in GT settings tab, as shown in figure 11. Otherwise, the host PC may not be able to recognise the connected card correctly.

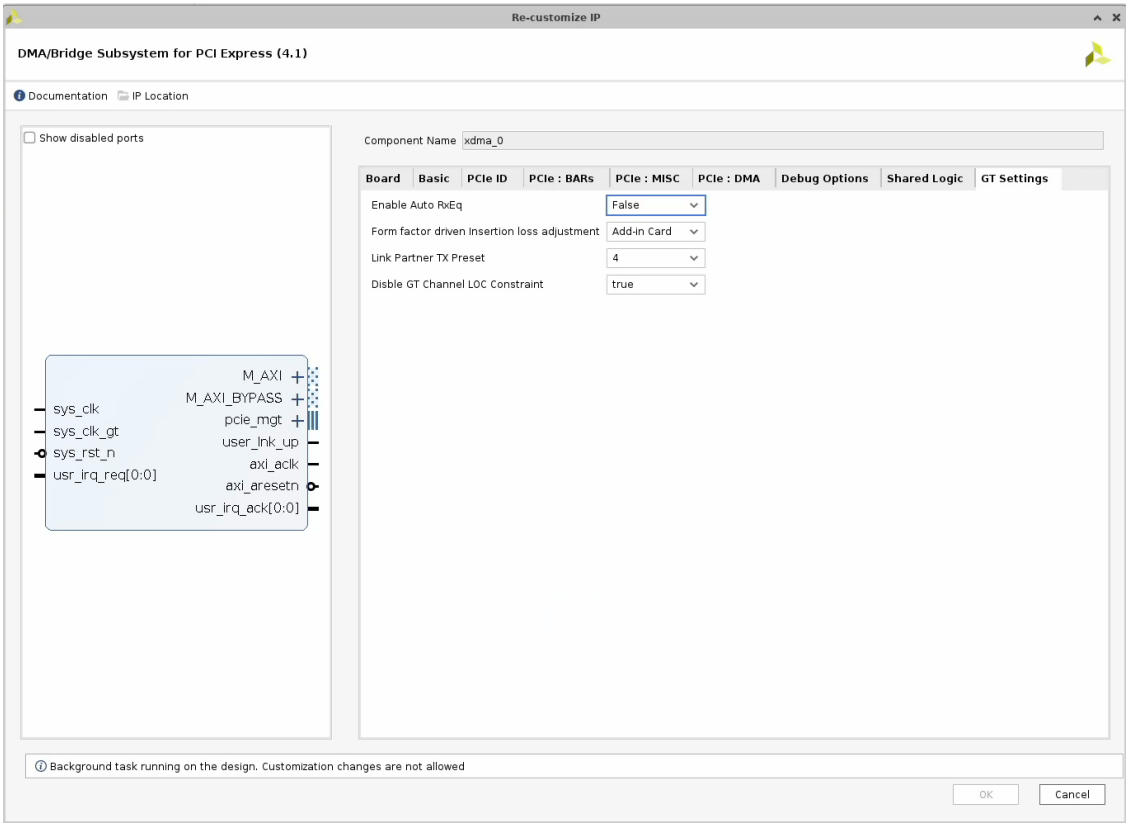


Figure 11: xdma IP - GT settings tab

Vivado project folder

The `Vivado_Project` folder contains two subfolders:

- **ELF**: contains the elf file included in the hardware platform creation, associated to the MicroBlaze configuration.
- **auboard_hdmi**: contains the source files required by Vivado to restore the project, together with the TCL script used to build the design.

In order to restore the Vivado project of the reference design, run the following commands:

```
cd Vivado\_Project/auboard
vivado -mode batch -source auboard\_hdmi.tcl
```

or alternatively, open Vivado GUI, and use tcl console to execute the TCL script:

```
source auboard\_hdmi.tcl
```

Vivado ELF File

For this Vivado project, the source code of the ELF file was taken from the article by **Marco Hoefle** and compiled using Vitis version 2023.1. The resulting ELF binary is embedded directly into the Vivado project. The source code can be retrieved from the original publication [4].

The FSBL (First Stage Boot Loader) is included directly in Vivado 2023.1 and associated as follows:

1. Open Vivado and navigate to:
   `Tools > Associate ELF Files...` (see Figure 12).

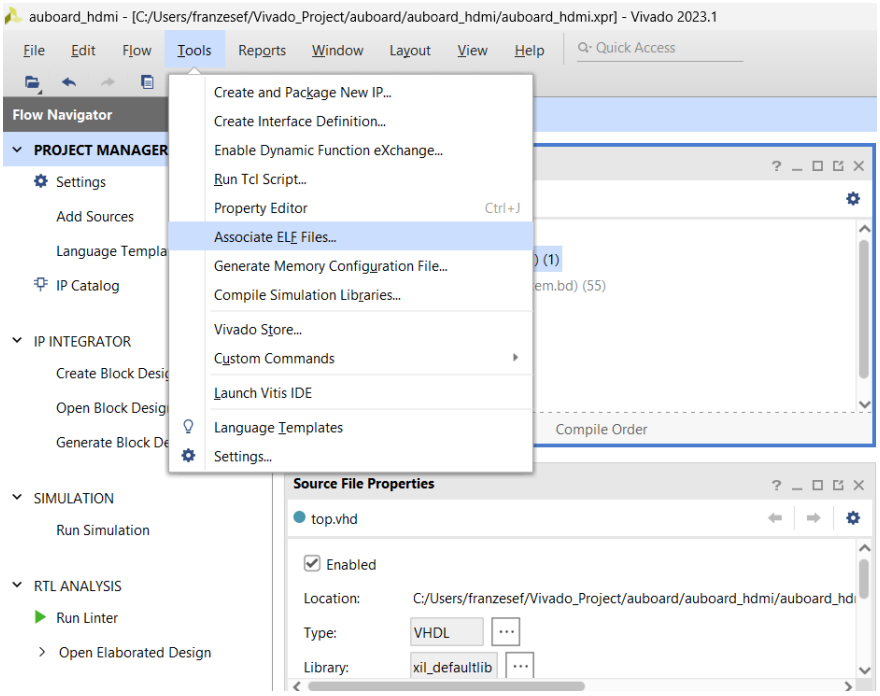


Figure 12: Vivado tool options

2. In the pop-up window, associate the compiled ELF file (e.g., `bobbyb.elf`) with the `microblaze_0` processor instance located under:
   `Design Sources > system > microblaze_0`.

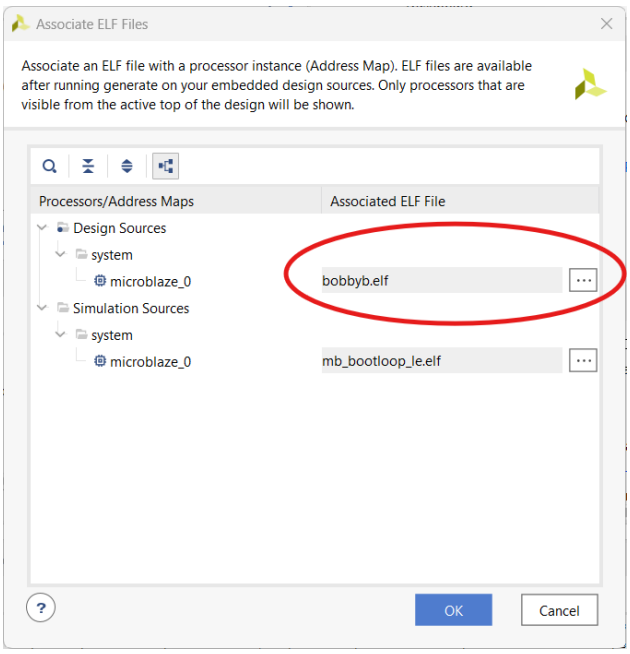


Figure 13: Associate ELF Files window

3. If the output products have not yet been generated for the block design, Vivado will prompt you to do so before completing the ELF association.
4. Once the project is compiled and the bitstream is generated, export the XSA file (including the bitstream) and place it in the following folder, using the correct filename (see Section 2.3.4 and 4.4.3):

```
/<poky-global-path>/mysources/meta-tria-technologies/meta-aag-auboard/recipes-bsp/hdf/files/
```

5. Finally, recompile the Yocto project. The generated output will include the updated bitstream with the embedded ELF.

4.9.1 Required Vivado Licenses

Using the HDMI IP subsystem requires an additional Xilinx license, which is free and can be downloaded from the AMD Product Licensing website. In the AMD Product Licensing page, search in the "Evaluation and No Charge cores catalog" the **LogiCORE HDMI Evaluation License**:

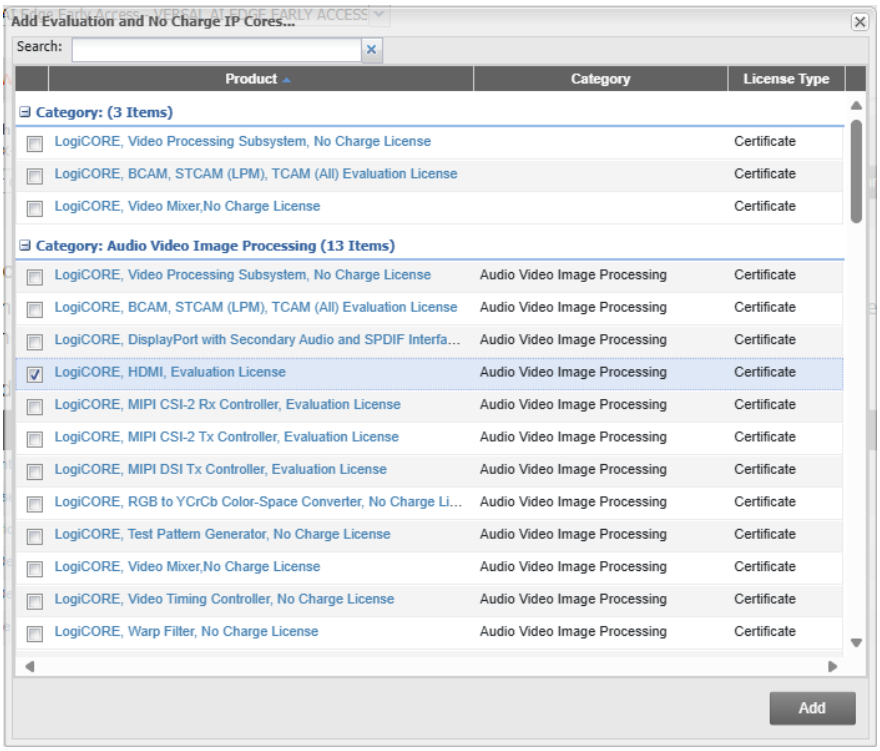


Figure 14: Vivado hdmi ip license

Then, use the Vivado License Manager (VLM) to install the license file, and ensure that it is correctly loaded before compiling the project.

5 CONCLUSION

This document has provided a complete walkthrough for configuring, compiling, and deploying a working Linux system on the AUBoard 15P Development Kit for microblaze architecture. By customizing recipes and layers, external kernel modules, and patching device drivers, you can successfully enable HDMI input and output.

Important Note on HDMI Resolutions

To ensure proper video capture and display, the HDMI-RX must be configured to use the same resolution as the HDMI-TX. Since the host PC sees the HDMI-RX as a standard monitor, make sure to manually set the PC's display resolution to match the expected format (for example, 1920x1080).

WARNING

For this demo the max supported resolution is 1920x1080.

Boot Duration

After powering on the AUBoard 15P, the initial boot process may take a few minutes to complete.

5.1 HDMI Screenshot Demo

After building the project and deploying the resulting binaries (`boot.bin`, `uImage`, `core-image-nano-microblaze-auboard-15p.cpio.tar.xz.u-boot`, and `fpga-hdf.bin`), the `hdmi-screenshot-demo-script` demonstrates the complete HDMI capture and display pipeline:

- Capture frames from the HDMI-RX port using `v4l2-ctl`.
- Display captured frames on the HDMI-TX output or framebuffer device.
- Convert raw frame data into a BMP image using the `raw2bmp` utility.
- Transfer the BMP image into `/dev/udmabuf0` for retrieval by the host PC over PCIe.

se viene usata un'alimentazione esterna si consiglia di accenderla prima del pc

Demo Instructions

1. Insert the AUBoard 15P into a PCIe slot on the host PC.
2. Connect the HDMI-TX port of the board to a monitor.
3. Connect the HDMI-RX port to a video source (for example, the HDMI output of a PC).
4. If using an external power supply, make sure to power on the board **before** the host PC.
5. Power up the PC. The Linux system will start, and the boot process may take several minutes to complete.
6. Once the system is ready, press the button labeled **PB6** on the AUBoard.
7. After a few seconds, the captured screenshot should appear on the monitor connected to the HDMI-TX port.

The script executed in the background performs the following actions: it sets the input resolution, waits for the button press, captures a single frame from HDMI-RX in raw data format, displays it on the framebuffer (HDMI-TX) pushing int in `/dev/fb0`, converts it to BMP format, and pushes it into `/dev/udmabuf0` for possible host-side retrieval.

5.2 Bonus Section: Dump Image from the PCIe

This section describes how to dump the contents of a PCIe device's BAR (Base Address Register) memory using the VFIO framework on a Linux host system. This method is compatible with Ubuntu 22.04 installation that has

IOMMU support enabled.

5.2.1 Identifying the PCIe Device

List all PCIe devices:

```
lspci -nn

# Example output:
02:00.0 RAM memory [10ee:9024]: Xilinx Corporation Device 9024
```

Record the PCIe address (`02:00.0`) and the Vendor:Device ID (`10ee:9024`).

5.2.2 Enable IOMMU Support (One-Time Setup)

Determine CPU Vendor

Run:

```
lscpu | grep "Vendor ID"
```

Then modify your GRUB config accordingly:

- Intel: `intel_iommu=on`
- AMD: `amd_iommu=on`

Update GRUB Configuration

Edit:

```
sudo nano /etc/default/grub
```

Update the line:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash intel_iommu=on"
```

Apply changes and reboot:

```
sudo update-grub
sudo reboot
```

5.2.3 Bind the PCIe Device to VFIO

1. Load the VFIO driver module:

   ```
   sudo modprobe vfio-pci

   ```

   On some Linux distributions (such as Ubuntu Server), this driver may already be built directly into the kernel. You can check with:

   ```
   grep VFIO_PCI /boot/config-$(uname -r)

   ```

   If the output is:

   ```
   CONFIG_VFIO=y
   CONFIG_VFIO_PCI=y

   ```

   then the driver is built-in, and you can safely skip the manual load step.

   If instead you see:

```
CONFIG_VFIO=m
CONFIG_VFIO_PCI=m

```

then the driver is compiled as a module and must be loaded manually using `modprobe`.

2. Unbind the device from its current driver (if applicable):

```
echo 0000:02:00.0 | sudo tee /sys/bus/pci/devices/0000:02:00.0/driver/unbind

```

3. Bind the device to `vfio-pci`:

```
echo 10ee 9024 | sudo tee /sys/bus/pci/drivers/vfio-pci/new_id

```

Verify:

```
lspci -nnk -d 10ee:9024
```

It should show: `Kernel driver in use: vfio-pci`

5.2.4 Determine IOMMU Group

Run:

```
readlink /sys/bus/pci/devices/0000:02:00.0/iommu_group
```

You will see a path like:

```
../../../../kernel/iommu_groups/12
```

This means your group number is 12, and the device will appear as `/dev/vfio/12`.

5.2.5 Enumerate BAR Regions

Use `lspci` to identify BARs:

```
sudo lspci -s 0000:02:00.0 -vv | grep "Region"
```

Example output:

```
Region 0: Memory at f7000000 (32-bit, non-prefetchable) [disabled] [size=64K]
Region 1: Memory at f6800000 (32-bit, non-prefetchable) [disabled] [size=8M]
```

In this example, region 1 is the target (8 MB).

5.2.6 Dump BAR1 Memory

Use the following C++ tool to dump the BAR contents:

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <linux/vfio.h>
#include <cerrno>
#include <cstring>

constexpr size_t MAX_SIZE   = 8 << 20;   // 8 MiB hard limit
constexpr size_t BLOCK_SIZE = 4096;      // alignment for O_DIRECT

struct Config {
```

```
    std::string device;              // e.g. 0000:02:00.0          (mandatory)
    std::string group;               // e.g. 12                    (mandatory)
    uint32_t    bar     = 0;         // BAR/region index           (mandatory)
    std::string output  = "pci_dump.bmp";
    size_t      size    = 0;         // 0 = dump full region
};

void show_help(const char* prog) {
    std::cout <<
        "Usage: " << prog << " -d DEV -g GRP -r IDX [ -o FILE ] [ -s SIZE ]\n\n"
        "  -d, --device   PCI device (e.g. 0000:02:00.0)            [mandatory]\n"
        "  -g, --group    VFIO group number (e.g. 12)               [mandatory]\n"
        "  -r, --region   BAR/region index (e.g. 1)                 [mandatory]\n"
        "  -o, --output   Output filename                           (default: pci_dump.bmp)\n"
        "  -s, --size     Size to dump (supports K/M suffix)        (default: full BAR)\n"
        "  -h, --help     Show this help and exit\n";
    std::exit(0);
}

size_t parse_size(const std::string& s) {
    char* end {};
    size_t v = std::strtoull(s.c_str(), &end, 0);
    if (*end == 'K' || *end == 'k') {
        v <<= 10;
    } else if (*end == 'M' || *end == 'm') {
        v <<= 20;
    }
    return v;
}

Config parse_args(int argc, char** argv) {
    Config cfg;
    bool have_dev  = false;
    bool have_grp  = false;
    bool have_bar  = false;

    auto next_arg = [&](int& i) -> std::string {
        ++i;
        if (i >= argc) {
            show_help(argv[0]);
        }
        return std::string(argv[i]);
    };

    for (int i = 1; i < argc; ++i) {
        std::string a = argv[i];

        if (a == "-h" || a == "--help") {
            show_help(argv[0]);
        } else if (a == "-d" || a.rfind("--device=", 0) == 0) {
            cfg.device = (a[1] == '-') ? a.substr(9) : next_arg(i);
            have_dev   = true;
        } else if (a == "-g" || a.rfind("--group=", 0) == 0) {
            cfg.group = (a[1] == '-') ? a.substr(8) : next_arg(i);
            have_grp  = true;
        } else if (a == "-r" || a.rfind("--region=", 0) == 0) {
            cfg.bar   = std::stoul((a[1] == '-') ? a.substr(9) : next_arg(i));
            have_bar  = true;
        } else if (a == "-o" || a.rfind("--output=", 0) == 0) {
            cfg.output = (a[1] == '-') ? a.substr(9) : next_arg(i);
        } else if (a == "-s" || a.rfind("--size=", 0) == 0) {
            cfg.size = parse_size((a[1] == '-') ? a.substr(7) : next_arg(i));
        } else {
            show_help(argv[0]);
        }
    }

    if (!have_dev || !have_grp || !have_bar) {
        std::cerr << "Error: --device, --group, and --region are mandatory.\n\n";
        show_help(argv[0]);
    }

    if (cfg.size > MAX_SIZE) {
        throw std::runtime_error("Size too large (>8 MiB)");
    }
    return cfg;
}

int main(int argc, char** argv) {
    try {
        Config cfg = parse_args(argc, argv);

        /* ---------------- VFIO setup ---------------- */
```

```
        int container = open("/dev/vfio/vfio", O_RDWR);
        int group_fd  = open(("/dev/vfio/" + cfg.group).c_str(), O_RDWR);

        if (container < 0 || group_fd < 0) {
            throw std::runtime_error("VFIO open failed");
        }

        if (ioctl(group_fd, VFIO_GROUP_SET_CONTAINER, &container) < 0 ||
                ioctl(container, VFIO_SET_IOMMU, VFIO_TYPE1_IOMMU)      < 0) {
            throw std::runtime_error("VFIO container/IOMMU setup failed");
        }

        int device_fd = ioctl(group_fd, VFIO_GROUP_GET_DEVICE_FD, cfg.device.c_str());
        if (device_fd < 0) {
            throw std::runtime_error("VFIO device FD failed");
        }

        vfio_region_info reg{ .argsz = sizeof(reg), .index = cfg.bar };
        if (ioctl(device_fd, VFIO_DEVICE_GET_REGION_INFO, &reg) < 0 || reg.size == 0) {
            throw std::runtime_error("Invalid BAR region");
        }

        size_t dump_size = cfg.size ? cfg.size : reg.size;
        if (dump_size > reg.size) {
            dump_size = reg.size;
        }
        if (dump_size > MAX_SIZE) {
            throw std::runtime_error("Requested size exceeds 8 MiB");
        }

        /* ---------------- mmap BAR ---------------- */
        void* bar = mmap(nullptr, dump_size, PROT_READ, MAP_SHARED, device_fd, reg.offset);
        if (bar == MAP_FAILED) {
            throw std::runtime_error("mmap BAR failed");
        }

        /* ---------------- allocate aligned buffer ---------------- */
        void* aligned = nullptr;
        if (posix_memalign(&aligned, BLOCK_SIZE, dump_size) != 0) {
            throw std::runtime_error("posix_memalign failed");
        }
        std::memcpy(aligned, bar, dump_size);

        /* ---------------- direct write ---------------- */
        int out_fd = open(cfg.output.c_str(),
                O_CREAT | O_TRUNC | O_WRONLY | O_DIRECT, 0644); if (out_fd < 0)
        {
            throw std::runtime_error("open output (O_DIRECT) failed: " + std::string(std::strerror(errno)));
        }

        ssize_t written = write(out_fd, aligned, dump_size);
        if (written != static_cast<ssize_t>(dump_size)) {
            throw std::runtime_error("Incomplete write: " + std::string(std::strerror(errno)));
        }

        close(out_fd);
        free(aligned);
        munmap(bar, dump_size);

        std::cout << "Dumped " << dump_size << " bytes from BAR "
            << cfg.bar << " (" << cfg.device << ") to "
            << cfg.output << "\n";

        close(device_fd);
        close(group_fd);
        close(container);
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
        return 1;
    }
    return 0;
}
```

Install dependencies (Ubuntu)

Before compiling, make sure the required build tools are installed:

```
sudo apt update
sudo apt install build-essential linux-headers-$(uname -r)
```

Compile and run:

```
g++ -o pci_dump pci_dump.cpp
sudo ./pci_dump --device=0000:02:00.0 --group=12 --region=1
```

You can optionally specify size (max 8 MB) and output filename:

```
sudo ./pci_dump --device=0000:02:00.0 --group=12 --region=1 --size=8M --output=pci_dump.bmp
```

5.2.7 Verify Output

Dumped memory is saved as `pci_dump.bmp`. View the contents:

```
hexdump -C pci_dump.bmp | less
```

The PCIe memory region is 8 MB in size. From the board, a framebuffer dump is captured, converted to a BMP image, and stored in this memory. When the host reads it back, the full 8 MB is transferred, even though only part of the region contains the actual BMP data. Some image viewers, such as `gwenview`, can still render the image correctly despite trailing garbage data.

If needed, install gwenview:

```
sudo apt install gwenview
```

5.2.8 Brief Recap

- Enable IOMMU via GRUB (one-time setup).
- Bind the PCIe device to the `vfio-pci` driver (after every boot).
- Identify the BAR region to access using `lspci`.
- Use VFIO to safely mmap and dump the PCIe BAR region to a file.

This procedure enables controlled, userspace-level access to PCIe BAR memory using the generic VFIO framework.

5.2.9 Doc source code

The LaTeX sources for this document can be found in the `mysource/meta-tria-technologies/docs` directory.

REFERENCES

[1] gerrit. *git-repo*. URL: https://gerrit.googlesource.com/git-repo.

[2] Glitchub. *arith64: Software 64-bit arithmetic for 32-bit systems*. https://github.com/glitchub/arith64. Accessed: 5 March 2025. 2023.

[3] ikwzm. *udmabuf*. u-dma-buf is a Linux kernel driver that allocates contiguous memory blocks in kernel space as DMA buffers and makes them available from user space. It is expected that these memory blocks will be used as DMA buffers when a user application implements the driver in user space using UIO (User space I/O). URL: https://github.com/ikwzm/udmabuf. tag: v4.5.3.

[4] Marco Hoefle. *One of the coolest FPGA Technologies: AMDs DFX*. URL: https://www.hackster.io/marco13/one-of-the-coolest-fpga-technologies-amds-dfx-654e04#toc-building-the-linux-system-using-amd-s-petalinux-5.

[5] The Yocto Project. *Yocto Project Documentation*. URL: https://docs.yoctoproject.org/langdale/index.html. Version: 4.1.999 (Langdale).

[6] Xilinx. *yocto-manifests*. Version rel-v2023.1. URL: https://github.com/Xilinx/yocto-manifests.