Application Note

AFE79xx Linux Driver

Akshay Kumar, Navin Maheshwari, Mustafa Chikhliwala, Phanindra

ABSTRACT

The AFE79xx high-performance analog front-end serves critical functions in wireless communications infrastructure, including 5G base stations, massive MIMO systems, and beamforming applications. This application note solves the complex integration challenge of deploying AFE79xx devices with Linux platforms. Communications equipment manufacturers, test and measurement companies, and defense contractors developing advanced RF systems will benefit from proven integration procedures covering both kernel driver and user-space SDK approaches. The kernel integration method provides production-ready systems with optimal performance for deployed infrastructure, while the user-space approach accelerates development cycles for prototype and testing environments. Complete step-by-step procedures include FPGA hardware design in Vivado, Linux kernel configuration with PetaLinux, Yocto, SPI interface setup, and bootable system deployment. Both methodologies are hardware-validated on ZCU102 platforms, enabling engineering teams to implement AFE79xx support efficiently while maintaining flexibility for future updates and customization.

Table of Contents

Trademarks

1 Introduction

AFE79xx is a family of high-performance, wide-bandwidth multi-channel transceivers, integrating four RF sampling transmitter chains, four RF sampling receiver chains, and up to two RF sampling digitizing auxiliary chains (feedback paths).

This application note provides instructions for integrating the AFE79xx SDK into Linux-based systems. Three integration methods are covered:

1. **Linux kernel driver using PetaLinux Tools** - Compiles directly with the Linux kernel source
2. **Linux kernel driver using Yocto Project** - Compiles directly with the Linux kernel source
3. **User-space application layer** - Operates without kernel modifications

The kernel approaches (methods 1 and 2) provide optimal performance and tight system integration, differing primarily in build system and workflow. The user-space approach offers development flexibility and simplified maintenance without kernel dependencies. Method selection depends on system architecture constraints, performance requirements, build infrastructure, and deployment scenarios. This document assumes familiarity with Linux kernel development, device driver concepts, and embedded system design fundamentals.

2 Linux Kernel Driver

The AFE79xx kernel driver helps isolate two components of device SDK, high level API functions to be used in user-space applications and device specific processing in kernel space. This simplifies SDK integration into user applications.

- All **SPI communications** are incorporated within the kernel.
- Integrated with kernel as a **SPI slave device** using standard Linux SPI subsystem, this makes driver the sole owner of slave device, ensuring atomic operations and preventing conflicts with other SPI accesses.
- User access is provided as a **character driver** for easier integration with user applications, which works as standard POSIX file operations (open, release, read, write, and ioctl)
- It shows connected devices under /dev as **/dev/device_nameX.Y** format, where X represents SPI master port and Y as a slave id, for easier detection and identification. Example /dev/afe79xx1.0
- Supports **auto invocation and slave assignment through device tree** using compatible string with format ***ti,device_name***. Example "ti,afe79xx". This approach enables dynamic device configuration without requiring driver recompilation.
- **Multiple devices** are presented as independent devices to the user with unique ids under /dev. Individual device contexts are managed entirely in kernel code using various standard Linux methods to make customer application simpler.
- Verifies **thread-safe operation** using various Linux provisions

Below are some notable features of the driver implementation

The following is the typical directory structure for device SDK supplied for device integration using linux kernel

Device_SDK/

├─ kernel-driver

└──── device_name/*

└─ userspace

├────device_name/*

├──── example/*

└──── Makefile

Here, the device folder under kernel-driver is used to build the kernel module whereas device folder under userspace contains high-level API definitions and is used as part of user application, while the example folder has user application examples and the Makefile shows the typical compilation method for integrating APIs with user applications.

2.1 Linux Kernel Driver using PetaLinux Tools

This section describes the process of building a Linux kernel with integrated AFE79xx SDK support using PetaLinux tools. This approach uses an in-tree build methodology, where the AFE79xx driver is compiled directly into the kernel source tree during the kernel build process.

This procedure assumes PetaLinux Tools 2023.2 (or later) are already installed and sourced in your environment.

Before beginning the PetaLinux build process, make sure the user has:

1. **Board Support Package (BSP)**
   - Download from AMD/Xilinx website or use the version included with PetaLinux installation
2. **Hardware Description File** (**.xsa**)
   - Generated from the Vivado hardware design
   - Must include the FPGA bitstream

See **Section 4** for instructions on creating the reference hardware design and generating the XSA file for the ZCU102 board.

Use the following steps:

Step 1: On the Linux terminal , create a petaLinux project by running command

```
petalinux-create -t project -n <Project_Name> -s <path to BSP>
```

Step 2: Change directory into the project folder <**Project_Name**> and run command

```
petalinux-config –get-hw-description <path to XSA>
```

Step 3: Configure kernel using command

```
petalinux-config -c kernel
```

The configuration menu opens. Navigate to *Device drivers SPI support* and enable options Cadence SPI controller, Xilinx SPI controller common module, Xilinx Zynq GQSPI controller, AMD SPI controller and User mode SPI device driver support.

Figure 2-1. Kernel Config Menu

Step 4: configure Root file system using command

```
petalinux-config -c rootfs
```

This will open the RootFs configuration menu. To enable GCC compiler navigate to *Filesystem packages-> misc > packagegroup-core-buildessential* under this select *packagegroup-core-buidessential* and *packagegroup-core-buidessential-dev*

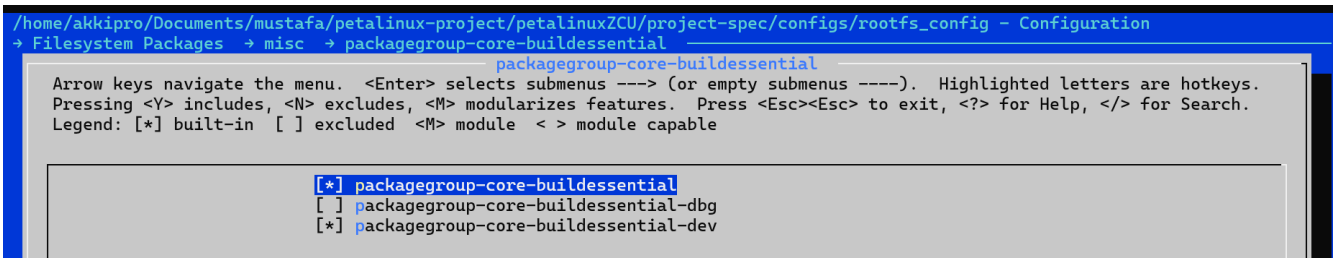

Figure 2-2. RootFs Config Menu

Step 5: Make the kernel source code modifiable by running:

```
petalinux-devtool modify linux-xlnx
```

This command extracts the kernel source to a working directory. Note the output path displayed by the command, generally the path is <Project_Name>/components/yocto/workspace/sources/linux-xlnx/. Navigate to that directory:

```
cd <Path_to_kernel_Src>
```

Step 6: Create a folder in <Path_to_kernel_Src>/drivers/char/ called afe79 and paste the afe79 folder from kernel-driver from SDK in this folder

Step 7: In the Makefile in <Path_to_kernel_Src>/drivers/char add this line at the end

```
obj-$(CONFIG_AFE79XX) += afe79/
```

This tells the kernel build system to compile the AFE79xx driver when the driver enabled in the configuration.

Step 8: In Kconfig file in <Path_to_kernel_Src>/drivers/char. Locate the *endmenu* line at the end of the file. Add this line before endmenu.

```
source "drivers/char/afe79/Kconfig"
```

System reads the AFE79xx Kconfig file. The driver appears in the menuconfig interface. Enable and disable the driver through the menu

Step 9: Configure the kernel to enable the AFE79xx driver. Run command

```
petalinux-config -c kernel
```

This will open the kernel configuration menu. Navigate device drivers -> character devices and enable AFE79xx by pressing *Y.*

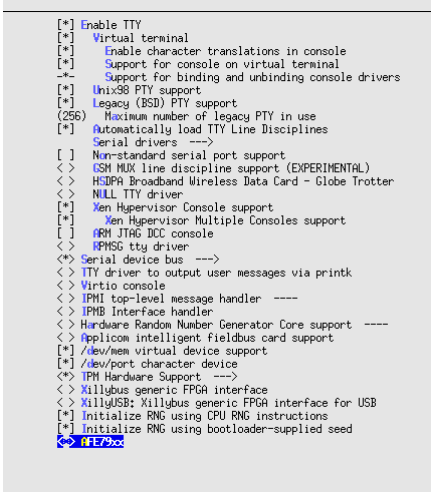

Figure 2-3. Kernel Config Menu for Char Drivers

Step 10: The device tree must be updated to declare the AFE79xx device on the SPI bus, enabling the kernel to recognize and initialize the hardware. The *system-user.dtsi* file is used for custom device tree modifications in PetaLinux projects.

Go to device tree, add SPI to device tree by editing the *system-user.dtsi* file in *path*

```
< Project_Name>/project-spec/meta-user/recipes-bsp/device-tree/files/
```

In the device tree, add entries for SPI slaves as shown in Figure 2-4

```
/include/ "system-conf.dtsi"
/ {
};

&spi0 {
    is-decoded-cs= <0>;
    status = "okay";
    num-cs = <1>;
    afe79xx@0 {
        compatible = "ti,afe79xx";
        reg = <0>;
        spi-max-frequency = <10000000>;
    };
};
```

Figure 2-4. Example Device Tree

This creates a device in /dev by the name */dev/afe79xx1.0* after bootup.

The node must specify the compatible string *ti,afe79xx,* for the device driver to bind.

Step 11: Build the project using command:

```
petalinux-build
```

This process can take 30-60 minutes depending on the host system performance.

Step 12: Package the file using

```
petalinux-package --boot --fsbl < Project_Name >/images/linux/zynqmp_fsbl.elf --fpga <path to bit file> --u-boot -force
```

After successfully built, the user has all required files for Preparing Linux Boot Images.

2.2 Linux Kernel Driver Using Yocto Project

This section describes how to integrate the AFE79xx SPI kernel driver into a Linux image built using the Yocto Project. This method is not tied to any specific board or vendor toolchain. It is applicable to any hardware platform for which a Yocto-compatible Board Support Package (BSP) exists.

The driver is delivered as a pre-built custom Yocto layer named **meta-afe79**, which can be added to any existing Yocto project. Users who already have a working Yocto build environment may proceed directly to Section 2.2.4. Users who are setting up Yocto for the first time should follow all steps in sequence.

Note

All configuration syntax in this section follows **Yocto Langdale (4.1.x)** conventions. Yocto has undergone significant syntax changes across releases. Variable assignment operators changed from underscore notation (for example, IMAGE_INSTALL_append) used in Dunfell and earlier releases, to colon notation (IMAGE_INSTALL:append) introduced in Honister and used in all later releases including Langdale. Users working with a different Yocto version must verify and update syntax accordingly. See the official Yocto Project migration guides at https://docs.yoctoproject.org for version-specific changes.

2.2.1 Overview of the Yocto Project

The Yocto Project is an open-source build framework that allows developers to create custom Linux distributions for embedded systems. It is not a Linux distribution itself, but a set of tools and methodologies for building one. At its core, Yocto uses a build engine called **BitBake**, which processes metadata files called **recipes** to fetch, compile, and package software components.

The fundamental organizational unit in Yocto is the **layer**. A layer is a collection of related recipes, configuration files, and patches that add specific functionality to a build. Official layers exist for most popular hardware platforms and are typically provided by the board or SoC vendor as a Board Support Package. Custom layers allow developers to add proprietary or project-specific software, such as the AFE79xx driver, without modifying upstream layers.

The AFE79xx driver is packaged as a Linux kernel driver and delivered as part of a custom Yocto layer. The driver source code is compiled as a loadable kernel module (.ko file), which is built as part of the Yocto image build process and installed into the root filesystem. Once the target system boots into Linux, the driver is automatically loaded into the running kernel, enabling communication with the AFE79xx device over SPI. Yocto provides built-in infrastructure for building and packaging kernel drivers of this type, and the provided meta-afe79 layer takes full advantage of this capability.

2.2.2 Host System Requirements

The following host system configuration is required to build a Yocto image.

Operating System

Ubuntu 18.04 LTS or later is recommended. Other Linux distributions supported by Yocto may also be used; refer to the official Yocto documentation for a full list.

Table 2-1. System Resources

Resource	Minimum	Recommended
RAM	8 GB	16 GB
Free Disk Space	100 GB	200 GB
CPU Cores	4	8 or more

Required Packages

Install the following packages on the host system before proceeding:

```
sudo apt-get update
```

```
sudo apt-get install -y gawk wget git diffstat unzip texinfo gcc \
build-essential chrpath socat cpio python3 python3-pip python3-pexpect \
xz-utils debianutils iputils-ping python3-git python3-jinja2 \
libegl1-mesa libsdl1.2-dev pylint3 xterm python3-subunit \ mesa-common-dev zstd liblz4-tool device-tree-compiler
```

2.2.3 Setting Up a Yocto Build Environment

This section provides instructions for initializing a Yocto build environment. Users who already have a working Yocto setup configured for their target board may skip to **Section 2.2.4.**

Step 1: Downloading Poky

Poky is the Yocto Project reference distribution and contains BitBake along with the core metadata layers. Clone the version of Poky that corresponds to your desired Yocto release. We will refer the poky root directory as **project_directory** moving forward. The following example uses the Langdale branch:

```
git clone -b langdale https://git.yoctoproject.org/poky.git
```

Step 2: Initializing the Build Environment

From the folder project_directory, source the Yocto environment setup script to create the build directory and configure the shell environment:

```
source poky/oe-init-build-env <build_directory>
```

This command creates a build directory, generates default configuration files, and adds BitBake to the shell path. It must be run every time a new terminal session is started.

IMPORTANT: Always run this command from the project root directory. Running it from inside the build directory will create a nested build folder and reset all configurations.

2.2.4 Configuring Board Specific Layers

Step 1: Download Board-Specific Layers

Every hardware platform requires a corresponding BSP layer that provides the machine configuration, kernel, and bootloader recipes for that board. BSP layers are typically provided by the board or SoC vendor and are cloned separately. Consult your board vendor's documentation to identify the correct BSP layer and branch compatible with your chosen Yocto release.

For Xilinx-based platforms, the relevant layers are meta-xilinx, meta-xilinx-bsp, meta-xilinx-standalone, and meta-xilinx-tools, all available from the Xilinx GitHub repository. For other platforms such as Raspberry Pi, NXP, or Intel, the equivalent vendor-provided BSP layers should be used. We can use git clone for the layers as shown in example below:

Xilinx

```
git clone -b langdale https://github.com/Xilinx/meta-xilinx.git
```

```
git clone -b langdale https://github.com/Xilinx/meta-xilinx-tools.git
```

Intel - **Altera**

```
git clone https://git.yoctoproject.org/meta-intel-fpga
```

Additionally, many BSP layers depend on packages from the OpenEmbedded layer collection (meta-openembedded). Clone the appropriate branch of this collection as well:

```
git clone -b langdale https://github.com/openembedded/meta-openembedded.git
```

Step 2 : Adding Layers

Add all required layers to the build using the bitbake-layers command. The exact set of layers depends on the target board. At minimum, the core Poky layers, the board BSP layers, and the AFE79 layer must be present. Only Layer cloned in previous step can be added using the command below:

```
bitbake-layers add-layer <Path_to_layer>
```

For example, on ZCU102 following layers are needed:

```
bitbake-layers add-layer ~/yocto-zcu102/meta-xilinx/meta-xilinx-core
```

```
bitbake-layers add-layer ~/yocto-zcu102/meta-xilinx/meta-xilinx-bsp
```

```
bitbake-layers add-layer ~/yocto-zcu102/meta-xilinx/meta-xilinx-standalone
```

```
bitbake-layers add-layer ~/yocto-zcu102/meta-xilinx-tools
```

```
bitbake-layers add-layer ~/yocto-zcu102/meta-openembedded/meta-oe
```

```
bitbake-layers add-layer ~/yocto-zcu102/meta-openembedded/meta-python
```

```
bitbake-layers add-layer ~/yocto-zcu102/meta-openembedded/meta-networking
```

```
bitbake-layers add-layer ~/yocto-zcu102/meta-openembedded/meta-multimedia
```

Verify that all required layers are active using the command.

```
bitbake-layers show-layers
```

Step 3: Adding hardware Description File and Target Machine

Many Yocto BSP workflows require a hardware description files that captures the complete hardware design of the target platform. Yocto uses hardware description file via **local.conf** format to automatically generate the correct FSBL, PMU firmware, and device tree for the specific hardware design.

conf/local.conf takes the hardware description file path and machine variables to correctly provide configs for build.

Open the file <project_directory/build_directory>/conf/local.conf in the build directory. and set the MACHINE variable to the identifier corresponding to your target board. This identifier is defined in the board's BSP layer. For example, the ZCU102 board uses the identifier zcu102-zynqmp.

.

Refer to your board's BSP documentation for the correct machine name.

Xilinx uses .XSA file and intel uses SOPC Information File.

For Xilinx:

Place the XSA file in a known location on the host system. The recommended location is within the custom layer:

meta-afe79/recipes-bsp/hdf/files/

Then add the following lines to the <project_directory/build_directory>/conf/local.conf file in the build directory, replacing the path and filename with those of your XSA file and the machine cariable:

```
MACHINE ??= "zcu102-zynqmp"
HDF_BASE = "file://"
FILESEXTRAPATHS:prepend:pn-external-hdf := "/absolute/path/to/meta-afe79/recipes-bsp/hdf/files:"
XILINX_RELEASE_VERSION = "v2023.2"
PREFERRED_VERSION_external-hdf = "2023.2"
LICENSE_FLAGS_ACCEPTED += "xilinx"
```

Note

The XILINX_RELEASE_VERSION must match the Vivado version that generated the XSA file. An XSA generated by Vivado 2023.2 requires XILINX_RELEASE_VERSION = "v2023.2". Using a mismatched version causes the XSCT tool to fail when processing the hardware description.

Configure the hardware handoff path according to the requirements of your specific BSP layer. Refer to the BSP layer README for the correct variable name and path format, as this varies between vendor platform generations.

2.2.5 Integrating the AFE79xx Yocto Layer

The AFE79xx driver is provided as a ready-to-use Yocto layer named **meta-afe79**. This layer contains all necessary recipes, the driver source code, and a reference device tree overlay. The integration procedure is identical regardless of the target board or Yocto version, except for the device tree configuration.

Step 1: Add the Device Layer to the Project

Copy the meta-afe79 layer directory from the TI provided SDK into your Yocto **project_directory**. Then, from the build directory with the Yocto environment active, register the layer with BitBake:

```
bitbake-layers add-layer <path_to_meta-afe79>
```

Confirm the layer appears in the active layer list:

```
bitbake-layers show-layers
```

Step 2: Include the Device Driver Package in the Final Linux Image

Add the following line to the <project_directory/build_directory>/conf/local.conf file in the build directory. This instructs Yocto to compile the AFE79xx driver module and include it in the root filesystem of the output image:

```
IMAGE_INSTALL:append = " afe79"
```

Note

For Yocto versions older than Honister, replace the colon notation with underscore notation:

```
IMAGE_INSTALL_append = " afe79"
```

Once included in the image, the driver module is configured to load automatically at system startup through a module autoload configuration file placed at /etc/modules-load.d/afe79.conf on the target filesystem.

To add the GCC compiler and development tools to the image add the following lines. This will help in compilation in userspace.

```
IMAGE_FEATURES += "tools-sdk"
```

Step 3: Configure Device Tree Overlay File

The Linux driver uses the device tree to discover the hardware it manages. For the device driver to bind , the device tree must contain a node with the compatible string **ti,afe79xx** on the appropriate SPI bus. The method for adding this node depends on the Yocto workflow used by your board's BSP.

For Xilinx-based boards, the meta-afe79 layer includes a ready-to-use overlay file at recipes-bsp/device-tree/files/system-user.dtsi and a corresponding bbappend file that automatically includes this overlay in the build. Before building, open the system-user.dtsi file and verify that the SPI node labels and properties match your board's hardware design. The file as provided references spi0, which correspond to the SPI controllers used on the ZCU102 evaluation board. If your board uses different SPI controllers or node labels, update the file accordingly.

As an example for Xilinx ZCU102 board, add entries for SPI slaves as shown below for SPI0:

Go to device tree, add SPI to device tree by editing the **system-user.dtsi** file in **path<path_to_meta-afe79>meta-afe79\recipes-bsp\device-tree\files**

As an example for Xilinx ZCU board, In device tree add entries for SPI slaves as shown below for SPI0:

```
/include/ "system-conf.dtsi"
/ {
};

&spi0 {
    is-decoded-cs= <0>;
    status = "okay";
    num-cs = <1>;
    afe79xx@0 {
        compatible = "ti,afe79xx";
        reg = <0>;
        spi-max-frequency = <10000000>;
    };
};
```

Figure 2-5. Example Device Tree

This will create a device in /dev by the name "/dev/afe79xx1.0" after bootup. verify that the SPI node labels and properties match your board's hardware design.

Step 4: Include our custom device tree file to the base device tree file

Xilinx uses device-tree.bbappend file and intel uses linux-socfpga.bbappend file to include custom dtsi file to base device tree without changing the original recipe.

The mata-afe79/recipes-bsp/device-tree/device-tree.bbappend file included in the meta-afe79 layer is specific to the Xilinx DTG workflow. Users on other platforms should either remove this file from the layer or replace it with the appropriate mechanism for their BSP.

Step 5: Build the Image

Run the BitBake build command for your target image recipe:

```
bitbake <image-name>
```

Replace <image-name> with the image recipe appropriate for your project. Common options include:

- **core-image-minimal** – A minimal bootable image with only essential packages. Suitable for production deployments where a small footprint is required.
- **core-image-minimal-dev** – Extends **core-image-minimal** with development tools and headers. Useful when compiling or debugging applications directly on the target board.

- **core-image-full-cmdline** – A console image with a fuller set of standard Linux command-line utilities. Suitable for development and testing environments.
- **petalinux-image-minimal** – Xilinx-specific minimal image available when using meta-xilinx BSP layers.

The first build will take between one and three hours depending on host system performance and network speed, as BitBake downloads and compiles all required sources. Subsequent builds are significantly faster due to shared state caching.

During the build, warning messages about failed primary download URLs are normal. BitBake automatically retries using mirror servers and these warnings do not indicate build failures.

2.2.6 Verifying the Integration

After flashing the built image to the target board and booting into Linux, run the following commands to confirm that the AFE79xx driver has loaded successfully.

Confirm the module is present in the loaded module list:

```
lsmod | grep afe79
```

Review the kernel log for driver initialization messages:

```
dmesg | grep -i "afe79"
```

A successful initialization produces output similar to the following:

```
afe79: loading out-of-tree module taints kernel.
```

AFE79xx SPI driver initialization

```
afe79xx spi1.0: AFE79xx device initialized successfully
```

```
afe79xx spi2.0: AFE79xx device initialized successfully
```

Confirm that the character device files have been created:

```
ls -la /dev/afe79*
```

The driver creates one device file per AFE79xx instance detected on the SPI bus, named according to the bus number and chip select index, for example /dev/afe79xx1.0 and /dev/afe79xx2.0.

2.2.7 Troubleshooting

Table 2-2. Troubleshooting

Issue	Symptom	Resolution
Module not loading at boot	lsmod shows no afe79 entry	Verify IMAGE_INSTALL:append = " afe79" is present in local.conf. Check that /etc/modules-load.d/afe79.conf exists on the target.
Build syntax errors	BitBake parse errors related to IMAGE_INSTALL or KERNEL_MODULE_AUTOLOAD	Verify Yocto version. Use underscore syntax for versions older than Honister.
Driver probe fails	No initialization messages in dmesg	Verify the device tree contains a node with compatible = "ti,afe79xx" on the correct SPI bus with status = "okay".
Device files not created	/dev/afe79* not present	Check dmesg for driver binding errors. Verify the SPI controller is enabled in the device tree.
SPI communication failure	Application returns incorrect read values or errors	Verify the correct device file is used. Confirm the SPI frequency and mode match the hardware. For Xilinx platforms, verify that the FPGA bitstream correctly routes SPI signals through the FMC connector to the AFE79xx device.
Nested build directory	Configurations reset unexpectedly	Always source the environment script from the project root directory, never from inside the build directory. Delete any nested build subdirectory if created accidentally.

2.3 Preparing Linux Boot Images

This section describes how to partition the SD card, copy the boot images and root filesystem to the appropriate partitions, and prepare the bootable media.

Follow the steps:

Step 1: Format the SD card and create 2 partitions using command

```
sudo gparted
```

First one will be fat32 and named BOOT (allocate 1GB for this) and second will be ext4 and named rootfs (allocate remaining space to this).

Step 2: Copy the BOOT.bin, image.ub and boot.scr from <project_name>/images/linux/ folder to BOOT partition

```
sudo cp <FileName> /media/<username>/ BOOT/
```

Step 3: Extract the file system (rootfs.tar.gz from <project_name>/images/linux/) to rootfs parition using

```
sudo tar -xf rootfs.tar.gz -C /media/<username>/rootfs/
```

Step 4: Copy the rootfs.cpio (from <project_name>/images/linux/) file using

```
sudo cp rootfs.cpio /media/<username>/rootfs/
```

Step 5: Eject the SD card (Eject to avoid corrupted rootfs).

Step 6: Now plug in the SD card to the ZCU board and connect UART cable to PC. Power up the board (set SW6 to SD card boot mode) and Linux will boot up and the terminal can be accessed via software like Putty.

With the completion of this section, the Linux kernel incorporating the AFE79xx driver has been successfully built using PetaLinux tools with in-tree build approach. The AFE SPI device is correctly registered in the device tree and exposed as a character device, making the system fully ready for AFE driver operation and application development.

2.4 Operating the AFE79xx Device from User Space

The CAFE package provides a high-level API for communicating with the AFE79xx kernel driver from user-space applications. Compile this along with your software stack to configure and control AFE79xx device.

The following example shows the representation of few high-level API calls with device instance.

```
#include <fcntl.h>
#include "tiAfe79xx.h"
int main(void) {
    const char *device = "/dev/afe79xx1.0"; // Assuming /dev/afe79xx1.0 is available
    int tiAfe79dev = open(device, O_RDWR); // Get the device instance using ioctl
    if (tiAfe79dev < 0){return 1;} // Sanity check if device instance is generated successfully
    Afe79_setAfeLogLvl(tiAfe79dev, 2); // Model API call with single argument
    Afe79_overrideTdd(tiAfe79dev, 0xF, 0x3, 0xF, 1); // Model API call with multiple arguments
    close(tiAfe79dev); // Release device instance to make it available to other applications
    return 0;
 }
```

Note

Please refer to device API documentation for exact API sequence to be followed and detailed information regarding APIs.

Installing user space library

Go to userspace folder in SDK and run:

1. make
2. sudo make install

Compiling user application with user space library:

Since the library and the headers are already installed in linux, users are not required to specify any path during compilation. Include single header file tiDeviceName.h(tiAfe79xx.h) wherever calling any device API and provide -ldevicename(-lafe79xx) during linking process.

3 Building the AFE79xx SDK in Application Layer

This section describes building the AFE79xx SDK as a user-space application, providing an alternative integration method to the in-tree kernel driver approach covered in Building the AFE79xx SDK in Application Layer. In this approach, the complete AFE79xx driver functionality is implemented in user space rather than kernel space, communicating directly with the hardware through SPI. This method offers greater flexibility for development and debugging, as user-space applications do not require kernel recompilation or system reboots for updates.

3.1 Building Linux Kernel Without AFE79xx SDK using PetaLinux Tools

This section describes building a standard Linux kernel for the ZCU102 board without AFE79xx driver integration, establishing the base system for the user-space driver approach. Since the AFE79xx SDK will be compiled and deployed as a user-space application in subsequent sections, the kernel build does not require custom driver modifications or AFE79xx-specific configuration. The resulting kernel image includes only the generic SPI device driver (spidev) and standard ZCU102 board support, providing the foundation upon which the user-space AFE79xx SDK will operate. This approach simplifies kernel management and allows the AFE79xx driver to be developed, updated, and deployed independently of the kernel.

The AFE79xx SDK needs a SPI interface to communicate with the device**. Section 5** describes reference design for adding processor and SPI to ZCU102 FPGA. We can follow exactly to generate the .xsa and .bit.

Follow the steps below to build kernel without AFE79xx driver. This includes only SPI needed for AFE79xx configuration:

Step 1: On the Linux terminal , create a petaLinux project by running command

```
petalinux-create -t project -n <Project_Name> -s <path to BSP>
```

Step 2: Change directory into the project folder <**Project_Name**> and run command

```
petalinux-config –get-hw-description <path to XSA>
```

Step 3: Configure kernel using command

```
petalinux-config -c kernel
```

The configuration menu will open. Navigate 'Device drivers' -> 'SPI support' and enable options Cadence SPI, Xilinx SPI, Xilinx Zynq QSPI and User mode SPI device driver.

Figure 3-1. Kernle Config Menu

Step 4: configure Root file system using command

```
petalinux-config -c rootfs
```

This will open the RootFs configuration menu. To enable GCC compiler navigate to 'Filesystem packages' -> 'misc' -> 'packagegroup-core-buildessential' under this select 'packagegroup-core-buidessential' and 'packagegroup-core-buidessential-dev'

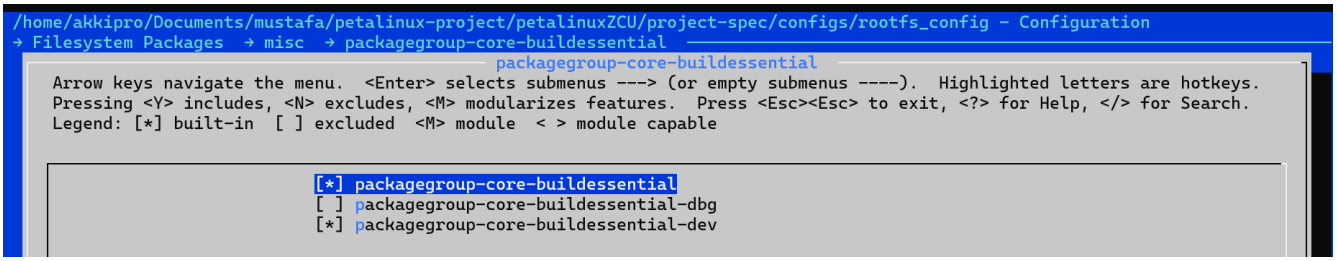

Figure 3-2. RootFs Config Menu for Build Dependencies

Step 5: The device tree must be updated to declare the AFE79xx device on the SPI bus, enabling the kernel to recognize and initialize the hardware. The **system-user.dtsi** file is used for custom device tree modifications in PetaLinux projects.

Go to device tree, add SPI to device tree by editing the **system-user.dtsi** file in **path**

```
<Project_Name>/project-spec/meta-user/recipes-bsp/device-tree/files/
```

In device tree add entries for SPI slaves as shown below:

```
/include/ "system-conf.dtsi"
/ {
};
&spi0 {
    num-cs = <1>;
    status = "okay";
    spidev@0 {
        compatible="rohm,dh2228fv";
        reg =<0>;
        spi-max-frequency = <10000000>;
    };
};
```

Figure 3-3. Example Device Tree

This will create a device in /dev by the name "/dev/spidev1.0" after bootup.

Step 6: Build your project using command:

```
petalinux-build
```

This process may take 30-60 minutes depending on your host system performance.

Step 7: Package the file using

```
petalinux-package --boot --fsbl < Project_Name >/images/linux/zynqmp_fsbl.elf --fpga <path to bit file> --u-boot –force
```

After successfully build, we should have all files needed for section 4.2. we can follow section 4.2 exactly to partition the SD card, copy the boot images and root filesystem to the appropriate partitions, and prepare the bootable media.

3.2 Integrating SPI with AFE79xx in User Space

The AFE79xx has the abstraction of low level SPI write and read function in *CAFE\Afe79xxUser\Src\tiAfe79_baseFunc.c*. Implement the SPI read writes function of SPIDEV for it to interact with the device.

This example uses halConfig member of struct afe79InstDeviceInfo to store the SPI handle. This is integrated in ti_afe79_afeSpiRawWrite and ti_afe79_afeSpiRawRead function.

```
/**
    @brief AFE SPI Write driver function.
    @details AFE SPI Write driver function. The contents of this function should be replaced by host SPI driver function.
    @param afeInst AFE Instance of AFE79_INST_TYPE type
    @param addr Address to be written to.
    @param data value to be written.
    @return Returns if the function execution passed or failed.
*/
TI_AFE_API_COMP uint8_t AFE79FNP(afeSpiRawWrite)(AFE79_INST_TYPE afeInst, uint16_t addr, uint8_t data)
{
    // afeLogDbg("WRITE: Addrress: 0x%X, data: 0x%X", addr, data);
    spi_write(afeInst->halConfig, addr, data);
    return TI_AFE_RET_EXEC_PASS;
}

/**
    @brief AFE SPI read driver function.
    @details AFE SPI read driver function and returns the read value as pointer. The contents of this function should be replaced by host SPI driver function.
    @param afeInst AFE Instance of AFE79_INST_TYPE type
    @param addr Address to be read from.
    @param readVal Pointer return of the value read.
    @return Returns if the function execution passed or failed.
*/
TI_AFE_API_COMP uint8_t AFE79FNP(afeSpiRawRead)(AFE79_INST_TYPE afeInst, uint16_t addr, uint8_t *readVal)
{
    spi_read(afeInst->halConfig, addr, readVal);
    // afeLogDbg("READ: Address: 0X%X, Read Val: 0X%X", addr, *readVal);
    return TI_AFE_RET_EXEC_PASS;
}
```

Figure 3-4. Example SPI Write and Read

The example code for creating the handle of spidev and compiling with SDK is provided as part of the package.

4 Reference Design for ZCU102 FPGA with Processor and SPI

This section describes the procedure for creating a hardware reference design in Vivado for the Zynq UltraScale+ MPSoC platform. The design incorporates the processing system with an SPI communication interface and produces the necessary hardware description files for software development and driver integration. Two key files generated are:

- Hardware Description (.xsa): Exports the complete hardware specification including the Zynq processing system configuration, SPI peripheral settings, memory addressing, and interrupt mappings. This file enables Linux build systems to automatically generate the device tree and configure kernel drivers that match your specific hardware implementation.
- FPGA Bitstream (.bit): Contains the synthesized hardware logic for programming the FPGA fabric. This file is packaged into the boot image (BOOT.BIN) and loaded during system initialization to configure the programmable logic.

Both files must be generated after completing the hardware design and are used in subsequent steps for building the Linux system and driver integration.

4.1 Configuring the Zynq Processor

This section describes how to set up the foundational block design in Vivado by adding and configuring the Zynq UltraScale+ MPSoC and Processor System Reset IP blocks. These form the base of the hardware design upon which the SPI interface and other peripherals are built.

Step 1: Open Vivado and create a new project. Select all default settings, then select required board. ZCU102 is used in example.

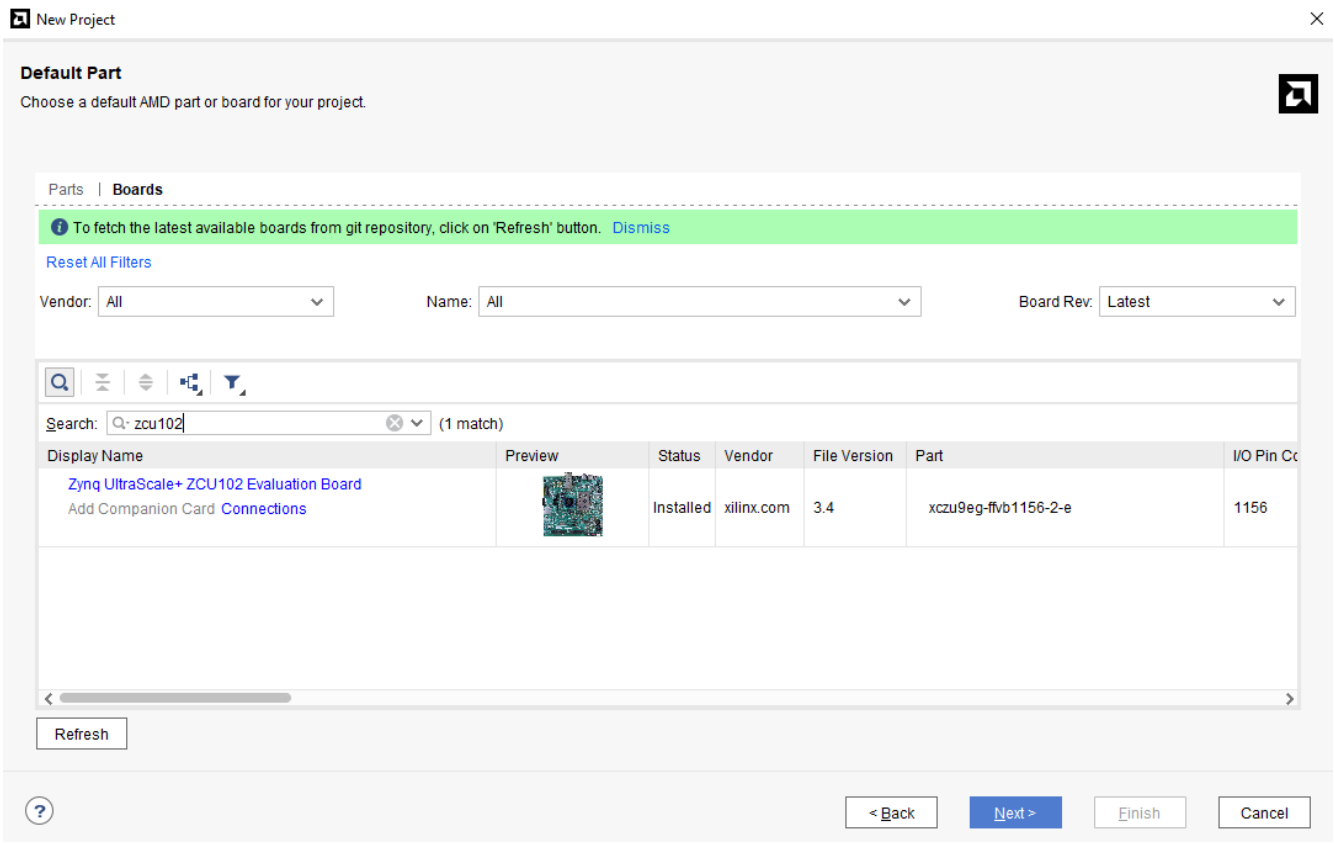

Figure 4-1. Board Selection for Project

Step 2: Create new block design by clicking on *Create Block Design*

Step 3: Click on Add IP or press Ctrl + I. Search Zynq UltraScale+ MPSoC IP Block and add by selecting it.

Figure 4-2. Adding IP to the Project

Step 4: Click on Add IP or press Ctrl + I. Search Processor System Reset IP Block and add by selecting it

Step 5: Run Block Automation. This design will look as below.

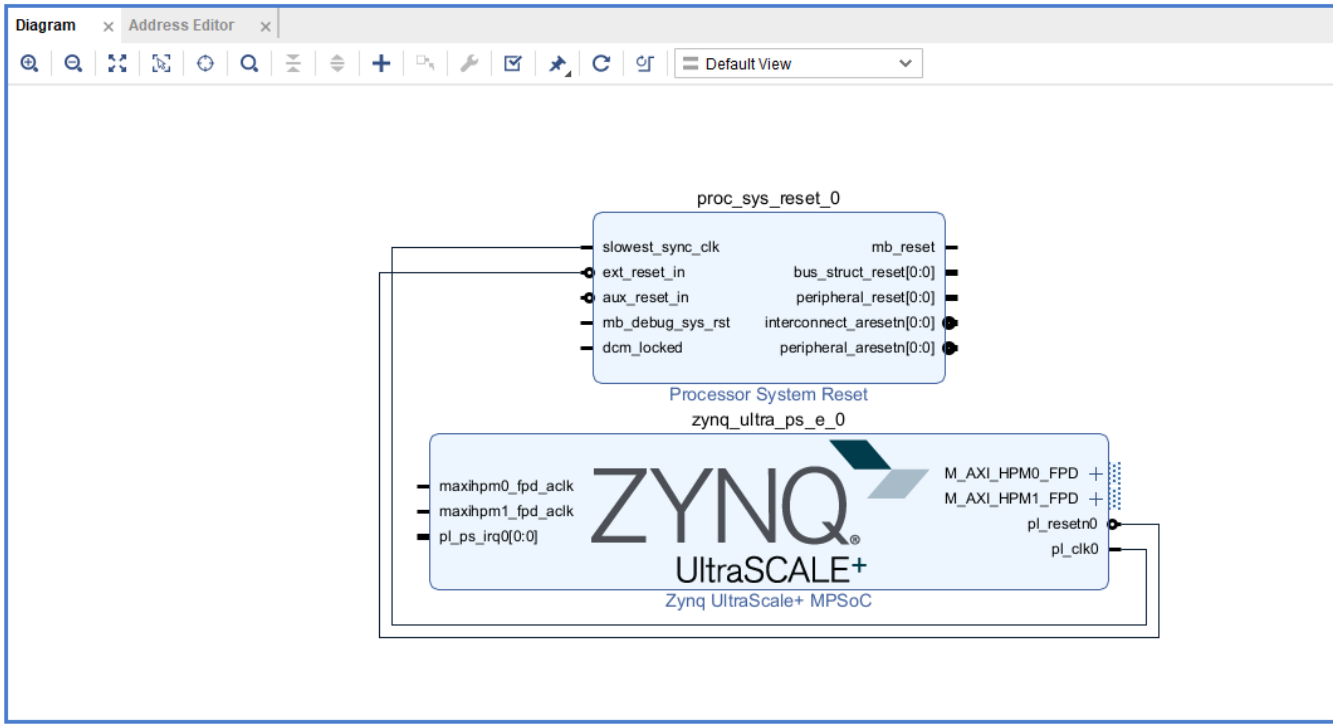

Figure 4-3. Adding Processor to the Project

4.2 Configuring the SPI Interface

This section describes how to enable and configure the integrated SPI controller in the Zynq processing system.

Follow these steps:

Step 1: Double Click on Zynq UltraScale+ MPSoC and go to I/O Configuration > Low Speed > I/O Peripherals and enable SPI (enable required number of chip selects) in Zynq UltraScale+ MPSoC and click OK.

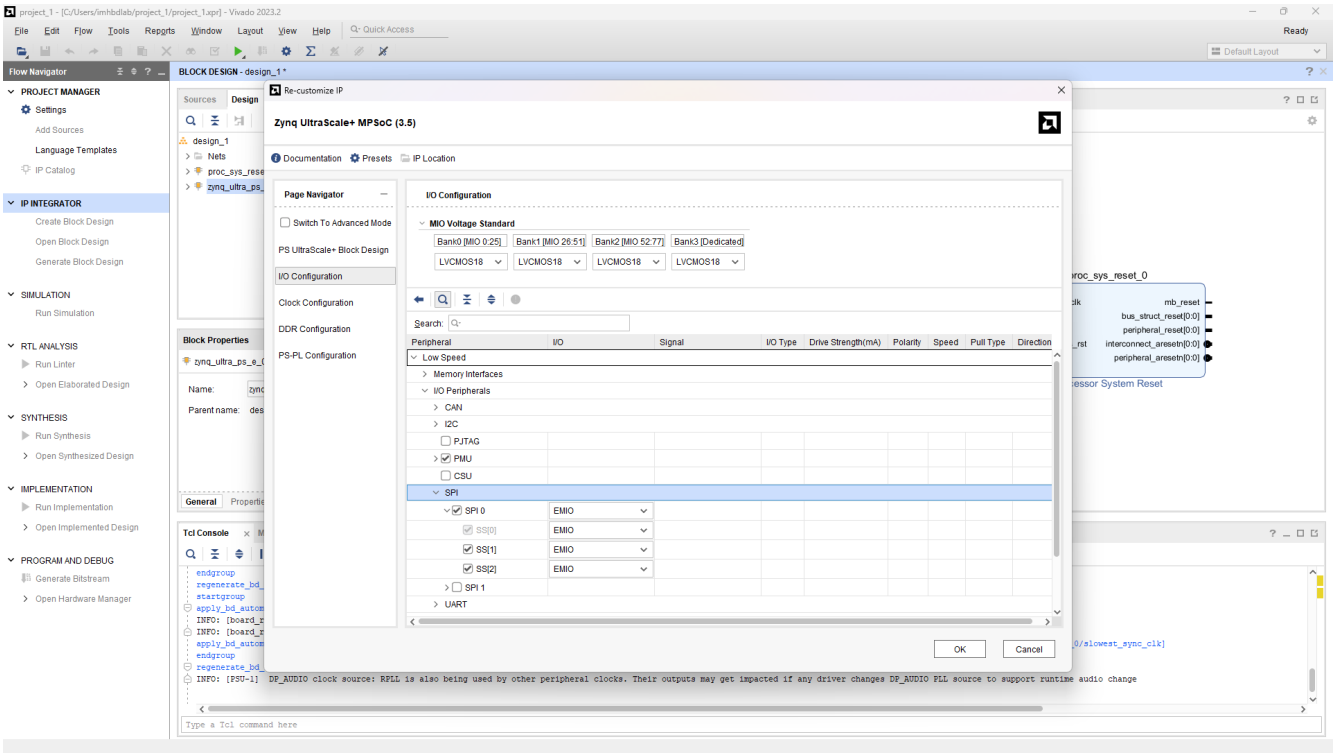

Figure 4-4. SPI Pin Config

Step 2: Click on + sign in front of SPI0. Right click on emio_spi0_sclk_o, emio_spi0_m_o, emio_spi0_m_i and emio_spi0_ss_o_n sequentially and select *Make External* or press *Ctrl+T* by selecting the SPI ports.

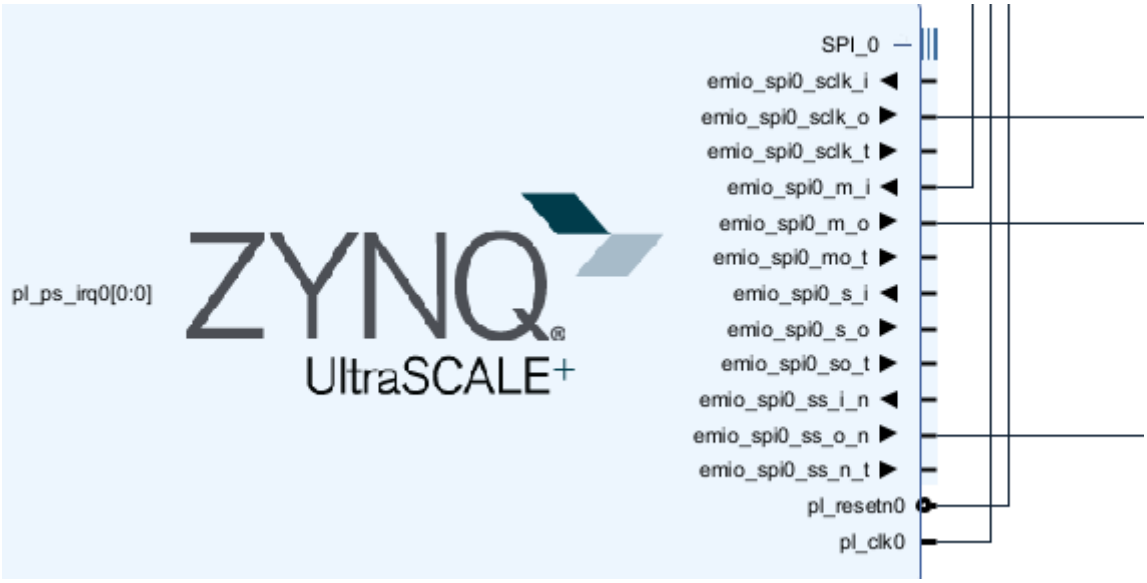

Figure 4-5. Selecting I/O Ports

Step 3: Double click on Zynq core and go to PS/PL Configuration > PS-PL Interfaces > Master Interface and disable AXI HPM0 FPD and AXI HPM1 FPD and click *OK*.

Step 4: Save and validate the design. Press Ctrl + S to save the design. Press F6 to validate the design. The final design should resemble Figure 4-6

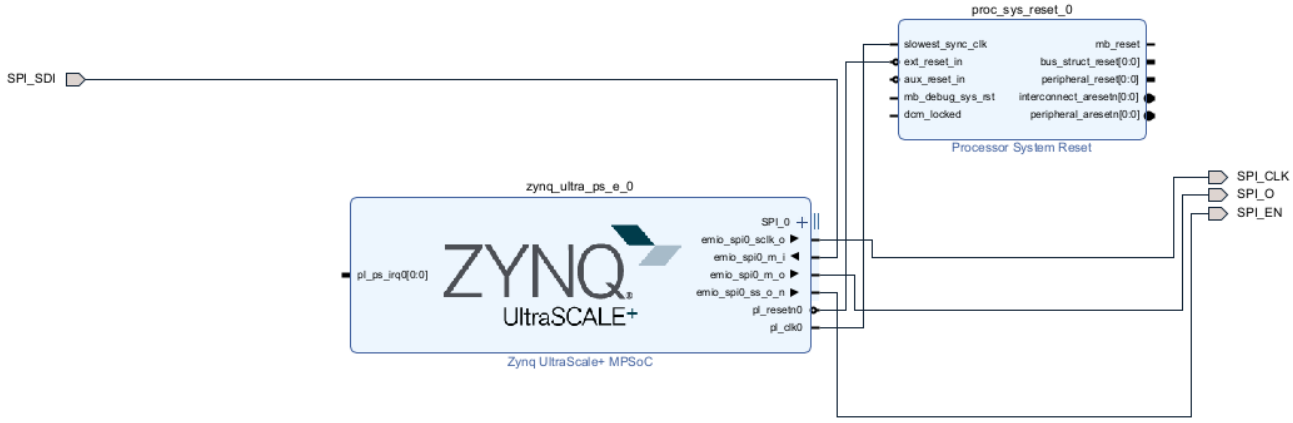

Figure 4-6. Example Project

Step 5: Under Sources, right click on block design and select *Create HDL Wrapper* and select default settings.

4.3 Generating Hardware Description

This section describes how to generate and export the hardware description file (.xsa) and FPGA bitstream (.bit) from the completed Vivado design.

Follow these steps:

Step 1: Run Synthesis. Open Synthesized design after completion.

Step 2: Go to Window -> IO Ports and assign required IO port pins to SPI lines.

SPI_0_14396 (4)	(Multiple)					✓	65	LVCMOS18	1.800		(Multiple)	
Scalar ports (4)												
SPI_CLK	OUT				AG3	✓	65	LVCMOS18	1.800		12	
SPI_EN	OUT				AJ5	✓	65	LVCMOS18	1.800		12	
SPI_O	OUT				AH3	✓	65	LVCMOS18	1.800		12	
SPI_SDI	IN				AJ6	✓	65	LVCMOS18	1.800			

Figure 4-7. SPI I/O Config

I/O mapping for AFE79xx EVM given below (For J4 HPC1 FMC connector on ZCU102).

Step 3: Click on Generate bitstream.

Step 4: Click on *File* > *Export* and select *Export Hardware*. This generates the .xsa file.

Step 5: Click on *File* > *Export* and select *Export bitstream file*. This generates the .bit file.

This completes the hardware design phase. The Zynq reference design with SPI interface has been exported from Vivado, generating the hardware description (.xsa) and bitstream (.bit) files required for the subsequent Linux kernel build and driver integration.

4.4 Booting ZCU102 Board with SD Card

This section describes the process of booting the ZCU102 board from the prepared SD card and verifying successful system initialization. The boot sequence includes loading the first stage bootloader, U-Boot, Linux kernel, device tree, and root filesystem from the SD card. Configure the boot mode settings of the board, connect to the serial console to monitor the boot process, and verify that the system boots successfully into a Linux command prompt.

Follow the steps:

Step 1: Configure Boot Mode

Set the ZCU102 boot mode switches for SD card boot:

- SW6[4:1] = OFF-OFF-OFF-ON

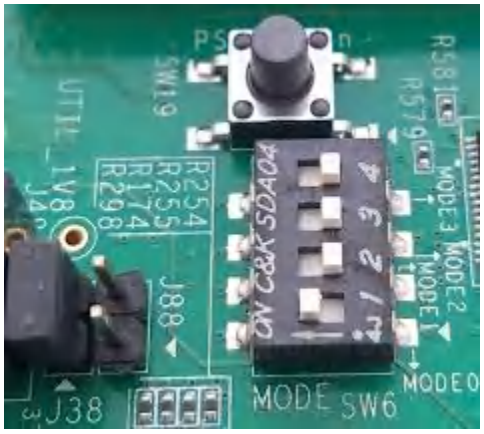

Figure 4-8. Switch State For SD Card Boot

Step 2: Insert the prepared SD card into the SD card slot on the ZCU102 board (J100).

Step 3: Connect Serial Console

Connect the USB-UART cable:

- Connect the micro-USB cable from the ZCU102 UART port (J83) to your PC
- The PC recognizes the FTDI USB-to-UART device

Step 4: Open Terminal Software

Launch a serial terminal application (PuTTY, Tera Term, or Minicom):

- **Port:** Select the COM port assigned to the ZCU102 (for example, COM3 on Windows, /dev/ttyUSB0 on Linux)
- **Baud rate:** 115200

Step 5: Power On the Board

Turn on the ZCU102 power switch (SW1).

Step 6: Monitor Boot Process

Observe the boot messages in the terminal. The system progresses through:

- First Stage Boot Loader (FSBL)
- U-Boot
- Linux kernel loading
- Root filesystem mount
- Login prompt

Step 7: Log In

At the login prompt, enter:

- **Username:** petalinux
- **Password:** \Set a new password.

5 Summary

This application note has provided complete procedures for integrating the AFE79xx SDK with Linux systems on the Zynq UltraScale+ ZCU102 platform using two distinct approaches. The in-tree kernel driver method demonstrated how to add AFE79xx driver source code to the Linux kernel tree, configure the kernel build system through Kconfig and Makefile modifications, and compile a complete Linux system with integrated driver support. This approach offers optimal performance and tight system integration suitable for production deployments. The user-space SDK method showed how to build AFE79xx functionality as application-layer software that communicates with hardware through the generic spidev interface, providing development flexibility without kernel dependencies. Both methods shared common foundational steps including Vivado hardware design creation with SPI interface configuration, hardware description file generation, and PetaLinux project setup. The document concluded with procedures for preparing bootable SD card media and booting the ZCU102 board to verify successful system initialization. Developers now have proven methodologies for implementing AFE79xx integration on Zynq platforms, with clear guidance for selecting between kernel-space and user-space approaches based on their specific project requirements, performance constraints, and maintenance considerations. The resulting systems provide complete AFE79xx device control capabilities ready for application development and deployment.

6 References

1. AMD, *PetaLinux Tools Documentation: Reference Guide*, reference guide.
2. Yocto Project, *Yocto Project Documentation*, webpage.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Last updated 10/2025