参考資料

Application Note

AFE79xx Linux ドライバ

Akshay Kumar, Navin Maheshwari, Mustafa Chikhliwala, Phanindra

概要

AFE79xx 高性能アナログ フロントエンドは、5G 基地局、Massive MIMO システム、ビーム フォーミング アプリケーションなど、無線通信インフラストラクチャにおいて重要な機能を担います。このアプリケーション ノートでは、Linux プラットフォームに AFE79xx デバイスを導入する際の複雑な統合上の課題を解決します。高度な RF システムを開発する通信機器メーカー、試験・計測機器メーカー、および防衛関連企業は、カーネル ドライバ方式とユーザー空間 SDK 方式の両方を対象とした実証済みの統合手順を活用できます。カーネル統合方式は、実運用インフラストラクチャ向けに最適な性能を備えた製品レベルのシステムを実現します。一方、ユーザー空間方式は、プロトタイプおよびテスト環境における開発サイクルを短縮します。一連の手順では、Vivado を使用した FPGA ハードウェア デザイン、PetaLinux および Yocto による Linux カーネル設定、SPI インターフェイスのセットアップ、ならびに起動可能なシステムの導入について、段階的に詳しく説明します。どちらの方式も ZCU102 プラットフォーム上でハードウェア検証済みであり、将来の更新やカスタマイズに対応できる柔軟性を維持しながら、AFE79xx サポートを効率的に実装できます。

目次

商標

English Document: SDAA427

1 概要

AFE79xx は、高性能かつ広帯域のマルチチャネル トランシーバ ファミリであり、四系統の RF サンプリング送信チェーン、四系統の RF サンプリング受信チェーン、および最大二系統の RF サンプリング補助デジタイズ チェーン (フィードバック パス) を統合しています。

このアプリケーション ノートでは、AFE79xx SDK を Linux ベース システムに統合するための手順を説明します。次の三つの統合方法について説明します:

1. **PetaLinux Tools を使用した Linux カーネル ドライバ** - Linux カーネル ソースと直接コンパイルします
2. **Yocto プロジェクトを使用する Linux カーネル ドライバ** - Linux カーネル ソースと直接コンパイルします
3. **ユーザ空間アプリケーション層** - カーネルを変更することなく動作します

カーネル方式 (方式 1 および方式 2) は、最適な性能とシステムとの緊密な統合を実現し、主にビルド システムとワークフローが異なります。ユーザー空間方式は、カーネルへの依存を避けながら、開発の柔軟性と保守の簡素化を実現します。方式の選択は、システムアーキテクチャ上の制約、性能要件、ビルド環境、および導入形態によって決まります。本書では、Linux カーネル開発、デバイス ドライバの概念、組込みシステム設計の基礎に精通していることを想定しています。

2 Linux カーネル ドライバ

AFE79xx カーネル ドライバは、デバイス SDK を、ユーザー空間アプリケーションで使用する高レベル API 関数と、カーネル空間で実行されるデバイス固有の処理という二つのコンポーネントに分離します。これにより、ユーザー アプリケーションへの SDK の統合が簡単になります。

- すべての **SPI 通信**はカーネル内に組み込まれています。
- 標準の Linux SPI サブシステムを使用して **SPI スレーブ デバイス**としてカーネルに統合することで、ドライバがスレーブ デバイスを排他的に管理し、処理のアトミック性を確保するとともに、他の SPI アクセスとの競合を防止します。
- ユーザー アプリケーションとの統合を容易にするため、ユーザー アクセスは**キャラクタ ドライバ**として提供され、標準の POSIX ファイル操作 (open、release、read、write、ioctl) を使用して動作します。
- 接続されたデバイスは、検出と識別を容易にするため、/dev 配下に **/dev/device_nameX.Y** 形式で表示されます。ここで、X は SPI マスタ ポート、Y はスレーブ ID を表します。例:/dev/afe79xx1.0
- **ti,device_name** 形式のコンパチブル文字列を使用することで、**デバイス ツリーによるドライバの自動呼び出しとスレーブ デバイスの割り当て**をサポートします。例:「ti,afe79xx」。この方式により、ドライバを再コンパイルすることなく、デバイスを動的に設定できます。
- **複数のデバイス**は、/dev の下に一意の ID を持つ独立したデバイスとしてユーザーに提示されます。ユーザー アプリケーションを簡素化するため、各デバイスのコンテキストは、各種の標準 Linux 機構を使用してカーネル コード内で一元的に管理されます。
- さまざまな Linux プロビジョニングを使用して、**スレッドセーフな動作**を検証します

以下に、ドライバ実装の主な機能を示します

以下に、Linux カーネルを使用したデバイス統合向けに提供されるデバイス SDK の一般的なディレクトリ構造を示します

```
Device_SDK/
├─ kernel-driver
└──── device_name/*
└─ userspace
├─────device_name/*
├──── example/*
└──── Makefile
```

kernel-driver 配下の device フォルダはカーネル モジュールのビルドに使用されます。一方、userspace 配下の device フォルダには高レベル API の定義が含まれており、ユーザー アプリケーションの一部として使用されます。また、example フォルダにはユーザー アプリケーションのサンプルが含まれ、Makefile には API をユーザー アプリケーションへ組み込む際の一般的なコンパイル方法が示されています。

2.1 PetaLinux Tools を使用した Linux カーネル ドライバ

このセクションでは、PetaLinux ツールを使用して、AFE79xx SDK のサポートを組み込んだ Linux カーネルをビルドする手順について説明します。この方法では、AFE79xx ドライバをカーネルのビルド中にカーネル ソース ツリー内で直接コンパイルする、インツリー ビルド方式を使用します。

この手順では、PetaLinux Tools 2023.2 以降がすでにインストールされ、使用する環境でセットアップ スクリプトが実行済みであることを前提としています。

PetaLinux のビルド プロセスを開始する前に、以下が準備されていることを確認します:

1. **ボード サポート パッケージ (BSP)**
   - AMD/Xilinx の Web サイトからダウンロードするか、PetaLinux のインストールに含まれているバージョンを使用します
2. **ハードウェア記述ファイル (.xsa)**
   - Vivado のハードウェア設計から生成されていますて
   - FPGA ビット ストリームを含める必要があります

リファレンス ハードウェア設計の作成と、ZCU102 ボード用の XSA ファイルの生成の手順については、**セクション 4** を参照してください。

以下の手順に従ってください:

ステップ 1: Linux ターミナルで、次のコマンドを実行して petaLinux プロジェクトを作成します

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

ステップ 2: ディレクトリをプロジェクト フォルダ **<Project_Name>** に変更し、コマンドを実行します

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

ステップ 3: コマンドを使用してカーネルを設定します

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

設定メニューが開きます。「デバイス ドライバ」「SPI サポート」に移動し、「Cadence SPI コントローラ」、「Xilinx SPI コントローラ共通モジュール」、「Xilinx Zynq GQSPI コントローラ」、「AMD SPI コントローラ」、および「ユーザー モード SPI デバイスドライバのサポート」の各オプションを有効にします。

図 2-1. カーネル設定メニュー

ステップ 4:コマンドを使用してルート ファイル システムを構成します

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

RootFs 設定メニューが開きます。GCC コンパイラを有効にするには、「Filesystem Packages」→「misc」→「packagegroup-core-buildessential」に移動し、「packagegroup-core-buildessential」と「packagegroup-core-buildessential-dev」を選択します

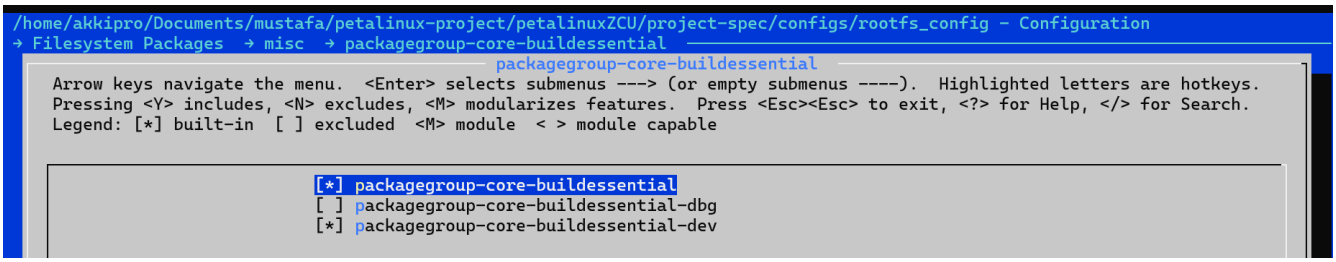

図 2-2. RootFs 設定メニュー

ステップ 5: 次のコマンドを実行して、カーネルのソースコードを編集可能な状態にします:

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

このコマンドを実行すると、カーネルのソースコードが作業ディレクトリに展開されます。コマンドの出力に表示されるパスを確認してください。通常、カーネル ソースは <Project_Name>/components/yocto/workspace/sources/linux-xlnx/ に展開されます。次のディレクトリに移動します:

```
cd <Path_to_kernel_Src>
```

ステップ 6: <Path_to_kernel_Src>/drivers/char/ の下に afe79 フォルダを作成し、SDK の kernel-driver 内にある afe79 フォルダの内容を、作成したフォルダにコピーします

ステップ 7: <Path_to_kernel_Src>/drivers/char/Makefile の末尾に、次の行を追加します

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

これにより、カーネル設定でドライバが有効になっている場合、カーネルのビルド システムによって AFE79xx ドライバがコンパイルされます。

ステップ 8: <Path_to_kernel_Src>/drivers/char/ にある Kconfig ファイル。ファイル末尾にある endmenu 行を見つけます。この行を endmenu の前に追加します。

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

これにより、カーネルの設定システムが AFE79xx の Kconfig ファイルを読み込みます。menuconfig インターフェイスにドライバが表示されます。メニューからドライバを有効または無効にします

ステップ 9: AFE79xx ドライバを有効にするようにカーネルを設定します。コマンドを実行します

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

これにより、カーネル設定メニューが開きます。「デバイス ドライバ」→「キャラクタ デバイス」に移動し、「AFE79xx」を選択して Y キーを押し、有効にします。

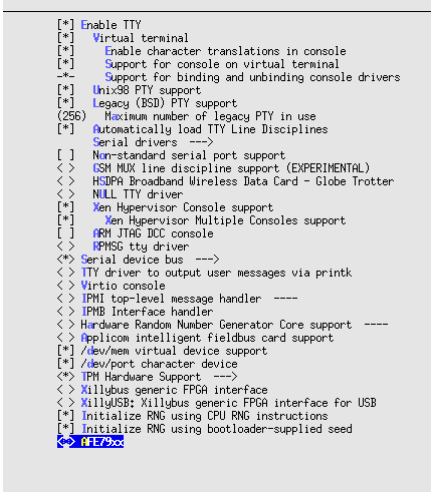

図 2-3. Char ドライバのカーネル設定メニュー

ステップ 10: カーネルがハードウェアを認識して初期化できるように、SPI バス上の AFE79xx デバイスを定義するデバイス ツリーの更新が必要です。PetaLinux プロジェクトでは、カスタムのデバイス ツリー変更に system-user.dtsi ファイルを使用します。

次の path にある system-user.dtsi ファイルを編集し、デバイス ツリーに SPI デバイスを追加します

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

デバイス ツリーで、図 2-4 に示すように SPI スレーブのエントリを追加します

```
/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>;
    };
};
```

図 2-4. デバイス ツリーの例

これにより、起動後に /dev/afe79xx1.0 という名前で /dev にデバイスが作成されます。

デバイス ドライバをバインドするには、ノードに互換性のある文字列 ti,afe79xx, を指定する必要があります。

ステップ 11: 次のコマンドを使用してプロジェクトをビルドします:

```
petalinux-build
```

この処理には、ホスト システムの性能に応じて 30 ～ 60 分程度かかる場合があります。

ステップ 12: 以下を使用してファイルをパッケージ化します

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

ビルドが正常に完了すると、ユーザーは Linux ブート イメージの準備に必要なすべてのファイルを取得できます。

2.2 Yocto プロジェクトを使用した Linux カーネル ドライバ

このセクションでは、Yocto プロジェクトを使用してビルドする Linux イメージに AFE79xx SPI カーネル ドライバを統合する方法について説明します。この方法は、特定のボードやベンダ固有のツールチェーンには依存しません。Yocto 互換のボード サポート パッケージ (BSP) が存在するあらゆるハードウェア プラットフォームに適用できます。

このドライバは、**meta-afe79** という名前の事前にビルド済みのカスタム Yocto レイヤとして提供され、既存の Yocto プロジェクトに追加できます。すでに動作可能な Yocto ビルド環境を構築済みのユーザーは、セクション 2.2.4 に直接進むことができます。初めて Yocto 環境をセットアップするユーザーは、すべての手順を順番に実行してください。

注

このセクションのすべての設定シンタックスは、**Yocto Langdale (4.1.x)** の規則に準拠しています。Yocto は、各リリースで大幅なシンタックスの変更を受けています。変数の代入演算子は、Dunfell 以前のリリースで使用されていたアンダースコア表記 (例：IMAGE_INSTALL_append) から、Honister で導入され、Langdale を含むそれ以降のすべてのリリースで使用されるコロン表記 (IMAGE_INSTALL:append) に変更されました。別の Yocto バージョンを使用して作業するユーザーは、シンタックスを確認し、それに応じて更新する必要があります。バージョン固有の変更については、https://docs.yoctoproject.org で Yocto プロジェクトの公式移行ガイドを参照してください。

2.2.1 Yocto プロジェクトの概要

Yocto プロジェクトは、組込みシステム向けのカスタム Linux ディストリビューションを開発者が作成できるオープン ソースのビルド フレームワークです。Yocto プロジェクト自体は Linux ディストリビューションではなく、Linux ディストリビューションを構築するためのツールと手法の集合です。Yocto の中核では、**BitBake** と呼ばれるビルド エンジンが使用され、**レシピ**と呼ばれるメタデータ ファイルを処理し、ソフトウェア コンポーネントのフェッチ、コンパイル、およびパッケージ化を行います。

Yocto の基本的な組織ユニットは**レイヤ**です。レイヤは、関連するレシピ、設定ファイル、およびパッチをまとめたもので、ビルドに特定の機能を追加します。主要なハードウェア プラットフォームの多くには公式レイヤが用意されており、通常はボードまたは SoC のベンダからボード サポート パッケージとして提供されています。カスタム レイヤを使用すると、アップストリーム レイヤを変更することなく、AFE79xx ドライバなどの独自ソフトウェアやプロジェクト固有のソフトウェアを追加できます。

AFE79xx ドライバは Linux カーネル ドライバとしてパッケージ化されており、カスタム Yocto レイヤの一部として供給されます。ドライバのソース コードは、ロード可能なカーネル モジュール (.ko ファイル) としてコンパイルされます。このモジュールは Yocto イメージのビルド プロセスの一部としてビルドされ、ルート ファイル システムにインストールされます。ターゲット システムが Linux を起動すると、ドライバが実行中のカーネルに自動的にロードされ、SPI 経由で AFE79xx デバイスとの通信が可能になります。Yocto には、この種のカーネル ドライバをビルドおよびパッケージ化するための仕組みが標準で用意されており、提供される meta-afe79 レイヤはこの機能を最大限に活用しています。

2.2.2 ホスト システム要件

Yocto イメージを構築するには、次のホスト システム構成が必要です。

オペレーティング システム

Ubuntu 18.04 LTS 以降を推奨します。Yocto でサポートされている他の Linux ディストリビューションも使用できます。完全な一覧については、Yocto の公式ドキュメントを参照してください。

表 2-1. システム リソース

関連資料	最小	推奨
RAM	8GB	16GB

表 2-1. システム リソース (続き)

空きディスク容量	100GB	200GB
CPU コア	4	8 人以上

必須パッケージ

続行する前に、次のパッケージをホスト システムにインストールします:

```
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 Yocto ビルド環境のセットアップ

このセクションでは、Yocto ビルド環境を初期化する手順について説明します。ターゲットボード用に、動作する Yocto の設定をすでに行っているユーザーは、**セクション 2.2.4** に進むことができます。

ステップ 1: Poky のダウンロード

Poky は Yocto プロジェクトのリファレンス ディストリビューションであり、BitBake およびコア メタデータ レイヤを含んでいます。使用する Yocto リリースに対応するバージョンの Poky をクローンします。以降、Poky のルート ディレクトリを **project_directory** と表記します。次の例では、Langdale 分岐を使用します:

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

ステップ 2: ビルド環境の初期化

project_directory ディレクトリから Yocto 環境セットアップ スクリプトを読み込み、ビルド ディレクトリを作成してシェル環境を設定します:

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

このコマンドは、ビルド ディレクトリを作成し、デフォルトの設定ファイルを生成して、シェルのパスに BitBake を追加します。このコマンドは、新しい端子セッションを開始するたびに実行する必要があります。

重要: このコマンドは、必ずプロジェクト ルート ディレクトリから実行してください。ビルド ディレクトリ内から実行すると、入れ子になったビルド フォルダが作成され、すべての設定がリセットされます。

2.2.4 ボード固有のレイヤの設定

ステップ 1: ボード固有のレイヤをダウンロードする

各ハードウェア プラットフォームには、ボード向けのマシン設定、カーネル、およびブートローダーのレシピを提供する、対応する BSP レイヤが必要です。BSP レイヤは通常、ボードまたは SoC のベンダから提供され、個別にクローンします。使用する Yocto リリースと互換性のある適切な BSP レイヤおよび分岐を特定するには、ボード ベンダのドキュメントを参照してください。

Xilinx ベースのプラットフォームでは、関連するレイヤとして meta-xilinx、meta-xilinx-bsp、meta-xilinx-standalone、および meta-xilinx-tools があり、いずれも Xilinx の GitHub リポジトリから入手できます。Raspberry Pi、NXP、Intel などの他のプラットフォームでは、ベンダが提供する BSP レイヤと同等のものを使用する必要があります。以下の例に示すように、git clone コマンドを使用して各レイヤをクローンできます:

AMD (旧 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
```

さらに、多くの BSP レイヤは OpenEmbedded レイヤ コレクション (meta-openembedded) のパッケージに依存しています。このレイヤ群についても、適切な分岐をクローンします:

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

ステップ 2:レイヤの追加

bitbake-layers コマンドを使用して、必要なすべてのレイヤをビルドに追加します。レイヤの正確なセットは、ターゲット ボードによって異なります。少なくとも、Poky のコア レイヤ、ボード用の BSP レイヤ、および AFE79 レイヤが必要です。前の手順で複製されたレイヤのみを追加するには、次のコマンドを使用します:

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

たとえば、ZCU102 では、次のレイヤが必要です:

```
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
```

コマンドを使用して、必要なすべてのレイヤがアクティブであることを確認します。

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

ステップ 3: ハードウェア記述ファイルとターゲット マシンの追加

多くの Yocto BSP ワークフローでは、ターゲット プラットフォームのハードウェア設計全体を記述したハードウェア記述ファイルが必要です。Yocto は、**local.conf** 形式経由でハードウェア記述ファイルを使用して、特定のハードウェア設計用の正しい FSBL、PMU ファームウェア、デバイス ツリーを自動的に生成します。

conf/local.conf は、ハードウェア記述ファイルのパスとマシン変数を取得して、ビルド用の構成を正しく提供します。

ビルド ディレクトリ内の <project_directory/build_directory>/conf/local.conf ファイルを開き、MACHINE 変数を対象ボードに対応する識別子に設定します。この識別子は、ボードの BSP レイヤで定義されます。たとえば、ZCU102 ボードでは、zcu102-zynqmp という識別子が使用されます。

。

正しいマシン名については、ボードの BSP ドキュメントを参照してください。

Xilinx では .XSA ファイルを使用し、Intel では SOPC 情報ファイルを使用します。

Xilinx の場合:

ホスト システム上の既知の場所に XSA ファイルを配置します。推奨される場所は、カスタム レイヤ内です:

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

次に、ビルド ディレクトリ内の <project_directory/build_directory>/conf/local.conf ファイルに以下の行を追加し、パス、ファイル名、および machine 変数を、使用する XSA ファイルと対象ボードに合わせて置き換えます:

```
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"
```

注

XILINX_RELEASE_VERSION には、XSA ファイルを生成した Vivado のバージョンと一致する値を指定する必要があります。Vivado 2023.2 で生成された XSA ファイルを使用する場合は、XILINX_RELEASE_VERSION = "v2023.2" と設定する必要があります。バージョンが一致していない場合、ハードウェア記述の処理中に XSCT ツールでエラーが発生します。

使用する BSP レイヤの要件に従って、ハードウェア ハンドオフのパスを設定します。正しい変数名とパス形式については、BSP レイヤの README を参照してください。これらはベンダやプラットフォームの世代によって異なります。

2.2.5 AFE79xx Yocto レイヤの統合

AFE79xx ドライバは、すぐに使用できる **meta-afe79** という名称の Yocto レイヤとして提供されています。このレイヤには、必要なすべてのレシピ、ドライバのソース コード、およびリファレンス用のデバイス ツリー オーバーレイが含まれています。デバイス ツリーの設定を除き、対象となるボードや Yocto のバージョンにかかわらず、統合手順は同一です。

ステップ 1: プロジェクトにデバイス レイヤを追加し

TI が提供する SDK から、meta-afe79 レイヤのディレクトリを Yocto の **project_directory** にコピーします。次に、Yocto 環境を有効にした状態でビルド ディレクトリから、以下のコマンドを実行してレイヤを BitBake に登録します:

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

アクティブなレイヤ リストにレイヤが表示されることを確認します:

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

ステップ 2: 最終的な Linux イメージにデバイス ドライバ パッケージを組み込む

ビルド ディレクトリ内の <project_directory/build_directory>/conf/local.conf ファイルに、次の行を追加します。これにより、Yocto は AFE79xx ドライバ モジュールをコンパイルし、出力イメージのルート ファイル システムに組み込みます:

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

注

Honister より前の Yocto バージョンでは、コロン表記をアンダースコア表記に置き換えます:

```
IMAGE_INSTALL_append = " afe79"
```

イメージに組み込まれたドライバ モジュールは、ターゲットのファイル システム上の /etc/modules-load.d/afe79.conf に配置されたモジュール自動ロード設定ファイルにより、システム起動時に自動的に読み込まれるよう設定されています。

GCC コンパイラと開発ツールをイメージに追加するには、以下の行を追加します。これにより、ユーザー空間でのコンパイルが容易になります。

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

ステップ 3: デバイス ツリー オーバーレイ ファイルを設定する

Linux ドライバは、管理対象のハードウェアを検出するためにデバイス ツリーを使用します。デバイス ドライバをバインドするには、適切な SPI バス上に、compatible プロパティとして **ti,afe79xx** を指定したノードがデバイス ツリーに含まれている必要があります。このノードを追加する方法は、使用するボードの BSP で採用されている Yocto ワークフローによって異なります。

Xilinx ベースのボード向けには、meta-afe79 レイヤに、すぐに使用できるオーバーレイ ファイル recipes-bsp/device-tree/files/system-user.dtsi と、このオーバーレイをビルドに自動的に組み込むための対応する.bbappend ファイルが含まれています。ビルドを開始する前に、system-user.dtsi ファイルを開き、SPI ノードのラベルとプロパティがボードのハードウェア設計と一致していることを確認します。提供されているファイルでは、ZCU102 評価ボードで使用される SPI コントローラに対応する spi0 が参照されています。使用するボードで異なる SPI コントローラやノード ラベルが使われている場合は、それに合わせてファイルを修正します。

Xilinx ZCU102 ボードの場合は、例として、以下に示すように SPI0 の SPI スレーブ用エントリを追加します:

デバイス ツリーに移動し、**<path_to_meta-afe79>meta-afe79\recipes-bsp\device-tree\files** の **system-user.dtsi** ファイルを編集して、デバイス ツリーに SPI を追加します

Xilinx ZCU ボードの場合は、例として、以下に示す SPI スレーブのエントリをデバイス ツリーの SPI0 ノードに追加します:

```
/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>;
    };
};
```

図 2-5. デバイス ツリーの例

これにより、システム起動後に /dev/afe79xx1.0 という名前のデバイス ノードが /dev 配下に作成されます。SPI ノードのラベルとプロパティが、使用するボードのハードウェア設計と一致していることを確認します。

ステップ 4: カスタム デバイス ツリー ファイルをベース デバイス ツリー ファイルに含める

Xilinx では device-tree.bbappend ファイルを、Intel では linux-socfpga.bbappend ファイルを使用し、元のレシピを変更することなく、カスタム .dtsi ファイルをベースのデバイス ツリーに組み込みます。

meta-afe79 レイヤに含まれる meta-afe79/recipes-bsp/device-tree/device-tree.bbappend ファイルは、Xilinx の DTG ワークフロー専用です。その他のプラットフォームを使用する場合は、このファイルをレイヤから削除するか、使用している BSP に適した仕組みに置き換えます。

ステップ 5: イメージをビルドする

対象のイメージ レシピに対して、BitBake のビルド コマンドを実行します:

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

<image-name> を、使用するプロジェクトに適したイメージ レシピ名に置き換えます。一般的なオプションは次のとおりです:

- **core-image-minimal** - 必要最小限のパッケージのみを含む、起動可能な最小構成のイメージ。小さなフットプリントが求められる本番環境への導入に適しています。
- **core-image-minimal-dev** - 開発ツールとヘッダを使用して **core-image-minimal** を拡張します。ターゲット ボード上でアプリケーションを直接コンパイルまたはデバッグする場合に便利です。
- **core-image-full-cmdline** - 標準的な Linux コマンド ライン ユーティリティをより豊富に含む、コンソール向けのイメージ。開発およびテスト環境に適しています。
- **petalinux-image-minimal** - Xilinx BSP レイヤを使用する場合に使用できる Xilinx 固有の最小イメージ。

初回のビルドでは、BitBake が必要なすべてのソース コードをダウンロードしてコンパイルするため、ホスト システムの性能やネットワーク速度によって 1 ～ 3 時間程度かかります。その後のビルドは、共有状態のキャッシュにより大幅に高速化されます。

ビルド中に、プライマリのダウンロード URL へのアクセス失敗を示す警告メッセージが表示されることがありますが、これは正常です。BitBake はミラー サーバーを使用して自動的に再試行するため、これらの警告はビルドの失敗を示すものではありません。

2.2.6 統合の確認

ビルドしたイメージをターゲット ボードに書き込み、Linux を起動した後、次のコマンドを実行して AFE79xx ドライバが正常に読み込まれていることを確認します。

モジュールがロード済みモジュール リストにあることを確認します:

```
lsmod | grep afe79
```

カーネル ログでドライバ初期化メッセージを確認します:

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

初期化が成功すると、次のような出力が生成されます:

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

AFE79xx SPI ドライバの初期化

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

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

キャラクタ デバイス ファイルが作成されたことを確認します:

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

ドライバは、SPI バス上で検出された AFE79xx インスタンスごとに一つのデバイス ファイルを作成します。ファイル名はバス番号とチップ セレクトのインデックスに基づいて付けられます。たとえば、/dev/afe79xx1.0 や /dev/afe79xx2.0 です。

2.2.7 トラブルシューティング

表 2-2. トラブルシューティング

問題	症状	分解能
モジュールが起動時にロードされていません	lsmod の出力に afe79 のエントリが表示されません	IMAGE_INSTALL:append = "afe79" が local.conf に存在することを確認します。ターゲットに /etc/modules-load.d/afe79.conf が存在することを確認します。
ビルド シンタックス エラー	IMAGE_INSTALL または KERNEL_MODULE_AUTOLOAD に関連する BitBake 解析エラー	Yocto のバージョンを確認します。Honister より前のバージョンでは、アンダースコア シンタックスを使用します
ドライバのプローブに失敗しました	dmesg に初期化メッセージがありません	デバイス ツリーの正しい SPI バス上に、compatible = "ti,afe79xx" および status = "okay" が設定されたノードが存在することを確認します。
デバイス ファイルが作成されていません	/dev/afe79* が存在しません	dmesg でドライバのバインディング エラーを確認します。デバイス ツリーで SPI コントローラが有効になっていることを確認します。
SPI 通信障害	アプリケーションから誤った読み取り値またはエラーが返されます	正しいデバイス ファイルが使用されていることを確認します。SPI 周波数とモードがハードウェアと一致していることを確認します。Xilinx プラットフォームでは、FPGA ビット ストリームによって SPI 信号が FMC コネクタ経由で AFE79xx デバイスへ正しく配線されていることを確認します。
ネストされたビルド ディレクトリ	設定が予期せずリセットされました	環境スクリプトは、必ずプロジェクトのルート ディレクトリから読み込んでください。ビルド ディレクトリ内から読み込まないでください。誤って作成した場合は、ネストされたビルド サブディレクトリを削除します。

2.3 Linux ブート イメージの準備

このセクションでは、SD カードのパーティション作成、ブート イメージとルート ファイル システムの各パーティションへのコピー、および起動用メディアの準備方法について説明します。

以下の手順に従います:

ステップ 1: 以下のコマンドを使用して SD カードをフォーマットし、2 つのパーティションを作成します

```
sudo gparted
```

一つ目のパーティションは fat32 形式とし、ラベルを BOOT に設定して 1GB を割り当てます。二つ目のパーティションは ext4 形式とし、ラベルを rootfs に設定して残りの領域をすべて割り当てます。

ステップ 2: <project_name>/images/linux / フォルダにある BOOT.bin、image.ub、および boot.scr を BOOT パーティションにコピーします

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

ステップ 3: 次のコマンドを使用して、<project_name>/images/linux/ にある rootfs.tar.gz を rootfs パーティションに展開します

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

ステップ 4: 次のコマンドを使用して、<project_name>/images/linux/ にある rootfs.cpio ファイルをコピーします

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

ステップ 5: SD カードを取り出します (rootfs の破損を防ぐため、必ず安全に取り出してください)。

ステップ 6: SD カードを ZCU ボードに挿入し、UART ケーブルで PC に接続します。SW6 を SD カード ブート モードに設定してボードの電源を入れます。Linux が起動し、Putty など端末ソフトウェアからコンソールにアクセスできます。

このセクションの完了により、PetaLinux ツールとインツリー ビルド方式を使用して、AFE79xx ドライバを組み込んだ Linux カーネルのビルドが正常に完了しました。AFE SPI デバイスがデバイス ツリーに正しく登録され、キャラクタ デバイスとして公開されたことで、AFE ドライバの動作およびアプリケーション開発を行うためのシステム準備がすべて整いました。

2.4 ユーザー空間から AFE79xx デバイスを動作させる

CAFE パッケージは、ユーザー空間アプリケーションから AFE79xx カーネル ドライバと通信するための高レベル API を提供します。AFE79xx デバイスを設定および制御するには、これをソフトウェア スタックとともにコンパイルしてください。

次の例は、デバイス インスタンスを使用していくつかの高レベル API を呼び出す方法を示しています。

```
#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;
}
```

注

従うべき正確な API 呼び出し手順および各 API の詳細については、デバイス API のドキュメントを参照してください。

ユーザ スペース ライブラリのインストール

SDK 内の userspace フォルダに移動し、次のコマンドを実行します:

1. make
2. sudo make install

ユーザー空間ライブラリを使用したユーザー アプリケーションのコンパイル:

ライブラリとヘッダは既に linux にインストールされているため、コンパイル時にパスを指定する必要はありません。デバイス API を呼び出すすべての箇所で、単一のヘッダ ファイル tiDeviceName.h (tiAfe79xx.h) をインクルードし、リンク時に -ldevicename (-lafe79xx) を指定します。

3 アプリケーション レイヤで AFE79xx SDK をビルドする

このセクションでは、AFE79xx SDK をユーザー空間アプリケーションとしてビルドする方法について説明します。これは、「アプリケーション層での AFE79xx SDK のビルド」で説明したインツリー カーネル ドライバ方式に代わる統合方法です。この方式では、AFE79xx ドライバの全機能をカーネル空間ではなくユーザー空間に実装し、SPI を介してハードウェアと直接通信します。この方法は、ユーザー空間アプリケーションを更新する際にカーネルの再コンパイルやシステムの再起動が不要なため、開発およびデバッグの柔軟性が向上します。

3.1 PetaLinux Tools を使用した AFE79xx SDK を含まない Linux カーネルのビルド

このセクションでは、AFE79xx ドライバを組み込まずに ZCU102 ボード向けの標準 Linux カーネルをビルドし、ユーザー空間ドライバ方式の基盤となるシステムを構築する手順について説明します。以降のセクションでは、AFE79xx SDK をユーザー空間アプリケーションとしてコンパイルおよび導入するため、カーネルのビルド時にドライバのカスタム変更や AFE79xx 固有の設定を行う必要はありません。生成されるカーネル イメージには、汎用 SPI デバイス ドライバ (spidev) と標準の ZCU102 ボードのサポートのみが含まれ、ユーザー空間で AFE79xx SDK を動作させるための基盤となります。この方式により、カーネル管理が簡素化され、AFE79xx ドライバをカーネルとは独立して開発、更新、導入できます。

AFE79xx SDK は、デバイスと通信するために SPI インターフェイスを必要とします。**セクション 5** では、ZCU102 FPGA にプロセッサと SPI を追加するためのリファレンス デザインについて説明します。以下の手順にそのまま従って、.xsa ファイルと .bit ファイルを生成できます。

以下の手順に従って、AFE79xx ドライバを組み込まずにカーネルをビルドします。これには、AFE79xx 構成に必要な SPI のみが含まれます:

ステップ 1: Linux ターミナルで、次のコマンドを実行して petaLinux プロジェクトを作成します

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

ステップ 2: ディレクトリをプロジェクト フォルダ **<Project_Name>** に変更し、コマンドを実行します

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

ステップ 3: コマンドを使用してカーネルを設定します

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

設定メニューが開きます。「デバイス ドライバ」→「SPI サポート」に移動し、「Cadence SPI」、「Xilinx SPI」、「Xilinx Zynq QSPI」、および「ユーザー モード SPI デバイス ドライバ」を有効にします。

図 3-1. カーネル設定メニュー

ステップ 4:コマンドを使用してルート ファイル システムを構成します

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

RootFs 設定メニューが開きます。GCC コンパイラを有効にするには、「Filesystem Packages」→「misc」→「packagegroup-core-buildessential」に移動し、「packagegroup-core-buildessential」と「packagegroup-core-buildessential-dev」を選択します

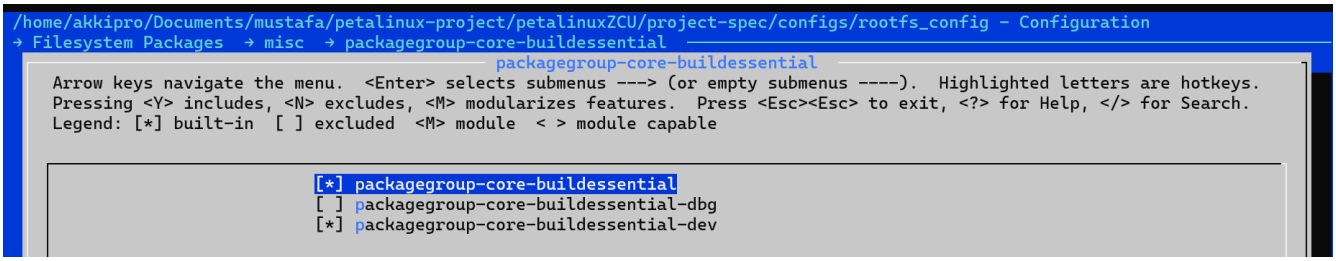

図 3-2. ビルド依存の RootFs 設定メニュー

ステップ 5: カーネルがハードウェアを認識して初期化できるように、SPI バス上の AFE79xx デバイスを定義するデバイス ツリーの更新が必要です。PetaLinux プロジェクトでは、カスタムのデバイス ツリー変更に system-user.dtsi ファイルを使用します。

次の **path** にある **system-user.dtsi** ファイルを編集し、デバイス ツリーに SPI デバイスを追加します

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

デバイスツリーで、以下に示すように SPI スレーブのエントリを追加します:

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

図 3-3. デバイス ツリーの例

これにより、システム起動後に /dev/spidev1.0 という名前のデバイス ノードが /dev 配下に作成されます。

ステップ 6: 次のコマンドを使用してご利用のプロジェクトをビルドします:

```
petalinux-build
```

この処理には、お客様のホスト システムの性能に応じて 30 ～ 60 分程度かかる場合があります。

ステップ 7: 以下を使用してファイルをパッケージ化します

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

ビルドが正常に完了すると、セクション 4.2 で必要となるすべてのファイルが生成されます。その後はセクション 4.2 の手順に従って、SD カードのパーティション作成、ブート イメージとルート ファイル システムの各パーティションへのコピー、および起動用メディアの準備を行います。

3.2 ユーザー空間で SPI と AFE79xx を統合

AFE79xx は、低レベルの SPI 書き込みおよび読み取り機能を CAFE\Afe79xxUser\Src\tiAfe79_baseFunc.c に抽象化しています。デバイスと通信できるように、SPIDEV の SPI 読み書き関数を実装します。

この例では、struct afe79InstDeviceInfo の halConfig メンバーを使用して SPI ハンドルを格納します。これは、ti_afe79_afeSpiRawWrite 関数および ti_afe79_afeSpiRawRead 関数に統合されています。

```
/**
    @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;
}
```

図 3-4. SPI 書き込みと読み取りの例

spidev のハンドルを作成し、SDK とともにコンパイルするためのサンプル コードがパッケージに含まれています。

4 プロセッサと SPI を搭載した ZCU102 FPGA 向けリファレンス デザイン

このセクションでは、Vivado を使用して Zynq UltraScale+ MPSoC プラットフォーム向けのハードウェア リファレンス デザインを作成する手順について説明します。このデザインには、SPI 通信インターフェイスを備えたプロセッシング システムが組み込まれており、ソフトウェア開発およびドライバ統合に必要なハードウェア記述ファイルが生成されます。生成される主なファイルは、次の二つです:

- ハードウェアの説明 (.xsa):Zynq プロセッシング システムの設定、SPI ペリフェラルの設定、メモリ アドレス指定、および割り込みマッピングを含む、完全なハードウェア仕様をエクスポートします。このファイルにより、Linux ビルド システムはデバイス ツリーを自動生成し、特定のハードウェア実装に適合するカーネル ドライバを設定できます。
- FPGA ビットストリーム (.bit):FPGA ファブリックをプログラミングするための合成ハードウェア ロジックが含まれています。このファイルはブート イメージ (BOOT.BIN) に組込まれ、システムの初期化時に読み込まれてプログラマブル ロジックを設定します。

どちらのファイルも、ハードウェア デザインの完成後に生成する必要があり、以降の Linux システムのビルドおよびドライバ統合の手順で使用されます。

4.1 Zynq プロセッサの設定

このセクションでは、Zynq UltraScale+ MPSoC およびプロセッサ システム リセット IP ブロックを追加して設定し、Vivado で基盤となるブロック デザインを構築する方法について説明します。これらは、SPI インターフェイスおよびその他のペリフェラルを構築するためのハードウェア デザインの基盤となります。

ステップ 1: Vivado を起動し、新しいプロジェクトを作成します。すべてのデフォルト設定を選択し、必要なボードを選択します。この例では ZCU102 を使用しています。

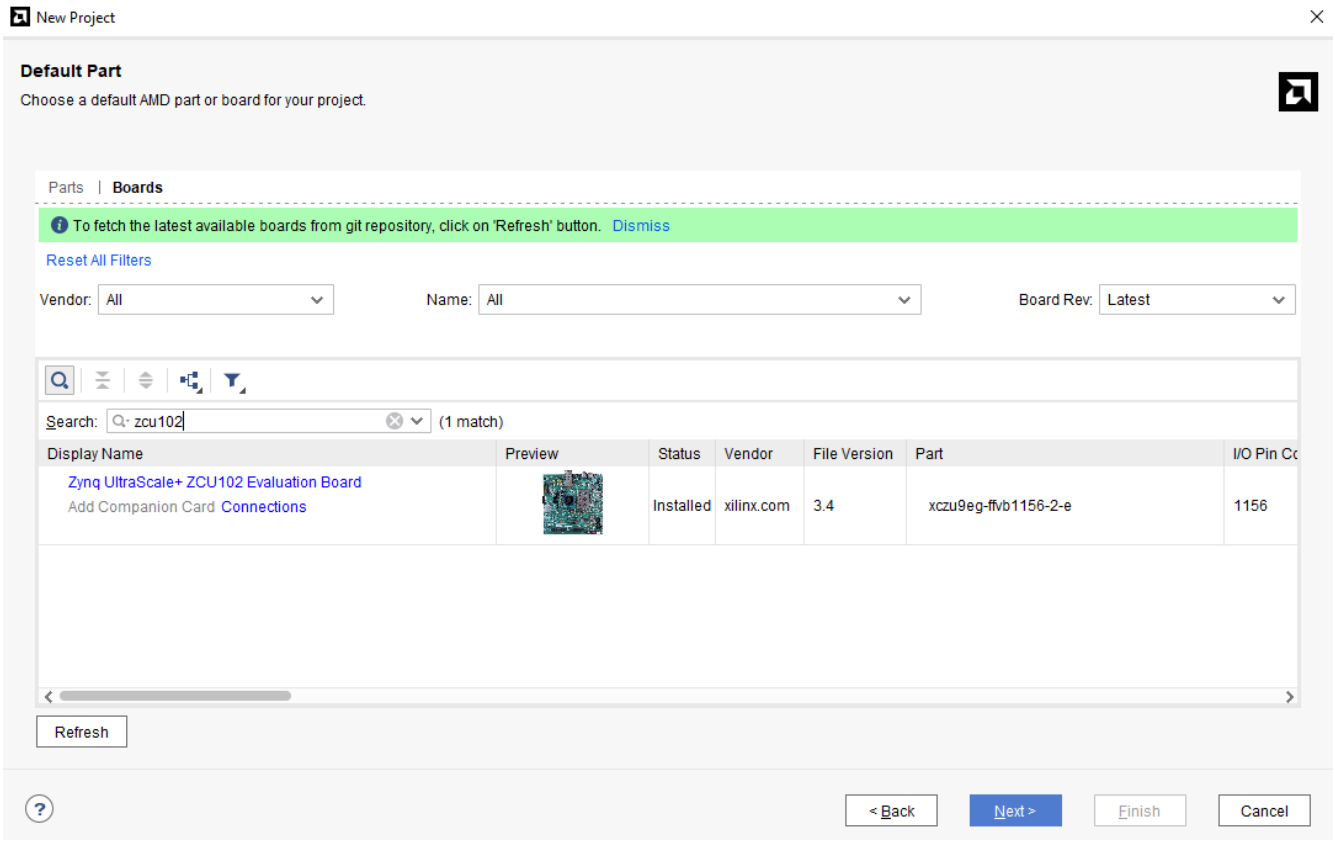

図 4-1. プロジェクトのためのボード選択

ステップ 2: 「ブロック デザインの作成」をクリックして、新しいブロックデザインを作成します

ステップ 3: 「IP を追加」をクリックするか、Ctrl + I を押します。「Zynq UltraScale+ MPSoC」IP ブロックを検索し、選択して追加します。

図 4-2. プロジェクトへの IP の追加

ステップ 4: 「IP を追加」をクリックするか、Ctrl + I を押します。「プロセッサ システム リセット」IP ブロックを検索し、選択して追加します

ステップ 5: ブロック オートメーションを実行します。このデザインは以下のようになります。

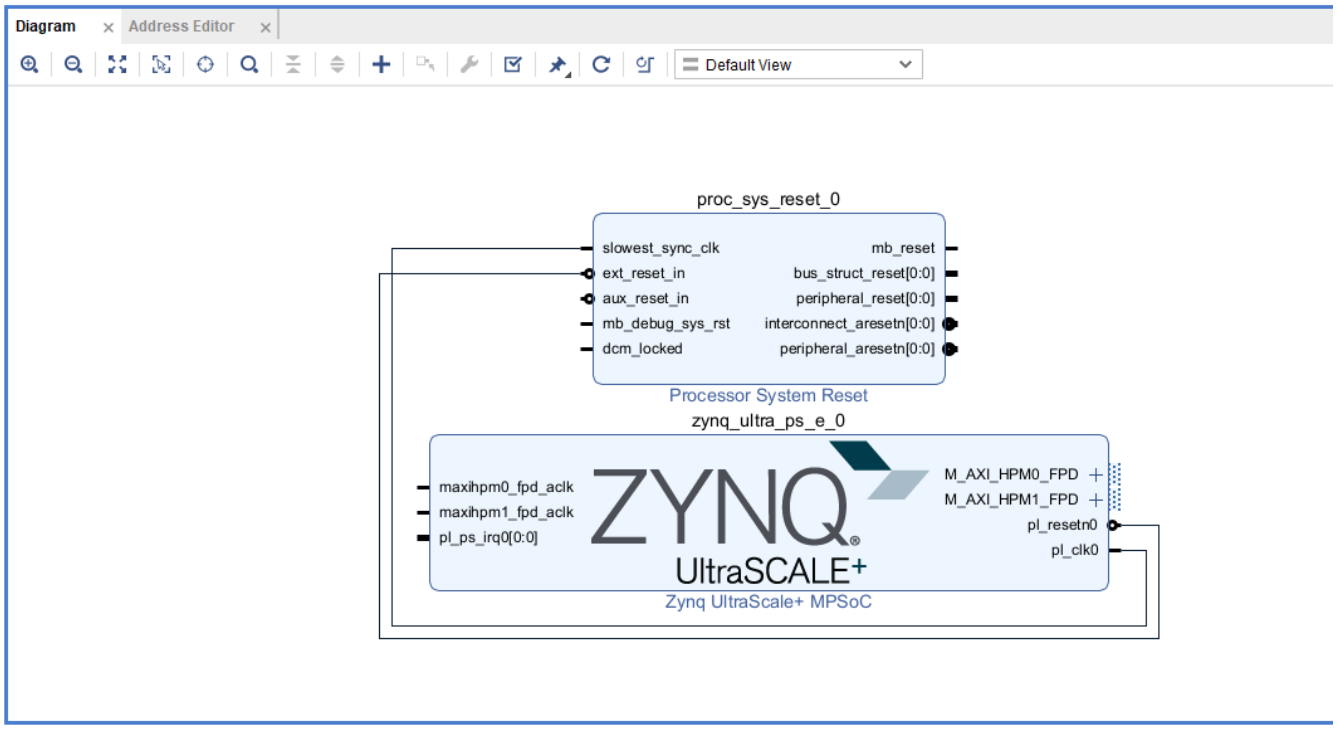

図 4-3. プロジェクトへのプロセッサの追加

4.2 SPI インターフェイスの構成

このセクションでは、Zynq プロセッシング システムに統合された SPI コントローラを有効化し、設定する方法について説明します。

次の手順に従います:

ステップ 1: 「Zynq UltraScale+ MPSoC」をダブルクリックし、「I/O 設定」>「低速」>「I/O ペリフェラル」に移動します。Zynq UltraScale+ MPSoC の SPI を有効化し、必要な数のチップ セレクトを有効にしてから、「OK」をクリックします。

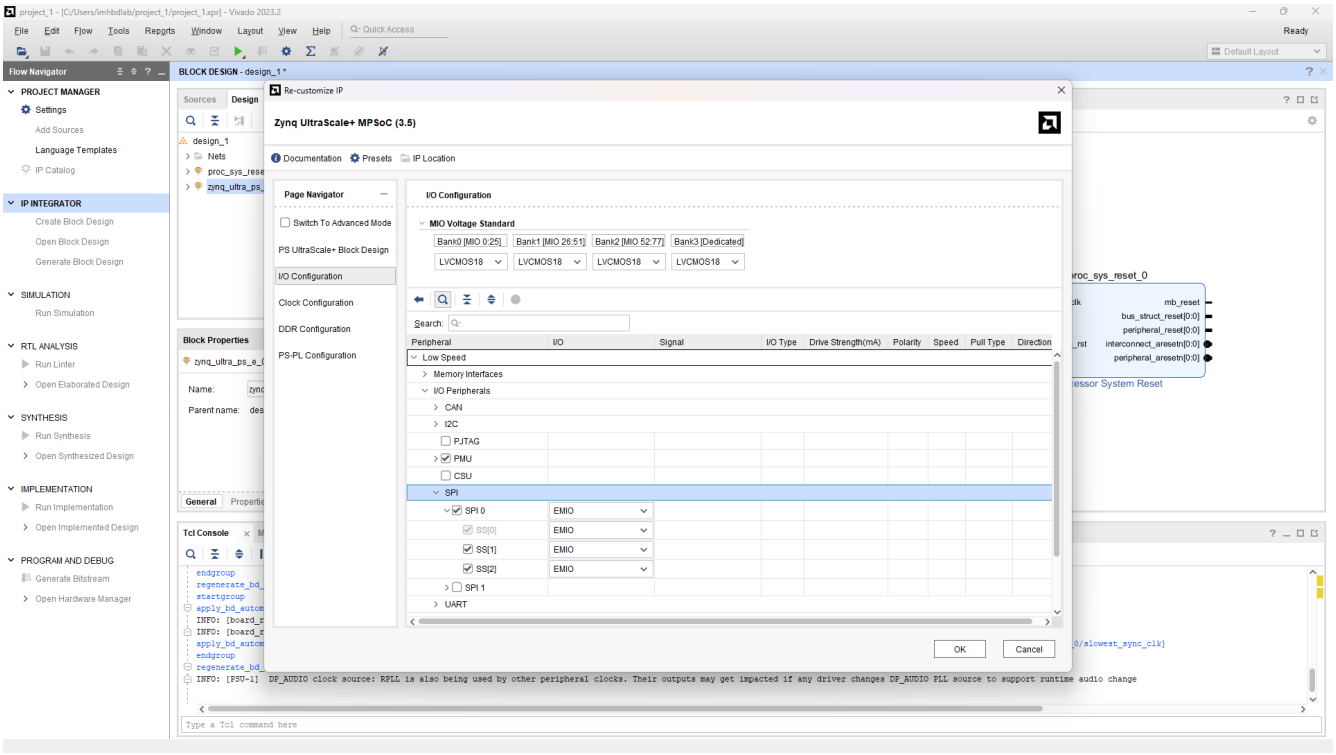

図 4-4. SPI ピンの構成

ステップ 2: SPI0 の前にある「+」記号をクリックします。emio_spi0_sclk_o、emio_spi0_m_o、emio_spi0_m_i、emio_spi0_ss_o_n を順に右クリックし、「Make External」を選択します。または、SPI ポートを選択して Ctrl + T を押します。

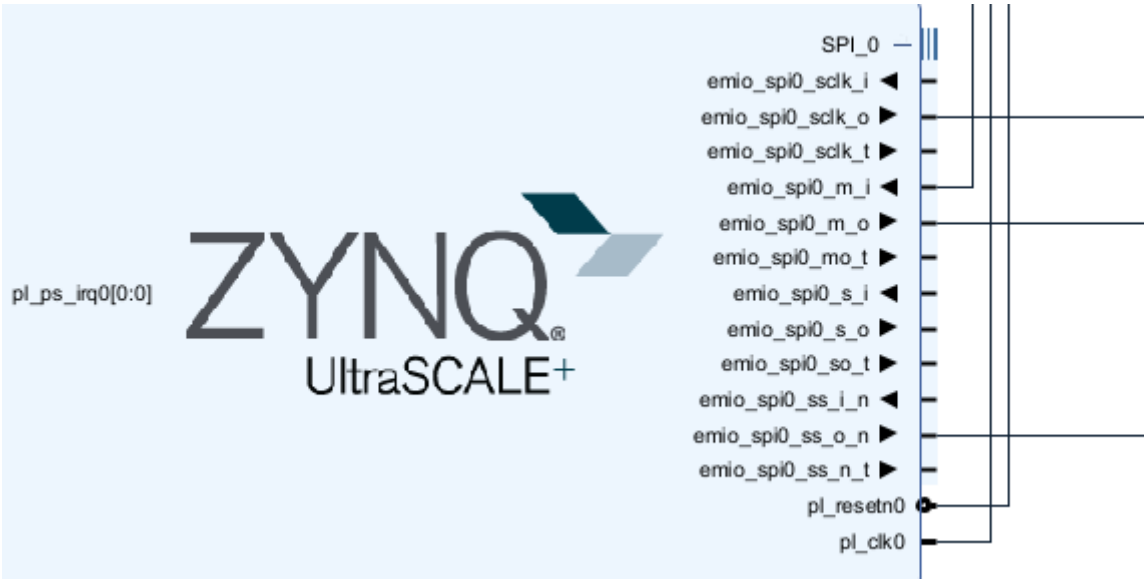

図 4-5. I/O ポートの選択

ステップ 3: Zynq コアをダブルクリックし、「PS/PL 構成」>「PS-PL インターフェイス」>「マスタ インターフェイス」に移動します。「AXI HPM0 FPD」および「AXI HPM1 FPD」を無効にし、「OK」をクリックします。

ステップ 4: 設計を保存して検証します。Ctrl + S を押して、デザインを保存します。F6 キーを押してデザインを検証します。最終的なデザインは図 4-6 と似ているはずです

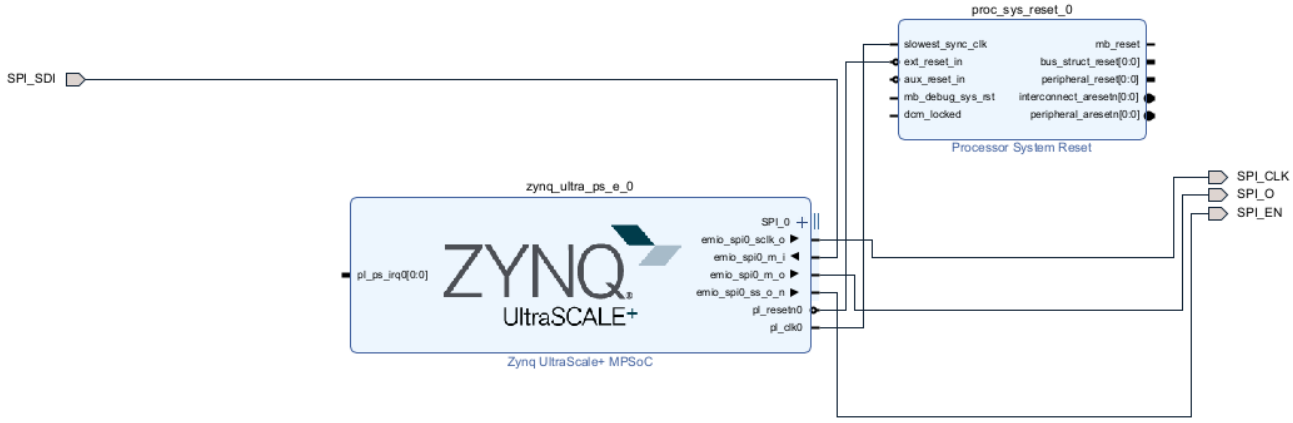

図 4-6. サンプル プロジェクト

ステップ 5: 「ソース」でブロック デザインを右クリックし、「HDL ラッパを作成」を選択して、デフォルト設定を適用します。

4.3 ハードウェア説明の生成

このセクションでは、完成した Vivado デザインからハードウェア記述ファイル (.xsa) および FPGA ビットストリーム (.bit) を生成してエクスポートする方法について説明します。

次の手順に従います:

ステップ 1: 合成を実行します。完了後に合成されたデザインを開きます。

ステップ 2: 「Window」->「IO ポート」に移動し、必要な IO ポート ピンを SPI ラインに割り当てます。

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			

図 4-7. SPI I/O 構成

AFE79xx 評価基板の I/O マッピングを以下に示します (ZCU102 の J4 HPC1 FMC コネクタ用)。

ステップ 3: 「ビットストリームを生成」をクリックします。

ステップ 4: 「ファイル」>「エクスポート」をクリックし、「ハードウェアのエクスポート」を選択します。これにより .xsa ファイルが生成されます。

ステップ 5: 「ファイル」>「エクスポート」をクリックし、「ビットストリーム ファイルのエクスポート」を選択します。これにより .bit ファイルが生成されます。

これでハードウェア設計フェーズは完了です。SPI インターフェイスを備えた Zynq リファレンス デザインが Vivado からエクスポートされ、以降の Linux カーネルのビルドおよびドライバ統合に必要なハードウェア記述ファイル (.xsa) とビットストリーム ファイル (.bit) が生成されました。

4.4 SD カードを使用した ZCU102 ボードの起動

このセクションでは、準備済みの SD カードから ZCU102 ボードを起動し、システムが正常に初期化されたことを確認する手順について説明します。起動シーケンスでは、SD カードから第一段階ブートローダー、U-Boot、Linux カーネル、デバイス ツリー、およびルート ファイル システムが順番にロードされます。ボードのブート モードを設定し、シリアル コンソールに接続して起動プロセスを監視した後、システムが正常に起動して Linux のコマンド プロンプトが表示されることを確認します。

以下の手順に従います:

ステップ 1: 評ブート モードの構成

ZCU102 のブート モード スイッチを SD カード ブートに設定します:

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

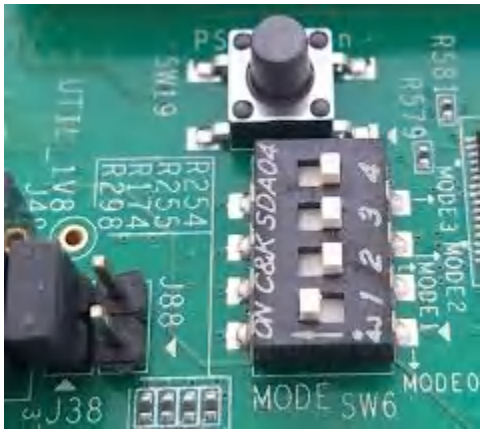

図 4-8. SD カード ブートのスイッチ状態

ステップ 2: 準備済みの SD カードを ZCU102 ボードの SD カード スロット (J100) に挿入します。

ステップ 3: シリアル コンソールを接続します

USB-UART ケーブルを接続します:

- ZCU102 の UART ポート (J83) と PC を micro-USB ケーブルで接続します
- PC が FTDI USB-UART 変換デバイスを認識します

ステップ 4: 端子ソフトウェアを開きます

シリアル端子アプリケーション (PuTTY、Tera Term、または Minicom) を起動します:

- **ポート:**ZCU102 に割り当てられた COM ポートを選択します (例:Windows では COM3、Linux では /dev/ttyUSB0)
- **ボーレート:**115200

ステップ 5: ボードの電源を入れます

ZCU102 電源スイッチ (SW1) をオンにします。

ステップ 6: 起動プロセスの監視

端末のブート メッセージを確認します。システムは、以下の段階を順に進みます:

- 第一段階ブート ローダ (FSBL)
- U-Boot
- Linux カーネル ロード
- ルート ファイルシステム マウント
- ログイン プロンプト

ステップ 7: ログイン

ログイン プロンプトで、次のように入力します:

- **ユーザー名:** petalinux
- **パスワード:** \ 新しいパスワードを設定します。

5 まとめ

このアプリケーション ノートでは、二つの異なる方式を使用して、Zynq UltraScale+ ZCU102 プラットフォーム上の Linux システムに AFE79xx SDK を統合するための一連の手順について説明しました。インツリー カーネル ドライバ方式では、AFE79xx ドライバのソース コードを Linux カーネル ツリーに追加し、Kconfig および Makefile を変更してカーネル ビルド システムを設定した後、ドライバ サポートを組み込んだ Linux システム全体をコンパイルする方法を示しました。このアプローチは、本番環境の導入に適した最適なパフォーマンスと緊密なシステム統合を提供します。ユーザー空間 SDK 方式では、汎用 spidev インターフェイスを介してハードウェアと通信するアプリケーション層ソフトウェアとして AFE79xx 機能を構築し、カーネルに依存しない柔軟な開発を実現する方法を示しました。両方式に共通する基盤手順として、SPI インターフェイスを設定した Vivado ハードウェア設計の作成、ハードウェア記述ファイルの生成、および PetaLinux プロジェクトのセットアップを行いました。本書の最後では、起動可能な SD カード メディアを準備し、ZCU102 ボードを起動してシステムが正常に初期化されたことを確認する手順について説明しました。これにより開発者は、Zynq プラットフォーム上で AFE79xx を統合するための実証済みの手法を利用できるようになりました。また、各プロジェクト固有の要件、性能上の制約、および保守性を考慮して、カーネル空間方式とユーザー空間方式のいずれを選択すべきかを明確に判断できます。構築されたシステムは、AFE79xx デバイスを完全に制御する機能を備えており、アプリケーションの開発および導入に使用できます。

6 参考資料

1. AMD、PetaLinux Tools ドキュメント: リファレンス ガイド、リファレンス ガイド。
2. Yocto プロジェクト、Yocto プロジェクトの資料、Web ページ。

重要なお知らせと免責事項

TI は、技術データと信頼性データ (データシートを含みます)、設計リソース (リファレンス デザインを含みます)、アプリケーションや設計に関する各種アドバイス、Web ツール、安全性情報、その他のリソースを、欠陥が存在する可能性のある「現状のまま」提供しており、商品性および特定目的に対する適合性の黙示保証、第三者の知的財産権の非侵害保証を含むいかなる保証も、明示的または黙示的にかかわらず拒否します。

これらのリソースは、TI 製品を使用する設計の経験を積んだ開発者への提供を意図したものです。(1) お客様のアプリケーションに適した TI 製品の選定、(2) お客様のアプリケーションの設計、検証、試験、(3) お客様のアプリケーションに該当する各種規格や、その他のあらゆる安全性、セキュリティ、規制、または他の要件への確実な適合に関する責任を、お客様のみが単独で負うものとします。

上記の各種リソースは、予告なく変更される可能性があります。これらのリソースは、リソースで説明されている TI 製品を使用するアプリケーションの開発の目的でのみ、TI はその使用をお客様に許諾します。これらのリソースに関して、他の目的で複製することや掲載することは禁止されています。TI や第三者の知的財産権のライセンスが付与されている訳ではありません。お客様は、これらのリソースを自身で使用した結果発生するあらゆる申し立て、損害、費用、損失、責任について、TI およびその代理人を完全に補償するものとし、TI は一切の責任を拒否します。

TI の製品は、 TI の販売条件 、 TI の総合的な品質ガイドライン 、 ti.com または TI 製品などに関連して提供される他の適用条件に従い提供されます。TI がこれらのリソースを提供することは、適用される TI の保証または他の保証の放棄の拡大や変更を意味するものではありません。 TI がカスタム、またはカスタマー仕様として明示的に指定していない限り、TI の製品は標準的なカタログに掲載される汎用機器です。

お客様がいかなる追加条項または代替条項を提案する場合も、TI はそれらに異議を唱え、拒否します。

最終更新日：2025 年 10 月