

# SDSoC 環境プラットフォーム開発ガイド

UG1146 (v2017.1) 2017 年 6 月 20 日

この資料は表記のバージョンの英語版を翻訳したもので、内容に相違が生じる場合には原文を優先します。資料によっては英語版の更新に対応していないものがあります。日本語版は参考用としてご使用の上、最新情報につきましては、必ず最新英語版をご参照ください。

# 改訂履歴

次の表に、この文書の改訂履歴を示します。

日付	バージョン	改訂内容
2017 年 6 月 20 日	2017.1	<ul style="list-style-type: none"><li>・ <a href="#">SDSoC プラットフォーム</a>を大幅に変更。</li><li>・ <a href="#">SDSoC プラットフォームの作成</a>の使用方法を追加。</li><li>・ <a href="#">ハードウェア プラットフォームの作成</a>に Vivado Design Suite プロジェクトの作成と Tcl スクリプトのサポートに関する情報を追加。</li><li>・ プラットフォーム メタデータ ファイルの詳細を削除。</li><li>・ <a href="#">PetaLinux</a> を使用した <a href="#">Linux ブート ファイルの作成</a>のプロセスをアップデート。</li><li>・ <a href="#">SDSoC プラットフォームのアップグレード</a>のプラットフォーム ファイルのアップデートに関する内容を変更。</li><li>・ チュートリアル: <a href="#">SDSoC プラットフォーム ユーティリティの使用</a>を追加。</li></ul>

# 目次

## 概要

### SDSoC プラットフォーム

SDSoC プラットフォームの作成 .....	9
メタデータ ファイル .....	14
プラットフォームのテストおよび使用 .....	14

### ハードウェア プラットフォームの作成

Vivado Design Suite プロジェクト .....	18
Vivado での SDSoC Tcl コマンド .....	20

### ソフトウェア プラットフォーム データの作成

ビルド済みハードウェア .....	29
ライブラリ ヘッダー ファイル .....	30
Linux ブート ファイル .....	31
PetaLinux を使用した Linux ブート ファイルの作成 .....	35
スタンドアロン ブート ファイル .....	39
FreeRTOS コンフィギュレーション/バージョン変更 .....	40

### プラットフォームのサンプル アプリケーション

### プラットフォームのチェックリスト

### SDSoC プラットフォームのアップグレード

### チュートリアル: SDSoC プラットフォーム ユーティリティの使用

演習 1: ZC702 プラットフォームの作成 .....	51
演習 2: ZC702_AXIS_IO プラットフォーム .....	55
演習 3: ZCU102 プラットフォーム .....	59

### SDSoC プラットフォームの例

例: SDSoC プラットフォームのダイレクト I/O .....	63
例: プラットフォーム IP AXI ポートの共有 .....	71

## その他のリソースおよび法的通知

参考資料 .....	74
お読みください: 重要な法的通知 .....	75

# 概要

SDx™ 環境は、Zynq®-7000 All Programmable SoC および Zynq UltraScale+™ MPSoC を使用してヘテロジニアス エンベデッド システムをインプリメントするための Eclipse ベースの統合設計環境 (IDE) です。SDx IDE では、Linux では SDSoC (Software-Defined System On Chip) および SDAccel デザイン フローの両方がサポートされ、Windows では SDSoC フローのみがサポートされます。SDSoC システム コンパイラ (sdsc または sds++) は、C または C++ で記述されたアプリケーション コードをハードウェア およびソフトウェアにコンパイルし、ターゲット プラットフォームを拡張するアプリケーション特定のシステム オン チップを生成します。SDx 環境には、アプリケーション開発用のプラットフォームが含まれます。その他のプラットフォームは、ザイリックス パートナーから提供されます。この資料では、Vivado® Design Suite を使用してビルドされたハードウェア システムと、OS カーネル、ブートローダー、ファイル システム、ライブラリなどのソフトウェア ランタイム環境から、カスタム SDSoC プラットフォームを作成する方法を説明します。

SDSoC プラットフォームは、ベース ハードウェアおよびソフトウェア アーキテクチャと、プロセッシング システム、外部メモリ インターフェイス、カスタム入力/出力、およびオペレーティング システム (ベアメタルの場合もあり)、ブートローダー、プラットフォーム ペリフェラルやルート ファイル システムなどのドライバー システム ランタイムを含むアプリケーション コンテキストを定義します。SDx 環境で作成するプロジェクトはすべて特定のハードウェア プラットフォームをターゲットとし、SDx 環境 IDE に含まれるツールを使用して、そのプラットフォームをアプリケーション特定のハードウェア アクセラレータおよびデータ モーション ネットワークでカスタマイズします。このようにすることで、さまざまなベース プラットフォーム用に高度にカスタマイズされたアプリケーション特定のシステム オン チップを簡単に作成でき、ベース プラットフォームを多数のアプリケーション特定のシステム オン チップに再利用できます。



**重要:** SDSoC 環境の使用に関する詳細は、『SDSoC 環境ユーザー ガイド』([UG1027](#)) を参照してください。

# SDSoC プラットフォーム

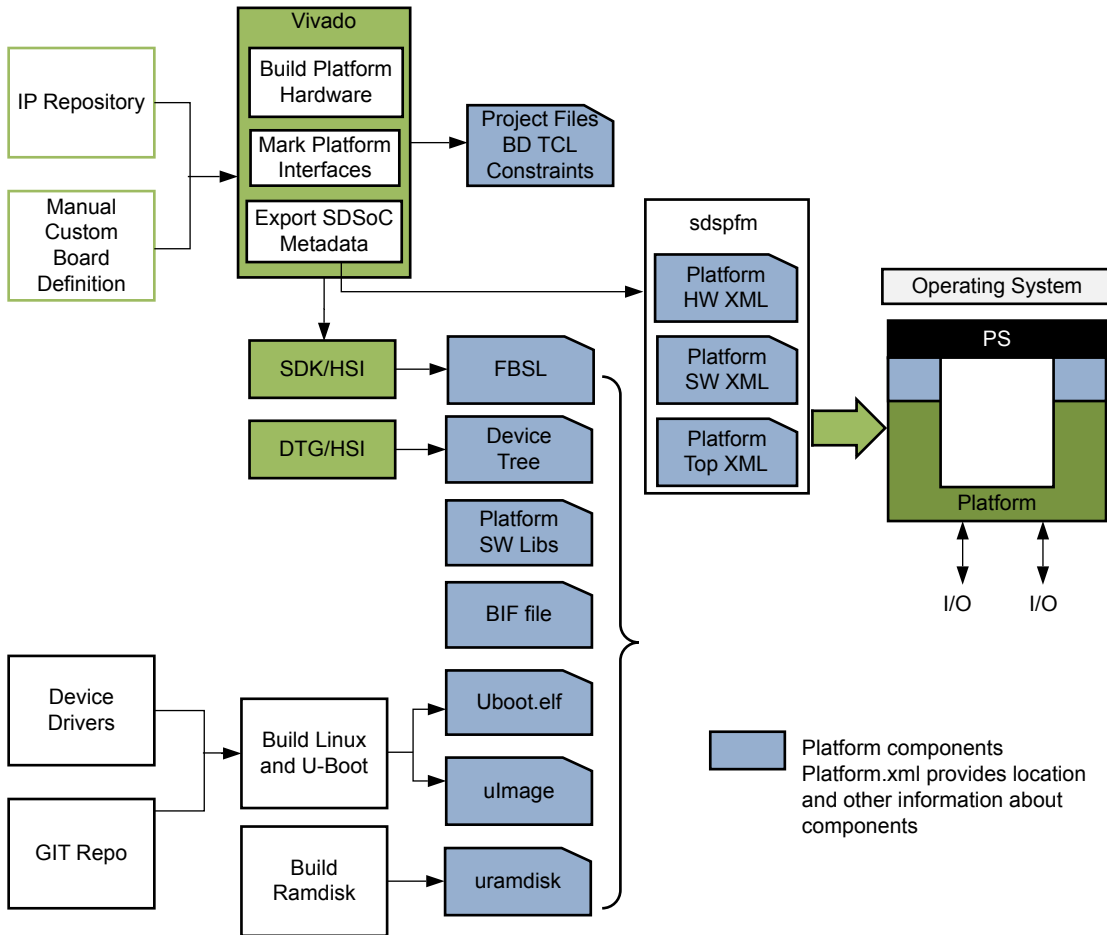
## 概要

SDSoC プラットフォームは、Vivado® Design Suite ハードウェア プロジェクト、ターゲット オペレーティング システム、ブート ファイル、およびオプションでプラットフォームをターゲットとするユーザー アプリケーションにリンクするソフトウェア ライブラリで構成されます。また、SDSoC コンパイラでプラットフォームをターゲットとするために使用されるハードウェアおよびソフトウェア インターフェイスを記述するサポートされる XML ファイルも含まれます。これらのファイルは、SDSoC プラットフォーム ユーティリティで生成されます。

プラットフォーム開発者は、Vivado Design Suite および IP インテグレーターを使用してプラットフォーム ハードウェアを設計します。ハードウェアをビルドして検証したら、Vivado ツール内で実行する Tcl スクリプトを使用して、SDSoC プラットフォーム ハードウェア インターフェイスを指定し、SDSoC プラットフォーム ハードウェア メタデータ ファイルを生成します。ハードウェア プラットフォームと必要な Tcl スクリプトの作成に関する詳細は、[ハードウェア プラットフォームの作成](#)を参照してください。

プラットフォーム開発者は、プラットフォームをブートするために必要なブートローダーおよびターゲット オペレーティング システムも供給する必要があります。プラットフォームには、SDSoC コンパイラを使用して、プラットフォームをターゲットとするアプリケーションにリンクするソフトウェア ライブラリもオプションで含めることができます。プラットフォームでターゲット Linux オペレーティング システムをサポートする場合は、コマンドラインまたは PetaLinux ツールスイートを使用してカーネルと U-Boot ブートローダーをビルドできます。プラットフォーム ライブラリをビルドするには、PetaLinux ツール、SDx IDE、またはザイリンクス SDK を使用できます。ソフトウェア プラットフォーム ライブラリの定義に関する詳細は、[ソフトウェア プラットフォーム データの作成](#)を参照してください。

図 1: SDSoC プラットフォームの主なコンポーネント



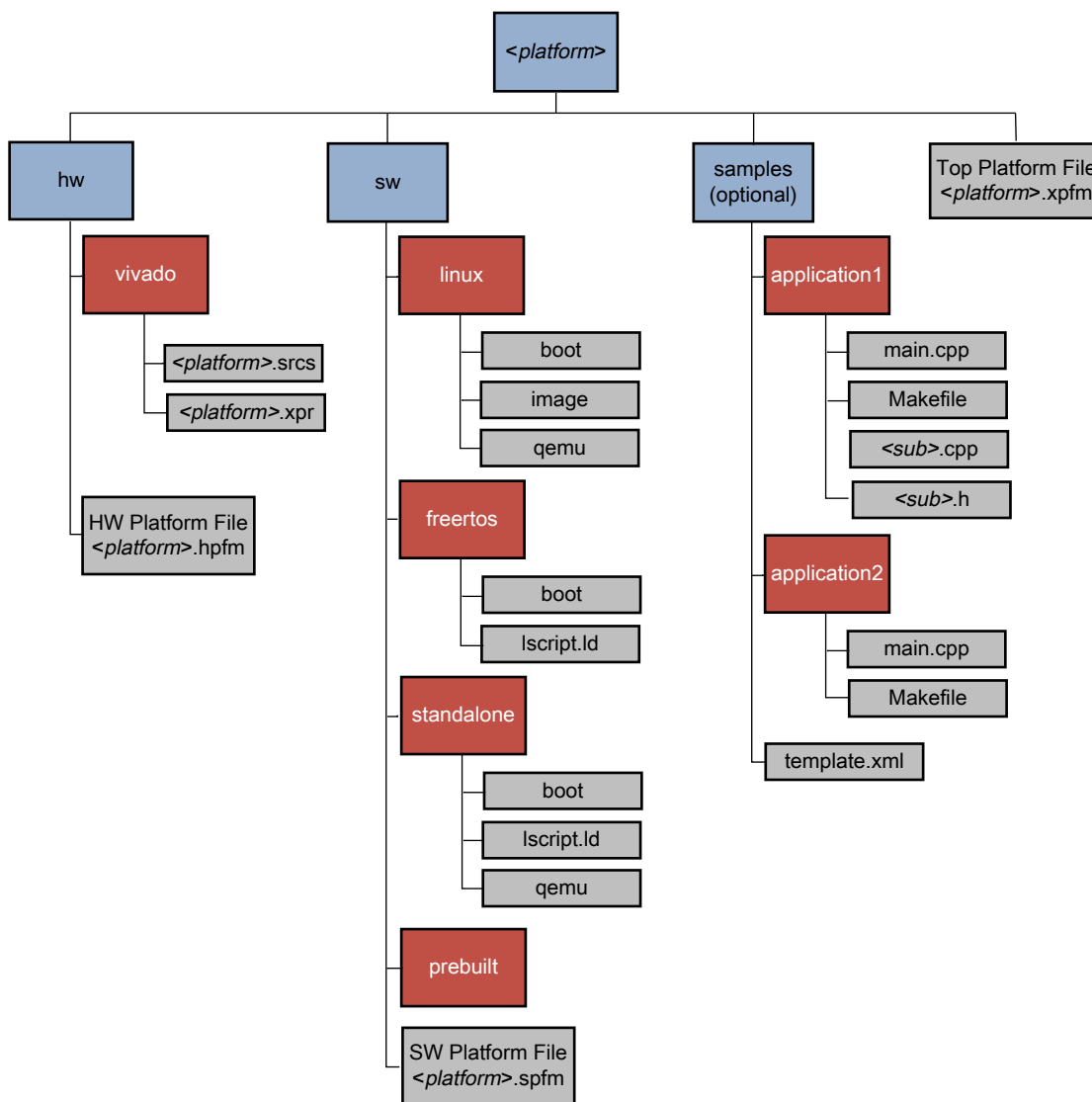
X14778-061517

SDSoC プラットフォームには、図 2 に示すように、次のエレメントが階層構造で含まれます。

- ・ SDSoC プラットフォーム ユーティリティで生成されたメタデータ ファイル:
- ・ Vivado Design Suite プロジェクト
  - デザイン ソース ファイル
  - タイミングおよび物理制約ファイル
  - IP ブロック
- ・ ソフトウェア ファイル
  - ライブラリ ヘッダー ファイル (オプション)
  - スタティック ライブラリ (オプション)
  - 共通ブート オブジェクト (FSBL (First Stage Boot Loader)、Zynq UltraScale+ MPSoC 用の ATF (ARM Trusted Firmware) および PMUFW (Power Management Unit FirmWare))
  - Linux 関連のオブジェクト (U-Boot および Linux デバイス ツリー、個別のファイルとしてまたは image.ub 統合ブート イメージとして供給されるカーネルおよび ramdisk)

- ・ ビルド済みハードウェア ファイル (オプション)
  - ビットストリーム
  - SDK 用にエクスポートされたハードウェア ファイル
  - 生成済みポート情報ソフトウェア ファイル
  - 生成済みハードウェアおよびソフトウェア インターフェイス ファイル
- ・ プラットフォーム サンプル アプリケーション (オプション)

図 2: 典型的な SDSoC プラットフォームのディレクトリ構造



通常は、プラットフォームが SDSoC 環境内で使用するのに正しいものであることを確実にできるのはプラットフォーム プロバイダーのみですが、SDx\_install\_path>/samples/platforms/Conformance フォルダには実行する必要がある基本的な生存性テストおよびサニティ テスト、およびそれらの実行方法が含まれています。これらのテストは正しくビルドされ、ハードウェア プラットフォーム上でテストされる必要があります。

プラットフォームには、すべてのカスタム インターフェイスに対するテストを含め、ユーザーがアプリケーション C/C++ コードからこれらのインターフェイスにアクセスする方法の例を取得できるようにする必要があります。



プラットフォームには、オプションでサンプル アプリケーションを含めることができます。アプリケーションのソース ファイルと `template.xml` メタデータ ファイルを含む `samples` というサブフォルダーを作成することにより、ユーザーが SDx 環境 IDE の New Project ウィザードを使用したときにこれらのサンプル アプリケーションのいずれかを選択してビルドできるようになります。アプリケーション テンプレートの作成については、[プラットフォームのサンプル アプリケーション](#)を参照してください。

SDSoC 環境には、SDSoC プラットフォームの生成を支援する SDSoC プラットフォーム ユーティリティ (`sdspfm`) が含まれます。必要な Vivado ハードウェア ファイルと Tcl スクリプトを生成した後このユーティリティを使用して、ソフトウェア プラットフォームのコンポーネントをビルドできます。

## SDSoC プラットフォームの作成

### 概要



**ヒント:** Vivado プロジェクトおよびブート ファイルを含むサンプル プラットフォーム ファイルは、SDSoC ソフトウェア インストール ディレクトリ (`<install>/samples/sdspfm`) に含まれています。SDSoC プラットフォームの作成のデモは、[チュートリアル: SDSoC プラットフォーム ユーティリティの使用](#)を参照してください。

SDSoC プラットフォーム ユーティリティを使用すると、SDSoC プラットフォームをシンプルに作成できます。このプラットフォーム ユーティリティには、プラットフォーム データを入力してプラットフォーム ファイルを見つける GUI のようなシンプルな形式のものと、オプションで定義したようにプラットフォームを生成するコマンドライン ユーティリティの 2 つがあります。

### SDSoC プラットフォーム ユーティリティ

SDSoC プラットフォーム ユーティリティを使用すると、SDSoC プラットフォームの作成がシンプルになります。このユーティリティは、GUI からは次のように起動します。

```
sdspfm -gui
```

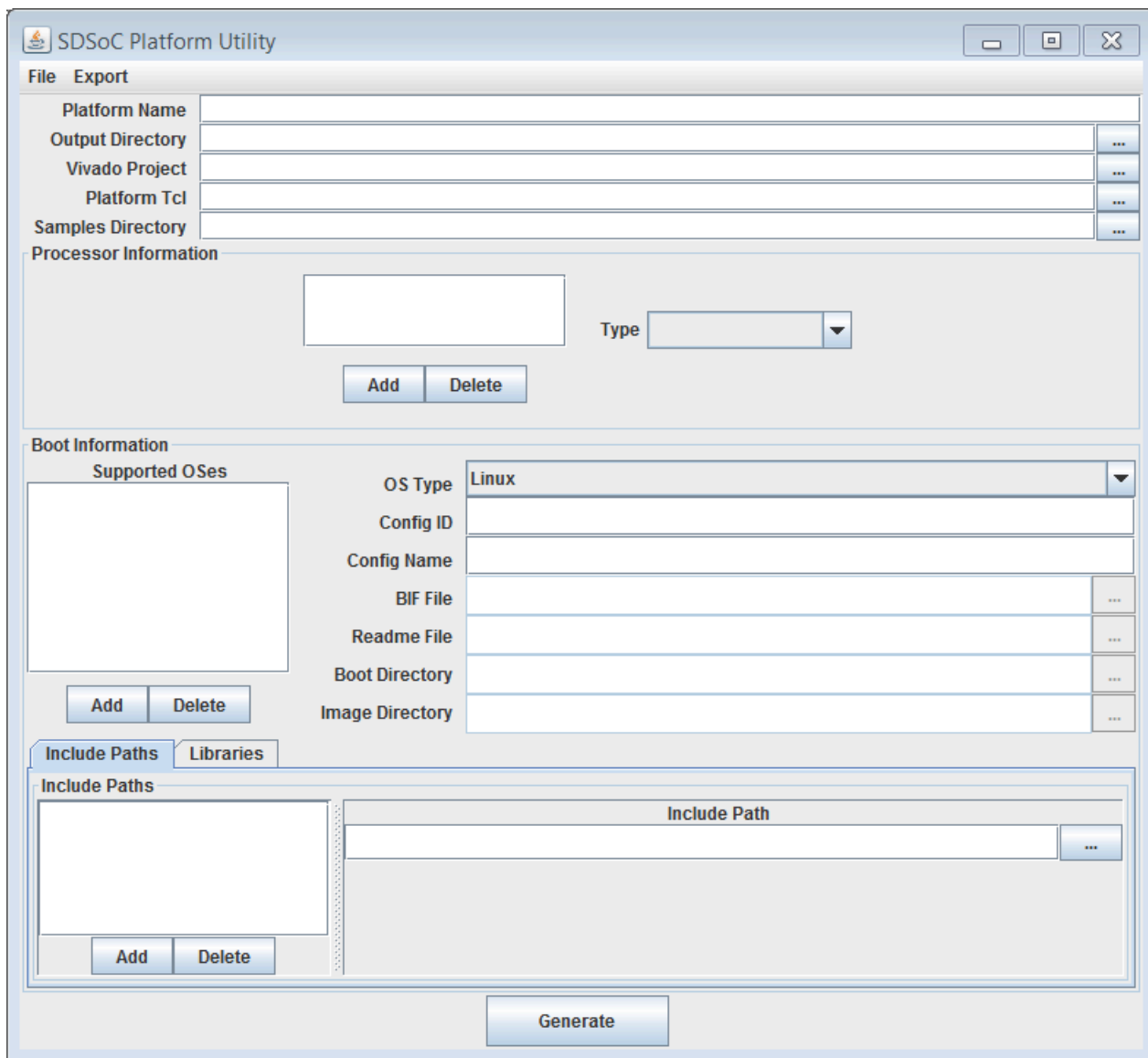


**重要:** SDSoC プラットフォーム ユーティリティは、Windows OS の C:/ または同等のルート ディレクトリから起動して、Windows でプラットフォーム ファイルを生成する際にパス名の長さの制限の問題が発生しないようにしてください。

GUI では、プラットフォーム設定を読み込み直すことが可能な XML ファイルとして保存して、プラットフォーム設定を変更したり、再利用したりできます。GUI のプラットフォーム ユーティリティには、次の図に示すように 3 つのプラットフォーム データ入力フィールドがあります。

- ・ [Base Info]: プラットフォーム名と Vivado ハードウェア プロジェクトに関連する基本情報。
- ・ [Processor Information]: プラットフォームで見つかったプロセッサのタイプを Vivado ハードウェア デザインで定義したように指定。
- ・ [Boot Information]: オペレーティング システム データとインクルード パスとリンクするライブラリのコンパイラ設定。

図 3: SDSoC プラットフォーム ユーティリティ



[Base Information] には、次が含まれます。

- ・ [Platform Name]: Vivado プロジェクトと IP インテグレーター ブロック図の名前と一致する必要あり (必須)
- ・ [Output Directory]: 新しく作成したプラットフォーム ディレクトリの記述される最上位プラットフォーム リポジトリを指定 (必須)
- ・ [Vivado Project]: [ハードウェア プラットフォームの作成](#)に示すように定義されたハードウェア プラットフォームを含む Vivado プロジェクト ファイル (.xpr) を指定 (必須)
- ・ [Platform Tcl]: [Vivado での SDSoC Tcl コマンド](#)で定義するように、プラットフォームのクロックとポートを含むスクリプト ファイル (必須)
- ・ [Samples Directory]: [プラットフォームのサンプル アプリケーション](#)で定義するように、プラットフォームとユーザーの作成した template.xml ファイルに関連するサンプル アプリケーションを含むディレクトリ (オプション)

[Processor Information] には、プラットフォームに使用可能なプロセッサ タイプを定義します。使用可能なプロセッサのタイプは、Vivado プロジェクトで選択したデバイスに基づいて設定されます。たとえば、XC7Z020 パーツのような Zynq-7000 デバイスを Vivado プロジェクトで選択した場合、GUI には ARM Cortex-A9 プロセッサ タイプしか表示されません。ただし、XCZU9 のような Zynq MPSoC デバイスを選択した場合、GUI には ARM Cortex-A53 と ARM Cortex-R5 プロセッサ タイプの両方が表示されます。これにより、A53 または R5 コアだけをターゲットにしたプラットフォームか、両方をターゲットにしたプラットフォームを作成できるようになります。



**ヒント:** プロセッサをコンフィギュレーションに追加するには、Vivado プロジェクト (\*.xpr ファイル) を選択する必要があります。選択しないと、プロセッサ タイプが空になります。

SDSoC では、プロジェクトをプロセッサのコア #0 をターゲットにするプロジェクトのみをビルドできます。つまり、a9\_0 にスタンドアロン プロジェクトを含めることはできますが、a9\_1 には含めることができません。また、a9\_0 および a9\_1 コアの両方に Linux を使用することはできますが、SDSoC で Linux が a9\_1 も使用していることを指定する必要はありません。この制限は、アプリケーション/OS の実行されているコアに関係なく、プロセッサですべてのコアを初期化するために FSBL をコア #0 で実行する必要があることが主な原因です。

[Boot Information] には、オペレーティング システムとコンパイラ設定が含まれます。SDSoC では、現在のところ Linux、FreeRTOS、およびスタンドアロン（ベアメタルなど）の 3 つのオペレーティング システムがサポートされています。各 OS ごとに、指定したアプリケーションに対してブート ファイルをコンパイル、リンク、生成するために含める必要のあるさまざまなファイルがあります。これらのファイルの詳細は、[ソフトウェア プラットフォーム データの作成](#)を参照してください。

- ・ [Config ID]: OS/ブート コンフィギュレーションの ID。SDSoC プラットフォーム ユーティリティを使用すると、プラットフォーム開発者は複数のビルド コンフィギュレーションを使用してプラットフォームを生成できます。各ビルド コンフィギュレーションには、OS、プロセッサ タイプ、およびブート ファイル/設定のセットが指定されます。[Config ID] では、ビルド コンフィギュレーションが認識されるので、ID はプラットフォームのビルド コンフィギュレーションごとに異なっている必要があります。[Config ID] には、英数字、アンダースコア ( ) またはダッシュ (-) を使用できますが、スペースや特殊文字は使用できません。この ID は、SDSoC makefile プロジェクトが作成される際に `-sds-sys-config<config_ID>` オプションで渡されます (すべての OS)
- ・ [Config Name]: OS/ブート コンフィギュレーションの名前。[Config Name] は、ユーザーが定義する名前、プラットフォームの各ビルド コンフィギュレーションごとに異なる名前を付ける必要があります。[Config Name] には、英数字、スペース、かっこ ( )、アンダースコア ( )、ダッシュ (-) のみを使用できます。SDSoC 開発環境 IDE で SDSoC プロジェクトを作成する際は、この名前がプロジェクト作成ウィザードに表示されます。
- ・ [BIF File]: Boot Information File (BIF) には、プロセッサに読み込む ELF ファイル (例: FSBL はコア 0 で実行)、ビットストリームの読み込み方法、プロセッサを設定するのに実行する必要があるその他のサブプログラムなどの情報が含まれます。BIF ファイルで参照したファイルはすべて [Boot Directory] フィールドで指定したディレクトリに含まれている必要があります。
- ・ [Readme File]: ユーザーにプラットフォームでこのブート コンフィギュレーションのアプリケーションを起動する方法を知らせるため必要とされるファイルです。ブート モードの設定をボードで切り替える方法など、ユーザーへの説明を含む単純なテキスト ファイルです。
- ・ [Image Directory]: SD カードにコピーする必要がある OS イメージ ファイルを含むディレクトリで、Linux の場合は `image.ub` ファイル、Linux ファイルのセットの場合は `image.gz`、`ramdisk`、`devicetree.dtb` が含まれている必要があります。これは、Linux でのみ必要な設定です。
- ・ [Linker Script]: セクションの ELF での統合方法、メモリ内での位置などをプログラマーが指定できるファイルで、ユーザーもスタックおよびヒープに対してどれくらい DDR メモリを割り当てるのかを指定できます。これは、FreeRTOS およびスタンドアロンでのみ必要な設定です。
- ・ [Boot Directory]: FSBL、U-Boot、ARM Trusted Firmware など、BIF ファイルで参照されるファイルをすべて含むディレクトリです。

[Compiler Settings]: プラットフォームのコンパイラ設定は、インクルード パスとリンクするライブラリを使用して OS コンフィギュレーションごとに指定できます。これらのディレクトリおよびライブラリは、プラットフォーム生成時に出力プラットフォームにコピーされます。ユーザーは複数のインクルード パスおよびライブラリを追加できます。

## プラットフォームの生成

SDSoC プラットフォームの設定が終了したら、SDSoC プラットフォーム ユーティリティの下にある [Generate] ボタンをクリックします。これで、ファイルのコピーおよび作成プロセスが開始され、プラットフォームのメタデータが生成されます。出力ディレクトリが既に存在する場合は、新しいプラットフォーム ファイルで自動的に上書きされます。プラットフォーム ファイルは必要に応じて生成し直すことができます。



**重要:** SDSoC プラットフォーム ユーティリティでプラットフォーム生成中にエラーが発生した場合は、原因は Vivado Design Suite プロジェクトおよび IP インテグレーター ブロック デザインに関係している可能性があります。SDSoC プラットフォーム ユーティリティで Vivado ツールが起動されて、ハードウェア プラットフォームが生成される場合に、IP がロックされていることを示すエラーが発生することがあります。これは、Vivado プロジェクトが [Vivado Design Suite プロジェクト](#) に示すように正しくコピーされていない場合や、[SDSoC プラットフォームのアップグレード](#) に示すように IP および出力ファイルがアップグレードされなかった場合に発生します。

## ユーティリティのメニュー

SDSoC プラットフォーム ユーティリティの GUI には、次の 2 つのメニューがあります。

- ・ [File]: 現在のコンフィギュレーションを保存したり、以前に保存したコンフィギュレーションを開いたり、またはユーティリティを終了したりするオプションがあります。
- ・ [Export]: プラットフォーム コンフィギュレーションを makefile としてエクスポートできます。GUI で入力したすべてのデータを使用して、プラットフォームを生成するコマンド ライン ユーティリティを呼び出すオプションを含めた最低限の makefile を含むファイルが出力されます。これは、上級ユーザーに役立つオプションです。エクスポートした makefile は SDSoC プラットフォーム ユーティリティに読み戻すことはできないので、makefile をエクスポートする際には、[File] → [Save] コマンドを使用してコンフィギュレーションを保存しておくことをお勧めします。

## ユーティリティのコマンド ライン

sdspfm -help コマンドを実行すると、次の情報が表示されます。

```
Usage: sdspfm -xpr <vivado_project.xpr> -pfmtcl <platform.tcl> [other
options]

Configuration Options:
[-sds-cfg <required options> -sds-end]
  Required options:
    -arch <process arch> : [cortex-a9, cortex-a53, cortex-r5, microblaze]
    -os <OS type> : [standalone, freertos, linux]
    -name <configuration name>
    -id <configuration ID>
    -bif <bif file path>
    -readme <readme file path>
    -boot <boot files directory path>
    -lscript <linker script path> : Only for Standalone/FreeRTOS OSes
    -image <image directory path> : Only for Linux OSes
    -inc <include path> : adds path to list of include paths (can use
multiple times)
    -lib <library file path> : adds path to list of library files (can
use multiple times)
    -libfreertos <FreeRTOS library file path> : use custom FreeRTOS
library, only for
FreeRTOS OSes
    -incfreertos <FreeRTOS include path> : use custom FreeRTOS includes,
only for
FreeRTOS OSes
    -qemu-args <QEMU arguments file>
```

```
Optional Options:
-o <output directory>
-samples <samples directory path containing template.xml>
-prebuilt <prebuilt directory path containing: portinfo.c/
h,apsys_0.xml,bitstream.bit,
<platform>.hdf,partitions.xml>
-force : overwrites output directory if it already exists
-cfg : load configuration saved from the GUI
-gui : launch the GUI
-verbose
-version
-help
```

## メタデータ ファイル

SDSoC プラットフォームには、SDSoC プラットフォーム ユーティリティで作成された、ハードウェアおよびソフトウェア インターフェイスを記述する次の XML メタデータ ファイルが含まれています。

- ・ 最上位プラットフォーム XML ファイル (.xpfm): 各 SDSoC プラットフォームには、プラットフォーム ハードウェア インターフェイスに関する情報を含むハードウェア プラットフォーム メタデータ ファイル (<platform>.hpfm) が含まれています。SDSoC は、デザインのハードウェア システムを作成する際にこの情報を使用し、データ モーション ネットワークおよびハードウェア アクセラレータと、クロック、データ、制御信号の必要な接続を追加します。SDSoC プラットフォーム ユーティリティでは、最上位プラットフォーム XML ファイルが <platform>/<platform>.xpfm として記述され、ハードウェアおよびソフトウェア XML ファイルとそれらを含むフォルダーへの参照も含まれます。
- ・ ハードウェア メタデータ ファイル (.hpfm): ハードウェア プラットフォーム XML メタデータ ファイルは、<platform>/hw/<platform>.hpfm として記述され、Vivado プラットフォーム プロジェクト (.xpr) と <platform>/hw/vivado のソースも同じディレクトリに含まれます。このファイルは、ベース ハードウェア、ハードウェア アクセラレータ、およびデータ モーション ネットワークを含むハードウェア システムを作成する際に SDSoC で使用されるプラットフォームのハードウェア インターフェイスを記述します。
- ・ ソフトウェア メタデータ ファイル (.spfm): ソフトウェア プラットフォーム XML ファイルは、<platform>/sw/<platform>.spfm として記述されます。.spfm ファイルは、ソフトウェア環境、つまりプラットフォームで使用可能なシステム コンフィギュレーションを記述します。各コンフィギュレーションにはオペレーティング システム (OS) が関連付けられており、ユーザーはハードウェア プラットフォームのデザインを作成するときにシステム コンフィギュレーションを選択します。

## プラットフォームのテストおよび使用

SDx 環境には、作成したプラットフォーム ファイルを読み込んでチェックするツールが含まれています。SDx ターミナル ウィンドウから、sdspfm の GUI で指定した [Output Directory] のディレクトリ内で次のコマンドを実行すると、SDSoC プラットフォーム ユーティリティで作成されたプラットフォーム ファイルが正しく読み込まれたかどうかを確認できます。

```
> sdscc -sds-pf-list
```



このコマンドは、現在の作業ディレクトリのプラットフォーム フォルダを読み込んでから、SDx インストール階層のプラットフォームを読み込んで、使用可能な SDx プラットフォームをリストします。カスタム プラットフォーム リポジトリからこのコマンドを指定した場合は、そこに入っているプラットフォームを読み込みます。

前のコマンドでリストされたプラットフォームは、次のコマンドを使用すると、さらに詳細に表示できます。

```
> sdscc -sds-pf-info <platform_name>
```

このコマンドは、指定したプラットフォームの詳細を表示します。

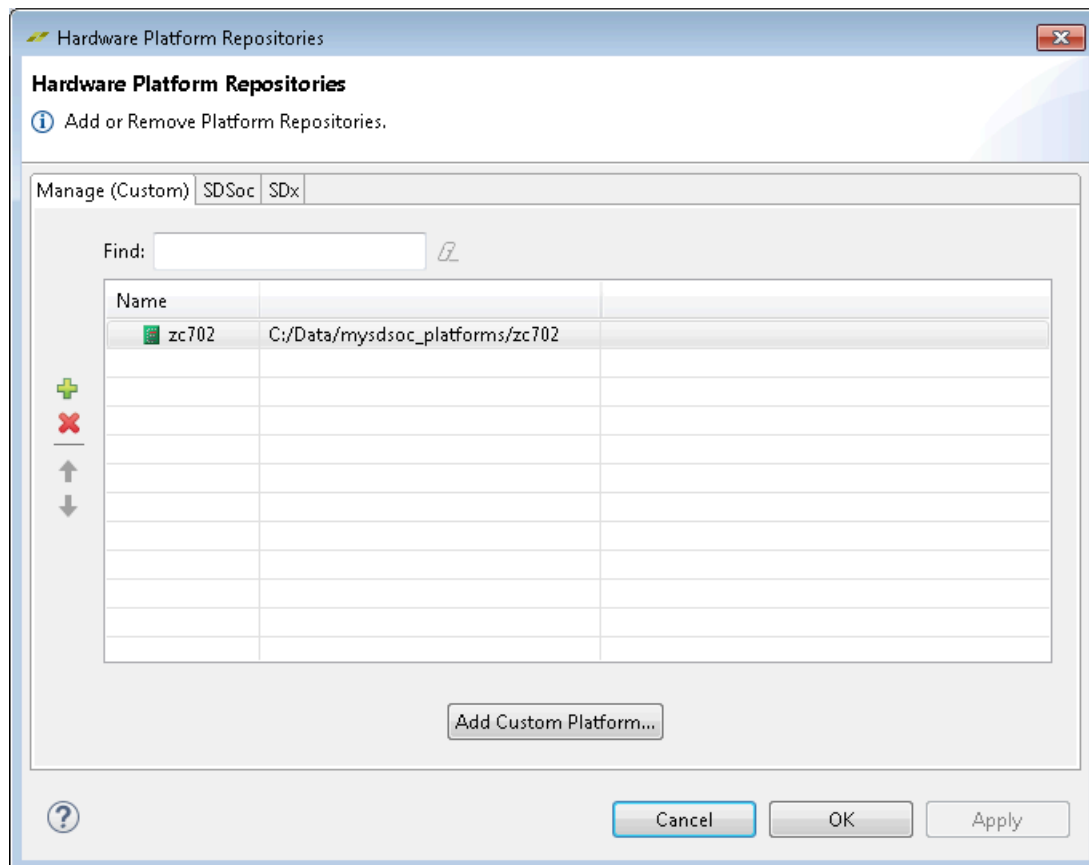
最後に、<platform>/hw フォルダ内で次のコマンドを使用すると、ハードウェア プラットフォーム ファイルを直接検証できます。

```
sds-pf-check <platform>.hpfm
```

これは、ハードウェア メタデータ ファイルが有効かどうかを単に確認するコマンドです。

プラットフォームの有効性を確認したら、ハードウェア プラットフォーム リポジトリを SDx に追加して新しいプロジェクトで使用できるようにします。SDx IDE で [Xilinx] → [Add Custom Platform] をクリックし、新規作成したプラットフォームを SDSoC プロジェクトで使用可能なプラットフォーム ライブラリに追加します。

図 4: ハードウェア プラットフォームの追加



次のコマンドを使用してプロジェクトに使用するプラットフォームを指定することもできます。

```
> sdscc -sds-pf <platform_name>
```

詳細は、『SDSoC 環境ユーザー ガイド』([UG1027](#)) を参照してください。



# ハードウェア プラットフォームの作成

## 概要

ハードウェア プラットフォームの作成プロセスでは、Vivado デザインをビルドし、クロック、割り込み、バス インターフェイスを含むサポートされるハードウェア インターフェイスを記述する SDSoC コマンドの Tcl スクリプトを作成します。SDSoC プラットフォーム ユーティリティでは、Vivado プロジェクトと関連するファイルがプラットフォーム フォルダー <path\_to\_platform>/hw/vivado に移動されます。

この章の説明は、Vivado Design Suite に関する知識があり、プラットフォームのハードウェア用の Vivado プロジェクトを作成できることを前提としています。プラットフォームのハードウェアに関する一般的な要件と、Vivado プロジェクトおよびハードウェア プラットフォーム フォルダーについて説明します。

## ハードウェア要件

このセクションでは、SDSoC プラットフォームのハードウェア デザイン コンポーネントの要件を説明します。通常、Vivado® Design Suite の IP インテグレーターを使用して作成した Zynq®-7000 All Programmable (AP) SoC または Zynq UltraScale+™ MPSoC デバイスをターゲットとするほぼすべてのデザインを、SDSoC プラットフォームのベースとして使用できます。SDSoC ハードウェア プラットフォームを記述するプロセスは、概念的には簡単です。

1. Vivado Design Suite および IP インテグレーター機能を使用してハードウェア システムをビルドおよび検証します。
2. [SDSoC Vivado Tcl コマンド](#)を使用して、Tcl スクリプトを作成します。
3. SDSoC プラットフォーム ユーティリティで Vivado プロジェクトと Tcl スクリプトを指定して、プラットフォームをビルドします。

プラットフォーム ハードウェア デザインでは、次の規則に従ってください。

- ・ Vivado Design Suite プロジェクト名は、ハードウェア プラットフォーム名と同じにする必要があります。



**ヒント:** Vivado プロジェクトに複数のブロック図が含まれる場合は、ハードウェア プラットフォームと同じ名前のブロック図が SDSoC Tcl スクリプトで使用されます。

- ・ プラットフォーム デザインで使用される IP で標準の Vivado IP カタログに含まれていないものは、Vivado Design Suite プロジェクトのローカルに配置されている必要があります。外部 IP リポジトリパスへの参照は、SDSoC Tcl スクリプトではサポートされません。
- ・ すべてのハードウェア プラットフォームに Vivado IP カタログからの Processing System IP ブロックを含める必要があります。

- ・ SDSoC プラットフォームへの各ハードウェア ポート インターフェイスは、AXI、AXI4-Stream、クロック、リセット、または割り込みインターフェイスにする必要があります。カスタム バス タイプまたはハードウェア インターフェイスは、ハードウェア プラットフォームに含める必要があります。
- ・ 各プラットフォームで、Processing System IP からの少なくとも 1 つの汎用 AXI マスター ポート、または AXI マスター ポートに接続されているインターコネクト IP を宣言する必要があります。これらは、SDSoC コンパイラでデータ ムーバーおよびアクセラレータ IP のソフトウェア制御に使用されます。
- ・ 各プラットフォームで、少なくとも 1 つの AXI スレーブ ポートを宣言する必要があります。このポートは、SDSoC コンパイラでデータ ムーバーおよびアクセラレータ IP から DDR にアクセスするために使用されます。
- ・ SDSoC 環境とプラットフォーム ロジック (s\_axi\_acp など) 間で AXI ポートを共有するには、対応する AXI ポートに接続されている AXI Interconnect IP の未使用の AXI マスターまたはスレーブをエクスポートし、プラットフォームで最下位インデックスのポートが使用されるようにする必要があります。
- ・ 各プラットフォーム AXI インターフェイスは、SDSoC 環境により 1 つのデータ モーション クロックに接続されます。



**ヒント:** SDSoC コンパイラで生成されたアクセラレータ関数は、プラットフォームで供給されるクロックとは異なるクロックで動作する場合があります。

- ・ エクスポートされた各プラットフォーム クロックに、Vivado IP カタログに含まれている Processing System Reset IP ブロックを付ける必要があります。
- ・ プラットフォームの割り込み入力、Processing System 7 IP IRQ\_F2P に接続されている Concat (xlconcat) IP でエクスポートされる必要があります。プラットフォームに含まれている IP ブロックでは 16 個のファブリック割り込みの一部を使用できますが、IRQ\_F2P ポートの最下位ビットからビットを飛ばさずに使用する必要があります。

## Vivado Design Suite プロジェクト

SDx™ IDE では、アプリケーション特定の SDSoC をビルドする際に、<platform>/vivado ディレクトリに含まれる Vivado® Design Suite プロジェクト ファイル (platform.xpr) が開始点として使用されます。プロジェクトには IP インテグレーター ブロック図が必ず含まれている必要があり、いくつでもソース ファイルを含めることができます。Zynq SoC または MPSoC をターゲットにするほとんどすべてのプロジェクトが SDSoC 環境プロジェクトの基盤となり得ますが、**ハードウェア プラットフォームの作成**に示すいくつかの制限があります。

Vivado Design Suite プロジェクトの名前はプラットフォームと同じにして、次のディレクトリに含めておく必要があります。

<platform>/hw/vivado/<platform>.xpr.

例: <path\_to\_install>/SDx/2017.x/platforms/zc702/hw/vivado/zc702.xpr.



#### 重要:

プロジェクト ファイルをプラットフォーム ディレクトリに移動する場合は、Vivado Design Suite プロジェクト全体をプロジェクト ファイル (.xpr) と同じディレクトリに含める必要があります。単に Vivado ツールのプロジェクトに含まれるファイルを別の場所にコピーすることはできません。Vivado Design Suite では、内部プロジェクト ステートとファイル同士の関係性が管理されているので、単にコピーするだけでは、これらが保持されません。Vivado Design Suite プロジェクトを正しくコピーするには、Vivado IDE で [File] → [Archive Project] をクリックして ZIP アーカイブを作成します。このアーカイブ ファイルを SDSoc <platform>/vivado ディレクトリにコピーして解凍します。

SDx IDE で Vivado ツールを起動する際に IP のロックを示すエラーが発生した場合は、Vivado プロジェクトが適切にコピーされなかったか、IP および出力ファイルがアップグレードされなかったことを意味します。

## Vivado プロジェクトの作成

次の手順で Vivado Design Suite プロジェクトを作成すると、SDSoC プラットフォームで使えるようになります。

1. <my\_platform>/hw/vivado のようなディレクトリ構造を作成します。
2. Vivado IDE を起動します。
3. Vivado Design Suite で [Create New Project] コマンドを使用して、そのディレクトリに <my\_platform> というプラットフォーム プロジェクトを作成します。

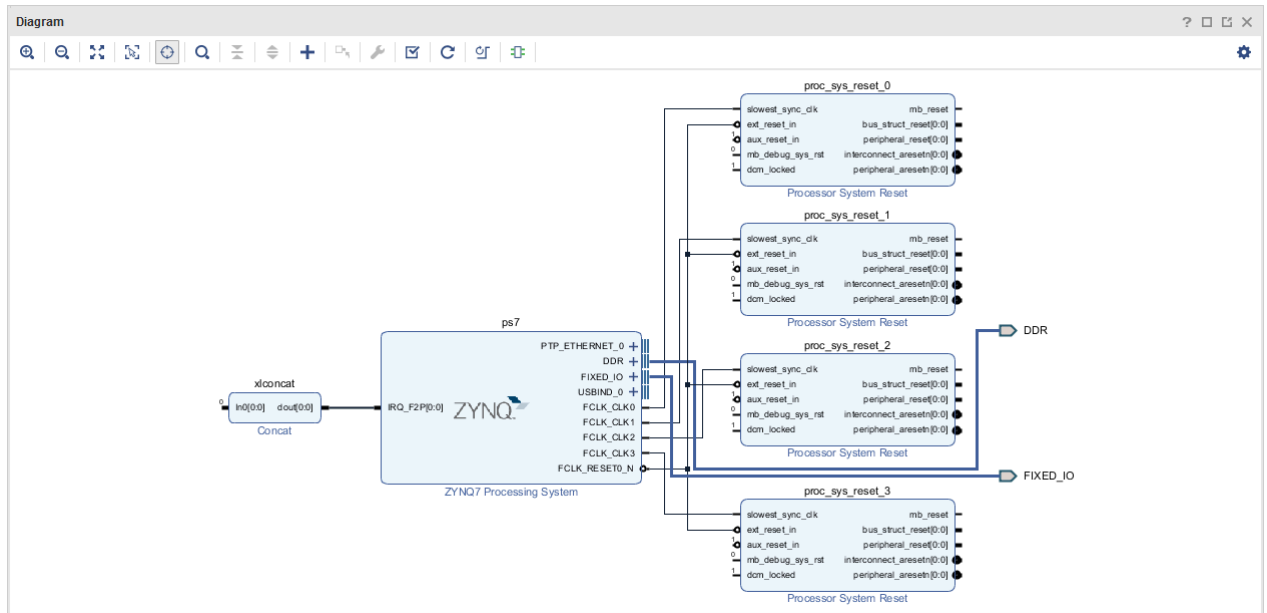


**ヒント:** 既存のプロジェクトを変更して、新しい SDSoc プラットフォームを作成する開始点として使用することもできます。

4. SDSoc プラットフォームで使用する ZC702 または Zybo ボードのようなザイリンクス デバイスまたはサポート デバイスを選択します。プロジェクトの作成およびパーツやボードの選択に関する詳細は、『Vivado Design Suite ユーザー ガイド: IP インテグレーターを使用した IP サブシステムの設計』(UG994) のこのセクションを参照してください。
5. Vivado IDE で最上位の Vivado プロジェクトが開いたら、[Create Block Design] コマンドを使用してブロック デザインを作成します。デザイン名はプラットフォーム (<my\_platform>) と同じ名前にします。

- 次の図のように、IP インテグレーターでブロック デザインを開いたまま、IP カタログからエンベデッド プロセッサ IP をインスタンスエートし、その他デザインを完成させるのに必要なザイリンクス IP やカスタム IP もインスタンスエートします。エンベデッド プロセッサ ブロック デザインの作成については、『Vivado Design Suite ユーザー ガイド: エンベデッド プロセッサ ハードウェア デザイン』(UG898) を参照してください。

図 5: IP インテグレーターでのブロック デザイン例



- [Validate] をクリックしてブロック デザインに問題がないかどうかを確認したら、[Save] をクリックしてデザインを保存します。
- [Generate Output Products] をクリックして、ブロック デザイン内の IP の出力ファイルを生成します。
- [Create Wrapper] をクリックして最上位 RTL デザインを作成します。
- [Export Hardware] をクリックして、プラットフォームのソフトウェア エLEMENTに必要なブート ロードーおよびターゲット オペレーティング システムを SDK にエクスポートします。ソフトウェア プラットフォーム ライブラリの定義に関する詳細は、[ソフトウェア プラットフォーム データの作成](#)を参照してください。
- 必要であれば、[Archive Project] でプロジェクトをアーカイブして、SDSoC プラットフォーム ディレクトリに移動します。

## Vivado での SDSoC Tcl コマンド

### 概要

Vivado Design Suite でハードウェア プラットフォーム デザイン プロジェクトを終了したら、SDSoC プラットフォーム ユーティリティで Vivado プロジェクトからのハードウェア プラットフォーム メタデータ ファイルを生成する Tcl スクリプトを作成する必要があります。Tcl スクリプトでは、次のセクションで説明するように SDSoC Vivado Tcl コマンドが使用されます。

## SDSoC Vivado Tcl コマンド

Vivado Design Suite には、クロック、リセット、割り込み、AXI、AXI4-Stream タイプのインターフェイスを含む、SDSoC プラットフォームのハードウェア インターフェイスを指定する SDSoC 用の Tcl コマンドが含まれます。スクリプトを作成するには、次の手順に従って Vivado Design Suite でこの SDSoC Tcl コマンドを使用する必要があります。

1. ハードウェア プラットフォームを宣言します (`sdsoc::create_pfm`)
2. ハードウェア プラットフォームの名前を定義します (`sdsoc::pfm_name`)
3. プラットフォームの簡単な説明を定義します (`sdsoc::pfm_description`)
4. プラットフォーム クロック ポートを宣言します (`sdsoc::pfm_clock`)
5. プラットフォーム AXI バス インターフェイスを宣言します (`sdsoc::pfm_axi_port`)
6. プラットフォーム AXI4-Stream バス インターフェイスを宣言します (`sdsoc::pfm_axis_port`)
7. 使用可能なプラットフォーム割り込みを宣言します (`sdsoc::pfm_irq`)
8. ハードウェア プラットフォーム記述メタデータ ファイルを記述します (`sdsoc::generate_hw_pfm`)

次にこれらのコマンドの詳細を示します。詳細は、[完全な例](#)も参照してください。

## ハードウェア プラットフォームの名前と説明の定義

次に、Vivado IP インテグレーターのブロック図内で使用される Tcl API について説明します。

- ・ 新しいハードウェア プラットフォーム ファイルを作成し、名前と説明を設定するには、次を使用します。

```
sdsoc::create_pfm <platform>.hpfm
```

引数:

```
<platform>      - platform name
```

戻り値:

```
new platform handle
```

- ・ プラットフォームの名前と説明を設定するには、次を使用します。

```
sdsoc::pfm_name      <platform handle> <vendor> <library> <platform>
<version>
sdsoc::pfm_description <platform handle> "<Description>"
```

例:

```
set pfm [sdsoc::create_pfm zc702.hpfm]
sdsoc::pfm_name      $pfm "xilinx.com" "xd" "zc702" "1.0"
sdsoc::pfm_description $pfm "Zynq ZC702 Board"
```

★ **重要:** Tcl スクリプトは、別のディレクトリではなく、Vivado プロジェクト ディレクトリ内にハードウェア プラットフォーム メタデータ ファイル (.hpfm 拡張子) を書き出す必要があります。この要件に従うには、Tcl スクリプトで次のコマンドを使用する必要があります。

```
set pfm [sdsoc::create_pfm <platform_name>.hpfm]
```

## クロックの宣言

クロック ソースはプラットフォームと一緒にエクスポートできますが、各クロックに対してプラットフォームの Processor System Reset IP ブロックを使用して、同期リセット信号もエクスポートする必要があります。クロックを定義するには、次を使用します。

```
sdsoc::pfm_clock <pfm_handle> <port> <instance> <id> <is_default>  
<proc_sys_reset>
```

引数:

引数	説明
pfm_handle	pfm ハンドル
port	クロック ポート名
instance	ポートを含むブロックのインスタンス名
id	クロック ID (ユーザー定義、固有の負でない整数)
is_default	デフォルトのクロックの場合は true、それ以外の場合は false
proc_sys_reset	同期リセット信号の対応する proc_sys_reset ブロック インスタンス

すべてのプラットフォームで、明示的にクロックが指定されていない場合は、SDSoC 環境で使用するデフォルト クロックを 1 つ宣言する必要があります。デフォルト クロックは、is\_default が true に設定されたクロックです。

例:

```
sdsoc::pfm_clock $pfm FCLK_CLK0 ps7 0 false proc_sys_reset_0  
sdsoc::pfm_clock $pfm FCLK_CLK1 ps7 1 false proc_sys_reset_1  
sdsoc::pfm_clock $pfm FCLK_CLK2 ps7 2 true proc_sys_reset_2  
sdsoc::pfm_clock $pfm FCLK_CLK3 ps7 3 false proc_sys_reset_3
```

## AXI ポートの宣言

AXI ポートを宣言するには、次を使用します。

```
sdsoc::pfm_axi_port    <pfm> <axi_port> <instance> <memport>
```

引数:

引数	説明
pfm	pfm ハンドル
port	AXI ポート名
instance	ポートを含むブロックのインスタンス名
memport	対応するメモリ インターフェイス ポート タイプ。値は次のとおりです。 <ul style="list-style-type: none"> <li>・ M_AXI_GP: 汎用 AXI マスター ポート</li> <li>・ S_AXI_HP: 高パフォーマンスの AXI スレーブ ポート</li> <li>・ S_AXI_ACP: アクセラレータ コヒーレンシ スレーブ ポート</li> <li>・ MIG: MIG メモリ コントローラーに接続されている AXI スレーブ</li> </ul>

例:

```
sdsoc::pfm_axi_port    $pfm M_AXI_GP0 ps7 M_AXI_GP
sdsoc::pfm_axi_port    $pfm M_AXI_GP1 ps7 M_AXI_GP
sdsoc::pfm_axi_port    $pfm S_AXI_ACP ps7 S_AXI_ACP
sdsoc::pfm_axi_port    $pfm S_AXI_HP0 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP1 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP2 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP3 ps7 S_AXI_HP
```

AXI インターコネクトの例:

```
sdsoc::pfm_axi_port    $pfm S01_AXI axi_interconnect_0 MIG
```

AXI インターコネクトのマスター ポートおよびスレーブ ポートのエクスポートには、いくつかの要件があります。

1. プラットフォームで使用されるインターコネクト上のすべてのポートは、宣言されたプラットフォーム インターフェイスの前にインデックス順で宣言する必要があります。
2. ポートのインデックス番号をスキップすることはできません。
3. S\_AXI\_ACP ポートのマスター ID の最大数は 8 なので、接続された AXI インターコネクト上で宣言する使用可能なポートは {S00\_AXI, S01\_AXI, ..., S07\_AXI} のいずれかである必要があります。プラットフォーム内で使用されるポートは宣言しないでください。できるだけ多くのポートを宣言すると、sds++ で生成されたユーザー システムでカスケード接続された axi\_interconnects を回避できます。
4. S\_AXI\_HP または MIG ポートのマスター ID の最大数は 16 なので、接続された AXI インターコネクト上で宣言する使用可能なポートは {S00\_AXI, S01\_AXI, ..., S15\_AXI} のいずれかである必要があります。プラットフォーム内で使用されるポートは宣言しないでください。できるだけ多くのポートを宣言すると、sds++ で生成されたユーザー システムでカスケード接続された axi\_interconnects を回避できます。



5. M\_AXI\_GP ポートに接続されるインターコネクト上で宣言されるマスター ポートの最大数は 64 なので、接続された AXI インターコネクト上で宣言する使用可能なポートは {M00\_AXI, M01\_AXI, ..., M63\_AXI} のいずれかである必要があります。プラットフォーム内で使用されるポートは宣言しないでください。できるだけ多くのポートを宣言すると、sds++ で生成されたユーザー システムでカスケード接続された axi\_interconnects を回避できます。

たとえば、zc702\_acp\_pfm.tcl ファイルには M\_AXI\_GP0 および S\_AXI\_ACP に接続されているインターコネクト ポート用に次の宣言が含まれます。

```
for {set i 1} {$i < 64} {incr i} {
    sdsoc::pfm_axi_port $pfm M[format %02d $i]_AXI axi_ic_gp0 M_AXI_GP
}
for {set i 1} {$i < 8} {incr i} {
    sdsoc::pfm_axi_port $pfm S[format %02d $i]_AXI axi_ic_acp S_AXI_ACP
}
```

## AXI4-Stream ポートの宣言

AXI4-Stream ポートを宣言するには、次を使用します。

```
sdsoc::pfm_axis_port <pfm> <axis_port> <instance> <type>
```

引数:

引数	説明
pfm	pfm ハンドル
port	AXI4-Stream のポート名
instance	ポートを含むブロックのインスタンス名
type	インターフェイス タイプ (値: M_AXIS、S_AXIS)

例:

```
sdsoc::pfm_axis_port $pfm S_AXIS axis2io S_AXIS
sdsoc::pfm_axis_port $pfm M_AXIS io2axis M_AXIS
```

## 割り込みポートの宣言

プラットフォーム プロセッシング システム 7 の IP ブロックには、IP インテグレーターの Concat ブロック (xlconcat) を介して割り込みを接続する必要があります。プラットフォームに含まれる IP に割り込みが含まれる場合、これらの割り込みで Concat ブロックの最下位ビットからビットを飛ばさずに使用する必要があります。



割り込みポートを宣言するには、次を使用します。

```
sdsoc::pfm_irq <pfm> <port> <instance>
```

引数:

引数	説明
pfm	pfm ハンドル
port	irq ポート名
instance	ポートを含む concat ブロックのインスタンス名

例:

```
for {set i 0} {$i < 16} {incr i} {
  sdsoc::pfm_irq $pfm In$i xlconcat
}
```

## I/O デバイスの宣言

Linux UIO フレームワークを使用する場合は、デバイスを宣言する必要があります。インスタンスを Linux I/O プラットフォーム デバイスとして宣言するには、次を使用します。

```
sdsoc::pfm_iodev <pfm> <port> <instance> <type>
```

引数:

引数	説明
pfm	pfm ハンドル
port	I/O ポート名
instance	UIO を含むブロックのインスタンス名
type	I/O デバイスタイプ (UIO、KIO など)

例:

```
sdsoc::pfm_iodev $pfm S_AXI axio_gpio_0 uio
```

## ハードウェア プラットフォーム記述ファイルの記述

上記の Tcl API コマンドを使用してプラットフォームを記述した後、次を使用してハードウェア プラットフォーム記述ファイルを記述します。

```
sdsoc::generate_hw_pfm <pfm_handle>
```

例:

```
sdsoc::generate_hw_pfm $pfm
```

このコマンドは、sdsoc::create\_pfm コマンドで指定されたファイルを記述します。

## 完全な例

SDSoC リリースに含まれるすべてのプラットフォームには、対応するハードウェア記述ファイルを生成する Tcl スクリプトが含まれます。この Tcl スクリプトは、hw/vivado ディレクトリに含まれる <platform>\_pfm.tcl というファイルです。

次に、ZC702 プラットフォームを生成するために Tcl API を使用する完全な例を示します。

```
# zc702_pfm.tcl --
#
# This file uses the SDSoC Tcl Platform API to create the
# zc702 hardware platform file
#
# Copyright (c) 2015 Xilinx, Inc.
#
# Uncomment and modify the line below to source the API script
# source -notrace <SDSOC_INSTALL>/scripts/vivado/sdsoc_pfm.tcl
set pfm [sdsoc::create_pfm zc702_hw.pfm]
sdsoc::pfm_name $pfm "xilinx.com" "xd" "zc702" "1.0"
sdsoc::pfm_description $pfm "Zynq ZC702 Board"
sdsoc::pfm_clock $pfm FCLK_CLK0 ps7 0 false proc_sys_reset_0
sdsoc::pfm_clock $pfm FCLK_CLK1 ps7 1 false proc_sys_reset_1
sdsoc::pfm_clock $pfm FCLK_CLK2 ps7 2 true proc_sys_reset_2
sdsoc::pfm_clock $pfm FCLK_CLK3 ps7 3 false proc_sys_reset_3
sdsoc::pfm_axi_port $pfm M_AXI_GP0 ps7 M_AXI_GP
sdsoc::pfm_axi_port $pfm M_AXI_GP1 ps7 M_AXI_GP
sdsoc::pfm_axi_port $pfm S_AXI_ACP ps7 S_AXI_ACP
sdsoc::pfm_axi_port $pfm S_AXI_HP0 ps7 S_AXI_HP
sdsoc::pfm_axi_port $pfm S_AXI_HP1 ps7 S_AXI_HP
sdsoc::pfm_axi_port $pfm S_AXI_HP2 ps7 S_AXI_HP
sdsoc::pfm_axi_port $pfm S_AXI_HP3 ps7 S_AXI_HP

for {set i 0} {$i < 16} {incr i} {
    sdsoc::pfm_irq $pfm In$i xlconcat
}

sdsoc::generate_hw_pfm $pfm
```

# ソフトウェア プラットフォーム データの作成

## 概要

ソフトウェア プラットフォーム データの作成プロセスでは、Zynq®-7000 All Programmable (AP) SoC または Zynq UltraScale+™ MPSoC デバイス上で実行されるサポートされた各 OS に対して、ライブラリおよびヘッダー ファイル、ブート ファイルなどのソフトウェア コンポーネントをビルドし、コンポーネントの使用法および場所を指定するソフトウェア プラットフォーム メタデータ XML ファイル (.spfm) を生成します。ソフトウェア コンポーネントはプラットフォーム フォルダー <path\_to\_platform>/sw に含まれ、ソフトウェア プラットフォーム メタデータ XML ファイル (.spfm) は <path\_to\_platform>/sw/<platform>.spfm にあります。

この章では、ソフトウェア プラットフォームに必須およびオプションのコンポーネントについて説明します。これらのコンポーネントは、プラットフォーム作成者が作成できると想定しています。たとえば、プラットフォームで Linux がサポートされる場合は、次を提供する必要があります。

- ・ ブート ファイル: FSBL (First Stage Boot Loader)、U-Boot、Linux ユニファイド ブート イメージ image.ub または個別の devicetree.dtb、カーネル、および ramdisk ファイル、BOOT.BIN ブート ファイルの作成に使用されるブート イメージ ファイル (BIF)。
- ・ 生成済みハードウェア ビットストリームや SDSoC データ ファイルなどのオプションのビルド済みファイル (SDSoC でハードウェア アクセラレータなしでアプリケーションをビルドする場合に時間を節約するために使用)。
- ・ オプションのヘッダー ファイルおよびライブラリ ファイル (プラットフォームでソフトウェア ライブラリが提供される場合)。
- ・ オプションのエミュレーション データ (プログラマブル ロジックおよびプロセッシング サブシステムの QEMU 用に Vivado シミュレータを使用したエミュレーション フローがプラットフォームでサポートされる場合)。

プラットフォームでザイリンクス スタンドアロン OS (ベアメタル ボード サポート パッケージ (BSP)) がサポートされる場合、ソフトウェア コンポーネントは Linux のものと似ていますが、ブート ファイルに FSBL および BIF ファイルが含まれます。



**ヒント:** Zynq UltraScale+ MPSoC ブート ファイルには、PMUFW (Power Management Unit FirmWare) および ATF (ARM Trusted Firmware) の ELF ファイルも必要です。

ターゲット OS 用のソフトウェア コンポーネントをビルドしたら、[SDSoC プラットフォームの作成](#)に従って、SDSoC プラットフォーム ユーティリティでこれらのコンポーネントをプラットフォームに追加します。

## ソフトウェア要件

このセクションでは、SDSoC プラットフォームのランタイム ソフトウェア コンポーネントの要件について説明します。

SDx IDE では、現在のところ Zynq®-7000 AP SoC ターゲットで実行される Linux、スタンドアロン (ベアメタル)、FreeRTOS オペレーティング システムがサポートされていますが、プラットフォームでそれらすべてをサポートする必要はありません。SDx IDE では、Zynq UltraScale+™ MPSoC 上で実行されている Linux およびスタンドアロン (ベアメタル) オペレーティング システムがサポートされます。

プラットフォーム ペリフェラルに Linux カーネルドライバが必要な場合は、drivers/staging/apf の linux-xlnx カーネル ソースを使用して、使用可能な SDx IDE 特定のドライバを複数含めるようにカーネルをコンフィギュレーションする必要があります。SDx 環境に含まれるベース プラットフォームには、platforms/zc702/sw/boot/generic.readme などの README ファイルに手順が含まれています。

この Linux カーネルと関連付けられているデバイス ツリーは、Linux カーネル 4.6 バージョンに基づいています。カーネルをビルドするには、次の手順に従います。

1. GitHub で Xilinx/linux-xlnx ツリーの master ブランチからクローン/プルし、xilinx-v2016.4-sdsoc タグをチェックアウトします。

```
git checkout -b sdsoc_release_tag xilinx-v2017.1-sdsoc
```

2. 次の CONFIG 属性を xilinx\_zynq\_defconfig に追加し、カーネルをコンフィギュレーションします。

```
CONFIG_STAGING=y
CONFIG_XILINX_APF=y
CONFIG_XILINX_DMA_APF=y
CONFIG_DMA_CMA=y
CONFIG_CMA_SIZE_MBYTES=256
CONFIG_CROSS_COMPILE="arm-linux-gnueabihf-"
CONFIG_LOCALVERSION="-xilinx-apf"
```

次はこれを実行するための 1 つの方法です。

```
cp arch/arm/configs/xilinx_zynq_defconfig arch/arm/configs/tmp_defconfig
```

3. テキスト エディターを使用して arch/arm/configs/tmp\_defconfig を編集し、上記の CONFIG 行をファイルの最後に追加します。

```
make ARCH=arm tmp_defconfig
```

4. 次を使用してカーネルをビルドします。

```
make ARCH=arm UIMAGE_LOADADDR=0x8000 uImage
```

デフォルトでは、SDSoC システム コンパイラでプラットフォームをブートするための SD カード イメージが生成されます。

スタンドアロン プラットフォームを作成するには、まず Vivado Design Suite の IP インテグレーターを使用してハードウェア コンポーネントをビルドし、ハードウェア エクスポート コマンド (write\_hwdef) を実行してハードウェア ハンドオフ ファイルを作成する必要があります。この新たに生成されたハードウェア ハンドオフ ファイルを使用し、SDx 環境 IDE でハードウェア プラットフォーム プロジェクトを作成します。このプロジェクトから、新しいボード サポート プロジェクトを作成できます。これで、system.mss ファイルおよびリンカー スクリプトをプラットフォームの一部として配布できます。このプロセスの詳細は、[SDSoC プラットフォームの例](#)を参照してください。

## ビルド済みハードウェア

プラットフォームには、オプションでビルド済みコンフィギュレーションを含めることができ、アプリケーションでハードウェア関数を指定しない場合にこれを使用できます。この場合、ビットストリームおよびその他の必要なファイルを作成するのに、プラットフォーム自体のハードウェア コンパイルを待つ必要はありません。

ビルド済みハードウェアは、プラットフォーム ソフトウェア ディレクトリのサブディレクトリに含める必要があります。サブディレクトリのデータは、該当するビルド済みハードウェアの <sdxc:configuration> の <sdxc:prebuilt> 要素で指定します。

プラットフォーム XML に <sdxc:prebuilt sdxc:data="prebuilt\_platform\_path"/> と指定されている場合、ディレクトリは次のようになります。

```
<path_to_platform>/sw/prebuilt_platform_path
```

パスは、ソフトウェア プラットフォーム フォルダーに相対するパスとして指定します。たとえば、次のように指定されているとします。

```
<sdxc:prebuilt sdxc:data="prebuilt_data"/>
```

この場合、ビルド済みのハードウェア ビットストリームおよび生成済みファイルは <path\_to\_platform>/sw/prebuilt\_data にあります。ZC702 プラットフォームの prebuilt\_data フォルダーには、bitstream.bit、zc702.hdf、partitions.xml、apsys\_0.xml、portinfo.c、および portinfo.h ファイルが含まれます。

ビルド済みハードウェア ファイルは、アプリケーションに通常のフラグを使用したハードウェア関数が含まれない場合、SDx 環境で自動的に使用されます。

```
-sds-pf zc702
```

Vivado ツールでビットストリームの生成および SD カード イメージの作成を強制的に実行するには、次の sdscc オプションを使用します。

```
-rebuild-hardware
```

platforms/<platform>/sw/prebuilt\_data フォルダーに含まれるファイルは、アプリケーション ELF とビットストリームを作成した後 \_sds フォルダーに含まれます。

- ・ bitstream.bit
  - \_sds/p0/ipi/<platform>.runs/impl\_1/bitstream.bit に含まれます。

- ・ <platform>.hdf
  - \_sds/p0/ipi/<platform>.sdk に含まれます。
- ・ partitions.xml、apsys\_0.xml
  - \_sds/.llvm に含まれます。
- ・ portinfo.c、portinfo.h
  - \_sds/swstubs に含まれます。

## ライブラリ ヘッダー ファイル

プラットフォームでアプリケーション コードにプラットフォーム特定のヘッダー ファイルを含める必要がある場合、これらのヘッダー ファイルをプラットフォーム ソフトウェア記述ファイルの該当する OS の `sdx:includePaths` 属性で指定されたプラットフォーム ディレクトリのサブディレクトリに含める必要があります。

プラットフォーム ソフトウェア記述ファイルに `sdx:includePaths="<relative_include_path>"` と指定されている場合、ディレクトリは次のようになります。

```
<platform root directory>/<relative_include_path>
```

例:

`sdx:includePaths="aarch32-linux/include"` の場合:

```
<sdx_root>/samples/platforms/zc702_axis_io/sw/aarch32-linux/include/  
zc702_axis_io.h
```

アプリケーション コードでヘッダー ファイルを使用するには、次の行を使用します。

```
#include "zc702_axis_io.h"
```

複数のインクルード パスは、コロン (:) で区切って指定します。

```
sdx:includePaths="<relative_include_path1>:<relative_include_path2>"
```

たとえば、プラットフォーム ソフトウェア記述ファイルでは、2 つのインクルード パスのリストが定義されます。

```
<platform_root_directory>/<relative_include_path1>  
<platform_root_directory>/<relative_include_path2>
```



**推奨:** ヘッダー ファイルが標準のディレクトリに含まれない場合は、SDSoC 環境のコンパイル コマンドで `-I` オプションを使用して指定する必要があります。ファイルは、プラットフォーム XML ファイルに記述されているように、標準ディレクトリに含めることをお勧めします。

## スタティック ライブラリ

プラットフォームで提供されるスタティック ライブラリにリンクすることを必須とする場合は、これらをプラットフォーム ソフトウェア記述ファイルの該当する OS の `sdx:libraryPaths` 属性で指定されたプラットフォーム ディレクトリのサブディレクトリに含める必要があります。

プラットフォーム ソフトウェア記述ファイルに `sdx:libraryPaths="<relative_lib_path>"` と指定されているとすると、ディレクトリは次のようになります。

```
<platform_root>/sw/<relative_lib_path>
```

例:

`sdx:libraryPaths="aarch32-linux/lib"` の場合:

```
<sdx_root>/samples/platforms/zc702_axis_io/sw/aarch32-linux/lib/  
libzc702_axis_io.a
```

ライブラリ ファイルを使用するには、次のリンカー オプションを使用します。

```
-lzc702_axis_io
```

複数のライブラリ パスは、コロン (:) で区切って指定します。次はその例です。

```
sdx:libraryPaths="<relative_lib_path1>:<relative_lib_path2>"
```

プラットフォーム ソフトウェア記述ファイルでは、2 つのライブラリ パスのリストが定義されます。

```
<platform_root>/sw/<relative_lib_path1>  
<platform_root>/sw/<relative_lib_path2>
```



**推奨:** スタティック ライブラリが標準ディレクトリには含まれない場合、`sdscc` リンク コマンドで `-L` オプションを使用してすべてのアプリケーションがそれらを指定するようにする必要があります。ファイルは、プラットフォーム ソフトウェア記述ファイルに記述されるように、標準ディレクトリに含めることをお勧めします。

## Linux ブート ファイル

SDx™ IDE では、ボード上の Linux オペレーティング システムを起動する SD カード イメージを作成できます。ブートが完了すると、Linux プロンプトからコンパイルされたアプリケーションを実行できるようになります。これには、プラットフォームの一部として次のオブジェクトが必要です。

- ・ [FSBL \(First Stage Boot Loader\)](#)
- ・ [U-Boot](#)
- ・ [デバイス ツリー](#) (オプション)
- ・ [Linux イメージ](#)
- ・ [ramdisk イメージ](#) (オプション)



SDx IDE では、次の 2 つの形式でパッケージ化された Linux イメージを使用できます。

1. カーネル イメージ、ramdisk イメージ、およびデバイス ツリーを個別のファイルとして含む個別のコンポーネント。
2. カーネル イメージ、ramdisk イメージ、およびデバイス ツリーを 1 つのファイルにまとめた image.ub という統合ブート イメージ。

Linux イメージは必須ですが、image.ub がどのようにパッケージ化されているかによって、ramdisk イメージおよびデバイス ツリーはオプションです。詳細は、『PetaLinux ツール資料: リファレンス ガイド』(UG1144) を参照してください。



**ヒント:** Zynq UltraScale+™ MPSoC デバイスをターゲットとするハードウェア プラットフォームでは、PMU ファームウェアと ATF (ARM Trusted Firmware) も必要です。これらのファイルは、後で BOOT.BIN に統合する必要があります。

SDx 環境では、ザイリンクスの bootgen ユーティリティ プログラムを使用して、FSBL、PMU-FW、ATF (ARM Trusted Firmware)、U-Boot ファイルとビットストリームを BOOT.BIN ファイルに統合します。このファイルは、必要なカーネル イメージ ファイル、オプションの ramdisk、デバイス ツリーと共に sd\_card フォルダーに含まれます。エンド ユーザーはこのフォルダーの中身を SD カードのルートにコピーして、プラットフォームをブートします。



**ヒント:** ブート ファイルのビルド方法の詳細は、<http://wiki.xilinx.com> にあるザイリンクス Wiki を参照してください。「Zynq AP SoC & Zynq UltraScale+ MPSoC (ZU+)」という見出しの下に、[U-Boot のビルド](#)、[Linux カーネルのビルド](#)などのトピックへのリンクがあります。

## FSBL (First Stage Boot Loader)

FSBL (First Stage Boot Loader) は、ブート時にビットストリームを読み込んで Zynq® アーキテクチャのプロセッシング システム (PS) をコンフィギュレーションします。

Vivado® Design Suite でプラットフォーム プロジェクトを開き、[File] → [Export] → [Export Hardware] をクリックします。

ザイリンクス SDK を使用する場合と同様に、[File] → [New] → [Application Project] をクリックし、fsbl という名前の新規ソフトウェア プロジェクトを作成します。

エクスポートされたハードウェア プラットフォームを使用して、リストから Zynq FSBL アプリケーションを選択します。これで、FSBL 実行ファイルが作成されます。

詳細は、[SDK ヘルプ](#)を参照してください。

FSBL を生成したら、SDx 環境フロー用の標準ディレクトリにコピーする必要があります。

例:

```
samples/platforms/zc702_axis_io/sw/boot/fsbl.elf
```

SDx システム コンパイラで FSBL が使用されるようにするには、<sd<:image> 要素の sd<:bif 属性で定義されている BIF ファイルで指定する必要があります。sd<:bif 属性の詳細は、[ソフトウェア プラットフォーム XML メタデータ リファレンス](#)を参照してください。ファイルは <path\_to\_platform>/sw/boot/ フォルダーに含める必要があります。





ヒント: Zynq AP SoC の BIF ファイルと Zynq UltraScale+ MPSoC デバイスの BIF ファイルは大きく異なります。

次に、Zynq®-7000 All Programmable (AP) SoC の boot.bif ファイルの例を示します。

```
/* linux */
the_ROM_image:
{
  [bootloader]<boot/fsbl.elf>
  <bitstream>
  <boot/u-boot.elf>
}
```

次に、Zynq UltraScale+™ MPSoC デバイスの boot.bif ファイルの例を示します。

```
/* linux */
the_ROM_image:
{
  [fsbl_config] a53_x64
  [bootloader]<boot/fsbl.elf>
  [pmufw_image]<boot/pmufw.elf>
  [destination_device=pl] <bitstream>
  [destination_cpu=a53-0, exception_level=el-3, trustzone] <boot/bl31.elf>
  [destination_cpu=a53-0, exception_level=el-2] <boot/u-boot.elf>
}
```

## U-Boot

U-Boot は、オープン ソースのブートローダーです。[wiki.xilinx.com](http://wiki.xilinx.com) の [U-Boot のビルド手順](#)に従って U-Boot をダウンロードし、プラットフォーム用にコンフィギュレーションします。PetaLinux を使用して Linux ブート ファイルを作成する場合は、U-Boot が自動的にコンフィギュレーションおよび作成されます。

SDx システム コンパイラで U-Boot が使用されるようにするには、<sdxc:image> 要素の `sdxc:bif` 属性で定義されている BIF ファイルで指定する必要があります。sdxc:bif 属性の詳細は、[ソフトウェア プラットフォーム XML メタデータ リファレンス](#)を参照してください。

```
/* linux */
the_ROM_image:
{
  [bootloader]<boot/fsbl.elf>
  <bitstream>
  <boot/u-boot.elf>
}
```

例: samples/platforms/zc702\_axis\_io/sw/boot/u-boot.elf

## デバイス ツリー

デバイス ツリーはハードウェアを記述するデータ構造であり、詳細をオペレーティング システムにハードコードする必要はありません。このデータ構造は、ブート時にオペレーティング システムに渡されます。ザイリンクス SDK を使用してプラットフォームのデバイス ツリーを生成します。デバイス ツリーの構築方法の詳細は、ザイリンクス Wiki ([wiki.xilinx.com](http://wiki.xilinx.com)) の [デバイス ツリー コンパイラの構築](#) を参照し、デバイス ツリー ジェネレーターのサポート ファイルをダウンロードしてインストールし、ザイリンクス SDK で使用してください。プラットフォームごとに 1 つのデバイス ツリーがあります。

ブートされた Linux システムで SDx:SDSoC を使用するには、デバイス ツリーに多少の変更を加える必要があります。最上位デバイス ツリー ファイルの末尾に次のテキストを手動で追加してください。

```
{
  xlnk {
    compatible = "xlnx,xlnk-1.0";
  };
};
```

★ **重要:** SDK ではこの変更は自動的に適用されません。最上位デバイス ツリー ファイルにこれを手動で追加する必要があります。

Linux ブート ファイルを PetaLinux を使用して作成している場合は、デバイス ツリーは統合ブート イメージ ファイル `image.ub` に含まれています。

`image.ub` ファイルまたは個別の Linux ブート ファイル (デバイス ツリー、カーネル、ramdisk) のどちらを使用している場合でも、`<sdx:image>` 要素の `sdx:imageData` 属性を定義してこれらのファイルを含むプラットフォーム フォルダを指定する必要があります。

```
sdx:imageData=image
```

場所: `samples/platforms/zc702_axis_io/sw/image`

## Linux イメージ

Linux イメージは、ハードウェア プラットフォームを起動するのに必要です。ザイリンクスでは、SDSoC プラットフォームすべてで動作する、プラットフォームに依存しないビルド済みの Linux イメージを 1 つ提供しています。

独自のプラットフォームに合わせて Linux をコンフィギュレーションする場合は、[wiki.xilinx.com](http://wiki.xilinx.com) の方法に従って、Linux カーネルをダウンロードおよびビルドしてください。独自のプラットフォーム用に Linux をコンフィギュレーションする場合は、SDx IDE の APF ドライバーおよび CMA (Contiguous Memory Allocator) をイネーブルにしてください。

Linux ブート ファイルを PetaLinux を使用して作成している場合は、Linux カーネルは統合ブート イメージ ファイル `image.ub` に含まれています。

`image.ub` ファイルまたは個別の Linux ブート ファイル (デバイス ツリー、カーネル、ramdisk) のどちらを使用している場合でも、`<sdx:image>` 要素の `sdx:imageData` 属性を定義してこれらのファイルを含むプラットフォーム フォルダを指定する必要があります。`sdx:imageData="image"` の場合、`image.ub` ファイルは `<path_to_platform>/sw/image/image.ub` にあります。

XML 記述の例:

```
sdx:imageData="image"
```

場所: samples/platforms/zc702\_axis\_io/sw/image

## ramdisk イメージ

ramdisk は起動に必要なイメージです。ramdisk イメージは SDx 環境 IDE インストールに含まれています。このイメージを変更したり新しい ramdisk を作成する必要がある場合は、[wiki.xilinx.com](http://wiki.xilinx.com) に記載されている手順に従います。

Linux ブート ファイルを PetaLinux を使用して作成している場合は、ramdisk は統合ブート イメージ ファイル image.ub に含まれています。

image.ub ファイルまたは個別の Linux ブート ファイル (デバイス ツリー、カーネル、ramdisk) のどちらかを使用している場合でも、<sdx:image> 要素の sdx:imageData 属性を定義してこれらのファイルを含むプラットフォーム フォルダを指定する必要があります。

XML 記述の例:

```
sdx:imageData="image"
```

場所: samples/platforms/zc702\_axis\_io/sw/image

## PetaLinux を使用した Linux ブート ファイルの作成

PetaLinux を使用すると、SDSoC プラットフォーム用の Linux ブート ファイルを生成できます。この方法については、『[PetaLinux ツール資料: ワークフロー チュートリアル](#)』(UG1156) を参照してください。SDSoC プラットフォームの全体的なワークフローは同じです。次に基本的な手順を示します。PetaLinux ツールの使用に慣れている場合は、Zynq-UltraScale+ MPSoC または Zynq-7000 All Programmable (AP) SoC デザインに対して次の手順を問題なく実行できるはずです。

開始する前に、次の手順を終了しておく必要があります。

- ・ PATH 環境変数で PetaLinux ツールを含むシェル環境を設定します。
- ・ 作業ディレクトリを作成し、cd コマンドを使用してそのディレクトリに移動します。
- ・ ターゲット ボードのタイプに対応する BSP をターゲットにした PetaLinux プロジェクトを作成します。

```
petalinux-create -t project <path_to_project> -s <path_to_base_BSP>
```

- ・ Vivado プロジェクトからハードウェア プラットフォームのハードウェア ハンドオフ ファイル (.hdf) のコピーを取得します。



**重要:** この資料では、プラットフォームに対して有効なハードウェア記述ファイル (HDF) が既に存在するものとして説明しています。このファイルは、Vivado Design Suite プロジェクトを使用して生成でき、<install>/SDx/2017.1/platforms/<platform name>/hw/vivado に含まれます。Vivado プロジェクトをインプリメントしてビットストリームを生成したら、そのビットストリームを含めて HDF ファイルをエクスポートします。

基本的な手順では、ハードウェア ハンドオフ ファイル、カーネル コンフィギュレーション、ルート ファイル システム コンフィギュレーションを読み込んで、Linux イメージ (.fsbl、.pmufw、.atf) をビルドします。この手順には、実行する作業と、実行する PetaLinux コマンドを引数と共に示します。ビルドが完了すると、デバイス ツリー、カーネル、および ramdisk を含むユニファイド ブート イメージ ファイル (image.ub) が作業ディレクトリに含まれます。基本設定の手順では、SDSoC プラットフォームに含まれるすべてのベース プラットフォームにパッケージされている Linux イメージをコンフィギュレーションします。

petalinux-config コマンドを使用する際は、階層状のメニュー システムを含むテキスト ベースのユーザー インターフェイスが表示されます。手順では、コマンドの階層と使用する設定を示します。同じインデントの選択肢は、同じ階層レベルにあることを示しています。たとえば、petalinux-config -c kernel では、最上位メニューから [Device Drivers] を選択し、[Generic Driver Options] を選択して、その下のレベルで設定を適用し、上のレベルの [Staging drivers] に戻って、設定をそのサブメニュー項目に適用します。

PetaLinux イメージをビルドするには、次の手順に従います。

1. PetaLinux の settings.sh を読み込みます。
2. 選択したボードに該当するベース BSP (たとえば、ZC702 の場合は xilinx-zc702-v2017.1-final.bsp) の PetaLinux プロジェクトを作成します。

```
petalinux-create -t project -n <platform name> -s <petalinux_install>/<base
BSP name>
```

3. 前に作っておいた関連するプラットフォームの HDF (作成方法は、概要を参照) を使用して PetaLinux を設定します。

```
petalinux-config -p <platform name> --get-hw-description=<HDF path>
```

デフォルトのコマンドの最後に quiet を含めるようにブート引数 (bootargs) を変更します。

- ・ [Kernel Bootargs] → [Generate boot args automatically] (オフ)
- ・ Zynq MPSoC の場合: [Kernel Bootargs] → ユーザーがカーネル bootargs を設定 (earlycon clk\_ignore\_unused quiet)
- ・ Zynq-7000 の場合: [Kernel Bootargs] → ユーザーがカーネル bootargs を設定 (console=ttyPS0,115200 earlyprintk quiet)

#### 4. PetaLinux カーネルを設定します。

```
petalinux-config -p <platform name> -c kernel
```

CMA のサイズをより大きく設定します。SDS-alloc バッファの場合は次のように設定します。

- ・ Zynq MPSoC の場合: [Device Drivers] → [Generic Driver Options] → [Size in Mega Bytes(1024)]
- ・ Zynq-7000 の場合: [Device Drivers] → [Generic Driver Options] → [Size in Mega Bytes(256)]

ステージングドライバーをイネーブルにします。

- ・ [Device Drivers] → [Staging drivers] (オン)

APF マネージメントドライバーをイネーブルにします。

- ・ [Device Drivers] → [Staging drivers] → [Xilinx APF Accelerator driver] (オン)

APF DMA ドライバーをイネーブルにします。

- ・ [Device Drivers] → [Staging drivers] → [Xilinx APF Accelerator driver] → [Xilinx APF DMA engines support] (オン)

**注記:** Zynq MPSoC の場合は、CPU アイドルと周波数スケーリングをオフにする必要があります。これには、次のようにオプションを指定します。

- ・ [CPU Power Management] → [CPU idle] → [CPU idle PM support] (オフ)
- ・ [CPU Power Management] → [CPU Frequency scaling] → [CPU Frequency scaling] (オフ)

#### 5. PetaLinux rootfs を設定します。

```
petalinux-config -p <platform name> -c rootfs
```

stdc++ libs を追加します。

- ・ [Filesystem Packages] → [misc] → [gcc-runtime] → [libstdc++] (オン)

#### 6. APF ドライバーのデバイス ツリー部分を追加します。<platform name>/project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi の下に次を追加します。

```
{
    xlnk {
        compatible = "xlnx,xlnk-1.0";
    };
};
```

#### 7. PetaLinux イメージをビルドします。

- ・ petalinux-build

8. <petalinux project>/images/linux/ ディレクトリには、重要なファイルが多くあり、2 つのカテゴリに分類されています。
  - a. BOOT.BIN にコンパイルされるファイルで、まとめてブート ファイルと呼ばれ、boot フォルダにコピーされるファイル。ブート ファイルには、u-boot.elf、zynq-fsbl.elf、または zynqmp-fsbl.elf が含まれるほか、Zynq UltraScale+ デバイスの場合は bl31.elf および pmufw.elf も含まれます。
  - b. SD カードに存在している必要のあるファイルですが、BOOT.BIN にはコンパイルされず、まとめてイメージ ファイルとよばれ、image フォルダにコピーされるファイル。PetaLinux ビルドからのイメージ ファイルは image.ub だけですが、その他のファイルも image フォルダに追加しておくと、プラットフォームのユーザーが使用できるようになります。

<petalinux project>/images/linux/ フォルダ内から次のコマンドを実行します。

```
$ mkdir ./boot
$ mkdir ./image
$ cp u-boot.elf ./boot/u-boot.elf
$ cp *fsbl.elf ./boot/fsbl.elf
$ cp bl31.elf ./boot/bl31.elf
$ cp linux/pmufw.elf ./boot/pmufw.elf
$ cp image.ub ./image/image.ub
```

9. 最後にブート イメージ ファイル (BIF) を作成します。このファイルは boot フォルダの内容を BOOT.BIN ファイルにコンパイルするのに使用されます。

SDSoC ブート イメージ ファイルは、標準的な BIF ファイルと類似していますが、指定したパスではなく、記号定数が使用される点が異なります。SDSoC では、ブート ファイルへの直接パスではなく、生成された内容に置き換えられるパターンを使用して記述した .bif ファイルが必ず必要となります。これは、ビットストリーム ファイルが手続き型で生成されるためと、一部の要素に BIF ファイルの生成時にわかっているファイル名がないためです。

次に、Zynq-UltraScale+ MPSoC デバイスの BIF の例を示します。

```
the_ROM_image:
{
  [fsbl_config] a53_x64
  [bootloader]<boot/fsbl.elf>
  [pmufw_image]<boot/pmufw.elf>
  [destination_device=pl] <bitstream>
  [destination_cpu=a53-0, exception_level=el-3, trustzone] <boot/bl31.elf>
  [destination_cpu=a53-0, exception_level=el-2] <boot/u-boot.elf>
}
```

boot ディレクトリ、image ディレクトリ、.bif ファイルなどがまとめられ、SDSoC プラットフォーム ユーティリティへの Linux OS の入力として必要とされます。

図 6: Linux ブート情報

OS Type	Linux	▼
Config ID	a53_linux	
Config Name	A53 Linux	
BIF File	boot.bif	...
Readme File		...
Boot Directory	./boot/	...
Image Directory	./image	...

## スタンドアロン ブート ファイル

OS が不要な場合は、生成された実行ファイルを自動的に実行するブート イメージを作成できます。

### FSBL (First Stage Boot Loader)

FSBL (First Stage Boot Loader) は、ブート時にビットストリームを読み込んで Zynq® アーキテクチャのプロセッシング システム (PS) をコンフィギュレーションします。

Vivado® Design Suite でプラットフォーム プロジェクトを開き、[File] → [Export] → [Export Hardware] をクリックします。

ザイリンクス SDK を使用する場合と同様に、[File] → [New] → [Application Project] をクリックし、fsbl という名前の新規ソフトウェア プロジェクトを作成します。

エクスポートされたハードウェア プラットフォームを使用して、リストから Zynq FSBL アプリケーションを選択します。これで、FSBL 実行ファイルが作成されます。

詳細は、[SDK ヘルプ](#)を参照してください。

FSBL を生成したら、SDx 環境フロー用の標準ディレクトリにコピーする必要があります。

例:

```
samples/platforms/zc702_axis_io/sw/boot/fsbl.elf
```

SDx システム コンパイラで FSBL が使用されるようにするには、<sdx:image> 要素の sdx:bif 属性で定義されている BIF ファイルで指定する必要があります。sdx:bif 属性の詳細は、[ソフトウェア プラットフォーム XML メタデータ リファレンス](#)を参照してください。ファイルは <path\_to\_platform>/sw/boot/ フォルダーに含める必要があります。



**ヒント:** Zynq AP SoC の BIF ファイルと Zynq UltraScale+ MPSoC デバイスの BIF ファイルは大きく異なります。



次に、Zynq®-7000 All Programmable (AP) SoC の boot.bif ファイルの例を示します。

```
/* linux */
the_ROM_image:
{
  [bootloader]<boot/fsbl.elf>
  <bitstream>
  <boot/u-boot.elf>
}
```

次に、Zynq UltraScale™ MPSoC デバイスの boot.bif ファイルの例を示します。

```
/* linux */
the_ROM_image:
{
  [fsbl_config] a53_x64
  [bootloader]<boot/fsbl.elf>
  [pmufw_image]<boot/pmufw.elf>
  [destination_device=pl] <bitstream>
  [destination_cpu=a53-0, exception_level=el-3, trustzone] <boot/bl31.elf>
  [destination_cpu=a53-0, exception_level=el-2] <boot/u-boot.elf>
}
```

## 実行ファイル

SDx 環境でブート イメージに含まれる実行ファイルが使用されるようにするには、BIT ファイルで指定する必要があります。

```
/* standalone */
the_ROM_image:
{
  [bootloader]<boot/fsbl.elf>
  <bitstream>
  <elf>
}
```

SDx 環境では、生成されたビットストリームと ELF ファイルが自動的に挿入されます。

## FreeRTOS コンフィギュレーション/バージョン変更

SDx™ IDE における FreeRTOS サポートでは、v8.2.3 ソフトウェア配布に含まれているデフォルトの FreeRTOSConfig.h を使用してビルド済みライブラリが定義済みリンカー スクリプトと共に使用されます。

FreeRTOS v8.2.3 コンフィギュレーションやそのリンカー スクリプトを変更、または別のバージョンの FreeRTOS を使用するには、次の手順に従ってください。

1. <path\_to\_install>/SDx/<version>/platforms/zc702 フォルダをローカルのフォルダにコピーします。
2. デフォルトのリンカー スクリプトを変更するには、<path\_to\_your\_platform>/zc702/sw/



freertos/lscript.ld を変更します。

3. FreeRTOS のコンフィギュレーション (FreeRTOSConfig.h) またはバージョンを変更するには、次を実行します。
  - a. FreeRTOS ライブラリを libfreertos.a としてビルドします。
  - b. インクルード ファイルを <path\_to\_your\_platform>/zc702/sw/freertos/include フォルダに追加します。
  - c. ライブラリ libfreertos.a を <path\_to\_your\_platform>/zc702/sw/freertos/li に追加します。
  - d. <path\_to\_your\_platform>/zc702/zc702.spfm ファイルの ("sdx:os  
sdx\_name="freertos" (sdx:includePaths="/aarch32-none/include and  
sdx:libraryPaths="/aarch-32-none/lib/freertos") 行を含むセクションでパスを変更します。
4. makefile で SDSoC プラットフォーム オプションを -sds-pf zc702 から -sds-pf <path\_to\_your\_platform>/zc702 に変更します。
5. ライブラリをビルドし直します。

SDx IDE の <path\_to\_install>/SDSoC/2016.1/tps/FreeRTOS フォルダには、コンフィギュレーション済み FreeRTOS v8.2.3 ライブラリ libfreertos.a をビルドするのに使用したソース ファイルと、単純な makefile および SDSoC\_readme.txt ファイルが含まれています。その他の必要条件や手順は、SDSoC\_readme.txt ファイルを参照してください。

- a. コマンド シェルを開きます。
- b. SDx IDE (<path\_to\_install>/SDx/2016.x/settings64.sh スクリプトまたは settings64.bat) を実行して、Zynq®-7000 AP SoC 向けの ARM GNU ツールチェーンを含むコマンド ライン ツールを実行する環境を設定します。
- c. フォルダをローカル フォルダにコピーします。
- d. FreeRTOSConfig.h を変更します。
- e. make コマンドを実行します。

FreeRTOS v8.2.3 を使用していない場合は、SDSoC\_readme.txt ファイルでソースがオフィシャル ソフトウェア ディストリビューションからどのように作成されたかを示す注記を確認してください。ZIP の解凍後、多少の変更はありますが (デモ アプリケーション main.c から memcpy、memset、および memcmp をライブラリ ソース ファイルに追加し、インクルード ファイルの参照を Task.h から task.h に変更)、フォルダ構造は元のままです。フォルダ構造を保持すると、コンフィギュレーション済み FreeRTOS v8.2.3 ライブラリをビルドするために作成された makefile を使用できます。

# プラットフォームのサンプル アプリケーション

## 概要

プラットフォームには、プラットフォームの使用法を示すサンプル アプリケーションをオプションで含めることができます。サンプル アプリケーションは、プラットフォームの `samples` ディレクトリの `template.xml` というファイルで定義される必要があります。次に、`<SDx_install>/SDx/2017.1/samples/sdspm/zc702_axis_io/src/samples` フォルダにある `zc702_axis_io` サンプル プラットフォームの例を示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest:Manifest xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
                  xmlns:manifest="http://www.xilinx.com/manifest">
  <template location="aximm" name="Unpacketized AXI4-Stream to DDR"
            description="Shows how to copy unpacketized AXI-Stream data
directly to DDR.">
    <supports>
      <and>
        <or>
          <os name="Linux"/>
          <os name="Standalone"/>
        </or>
      </and>
    </supports>
    <accelerator name="s2mm_data_copy" location="main.cpp"/>
  </template>
  <template location="stream" name="Packetize an AXI4-Stream"
            description="Shows how to packetize an unpacketized
AXI4-Stream.">
    <supports>
      <and>
        <or>
          <os name="Linux"/>
          <os name="Standalone"/>
        </or>
      </and>
    </supports>
    <accelerator name="packetize" location="packetize.cpp"/>
    <accelerator name="minmax" location="minmax.cpp"/>
  </template>
  <template location="pull_packet" name="Lossless data capture from
```

```
AXI4-Stream to DDR"
    description="Illustrates a technique to enable lossless data
capture from a free-running input source.">
    <supports>
        <and>
            <or>
                <os name="Linux"/>
            </or>
        </and>
    </supports>
    <accelerator name="PullPacket" location="main.cpp"/>
</template>
</manifest:Manifest>
```

最初の行は、テンプレート ファイルのフォーマットが XML であることを定義しており、必須です。

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

<manifest:Manifest> XML 要素は、テンプレート ファイルで定義されているすべてのアプリケーション テンプレートのコンテナとして必要です。

```
<manifest:Manifest xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:manifest="http://www.xilinx.com/manifest">
    <!-- ONE OR MORE TEMPLATE XML ELEMENTS GO HERE -->
</manifest:Manifest>
```

## <template> 要素

各アプリケーション テンプレートは <template> 要素内で定義され、<manifest> 要素内に複数の <template> タグを定義できます。<template> 要素には、次の表に示す複数の属性を設定できます。

表 1: <template> 要素の属性

属性	説明
location	テンプレート アプリケーションへの相対パス
name	SDSoC 環境に表示されるアプリケーション名
description	SDSoC 環境に表示されるアプリケーションの説明

例:

```
<template location="aximm" name="Unpacketized AXI4-Stream to DDR"
    description="Shows how to copy unpacketized AXI-Stream data
directly to DDR.">
```

<template> 要素には、アプリケーションの異なる特性を定義する複数の要素を含めることもできます。

表 2: <template> のサブ要素

要素	説明
<supports>	対応するテンプレートを定義するブール関数。
<includepaths>	コンパイラに -I フラグを使用して追加するアプリケーションに相対するパス。
<librarypaths>	リンカーに -L フラグを使用して追加するアプリケーションに相対するパス。
<libraries>	リンカーの -l フラグを使用してリンクされるプラットフォーム ライブラリ。
<exclude>	SDSoC プロジェクトにコピーしないディレクトリまたはファイル。
<system>	システムのアプリケーション プロジェクト設定 (データ モーション クロックなど)。
<accelerator>	ハードウェアをターゲットとする関数を指定するアプリケーション プロジェクト設定。
<compiler>	コンパイラ オプションを定義するアプリケーション プロジェクト設定。
<linker>	リンカー オプションを定義するアプリケーション プロジェクト設定。

## <supports> 要素

<supports> 要素は、選択した SDx プラットフォームに対応するオペレーティング システムを定義します。ブール関数を定義するには、<os> 要素を <and> および <or> 要素に含める必要があります。

次の例では、Linux、スタンドアロン、または FreeRTOS のいずれかを選択したときにオペレーティング システムとして選択可能なアプリケーションを定義しています。

```
<supports>
  <and>
    <or>
      <os name="Linux"/>
      <os name="Standalone"/>
      <os name="FreeRTOS"/>
    </or>
  </and>
</supports>
```

## <includepaths> 要素

<includepaths> 要素は、-I フラグを使用してコンパイラに渡すアプリケーションに相対するパスを定義します。各 <path> 要素には、location 属性を指定します。

次の例では、コンパイラに -I"./src/myinclude" -I"./src/dir/include" が追加されます。

```
<includepaths>
  <path location="myinclude"/>
  <path location="dir/include"/>
</includepaths>
```

## <librarypaths> 要素

<librarypaths> 要素は、-L フラグを使用してリンカーに渡すアプリケーションに相対するパスを定義します。各 <path> 要素には、location 属性を指定します。

次の例では、リンカーに -L "../src/mylibrary" -L "../src/dir/lib" が追加されます。

```
<librarypaths>
  <path location="mylibrary"/>
  <path location="dir/lib"/>
</librarypaths>
```

## <libraries> 要素

<libraries> 要素は、リンカーに -l フラグを使用して渡すライブラリを定義します。各 <lib> 要素には、name 属性を含みます。

次の例では、リンカーに -lmylib2 -lmylib2 が追加されます。

```
<libraries>
  <lib name="mylib1"/>
  <lib name="mylib2"/>
</libraries>
```

## <exclude> 要素

<exclude> 要素は、SDx で新規プロジェクトを作成する際にコピーしないディレクトリとファイルを定義します。

次の例では、新規プロジェクトを作成したときに、MyDir および MyOtherDir ディレクトリと、MyFile.txt および MyOtherFile.txt ファイルのコピーは作成されません。これにより、アプリケーション ディレクトリにアプリケーションのビルドには必要ないファイルまたはディレクトリを含めることができます。

```
<exclude>
  <directory name="MyDir"/>
  <directory name="MyOtherDir"/>
  <file name="MyFile.txt"/>
  <file name="MyOtherFile.txt"/>
</exclude>
```

## <system> 要素

オプションの <system> 要素は、新規プロジェクトを作成する際のシステムのアプリケーション プロジェクト設定を定義します。dmclkid 属性は、データ モーション クロック ID を定義します。<system> 要素が指定されていない場合は、データ モーション クロックにデフォルトのクロック ID が使用されます。

次の例では、新規プロジェクトを作成したときに、データ モーション クロック ID がデフォルトのクロック ID ではなく 2 に設定されます。

```
<system dmclkid="2"/>
```

## <accelerator> 要素

オプションの <accelerator> 要素は、新規プロジェクトを作成する際のハードウェアにインプリメントするよう設定されている関数のアプリケーション プロジェクト設定を定義します。name 属性は、関数の名前、location 属性は関数を含むソース ファイルへのパス (アプリケーション ソース ファイルを含むプラットフォームのフォルダーに相対するパス) を定義します。name および location は、<accelerator> 要素の必須の属性です。オプションの clkid 属性は、デフォルトの代わりに使用するアクセラレータ クロックを指定します。オプションのサブ要素 <hlsfiles> は、別のファイルにあるアクセラレータで呼び出されるコードを含むソース ファイル (アプリケーション ソース ファイルを含むプラットフォームのフォルダーに相対するパス) の名前 (name) を指定します。SDx 環境では通常、アプリケーションに対して <hlsfiles> の情報が推論されるので、デフォルトの動作を無効にする必要がある場合以外はこのサブ要素を指定する必要はありません。

次の例では、新規プロジェクトを作成したときに、2 つの関数 func1 および func2 がハードウェアに移動するよう指定されます。

```
<accelerator name="func1" location="func1.cpp"/>
<accelerator name="func2" location="func2.cpp" clkid="2">
  <hlsfiles name="func2_helper_a.cpp"/>
  <hlsfiles name="func2_helper_b.cpp"/>
</accelerator>
```

## <compiler> 要素

オプションの <compiler> 要素は、新規プロジェクトを作成する際のコンパイラのアプリケーション プロジェクト設定を定義します。inferredOptions 属性は、アプリケーションをビルドするのに必要なコンパイラ オプションを定義し、SDx 環境の [C/C++ Build Settings] ダイアログ ボックスの [Software Platform] の下にコンパイラの [Inferred Options] として表示されます。

次の例では、新規プロジェクトを作成したときにコンパイラ オプション -D MYAPPMACRO が追加されます。

```
<compiler inferredOptions="-D MYAPPMACRO"/>
```

## <linker> 要素

オプションの <linker> 要素は、新規プロジェクトを作成する際のリンカーのアプリケーション プロジェクト設定を定義します。inferredOptions 属性は、アプリケーションをビルドするのに必要なリンカー オプションを定義し、SDx 環境の [C/C++ Build Settings] ダイアログ ボックスに [linker Miscellaneous] として表示されます。

次の例では、新規プロジェクトを作成したときにリンカー オプション -poll-mode 1 が追加されます。

```
<linker inferredOptions="-poll-mode 1"/>
```

# プラットフォームのチェックリスト

## 概要

この付録のプラットフォーム作成プロセスの概要では、SDSoC でカスタム プラットフォームを使用できるようにするためのハードウェアおよびソフトウェア プラットフォームの要件とコンポーネント、プラットフォーム検証、サンプル アプリケーション サポート、ディレクトリ構造、プラットフォーム メタデータ XML ファイルについて説明しました。

SDSoC プラットフォーム作成プロセスでは、Vivado Design Suite についておよび Vivado Design Suite で Zynq-7000 または Zynq UltraScale+ MPSoC デザインを作成する方法、プラットフォームの観点からの SDSoC、ザイリンクス ソフトウェア開発キット (SDK) などのザイリンクス ソフトウェア開発ツール、エンベデッド ソフトウェア環境 (Linux またはベアメタル) を理解している必要があります。

SDSoC プラットフォームの作成プロセスに慣れていない場合は、主要な概念に関する記述に簡単に目を通し、このガイドの章の手順を読んで、[SDSoC プラットフォームの例](#)の例をいくつか確認してください。

以前に SDSoC プラットフォームを作成したことがある場合でも、各章をよく読んでください。アップグレード情報については、[SDSoC プラットフォームのアップグレード](#)を参照してください。

次のチェックリストに、SDSoC プラットフォーム作成に関連する主なタスクをまとめます。

1. Vivado Design Suite を使用して Zynq-7000 または Zynq UltraScale+ MPSoC ベースのデザインを作成します。
  - ・ Vivado ハードウェア プロジェクトを作成する際の要件およびガイドラインは、[ハードウェア プラットフォームの作成](#)を参照してください。Vivado Design Suite ツールを使用してハードウェア デザインをテストします。
2. サポートされるターゲット オペレーティング システムに対して、ブートおよびユーザー アプリケーション用のソフトウェア コンポーネントを提供します。
  - ・ プラットフォームに含まれるブート ファイルを使用して (該当する場合)、SDSoC により OS をブートするための SD イメージが作成されます。FSBL (First Stage Boot Loader) と、ブート用の BOOT.BIN ファイルの作成方法を記述したブート イメージ ファイル (BIF) が必要です。Linux ブートの場合は、U-Boot および Linux イメージ (デバイス ツリー、カーネル イメージ、およびルート ファイル システムの個別のファイルまたは統合ブート イメージ .ub ファイル) を提供します。ベアメタル アプリケーションの場合は、リンカー スクリプトを作成します。Zynq UltraScale+ MPSoC プラットフォームでは、ATF (ARM Trusted Firmware) および PMUFW (Power Management Unit FirmWare) も必要です。さらに、SD カード イメージに含める必要のある README ファイルおよびその他のファイルを作成します。
  - ・ プラットフォームでユーザーのアプリケーションにリンクするライブラリを提供する場合は、プラットフォームの一部としてヘッダーおよびライブラリを含めると便利です。
  - ・ 詳細は、[ソフトウェア プラットフォーム データの作成](#)を参照してください。



3. 1 つまたは複数のサンプル アプリケーションを作成します (オプション)。
  - ・ プラットフォーム フォルダに複数のサブフォルダを含む samples というフォルダを作成して、各サブフォルダにアプリケーションのソース コードを配置します。samples フォルダには、SDx 環境 IDE の New Project ウィザードでアプリケーションを作成するときに使用される template.xml ファイルも含めます。
  - ・ [プラットフォームのサンプル アプリケーション](#)を参照してください。
4. プラットフォーム ディレクトリを作成し、プラットフォーム XML メタデータ ファイルを含めます。
  - ・ 次に、プラットフォーム ディレクトリの基本的な構造を示します。ここに示すフォルダ名およびファイル名を使用してください (<platform> は実際のプラットフォーム名に置換)。ここにリストされていないフォルダおよびファイルの名前には決まりはありません。samples フォルダはオプションです。
    - myplatforms
      - <platform>
        - ・ <platform>.xpfm
        - ・ hw
          - ・ vivado
          - ・ <platform>.hpfm
        - ・ sw
          - ・ <processor\_boot\_folder>
          - ・ <platform>.spfm
      - samples
        - ・ <sample\_application\_folder>
        - ・ template.xml
5. プラットフォームで SDSoC 環境がサポートされることを確認します。
  - ・ [SDSoC プラットフォーム](#)の章では、SDSoC システム コンパイラで使用されるデータ ユーバーの platform/Conformance テストについて説明されています。各テストは、SDx ターミナルを起動するか SDx のインストールに含まれる settings64 スクリプト (.bat、.sh、または .csh) を実行して SDx 環境を使用可能な状態にして、シェルのコマンドラインから make を実行することにより正しく構築する必要があります。これらのテストは、プラットフォーム ボード上でも実行する必要があります。
6. SDx 環境 IDE でプラットフォームを使用してプロジェクトを作成できることを確認します。
  - ・ SDx 環境 IDE を起動し、New Project ウィザードを使用して SDSoC プロジェクトを作成します。プロジェクト名を指定すると、プラットフォームのリストが表示されます。[Add Custom Platform] をクリックしてプラットフォーム フォルダを選択します。プラットフォームに samples フォルダが含まれる場合、アプリケーションのいずれかを選択するか、Empty アプリケーションを選択して後でソース ファイルを追加できます。プロジェクトを作成したら、ビルドして実行し、ELF ファイルをデバッグします。

# SDSoC プラットフォームのアップグレード

## 概要

新しい SDx 開発環境のリリースがサポートされるようにするには、プラットフォームも含めて、Vivado Design Suite を最新のバージョンにアップグレードする必要があります。IP インテグレーター デザインのアップデートは、現在のリリースの最新の IP バージョンを使用するだけの単純なプロセスで済むこともありますが、リリース間で主なバージョン変更があった場合などは、IP のインターフェイス信号を追加したり削除したり、パラメーターをアップデートしたりすることで、複雑になることもあり、Vivado Design Suite プロジェクトのアップグレードの手間が増えることがあります。

ハードウェア プラットフォームの新機能と互換性のあるように、または新機能の利点を生かせるようにするには、ソフトウェア プラットフォームもアップデートする必要があります。最後に、SDSoC プラットフォーム ユーティリティを使用して最上位プラットフォームとハードウェア記述を生成し直す必要があります。これには、単に Vivado IP インテグレーター ブロック デザインを最新リリースにアップグレードして、ソフトウェア コンポーネントを最新の SDSoC ツールでビルドし直すだけで済むこともあります。



### 重要:

Vivado ツールでは、Vivado Design Suite の新しいバージョンがリリースされるたびに [Upgrade IP] を実行する必要があります。SDSoC プラットフォーム ユーティリティでプラットフォームが生成される際や SDx IDE で Vivado ツールを起動する際に IP のロックを示すエラーが発生した場合は、[Vivado Design Suite プロジェクト](#)に示すように Vivado プロジェクトが適切にコピーされなかったか、Vivado プロジェクトで使用された IP が新しいリリース用にアップグレードされなかったことを意味します。

以前のリリースから SDSoC ハードウェア プラットフォームをアップグレードするには、Vivado ツールの新しいバージョンで Vivado プロジェクトを開き、IP インテグレーター ブロック デザインおよびすべての IP をアップグレードしてから、出力ファイルを生成し直します。ブロック デザイン プロジェクトのアップデートに関する詳細は、『Vivado Design Suite ユーザー ガイド: IP インテグレーターを使用した IP サブシステムの設計』(UG994) の[このセクション](#)を参照してください。

# チュートリアル: SDSoC プラットフォーム ユーティリティの使用

## 概要

この付録には、サンプル プラットフォームを作成する単純なチュートリアルを含めます。プラットフォーム設定のほとんどが SDSoC プラットフォーム ユーティリティ GUI で設定できます。コンフィギュレーションからプラットフォームを生成することもできます。プラットフォームが作成されたら、さらにカスタマイズすることもできます。

次のチュートリアルを実行する際は、まず `<install>/SDx/2017.1/samples/sdspfm` のサンプル ファイルを `C:/temp` や `/temp` などの別のディレクトリに保存してください。samples/sdspfm フォルダをコピーしておく、元のファイルはそのまま、ソース ファイルに変更したり、インストール ディレクトリのパーミッション問題などを回避したりできます。



**ヒント:** 次のチュートリアルプラットフォーム例だけでなく、`<install>/SDx/2017.1/platforms` に含まれる標準的な SDx プラットフォームも参照してください。

## 演習 1: ZC702 プラットフォームの作成



**重要:** 次のチュートリアル演習では、サンプル フォルダが `<samples_dir>` と示され、SDx インストールからローカルにコピーしてきたサンプル プラットフォームのディレクトリを意味します。

このチュートリアルでは、SDSoC インストールに含まれる ZC702 プラットフォームの基本的なプラットフォームを作成し直します。SDSoC プラットフォーム ユーティリティの GUI の開始方法、使用方法をお見せし、Linux、FreeRTOS、スタンドアロン OS をターゲットにした単純なプラットフォームを作成します。SDSoC プラットフォーム ユーティリティの詳細は、[SDSoC プラットフォームの作成](#)を参照してください。

1. SDx ターミナル ウィンドウを起動します。

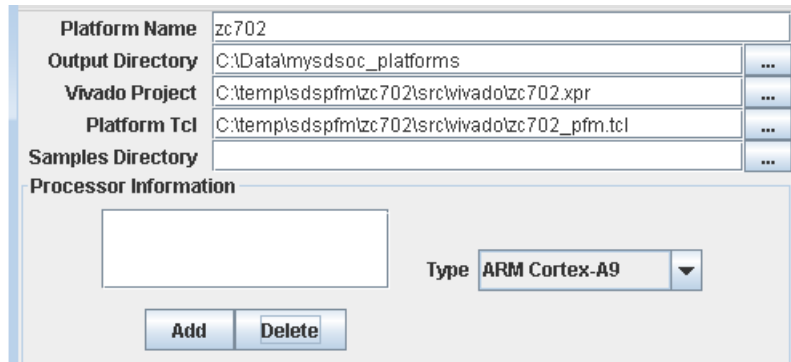
- ・ Windows の場合は、[スタート] → [すべてのプログラム] → [Xilinx Design Tools] → [SDx 2017.1] → [SDx Terminal 2017.1] をクリックします。
- ・ Linux の場合は、使用しているシェル タイプに合わせて `settings.sh/csh` ファイルを読み込みます。

- SDx ターミナルから SDSoC プラットフォーム ユーティリティ の GUI を起動します。

★ **重要:** SDSoC プラットフォーム ユーティリティは、Windows OS の C:/ または同等のルート ディレクトリから起動して、Windows でプラットフォーム ファイルを生成する際にパス名の長さの制限の問題が発生しないようにしてください。

- ・ ターミナルまたはシェルで `sdspfm -gui` と入力して GUI を起動します。
- SDSoC プラットフォーム ユーティリティの GUI が開いたら、次の図のように基本プラットフォームの情報を入力します。

図 7: ハードウェア プラットフォームの指定



- ・ [Platform Name]: zc702

★ **重要:** プラットフォーム名は、ハードウェアを定義する Vivado Design Suite プロジェクトおよびプロジェクトに含まれる IP インテグレーター ブロック デザインの名前と同じである必要があります。

- ・ システムに適した [Output Directory] を選択してください。
  - ・ [Vivado Project] および [Platform Tcl] には、<samples\_dir>/zc702/src/vivado ディレクトリに含まれるファイルを指定します。
- Vivado プロジェクト ファイル (zc702.xpr) を指定すると、[Processor Type] フィールドに [ARM Cortex-A9] と表示されます。
  - [Processor Information] フィールドで [Add] ボタンをクリックし、プラットフォーム コンフィギュレーションに A9 プロセッサを追加します。

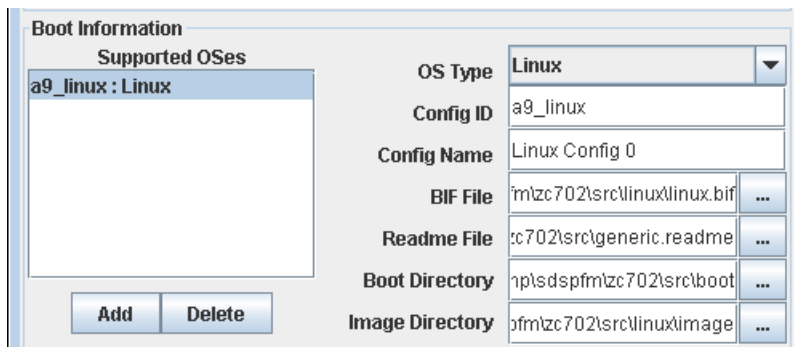
💡 **ヒント:** プラットフォーム コンフィギュレーションに必要なプロセッサは、プラットフォームでまだすべてのコアが使用できる状態であっても、各タイプごとに 1 つのみです。

図 8: プロセッサの追加



6. [Boot Information] フィールドでは、まず Linux OS を設定しますが、OS を設定する順番は重要ではありません。

図 9: Linux ブート情報



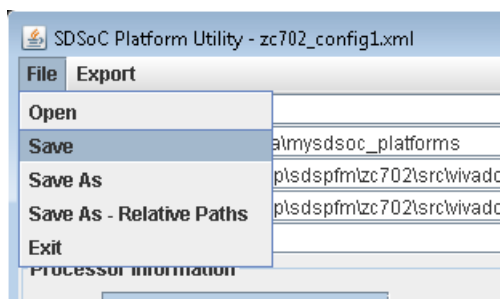
- a. [Boot Information] フィールドで [Add] ボタンをクリックし、新しいコンフィギュレーションを追加します。
- b. [Config ID]: a9\_linux になっていますが、どの名前でも指定できます。
- c. [OS Type]: ドロップダウン リストから [Linux] を選択します。
- d. [Config Name]: この名前は自動的に生成されますが、必要であれば変更できます。
- e. [BIF File]: <samples\_dir>/zc702/src/linux/linux.bif
- f. [Readme File]: <samples\_dir>/zc702/src/generic.readme
- g. [Boot Directory]: <samples\_dir>/zc702/src/boot
- h. [Image Directory]: <samples\_dir>/zc702/src/linux/image



**ヒント:** OS 設定は、右側の設定を変更すると、選択したコンフィギュレーションに自動的に保存されます。

7. この段階では、GUI の上のメニューから [File] → [Save] をクリックして、ファイルにプラットフォーム コンフィギュレーションを保存できます。コンフィギュレーション ファイルの名前とディレクトリを入力する画面が表示されます。プラットフォーム コンフィギュレーション ファイルはどのディレクトリにも保存できますが、生成した SDSoC プラットフォーム内に保存すると、プラットフォームを生成し直したときにコンフィギュレーション ファイルが削除されてしまいますので、保存しないようにしてください。

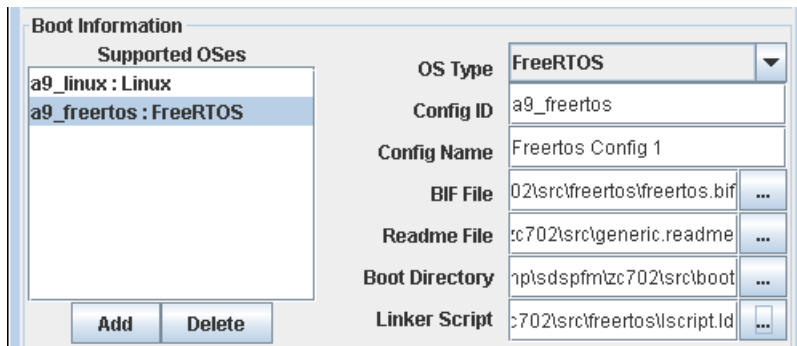
図 10: コンフィギュレーションの保存



プラットフォーム コンフィギュレーション ファイルを保存しておくと、SDSoC プラットフォーム ユーティリティにプラットフォームの詳細を読み込み直して、必要に応じて再生成したり、新しいプラットフォームの開始点として使用したりできます。保存したコンフィギュレーション ファイルは [File] → [Open] コマンドで読み込むことができます。

8. 次の設定を使用して FreeRTOS を設定してプラットフォームに追加します。

図 11: FreeRTOS コンフィギュレーション

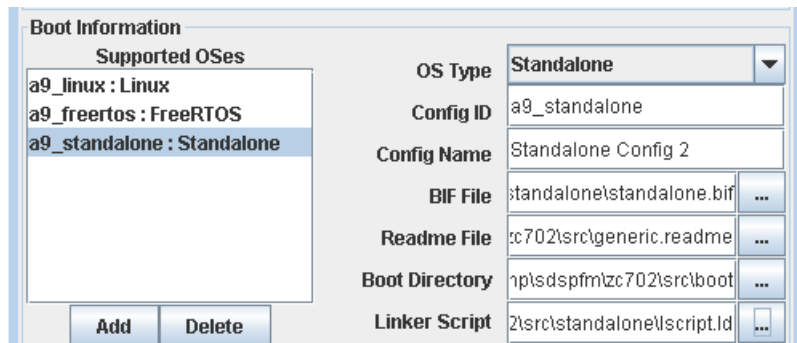


The screenshot shows the 'Boot Information' dialog box. On the left, under 'Supported OSes', 'a9\_freertos : FreeRTOS' is selected. On the right, the configuration fields are as follows:

OS Type	FreeRTOS
Config ID	a9_freertos
Config Name	Freertos Config 1
BIF File	02\src\freertos\freertos.bif
Readme File	zc702\src\generic.readme
Boot Directory	np\sdspfm\zc702\src\boot
Linker Script	zc702\src\freertos\lscript.ld

- [Boot Information] フィールドで [Add] ボタンをクリックし、新しいコンフィギュレーションを追加します。
  - [Config ID]: a9\_freertos.
  - [OS Type]: ドロップダウン リストから [FreeRTOS] を選択します。
  - [Config Name]: FreeRTOS Config 1 に変更します。
  - [BIF File]: <samples\_dir>/zc702/src/freertos/freertos.bif
  - [Readme File]: <samples\_dir>/zc702/src/generic.readme
  - [Boot Directory]: <samples\_dir>/zc702/src/boot
  - [Linker Script]: <samples\_dir>/zc702/src/freertos/lscript.ld
9. 次のようにスタンドアロン OS を設定してプラットフォームに追加します。

図 12: スタンドアロン コンフィギュレーション



The screenshot shows the 'Boot Information' dialog box. On the left, under 'Supported OSes', 'a9\_standalone : Standalone' is selected. On the right, the configuration fields are as follows:

OS Type	Standalone
Config ID	a9_standalone
Config Name	Standalone Config 2
BIF File	standalone\standalone.bif
Readme File	zc702\src\generic.readme
Boot Directory	np\sdspfm\zc702\src\boot
Linker Script	2\src\standalone\lscript.ld

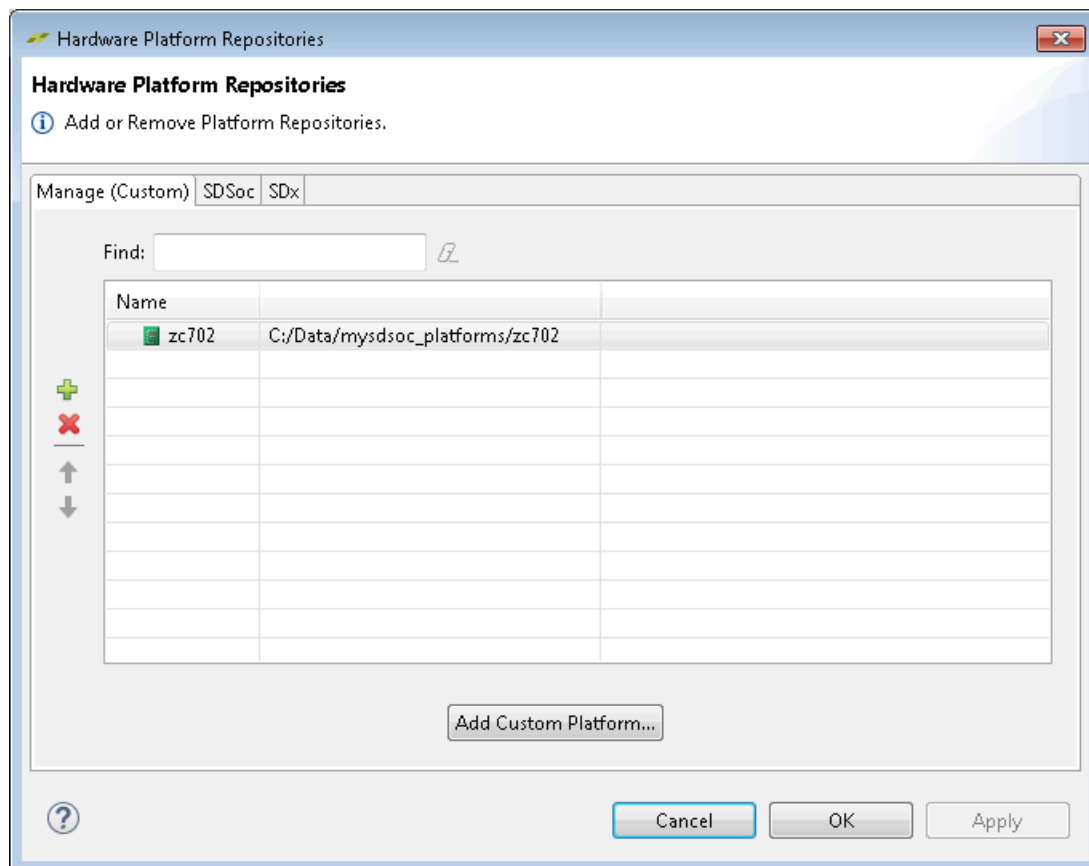
- [Boot Information] フィールドで [Add] ボタンをクリックし、新しいコンフィギュレーションを追加します。
  - [Config ID]: a9\_standalone.
  - [OS Type]: ドロップダウン リストから [Standalone] を選択します。
  - [Config Name]: Standalone Config 2 に変更します。
  - [BIF File]: <samples\_dir>/zc702/src/standalone/standalone.bif
  - [Readme File]: <samples\_dir>/zc702/src/generic.readme
  - [Boot Directory]: <samples\_dir>/zc702/src/boot
  - [Linker Script]: <samples\_dir>/zc702/src/standalone/lscript.ld
10. [File] → [Save] コマンドを使用してコンフィギュレーション ファイルを保存し直します。

11. GUI の一番下で [Generate] ボタンをクリックします。これでプラットフォーム コンフィギュレーション設定がプラットフォームを作成する `sdspfm` コマンドライン ユーティリティに送信されます。プラットフォームの作成状況は、GUI の起動時に使用した SDx ターミナルでアップデートされていきます。プラットフォームの生成が終了したら、エラー メッセージがある場合はそれが SDx ターミナルに表示されます。[OK] をクリックしてメッセージを閉じます。
12. [File] → [Exit] コマンドを使用して SDSoc プラットフォーム ユーティリティを終了します。

プラットフォームが作成されたら、出力ディレクトリで生成されたプラットフォーム ファイルを確認します。Vivado プロジェクトは `hw` フォルダに、さまざまな OS コンフィギュレーションは `sw` フォルダに含まれます。

SDx IDE で [Xilinx] → [Add Custom Platform] をクリックすると、新しく作成したプラットフォームが SDx プロジェクトで使用可能なプラットフォーム ライブラリに追加されます。また、プラットフォームは、ライブラリに追加する前であればファイル システムのどこにでも移動できます。

図 13: ハードウェア プラットフォームの追加



## 演習 2: ZC702\_AXIS\_IO プラットフォーム

★ 重要: 次のチュートリアル演習では、サンプル フォルダが `<samples_dir>` と示され、SDx インストールからローカルにコピーしてきたサンプル プラットフォームのディレクトリを意味します。



演習 1 では、Linux、FreeRTOS、スタンドアロン OS を使用して基本的な ZC702 プラットフォームを作成しました。また、SDSoC プラットフォーム ユーティリティの GUI を使用して、プラットフォームを設定し、その設定を保存し、プラットフォームを生成しました。このチュートリアルでは、プラットフォーム ユーティリティを使用して `zc702_axis_io` を作成します。このプラットフォームの詳細は、[例: SDSoC プラットフォームのダイレクト I/O](#)を参照してください。

`zc702_axis_io` プラットフォームには、プラットフォーム専用のインクルード ヘッダー、およびリンクする必要のあるライブラリがあり、サンプル アプリケーションも数個含まれて射ます。この演習を開始するには、まずプラットフォームのスタティック ライブラリ ファイルを作成します。

#### 1. SDx ターミナル ウィンドウを起動します。

- ・ Windows の場合は、[スタート] → [すべてのプログラム] → [Xilinx Design Tools] → [SDx 2017.1] → [SDx Terminal 2017.1] をクリックします。
- ・ Linux の場合は、使用しているシェル タイプに合わせて `settings.sh/csh` ファイルを読み込みます。

#### 2. このターミナル ウィンドウで `zc702_axis_io` プラットフォームのインクルード フォルダーに移動して、Linux とスタンドアロン オペレーティング システム両方に対して `make` コマンドでスタティック ライブラリ (`.a`) を作成します。

```
cd <samples_dir>/zc702_axis_io/src/src
cd linux
make
cd ../standalone
make
cd C:/
```

スタティック ライブラリ ファイルが作成されたら、SDx ターミナル ウィンドウを使用して `zc702_axis_io` プラットフォーム コンフィギュレーション ファイルを定義して、プラットフォームを作成できます。

#### 1. SDx ターミナルから SDSoC プラットフォーム ユーティリティ の GUI を起動します。

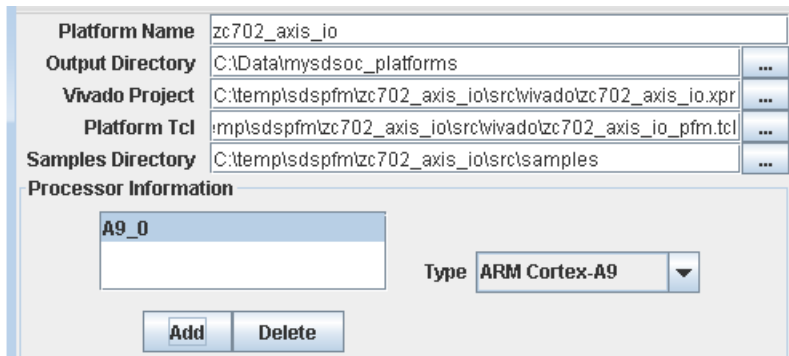
★ **重要:** SDSoC プラットフォーム ユーティリティは、Windows OS の C:/ または同等のルート ディレクトリから起動して、Windows でプラットフォーム ファイルを生成する際にパス名の長さの制限の問題が発生しないようにしてください。

- ・ ターミナルまたはシェルで `sdspfm -gui` と入力して GUI を起動します。



2. SDSoC プラットフォーム ユーティリティの GUI が開いたら、次の図のように基本プラットフォームの情報を入力します。

図 14: ハードウェア プラットフォームの指定



The screenshot shows the 'Platform Name' field set to 'zc702\_axis\_io'. The 'Output Directory' is 'C:\Data\mysdsoc\_platforms'. The 'Vivado Project' is 'C:\temp\sdspfm\zc702\_axis\_io\src\vivado\zc702\_axis\_io.xpr'. The 'Platform Tcl' is 'C:\temp\sdspfm\zc702\_axis\_io\src\vivado\zc702\_axis\_io\_pfm.tcl'. The 'Samples Directory' is 'C:\temp\sdspfm\zc702\_axis\_io\src\samples'. Under 'Processor Information', 'A9\_0' is entered in the name field, and 'ARM Cortex-A9' is selected in the 'Type' dropdown. 'Add' and 'Delete' buttons are at the bottom.

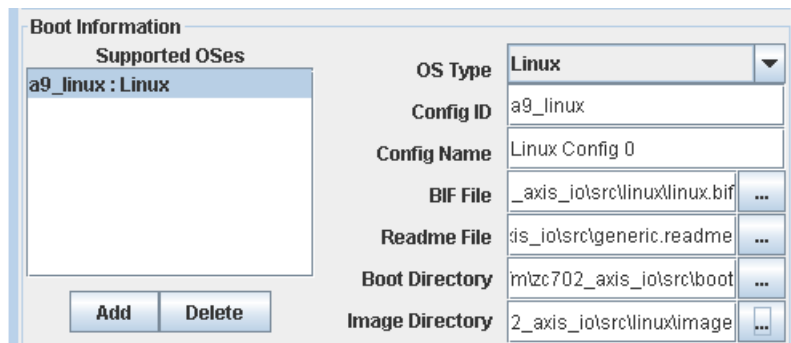
- ・ [Platform Name]: zc702\_axis\_io
  - ・ システムに適した [Output Directory] を選択します。
  - ・ [Vivado Project] および [Platform Tcl] には、<samples\_dir>/zc702\_axis\_io/src/vivado ディレクトリに含まれるファイルを指定します。
  - ・ [Samples Directory]: <samples\_dir>/zc702\_axis\_io/src/samples
3. [Processor Information] フィールドで [Add] ボタンをクリックし、プラットフォーム コンフィギュレーションに A9 プロセッサを追加します。



**ヒント:** プラットフォーム コンフィギュレーションに必要なプロセッサは、プラットフォームでまだすべてのコアが使用できる状態であっても、各タイプごとに 1 つのみです。

4. [Boot Information] フィールドでは、まず Linux OS を設定しますが、OS を設定する順番は重要ではありません。

図 15: Linux ブート情報

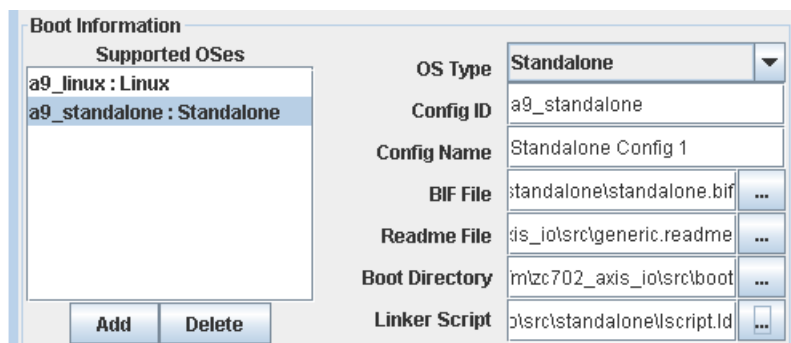


The screenshot shows the 'Boot Information' dialog box. On the left, under 'Supported OSes', there is a list with 'a9\_linux : Linux'. Below this list are 'Add' and 'Delete' buttons. On the right, the configuration fields are as follows:

OS Type	Linux
Config ID	a9_linux
Config Name	Linux Config 0
BIF File	_axis_io\src\linux\linux.bif
Readme File	dis_io\src\generic.readme
Boot Directory	m\zc702_axis_io\src\boot
Image Directory	2_axis_io\src\linux\image

- a. [Boot Information] フィールドで [Add] ボタンをクリックし、新しいコンフィギュレーションを追加します。
  - b. [Config ID]: a9\_linux になっていますが、どの名前でも指定できます。
  - c. [OS Type]: ドロップダウン リストから [Linux] を選択します。
  - d. [Config Name]: この名前は自動的に生成されますが、必要であれば変更できます。
  - e. [BIF File]: <samples\_dir>/zc702\_axis\_io/src/linux/linux.bif
  - f. [Readme File]: <samples\_dir>/zc702\_axis\_io/src/generic.readme
  - g. [Boot Directory]: <samples\_dir>/zc702\_axis\_io/src/boot
  - h. [Image Directory]: <samples\_dir>/zc702\_axis\_io/src/linux/image
5. 次のようにスタンドアロン OS を設定してプラットフォームに追加します。

図 16: スタンドアロン コンフィギュレーション



The screenshot shows the 'Boot Information' dialog box with the 'Standalone' configuration. On the left, under 'Supported OSes', there is a list with 'a9\_linux : Linux' and 'a9\_standalone : Standalone'. Below this list are 'Add' and 'Delete' buttons. On the right, the configuration fields are as follows:

OS Type	Standalone
Config ID	a9_standalone
Config Name	Standalone Config 1
BIF File	standalone\standalone.bif
Readme File	dis_io\src\generic.readme
Boot Directory	m\zc702_axis_io\src\boot
Linker Script	l\src\standalone\lscript.ld

- a. [Boot Information] フィールドで [Add] ボタンをクリックし、新しいコンフィギュレーションを追加します。
- b. Config ID: a9\_standalone.
- c. [OS Type]: ドロップダウン リストから [Standalone] を選択します。
- d. [Config Name]: Standalone Config 1 に変更します。
- e. [BIF File]: <samples\_dir>/zc702\_axis\_io/src/standalone/standalone.bif
- f. [Readme File]: <samples\_dir>/zc702\_axis\_io/src/generic.readme
- g. [Boot Directory]: <samples\_dir>/zc702\_axis\_io/src/boot
- h. [Linker Script]: <samples\_dir>/zc702\_axis\_io/src/standalone/lscript.ld

6. [Supported OSes] の下で Linux OS の a9\_linux を選択し、このコンフィギュレーションのインクルードパスとライブラリを設定します。
  - ・ [Include Paths] タブでインクルードパスを <samples\_dir>/zc702\_axis\_io/src/src/inc に指定し、[Add] をクリックしてリストに追加します。
  - ・ [Libraries] タブの [Library Path] フィールドで <samples\_dir>/zc702\_axis\_io/src/src/linux/libzc702\_axis\_io.a ファイルを選択し、[Add] をクリックしてリストに追加します。

図 17: インクルードパスおよびライブラリ



7. スタンドアロン OS の a9\_standalone を選択し、このコンフィギュレーションのインクルードパスとライブラリを設定します。
  - ・ [Include Paths] タブでインクルードパスを <samples\_dir>/zc702\_axis\_io/src/src/inc に指定し、[Add] をクリックしてリストに追加します。
  - ・ [Libraries] タブの [Library Path] フィールドで <samples\_dir>/zc702\_axis\_io/src/src/standalone/libzc702\_axis\_io.a ファイルを選択し、[Add] をクリックしてリストに追加します。
8. [File] → [Save] をクリックしてプラットフォーム コンフィギュレーション ファイルの名前とディレクトリを指定して保存します。
9. [Generate] ボタンをクリックしてプラットフォームを作成します。プラットフォームの作成状況は、GUI の起動時に使用した SDx ターミナルでアップデートされていきます。プラットフォームの生成が終了したら、エラーメッセージがある場合はそれが SDx ターミナルに表示されます。[OK] をクリックしてメッセージを閉じます。
10. [File] → [Exit] コマンドを使用して SDSoC プラットフォーム ユーティリティを終了します。

## 演習 3: ZCU102 プラットフォーム



**重要:** 次のチュートリアル演習では、サンプル フォルダーが <samples\_dir> と示され、SDx インストールからローカルにコピーしてきたサンプル プラットフォームのディレクトリを意味します。

この演習では、zcu102\_es1 プラットフォームを作成します。

1. SDx ターミナル ウィンドウを起動します。

- ・ Windows の場合は、[スタート] → [すべてのプログラム] → [Xilinx Design Tools] → [SDx 2017.1] → [SDx Terminal 2017.1] をクリックします。
- ・ Linux の場合は、使用しているシェル タイプに合わせて settings.sh/csh ファイルを読み込みます。

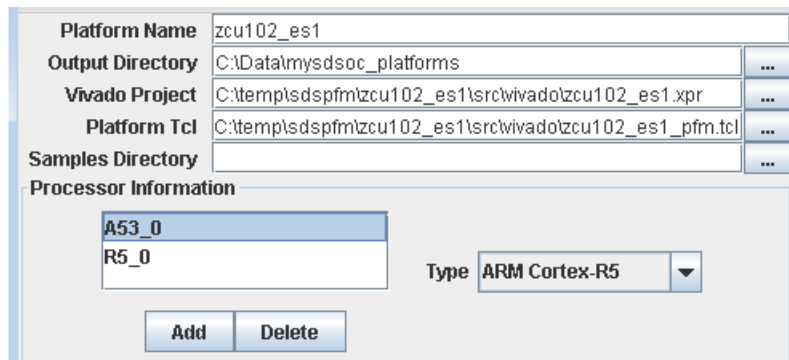
2. SDx ターミナルから SDSoC プラットフォーム ユーティリティ の GUI を起動します。

★ **重要:** SDSoC プラットフォーム ユーティリティは、Windows OS の C:/ または同等のルート ディレクトリから起動して、Windows でプラットフォーム ファイルを生成する際にパス名の長さの制限の問題が発生しないようにしてください。

- ・ ターミナルまたはシェルで sdspfm -gui を入力して GUI を起動します。

3. SDSoC プラットフォーム ユーティリティの GUI が開いたら、次の図のように基本プラットフォームの情報を入力します。

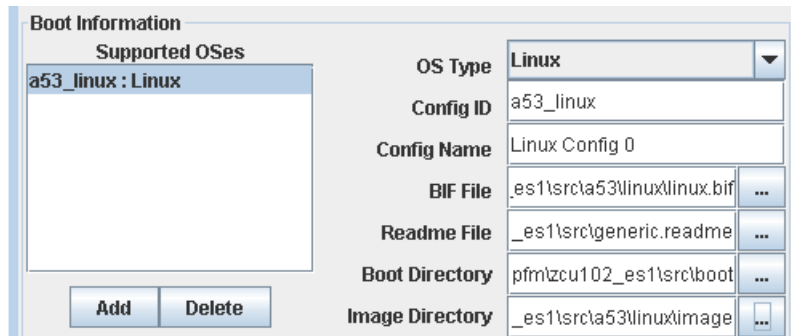
図 18: ハードウェア プラットフォームの指定



- ・ [Platform Name]: zcu102\_es1
  - ・ システムに適した [Output Directory] を選択してください。
  - ・ [Vivado Project] および [Platform Tcl] には、<samples\_dir>/zcu102\_es1/src/vivado ディレクトリに含まれるファイルを指定します。
4. [Processor Information] の [Type] フィールドで ARM Cortex-A53 プロセッサを選択して、[Add] でそのプロセッサ コアをプラットフォーム コンフィギュレーションに追加します。
5. 再び [Type] フィールドで ARM Cortex-R5 プロセッサを選択して、[Add] でこのコアもプラットフォーム コンフィギュレーションに追加します。

6. A53\_0 プロセッサを選択し、[Boot Information] の下に Linux OS を追加して設定します。

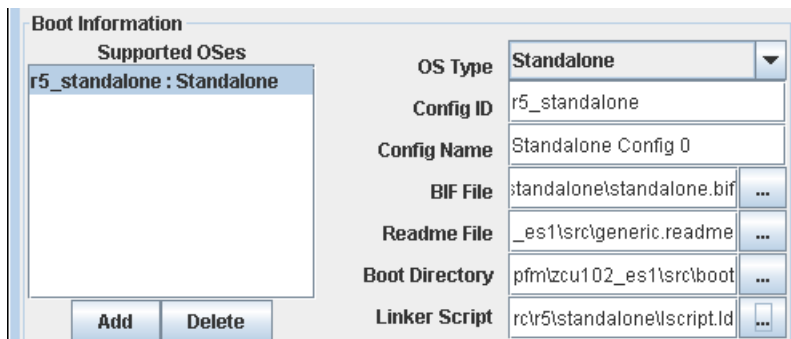
図 19: A53 プロセッサ コア – Linux ブート情報



- a. [Boot Information] フィールドで [Add] ボタンをクリックし、新しいコンフィギュレーションを追加します。
  - b. [Config ID]: a53\_linux になっていますが、どの名前でも指定できます。
  - c. [OS Type]: ドロップダウン リストから [Linux] を選択します。
  - d. [Config Name]: この名前は自動的に生成されますが、必要であれば変更できます。
  - e. [BIF File]: <samples\_dir>/zcu102\_es1/src/a53/linux/linux.bif
  - f. [Readme File]: <samples\_dir>/zcu102\_es1/src/generic.readme
  - g. [Boot Directory]: <samples\_dir>/zcu102\_es1/src/boot
  - h. [Image Directory]: <samples\_dir>/zcu102\_es1/src/a53/linux/image
7. A53\_0 プロセッサを選択したまま、次のように スタンドアロン OS を設定してプラットフォームに追加します。
    - a. [Boot Information] フィールドで [Add] ボタンをクリックし、新しいコンフィギュレーションを追加します。
    - b. [Config ID]: a53\_standalone.
    - c. [OS Type]: ドロップダウン リストから [Standalone] を選択します。
    - d. [Config Name]: Standalone Config 1 に変更します。
    - e. [BIF File]: <samples\_dir>/zcu102\_es1/src/a53/standalone/standalone.bif
    - f. [Readme File]: <samples\_dir>/zcu102\_es1/src/generic.readme
    - g. [Boot Directory]: <samples\_dir>/zcu102\_es1/src/boot
    - h. [Linker Script]: <samples\_dir>/zcu102\_es1/src/a53/standalone/lscript.ld

8. [Processor Information] の下で R5\_0 プロセッサを選択して、このプロセッサ コアに対してスタンドアロン OS を追加して設定します。

図 20: R5 プロセッサ コア – スタンドアロン ブート情報



- a. [Boot Information] フィールドで [Add] ボタンをクリックし、新しいコンフィギュレーションを追加します。
- b. [Config ID]: r5\_standalone.
- c. [OS Type]: ドロップダウン リストから [Standalone] を選択します。
- d. [Config Name]: Standalone Config 0 に変更します。
- e. [BIF File]: <samples\_dir>/zcu102\_es1/src/r5/standalone/standalone.bif
- f. [Readme File]: <samples\_dir>/zcu102\_es1/src/generic.readme
- g. [Boot Directory]: <samples\_dir>/zcu102\_es1/src/boot
- h. [Linker Script]: <samples\_dir>/zcu102\_es1/src/r5/standalone/lscript.ld
9. [File] → [Save] をクリックしてプラットフォーム コンフィギュレーション ファイルの名前とディレクトリを指定して保存します。
10. [Generate] ボタンをクリックしてプラットフォームを作成します。プラットフォームの作成状況は、GUI の起動時に使用した SDx ターミナルでアップデートされていきます。プラットフォームの生成が終了したら、エラー メッセージがある場合はそれが SDx ターミナルに表示されます。[OK] をクリックしてメッセージを閉じます。
11. [File] → [Exit] コマンドを使用して SDSoC プラットフォーム ユーティリティを終了します。

# SDSoC プラットフォームの例

## 概要

この付録には、ハードウェアだけでなく、OS カーネル、ブートローダー、ファイル システム、ライブラリなどを含め、Vivado® Design Suite とソフトウェア ランタイム環境を使用してビルドされたハードウェア システムから作成された SDSoC プラットフォームの単純な例が含まれます。各例はよく使用されるプラットフォーム機能を示しており、ザイリンクスの ZC702 ボードを使用してビルドされています。

- ・ `zc702_axis_io`: SDSoC プラットフォームで FPGA ピンからのダイレクト I/O で、データ ストリームへアクセス
- ・ `zc702_acp`: プラットフォームと `sdscc` システム コンパイラ間でプロセッシングシステムの AXI バス インターフェイスを共有

各例には、次の情報が含まれます。

- ・ プラットフォームとその説明の記述
- ・ SDSoC ハードウェア プラットフォーム メタデータ ファイルの生成方法
- ・ 必要な場合のためのプラットフォーム ソフトウェア ライブラリの作成方法
- ・ SDSoC ソフトウェア プラットフォーム メタデータ ファイルの記述
- ・ 基本的なプラットフォーム テスト

これらのプラットフォーム例だけでなく、`<sdx_root>/platforms` ディレクトリの SDx IDE に含まれる標準 SDSoC プラットフォームについても確認することをお勧めします。

## 例: SDSoC プラットフォームのダイレクト I/O

SDSoC プラットフォームには、生の物理的なデータ ストリームをプラットフォーム インターフェイス仕様の一部としてエクスポートできるように AXI4-Stream インターフェイスに変換し、入力および出力サブシステム (アナログ/デジタル コンバーターおよびデジタル/アナログ コンバーター、または Video I/O) を含めることができます。`zc702_axis_io` サンプル プラットフォームについては、『SDSoC 環境ユーザーガイド』(UG1027) の「外部 I/O の使用」を参照してください。この例には、入力データ ストリームがデータ損失なしにメモリ バッファに書き込まれると、アプリケーションにより AXI 転送レベルでデータ ストリームをパケットにして、パケット フレーミングを必要とするその他のファンクション (DMA を含むが、これだけではない) とのデータのやり取りを示します。



**推奨:** このプラットフォームのソース コードについては、`<sdx_root>/samples/sdspm/zc702_axis_io` を参照してください。







## SDSoC ハードウェア プラットフォーム記述の生成

Vivado での SDSoC Tcl コマンドに示すように、プラットフォーム ハードウェアのポート インターフェイスは Vivado プロジェクトで定義され、スクリプト ファイル (zc702\_axis\_io/src/vivado/zc702\_axis\_io\_pfm.tcl) に含まれる Tcl コマンドを使用して SDSoC プラットフォーム ユーティリティで抽出されます。

1. 次のコマンドでは、ハードウェア プラットフォーム オブジェクトが作成されます。

```
set pfm [sdsoc::create_pfm zc702_axis_io.hpfm]
```

2. 次のコマンドでプラットフォーム名を宣言して、sdsoc -sds-pf-info zc702\_axis\_io を実行したときに表示される簡単な説明を指定します。

```
sdsoc::pfm_name      $pfm "xilinx.com" "xd" "zc702_axis_io" "1.0"
sdsoc::pfm_description $pfm "Zynq ZC702 Board With Direct I/O"
```

3. 次のコマンドでクロックを宣言します。デフォルトのプラットフォーム クロックの id は 1 です。true 引数は、このクロックがプラットフォームのデフォルトであることを示します。

```
sdsoc::pfm_clock      $pfm FCLK_CLK0 ps7 0 false psr0
sdsoc::pfm_clock      $pfm FCLK_CLK1 ps7 1 true  psr1
```



**ヒント:** このコマンドでは、各プラットフォーム クロックに必要な関連する proc\_sys\_reset IP インスタンス (psr0、psr1) も指定されます。

4. 次のコマンドでプラットフォーム AXI インターフェイスを宣言します。各 AXI ポートにメモリタイプの宣言が必要です。M\_AXI\_GP (汎用 AXI マスター)、S\_AXI\_ACP (キャッシュ コヒーレント スレーブ インターフェイス)、S\_AXI\_HP (高パフォーマンス ポート)、または MIG (外部メモリ コントローラーへのインターフェイス) のいずれかを指定します。AXI ポートの選択は、プラットフォーム作成者しだいです。このプラットフォームでは汎用マスター、コヒーレント ポート、およびプロセッシング システム IP ブロックの 4 つのハイ パフォーマンス ポートすべてが宣言されていますが、宣言する必要があるのは 1 つの汎用 AXI マスターと 1 つの AXI スレーブ ポートのみです。

```
sdsoc::pfm_axi_port    $pfm M_AXI_GP0 ps7 M_AXI_GP
sdsoc::pfm_axi_port    $pfm M_AXI_GP1 ps7 M_AXI_GP
sdsoc::pfm_axi_port    $pfm S_AXI_ACP ps7 S_AXI_ACP
sdsoc::pfm_axi_port    $pfm S_AXI_HP0 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP1 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP2 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP3 ps7 S_AXI_HP
```

5. 次のコマンドでダイレクト I/O をプロキシする stream\_fifo マスター AXI4-Stream バス インターフェイスを宣言します。

```
sdsoc::pfm_axis_port   $pfm M_AXIS stream_fifo M_AXIS
```

6. 次のコマンドで割り込み入力を宣言します。

```
for {set i 0} {$i < 16} {incr i} {
    sdsoc::pfm_irq      $pfm In$i irq_concat
}
```

- すべてのインターフェイスを宣言しておく、SDSoC プラットフォーム ユーティリティでプラットフォームが生成される際にハードウェア プラットフォームのメタデータ ファイル (zc702\_axis\_io/src/vivado/zc702\_axis\_io\_pfm.tcl) が生成されます。

```
sdsoc::generate_hw_pfm $pfm
```

次のコマンドでプラットフォーム ハードウェア記述ファイルを有効にします。

```
sds-pf-check <sdx_root>/samples/platforms/zc702_axis_io/hw/  
zc702_axis_io.hpfm
```

<sdx\_root>/samples/platforms/zc702\_axis\_io/hw/zc702\_axis\_io\_hw.pfm validates.」などのメッセージが表示されます。

## SDSoC プラットフォーム ソフトウェア ライブラリ

AXI4-Stream インターフェイスをエクスポートするすべてのプラットフォーム IP には、C 呼び出し可能ライブラリにパッケージされたハードウェア関数が必ず必要で、これがアプリケーションで呼び出されると、アクセラレータがエクスポートされたインターフェイスに接続されます。SDSoC の sdslib ユーティリティを使用すると、[ライブラリの作成](#)に示すように、プラットフォーム用にスタティックな C 呼び出し可能ライブラリを作成できます。

- プラットフォームのソース コードは、<sdx\_root>/samples/platforms/zc702\_axis\_io/src ディレクトリにあります。

Platform AXI4-Stream Data FIFO IP には、M\_AXIS ポートにアクセスするために C 呼び出し可能関数が必要です。この場合は、pf\_read\_stream になります。ハードウェア関数は、次のように pf\_read.cpp で定義されます。関数宣言は zc702\_axis\_io.h ファイルに含まれます。

```
void pf_read_stream(unsigned *rbuf) {}
```

関数の本体は空で、アプリケーションから呼び出されると、sdsc コンパイラによりスタブ関数の本体にデータを移動する適切なコードが挿入されます。関数の引数がすべて IP ポートに一貫してマップされている場合にのみ、複数の関数を 1 つの IP にマップできます。たとえば、サイズの異なる 2 つの配列引数は、対応する IP の 1 つの AXIS ポートにはマップできません。

- C 呼び出し可能インターフェイスの各関数では、関数引数から IP ポートへのマップを定義する必要があります。pf\_read\_stream IP のマップは、zc702\_axis\_io.fcnmap.xml に読み込まれます。

```
<xd:repository xmlns:xd="http://www.xilinx.com/xd">  
  <xd:fcnMap xd:fcnName="pf_read_stream"  
    xd:componentRef="zc702_axis_io">  
    <xd:arg  
      xd:name="rbuf"  
      xd:direction="out"  
      xd:busInterfaceRef="stream_fifo_M_AXIS"  
      xd:portInterfaceType="axis"  
      xd:dataWidth="32"  
    />  
  </xd:fcnMap>  
</xd:repository>
```

各関数引数には、名前、方向、IP バス インターフェイス名、インターフェイス タイプ、およびデータ幅が必要です。



**重要:** fcnMap はプラットフォーム関数 pf\_read\_stream をプラットフォーム コンポーネント zc702\_axis\_io のプラットフォーム バス インターフェイス stream\_fifo\_M\_AXIS に関連付けます。これは、関数をインプリメントするプラットフォーム内の IP のバス インターフェイスへの参照です。zc702\_axis\_io.hpfm の stream\_fifo\_M\_AXIS という名前のプラットフォーム バス インターフェイス (ポート) には、xd:instanceRef 属性内に IP へのマップが含まれています。

3. IP カスタマイズ パラメーターは、コンパイル時に XML ファイルで設定する必要があります。この例では、プラットフォーム IP にパラメーターが設定されていないので、zc702\_axis\_io.params.xml は特に単純です。別の例を参照する場合は、SDx インストール ディレクトリから <sdx\_root>/samples/fir\_lib/build/fir\_compiler.params.xml を開いてください。
4. src/linux ディレクトリには、次のコマンドでライブラリをビルドするための makefile が含まれます。

```
sdslib -lib libzc702_axis_io.a \
  pf_read_stream pf_read.cpp \
  -vlnv xilinx.com:ip:axis_data_fifo:1.1 \
  -ip-map zc702_axis_io.fcnmap.xml \
  -ip-params zc702_axis_io.params.xml
```

ライブラリ ファイルは、zc702\_axis\_io/sw/aarch32-linux/lib/libzc702\_axis\_io.a に含まれます。

## プラットフォームのサンプル デザイン

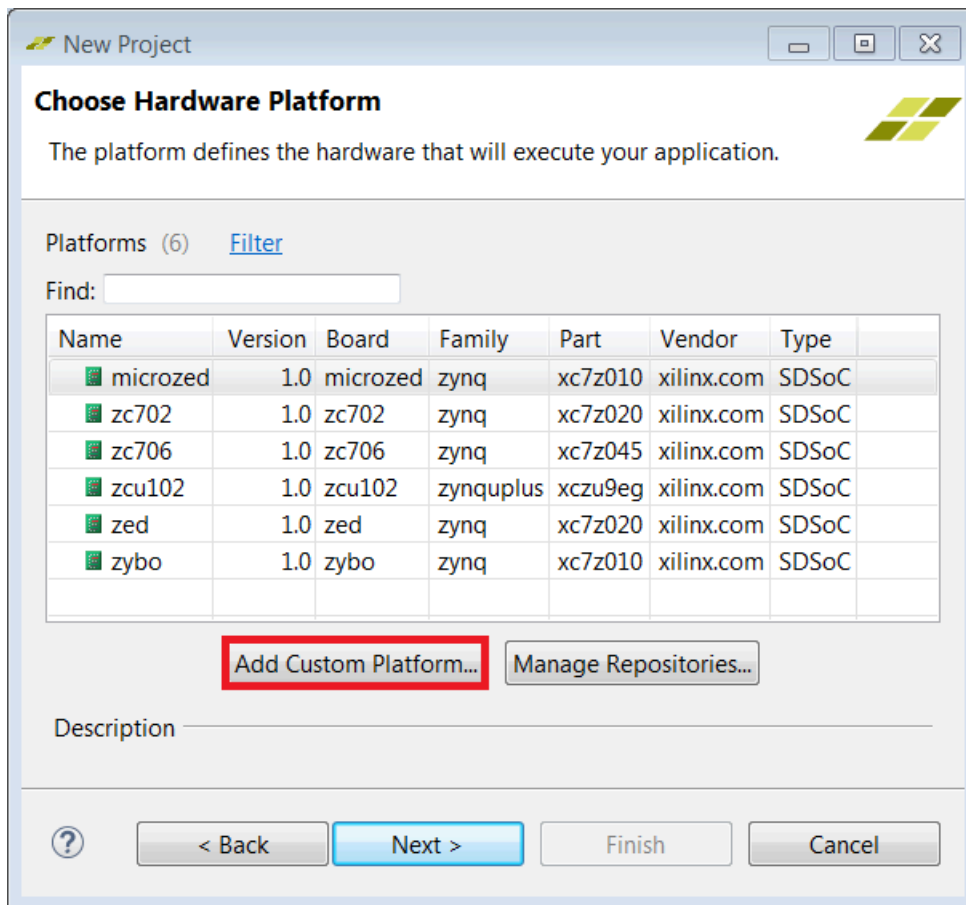
SDSoC プラットフォームには、その使用方法を示すサンプル アプリケーションを含めることができます。SDx IDE では、プラットフォーム内のサンプル ソース ファイルの場所に関する情報を取得するため、samples/template.xml というファイルが検索されます。zc702\_axis\_io プラットフォームのテンプレート ファイルには、それぞれ目的の違う複数のテスト アプリケーションがリストされています。

```
<template location="aximm" name="Unpacketized AXI4-Stream to DDR"
      description="Shows how to copy unpacketized AXI-Stream data
directly to DDR.">
  <supports>
    <and>
      <or>
        <os name="Linux"/>
        <os name="Standalone"/>
      </or>
    </and>
  </supports>
  <accelerator name="s2mm_data_copy" location="main.cpp"/>
</template>
<template location="stream" name="Packetize an AXI4-Stream"
      description="Shows how to packetize an unpacketized
AXI4-Stream.">
  <supports>
    <and>
      <or>
        <os name="Linux"/>
        <os name="Standalone"/>
      </or>
    </and>
  </supports>
  <accelerator name="packetize" location="packetize.cpp"/>
  <accelerator name="minmax" location="minmax.cpp"/>
</template>
<template location="pull_packet" name="Lossless data capture from
AXI4-Stream to DDR"
      description="Illustrates a technique to enable lossless data
capture from a free-running input source.">
  <supports>
    <and>
      <or>
        <os name="Linux"/>
        <os name="Standalone"/>
      </or>
    </and>
  </supports>
  <accelerator name="PullPacket" location="main.cpp"/>
</template>
```

SDx IDE でプラットフォームを使用するには、次の手順に従って、それを Eclipse ワークスペース用のプラットフォーム リポジトリに追加する必要があります。

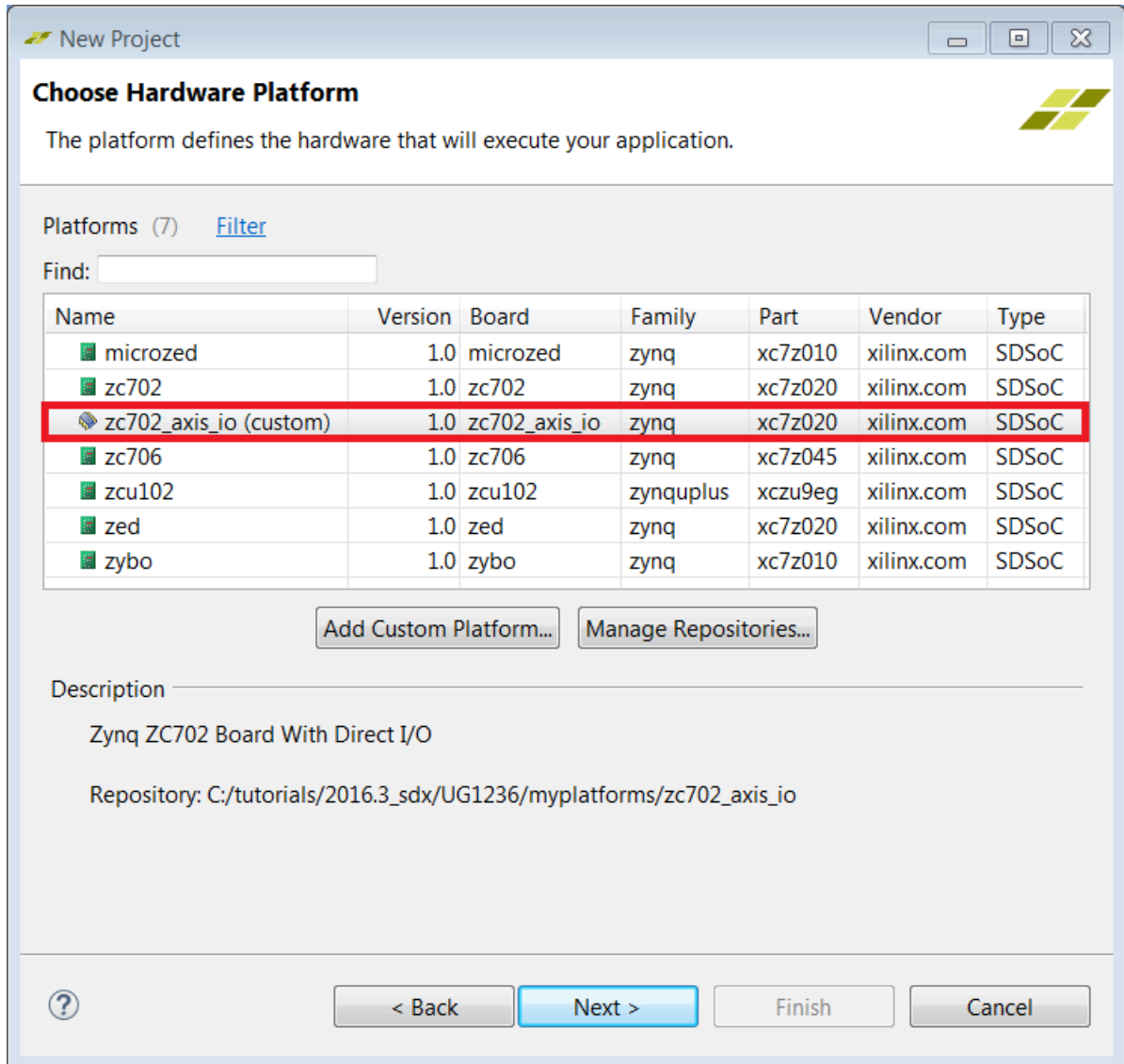
1. ザイリンクス SDx を起動して、<path\_to\_tutorial>/myplatforms/ のように、ワークスペースのパスを指定します。
2. [File] → [New] → [Xilinx SDx Project] をクリックし、新しいプロジェクトを作成します。

3. [Create New SDx Project] ページで `my_zc702_axis_io` のようにプロジェクト名を指定したら、[Next] をクリックします。
4. [Choose Hardware Platform] ページで [Add Custom Platform] をクリックします。



5. プラットフォーム `<sdx_root>/samples/platforms/zc702_axis_io` を選択します。

- このプラットフォームが [Choose Hardware Platform] ページに表示されます。zc702\_axis\_io (custom) を選択して [Next] をクリックします。



- [Choose Software Platform and Target CPU] ページの [System Configuration] をデフォルトの [Linux SMP (Zynq 7000)] にしたまま [Next] をクリックします。
- サンプル アプリケーションの 1 つを含むプラットフォームをテストする場合は、[Unpacketized AXI4-Stream to DDR] を選択して [Finish] をクリックします。s2mm\_data\_copy 関数がハードウェア用に前もって選択されています。s2mm\_data\_copy\_wrapper 内のプログラム データ フローにより、プラットフォーム入力から s2mm\_data\_copy というハードウェア関数までのダイレクト信号パスが作成され、zero\_copy データムーバーとしてメモリにデータが送信されます。つまり、s2mm\_data\_copy 関数はカスタム DMA として動作します。メイン プログラムは 4 つのバッファに割り当てられ、s2mm\_data\_copy\_wrapper が起動されてから、書き込まれたバッファがチェックされて、データ値がシーケンシャルになる (データがバブルなしで書き込まれる) ようになります。このプログラムでは簡単にするため、カウンタがリセットされないようになっているので、初期値はボードの電源投入時からプログラムの起動時までの時間によって変わります。

9. main.cpp を開きます。次の点を確認します。

- ・ zero\_copy データムーバーに必要な物理的に隣接した割り当てになるように、sds\_alloc を使用してバッファが割り当てられる方法

```
unsigned *bufs[NUM_BUFFERS];
bool error = false;
for(int i=0; i<NUM_BUFFERS; i++) {
    bufs[i] = (unsigned*) sds_alloc(BUF_SIZE * sizeof(unsigned));
}
// Flush the platform FIFO of start-up garbage
s2mm_data_copy_wrapper(bufs[0]);
for(int i=0; i<NUM_BUFFERS; i++) {
    s2mm_data_copy_wrapper(bufs[i]);
}
```

- ・ プラットフォーム入力から読み出すためにプラットフォームの関数が呼び出される方法

```
void copy_data_wrapper(unsigned int *buf)
void s2mm_data_copy_wrapper(unsigned *buf)
{
    unsigned rbuf0[1];
    pf_read_stream(rbuf0);
    s2mm_data_copy(rbuf0,buf);
}
```

10. ツールバーの [Build] アイコンをクリックしてアプリケーションをビルドします。ビルドが完了したら、Debug フォルダの sd\_card フォルダにブートイメージとアプリケーション ELF が含まれます。
11. ビルドが終了したら、sd\_card ディレクトリの内容を SD カードにコピーし、ブートして、my\_zc702\_axis\_io.elf を実行します。

```
sh-4.3# cd /mnt
sh-4.3# ./my_zc702_axis_io.elf
TEST PASSED!
sh-4.3#
```

## 例: プラットフォーム IP AXI ポートの共有

プラットフォーム IP、アクセラレータ、および SDSoC コンパイラで生成されたデータ モーション IP の間で AXI マスター (スレーブ) インターフェイスを共有するには、SDSoC Tcl API を使用して、共有インターフェイスに接続されている AXI Interconnect IP ブロック上のインデックス順で最初の未使用 AXI マスター (スレーブ) ポートを宣言します。プラットフォームでこの AXI Interconnect の各下位インデックスのマスター (スレーブ) を使用する必要があります。

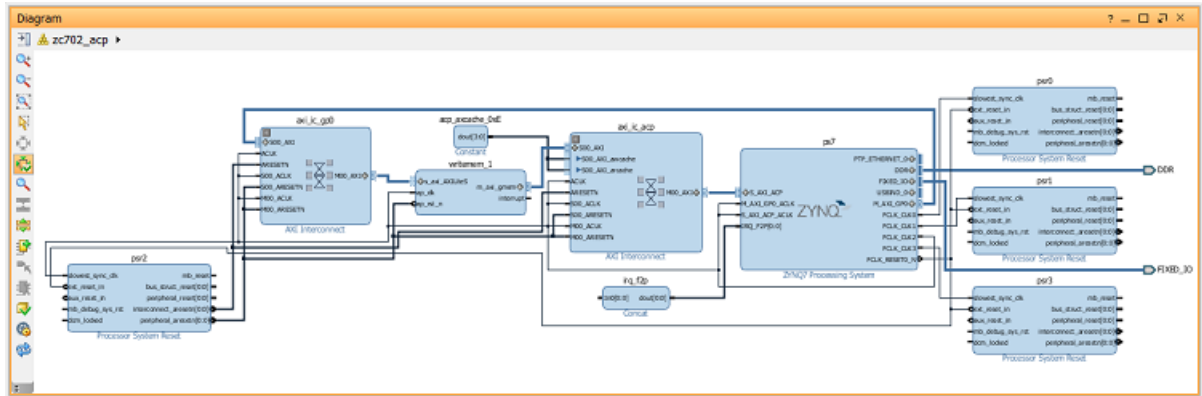


## SDSoC プラットフォーム ハードウェア記述

ハードウェア システムは Vivado プロジェクト <sdx\_root>/samples/sdspfm/zc702\_acp/src/vivado/zc702\_acp.xpr に含まれます。

1. ブロック図は、次のようになります。

図 22: ZC702\_acp ブロック図



2. Vivado での SDSoC Tcl コマンドに示すように、プラットフォーム ハードウェアのポート インターフェイスは Vivado プロジェクトで定義され、スクリプトファイル (src/vivado/zc702\_acp\_pfm.tcl) に含まれる Tcl コマンドを使用して SDSoC プラットフォーム ユーティリティで抽出されます。次のコマンドでハードウェア プラットフォーム オブジェクトを作成して、名前と簡単な説明を追加します。

```
set pfm [sdsoc::create_pfm zc702_acp.hpfm]
sdsoc::pfm_name $pfm "xilinx.com" "xd" "zc702_acp" "1.0"
sdsoc::pfm_description $pfm "Zynq XC702 platform with shared GP and ACP ports"
```

3. このコマンドでデフォルトのプラットフォーム クロックの ID を 2 に宣言します。true 引数は、このクロックがプラットフォームのデフォルトであることを示します。

```
sdsoc::pfm_clock $pfm FCLK_CLK2 ps7 2 true psr2
```



ヒント: このコマンドは、[クロックの宣言](#)に示すように、各プラットフォーム クロックごとに必要な関連する proc\_sys\_reset IP インスタンス (psr2) も指定しています。

4. 次のコマンドでプラットフォーム AXI インターフェイスを宣言します。

```
sdsoc::pfm_axi_port $pfm M_AXI_GP0 ps7 M_AXI_GP
sdsoc::pfm_axi_port $pfm S_AXI_HP0 ps7 S_AXI_HP
sdsoc::pfm_axi_port $pfm S_AXI_HP1 ps7 S_AXI_HP
sdsoc::pfm_axi_port $pfm S_AXI_HP2 ps7 S_AXI_HP
sdsoc::pfm_axi_port $pfm S_AXI_HP3 ps7 S_AXI_HP
for {set i 1} {$i < 64} {incr i} {
    sdsoc::pfm_axi_port $pfm M[format %02d $i]_AXI axi_ic_gp0 M_AXI_GP
}
for {set i 1} {$i < 8} {incr i} {
    sdsoc::pfm_axi_port $pfm S[format %02d $i]_AXI axi_ic_acp S_AXI_ACP
}
```



各 AXI ポートにメモリ タイプの宣言が必要です。M\_AXI\_GP (汎用 AXI マスター)、S\_AXI\_ACP (キャッシュ コヒーレント スレーブ インターフェイス)、S\_AXI\_HP (高パフォーマンス ポート)、または MIG (外部メモリ コントローラーへのインターフェイス) のいずれかを指定します。

Mxy\_AXI ( $y > 0$ ) ポートへの API 呼び出しを使用した for ループでは、プロセッシング システムの M\_AXI\_GP0 ポートに接続されたインターコネクトで使用するマスター ポートを宣言し、Sxy\_AXI ( $y > 0$ ) ポートへの API 呼び出しを使用したループでは、プロセッシング システムの S\_AXI\_ACP ポートに接続されたインターコネクトで使用するスレーブ ポートを宣言します。

上記の Vivado ブロック図で、各インターコネクトのインデックスが最下位のポートがプラットフォーム内で要件どおりに使用されていることを確認します。

5. 次のコマンドで割り込み入力を宣言します。

```
for {set i 0} {$i < 16} {incr i} {
    sdsoc::pfm_irq      $pfm In$i irq_f2p
}
```

6. 次のコマンドで、プラットフォームが SDSoC プラットフォーム ユーティリティで生成される際に、zc702\_acp.hpfm ハードウェア プラットフォームのメタデータ ファイルを作成します。

```
sdsoc::generate_hw_pfm $pfm
```

このプラットフォームが使用されると、sdsc コンパイラで M\_AXI\_GP0 および S\_AXI\_ACP ポートに接続されたプラットフォーム インターコネクトが必要に応じて展開され、プラットフォームと SDSoC アプリケーション ロジック間で CPU と DDR メモリ アクセスが共有されます。

# その他のリソースおよび法的通知

---

## ザイリンクス リソース

アンサー、資料、ダウンロード、フォーラムなどのサポート リソースは、[ザイリンクス サポート](#) サイトを参照してください。

---

## ソリューション センター

デバイス、ツール、IP のサポートについては、[ザイリンクス ソリューション センター](#)を参照してください。デザイン アシスタント、デザイン アドバイザリ、トラブルシューティングのヒントなどが含まれます。

---

## 参考資料

このガイドの補足情報は、次の資料を参照してください。

日本語版のバージョンは、英語版より古い場合があります。

1. 『SDx 環境リリース ノート、インストールおよびライセンス ガイド』(UG1238)
2. 『SDSoC 環境ユーザー ガイド』(UG1027)
3. 『SDSoC 環境最適化ガイド』(UG1235)
4. 『SDSoC 環境チュートリアル: 入門』(UG1028)
5. 『SDSoC 環境プラットフォーム開発ガイド』(UG1146)
6. [SDSoC 開発環境ウェブ ページ](#)
7. 『UltraFast エンベデッド デザイン設計手法ガイド』(UG1046: [英語版](#)、[日本語版](#))
8. 『ZC702 評価ボード (Zynq-7000 XC7Z020 All Programmable SoC 用) ユーザー ガイド』(UG850)
9. 『Vivado Design Suite ユーザー ガイド: 高位合成』(UG902)
10. 『PetaLinux ツール資料: ワークフロー チュートリアル』(UG1156)
11. [Vivado® Design Suite の資料](#)
12. 『Vivado Design Suite ユーザー ガイド: カスタム IP の作成とパッケージ』(UG1118)

## お読みください: 重要な法的通知

本通知に基づいて貴殿または貴社（本通知の被通知者が個人の場合には「貴殿」、法人その他の団体の場合には「貴社」。以下同じ）に開示される情報（以下「本情報」といいます）は、ザイリンクスの製品を選択および使用することのためにのみ提供されます。適用される法律が許容する最大限の範囲で、(1) 本情報は「現状有姿」、およびすべて受領者の責任で (with all faults) という状態で提供され、ザイリンクスは、本通知をもって、明示、黙示、法定を問わず（商品性、非侵害、特定目的適合性の保証を含みますがこれらに限られません）、すべての保証および条件を負わない（否認する）ものとします。また、(2) ザイリンクスは、本情報（貴殿または貴社による本情報の使用を含む）に関係し、起因し、関連する、いかなる種類・性質の損失または損害についても、責任を負わない（契約上、不法行為上（過失の場合を含む）、その他のいかなる責任の法理によるかを問わない）ものとし、当該損失または損害には、直接、間接、特別、付随的、結果的な損失または損害（第三者が起こした行為の結果被った、データ、利益、業務上の信用の損失、その他あらゆる種類の損失や損害を含みます）が含まれるものとし、それは、たとえ当該損害や損失が合理的に予見可能であったり、ザイリンクスがそれらの可能性について助言を受けていた場合であったとしても同様です。ザイリンクスは、本情報に含まれるいかなる誤りも訂正する義務を負わず、本情報または製品仕様のアップデートを貴殿または貴社に知らせる義務も負いません。事前の書面による同意のない限り、貴殿または貴社は本情報を再生産、変更、頒布、または公に展示してはなりません。一定の製品は、ザイリンクスの限定的保証の諸条件に従うこととなるので、<https://japan.xilinx.com/legal.htm#tos> で見られるザイリンクスの販売条件を参照してください。IP コアは、ザイリンクスが貴殿または貴社に付与したライセンスに含まれる保証と補助的条件に従うこととなります。ザイリンクスの製品は、フェイルセーフとして、または、フェイルセーフの動作を要求するアプリケーションに使用するために、設計されたり意図されたりしていません。そのような重大なアプリケーションにザイリンクスの製品を使用する場合のリスクと責任は、貴殿または貴社が単独で負うものです。<https://japan.xilinx.com/legal.htm#tos> で見られるザイリンクスの販売条件を参照してください。

## 自動車用のアプリケーションの免責条項

オートモーティブ製品（製品番号に「XA」が含まれる）は、ISO 26262 自動車用機能安全規格に従った安全コンセプトまたは余剰性の機能（「セーフティ設計」）がない限り、エアバッグの展開における使用または車両の制御に影響するアプリケーション（「セーフティ アプリケーション」）における使用は保証されていません。顧客は、製品を組み込むすべてのシステムについて、その使用前または提供前に安全を目的として十分なテストを行うものとします。セーフティ設計なしにセーフティ アプリケーションで製品を使用するリスクはすべて顧客が負い、製品の責任の制限を規定する適用法令および規則にのみ従うものとします。

© Copyright 2017 Xilinx, Inc. Xilinx, Xilinx のロゴ、Artix、ISE、Kintex、Spartan、Virtex、Vivado、Zynq、およびこの文書に含まれるその他の指定されたブランドは、米国およびその他の各国のザイリンクス社の商標です。OpenCL および OpenCL のロゴは Apple Inc. の商標であり、Khronos による許可を受けて使用されています。PCI、PCIe、および PCI Express は PCI-SIG の商標であり、ライセンスに基づいて使用されています。すべてのその他の商標は、それぞれの所有者に帰属します。

この資料に関するフィードバックおよびリンクなどの問題につきましては、[jpn\\_trans\\_feedback@xilinx.com](mailto:jpn_trans_feedback@xilinx.com) まで、または各ページの右下にある [フィードバック送信] ボタンをクリックすると表示されるフォームからお知らせください。フィードバックは日本語で入力可能です。いただきましたご意見を参考に早急に対応させていただきます。なお、このメール アドレスへのお問い合わせは受け付けておりません。あらかじめご了承ください。