SDSoC 環境ユーザー ガイド

プラットフォームおよびライブラリ

UG1146 (v2015.4) 2015 年 12 月 14 日

本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。

XILINX
ALL PROGRAMMABLE.

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
2015 年 12 月 14 日	2015.4	・ ハードウェア プラットフォーム ファイル生成用の Tcl API を追加 ・ サンプル アプリケーション (template.xml) をアップデート
2015 年 9 月 30 日	2015.2.1	・ ZC702 スタンドアロン ボードのサポートを追加 ・ ZC702 チュートリアルをアップデート ・ プラットフォーム ダイレクト I/O チュートリアルをアップデート
2015 年 7 月 20 日	2015.2	初版

目次

第 1 章

概要

SDSoC™ (Software-Defined Development Environment for System-on-Chip) 環境は、Zynq®-7000 All Programmable SoC 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。SDSoC システム コンパイラは、C/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。SDSoC 環境には、アプリケーション開発用の多数のプラットフォームおよびザイリンクス パートナーから提供されるプラットフォームが含まれています。

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。SDSoC 環境で作成するプロジェクトはすべて特定のプラットフォームをターゲットとし、SDSoC 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アクセラレータをプラットフォームに接続するデータ モーション ネットワークでカスタマイズします。この方法を使用すると、さまざまなベース プラットフォーム向け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さまざまな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。

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 システムからカスタムの SDSoC プラットフォームを作成する方法および OS カーネル、ブート ローダー、ファイル システム、およびライブラリ、などのソフトウェア ランタイム環境について説明します。

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。

第 2 章

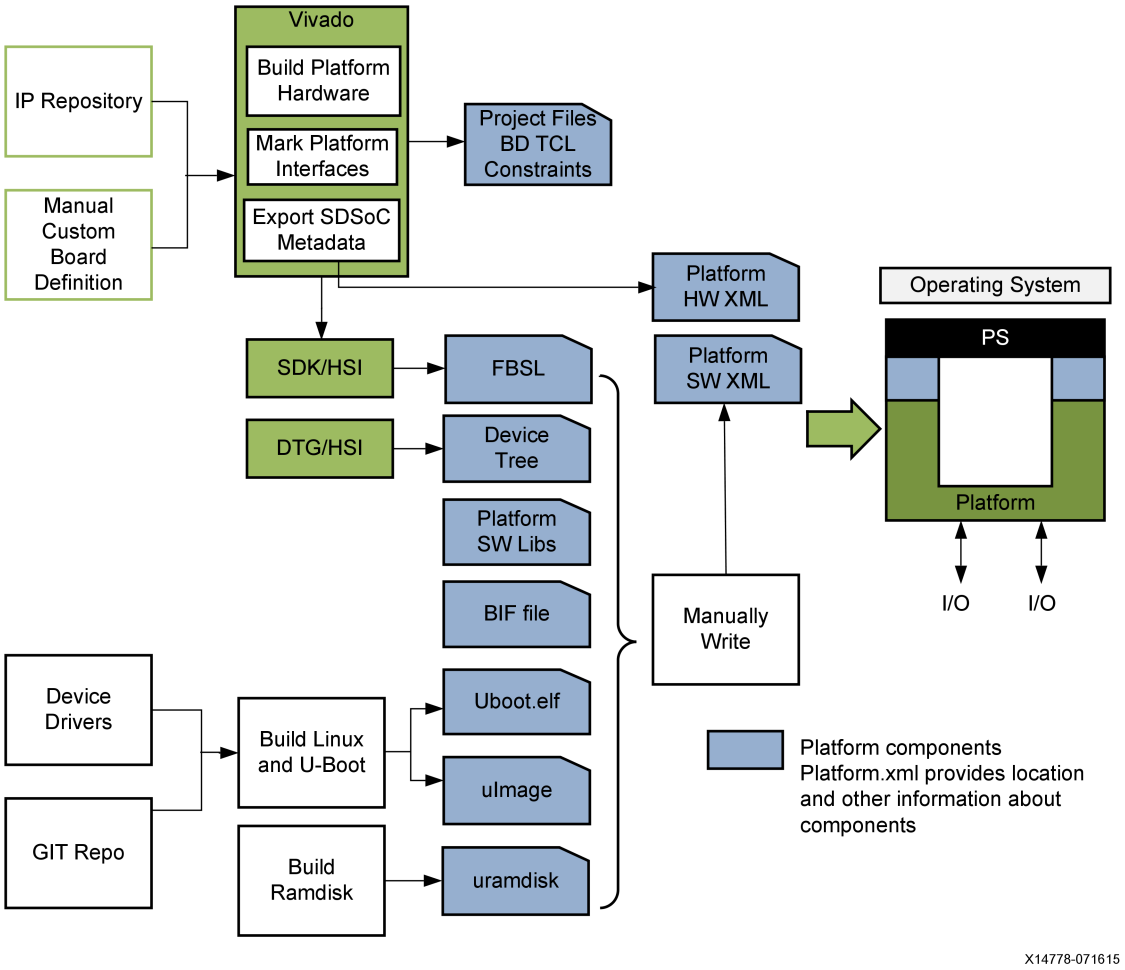
SDSoC プラットフォーム

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 XML メタデータ ファイルも含まれます。

プラットフォーム プロバイダーにより、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 ハードウェアがビルドされます。ハードウェアがビルドされ、検証されると、プラットフォーム プロバイダーにより Vivado ツール内で Tcl コマンドが実行され、SDSoC プラットフォーム ハードウェア インターフェイスが指定されて、SDSoC プラットフォーム ハードウェア メタデータ ファイルが生成されます。

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の Linux オペレーティング システムがサポートされる場合は、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、SDSoC 環境、またはザイリンクス SDK を使用できます。現時点では、ソフトウェア プラットフォームのメタデータ ファイルは手動で作成する必要があります。

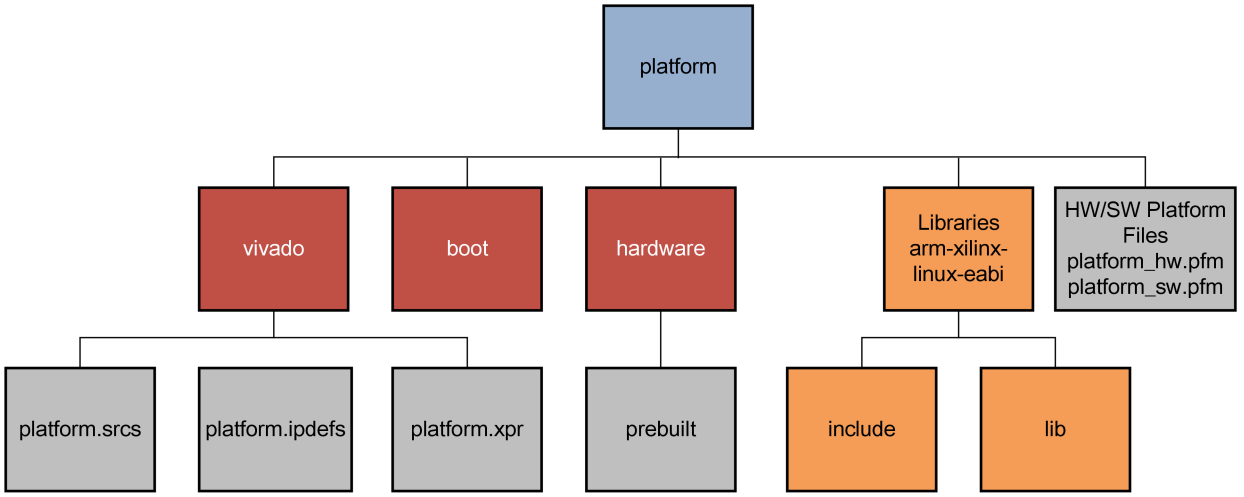
図 2-1 : SDSoC プラットフォームの主なコンポーネント



SDSoC プラットフォームには、次が含まれます。

- メタデータ ファイル
  - Vivado ツールを使用して生成されたプラットフォーム ハードウェア記述 (`<platform>_hw.pfm`)
  - 手動で記述したプラットフォーム ソフトウェア記述ファイル (`<platform>_sw.pfm`)
- Vivado Design Suite プロジェクト
  - ソース
  - 制約
  - 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)
  - Linux 関連オブジェクト (デバイス ツリー、U-Boot、Linux カーネル、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
  - 前もって生成されたデバイス登録およびポート情報ソフトウェア ファイル
  - 前もって生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 ファイル

図 2-2 : 典型的な SDSoC プラットフォームのディレクトリ構造



X14784-070915

通常は、プラットフォーム ビルダーだけでプラットフォームが SDSoC 環境内で正しく使用されるようにできますが、`<sdsoc_root>/docs/SDSoC_platform_checklist.xlsx` のプラットフォーム チェックリストも参照できます。

このチェックリストには、SDSoC システム コンパイラで使用された各データ ムーバーの基本的なテストを含む platform_dm_test.zip ファイルが含まれています。platform_dm_test.zip を作業エリアで解凍し、SDSoC 開発のターミナル シェルから次を実行します。

```
$ make PLATFORM=<platform_path> axidma_simple
$ make PLATFORM=<platform_path> axidma_sg
$ make PLATFORM=<platform_path> axidma_2d
$ make PLATFORM=<platform_path> axififo
$ make PLATFORM=<platform_path> zero_copy
$ make PLATFORM=<platform_path> xd_adapter
```

これらの各テストは正しくビルドされ、ボードでテストされるはずです。

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る必要があります。

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について説明します。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 SoC 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 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 API を読み込みます。
3. Vivado Tcl コンソールで Tcl API を実行し、次の手順を実行します。
   a. ハードウェア プラットフォーム名を宣言します。
   b. プラットフォームの簡単な説明を宣言します。
   c. プラットフォーム クロック ポートを宣言します。
   d. 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。
   e. 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。
   f. 使用可能なプラットフォーム割り込みを宣言します。
   g. プラットフォーム ハードウェア記述のメタデータ ファイルを生成します。

プラットフォーム ハードウェア デザインでは、次の規則に必ず従ってください。

1. Vivado プロジェクトと IP インテグレーター ブロック図の名前はプラットフォーム名と同じにし、ブロック図のハードウェア ラッパーの名前を `<platform_name>_wrapper.v` にする必要があります。たとえばプラットフォーム `zc702` の場合、Vivado プロジェクトは `zc702.xpr`、ブロック図は `zc702.bd` (`zc702.srcs/sources_1/bd/zc702/zc702.bd`)、ハードウェア ラッパーは `zc702_wrapper.v` とします。
2. 標準の Vivado IP カタログに含まれていないプラットフォーム IP は、プラットフォーム Vivado Design Suite プロジェクトに含まれている必要があります。外部 IP リポジトリ パスを参照することはできません。
3. いずれのプラットフォームでも Vivado IP カタログに含まれている Processing System IP ブロックを含める必要があります。
4. SDSoC プラットフォームのハードウェア 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プラットフォーム内に含める必要があります。
5. 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 Interconnect 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。
6. 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。
7. SDSoC 環境とプラットフォーム ロジック (たとえば `S_AXI_ACP`) 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を使用する必要があります。
8. 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。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。
9. 各プラットフォーム AXI4-Stream インターフェイスには、SDSoC 環境で使用される Vivado ツールのデータ ムーバー IP に準拠するために `TLAST` および `TKEEP` 側帯波信号が必要です。
10. 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。
11. プラットフォームの割り込み入力は、Processing System 7 IP `IRQ_F2P` に接続されている Concat (`xlconcat`) ブロック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。

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。

SDSoC 環境では、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はそれらすべてをサポートする必要はありません。

プラットフォーム ペリフェラルに Linux カーネル ドライバーが必要な場合は、drivers/staging/apf の linux-xlnx カーネル ソースを使用して、使用可能な SDSoC 環境特有のドライバー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。SDSoC 環境に含まれるベース プラットフォームには、たとえば platforms/zc702/boot/how-to-build-this-linux-kernel.txt のような手順が含まれています。

```
This linux kernel (uImage) and the associated device tree (devicetree.dtb) are based on
the 3.19 version of the linux kernel
To build the kernel:

Clone/pull from the master branch of the Xilinx/linux-xlnx tree at github,
and checkout the xilinx-v2015.2.03 tag
 git checkout -b sdsoc_release_tag xilinx-v2015.2.03

Add the following CONFIGs to xilinx_zynq_defconfig and then configure the kernel
CONFIG_STAGING=y
CONFIG_XILINX_APF=y
CONFIG_XILINX_DMA_APF=y
CONFIG_DMA_CMA=y
CONFIG_CMA=y
CONFIG_CMA_SIZE_MBYTES=256
CONFIG_LOCALVERSION="-xilinx-apf"
# The following configs are optional, and remove some debug settings
CONFIG_PRINTK_TIME=n
CONFIG_LOCKUP_DETECTOR=n
CONFIG_DEBUG_RT_MUTEXES=n
CONFIG_DEBUG_WW_MUTEX_SLOWPATH=n
CONFIG_PROVE_LOCKING=n
CONFIG_DEBUG_ATOMIC_SLEEP=n
CONFIG_PROVE_RCU=n
CONFIG_DMA_API_DEBUG=n
One way to do this is to
 cp arch/arm/configs/xilinx_zynq_defconfig arch/arm/configs/tmp_defconfig
Edit arch/arm/configs/tmp_defconfig using a text editor and add the above
config lines to the bottom of the file
 make ARCH=arm tmp_defconfig

Build the kernel using
make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- 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場合は、まずハードウェア コンポーネントをビルドする必要があります。ハードウェア コンポーネントをビルドしたら、Vivado® ハードウェア エクスポート コマンドを実行してハードウェア ハンドオフ ファイルを作成します。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、SDSoC 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、第 4 章の「スタンドアロン プラットフォーム ソフトウェアの作成」を参照してください。プラットフォームは、「プラットフォーム ソフトウェア記述ファイル」に示されている手順を使用して、ライブラリおよびヘッダー ファイルをまとめることができます。

メタデータ ファイル

SDSoC プラットフォームには次の XML メタデータ ファイルが含まれており、これらのファイルにはハードウェアおよびソフトウェア インターフェイスが記述されています。

- プラットフォーム ハードウェア記述ファイル
- プラットフォーム ソフトウェア記述ファイル

プラットフォーム ハードウェア記述ファイル

プラットフォーム ハードウェア記述ファイル `<platform>_hw.pfm` は SDSoC 環境用にハードウェア システムを記述した XML メタデータ ファイルで、使用可能なクロック周波数、割り込み、SDSoC 環境でハードウェア関数との通信に使用できるハードウェア インターフェイスを含みます。

「SDSoC プラットフォーム」の図に示すように、Vivado Design Suite を使用してベース ハードウェア プラットフォーム デザインを構築することによりこのファイルを作成します。Vivado Tcl API を使用して、SDSoC プラットフォーム ポート インターフェイスを宣言し、SDSoC プラットフォーム ハードウェア記述を生成します。この資料に含まれているチュートリアルでは、これらの Vivado Tcl API の使用方法の例を紹介しています。

SDSoC Vivado Tcl コマンド

このセクションでは、クロック情報およびクロック、リセット、割り込み、AXI、AXI4-Stream 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 Vivado® IP インテグレーターの Tcl コマンドについて説明します。Vivado Design Suite でハードウェア システムをビルドして検証したら、SDSoC プラットフォーム ハードウェア記述ファイルを作成する手順は次のとおりです。

1. Tcl コンソールで次のコマンドを実行して SDSoC Vivado Tcl API を読み込みます。

```
source -notrace <sdsoc_root>/scripts/vivado/sdsoc_pfm.tcl
```

2. Vivado で Tcl API を実行し、次の手順を実行します。
   a. ハードウェア プラットフォーム名を宣言します。
   b. プラットフォームの簡単な説明を宣言します。
   c. プラットフォーム クロック ポートを宣言します。
   d. 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。
   e. 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。
   f. 使用可能なプラットフォーム割り込みを宣言します。
   g. プラットフォーム ハードウェア記述のメタデータ ファイルを生成します。

次に、ブロック図内で使用される Tcl API を説明します。

ハードウェア プラットフォーム記述の作成

新しいハードウェア PFM 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>_hw.pfm
```

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_hw.pfm]

sdsoc::pfm_name        $pfm "xilinx.com" "xd" "zc702" "1.0"

sdsoc::pfm_description $pfm "Zynq ZC702 Board"
```

クロック

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、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> <port> <instance> <id> <is_default> <proc_sys_reset>
```

引数 :

```
pfm            - pfm handle

port           - clock port name

instance       - instance name of the block that contains the port

id             - clock id (user-defined, must be unique non-negative

                 integer)

is_default     - true if this is the default clock, false otherwise

proc_sys_reset - corresponding proc_sys_reset block instance for

                 synchronized reset signals
```

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

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 :

```
pfm      - pfm handle

port     - axi port name

instance - instance name of the block that contains the port

memport  - corresponding memory interface port type

             (values: M_AXI_GP - a general purpose AXI master port

                      S_AXI_HP - a high-performance AXI slave port

                      S_AXI_ACP - an Accelerator Coherent slave port

                      MIG - an AXI slave connected to a MIG memory controller)
```

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
```

AXI4-Stream ポート

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 :

```
pfm      - pfm handle

port     - axi stream port name

instance - instance name of the block that contains the port

type     - interface type (values: 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

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 :

```
pfm      - pfm handle

port     - irq port name

instance - instance name of the concat block that contains the port
```

例 :

```
for {set i 0} {$i < 16} {incr i} {

  sdsoc::pfm_irq       $pfm In$i xlconcat

}
```

I/O デバイス

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 :

```
pfm      - pfm handle

port     - io port name

instance - instance name of the block that contains the uio

type     - io device type (e.g. uio, kio)
```

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>
```

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、`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
```

プラットフォーム ソフトウェア記述ファイル

「SDSoC プラットフォーム」で説明されているように、SDSoC プラットフォームにはソフトウェア コンポーネントが含まれており、このコンポーネントにはオペレーティング システム、ブート ローダー、およびライブラリが含まれています。プラットフォーム ソフトウェア記述ファイルには、SDSoC システム コンパイラがプラットフォーム上にビルドされるアプリケーション特定のシステム オン チップを生成するのに必要なソフトウェア ランタイムに関するメタデータが含まれています。

ブート ファイル

SDSoC 環境では、ボードをブートして Linux オペレーティング システムをブートするかまたはスタンドアロン プログラムを実行する SD カードを作成できます。

次のフォーマットを使用して Linux のファイルを記述します。カーネル イメージ、デバイス ツリー、およびルート ファイル システムを含むユニファイド ブート イメージの `.ub` ファイルを使用している場合は、`xd:devicetree` および `xd:ramdisk` を削除して `xd:linuxImage="boot/image.ub"` を指定します。オプションの `xd:sdcard` フォルダーには、SD カード イメージのルートに追加されるフォルダーおよびファイルが含まれています。オプションの `xd:sdcardMountPath` は、SD カードのマウント パスを指定します。指定しない場合のデフォルトは `/mnt` です。次の例では、PetaLinux のマウント パス `/media/card` を指定しています。

```
<xd:bootFiles
 xd:os="linux"
 xd:bif="boot/linux.bif"
 xd:readme="boot/generic.readme"
 xd:devicetree="boot/devicetree.dtb"
 xd:linuxImage="boot/uImage"
 xd:ramdisk="boot/ramdisk.image.gz"
 xd:sdcard="boot/sdcard"
 xd:sdcardMountPath="/media/card"/>
```

OS を使用しないスタンドアロン プログラムでは、次を記述します。

```
<xd:bootFiles
 xd:os="standalone"
 xd:bif="boot/standalone.bif"
 xd:readme="boot/generic.readme"
 xd:sdcard="boot/sdcard"
/>
```

注記 : これらのエレメントではブート イメージ ファイル (BIF) が参照されることに注意してください。指定ディレクトリに BIF ファイルが含まれている必要があります。

Linux ターゲットのサンプル プラットフォームの BIF ファイル テンプレートには、次が含まれています。

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

システム生成中、SDSoC システム コンパイラでは、このテンプレートを読み込んで、アプリケーション特有のファイル名を挿入して BIF ファイルが生成されます。このファイルが bootgen ユーティリティに渡されてブート イメージが作成されます。

```
/* linux */
the_ROM_image:
 {
    [bootloader]<path_to_platform>/boot/fsbl.elf
    <path_to_generated_bitstream>/<project_name>.elf.bit
    <path_to_platform>/boot/u-boot.elf
}
```

standalone.bif サンプル ファイルには、次の内容が含まれます。

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

システム生成中、SDSoC システム コンパイラでは、このテンプレートを読み込んで、アプリケーション特有のファイル名を挿入して BIF ファイルが生成されます。このファイルが bootgen ユーティリティに渡されてブート イメージが作成されます。

```
/* standalone */
the_ROM_image:
 {
     [bootloader]<path_to_platform>/boot/fsbl.elf
     <path_to_generated_bitstream_directory>/<project_name>.elf.bin
     <path_to_generated_application_elf_directory>/<project_name>.elf
 }
```

ライブラリ ファイル

プラットフォームには、オプションでライブラリを含めることができます。次のフォーマットを使用してライブラリ ファイルを記述する場合は、コンパイラの呼び出し時に SDSoC 環境で該当するインクルード パスとライブラリ パスが -I および -L オプションを使用して自動的に追加されます。

```
 <xd:libraryFiles
  xd:os="linux"
  xd:includeDir="arm-xilinx-linux-gnueabi/include"
  xd:libDir="arm-xilinx-linux-gnueabi/lib"/>
<xd:libraryFiles
 xd:os="standalone"
 xd:includeDir="arm-xilinx-eabi/include"
 xd:libDir="arm-xilinx-eabi/lib"
 xd:bspconfig="arm-xilinx-eabi/system.mss"
 xd:bsprepo="arm-xilinx-eabi/bsprepo"/>
```

説明

次に、xd:libraryFiles の情報スキーマを示します。

```
<xd:libraryFiles
 xd:os            Operating system. Valid values: linux, standalone
 xd:includeDir    Directory passed to compiler using -I.
                  Separate multiple paths with a colon ':' character.
                  When sdscc/sds++ compiles source files, the include path
                  order is: (1) user paths, (2) platform paths, (3) SDSoC install paths
                  and (4) Vivado HLS paths (if required).
 xd:libDir        Directory paths passed to the linker using -L.
                  Separate multiple paths with a colon ':' character. Each path
                  must be a directory within the platform directory, containing libraries
                  that can be linked with the user application.  The library path
                  link order is: (1) user specified paths, (2) path to SDSoC-generated
                  BSP (standalone/FreeRTOS only), (3) platform paths, (4) SDSoC install
                  paths, and (5) SDSoC-generated project.  Do not place standalone
                  BSP library libxil.a, which sdscc/sds++ generates for a BSP configuration
                  file in a directory on the xd:libDir path.
 xd:libName       Library names passed to the linker using -l.
                  Separate multiple library names with a colon ':' character.
                  When specified, sdscc/sds++ automatically adds the -l option
                  when linking the ELF.
 xd:bspconfig     BSP configuration file (.mss) for standalone/FreeRTOS. When specified,
                  the platform must also specify an xd:includeDir containing BSP header files.
                  sdscc/sds++ uses this .mss instead of generating a default BSP
                  configuration file based on the both the platform and hardware
                  in the PL.  Consequently, the .mss file must specify
                  drivers required in SDSoC user designs, including the Xilinx
                  AXI DMA driver (scatter-gather mode). See Generating
                  Basic Software Platforms (UG1138) for information about
                  BSP configuration files (.mss).
 xd:bsprepo       BSP repository folder. When specified,
                  xd:bspconfig must also be specified.  sdscc/sds++ adds this folder
                  to the BSP repository search path used to create a standalone BSP.
                  Refer to Generating
                  Basic Software Platforms (UG1138) for information
                  about BSP repositories.
/>
```

ビルド済みハードウェア ファイル

プラットフォームには、オプションでビルド済みハードウェア ファイルを含めることができます。アプリケーションにハードウェア関数が含まれない場合、SDSoC 環境では、ビットストリームとブート イメージがビルドし直される代わりにこれらがプロジェクトにコピーされます。これにより、ターゲットでアプリケーション ソフトウェアを実行するためのコンパイルが高速に実行できます。プラットフォームにビルド済みハードウェア ファイルが含まれる場合、sdscc-rebuild-hardware オプションを使用してビットストリーム コンパイルを強制的に実行すると、ハードウェア ファイルが作成し直すことができます。

次の例は、ZC702 プラットフォームに含まれるビルド済みハードウェアを示しています。

```
<xd:hardware
 xd:system="prebuilt"
 xd:bitstream="prebuilt/bitstream.bit"
 xd:export="prebuilt/export"
 xd:hwcf="prebuilt/hwcf"
 xd:swcf="prebuilt/swcf"/>
```

説明

次に、xd:hardware の情報スキーマを示します。

```
<xd:hardware
 xd:system          Identifier associated with predefined hardware; when
                    the SDSoC environment searches for a pre-built bitstream, it looks
                    for the keyword "prebuilt"
 xd:bitstream       Path to the bitstream.bit file for the pre-built hardware
 xd:export          Path to the folder containing SDK-compatible files
                    created using the Vivado tools export_hardware command.
                    This folder contains the hardware handoff file <platform>.hdf,
                    for example, zc702.hdf.
 xd:hwcf            Path to the folder containing hardware system
                    information files. Files found in this folder
                    are partitions.xml and apsys_0.xml.
 xd:swcf            Path to the folder containing device registration and
                    port information files. Files found in this folder
                    are devreg.c, devreg.h, portinfo.c and portinfo.h.
 />
```

ビルド済みプラットフォーム ファイルは SDSoC システム コンパイラで Hello world プログラムをビルドすることで作成できます。

すべてのベース プラットフォームの例は、<sdsoc_install_directory>/platforms/*/hardware/prebuilt に含まれています。

プラットフォーム ハードウェア記述ファイルのテスト

SDSoC 環境にはプラットフォーム ハードウェア記述ファイルを検証するための XML スキーマが含まれています。たとえば、SDSoC でプラットフォーム ハードウェア記述 XML ファイルを検証するには、次のコマンドを実行します。

```
sds-pf-check <platform>_hw.pfm
```

ハードウェア プラットフォーム記述ファイル (`<platform>_hw.pfm`) とソフトウェア プラットフォーム記述ファイル (`<platform>_sw.pfm`) をプラットフォーム ディレクトリに含めた後に、すべての使用可能なプラットフォームをリストする次のコマンドを実行して、SDSoC で正しくファイルを読み込むことができるかを検証できます。表示されたリストに作成したプラットフォームが表示されていれば、SDSoC 環境で正しく読み込むことができたことを意味します。

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

プラットフォームに関する詳細を表示するには、次のコマンドを実行します。

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

Vivado Design Suite プロジェクト

SDSoC™ 環境では、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 プロジェクトを開始点として使用します。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 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えますが、「ハードウェア要件」に示すような制約もいくつかあります。

ファイル名とディレクトリ : `platforms/<platform>/vivado/<platform>.xpr`

例 : `platforms/zc702/vivado/zc702.xpr`

注記 : プロジェクトは、`xpr` ファイルと同じディレクトリに含める必要があります。

重要 : ファイルは Vivado プロジェクトで単純にコピーすることはできません。Vivado ツールでは、単純なファイル コピーでは保持されないような方式で内部ステートが管理されているからです。プロジェクトのコピーを作成するには、Vivado 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 ファイルをハードウェア プラットフォームの含まれる SDSoC のプラットフォーム ディレクトリで解凍します。

Vivado ツールでは、新しい Vivado Design Suite の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 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、新しいツール バージョンでプロジェクトを開いてから、すべての IP をアップグレードします。プロジェクトをアーカイブして、このアーカイブを SDSoC プラットフォーム ハードウェア プロジェクトで解凍します。

SDSoC 環境で 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、プラットフォームをコピーできなかったことを意味します。

ライブラリ ヘッダー ファイル

プラットフォームにプラットフォーム特定のヘッダー ファイルを含めるためのアプリケーション コード `#include` が必要な場合、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 `xd:includeDir` 属性で指定されたプラットフォーム ディレクトリの下位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 xd:includeDir="<relative_include_path>" とある場合、ディレクトリは次になります。

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

例 :

xd:includeDir="arm-xilinx-linux-gnueabi/include" の場合、次のディレクトリになります。

```
<sdsoc_root>/samples/platforms/zc702_axis_io/arm-xilinx-linux-gnueabi/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、次を使用します。

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

複数のインクルード パスを区切るためには、コロン (:) を使用します。次はその例です。

```
xd:includeDir="<relative_include_path1>:<relative_include_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 ファイルは、標準ディレクトリには含まれないので、ユーザーが SDSoC 環境のコンパイル コマンドに -I オプションを使用してそれらを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

プラットフォームに含まれるスタティック ライブラリに対してユーザーがリンクする必要がある場合、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 xd:libDir 属性で指定されたプラットフォーム ディレクトリの下位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 xd:libDir="<relative_lib_path>" とある場合、ディレクトリは次になります。

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

例 :

xd:libDir="arm-xilinx-linux-gnueabi/lib" の場合、次のディレクトリになります。

```
<sdsoc_root>/samples/platforms/zc702_axis_io/arm-xilinx-linux-gnueabi/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 パスを区切るためには、コロン (:) を使用します。次はその例です。

```
xd:libDir="<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>/<relative_lib_path1>
<platform root>/<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 コマンドに -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。

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 ディレクトリの下位ディレクトリに含める必要があります。下位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に対して xd:bitstream、xd:export、xd:hwcf、および xd:swcf 属性で指定します。

プラットフォーム XML に xd:bitsteam="<relative_lib_path>/bitstream.bit" とある場合、ディレクトリは次になります。

```
platforms/<platform>/<relative_lib_path>/bitstream.bit
```

プラットフォーム XML に xd:export="<relative_export_path>" とある場合、ディレクトリは次になります。

```
platforms/<platform>/<relative_export_path>
```

プラットフォーム XML に xd:hwcf="<relative_hwcf_path>" とある場合、ディレクトリは次になります。

```
platforms/<platform>/<relative_hwcf_path>
```

プラットフォーム XML に xd:swcf="<relative_swcf_path>" とある場合、ディレクトリは次になります。

```
platforms/<platform>/<relative_swcf_path>
```

例 :

xd:bitstream="prebuilt/bitstream.bit" の場合は、次のディレクトリになります。

```
platforms/zc702/hardware/prebuilt/bitstream.bit
```

xd:export="prebuilt/export" の場合は、次のディレクトリになります。

```
platforms/zc702/hardware/prebuilt/export
```

これには zc702.hdf が含まれます。

xd:hwcf="prebuilt/hwcf" の場合は、次のディレクトリになります。

```
platforms/zc702/hardware/prebuilt/hwcf
```

これには partitions.xml および apsys_0.xml が含まれます。

xd:swcf="prebuilt/swcf" の場合は、次のディレクトリになります。

```
platforms/zc702/hardware/prebuilt/swcf
```

これには devreg.c、devreg.h、portinfo.c および portinfo.h 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フラグを使用したハードウェア関数が含まれない場合、SDSoC 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>/hardware/prebuilt フォルダーを生成するために使用したファイルは、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。
- export

  _sds/p0/ipi/<platform>.sdk (<platform>.hdf) に含まれます。
- hwcf

  _sds/.llvm (partitions.xml, apsys_0.xml) に含まれます。
- swcf

  _sds/swstubs (devreg.c、devreg.h、portinfo.c、portinfo.h) に含まれます。

Linux ブート ファイル

SDSoC™ 環境では、ボードをブートして Linux オペレーティング システムをブートする SD カードを作成できます。ブートが完了すると、Linux プロンプが開き、コンパイルしたアプリケーションを実行できます。このために、SDSoC 環境では、プラットフォームの一部として次のようなオプションジェクトが必要です。

- 「FSBL (First Stage Boot Loader)」
- 「U-Boot」
- 「デバイス ツリー」
- 「Linux イメージ」
- 「ramdisk イメージ」

SDSoC 環境ではザイリンクスの bootgen ユーティリティ プログラムが使用され、ビットストリームと共に必要なファイルが 1 つの BOOT.BIN ファイルにまとめられます。このファイルは 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。

重要 : 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。

FSBL (First Stage Boot Loader)

FSBL では、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) がコンフィギュレーションされます。

Vivado® Design Suite でプラットフォーム プロジェクトが開いたら、[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] をクリックして `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、SDSoC 環境フロー用の標準ディレクトリにコピーしておく必要があります。

SDSoC システム コンパイラで FSBL が使用されるようにするには、BIF ファイルで指定する必要があります (「ブート ファイル」を参照してください)。ファイルは、`<platform_root>/boot/fsbl.elf` フォルダーに含まれる必要があります。

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

例 :

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

U-Boot

Das U-Boot は、オープン ソースのブートローダーです。wiki.xilinx.com の手順に従って 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。

SDSoC 環境で U-Boot が使用されるようにするには、BIF ファイルで指定する必要があります (「ブート ファイル」を参照してください)。ファイルは、`<platform_root>/boot/fsbl.elf` フォルダーに含まれる必要があります。

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

例 : `samples/platforms/zc702_axis_io/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。wiki.xilinx.com のデバイス ツリーに関する手順に従って、デバイス ツリー ジェネレーター サポート ファイルをダウンロードし、ザイリンクス SDK で使用できるようにインストールします。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。

ファイル名およびファイルのディレクトリは、プラットフォーム XML で定義されています。xd:devicetree 属性を xd:bootFiles エレメントで使用します。カーネルを含むユニファイド ブート イメージ (.ub ファイル) を使用する場合、デバイス ツリーおよびルート ファイル システムでは xd:devicetree 属性は定義されません。

XML 記述サンプル :

```
xd:devicetree="boot/devicetree.dtb"
```

ディレクトリ : samples/platforms/zc702_axis_io/boot/devicetree.dtb

注記 : プラットフォームで processing_system7 FCLK_CLK ポートから供給されるクロックがエクスポートされる場合、Vivado 内部ロジック アナライザー IP コアを使用したハードウェア デバッグがサポートされるよう PetaLinux で生成される標準のデバイス ツリーをブート時に変更してこのクロックをイネーブルにする必要があります。次のコマンドを使用して FCLK_CLK クロックをイネーブルにするようプラットフォームでデバイス ツリーを変更します。

```
fclk-enable = <0xf>;
```

Linux イメージ

Linux イメージは起動に必要なイメージです。ザイリンクスからは、プラットフォームに依存しない SDSoC プラットフォームすべてで動作するビルド済みの Linux イメージが 1 つ提供されています。

ただし、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には、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。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、SDSoC 環境の 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。SDSoC プラットフォーム用に Linux カーネルをビルドする手順は、<sdsoc_root>/<platform>/boot/how-to-build-this-linux-kernel.txt に記述されています。

ファイル名およびファイルのディレクトリは、プラットフォーム XML で定義されています。xd:linuxImage 属性を xd:bootFiles エレメントで使用します。カーネル、デバイス ツリー、ルートファイルを含むユニファイド ブート イメージ (.ub ファイル) を使用する場合、xd:linuxImage 属性を定義して、xd:linuxImage="boot/image.ub" など .ub ファイルのディレクトリを指定します。

XML 記述サンプル :

```
xd:linuxImage="boot/uImage"
```

ディレクトリ : samples/platforms/zc702_axis_io/boot/uImage

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 SDSoC 環境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 に記載されている手順に従います。

ファイル名およびファイルのディレクトリは、プラットフォーム XML で定義されています。xd:ramdisk 属性を xd:bootFiles エレメントで使用してください。カーネル、デバイス ツリーおよびルート ファイル システムを含むユニファイド ブート イメージ (.ub ファイル) を使用する場合は、xd:ramdisk 属性は定義しないでください。

XML 記述サンプル :

```
xd:ramdisk="boot/uramdisk.image.gz"
```

ディレクトリ : samples/platforms/zc702_axis_io/boot/uramdisk.image.gz

Petalinux を使用した Linux ブート ファイルの作成

PetaLinux を使用すると、Linux ブート ファイルすべてを生成できます。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。PetaLinux を使用した場合も全体的なワークフローは同じですが、SDSoC 環境を使用して Linux ブート ファイルを生成するには、追加の手順を実行する必要があります。このため、SDSoC 環境で使用できるようにコンフィギュレーションされた ZC702 用の BSP をあらかじめ提供しています。

プラットフォームのクロック ソースに `processing_system7` IP ブロックの `FCLK_CLK` ポートが含まれている場合は、1 つ前のセクションに従いデバイス ツリーを変更する必要があります。

SDSoC 環境で使用できる ZC702 用の PetaLinux イメージを構築する場合は、次の手順に従ってください。

1. 提供されている BSP を使用してPetaLinux プロジェクトを新規作成します。

```
$ petalinux-create -t project /path/to/Xilinx-ZC702-SDSoC-2015.2.1.bsp
```

2. プロジェクトを構築します。

```
$ petalinux-build
```

3. 生成したカーネルおよび rootfs を U-Boot ヘッダーを使用してラップします。

```
$ petalinux-package --image -c kernel --format uImage
```

4. デバイス ツリー BLOB および ramdisk の名前を変更します。

```
$ mv images/linux/system.dtb images/linux/devicetree.dtb
$ mv images/linux/urootfs.cpio.gz images/linux/uramdisk.image.gz
```

5. 出力ファイルは `./images/linux` ディレクトリに含められており、SD カードにコピーできます。

SDSoC のインストールで提供されている ZC702 BSP には、PetaLinux で提供されているデフォルトの BSP に次の手順に従った変更が加えられています。

1. `<project-root>/subsystems/linux/config` ファイルを開いて、次のコードを含めます。

```
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-xilinx-linux-gnueabi-"
CONFIG_LOCALVERSION="-xilinx-apf"
CONFIG_PRINTK_TIME=n
CONFIG_DEBUG_KERNEL=n
CONFIG_HAVE_DEBUG_KMEMLEAK=n
CONFIG_LOCKUP_DETECTOR=n
CONFIG_DEBUG_RT_MUTEXES=n
CONFIG_DEBUG_WW_MUTEX_SLOWPATH=n
CONFIG_PROVE_LOCKING=n
CONFIG_DEBUG_ATOMIC_SLEEP=n
CONFIG_PROVE_RCU=n
CONFIG_DMA_API_DEBUG=n
```

2. `<project-root>/subsystems/linux/configs/device-tree/system-top.dts` ファイルを開いて、次のコードを含めます。

```
&clkc {
   fclk-enable = <0xf>;
};
/ {
   xlnk {
      compatible = "xlnx,xlnk-1.0";
      clock-names = "xclk0", "xclk1", "xclk2", "xclk3";
      clocks = <&clkc 15>, <&clkc 16>, <&clkc 17>, <&clkc 18>;
   };
};
```

3. `petalinux-config -c rootfs` を実行して、menuconfig システムを起動します。
4. [Filesystem Packages] をクリックします。
5. [base] をクリックします。
   a. [external-xilinx-toolchain] → [libstdc++6] をクリックします。
   b. [tcf-agent] → [tcf-agent] をクリックします。
6. `petalinux-build` コマンドを実行してプロジェクトをビルドし、`<project-root>/images/linux` フォルダーに `image.ub` というファイルを生成します。これには、カーネル、デバイス ツリー、ファイル システムなどがパッケージされます。

スタンドアロン ブート ファイル

OS が不要な場合は、エンドユーザーが生成し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 では、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) がコンフィギュレーションされます。

Vivado® Design Suite でプラットフォーム プロジェクトが開いたら、[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] をクリックして `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、SDSoC 環境フロー用の標準ディレクトリにコピーしておく必要があります。

SDSoC システム コンパイラで FSBL が使用されるようにするには、BIF ファイルで指定する必要があります (「ブート ファイル」を参照してください)。ファイルは、<platform_root>/boot/fsbl.elf フォルダーに含まれる必要があります。

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

例 :

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

実行ファイル

SDSoC 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 ファイルで設定する必要があります (「ブート ファイル」を参照)。

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

SDSoC 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。

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 テンプレート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 ディレクトリに含める必要があります。このアプリケーションを記述するファイルは template.xml と呼ばれ、samples ディレクトリに含まれます。

template.xml ファイルのフォーマットは非常に単純です。次に、zc702_led 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="arraycopy" name="Array copy"
            description="Simple test application">
    <supports>
      <and>
        <os name="Linux"/>
      </and>
```

```
      </supports>
    </template>
    <template location="arraycopy_sa" name="Array copy"
              description="Simple test application">
      <supports>
        <and>
          <os name="Standalone"/>
        </and>
      </supports>
    </template>
  </manifest:Manifest>
```

最初の行は、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、すべてのアプリケーション テンプレートのコンテナーとして必要です。

```
<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

`<template>` 要素には、複数の属性を含めることができます。

表 2-1 : template 要素

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="myapp" name="My App" description="Sample application">
```

`<template>` 要素には、複数の XML サブ要素も含めることができます。

表 2-2 : template 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

supports 要素

<supports> 要素は、選択した SDSoC 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、コンパイラに -I フラグを使用して渡す、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、リンカーに -L フラグを使用して渡す、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 属性を指定します。

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、SDSoC 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>
```

template.xml の例

次に、完全な template.xml ファイルの例を示します。

```
<?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="myapp" name="My App"
            description="Sample application">
    <supports>
      <and>
        <or>
          <os name="Linux"/>
          <os name="Standalone"/>
          <os name="FreeRTOS"/>
        </or>
      </and>
    </supports>
    <includepaths>
      <path location="myinclude"/>
      <path location="dir/include"/>
    </includepaths>
    <libraries>
      <lib name="mylib1"/>
      <lib name="mylib2"/>
    </libraries>
    <exclude>
      <directory name="MyDir"/>
      <directory name="MyOtherDir"/>
      <file name="MyFile.txt"/>
      <file name="MyOtherFile.txt:/>
    </exclude>
  </template>
  <!-- Multiple template elements allowed -->
</manifest:Manifest>
```

FreeRTOS コンフィギュレーション/バージョン変更

SDSoC™ 環境における FreeRTOS サポートでは、v8.2.1 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 v8.2.1 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>/SDSoC/<version>/platforms/zc702` フォルダーをローカルのフォルダーにコピーします。

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/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. FeeRTOS 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/freertos/include フォルダーに追加します。
   c. ライブラリ libfreertos.a を <path_to_your_platform>/zc702/freertos/li に追加します。
   d. <path_to_your_platform>/zc702/zc702_sw.pfm ファイルの ("xd:os="freertos" (xd:includeDir="freertos/include" and xd:libDir="freertos/lib") 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。

   SDSoC 環境のフォルダー <path_to_install>/SDSoC/2015.4/tps/FreeRTOS には、コンフィギュレーション済み FreeRTOS v8.2.1 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、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. SDSoC環境の <path_to_install>/SDSoC/2015.4/settings64 スクリプトを実行して、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. フォルダーをローカル フォルダーにコピーします。
   d. FreeRTOSConfig.h を変更します。
   e. make コマンドを実行します。

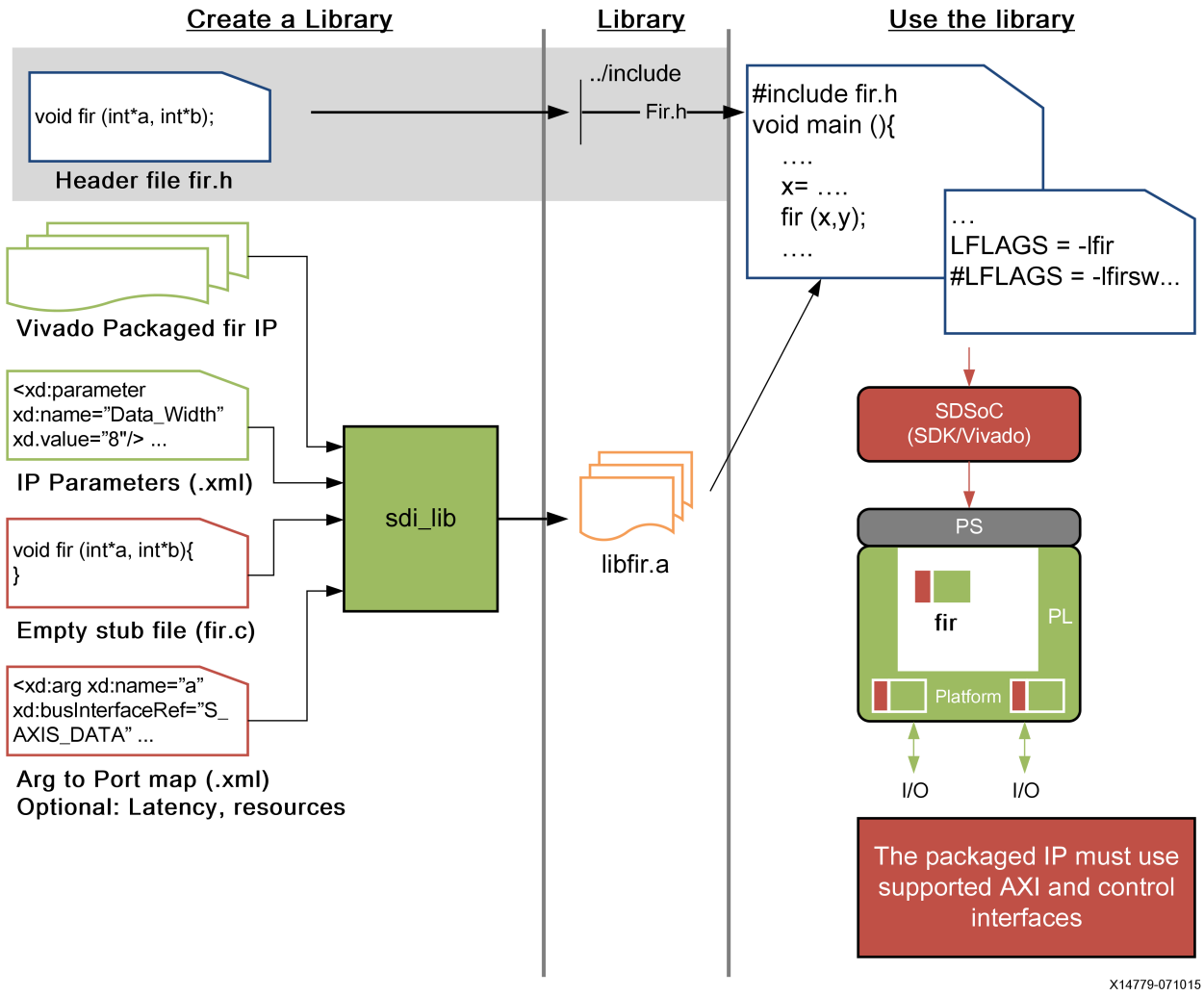
FreeRTOS v8.2.1 を使用していない場合は、SDSoC_readme.txt ファイルでソースがオフィシャル ソフトウェア ディストリビューションからどのように作成されたかを示す注記を確認してください。ZIP の解凍後、少数の変更 (デモ アプリケーション main.c に含まれる memcpy、memset、および memcmp をライブラリ ソース ファイルへ追加し、Task.h から task.h へのインクルード ファイル リファレンスを変更) が加わっていますが、フォルダー構造は元のままです。フォルダー構造を維持すると、コンフィギュレーション済み FreeRTOS v8.2.1 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。

第 3 章

C 呼び出し可能ライブラリ

このセクションでは、VHDL や Verilog のようなハードウェア記述言語で記述された IP ブロックの C 呼び出し可能ライブラリを作成する方法について説明します。ユーザー アプリケーションは、SDSoC システム コンパイラーを使用してこのようなライブラリとスタティックにリンクでき、IP ブロックは生成されたハードウェア システムにインスタンシエートされます。C 呼び出し可能ライブラリでは、sdscc でコンパイルされたアプリケーションによるプラットフォームに含まれる IP ブロックへのアクセスも提供されます (「例 : SDSoC プラットフォームのダイレクト I/O」を参照)。

図 3-1 : C 呼び出し可能ライブラリの作成および使用



次に、SDSoC プラットフォームのソフトウェア呼び出し可能ライブラリに含まれているエレメントを示します。

- 「ヘッダー ファイル」

  関数プロトタイプ
- 「スタティック ライブラリ」
  - 関数定義
  - IP コア
  - IP コンフィギュレーション パラメーター
  - 関数引数マップ

ヘッダー ファイル

ライブラリでは、ユーザー アプリケーションのソース ファイルに含めることができるヘッダー ファイルの IP にマップされる関数プロトタイプを宣言する必要があります。これらの関数で、ソフトウェア アプリケーション コードを介して IP にアクセスするための関数呼び出しインターフェイスが定義されます。

例 :

```
 // FILE: fir.h
#define N 256
void fir(signed char X[N], short Y[N]);
```

スタティック ライブラリ

SDSoC 環境のスタティック ライブラリには、プログラマブル リソース上でソフトウェア関数を実行できるようにするエレメントが複数含まれています。

関数定義

関数インターフェイスでは、ライブラリへのエントリ ポイントが関数または関数のセットとして定義されます。これをユーザー コードで呼び出して IP をターゲットにすることができます。関数定義には、空の関数ボディを含めることができます。SDSoC では、これらが API 呼び出しに置き換えられ、IP ブロック間とのデータ転送が実行されます。これらの呼び出しのインプリメンテーションは、SDSoC システム コンパイラーで作成されたデータ モーション ネットワークによって異なります。

例 :

```
// FILE: fir.c
#include "fir.h"
#include <stdlib.h>
#include <stdio.h>
void fir(signed char X[N], short Y[N])
{
    // SDSoC replaces function body with API calls for data transfer
}
```

注記 : ライブラリへリンクするアプリケーション コードでは、#include stdlib.h および stdio.h も使用する必要があります。これらは、SDSoC システム コンパイラーで生成されたスタブの API 呼び出しに必要です。

IP コア

C 呼び出し可能ライブラリの HDL IP コアは、Vivado® ツールを使用してパッケージする必要があります。この IP コアは、Vivado ツールの IP リポジトリまたは別のディレクトリに含めることができます。ライブラリが使用されるときに該当する IP コアがハードウェア システムにインスタンシエートされます。

『Vivado Design Suite ユーザー ガイド : IP を使用した設計』(UG896) に説明されているように、Vivado Design Suite で IP をパッケージする必要があります。Vivado IP パッケージャー ツールでは、HDL、その他のソース ファイル、および IEEE-1685 IP-XACT 規格に準拠した IP 定義ファイル (`component.xml`) のディレクトリ構造が作成されます。また、パッケージャーでは Vivado Design Suite で必要なディレクトリおよびそのディレクトリに含まれるファイルを含んだアーカイブ ZIPファイルも作成されます。

IP では AXI4、AXI4-Lite、および AXI4 Stream インターフェイスをエクスポートできます。IP 制御レジスタは、アドレス オフセット `0x0` に配置して、次の仕様に準拠させる必要があります。この準拠は、Vivado HLS で生成される IP のネイティブ `axilite` 制御インターフェイスに一致しています。

制御信号は通常は簡単に判別できます。`ap_start` 信号で IP 実行が開始され、`ap_done` 信号で IP のタスクの完了が示され、`ap_ready` 信号で IP を開始できることが示されます。`ap_ctrl_hs` 定義の詳細は、Vivado の高位合成に関する資料を参照してください。

```
// 0x00 : Control signals
// bit 0 - ap_start (Read/Write/COH)
// bit 1 - ap_done (Read/COR)
// bit 2 - ap_idle (Read)
// bit 3 - ap_ready (Read)
// bit 7 - auto_restart (Read/Write)
// others - reserved
// (COR = Clear on Read, COH = Clear on Handshake)
```

重要 : HDL IP を Vivado Design Suite に統合する方法の詳細は、『Vivado Design Suite ユーザー ガイド : カスタム IP の作成およびパッケージ』(UG1118) を参照してください。

IP コンフィギュレーション パラメーター

HDL IP コアのほとんどは、合成時にカスタマイズできます。カスタマイズは、IP コアの動作を定義する IP パラメーターで設定できます。SDSoC 環境ではコアが生成されたシステムにインスタンシエートされるときにこの情報が使用されます。この情報は XML ファイルに含まれています。

`xd:component` 名は `spirit:component` 名と同じで、`xd:parameter` 名はそれぞれ IP のパラメーター名にする必要があります。IP インテグレーターに含まれるパラメーター名を表示するには、ブロックを右クリックして [Edit IP Meta Data] をクリックし、[IP Customization Parameters] を表示します。

例 :

```
 <!-- FILE: fir.params.xml --
><?xml version="1.0" encoding="UTF-8"?>
<xd:component xmlns:xd="http://www.xilinx.com/xd" xd:name="fir_compiler">
<xd:parameter xd:name="DATA_Has_TLAST" xd:value="Packet_Framing"/>
<xd:parameter xd:name="M_DATA_Has_TREADY" xd:value="true"/>
<xd:parameter xd:name="Coefficient_Width" xd:value="8"/>
<xd:parameter xd:name="Data_Width" xd:value="8"/>
<xd:parameter xd:name="Quantization" xd:value="Integer_Coefficients"/>
<xd:parameter xd:name="Output_Rounding_Mode" xd:value="Full_Precision"/>
<xd:parameter xd:name="CoefficientVector"
 xd:value="6,0,-4,-3,5,6,-6,-13,7,44,64,44,7,-13,-6,6,5,-3,-4,0,6"/></xd:component>
```

関数引数マップ

SDSoC システム コンパイラには、ライブラリの関数プロトタイプから、関数をインプリメントする IP ブロックで定義されているハードウェア インターフェイスへのマップが必要です。この情報は、関数マップ XML ファイルに含まれます。

次の情報が含まれます。

- 関数名 : コンポーネントにマップされる関数の名前
- コンポーネント リファレンス : IP-XACT VNLV (Vendor-Name-Library-Version) ID からの IP タイプの名前

  関数がプラットフォームに関連付けられている場合は、コンポーネント リファレンスはプラットフォーム名です。「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。
- C 引数名 : 関数引数のアドレス表現 (例 : `x` (スカラーを値で渡す) または `*p` (ポインターで渡す))

  注記 : 関数マップの引数名は関数定義の引数と一致しており、同じ順序である必要があります。
- 関数引数の方向 : `in` (関数への入力引数) または `out` (関数への出力引数)。SDSoC 環境では現在のところ `inout` 関数引数はサポートされません。
- バス インターフェイス : 関数引数に対応する IP ポートの名前。プラットフォーム コンポーネントでは、この名前はプラットフォーム インターフェイス `xd:name` であり、対応するプラットフォーム IP の実際のポート名ではありません。
- ポート インターフェイス タイプ : 対応する IP ポート インターフェイス タイプ。現在のところ、`aximm` (スレーブのみ) または `axis` である必要があります。
- アドレス オフセット : `aximm` スレーブ ポートにマップされる引数に必要な 16 進数アドレス (0x40 など)
- データ幅 : データごとのビット数
- 配列のサイズ : 配列引数のエレメント数

次に、samples/fir_lib/build からの Vivado FIR Filter Compiler IP コンフィギュレーション用の関数マップを示します。

```
<!-- FILE: fir.fcnmap.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<xd:repository xmlns:xd="http://www.xilinx.com/xd">
    <xd:fcnMap xd:fcnName="fir" xd:componentRef="fir_compiler">
        <xd:arg xd:name="X"
            xd:direction="in"
            xd:portInterfaceType="axis"
            xd:dataWidth="8"
            xd:busInterfaceRef="S_AXIS_DATA"
            xd:arraySize="32"/>
        <xd:arg xd:name="Y"
            xd:direction="out"
            xd:portInterfaceType="axis"
            xd:dataWidth="16"
            xd:busInterfaceRef="M_AXIS_DATA"
            xd:arraySize="32"/>
        <xd:latencyEstimates xd:worst-case="20" xd:average-case="20" xd:best-case="20"/>
        <xd:resourceEstimates xd:BRAM="0" xd:DSP="1 xd:FF="200" xd:LUT="200"/>
    </xd:fcnMap>
</xd:repository>
```

ライブラリの作成

ザイリンクスでは、SDSoC ライブラリを作成するための sdslib というユーティリティを提供しています。

使用方法

```
sdslib [arguments] [options]
```

引数 (必須)

引数	説明
-lib <libname>	作成または追加するライブラリの名前
<function_name file_name>+	関数名とファイル名のペアを 1 つ以上指定します。 例 : fir fir.c
-vlnv <v>:<l>:<n>:<v>	使用する IP コアを VLNV で指定します。たとえば「-vlnv xilinx.com:ip:fir_compiler:7.1」のように指定します。
-ip-map <file>	IP 関数マップとして使用するファイルを指定します。
-ip-params <file>	IP パラメーターとして使用するファイルを指定します。

オプション	説明
-ip-repo <path>	HDL IP リポジトリ検索パスを追加します。
-os <name>	ターゲット オペレーティング システムを指定します。 · linux (デフォルト) · standalone (ベアメタル)
--help	ヘルプ情報を表示します。

たとえば fir filter IP コア用の SDSoC ライブラリを作成するには、次のコードを使用します。

```
> sdslib -lib libfir.a \
    fir fir.c \
    fir_reload fir_reload.c \
    fir_config fir_config.c \
    -vlnv xilinx.com:ip:fir_compiler:7.1 \
    -ip-map fir_compiler.fcnmap.xml \
    -ip-params fir_compiler.params.xml
```

この例では、sdslib により fir.c ファイルに含まれる fir 関数、fir_reload.c ファイルに含まれる fir_reload 関数、および fir_config.c ファイルに含まれる fir_config 関数が libfir.a スタティック ライブラリにアーカイブされます。fir_compiler IP コアは -vlnv を使用して指定され、関数マップは -ip-map、IP パラメーターは -ip-params を使用して指定されています。

ライブラリのテスト

ライブラリをテストするには、そのライブラリを使用するプログラムを作成します。ソース コードに該当するヘッダー ファイルを含めます。ライブラリ関数を呼び出すコードをコンパイルするときに、-I オプションを使用してヘッダー ファイルへのパスを含めます。

```
> sdscc -c -I<path to header> -o main.o main.c
```

ライブラリにリンクするには、-L および -l オプションを使用します。

```
> sdscc -sds-pf zc702 ${OBJECTS} -L<path to library> -lfir -o
 fir.elf
```

上記の例では、コンパイラにより <path to library> に含まれるライブラリ libfir.a が使用されます。SDSoC IDE でライブラリを使用する方法については、『SDSoC 環境ユーザー ガイド』(UG1027) の「C 呼び出し可能な IP ライブラリの使用」も参照してください。

C 呼び出し可能ライブラリの例 : Vivado FIR Compiler IP

SDSoC 環境インストールの samples/fir_lib/build ディレクトリには、ライブラリをビルドする方法を示す例が含まれています。この例では、Vivado® Design Suite で FIR Compiler IP のシングル チャネルのリロード可能なフィルター コンフィギュレーションが使用されています。IP のデザインと一致するよう、すべての通信および制御は AXI4-Stream チャネルを介して実行します。

SDSoC 環境インストールの samples/fir_lib/build ディレクトリには、ライブラリを使用する方法を示す例が含まれています。SDSoC IDE でライブラリを使用する方法については、『SDSoC 環境ユーザー ガイド』(UG1027) の「C 呼び出し可能な IP ライブラリの使用」も参照してください。

C 呼び出し可能ライブラリの例 : HDL IP

samples/rtl_lib/arraycopy/build ディレクトリには、Vivado ツールでパッケージされた RTL IP の例が含まれています。この例には、2 個の IP コアが含まれており、それぞれのコアで配列の M エレメントが入力から出力にコピーされます。ここでの M は、スカラー値を意味し、この値は関数呼び出しごとに異なる可能性があります。

- arraycopy_aximm : IP の AXI マスター インターフェイスを使用して配列が転送されます。
- arraycopy_axis : AXI4-Stream インターフェイスを使用して配列が転送されます。

次に IP のレジスタ マップを示します。

```
// arraycopy_aximm
// 0x00 : Control signals
// bit 0 - ap_start (Read/Write/COH)
// bit 1 - ap_done (Read/COR)
// bit 2 - ap_idle (Read)
// bit 3 - ap_ready (Read)
// bit 7 - auto_restart (Read/Write)
// others - reserved
// 0x10 : Data signal of ap_return
// bit 31~0 - ap_return[31:0] (Read)
// 0x18 : Data signal of a
// bit 31~0 - a[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of b
// bit 31~0 - b[31:0] (Read/Write)
// 0x24 : reserved
// 0x28 : Data signal of M
// bit 31~0 - M[31:0] (Read/Write)
// 0x2c : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

// arraycopy_axis
// 0x00 : Control signals
// bit 0 - ap_start (Read/Write/COH)
// bit 1 - ap_done (Read/COR)
// bit 2 - ap_idle (Read)
// bit 3 - ap_ready (Read)
// bit 7 - auto_restart (Read/Write)
// others - reserved
// 0x10 : Data signal of ap_return
// bit 31~0 - ap_return[31:0] (Read)
// 0x18 : Data signal of M
// bit 31~0 - M[31:0] (Read/Write)
// 0x1c : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)
```

makefile では `stdlib` を使用してライブラリを作成する方法が示されます。ライブラリをビルドするには、SDSoC IDE でターミナル シェルを開いてビルド ディレクトリから次のコマンドを実行します。

- `make librtl_arraycopy.a` : Linux アプリケーションのライブラリをビルドします。
- `make standalone/lib_rtl_arraycopy.a` : スタンドアロン アプリケーションのライブラリをビルドします。

`samples/rtl_lib/arraycopy/use` ディレクトリには、両方の IP を使用した単純なテスト例が含まれています。SDSoC ターミナル シェルで `make` コマンドを実行し、両方のハードウェア関数を実行する Linux アプリケーションを作成してください。

SDSoC IDE でライブラリを使用する方法については、『SDSoC 環境ユーザー ガイド』(UG1027) の「C 呼び出し可能な IP ライブラリの使用」も参照してください。

第 4 章

チュートリアル：SDSoC プラットフォームの作成

このチュートリアルでは、SDSoC™ 環境で Vivado® Design Suite を使用してビルドされたハードウェア システムから単純なサンプル プラットフォームを作成します。

Vivado ツールでビルドされたデザインのコネクティビティ インターフェイス (AXI および AXI4-Stream、クロック、リセット、および割り込みポートを含む) は、プラットフォーム ハードウェア記述で定義します。このコネクティビティ インターフェイスは、次の手順に従って、「SDSoC Vivado Tcl コマンド」で説明される Tcl API のセットを使用して Vivado Tcl コンソール内から宣言します。

1. Vivado Design Suite 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. Vivado Design Suite GUI でハードウェア プロジェクトを開きます。このプロセスは、スクリプト化することもできます。
3. SDSoC Vivado Tcl API を読み込みます。
4. Vivado で Tcl API を実行し、次の手順を実行します。
   a. ハードウェア プラットフォーム名を宣言します。
   b. プラットフォームの簡単な説明を宣言します。
   c. プラットフォーム クロック ポートを宣言します。
   d. 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。
   e. 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。
   f. 使用可能なプラットフォーム割り込みを宣言します。
   g. プラットフォーム ハードウェア記述のメタデータ ファイルを生成します。

このセクションのプラットフォーム サンプルでの操作を終了した後、`<sdsoc_root>/platforms` ディレクトリにある SDSoC 環境に含まれるプラットフォームを調べてみると有益です。次のサンプルでは、プラットフォームのビルドに関する異なる側面を示します。

- `zc702_axis_io`：SDSoC プラットフォームのダイレクト I/O のエクスポート
- `zc702_led`：プラットフォーム内の IP のソフトウェア制御
- `zc702_acp`：プラットフォームと `sdscc` での AXI バス インターフェイスの共有

例： SDSoC プラットフォームのダイレクト I/O

SDSoC システム コンパイラでは、ADC、DAC、ビデオ I/O などの入力および出力サブシステムと直接通信するハードウェア アクセラレータ ネットワークを生成できます。これらの接続は、未処理の物理データ 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された AXI4-Stream インターフェイスに変換することにより達成されます。このチュートリアルでは、このようなプラットフォームを作成します。

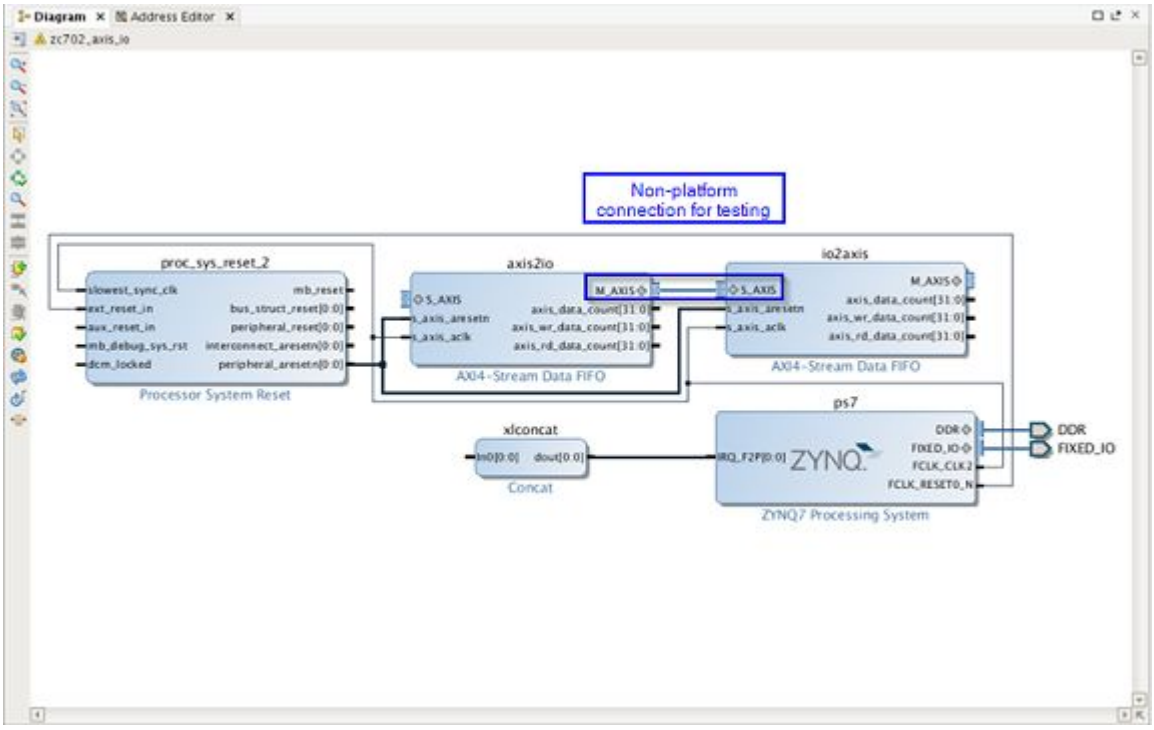
推奨： このチュートリアルでは、`<sdsoc_root>/samples/platforms/zc702_axis_io` のサンプルを使用します。

SDSoC 環境ターミナル シェルを開き、`<sdsoc_root>/samples/platforms/zc702_axis_io` を新しいディレクトリにコピーし、`cd` を使用してこのディレクトリに移動します。このプラットフォームはそのままで完全に機能しますが、このチュートリアルでは再作成します。ターミナルで次のコマンドを実行し、再作成するファイルを保存します。

```
mkdir myplatforms

cp -rf <sdsoc_root>/samples/platforms/zc702_axis_io myplatforms

cd myplatforms/zc702_axis_io

mkdir solution

mv zc702_axis_io_hw.pfm solution

mv arm-xilinx-linux-gnueabi/include/zc702_axis_io.h solution

mv arm-xilinx-linux-gnueabi/lib/libzc702_axis_io.a solution
```

SDSoC プラットフォームのハードウェア コンポーネントには Vivado プロジェクトが含まれますが、このチュートリアルではプロジェクトは再作成しません。ターミナルで、`cd` を使用して `vivado` サブディレクトリに移動します。`vivado zc702_axis_io.xpr` コマンドを使用して Vivado プロジェクトを開き、ブロック図を開きます。

図 4-1： zc702_axis_io ブロック図



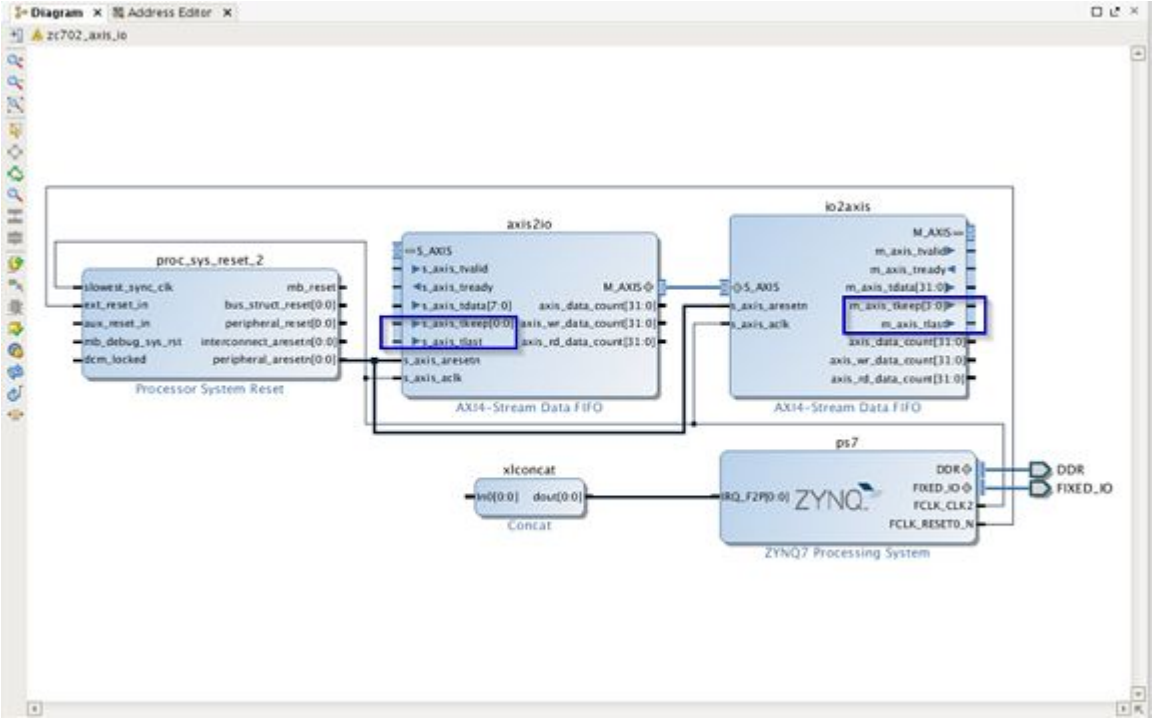
プラットフォームの直接入力ポートは、io2axis IP ブロックの終端されていない M_AXIS ポートで、プラットフォーム出力ポートは axis2io IP ブロックの S_AXIS ポートです。実際のプラットフォームでは、io2axis IP ブロックの S_AXIS ポートはほかのロジックまたはデバイス ピンに接続されますが (axis2io の M_AXIS ポートも同様)、このサンプル デザインでは、ブロック図に示すように、外部ループバック テストをエミュレートするためプラットフォームでない接続を使用します。

重要： axis_to_io ブロックの M_AXI ポートから io2axis_0 ブロックの S_AXI ポートへの接続は、プラットフォームの一部ではありません。実際のプラットフォームでは、これらのポートはほかの IP に接続され、最終的に Zynq デバイスのピンに接続されます。

このサンプルでは、この接続は出力から入力への外部ループバックを表しているので、データをプラットフォーム出力に書き込んで同じデータをプラットフォーム入力からリードバックすることにより、SDSoC でプラットフォームをテストできます。このループバック接続はプログラマブル ロジックを使用してインプリメントされますが、実際のプラットフォームでは、このようなループバック インターフェイスにはデバイス外の接続が必要です。

SDSoC プラットフォーム インターフェイスでは、データ ムーバー IP に準拠するため、AXI4-Stream インターフェイスに TLAST および TKEEP 側帯波信号が必要です。これらの側帯波信号は、次の図に示すように IP インテグレーターでポートを展開すると確認できます。プラットフォーム内でのみ使用される AXI4-Stream のバスには、これらの側帯波信号は必要ありません。

図 4-2： AXI4-Stream のバス接続



推奨： 現在のところ、エクスポートされた AXI および AXI4-Stream プラットフォーム インターフェイスは同じデータ モーション クロック (dmclkid) で動作する必要があります。プラットフォーム I/O に SDSoC 環境のプラットフォーム クロック以外のクロックが必要な場合は、Vivado IP カタログの AXI4-Stream Data FIFO IP をクロック乗せ換えに使用できます。

SDSoC ハードウェア プラットフォーム記述の生成

次の手順を実行します。これらの Tcl コマンドは、`solution/zc702_axis_io_pfm.tcl` に移動したプラットフォーム ファイル `vivado/zc702_axis_ip_pfm.tcl` にも含まれます。

1. Vivado Tcl コンソールで次のコマンドを実行し、SDSoC Vivado Tcl API を読み込んでハードウェア プラットフォーム オブジェクトを作成します。

```
source -notrace <sdsoc_root>/scripts/vivado/sdsoc_pfm.tcl

set pfm [sdsoc::create_pfm zc702_axis_io_hw.pfm]
```

2. 次のコマンドを入力し、プラットフォーム名を宣言して、sdscc -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. 次のコマンドを使用して、デフォルトのプラットフォーム クロックの ID を 2 にします。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。各プラットフォーム クロックに対して、関連の proc_sys_reset_2 IP インスタンスも宣言する必要があることに注意してください。

```
sdsoc::pfm_clock       $pfm FCLK_CLK2 ps7 2 true  proc_sys_reset_2
```

4. 次のコマンドを入力してプラットフォーム 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. 次のコマンドを使用して、ダイレクト I/O をプロキシする io2axis マスターと axis2io スレーブ AXI4-Stream バス インターフェイスを宣言します。

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

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

6. 次のコマンドを入力し、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 xlconcat
}
```

7. これですべてのインターフェイスが宣言されたので、次のコマンドを使用してプラットフォーム ルート ディレクトリに SDSoC プラットフォーム ハードウェア記述ファイル zc702_axis_io_hw.pfm を作成します。

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

Vivado を終了し、SDSoC ターミナル から vivado ディレクトリで生成されたプラットフォーム ハードウェア記述を確認し、プラットフォーム ルート ディレクトリに移動して、不要なプロジェクト ファイルを削除します。

```
sds-pf-check zc702_axis_io_hw.pfm

mv -f zc702_axis_io_hw.pfm ..

rm -rf zc702_axis_io.cache

rm -rf zc702_axis_io.hw

rm -rf zc702_axis_io.runs

rm -rf zc702_axis_io.sdk

rm -rf zc702_axis_io.sim

rm -rf vivado*
```

SDSoC プラットフォーム ソフトウェア ライブラリ

ダイレクト I/O 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、アプリケーションを呼び出してエクスポートされたインターフェイスに接続する C 呼び出し可能ライブラリが含まれている必要があります。このセクションでは、「ライブラリの作成」に説明されているように、SDSoC sdslib ユーティリティを使用してスタティック C 呼び出し可能ライブラリを作成します。

1. I/O IP の C 呼び出し可能インターフェイスを定義します。SDSoC ツールのコマンド シェルで cd を使用して src ディレクトリに移動します。

   C 呼び出し可能な関数を必要とするプラットフォーム IP は、pf_read および pf_write の 2 つです。ハードウェア関数は、次のように pf_read.cpp および pf_write.cpp で定義されます。data_t 型定義および N の #define は zc702_axis_io.h ファイルに含まれます。

   ```
   void pf_read(data_t rbuf[N]) {}

   void pf_write(data_t wbuf[N]) {}
   ```

   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、sdscc 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。

2. 関数インターフェイスから該当する IP ポートへのマップを定義します。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 および pf_write IP のマップは、zc702_axis_io.fcnmap.xml にキャプチャされます。

```
<xd:repository xmlns:xd="http://www.xilinx.com/xd">
    <xd:fcnMap xd:fcnName="pf_read" xd:componentRef="zc702_axis_io">
        <xd:arg
       xd:name="rbuf"
       xd:direction="out"
       xd:busInterfaceRef="io2axis_M_AXIS"
       xd:portInterfaceType="axis"
       xd:arraySize="64"
       xd:dataWidth="32"
    />
    </xd:fcnMap>
    <xd:fcnMap xd:fcnName="pf_write" xd:componentRef="zc702_axis_io">
        <xd:arg
       xd:name="wbuf"
       xd:direction="in"
       xd:busInterfaceRef="axis2io_S_AXIS"
       xd:portInterfaceType="axis"
       xd:arraySize="64"
       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。配列引数には配列サイズを、スカラー引数にはレジスタ オフセットを指定する必要があります。

重要： fcnMap はプラットフォーム関数 pf_read 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 io2axis_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_hw.pfm の io2axis_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 のパラメーターを指定します。

   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。この演習では、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、SDSoC インストール ディレクトリから <sdsroot>/samples/fir_lib/build/fir_compiler.{fcnmap,params}.xml を開いてください。

4. ライブラリをビルドします。

sdslib コマンドは次のようになります。

```
sdslib -lib libzc702_axis_io.a \
 pf_read 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
```

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

ライブラリの sdslib は繰り返し呼び出すことができます。呼び出しごとに、ライブラリに関数が追加されます。libzc702_axis_io.a を ../arm-xilinx-linux-gnueabi/lib に、zc702_axis_io.h を ../arm-xilinx-linux-gnueabi/include にコピーして、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でライブラリを使用できるようにします。

SDSoC プラットフォーム ソフトウェア記述

SDSoC™ プラットフォーム ソフトウェア記述ファイルは XML ファイルです。このファイルには プラットフォーム ライブラリにリンクし、ハードウェア プラットフォームでアプリケーションを実行するブート イメージを作成するのに必要な情報が含まれています。現段階では、この手順は自動的に実行されません。

zc702_axis_io プラットフォームでは ZC702 ブート ファイルがすべて再利用されます。

1. プラットフォーム ソフトウェア記述 zc702_axis_io_sw.pfm を開きます。

   次のエレメントではプラットフォーム ディレクトリで作成されたプラットフォーム ソフトウェア ライブラリの保存場所が SDSoC 環境に示されます。

```
<xd:libraryFiles
    xd:os="linux"
    xd:includeDir="arm-xilinx-linux-gnueabi/include"
    xd:libDir="arm-xilinx-linux-gnueabi/lib"
    xd:libName="zc702_axis_io"
/>
```

   同様にブート ファイルは次のように指定されます。

```
<xd:bootFiles xd:os="linux"
    xd:bif="boot/linux.bif"
    xd:readme="boot/generic.readme"
    xd:devicetree="boot/devicetree.dtb"
    xd:linuxImage="boot/uImage"
    xd:ramdisk="boot/uramdisk.image.gz"
/>
```

zc702_axis_io プラットフォームのテスト

プラットフォームをテストするには、IDE 内で SDSoC プロジェクトを新規に作成し、プラットフォームに [Other] を選択してプラットフォームのディレクトリを指定します (フォルダーとプラットフォーム名は同じにしてください)。プラットフォームには samples ディレクトリが含まれ、arraycopy というテスト アプリケーションが含まれます。この samples/template.xml ファイルには、SDSoC IDE にサンプル アプリケーションを登録します。

```
<template location="arraycopy" name="Array copy" description="Simple test application">

    <supports>

        <and>

            <os name="Linux"/>

        </and>

    </supports>

</template>
```

1. ハードウェアに arraycopy 関数を選択します。プログラム データフローにより、出力をメモリに転送する前に、プラットフォーム入力からハードウェア関数への直接信号パスが作成されます。
2. [C/C++ Build Settings] から、プラットフォーム I/O 関数を含む zc702_axis_io ライブラリを -l ライブラリ リンカー オプションに追加します。
3. arraycopy.cpp を開きます。次の点を確認します。
   - sds_alloc を使用してバッファーが割り当てられている方法

     ```
     data_t *wbuf = (data_t *) sds_alloc(N * sizeof(data_t));

     data_t *rbuf = (data_t *) sds_alloc(N * sizeof(data_t));

     data_t tmp[N];
     ```

   - プラットフォーム入力から読み出すか、プラットフォーム出力に書き込むためにプラットフォームの関数が呼び出される方法

     ```
     pf_write(wbuf);  // write to platform output

     pf_read(tmp);    // read from platform input

     arraycopy(tmp, rbuf);  // direct signal path from platform input
     ```

4. アプリケーションをビルドします。ビルドが完了したら、SDDebug フォルダーの sd_card フォルダーにブートイメージとアプリケーション ELF が含まれます。コマンド ラインからプロジェクトをビルドするには、SDSoC ターミナルで cd コマンドを使用して samples/arraycopy ディレクトリに移動し、make all を実行します。
5. 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、zc702_axis_io.elf を実行します。

```
sh-4.3# ./zc702_axis_io.elf

rbuf: dead4ead dead4eae dead4eaf ... (more output)

wbuf: dead4ead dead4eae dead4eaf ... (more output)

Test PASSED!

sh-4.3#
```

例：プラットフォーム IP のソフトウェア制御

この例では、システム推論および生成プロセスからは独立した、SDSoC プラットフォームでアプリケーションでリンク可能なソフトウェア ライブラリを提供する方法を示します。このようなライブラリは、たとえばプラットフォーム内の IP ブロックを制御するのに使用できます。

このプラットフォーム例には、ZC702 ボードの LED への書き込めるようにするため、プログラマブル ロジックにインプリメントされた汎用 I/O (AXI GPIO) IP ブロックが含まれます。同じプラットフォームのハードウェア システムでは、スタンドアロンと Linux アプリケーションの両方がサポートされます。スタンドアロン ソフトウェア ライブラリでは GPIO スタンドアロン ドライバーが使用されますが、Linux ターゲット ライブラリではアプリケーション コードから直接 GPIO ペリフェラルと通信できるようにするため、Linux UIO (ユーザー空間 I/O) フレームワークが使用されます。

推奨： このチュートリアルでは、`<sdsoc_root>/samples/platforms/zc702_led/` のサンプルを使用します。

SDSoC プラットフォーム ハードウェア記述の作成

次の手順を実行します。

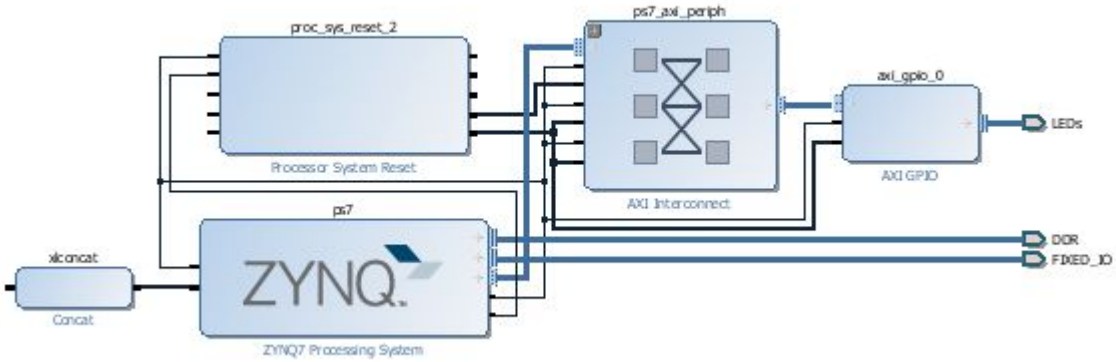
1. `samples/platforms/zc702_led` のローカル コピーを作成し、SDSoC™ 環境のターミナル シェルから `cd` を使用してこの新しいディレクトリに移動します。このプラットフォームはそのままで完全に機能しますが、このチュートリアルでは再作成します。ターミナルで次のコマンドを実行し、再作成するファイルを保存します (次のコマンドは、ファイルを `solution` というフォルダーに保存します。このファイルを作成したファイルと比較できます)。

```
mkdir myplatforms

cp -rf <sdsoc_root>/samples/platforms/zc702_led myplatforms

cd myplatforms/zc702_led

mkdir solution

mv zc702_led_hw.pfm solution

mv arm-xilinx-linux-gnueabi/include/uio_axi_gpio.h solution

mv arm-xilinx-linux-gnueabi/lib/libzc702_led.a solution

mkdir solution/arm-xilinx-eabi

mv arm-xilinx-eabi/include solution/arm-xilinx-eabi
```

2. cd コマンドで vivado ディレクトリに移動し、vivado zc702_led.xpr を実行し、Vivado IDE で [Open Block Diagram] をクリックします。

   次の図は、Vivado AXI_GPIO IP ブロックを使用して ZC702 ボードの LED に接続するところを示しています。

図 4-3： zc702_led ブロック図



3. Vivado Tcl コンソールで次のコマンドを実行し、SDSoC Vivado Tcl API を読み込んでハードウェア プラットフォーム オブジェクトを作成します。

```
source -notrace <sdsoc_root>/scripts/vivado/sdsoc_pfm.tcl

set pfm [sdsoc::create_pfm zc702_led_hw.pfm]
```

4. 次のコマンドを入力し、プラットフォーム名を宣言して、sdscc -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_led" "1.0"

sdsoc::pfm_description $pfm "Zynq ZC702 Board With Software Control of Platform IP"
```

5. 次のコマンドを使用して、デフォルトのプラットフォーム クロックの ID を 2 にします。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。各プラットフォーム クロックに対して、関連の proc_sys_reset_2 IP インスタンスも宣言する必要があることに注意してください。

```
sdsoc::pfm_clock       $pfm FCLK_CLK2 ps7 2 true  proc_sys_reset_2
```

6. 次のコマンドを入力してプラットフォーム 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。このプラットフォームでは、プラットフォーム内で LED を書き込むために使用する M_AXI_GP0 ポートは宣言しません。

```
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
```

7. Linux アプリケーションをサポートするには、Linux UIO フレームワークにビルドされた LED に書き込むユーザー空間ライブラリをプラットフォームで提供します。上記の Tcl コマンドでは、次の API 呼び出しで axi_gpio_0 IP ブロックが UIO デバイスとして宣言されます。この宣言は、SDSoC コンパイラでアプリケーション コードのほかの UIO デバイスが正しくコンフィギュレーションされるようにするため必要です。

```
sdsoc::pfm_iodev       $pfm S_AXI axi_gpio_0 uio
```

8. 次のコマンドを入力し、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 xlconcat

}
```

9. これですべてのインターフェイスが宣言されたので、次のコマンドを使用してプラットフォーム ルート ディレクトリに SDSoC プラットフォーム ハードウェア記述ファイル zc702_led_hw.pfm を作成します。

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

Vivado を終了し、SDSoC ターミナル ウィンドウから vivado ディレクトリでプラットフォーム ハードウェア記述を確認し、ファイルをプラットフォーム ディレクトリに移動して、不要なファイルを削除します。

```
sds-pf-check zc702_led_hw.pfm

mv -f zc702_led_hw.pfm ..

rm -rf zc702_led.cache

rm -rf zc702_led.hw

rm -rf zc702_led.runs

rm -rf zc702_led.sim

rm -rf vivado*
```

zc702_led プラットフォーム ソフトウェア記述の作成

- スタンドアロン アプリケーションでは、SDSoC プラットフォームにリンカー スクリプトとヘッダー ファイルが必要です。これらから、SDSoC コンパイラによりアプリケーション特定のボード サポート パッケージ (BSP) が作成され、アプリケーション コードがプラットフォーム特定の libXil.a ライブラリに対してリンクされます。リンカー スクリプトは、次のように作成します。
  1. Vivado IDE でハードウェア システムを開き、ハードウェア エクスポート機能を使用します。
  2. ザイリンクス SDK を使用する場合と同様に、エクスポートされたハードウェア システムからハードウェア プラットフォーム仕様プロジェクトを作成します。
  3. ザイリンクス SDK を使用する場合と同様に、ボード サポート パッケージ (BSP) プロジェクトを作成します。
  4. 手順 2 および 3 で作成したハードウェア仕様と BSP を使用して、Hello World アプリケーション プロジェクトを作成します。

  Hello World プロジェクト用に作成されたリンカー スクリプトとBSP からのヘッダー ファイルが、SDSoC プラットフォーム ソフトウェア コンポーネントとなります。
- Linux アプリケーションでは、SDSoC プラットフォームのソフトウェア コンポーネントが zc702_led の Linux ブート環境を提供します。この Linux ブート環境は、AXI GPIO プラットフォームのペリフェラルの登録に必要な devicetree.dtb を除き、SDSoC 環境の一部として提供される ZC702 プラットフォームと同じです。zc702_led プラットフォームには、Linux UIO ドライバー フレームワークを介して AXI GPIO ペリフェラルにアクセスするソフトウェア ライブラリも含まれます。

スタンドアロン プラットフォーム ソフトウェアの作成

スタンドアロン アプリケーション用の SDSoC プラットフォームのソフトウェア コンポーネントを作成する前に、まず前のセクションで説明したハードウェア コンポーネントをビルドしておく必要があります。前のセクションでは既に Vivado ハードウェアのエクスポート コマンド (ブロック図を開いた状態で [File] → [Export] → [Export Hardware] をクリック) を実行してハードウェア ハンドオフ ファイル zc702_led.sdk/zc702_led_wrapper.hdf を作成しました。

1. SDSoC IDE で [New] → [Project] → [Xilinx] → [Hardware Platform Specification] をクリックし、ザイリンクス SDK を使用する場合と同様に、作成した zc702_led.sdk/zc702_led_wrapper.hdf ハンドオフ ファイルを選択してハードウェア プラットフォーム プロジェクトを作成します。
2. SDSoC IDE で [New] → [Project] → [Xilinx] → [Board Support Project] をクリックし、ザイリンクス SDK を使用する場合と同様に、作成したハードウェア プラットフォーム プロジェクトを選択して BSP を作成します。ポップアップ ウィンドウが表示されたら BSP プロジェクトに xilffs ライブラリを追加します。

3. SDSoC ターミナル ウィンドウでは、ヘッダー ファイルを BSP プロジェクト ディレクトリから SDSoC プラットフォーム プロジェクトの zc702_led/arm-xilinx-eabi/include ディレクトリにコピーします。これらのヘッダー ファイルがスタンドアロン アプリケーションの zc702_led プラットフォームに使用されます。
4. BSP プロジェクトには、ドライバーのバージョン、オプション、およびその他の設定を指定する system.mss BSP コンフィギュレーション ファイルが含まれます。BSP はデフォルト設定を使用して作成されているので、プラットフォームに .mss ファイルは必要ありません。アプリケーション特定のハードウェア システムを生成した後、sdscc/sds++ が BSP を生成するときに .mss ファイルを自動的に作成し、アプリケーション ELF にリンクします。
5. SDSoC IDE で [New] → [Project] → [Application Project] をクリックし、ザイリンクス SDK を使用する場合と同様に、このハードウェア仕様および BSP のアプリケーション プロジェクトを作成します。[Hello World] を選択し、アプリケーション ソフトウェア プロジェクトを作成します。
6. SDSoC IDE の [Project Explorer] タブで [Hello World] プロジェクトをクリックし、タスクバーのハンマーアイコンをクリックするか、[Project Explorer] タブで右クリックして [Build Project] をクリックしてプロジェクトをビルドします。

リンカー スクリプト src/lscript.ld をテキスト エディターでクリックして、ヘッダー サイズを次のように変更します。

```
_HEAP_SIZE = DEFINED(_HEAP_SIZE) ? _HEAP_SIZE : 0x8000000;
```

次に、リンカー スクリプトを zc702_led/arm-xilinx-eabi プラットフォーム ディレクトリにコピーし、テキスト エディターで zc702_led/zc702_led_sw.pfm を開いて、次のエレメントを追加します。

```
<xd:libraryFiles

  xd:os="standalone"

  xd:includeDir="arm-xilinx-eabi/include"

  xd:ldscript="arm-xilinx-eabi/lscript.ld"

/>
```

SDSoC コンパイラでアプリケーションをコンパイルしてリンクすると、プラットフォーム用のスタンドアロン BSP が自動的に作成され、作成したリンカー スクリプトを使用してアプリケーション ELF がリンクされます。

Linux ソフトウェア プラットフォームの作成

重要： このチュートリアルでは、組み込み Linux をターゲットとするプラットフォームのメモリ マップド プラットフォーム IP をソフトウェアで制御する単純な例を示します。組み込み Linux、ユーザー空間ドライバー、デバイス ツリーの自己完結した基盤としては使用できません。概念をよく理解していない場合は、詳細を説明する参考資料が多数ありますので、このチュートリアルを実行する前に参照してください。

zc702_led/src ディレクトリで SDSoC ターミナル シェルを開き、テキスト エディターで Makefile を開きます。デフォルトのビルド ターゲットにより、uio_axi_gpio.[ch] ファイルで構成される axi_gpio ブロックの UIO ドライバーを含むソフトウェア ライブラリが作成されるのを確認します。このユーザー空間ドライバーは、プラットフォーム内のほかのメモリ マップド IP ブロックに適用可能な GPIO IP を制御する単純な API を提供します。

ライブラリを構築するには、ターミナル シェルで次を実行します。

```
make
```

その後、次のようにライブラリおよびヘッダー ファイルをプラットフォーム ディレクトリにコピーします。

```
cp libzc702_led.a ../arm-xilinx-linux-gnueabi/lib
cp uio_axi_gpio.h ../arm-xilinx-linux-gnueabi/include
```

samples/arraycopy.cpp に含まれる SDSoC テスト プログラムは、API の使用例を示します。

SDSoC プラットフォームで Linux アプリケーションをサポートするには、プラットフォームをブートするのに使用される zc702 プラットフォームで提供される Linux デバイス ツリーをアップデートする必要があります。zc702_led プラットフォームの一部として提供されているデバイス ツリーは、ZC702 プラットフォームの devicetree.dtb を変更して手動で作成されています。まず、zc702 devicetree.dtb が dtc コンパイラを使用してテキスト フォーマット (.dts またはデバイス ツリー ソース) に変換されます。

```
dtc -I dtb -O dts -o devicetree.dts boot/devicetree.dtb
```

axi_gpio_0 プラットフォーム ペリフェラルを Linux に登録するには、デバイス ツリー ファイル devicetree.dts に 2 つの変更を加える必要があります。まず、次のように bootargs に uio_pdrv_genirq.of_id=generic-uio を追加します。

```
bootargs = "console=ttyPS0,115200 root=/dev/ram rw earlyprintk uio_pdrv_genirq.of=generic-uio";
```

そして、次のデバイス ツリー ブロブを追加します。

```
gpio@41200000 {
    compatible = "generic-uio";
    reg = <0x41200000 0x10000>;
};
```

このブロブを、デバイス ツリーの amba レコード内で最初に現れる generic-uio デバイスとしてデバイス ツリーに挿入します。

後のブロブの名前は固有のものにする必要があります。ザイリンクスでは、システム生成中に Vivado で計算されたペリフェラルのベース アドレスを使用した命名規則を採用しています。reg メンバーの値は、そのペリフェラルのベース アドレスおよび その IP の該当するアドレス セグメントのバイト アドレス番号になるはずです。これらはどちらも Vivado IP インテグレーターの [Address Editor] タブで確認できます。

デバイス ツリーを Linux カーネルで必要とされるバイナリ形式に変換し戻すには、dtc デバイス ツリー コンパイラーを再び使用します。

```
dtc -I dts -O dtb -o devicetree.dtb boot/devicetree.dts
```

zc702_led/lib ディレクトリの UIO ドライバーには、UIO フレームワークに必要なフックが含まれています。

```
int axi_gpio_init(axi_gpio *inst, const char* instnm);
int axi_gpio_release(axi_gpio *inst);
```

ペリフェラルにアクセスするアプリケーションでは、ペリフェラルにアクセスする前に初期化関数を呼び出して、終了したときにそのリソースを解放する必要があります。

デバイス ツリーと Linux OS フレームワークの詳細は、次のウェブサイトなどから入手可能なトレーニング資料を参照してください。

http://www.free-electrons.com/docs

テキスト エディターで zc702_led_sw.pfm を開き、次のエレメントを追加します。

```
<xd:libraryFiles
  xd:os="linux"
  xd:libName="zc702_led"
  xd:libDir="arm-xilinx-linux-gnueabi/lib"
  xd:includeDir="arm-xilinx-linux-gnueabi/include"
/>
```

zc702_led プラットフォームのテスト

プラットフォームをテストするには、SDSoC IDE で新規プロジェクトを作成します。プラットフォームに [Other] を選択してプラットフォームのディレクトリを指定します (フォルダーとプラットフォーム名は同じにしてください)。プラットフォームには samples ディレクトリが含まれ、arraycopy というテスト アプリケーションが含まれます。このテスト アプリケーションにはループ内で起動するシンプルな arraycopy ハードウェア関数が含まれます。このアプリケーション コードでは、配列入力をハードウェア関数出力にコピーするだけでなく、ZC702 ボードの LED をバイナリ表現と一致するように点灯します。

このディレクトリの template.xml ファイルには、SDSoC IDE を使用したサンプル アプリケーションが含まれます。

```
<template location="arraycopy" name="Array copy" description="Linux test application">
    <supports>
        <and>
            <os name="Linux"/>
        </and>
    </supports>
</template>

<template location="arraycopy_sa" name="Array copy" description="Standalone test application">
    <supports>
        <and>
            <os name="standalone"/>
        </and>
    </supports>

</template>
```

- コマンド ラインから Linux プラットフォームをテストするには、次を実行します。
  a. SDSoC ターミナルで cd コマンドで samples/arraycopy ディレクトリに移動して make を実行します。
  b. ビルドが終了したら、sd_card ディレクトリの内容を SD カードにコピーして、ZC702 ボードに挿入して電源を入れます。Linux の起動後にボードに接続されたシリアル ターミナルから /mnt/arraycopy.elf を実行します。
- コマンド ラインからスタンドアロン プラットフォームをテストするには、次を実行します。
  a. SDSoC ターミナルで cd コマンドで samples/arraycopy_sa ディレクトリに移動して make を実行します。
  b. ビルドが終了したら、sd_card ディレクトリの内容を SD カードにコピーして、ZC702 ボードに挿入して、stdout を監視するためにシリアル ターミナルを接続し、電源を入れます。

かなりシンプルですが、zc702_led プラットフォームは、SDSoC ソフトウェア ランタイム外のプラットフォーム ペリフェラルへのアクセス方法を示しています。スタンドアロン アプリケーションにはペリフェラル デバイス ドライバー API への直接呼出しが含まれます。Linux アプリケーションは Linux UIO フレームワーク (メモリ マップされた読み出し/書き込み) を使用して、プラットフォーム ペリフェラルを制御し、SDSoC プラットフォーム ライブラリからアクセスできるようにします。

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。

Vivado ツールでビルドされたデザインのコネクティビティ インターフェイス (AXI および AXI4-Stream、クロック、リセット、および割り込みポートを含む) は、プラットフォーム ハードウェア記述で定義します。このコネクティビティ インターフェイスは、次の手順に従って、「SDSoC Vivado Tcl コマンド」で説明される Tcl API のセットを使用して Vivado Tcl コンソール内から宣言します。

1. Vivado Design Suite 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. Vivado Design Suite GUI でハードウェア プロジェクトを開きます。このプロセスは、スクリプト化することもできます。
3. SDSoC Vivado Tcl API を読み込みます。
4. Vivado で Tcl API を実行し、次の手順を実行します。
   a. ハードウェア プラットフォーム名を宣言します。
   b. プラットフォームの簡単な説明を宣言します。
   c. プラットフォーム クロック ポートを宣言します。
   d. 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。
   e. 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。
   f. 使用可能なプラットフォーム割り込みを宣言します。
   g. プラットフォーム ハードウェア記述のメタデータ ファイルを生成します。

推奨： このチュートリアルでは、`<sdsoc_root>/samples/platforms/zc702_acp/` のサンプルを使用します。プラットフォームのビルドおよびテスト手順は、`readme.txt` ファイルを参照してください。

SDSoC プラットフォーム ハードウェア記述の作成

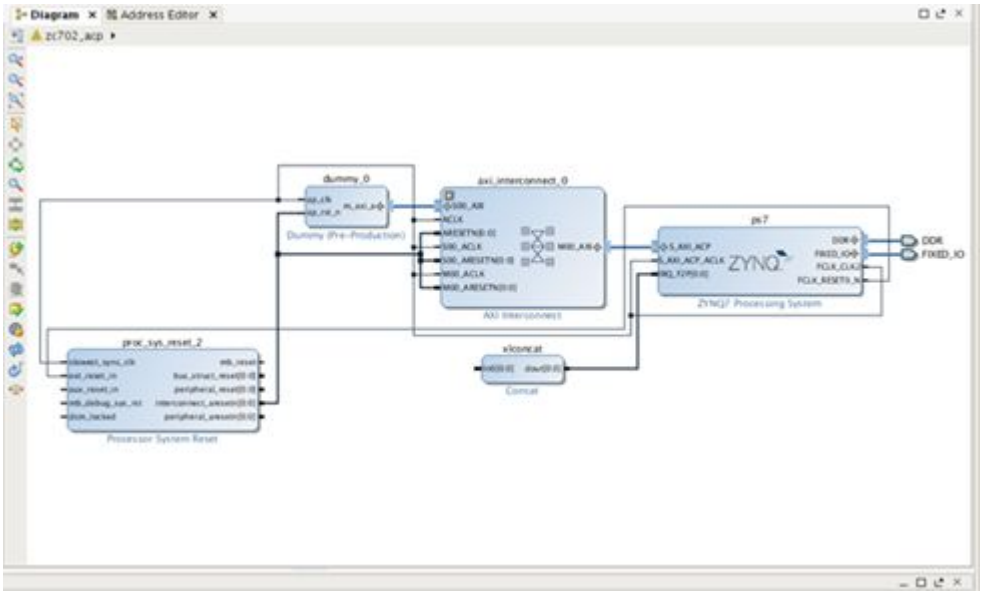
1. `samples/platforms/zc702_acp` のローカル コピーを作成し、SDSoC™ 環境のターミナル シェルから `cd` を使用して `zc702_acp/vivado` ディレクトリに移動します。このプラットフォームはそのままで完全に機能しますが、このチュートリアルでは再作成します。ターミナルで次のコマンドを実行し、再作成するファイルを保存します。

```
mkdir myplatforms
cp -rf <sdsoc_root>/samples/platforms/zc702_acp myplatforms
cd myplatforms/zc702_acp
mkdir solution
mv zc702_acp_hw.pfm solution
```

2. Vivado ツールで zc702_acp.xpr を開き、ブロック図を開きます。

   ブロック図は次のようになります。

図 4-4： ZC702_acp ブロック図



3. Vivado Tcl コンソールで次のコマンドを実行し、SDSoC Vivado Tcl API を読み込んでハードウェア プラットフォーム オブジェクトを作成します。

```
source -notrace <sdsoc_root>/scripts/vivado/sdsoc_pfm.tcl

set pfm [sdsoc::create_pfm zc702_acp_hw.pfm]
```

4. 次のコマンドを入力し、プラットフォーム名を宣言して、sdscc -sds-pf-info zc702_acp 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_acp" "1.0"

sdsoc::pfm_description $pfm "Zynq ZC702 board with a shared ACP port"
```

5. 次のコマンドを使用して、デフォルトのプラットフォーム クロックの ID を 2 にします。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。各プラットフォーム クロックに対して、関連の proc_sys_reset_2 IP インスタンスも宣言する必要があることに注意してください。

```
sdsoc::pfm_clock       $pfm FCLK_CLK2 ps7 2 true  proc_sys_reset_2
```

6. 次のコマンドを入力してプラットフォーム 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。S01_AXI ポートの API 呼び出しは、インターコネクト ポートをプラットフォーム インターフェイスの一部として宣言し、プラットフォーム内の processing_system7 IP ブロック上にあるハードウェア コヒーレント S_AXI_ACP ポートにアクセスできるようにします。Vivado ブロック図で、s00_AXI ポート (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。

```
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

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

7. 次のコマンドを入力し、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 xlconcat

}
```

8. これですべてのインターフェイスが宣言されたので、次のコマンドを使用してプラットフォーム ルート ディレクトリに SDSoC プラットフォーム ハードウェア記述ファイル zc702_acp_hw.pfm を作成します。

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

Vivado を終了し、SDSoC ターミナル から vivado ディレクトリでハードウェア記述を確認し、ファイルをプラットフォーム ディレクトリに移動して、不要なプロジェクト ファイルを削除します。

```
sds-pf-check zc702_acp_hw.pfm

mv -f zc702_acp_hw.pfm ..

rm -rf zc702_acp.cache

rm -rf zc702_acp.hw

rm -rf zc702_acp.runs

rm -rf zc702_acp.sdk

rm -rf zc702_acp.sim

rm -rf vivado*
```

zc702_acp プラットフォームのテスト

プラットフォームをテストするには、IDE 内で SDSoC プロジェクトを新規に作成し、プラットフォームに [Other] を選択してプラットフォームのディレクトリを指定します (フォルダーとプラットフォーム名は同じにしてください)。プラットフォームには samples ディレクトリが含まれ、arraycopy というテスト アプリケーションが含まれます。このディレクトリの template.xml ファイルには、SDSoC IDE を使用したサンプル アプリケーションが含まれます。

```
<template location="arraycopy" name="Array copy" description="Simple test application">
    <supports>
        <and>
            <os name="Linux"/>
        </and>
    </supports>
</template>
```

SDSoC GUI でプラットフォームをテストするには、次の手順を実行します。

1. [File] → [New] → [SDSoC Project] をクリックし、zc702_acp という新しい SDSoC プロジェクトを作成します。プラットフォームを選択するには、[Other] ボタンをクリックして <sdsoc_root>/samples/platforms/zc702_acp を選択します。[Next] をクリックします。
2. [Array Copy] テンプレートを選択して [Finish] をクリックします。
3. [Project Explorer] タブで zc702_acp/src/arraycopy.cpp を展開し、arraycopy 関数を右クリックして [Toggle HW/SW] をクリックしてハードウェアの関数を選択します。
4. プロジェクトをビルドします。ビルドが完了したら、SDDebug/sd_card ディレクトリの内容を SD カードにコピーします。
5. ZC702 ボードをブートし、次を実行します。

```
$ /mnt/zc702_acp.elf
```

コマンド ラインからプラットフォームをテストするには、SDSoC ターミナルで cd コマンドを使用して samples/arraycopy ディレクトリに移動します。

1. プラットフォームをテストするには、ターミナル シェルで make コマンドを実行します。
2. テスト アプリケーションには、SDSoC 環境の axi_dma_simple データムーバーを使用した単純な arraycopy ハードウェア関数が含まれています。samples/arraycopy/sd_card の内容を SD カードに読み込んで起動します。

```
$ /mnt/zc702_acp.elf
```

この例では、プラットフォームと SDSoC 環境で生成されたロジック間で Processing System 7 IP ポートを共有する方法を示しています。

付録 A

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 リソースについては、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、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. 『SDSoC 環境ユーザー ガイド : SDSoC 環境の概要』(UG1028) (SDSoC 環境の docs フォルダーからも入手可能)
2. 『SDSoC 環境ユーザー ガイド』(UG1027) (SDSoC 環境の docs フォルダーからも入手可能)
3. 『SDSoC 環境ユーザー ガイド : プラットフォームおよびライブラリ』(UG1146) (SDSoC 環境の docs フォルダーからも入手可能)
4. 『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)
5. 『ZC702 評価ボード (Zynq-7000 XC7Z020 All Programmable SoC 用) ユーザー ガイド』(UG850)
6. 『Vivado Design Suite ユーザー ガイド : 高位合成』(UG902)
7. 『PetaLinux ツール資料ワークフロー チュートリアル』(UG1156)
8. Vivado® Design Suite 資料
9. 『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、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になります。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。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。

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 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。