Xilinx Standalone Library Documentation

XilFPGA Library v6.0

UG1229 (v2021.1) June 16, 2021

Table of Contents

Overview

The XilFPGA library provides an interface for the users to configure the programmable logic (PL) from PS. The library is designed to run on top of Xilinx® standalone BSPs. It acts as a bridge between the user application and the PL device. It provides the required functionality to the user application for configuring the PL device with the required bitstream.

Note: XilFPGA does not support a system with no DDR memory.

Supported Features

Zynq UltraScale+ MPSoC Platform

The following features are supported in Zynq UltraScale+ MPSoC platform:

- Full bitstream loading
- Partial bitstream loading
- Encrypted bitstream loading
- Authenticated bitstream loading
- Authenticated and encrypted bitstream loading
- Readback of configuration registers
- Readback of configuration data

Versal ACAP

The following features are supported in Versal™ platform:

- Full/Partial bitstream loading
- Device Key Encrypted bitstream loading
- Authenticated bitstream loading
- Authenticated and Device-key encrypted bitstream loading

XILINX

Chapter 2

Zynq UltraScale+ MPSoC XilFPGA Library

The library when used for Zynq UltraScale+ MPSoC runs on top of Xilinx standalone BSPs. It is tested for Arm Cortex-A53, Arm Cortex-R5F and MicroBlaze. In the most common use case, you should run this library on the PMU MicroBlaze with PMU firmware to serve requests from either Linux or U-Boot for bitstream programming.

XilFPGA Library Interface Modules

XilFPGA library uses the below major components to configure the PL through PS.

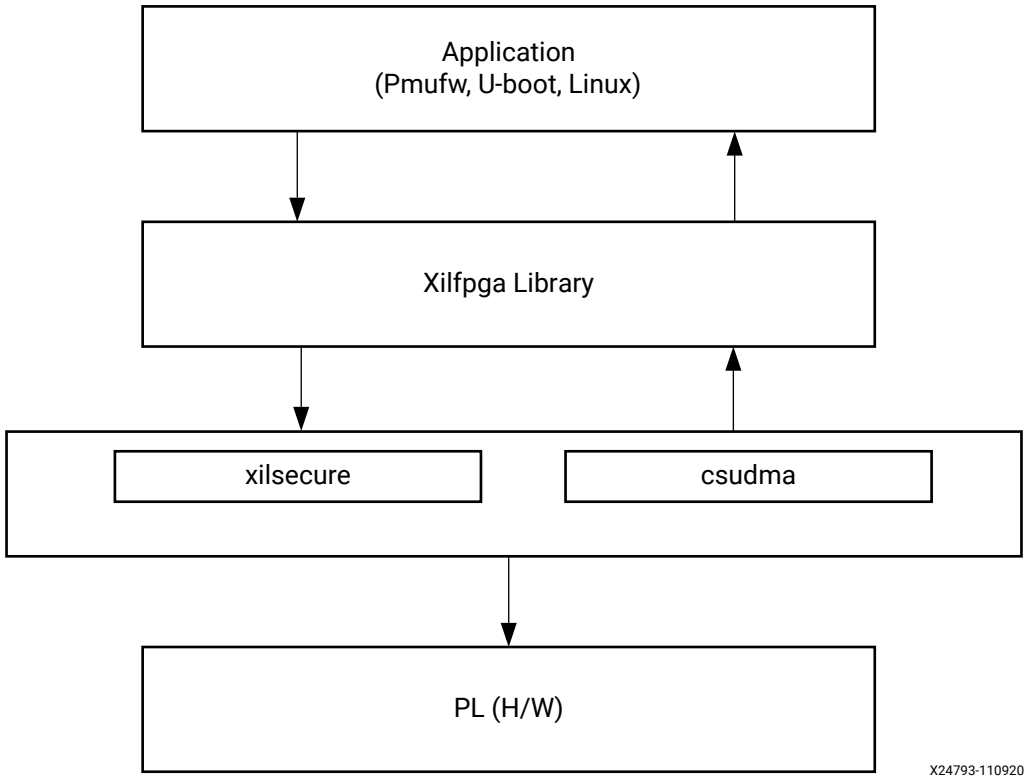
- **Processor Configuration Access Port (PCAP):** The processor configuration access port (PCAP) is used to configure the programmable logic (PL) through the PS.
- **CSU DMA Driver:** The CSU DMA driver is used to transfer the actual bitstream file for the PS to PL after PCAP initialization.
- **XilSecure Library:** The XilSecure library provides APIs to access secure hardware on the Zynq UltraScale+ MPSoCs.

Design Summary

XilFPGA library acts as a bridge between the user application and the PL device.

It provides the required functionality to the user application for configuring the PL Device with the required bitstream. The following figure illustrates an implementation where the XilFPGA library needs the CSU DMA driver APIs to transfer the bitstream from the DDR to the PL region. The XilFPGA library also needs the XilSecure library APIs to support programming authenticated and encrypted bitstream files.

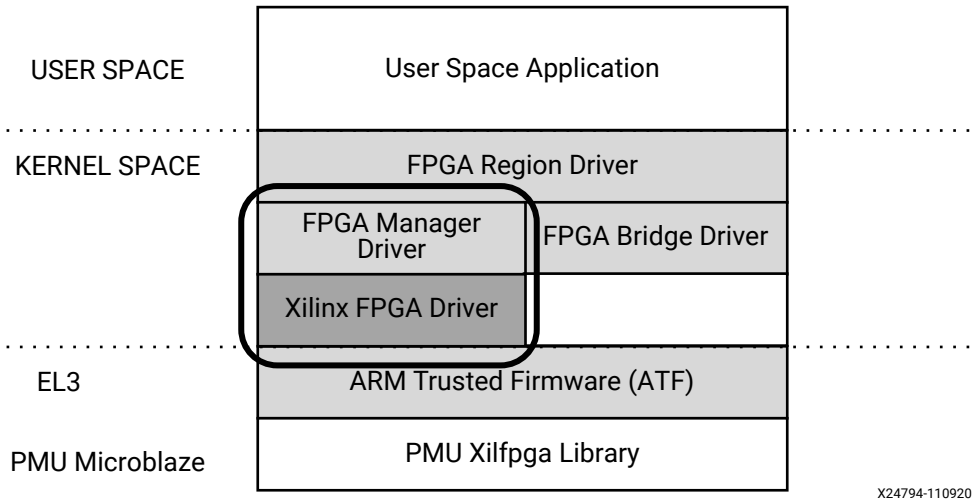
Figure 1: **XilFPGA Design Summary**



Flow Diagram

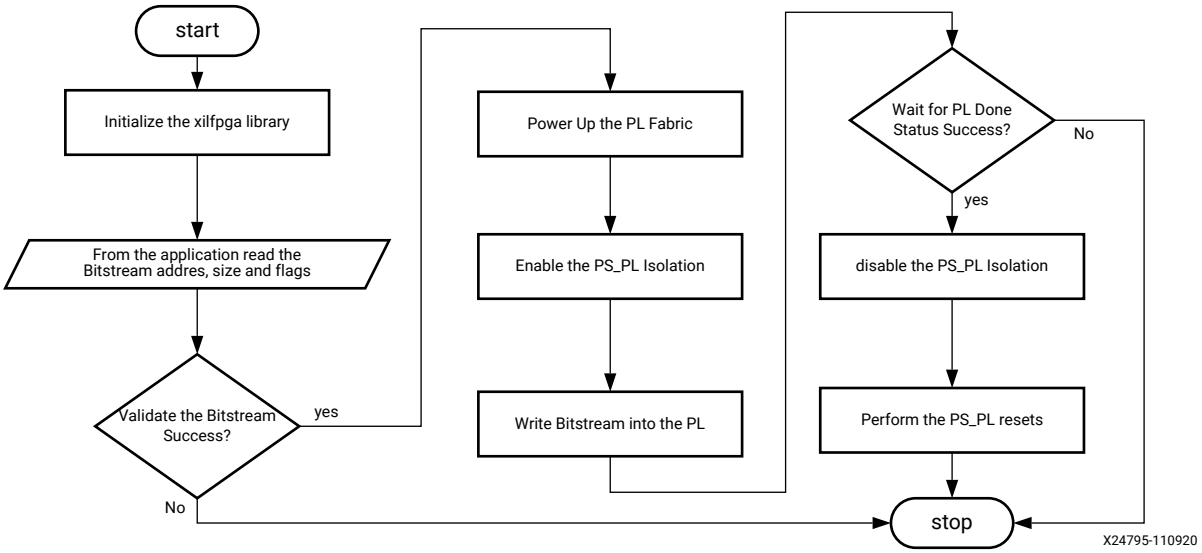
The following figure illustrates the Bitstream loading flow on the Linux operating system.

Figure 2: **Bitstream loading on Linux:**



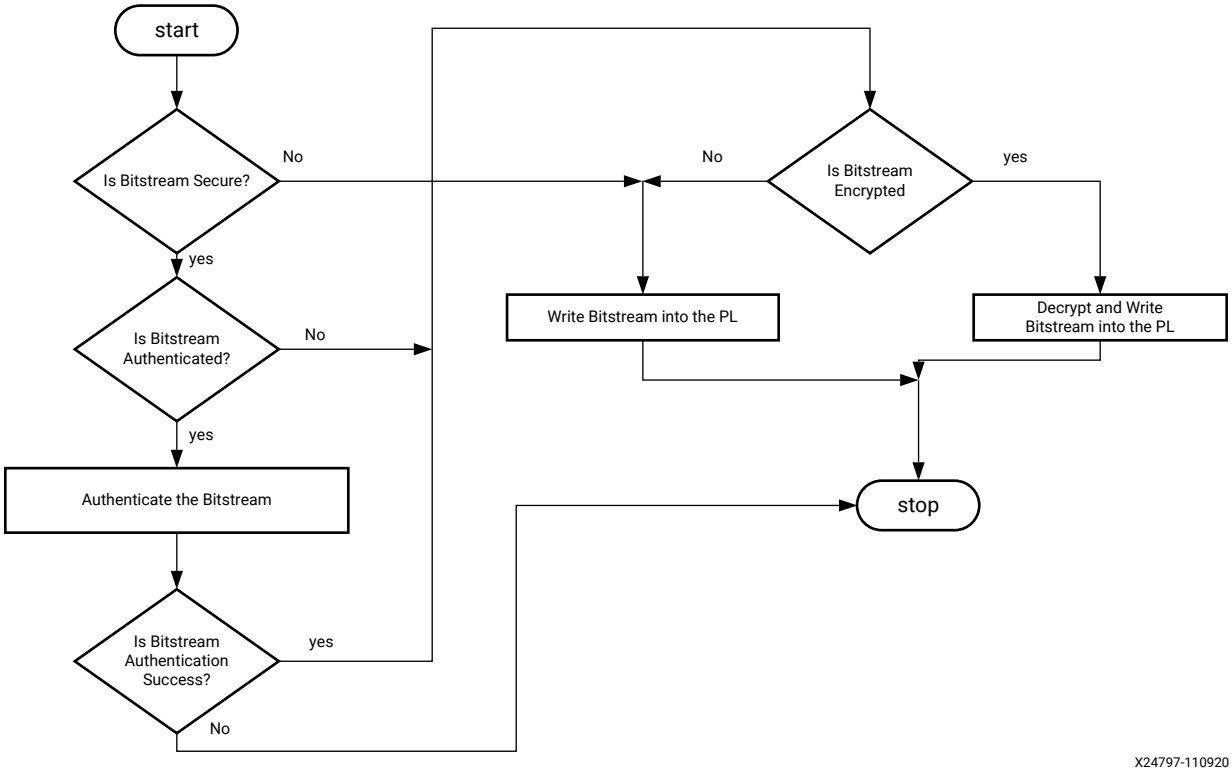
The following figure illustrates the XilFPGA PL configuration sequence.

Figure 3: **XilFPGA PL Configuration Sequence**



The following figure illustrates the Bitstream write sequence.

Figure 4: **Bitstream write Sequence**



XilFPGA BSP Configuration Settings

XilFPGA provides the following user configuration BSP settings.

Table 1: **User Configuration BSP Settings**

Parameter Name	Type	Default Value	Description
secure_mode	bool	TRUE	Enables secure Bitstream loading support.
debug_mode	bool	FALSE	Enables the Debug messages in the library.
ocm_address	int	0xfffc0000	Address used for the Bitstream authentication.
base_address	int	0x80000	Holds the Bitstream Image address. This flag is valid only for the Cortex-A53 or the Cortex-R5F processors.
secure_readback	bool	FALSE	Should be set to TRUE to allow the secure Bitstream configuration data read back. The application environment should be secure and trusted to enable this flag.
secure_environment	bool	FALSE	Enable the secure PL configuration using the IPI. This flag is valid only for the Cortex-A53 or the Cortex-R5F processors.

Setting up the Software System

To use XilFPGA in a software application, you must first compile the XilFPGA library as part of software application.

1. Click **File** > **New** > **Platform Project**.
2. Click **Specify** to create a new Hardware Platform Specification.
3. Provide a new name for the domain in the **Project name** field if you wish to override the default value.
4. Select the location for the board support project files. To use the default location, as displayed in the **Location** field, leave the **Use default location** check box selected. Otherwise, deselect the checkbox and then type or browse to the directory location.
5. From the **Hardware Platform** drop-down choose the appropriate platform for your application or click the **New** button to browse to an existing Hardware Platform.
6. Select the target CPU from the drop-down list.

7. From the **Board Support Package OS** list box, select the type of board support package to create. A description of the platform types displays in the box below the drop-down list.
8. Click **Finish**. The wizard creates a new software platform and displays it in the Vitis Navigator pane.
9. Select **Project** > **Build Automatically** to automatically build the board support package. The Board Support Package Settings dialog box opens. Here you can customize the settings for the domain.
10. Click **OK** to accept the settings, build the platform, and close the dialog box.
11. From the Explorer, double-click platform.spr file and select the appropriate domain/board support package. The overview page opens.
12. In the overview page, click **Modify BSP Settings**.
13. Using the Board Support Package Settings page, you can select the OS Version and which of the Supported Libraries are to be enabled in this domain/BSP.
14. Select the **xilfpga** library from the list of **Supported Libraries**.
15. Expand the **Overview** tree and select **xilfpga**. The configuration options for xilfpga are listed.
16. Configure the xilfpga by providing the base address of the Bit-stream file (DDR address) and the size (in bytes).
17. Click **OK**. The board support package automatically builds with XilFPGA library included in it.
18. Double-click the **system.mss** file to open it in the **Editor** view.
19. Scroll-down and locate the **Libraries** section.
20. Click **Import Examples** adjacent to the XilFPGA entry.

Enabling Security

To support encrypted and/or authenticated bitstream loading, you must enable security in PMUFW.

1. Click **File** > **New** > **Platform Project**.
2. Click **Specify** to create a new Hardware Platform Specification.
3. Provide a new name for the domain in the **Project name** field if you wish to override the default value.
4. Select the location for the board support project files. To use the default location, as displayed in the **Location** field, leave the **Use default location** check box selected. Otherwise, deselect the checkbox and then type or browse to the directory location.
5. From the **Hardware Platform** drop-down choose the appropriate platform for your application or click the **New** button to browse to an existing Hardware Platform.

6. Select the target CPU from the drop-down list.
7. From the **Board Support Package OS** list box, select the type of board support package to create. A description of the platform types displays in the box below the drop-down list.
8. Click **Finish**. The wizard creates a new software platform and displays it in the Vitis Navigator pane.
9. Select **Project** > **Build Automatically** to automatically build the board support package. The Board Support Package Settings dialog box opens. Here you can customize the settings for the domain.
10. Click **OK** to accept the settings, build the platform, and close the dialog box.
11. From the Explorer, double-click platform.spr file and select the appropriate domain/board support package. The overview page opens.
12. In the overview page, click **Modify BSP Settings**.
13. Using the Board Support Package Settings page, you can select the OS Version and which of the Supported Libraries are to be enabled in this domain/BSP.
14. Expand the **Overview** tree and select **Standalone**.
15. Select a supported hardware platform.
16. Select **psu_pmu_0** from the **Processor** drop-down list.
17. Click Next. The **Templates** page appears.
18. Select **ZynqMP PMU Firmware** from the **Available Templates** list.
19. Click **Finish**. A PMUFW application project is created with the required BSPs.
20. Double-click the **system.mss** file to open it in the **Editor** view.
21. Click the **Modify this BSP's Settings** button. The **Board Support Package Settings** dialog box appears.
22. Select **xilfpga**. Various settings related to the library appears.
23. Select **secure_mode** and modify its value to **true** .
24. Click **OK** to save the configuration.

Note: By default the secure mode is enabled. To disable modify the secure_mode value to FALSE.

Bitstream Authentication Using External Memory

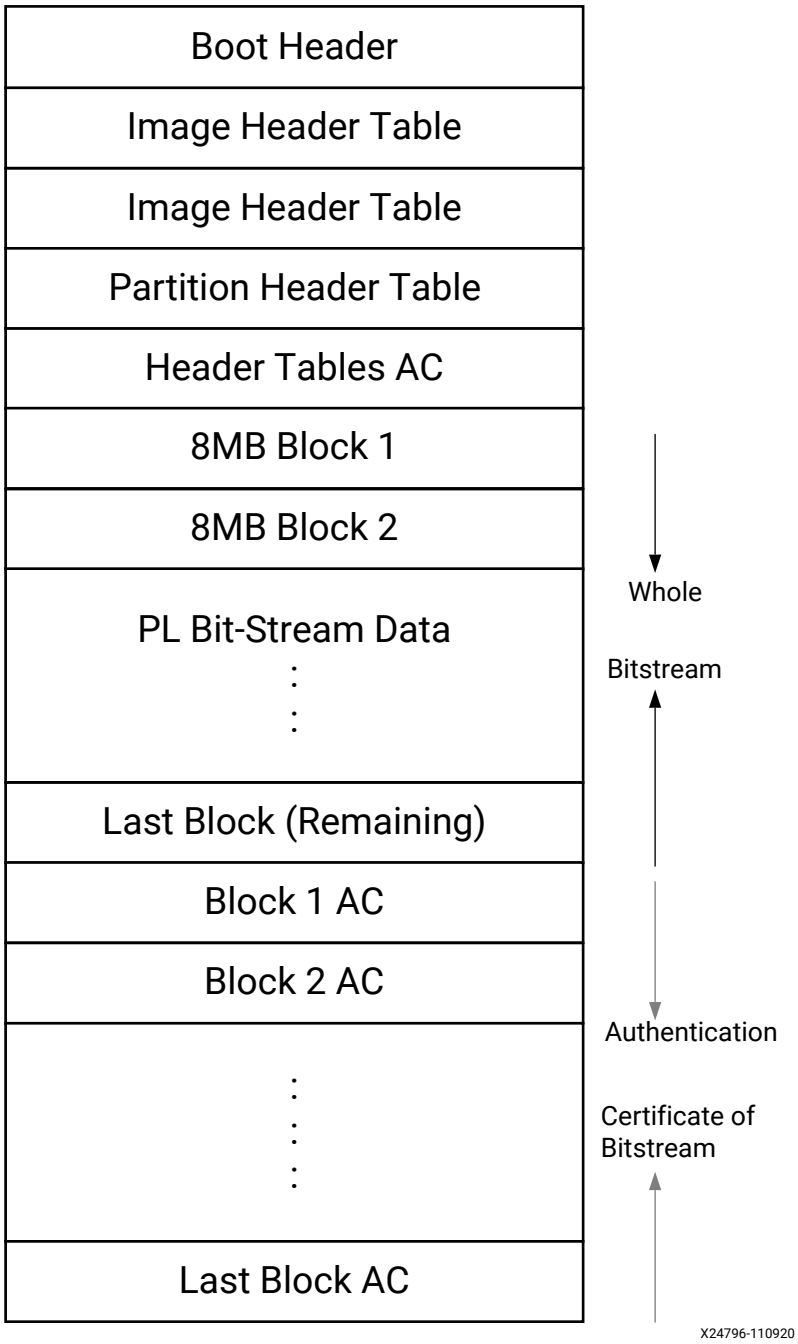
The size of the bitstream is too large to be contained inside the device, therefore external memory must be used. The use of external memory could create a security risk. Therefore, two methods are provided to authenticate and decrypt a Bitstream.

- The first method uses the internal OCM as temporary buffer for all cryptographic operations. For details, see Loading an Authenticated and Encrypted Bitstream using OCM. This method does not require trust in external DDR memory.
- The second method uses external DDR memory for authentication prior to sending the data to the decryptor, there by requiring trust in the external DDR memory. For details, see Loading an Authenticated and Encrypted Bitstream using DDR Memory Controller.

Bootgen

When a bitstream is requested for authentication, Bootgen divides the bitstream into blocks of 8 MB each and assigns an authentication certificate for each block. If the size of a bitstream is not in multiples of 8 MB, the last block contains the remaining Bitstream data.

Figure 5: **Bitstream Blocks**



When both authentication and encryption are enabled, encryption is first done on the Bitstream. Bootgen then divides the encrypted data into blocks and assigns an Authentication certificate for each block.

Loading an Authenticated and Encrypted Bitstream using OCM

To authenticate the bitstream partition securely, XilFPGA uses the FSBL section's OCM memory to copy the bitstream in chunks from DDR memory. This method does not require trust in the external DDR memory to securely authenticate and decrypt a bitstream.

The software workflow for authenticating Bitstream is as follows:

1. XilFPGA identifies DDR-secure bitstream image base address. XilFPGA has two buffers in OCM, the Read Buffer is of size 56 KB and hash of chunks to store intermediate hashes calculated for each 56 KB of every 8 MB block.
2. XilFPGA copies a 56 KB chunk from the first 8 MB block to Read Buffer.
3. XilFPGA calculates hash on 56 KB and stores in HashsOfChunks.
4. XilFPGA repeats steps 1 to 3 until the entire 8 MB of block is completed.

   ***Note*:** The chunk that XilFPGA copies can be of any size. A 56 KB chunk is taken for better performance.
5. XilFPGA authenticates the 8 MB Bitstream chunk.
6. Once the authentication is successful, XilFPGA starts copying information in batches of 56 KB starting from the first block which is located in DDR memory to Read Buffer, calculates the hash, and then compares it with the hash stored at HashsOfChunks.
7. If the hash comparison is successful, FSBL transmits data to PCAP using DMA (for un-encrypted Bitstream) or AES (if encryption is enabled).
8. XilFPGA repeats steps 6 and 7 until the entire 8 MB block is completed.
9. Repeats steps 1 through 8 for all the blocks of Bitstream.

***Note*:** You can perform warm restart even when the FSBL OCM memory is used to authenticate the Bitstream. PMU stores the FSBL image in the PMU reserved DDR memory which is visible and accessible only to the PMU and restores back to the OCM when APU-only restart needs to be performed. PMU uses the SHA3 hash to validate the FSBL image integrity before restoring the image to OCM (PMU takes care of only image integrity and not confidentiality).

Loading an Authenticated and Encrypted Bitstream using DDR Memory Controller

The software workflow for authenticating bitstream is as follows:

1. XilFPGA identifies DDR-secure bitstream image base address.

2. XilFPGA calculates hash for the first 8 MB block.
3. XilFPGA authenticates the 8 MB block while stored in the external DDR memory.
4. If Authentication is successful, XilFPGA transmits data to PCAP via DMA (for unencrypted Bitstream) or AES (if encryption is enabled).
5. Repeats steps 1 through 4 for all the blocks of bitstream.

Chapter 3

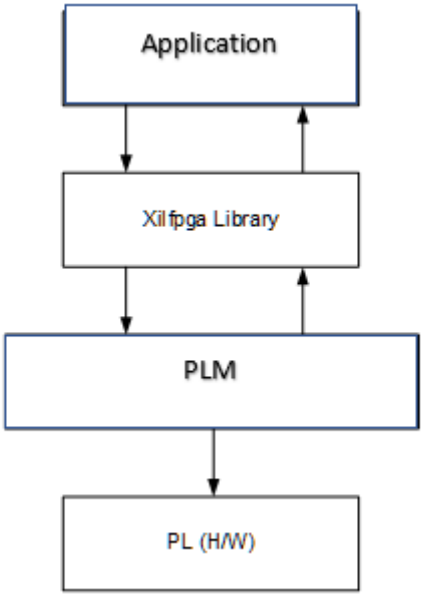
Versal ACAP XilFPGA Library

The library, when used for Versal ACAP, runs on top of Xilinx standalone BSPs. It is tested for Arm Cortex A72 and Arm Cortex-R5F. The most common use-case is that the user can run this on either Arm Cortex-A72 or Arm Cortex-R5F and requests PLM to load the bitstream (PDI) on to the PL. In Versal the bitstream always comes in the format of PDI file.

Design Summary

The following figure shows the flow diagram of how an user application interacts with XilFPGA interacts and other SW components for loading the bitstream from DDR to the PL region.

Figure 6: **XilFPGA Design Summary**



BSP Configuration Settings

XilFPGA provides the following user configuration BSP settings.

Table 2: **BSP Configuration Settings**

Parameter Name	Type	Default Value	Description
base_address	int	0x80000	Holds the bitstream image address. This flag is valid only for the Cortex-A72 or the Cortex-R5F processors.

Setting up the Software System

To use XilFPGA in a software application, you must first compile the XilFPGA library as part of software application.

1. Click **File→ New→ Platform Project**.
2. Click **Specify** to create a new Hardware Platform Specification.
3. Provide a new name for the domain in the Project name field if you wish to override the default value.
4. Select the location for the board support project files. To use the default location, as displayed in the Location field, leave the Use default location check box selected. Otherwise, deselect the checkbox and then type or browse to the directory location.
5. From the Hardware Platform drop-down choose the appropriate platform for your application or click the New button to browse to an existing Hardware Platform.
6. Select the target CPU from the drop-down list.
7. From the Board Support Package OS list box, select the type of board support package to create. A description of the platform types displays in the box below the drop-down list.
8. Click Finish. The wizard creates a new software platform and displays it in the Vitis Navigator pane.
9. Select **Project→Build Automatically** to automatically build the board support package. The Board Support Package Settings dialog box opens. Here you can customize the settings for the domain.
10. Click OK to accept the settings, build the platform, and close the dialog box.
11. From the Explorer, double-click **platform.spr** file and select the appropriate domain/board support package. The overview page opens.
12. In the overview page, click **Modify BSP Settings**.
13. Using the Board Support Package Settings page, you can select the OS Version and which of the Supported Libraries are to be enabled in this domain/BSP.
14. Select the xilfpga and xilmailbox library from the list of Supported Libraries.
15. Expand the Overview tree and select xilfpga. The configuration options for xilfpga are listed.

16. Configure the xilfpga by providing the base address of the bitstream file (DDR address) and the size (in bytes).
17. Click **OK**. The board support package automatically builds with XilFPGA library included in it.
18. Double-click the `system.mss` file to open it in the Editor view.
19. Scroll-down and locate the Libraries section.
20. Click **Import Examples** adjacent to the XilFPGA entry.

XilFPGA APIs

This section provides detailed descriptions of the XilFPGA library APIs.

XilFPGA error = Lower-level errors + Interface-specific errors + XilFPGA top-layer errors

Table 3: **XilFPGA Error**

Lower-level Errors (other libraries or drivers used by XilFPGA)	Interface-specific Errors (PCAP Interface)	XilFPGA Top-layer Errors
31 - 16 bits	15 - 8 bits	7 - 0 bits

- **XilFPGA Top Layer:** The functionality exist in this layers is completely interface agnostic. It provides a unique interface to load the Bitstream across multiple platforms.
- **Interface Specific Layer:** This layer is responsible for providing the interface-specific errors. In case of Zynq UltraScale+ MPSoC, it provides the errors related to PCAP interface.
- **XilFPGA Lower Layer:** This layer is responsible for providing the error related to the lower level drivers used by interface layer.

XilFPGA APIs for Versal ACAPs and Zynq UltraScale+ MPSoCs

The following APIs are supported by Versal ACAPs and Zynq UltraScale+ MPSoCs.

Table 4: **Quick Function Reference**

Type	Name	Arguments
u32	XFpga_Initialize	XFpga * InstancePtr

Table 4: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
u32	XFpga_PL_BitStream_Load	XFpga * InstancePtr UINTPTR BitstreamImageAddr UINTPTR AddrPtr_Size u32 Flags
u32	XFpga_BitStream_Load	XFpga * InstancePtr UINTPTR BitstreamImageAddr UINTPTR KeyAddr u32 Size u32 Flags
u32	XFpga_PL_ValidateImage	XFpga * InstancePtr UINTPTR BitstreamImageAddr UINTPTR AddrPtr_Size u32 Flags
u32	XFpga_ValidateImage	XFpga * InstancePtr UINTPTR BitstreamImageAddr UINTPTR KeyAddr u32 Size u32 Flags
u32	XFpga_PL_Preconfig	XFpga * InstancePtr
u32	XFpga_PL_Write	XFpga * InstancePtr UINTPTR BitstreamImageAddr UINTPTR AddrPtr_Size u32 Flags
u32	XFpga_Write_PI	XFpga * InstancePtr UINTPTR BitstreamImageAddr UINTPTR KeyAddr u32 Size u32 Flags
u32	XFpga_PL_PostConfig	XFpga * InstancePtr

Functions

XFpga_PL_BitStream_Load

The API is used to load the bitstream file into the PL region.

It supports the Vivado-generated bitstream(*.bit, *.bin) and Bootgen-generated bitstream(*.bin) loading, Passing valid bitstream size(Size) information is mandatory for Vivado-generated bitstream, For Bootgen-generated bitstreams bitstream size is taken from the bitstream header.

***Note*:**

- This API will be deprecated in the 2022.1 release. Use the updated ' `XFpga_BitStream_Load()` ' API to perform the same functionality.

Prototype

```
u32 XFpga_PL_BitStream_Load(XFpga *InstancePtr, UINTPTR BitstreamImageAddr,
UINTPTR AddrPtr_Size, u32 Flags);
```

Parameters

The following table lists the `XFpga_PL_BitStream_Load` function arguments.

Table 5: **XFpga_PL_BitStream_Load Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure.
UINTPTR	BitstreamImageAddr	Linear memory bitstream image base address
UINTPTR	AddrPtr_Size	Aes key address which is used for decryption (or) In none secure bitstream used it is used store size of bitstream image.

Table 5: **XFpga_PL_BitStream_Load Arguments** *(cont'd)*

Type	Name	Description
u32	Flags	Flags are used to specify the type of bitstream file. • BIT(0) - Bitstream type ◦ 0 - Full bitstream ◦ 1 - Partial bitstream ◦ 1 - Enable • BIT(1) - Authentication using DDR ◦ 1 - Enable ◦ 0 - Disable • BIT(2) - Authentication using OCM ◦ 1 - Enable ◦ 0 - Disable • BIT(3) - User-key Encryption ◦ 1 - Enable ◦ 0 - Disable • BIT(4) - Device-key Encryption ◦ 1 - Enable ◦ 0 - Disable

Returns

- XFPGA_SUCCESS on success
- Error code on failure.
- XFPGA_VALIDATE_ERROR.
- XFPGA_PRE_CONFIG_ERROR.
- XFPGA_WRITE_BITSTREAM_ERROR.
- XFPGA_POST_CONFIG_ERROR.

XFpga_BitStream_Load

The API is used to load the bitstream file into the PL region.

It supports the Vivado-generated bitstream(*.bit, *.bin) and Bootgen-generated bitstream(*.bin) loading, Passing valid bitstream size(Size) information is mandatory for Vivado-generated bitstream, For Bootgen-generated bitstreams bitstream size is taken from the bitstream header.

Prototype

```
u32 XFpga_BitStream_Load(XFpga *InstancePtr, UINTPTR BitstreamImageAddr,
UINTPTR KeyAddr, u32 Size, u32 Flags);
```

Parameters

The following table lists the `XFpga_BitStream_Load` function arguments.

Table 6: **XFpga_BitStream_Load Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure.
UINTPTR	BitstreamImageAddr	Linear memory bitstream image base address
UINTPTR	KeyAddr	Aes key address which is used for decryption.
u32	Size	Used to store size of bitstream image.
u32	Flags	Flags are used to specify the type of bitstream file. • BIT(0) - Bitstream type ◦ 0 - Full bitstream ◦ 1 - Partial bitstream • BIT(1) - Authentication using DDR ◦ 1 - Enable ◦ 0 - Disable • BIT(2) - Authentication using OCM ◦ 1 - Enable ◦ 0 - Disable • BIT(3) - User-key Encryption ◦ 1 - Enable ◦ 0 - Disable • BIT(4) - Device-key Encryption ◦ 1 - Enable ◦ 0 - Disable

Returns

- XFPGA_SUCCESS on success
- Error code on failure.
- XFPGA_VALIDATE_ERROR.
- XFPGA_PRE_CONFIG_ERROR.

- XFPGA_WRITE_BITSTREAM_ERROR.
- XFPGA_POST_CONFIG_ERROR.

XFpga_PL_ValidateImage

This function is used to validate the bitstream image.

***Note*:**

- This API will be deprecated in the 2022.1 release. Use the updated ' `XFpga_ValidateImage()` ' API to perform the same functionality.

Prototype

```
u32 XFpga_PL_ValidateImage(XFpga *InstancePtr, UINTPTR BitstreamImageAddr,
UINTPTR AddrPtr_Size, u32 Flags);
```

Parameters

The following table lists the `XFpga_PL_ValidateImage` function arguments.

Table 7: **XFpga_PL_ValidateImage Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure
UINTPTR	BitstreamImageAddr	Linear memory bitstream image base address
UINTPTR	AddrPtr_Size	Aes key address which is used for decryption (or) In none secure bitstream used it is used store size of bitstream image.

Table 7: **XFpga_PL_ValidateImage Arguments** *(cont'd)*

Type	Name	Description
u32	Flags	Flags are used to specify the type of bitstream file. • BIT(0) - bitstream type ◦ 0 - Full bitstream ◦ 1 - Partial bitstream • BIT(1) - Authentication using DDR ◦ 1 - Enable ◦ 0 - Disable • BIT(2) - Authentication using OCM ◦ 1 - Enable ◦ 0 - Disable • BIT(3) - User-key Encryption ◦ 1 - Enable ◦ 0 - Disable • BIT(4) - Device-key Encryption ◦ 1 - Enable ◦ 0 - Disable

Returns

Codes as mentioned in xilfpga.h

XFpga_ValidateImage

This function is used to validate the bitstream image.

Prototype

```
u32 XFpga_ValidateImage(XFpga *InstancePtr, UINTPTR BitstreamImageAddr,
UINTPTR KeyAddr, u32 Size, u32 Flags);
```

Parameters

The following table lists the `XFpga_ValidateImage` function arguments.

Table 8: **XFpga_ValidateImage Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure

Table 8: **XFpga_ValidateImage Arguments** *(cont'd)*

Type	Name	Description
UINTPTR	BitstreamImageAddr	Linear memory bitstream image base address
UINTPTR	KeyAddr	Aes key address which is used for decryption.
u32	Size	Used to store size of bitstream image.
u32	Flags	Flags are used to specify the type of bitstream file. • BIT(0) - Bitstream type ◦ 0 - Full bitstream ◦ 1 - Partial bitstream • BIT(1) - Authentication using DDR ◦ 1 - Enable ◦ 0 - Disable • BIT(2) - Authentication using OCM ◦ 1 - Enable ◦ 0 - Disable • BIT(3) - User-key Encryption ◦ 1 - Enable ◦ 0 - Disable • BIT(4) - Device-key Encryption ◦ 1 - Enable ◦ 0 - Disable

Returns

Codes as mentioned in xilfpga.h

XFpga_PL_Preconfig

This function prepares the FPGA to receive configuration data.

Prototype

```
u32 XFpga_PL_Preconfig(XFpga *InstancePtr);
```

Parameters

The following table lists the `XFpga_PL_Preconfig` function arguments.

Table 9: **XFpga_PL_Preconfig Arguments**

Type	Name	Description
XFpga *	InstancePtr	is the pointer to the XFpga.

Returns

Codes as mentioned in xilfpga.h

XFpga_PL_Write

This function writes the count bytes of configuration data into the PL.

***Note*:**

- This API will be deprecated in the 2022.1 release. Use the updated ' `XFpga_Write_Pl()` ' API to perform the same functionality.

Prototype

```
u32 XFpga_PL_Write(XFpga *InstancePtr, UINTPTR BitstreamImageAddr, UINTPTR
AddrPtr_Size, u32 Flags);
```

Parameters

The following table lists the `XFpga_PL_Write` function arguments.

Table 10: **XFpga_PL_Write Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure
UINTPTR	BitstreamImageAddr	Linear memory bitstream image base address
UINTPTR	AddrPtr_Size	Aes key address which is used for decryption (or) In none secure bitstream used it is used store size of bitstream image.

Table 10: **XFpga_PL_Write Arguments** *(cont'd)*

Type	Name	Description
u32	Flags	Flags are used to specify the type of bitstream file. • BIT(0) - Bitstream type ◦ 0 - Full bitstream ◦ 1 - Partial bitstream • BIT(1) - Authentication using DDR ◦ 1 - Enable ◦ 0 - Disable • BIT(2) - Authentication using OCM ◦ 1 - Enable ◦ 0 - Disable • BIT(3) - User-key Encryption ◦ 1 - Enable ◦ 0 - Disable • BIT(4) - Device-key Encryption ◦ 1 - Enable ◦ 0 - Disable

Returns

Codes as mentioned in xilfpga.h

XFpga_Write_Pl

This function writes the count bytes of configuration data into the PL.

Prototype

```
u32 XFpga_Write_Pl(XFpga *InstancePtr, UINTPTR BitstreamImageAddr, UINTPTR
KeyAddr, u32 Size, u32 Flags);
```

Parameters

The following table lists the `XFpga_Write_Pl` function arguments.

Table 11: **XFpga_Write_Pl Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure

Table 11: **XFpga_Write_Pl Arguments** *(cont'd)*

Type	Name	Description
UINTPTR	BitstreamImageAddr	Linear memory bitstream image base address
UINTPTR	KeyAddr	Aes key address which is used for decryption.
u32	Size	Used to store size of bitstream image.
u32	Flags	Flags are used to specify the type of bitstream file. • BIT(0) - Bitstream type ◦ 0 - Full bitstream ◦ 1 - Partial bitstream • BIT(1) - Authentication using DDR ◦ 1 - Enable ◦ 0 - Disable • BIT(2) - Authentication using OCM ◦ 1 - Enable ◦ 0 - Disable • BIT(3) - User-key Encryption ◦ 1 - Enable ◦ 0 - Disable • BIT(4) - Device-key Encryption ◦ 1 - Enable ◦ 0 - Disable

Returns

Codes as mentioned in xilfpga.h

XFpga_PL_PostConfig

This function sets the FPGA to the operating state after writing.

Prototype

```
u32 XFpga_PL_PostConfig(XFpga *InstancePtr);
```

Parameters

The following table lists the `XFpga_PL_PostConfig` function arguments.

Table 12: **XFpga_PL_PostConfig Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure

Returns

Codes as mentioned in xilfpga.h

XFpga_Initialize

This API, when called, initializes the XFPGA interface with default settings.

Prototype

```
u32 XFpga_Initialize(XFpga *InstancePtr);
```

Parameters

The following table lists the `XFpga_Initialize` function arguments.

Table 13: **XFpga_Initialize Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure.

Returns

Returns Status

- XFPGA_SUCCESS on success
- Error code on failure

XilFPGA APIs for Zynq UltraScale+ MPSoC

The following APIs are supported only by Zynq UltraScale+ MPSoCs.

Table 14: **Quick Function Reference**

Type	Name	Arguments
u32	XFpga_GetPlConfigData	XFpga * InstancePtr UINTPTR ReadbackAddr u32 NumFrames

Table 14: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
u32	XFpga_GetPlConfigReg	XFpga * InstancePtr UINTPTR ReadbackAddr u32 ConfigRegAddr
u32	XFpga_InterfaceStatus	XFpga * InstancePtr

Functions

XFpga_GetPlConfigData

This function provides functionality to read back the PL configuration data.

Note:

- This API is not supported for the Versal platform.

Prototype

```
u32 XFpga_GetPlConfigData(XFpga *InstancePtr, UINTPTR ReadbackAddr, u32
NumFrames);
```

Parameters

The following table lists the `XFpga_GetPlConfigData` function arguments.

Table 15: **XFpga_GetPlConfigData Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure
UINTPTR	ReadbackAddr	Address which is used to store the PL readback data.
u32	NumFrames	The number of FPGA configuration frames to read.

Returns

- XFPGA_SUCCESS, if successful
- XFPGA_FAILURE, if unsuccessful
- XFPGA_OPS_NOT_IMPLEMENTED, if implementation not exists.

XFpga_GetPlConfigReg

This function provides PL specific configuration register values.

Note:

- This API is not supported for the Versal platform.

Prototype

```
u32 XFpga_GetPlConfigReg(XFpga *InstancePtr, UINTPTR ReadbackAddr, u32
ConfigRegAddr);
```

Parameters

The following table lists the `XFpga_GetPlConfigReg` function arguments.

Table 16: **XFpga_GetPlConfigReg Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure
UINTPTR	ReadbackAddr	Address which is used to store the PL Configuration register data.
u32	ConfigRegAddr	Configuration register address as mentioned in the UG570.

Returns

- XFPGA_SUCCESS if, successful
- XFPGA_FAILURE if, unsuccessful
- XFPGA_OPS_NOT_IMPLEMENTED, if implementation not exists.

XFpga_InterfaceStatus

This function provides the status of the PL programming interface.

Note:

- This API is not supported for the Versal platform.

Prototype

```
u32 XFpga_InterfaceStatus(XFpga *InstancePtr);
```

Parameters

The following table lists the `XFpga_InterfaceStatus` function arguments.

Table 17: **XFpga_InterfaceStatus Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure

Returns

Status of the PL programming interface

XILINX

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright