Xilinx Standalone Library Documentation

OS and Libraries Document Collection

UG643 (v2020.1) September 4, 2020

Table of Contents

Chapter 1

Xilinx OS and Libraries Overview

The Vitis™ Unified Software Development Environment provides a variety of Xilinx software packages, including drivers, libraries, board support packages, and complete operating systems to help you develop a software platform. This document collection provides information on these. Complete documentation for other operating systems can be found in the their respective reference guides. Device drivers are documented along with the corresponding peripheral documentation. The documentation is listed in the following table; click the name to open the document.

Library Name	Summary
Chapter 2: Xilinx Standard C Libraries	Describes the software libraries available for the embedded processors.
Chapter 3: Standalone Library v7.2	Describes the Standalone platform, a single-threaded, simple operating system (OS) platform that provides the lowest layer of software modules used to access processor-specific functions. Some typical functions offered by the Standalone platform include setting up the interrupts and exceptions systems, configuring caches, and other hardware specific functions. The Hardware Abstraction Layer (HAL) is described in this document.
Chapter 4: LwIP 2.1.1 Library	Describes the SDK port of the third party networking library, Light Weight IP (lwIP) for embedded processors.
Chapter 5: XilIsf Library v5.15	Describes the In System Flash hardware library, which enables higher-layer software (such as an application) to communicate with the Isf. XilIsf supports the Xilinx In-System Flash and external Serial Flash memories from Atmel (AT45XXXD), Spansion(S25FLXX), Winbond W25QXX, and Micron N25QXX.
Chapter 6: XilFFS Library v4.3	Xilffs is a generic FAT file system that is primarily added for use with SD/eMMC driver. The file system is open source and a glue layer is implemented to link it to the SD/eMMC driver.
Chapter 7: XilSecure Library v4.2	The XilSecure library provides APIs to access secure hardware on the Zynq UltraScale+ MPSoC devices.
Chapter 8: XilSkey Library v4.9	The XilSKey library provides a programming mechanism for user-defined eFUSE bits and for programming the KEY into battery-backed RAM (BBRAM) of Zynq SoC, provides programming mechanisms for eFUSE bits of UltraScale devices. The library also provides programming mechanisms for eFUSE bits and BBRAM key of the Zynq UltraScale+ MPSoC devices.
Chapter 9: XilPM Library v3.1	The Zynq UltraScale+ MPSoC power management framework is a set of power management options, based upon an implementation of the extensible energy management interface (EEMI). The power management framework allows software components running across different processing units (PUs) on a chip or device to issue or respond to requests for power management.
Chapter 10: XilFPGA Library v5.2	The XilFPGA library provides an interface to the Linux or bare-metal users for configuring the programmable logic (PL) over PCAP from PS. The library is designed for Zynq UltraScale+ MPSoC devices to run on top of Xilinx standalone BSPs.
Chapter 11: XilMailbox v1.2	The XilMailbox library provides the top-level hooks for sending or receiving an inter-processor interrupt (IPI) message using the Zynq UltraScale+ MPSoC IPI hardware.

About the Libraries

The Standard C support library consists of the `newlib`, `libc`, which contains the standard C functions such as `stdio`, `stdlib`, and `string` routines. The math library is an enhancement over the `newlib` math library, `libm`, and provides the standard math routines. The LibXil libraries consist of the following:

- LibXil Driver (Xilinx device drivers)
- XilMFS (Xilinx memory file system)
- XilFlash (a parallel flash programming library)
- Xilisf (a serial flash programming library)

The Hardware Abstraction Layer (HAL) provides common functions related to register IO, exception, and cache. These common functions are uniform across MicroBlaze™ and Cortex A9 processors. The Standalone platform document provides some processor specific functions and macros for accessing the processor-specific features. Most routines in the library are written in C and can be ported to any platform. User applications must include appropriate headers and link with required libraries for proper compilation and inclusion of required functionality. These libraries and their corresponding include files are created in the processor `\lib` and `\include` directories, under the current project, respectively. The `-I` and `-L` options of the compiler being used should be leveraged to add these directories to the search paths.

Library Organization

Your application can interface with the components in a variety of ways. The libraries are independent of each other, with the exception of some interactions. The LibXil drivers and the Standalone form the lowermost hardware abstraction layer. The library and OS components rely on standard C library components. The math library, `libm.a` is also available for linking with the user applications.

Note: "LibXil Drivers" are the device drivers included in the software platform to provide an interface to the peripherals in the system. These drivers are provided along with the Vitis Unified Software Development Environment and are configured by Libgen. This document collection contains a section that briefly discusses the concept of device drivers and the way they integrate with the board support package in Vitis.

Taking into account some restrictions and implications, which are described in the reference guides for each component, you can mix and match the component libraries.

Xilinx Standard C Libraries

The Vitis™ Unified Software Development Environment libraries and device drivers provide standard C library functions, as well as functions to access peripherals. The SDK libraries are automatically configured based on the Microprocessor Software Specification (MSS) file. These libraries and include files are saved in the current project lib and include directories, respectively. The -I and -L options of mb-gcc are used to add these directories to its library search paths.

Standard C Library (libc.a)

The standard C library, `libc.a`, contains the standard C functions compiled for the MicroBlaze™ processor or the Cortex A9 processor. You can find the header files corresponding to these C standard functions in the `<XILINX_SDK>/gnu/<processor>/<platform>/<processor-lib>/include` folder, where:

- <Vitis> is the Vitis Unified Software Development Environment installation path
- <processor> is either ARM or MicroBlaze
- <platform> is either Solaris (sol), or Windows (nt), or Linux (lin)
- <processor-lib> is either arm-xilinx-eabi or microblaze-xilinx-elf

The `lib.c` directories and functions are:

```
_ansi.h    fastmath.h machine/   reent.h    stdlib.h  utime.h   _syslist.h
fcntl.h    malloc.h   regdef.h   string.h   utmp.h    ar.h      float.h
math.h     setjmp.h   sys/       assert.h   grp.h     paths.h   signal.h
termios.h  ctype.h    ieeefp.h   process.h  stdarg.h  time.h    dirent.h
imits.h    pthread.h  stddef.h   nctrl.h    errno.h   locale.h  pwd.h
stdio.h    unistd.h
```

Programs accessing standard C library functions must be compiled as follows:

- For MicroBlaze processors:

  ```
  mb-gcc <C files>
  ```

- For Cortex A9 processors:

  ```
  arm-xilinx-eabi-gcc <C files>
  ```

The `libc` library is included automatically. For programs that access `libm` math functions, specify the `lm` option. For more information on the C runtime library, see *MicroBlaze Processor Reference Guide* (UG081).

Xilinx C Library (libxil.a)

The Xilinx C library, `libxil.a`, contains the following object files for the MicroBlaze processor embedded processor:

- _exception_handler.o
- _interrupt_handler.o
- _program_clean.o
- _program_init.o

Default exception and interrupt handlers are provided. The `libxil.a` library is included automatically. Programs accessing Xilinx C library functions must be compiled as follows:

```
mb-gcc <C files>
```

Memory Management Functions

The MicroBlaze processor and Cortex A9 processor C libraries support the standard memory management functions such as `malloc()`, `calloc()`, and `free()`. Dynamic memory allocation provides memory from the program heap. The heap pointer starts at low memory and grows toward high memory. The size of the heap cannot be increased at runtime. Therefore an appropriate value must be provided for the heap size at compile time. The `malloc()` function requires the heap to be at least 128 bytes in size to be able to allocate memory dynamically (even if the dynamic requirement is less than 128 bytes).

***Note*:** The return value of malloc must always be checked to ensure that it could actually allocate the memory requested.

Arithmetic Operations

Software implementations of integer and floating point arithmetic is available as library routines in libgcc.a for both processors. The compiler for both the processors inserts calls to these routines in the code produced, in case the hardware does not support the arithmetic primitive with an instruction.

Details of the software implementations of integer and floating point arithmetic for MicroBlaze processors are listed below:

Integer Arithmetic

By default, integer multiplication is done in software using the library function `__mulsi3`. Integer multiplication is done in hardware if the `-mno-xl-soft-mul` mb-gcc option is specified. Integer divide and mod operations are done in software using the library functions `__divsi3` and `__modsi3`. The MicroBlaze processor can also be customized to use a hard divider, in which case the `div` instruction is used in place of the `__divsi3` library routine. Double precision multiplication, division and mod functions are carried out by the library functions `__muldi3`, `__divdi3`, and `__moddi3` respectively. The unsigned version of these operations correspond to the signed versions described above, but are prefixed with an `__u` instead of `__`.

Floating Point Arithmetic

All floating point addition, subtraction, multiplication, division, and conversions are implemented using software functions in the C library.

Thread Safety

The standard C library provided with SDK is not built for a multi-threaded environment. STDIO functions like `printf()`, `scanf()` and memory management functions like `malloc()` and `free()` are common examples of functions that are not thread-safe. When using the C library in a multi-threaded environment, proper mutual exclusion techniques must be used to protect thread unsafe functions.

Input/Output Functions

The SDK libraries contains standard C functions for I/O, such as printf and scanf. These functions are large and might not be suitable for embedded processors. The prototypes for these functions are available in the stdio.h file.

These Input/Output routines require that a newline is terminated with both a CR and LF. Ensure that your terminal CR/LF behavior corresponds to this requirement.

***Note*:** The C standard I/O routines such as printf, scanf, vfprintf are, by default, line buffered. To change the buffering scheme to no buffering, you must call setvbuf appropriately. For example:

```
setvbuf (stdout, NULL, _IONBF, 0);
```

These Input/Output routines require that a newline is terminated with both a CR and LF. Ensure that your terminal CR/LF behavior corresponds to this requirement.

For more information on setting the standard input and standard output devices for a system, see *Embedded System Tools Reference Manual* (UG1043). In addition to the standard C functions, the SDK processors library provides the following smaller I/O functions:

Table 1: **Quick Function Reference**

Type	Name	Arguments
void	print	void
void	putnum	void
void	xil_printf	void

Functions

print

This function prints a string to the peripheral designated as standard output in the Microprocessor Software Specification (MSS) file. This function outputs the passed string as is and there is no interpretation of the string passed. For example, a \n passed is interpreted as a new line character and not as a carriage return and a new line as is the case with ANSI C printf function.

Prototype

```
void print(char *);
```

putnum

This function converts an integer to a hexadecimal string and prints it to the peripheral designated as standard output in the MSS file.

Prototype

```
void putnum(int);
```

xil_printf

`xil_printf()` is a light-weight implementation of printf. It is much smaller in size (only 1 Kb). It does not have support for floating point numbers. `xil_printf()` also does not support printing of long (such as 64-bit) numbers.

About format string support:

The format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %, and ends with a conversion specifier.

In between there can be (in order) zero or more flags, an optional minimum field width and an optional precision. Supported flag characters are:

The character % is followed by zero or more of the following flags:

- 0 The value should be zero padded. For d, x conversions, the converted value is padded on the left with zeros rather than blanks. If the 0 and - flags both appear, the 0 flag is ignored.
- - The converted value is to be left adjusted on the field boundary. (The default is right justification.) Except for n conversions, the converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A - overrides a 0 if both are given.

About supported field widths

Field widths are represented with an optional decimal digit string (with a nonzero in the first digit) specifying a minimum field width. If the converted value has fewer characters than the field width, it is padded with spaces on the left (or right, if the left-adjustment flag has been given). The supported conversion specifiers are:

- d The int argument is converted to signed decimal notation.
- l The int argument is converted to a signed long notation.
- x The unsigned int argument is converted to unsigned hexadecimal notation. The letters abcdef are used for x conversions.
- c The int argument is converted to an unsigned char, and the resulting character is written.
- s The const char* argument is expected to be a pointer to an array of character type (pointer to a string).

Characters from the array are written up to (but not including) a terminating NULL character; if a precision is specified, no more than the number specified are written. If a precision s given, no null character need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating NULL character.

Prototype

```
void xil_printf(const *char ctrl1,...);
```

Chapter 3

Standalone Library v7.2

Xilinx Hardware Abstraction Layer API

This section describes the Xilinx Hardware Abstraction Layer API, These APIs are applicable for all processors supported by Xilinx.

Assert APIs and Macros

The xil_assert.h file contains assert related functions and macros. Assert APIs/Macros specifies that a application program satisfies certain conditions at particular points in its execution. These function can be used by application programs to ensure that, application code is satisfying certain conditions.

Table 2: **Quick Function Reference**

Type	Name	Arguments
void	Xil_Assert	file line
void	XNullHandler	void * NullParameter
void	Xil_AssertSetCallback	routine

Functions

Xil_Assert

Implement assert.

Currently, it calls a user-defined callback function if one has been set. Then, it potentially enters an infinite loop depending on the value of the Xil_AssertWait variable.

***Note*:** None.

Prototype

```
void Xil_Assert(const char8 *File, s32 Line);
```

Parameters

The following table lists the `Xil_Assert` function arguments.

Table 3: **Xil_Assert Arguments**

Name	Description
file	filename of the source
line	linenumber within File

Returns

None.

XNullHandler

Null handler function.

This follows the XInterruptHandler signature for interrupt handlers. It can be used to assign a null handler (a stub) to an interrupt controller vector table.

***Note*:** None.

Prototype

```
void XNullHandler(void *NullParameter);
```

Parameters

The following table lists the `XNullHandler` function arguments.

Table 4: **XNullHandler Arguments**

Name	Description
NullParameter	arbitrary void pointer and not used.

Returns

None.

Xil_AssertSetCallback

Set up a callback function to be invoked when an assert occurs.

If a callback is already installed, then it will be replaced.

***Note*:** This function has no effect if NDEBUG is set

Prototype

```
void Xil_AssertSetCallback(Xil_AssertCallback Routine);
```

Parameters

The following table lists the `Xil_AssertSetCallback` function arguments.

Table 5: **Xil_AssertSetCallback Arguments**

Name	Description
routine	callback to be invoked when an assert is taken

Returns

None.

Register IO interfacing APIs

The xil_io.h file contains the interface for the general I/O component, which encapsulates the Input/Output functions for the processors that do not require any special I/O handling.

Table 6: **Quick Function Reference**

Type	Name	Arguments
u16	Xil_EndianSwap16	u16 Data
u32	Xil_EndianSwap32	u32 Data
INLINE u8	Xil_In8	UINTPTR Addr
INLINE u16	Xil_In16	UINTPTR Addr
INLINE u32	Xil_In32	UINTPTR Addr
INLINE u64	Xil_In64	UINTPTR Addr
INLINE void	Xil_Out8	UINTPTR Addr u8 Value

Table 6: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
INLINE void	Xil_Out16	UINTPTR Addr u16 Value
INLINE void	Xil_Out32	UINTPTR Addr u32 Value
INLINE void	Xil_Out64	UINTPTR Addr u64 Value
INLINE u32	Xil_SecureOut32	UINTPTR Addr u32 Value
INLINE u16	Xil_In16BE	void
INLINE u32	Xil_In32BE	void
INLINE void	Xil_Out16BE	void
INLINE void	Xil_Out32BE	void

Functions

Xil_EndianSwap16

Perform a 16-bit endian conversion.

Prototype

```
u16 Xil_EndianSwap16(u16 Data);
```

Parameters

The following table lists the `Xil_EndianSwap16` function arguments.

Table 7: **Xil_EndianSwap16 Arguments**

Name	Description
Data	16 bit value to be converted

Returns

16 bit Data with converted endianness

Xil_EndianSwap32

Perform a 32-bit endian conversion.

Prototype

```
u32 Xil_EndianSwap32(u32 Data);
```

Parameters

The following table lists the `Xil_EndianSwap32` function arguments.

Table 8: **Xil_EndianSwap32 Arguments**

Name	Description
Data	32 bit value to be converted

Returns

32 bit data with converted endianness

Xil_In8

Performs an input operation for a memory location by reading from the specified address and returning the 8 bit Value read from that address.

Prototype

```
INLINE u8 Xil_In8(UINTPTR Addr);
```

Parameters

The following table lists the `Xil_In8` function arguments.

Table 9: **Xil_In8 Arguments**

Name	Description
Addr	contains the address to perform the input operation

Returns

The 8 bit Value read from the specified input address.

Xil_In16

Performs an input operation for a memory location by reading from the specified address and returning the 16 bit Value read from that address.

Prototype

```
INLINE u16 Xil_In16(UINTPTR Addr);
```

Parameters

The following table lists the `Xil_In16` function arguments.

Table 10: **Xil_In16 Arguments**

Name	Description
Addr	contains the address to perform the input operation

Returns

The 16 bit Value read from the specified input address.

Xil_In32

Performs an input operation for a memory location by reading from the specified address and returning the 32 bit Value read from that address.

Prototype

```
INLINE u32 Xil_In32(UINTPTR Addr);
```

Parameters

The following table lists the `Xil_In32` function arguments.

Table 11: **Xil_In32 Arguments**

Name	Description
Addr	contains the address to perform the input operation

Returns

The 32 bit Value read from the specified input address.

Xil_In64

Performs an input operation for a memory location by reading the 64 bit Value read from that address.

Prototype

```
INLINE u64 Xil_In64(UINTPTR Addr);
```

Parameters

The following table lists the `Xil_In64` function arguments.

Table 12: **Xil_In64 Arguments**

Name	Description
Addr	contains the address to perform the input operation

Returns

The 64 bit Value read from the specified input address.

Xil_Out8

Performs an output operation for an memory location by writing the 8 bit Value to the the specified address.

Prototype

```
INLINE void Xil_Out8(UINTPTR Addr, u8 Value);
```

Parameters

The following table lists the `Xil_Out8` function arguments.

Table 13: **Xil_Out8 Arguments**

Name	Description
Addr	contains the address to perform the output operation
Value	contains the 8 bit Value to be written at the specified address.

Returns

None.

Xil_Out16

Performs an output operation for a memory location by writing the 16 bit Value to the the specified address.

Prototype

```
INLINE void Xil_Out16(UINTPTR Addr, u16 Value);
```

Parameters

The following table lists the `Xil_Out16` function arguments.

Table 14: **Xil_Out16 Arguments**

Name	Description
Addr	contains the address to perform the output operation
Value	contains the Value to be written at the specified address.

Returns

None.

Xil_Out32

Performs an output operation for a memory location by writing the 32 bit Value to the the specified address.

Prototype

```
INLINE void Xil_Out32(UINTPTR Addr, u32 Value);
```

Parameters

The following table lists the `Xil_Out32` function arguments.

Table 15: **Xil_Out32 Arguments**

Name	Description
Addr	contains the address to perform the output operation
Value	contains the 32 bit Value to be written at the specified address.

Returns

None.

Xil_Out64

Performs an output operation for a memory location by writing the 64 bit Value to the the specified address.

Prototype

```
INLINE void Xil_Out64(UINTPTR Addr, u64 Value);
```

Parameters

The following table lists the `Xil_Out64` function arguments.

Table 16: **Xil_Out64 Arguments**

Name	Description
Addr	contains the address to perform the output operation
Value	contains 64 bit Value to be written at the specified address.

Returns

None.

Xil_SecureOut32

Performs an output operation for a memory location by writing the 32 bit Value to the the specified address and then reading it back to verify the value written in the register.

Prototype

```
INLINE u32 Xil_SecureOut32(UINTPTR Addr, u32 Value);
```

Parameters

The following table lists the `Xil_SecureOut32` function arguments.

Table 17: **Xil_SecureOut32 Arguments**

Name	Description
Addr	contains the address to perform the output operation
Value	contains 32 bit Value to be written at the specified address

Returns

Returns Status

- XST_SUCCESS on success

- XST_FAILURE on failure

Definitions for available xilinx platforms

The xplatform_info.h file contains definitions for various available Xilinx platforms. Also, it contains prototype of APIs, which can be used to get the platform information.

Table 18: **Quick Function Reference**

Type	Name	Arguments
u32	XGetPlatform_Info	None.

Functions

XGetPlatform_Info

This API is used to provide information about platform.

Prototype

```
u32 XGetPlatform_Info(void);
```

Parameters

The following table lists the `XGetPlatform_Info` function arguments.

Table 19: **XGetPlatform_Info Arguments**

Name	Description
None.	

Returns

The information about platform defined in xplatform_info.h

Data types for Xilinx Software IP Cores

The xil_types.h file contains basic types for Xilinx software IP. These data types are applicable for all processors supported by Xilinx.

Customized APIs for Memory Operations

The xil_mem.h file contains prototype for functions related to memory operations. These APIs are applicable for all processors supported by Xilinx.

Table 20: **Quick Function Reference**

Type	Name	Arguments
void	Xil_MemCpy	void * dst const void * src u32 cnt

Functions

Xil_MemCpy

This function copies memory from once location to other.

Prototype

```
void Xil_MemCpy(void *dst, const void *src, u32 cnt);
```

Parameters

The following table lists the `Xil_MemCpy` function arguments.

Table 21: **Xil_MemCpy Arguments**

Name	Description
dst	pointer pointing to destination memory
src	pointer pointing to source memory
cnt	32 bit length of bytes to be copied

Xilinx software status codes

The xstatus.h file contains the Xilinx software status codes.These codes are used throughout the Xilinx device drivers.

Test Utilities for Memory and Caches

The xil_testcache.h, xil_testio.h and the xil_testmem.h files contain utility functions to test cache and memory. Details of supported tests and subtests are listed below.

The xil_testcache.h file contains utility functions to test cache.

The xil_testio.h file contains utility functions to test endian related memory IO functions.

A subset of the memory tests can be selected or all of the tests can be run in order. If there is an error detected by a subtest, the test stops and the failure code is returned. Further tests are not run even if all of the tests are selected.

The xil_testmem.h file contains utility functions to test memory. A subset of the memory tests can be selected or all of the tests can be run in order. If there is an error detected by a subtest, the test stops and the failure code is returned. Further tests are not run even if all of the tests are selected. Following list describes the supported memory tests:

- XIL_TESTMEM_ALLMEMTESTS: This test runs all of the subtests.
- XIL_TESTMEM_INCREMENT: This test starts at 'XIL_TESTMEM_INIT_VALUE' and uses the incrementing value as the test value for memory.
- XIL_TESTMEM_WALKONES: Also known as the Walking ones test. This test uses a walking '1' as the test value for memory.

```
location 1 = 0x00000001
location 2 = 0x00000002
...
```

- XIL_TESTMEM_WALKZEROS: Also known as the Walking zero's test. This test uses the inverse value of the walking ones test as the test value for memory.

```
location 1 = 0xFFFFFFFE
location 2 = 0xFFFFFFFD
...
```

- XIL_TESTMEM_INVERSEADDR: Also known as the inverse address test. This test uses the inverse of the address of the location under test as the test value for memory.
- XIL_TESTMEM_FIXEDPATTERN: Also known as the fixed pattern test. This test uses the provided patters as the test value for memory. If zero is provided as the pattern the test uses '0xDEADBEEF".

CAUTION! *The tests are **DESTRUCTIVE**. Run before any initialized memory spaces have been set up. The address provided to the memory tests is not checked for validity except for the NULL case. It is possible to provide a code-space pointer for this test to start with and ultimately destroy executable code causing random failures.*

Note: Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** width, the patterns used in XIL_TESTMEM_WALKONES and XIL_TESTMEM_WALKZEROS will repeat on a boundary of a power of two making it more difficult to detect addressing errors. The XIL_TESTMEM_INCREMENT and XIL_TESTMEM_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

Table 22: **Quick Function Reference**

Type	Name	Arguments
s32	Xil_TestDCacheRange	void

Table 22: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
s32	Xil_TestDCacheAll	void
s32	Xil_TestICacheRange	void
s32	Xil_TestICacheAll	void
s32	Xil_TestIO8	u8 * Addr s32 Length u8 Value
s32	Xil_TestIO16	u16 * Addr s32 Length u16 Value s32 Kind s32 Swap
s32	Xil_TestIO32	u32 * Addr s32 Length u32 Value s32 Kind s32 Swap

Functions

Xil_TestIO8

Perform a destructive 8-bit wide register IO test where the register is accessed using Xil_Out8 and Xil_In8, and comparing the written values by reading them back.

Prototype

```
s32 Xil_TestIO8(u8 *Addr, s32 Length, u8 Value);
```

Parameters

The following table lists the `Xil_TestIO8` function arguments.

Table 23: **Xil_TestIO8 Arguments**

Name	Description
Addr	a pointer to the region of memory to be tested.
Length	Length of the block.

Table 23: **Xil_TestIO8 Arguments** *(cont'd)*

Name	Description
Value	constant used for writing the memory.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

Xil_TestIO16

Perform a destructive 16-bit wide register IO test.

Each location is tested by sequentially writing a 16-bit wide register, reading the register, and comparing value. This function tests three kinds of register IO functions, normal register IO, little-endian register IO, and big-endian register IO. When testing little/big-endian IO, the function performs the following sequence, Xil_Out16LE/Xil_Out16BE, Xil_In16, Compare In-Out values, Xil_Out16, Xil_In16LE/Xil_In16BE, Compare In-Out values. Whether to swap the read-in value before comparing is controlled by the 5th argument.

Prototype

```
s32 Xil_TestIO16(u16 *Addr, s32 Length, u16 Value, s32 Kind, s32 Swap);
```

Parameters

The following table lists the `Xil_TestIO16` function arguments.

Table 24: **Xil_TestIO16 Arguments**

Name	Description
Addr	a pointer to the region of memory to be tested.
Length	Length of the block.
Value	constant used for writing the memory.
Kind	Type of test. Acceptable values are: XIL_TESTIO_DEFAULT, XIL_TESTIO_LE, XIL_TESTIO_BE.
Swap	indicates whether to byte swap the read-in value.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

Xil_TestIO32

Perform a destructive 32-bit wide register IO test.

Each location is tested by sequentially writing a 32-bit wide register, reading the register, and comparing value. This function tests three kinds of register IO functions, normal register IO, little-endian register IO, and big-endian register IO. When testing little/big-endian IO, the function perform the following sequence, Xil_Out32LE/ Xil_Out32BE, Xil_In32, Compare, Xil_Out32, Xil_In32LE/Xil_In32BE, Compare. Whether to swap the read-in value *before comparing is controlled by the 5th argument.

Prototype

```
s32 Xil_TestIO32(u32 *Addr, s32 Length, u32 Value, s32 Kind, s32 Swap);
```

Parameters

The following table lists the `Xil_TestIO32` function arguments.

Table 25: **Xil_TestIO32 Arguments**

Name	Description
Addr	a pointer to the region of memory to be tested.
Length	Length of the block.
Value	constant used for writing the memory.
Kind	type of test. Acceptable values are: XIL_TESTIO_DEFAULT, XIL_TESTIO_LE, XIL_TESTIO_BE.
Swap	indicates whether to byte swap the read-in value.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

MicroBlaze Processor API

This section provides a linked summary and detailed descriptions of the MicroBlaze Processor APIs.

MicroBlaze Pseudo-asm Macros and Interrupt Handling APIs

MicroBlaze BSP includes macros to provide convenient access to various registers in the MicroBlaze processor. Some of these macros are very useful within exception handlers for retrieving information about the exception.Also, the interrupt handling functions help manage interrupt handling on MicroBlaze processor devices.To use these functions, include the header file mb_interface.h in your source code

Table 26: **Quick Function Reference**

Type	Name	Arguments
void	microblaze_enable_interrupts	void
void	microblaze_disable_interrupts	void
void	microblaze_enable_icache	void
void	microblaze_disable_icache	void
void	microblaze_enable_dcache	void
void	microblaze_disable_dcache	void
void	microblaze_enable_exceptions	void
void	microblaze_disable_exceptions	void
void	microblaze_register_handler	`XInterruptHandler` Handler void * DataPtr
void	microblaze_register_exception_handler	u32 ExceptionId Top void * DataPtr
void	microblaze_invalidate_icache	void
void	microblaze_invalidate_dcache	void
void	microblaze_flush_dcache	void
void	microblaze_invalidate_icache_range	void

Table 26: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	microblaze_invalidate_dcache_range	void
void	microblaze_flush_dcache_range	void
void	microblaze_scrub	void
void	microblaze_invalidate_cache_ext	void
void	microblaze_flush_cache_ext	void
void	microblaze_flush_cache_ext_range	void
void	microblaze_invalidate_cache_ext_range	void
void	microblaze_update_icache	void
void	microblaze_init_icache_range	void
void	microblaze_update_dcache	void
void	microblaze_init_dcache_range	void

Functions

microblaze_register_handler

Registers a top-level interrupt handler for the MicroBlaze.

The argument provided in this call as the DataPtr is used as the argument for the handler when it is called.

Prototype

```
void microblaze_register_handler(XInterruptHandler Handler, void *DataPtr);
```

Parameters

The following table lists the `microblaze_register_handler` function arguments.

Table 27: **microblaze_register_handler Arguments**

Name	Description
Handler	Top level handler.
DataPtr	a reference to data that will be passed to the handler when it gets called.

Returns

None.

microblaze_register_exception_handler

Registers an exception handler for the MicroBlaze.

The argument provided in this call as the DataPtr is used as the argument for the handler when it is called.

None.

Note:

Prototype

```
void microblaze_register_exception_handler(u32 ExceptionId,
Xil_ExceptionHandler Handler, void *DataPtr);
```

Parameters

The following table lists the `microblaze_register_exception_handler` function arguments.

Table 28: **microblaze_register_exception_handler Arguments**

Name	Description
ExceptionId	is the id of the exception to register this handler for.
Top	level handler.
DataPtr	is a reference to data that will be passed to the handler when it gets called.

Returns

None.

MicroBlaze exception APIs

The xil_exception.h file contains MicroBlaze specific exception related APIs and macros. Application programs can use these APIs/Macros for various exception related operations (i.e. enable exception, disable exception, register exception handler etc.)

***Note*:** To use exception related functions, the xil_exception.h file must be added in source code

Table 29: **Quick Function Reference**

Type	Name	Arguments
void	microblaze_enable_exceptions	void
void	microblaze_disable_exceptions	void
void	microblaze_enable_interrupts	void
void	microblaze_disable_interrupts	void
void	Xil_ExceptionNullHandler	void * Data
void	Xil_ExceptionInit	None.
void	Xil_ExceptionEnable	void
void	Xil_ExceptionDisable	None.
void	Xil_ExceptionRegisterHandler	u32 Id `Xil_ExceptionHandler` Handler void * Data
void	Xil_ExceptionRemoveHandler	u32 Id

Functions

Xil_ExceptionNullHandler

This function is a stub handler that is the default handler that gets called if the application has not setup a handler for a specific exception.

The function interface has to match the interface specified for a handler even though none of the arguments are used.

Prototype

```
void Xil_ExceptionNullHandler(void *Data);
```

Parameters

The following table lists the `Xil_ExceptionNullHandler` function arguments.

Table 30: **Xil_ExceptionNullHandler Arguments**

Name	Description
Data	unused by this function.

Xil_ExceptionInit

Initialize exception handling for the processor.

The exception vector table is setup with the stub handler for all exceptions.

Prototype

```
void Xil_ExceptionInit(void);
```

Parameters

The following table lists the `Xil_ExceptionInit` function arguments.

Table 31: **Xil_ExceptionInit Arguments**

Name	Description
None.	

Returns

None.

Xil_ExceptionEnable

Enable Exceptions.

Prototype

```
void Xil_ExceptionEnable(void);
```

Returns

None.

Xil_ExceptionDisable

Disable Exceptions.

Prototype

```
void Xil_ExceptionDisable(void);
```

Parameters

The following table lists the `Xil_ExceptionDisable` function arguments.

Table 32: **Xil_ExceptionDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_ExceptionRegisterHandler

Makes the connection between the Id of the exception source and the associated handler that is to run when the exception is recognized.

The argument provided in this call as the DataPtr is used as the argument for the handler when it is called.

Prototype

```
void Xil_ExceptionRegisterHandler(u32 Id, Xil_ExceptionHandler Handler,
void *Data);
```

Parameters

The following table lists the `Xil_ExceptionRegisterHandler` function arguments.

Table 33: **Xil_ExceptionRegisterHandler Arguments**

Name	Description
Id	contains the 32 bit ID of the exception source and should be XIL_EXCEPTION_INT or be in the range of 0 to XIL_EXCEPTION_LAST. See xil_mach_exception.h for further information.
Handler	handler function to be registered for exception
Data	a reference to data that will be passed to the handler when it gets called.

Xil_ExceptionRemoveHandler

Removes the handler for a specific exception Id.

The stub handler is then registered for this exception Id.

Prototype

```
void Xil_ExceptionRemoveHandler(u32 Id);
```

Parameters

The following table lists the `Xil_ExceptionRemoveHandler` function arguments.

Table 34: **Xil_ExceptionRemoveHandler Arguments**

Name	Description
Id	contains the 32 bit ID of the exception source and should be XIL_EXCEPTION_INT or in the range of 0 to XIL_EXCEPTION_LAST. See xexception_l.h for further information.

MicroBlaze Processor FSL Macros

MicroBlaze BSP includes macros to provide convenient access to accelerators connected to the MicroBlaze Fast Simplex Link (FSL) Interfaces.To use these functions, include the header file fsl.h in your source code

MicroBlaze PVR access routines and macros

MicroBlaze processor v5.00.a and later versions have configurable Processor Version Registers (PVRs). The contents of the PVR are captured using the pvr_t data structure, which is defined as an array of 32-bit words, with each word corresponding to a PVR register on hardware. The number of PVR words is determined by the number of PVRs configured in the hardware. You should not attempt to access PVR registers that are not present in hardware, as the pvr_t data structure is resized to hold only as many PVRs as are present in hardware. To access information in the PVR:

1. Use the `microblaze_get_pvr()` function to populate the PVR data into a pvr_t data structure.
2. In subsequent steps, you can use any one of the PVR access macros list to get individual data stored in the PVR.
3. pvr.h header file must be included to source to use PVR macros.

Table 35: **Quick Function Reference**

Type	Name	Arguments
int	microblaze_get_pvr	pvr-

Functions

microblaze_get_pvr

Populate the PVR data structure to which pvr points, with the values of the hardware PVR registers.

Prototype

```
int microblaze_get_pvr(pvr_t *pvr);
```

Parameters

The following table lists the `microblaze_get_pvr` function arguments.

Table 36: **microblaze_get_pvr Arguments**

Name	Description
pvr-	address of PVR data structure to be populated

Returns

0 - SUCCESS -1 - FAILURE

Sleep Routines for MicroBlaze

The microblaze_sleep.h file contains microblaze sleep APIs. These APIs provides delay for requested duration.

***Note*:** The microblaze_sleep.h file may contain architecture-dependent items.

Table 37: **Quick Function Reference**

Type	Name	Arguments
void	MB_Sleep	MilliSeconds-

Functions

MB_Sleep

Provides delay for requested duration.

***Note*:** Instruction cache should be enabled for this to work.

Prototype

```
void MB_Sleep(u32 MilliSeconds) __attribute__((__deprecated__));
```

Parameters

The following table lists the `MB_Sleep` function arguments.

Table 38: **MB_Sleep Arguments**

Name	Description
MilliSeconds-	Delay time in milliseconds.

Returns

None.

Cortex R5 Processor API

Standalone BSP contains boot code, cache, exception handling, file and memory management, configuration, time and processor-specific include functions. It supports gcc compiler. This section provides a linked summary and detailed descriptions of the Cortex R5 processor APIs.

Cortex R5 Processor Boot Code

The boot code performs minimum configuration which is required for an application to run starting from processor's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling
2. Program stack pointer for various modes (IRQ, FIQ, supervisor, undefine, abort, system)
3. Disable instruction cache, data cache and MPU
4. Invalidate instruction and data cache
5. Configure MPU with short descriptor translation table format and program base address of translation table
6. Enable data cache, instruction cache and MPU
7. Enable Floating point unit
8. Transfer control to _start which clears BSS sections and jumping to main application

Cortex R5 Processor MPU specific APIs

MPU functions provides access to MPU operations such as enable MPU, disable MPU and set attribute for section of memory. Boot code invokes Init_MPU function to configure the MPU. A total of 10 MPU regions are allocated with another 6 being free for users. Overview of the memory attributes for different MPU regions is as given below,

	Memory Range	Attributes of MPURegion
DDR	0x00000000 - 0x7FFFFFFF	Normal write-back Cacheable
PL	0x80000000 - 0xBFFFFFFF	Strongly Ordered
QSPI	0xC0000000 - 0xDFFFFFFF	Device Memory
PCIe	0xE0000000 - 0xEFFFFFFF	Device Memory
STM_CORESIGHT	0xF8000000 - 0xF8FFFFFF	Device Memory
RPU_R5_GIC	0xF9000000 - 0xF90FFFFF	Device memory
FPS	0xFD000000 - 0xFDFFFFFF	Device Memory
LPS	0xFE000000 - 0xFFFFFFFF	Device Memory
OCM	0xFFFC0000 - 0xFFFFFFFF	Normal write-back Cacheable

***Note*:** For a system where DDR is less than 2GB, region after DDR and before PL is marked as undefined in translation table. Memory range 0xFE000000-0xFEFFFFFF is allocated for upper LPS slaves, where as memory region 0xFF000000-0xFFFFFFFF is allocated for lower LPS slaves.

Table 39: **Quick Function Reference**

Type	Name	Arguments
void	Xil_SetTlbAttributes	INTPTR Addr u32 attrib
void	Xil_EnableMPU	None.
void	Xil_DisableMPU	None.
u32	Xil_SetMPURegion	Addr u64 size u32 attrib
u32	Xil_UpdateMPUConfig	u32 reg_num INTPTR address u32 size u32 attrib
void	Xil_GetMPUConfig	XMpu_Config mpuconfig
u32	Xil_GetNumOfFreeRegions	none
u32	Xil_GetNextMPURegion	none
u32	Xil_DisableMPURegionByRegNum	u32 reg_num
u16	Xil_GetMPUFreeRegMask	none

Table 39: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
u32	Xil_SetMPURegionByRegNum	u32 reg_num address u64 size u32 attrib
void *	Xil_MemMap	void

Functions

Xil_SetTlbAttributes

This function sets the memory attributes for a section covering 1MB, of memory in the translation table.

Prototype

```
void Xil_SetTlbAttributes(INTPTR Addr, u32 attrib);
```

Parameters

The following table lists the `Xil_SetTlbAttributes` function arguments.

Table 40: **Xil_SetTlbAttributes Arguments**

Name	Description
Addr	32-bit address for which memory attributes need to be set.
attrib	Attribute for the given memory region.

Returns

None.

Xil_EnableMPU

Enable MPU for Cortex R5 processor.

This function invalidates I cache and flush the D Caches, and then enables the MPU.

Prototype

```
void Xil_EnableMPU(void);
```

Parameters

The following table lists the `Xil_EnableMPU` function arguments.

Table 41: **Xil_EnableMPU Arguments**

Name	Description
None.	

Returns

None.

Xil_DisableMPU

Disable MPU for Cortex R5 processors.

This function invalidates I cache and flush the D Caches, and then disabes the MPU.

Prototype

```
void Xil_DisableMPU(void);
```

Parameters

The following table lists the `Xil_DisableMPU` function arguments.

Table 42: **Xil_DisableMPU Arguments**

Name	Description
None.	

Returns

None.

Xil_SetMPURegion

Set the memory attributes for a section of memory in the translation table.

Prototype

```
u32 Xil_SetMPURegion(INTPTR addr, u64 size, u32 attrib);
```

Parameters

The following table lists the `Xil_SetMPURegion` function arguments.

Table 43: **Xil_SetMPURegion Arguments**

Name	Description
Addr	32-bit address for which memory attributes need to be set..
size	size is the size of the region.
attrib	Attribute for the given memory region.

Returns

None.

Xil_UpdateMPUConfig

Update the MPU configuration for the requested region number in the global MPU configuration table.

Prototype

```
u32 Xil_UpdateMPUConfig(u32 reg_num, INTPTR address, u32 size, u32 attrib);
```

Parameters

The following table lists the `Xil_UpdateMPUConfig` function arguments.

Table 44: **Xil_UpdateMPUConfig Arguments**

Name	Description
reg_num	The requested region number to be updated information for.
address	32 bit address for start of the region.
size	Requested size of the region.
attrib	Attribute for the corresponding region.

Returns

XST_FAILURE: When the requested region number if 16 or more. XST_SUCCESS: When the MPU configuration table is updated.

Xil_GetMPUConfig

The MPU configuration table is passed to the caller.

Prototype

```
void Xil_GetMPUConfig(XMpu_Config mpuconfig);
```

Parameters

The following table lists the `Xil_GetMPUConfig` function arguments.

Table 45: **Xil_GetMPUConfig Arguments**

Name	Description
mpuconfig	This is of type XMpu_Config which is an array of 16 entries of type structure representing the MPU config table

Returns

none

Xil_GetNumOfFreeRegions

Returns the total number of free MPU regions available.

Prototype

```
u32 Xil_GetNumOfFreeRegions(void);
```

Parameters

The following table lists the `Xil_GetNumOfFreeRegions` function arguments.

Table 46: **Xil_GetNumOfFreeRegions Arguments**

Name	Description
none	

Returns

Number of free regions available to users

Xil_GetNextMPURegion

Returns the next available free MPU region.

Prototype

```
u32 Xil_GetNextMPURegion(void);
```

Parameters

The following table lists the `Xil_GetNextMPURegion` function arguments.

Table 47: **Xil_GetNextMPURegion Arguments**

Name	Description
none	

Returns

The free MPU region available

Xil_DisableMPURegionByRegNum

Disables the corresponding region number as passed by the user.

Prototype

```
u32 Xil_DisableMPURegionByRegNum(u32 reg_num);
```

Parameters

The following table lists the `Xil_DisableMPURegionByRegNum` function arguments.

Table 48: **Xil_DisableMPURegionByRegNum Arguments**

Name	Description
reg_num	The region number to be disabled

Returns

XST_SUCCESS: If the region could be disabled successfully XST_FAILURE: If the requested region number is 16 or more.

Xil_GetMPUFreeRegMask

Returns the total number of free MPU regions available in the form of a mask.

A bit of 1 in the returned 16 bit value represents the corresponding region number to be available. For example, if this function returns 0xC0000, this would mean, the regions 14 and 15 are available to users.

Prototype

```
u16 Xil_GetMPUFreeRegMask(void);
```

Parameters

The following table lists the `Xil_GetMPUFreeRegMask` function arguments.

Table 49: **Xil_GetMPUFreeRegMask Arguments**

Name	Description
none	

Returns

The free region mask as a 16 bit value

Xil_SetMPURegionByRegNum

Enables the corresponding region number as passed by the user.

Prototype

```
u32 Xil_SetMPURegionByRegNum(u32 reg_num, INTPTR addr, u64 size, u32
attrib);
```

Parameters

The following table lists the `Xil_SetMPURegionByRegNum` function arguments.

Table 50: **Xil_SetMPURegionByRegNum Arguments**

Name	Description
reg_num	The region number to be enabled
address	32 bit address for start of the region.
size	Requested size of the region.
attrib	Attribute for the corresponding region.

Returns

XST_SUCCESS: If the region could be created successfully XST_FAILURE: If the requested region number is 16 or more.

Cortex R5 Processor Cache Functions

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Table 51: **Quick Function Reference**

Type	Name	Arguments
void	Xil_DCacheEnable	None.
void	Xil_DCacheDisable	None.
void	Xil_DCacheInvalidate	None.
void	Xil_DCacheInvalidateRange	INTPTR adr u32 len
void	Xil_DCacheFlush	None.
void	Xil_DCacheFlushRange	INTPTR adr u32 len
void	Xil_DCacheInvalidateLine	INTPTR adr
void	Xil_DCacheFlushLine	INTPTR adr
void	Xil_DCacheStoreLine	INTPTR adr
void	Xil_ICacheEnable	None.
void	Xil_ICacheDisable	None.
void	Xil_ICacheInvalidate	None.
void	Xil_ICacheInvalidateRange	INTPTR adr u32 len
void	Xil_ICacheInvalidateLine	INTPTR adr

Functions

Xil_DCacheEnable

Enable the Data cache.

***Note*:** None.

Prototype

```
void Xil_DCacheEnable(void);
```

Parameters

The following table lists the `Xil_DCacheEnable` function arguments.

Table 52: **Xil_DCacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheDisable(void);
```

Parameters

The following table lists the `Xil_DCacheDisable` function arguments.

Table 53: **Xil_DCacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidate

Invalidate the entire Data cache.

Prototype

```
void Xil_DCacheInvalidate(void);
```

Parameters

The following table lists the `Xil_DCacheInvalidate` function arguments.

Table 54: **Xil_DCacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidateRange

Invalidate the Data cache for the given address range.

If the bytes specified by the address (adr) are cached by the Data cache,the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Prototype

```
void Xil_DCacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 55: **Xil_DCacheInvalidateRange Arguments**

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of range to be invalidated in bytes.

Returns

None.

Xil_DCacheFlush

Flush the entire Data cache.

Prototype

```
void Xil_DCacheFlush(void);
```

Parameters

The following table lists the `Xil_DCacheFlush` function arguments.

Table 56: **Xil_DCacheFlush Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheFlushRange

Flush the Data cache for the given address range.

If the bytes specified by the address (adr) are cached by the Data cache, the cacheline containing those bytes is invalidated.If the cacheline is modified (dirty), the written to system memory before the lines are invalidated.

Prototype

```
void Xil_DCacheFlushRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheFlushRange` function arguments.

Table 57: **Xil_DCacheFlushRange Arguments**

Name	Description
adr	32bit start address of the range to be flushed.
len	Length of the range to be flushed in bytes

Returns

None.

Xil_DCacheInvalidateLine

Invalidate a Data cache line.

If the byte specified by the address (adr) is cached by the data cache, the cacheline containing that byte is invalidated.If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheInvalidateLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheInvalidateLine` function arguments.

Table 58: **Xil_DCacheInvalidateLine Arguments**

Name	Description
adr	32bit address of the data to be flushed.

Returns

None.

Xil_DCacheFlushLine

Flush a Data cache line.

If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheFlushLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheFlushLine` function arguments.

Table 59: **Xil_DCacheFlushLine Arguments**

Name	Description
adr	32bit address of the data to be flushed.

Returns

None.

Xil_DCacheStoreLine

Store a Data cache line.

If the byte specified by the address (adr) is cached by the Data cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory.After the store completes, the cacheline is marked as unmodified (not dirty).

***Note*:** The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheStoreLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheStoreLine` function arguments.

Table 60: **Xil_DCacheStoreLine Arguments**

Name	Description
adr	32bit address of the data to be stored

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

Prototype

```
void Xil_ICacheEnable(void);
```

Parameters

The following table lists the `Xil_ICacheEnable` function arguments.

Table 61: **Xil_ICacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Prototype

```
void Xil_ICacheDisable(void);
```

Parameters

The following table lists the `Xil_ICacheDisable` function arguments.

Table 62: **Xil_ICacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

Prototype

```
void Xil_ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_ICacheInvalidate` function arguments.

Table 63: **Xil_ICacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidateRange

Invalidate the instruction cache for the given address range.

If the bytes specified by the address (adr) are cached by the Data cache, the cacheline containing that byte is invalidated. If the cachelineis modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Prototype

```
void Xil_ICacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 64: **Xil_ICacheInvalidateRange Arguments**

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_ICacheInvalidateLine

Invalidate an instruction cache line.If the instruction specified by the address is cached by the instruction cache, the cacheline containing that instruction is invalidated.

***Note*:** The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_ICacheInvalidateLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_ICacheInvalidateLine` function arguments.

Table 65: **Xil_ICacheInvalidateLine Arguments**

Name	Description
adr	32bit address of the instruction to be invalidated.

Returns

None.

Cortex R5 Time Functions

The xtime_l.h provides access to 32-bit TTC timer counter. These functions can be used by applications to track the time.

Table 66: **Quick Function Reference**

Type	Name	Arguments
void	XTime_SetTime	XTime Xtime_Global
void	XTime_GetTime	XTime * Xtime_Global

Functions

XTime_SetTime

TTC Timer runs continuously and the time can not be set as desired.

This API doesn't contain anything. It is defined to have uniformity across platforms.

Note: In multiprocessor environment reference time will reset/lost for all processors, when this function called by any one processor.

Prototype

```
void XTime_SetTime(XTime Xtime_Global);
```

Parameters

The following table lists the `XTime_SetTime` function arguments.

Table 67: **XTime_SetTime Arguments**

Name	Description
Xtime_Global	32 bit value to be written to the timer counter register.

Returns

None.

XTime_GetTime

Get the time from the timer counter register.

Prototype

```
void XTime_GetTime(XTime *Xtime_Global);
```

Parameters

The following table lists the `XTime_GetTime` function arguments.

Table 68: **XTime_GetTime Arguments**

Name	Description
Xtime_Global	Pointer to the 32 bit location to be updated with the time current value of timer counter register.

Returns

None.

Cortex R5 Event Counters Functions

Cortex R5 event counter functions can be utilized to configure and control the Cortex-R5 performance monitor events. Cortex-R5 Performance Monitor has 3 event counters which can be used to count a variety of events described in Coretx-R5 TRM. The xpm_counter.h file defines configurations XPM_CNTRCFGx which can be used to program the event counters to count a set of events.

Table 69: **Quick Function Reference**

Type	Name	Arguments
void	Xpm_SetEvents	s32 PmcrCfg
void	Xpm_GetEventCounters	u32 * PmCtrValue
u32	Xpm_DisableEvent	Event
u32	Xpm_SetUpAnEvent	Event
u32	Xpm_GetEventCounter	Event Pointer
void	Xpm_DisableEventCounters	None.
void	Xpm_EnableEventCounters	None.
void	Xpm_ResetEventCounters	None.
void	Xpm_SleepPerfCounter	u32 delay u64 frequency

Functions

Xpm_SetEvents

This function configures the Cortex R5 event counters controller, with the event codes, in a configuration selected by the user and enables the counters.

Prototype

```
void Xpm_SetEvents(s32 PmcrCfg);
```

Parameters

The following table lists the `Xpm_SetEvents` function arguments.

Table 70: **Xpm_SetEvents Arguments**

Name	Description
PmcrCfg	Configuration value based on which the event counters are configured.XPM_CNTRCFG* values defined in xpm_counter.h can be utilized for setting configuration

Returns

None.

Xpm_GetEventCounters

This function disables the event counters and returns the counter values.

Prototype

```
void Xpm_GetEventCounters(u32 *PmCtrValue);
```

Parameters

The following table lists the `Xpm_GetEventCounters` function arguments.

Table 71: **Xpm_GetEventCounters Arguments**

Name	Description
PmCtrValue	Pointer to an array of type u32 PmCtrValue[6]. It is an output parameter which is used to return the PM counter values.

Returns

None.

Xpm_DisableEvent

Disables the requested event counter.

***Note*:** None.

Prototype

```
u32 Xpm_DisableEvent(u32 EventHandlerId);
```

Parameters

The following table lists the `Xpm_DisableEvent` function arguments.

Table 72: **Xpm_DisableEvent Arguments**

Name	Description
Event	Counter ID. The counter ID is the same that was earlier returned through a call to Xpm_SetUpAnEvent. Cortex-R5 supports only 3 counters. The valid values are 0, 1, or 2.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if the passed Counter ID is invalid (i.e. greater than 2).

Xpm_SetUpAnEvent

Sets up one of the event counters to count events based on the Event ID passed.

For supported Event IDs please refer xpm_counter.h. Upon invoked, the API searches for an available counter. After finding one, it sets up the counter to count events for the requested event.

***Note*:** None.

Prototype

```
u32 Xpm_SetUpAnEvent(u32 EventID);
```

Parameters

The following table lists the `Xpm_SetUpAnEvent` function arguments.

Table 73: **Xpm_SetUpAnEvent Arguments**

Name	Description
Event	ID. For valid values, please refer xpm_counter.h.

Returns

- Counter Number if successful. For Cortex-R5, valid return values are 0, 1, or 2.
- XPM_NO_COUNTERS_AVAILABLE (0xFF) if all counters are being used

Xpm_GetEventCounter

Reads the counter value for the requested counter ID.

This is used to read the number of events that has been counted for the requsted event ID. This can only be called after a call to Xpm_SetUpAnEvent.

***Note*:** None.

Prototype

```
u32 Xpm_GetEventCounter(u32 EventHandlerId, u32 *CntVal);
```

Parameters

The following table lists the `Xpm_GetEventCounter` function arguments.

Table 74: **Xpm_GetEventCounter Arguments**

Name	Description
Event	Counter ID. The counter ID is the same that was earlier returned through a call to Xpm_SetUpAnEvent. Cortex-R5 supports only 3 counters. The valid values are 0, 1, or 2.
Pointer	to a 32 bit unsigned int type. This is used to return the event counter value.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if the passed Counter ID is invalid (i.e. greater than 2).

Xpm_DisableEventCounters

This function disables the Cortex R5 event counters.

Prototype

```
void Xpm_DisableEventCounters(void);
```

Parameters

The following table lists the `Xpm_DisableEventCounters` function arguments.

Table 75: **Xpm_DisableEventCounters Arguments**

Name	Description
None.	

Returns

None.

Xpm_EnableEventCounters

This function enables the Cortex R5 event counters.

Prototype

```
void Xpm_EnableEventCounters(void);
```

Parameters

The following table lists the `Xpm_EnableEventCounters` function arguments.

Table 76: **Xpm_EnableEventCounters Arguments**

Name	Description
None.	

Returns

None.

Xpm_ResetEventCounters

This function resets the Cortex R5 event counters.

Prototype

```
void Xpm_ResetEventCounters(void);
```

Parameters

The following table lists the `Xpm_ResetEventCounters` function arguments.

Table 77: **Xpm_ResetEventCounters Arguments**

Name	Description
None.	

Returns

None.

Xpm_SleepPerfCounter

This is helper function used by sleep/usleep APIs to generate delay in sec/usec.

Prototype

```
void Xpm_SleepPerfCounter(u32 delay, u64 frequency);
```

Parameters

The following table lists the `Xpm_SleepPerfCounter` function arguments.

Table 78: **Xpm_SleepPerfCounter Arguments**

Name	Description
delay	- delay time in sec/usec
frequency	- Number of countes in second/micro second

Returns

None.

Cortex R5 Processor Specific Include Files

The xpseudo_asm.h includes xreg_cortexr5.h and xpseudo_asm_gcc.h.

The xreg_cortexr5.h file contains definitions for inline assembler code. It provides inline definitions for Cortex R5 GPRs, SPRs,co-processor registers and Debug register

The xpseudo_asm_gcc.h contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization,or cache manipulation. These inline assembler instructions can be used from drivers and user applications written in C.

Cortex R5 peripheral definitions

The xparameters_ps.h file contains the canonical definitions and constant declarations for peripherals within hardblock, attached to the ARM Cortex R5 core. These definitions can be used by drivers or applications to access the peripherals.

ARM Processor Common API

This section provides a linked summary and detailed descriptions of the ARM Processor Common APIs.

ARM Processor Exception Handling

ARM processors specific exception related APIs for cortex A53,A9 and R5 can utilized for enabling/disabling IRQ, registering/removing handler for exceptions or initializing exception vector table with null handler.

Table 79: **Quick Function Reference**

Type	Name	Arguments
void	Xil_ExceptionRegisterHandler	exception_id `Xil_ExceptionHandler` Handler void * Data
void	Xil_ExceptionRemoveHandler	exception_id
void	Xil_GetExceptionRegisterHandler	exception_id `Xil_ExceptionHandler` * Handler void ** Data
void	Xil_ExceptionInit	None.
void	Xil_DataAbortHandler	void
void	Xil_PrefetchAbortHandler	void
void	Xil_UndefinedExceptionHandler	void

Functions

Xil_ExceptionRegisterHandler

Register a handler for a specific exception.

This handler is being called when the processor encounters the specified exception.

***Note*:** None.

Prototype

```
void Xil_ExceptionRegisterHandler(u32 Exception_id, Xil_ExceptionHandler
Handler, void *Data);
```

Parameters

The following table lists the `Xil_ExceptionRegisterHandler` function arguments.

Table 80: **Xil_ExceptionRegisterHandler Arguments**

Name	Description
exception_id	contains the ID of the exception source and should be in the range of 0 to XIL_EXCEPTION_ID_LAST. See xil_exception.h for further information.
Handler	to the Handler for that exception.
Data	is a reference to Data that will be passed to the Handler when it gets called.

Returns

None.

Xil_ExceptionRemoveHandler

Removes the Handler for a specific exception Id.

The stub Handler is then registered for this exception Id.

***Note*:** None.

Prototype

```
void Xil_ExceptionRemoveHandler(u32 Exception_id);
```

Parameters

The following table lists the `Xil_ExceptionRemoveHandler` function arguments.

Table 81: **Xil_ExceptionRemoveHandler Arguments**

Name	Description
exception_id	contains the ID of the exception source and should be in the range of 0 to XIL_EXCEPTION_ID_LAST. See xil_exception.h for further information.

Returns

None.

Xil_GetExceptionRegisterHandler

Get a handler for a specific exception.

This handler is being called when the processor encounters the specified exception.

Note: None.

Prototype

```
void Xil_GetExceptionRegisterHandler(u32 Exception_id, Xil_ExceptionHandler
*Handler, void **Data);
```

Parameters

The following table lists the `Xil_GetExceptionRegisterHandler` function arguments.

Table 82: **Xil_GetExceptionRegisterHandler Arguments**

Name	Description
exception_id	contains the ID of the exception source and should be in the range of 0 to XIL_EXCEPTION_ID_LAST. See xil_exception.h for further information.
Handler	to the Handler for that exception.
Data	is a reference to Data that will be passed to the Handler when it gets called.

Returns

None.

Xil_ExceptionInit

The function is a common API used to initialize exception handlers across all supported arm processors.

For ARM Cortex-A53, Cortex-R5, and Cortex-A9, the exception handlers are being initialized statically and this function does not do anything. However, it is still present to take care of backward compatibility issues (in earlier versions of BSPs, this API was being used to initialize exception handlers).

Note: None.

Prototype

```
void Xil_ExceptionInit(void);
```

Parameters

The following table lists the `Xil_ExceptionInit` function arguments.

Table 83: **Xil_ExceptionInit Arguments**

Name	Description
None.	

Returns

None.

Xil_DataAbortHandler

Default Data abort handler which prints data fault status register through which information about data fault can be acquired

Prototype

```
void Xil_DataAbortHandler(void *CallBackRef);
```

Xil_PrefetchAbortHandler

Default Prefetch abort handler which prints prefetch fault status register through which information about instruction prefetch fault can be acquired.

Prototype

```
void Xil_PrefetchAbortHandler(void *CallBackRef);
```

Xil_UndefinedExceptionHandler

Default undefined exception handler which prints address of the undefined instruction if debug prints are enabled.

Prototype

```
void Xil_UndefinedExceptionHandler(void *CallBackRef);
```

Cortex A9 Processor API

Standalone BSP contains boot code, cache, exception handling, file and memory management, configuration, time and processor-specific include functions. It supports gcc compilers.

Cortex A9 Processor Boot Code

The boot code performs minimum configuration which is required for an application to run starting from processor's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling
2. Invalidate instruction cache, data cache and TLBs
3. Program stack pointer for various modes (IRQ, FIQ, supervisor, undefine, abort, system)
4. Configure MMU with short descriptor translation table format and program base address of translation table
5. Enable data cache, instruction cache and MMU
6. Enable Floating point unit
7. Transfer control to _start which clears BSS sections, initializes global timer and runs global constructor before jumping to main application

None.

***Note*:**

translation_table.S contains a static page table required by MMU for cortex-A9. This translation table is flat mapped (input address = output address) with default memory attributes defined for zynq architecture. It utilizes short descriptor translation table format with each section defining 1MB of memory.

The overview of translation table memory attributes is described below.

	Memory Range	Definition in Translation Table
DDR	0x00000000 - 0x3FFFFFFF	Normal write-back Cacheable
PL	0x40000000 - 0xBFFFFFFF	Strongly Ordered
Reserved	0xC0000000 - 0xDFFFFFFF	Unassigned
Memory mapped devices	0xE0000000 - 0xE02FFFFF	Device Memory
Reserved	0xE0300000 - 0xE0FFFFFF	Unassigned
NAND, NOR	0xE1000000 - 0xE3FFFFFF	Device memory
SRAM	0xE4000000 - 0xE5FFFFFF	Normal write-back Cacheable
Reserved	0xE6000000 - 0xF7FFFFFF	Unassigned
AMBA APB Peripherals	0xF8000000 - 0xF8FFFFFF	Device Memory
Reserved	0xF9000000 - 0xFBFFFFFF	Unassigned
Linear QSPI - XIP	0xFC000000 - 0xFDFFFFFF	Normal write-through cacheable
Reserved	0xFE000000 - 0xFFEFFFFF	Unassigned
OCM	0xFFF00000 - 0xFFFFFFFF	Normal inner write-back cacheable

For region 0x00000000 - 0x3FFFFFFF, a system where DDR is less than 1GB, region after DDR and before PL is marked as undefined/reserved in translation table. In 0xF8000000 - 0xF8FFFFFF, 0xF8000C00 - 0xF8000FFF, 0xF8010000 - 0xF88FFFFF and 0xF8F03000 to 0xF8FFFFFF are reserved but due to granual size of 1MB, it is not possible to define separate regions for them. For region 0xFFF00000 - 0xFFFFFFFF, 0xFFF00000 to 0xFFFB0000 is reserved but due to 1MB granual size, it is not possible to define separate region for it

Note:

Cortex A9 Processor Cache Functions

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Table 84: **Quick Function Reference**

Type	Name	Arguments
void	Xil_DCacheEnable	None.
void	Xil_DCacheDisable	None.
void	Xil_DCacheInvalidate	None.
void	Xil_DCacheInvalidateRange	INTPTR adr u32 len
void	Xil_DCacheFlush	None.
void	Xil_DCacheFlushRange	INTPTR adr u32 len
void	Xil_ICacheEnable	None.
void	Xil_ICacheDisable	None.
void	Xil_ICacheInvalidate	None.
void	Xil_ICacheInvalidateRange	INTPTR adr u32 len
void	Xil_DCacheInvalidateLine	u32 adr

Table 84: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	Xil_DCacheFlushLine	u32 adr
void	Xil_DCacheStoreLine	u32 adr
void	Xil_ICacheInvalidateLine	u32 adr
void	Xil_L1DCacheEnable	None.
void	Xil_L1DCacheDisable	None.
void	Xil_L1DCacheInvalidate	None.
void	Xil_L1DCacheInvalidateLine	u32 adr
void	Xil_L1DCacheInvalidateRange	u32 adr u32 len
void	Xil_L1DCacheFlush	None.
void	Xil_L1DCacheFlushLine	u32 adr
void	Xil_L1DCacheFlushRange	u32 adr u32 len
void	Xil_L1DCacheStoreLine	Address
void	Xil_L1ICacheEnable	None.
void	Xil_L1ICacheDisable	None.
void	Xil_L1ICacheInvalidate	None.
void	Xil_L1ICacheInvalidateLine	u32 adr
void	Xil_L1ICacheInvalidateRange	u32 adr u32 len
void	Xil_L2CacheEnable	None.

Table 84: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	Xil_L2CacheDisable	None.
void	Xil_L2CacheInvalidate	None.
void	Xil_L2CacheInvalidateLine	u32 adr
void	Xil_L2CacheInvalidateRange	u32 adr u32 len
void	Xil_L2CacheFlush	None.
void	Xil_L2CacheFlushLine	u32 adr
void	Xil_L2CacheFlushRange	u32 adr u32 len
void	Xil_L2CacheStoreLine	u32 adr

Functions

Xil_DCacheEnable

Enable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheEnable(void);
```

Parameters

The following table lists the `Xil_DCacheEnable` function arguments.

Table 85: **Xil_DCacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheDisable(void);
```

Parameters

The following table lists the `Xil_DCacheDisable` function arguments.

Table 86: **Xil_DCacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidate

Invalidate the entire Data cache.

Note: None.

Prototype

```
void Xil_DCacheInvalidate(void);
```

Parameters

The following table lists the `Xil_DCacheInvalidate` function arguments.

Table 87: **Xil_DCacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidateRange

Invalidate the Data cache for the given address range.

If the bytes specified by the address range are cached by the Data cache, the cachelines containing those bytes are invalidated. If the cachelines are modified (dirty), the modified contents are lost and NOT written to the system memory before the lines are invalidated. data. This issue raises few possibilities. work.

1. Avoid situations where invalidation has to be done after the data is updated by peripheral/DMA directly into the memory. It is not tough to achieve (may be a bit risky). The common use case to do invalidation is when a DMA happens. Generally for such use cases, buffers can be allocated first and then start the DMA. The practice that needs to be followed here is, immediately after buffer allocation and before starting the DMA, do the invalidation. With this approach, invalidation need not to be done after the DMA transfer is over. are brought into cache (between the time it is invalidated and DMA completes) because of some speculative prefetching or reading data for a variable present in the same cache line, then we will have to invalidate the cache after DMA is complete.

***Note*:** None.

Prototype

```
void Xil_DCacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 88: **Xil_DCacheInvalidateRange Arguments**

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_DCacheFlush

Flush the entire Data cache.

***Note*:** None.

Prototype

```
void Xil_DCacheFlush(void);
```

Parameters

The following table lists the `Xil_DCacheFlush` function arguments.

Table 89: **Xil_DCacheFlush Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheFlushRange

Flush the Data cache for the given address range.

If the bytes specified by the address range are cached by the data cache, the cachelines containing those bytes are invalidated. If the cachelines are modified (dirty), they are written to the system memory before the lines are invalidated.

Note: None.

Prototype

```
void Xil_DCacheFlushRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheFlushRange` function arguments.

Table 90: **Xil_DCacheFlushRange Arguments**

Name	Description
adr	32bit start address of the range to be flushed.
len	Length of the range to be flushed in bytes.

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

Note: None.

Prototype

```
void Xil_ICacheEnable(void);
```

Parameters

The following table lists the `Xil_ICacheEnable` function arguments.

Table 91: **Xil_ICacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Note: None.

Prototype

```
void Xil_ICacheDisable(void);
```

Parameters

The following table lists the `Xil_ICacheDisable` function arguments.

Table 92: **Xil_ICacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

Note: None.

Prototype

```
void Xil_ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_ICacheInvalidate` function arguments.

Table 93: **Xil_ICacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidateRange

Invalidate the instruction cache for the given address range.

If the instructions specified by the address range are cached by the instrunction cache, the cachelines containing those instructions are invalidated.

***Note*:** None.

Prototype

```
void Xil_ICacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 94: **Xil_ICacheInvalidateRange Arguments**

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_DCacheInvalidateLine

Invalidate a Data cache line.

If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to the system memory before the line is invalidated.

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheInvalidateLine` function arguments.

Table 95: **Xil_DCacheInvalidateLine Arguments**

Name	Description
adr	32bit address of the data to be flushed.

Returns

None.

Xil_DCacheFlushLine

Flush a Data cache line.

If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheFlushLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheFlushLine` function arguments.

Table 96: **Xil_DCacheFlushLine Arguments**

Name	Description
adr	32bit address of the data to be flushed.

Returns

None.

Xil_DCacheStoreLine

Store a Data cache line.

If the byte specified by the address (adr) is cached by the Data cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

***Note*:** The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheStoreLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheStoreLine` function arguments.

Table 97: **Xil_DCacheStoreLine Arguments**

Name	Description
adr	32bit address of the data to be stored.

Returns

None.

Xil_ICacheInvalidateLine

Invalidate an instruction cache line.

If the instruction specified by the address is cached by the instruction cache, the cacheline containing that instruction is invalidated.

***Note*:** The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_ICacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_ICacheInvalidateLine` function arguments.

Table 98: **Xil_ICacheInvalidateLine Arguments**

Name	Description
adr	32bit address of the instruction to be invalidated.

Returns

None.

Xil_L1DCacheEnable

Enable the level 1 Data cache.

***Note*:** None.

Prototype

```
void Xil_L1DCacheEnable(void);
```

Parameters

The following table lists the `Xil_L1DCacheEnable` function arguments.

Table 99: **Xil_L1DCacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_L1DCacheDisable

Disable the level 1 Data cache.

***Note*:** None.

Prototype

```
void Xil_L1DCacheDisable(void);
```

Parameters

The following table lists the `Xil_L1DCacheDisable` function arguments.

Table 100: **Xil_L1DCacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_L1DCacheInvalidate

Invalidate the level 1 Data cache.

***Note*:** In Cortex A9, there is no cp instruction for invalidating the whole D-cache. This function invalidates each line by set/way.

Prototype

```
void Xil_L1DCacheInvalidate(void);
```

Parameters

The following table lists the `Xil_L1DCacheInvalidate` function arguments.

Table 101: **Xil_L1DCacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_L1DCacheInvalidateLine

Invalidate a level 1 Data cache line.

If the byte specified by the address (Addr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

***Note*:** The bottom 5 bits are set to 0, forced by architecture.

Prototype

```
void Xil_L1DCacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_L1DCacheInvalidateLine` function arguments.

Table 102: **Xil_L1DCacheInvalidateLine Arguments**

Name	Description
adr	32bit address of the data to be invalidated.

Returns

None.

Xil_L1DCacheInvalidateRange

Invalidate the level 1 Data cache for the given address range.

If the bytes specified by the address range are cached by the Data cache, the cachelines containing those bytes are invalidated. If the cachelines are modified (dirty), the modified contents are lost and NOT written to the system memory before the lines are invalidated.

Note: None.

Prototype

```
void Xil_L1DCacheInvalidateRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L1DCacheInvalidateRange` function arguments.

Table 103: **Xil_L1DCacheInvalidateRange Arguments**

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_L1DCacheFlush

Flush the level 1 Data cache.

Note: In Cortex A9, there is no cp instruction for flushing the whole D-cache. Need to flush each line.

Prototype

```
void Xil_L1DCacheFlush(void);
```

Parameters

The following table lists the `Xil_L1DCacheFlush` function arguments.

Table 104: **Xil_L1DCacheFlush Arguments**

Name	Description
None.	

Returns

None.

Xil_L1DCacheFlushLine

Flush a level 1 Data cache line.

If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

***Note*:** The bottom 5 bits are set to 0, forced by architecture.

Prototype

```
void Xil_L1DCacheFlushLine(u32 adr);
```

Parameters

The following table lists the `Xil_L1DCacheFlushLine` function arguments.

Table 105: **Xil_L1DCacheFlushLine Arguments**

Name	Description
adr	32bit address of the data to be flushed.

Returns

None.

Xil_L1DCacheFlushRange

Flush the level 1 Data cache for the given address range.

If the bytes specified by the address range are cached by the Data cache, the cacheline containing those bytes are invalidated. If the cachelines are modified (dirty), they are written to system memory before the lines are invalidated.

Note: None.

Prototype

```
void Xil_L1DCacheFlushRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L1DCacheFlushRange` function arguments.

Table 106: **Xil_L1DCacheFlushRange Arguments**

Name	Description
adr	32bit start address of the range to be flushed.
len	Length of the range to be flushed in bytes.

Returns

None.

Xil_L1DCacheStoreLine

Store a level 1 Data cache line.

If the byte specified by the address (adr) is cached by the Data cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

Note: The bottom 5 bits are set to 0, forced by architecture.

Prototype

```
void Xil_L1DCacheStoreLine(u32 adr);
```

Parameters

The following table lists the `Xil_L1DCacheStoreLine` function arguments.

Table 107: **Xil_L1DCacheStoreLine Arguments**

Name	Description
Address	to be stored.

Returns

None.

Xil_L1ICacheEnable

Enable the level 1 instruction cache.

Note: None.

Prototype

```
void Xil_L1ICacheEnable(void);
```

Parameters

The following table lists the `Xil_L1ICacheEnable` function arguments.

Table 108: **Xil_L1ICacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_L1ICacheDisable

Disable level 1 the instruction cache.

Note: None.

Prototype

```
void Xil_L1ICacheDisable(void);
```

Parameters

The following table lists the `Xil_L1ICacheDisable` function arguments.

Table 109: **Xil_L1ICacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_L1ICacheInvalidate

Invalidate the entire level 1 instruction cache.

Note: None.

Prototype

```
void Xil_L1ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_L1ICacheInvalidate` function arguments.

Table 110: **Xil_L1ICacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_L1ICacheInvalidateLine

Invalidate a level 1 instruction cache line.

If the instruction specified by the address is cached by the instruction cache, the cacheline containing that instruction is invalidated.

Note: The bottom 5 bits are set to 0, forced by architecture.

Prototype

```
void Xil_L1ICacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_L1ICacheInvalidateLine` function arguments.

Table 111: **Xil_L1ICacheInvalidateLine Arguments**

Name	Description
adr	32bit address of the instruction to be invalidated.

Returns

None.

Xil_L1ICacheInvalidateRange

Invalidate the level 1 instruction cache for the given address range.

If the instrucions specified by the address range are cached by the instruction cache, the cacheline containing those bytes are invalidated.

***Note*:** None.

Prototype

```
void Xil_L1ICacheInvalidateRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L1ICacheInvalidateRange` function arguments.

Table 112: **Xil_L1ICacheInvalidateRange Arguments**

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_L2CacheEnable

Enable the L2 cache.

***Note*:** None.

Prototype

```
void Xil_L2CacheEnable(void);
```

Parameters

The following table lists the `Xil_L2CacheEnable` function arguments.

Table 113: **Xil_L2CacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_L2CacheDisable

Disable the L2 cache.

***Note*:** None.

Prototype

```
void Xil_L2CacheDisable(void);
```

Parameters

The following table lists the `Xil_L2CacheDisable` function arguments.

Table 114: **Xil_L2CacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_L2CacheInvalidate

Invalidate the entire level 2 cache.

***Note*:** None.

Prototype

```
void Xil_L2CacheInvalidate(void);
```

Parameters

The following table lists the `Xil_L2CacheInvalidate` function arguments.

Table 115: **Xil_L2CacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_L2CacheInvalidateLine

Invalidate a level 2 cache line.

If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_L2CacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_L2CacheInvalidateLine` function arguments.

Table 116: **Xil_L2CacheInvalidateLine Arguments**

Name	Description
adr	32bit address of the data/instruction to be invalidated.

Returns

None.

Xil_L2CacheInvalidateRange

Invalidate the level 2 cache for the given address range.

If the bytes specified by the address range are cached by the L2 cache, the cacheline containing those bytes are invalidated. If the cachelines are modified (dirty), the modified contents are lost and are NOT written to system memory before the lines are invalidated.

Note: None.

Prototype

```
void Xil_L2CacheInvalidateRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L2CacheInvalidateRange` function arguments.

Table 117: **Xil_L2CacheInvalidateRange Arguments**

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_L2CacheFlush

Flush the entire level 2 cache.

Note: None.

Prototype

```
void Xil_L2CacheFlush(void);
```

Parameters

The following table lists the `Xil_L2CacheFlush` function arguments.

Table 118: **Xil_L2CacheFlush Arguments**

Name	Description
None.	

Returns

None.

Xil_L2CacheFlushLine

Flush a level 2 cache line.

If the byte specified by the address (adr) is cached by the L2 cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_L2CacheFlushLine(u32 adr);
```

Parameters

The following table lists the `Xil_L2CacheFlushLine` function arguments.

Table 119: **Xil_L2CacheFlushLine Arguments**

Name	Description
adr	32bit address of the data/instruction to be flushed.

Returns

None.

Xil_L2CacheFlushRange

Flush the level 2 cache for the given address range.

If the bytes specified by the address range are cached by the L2 cache, the cacheline containing those bytes are invalidated. If the cachelines are modified (dirty), they are written to the system memory before the lines are invalidated.

Note: None.

Prototype

```
void Xil_L2CacheFlushRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L2CacheFlushRange` function arguments.

Table 120: **Xil_L2CacheFlushRange Arguments**

Name	Description
adr	32bit start address of the range to be flushed.
len	Length of the range to be flushed in bytes.

Returns

None.

Xil_L2CacheStoreLine

Store a level 2 cache line.

If the byte specified by the address (adr) is cached by the L2 cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_L2CacheStoreLine(u32 adr);
```

Parameters

The following table lists the `Xil_L2CacheStoreLine` function arguments.

Table 121: **Xil_L2CacheStoreLine Arguments**

Name	Description
adr	32bit address of the data/instruction to be stored.

Returns

None.

Cortex A9 Processor MMU Functions

MMU functions equip users to enable MMU, disable MMU and modify default memory attributes of MMU table as per the need.

Table 122: **Quick Function Reference**

Type	Name	Arguments
void	Xil_SetTlbAttributes	INTPTR Addr u32 attrib
void	Xil_EnableMMU	None.
void	Xil_DisableMMU	None.
void *	Xil_MemMap	UINTPTR PhysAddr size_t size u32 flags

Functions

Xil_SetTlbAttributes

This function sets the memory attributes for a section covering 1MB of memory in the translation table.

Note: The MMU or D-cache does not need to be disabled before changing a translation table entry.

Prototype

```
void Xil_SetTlbAttributes(INTPTR Addr, u32 attrib);
```

Parameters

The following table lists the `Xil_SetTlbAttributes` function arguments.

Table 123: **Xil_SetTlbAttributes Arguments**

Name	Description
Addr	32-bit address for which memory attributes need to be set.
attrib	Attribute for the given memory region. xil_mmu.h contains definitions of commonly used memory attributes which can be utilized for this function.

Returns

None.

Xil_EnableMMU

Enable MMU for cortex A9 processor.

This function invalidates the instruction and data caches, and then enables MMU.

Prototype

```
void Xil_EnableMMU(void);
```

Parameters

The following table lists the `Xil_EnableMMU` function arguments.

Table 124: **Xil_EnableMMU Arguments**

Name	Description
None.	

Returns

None.

Xil_DisableMMU

Disable MMU for Cortex A9 processors.

This function invalidates the TLBs, Branch Predictor Array and flushed the D Caches before disabling the MMU.

***Note*:** When the MMU is disabled, all the memory accesses are treated as strongly ordered.

Prototype

```
void Xil_DisableMMU(void);
```

Parameters

The following table lists the `Xil_DisableMMU` function arguments.

Table 125: **Xil_DisableMMU Arguments**

Name	Description
None.	

Returns

None.

Xil_MemMap

Memory mapping for Cortex A9 processor.

***Note*:** : Previously this was implemented in libmetal. Move to embeddedsw as this functionality is specific to A9 processor.

Prototype

```
void * Xil_MemMap(UINTPTR PhysAddr, size_t size, u32 flags);
```

Parameters

The following table lists the `Xil_MemMap` function arguments.

Table 126: **Xil_MemMap Arguments**

Name	Description
PhysAddr	is physical address.
size	is size of region.
flags	is flags used to set translation table.

Returns

Pointer to virtual address.

Cortex A9 Time Functions

xtime_l.h provides access to the 64-bit Global Counter in the PMU. This counter increases by one at every two processor cycles. These functions can be used to get/set time in the global timer.

Table 127: **Quick Function Reference**

Type	Name	Arguments
void	XTime_SetTime	XTime Xtime_Global
void	XTime_GetTime	XTime * Xtime_Global

Functions

XTime_SetTime

Set the time in the Global Timer Counter Register.

Note: When this function is called by any one processor in a multi- processor environment, reference time will reset/lost for all processors.

Prototype

```
void XTime_SetTime(XTime Xtime_Global);
```

Parameters

The following table lists the `XTime_SetTime` function arguments.

Table 128: **XTime_SetTime Arguments**

Name	Description
Xtime_Global	64-bit Value to be written to the Global Timer Counter Register.

Returns

None.

XTime_GetTime

Get the time from the Global Timer Counter Register.

Note: None.

Prototype

```
void XTime_GetTime(XTime *Xtime_Global);
```

Parameters

The following table lists the `XTime_GetTime` function arguments.

Table 129: **XTime_GetTime Arguments**

Name	Description
Xtime_Global	Pointer to the 64-bit location which will be updated with the current timer value.

Returns

None.

Cortex A9 Event Counter Function

Cortex A9 event counter functions can be utilized to configure and control the Cortex-A9 performance monitor events.

Cortex-A9 performance monitor has six event counters which can be used to count a variety of events described in Coretx-A9 TRM. xpm_counter.h defines configurations XPM_CNTRCFGx which can be used to program the event counters to count a set of events.

Note: It doesn't handle the Cortex-A9 cycle counter, as the cycle counter is being used for time keeping.

Table 130: **Quick Function Reference**

Type	Name	Arguments
void	Xpm_SetEvents	s32 PmcrCfg
void	Xpm_GetEventCounters	u32 * PmCtrValue

Functions

Xpm_SetEvents

This function configures the Cortex A9 event counters controller, with the event codes, in a configuration selected by the user and enables the counters.

Note: None.

Prototype

```
void Xpm_SetEvents(s32 PmcrCfg);
```

Parameters

The following table lists the `Xpm_SetEvents` function arguments.

Table 131: **Xpm_SetEvents Arguments**

Name	Description
PmcrCfg	Configuration value based on which the event counters are configured. XPM_CNTRCFG* values defined in xpm_counter.h can be utilized for setting configuration.

Returns

None.

Xpm_GetEventCounters

This function disables the event counters and returns the counter values.

Note: None.

Prototype

```
void Xpm_GetEventCounters(u32 *PmCtrValue);
```

Parameters

The following table lists the `Xpm_GetEventCounters` function arguments.

Table 132: **Xpm_GetEventCounters Arguments**

Name	Description
PmCtrValue	Pointer to an array of type u32 PmCtrValue[6]. It is an output parameter which is used to return the PM counter values.

Returns

None.

PL310 L2 Event Counters Functions

xl2cc_counter.h contains APIs for configuring and controlling the event counters in PL310 L2 cache controller. PL310 has two event counters which can be used to count variety of events like DRHIT, DRREQ, DWHIT, DWREQ, etc. xl2cc_counter.h contains definitions for different configurations which can be used for the event counters to count a set of events.

Table 133: **Quick Function Reference**

Type	Name	Arguments
void	XL2cc_EventCtrInit	s32 Event0 s32 Event1
void	XL2cc_EventCtrStart	None.
void	XL2cc_EventCtrStop	u32 * EveCtr0

Functions

XL2cc_EventCtrInit

This function initializes the event counters in L2 Cache controller with a set of event codes specified by the user.

***Note*:** The definitions for event codes XL2CC_* can be found in xl2cc_counter.h.

Prototype

```
void XL2cc_EventCtrInit(s32 Event0, s32 Event1);
```

Parameters

The following table lists the `XL2cc_EventCtrInit` function arguments.

Table 134: **XL2cc_EventCtrInit Arguments**

Name	Description
Event0	Event code for counter 0.
Event1	Event code for counter 1.

Returns

None.

XL2cc_EventCtrStart

This function starts the event counters in L2 Cache controller.

Note: None.

Prototype

```
void XL2cc_EventCtrStart(void);
```

Parameters

The following table lists the `XL2cc_EventCtrStart` function arguments.

Table 135: **XL2cc_EventCtrStart Arguments**

Name	Description
None.	

Returns

None.

XL2cc_EventCtrStop

This function disables the event counters in L2 Cache controller, saves the counter values and resets the counters.

Note: None.

Prototype

```
void XL2cc_EventCtrStop(u32 *EveCtr0, u32 *EveCtr1);
```

Parameters

The following table lists the `XL2cc_EventCtrStop` function arguments.

Table 136: **XL2cc_EventCtrStop Arguments**

Name	Description
EveCtr0	Output parameter which is used to return the value in event counter 0. EveCtr1: Output parameter which is used to return the value in event counter 1.

Returns

None.

Cortex A9 Processor and pl310 Errata Support

Various ARM errata are handled in the standalone BSP. The implementation for errata handling follows ARM guidelines and is based on the open source Linux support for these errata.

Note: The errata handling is enabled by default. To disable handling of all the errata globally, un-define the macro ENABLE_ARM_ERRATA in xil_errata.h. To disable errata on a per-erratum basis, un-define relevant macros in xil_errata.h.

Cortex A9 Processor Specific Include Files

The xpseudo_asm.h includes xreg_cortexa9.h and xpseudo_asm_gcc.h.

The xreg_cortexa9.h file contains definitions for inline assembler code. It provides inline definitions for Cortex A9 GPRs, SPRs, MPE registers, co-processor registers and Debug registers.

The xpseudo_asm_gcc.h contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation etc. These inline assembler instructions can be used from drivers and user applications written in C.

Cortex A53 32-bit Processor API

Cortex-A53 standalone BSP contains two separate BSPs for 32-bit mode and 64-bit mode. The 32-bit mode of cortex-A53 is compatible with ARMv7-A architecture.

Cortex A53 32-bit Processor Boot Code

The boot code performs minimum configuration which is required for an application to run starting from processor's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling
2. Invalidate instruction cache, data cache and TLBs
3. Program stack pointer for various modes (IRQ, FIQ, supervisor, undefine, abort, system)
4. Program counter frequency
5. Configure MMU with short descriptor translation table format and program base address of translation table

6. Enable data cache, instruction cache and MMU
7. Transfer control to _start which clears BSS sections and runs global constructor before jumping to main application

Cortex A53 32-bit Processor Cache Functions

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Table 137: **Quick Function Reference**

Type	Name	Arguments
void	Xil_DCacheEnable	None.
void	Xil_DCacheDisable	None.
void	Xil_DCacheInvalidate	None.
void	Xil_DCacheInvalidateRange	INTPTR adr u32 len
void	Xil_DCacheFlush	None.
void	Xil_DCacheFlushRange	INTPTR adr u32 len
void	Xil_DCacheInvalidateLine	u32 adr
void	Xil_DCacheFlushLine	u32 adr
void	Xil_ICacheInvalidateLine	u32 adr
void	Xil_ICacheEnable	None.
void	Xil_ICacheDisable	None.
void	Xil_ICacheInvalidate	None.
void	Xil_ICacheInvalidateRange	INTPTR adr u32 len

Functions

Xil_DCacheEnable

Enable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheEnable(void);
```

Parameters

The following table lists the `Xil_DCacheEnable` function arguments.

Table 138: **Xil_DCacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheDisable(void);
```

Parameters

The following table lists the `Xil_DCacheDisable` function arguments.

Table 139: **Xil_DCacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidate

Invalidate the Data cache.

The contents present in the data cache are cleaned and invalidated.

Note: In Cortex-A53, functionality to simply invalid the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate to avoid such corruption.

Prototype

```
void Xil_DCacheInvalidate(void);
```

Parameters

The following table lists the `Xil_DCacheInvalidate` function arguments.

Table 140: **Xil_DCacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidateRange

Invalidate the Data cache for the given address range.

The cachelines present in the adderss range are cleaned and invalidated

@notice In Cortex-A53, functionality to simply invalid the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate to avoid such corruption.

Prototype

```
void Xil_DCacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 141: **Xil_DCacheInvalidateRange Arguments**

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_DCacheFlush

Flush the Data cache.

@notice None.

Prototype

```
void Xil_DCacheFlush(void);
```

Parameters

The following table lists the `Xil_DCacheFlush` function arguments.

Table 142: **Xil_DCacheFlush Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheFlushRange

Flush the Data cache for the given address range.

If the bytes specified by the address range are cached by the Data cache, the cachelines containing those bytes are invalidated. If the cachelines are modified (dirty), they are written to system memory before the lines are invalidated.

@notice None.

Prototype

```
void Xil_DCacheFlushRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheFlushRange` function arguments.

Table 143: **Xil_DCacheFlushRange Arguments**

Name	Description
adr	32bit start address of the range to be flushed.
len	Length of range to be flushed in bytes.

Returns

None.

Xil_DCacheInvalidateLine

Invalidate a Data cache line.

The cacheline is cleaned and invalidated.

Note: In Cortex-A53, functionality to simply invalid the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate to avoid such corruption.

Prototype

```
void Xil_DCacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheInvalidateLine` function arguments.

Table 144: **Xil_DCacheInvalidateLine Arguments**

Name	Description
adr	32 bit address of the data to be invalidated.

Returns

None.

Xil_DCacheFlushLine

Flush a Data cache line.

If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

@notice The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheFlushLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheFlushLine` function arguments.

Table 145: **Xil_DCacheFlushLine Arguments**

Name	Description
adr	32bit address of the data to be flushed.

Returns

None.

Xil_ICacheInvalidateLine

Invalidate an instruction cache line.

If the instruction specified by the address is cached by the instruction cache, the cachecline containing that instruction is invalidated.

@notice The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_ICacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_ICacheInvalidateLine` function arguments.

Table 146: **Xil_ICacheInvalidateLine Arguments**

Name	Description
adr	32bit address of the instruction to be invalidated..

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

@notice None.

Prototype

```
void Xil_ICacheEnable(void);
```

Parameters

The following table lists the `Xil_ICacheEnable` function arguments.

Table 147: **Xil_ICacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Note: None.

Prototype

```
void Xil_ICacheDisable(void);
```

Parameters

The following table lists the `Xil_ICacheDisable` function arguments.

Table 148: **Xil_ICacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

***Note*:** None.

Prototype

```
void Xil_ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_ICacheInvalidate` function arguments.

Table 149: **Xil_ICacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidateRange

Invalidate the instruction cache for the given address range.

If the instructions specified by the address range are cached by the instrunction cache, the cachelines containing those instructions are invalidated.

@notice None.

Prototype

```
void Xil_ICacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 150: **Xil_ICacheInvalidateRange Arguments**

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Cortex A53 32-bit Processor MMU Handling

MMU functions equip users to enable MMU, disable MMU and modify default memory attributes of MMU table as per the need.

None.

***Note*:**

Table 151: **Quick Function Reference**

Type	Name	Arguments
void	Xil_SetTlbAttributes	UINTPTR Addr u32 attrib
void	Xil_EnableMMU	None.
void	Xil_DisableMMU	None.

Functions

Xil_SetTlbAttributes

This function sets the memory attributes for a section covering 1MB of memory in the translation table.

***Note*:** The MMU or D-cache does not need to be disabled before changing a translation table entry.

Prototype

```
void Xil_SetTlbAttributes(UINTPTR Addr, u32 attrib);
```

Parameters

The following table lists the `Xil_SetTlbAttributes` function arguments.

Table 152: **Xil_SetTlbAttributes Arguments**

Name	Description
Addr	32-bit address for which the attributes need to be set.
attrib	Attributes for the specified memory region. xil_mmu.h contains commonly used memory attributes definitions which can be utilized for this function.

Returns

None.

Xil_EnableMMU

Enable MMU for Cortex-A53 processor in 32bit mode.

This function invalidates the instruction and data caches before enabling MMU.

Prototype

```
void Xil_EnableMMU(void);
```

Parameters

The following table lists the `Xil_EnableMMU` function arguments.

Table 153: **Xil_EnableMMU Arguments**

Name	Description
None.	

Returns

None.

Xil_DisableMMU

Disable MMU for Cortex A53 processors in 32bit mode.

This function invalidates the TLBs, Branch Predictor Array and flushed the data cache before disabling the MMU.

***Note*:** When the MMU is disabled, all the memory accesses are treated as strongly ordered.

Prototype

```
void Xil_DisableMMU(void);
```

Parameters

The following table lists the `Xil_DisableMMU` function arguments.

Table 154: **Xil_DisableMMU Arguments**

Name	Description
None.	

Returns

None.

Cortex A53 32-bit Mode Time Functions

xtime_l.h provides access to the 64-bit physical timer counter.

Table 155: **Quick Function Reference**

Type	Name	Arguments
void	XTime_StartTimer	None.
void	XTime_SetTime	XTime Xtime_Global
void	XTime_GetTime	XTime * Xtime_Global

Functions

XTime_StartTimer

Start the 64-bit physical timer counter.

Note: The timer is initialized only if it is disabled. If the timer is already running this function does not perform any operation.

Prototype

```
void XTime_StartTimer(void);
```

Parameters

The following table lists the `XTime_StartTimer` function arguments.

Table 156: **XTime_StartTimer Arguments**

Name	Description
None.	

Returns

None.

XTime_SetTime

Timer of A53 runs continuously and the time can not be set as desired.

This API doesn't contain anything. It is defined to have uniformity across platforms.

Note: None.

Prototype

```
void XTime_SetTime(XTime Xtime_Global);
```

Parameters

The following table lists the `XTime_SetTime` function arguments.

Table 157: **XTime_SetTime Arguments**

Name	Description
Xtime_Global	64bit Value to be written to the Global Timer Counter Register. But since the function does not contain anything, the value is not used for anything.

Returns

None.

XTime_GetTime

Get the time from the physical timer counter register.

Note: None.

Prototype

```
void XTime_GetTime(XTime *Xtime_Global);
```

Parameters

The following table lists the `XTime_GetTime` function arguments.

Table 158: **XTime_GetTime Arguments**

Name	Description
Xtime_Global	Pointer to the 64-bit location to be updated with the current value in physical timer counter.

Returns

None.

Cortex A53 32-bit Processor Specific Include Files

The xpseudo_asm.h includes xreg_cortexa53.h and xpseudo_asm_gcc.h. The xreg_cortexa53.h file contains definitions for inline assembler code. It provides inline definitions for Cortex A53 GPRs, SPRs, co-processor registers and floating point registers.

The xpseudo_asm_gcc.h contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation etc. These inline assembler instructions can be used from drivers and user applications written in C.

Cortex A53 64-bit Processor Boot Code

Cortex-A53 standalone BSP contains two separate BSPs for 32-bit mode and 64-bit mode. The 64-bit mode of cortex-A53 contains ARMv8-A architecture. This section provides a linked summary and detailed descriptions of the Cortex A53 64-bit Processor APIs.

Cortex A53 64-bit Processor Cache Functions

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Table 159: **Quick Function Reference**

Type	Name	Arguments
void	Xil_DCacheEnable	None.
void	Xil_DCacheDisable	None.
void	Xil_DCacheInvalidate	None.
void	Xil_DCacheInvalidateRange	INTPTR adr INTPTR len
void	Xil_DCacheInvalidateLine	INTPTR adr
void	Xil_DCacheFlush	None.
void	Xil_DCacheFlushLine	INTPTR adr

Table 159: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	Xil_ICacheEnable	None.
void	Xil_ICacheDisable	None.
void	Xil_ICacheInvalidate	None.
void	Xil_ICacheInvalidateRange	INTPTR adr INTPTR len
void	Xil_ICacheInvalidateLine	INTPTR adr
void	Xil_ConfigureL1Prefetch	u8 num

Functions

Xil_DCacheEnable

Enable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheEnable(void);
```

Parameters

The following table lists the `Xil_DCacheEnable` function arguments.

Table 160: **Xil_DCacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheDisable(void);
```

Parameters

The following table lists the `Xil_DCacheDisable` function arguments.

Table 161: **Xil_DCacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidate

Invalidate the Data cache.

The contents present in the cache are cleaned and invalidated.

Note: In Cortex-A53, functionality to simply invalid the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate which avoids such corruption.

Prototype

```
void Xil_DCacheInvalidate(void);
```

Parameters

The following table lists the `Xil_DCacheInvalidate` function arguments.

Table 162: **Xil_DCacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidateRange

Invalidate the Data cache for the given address range.

The cachelines present in the adderss range are cleaned and invalidated

***Note*:** In Cortex-A53, functionality to simply invalid the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate which avoids such corruption.

Prototype

```
void Xil_DCacheInvalidateRange(INTPTR adr, INTPTR len);
```

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 163: **Xil_DCacheInvalidateRange Arguments**

Name	Description
adr	64bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_DCacheInvalidateLine

Invalidate a Data cache line.

The cacheline is cleaned and invalidated.

***Note*:** In Cortex-A53, functionality to simply invalid the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate which avoids such corruption.

Prototype

```
void Xil_DCacheInvalidateLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheInvalidateLine` function arguments.

Table 164: **Xil_DCacheInvalidateLine Arguments**

Name	Description
adr	64bit address of the data to be flushed.

Returns

None.

Xil_DCacheFlush

Flush the Data cache.

***Note*:** None.

Prototype

```
void Xil_DCacheFlush(void);
```

Parameters

The following table lists the `Xil_DCacheFlush` function arguments.

Table 165: **Xil_DCacheFlush Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheFlushLine

Flush a Data cache line.

If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

***Note*:** The bottom 6 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheFlushLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheFlushLine` function arguments.

Table 166: **Xil_DCacheFlushLine Arguments**

Name	Description
adr	64bit address of the data to be flushed.

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

***Note*:** None.

Prototype

```
void Xil_ICacheEnable(void);
```

Parameters

The following table lists the `Xil_ICacheEnable` function arguments.

Table 167: **Xil_ICacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

***Note*:** None.

Prototype

```
void Xil_ICacheDisable(void);
```

Parameters

The following table lists the `Xil_ICacheDisable` function arguments.

Table 168: **Xil_ICacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

Note: None.

Prototype

```
void Xil_ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_ICacheInvalidate` function arguments.

Table 169: **Xil_ICacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidateRange

Invalidate the instruction cache for the given address range.

If the instructions specified by the address range are cached by the instrunction cache, the cachelines containing those instructions are invalidated.

Note: None.

Prototype

```
void Xil_ICacheInvalidateRange(INTPTR adr, INTPTR len);
```

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 170: **Xil_ICacheInvalidateRange Arguments**

Name	Description
adr	64bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_ICacheInvalidateLine

Invalidate an instruction cache line.

If the instruction specified by the parameter adr is cached by the instruction cache, the cacheline containing that instruction is invalidated.

***Note*:** The bottom 6 bits are set to 0, forced by architecture.

Prototype

```
void Xil_ICacheInvalidateLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_ICacheInvalidateLine` function arguments.

Table 171: **Xil_ICacheInvalidateLine Arguments**

Name	Description
adr	64bit address of the instruction to be invalidated.

Returns

None.

Xil_ConfigureL1Prefetch

Configure the maximum number of outstanding data prefetches allowed in L1 cache.

***Note*:** This function is implemented only for EL3 privilege level.

Prototype

```
void Xil_ConfigureL1Prefetch(u8 num);
```

Parameters

The following table lists the `Xil_ConfigureL1Prefetch` function arguments.

Table 172: **Xil_ConfigureL1Prefetch Arguments**

Name	Description
num	maximum number of outstanding data prefetches allowed, valid values are 0-7.

Returns

None.

Cortex A53 64-bit Processor MMU Handling

MMU function equip users to modify default memory attributes of MMU table as per the need.

None.

***Note*:**

Table 173: **Quick Function Reference**

Type	Name	Arguments
void	Xil_SetTlbAttributes	UINTPTR Addr u64 attrib

Functions

Xil_SetTlbAttributes

brief It sets the memory attributes for a section, in the translation table.

If the address (defined by Addr) is less than 4GB, the memory attribute(attrib) is set for a section of 2MB memory. If the address (defined by Addr) is greater than 4GB, the memory attribute (attrib) is set for a section of 1GB memory.

***Note*:** The MMU and D-cache need not be disabled before changing an translation table attribute.

Prototype

```
void Xil_SetTlbAttributes(UINTPTR Addr, u64 attrib);
```

Parameters

The following table lists the `Xil_SetTlbAttributes` function arguments.

Table 174: **Xil_SetTlbAttributes Arguments**

Name	Description
Addr	64-bit address for which attributes are to be set.
attrib	Attribute for the specified memory region. xil_mmu.h contains commonly used memory attributes definitions which can be utilized for this function.

Returns

None.

Cortex A53 64-bit Mode Time Functions

xtime_l.h provides access to the 64-bit physical timer counter.

Table 175: **Quick Function Reference**

Type	Name	Arguments
void	XTime_StartTimer	None.
void	XTime_SetTime	XTime Xtime_Global
void	XTime_GetTime	XTime * Xtime_Global

Functions

XTime_StartTimer

Start the 64-bit physical timer counter.

Note: The timer is initialized only if it is disabled. If the timer is already running this function does not perform any operation. This API is effective only if BSP is built for EL3. For EL1 Non-secure, it simply exits.

Prototype

```
void XTime_StartTimer(void);
```

Parameters

The following table lists the `XTime_StartTimer` function arguments.

Table 176: **XTime_StartTimer Arguments**

Name	Description
None.	

Returns

None.

XTime_SetTime

Timer of A53 runs continuously and the time can not be set as desired.

This API doesn't contain anything. It is defined to have uniformity across platforms.

***Note*:** None.

Prototype

```
void XTime_SetTime(XTime Xtime_Global);
```

Parameters

The following table lists the `XTime_SetTime` function arguments.

Table 177: **XTime_SetTime Arguments**

Name	Description
Xtime_Global	64bit value to be written to the physical timer counter register. Since API does not do anything, the value is not utilized.

Returns

None.

XTime_GetTime

Get the time from the physical timer counter register.

***Note*:** None.

Prototype

```
void XTime_GetTime(XTime *Xtime_Global);
```

Parameters

The following table lists the `XTime_GetTime` function arguments.

Table 178: **XTime_GetTime Arguments**

Name	Description
Xtime_Global	Pointer to the 64-bit location to be updated with the current value of physical timer counter register.

Returns

None.

Cortex A53 64-bit Processor Specific Include Files

The xpseudo_asm.h includes xreg_cortexa53.h and xpseudo_asm_gcc.h. The xreg_cortexa53.h file contains definitions for inline assembler code. It provides inline definitions for Cortex A53 GPRs, SPRs and floating point registers.

The xpseudo_asm_gcc.h contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation etc. These inline assembler instructions can be used from drivers and user applications written in C.

Chapter 4

LwIP 2.1.1 Library

Introduction

The lwIP is an open source TCP/IP protocol suite available under the BSD license. The lwIP is a standalone stack; there are no operating systems dependencies, although it can be used along with operating systems. The lwIP provides two A05PIs for use by applications:

- RAW API: Provides access to the core lwIP stack.
- Socket API: Provides a BSD sockets style interface to the stack.

The lwip211_v1.2 is built on the open source lwIP library version 2.1.1. The lwip211 library provides adapters for the Ethernetlite (axi_ethernetlite), the TEMAC (axi_ethernet), and the Gigabit Ethernet controller and MAC (GigE) cores. The library can run on MicroBlaze™, ARM Cortex-A9, ARM Cortex-A53, and ARM Cortex-R5 processors. The Ethernetlite and TEMAC cores apply for MicroBlaze systems. The Gigabit Ethernet controller and MAC (GigE) core is applicable only for ARM Cortex-A9 system (Zynq-7000 processor devices) and ARM Cortex-A53 & ARM Cortex-R5 system (Zynq UltraScale+ MPSoC).

Features

The lwIP provides support for the following protocols:

- Internet Protocol (IP)
- Internet Control Message Protocol (ICMP)
- User Datagram Protocol (UDP)
- TCP (Transmission Control Protocol (TCP)
- Address Resolution Protocol (ARP)
- Dynamic Host Configuration Protocol (DHCP)
- Internet Group Message Protocol (IGMP)

References

- FreeRTOS: http://www.freertos.org/Interactive_Frames/Open_Frames.html?http://interactive.freertos.org/forums
- lwIP wiki: http://lwip.scribblewiki.com
- Xilinx® lwIP designs and application examples: http://www.xilinx.com/support/documentation/application_notes/xapp1026.pdf
- lwIP examples using RAW and Socket APIs: http://savannah.nongnu.org/projects/lwip/
- FreeRTOS Port for Zynq is available for download from the [FreeRTOS][freertos] website

Using lwIP

Overview

The following are the key steps to use lwIP for networking:

1. Creating a hardware system containing the processor, ethernet core, and a timer. The timer and ethernet interrupts must be connected to the processor using an interrupt controller.
2. Configuring lwip211_v1.2 to be a part of the software platform. For operating with lwIP socket API, the Xilkernel library or FreeRTOS BSP is a prerequisite. See the Note below.

***Note*:** The Xilkernel library is available only for MicroBlaze systems. For Cortex-A9 based systems (Zynq) and Cortex-A53 or Cortex-R5 based systems (Zynq® UltraScale™+ MPSoC), there is no support for Xilkernel. Instead, use FreeRTOS. A FreeRTOS BSP is available for Zynq and Zynq UltraScale+ MPSoC systems and must be included for using lwIP socket API. The FreeRTOS BSP for Zynq and Zynq UltraScale+ MPSoC is available for download from the the [FreeRTOS][http://www.freertos.org/Interactive_Frames/Open_Frames.html?http://interactive.freertos.org/forums] website.

Setting up the Hardware System

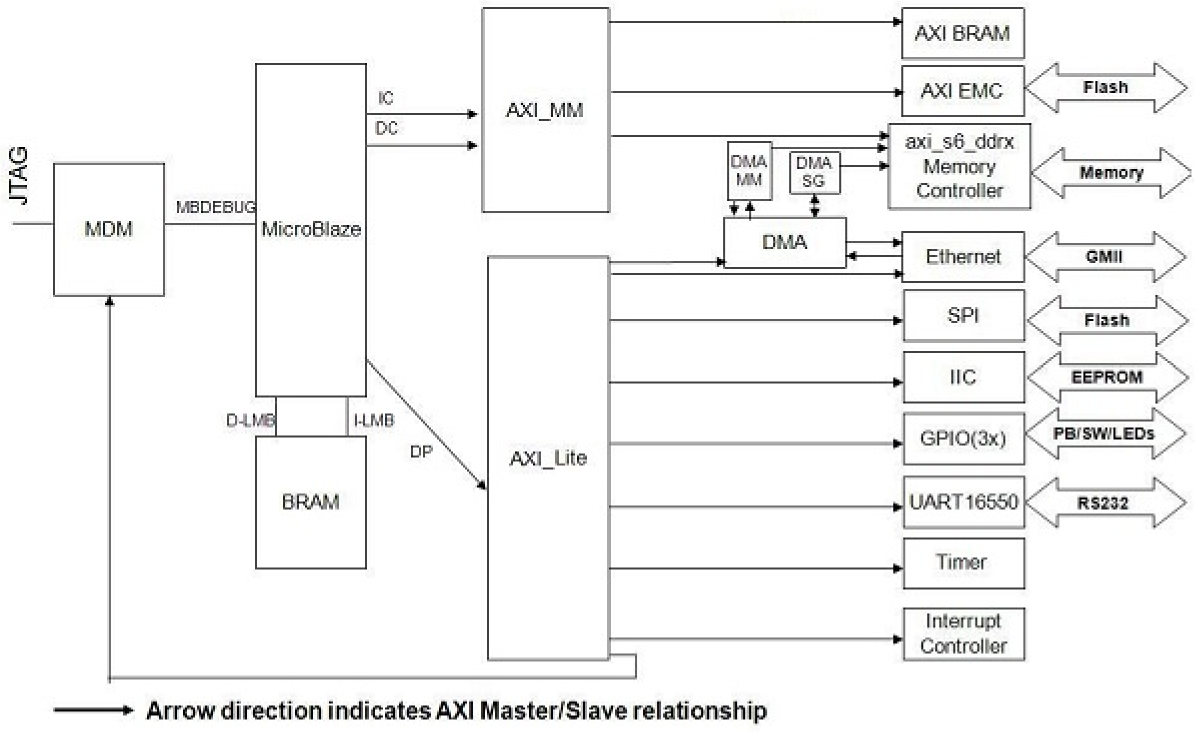
This section describes the hardware configurations supported by lwIP. The key components of the hardware system include:

- Processor: Either a MicroBlaze or a Cortex-A9 or a Cortex-A53 or a Cortex-R5 processor. The Cortex-A9 processor applies to Zynq systems. The Cortex-A53 and Cortex-R5 processors apply to Zynq UltraScale+ MPSoC systems.
- MAC: LwIP supports axi_ethernetlite, axi_ethernet, and Gigabit Ethernet controller and MAC (GigE) cores.
- Timer: to maintain TCP timers, lwIP raw API based applications require that certain functions are called at periodic intervals by the application. An application can do this by registering an interrupt handler with a timer.

- DMA: For axi_ethernet based systems, the axi_ethernet cores can be configured with a soft DMA engine (AXI DMA and MCDMA) or a FIFO interface. For GigE-based Zynq and Zynq UltraScale+ MPSoC systems, there is a built-in DMA and so no extra configuration is needed. Same applies to axi_ethernetlite based systems, which have their built-in buffer management provisions.

The following figure shows a sample system architecture with a Kintex-6 device utilizing the axi_ethernet core with DMA.

Figure 1: **System Architecture using axi_ethernet core with DMA**



Setting up the Software System

To use lwIP in a software application, you must first compile the lwIP library as a part of the 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 lwip211 library from the list of Supported Libraries.
15. Expand the Overview tree and select lwip211. The configuration options for the lwip211 library are listed.
16. Configure the lwip211 library and click OK.

Configuring lwIP Options

The lwIP library provides configurable parameters. There are two major categories of configurable options:

- Xilinx Adapter to lwIP options: These control the settings used by Xilinx adapters for the ethernet cores.
- Base lwIP options: These options are part of lwIP library itself, and include parameters for TCP, UDP, IP and other protocols supported by lwIP. The following sections describe the available lwIP configurable options.

Customizing lwIP API Mode

The lwip211_v1.3 supports both raw API and socket API:

- The raw API is customized for high performance and lower memory overhead. The limitation of raw API is that it is callback-based, and consequently does not provide portability to other TCP stacks.

- The socket API provides a BSD socket-style interface and is very portable; however, this mode is not as efficient as raw API mode in performance and memory requirements. The lwip211_v1.3 also provides the ability to set the priority on TCP/IP and other lwIP application threads.

The following table describes the lwIP library API mode options.

Attribute	Description	Type	Default
api_mode {RAW_API | SOCKET_API}	The lwIP library mode of operation	enum	RAW_API
socket_mode_thread_prio	Priority of lwIP TCP/IP thread and all lwIP application threads. This setting applies only when Xilkernel is used in priority mode. It is recommended that all threads using lwIP run at the same priority level. For GigE based Zynq-7000 and Zynq UltraScale+ MPSoC systems using FreeRTOS, appropriate priority should be set. The default priority of 1 will not give the expected behaviour. For FreeRTOS (Zynq-7000 and Zynq UltraScale+ MPSoC systems), all internal lwIP tasks (except the main TCP/IP task) are created with the priority level set for this attribute. The TCP/IP task is given a higher priority than other tasks for improved performance. The typical TCP/IP task priority is 1 more than the priority set for this attribute for FreeRTOS.	integer	1
use_axieth_on_zynq	In the event that the AxiEthernet soft IP is used on a Zynq-7000 device or a Zynq UltraScale+ MPSoC device. This option ensures that the GigE on the Zynq-7000 PS (EmacPs) is not enabled and the device uses the AxiEthernet soft IP for Ethernet traffic. The existing Xilinx-provided lwIP adapters are not tested for multiple MACs. Multiple Axi Ethernet's are not supported on Zynq UltraScale+ MPSOC devices.	integer	0 = Use Zynq-7000 PS-based or ZynMP PS-based GigE controller 1= User AxiEthernet

Configuring Xilinx Adapter Options

The Xilinx adapters for EMAC/GigE cores are configurable.

Ethernetlite Adapter Options

The following table describes the configuration parameters for the axi_ethernetlite adapter.

Attribute	Description	Type	Default
sw_rx_fifo_size	Software Buffer Size in bytes of the receive data between EMAC and processor	integer	8192
sw_tx_fifo_size	Software Buffer Size in bytes of the transmit data between processor and EMAC	integer	8192

TEMAC Adapter Options

The following table describes the configuration parameters for the axi_ethernet and GigE adapters.

Attribute	Type	Description
n_tx_descriptors	integer	Number of Tx descriptors to be used. For high performance systems there might be a need to use a higher value. Default is 64.
n_rx_descriptors	integer	Number of Rx descriptors to be used. For high performance systems there might be a need to use a higher value. Typical values are 128 and 256. Default is 64.
n_tx_coalesce	integer	Setting for Tx interrupt coalescing. Default is 1.
n_rx_coalesce	integer	Setting for Rx interrupt coalescing. Default is 1.
tcp_rx_checksum_offload	boolean	Offload TCP Receive checksum calculation (hardware support required). For GigE in Zynq and Zynq UltraScale+ MPSoC, the TCP receive checksum offloading is always present, so this attribute does not apply. Default is false.
tcp_tx_checksum_offload	boolean	Offload TCP Transmit checksum calculation (hardware support required). For GigE cores (Zynq and Zynq UltraScale+ MPSoC), the TCP transmit checksum offloading is always present, so this attribute does not apply. Default is false.
tcp_ip_rx_checksum_ofload	boolean	Offload TCP and IP Receive checksum calculation (hardware support required). Applicable only for AXI systems. For GigE in Zynq and Zynq UltraScale+ MPSoC devices, the TCP and IP receive checksum offloading is always present, so this attribute does not apply. Default is false.

Attribute	Type	Description
tcp_ip_tx_checksum_ofload	boolean	Offload TCP and IP Transmit checksum calculation (hardware support required). Applicable only for AXI systems. For GigE in Zynq and Zynq UltraScale+ MPSoC devices, the TCP and IP transmit checksum offloading is always present, so this attribute does not apply. Default is false.
phy_link_speed	CONFIG_LINKSPEED_ AUTODETECT	Link speed as auto-negotiated by the PHY. lwIP configures the TEMAC/GigE for this speed setting. This setting must be correct for the TEMAC/GigE to transmit or receive packets. The CONFIG_LINKSPEED_ AUTODETECT setting attempts to detect the correct linkspeed by reading the PHY registers; however, this is PHY dependent, and has been tested with the Marvell and TI PHYs present on Xilinx development boards. For other PHYs, select the correct speed. Default is enum.
temac_use_jumbo_ frames_experimental	boolean	Use TEMAC jumbo frames (with a size up to 9k bytes). If this option is selected, jumbo frames are allowed to be transmitted and received by the TEMAC. For GigE in Zynq there is no support for jumbo frames, so this attribute does not apply. Default is false.

Configuring Memory Options

The lwIP stack provides different kinds of memories. Similarly, when the application uses socket mode, different memory options are used. All the configurable memory options are provided as a separate category. Default values work well unless application tuning is required. The following table describes the memory parameter options.

Attribute	Default	Type	Description
mem_size	131072	Integer	Total size of the heap memory available, measured in bytes. For applications which use a lot of memory from heap (using C library malloc or lwIP routine mem_malloc or pbuf_alloc with PBUF_RAM option), this number should be made higher as per the requirements.
memp_n_pbuf	16	Integer	The number of memp struct pbufs. If the application sends a lot of data out of ROM (or other static memory), this should be set high.
memp_n_udp_pcb	4	Integer	The number of UDP protocol control blocks. One per active UDP connection.

Attribute	Default	Type	Description
memp_n_tcp_pcb	32	Integer	The number of simultaneously active TCP connections.
memp_n_tcp_pcb _listen	8	Integer	The number of listening TC connections.
memp_n_tcp_seg	256	Integer	The number of simultaneously queued TCP segments.
memp_n_sys_timeout	8	Integer	Number of simultaneously active timeouts.
memp_num_netbuf	8	Integer	Number of allowed structure instances of type netbufs. Applicable only in socket mode.
memp_num_netconn	16	Integer	Number of allowed structure instances of type netconns. Applicable only in socket mode.
memp_num_api_msg	16	Integer	Number of allowed structure instances of type api_msg. Applicable only in socket mode.
memp_num_tcpip_msg	64	Integer	Number of TCPIP msg structures (socket mode only).

***Note*:** Because Sockets Mode support uses Xilkernel services, the number of semaphores chosen in the Xilkernel configuration must take the value set for the memp_num_netbuf parameter into account. For FreeRTOS BSP there is no setting for the maximum number of semaphores. For FreeRTOS, you can create semaphores as long as memory is available.

Configuring Packet Buffer (Pbuf) Memory Options

Packet buffers (Pbufs) carry packets across various layers of the TCP/IP stack. The following are the pbuf memory options provided by the lwIP stack. Default values work well unless application tuning is required. The following table describes the parameters for the Pbuf memory options.

Attribute	Default	Type	Description
pbuf_pool_size	256	Integer	Number of buffers in pbuf pool. For high performance systems, you might consider increasing the pbuf pool size to a higher value, such as 512.
pbuf_pool_bufsize	1700	Integer	Size of each pbuf in pbuf pool. For systems that support jumbo frames, you might consider using a pbuf pool buffer size that is more than the maximum jumbo frame size.

Attribute	Default	Type	Description
pbuf_link_hlen	16	Integer	Number of bytes that should be allocated for a link level header.

Configuring ARP Options

The following table describes the parameters for the ARP options. Default values work well unless application tuning is required.

Attribute	Default	Type	Description
arp_table_size	10	Integer	Number of active hardware address IP address pairs cached.
arp_queueing	1	Integer	If enabled outgoing packets are queued during hardware address resolution. This attribute can have two values: 0 or 1.

Configuring IP Options

The following table describes the IP parameter options. Default values work well unless application tuning is required.

Attribute	Default	Type	Description
ip_forward	0	Integer	Set to 1 for enabling ability to forward IP packets across network interfaces. If running lwIP on a single network interface, set to 0. This attribute can have two values: 0 or 1.
ip_options	0	Integer	When set to 1, IP options are allowed (but not parsed). When set to 0, all packets with IP options are dropped. This attribute can have two values: 0 or 1.
ip_reassembly	1	Integer	Reassemble incoming fragmented IP packets.
ip_frag	1	Integer	Fragment outgoing IP packets if their size exceeds MTU.
ip_reass_max_pbufs	128	Integer	Reassembly pbuf queue length.
ip_frag_max_mtu	1500	Integer	Assumed max MTU on any interface for IP fragmented buffer.
ip_default_ttl	255	Integer	Global default TTL used by transport layers.

Configuring ICMP Options

The following table describes the parameter for ICMP protocol option. Default values work well unless application tuning is required.

For GigE cores (for Zynq and Zynq MPSoC) there is no support for ICMP in the hardware.

Attribute	Default	Type	Description
icmp_ttl	255	Integer	ICMP TTL value.

Configuring IGMP Options

The IGMP protocol is supported by lwIP stack. When set true, the following option enables the IGMP protocol.

Attribute	Default	Type	Description
imgp_options	false	Boolean	Specify whether IGMP is required.

Configuring UDP Options

The following table describes UDP protocol options. Default values work well unless application tuning is required.

Attribute	Default	Type	Description
lwip_udp	true	Boolean	Specify whether UDP is required.
udp_ttl	255	Integer	UDP TTL value.

Configuring TCP Options

The following table describes the TCP protocol options. Default values work well unless application tuning is required.

Attribute	Default	Type	Description
lwip_tcp	true	Boolean	Require TCP.
tcp_ttl	255	Integer	TCP TTL value.
tcp_wnd	2048	Integer	TCP Window size in bytes.
tcp_maxrtx	12	Integer	TCP Maximum retransmission value.
tcp_synmaxrtx	4	Integer	TCP Maximum SYN retransmission value.
tcp_queue_ooseq	1	Integer	Accept TCP queue segments out of order. Set to 0 if your device is low on memory.

Attribute	Default	Type	Description
tcp_mss	1460	Integer	TCP Maximum segment size.
tcp_snd_buf	8192	Integer	TCP sender buffer space in bytes.

Configuring DHCP Options

The DHCP protocol is supported by lwIP stack. The following table describes DHCP protocol options. Default values work well unless application tuning is required.

Attribute	Default	Type	Description
lwip_dhcp	false	Boolean	Specify whether DHCP is required.
dhcp_does_arp_check	false	Boolean	Specify whether ARP checks on offered addresses.

Configuring the Stats Option

lwIP stack has been written to collect statistics, such as the number of connections used; amount of memory used; and number of semaphores used, for the application. The library provides the stats_display() API to dump out the statistics relevant to the context in which the call is used. The stats option can be turned on to enable the statistics information to be collected and displayed when the stats_display API is called from user code. Use the following option to enable collecting the stats information for the application.

Attribute	Description	Type	Default
lwip_stats	Turn on lwIP Statistics	int	0

Configuring the Debug Option

lwIP provides debug information. The following table lists all the available options.

Attribute	Default	Type	Description
lwip_debug	false	Boolean	Turn on/off lwIP debugging.
ip_debug	false	Boolean	Turn on/off IP layer debugging.
tcp_debug	false	Boolean	Turn on/off TCP layer debugging.
udp_debug	false	Boolean	Turn on/off UDP layer debugging.
icmp_debug	false	Boolean	Turn on/off ICMP protocol debugging.
igmp_debug	false	Boolean	Turn on/off IGMP protocol debugging.
netif_debug	false	Boolean	Turn on/off network interface layer debugging.

Attribute	Default	Type	Description
sys_debug	false	Boolean	Turn on/off sys arch layer debugging.
pbuf_debug	false	Boolean	Turn on/off pbuf layer debugging

LwIP Library APIs

The lwIP library provides two different APIs: RAW API and Socket API.

Raw API

The Raw API is callback based. Applications obtain access directly into the TCP stack and vice-versa. As a result, there is no extra socket layer, and using the Raw API provides excellent performance at the price of compatibility with other TCP stacks.

Xilinx Adapter Requirements when using the RAW API

In addition to the lwIP RAW API, the Xilinx adapters provide the `xemacif_input` utility function for receiving packets. This function must be called at frequent intervals to move the received packets from the interrupt handlers to the lwIP stack. Depending on the type of packet received, lwIP then calls registered application callbacks. The `<Vitis_install_path>/sw/ThirdParty/sw_services/lwip211/src/lwip-2.1.1/doc/rawapi.txt` file describes the lwIP Raw API.

LwIP Performance

The following table provides the maximum TCP throughput achievable by FPGA, CPU, EMAC, and system frequency in RAW modes. Applications requiring high performance should use the RAW API.

FPGA	CPU	EMAC	System Frequency	Max TCP Throughput in RAW Mode (Mbps)
Virtex	MicroBlaze	axi-ethernet	100 MHz	RX Side: 182 TX Side: 100
Virtex	MicroBlaze	xps-ll-temac	100 MHz	RX Side: 178 TX Side: 100
Virtex	MicroBlaze	xps-ethernetlite	100 MHz	RX Side: 50 TX Side: 38

RAW API Example

Applications using the RAW API are single threaded. The following pseudo-code illustrates a typical RAW mode program structure.

```
int main()
{
        struct netif *netif, server_netif;
        ip_addr_t ipaddr, netmask, gw;


        unsigned char mac_ethernet_address[] =
                {0x00, 0x0a, 0x35, 0x00, 0x01, 0x02};

        lwip_init();


        if (!xemac_add(netif, &ipaddr, &netmask,
                &gw, mac_ethernet_address,
                EMAC_BASEADDR)) {
                printf("Error adding N/W interface\n\r");
                return -1;
        }
        netif_set_default(netif);


        platform_enable_interrupts();


        netif_set_up(netif);


        start_application();


        while (1) {
                xemacif_input(netif);

                transfer_data();
        }
}
```

Socket API

The lwIP socket API provides a BSD socket-style API to programs. This API provides an execution model that is a blocking, open-read-write-close paradigm.

Xilinx Adapter Requirements when using the Socket API

Applications using the Socket API with Xilinx adapters need to spawn a separate thread called `xemacif_input_thread`. This thread takes care of moving received packets from the interrupt handlers to the `tcpip_thread` of the lwIP. Application threads that use lwIP must be created using the lwIP `sys_thread_new` API. Internally, this function makes use of the appropriate thread or task creation routines provided by XilKernel or FreeRTOS.

Xilkernel/FreeRTOS scheduling policy when using the Socket API

lwIP in socket mode requires the use of the Xilkernel or FreeRTOS, which provides two policies for thread scheduling: round-robin and priority based. There are no special requirements when round-robin scheduling policy is used because all threads or tasks with same priority receive the same time quanta. This quanta is fixed by the RTOS (Xilkernel or FreeRTOS) being used. With priority scheduling, care must be taken to ensure that lwIP threads or tasks are not starved. For Xilkernel, lwIP internally launches all threads at the priority level specified in `socket_mode_thread_prio`. For FreeRTOS, lwIP internally launches all tasks except the main TCP/IP task at the priority specified in `socket_mode_thread_prio`. The TCP/IP task in FreeRTOS is launched with a higher priority (one more than priority set in `socket_mode_thread_prio`). In addition, application threads must launch `xemacif_input_thread`. The priorities of both `xemacif_input_thread`, and the lwIP internal threads (`socket_mode_thread_prio`) must be high enough in relation to the other application threads so that they are not starved.

Socket API Example

XilKernel-based applications in socket mode can specify a static list of threads that Xilkernel spawns on startup in the Xilkernel Software Platform Settings dialog box. Assuming that `main_thread()` is a thread specified to be launched by XIlkernel, control reaches this first thread from application `main` after the Xilkernel schedule is started. In `main_thread`, one more thread (network_thread) is created to initialize the MAC layer. For FreeRTOS (Zynq and Zynq Ultrascale+ MPSoC processor systems) based applications, once the control reaches application `main` routine, a task (can be termed as main_thread) with an entry point function as main_thread() is created before starting the scheduler. After the FreeRTOS scheduler starts, the control reaches `main_thread()`, where the lwIP internal initialization happens. The application then creates one more thread (network_thread) to initialize the MAC layer. The following pseudo-code illustrates a typical socket mode program structure.

```
void network_thread(void *p)
{
        struct netif *netif;
        ip_addr_t ipaddr, netmask, gw;


        unsigned char mac_ethernet_address[] =
                {0x00, 0x0a, 0x35, 0x00, 0x01, 0x02};

        netif = &server_netif;


        IP4_ADDR(&ipaddr,192,168,1,10);
        IP4_ADDR(&netmask,255,255,255,0);
        IP4_ADDR(&gw,192,168,1,1);


        if (!xemac_add(netif, &ipaddr, &netmask,
                        &gw, mac_ethernet_address,
                        EMAC_BASEADDR)) {
                printf("Error adding N/W interface\n\r");
                return;
        }
        netif_set_default(netif);


        netif_set_up(netif);


        sys_thread_new(“xemacif_input_thread”, xemacif_input_thread,
                netif,
                THREAD_STACKSIZE, DEFAULT_THREAD_PRIO);


        sys_thread_new(“httpd” web_application_thread, 0,
                        THREAD_STACKSIZE DEFAULT_THREAD_PRIO);
}

int main_thread()
{

        lwip_init();


        sys_thread_new(“network_thread” network_thread, NULL,
                        THREAD_STACKSIZE DEFAULT_THREAD_PRIO);

        return 0;
}
```

Using the Xilinx Adapter Helper Functions

The Xilinx adapters provide the following helper functions to simplify the use of the lwIP APIs.

Table 179: **Quick Function Reference**

Type	Name	Arguments
void	xemacif_input_thread	void
struct netif *	xemac_add	void
void	lwip_init	void
int	xemacif_input	void
void	xemacpsif_resetrx_on_no_rxdata	void

Functions

xemacif_input_thread

In the socket mode, the application thread must launch a separate thread to receive the input packets. This performs the same work as the RAW mode function, `xemacif_input()` , except that it resides in its own separate thread; consequently, any lwIP socket mode application is required to have code similar to the following in its main thread:

Note: For Socket mode only.

```
sys_thread_new("xemacif_input_thread",

               xemacif_input_thread
            , netif, THREAD_STACK_SIZE, DEFAULT_THREAD_PRIO);
```

The application can then continue launching separate threads for doing application specific tasks. The `xemacif_input_thread()` receives data processed by the interrupt handlers, and passes them to the lwIP tcpip_thread.

Prototype

```
void xemacif_input_thread(struct netif *netif);
```

Returns

xemac_add

The `xemac_add()` function provides a unified interface to add any Xilinx EMAC IP as well as GigE core. This function is a wrapper around the lwIP netif_add function that initializes the network interface 'netif' given its IP address ipaddr, netmask, the IP address of the gateway, gw, the 6 byte ethernet address mac_ethernet_address, and the base address, mac_baseaddr, of the axi_ethernetlite or axi_ethernet MAC core.

Prototype

```
struct netif * xemac_add(struct netif *netif, ip_addr_t *ipaddr, ip_addr_t
*netmask, ip_addr_t *gw, unsigned char *mac_ethernet_address, unsigned
mac_baseaddr);
```

lwip_init

Initialize all modules. Use this in NO_SYS mode. Use tcpip_init() otherwise.

This function provides a single initialization function for the lwIP data structures. This replaces specific calls to initialize stats, system, memory, pbufs, ARP, IP, UDP, and TCP layers.

Prototype

```
void lwip_init(void);
```

xemacif_input

The Xilinx lwIP adapters work in interrupt mode. The receive interrupt handlers move the packet data from the EMAC/GigE and store them in a queue. The `xemacif_input()` function takes those packets from the queue, and passes them to lwIP; consequently, this function is required for lwIP operation in RAW mode. The following is a sample lwIP application in RAW mode.

Note: For RAW mode only.

```
                while (1) {


                    xemacif_input
                (netif);


}
```

Note: The program is notified of the received data through callbacks.

Prototype

```
int xemacif_input(struct netif *netif);
```

Returns

xemacpsif_resetrx_on_no_rxdata

There is an errata on the GigE controller that is related to the Rx path. The errata describes conditions whereby the Rx path of GigE becomes completely unresponsive with heavy Rx traffic of small sized packets. The condition occurrence is rare; however a software reset of the Rx logic in the controller is required when such a condition occurs. This API must be called periodically (approximately every 100 milliseconds using a timer or thread) from user applications to ensure that the Rx path never becomes unresponsive for more than 100 milliseconds.

Note: Used in both Raw and Socket mode and applicable only for the Zynq-7000 and Zynq MPSoC processors and the GigE controller

Prototype

```
void xemacpsif_resetrx_on_no_rxdata(struct netif *netif);
```

Returns

Chapter 5

XilIsf Library v5.15

Overview

The LibXil Isf library:

- Allows you to Write, Read, and Erase the Serial Flash.
- Allows protection of the data stored in the Serial Flash from unwarranted modification by enabling the Sector Protection feature.
- Supports multiple instances of Serial Flash at a time, provided they are of the same device family (Atmel, Intel, STM, Winbond, SST, or Spansion) as the device family is selected at compile time.
- Allows your application to perform Control operations on Intel, STM, Winbond, SST, and Spansion Serial Flash.
- Requires the underlying hardware platform to contain the axi_quad_spi, ps7_spi, ps7_qspi, psu_qspi, psv_ospi, or psu_spi device for accessing the Serial Flash.
- Uses the Xilinx SPI interface drivers in interrupt-driven mode or polled mode for communicating with the Serial Flash. In interrupt mode, the user application must acknowledge any associated interrupts from the Interrupt Controller.

Additional information

- In interrupt mode, the application is required to register a callback to the library and the library registers an internal status handler to the selected interface driver.
- When your application requests a library operation, it is initiated and control is given back to the application. The library tracks the status of the interface transfers, and notifies the user application upon completion of the selected library operation.
- Added support in the library for SPI PS and QSPI PS. You must select one of the interfaces at compile time.
- Added support for QSPIPSU and SPIPS flash interface on Zynq UltraScale+ MPSoC.
- Added support for OSPIPSV flash interface
- When your application requests selection of QSPIPS interface during compilation, the QSPI PS or QSPI PSU interface, based on the hardware platform, are selected.

- When the SPIPS interface is selected during compilation, the SPI PS or the SPI PSU interface is selected.
- When the OSPI interface is selected during compilation, the OSPIPSV interface is selected.

Supported Devices

The table below lists the supported Xilinx in-system and external serial flash memories.

Device Series	Manufacturer
AT45DB011D AT45DB021D AT45DB041D AT45DB081D AT45DB161D AT45DB321D AT45DB642D	Atmel
W25Q16 W25Q32 W25Q64 W25Q80 W25Q128 W25X10 W25X20 W25X40 W25X80 W25X16 W25X32 W25X64	Winbond
S25FL004 S25FL008 S25FL016 S25FL032 S25FL064 S25FL128 S25FL129 S25FL256 S25FL512 S70FL01G	Spansion
SST25WF080	SST
N25Q032 N25Q064 N25Q128 N25Q256 N25Q512 N25Q00AA MT25Q01 MT25Q02 MT25Q512 MT25QL02G MT25QU02G MT35XU512ABA	Micron
MX66L1G45G MX66U1G45G	Macronix
IS25WP256D IS25LP256D IS25LWP512M IS25LP512M IS25WP064A IS25LP064A IS25WP032D IS25LP032D IS25WP016D IS25LP016D IS25WP080D IS25LP080D IS25LP128F IS25WP128F	ISSI

***Note*:** Intel, STM, and Numonyx serial flash devices are now a part of Serial Flash devices provided by Micron.

References

- Spartan-3AN FPGA In-System Flash User Guide (UG333):http://www.xilinx.com/support/documentation/user_guides/ug333.pdf
- Winbond Serial Flash Page:http://www.winbond.com/hq/product/code-storage-flash-memory/ serial-nor-flash/?__locale=en
- Intel (Numonyx) S33 Serial Flash Memory, SST SST25WF080, Micron N25Q flash family : https://www.micron.com/products/nor-flash/serial-nor-flash

XilIsf Library API

This section provides a linked summary and detailed descriptions of the Xillsf library APIs.

Table 180: **Quick Function Reference**

Type	Name	Arguments
int	XIsf_Initialize	XIsf * InstancePtr XIsf_Iface * SpiInstPtr u8 SlaveSelect u8 * WritePtr
int	XIsf_GetStatus	XIsf * InstancePtr u8 * ReadPtr
int	XIsf_GetStatusReg2	XIsf * InstancePtr u8 * ReadPtr
int	XIsf_GetDeviceInfo	XIsf * InstancePtr u8 * ReadPtr
int	XIsf_Transfer	void
u32	GetRealAddr	XIsf_Iface * QspiPtr u32 Address
int	XIsf_Write	XIsf * InstancePtr XIsf_WriteOperation Operation void * OpParamPtr
int	XIsf_Read	XIsf * InstancePtr XIsf_ReadOperation Operation void * OpParamPtr
int	XIsf_Erase	XIsf * InstancePtr XIsf_EraseOperation Operation u32 Address
int	XIsf_SectorProtect	XIsf * InstancePtr XIsf_SpOperation Operation u8 * BufferPtr
int	XIsf_Ioctl	XIsf * InstancePtr XIsf_IoctlOperation Operation
int	XIsf_WriteEnable	XIsf * InstancePtr u8 WriteEnable

Table 180: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	XIsf_RegisterInterface	XIsf * InstancePtr
int	XIsf_SetSpiConfiguration	XIsf * InstancePtr XIsf_Iface * SpiInstPtr u32 Options u8 PreScaler
void	XIsf_SetStatusHandler	XIsf * InstancePtr XIsf_Iface * XIfaceInstancePtr XIsf_StatusHandler XilIsf_Handler
void	XIsf_IfaceHandler	void * CallBackRef u32 StatusEvent unsigned int ByteCount

Functions

XIsf_Initialize

This API when called initializes the SPI interface with default settings.

With custom settings, user should call `XIsf_SetSpiConfiguration()` and then call this API. The geometry of the underlying Serial Flash is determined by reading the Joint Electron Device Engineering Council (JEDEC) Device Information and the Status Register of the Serial Flash.

***Note*:**

- The `XIsf_Initialize()` API is a blocking call (for both polled and interrupt modes of the Spi driver). It reads the JEDEC information of the device and waits till the transfer is complete before checking if the information is valid.
- This library can support multiple instances of Serial Flash at a time, provided they are of the same device family (either Atmel, Intel or STM, Winbond or Spansion) as the device family is selected at compile time.

Prototype

```
int XIsf_Initialize(XIsf *InstancePtr, XIsf_Iface *SpiInstPtr, u8
SlaveSelect, u8 *WritePtr);
```

Parameters

The following table lists the `XIsf_Initialize` function arguments.

Table 181: **XIsf_Initialize Arguments**

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
XIsf_Iface *	SpiInstPtr	Pointer to XIsf_Iface instance to be worked on.
u8	SlaveSelect	It is a 32-bit mask with a 1 in the bit position of slave being selected. Only one slave can be selected at a time.
u8 *	WritePtr	Pointer to the buffer allocated by the user to be used by the In-system and Serial Flash Library to perform any read/write operations on the Serial Flash device. User applications must pass the address of this buffer for the Library to work. • Write operations : ◦ The size of this buffer should be equal to the Number of bytes to be written to the Serial Flash + XISF_CMD_MAX_EXTRA_BYTES. ◦ The size of this buffer should be large enough for usage across all the applications that use a common instance of the Serial Flash. ◦ A minimum of one byte and a maximum of ISF_PAGE_SIZE bytes can be written to the Serial Flash, through a single Write operation. • Read operations : ◦ The size of this buffer should be equal to XISF_CMD_MAX_EXTRA_BYTES, if the application only reads from the Serial Flash (no write operations).

Returns

- XST_SUCCESS if successful.
- XST_DEVICE_IS_STOPPED if the device must be started before transferring data.
- XST_FAILURE, otherwise.

XIsf_GetStatus

This API reads the Serial Flash Status Register.

Note: The contents of the Status Register is stored at second byte pointed by the ReadPtr.

Prototype

```
int XIsf_GetStatus(XIsf *InstancePtr, u8 *ReadPtr);
```

Parameters

The following table lists the `XIsf_GetStatus` function arguments.

Table 182: **XIsf_GetStatus Arguments**

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
u8 *	ReadPtr	Pointer to the memory where the Status Register content is copied.

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_GetStatusReg2

This API reads the Serial Flash Status Register 2.

***Note*:** The contents of the Status Register 2 is stored at the second byte pointed by the ReadPtr. This operation is available only in Winbond Serial Flash.

Prototype

```
int XIsf_GetStatusReg2(XIsf *InstancePtr, u8 *ReadPtr);
```

Parameters

The following table lists the `XIsf_GetStatusReg2` function arguments.

Table 183: **XIsf_GetStatusReg2 Arguments**

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
u8 *	ReadPtr	Pointer to the memory where the Status Register content is copied.

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_GetDeviceInfo

This API reads the Joint Electron Device Engineering Council (JEDEC) information of the Serial Flash.

***Note*:** The Device information is stored at the second byte pointed by the ReadPtr.

Prototype

```
int XIsf_GetDeviceInfo(XIsf *InstancePtr, u8 *ReadPtr);
```

Parameters

The following table lists the `XIsf_GetDeviceInfo` function arguments.

Table 184: **XIsf_GetDeviceInfo Arguments**

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
u8 *	ReadPtr	Pointer to the buffer where the Device information is copied.

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_Transfer

Prototype

```
int XIsf_Transfer(XIsf *InstancePtr, u8 *WritePtr, u8 *ReadPtr, u32
ByteCount);
```

GetRealAddr

Function to get the real address of flash in case dual parallel and stacked configuration.

Function to get the real address of flash in case dual parallel and stacked configuration.

This functions translates the address based on the type of interconnection. In case of stacked, this function asserts the corresponding slave select.

Note: None.

Prototype

```
u32 GetRealAddr(XIsf_Iface *QspiPtr, u32 Address);
```

Parameters

The following table lists the `GetRealAddr` function arguments.

Table 185: **GetRealAddr Arguments**

Type	Name	Description
XIsf_Iface *	QspiPtr	is a pointer to XIsf_Iface instance to be worked on.
u32	Address	which is to be accessed (for erase, write or read)

Returns

RealAddr is the translated address - for single it is unchanged for stacked, the lower flash size is subtracted for parallel the address is divided by 2.

XIsf_Write

This API writes the data to the Serial Flash.

Operations

- Normal Write(XISF_WRITE), Dual Input Fast Program (XISF_DUAL_IP_PAGE_WRITE), Dual Input Extended Fast Program(XISF_DUAL_IP_EXT_PAGE_WRITE), Quad Input Fast Program(XISF_QUAD_IP_PAGE_WRITE), Quad Input Extended Fast Program (XISF_QUAD_IP_EXT_PAGE_WRITE):
  - The OpParamPtr must be of type struct XIsf_WriteParam.
    - OpParamPtr->Address is the start address in the Serial Flash.
    - OpParamPtr->WritePtr is a pointer to the data to be written to the Serial Flash.
    - OpParamPtr->NumBytes is the number of bytes to be written to Serial Flash.
    - This operation is supported for Atmel, Intel, STM, Winbond and Spansion Serial Flash.
- Auto Page Write (XISF_AUTO_PAGE_WRITE):
  - The OpParamPtr must be of 32 bit unsigned integer variable.
  - This is the address of page number in the Serial Flash which is to be refreshed.
  - This operation is only supported for Atmel Serial Flash.
- Buffer Write (XISF_BUFFER_WRITE):
  - The OpParamPtr must be of type struct XIsf_BufferToFlashWriteParam.
  - OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
  - OpParamPtr->WritePtr is a pointer to the data to be written to the Serial Flash SRAM Buffer.
  - OpParamPtr->ByteOffset is byte offset in the buffer from where the data is to be written.
  - OpParamPtr->NumBytes is number of bytes to be written to the Buffer. This operation is supported only for Atmel Serial Flash.
- Buffer To Memory Write With Erase (XISF_BUF_TO_PAGE_WRITE_WITH_ERASE)/ Buffer To Memory Write Without Erase (XISF_BUF_TO_PAGE_WRITE_WITHOUT_ERASE):
  - The OpParamPtr must be of type struct XIsf_BufferToFlashWriteParam.

- OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
- OpParamPtr->Address is starting address in the Serial Flash memory from where the data is to be written. These operations are only supported for Atmel Serial Flash.

- Write Status Register (XISF_WRITE_STATUS_REG):
    - The OpParamPtr must be of type of 8 bit unsigned integer variable. This is the value to be written to the Status Register.
    - This operation is only supported for Intel, STM Winbond and Spansion Serial Flash.
- Write Status Register2 (XISF_WRITE_STATUS_REG2):
    - The OpParamPtr must be of type (u8 *) and should point to two 8 bit unsigned integer values. This is the value to be written to the 16 bit Status Register. This operation is only supported in Winbond (W25Q) Serial Flash.
- One Time Programmable Area Write(XISF_OTP_WRITE):
    - The OpParamPtr must be of type struct XIsf_WriteParam.
    - OpParamPtr->Address is the address in the SRAM Buffer of the Serial Flash to which the data is to be written.
    - OpParamPtr->WritePtr is a pointer to the data to be written to the Serial Flash.
    - OpParamPtr->NumBytes should be set to 1 when performing OTPWrite operation. This operation is only supported for Intel Serial Flash.

***Note*:**

- Application must fill the structure elements of the third argument and pass its pointer by type casting it with void pointer.
- For Intel, STM, Winbond and Spansion Serial Flash, the user application must call the `XIsf_WriteEnable()` API by passing XISF_WRITE_ENABLE as an argument, before calling the `XIsf_Write()` API.

Prototype

```
int XIsf_Write(XIsf *InstancePtr, XIsf_WriteOperation Operation, void
*OpParamPtr);
```

Parameters

The following table lists the `XIsf_Write` function arguments.

Table 186: **XIsf_Write Arguments**

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.

Table 186: **XIsf_Write Arguments** *(cont'd)*

Type	Name	Description
XIsf_WriteOperation	Operation	Type of write operation to be performed on the Serial Flash. The different operations are • XISF_WRITE: Normal Write • XISF_DUAL_IP_PAGE_WRITE: Dual Input Fast Program • XISF_DUAL_IP_EXT_PAGE_WRITE: Dual Input Extended Fast Program • XISF_QUAD_IP_PAGE_WRITE: Quad Input Fast Program • XISF_QUAD_IP_EXT_PAGE_WRITE: Quad Input Extended Fast Program • XISF_AUTO_PAGE_WRITE: Auto Page Write • XISF_BUFFER_WRITE: Buffer Write • XISF_BUF_TO_PAGE_WRITE_WITH_ERASE: Buffer to Page Transfer with Erase • XISF_BUF_TO_PAGE_WRITE_WITHOUT_ERASE: Buffer to Page Transfer without Erase • XISF_WRITE_STATUS_REG: Status Register Write • XISF_WRITE_STATUS_REG2: 2 byte Status Register Write • XISF_OTP_WRITE: OTP Write.
void *	OpParamPtr	Pointer to a structure variable which contains operational parameters of the specified operation. This parameter type is dependant on value of first argument(Operation). For more details, refer `Operations` .

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_Read

This API reads the data from the Serial Flash.

Operations

- Normal Read (XISF_READ), Fast Read (XISF_FAST_READ), One Time Programmable Area Read(XISF_OTP_READ), Dual Output Fast Read (XISF_CMD_DUAL_OP_FAST_READ), Dual Input/Output Fast Read (XISF_CMD_DUAL_IO_FAST_READ), Quad Output Fast Read (XISF_CMD_QUAD_OP_FAST_READ) and Quad Input/Output Fast Read (XISF_CMD_QUAD_IO_FAST_READ):
    - The OpParamPtr must be of type struct XIsf_ReadParam.
    - OpParamPtr->Address is start address in the Serial Flash.

   - OpParamPtr->ReadPtr is a pointer to the memory where the data read from the Serial Flash is stored.
   - OpParamPtr->NumBytes is number of bytes to read.
   - OpParamPtr->NumDummyBytes is the number of dummy bytes to be transmitted for the Read command. This parameter is only used in case of Dual and Quad reads.
   - Normal Read and Fast Read operations are supported for Atmel, Intel, STM, Winbond and Spansion Serial Flash.
   - Dual and quad reads are supported for Winbond (W25QXX), Numonyx(N25QXX) and Spansion (S25FL129) quad flash.
   - OTP Read operation is only supported in Intel Serial Flash.
- Page To Buffer Transfer (XISF_PAGE_TO_BUF_TRANS):
   - The OpParamPtr must be of type struct XIsf_FlashToBufTransferParam .
   - OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
   - OpParamPtr->Address is start address in the Serial Flash. This operation is only supported in Atmel Serial Flash.
- Buffer Read (XISF_BUFFER_READ) and Fast Buffer Read(XISF_FAST_BUFFER_READ):
   - The OpParamPtr must be of type struct XIsf_BufferReadParam.
   - OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
   - OpParamPtr->ReadPtr is pointer to the memory where data read from the SRAM buffer is to be stored.
   - OpParamPtr->ByteOffset is byte offset in the SRAM buffer from where the first byte is read.
   - OpParamPtr->NumBytes is the number of bytes to be read from the Buffer. These operations are supported only in Atmel Serial Flash.

***Note*:**

- Application must fill the structure elements of the third argument and pass its pointer by type casting it with void pointer.
- The valid data is available from the fourth location pointed to by the ReadPtr for Normal Read and Buffer Read operations.
- The valid data is available from fifth location pointed to by the ReadPtr for Fast Read, Fast Buffer Read and OTP Read operations.
- The valid data is available from the (4 + NumDummyBytes)th location pointed to by ReadPtr for Dual/ Quad Read operations.

Prototype

```
int XIsf_Read(XIsf *InstancePtr, XIsf_ReadOperation Operation, void
*OpParamPtr);
```

Parameters

The following table lists the `XIsf_Read` function arguments.

Table 187: **XIsf_Read Arguments**

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
XIsf_ReadOperation	Operation	Type of the read operation to be performed on the Serial Flash. The different operations are • XISF_READ: Normal Read • XISF_FAST_READ: Fast Read • XISF_PAGE_TO_BUF_TRANS: Page to Buffer Transfer • XISF_BUFFER_READ: Buffer Read • XISF_FAST_BUFFER_READ: Fast Buffer Read • XISF_OTP_READ: One Time Programmable Area (OTP) Read • XISF_DUAL_OP_FAST_READ: Dual Output Fast Read • XISF_DUAL_IO_FAST_READ: Dual Input/Output Fast Read • XISF_QUAD_OP_FAST_READ: Quad Output Fast Read • XISF_QUAD_IO_FAST_READ: Quad Input/Output Fast Read
void *	OpParamPtr	Pointer to structure variable which contains operational parameter of specified Operation. This parameter type is dependant on the type of Operation to be performed. For more details, refer `Operations`.

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_Erase

This API erases the contents of the specified memory in the Serial Flash.

***Note*:**

- The erased bytes will read as 0xFF.
- For Intel, STM, Winbond or Spansion Serial Flash the user application must call `XIsf_WriteEnable()` API by passing XISF_WRITE_ENABLE as an argument before calling `XIsf_Erase()` API.

- Atmel Serial Flash support Page/Block/Sector Erase operations.
- Intel, Winbond, Numonyx (N25QXX) and Spansion Serial Flash support Sector/Block/Bulk Erase operations.
- STM (M25PXX) Serial Flash support Sector/Bulk Erase operations.

Prototype

```
int XIsf_Erase(XIsf *InstancePtr, XIsf_EraseOperation Operation, u32
Address);
```

Parameters

The following table lists the `XIsf_Erase` function arguments.

Table 188: **XIsf_Erase Arguments**

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
XIsf_EraseOperation	Operation	Type of Erase operation to be performed on the Serial Flash. The different operations are • XISF_PAGE_ERASE: Page Erase • XISF_BLOCK_ERASE: Block Erase • XISF_SECTOR_ERASE: Sector Erase • XISF_BULK_ERASE: Bulk Erase
u32	Address	Address of the Page/Block/Sector to be erased. The address can be either Page address, Block address or Sector address based on the Erase operation to be performed.

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_SectorProtect

This API is used for performing Sector Protect related operations.

***Note*:**

- The SPR content is stored at the fourth location pointed by the BufferPtr when performing XISF_SPR_READ operation.
- For Intel, STM, Winbond and Spansion Serial Flash, the user application must call the `XIsf_WriteEnable()` API by passing XISF_WRITE_ENABLE as an argument, before calling the `XIsf_SectorProtect()` API, for Sector Protect Register Write (XISF_SPR_WRITE) operation.
- Atmel Flash supports all these Sector Protect operations.

- Intel, STM, Winbond and Spansion Flash support only Sector Protect Read and Sector Protect Write operations.

Prototype

```
int XIsf_SectorProtect(XIsf *InstancePtr, XIsf_SpOperation Operation, u8
*BufferPtr);
```

Parameters

The following table lists the `XIsf_SectorProtect` function arguments.

Table 189: **XIsf_SectorProtect Arguments**

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
XIsf_SpOperation	Operation	Type of Sector Protect operation to be performed on the Serial Flash. The different operations are • XISF_SPR_READ: Read Sector Protection Register • XISF_SPR_WRITE: Write Sector Protection Register • XISF_SPR_ERASE: Erase Sector Protection Register • XISF_SP_ENABLE: Enable Sector Protection • XISF_SP_DISABLE: Disable Sector Protection
u8 *	BufferPtr	Pointer to the memory where the SPR content is read to/written from. This argument can be NULL if the Operation is SprErase, SpEnable and SpDisable.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if it fails.

XIsf_Ioctl

This API configures and controls the Intel, STM, Winbond and Spansion Serial Flash.

Note:

- Atmel Serial Flash does not support any of these operations.
- Intel Serial Flash support Enter/Release from DPD Mode and Clear Status Register Fail Flags.
- STM, Winbond and Spansion Serial Flash support Enter/Release from DPD Mode.
- Winbond (W25QXX) Serial Flash support Enable High Performance mode.

Prototype

```
int XIsf_Ioctl(XIsf *InstancePtr, XIsf_IoctlOperation Operation);
```

Parameters

The following table lists the `XIsf_Ioctl` function arguments.

Table 190: **XIsf_Ioctl Arguments**

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
XIsf_IoctlOperation	Operation	Type of Control operation to be performed on the Serial Flash. The different control operations are • XISF_RELEASE_DPD: Release from Deep Power Down (DPD) Mode • XISF_ENTER_DPD: Enter DPD Mode • XISF_CLEAR_SR_FAIL_FLAGS: Clear Status Register Fail Flags

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_WriteEnable

This API Enables/Disables writes to the Intel, STM, Winbond and Spansion Serial Flash.

Note: This API works only for Intel, STM, Winbond and Spansion Serial Flash. If this API is called for Atmel Flash, XST_FAILURE is returned.

Prototype

```
int XIsf_WriteEnable(XIsf *InstancePtr, u8 WriteEnable);
```

Parameters

The following table lists the `XIsf_WriteEnable` function arguments.

Table 191: **XIsf_WriteEnable Arguments**

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
u8	WriteEnable	Specifies whether to Enable (XISF_CMD_ENABLE_WRITE) or Disable (XISF_CMD_DISABLE_WRITE) the writes to the Serial Flash.

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_RegisterInterface

This API registers the interface SPI/SPI PS/QSPI PS.

Prototype

```
void XIsf_RegisterInterface(XIsf *InstancePtr);
```

Parameters

The following table lists the `XIsf_RegisterInterface` function arguments.

Table 192: **XIsf_RegisterInterface Arguments**

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.

Returns

None

XIsf_SetSpiConfiguration

This API sets the configuration of SPI.

This will set the options and clock prescaler (if applicable).

Note: This API can be called before calling `XIsf_Initialize()` to initialize the SPI interface in other than default options mode. PreScaler is only applicable to PS SPI/QSPI.

Prototype

```
int XIsf_SetSpiConfiguration(XIsf *InstancePtr, XIsf_Iface *SpiInstPtr, u32
Options, u8 PreScaler);
```

Parameters

The following table lists the `XIsf_SetSpiConfiguration` function arguments.

Table 193: **XIsf_SetSpiConfiguration Arguments**

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.

Table 193: **XIsf_SetSpiConfiguration Arguments** *(cont'd)*

Type	Name	Description
XIsf_Iface *	SpiInstPtr	Pointer to XIsf_Iface instance to be worked on.
u32	Options	Specified options to be set.
u8	PreScaler	Value of the clock prescaler to set.

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_SetStatusHandler

This API is to set the Status Handler when an interrupt is registered.

Note: None.

Prototype

```
void XIsf_SetStatusHandler(XIsf *InstancePtr, XIsf_Iface
*XIfaceInstancePtr, XIsf_StatusHandler XilIsf_Handler);
```

Parameters

The following table lists the `XIsf_SetStatusHandler` function arguments.

Table 194: **XIsf_SetStatusHandler Arguments**

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf Instance.
XIsf_Iface *	XIfaceInstancePtr	Pointer to the XIsf_Iface instance to be worked on.
XIsf_StatusHandler	XilIsf_Handler	Status handler for the application.

Returns

None

XIsf_IfaceHandler

This API is the handler which performs processing for the QSPI driver.

It is called from an interrupt context such that the amount of processing performed should be minimized. It is called when a transfer of QSPI data completes or an error occurs.

This handler provides an example of how to handle QSPI interrupts but is application specific.

Note: None.

Prototype

```
void XIsf_IfaceHandler(void *CallBackRef, u32 StatusEvent, unsigned int
ByteCount);
```

Parameters

The following table lists the `XIsf_IfaceHandler` function arguments.

Table 195: **XIsf_IfaceHandler Arguments**

Type	Name	Description
void *	CallBackRef	Reference passed to the handler.
u32	StatusEvent	Status of the QSPI .
unsigned int	ByteCount	Number of bytes transferred.

Returns

None

Library Parameters in MSS File

Xillsf Library can be integrated with a system using the following snippet in the Microprocessor Software Specification (MSS) file:

```
BEGIN LIBRARY`
PARAMETER LIBRARY_NAME = xilisf
PARAMETER LIBRARY_VER = 5.15
PARAMETER serial_flash_family = 1
PARAMETER serial_flash_interface = 1
END
```

The table below describes the libgen customization parameters.

Parameter	Default Value	Description
LIBRARY_NAME	xilisf	Specifies the library name.
LIBRARY_VER	5.15	Specifies the library version.
serial_flash_family	1	Specifies the serial flash family. Supported numerical values are: 1 = Xilinx In-system Flash or Atmel Serial Flash 2 = Intel (Numonyx) S33 Serial Flash 3 = STM (Numonyx) M25PXX/ N25QXX Serial Flash 4 = Winbond Serial Flash 5 = Spansion Serial Flash/Micron Serial Flash/Cypress Serial Flash 6 = SST Serial Flash

Parameter	Default Value	Description
Serial_flash_interface	1	Specifies the serial flash interface. Supported numerical values are: 1 = AXI QSPI Interface 2 = SPI PS Interface 3 = QSPI PS Interface or QSPI PSU Interface 4 = OSPIPSV Interface for OSPI

Note: Intel, STM, and Numonyx serial flash devices are now a part of Serial Flash devices provided by Micron.

XILINX

Chapter 6

XilFFS Library v4.3

XilFFS Library API Reference

The Xilinx fat file system (FFS) library consists of a file system and a glue layer. This FAT file system can be used with an interface supported in the glue layer. The file system code is open source and is used as it is. Currently, the Glue layer implementation supports the SD/eMMC interface and a RAM based file system. Application should make use of APIs provided in ff.h. These file system APIs access the driver functions through the glue layer.

The file system supports FAT16, FAT32, and exFAT (optional). The APIs are standard file system APIs. For more information, see the http://elm-chan.org/fsw/ff/00index_e.html.

***Note*:** The XilFFS library uses Revision R0.13b of the generic FAT filesystem module.

Library Files

The table below lists the file system files.

File	Description
ff.c	Implements all the file system APIs
ff.h	File system header
ffconf.h	File system configuration header – File system configurations such as READ_ONLY, MINIMAL, can be set here. This library uses FF_FS_MINIMIZE and FF_FS_TINY and Read/Write (NOT read only)

The table below lists the glue layer files.

File	Description
diskio.c	Glue layer – implements the function used by file system to call the driver APIs
ff.h	File system header
diskio.h	Glue layer header

Selecting a File System with an SD Interface

To select a file system with an SD interface:

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 **xilffs** library from the list of **Supported Libraries**.
15. Expand the **Overview** tree and select **xilffs**. The configuration options for xilffs are listed.
16. Configure the xilffs by setting the `fs_interface = 1` to select the `SD/eMMC`. This is the default value. Ensure that the SD/eMMC interface is available, prior to selecting the `fs_interface = 1` option.
17. Build the bsp and the application to use the file system with SD/eMMC. SD or eMMC will be recognized by the low level driver.

Selecting a RAM Based File System

To select a RAM based file system:

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 **xilffs** library from the list of **Supported Libraries**.
15. Expand the **Overview** tree and select **xilffs**. The configuration options for xilffs are listed.
16. Configure the xilffs by setting the `fs_interface = 2` to select the `RAM`.
17. As this project is used by LWIP based application, select **lwip library** and configure according to your requirements. For more information, see the LwIP Library API Reference documentation.
18. Use any lwip application that requires a RAM based file system - TCP/UDP performance test apps or tftp or webserver examples.
19. Build the bsp and the application to use the RAM based file system.

Library Parameters in MSS File

XilFFS Library can be integrated with a system using the following code snippet in the Microprocessor Software Specification (MSS) file:

```
    BEGIN LIBRARY
        PARAMETER LIBRARY_NAME = xilffs
        PARAMETER LIBRARY_VER = 4.3
        PARAMETER fs_interface = 1
        PARAMETER read_only = false
        PARAMETER use_lfn = 0
        PARAMETER enable_multi_partition = false
        PARAMETER num_logical_vol = 2
        PARAMETER use_mkfs = true
        PARAMETER use_strfunc = 0
        PARAMETER set_fs_rpath = 0
        PARAMETER enable_exfat = false
        PARAMETER word_access = true
        PARAMETER use_chmod = false
    END
```

The table below describes the libgen customization parameters.

Parameter	Default Value	Description
LIBRARY_NAME	xilffs	Specifies the library name.
LIBRARY_VER	4.3	Specifies the library version.
fs_interface	1 for SD/eMMC 2 for RAM	File system interface. SD/eMMC and RAM based file system are supported.
read_only	False	Enables the file system in Read Only mode, if true. Default is false. For Zynq UltraScale+ MPSoC devices, sets this option as true.
use_lfn	0	Enables the Long File Name(LFN) support if non-zero. 0: Disabled (Default) 1: LFN with static working buffer 2 (on stack) or 3 (on heap): Dynamic working buffer
enable_multi_partitio	False	Enables the multi partition support, if true.
num_logical_vol	2	Number of volumes (logical drives, from 1 to 10) to be used.
use_mkfs	True	Enables the mkfs support, if true. For Zynq UltraScale+ MPSoC devices, set this option as false.
use_strfunc	0	Enables the string functions (valid values 0 to 2). Default is 0.
set_fs_rpath	0	Configures relative path feature (valid values 0 to 2). Default is 0.
ramfs_size	3145728	Ram FS size is applicable only when RAM based file system is selected.
ramfs_start_addr	0x10000000	RAM FS start address is applicable only when RAM based file system is selected.

Parameter	Default Value	Description
enable_exfat	false	Enables support for exFAT file system. 0: Disable exFAT 1: Enable exFAT(Also Enables LFN)
word_access	True	Enables word access for misaligned memory access platform.
use_chmod	false	Enables use of CHMOD functionality for changing attributes (valid only with read_only set to false).

Chapter 7

XilSecure Library v4.2

Overview

The XilSecure library provides APIs to access cryptographic accelerators on the Zynq UltraScale+ MPSoC devices. The library is designed to run on top of Xilinx standalone BSPs. It is tested for A53, R5 and MicroBlaze. XilSecure is used during the secure boot process. The primary post-boot use case is to run this library on the PMU MicroBlaze with PMUFW to service requests from Uboot or Linux for cryptographic acceleration.

***Note*:** The XilSecure library does not check for memory bounds while performing cryptographic operations. You must check the bounds before using the functions provided in this library. If needed, you can take advantage of the XMPU, the XPPU, and/or TrustZone to limit memory access.

The XilSecure library includes:

- SHA-3/384 engine for 384 bit hash calculation.
- AES-GCM engine for symmetric key encryption and decryption using a 256-bit key.
- RSA engine for signature generation, signature verification, encryption and decryption. Key sizes supported include 2048, 3072, and 4096.

CAUTION! *SDK defaults to using a software stack in DDR and any variables used by XilSecure will be placed in DDR. For better security, change the linker settings to make sure the stack used by XilSecure is either in the OCM or the TCM.*

Board Support Package Settings

XilSecure provides an user configuration under BSP settings to enable or disable secure environment, this bsp parameter is valid only when BSP is build for the PMU MicroBlaze for post boot use cases and XilSecure is been accessed using the IPI response calls to PMUFW from Linux or U-boot or baremetal applications. When the application environment is secure and trusted this variable should be set to TRUE.

Parameter	Description
secure_environment	Default = FALSE. Set the value to TRUE to allow usage of device key through the IPI response calls.

By default, PMUFW will not allow device key for any decryption operation requested through IPI response unless authentication is enabled. If the user space is secure and trusted PMUFW can be build by setting the secure_environment variable. Only then the PMUFW allows usage of the device key for encrypting or decrypting the data blobs, decryption of bitstream or image.

Source Files

The source files for the library can be found at:

- https://github.com/Xilinx/embeddedsw/blob/master/lib/sw_services/xilsecure/
- https://github.com/Xilinx/embeddedsw/tree/master/lib/sw_services/xilsecure/src/common

AES-GCM

This software uses AES-GCM hardened cryptographic accelerator to encrypt or decrypt the provided data and requires a key of size 256 bits and initialization vector(IV) of size 96 bits.

XilSecure library supports the following features:

- Encryption of data with provided key and IV
- Decryption of data with provided key and IV
- Authentication using a GCM tag.
- Key loading based on key selection, the key can be either the user provided key loaded into the KUP key or the device key used during boot.

For either encryption or decryption the AES-GCM engine should be initialized first using the XSecure_AesInitiaze function.

AES Encryption Function Usage

When all the data to be encrypted is available, the `XSecure_AesEncryptData()` can be used. When all the data is not available, use the following functions in the suggested order:

1. `XSecure_AesEncryptInit()`
2. `XSecure_AesEncryptUpdate()` - This function can be called multiple times till input data is completed.

AES Decryption Function Usage

When all the data to be decrypted is available, the `XSecure_AesDecryptData()` can be used. When all the data is not available, use the following functions in the suggested order:

1. `XSecure_AesDecryptInit()`

2. `XSecure_AesDecryptUpdate()` - This function can be called multiple times till input data is completed.

During decryption, the passed in GCM tag will be compared to the GCM tag calculated by the engine. The two tags are then compared in the software and returned to the user as to whether or not the tags matched.

CAUTION! *when using the KUP key for encryption/decryption of the data, where the key is stored should be carefully considered. Key should be placed in an internal memory region that has access controls. Not doing so may result in security vulnerability.*

Table 196: **Quick Function Reference**

Type	Name	Arguments
s32	XSecure_AesInitialize	XSecure_Aes * InstancePtr XCsuDma * CsuDmaPtr u32 KeySel Iv Key
u32	XSecure_AesDecryptInit	XSecure_Aes * InstancePtr u8 * DecData u32 Size u8 * GcmTagAddr
s32	XSecure_AesDecryptUpdate	XSecure_Aes * InstancePtr u8 * EncData u32 Size
s32	XSecure_AesDecryptData	XSecure_Aes * InstancePtr u8 * DecData u8 * EncData u32 Size
s32	XSecure_AesDecrypt	XSecure_Aes * InstancePtr const u8 * Src u8 * Dst u32 Length
u32	XSecure_AesEncryptInit	XSecure_Aes * InstancePtr u8 * EncData u32 Size
u32	XSecure_AesEncryptUpdate	XSecure_Aes * InstancePtr const u8 * Data u32 Size

Table 196: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
u32	XSecure_AesEncryptData	XSecure_Aes * InstancePtr u8 * Dst const u8 * Src u32 Len
void	XSecure_AesReset	XSecure_Aes * InstancePtr

Functions

XSecure_AesInitialize

This function initializes the instance pointer.

Note: All the inputs are accepted in little endian format but the AES engine accepts the data in big endian format, The decryption and encryption functions in xsecure_aes handle the little endian to big endian conversion using few API's, Xil_Htonl (provided by Xilinx xil_io library) and XSecure_AesCsuDmaConfigureEndiannes for handling data endianness conversions. If higher performance is needed, users can strictly use data in big endian format and modify the xsecure_aes functions to remove the use of the Xil_Htonl and XSecure_AesCsuDmaConfigureEndiannes functions as required.

Prototype

```
s32 XSecure_AesInitialize(XSecure_Aes *InstancePtr, XCsuDma *CsuDmaPtr, u32
KeySel, u32 *IvPtr, u32 *KeyPtr);
```

Parameters

The following table lists the `XSecure_AesInitialize` function arguments.

Table 197: **XSecure_AesInitialize Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.
CsuDmaPtr	Pointer to the XCsuDma instance.
KeySel	Key source for decryption, can be KUP/device key • XSECURE_CSU_AES_KEY_SRC_KUP :For KUP key • XSECURE_CSU_AES_KEY_SRC_DEV :For Device Key
Iv	Pointer to the Initialization Vector for decryption
Key	Pointer to Aes key in case KUP key is used. Pass Null if the device key is to be used.

Returns

XST_SUCCESS if initialization was successful.

XSecure_AesDecryptInit

This function initializes the AES engine for decryption and is required to be called before calling XSecure_AesDecryptUpdate.

Note: If all of the data to be decrypted is available, the XSecure_AesDecryptData function can be used instead.

Prototype

```
u32 XSecure_AesDecryptInit(XSecure_Aes *InstancePtr, u8 *DecData, u32 Size,
u8 *GcmTagAddr);
```

Parameters

The following table lists the `XSecure_AesDecryptInit` function arguments.

Table 198: **XSecure_AesDecryptInit Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.
DecData	Pointer in which decrypted data will be stored.
Size	Expected size of the data in bytes whereas the number of bytes provided should be multiples of 4.
GcmTagAddr	Pointer to the GCM tag which needs to be verified during decryption of the data.

Returns

None

XSecure_AesDecryptUpdate

This function decrypts the encrypted data passed in and updates the GCM tag from any previous calls. The size from XSecure_AesDecryptInit is decremented from the size passed into this function to determine when the GCM tag passed to XSecure_AesDecryptInit needs to be compared to the GCM tag calculated in the AES engine.

Note: When Size of the data equals to size of the remaining data that data will be treated as final data. This API can be called multpile times but sum of all Sizes should be equal to Size mention in init. Return of the final call of this API tells whether GCM tag is matching or not.

Prototype

```
s32 XSecure_AesDecryptUpdate(XSecure_Aes *InstancePtr, u8 *EncData, u32
Size);
```

Parameters

The following table lists the `XSecure_AesDecryptUpdate` function arguments.

Table 199: **XSecure_AesDecryptUpdate Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.
EncData	Pointer to the encrypted data which needs to be decrypted.
Size	Expected size of data to be decrypted in bytes, whereas the number of bytes should be multiples of 4.

Returns

Final call of this API returns the status of GCM tag matching.

- XSECURE_CSU_AES_GCM_TAG_MISMATCH: If GCM tag is mismatched
- XSECURE_CSU_AES_ZEROIZATION_ERROR: If GCM tag is mismatched, zeroize the decrypted data and send the status of zeroization.
- XST_SUCCESS: If GCM tag is matching.

XSecure_AesDecryptData

This function decrypts the encrypted data provided and updates the DecData buffer with decrypted data.

***Note*:** When using this function to decrypt data that was encrypted with XSecure_AesEncryptData, the GCM tag will be stored as the last sixteen (16) bytes of data in XSecure_AesEncryptData's Dst (destination) buffer and should be used as the GcmTagAddr's pointer.

Prototype

```
s32 XSecure_AesDecryptData(XSecure_Aes *InstancePtr, u8 *DecData, u8
*EncData, u32 Size, u8 *GcmTagAddr);
```

Parameters

The following table lists the `XSecure_AesDecryptData` function arguments.

Table 200: **XSecure_AesDecryptData Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.
DecData	Pointer to a buffer in which decrypted data will be stored.
EncData	Pointer to the encrypted data which needs to be decrypted.
Size	Size of data to be decrypted in bytes, whereas the number of bytes should be multiples of 4.

Returns

This API returns the status of GCM tag matching.

- XSECURE_CSU_AES_GCM_TAG_MISMATCH: If GCM tag was mismatched
- XST_SUCCESS: If GCM tag was matched.

XSecure_AesDecrypt

This function will handle the AES-GCM Decryption.

***Note*:** This function is used for decrypting the Image's partition encrypted by Bootgen

Prototype

```
s32 XSecure_AesDecrypt(XSecure_Aes *InstancePtr, u8 *Dst, const u8 *Src,
u32 Length);
```

Parameters

The following table lists the `XSecure_AesDecrypt` function arguments.

Table 201: **XSecure_AesDecrypt Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.
Src	Pointer to encrypted data source location
Dst	Pointer to location where decrypted data will be written.
Length	Expected total length of decrypted image expected.

Returns

returns XST_SUCCESS if successful, or the relevant errorcode.

XSecure_AesEncryptInit

This function is used to initialize the AES engine for encryption.

Note: If all of the data to be encrypted is available, the XSecure_AesEncryptData function can be used instead.

Prototype

```
u32 XSecure_AesEncryptInit(XSecure_Aes *InstancePtr, u8 *EncData, u32 Size);
```

Parameters

The following table lists the `XSecure_AesEncryptInit` function arguments.

Table 202: **XSecure_AesEncryptInit Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.
EncData	Pointer of a buffer in which encrypted data along with GCM TAG will be stored. Buffer size should be Size of data plus 16 bytes.
Size	A 32 bit variable, which holds the size of the input data to be encrypted in bytes, whereas number of bytes provided should be multiples of 4.

Returns

None

XSecure_AesEncryptUpdate

This function encrypts the clear-text data passed in and updates the GCM tag from any previous calls. The size from XSecure_AesEncryptInit is decremented from the size passed into this function to determine when the final CSU DMA transfer of data to the AES-GCM cryptographic core.

Note: If all of the data to be encrypted is available, the XSecure_AesEncryptData function can be used instead.

Prototype

```
u32 XSecure_AesEncryptUpdate(XSecure_Aes *InstancePtr, const u8 *Data, u32
Size);
```

Parameters

The following table lists the `XSecure_AesEncryptUpdate` function arguments.

Table 203: **XSecure_AesEncryptUpdate Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.

Table 203: **XSecure_AesEncryptUpdate Arguments** *(cont'd)*

Name	Description
Data	Pointer to the data for which encryption should be performed.
Size	A 32 bit variable, which holds the size of the input data in bytes, whereas the number of bytes provided should be multiples of 4.

Returns

None

XSecure_AesEncryptData

This function encrypts Len (length) number of bytes of the passed in Src (source) buffer and stores the encrypted data along with its associated 16 byte tag in the Dst (destination) buffer.

***Note*:** If data to be encrypted is not available in one buffer one can call `XSecure_AesEncryptInit()` and update the AES engine with data to be encrypted by calling `XSecure_AesEncryptUpdate()` API multiple times as required.

Prototype

```
u32 XSecure_AesEncryptData(XSecure_Aes *InstancePtr, u8 *Dst, const u8
*Src, u32 Len);
```

Parameters

The following table lists the `XSecure_AesEncryptData` function arguments.

Table 204: **XSecure_AesEncryptData Arguments**

Name	Description
InstancePtr	A pointer to the XSecure_Aes instance.
Dst	A pointer to a buffer where encrypted data along with GCM tag will be stored. The Size of buffer provided should be Size of the data plus 16 bytes
Src	A pointer to input data for encryption.
Len	Size of input data in bytes, whereas the number of bytes provided should be multiples of 4.

Returns

None

XSecure_AesReset

This function sets and then clears the AES-GCM's reset line.

Prototype

```
void XSecure_AesReset(XSecure_Aes *InstancePtr);
```

Parameters

The following table lists the `XSecure_AesReset` function arguments.

Table 205: **XSecure_AesReset Arguments**

Name	Description
InstancePtr	is a pointer to the XSecure_Aes instance.

Returns

None

Definitions

XSecure_AesWaitForDone

This macro waits for AES engine completes configured operation.

Definition

```
#define XSecure_AesWaitForDoneXil_WaitForEvent((InstancePtr)->BaseAddress +
XSECURE_CSU_AES_STS_OFFSET,\
                    XSECURE_CSU_AES_STS_AES_BUSY,     \
                    0U,     \
                    XSECURE_AES_TIMEOUT_MAX)
```

Parameters

The following table lists the `XSecure_AesWaitForDone` definition values.

Table 206: **XSecure_AesWaitForDone Values**

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.

Returns

XST_SUCCESS if the AES engine completes configured operation. XST_FAILURE if a timeout has occurred.

AES-GCM Error Codes

The table below lists the AES-GCM error codes.

Error Code	Error Value	Description
XSECURE_CSU_AES_GCM_TAG_MISMATCH	0x1	User provided GCM tag does not match with GCM calculated on data
XSECURE_CSU_AES_IMAGE_LEN_MISMATCH	0x2	When there is a Image length mismatch
XSECURE_CSU_AES_DEVICE_COPY_ERROR	0x3	When there is device copy error.
XSECURE_CSU_AES_ZEROIZATION_ERROR	0x4	When there is an error with Zeroization. ***Note***: In case of any error during Aes decryption, we perform zeroization of the decrypted data.
XSECURE_CSU_AES_KEY_CLEAR_ERROR	0x20	Error when clearing key storage registers after Aes operation.

AES-GCM API Example Usage

The following example illustrates the usage of AES encryption and decryption APIs.

```
static s32 SecureAesExample(void)
{
        XCsuDma_Config *Config;
        s32 Status;
        u32 Index;
        XCsuDma CsuDmaInstance;
        XSecure_Aes Secure_Aes;

        /* Initialize CSU DMA driver */
        Config = XCsuDma_LookupConfig(XSECURE_CSUDMA_DEVICEID);
        if (NULL == Config) {
                return XST_FAILURE;
        }

        Status = XCsuDma_CfgInitialize(&CsuDmaInstance, Config,
                                        Config->BaseAddress);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        /* Initialize the Aes driver so that it's ready to use */
        XSecure_AesInitialize(&Secure_Aes, &CsuDmaInstance,
                                XSECURE_CSU_AES_KEY_SRC_KUP,
                                (u32 *)Iv, (u32 *)Key);

        xil_printf("Data to be encrypted: \n\r");
        for (Index = 0; Index < XSECURE_DATA_SIZE; Index++) {
                xil_printf("%02x", Data[Index]);
        }
        xil_printf( "\r\n\n");

        /* Encryption of Data */
        /*
         * If all the data to be encrypted is contiguous one can call
         * XSecure_AesEncryptData API directly.
         */
        XSecure_AesEncryptInit(&Secure_Aes, EncData, XSECURE_DATA_SIZE);
        XSecure_AesEncryptUpdate(&Secure_Aes, Data, XSECURE_DATA_SIZE);

        xil_printf("Encrypted data: \n\r");
        for (Index = 0; Index < XSECURE_DATA_SIZE; Index++) {
                xil_printf("%02x", EncData[Index]);
        }
        xil_printf( "\r\n");

        xil_printf("GCM tag: \n\r");
        for (Index = 0; Index < XSECURE_SECURE_GCM_TAG_SIZE; Index++) {
                xil_printf("%02x", EncData[XSECURE_DATA_SIZE + Index]);
        }
        xil_printf( "\r\n\n");

        /* Decrypt's the encrypted data */
        /*
         * If data to be decrypted is contiguous one can also call
         * single API XSecure_AesDecryptData
         */
        XSecure_AesDecryptInit(&Secure_Aes, DecData, XSECURE_DATA_SIZE,
                                        EncData + XSECURE_DATA_SIZE);
        /* Only the last update will return the GCM TAG matching status */
        Status = XSecure_AesDecryptUpdate(&Secure_Aes, EncData,
                                                 XSECURE_DATA_SIZE);
        if (Status != XST_SUCCESS) {
                xil_printf("Decryption failure- GCM tag was not matched\n
\r");
                return Status;
        }

        xil_printf("Decrypted data\n\r");
        for (Index = 0; Index < XSECURE_DATA_SIZE; Index++) {
                xil_printf("%02x", DecData[Index]);
        }
        xil_printf( "\r\n");

        /* Comparison of Decrypted Data with original data */
        for(Index = 0; Index < XSECURE_DATA_SIZE; Index++) {
                if (Data[Index] != DecData[Index]) {
                        xil_printf("Failure during comparison of the data\n
\r");
                        return XST_FAILURE;
                }
        }

        return XST_SUCCESS;
}
```

Note: Relevant examples are available in the <library-install-path>\examples folder. Where <library-install-path> is the XilSecure library installation path.

AES-GCM Usage to decrypt Boot Image

The Multiple key(Key Rolling) or Single key encrypted images will have the same format. The images include:

- Secure header - This includes the dummy AES key of 32byte + Block 0 IV of 12byte + DLC for Block 0 of 4byte + GCM tag of 16byte(Un-Enc).
- Block N - This includes the boot image data for the block N of n size + Block N+1 AES key of 32byte + Block N+1 IV of 12byte + GCM tag for Block N of 16byte(Un-Enc).

The Secure header and Block 0 will be decrypted using the device key or user provided key. If more than one block is found then the key and the IV obtained from previous block will be used for decryption.

Following are the instructions to decrypt an image:

1. Read the first 64 bytes and decrypt 48 bytes using the selected Device key.
2. Decrypt Block 0 using the IV + Size and the selected Device key.
3. After decryption, you will get the decrypted data+KEY+IV+Block Size. Store the KEY/IV into KUP/IV registers.
4. Using Block size, IV and the next Block key information, start decrypting the next block.
5. If the current image size is greater than the total image length, perform the next step. Else, go back to the previous step.
6. If there are failures, an error code is returned. Else, the decryption is successful.

RSA

The `xsecure_rsa.h` file contains hardware interface related information for the RSA hardware accelerator. This hardened cryptographic accelerator, within the CSU, performs the modulus math based on the Rivest-Shamir-Adelman (RSA) algorithm. It is an asymmetric algorithm.

Initialization & Configuration

The RSA driver instance can be initialized by using the `XSecure_RsaInitialize()` function. The method used for RSA implementation can take a pre-calculated value of `R^2 mod N`. If you do not have the pre-calculated exponential value pass NULL, the controller will take care of the exponential value.

***Note*:**

- From the RSA key modulus, the exponent should be extracted.
- For verification, PKCS v1.5 padding scheme has to be applied for comparing the data hash with decrypted hash.

Table 207: **Quick Function Reference**

Type	Name	Arguments
s32	XSecure_RsaInitialize	XSecure_Rsa * InstancePtr u8 * Mod u8 * ModExt u8 * ModExpo
u32	XSecure_RsaSignVerification	u8 * Signature u8 * Hash u32 HashLen
s32	XSecure_RsaPublicEncrypt	XSecure_Rsa * InstancePtr u8 * Input u32 Size u8 * Result
s32	XSecure_RsaPrivateDecrypt	XSecure_Rsa * InstancePtr u8 * Input u32 Size u8 * Result

Functions

XSecure_RsaInitialize

This function initializes a a XSecure_Rsa structure with the default values required for operating the RSA cryptographic engine.

Note: Modulus, ModExt and ModExpo are part of prtition signature when authenticated boot image is generated by bootgen, else the all of them should be extracted from the key.

Prototype

```
s32 XSecure_RsaInitialize(XSecure_Rsa *InstancePtr, u8 *Mod, u8 *ModExt, u8
*ModExpo);
```

Parameters

The following table lists the `XSecure_RsaInitialize` function arguments.

Table 208: **XSecure_RsaInitialize Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Rsa instance.

Table 208: **XSecure_RsaInitialize Arguments** *(cont'd)*

Name	Description
Mod	A character Pointer which contains the key Modulus of key size.
ModExt	A Pointer to the pre-calculated exponential (R^2 Mod N) value. • NULL - if user doesn't have pre-calculated R^2 Mod N value, control will take care of this calculation internally.
ModExpo	Pointer to the buffer which contains key exponent.

Returns

XST_SUCCESS if initialization was successful.

XSecure_RsaSignVerification

This function verifies the RSA decrypted data provided is either matching with the provided expected hash by taking care of PKCS padding.

Prototype

```
u32 XSecure_RsaSignVerification(u8 *Signature, u8 *Hash, u32 HashLen);
```

Parameters

The following table lists the `XSecure_RsaSignVerification` function arguments.

Table 209: **XSecure_RsaSignVerification Arguments**

Name	Description
Signature	Pointer to the buffer which holds the decrypted RSA signature
Hash	Pointer to the buffer which has the hash calculated on the data to be authenticated.
HashLen	Length of Hash used. • For SHA3 it should be 48 bytes • For SHA2 it should be 32 bytes

Returns

- XST_SUCCESS if decryption was successful.
- XST_FAILURE in case of mismatch.

XSecure_RsaPublicEncrypt

This function handles the RSA encryption with the public key components provided when initializing the RSA cryptographic core with the XSecure_RsaInitialize function.

Note: The Size passed here needs to match the key size used in the XSecure_RsaInitialize function.

Prototype

```
s32 XSecure_RsaPublicEncrypt(XSecure_Rsa *InstancePtr, u8 *Input, u32 Size,
u8 *Result);
```

Parameters

The following table lists the `XSecure_RsaPublicEncrypt` function arguments.

Table 210: **XSecure_RsaPublicEncrypt Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Rsa instance.
Input	Pointer to the buffer which contains the input data to be encrypted.
Size	Key size in bytes, Input size also should be same as Key size mentioned.Inputs supported are • XSECURE_RSA_4096_KEY_SIZE • XSECURE_RSA_2048_KEY_SIZE • XSECURE_RSA_3072_KEY_SIZE
Result	Pointer to the buffer where resultant decrypted data to be stored .

Returns

- XST_SUCCESS if encryption was successful.
- Error code on failure

XSecure_RsaPrivateDecrypt

This function handles the RSA decryption with the private key components provided when initializing the RSA cryptographic core with the XSecure_RsaInitialize function.

Note: The Size passed in needs to match the key size used in the XSecure_RsaInitialize function..

Prototype

```
s32 XSecure_RsaPrivateDecrypt(XSecure_Rsa *InstancePtr, u8 *Input, u32
Size, u8 *Result);
```

Parameters

The following table lists the `XSecure_RsaPrivateDecrypt` function arguments.

Table 211: **XSecure_RsaPrivateDecrypt Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Rsa instance.
Input	Pointer to the buffer which contains the input data to be decrypted.
Size	Key size in bytes, Input size also should be same as Key size mentioned. Inputs supported are • XSECURE_RSA_4096_KEY_SIZE, • XSECURE_RSA_2048_KEY_SIZE • XSECURE_RSA_3072_KEY_SIZE
Result	Pointer to the buffer where resultant decrypted data to be stored .

Returns

- XST_SUCCESS if decryption was successful.
- XSECURE_RSA_DATA_VALUE_ERROR - if input data is greater than modulus.
- XST_FAILURE - on RSA operation failure.

RSA API Example Usage

The following example illustrates the usage of the RSA library to encrypt data using the public key and to decrypt the data using private key.

***Note*:** Application should take care of the padding.

```
u32 SecureRsaExample(void)
{
        u32 Index;

        /* RSA signature decrypt with private key */
        /*
         * Initialize the Rsa driver with private key components
         * so that it's ready to use
         */
        XSecure_RsaInitialize(&Secure_Rsa, Modulus, NULL, PrivateExp);


        if(XST_SUCCESS != XSecure_RsaPrivateDecrypt(&Secure_Rsa, Data,
                                        Size, Signature))       {
                xil_printf("Failed at RSA signature decryption\n\r");
                return XST_FAILURE;
        }

        xil_printf("\r\n Decrypted Signature with private key\r\n ");

        for(Index = 0; Index < Size; Index++) {
                xil_printf(" %02x ", Signature[Index]);
        }
        xil_printf(" \r\n ");

        /* Verification if Data is expected */
        for(Index = 0; Index < Size; Index++) {
                if (Signature[Index] != ExpectedSign[Index]) {
                        xil_printf("\r\nError at verification of RSA 
signature"
                                        " Decryption\n\r");
                        return XST_FAILURE;
                }
        }

        /* RSA signature encrypt with Public key components */

        /*
         * Initialize the Rsa driver with public key components
         * so that it's ready to use
         */

        XSecure_RsaInitialize(&Secure_Rsa, Modulus, NULL, (u8 *)&PublicExp);

        if(XST_SUCCESS != XSecure_RsaPublicEncrypt(&Secure_Rsa, Signature,
                                        Size,
EncryptSignatureOut))     {
                xil_printf("\r\nFailed at RSA signature encryption\n\r");
                return XST_FAILURE;
        }
        xil_printf("\r\n Encrypted Signature with public key\r\n ");

        for(Index = 0; Index < Size; Index++) {
                xil_printf(" %02x ", EncryptSignatureOut[Index]);
        }

        /* Verification if Data is expected */
        for(Index = 0; Index < Size; Index++) {
                if (EncryptSignatureOut[Index] != Data[Index]) {
                        xil_printf("\r\nError at verification of RSA 
signature"
                                        " encryption\n\r");
                        return XST_FAILURE;
                }
        }


        return XST_SUCCESS;
}
```

Note: Relevant examples are available in the <library-install-path>\examples folder. Where <library-install-path> is the XilSecure library installation path.

SHA-3

This block uses the NIST-approved SHA-3 algorithm to generate a 384-bit hash on the input data. Because the SHA-3 hardware only accepts 104 byte blocks as the minimum input size, the input data will be padded with user selectable Keccak or NIST SHA-3 padding and is handled internally in the SHA-3 library.

Initialization & Configuration

The SHA-3 driver instance can be initialized using the `XSecure_Sha3Initialize()` function. A pointer to CsuDma instance has to be passed during initialization as the CSU DMA will be used for data transfers to the SHA module.

SHA-3 Function Usage

When all the data is available on which the SHA3 hash must be calculated, the `XSecure_Sha3Digest()` can be used with the appropriate parameters as described. When all the data is not available, use the SHA3 functions in the following order:

1. `XSecure_Sha3Start()`
2. `XSecure_Sha3Update()` - This function can be called multiple times until all input data has been passed to the SHA-3 cryptographic core.
3. `XSecure_Sha3Finish()` - Provides the final hash of the data. To get intermediate hash values after each `XSecure_Sha3Update()` , you can call `XSecure_Sha3_ReadHash()` after the `XSecure_Sha3Update()` call.

Table 212: **Quick Function Reference**

Type	Name	Arguments
s32	XSecure_Sha3Initialize	XSecure_Sha3 * InstancePtr XCsuDma * CsuDmaPtr
void	XSecure_Sha3Start	XSecure_Sha3 * InstancePtr
u32	XSecure_Sha3Update	XSecure_Sha3 * InstancePtr const u8 * Data const u32 Size
u32	XSecure_Sha3Finish	XSecure_Sha3 * InstancePtr u8 * Hash

Table 212: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
u32	XSecure_Sha3Digest	XSecure_Sha3 * InstancePtr const u8 * In const u32 Size u8 * Out
void	XSecure_Sha3_ReadHash	XSecure_Sha3 * InstancePtr u8 * Hash
s32	XSecure_Sha3PadSelection	XSecure_Sha3 * InstancePtr Sha3Type
s32	XSecure_Sha3LastUpdate	XSecure_Sha3 * InstancePtr
u32	XSecure_Sha3WaitForDone	XSecure_Sha3 * InstancePtr

Functions

XSecure_Sha3Initialize

This function initializes a XSecure_Sha3 structure with the default values required for operating the SHA3 cryptographic engine.

Note: The base address is initialized directly with value from xsecure_hw.h The default is NIST SHA3 padding, to change to KECCAK padding call `XSecure_Sha3PadSelection()` after `XSecure_Sha3Initialize()`.

Prototype

```
s32 XSecure_Sha3Initialize(XSecure_Sha3 *InstancePtr, XCsuDma *CsuDmaPtr);
```

Parameters

The following table lists the `XSecure_Sha3Initialize` function arguments.

Table 213: **XSecure_Sha3Initialize Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.
CsuDmaPtr	Pointer to the XCsuDma instance.

Returns

XST_SUCCESS if initialization was successful

XSecure_Sha3Start

This function configures Secure Stream Switch and starts the SHA-3 engine.

Prototype

```
void XSecure_Sha3Start(XSecure_Sha3 *InstancePtr);
```

Parameters

The following table lists the `XSecure_Sha3Start` function arguments.

Table 214: **XSecure_Sha3Start Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.

Returns

None

XSecure_Sha3Update

This function updates the SHA3 engine with the input data.

Prototype

```
u32 XSecure_Sha3Update(XSecure_Sha3 *InstancePtr, const u8 *Data, const u32
Size);
```

Parameters

The following table lists the `XSecure_Sha3Update` function arguments.

Table 215: **XSecure_Sha3Update Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.
Data	Pointer to the input data for hashing.
Size	Size of the input data in bytes.

Returns

XST_SUCCESS if the update is successful XST_FAILURE if there is a failure in SSS config

XSecure_Sha3Finish

This function updates SHA3 engine with final data which includes SHA3 padding and reads final hash on complete data.

Prototype

```
u32 XSecure_Sha3Finish(XSecure_Sha3 *InstancePtr, u8 *Hash);
```

Parameters

The following table lists the `XSecure_Sha3Finish` function arguments.

Table 216: **XSecure_Sha3Finish Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.
Hash	Pointer to location where resulting hash will be written

Returns

XST_SUCCESS if finished without any errors XST_FAILURE if Sha3PadType is other than KECCAK or NIST

XSecure_Sha3Digest

This function calculates the SHA-3 digest on the given input data.

Prototype

```
u32 XSecure_Sha3Digest(XSecure_Sha3 *InstancePtr, const u8 *In, const u32
Size, u8 *Out);
```

Parameters

The following table lists the `XSecure_Sha3Digest` function arguments.

Table 217: **XSecure_Sha3Digest Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.
In	Pointer to the input data for hashing
Size	Size of the input data

Table 217: **XSecure_Sha3Digest Arguments** *(cont'd)*

Name	Description
Out	Pointer to location where resulting hash will be written.

Returns

XST_SUCCESS if digest calculation done successfully XST_FAILURE if any error from Sha3Update or Sha3Finish.

XSecure_Sha3_ReadHash

This function reads the SHA3 hash of the data and it can be called between calls to XSecure_Sha3Update.

Prototype

```
void XSecure_Sha3_ReadHash(XSecure_Sha3 *InstancePtr, u8 *Hash);
```

Parameters

The following table lists the `XSecure_Sha3_ReadHash` function arguments.

Table 218: **XSecure_Sha3_ReadHash Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.
Hash	Pointer to a buffer in which read hash will be stored.

Returns

None

XSecure_Sha3PadSelection

This function provides an option to select the SHA-3 padding type to be used while calculating the hash.

***Note*:** The default provides support for NIST SHA-3. If a user wants to change the padding to Keccak SHA-3, this function should be called after `XSecure_Sha3Initialize()`

Prototype

```
s32 XSecure_Sha3PadSelection(XSecure_Sha3 *InstancePtr, XSecure_Sha3PadType
Sha3PadType);
```

Parameters

The following table lists the `XSecure_Sha3PadSelection` function arguments.

Table 219: **XSecure_Sha3PadSelection Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.
Sha3Type	Type of SHA3 padding to be used. • For NIST SHA-3 padding - XSECURE_CSU_NIST_SHA3 • For KECCAK SHA-3 padding - XSECURE_CSU_KECCAK_SHA3

Returns

XST_SUCCESS if pad selection is successful. XST_FAILURE if pad selecction is failed.

XSecure_Sha3LastUpdate

This function is to notify this is the last update of data where sha padding is also been included along with the data in the next update call.

Prototype

```
s32 XSecure_Sha3LastUpdate(XSecure_Sha3 *InstancePtr);
```

Parameters

The following table lists the `XSecure_Sha3LastUpdate` function arguments.

Table 220: **XSecure_Sha3LastUpdate Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.

Returns

XST_SUCCESS if last update can be accepted

XSecure_Sha3WaitForDone

This inline function waits till SHA3 completes its operation.

Prototype

```
u32 XSecure_Sha3WaitForDone(XSecure_Sha3 *InstancePtr);
```

Parameters

The following table lists the `XSecure_Sha3WaitForDone` function arguments.

Table 221: **XSecure_Sha3WaitForDone Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.

Returns

XST_SUCCESS if the SHA3 completes its operation. XST_FAILURE if a timeout has occurred.

SHA-3 API Example Usage

The xilsecure_sha_example.c file is a simple example application that demonstrates the usage of SHA-3 accelerator to calculate a 384-bit hash on the Hello World string. A typical use case for the SHA3 accelerator is for calcuation of the boot image hash as part of the autentication operation. This is illustrated in the xilsecure_rsa_example.c.

The contents of the xilsecure_sha_example.c file are shown below:

```
int SecureHelloWorldExample()
{
        u8 HelloWorld[4] = {'h','e','l','l'};
        u32 Size = sizeof(HelloWorld);
        u8 Out[384/8];
        XCsuDma_Config *Config;

        int Status;

        Config = XCsuDma_LookupConfig(0);
        if (NULL == Config) {
                xil_printf("config  failed\n\r");
                return XST_FAILURE;
        }

        Status = XCsuDma_CfgInitialize(&CsuDma, Config, Config-
>BaseAddress);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        /*
         * Initialize the SHA-3 driver so that it's ready to use
         */
        XSecure_Sha3Initialize(&Secure_Sha3, &CsuDma);

        XSecure_Sha3Digest(&Secure_Sha3, HelloWorld, Size, Out);

        xil_printf(" Calculated Digest \r\n ");
        int i= 0;
        for(i=0; i< (384/8); i++)
        {
                xil_printf(" %0x ", Out[i]);
        }
        xil_printf(" \r\n ");

        return XST_SUCCESS;
}
```

Note: The xilsecure_sha_example.c and xilsecure_rsa_example.c example files are available in the <library-install-path>\examples folder. Where <library-install-path> is the XilSecure library installation path.

XilSecure Utilities

The `xsecure_utils.h` file contains common functions used among the XilSecure library like holding hardware crypto engines in Reset or bringing them out of reset, and secure stream switch configuration for AES and SHA3.

Table 222: **Quick Function Reference**

Type	Name	Arguments
u32	XSecure_ReadReg	u32 BaseAddress u16 RegOffset
void	XSecure_WriteReg	u32 BaseAddress u32 RegOffset u32 RegisterValue
void	XSecure_SetReset	u32 BaseAddress u32 BaseAddress
void	XSecure_ReleaseReset	u32 BaseAddress u32 BaseAddress

Functions

XSecure_ReadReg

Read from the register.

Note: C-Style signature: u32 XSecure_ReadReg(u32 BaseAddress, u16 RegOffset)

Prototype

```
u32 XSecure_ReadReg(u32 BaseAddress, u16 RegOffset);
```

Parameters

The following table lists the `XSecure_ReadReg` function arguments.

Table 223: **XSecure_ReadReg Arguments**

Name	Description
BaseAddress	contains the base address of the device.
RegOffset	contains the offset from the base address of the device.

Returns

The value read from the register.

XSecure_WriteReg

Write to the register.

***Note*:** C-Style signature: void XSecure_WriteReg(u32 BaseAddress, u16 RegOffset, u16 RegisterValue)

Prototype

```
void XSecure_WriteReg(u32 BaseAddress, u32 RegOffset, u32 RegisterValue);
```

Parameters

The following table lists the `XSecure_WriteReg` function arguments.

Table 224: **XSecure_WriteReg Arguments**

Name	Description
BaseAddress	contains the base address of the device.
RegOffset	contains the offset from the base address of the device.
RegisterValue	is the value to be written to the register

Returns

None.

XSecure_SetReset

This function places the hardware core into the reset.

Prototype

```
void XSecure_SetReset(u32 BaseAddress, u32 Offset);
```

Parameters

The following table lists the `XSecure_SetReset` function arguments.

Table 225: **XSecure_SetReset Arguments**

Name	Description
BaseAddress	Base address of the core.
BaseAddress	Offset of the reset register.

Returns

None

XSecure_ReleaseReset

This function takes the hardware core out of reset.

Prototype

```
void XSecure_ReleaseReset(u32 BaseAddress, u32 Offset);
```

Parameters

The following table lists the `XSecure_ReleaseReset` function arguments.

Table 226: **XSecure_ReleaseReset Arguments**

Name	Description
BaseAddress	Base address of the core.
BaseAddress	Offset of the reset register.

Returns

None

Chapter 8

XilSkey Library v4.9

Overview

The XilSKey library provides APIs for programming and reading eFUSE bits and for programming the battery-backed RAM (BBRAM) of Zynq-7000 SoC, UltraScale, UltraScale+ and the Zynq UltraScale+ MPSoC devices.

- In Zynq-7000 devices:
    - PS eFUSE holds the RSA primary key hash bits and user feature bits, which can enable or disable some Zynq-7000 processor features.
    - PL eFUSE holds the AES key, the user key and some of the feature bits.
    - PL BBRAM holds the AES key.
- In Kintex/Virtex UltraScale or UltraScale+:
    - PL eFUSE holds the AES key, 32 bit and 128 bit user key, RSA hash and some of the feature bits.
    - PL BBRAM holds AES key with or without DPA protection enable or obfuscated key programming.
- In Zynq UltraScale+ MPSoC:
    - PUF registration and Regeneration.
    - PS eFUSE holds:

Programming AES key and can perform CRC verification of AES key

- Programming/Reading User fuses
- Programming/Reading PPK0/PPK1 sha3 hash
- Programming/Reading SPKID
- Programming/Reading secure control bits
    - PS BBRAM holds the AES key.
    - PL eFUSE holds the AES key, 32 bit and 128 bit user key, RSA hash and some of the feature bits.

- PL BBRAM holds AES key with or without DPA protection enable or obfuscated key programming.

Board Support Package Settings

There are few configurable parameters available under bsp settings, which can be configured during compilation of board support package.

Configurations For Adding New device

The below configurations helps in adding new device information not supported by default. Currently, MicroBlaze, Zynq UltraScale and Zynq UltraScale+ MPSoC devices are supported.

Parameter Name	Description
device_id	Mention the device ID
device_irlen	Mention IR length of the device. Default is 0
device_numslr	Mention number of SLRs available. Range of values can be 1 to 4. Default is 1. If no slaves are present and only one master SLR is available then only 1 number of SLR is available.
device_series	Select the device series. Default is FPGA SERIES ZYNQ. The following device series are supported:XSK_FPGA_SERIES_ZYNQ - Select if the device belongs to the Zynq-7000 family. XSK_FPGA_SERIES_ULTRA - Select if the device belongs to the Zynq UltraScale family. XSK_FPGA_SERIES_ULTRA_PLUS - Select if the device belongs to Zynq UltraScale MPSoC family.
device_masterslr	Mention the master SLR number. Default is 0.

Configurations For Zynq UltraScale+ MPSoC devices

Parameter Name	Description
override_sysmon_cfg	Default = TRUE, library configures sysmon before accessing efuse memory. If you are using the Sysmon library and XilSkey library together, XilSkey overwrites the user defined sysmon configuration by default. When override_sysmon_cfg is set to false, XilSkey expects you to configure the sysmon to read the 3 ADC channels - Supply 1 (VPINT), Supply 3 (VPAUX) and LPD Temperature. XilSkey validates the user defined sysmon configuration is correct before performing the eFuse operations.

***Note*:** On Ultrascale and Ultrascale plus devices there can be multiple or single SLRs and among which one can be master and the others are slaves, where SLR 0 is not always the master SLR. Based on master and slave SLR order SLRs in this library are referred with config order index. Master SLR is mentioned with CONFIG ORDER 0, then follows the slaves config order, CONFIG ORDER 1,2 and 3 are for slaves in order. Due to the added support for the SSIT devices, it is recommended to use the updated library with updated examples only for the UltraScale and the UltraScale+ devices.

Hardware Setup

This section describes the hardware setup required for programming PL BBRAM or PL eFUSE.

Hardware setup for Zynq PL

This section describes the hardware setup required for programming BBRAM or eFUSE of Zynq PL devices. PL eFUSE or PL BBRAM is accessed through PS via MIO pins which are used for communication PL eFUSE or PL BBRAM through JTAG signals, these can be changed depending on the hardware setup. A hardware setup which dedicates four MIO pins for JTAG signals should be used and the MIO pins should be mentioned in application header file (xilskey_input.h). There should be a method to download this example and have the MIO pins connected to JTAG before running this application. You can change the listed pins at your discretion.

MUX Usage Requirements

To write the PL eFUSE or PL BBRAM using a driver you must:

- Use four MIO lines (TCK,TMS,TDO,TDI)
- Connect the MIO lines to a JTAG port

If you want to switch between the external JTAG and JTAG operation driven by the MIOs, you must:

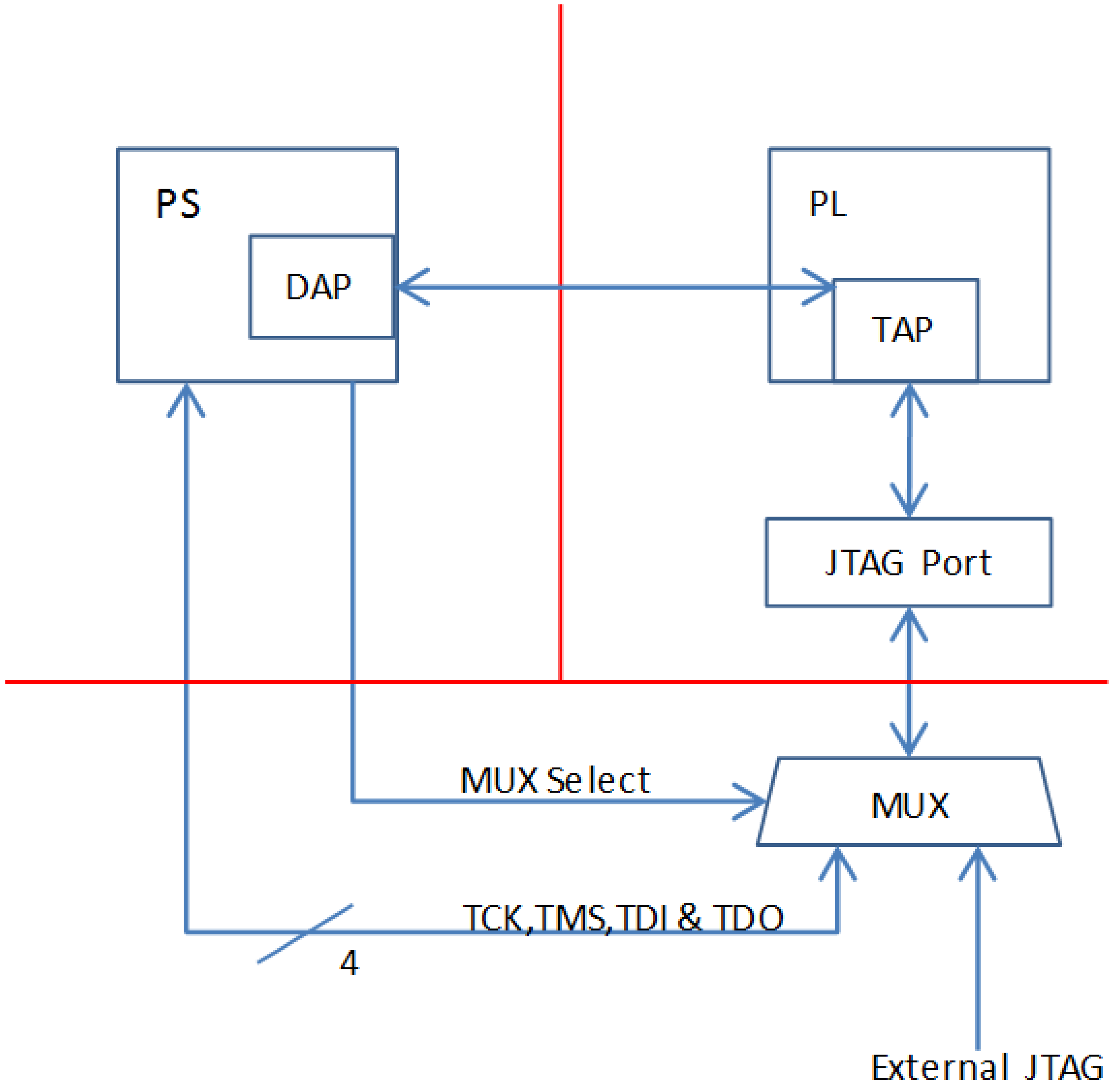
- Include a MUX between the external JTAG and the JTAG operation driven by the MIOs
- Assign a MUX selection PIN

To rephrase, to select JTAG for PL EFUSE or PL BBRAM writing, you must define the following:

- The MIOs used for JTAG operations (TCK,TMS,TDI,TDO).
- The MIO used for the MUX Select Line.
- The Value on the MUX Select line, to select JTAG for PL eFUSE or PL BBRAM writing.

The following graphic illustrates the correct MUX usage:

Figure 2: **MUX Usage**



Note: If you use the Vivado Device Programmer tool to burn PL eFUSEs, there is no need for MUX circuitry or MIO pins.

Hardware setup for UltraScale or UltraScale+

This section describes the hardware setup required for programming BBRAM or eFUSE of UltraScale devices. Accessing UltraScale MicroBlaze eFuse is done by using block RAM initialization. UltraScale eFUSE programming is done through MASTER JTAG. Crucial Programming sequence will be taken care by Hardware module. It is mandatory to add Hardware module in the design. Use hardware module's vhd code and instructions provided to add Hardware module in the design.

- You need to add the Master JTAG primitive to design, that is, the MASTER_JTAG_inst instantiation has to be performed and AXI GPIO pins have to be connected to TDO, TDI, TMS and TCK signals of the MASTER_JTAG primitive.
- For programming eFUSE, along with master JTAG, hardware module(HWM) has to be added in design and it's signals XSK_EFUSEPL_AXI_GPIO_HWM_READY , XSK_EFUSEPL_AXI_GPIO_HWM_END and XSK_EFUSEPL_AXI_GPIO_HWM_START, needs to be connected to AXI GPIO pins to communicate with HWM. Hardware module is not mandatory for programming BBRAM. If your design has a HWM, it is not harmful for accessing BBRAM.
- All inputs (Master JTAG's TDO and HWM's HWM_READY, HWM_END) and all outputs (Master JTAG TDI, TMS, TCK and HWM's HWM_START) can be connected in one channel (or) inputs in one channel and outputs in other channel.
- Some of the outputs of GPIO in one channel and some others in different channels are not supported.
- The design should contain AXI BRAM control memory mapped (1MB).

Note: MASTER_JTAG will disable all other JTAGs.

For providing inputs of MASTER JTAG signals and HWM signals connected to the GPIO pins and GPIO channels, refer GPIO Pins Used for PL Master JTAG Signal and GPIO Channels sections of the UltraScale User-Configurable PL eFUSE Parameters and UltraScale User-Configurable PL BBRAM Parameters. The procedure for programming BBRAM of eFUSE of UltraScale or UltraScale+ can be referred at UltraScale BBRAM Access Procedure and UltraScale eFUSE Access Procedure.

Source Files

The following is a list of eFUSE and BBRAM application project files, folders and macros.

- xilskey_efuse_example.c: This file contains the main application code. The file helps in the PS/PL structure initialization and writes/reads the PS/PL eFUSE based on the user settings provided in the xilskey_input.h file.
- xilskey_input.h: This file ontains all the actions that are supported by the eFUSE library. Using the preprocessor directives given in the file, you can read/write the bits in the PS/PL eFUSE. More explanation of each directive is provided in the following sections. Burning or reading the PS/PL eFUSE bits is based on the values set in the xilskey_input.h file. Also contains GPIO pins and channels connected to MASTER JTAG primitive and hardware module to access Ultrascale eFUSE.

  In this file:

  - specify the 256 bit key to be programmed into BBRAM.
  - specify the AES(256 bit) key, User (32 bit and 128 bit) keys and RSA key hash(384 bit) key to be programmed into UltraScale eFUSE.

  - XSK_EFUSEPS_DRIVER: Define to enable the writing and reading of PS eFUSE.
  - XSK_EFUSEPL_DRIVER: Define to enable the writing of PL eFUSE.

- xilskey_bbram_example.c: This file contains the example to program a key into BBRAM and verify the key.

  Note: This algorithm only works when programming and verifying key are both executed in the recommended order.

- xilskey_efuseps_zynqmp_example.c: This file contains the example code to program the PS eFUSE and read back of eFUSE bits from the cache.
- xilskey_efuseps_zynqmp_input.h: This file contains all the inputs supported for eFUSE PS of Zynq UltraScale+ MPSoC. eFUSE bits are programmed based on the inputs from the xilskey_efuseps_zynqmp_input.h file.
- xilskey_bbramps_zynqmp_example.c: This file contains the example code to program and verify BBRAM key of Zynq UltraScale+ MPSoC. Default is zero. You can modify this key on top of the file.
- xilskey_bbram_ultrascale_example.c: This file contains example code to program and verify BBRAM key of UltraScale.

  Note: Programming and verification of BBRAM key cannot be done separately.

- xilskey_bbram_ultrascale_input.h: This file contains all the preprocessor directives you need to provide. In this file, specify BBRAM AES key or Obfuscated AES key to be programmed, DPA protection enable and, GPIO pins and channels connected to MASTER JTAG primitive.
- xilskey_puf_registration.c: This file contains all the PUF related code. This example illustrates PUF registration and generating black key and programming eFUSE with PUF helper data, CHash and Auxilary data along with the Black key.
- xilskey_puf_registration.h: This file contains all the preprocessor directives based on which read/write the eFUSE bits and Syndrome data generation. More explanation of each directive is provided in the following sections.

CAUTION! *Ensure that you enter the correct information before writing or 'burning' eFUSE bits. Once burned, they cannot be changed. The BBRAM key can be programmed any number of times.*

Note: POR reset is required for the eFUSE values to be recognized.

BBRAM PL API

This section provides a linked summary and detailed descriptions of the battery-backed RAM (BBRAM) APIs of Zynq PL and UltraScale devices.

Example Usage

- Zynq BBRAM PL example usage:
  - The Zynq BBRAM PL example application should contain the xilskey_bbram_example.c and xilskey_input.h files.
  - You should provide user configurable parameters in the xilskey_input.h file. For more information, refer `Zynq User-Configurable PL BBRAM Parameters`.
- UltraScale BBRAM example usage:
  - The UltraScale BBRAM example application should contain the xilskey_bbram_ultrascale_input.h and xilskey_bbram_ultrascale_example.c files.
  - You should provide user configurable parameters in the xilskey_bbram_ultrascale_input.h file. For more information, refer `UltraScale or UltraScale+ User-Configurable BBRAM PL Parameters`.

Note: It is assumed that you have set up your hardware prior to working on the example application. For more information, refer `Hardware Setup`.

Table 227: **Quick Function Reference**

Type	Name	Arguments
int	XilSKey_Bbram_Program	XilSKey_Bbram * InstancePtr
int	XilSKey_Bbram_JTAGServerInit	void

Functions

XilSKey_Bbram_Program

This function implements the BBRAM algorithm for programming and verifying key.

The program and verify will only work together in and in that order.

Note: This function will program BBRAM of Ultrascale and Zynq as well.

Prototype

```
int XilSKey_Bbram_Program(XilSKey_Bbram *InstancePtr);
```

Parameters

The following table lists the `XilSKey_Bbram_Program` function arguments.

Table 228: **XilSKey_Bbram_Program Arguments**

Type	Name	Description
XilSKey_Bbram *	InstancePtr	Pointer to XilSKey_Bbram

Returns

XilSKey_Bbram_JTAGServerInit

Prototype

```
int XilSKey_Bbram_JTAGServerInit(XilSKey_Bbram *InstancePtr);
```

Zynq UltraScale+ MPSoC BBRAM PS API

This section provides a linked summary and detailed descriptions of the battery-backed RAM (BBRAM) APIs for Zynq UltraScale+ MPSoC devices.

Example Usage

- The Zynq UltraScale+ MPSoc example application should contain the xilskey_bbramps_zynqmp_example.c file.
- User configurable key can be modified in the same file (xilskey_bbramps_zynqmp_example.c), at the XSK_ZYNQMP_BBRAMPS_AES_KEY macro.

Table 229: **Quick Function Reference**

Type	Name	Arguments
u32	XilSKey_ZynqMp_Bbram_Program	u32 * AesKey
u32	XilSKey_ZynqMp_Bbram_Zeroise	None.

Functions

XilSKey_ZynqMp_Bbram_Program

This function implements the BBRAM programming and verifying the key written.

Program and verification of AES will work only together. CRC of the provided key will be calculated internally and verified after programming.

Prototype

```
u32 XilSKey_ZynqMp_Bbram_Program(u32 *AesKey);
```

Parameters

The following table lists the `XilSKey_ZynqMp_Bbram_Program` function arguments.

Table 230: **XilSKey_ZynqMp_Bbram_Program Arguments**

Type	Name	Description
u32 *	AesKey	Pointer to the key which has to be programmed.

Returns

- Error code from XskZynqMp_Ps_Bbram_ErrorCodes enum if it fails
- XST_SUCCESS if programming is done.

XilSKey_ZynqMp_Bbram_Zeroise

This function zeroize's Bbram Key.

Note: BBRAM key will be zeroized.

Prototype

```
u32 XilSKey_ZynqMp_Bbram_Zeroise(void);
```

Parameters

The following table lists the `XilSKey_ZynqMp_Bbram_Zeroise` function arguments.

Table 231: **XilSKey_ZynqMp_Bbram_Zeroise Arguments**

Type	Name	Description
Commented parameter None. does not exist in function XilSKey_ZynqMp_Bbram_Zeroise.	None.	

Returns

None.

Zynq eFUSE PS API

This chapter provides a linked summary and detailed descriptions of the Zynq eFUSE PS APIs.

Example Usage

- The Zynq eFUSE PS example application should contain the xilskey_efuse_example.c and the xilskey_input.h files.
- There is no need of any hardware setup. By default, both the eFUSE PS and PL are enabled in the application. You can comment 'XSK_EFUSEPL_DRIVER' to execute only the PS. For more details, refer `Zynq User-Configurable PS eFUSE Parameters`.

Table 232: **Quick Function Reference**

Type	Name	Arguments
u32	XilSKey_EfusePs_Write	InstancePtr
u32	XilSKey_EfusePs_Read	InstancePtr
u32	XilSKey_EfusePs_ReadStatus	XilSKey_EPs * InstancePtr u32 * StatusBits

Functions

XilSKey_EfusePs_Write

PS eFUSE interface functions.

PS eFUSE interface functions.

Note: When called, this Initializes the timer, XADC subsystems. Unlocks the PS eFUSE controller.Configures the PS eFUSE controller. Writes the hash and control bits if requested. Programs the PS eFUSE to enable the RSA authentication if requested. Locks the PS eFUSE controller. Returns an error, if the reference clock frequency is not in between 20 and 60 MHz or if the system not in a position to write the requested PS eFUSE bits (because the bits are already written or not allowed to write) or if the temperature and voltage are not within range

Prototype

```
u32 XilSKey_EfusePs_Write(XilSKey_EPs *PsInstancePtr);
```

Parameters

The following table lists the `XilSKey_EfusePs_Write` function arguments.

Table 233: **XilSKey_EfusePs_Write Arguments**

Type	Name	Description
Commented parameter InstancePtr does not exist in function XilSKey_EfusePs_Write.	InstancePtr	Pointer to the PsEfuseHandle which describes which PS eFUSE bit should be burned.

Returns

- XST_SUCCESS.
- In case of error, value is as defined in xilskey_utils.h Error value is a combination of Upper 8 bit value and Lower 8 bit value. For example, 0x8A03 should be checked in error.h as 0x8A00 and 0x03. Upper 8 bit value signifies the major error and lower 8 bit values tells more precisely.

XilSKey_EfusePs_Read

This function is used to read the PS eFUSE.

Note: When called: This API initializes the timer, XADC subsystems. Unlocks the PS eFUSE Controller. Configures the PS eFUSE Controller and enables read-only mode. Reads the PS eFUSE (Hash Value), and enables read-only mode. Locks the PS eFUSE Controller. Returns an error, if the reference clock frequency is not in between 20 and 60MHz. or if unable to unlock PS eFUSE controller or requested address corresponds to restricted bits. or if the temperature and voltage are not within range

Prototype

```
u32 XilSKey_EfusePs_Read(XilSKey_EPs *PsInstancePtr);
```

Parameters

The following table lists the `XilSKey_EfusePs_Read` function arguments.

Table 234: **XilSKey_EfusePs_Read Arguments**

Type	Name	Description
Commented parameter InstancePtr does not exist in function XilSKey_EfusePs_Read.	InstancePtr	Pointer to the PsEfuseHandle which describes which PS eFUSE should be burned.

Returns

- XST_SUCCESS no errors occurred.

- In case of error, value is as defined in xilskey_utils.h. Error value is a combination of Upper 8 bit value and Lower 8 bit value. For example, 0x8A03 should be checked in error.h as 0x8A00 and 0x03. Upper 8 bit value signifies the major error and lower 8 bit values tells more precisely.

XilSKey_EfusePs_ReadStatus

This function is used to read the PS efuse status register.

***Note*:** This API unlocks the controller and reads the Zynq PS eFUSE status register.

Prototype

```
u32 XilSKey_EfusePs_ReadStatus(XilSKey_EPs *InstancePtr, u32 *StatusBits);
```

Parameters

The following table lists the `XilSKey_EfusePs_ReadStatus` function arguments.

Table 235: **XilSKey_EfusePs_ReadStatus Arguments**

Type	Name	Description
XilSKey_EPs *	InstancePtr	Pointer to the PS eFUSE instance.
u32 *	StatusBits	Buffer to store the status register read.

Returns

- XST_SUCCESS.
- XST_FAILURE

Zynq UltraScale+ MPSoC eFUSE PS API

This chapter provides a linked summary and detailed descriptions of the Zynq MPSoC UltraScale + eFUSE PS APIs.

Example Usage

- For programming eFUSEs other than the PUF, the Zynq UltraScale+ MPSoC example application should contain the xilskey_efuseps_zynqmp_example.c and the xilskey_efuseps_zynqmp_input.h files.
- For PUF registration, programming PUF helper data, AUX, chash, and black key, the Zynq UltraScale+ MPSoC example application should contain the xilskey_puf_registration.c and the xilskey_puf_registration.h files.

- For more details on the user configurable parameters, refer `Zynq UltraScale+ MPSoC User-Configurable PS eFUSE Parameters` and `Zynq UltraScale+ MPSoC User-Configurable PS PUF Parameters`.

Table 236: **Quick Function Reference**

Type	Name	Arguments
u32	XilSKey_ZynqMp_EfusePs_CheckAesKeyCrc	u32 CrcValue
u32	XilSKey_ZynqMp_EfusePs_ReadUserFuse	u32 * UseFusePtr u8 UserFuse_Num u8 ReadOption
u32	XilSKey_ZynqMp_EfusePs_ReadPpk0Hash	u32 * Ppk0Hash u8 ReadOption
u32	XilSKey_ZynqMp_EfusePs_ReadPpk1Hash	u32 * Ppk1Hash u8 ReadOption
u32	XilSKey_ZynqMp_EfusePs_ReadSpkId	u32 * SpkId u8 ReadOption
void	XilSKey_ZynqMp_EfusePs_ReadDna	u32 * DnaRead
u32	XilSKey_ZynqMp_EfusePs_ReadSecCtrlBits	XilSKey_SecCtrlBits * ReadBackSecCtrlBits u8 ReadOption
u32	XilSKey_ZynqMp_EfusePs_CacheLoad	void
u32	XilSKey_ZynqMp_EfusePs_Write	XilSKey_ZynqMpEPs * InstancePtr
u32	XilSkey_ZynqMpEfuseAccess	void
u32	XilSKey_ZynqMp_EfusePs_WritePufHelprData	XilSKey_Puf * InstancePtr
u32	XilSKey_ZynqMp_EfusePs_ReadPufHelprData	u32 * Address
u32	XilSKey_ZynqMp_EfusePs_WritePufChash	XilSKey_Puf * InstancePtr
u32	XilSKey_ZynqMp_EfusePs_ReadPufChash	u32 * Address u8 ReadOption
u32	XilSKey_ZynqMp_EfusePs_WritePufAux	XilSKey_Puf * InstancePtr

Table 236: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
u32	XilSKey_ZynqMp_EfusePs_ReadPufAux	u32 * Address u8 ReadOption
u32	XilSKey_Write_Puf_EfusePs_SecureBits	XilSKey_Puf_Secure * WriteSecureBits
u32	XilSKey_Read_Puf_EfusePs_SecureBits	SecureBits u8 ReadOption
u32	XilSKey_Puf_Registration	XilSKey_Puf * InstancePtr
u32	XilSKey_Puf_Regeneration	XilSKey_Puf * InstancePtr

Functions

XilSKey_ZynqMp_EfusePs_CheckAesKeyCrc

This function performs the CRC check of AES key.

Note: For Calculating the CRC of the AES key use the `XilSKey_CrcCalculation()` function or `XilSkey_CrcCalculation_AesKey()` function

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_CheckAesKeyCrc(u32 CrcValue);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_CheckAesKeyCrc` function arguments.

Table 237: **XilSKey_ZynqMp_EfusePs_CheckAesKeyCrc Arguments**

Type	Name	Description
u32	CrcValue	A 32 bit CRC value of an expected AES key.

Returns

- XST_SUCCESS on successful CRC check.
- ErrorCode on failure

XilSKey_ZynqMp_EfusePs_ReadUserFuse

This function is used to read a user fuse from the eFUSE or cache.

Note: It is highly recommended to read from eFuse cache. Because reading from efuse may reduce the life of the efuse. And Cache reload is required for obtaining updated values for ReadOption 0.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadUserFuse(u32 *UseFusePtr, u8 UserFuse_Num,
u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadUserFuse` function arguments.

Table 238: **XilSKey_ZynqMp_EfusePs_ReadUserFuse Arguments**

Type	Name	Description
u32 *	UseFusePtr	Pointer to an array which holds the readback user fuse.
u8	UserFuse_Num	A variable which holds the user fuse number. Range is (User fuses: 0 to 7)
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. • 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from eFUSE cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS on successful read
- ErrorCode on failure

XilSKey_ZynqMp_EfusePs_ReadPpk0Hash

This function is used to read the PPK0 hash from an eFUSE or eFUSE cache.

Note: It is highly recommended to read from eFuse cache. Because reading from efuse may reduce the life of the efuse. And Cache reload is required for obtaining updated values for ReadOption 0.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadPpk0Hash(u32 *Ppk0Hash, u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadPpk0Hash` function arguments.

Table 239: **XilSKey_ZynqMp_EfusePs_ReadPpk0Hash Arguments**

Type	Name	Description
u32 *	Ppk0Hash	A pointer to an array which holds the readback PPK0 hash.
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. • 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from eFUSE cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS on successful read
- ErrorCode on failure

XilSKey_ZynqMp_EfusePs_ReadPpk1Hash

This function is used to read the PPK1 hash from eFUSE or cache.

Note: It is highly recommended to read from eFuse cache. Because reading from efuse may reduce the life of the efuse. And Cache reload is required for obtaining updated values for ReadOption 0.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadPpk1Hash(u32 *Ppk1Hash, u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadPpk1Hash` function arguments.

Table 240: **XilSKey_ZynqMp_EfusePs_ReadPpk1Hash Arguments**

Type	Name	Description
u32 *	Ppk1Hash	Pointer to an array which holds the readback PPK1 hash.
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. • 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from eFUSE cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS on successful read
- ErrorCode on failure

XilSKey_ZynqMp_EfusePs_ReadSpkId

This function is used to read SPKID from eFUSE or cache based on user's read option.

***Note*:** It is highly recommended to read from eFuse cache. Because reading from efuse may reduce the life of the efuse. And Cache reload is required for obtaining updated values for ReadOption 0.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadSpkId(u32 *SpkId, u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadSpkId` function arguments.

Table 241: **XilSKey_ZynqMp_EfusePs_ReadSpkId Arguments**

Type	Name	Description
u32 *	SpkId	Pointer to a 32 bit variable which holds SPK ID.
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. • 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from eFUSE cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS on successful read
- ErrorCode on failure

XilSKey_ZynqMp_EfusePs_ReadDna

This function is used to read DNA from eFUSE.

Prototype

```
void XilSKey_ZynqMp_EfusePs_ReadDna(u32 *DnaRead);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadDna` function arguments.

Table 242: **XilSKey_ZynqMp_EfusePs_ReadDna Arguments**

Type	Name	Description
u32 *	DnaRead	Pointer to an array of 3 x u32 words which holds the readback DNA.

Returns

None.

XilSKey_ZynqMp_EfusePs_ReadSecCtrlBits

This function is used to read the PS eFUSE secure control bits from cache or eFUSE based on user input provided.

***Note*:** It is highly recommended to read from eFuse cache. Because reading from efuse may reduce the life of the efuse. And Cache reload is required for obtaining updated values for ReadOption 0.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadSecCtrlBits(XilSKey_SecCtrlBits
*ReadBackSecCtrlBits, u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadSecCtrlBits` function arguments.

Table 243: **XilSKey_ZynqMp_EfusePs_ReadSecCtrlBits Arguments**

Type	Name	Description
XilSKey_SecCtrlBits *	ReadBackSecCtrlBits	Pointer to the XilSKey_SecCtrlBits which holds the read secure control bits.
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. • 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from eFUSE cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS if reads successfully
- XST_FAILURE if reading is failed

XilSKey_ZynqMp_EfusePs_CacheLoad

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_CacheLoad(void);
```

XilSKey_ZynqMp_EfusePs_Write

This function is used to program the PS eFUSE of ZynqMP, based on user inputs.

Note: After eFUSE programming is complete, the cache is automatically reloaded so all programmed eFUSE bits can be directly read from cache.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_Write(XilSKey_ZynqMpEPs *InstancePtr);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_Write` function arguments.

Table 244: **XilSKey_ZynqMp_EfusePs_Write Arguments**

Type	Name	Description
XilSKey_ZynqMpEPs *	InstancePtr	Pointer to the XilSKey_ZynqMpEPs.

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

XilSkey_ZynqMpEfuseAccess

Prototype

```
u32 XilSkey_ZynqMpEfuseAccess(const u32 AddrHigh, const u32 AddrLow);
```

XilSKey_ZynqMp_EfusePs_WritePufHelprData

This function programs the PS eFUSEs with the PUF helper data.

Note: To generate PufSyndromeData please use XilSKey_Puf_Registration API

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_WritePufHelprData(XilSKey_Puf *InstancePtr);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_WritePufHelprData` function arguments.

Table 245: **XilSKey_ZynqMp_EfusePs_WritePufHelprData Arguments**

Type	Name	Description
XilSKey_Puf *	InstancePtr	Pointer to the XilSKey_Puf instance.

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

XilSKey_ZynqMp_EfusePs_ReadPufHelprData

This function reads the PUF helper data from eFUSE.

Note: This function only reads from eFUSE non-volatile memory. There is no option to read from Cache.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadPufHelprData(u32 *Address);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadPufHelprData` function arguments.

Table 246: **XilSKey_ZynqMp_EfusePs_ReadPufHelprData Arguments**

Type	Name	Description
u32 *	Address	Pointer to data array which holds the PUF helper data read from eFUSEs.

Returns

- XST_SUCCESS if reads successfully.
- Errorcode on failure.

XilSKey_ZynqMp_EfusePs_WritePufChash

This function programs eFUSE with CHash value.

Note: To generate the CHash value, please use XilSKey_Puf_Registration function.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_WritePufChash(XilSKey_Puf *InstancePtr);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_WritePufChash` function arguments.

Table 247: **XilSKey_ZynqMp_EfusePs_WritePufChash Arguments**

Type	Name	Description
XilSKey_Puf *	InstancePtr	Pointer to the XilSKey_Puf instance.

Returns

- XST_SUCCESS if chash is programmed successfully.
- An Error code on failure

XilSKey_ZynqMp_EfusePs_ReadPufChash

This function reads eFUSE PUF CHash data from the eFUSE array or cache based on the user read option.

***Note*:** Cache reload is required for obtaining updated values for reading from cache..

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadPufChash(u32 *Address, u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadPufChash` function arguments.

Table 248: **XilSKey_ZynqMp_EfusePs_ReadPufChash Arguments**

Type	Name	Description
u32 *	Address	Pointer which holds the read back value of the chash.
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. • 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

XilSKey_ZynqMp_EfusePs_WritePufAux

This function programs eFUSE PUF auxiliary data.

Note: To generate auxiliary data, please use XilSKey_Puf_Registration function.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_WritePufAux(XilSKey_Puf *InstancePtr);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_WritePufAux` function arguments.

Table 249: **XilSKey_ZynqMp_EfusePs_WritePufAux Arguments**

Type	Name	Description
XilSKey_Puf *	InstancePtr	Pointer to the XilSKey_Puf instance.

Returns

- XST_SUCCESS if the eFUSE is programmed successfully.
- Errorcode on failure

XilSKey_ZynqMp_EfusePs_ReadPufAux

This function reads eFUSE PUF auxiliary data from eFUSE array or cache based on user read option.

Note: Cache reload is required for obtaining updated values for reading from cache.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadPufAux(u32 *Address, u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadPufAux` function arguments.

Table 250: **XilSKey_ZynqMp_EfusePs_ReadPufAux Arguments**

Type	Name	Description
u32 *	Address	Pointer which holds the read back value of PUF's auxiliary data.
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. • 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS if PUF auxiliary data is read successfully.
- Errorcode on failure

XilSKey_Write_Puf_EfusePs_SecureBits

This function programs the eFUSE PUF secure bits.

Prototype

```
u32 XilSKey_Write_Puf_EfusePs_SecureBits(XilSKey_Puf_Secure
*WriteSecureBits);
```

Parameters

The following table lists the `XilSKey_Write_Puf_EfusePs_SecureBits` function arguments.

Table 251: **XilSKey_Write_Puf_EfusePs_SecureBits Arguments**

Type	Name	Description
XilSKey_Puf_Secure *	WriteSecureBits	Pointer to the XilSKey_Puf_Secure structure

Returns

- XST_SUCCESS if eFUSE PUF secure bits are programmed successfully.
- Errorcode on failure.

XilSKey_Read_Puf_EfusePs_SecureBits

This function is used to read the PS eFUSE PUF secure bits from cache or from eFUSE array.

Prototype

```
u32 XilSKey_Read_Puf_EfusePs_SecureBits(XilSKey_Puf_Secure *SecureBitsRead,
u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_Read_Puf_EfusePs_SecureBits` function arguments.

Table 252: **XilSKey_Read_Puf_EfusePs_SecureBits Arguments**

Type	Name	Description
Commented parameter SecureBits does not exist in function XilSKey_Read_Puf_EfuseP s_SecureBits.	SecureBits	Pointer to the XilSKey_Puf_Secure structure which holds the read eFUSE secure bits from the PUF.
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. • 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS if reads successfully.
- Errorcode on failure.

XilSKey_Puf_Registration

This function performs registration of PUF which generates a new KEK and associated CHash, Auxiliary and PUF-syndrome data which are unique for each silicon.

***Note*:** With the help of generated PUF syndrome data, it will be possible to re-generate same PUF KEK.

Prototype

```
u32 XilSKey_Puf_Registration(XilSKey_Puf *InstancePtr);
```

Parameters

The following table lists the `XilSKey_Puf_Registration` function arguments.

Table 253: **XilSKey_Puf_Registration Arguments**

Type	Name	Description
XilSKey_Puf *	InstancePtr	Pointer to the XilSKey_Puf instance.

Returns

- XST_SUCCESS if registration/re-registration was successful.
- ERROR if registration was unsuccessful

XilSKey_Puf_Regeneration

This function regenerates the PUF data so that the PUF's output can be used as the key source to the AES-GCM hardware cryptographic engine.

Prototype

```
u32 XilSKey_Puf_Regeneration(XilSKey_Puf *InstancePtr);
```

Parameters

The following table lists the `XilSKey_Puf_Regeneration` function arguments.

Table 254: **XilSKey_Puf_Regeneration Arguments**

Type	Name	Description
XilSKey_Puf *	InstancePtr	is a pointer to the XilSKey_Puf instance.

Returns

- XST_SUCCESS if regeneration was successful.
- ERROR if regeneration was unsuccessful

eFUSE PL API

This chapter provides a linked summary and detailed descriptions of the eFUSE APIs of Zynq eFUSE PL and UltraScale eFUSE.

Example Usage

- The Zynq eFUSE PL and UltraScale example application should contain the xilskey_efuse_example.c and the xilskey_input.h files.
- By default, both the eFUSE PS and PL are enabled in the application. You can comment 'XSK_EFUSEPL_DRIVER' to execute only the PS.
- For UltraScale, it is mandatory to comment `XSK_EFUSEPS_DRIVER else the example will generate an error.

- For more details on the user configurable parameters, refer `Zynq User-Configurable PL eFUSE Parameters` and `UltraScale or UltraScale+ User-Configurable PL eFUSE Parameters`.
- Requires hardware setup to program PL eFUSE of Zynq or UltraScale.

Table 255: **Quick Function Reference**

Type	Name	Arguments
u32	XilSKey_EfusePl_SystemInit	`XilSKey_EPl` * InstancePtr
u32	XilSKey_EfusePl_Program	InstancePtr
u32	XilSKey_EfusePl_ReadStatus	`XilSKey_EPl` * InstancePtr u32 * StatusBits
u32	XilSKey_EfusePl_ReadKey	`XilSKey_EPl` * InstancePtr

Functions

XilSKey_EfusePl_SystemInit

Note: Updates the global variable ErrorCode with error code(if any).

Prototype

```
u32 XilSKey_EfusePl_SystemInit(XilSKey_EPl *InstancePtr);
```

Parameters

The following table lists the `XilSKey_EfusePl_SystemInit` function arguments.

Table 256: **XilSKey_EfusePl_SystemInit Arguments**

Type	Name	Description
`XilSKey_EPl` *	InstancePtr	- Input data to be written to PL eFUSE

Returns

XilSKey_EfusePl_Program

Programs PL eFUSE with input data given through InstancePtr.

Note: When this API is called: Initializes the timer, XADC/xsysmon and JTAG server subsystems. Returns an error in the following cases, if the reference clock frequency is not in the range or if the PL DAP ID is not identified, if the system is not in a position to write the requested PL eFUSE bits (because the bits are already written or not allowed to write) if the temperature and voltage are not within range.

Prototype

```
u32 XilSKey_EfusePl_Program(XilSKey_EPl *PlInstancePtr);
```

Parameters

The following table lists the `XilSKey_EfusePl_Program` function arguments.

Table 257: **XilSKey_EfusePl_Program Arguments**

Type	Name	Description
Commented parameter InstancePtr does not exist in function XilSKey_EfusePl_Program.	InstancePtr	Pointer to PL eFUSE instance which holds the input data to be written to PL eFUSE.

Returns

- XST_FAILURE - In case of failure
- XST_SUCCESS - In case of Success

XilSKey_EfusePl_ReadStatus

Reads the PL efuse status bits and gets all secure and control bits.

Prototype

```
u32 XilSKey_EfusePl_ReadStatus(XilSKey_EPl *InstancePtr, u32 *StatusBits);
```

Parameters

The following table lists the `XilSKey_EfusePl_ReadStatus` function arguments.

Table 258: **XilSKey_EfusePl_ReadStatus Arguments**

Type	Name	Description
`XilSKey_EPl` *	InstancePtr	Pointer to PL eFUSE instance.
u32 *	StatusBits	Buffer to store the status bits read.

Returns

XilSKey_EfusePl_ReadKey

Reads the PL efuse keys and stores them in the corresponding arrays in instance structure.

Note: This function initializes the timer, XADC and JTAG server subsystems, if not already done so. In Zynq - Reads AES key and User keys. In Ultrascale - Reads 32 bit and 128 bit User keys and RSA hash But AES key cannot be read directly it can be verified with CRC check (for that we need to update the instance with 32 bit CRC value, API updates whether provided CRC value is matched with actuals or not). To calculate the CRC of expected AES key one can use any of the following APIs `XilSKey_CrcCalculation()` or `XilSkey_CrcCalculation_AesKey()`

Prototype

```
u32 XilSKey_EfusePl_ReadKey(XilSKey_EPl *InstancePtr);
```

Parameters

The following table lists the `XilSKey_EfusePl_ReadKey` function arguments.

Table 259: **XilSKey_EfusePl_ReadKey Arguments**

Type	Name	Description
`XilSKey_EPl` *	InstancePtr	Pointer to PL eFUSE instance.

Returns

CRC Calculation API

This chapter provides a linked summary and detailed descriptions of the CRC calculation APIs. For UltraScale and Zynq UltraScale+ MPSoC devices, the programmed AES cannot be read back. The programmed AES key can only be verified by reading the CRC value of AES key.

Table 260: **Quick Function Reference**

Type	Name	Arguments
u32	XilSKey_CrcCalculation	u8 * Key
u32	XilSKey_CrcCalculation_AesKey	u8 * Key

Functions

XilSKey_CrcCalculation

This function Calculates CRC value based on hexadecimal string passed.

Note: If the length of the string provided is less than 64, this function appends the string with zeros. For calculation of AES key's CRC one can use u32 `XilSKey_CrcCalculation(u8 *Key)` API or reverse polynomial 0x82F63B78.

Prototype

```
u32 XilSKey_CrcCalculation(u8 *Key);
```

Parameters

The following table lists the `XilSKey_CrcCalculation` function arguments.

Table 261: **XilSKey_CrcCalculation Arguments**

Type	Name	Description
u8 *	Key	Pointer to the string contains AES key in hexadecimal of length less than or equal to 64.

Returns

- On Success returns the Crc of AES key value.
- On failure returns the error code when string length is greater than 64

XilSkey_CrcCalculation_AesKey

Calculates CRC value of the provided key.

Key should be provided in hexa buffer.

Prototype

```
u32 XilSkey_CrcCalculation_AesKey(u8 *Key);
```

Parameters

The following table lists the `XilSkey_CrcCalculation_AesKey` function arguments.

Table 262: **XilSkey_CrcCalculation_AesKey Arguments**

Type	Name	Description
u8 *	Key	Pointer to an array of 32 bytes, which holds an AES key.

Returns

Crc of provided AES key value. To calculate CRC on the AES key in string format please use XilSKey_CrcCalculation.

User-Configurable Parameters

This section provides detailed descriptions of the various user configurable parameters.

Zynq User-Configurable PS eFUSE Parameters

Define the XSK_EFUSEPS_DRIVER macro to use the PS eFUSE. After defining the macro, provide the inputs defined with XSK_EFUSEPS_DRIVER to burn the bits in PS eFUSE. If the bit is to be burned, define the macro as TRUE; otherwise define the macro as FALSE. For details, refer the following table.

Macro Name	Description
XSK_EFUSEPS_ENABLE_WRITE_PROTECT	Default = FALSE. TRUE to burn the write-protect bits in eFUSE array. Write protect has two bits. When either of the bits is burned, it is considered write-protected. So, while burning the write-protected bits, even if one bit is blown, write API returns success. As previously mentioned, POR reset is required after burning for write protection of the eFUSE bits to go into effect. It is recommended to do the POR reset after write protection. Also note that, after write-protect bits are burned, no more eFUSE writes are possible. If the write-protect macro is TRUE with other macros, write protect is burned in the last iteration, after burning all the defined values, so that for any error while burning other macros will not effect the total eFUSE array. FALSE does not modify the write-protect bits.
XSK_EFUSEPS_ENABLE_RSA_AUTH	Default = FALSE. Use TRUE to burn the RSA enable bit in the PS eFUSE array. After enabling the bit, every successive boot must be RSA-enabled apart from JTAG. Before burning (blowing) this bit, make sure that eFUSE array has the valid PPK hash. If the PPK hash burning is enabled, only after writing the hash successfully, RSA enable bit will be blown. For the RSA enable bit to take effect, POR reset is required. FALSE does not modify the RSA enable bit.

Macro Name	Description
XSK_EFUSEPS_ENABLE_ROM_128K_CRC	Default = FALSE. TRUE burns the ROM 128K CRC bit. In every successive boot, BootROM calculates 128k CRC. FALSE does not modify the ROM CRC 128K bit.
XSK_EFUSEPS_ENABLE_RSA_KEY_HASH	Default = FALSE. TRUE burns (blows) the eFUSE hash, that is given in XSK_EFUSEPS_RSA_KEY_HASH_VALUE when write API is used. TRUE reads the eFUSE hash when the read API is used and is read into structure. FALSE ignores the provided value.
XSK_EFUSEPS_RSA_KEY_HASH_VALUE	Default = The specified value is converted to a hexadecimal buffer and written into the PS eFUSE array when the write API is used. This value should be the Primary Public Key (PPK) hash provided in string format. The buffer must be 64 characters long: valid characters are 0-9, a-f, and A-F. Any other character is considered an invalid string and will not burn RSA hash. When the Xilskey_EfusePs_Write() API is used, the RSA hash is written, and the XSK_EFUSEPS_ENABLE_RSA_KEY_HASH must have a value of TRUE.
XSK_EFUSEPS_DISABLE_DFT_JTAG	Default = FALSE TRUE disables DFT JTAG permanently. FALSE will not modify the eFuse PS DFT JTAG disable bit.
XSK_EFUSEPS_DISABLE_DFT_MODE	Default = FALSE TRUE disables DFT mode permanently. FALSE will not modify the eFuse PS DFT mode disable bit.

Zynq User-Configurable PL eFUSE Parameters

Define the XSK_EFUSEPL_DRIVER macro to use the PL eFUSE. After defining the macro, provide the inputs defined with XSK_EFUSEPL_DRIVER to burn the bits in PL eFUSE bits. If the bit is to be burned, define the macro as TRUE; otherwise define the macro as FALSE. The table below lists the user-configurable PL eFUSE parameters for Zynq devices.

Macro Name	Description
XSK_EFUSEPL_FORCE_PCYCLE_RECONFIG	Default = FALSE If the value is set to TRUE, then the part has to be power-cycled to be reconfigured. FALSE does not set the eFUSE control bit.
XSK_EFUSEPL_DISABLE_KEY_WRITE	Default = FALSE TRUE disables the eFUSE write to FUSE_AES and FUSE_USER blocks. FALSE does not affect the EFUSE bit.
XSK_EFUSEPL_DISABLE_AES_KEY_READ	Default = FALSE TRUE disables the write to FUSE_AES and FUSE_USER key and disables the read of FUSE_AES. FALSE does not affect the eFUSE bit.

Macro Name	Description
XSK_EFUSEPL_DISABLE_USER_KEY_READ	Default = FALSE. TRUE disables the write to FUSE_AES and FUSE_USER key and disables the read of FUSE_USER. FALSE does not affect the eFUSE bit.
XSK_EFUSEPL_DISABLE_FUSE_CNTRL_WRITE	Default = FALSE. TRUE disables the eFUSE write to FUSE_CTRL block. FALSE does not affect the eFUSE bit.
XSK_EFUSEPL_FORCE_USE_AES_ONLY	Default = FALSE. TRUE forces the use of secure boot with eFUSE AES key only. FALSE does not affect the eFUSE bit.
XSK_EFUSEPL_DISABLE_JTAG_CHAIN	Default = FALSE. TRUE permanently disables the Zynq ARM DAP and PL TAP. FALSE does not affect the eFUSE bit.
XSK_EFUSEPL_BBRAM_KEY_DISABLE	Default = FALSE. TRUE forces the eFUSE key to be used if booting Secure Image. FALSE does not affect the eFUSE bit.

MIO Pins for Zynq PL eFUSE JTAG Operations

The table below lists the MIO pins for Zynq PL eFUSE JTAG operations. You can change the listed pins at your discretion.

Note**:** The pin numbers listed in the table below are examples. You must assign appropriate pin numbers as per your hardware design.

Pin Name	Pin Number
XSK_EFUSEPL_MIO_JTAG_TDI	(17)
XSK_EFUSEPL_MIO_JTAG_TDO	(21)
XSK_EFUSEPL_MIO_JTAG_TCK	(19)
XSK_EFUSEPL_MIO_JTAG_TMS	(20)

MUX Selection Pin for Zynq PL eFUSE JTAG Operations

The table below lists the MUX selection pin.

Pin Name	Pin Number	Description
XSK_EFUSEPL_MIO_JTAG_ MUX_SELECT	(11)	This pin toggles between the external JTAG or MIO driving JTAG operations.

MUX Parameter for Zynq PL eFUSE JTAG Operations

The table below lists the MUX parameter.

Parameter Name	Description
XSK_EFUSEPL_MIO_MUX_SEL_DEFAULT_VAL	Default = LOW. LOW writes zero on the MUX select line before PL_eFUSE writing. HIGH writes one on the MUX select line before PL_eFUSE writing.

AES and User Key Parameters

The table below lists the AES and user key parameters.

Parameter Name	Description
XSK_EFUSEPL_PROGRAM_AES_AND_USER_ LOW_KEY	Default = FALSE. TRUE burns the AES and User Low hash key, which are given in the XSK_EFUSEPL_AES_KEY and the XSK_EFUSEPL_USER_LOW_KEY respectively. FALSE ignores the provided values. You cannot write the AES Key and the User Low Key separately.
XSK_EFUSEPL_PROGRAM_USER_HIGH_KEY	Default =FALSE. TRUE burns the User High hash key, given in XSK_EFUSEPL_PROGRAM_USER_HIGH_KEY. FALSE ignores the provided values.
XSK_EFUSEPL_AES_KEY	Default = 0000000000000000000000000000000000000000000000000000000000000000 This value converted to hex buffer and written into the PL eFUSE array when write API is used. This value should be the AES Key, given in string format. It must be 64 characters long. Valid characters are 0-9, a-f, A-F. Any other character is considered an invalid string and will not burn AES Key. To write AES Key, XSK_EFUSEPL_PROGRAM_AES_AND_USER_ LOW_KEY must have a value of TRUE.
XSK_EFUSEPL_USER_LOW_KEY	Default = 00 This value is converted to a hexadecimal buffer and written into the PL eFUSE array when the write API is used. This value is the User Low Key given in string format. It must be two characters long; valid characters are 0-9,a-f, and A-F. Any other character is considered as an invalid string and will not burn the User Low Key. To write the User Low Key, XSK_EFUSEPL_PROGRAM_AES_AND_USER_ LOW_KEY must have a value of TRUE.

Parameter Name	Description
XSK_EFUSEPL_USER_HIGH_KEY	Default = 000000 The default value is converted to a hexadecimal buffer and written into the PL eFUSE array when the write API is used. This value is the User High Key given in string format. The buffer must be six characters long: valid characters are 0-9, a-f, A-F. Any other character is considered to be an invalid string and does not burn User High Key. To write the User High Key, the XSK_EFUSEPL_PROGRAM_USER_HIGH_ KEY must have a value of TRUE.

Zynq User-Configurable PL BBRAM Parameters

The table below lists the MIO pins for Zynq PL BBRAM JTAG operations.

The table below lists the MUX selection pin for Zynq BBRAM PL JTAG operations.

Note: The pin numbers listed in the table below are examples. You must assign appropriate pin numbers as per your hardware design.

Pin Name	Pin Number
XSK_BBRAM_MIO_JTAG_TDI	(17)
XSK_BBRAM_MIO_JTAG_TDO	(21)
XSK_BBRAM_MIO_JTAG_TCK	(19)
XSK_BBRAM_MIO_JTAG_TMS	(20)

Pin Name	Pin Number
XSK_BBRAM_MIO_JTAG_MUX_SELECT	(11)

MUX Parameter for Zynq BBRAM PL JTAG Operations

The table below lists the MUX parameter for Zynq BBRAM PL JTAG operations.

Parameter Name	Description
XSK_BBRAM_MIO_MUX_SEL_DEFAULT_VAL	Default = LOW. LOW writes zero on the MUX select line before PL_eFUSE writing. HIGH writes one on the MUX select line before PL_eFUSE writing.

AES and User Key Parameters

The table below lists the AES and user key parameters.

Parameter Name	Description
XSK_BBRAM_AES_KEY	Default = XX. AES key (in HEX) that must be programmed into BBRAM.
XSK_BBRAM_AES_KEY_SIZE_IN_BITS	Default = 256. Size of AES key. Must be 256 bits.

UltraScale or UltraScale+ User-Configurable BBRAM PL Parameters

Following parameters need to be configured. Based on your inputs, BBRAM is programmed with the provided AES key.

AES Keys and Related Parameters

The following table shows AES key related parameters.

Parameter Name	Description
XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_ CONFIG_ORDER_0	Default = FALSE By default, XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_ CONFIG_ORDER_0 is FALSE. BBRAM is programmed with a non-obfuscated key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_0 and DPA protection can be either in enabled/disabled state. TRUE programs the BBRAM with key provided in XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ ORDER_0 and DPA protection cannot be enabled.
XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_ CONFIG_ORDER_1	Default = FALSE By default, XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_ CONFIG_ORDER_1 is FALSE. BBRAM is programmed with a non-obfuscated key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_1 and DPA protection can be either in enabled/disabled state. TRUE programs the BBRAM with key provided in XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ ORDER_1 and DPA protection cannot be enabled.
XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_ CONFIG_ORDER_2	Default = FALSE By default, XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_ CONFIG_ORDER_2 is FALSE. BBRAM is programmed with a non-obfuscated key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_2 and DPA protection can be either in enabled/disabled state. TRUE programs the BBRAM with key provided in XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ ORDER_2 and DPA protection cannot be enabled.

Parameter Name	Description
XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_ CONFIG_ORDER_3	Default = FALSE By default, XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_ CONFIG_ORDER_3 is FALSE. BBRAM is programmed with a non-obfuscated key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_3 and DPA protection can be either in enabled/disabled state. TRUE programs the BBRAM with key provided in XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ ORDER_3 and DPA protection cannot be enabled.
XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ ORDER_0	Default = b1c276899d71fb4cdd4a0a7905ea46c2e11f9574d09c7ea23b70b67de713ccd1 The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM. ***Note***: For writing the OBFUSCATED Key, XSK_BBRAM_PGM_OBFUSCATED_KEY_ SLR_CONFIG_ORDER_0 should have TRUE value.
XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ ORDER_1	Default = b1c276899d71fb4cdd4a0a7905ea46c2e11f9574d09c7ea23b70b67de713ccd1 The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM. ***Note***: For writing the OBFUSCATED Key, XSK_BBRAM_PGM_OBFUSCATED_KEY_ SLR_CONFIG_ORDER_1 should have TRUE value.
XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ ORDER_2	Default = b1c276899d71fb4cdd4a0a7905ea46c2e11f9574d09c7ea23b70b67de713ccd1 The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM. ***Note***: For writing the OBFUSCATED Key, XSK_BBRAM_PGM_OBFUSCATED_KEY_ SLR_CONFIG_ORDER_2 should have TRUE value.

Parameter Name	Description
XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ ORDER_3	Default = b1c276899d71fb4cdd4a0a7905ea46c2e11f9574d09c7ea23b7 0b67de713ccd1 The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM. ***Note*:** For writing the OBFUSCATED Key, XSK_BBRAM_PGM_OBFUSCATED_KEY_ SLR_CONFIG_ORDER_3 should have TRUE value.
XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG_ ORDER_0	Default = FALSE TRUE will program BBRAM with AES key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_0
XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG_ ORDER_1	Default = FALSE TRUE will program BBRAM with AES key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_1
XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG_ ORDER_2	Default = FALSE TRUE will program BBRAM with AES key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_2
XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG_ ORDER_3	Default = FALSE TRUE will program BBRAM with AES key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_3
XSK_BBRAM_AES_KEY_SLR_CONFIG_ ORDER_0	Default = 0000000000000000524156a63950bcedafeadcdeabaadee3421 6615aaaabbaaa The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM,when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM. ***Note*:** For writing AES key, XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG _ORDER_0 should have TRUE value , and XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR _CONFIG_ORDER_0 should have FALSE value.

Parameter Name	Description
XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_1	Default = 0000000000000000524156a63950bcedafeadcdeabaadee34216615aaaabbaaa The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM,when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM. ***Note***: For writing AES key, XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG _ORDER_1 should have TRUE value , and XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR _CONFIG_ORDER_1 should have FALSE value
XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_2	Default = 0000000000000000524156a63950bcedafeadcdeabaadee34216615aaaabbaaa The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM. ***Note***: For writing AES key, XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG _ORDER_2 should have TRUE value , and XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR _CONFIG_ORDER_2 should have FALSE value
XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_3	Default = 0000000000000000524156a63950bcedafeadcdeabaadee34216615aaaabbaaa The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM. ***Note***: For writing AES key, XSK_BBRAM_PGM_AES_KEY_SLR _CONFIG_ORDER_3 should have TRUE value , and XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR _CONFIG_ORDER_3 should have FALSE value
XSK_BBRAM_AES_KEY_SIZE_IN_BITS	Default= 256 Size of AES key must be 256 bits.

DPA Protection for BBRAM key

The following table shows DPA protection configurable parameter

Parameter Name	Description
XSK_BBRAM_DPA_PROTECT_ENABLE	Default = FALSE By default, the DPA protection will be in disabled state. TRUE will enable DPA protection with provided DPA count and configuration in XSK_BBRAM_DPA_COUNT and XSK_BBRAM_DPA_MODE respectively. DPA protection cannot be enabled if BBRAM is been programmed with an obfuscated key.
XSK_BBRAM_DPA_COUNT	Default = 0 This input is valid only when DPA protection is enabled. Valid range of values are 1 - 255 when DPA protection is enabled else 0.
XSK_BBRAM_DPA_MODE	Default = XSK_BBRAM_INVALID_CONFIGURATIONS When DPA protection is enabled it can be XSK_BBRAM_INVALID_CONFIGURATIONS or XSK_BBRAM_ALL_CONFIGURATIONS If DPA protection is disabled this input provided over here is ignored.

GPIO Device Used for Connecting PL Master JTAG Signals

In hardware design MASTER JTAG can be connected to any one of the available GPIO devices, based on the design the following parameter should be provided with corresponding device ID of selected GPIO device.

Master JTAG Signal	Description
XSK_BBRAM_AXI_GPIO_DEVICE_ID	Default = XPAR_AXI_GPIO_0_DEVICE_ID This is for providing exact GPIO device ID, based on the design configuration this parameter can be modified to provide GPIO device ID which is used for connecting master jtag pins.

GPIO Pins Used for PL Master JTAG Signals

In Ultrascale the following GPIO pins are used for connecting MASTER_JTAG pins to access BBRAM. These can be changed depending on your hardware.The table below shows the GPIO pins used for PL MASTER JTAG signals.

Master JTAG Signal	Default PIN Number
XSK_BBRAM_AXI_GPIO_JTAG_TDO	0
XSK_BBRAM_AXI_GPIO_JTAG_TDI	0
XSK_BBRAM_AXI_GPIO_JTAG_TMS	1
XSK_BBRAM_AXI_GPIO_JTAG_TCK	2

GPIO Channels

The following table shows GPIO channel number.

Parameter	Default Channel Number	Master JTAG Signal Connected
XSK_BBRAM_GPIO_INPUT_CH	2	TDO
XSK_BBRAM_GPIO_OUTPUT_CH	1	TDI, TMS, TCK

Note: All inputs and outputs of GPIO should be configured in single channel. For example, XSK_BBRAM_GPIO_INPUT_CH = XSK_BBRAM_GPIO_OUTPUT_CH = 1 or 2. Among (TDI, TCK, TMS) Outputs of GPIO cannot be connected to different GPIO channels all the 3 signals should be in same channel. TDO can be a other channel of (TDI, TCK, TMS) or the same. DPA protection can be enabled only when programming non-obfuscated key.

UltraScale or UltraScale+ User-Configurable PL eFUSE Parameters

The table below lists the user-configurable PL eFUSE parameters for UltraScale devices.

Macro Name	Description
XSK_EFUSEPL_DISABLE_AES_KEY_READ	Default = FALSE TRUE will permanently disable the write to FUSE_AES and check CRC for AES key by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_USER_KEY_READ	Default = FALSE TRUE will permanently disable the write to 32 bit FUSE_USER and read of FUSE_USER key by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_SECURE_READ	Default = FALSE TRUE will permanently disable the write to FUSE_Secure block and reading of secure block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_FUSE_CNTRL_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_CNTRL block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_RSA_KEY_READ	Default = FALSE. TRUE will permanently disable the write to FUSE_RSA block and reading of FUSE_RSA Hash by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_KEY_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_AES block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.

Macro Name	Description
XSK_EFUSEPL_DISABLE_USER_KEY_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_USER block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_SECURE_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_SECURE block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_RSA_HASH_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_RSA authentication key by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_128BIT_USER_KEY _WRITE	Default = FALSE. TRUE will permanently disable the write to 128 bit FUSE_USER by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_ALLOW_ENCRYPTED_ONLY	Default = FALSE. TRUE will permanently allow encrypted bitstream only. FALSE will not modify this Secure bit of eFuse.
XSK_EFUSEPL_FORCE_USE_FUSE_AES_ONLY	Default = FALSE. TRUE then allows only FUSE's AES key as source of encryption FALSE then allows FPGA to configure an unencrypted bitstream or bitstream encrypted using key stored BBRAM or eFuse.
XSK_EFUSEPL_ENABLE_RSA_AUTH	Default = FALSE. TRUE will enable RSA authentication of bitstream FALSE will not modify this secure bit of eFuse.
XSK_EFUSEPL_DISABLE_JTAG_CHAIN	Default = FALSE. TRUE will disable JTAG permanently. FALSE will not modify this secure bit of eFuse.
XSK_EFUSEPL_DISABLE_TEST_ACCESS	Default = FALSE. TRUE will disables Xilinx test access. FALSE will not modify this secure bit of eFuse.
XSK_EFUSEPL_DISABLE_AES_DECRYPTOR	Default = FALSE. TRUE will disables decoder completely. FALSE will not modify this secure bit of eFuse.
XSK_EFUSEPL_ENABLE_OBFUSCATION_ EFUSEAES	Default = FALSE. TRUE will enable obfuscation feature for eFUSE AES key.

GPIO Device Used for Connecting PL Master JTAG Signals

In hardware design MASTER JTAG can be connected to any one of the available GPIO devices, based on the design the following parameter should be provided with corresponding device ID of selected GPIO device.

Master JTAG Signal	Description
XSK_EFUSEPL_AXI_GPIO_DEVICE_ID	Default = XPAR_AXI_GPIO_0_DEVICE_ID This is for providing exact GPIO device ID, based on the design configuration this parameter can be modified to provide GPIO device ID which is used for connecting master jtag pins.

GPIO Pins Used for PL Master JTAG and HWM Signals

In Ultrascale the following GPIO pins are used for connecting MASTER_JTAG pins to access eFUSE. These can be changed depending on your hardware.The table below shows the GPIO pins used for PL MASTER JTAG signals.

Master JTAG Signal	Default PIN Number
XSK_EFUSEPL_AXI_GPIO_JTAG_TDO	0
XSK_EFUSEPL_AXI_GPIO_HWM_READY	0
XSK_EFUSEPL_AXI_GPIO_HWM_END	1
XSK_EFUSEPL_AXI_GPIO_JTAG_TDI	2
XSK_EFUSEPL_AXI_GPIO_JTAG_TMS	1
XSK_EFUSEPL_AXI_GPIO_JTAG_TCK	2
XSK_EFUSEPL_AXI_GPIO_HWM_START	3

GPIO Channels

The following table shows GPIO channel number.

Parameter	Default Channel Number	Master JTAG Signal Connected
XSK_EFUSEPL_GPIO_INPUT_ CH	2	TDO
XSK_EFUSEPL_GPIO_OUTPUT_ CH	1	TDI, TMS, TCK

***Note*:** All inputs and outputs of GPIO should be configured in single channel. For example, XSK_EFUSEPL_GPIO_INPUT_CH = XSK_EFUSEPL_GPIO_OUTPUT_CH = 1 or 2. Among (TDI, TCK, TMS) Outputs of GPIO cannot be connected to different GPIO channels all the 3 signals should be in same channel. TDO can be a other channel of (TDI, TCK, TMS) or the same.

SLR Selection to Program eFUSE on MONO/SSIT Devices

The following table shows parameters for programming different SLRs.

Parameter Name	Description
XSK_EFUSEPL_PGM_SLR_CONFIG_ORDER_0	Default = FALSE TRUE will enable programming SLR config order 0's eFUSE. FALSE will disable programming.

Parameter Name	Description
XSK_EFUSEPL_PGM_SLR_CONFIG_ORDER_1	Default = FALSE TRUE will enable programming SLR config order 1's eFUSE. FALSE will disable programming.
XSK_EFUSEPL_PGM_SLR_CONFIG_ORDER_2	Default = FALSE TRUE will enable programming SLR config order 2's eFUSE. FALSE will disable programming.
XSK_EFUSEPL_PGM_SLR_CONFIG_ORDER_3	Default = FALSE TRUE will enable programming SLR config order 3's eFUSE. FALSE will disable programming.

eFUSE PL Read Parameters

The following table shows parameters related to read USER 32/128bit keys and RSA hash.

By enabling any of the below parameters, by default will read corresponding hash/key associated with all the available SLRs. For example, if XSK_EFUSEPL_READ_USER_KEY is TRUE, USER key for all the available SLRs will be read.

***Note*:** For only reading keys it is not required to enable XSK_EFUSEPL_PGM_SLR1, XSK_EFUSEPL_PGM_SLR2, XSK_EFUSEPL_PGM_SLR3, XSK_EFUSEPL_PGM_SLR4 macros, they can be in FALSE state.

Parameter Name	Description
XSK_EFUSEPL_READ_USER_KEY	Default = FALSE TRUE will read 32 bit FUSE_USER from eFUSE of all available SLRs and each time updates in `XilSKey_EP1` instance parameter UserKeyReadback, which will be displayed on UART by example before reading next SLR. FALSE 32-bit FUSE_USER key read will not be performed.
XSK_EFUSEPL_READ_RSA_KEY_HASH	Default = FALSE TRUE will read FUSE_USER from eFUSE of all available SLRs and each time updates in `XilSKey_EP1` instance parameter RSAHashReadback, which will be displayed on UART by example before reading next SLR. FALSE FUSE_RSA_HASH read will not be performed.
XSK_EFUSEPL_READ_USER_KEY128_BIT	Default = FALSE TRUE will read 128 bit USER key eFUSE of all available SLRs and each time updates in `XilSKey_EP1` instance parameter User128BitReadBack, which will be displayed on UART by example before reading next SLR. FALSE 128 bit USER key read will not be performed.

AES Keys and Related Parameters

***Note*:** For programming AES key for MONO/SSIT device, the corresponding SLR should be selected and AES key programming should be enabled.

USER Keys (32-bit) and Related Parameters

Note: For programming USER key for MONO/SSIT device, the corresponding SLR should be selected and USER key programming should be enabled.

RSA Hash and Related Parameters

Note: For programming RSA hash for MONO/SSIT device, the corresponding SLR should be selected and RSA hash programming should be enabled.

USER Keys (128-bit) and Related Parameters

Note: For programming USER key 128 bit for MONO/SSIT device, the corresponding SLR and programming for USER key 128 bit should be enabled.

AES key CRC verification

You cannot read the AES key. You can verify only by providing the CRC of the expected AES key. The following lists the parameters that may help you in verifying the AES key:

Parameter Name	Description
XSK_EFUSEPL_CHECK_AES_KEY_ CRC	Default = FALSE TRUE will perform CRC check of FUSE_AES with provided CRC value in macro XSK_EFUSEPL_CRC_OF_EXPECTED_ AES_KEY. And result of CRC check will be updated in `XilSKey_EPl` instance parameter AESKeyMatched with either TRUE or FALSE. FALSE CRC check of FUSE_AES will not be performed.
XSK_EFUSEPL_CRC_OF_EXPECTED_AES_KEY_ CONFIG_ORDER_0	Default = XSK_EFUSEPL_AES_CRC_OF_ALL_ ZEROS CRC value of FUSE_AES with all Zeros. Expected FUSE_AES key's CRC value of SLR config order 0 has to be updated in place of XSK_EFUSEPL_AES_CRC_OF_ALL_ ZEROS. For Checking CRC of FUSE_AES XSK_EFUSEPL_CHECK_AES_KEY_ULTRA macro should be TRUE otherwise CRC check will not be performed. For calculation of AES key's CRC one can use u32 XilSKey_CrcCalculation(u8_Key) API. For UltraScale, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ ZEROS is 0x621C42AA(XSK_EFUSEPL_CRC_ FOR_AES_ZEROS). For UltraScale+, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS is 0x3117503A(XSK_EFUSEPL_CRC_FOR_ AES_ZEROS_ULTRA_PLUS)

Parameter Name	Description
XSK_EFUSEPL_CRC_OF_EXPECTED_AES_KEY_ CONFIG_ORDER_1	Default = XSK_EFUSEPL_AES_CRC_OF_ALL_ ZEROS CRC value of FUSE_AES with all Zeros. Expected FUSE_AES key's CRC value of SLR config order 1 has to be updated in place of XSK_EFUSEPL_AES_CRC_OF_ALL_ ZEROS. For Checking CRC of FUSE_AES XSK_EFUSEPL_CHECK_AES_KEY_ULTRA macro should be TRUE otherwise CRC check will not be performed. For calculation of AES key's CRC one can use u32 XilSKey_CrcCalculation(u8_Key) API. For UltraScale, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ ZEROS is 0x621C42AA(XSK_EFUSEPL_CRC_ FOR_AES_ZEROS). For UltraScale+, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS is 0x3117503A(XSK_EFUSEPL_CRC_FOR_ AES_ZEROS_ULTRA _PLUS)
XSK_EFUSEPL_CRC_OF_EXPECTED_AES_KEY_ CONFIG_ORDER_2	Default = XSK_EFUSEPL_AES_CRC_OF_ALL_ ZEROS CRC value of FUSE_AES with all Zeros. Expected FUSE_AES key's CRC value of SLR config order 2 has to be updated in place of XSK_EFUSEPL_AES_CRC_OF_ALL_ ZEROS. For Checking CRC of FUSE_AES XSK_EFUSEPL_CHECK_AES_KEY_ULTRA macro should be TRUE otherwise CRC check will not be performed. For calculation of AES key's CRC one can use u32 XilSKey_CrcCalculation(u8_Key) API. For UltraScale, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ ZEROS is 0x621C42AA(XSK_EFUSEPL_CRC_ FOR_AES_ZEROS). For UltraScale+, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS is 0x3117503A(XSK_EFUSEPL_CRC_FOR_ AES_ZEROS_ULTRA _PLUS)
XSK_EFUSEPL_CRC_OF_EXPECTED_AES_KEY_ CONFIG_ORDER_3	Default = XSK_EFUSEPL_AES_CRC_OF_ALL_ ZEROS CRC value of FUSE_AES with all Zeros. Expected FUSE_AES key's CRC value of SLR config order 3 has to be updated in place of XSK_EFUSEPL_AES_CRC_OF_ALL_ ZEROS. For Checking CRC of FUSE_AES XSK_EFUSEPL_CHECK_AES_KEY_ULTRA macro should be TRUE otherwise CRC check will not be performed. For calculation of AES key's CRC one can use u32 XilSKey_CrcCalculation(u8_Key) API. For UltraScale, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ ZEROS is 0x621C42AA(XSK_EFUSEPL_CRC_ FOR_AES_ZEROS). For UltraScale+, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS is 0x3117503A(XSK_EFUSEPL_CRC_FOR_ AES_ZEROS_ULTRA _PLUS)

Zynq UltraScale+ MPSoC User-Configurable PS eFUSE Parameters

The table below lists the user-configurable PS eFUSE parameters for Zynq UltraScale+ MPSoC devices.

Macro Name	Description
XSK_EFUSEPS_AES_RD_LOCK	Default = FALSE TRUE will permanently disable the CRC check of FUSE_AES. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_AES_WR_LOCK	Default = FALSE TRUE will permanently disable the writing to FUSE_AES block. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_ENC_ONLY	Default = FALSE TRUE will permanently enable encrypted booting only using the Fuse key. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_BBRAM_DISABLE	Default = FALSE TRUE will permanently disable the BBRAM key. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_ERR_DISABLE	Default = FALSE TRUE will permanently disables the error messages in JTAG status register. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_JTAG_DISABLE	Default = FALSE TRUE will permanently disable JTAG controller. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_DFT_DISABLE	Default = FALSE TRUE will permanently disable DFT boot mode. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PROG_GATE_DISABLE	Default = FALSE TRUE will permanently disable PROG_GATE feature in PPD. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_SECURE_LOCK	Default = FALSE TRUE will permanently disable reboot into JTAG mode when doing a secure lockdown. FALSE will not modify thi s control bit of eFuse.
XSK_EFUSEPS_RSA_ENABLE	Default = FALSE TRUE will permanently enable RSA authentication during boot. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PPK0_WR_LOCK	Default = FALSE TRUE will permanently disable writing to PPK0 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PPK0_INVLD	Default = FALSE TRUE will permanently revoke PPK0. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PPK1_WR_LOCK	Default = FALSE TRUE will permanently disable writing PPK1 efuses. FALSE will not modify this control bit of eFuse.

Macro Name	Description
XSK_EFUSEPS_PPK1_INVLD	Default = FALSE TRUE will permanently revoke PPK1. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_0	Default = FALSE TRUE will permanently disable writing to USER_0 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_1	Default = FALSE TRUE will permanently disable writing to USER_1 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_2	Default = FALSE TRUE will permanently disable writing to USER_2 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_3	Default = FALSE TRUE will permanently disable writing to USER_3 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_4	Default = FALSE TRUE will permanently disable writing to USER_4 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_5	Default = FALSE TRUE will permanently disable writing to USER_5 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_6	Default = FALSE TRUE will permanently disable writing to USER_6 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_7	Default = FALSE TRUE will permanently disable writing to USER_7 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_LBIST_EN	Default = FALSE TRUE will permanently enables logic BIST to be run during boot. FALSE will not modify this control bit of eFUSE.
XSK_EFUSEPS_LPD_SC_EN	Default = FALSE TRUE will permanently enables zeroization of registers in Low Power Domain(LPD) during boot. FALSE will not modify this control bit of eFUSE.
XSK_EFUSEPS_FPD_SC_EN	Default = FALSE TRUE will permanently enables zeroization of registers in Full Power Domain(FPD) during boot. FALSE will not modify this control bit of eFUSE.
XSK_EFUSEPS_PBR_BOOT_ERR	Default = FALSE TRUE will permanently enables the boot halt when there is any PMU error. FALSE will not modify this control bit of eFUSE.

AES Keys and Related Parameters

The following table shows AES key related parameters.

Parameter Name	Description
XSK_EFUSEPS_WRITE_AES_KEY	Default = FALSE TRUE will burn the AES key provided in XSK_EFUSEPS_AES_KEY. FALSE will ignore the key provide XSK_EFUSEPS_AES_KEY.
XSK_EFUSEPS_AES_KEY	Default = 00000000000000000000000000000000000000000000000000000000000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn AES Key. ***Note***: For writing the AES Key, XSK_EFUSEPS_WRITE_AES_KEY should have TRUE value.
XSK_EFUSEPS_CHECK_AES_KEY_CRC	Default value is FALSE. TRUE will check the CRC provided in XSK_EFUSEPS_AES_KEY. CRC verification is done after programming AES key to verify the key is programmed properly or not, if not library error outs the same. So While programming AES key it is not necessary to verify the AES key again. ***Note***: Please make sure if intention is to check only CRC of the provided key and not programming AES key then do not modify XSK_EFUSEPS_WRITE_AES_ KEY (TRUE will Program key).

User Keys and Related Parameters

Single bit programming is allowed for all the user eFUSEs. When you request to revert already programmed bit, the library will return an error. Also, if the user eFUSEs is non-zero, the library will not throw an error for valid requests. The following table shows the user keys and related parameters.

Parameter Name	Description
XSK_EFUSEPS_WRITE_USER0_FUSE	Default = FALSE TRUE will burn User0 Fuse provided in XSK_EFUSEPS_USER0_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER0_FUSES
XSK_EFUSEPS_WRITE_USER1_FUSE	Default = FALSE TRUE will burn User1 Fuse provided in XSK_EFUSEPS_USER1_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER1_FUSES
XSK_EFUSEPS_WRITE_USER2_FUSE	Default = FALSE TRUE will burn User2 Fuse provided in XSK_EFUSEPS_USER2_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER2_FUSES

Parameter Name	Description
XSK_EFUSEPS_WRITE_USER3_FUSE	Default = FALSE TRUE will burn User3 Fuse provided in XSK_EFUSEPS_USER3_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER3_FUSES
XSK_EFUSEPS_WRITE_USER4_FUSE	Default = FALSE TRUE will burn User4 Fuse provided in XSK_EFUSEPS_USER4_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER4_FUSES
XSK_EFUSEPS_WRITE_USER5_FUSE	Default = FALSE TRUE will burn User5 Fuse provided in XSK_EFUSEPS_USER5_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER5_FUSES
XSK_EFUSEPS_WRITE_USER6_FUSE	Default = FALSE TRUE will burn User6 Fuse provided in XSK_EFUSEPS_USER6_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER6_FUSES
XSK_EFUSEPS_WRITE_USER7_FUSE	Default = FALSE TRUE will burn User7 Fuse provided in XSK_EFUSEPS_USER7_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER7_FUSES
XSK_EFUSEPS_USER0_FUSES	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID. ***Note***: For writing the User0 Fuse, XSK_EFUSEPS_WRITE_USER0_FUSE should have TRUE value
XSK_EFUSEPS_USER1_FUSES	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID. ***Note***: For writing the User1 Fuse, XSK_EFUSEPS_WRITE_USER1_FUSE should have TRUE value
XSK_EFUSEPS_USER2_FUSES	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID. ***Note***: For writing the User2 Fuse, XSK_EFUSEPS_WRITE_USER2_FUSE should have TRUE value

Parameter Name	Description
XSK_EFUSEPS_USER3_FUSES	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID. ***Note*:** For writing the User3 Fuse, XSK_EFUSEPS_WRITE_USER3_FUSE should have TRUE value
XSK_EFUSEPS_USER4_FUSES	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID. ***Note*:** For writing the User4 Fuse, XSK_EFUSEPS_WRITE_USER4_FUSE should have TRUE value
XSK_EFUSEPS_USER5_FUSES	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID. ***Note*:** For writing the User5 Fuse, XSK_EFUSEPS_WRITE_USER5_FUSE should have TRUE value
XSK_EFUSEPS_USER6_FUSES	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID. ***Note*:** For writing the User6 Fuse, XSK_EFUSEPS_WRITE_USER6_FUSE should have TRUE value
XSK_EFUSEPS_USER7_FUSES	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID. ***Note*:** For writing the User7 Fuse, XSK_EFUSEPS_WRITE_USER7_FUSE should have TRUE value

PPK0 Keys and Related Parameters

The following table shows the PPK0 keys and related parameters.

Parameter Name	Description
XSK_EFUSEPS_WRITE_PPK0_SHA3_HASH	Default = FALSE TRUE will burn PPK0 sha3 hash provided in XSK_EFUSEPS_PPK0_SHA3_HASH. FALSE will ignore the hash provided in XSK_EFUSEPS_PPK0_SHA3_HASH.
XSK_EFUSEPS_PPK0_IS_SHA3	Default = TRUE TRUE XSK_EFUSEPS_PPK0_SHA3_HASH should be of string length 96 it specifies that PPK0 is used to program SHA3 hash. FALSE XSK_EFUSEPS_PPK0_SHA3_HASH should be of string length 64 it specifies that PPK0 is used to program SHA2 hash.
XSK_EFUSEPS_PPK0_HASH	Default = 0000000000000000000000000000000000000000000000000000 00000000000000000000000000000000000000000000 The value mentioned in this will be converted to hex buffer and into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 96 or 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn PPK0 hash. Note that,for writing the PPK0 hash, XSK_EFUSEPS_WRITE_PPK0_SHA3_HASH should have TRUE value. While writing SHA2 hash, length should be 64 characters long XSK_EFUSEPS_PPK0_IS_SHA3 macro has to be made FALSE. While writing SHA3 hash, length should be 96 characters long and XSK_EFUSEPS_PPK0_IS_SHA3 macro should be made TRUE

PPK1 Keys and Related Parameters

The following table shows the PPK1 keys and related parameters.

Parameter Name	Description
XSK_EFUSEPS_WRITE_PPK1_SHA3_HASH	Default = FALSE TRUE will burn PPK1 sha3 hash provided in XSK_EFUSEPS_PPK1_SHA3_HASH. FALSE will ignore the hash provided in XSK_EFUSEPS_PPK1_SHA3_HASH.
XSK_EFUSEPS_PPK1_IS_SHA3	Default = TRUE TRUE XSK_EFUSEPS_PPK1_SHA3_HASH should be of string length 96 it specifies that PPK1 is used to program SHA3 hash. FALSE XSK_EFUSEPS_PPK1_SHA3_HASH should be of string length 64 it specifies that PPK1 is used to program SHA2 hash.

Parameter Name	Description
XSK_EFUSEPS_PPK1_HASH	Default = 0000000000000000000000000000000000000000000000000000000000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale + MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 64 or 96 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn PPK1 hash. Note that,for writing the PPK11 hash, XSK_EFUSEPS_WRITE_PPK1_SHA3_HASH should have TRUE value. By default, PPK1 hash will be provided with 64 character length to program PPK1 hash with sha2 hash so XSK_EFUSEPS_PPK1_IS_SHA3 also will be in FALSE state. But to program PPK1 hash with SHA3 hash make XSK_EFUSEPS_PPK1_IS_SHA3 to TRUE and provide sha3 hash of length 96 characters XSK_EFUSEPS_PPK1_HASH so that one can program sha3 hash.

SPK ID and Related Parameters

The following table shows the SPK ID and related parameters.

Parameter Name	Description
XSK_EFUSEPS_WRITE_SPKID	Default = FALSE TRUE will burn SPKID provided in XSK_EFUSEPS_SPK_ID. FALSE will ignore the hash provided in XSK_EFUSEPS_SPK_ID.
XSK_EFUSEPS_SPK_ID	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID. ***Note***: For writing the SPK ID, XSK_EFUSEPS_WRITE_SPKID should have TRUE value.

Note: PPK hash should be unmodified hash generated by bootgen. Single bit programming is allowed for User FUSEs (0 to 7), if you specify a value that tries to set a bit that was previously programmed to 1 back to 0, you will get an error. you have to provide already programmed bits also along with new requests.

Zynq UltraScale+ MPSoC User-Configurable PS BBRAM Parameters

The table below lists the AES and user key parameters.

Parameter Name	Description
XSK_ZYNQMP_BBRAMPS_AES_KEY	Default = 0000000000000000000000000000000000000000000000000000000000000 AES key (in HEX) that must be programmed into BBRAM.

Parameter Name	Description
XSK_ZYNQMP_BBRAMPS_AES_KEY_LEN_IN_ BYTES	Default = 32. Length of AES key in bytes.
XSK_ZYNQMP_BBRAMPS_AES_KEY_LEN_IN_ BITS	Default = 256. Length of AES key in bits.
XSK_ZYNQMP_BBRAMPS_AES_KEY_STR_LEN	Default = 64. String length of the AES key.

Zynq UltraScale+ MPSoC User-Configurable PS PUF Parameters

The table below lists the user-configurable PS PUF parameters for Zynq UltraScale+ MPSoC devices.

Macro Name	Description
XSK_PUF_INFO_ON_UART	Default = FALSE TRUE will display syndrome data on UART com port FALSE will display any data on UART com port.
XSK_PUF_PROGRAM_EFUSE	Default = FALSE TRUE will program the generated syndrome data, CHash and Auxilary values, Black key. FALSE will not program data into eFUSE.
XSK_PUF_IF_CONTRACT_MANUFACTURER	Default = FALSE This should be enabled when application is hand over to contract manufacturer. TRUE will allow only authenticated application. FALSE authentication is not mandatory.
XSK_PUF_REG_MODE	Default = XSK_PUF_MODE4K PUF registration is performed in 4K mode. For only understanding it is provided in this file, but user is not supposed to modify this.
XSK_PUF_READ_SECUREBITS	Default = FALSE TRUE will read status of the puf secure bits from eFUSE and will be displayed on UART. FALSE will not read secure bits.
XSK_PUF_PROGRAM_SECUREBITS	Default = FALSE TRUE will program PUF secure bits based on the user input provided at XSK_PUF_SYN_INVALID, XSK_PUF_SYN_WRLK and XSK_PUF_REGISTER_DISABLE. FALSE will not program any PUF secure bits.
XSK_PUF_SYN_INVALID	Default = FALSE TRUE will permanently invalidate the already programmed syndrome data. FALSE will not modify anything

Macro Name	Description
XSK_PUF_SYN_WRLK	Default = FALSE TRUE will permanently disable programming syndrome data into eFUSE. FALSE will not modify anything.
XSK_PUF_REGISTER_DISABLE	Default = FALSE TRUE permanently does not allow PUF syndrome data registration. FALSE will not modify anything.
XSK_PUF_RESERVED	Default = FALSE TRUE programs this reserved eFUSE bit. FALSE will not modify anything.
XSK_PUF_AES_KEY	Default = 0000000000000000000000000000000000000000000000000000000000000000 The value mentioned in this will be converted to hex buffer and encrypts this with PUF helper data and generates a black key and written into the Zynq UltraScale+ MPSoC PS eFUSE array when XSK_PUF_PROGRAM_EFUSE macro is TRUE. This value should be given in string format. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn AES Key. Note Provided here should be red key and application calculates the black key and programs into eFUSE if XSK_PUF_PROGRAM_EFUSE macro is TRUE. To avoid programming eFUSE results can be displayed on UART com port by making XSK_PUF_INFO_ON_UART to TRUE.
XSK_PUF_BLACK_KEY_IV	Default = 000000000000000000000000 The value mentioned here will be converted to hex buffer. This is Initialization vector(IV) which is used to generated black key with provided AES key and generated PUF key. This value should be given in string format. It should be 24 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string.

Error Codes

The application error code is 32 bits long. For example, if the error code for PS is 0x8A05:

- 0x8A indicates that a write error has occurred while writing RSA Authentication bit.
- 0x05 indicates that write error is due to the write temperature out of range.

Applications have the following options on how to show error status. Both of these methods of conveying the status are implemented by default. However, UART is required to be present and initialized for status to be displayed through UART.

- Send the error code through UART pins

- Write the error code in the reboot status register

PL eFUSE Error Codes

Enumerations

Enumeration XSKEfusePl_ErrorCodes

Table 263: **Enumeration XSKEfusePl_ErrorCodes Values**

Value	Description
XSK_EFUSEPL_ERROR_NONE	0 No error.
XSK_EFUSEPL_ERROR_ROW_NOT_ZERO	0x10 Row is not zero.
XSK_EFUSEPL_ERROR_READ_ROW_OUT_OF_RANGE	0x11 Read Row is out of range.
XSK_EFUSEPL_ERROR_READ_MARGIN_OUT_OF_RANGE	0x12 Read Margin is out of range.
XSK_EFUSEPL_ERROR_READ_BUFFER_NULL	0x13 No buffer for read.
XSK_EFUSEPL_ERROR_READ_BIT_VALUE_NOT_SET	0x14 Read bit not set.
XSK_EFUSEPL_ERROR_READ_BIT_OUT_OF_RANGE	0x15 Read bit is out of range.
XSK_EFUSEPL_ERROR_READ_TMEPERATURE_OUT_OF_RANGE	0x16 Temperature obtained from XADC is out of range to read.
XSK_EFUSEPL_ERROR_READ_VCCAUX_VOLTAGE_OUT_OF_RANGE	0x17 VCCAUX obtained from XADC is out of range to read.
XSK_EFUSEPL_ERROR_READ_VCCINT_VOLTAGE_OUT_OF_RANGE	0x18 VCCINT obtained from XADC is out of range to read.
XSK_EFUSEPL_ERROR_WRITE_ROW_OUT_OF_RANGE	0x19 To write row is out of range.
XSK_EFUSEPL_ERROR_WRITE_BIT_OUT_OF_RANGE	0x1A To read bit is out of range.
XSK_EFUSEPL_ERROR_WRITE_TMEPERATURE_OUT_OF_RANGE	0x1B To eFUSE write Temperature obtained from XADC is outof range.
XSK_EFUSEPL_ERROR_WRITE_VCCAUX_VOLTAGE_OUT_OF_RANGE	0x1C To write eFUSE VCCAUX obtained from XADC is out of range.
XSK_EFUSEPL_ERROR_WRITE_VCCINT_VOLTAGE_OUT_OF_RANGE	0x1D To write into eFUSE VCCINT obtained from XADC is out of range.
XSK_EFUSEPL_ERROR_FUSE_CNTRL_WRITE_DISABLED	0x1E Fuse control write is disabled.

Table 263: **Enumeration XSKEfusePl_ErrorCodes Values** *(cont'd)*

Value	Description
XSK_EFUSEPL_ERROR_CNTRL_WRITE_BUFFER_NULL	0x1F Buffer pointer that is supposed to contain control data is null.
XSK_EFUSEPL_ERROR_NOT_VALID_KEY_LENGTH	0x20 Key length invalid.
XSK_EFUSEPL_ERROR_ZERO_KEY_LENGTH	0x21 Key length zero.
XSK_EFUSEPL_ERROR_NOT_VALID_KEY_CHAR	0x22 Invalid key characters.
XSK_EFUSEPL_ERROR_NULL_KEY	0x23 Null key.
XSK_EFUSEPL_ERROR_FUSE_SEC_WRITE_DISABLED	0x24 Secure bits write is disabled.
XSK_EFUSEPL_ERROR_FUSE_SEC_READ_DISABLED	0x25 Secure bits reading is disabled.
XSK_EFUSEPL_ERROR_SEC_WRITE_BUFFER_NULL	0x26 Buffer to write into secure block is NULL.
XSK_EFUSEPL_ERROR_READ_PAGE_OUT_OF_RANGE	0x27 Page is out of range.
XSK_EFUSEPL_ERROR_FUSE_ROW_RANGE	0x28 Row is out of range.
XSK_EFUSEPL_ERROR_IN_PROGRAMMING_ROW	0x29 Error programming fuse row.
XSK_EFUSEPL_ERROR_PRGRMG_ROWS_NOT_EMPTY	0x2A Error when tried to program non Zero rows of eFUSE.
XSK_EFUSEPL_ERROR_HWM_TIMEOUT	0x80 Error when hardware module is exceeded the time for programming eFUSE.
XSK_EFUSEPL_ERROR_USER_FUSE_REVERT	0x90 Error occurs when user requests to revert already programmed user eFUSE bit.
XSK_EFUSEPL_ERROR_KEY_VALIDATION	0xF000 Invalid key.
XSK_EFUSEPL_ERROR_PL_STRUCT_NULL	0x1000 Null PL structure.
XSK_EFUSEPL_ERROR_JTAG_SERVER_INIT	0x1100 JTAG server initialization error.
XSK_EFUSEPL_ERROR_READING_FUSE_CNTRL	0x1200 Error reading fuse control.
XSK_EFUSEPL_ERROR_DATA_PROGRAMMING_NOT_ALLOWED	0x1300 Data programming not allowed.
XSK_EFUSEPL_ERROR_FUSE_CTRL_WRITE_NOT_ALLOWED	0x1400 Fuse control write is disabled.

Table 263: **Enumeration XSKEfusePl_ErrorCodes Values** *(cont'd)*

Value	Description
XSK_EFUSEPL_ERROR_READING_FUSE_AES_ROW	0x1500 Error reading fuse AES row.
XSK_EFUSEPL_ERROR_AES_ROW_NOT_EMPTY	0x1600 AES row is not empty.
XSK_EFUSEPL_ERROR_PROGRAMMING_FUSE_AES_ROW	0x1700 Error programming fuse AES row.
XSK_EFUSEPL_ERROR_READING_FUSE_USER_DATA_ROW	0x1800 Error reading fuse user row.
XSK_EFUSEPL_ERROR_USER_DATA_ROW_NOT_EMPTY	0x1900 User row is not empty.
XSK_EFUSEPL_ERROR_PROGRAMMING_FUSE_DATA_ROW	0x1A00 Error programming fuse user row.
XSK_EFUSEPL_ERROR_PROGRAMMING_FUSE_CNTRL_ROW	0x1B00 Error programming fuse control row.
XSK_EFUSEPL_ERROR_XADC	0x1C00 XADC error.
XSK_EFUSEPL_ERROR_INVALID_REF_CLK	0x3000 Invalid reference clock.
XSK_EFUSEPL_ERROR_FUSE_SEC_WRITE_NOT_ALLOWED	0x1D00 Error in programming secure block.
XSK_EFUSEPL_ERROR_READING_FUSE_STATUS	0x1E00 Error in reading FUSE status.
XSK_EFUSEPL_ERROR_FUSE_BUSY	0x1F00 Fuse busy.
XSK_EFUSEPL_ERROR_READING_FUSE_RSA_ROW	0x2000 Error in reading FUSE RSA block.
XSK_EFUSEPL_ERROR_TIMER_INTIALISE_ULTRA	0x2200 Error in initiating Timer.
XSK_EFUSEPL_ERROR_READING_FUSE_SEC	0x2300 Error in reading FUSE secure bits.
XSK_EFUSEPL_ERROR_PRGRMG_FUSE_SEC_ROW	0x2500 Error in programming Secure bits of efuse.
XSK_EFUSEPL_ERROR_PRGRMG_USER_KEY	0x4000 Error in programming 32 bit user key.
XSK_EFUSEPL_ERROR_PRGRMG_128BIT_USER_KEY	0x5000 Error in programming 128 bit User key.
XSK_EFUSEPL_ERROR_PRGRMG_RSA_HASH	0x8000 Error in programming RSA hash.

PS eFUSE Error Codes

Enumerations

Enumeration XSKEfusePs_ErrorCodes

Table 264: **Enumeration XSKEfusePs_ErrorCodes Values**

Value	Description
XSK_EFUSEPS_ERROR_NONE	0 No error.
XSK_EFUSEPS_ERROR_ADDRESS_XIL_RESTRICTED	0x01 Address is restricted.
XSK_EFUSEPS_ERROR_READ_TMEPERATURE_OUT_OF_RANGE	0x02 Temperature obtained from XADC is out of range.
XSK_EFUSEPS_ERROR_READ_VCCPAUX_VOLTAGE_OUT_OF_RANGE	0x03 VCCAUX obtained from XADC is out of range.
XSK_EFUSEPS_ERROR_READ_VCCPINT_VOLTAGE_OUT_OF_RANGE	0x04 VCCINT obtained from XADC is out of range.
XSK_EFUSEPS_ERROR_WRITE_TEMPERATURE_OUT_OF_RANGE	0x05 Temperature obtained from XADC is out of range.
XSK_EFUSEPS_ERROR_WRITE_VCCPAUX_VOLTAGE_OUT_OF_RANGE	0x06 VCCAUX obtained from XADC is out of range.
XSK_EFUSEPS_ERROR_WRITE_VCCPINT_VOLTAGE_OUT_OF_RANGE	0x07 VCCINT obtained from XADC is out of range.
XSK_EFUSEPS_ERROR_VERIFICATION	0x08 Verification error.
XSK_EFUSEPS_ERROR_RSA_HASH_ALREADY_PROGRAMMED	0x09 RSA hash was already programmed.
XSK_EFUSEPS_ERROR_CONTROLLER_MODE	0x0A Controller mode error
XSK_EFUSEPS_ERROR_REF_CLOCK	0x0B Reference clock not between 20 to 60MHz
XSK_EFUSEPS_ERROR_READ_MODE	0x0C Not supported read mode
XSK_EFUSEPS_ERROR_XADC_CONFIG	0x0D XADC configuration error.
XSK_EFUSEPS_ERROR_XADC_INITIALIZE	0x0E XADC initialization error.
XSK_EFUSEPS_ERROR_XADC_SELF_TEST	0x0F XADC self-test failed.
XSK_EFUSEPS_ERROR_PARAMETER_NULL	0x10 Passed parameter null.

Table 264: **Enumeration XSKEfusePs_ErrorCodes Values** *(cont'd)*

Value	Description
XSK_EFUSEPS_ERROR_STRING_INVALID	0x20 Passed string is invalid.
XSK_EFUSEPS_ERROR_AES_ALREADY_PROGRAMMED	0x12 AES key is already programmed.
XSK_EFUSEPS_ERROR_SPKID_ALREADY_PROGRAMMED	0x13 SPK ID is already programmed.
XSK_EFUSEPS_ERROR_PPK0_HASH_ALREADY_PROGRAMMED	0x14 PPK0 hash is already programmed.
XSK_EFUSEPS_ERROR_PPK1_HASH_ALREADY_PROGRAMMED	0x15 PPK1 hash is already programmed.
XSK_EFUSEPS_ERROR_IN_TBIT_PATTERN	0x16 Error in TBITS pattern .
XSK_EFUSEPS_ERROR_PROGRAMMING	0x00A0 Error in programming eFUSE.
XSK_EFUSEPS_ERROR_PGM_NOT_DONE	0x00A1 Program not done
XSK_EFUSEPS_ERROR_READ	0x00B0 Error in reading.
XSK_EFUSEPS_ERROR_BYTES_REQUEST	0x00C0 Error in requested byte count.
XSK_EFUSEPS_ERROR_RESRVD_BITS_PRGRMG	0x00D0 Error in programming reserved bits.
XSK_EFUSEPS_ERROR_ADDR_ACCESS	0x00E0 Error in accessing requested address.
XSK_EFUSEPS_ERROR_READ_NOT_DONE	0x00F0 Read not done
XSK_EFUSEPS_ERROR_PS_STRUCT_NULL	0x8100 PS structure pointer is null.
XSK_EFUSEPS_ERROR_XADC_INIT	0x8200 XADC initialization error.
XSK_EFUSEPS_ERROR_CONTROLLER_LOCK	0x8300 PS eFUSE controller is locked.
XSK_EFUSEPS_ERROR_EFUSE_WRITE_PROTECTED	0x8400 PS eFUSE is write protected.
XSK_EFUSEPS_ERROR_CONTROLLER_CONFIG	0x8500 Controller configuration error.
XSK_EFUSEPS_ERROR_PS_PARAMETER_WRONG	0x8600 PS eFUSE parameter is not TRUE/FALSE.
XSK_EFUSEPS_ERROR_WRITE_128K_CRC_BIT	0x9100 Error in enabling 128K CRC.

Table 264: **Enumeration XSKEfusePs_ErrorCodes Values** *(cont'd)*

Value	Description
XSK_EFUSEPS_ERROR_WRITE_NONSECURE_INITB_BIT	0x9200 Error in programming NON secure bit.
XSK_EFUSEPS_ERROR_WRITE_UART_STATUS_BIT	0x9300 Error in writing UART status bit.
XSK_EFUSEPS_ERROR_WRITE_RSA_HASH	0x9400 Error in writing RSA key.
XSK_EFUSEPS_ERROR_WRITE_RSA_AUTH_BIT	0x9500 Error in enabling RSA authentication bit.
XSK_EFUSEPS_ERROR_WRITE_WRITE_PROTECT_BIT	0x9600 Error in writing write-protect bit.
XSK_EFUSEPS_ERROR_READ_HASH_BEFORE_PROGRAMMING	0x9700 Check RSA key before trying to program.
XSK_EFUSEPS_ERROR_WRTIE_DFT_JTAG_DIS_BIT	0x9800 Error in programming DFT JTAG disable bit.
XSK_EFUSEPS_ERROR_WRTIE_DFT_MODE_DIS_BIT	0x9900 Error in programming DFT MODE disable bit.
XSK_EFUSEPS_ERROR_WRTIE_AES_CRC_LK_BIT	0x9A00 Error in enabling AES's CRC check lock.
XSK_EFUSEPS_ERROR_WRTIE_AES_WR_LK_BIT	0x9B00 Error in programming AES write lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USE_AESONLY_EN_BIT	0x9C00 Error in programming use AES only bit.
XSK_EFUSEPS_ERROR_WRTIE_BBRAM_DIS_BIT	0x9D00 Error in programming BBRAM disable bit.
XSK_EFUSEPS_ERROR_WRTIE_PMU_ERR_DIS_BIT	0x9E00 Error in programming PMU error disable bit.
XSK_EFUSEPS_ERROR_WRTIE_JTAG_DIS_BIT	0x9F00 Error in programming JTAG disable bit.
XSK_EFUSEPS_ERROR_READ_RSA_HASH	0xA100 Error in reading RSA key.
XSK_EFUSEPS_ERROR_WRONG_TBIT_PATTERN	0xA200 Error in programming TBIT pattern.
XSK_EFUSEPS_ERROR_WRITE_AES_KEY	0xA300 Error in programming AES key.
XSK_EFUSEPS_ERROR_WRITE_SPK_ID	0xA400 Error in programming SPK ID.
XSK_EFUSEPS_ERROR_WRITE_USER_KEY	0xA500 Error in programming USER key.
XSK_EFUSEPS_ERROR_WRITE_PPK0_HASH	0xA600 Error in programming PPK0 hash.

Table 264: **Enumeration XSKEfusePs_ErrorCodes Values** *(cont'd)*

Value	Description
XSK_EFUSEPS_ERROR_WRITE_PPK1_HASH	0xA700 Error in programming PPK1 hash.
XSK_EFUSEPS_ERROR_WRITE_USER0_FUSE	0xC000 Error in programming USER 0 Fuses.
XSK_EFUSEPS_ERROR_WRITE_USER1_FUSE	0xC100 Error in programming USER 1 Fuses.
XSK_EFUSEPS_ERROR_WRITE_USER2_FUSE	0xC200 Error in programming USER 2 Fuses.
XSK_EFUSEPS_ERROR_WRITE_USER3_FUSE	0xC300 Error in programming USER 3 Fuses.
XSK_EFUSEPS_ERROR_WRITE_USER4_FUSE	0xC400 Error in programming USER 4 Fuses.
XSK_EFUSEPS_ERROR_WRITE_USER5_FUSE	0xC500 Error in programming USER 5 Fuses.
XSK_EFUSEPS_ERROR_WRITE_USER6_FUSE	0xC600 Error in programming USER 6 Fuses.
XSK_EFUSEPS_ERROR_WRITE_USER7_FUSE	0xC700 Error in programming USER 7 Fuses.
XSK_EFUSEPS_ERROR_WRTIE_USER0_LK_BIT	0xC800 Error in programming USER 0 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USER1_LK_BIT	0xC900 Error in programming USER 1 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USER2_LK_BIT	0xCA00 Error in programming USER 2 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USER3_LK_BIT	0xCB00 Error in programming USER 3 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USER4_LK_BIT	0xCC00 Error in programming USER 4 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USER5_LK_BIT	0xCD00 Error in programming USER 5 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USER6_LK_BIT	0xCE00 Error in programming USER 6 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USER7_LK_BIT	0xCF00 Error in programming USER 7 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_PROG_GATE0_DIS_BIT	0xD000 Error in programming PROG_GATE0 disabling bit.
XSK_EFUSEPS_ERROR_WRTIE_PROG_GATE1_DIS_BIT	0xD100 Error in programming PROG_GATE1 disabling bit.
XSK_EFUSEPS_ERROR_WRTIE_PROG_GATE2_DIS_BIT	0xD200 Error in programming PROG_GATE2 disabling bit.

Table 264: **Enumeration XSKEfusePs_ErrorCodes Values** *(cont'd)*

Value	Description
XSK_EFUSEPS_ERROR_WRTIE_SEC_LOCK_BIT	0xD300 Error in programming SEC_LOCK bit.
XSK_EFUSEPS_ERROR_WRTIE_PPK0_WR_LK_BIT	0xD400 Error in programming PPK0 write lock bit.
XSK_EFUSEPS_ERROR_WRTIE_PPK0_RVK_BIT	0xD500 Error in programming PPK0 revoke bit.
XSK_EFUSEPS_ERROR_WRTIE_PPK1_WR_LK_BIT	0xD600 Error in programming PPK1 write lock bit.
XSK_EFUSEPS_ERROR_WRTIE_PPK1_RVK_BIT	0xD700 Error in programming PPK0 revoke bit.
XSK_EFUSEPS_ERROR_WRITE_PUF_SYN_INVLD	0xD800 Error while programming the PUF syndrome invalidate bit.
XSK_EFUSEPS_ERROR_WRITE_PUF_SYN_WRLK	0xD900 Error while programming Syndrome write lock bit.
XSK_EFUSEPS_ERROR_WRITE_PUF_SYN_REG_DIS	0xDA00 Error while programming PUF syndrome register disable bit.
XSK_EFUSEPS_ERROR_WRITE_PUF_RESERVED_BIT	0xDB00 Error while programming PUF reserved bit.
XSK_EFUSEPS_ERROR_WRITE_LBIST_EN_BIT	0xDC00 Error while programming LBIST enable bit.
XSK_EFUSEPS_ERROR_WRITE_LPD_SC_EN_BIT	0xDD00 Error while programming LPD SC enable bit.
XSK_EFUSEPS_ERROR_WRITE_FPD_SC_EN_BIT	0xDE00 Error while programming FPD SC enable bit.
XSK_EFUSEPS_ERROR_WRITE_PBR_BOOT_ERR_BIT	0xDF00 Error while programming PBR boot error bit.
XSK_EFUSEPS_ERROR_PUF_INVALID_REG_MODE	0xE000 Error when PUF registration is requested with invalid registration mode.
XSK_EFUSEPS_ERROR_PUF_REG_WO_AUTH	0xE100 Error when write not allowed without authentication enabled.
XSK_EFUSEPS_ERROR_PUF_REG_DISABLED	0xE200 Error when trying to do PUF registration and when PUF registration is disabled.
XSK_EFUSEPS_ERROR_PUF_INVALID_REQUEST	0xE300 Error when an invalid mode is requested.
XSK_EFUSEPS_ERROR_PUF_DATA_ALREADY_PROGRAMMED	0xE400 Error when PUF is already programmed in eFUSE.
XSK_EFUSEPS_ERROR_PUF_DATA_OVERFLOW	0xE500 Error when an over flow occurs.
XSK_EFUSEPS_ERROR_SPKID_BIT_CANT_REVERT	0xE600 Already programmed SPKID bit cannot be reverted

Table 264: **Enumeration XSKEfusePs_ErrorCodes Values** *(cont'd)*

Value	Description
XSK_EFUSEPS_ERROR_PUF_DATA_UNDERFLOW	0xE700 Error when an under flow occurs.
XSK_EFUSEPS_ERROR_PUF_TIMEOUT	0xE800 Error when an PUF generation timedout.
XSK_EFUSEPS_ERROR_PUF_ACCESS	0xE900 Error when an PUF Access violation.
XSK_EFUSEPS_ERROR_PUF_CHASH_ALREADY_PROGRAMMED	
XSK_EFUSEPS_ERROR_PUF_AUX_ALREADY_PROGRAMMED	0xEA00 Error When PUF Chash already programmed in eFuse.
XSK_EFUSEPS_ERROR_CMPLTD_EFUSE_PRGRM_WITH_ERR	0xEB00 Error When PUF AUX already programmed in eFuse. 0x10000 eFUSE programming is completed with temp and vol read errors.
XSK_EFUSEPS_ERROR_CACHE_LOAD	0x20000U Error in re-loading CACHE.
XSK_EFUSEPS_RD_FROM_EFUSE_NOT_ALLOWED	0x30000U Read from eFuse is not allowed.
XSK_EFUSEPS_ERROR_FUSE_PROTECTED	0x00080000 Requested eFUSE is write protected.
XSK_EFUSEPS_ERROR_USER_BIT_CANT_REVERT	0x00800000 Already programmed user FUSE bit cannot be reverted.
XSK_EFUSEPS_ERROR_BEFORE_PROGRAMMING	0x08000000U Error occurred before programming.

Zynq UltraScale+ MPSoC BBRAM PS Error Codes

Enumerations

Enumeration XskZynqMp_Ps_Bbram_ErrorCodes

Table 265: **Enumeration XskZynqMp_Ps_Bbram_ErrorCodes Values**

Value	Description
XSK_ZYNQMP_BBRAMPS_ERROR_NONE	0 No error.
XSK_ZYNQMP_BBRAMPS_ERROR_IN_PRGRMG_ENABLE	0x010 If this error is occurred programming is not possible.
XSK_ZYNQMP_BBRAMPS_ERROR_IN_ZEROISE	0x20 zeroize bbram is failed.

Table 265: **Enumeration XskZynqMp_Ps_Bbram_ErrorCodes Values** *(cont'd)*

Value	Description
XSK_ZYNQMP_BBRAMPS_ERROR_IN_CRC_CHECK	0xB000 If this error is occurred programming is done but CRC check is failed.
XSK_ZYNQMP_BBRAMPS_ERROR_IN_PRGRMG	0xC000 programming of key is failed.
XSK_ZYNQMP_BBRAMPS_ERROR_IN_WRITE_CRC	0xE800 error write CRC value.

Status Codes

For Zynq and UltraScale, the status in the xilskey_efuse_example.c file is conveyed through a UART or reboot status register in the following format: 0xYYYYZZZZ, where:

- YYYY represents the PS eFUSE Status.
- ZZZZ represents the PL eFUSE Status.

The table below lists the status codes.

Status Code Values	Description
0x0000ZZZZ	Represents PS eFUSE is successful and PL eFUSE process returned with error.
0xYYYY0000	Represents PL eFUSE is successful and PS eFUSE process returned with error.
0xFFFF0000	Represents PS eFUSE is not initiated and PL eFUSE is successful.
0x0000FFFF	Represents PL eFUSE is not initiated and PS eFUSE is successful.
0xFFFFZZZZ	Represents PS eFUSE is not initiated and PL eFUSE is process returned with error.
0xYYYYFFFF	Represents PL eFUSE is not initiated and PS eFUSE is process returned with error.

For Zynq UltraScale+ MPSoC, the status in the xilskey_bbramps_zynqmp_example.c, xilskey_puf_registration.c and xilskey_efuseps_zynqmp_example.c files is conveyed as 32 bit error code. Where Zero represents that no error has occurred and if the value is other than Zero, a 32 bit error code is returned.

Procedures

This section provides detailed descriptions of the various procedures.

Zynq eFUSE Writing Procedure Running from DDR as an Application

This sequence is same as the existing flow described below.

1. Provide the required inputs in `xilskey_input.h`, then compile the platform project.
2. Take the latest FSBL (ELF), stitch the `<output>.elf` generated to it (using the bootgen utility), and generate a bootable image.
3. Write the generated binary image into the flash device (for example: QSPI, NAND).
4. To burn the eFUSE key bits, execute the image.

Zynq eFUSE Driver Compilation Procedure for OCM

The procedure is as follows:

1. Open the linker script (`lscript.ld`) in the platform project.
2. Map all the sections to point to ps7_ram_0_S_AXI_BASEADDR instead of ps7_ddr_0_S_AXI_BASEADDR. For example, Click the Memory Region tab for the .text section and select ps7_ram_0_S_AXI_BASEADDR from the drop-down list.
3. Copy the ps7_init.c and ps7_init.h files from the hw_platform folder into the example folder.
4. In `xilskey_efuse_example.c`, un-comment the code that calls the `ps7_init()` routine.
5. Compile the project.

   The `<Project name>.elf` file is generated and is executed out of OCM.

When executed, this example displays the success/failure of the eFUSE application in a display message via UART (if UART is present and initialized) or the reboot status register.

UltraScale eFUSE Access Procedure

The procedure is as follows:

1. After providing the required inputs in `xilskey_input.h`, compile the project.
2. Generate a memory mapped interface file using TCL command write_mem_info
3. Update memory has to be done using the tcl command updatemem.
4. Program the board using `$Final.bit bitstream`.
5. Output can be seen in UART terminal.

UltraScale BBRAM Access Procedure

The procedure is as follows:

1. After providing the required inputs in the `xilskey_bbram_ultrascale_input.h`` file, compile the project.
2. Generate a memory mapped interface file using TCL command
3. Update memory has to be done using the tcl command updatemem:
4. Program the board using `$Final.bit bitstream`.
5. Output can be seen in UART terminal.

Data Structure Index

The following is a list of data structures:

- XilSKey_EPl

XilSKey_EPl

XEfusePl is the PL eFUSE driver instance.

Using this structure, user can define the eFUSE bits to be blown.

Declaration

```
typedef struct
{
  u32 ForcePowerCycle,
  u32 KeyWrite,
  u32 AESKeyRead,
  u32 UserKeyRead,
  u32 CtrlWrite,
  u32 RSARead,
  u32 UserKeyWrite,
  u32 SecureWrite,
  u32 RSAWrite,
  u32 User128BitWrite,
  u32 SecureRead,
  u32 AESKeyExclusive,
  u32 JtagDisable,
  u32 UseAESOnly,
  u32 EncryptOnly,
  u32 IntTestAccessDisable,
  u32 DecoderDisable,
  u32 RSAEnable,
  u32 FuseObfusEn,
  u32 ProgAESandUserLowKey,
  u32 ProgUserHighKey,
  u32 ProgAESKeyUltra,
  u32 ProgUserKeyUltra,
  u32 ProgRSAKeyUltra,
  u32 ProgUser128BitUltra,
  u32 CheckAESKeyUltra,
  u32 ReadUserKeyUltra,
  u32 ReadRSAKeyUltra,
  u32 ReadUser128BitUltra,
  u8 AESKey[XSK_EFUSEPL_AES_KEY_SIZE_IN_BYTES],
  u8 UserKey[XSK_EFUSEPL_USER_KEY_SIZE_IN_BYTES],
  u8 RSAKeyHash[XSK_EFUSEPL_RSA_KEY_HASH_SIZE_IN_BYTES],
  u8 User128Bit[XSK_EFUSEPL_128BIT_USERKEY_SIZE_IN_BYTES],
  u32 JtagMioTDI,
  u32 JtagMioTDO,
  u32 JtagMioTCK,
  u32 JtagMioTMS,
  u32 JtagMioMuxSel,
  u32 JtagMuxSelLineDefVal,
  u32 JtagGpioID,
  u32 HwmGpioStart,
  u32 HwmGpioReady,
  u32 HwmGpioEnd,
  u32 JtagGpioTDI,
  u32 JtagGpioTDO,
  u32 JtagGpioTMS,
  u32 JtagGpioTCK,
  u32 GpioInputCh,
  u32 GpioOutPutCh,
  u8 AESKeyReadback[XSK_EFUSEPL_AES_KEY_SIZE_IN_BYTES],
  u8 UserKeyReadback[XSK_EFUSEPL_USER_KEY_SIZE_IN_BYTES],
  u32 CrcOfAESKey,
  u8 AESKeyMatched,
  u8 RSAHashReadback[XSK_EFUSEPL_RSA_KEY_HASH_SIZE_IN_BYTES],
  u8 User128BitReadBack[XSK_EFUSEPL_128BIT_USERKEY_SIZE_IN_BYTES],
  u32 SystemInitDone,
  XSKEfusePl_Fpga FpgaFlag,
  u32 CrcToVerify,
  u32 NumSlr,
  u32 MasterSlr,
  u32 SlrConfigOrderIndex
} XilSKey_EPl;
```

Table 266: **Structure XilSKey_EPl member description**

Member	Description
ForcePowerCycle	Following are the FUSE CNTRL bits[1:5, 8-10]. If XTRUE then part has to be power cycled to be able to be reconfigured only for zynq
KeyWrite	If XTRUE will disable eFUSE write to FUSE_AES and FUSE_USER blocks valid only for zynq but in ultrascale If XTRUE will disable eFUSE write to FUSE_AESKEY block in Ultrascale.
AESKeyRead	If XTRUE will disable eFUSE read to FUSE_AES block and also disables eFUSE write to FUSE_AES and FUSE_USER blocks in Zynq Pl.but in Ultrascale if XTRUE will disable eFUSE read to FUSE_KEY block and also disables eFUSE write to FUSE_KEY blocks.
UserKeyRead	If XTRUE will disable eFUSE read to FUSE_USER block and also disables eFUSE write to FUSE_AES and FUSE_USER blocks in zynq but in ultrascale if XTRUE will disable eFUSE read to FUSE_USER block and also disables eFUSE write to FUSE_USER blocks.
CtrlWrite	If XTRUE will disable eFUSE write to FUSE_CNTRL block in both Zynq and Ultrascale.
RSARead	If XTRUE will disable eFuse read to FUSE_RSA block and also disables eFuse write to FUSE_RSA block in Ultrascale.
UserKeyWrite	

Table 266: **Structure XilSKey_EPl member description** *(cont'd)*

Member	Description
SecureWrite	
RSAWrite	
User128BitWrite	If TRUE will disable eFUSE write to 128BIT FUSE_USER block in Ultrascale.
SecureRead	IF XTRUE will disable eFuse read to FUSE_SEC block and also disables eFuse write to FUSE_SEC block in Ultrascale.
AESKeyExclusive	If XTRUE will force eFUSE key to be used if booting Secure Image In Zynq.
JtagDisable	If XTRUE then permanently sets the Zynq ARM DAP controller in bypass mode in both zynq and ultrascale.
UseAESOnly	If XTRUE will force to use Secure boot with eFUSE key only for both Zynq and Ultrascale.
EncryptOnly	If XTRUE will only allow encrypted bitstreams only.
IntTestAccessDisable	If XTRUE then sets the disable's Xilinx internal test access in Ultrascale.
DecoderDisable	If XTRUE then permanently disables the decryptor in Ultrascale.
RSAEnable	Enable RSA authentication in ultrascale.
FuseObfusEn	
ProgAESandUserLowKey	Following is the define to select if the user wants to select AES key and User Low Key for Zynq.
ProgUserHighKey	Following is the define to select if the user wants to select User Low Key for Zynq.
ProgAESKeyUltra	Following is the define to select if the user wants to select User key for Ultrascale.
ProgUserKeyUltra	Following is the define to select if the user wants to select User key for Ultrascale.
ProgRSAKeyUltra	Following is the define to select if the user wants to select RSA key for Ultrascale.
ProgUser128BitUltra	Following is the define to select if the user wants to program 128 bit User key for Ultrascale.
CheckAESKeyUltra	Following is the define to select if the user wants to read AES key for Ultrascale.
ReadUserKeyUltra	Following is the define to select if the user wants to read User key for Ultrascale.
ReadRSAKeyUltra	Following is the define to select if the user wants to read RSA key for Ultrascale.
ReadUser128BitUltra	Following is the define to select if the user wants to read 128 bit User key for Ultrascale.
AESKey	This is the REF_CLK value in Hz. This is for the aes_key value
UserKey	This is for the user_key value.
RSAKeyHash	This is for the rsa_key value for Ultrascale.
User128Bit	This is for the User 128 bit key value for Ultrascale.
JtagMioTDI	TDI MIO Pin Number for ZYNQ.
JtagMioTDO	TDO MIO Pin Number for ZYNQ.
JtagMioTCK	TCK MIO Pin Number for ZYNQ.
JtagMioTMS	TMS MIO Pin Number for ZYNQ.

Table 266: **Structure XilSKey_EPl member description** *(cont'd)*

Member	Description
JtagMioMuxSel	MUX Selection MIO Pin Number for ZYNQ.
JtagMuxSelLineDefVal	Value on the MUX Selection line for ZYNQ.
JtagGpioID	GPIO device ID.
HwmGpioStart	
HwmGpioReady	
HwmGpioEnd	
JtagGpioTDI	
JtagGpioTDO	
JtagGpioTMS	
JtagGpioTCK	
GpioInputCh	
GpioOutPutCh	
AESKeyReadback	AES key read only for Zynq.
UserKeyReadback	User key read in Ultrascale and Zynq.
CrcOfAESKey	Expected AES key's CRC for Ultrascale here we can't read AES key directly.
AESKeyMatched	
RSAHashReadback	
User128BitReadBack	User 128 bit key read back for Ultrascale.
SystemInitDone	Internal variable to check if timer, XADC and JTAG are initialized.
FpgaFlag	
CrcToVerify	
NumSlr	
MasterSlr	
SlrConfigOrderIndex	

Chapter 9

XilPM Library v3.1

XilPM Zynq UltraScale+ MPSoC APIs

Xilinx Power Management (XilPM) provides Embedded Energy Management Interface (EEMI) APIs for power management on Zynq UltraScale+ MPSoC. For more details about EEMI, see the Embedded Energy Management Interface (EEMI) API User Guide (UG1200).

Table 267: **Quick Function Reference**

Type	Name	Arguments
XStatus	XPm_InitXilpm	XIpiPsu * IpiInst
enum XPmBootStatus	XPm_GetBootStatus	void
void	XPm_SuspendFinalize	void
XStatus	pm_ipi_send	struct XPm_Master *const master u32 payload
XStatus	pm_ipi_buff_read32	struct XPm_Master *const master u32 * value1 u32 * value2 u32 * value3
XStatus	XPm_SelfSuspend	const enum XPmNodeId nid const u32 latency const u8 state const u64 address
XStatus	XPm_SetConfiguration	const u32 address
XStatus	XPm_InitFinalize	void

Table 267: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
XStatus	XPm_RequestSuspend	const enum `XPmNodeId` target const enum `XPmRequestAck` ack const u32 latency const u8 state
XStatus	XPm_RequestWakeUp	const enum `XPmNodeId` target const bool setAddress const u64 address const enum `XPmRequestAck` ack
XStatus	XPm_ForcePowerDown	const enum `XPmNodeId` target const enum `XPmRequestAck` ack
XStatus	XPm_AbortSuspend	const enum `XPmAbortReason` reason
XStatus	XPm_SetWakeUpSource	const enum `XPmNodeId` target const enum `XPmNodeId` wkup_node const u8 enable
XStatus	XPm_SystemShutdown	restart
XStatus	XPm_RequestNode	const enum `XPmNodeId` node const u32 capabilities const u32 qos const enum `XPmRequestAck` ack
XStatus	XPm_SetRequirement	const enum `XPmNodeId` nid const u32 capabilities const u32 qos const enum `XPmRequestAck` ack
XStatus	XPm_ReleaseNode	const enum `XPmNodeId` node
XStatus	XPm_SetMaxLatency	const enum `XPmNodeId` node const u32 latency
void	XPm_InitSuspendCb	const enum `XPmSuspendReason` reason const u32 latency const u32 state const u32 timeout

Table 267: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	XPm_AcknowledgeCb	const enum XPmNodeId node const XStatus status const u32 oppoint
void	XPm_NotifyCb	const enum XPmNodeId node const enum XPmNotifyEvent event const u32 oppoint
XStatus	XPm_GetApiVersion	u32 * version
XStatus	XPm_GetNodeStatus	const enum XPmNodeId node XPm_NodeStatus *const nodestatus
XStatus	XPm_GetOpCharacteristic	const enum XPmNodeId node const enum XPmOpCharType type u32 *const result
XStatus	XPm_ResetAssert	const enum XPmReset reset assert
XStatus	XPm_ResetGetStatus	const enum XPmReset reset u32 * status
XStatus	XPm_RegisterNotifier	XPm_Notifier *const notifier
XStatus	XPm_UnregisterNotifier	XPm_Notifier *const notifier
XStatus	XPm_MmioWrite	const u32 address const u32 mask const u32 value
XStatus	XPm_MmioRead	const u32 address u32 *const value
XStatus	XPm_ClockEnable	const enum XPmClock clock
XStatus	XPm_ClockDisable	const enum XPmClock clock
XStatus	XPm_ClockGetStatus	const enum XPmClock clock u32 *const status

Table 267: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
XStatus	XPm_ClockSetOneDivider	const enum XPmClock clock const u32 divider const u32 divId
XStatus	XPm_ClockSetDivider	const enum XPmClock clock const u32 divider
XStatus	XPm_ClockGetOneDivider	const enum XPmClock clock u32 *const divider
XStatus	XPm_ClockGetDivider	const enum XPmClock clock u32 *const divider
XStatus	XPm_ClockSetParent	const enum XPmClock clock const enum XPmClock parent
XStatus	XPm_ClockGetParent	const enum XPmClock clock enum XPmClock *const parent
XStatus	XPm_ClockSetRate	const enum XPmClock clock const u32 rate
XStatus	XPm_ClockGetRate	const enum XPmClock clock u32 *const rate
XStatus	XPm_PllSetParameter	const enum XPmNodeId node const enum XPmPllParam parameter const u32 value
XStatus	XPm_PllGetParameter	const enum XPmNodeId node const enum XPmPllParam parameter u32 *const value
XStatus	XPm_PllSetMode	const enum XPmNodeId node const enum XPmPllMode mode
XStatus	XPm_PllGetMode	const enum XPmNodeId node enum XPmPllMode *const mode
XStatus	XPm_PinCtrlAction	const u32 pin

Table 267: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
XStatus	XPm_PinCtrlRequest	const u32 pin
XStatus	XPm_PinCtrlRelease	const u32 pin
XStatus	XPm_PinCtrlSetFunction	const u32 pin const enum XPmPinFn fn
XStatus	XPm_PinCtrlGetFunction	const u32 pin enum XPmPinFn *const fn
XStatus	XPm_PinCtrlSetParameter	const u32 pin const enum XPmPinParam param const u32 value
XStatus	XPm_PinCtrlGetParameter	const u32 pin const enum XPmPinParam param u32 *const value

Functions

XPm_InitXilpm

Initialize xilpm library.

Note: None

Prototype

```
XStatus XPm_InitXilpm(XIpiPsu *IpiInst);
```

Parameters

The following table lists the `XPm_InitXilpm` function arguments.

Table 268: **XPm_InitXilpm Arguments**

Type	Name	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_GetBootStatus

This Function returns information about the boot reason. If the boot is not a system startup but a resume, power down request bitfield for this processor will be cleared.

Note: None

Prototype

```
enum
               XPmBootStatus
             XPm_GetBootStatus(void);
```

Returns

Returns processor boot status

- PM_RESUME : If the boot reason is because of system resume.
- PM_INITIAL_BOOT : If this boot is the initial system startup.

XPm_SuspendFinalize

This Function waits for PMU to finish all previous API requests sent by the PU and performs client specific actions to finish suspend procedure (e.g. execution of wfi instruction on A53 and R5 processors).

Note: This function should not return if the suspend procedure is successful.

Prototype

```
void XPm_SuspendFinalize(void);
```

Returns

pm_ipi_send

Sends IPI request to the PMU.

Note: None

Prototype

```
XStatus pm_ipi_send(struct XPm_Master *const master, u32
payload[PAYLOAD_ARG_CNT]);
```

Parameters

The following table lists the `pm_ipi_send` function arguments.

Table 269: **pm_ipi_send Arguments**

Type	Name	Description
struct `XPm_Master` *const	master	Pointer to the master who is initiating request
u32	payload	API id and call arguments to be written in IPI buffer

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

pm_ipi_buff_read32

Reads IPI response after PMU has handled interrupt.

***Note*:** None

Prototype

```
XStatus pm_ipi_buff_read32(struct XPm_Master *const master, u32 *value1,
u32 *value2, u32 *value3);
```

Parameters

The following table lists the `pm_ipi_buff_read32` function arguments.

Table 270: **pm_ipi_buff_read32 Arguments**

Type	Name	Description
struct `XPm_Master` *const	master	Pointer to the master who is waiting and reading response
u32 *	value1	Used to return value from 2nd IPI buffer element (optional)
u32 *	value2	Used to return value from 3rd IPI buffer element (optional)
u32 *	value3	Used to return value from 4th IPI buffer element (optional)

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SelfSuspend

This function is used by a CPU to declare that it is about to suspend itself. After the PMU processes this call it will wait for the requesting CPU to complete the suspend procedure and become ready to be put into a sleep state.

***Note*:** This is a blocking call, it will return only once PMU has responded

Prototype

```
XStatus XPm_SelfSuspend(const enum XPmNodeId nid, const u32 latency, const
u8 state, const u64 address);
```

Parameters

The following table lists the `XPm_SelfSuspend` function arguments.

Table 271: **XPm_SelfSuspend Arguments**

Type	Name	Description
const enum `XPmNodeId`	nid	Node ID of the CPU node to be suspended.
const u32	latency	Maximum wake-up latency requirement in us(microsecs)
const u8	state	Instead of specifying a maximum latency, a CPU can also explicitly request a certain power state.
const u64	address	Address from which to resume when woken up.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SetConfiguration

This function is called to configure the power management framework. The call triggers power management controller to load the configuration object and configure itself according to the content of the object.

***Note*:** The provided address must be in 32-bit address space which is accessible by the PMU.

Prototype

```
XStatus XPm_SetConfiguration(const u32 address);
```

Parameters

The following table lists the `XPm_SetConfiguration` function arguments.

Table 272: **XPm_SetConfiguration Arguments**

Type	Name	Description
const u32	address	Start address of the configuration object

Returns

XST_SUCCESS if successful, otherwise an error code

XPm_InitFinalize

This function is called to notify the power management controller about the completed power management initialization.

Note: It is assumed that all used nodes are requested when this call is made. The power management controller may power down the nodes which are not requested after this call is processed.

Prototype

```
XStatus XPm_InitFinalize(void);
```

Returns

XST_SUCCESS if successful, otherwise an error code

XPm_RequestSuspend

This function is used by a PU to request suspend of another PU. This call triggers the power management controller to notify the PU identified by 'nodeID' that a suspend has been requested. This will allow said PU to gracefully suspend itself by calling XPm_SelfSuspend for each of its CPU nodes, or else call XPm_AbortSuspend with its PU node as argument and specify the reason.

Note: If 'ack' is set to PM_ACK_NON_BLOCKING, the requesting PU will be notified upon completion of suspend or if an error occurred, such as an abort. REQUEST_ACK_BLOCKING is not supported for this command.

Prototype

```
XStatus XPm_RequestSuspend(const enum XPmNodeId target, const enum
XPmRequestAck ack, const u32 latency, const u8 state);
```

Parameters

The following table lists the `XPm_RequestSuspend` function arguments.

Table 273: **XPm_RequestSuspend Arguments**

Type	Name	Description
const enum `XPmNodeId`	target	Node ID of the PU node to be suspended
const enum `XPmRequestAck`	ack	Requested acknowledge type
const u32	latency	Maximum wake-up latency requirement in us(micro sec)
const u8	state	Instead of specifying a maximum latency, a PU can also explicitly request a certain power state.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_RequestWakeUp

This function can be used to request power up of a CPU node within the same PU, or to power up another PU.

Note: If acknowledge is requested, the calling PU will be notified by the power management controller once the wake-up is completed.

Prototype

```
XStatus XPm_RequestWakeUp(const enum XPmNodeId target, const bool
setAddress, const u64 address, const enum XPmRequestAck ack);
```

Parameters

The following table lists the `XPm_RequestWakeUp` function arguments.

Table 274: **XPm_RequestWakeUp Arguments**

Type	Name	Description
const enum `XPmNodeId`	target	Node ID of the CPU or PU to be powered/woken up.
const bool	setAddress	Specifies whether the start address argument is being passed. • 0 : do not set start address • 1 : set start address
const u64	address	Address from which to resume when woken up. Will only be used if set_address is 1.
const enum `XPmRequestAck`	ack	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ForcePowerDown

One PU can request a forced poweroff of another PU or its power island or power domain. This can be used for killing an unresponsive PU, in which case all resources of that PU will be automatically released.

Note: Force power down may not be requested by a PU for itself.

Prototype

```
XStatus XPm_ForcePowerDown(const enum XPmNodeId target, const enum
XPmRequestAck ack);
```

Parameters

The following table lists the `XPm_ForcePowerDown` function arguments.

Table 275: **XPm_ForcePowerDown Arguments**

Type	Name	Description
const enum `XPmNodeId`	target	Node ID of the PU node or power island/domain to be powered down.
const enum `XPmRequestAck`	ack	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_AbortSuspend

This function is called by a CPU after a XPm_SelfSuspend call to notify the power management controller that CPU has aborted suspend or in response to an init suspend request when the PU refuses to suspend.

Note: Calling PU expects the PMU to abort the initiated suspend procedure. This is a non-blocking call without any acknowledge.

Prototype

```
XStatus XPm_AbortSuspend(const enum XPmAbortReason reason);
```

Parameters

The following table lists the `XPm_AbortSuspend` function arguments.

Table 276: **XPm_AbortSuspend Arguments**

Type	Name	Description
const enum `XPmAbortReason`	reason	Reason code why the suspend can not be performed or completed • ABORT_REASON_WKUP_EVENT : local wakeup-event received • ABORT_REASON_PU_BUSY : PU is busy • ABORT_REASON_NO_PWRDN : no external powerdown supported • ABORT_REASON_UNKNOWN : unknown error during suspend procedure

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SetWakeUpSource

This function is called by a PU to add or remove a wake-up source prior to going to suspend. The list of wake sources for a PU is automatically cleared whenever the PU is woken up or when one of its CPUs aborts the suspend procedure.

Note: Declaring a node as a wakeup source will ensure that the node will not be powered off. It also will cause the PMU to configure the GIC Proxy accordingly if the FPD is powered off.

Prototype

```
XStatus XPm_SetWakeUpSource(const enum XPmNodeId target, const enum
XPmNodeId wkup_node, const u8 enable);
```

Parameters

The following table lists the `XPm_SetWakeUpSource` function arguments.

Table 277: **XPm_SetWakeUpSource Arguments**

Type	Name	Description
const enum `XPmNodeId`	target	Node ID of the target to be woken up.
const enum `XPmNodeId`	wkup_node	Node ID of the wakeup device.
const u8	enable	Enable flag: • 1 : the wakeup source is added to the list • 0 : the wakeup source is removed from the list

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SystemShutdown

This function can be used by a privileged PU to shut down or restart the complete device.

Note: In either case the PMU will call XPm_InitSuspendCb for each of the other PUs, allowing them to gracefully shut down. If a PU is asleep it will be woken up by the PMU. The PU making the XPm_SystemShutdown should perform its own suspend procedure after calling this API. It will not receive an init suspend callback.

Prototype

```
XStatus XPm_SystemShutdown(u32 type, u32 subtype);
```

Parameters

The following table lists the `XPm_SystemShutdown` function arguments.

Table 278: **XPm_SystemShutdown Arguments**

Type	Name	Description
Commented parameter restart does not exist in function XPm_SystemShutdown.	restart	Should the system be restarted automatically? • PM_SHUTDOWN : no restart requested, system will be powered off permanently • PM_RESTART : restart is requested, system will go through a full reset

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_RequestNode

Used to request the usage of a PM-slave. Using this API call a PU requests access to a slave device and asserts its requirements on that device. Provided the PU is sufficiently privileged, the PMU will enable access to the memory mapped region containing the control registers of that device. For devices that can only be serving a single PU, any other privileged PU will now be blocked from accessing this device until the node is released.

Note: None

Prototype

```
XStatus XPm_RequestNode(const enum XPmNodeId node, const u32 capabilities,
const u32 qos, const enum XPmRequestAck ack);
```

Parameters

The following table lists the `XPm_RequestNode` function arguments.

Table 279: **XPm_RequestNode Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	Node ID of the PM slave requested
const u32	capabilities	Slave-specific capabilities required, can be combined • PM_CAP_ACCESS : full access / functionality • PM_CAP_CONTEXT : preserve context • PM_CAP_WAKEUP : emit wake interrupts
const u32	qos	Quality of Service (0-100) required
const enum `XPmRequestAck`	ack	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SetRequirement

This function is used by a PU to announce a change in requirements for a specific slave node which is currently in use.

Note: If this function is called after the last awake CPU within the PU calls SelfSuspend, the requirement change shall be performed after the CPU signals the end of suspend to the power management controller, (e.g. WFI interrupt).

Prototype

```
XStatus XPm_SetRequirement(const enum XPmNodeId nid, const u32
capabilities, const u32 qos, const enum XPmRequestAck ack);
```

Parameters

The following table lists the `XPm_SetRequirement` function arguments.

Table 280: **XPm_SetRequirement Arguments**

Type	Name	Description
const enum `XPmNodeId`	nid	Node ID of the PM slave.
const u32	capabilities	Slave-specific capabilities required.
const u32	qos	Quality of Service (0-100) required.
const enum `XPmRequestAck`	ack	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ReleaseNode

This function is used by a PU to release the usage of a PM slave. This will tell the power management controller that the node is no longer needed by that PU, potentially allowing the node to be placed into an inactive state.

***Note*:** None

Prototype

```
XStatus XPm_ReleaseNode(const enum XPmNodeId node);
```

Parameters

The following table lists the `XPm_ReleaseNode` function arguments.

Table 281: **XPm_ReleaseNode Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	Node ID of the PM slave.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SetMaxLatency

This function is used by a PU to announce a change in the maximum wake-up latency requirements for a specific slave node currently used by that PU.

***Note*:** Setting maximum wake-up latency can constrain the set of possible power states a resource can be put into.

Prototype

```
XStatus XPm_SetMaxLatency(const enum XPmNodeId node, const u32 latency);
```

Parameters

The following table lists the `XPm_SetMaxLatency` function arguments.

Table 282: **XPm_SetMaxLatency Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	Node ID of the PM slave.
const u32	latency	Maximum wake-up latency required.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_InitSuspendCb

Callback function to be implemented in each PU, allowing the power management controller to request that the PU suspend itself.

***Note*:** If the PU fails to act on this request the power management controller or the requesting PU may choose to employ the forceful power down option.

Prototype

```
void XPm_InitSuspendCb(const enum XPmSuspendReason reason, const u32
latency, const u32 state, const u32 timeout);
```

Parameters

The following table lists the `XPm_InitSuspendCb` function arguments.

Table 283: **XPm_InitSuspendCb Arguments**

Type	Name	Description
const enum `XPmSuspendReason`	reason	Suspend reason: • SUSPEND_REASON_PU_REQ : Request by another PU • SUSPEND_REASON_ALERT : Unrecoverable SysMon alert • SUSPEND_REASON_SHUTDOWN : System shutdown • SUSPEND_REASON_RESTART : System restart
const u32	latency	Maximum wake-up latency in us(micro secs). This information can be used by the PU to decide what level of context saving may be required.

Table 283: **XPm_InitSuspendCb Arguments** *(cont'd)*

Type	Name	Description
const u32	state	Targeted sleep/suspend state.
const u32	timeout	Timeout in ms, specifying how much time a PU has to initiate its suspend procedure before it's being considered unresponsive.

Returns

None

XPm_AcknowledgeCb

This function is called by the power management controller in response to any request where an acknowledge callback was requested, i.e. where the 'ack' argument passed by the PU was REQUEST_ACK_NON_BLOCKING.

Note: None

Prototype

```
void XPm_AcknowledgeCb(const enum XPmNodeId node, const XStatus status,
const u32 oppoint);
```

Parameters

The following table lists the `XPm_AcknowledgeCb` function arguments.

Table 284: **XPm_AcknowledgeCb Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	ID of the component or sub-system in question.
const XStatus	status	Status of the operation: • OK: the operation completed successfully • ERR: the requested operation failed
const u32	oppoint	Operating point of the node in question

Returns

None

XPm_NotifyCb

This function is called by the power management controller if an event the PU was registered for has occurred. It will populate the notifier data structure passed when calling XPm_RegisterNotifier.

***Note*:** None

Prototype

```
void XPm_NotifyCb(const enum XPmNodeId node, const enum XPmNotifyEvent
event, const u32 oppoint);
```

Parameters

The following table lists the `XPm_NotifyCb` function arguments.

Table 285: **XPm_NotifyCb Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	ID of the node the event notification is related to.
const enum `XPmNotifyEvent`	event	ID of the event
const u32	oppoint	Current operating state of the node.

Returns

None

XPm_GetApiVersion

This function is used to request the version number of the API running on the power management controller.

***Note*:** None

Prototype

```
XStatus XPm_GetApiVersion(u32 *version);
```

Parameters

The following table lists the `XPm_GetApiVersion` function arguments.

Table 286: **XPm_GetApiVersion Arguments**

Type	Name	Description
u32 *	version	Returns the API 32-bit version number. Returns 0 if no PM firmware present.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_GetNodeStatus

This function is used to obtain information about the current state of a component. The caller must pass a pointer to an `XPm_NodeStatus` structure, which must be pre-allocated by the caller.

- status - The current power state of the requested node.
  - For CPU nodes:
    - 0 : if CPU is powered down,
    - 1 : if CPU is active (powered up),
    - 2 : if CPU is suspending (powered up)
  - For power islands and power domains:
    - 0 : if island is powered down,
    - 1 : if island is powered up
  - For PM slaves:
    - 0 : if slave is powered down,
    - 1 : if slave is powered up,
    - 2 : if slave is in retention
- requirement - Slave nodes only: Returns current requirements the requesting PU has requested of the node.
- usage - Slave nodes only: Returns current usage status of the node:
  - 0 : node is not used by any PU,
  - 1 : node is used by caller exclusively,
  - 2 : node is used by other PU(s) only,
  - 3 : node is used by caller and by other PU(s)

***Note*:** None

Prototype

```
XStatus XPm_GetNodeStatus(const enum XPmNodeId node, XPm_NodeStatus *const
nodestatus);
```

Parameters

The following table lists the `XPm_GetNodeStatus` function arguments.

Table 287: **XPm_GetNodeStatus Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	ID of the component or sub-system in question.
`XPm_NodeStatus` *const	nodestatus	Used to return the complete status of the node.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_GetOpCharacteristic

Call this function to request the power management controller to return information about an operating characteristic of a component.

Note: None

Prototype

```
XStatus XPm_GetOpCharacteristic(const enum XPmNodeId node, const enum
XPmOpCharType type, u32 *const result);
```

Parameters

The following table lists the `XPm_GetOpCharacteristic` function arguments.

Table 288: **XPm_GetOpCharacteristic Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	ID of the component or sub-system in question.
const enum `XPmOpCharType`	type	Type of operating characteristic requested: • power (current power consumption), • latency (current latency in us to return to active state), • temperature (current temperature),
u32 *const	result	Used to return the requested operating characteristic.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ResetAssert

This function is used to assert or release reset for a particular reset line. Alternatively a reset pulse can be requested as well.

Note: None

Prototype

```
XStatus XPm_ResetAssert(const enum XPmReset reset, const enum
XPmResetAction resetaction);
```

Parameters

The following table lists the `XPm_ResetAssert` function arguments.

Table 289: **XPm_ResetAssert Arguments**

Type	Name	Description
const enum `XPmReset`	reset	ID of the reset line
Commented parameter assert does not exist in function XPm_ResetAssert.	assert	Identifies action: • PM_RESET_ACTION_RELEASE : release reset, • PM_RESET_ACTION_ASSERT : assert reset, • PM_RESET_ACTION_PULSE : pulse reset,

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ResetGetStatus

Call this function to get the current status of the selected reset line.

Note: None

Prototype

```
XStatus XPm_ResetGetStatus(const enum XPmReset reset, u32 *status);
```

Parameters

The following table lists the `XPm_ResetGetStatus` function arguments.

Table 290: **XPm_ResetGetStatus Arguments**

Type	Name	Description
const enum `XPmReset`	reset	Reset line
u32 *	status	Status of specified reset (true - asserted, false - released)

Returns

Returns 1/XST_FAILURE for 'asserted' or 0/XST_SUCCESS for 'released'.

XPm_RegisterNotifier

A PU can call this function to request that the power management controller call its notify callback whenever a qualifying event occurs. One can request to be notified for a specific or any event related to a specific node.

- nodeID : ID of the node to be notified about,
- eventID : ID of the event in question, '-1' denotes all events (- EVENT_STATE_CHANGE, EVENT_ZERO_USERS),
- wake : true: wake up on event, false: do not wake up (only notify if awake), no buffering/ queueing
- callback : Pointer to the custom callback function to be called when the notification is available. The callback executes from interrupt context, so the user must take special care when implementing the callback. Callback is optional, may be set to NULL.
- received : Variable indicating how many times the notification has been received since the notifier is registered.

***Note*:** The caller shall initialize the notifier object before invoking the XPm_RegisteredNotifier function. While notifier is registered, the notifier object shall not be modified by the caller.

Prototype

```
XStatus XPm_RegisterNotifier(XPm_Notifier *const notifier);
```

Parameters

The following table lists the `XPm_RegisterNotifier` function arguments.

Table 291: **XPm_RegisterNotifier Arguments**

Type	Name	Description
`XPm_Notifier` *const	notifier	Pointer to the notifier object to be associated with the requested notification. The notifier object contains the following data related to the notification:

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_UnregisterNotifier

A PU calls this function to unregister for the previously requested notifications.

***Note*:** None

Prototype

```
XStatus XPm_UnregisterNotifier(XPm_Notifier *const notifier);
```

Parameters

The following table lists the `XPm_UnregisterNotifier` function arguments.

Table 292: **XPm_UnregisterNotifier Arguments**

Type	Name	Description
`XPm_Notifier` *const	notifier	Pointer to the notifier object associated with the previously requested notification

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_MmioWrite

Call this function to write a value directly into a register that isn't accessible directly, such as registers in the clock control unit. This call is bypassing the power management logic. The permitted addresses are subject to restrictions as defined in the PCW configuration.

***Note*:** If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_MmioWrite(const u32 address, const u32 mask, const u32 value);
```

Parameters

The following table lists the `XPm_MmioWrite` function arguments.

Table 293: **XPm_MmioWrite Arguments**

Type	Name	Description
const u32	address	Physical 32-bit address of memory mapped register to write to.
const u32	mask	32-bit value used to limit write to specific bits in the register.
const u32	value	Value to write to the register bits specified by the mask.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_MmioRead

Call this function to read a value from a register that isn't accessible directly. The permitted addresses are subject to restrictions as defined in the PCW configuration.

***Note*:** If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_MmioRead(const u32 address, u32 *const value);
```

Parameters

The following table lists the `XPm_MmioRead` function arguments.

Table 294: **XPm_MmioRead Arguments**

Type	Name	Description
const u32	address	Physical 32-bit address of memory mapped register to read from.
u32 *const	value	Returns the 32-bit value read from the register

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ClockEnable

Call this function to enable (activate) a clock.

***Note*:** If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockEnable(const enum XPmClock clock);
```

Parameters

The following table lists the `XPm_ClockEnable` function arguments.

Table 295: **XPm_ClockEnable Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock to be enabled

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockDisable

Call this function to disable (gate) a clock.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockDisable(const enum XPmClock clock);
```

Parameters

The following table lists the `XPm_ClockDisable` function arguments.

Table 296: **XPm_ClockDisable Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock to be disabled

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockGetStatus

Call this function to get status of a clock gate state.

Prototype

```
XStatus XPm_ClockGetStatus(const enum XPmClock clock, u32 *const status);
```

Parameters

The following table lists the `XPm_ClockGetStatus` function arguments.

Table 297: **XPm_ClockGetStatus Arguments**

Type	Name	Description
const enum XPmClock	clock	Identifier of the target clock
u32 *const	status	Location to store clock gate state (1=enabled, 0=disabled)

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockSetOneDivider

Call this function to set divider for a clock.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockSetOneDivider(const enum XPmClock clock, const u32
divider, const u32 divId);
```

Parameters

The following table lists the `XPm_ClockSetOneDivider` function arguments.

Table 298: **XPm_ClockSetOneDivider Arguments**

Type	Name	Description
const enum XPmClock	clock	Identifier of the target clock
const u32	divider	Divider value to be set
const u32	divId	ID of the divider to be set

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockSetDivider

Call this function to set divider for a clock.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockSetDivider(const enum XPmClock clock, const u32 divider);
```

Parameters

The following table lists the `XPm_ClockSetDivider` function arguments.

Table 299: **XPm_ClockSetDivider Arguments**

Type	Name	Description
const enum XPmClock	clock	Identifier of the target clock
const u32	divider	Divider value to be set

Returns

XST_INVALID_PARAM or status of performing the operation as returned by the PMU-FW

XPm_ClockGetOneDivider

Local function to get one divider (DIV0 or DIV1) of a clock.

Prototype

```
XStatus XPm_ClockGetOneDivider(const enum XPmClock clock, u32 *const
divider, const u32 divId);
```

Parameters

The following table lists the `XPm_ClockGetOneDivider` function arguments.

Table 300: **XPm_ClockGetOneDivider Arguments**

Type	Name	Description
const enum XPmClock	clock	Identifier of the target clock
u32 *const	divider	Location to store the divider value

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockGetDivider

Call this function to get divider of a clock.

Prototype

```
XStatus XPm_ClockGetDivider(const enum XPmClock clock, u32 *const divider);
```

Parameters

The following table lists the `XPm_ClockGetDivider` function arguments.

Table 301: **XPm_ClockGetDivider Arguments**

Type	Name	Description
const enum XPmClock	clock	Identifier of the target clock
u32 *const	divider	Location to store the divider value

Returns

XST_INVALID_PARAM or status of performing the operation as returned by the PMU-FW

XPm_ClockSetParent

Call this function to set parent for a clock.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockSetParent(const enum XPmClock clock, const enum XPmClock
parent);
```

Parameters

The following table lists the `XPm_ClockSetParent` function arguments.

Table 302: **XPm_ClockSetParent Arguments**

Type	Name	Description
const enum XPmClock	clock	Identifier of the target clock
const enum XPmClock	parent	Identifier of the target parent clock

Returns

XST_INVALID_PARAM or status of performing the operation as returned by the PMU-FW.

XPm_ClockGetParent

Call this function to get parent of a clock.

Prototype

```
XStatus XPm_ClockGetParent(const enum XPmClock clock, enum XPmClock *const
parent);
```

Parameters

The following table lists the `XPm_ClockGetParent` function arguments.

Table 303: **XPm_ClockGetParent Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock
enum `XPmClock` *const	parent	Location to store clock parent ID

Returns

XST_INVALID_PARAM or status of performing the operation as returned by the PMU-FW.

XPm_ClockSetRate

Call this function to set rate of a clock.

***Note*:** If the action isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockSetRate(const enum XPmClock clock, const u32 rate);
```

Parameters

The following table lists the `XPm_ClockSetRate` function arguments.

Table 304: **XPm_ClockSetRate Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock
const u32	rate	Clock frequency (rate) to be set

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockGetRate

Call this function to get rate of a clock.

Prototype

```
XStatus XPm_ClockGetRate(const enum XPmClock clock, u32 *const rate);
```

Parameters

The following table lists the `XPm_ClockGetRate` function arguments.

Table 305: **XPm_ClockGetRate Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock
u32 *const	rate	Location where the rate should be stored

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PllSetParameter

Call this function to set a PLL parameter.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_PllSetParameter(const enum XPmNodeId node, const enum
XPmPllParam parameter, const u32 value);
```

Parameters

The following table lists the `XPm_PllSetParameter` function arguments.

Table 306: **XPm_PllSetParameter Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	PLL node identifier
const enum XPmPllParam	parameter	PLL parameter identifier
const u32	value	Value of the PLL parameter

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PllGetParameter

Call this function to get a PLL parameter.

Prototype

```
XStatus XPm_PllGetParameter(const enum XPmNodeId node, const enum
XPmPllParam parameter, u32 *const value);
```

Parameters

The following table lists the `XPm_PllGetParameter` function arguments.

Table 307: **XPm_PllGetParameter Arguments**

Type	Name	Description
const enum XPmNodeId	node	PLL node identifier
const enum XPmPllParam	parameter	PLL parameter identifier
u32 *const	value	Location to store value of the PLL parameter

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PllSetMode

Call this function to set a PLL mode.

***Note*:** If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_PllSetMode(const enum XPmNodeId node, const enum XPmPllMode
mode);
```

Parameters

The following table lists the `XPm_PllSetMode` function arguments.

Table 308: **XPm_PllSetMode Arguments**

Type	Name	Description
const enum XPmNodeId	node	PLL node identifier
const enum XPmPllMode	mode	PLL mode to be set

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PllGetMode

Call this function to get a PLL mode.

Prototype

```
XStatus XPm_PllGetMode(const enum XPmNodeId node, enum XPmPllMode *const
mode);
```

Parameters

The following table lists the `XPm_PllGetMode` function arguments.

Table 309: **XPm_PllGetMode Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	PLL node identifier
enum XPmPllMode *const	mode	Location to store the PLL mode

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlAction

Locally used function to request or release a pin control.

Prototype

```
XStatus XPm_PinCtrlAction(const u32 pin, const enum XPmApiId api);
```

Parameters

The following table lists the `XPm_PinCtrlAction` function arguments.

Table 310: **XPm_PinCtrlAction Arguments**

Type	Name	Description
const u32	pin	PIN identifier (index from range 0-77) @api API identifier (request or release pin control)

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlRequest

Call this function to request a pin control.

Prototype

```
XStatus XPm_PinCtrlRequest(const u32 pin);
```

Parameters

The following table lists the `XPm_PinCtrlRequest` function arguments.

Table 311: **XPm_PinCtrlRequest Arguments**

Type	Name	Description
const u32	pin	PIN identifier (index from range 0-77)

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlRelease

Call this function to release a pin control.

Prototype

```
XStatus XPm_PinCtrlRelease(const u32 pin);
```

Parameters

The following table lists the `XPm_PinCtrlRelease` function arguments.

Table 312: **XPm_PinCtrlRelease Arguments**

Type	Name	Description
const u32	pin	PIN identifier (index from range 0-77)

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlSetFunction

Call this function to set a pin function.

***Note*:** If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_PinCtrlSetFunction(const u32 pin, const enum XPmPinFn fn);
```

Parameters

The following table lists the `XPm_PinCtrlSetFunction` function arguments.

Table 313: **XPm_PinCtrlSetFunction Arguments**

Type	Name	Description
const u32	pin	Pin identifier
const enum XPmPinFn	fn	Pin function to be set

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlGetFunction

Call this function to get currently configured pin function.

Prototype

```
XStatus XPm_PinCtrlGetFunction(const u32 pin, enum XPmPinFn *const fn);
```

Parameters

The following table lists the `XPm_PinCtrlGetFunction` function arguments.

Table 314: **XPm_PinCtrlGetFunction Arguments**

Type	Name	Description
const u32	pin	PLL node identifier
enum XPmPinFn *const	fn	Location to store the pin function

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlSetParameter

Call this function to set a pin parameter.

***Note*:** If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_PinCtrlSetParameter(const u32 pin, const enum XPmPinParam
param, const u32 value);
```

Parameters

The following table lists the `XPm_PinCtrlSetParameter` function arguments.

Table 315: **XPm_PinCtrlSetParameter Arguments**

Type	Name	Description
const u32	pin	Pin identifier
const enum XPmPinParam	param	Pin parameter identifier
const u32	value	Value of the pin parameter to set

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlGetParameter

Call this function to get currently configured value of pin parameter.

Prototype

```
XStatus XPm_PinCtrlGetParameter(const u32 pin, const enum XPmPinParam
param, u32 *const value);
```

Parameters

The following table lists the `XPm_PinCtrlGetParameter` function arguments.

Table 316: **XPm_PinCtrlGetParameter Arguments**

Type	Name	Description
const u32	pin	Pin identifier
const enum XPmPinParam	param	Pin parameter identifier
u32 *const	value	Location to store value of the pin parameter

Returns

Status of performing the operation as returned by the PMU-FW

Error Status

This section lists the Power management specific return error statuses.

PLM error codes format is '0xXXXXYYYY'. Where:

- XXXX - PLM/LOADER/XPLMI error codes as defined in the xplmi_status.h file.
- YYYY - Libraries / Drivers error code as defined in respective modules.

Definitions

Define XST_PM_INTERNAL

Definition

```
#define XST_PM_INTERNAL2000L
```

Description

An internal error occurred while performing the requested operation

Define XST_PM_CONFLICT

Definition

```
#define XST_PM_CONFLICT2001L
```

Description

Conflicting requirements have been asserted when more than one processing cluster is using the same PM slave

Define XST_PM_NO_ACCESS

Definition

```
#define XST_PM_NO_ACCESS2002L
```

Description

The processing cluster does not have access to the requested node or operation

Define XST_PM_INVALID_NODE

Definition

```
#define XST_PM_INVALID_NODE2003L
```

Description

The API function does not apply to the node passed as argument

Define XST_PM_DOUBLE_REQ

Definition

```
#define XST_PM_DOUBLE_REQ2004L
```

Description

A processing cluster has already been assigned access to a PM slave and has issued a duplicate request for that PM slave

Define XST_PM_ABORT_SUSPEND

Definition

```
#define XST_PM_ABORT_SUSPEND2005L
```

Description

The target processing cluster has aborted suspend

Define XST_PM_TIMEOUT

Definition

```
#define XST_PM_TIMEOUT2006L
```

Description

A timeout occurred while performing the requested operation

Define XST_PM_NODE_USED

Definition

```
#define XST_PM_NODE_USED2007L
```

Description

Slave request cannot be granted since node is non-shareable and used

Data Structure Index

The following is a list of data structures:

- XPm_Master
- XPm_NodeStatus
- XPm_Notifier
- pm_acknowledge
- pm_init_suspend

pm_acknowledge

Declaration

```
typedef struct
{
  u8 received,
  u32 node,
  XStatus status,
  u32 opp
} pm_acknowledge;
```

Table 317: **Structure pm_acknowledge member description**

Member	Description
received	Has acknowledge argument been received?
node	Node argument about which the acknowledge is
status	Acknowledged status
opp	Operating point of node in question

pm_init_suspend

Declaration

```
typedef struct
{
  u8 received,
  enum XPmSuspendReason reason,
  u32 latency,
  u32 state,
  u32 timeout
} pm_init_suspend;
```

Table 318: **Structure pm_init_suspend member description**

Member	Description
received	Has init suspend callback been received/handled
reason	Reason of initializing suspend
latency	Maximum allowed latency
state	Targeted sleep/suspend state
timeout	Period of time the client has to response

XPm_Master

`XPm_Master` - Master structure

Declaration

```
typedef struct
{
  enum XPmNodeId node_id,
  const u32 pwrctl,
  const u32 pwrdn_mask,
  XIpiPsu * ipi
} XPm_Master;
```

Table 319: **Structure XPm_Master member description**

Member	Description
node_id	Node ID
pwrctl	
pwrdn_mask	< Power Control Register Address Power Down Mask
ipi	IPI Instance

XPm_NodeStatus

`XPm_NodeStatus` - struct containing node status information

Declaration

```
typedef struct
{
  u32 status,
  u32 requirements,
  u32 usage
} XPm_NodeStatus;
```

Table 320: **Structure XPm_NodeStatus member description**

Member	Description
status	Node power state
requirements	Current requirements asserted on the node (slaves only)
usage	Usage information (which master is currently using the slave)

XPm_Notifier

`XPm_Notifier` - Notifier structure registered with a callback by app

Declaration

```
typedef struct
{
  void(*const callback)(struct XPm_Ntfier *const notifier),
  const u32 node,
  enum XPmNotifyEvent event,
  u32 flags,
  u32 oppoint,
  u32 received,
  struct XPm_Ntfier * next
} XPm_Notifier;
```

Table 321: **Structure XPm_Notifier member description**

Member	Description
callback	Custom callback handler to be called when the notification is received. The custom handler would execute from interrupt context, it shall return quickly and must not block! (enables event-driven notifications)
node	Node argument (the node to receive notifications about)
event	Event argument (the event type to receive notifications about)
flags	Flags
oppoint	Operating point of node in question. Contains the value updated when the last event notification is received. User shall not modify this value while the notifier is registered.
received	How many times the notification has been received - to be used by application (enables polling). User shall not modify this value while the notifier is registered.
next	Pointer to next notifier in linked list. Must not be modified while the notifier is registered. User shall not ever modify this value.

XilFPGA Library v5.2

Overview

The XilFPGA library provides an interface to the Linux or bare-metal users for configuring the programmable logic (PL) over PCAP from PS. The library is designed for Zynq UltraScale+ MPSoC to run on top of Xilinx standalone BSPs. It is tested for A53, R5 and MicroBlaze. In the most common use case, we expect users to run this library on the PMU MicroBlaze with PMUFW to serve requests from either Linux or Uboot for Bitstream programming.

Note: XILFPGA does not support a DDR less system. DDR must be present for use of XilFPGA.

Supported Features

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

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

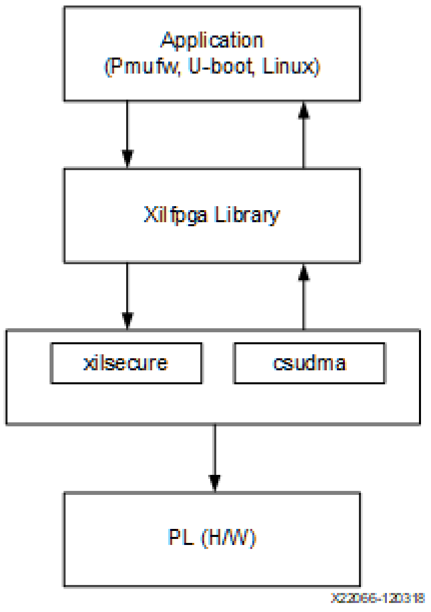
The XilSecure library provides APIs to access secure hardware on the Zynq UltraScale+ MPSoC devices.

***Note*:** The current version of library supports only Zynq UltraScale MPSoC devices.

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.

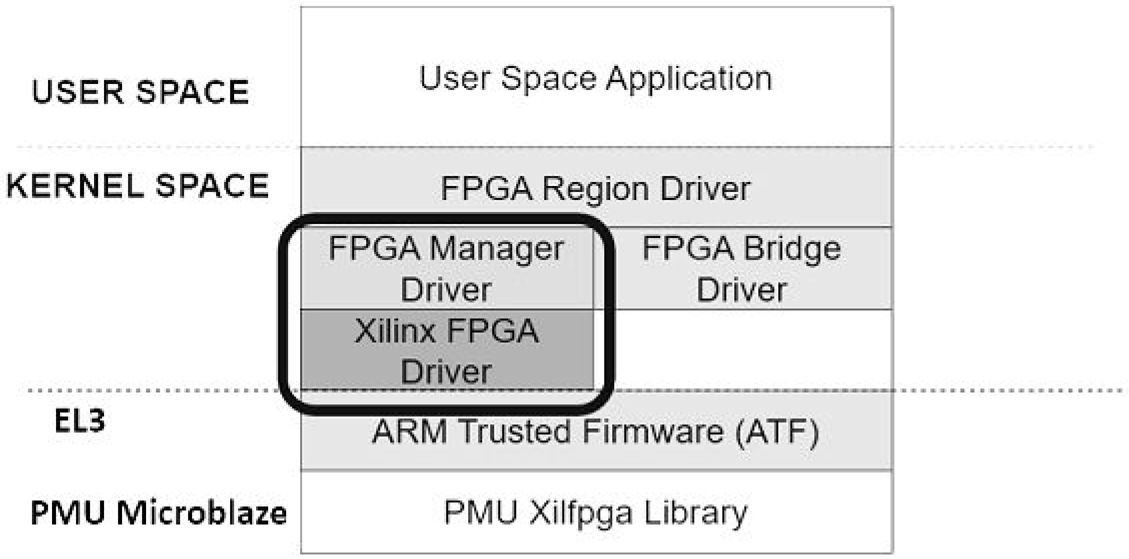
Figure 3: **XilFPGA Design Summary**



Flow Diagram

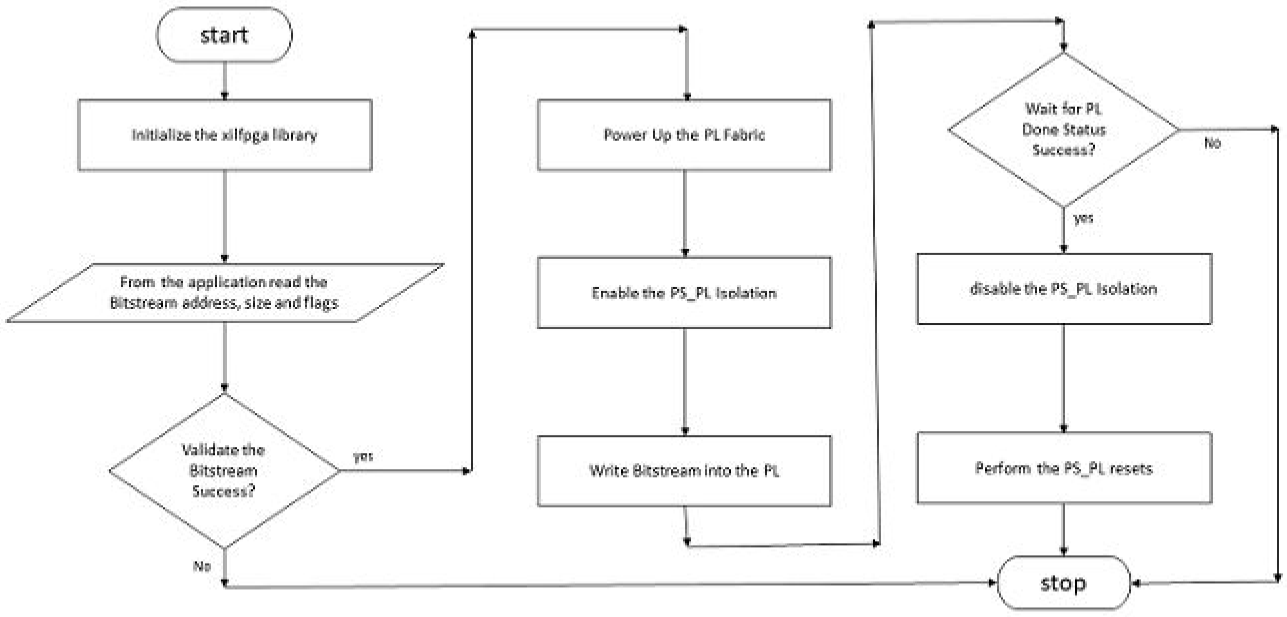
The following figure illustrates the Bitstream loading flow on the Linux operating system.

Figure 4: **Bitstream loading on Linux:**



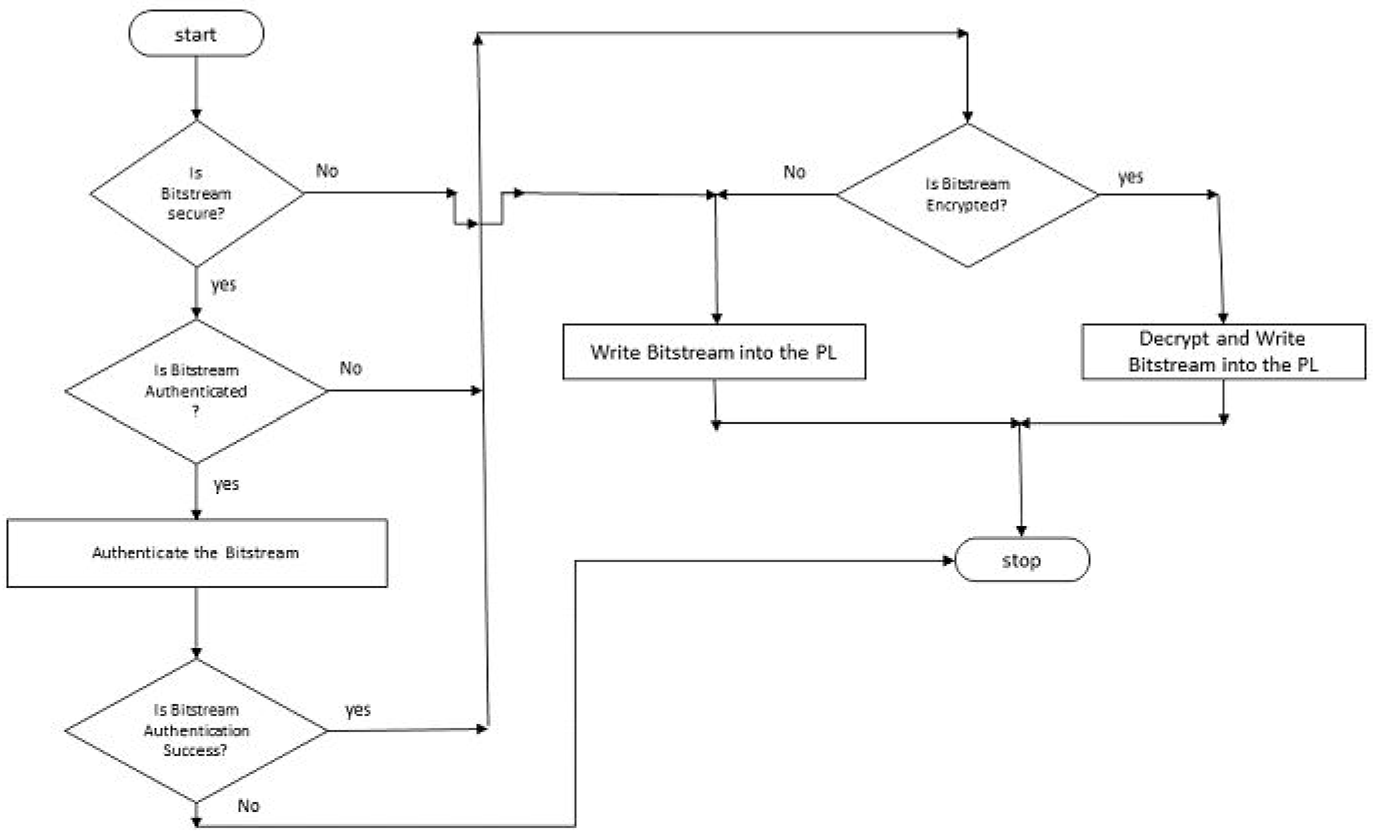
The following figure illustrates the XilFPGA PL configuration sequence.

Figure 5: **XilFPGA PL Configuration Sequence**


start
Initialize the xilfpga library
From the application read the Bitstream address, size and flags
Validate the Bitstream Success?
yes
No
Power Up the PL Fabric
Enable the PS_PL Isolation
Write Bitstream into the PL
Wait for PL Done Status Success?
No
yes
disable the PS_PL Isolation
Perform the PS_PL resets
stop

The following figure illustrates the Bitstream write sequence.

Figure 6: **Bitstream write Sequence**


start
Is Bitstream secure?
No
yes
Is Bitstream Authenticated ?
No
yes
Authenticate the Bitstream
Is Bitstream Authentication Success?
yes
No
Is Bitstream Encrypted?
No
yes
Write Bitstream into the PL
Decrypt and Write Bitstream into the PL
stop

XilFPGA BSP Configuration Settings

The XilFPGA library provides user configuration BSP settings. The following table describes the parameters and their default value:

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-R5 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.

Parameter Name	Type	Default Value	Description
secure_environment	bool	FALSE	Enable the secure PL configuration using the IPI. This flag is valid only for the Cortex-A53 or the Cortex-R5 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

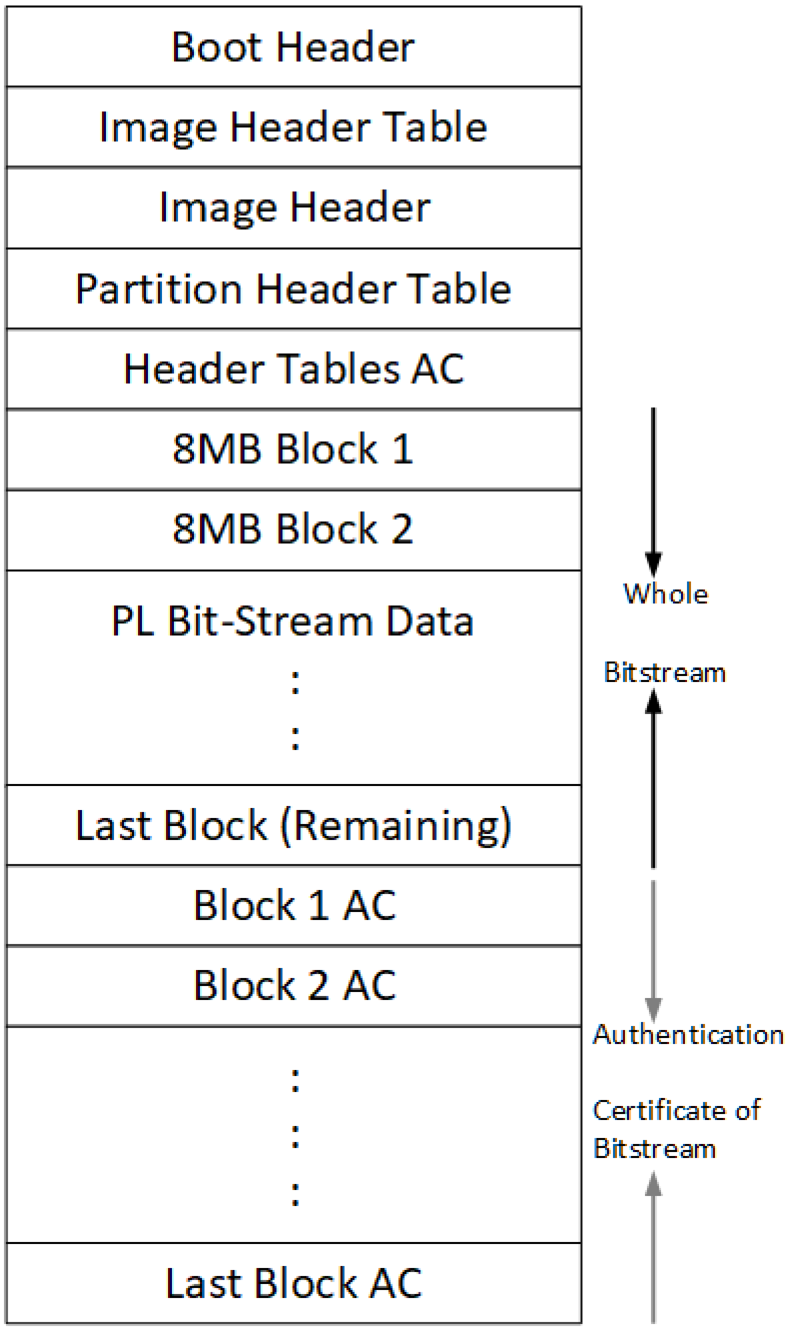
The size of the Bitstream is too large to be contained inside the device, therefore external memory must be used. The use of external memory could create a security risk. Therefore, two methods are provided to authenticate and decrypt a Bitstream.

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

Bootgen

When a Bitstream is requested for authentication, Bootgen divides the Bitstream into blocks of 8MB 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 7: **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.

Authenticated and Encrypted Bitstream Loading Using OCM

To authenticate the Bitstream partition securely, XilFPGA uses the FSBL section's OCM memory to copy the bitstream in chunks from DDR. This method does not require trust in the external DDR 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 56KB and hash of chunks to store intermediate hashes calculated for each 56 KB of every 8MB block.
2. XilFPGA copies a 56KB chunk from the first 8MB 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 8MB of block is completed.

   Note: The chunk that XilFPGA copies can be of any size. A 56KB chunk is taken for better performance.
5. XilFPGA authenticates the 8MB Bitstream chunk.
6. Once the authentication is successful, XilFPGA starts copying information in batches of 56KB starting from the first block which is located in DDR 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 8MB 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). PMU checks if FSBL image is encrypted and skips copying FSBL from OCM to reserved DDR memory. In this case, If XilFPGA uses OCM memory for authenticating bitstream, APU restart feature will not work.

Also, copying FSBL to DDR for APU restart feature can be disabled by setting USE_DDR_FOR_APU_RESTART_VAL macro value in xpfw_config.h file to 0. If XilFPGA uses OCM memory for authenticating bitstream, APU restart feature will not work.

Authenticated and Encrypted Bitstream Loading Using DDR

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 8MB block.
3. XilFPGA authenticates the 8MB block while stored in the external DDR.
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.

XilFPGA APIs

This section provides detailed descriptions of the XilFPGA library APIs.

Table 322: **Quick Function Reference**

Type	Name	Arguments
u32	XFpga_Initialize	void
u32	XFpga_PL_BitStream_Load	XFpga * InstancePtr UINTPTR BitstreamImageAddr UINTPTR AddrPtr_Size u32 Flags
u32	XFpga_PL_Preconfig	void
u32	XFpga_PL_Write	void
u32	XFpga_PL_PostConfig	XFpga * InstancePtr
u32	XFpga_PL_ValidateImage	XFpga * InstancePtr UINTPTR BitstreamImageAddr UINTPTR AddrPtr_Size u32 Flags
u32	XFpga_GetPlConfigData	XFpga * InstancePtr
u32	XFpga_GetPlConfigReg	XFpga * InstancePtr ConfigReg Address
u32	XFpga_InterfaceStatus	XFpga * InstancePtr

Functions

XFpga_Initialize

Prototype

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

XFpga_PL_BitStream_Load

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

It supports vivado generated Bitstream(*.bit, *.bin) and bootgen generated Bitstream(*.bin) loading, Passing valid Bitstream size (AddrPtr_Size) info is mandatory for vivado * generated Bitstream, For bootgen generated Bitstreams it will take Bitstream size from the Bitstream Header.

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 323: **XFpga_PL_BitStream_Load Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFgpa 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 to store size of Bitstream Image.

Table 323: **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
		• 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_Preconfig

Prototype

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

XFpga_PL_Write

Prototype

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

XFpga_PL_PostConfig

This function set FPGA to operating state after writing.

Prototype

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

Parameters

The following table lists the `XFpga_PL_PostConfig` function arguments.

Table 324: **XFpga_PL_PostConfig Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFgpa structure

Returns

Codes as mentioned in xilfpga.h

XFpga_PL_ValidateImage

This function is used to validate the Bitstream Image.

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 325: **XFpga_PL_ValidateImage Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFgpa 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 to store size of Bitstream Image.

Table 325: **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_GetPlConfigData

Provides functionality to read back the PL configuration data.

Prototype

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

Parameters

The following table lists the `XFpga_GetPlConfigData` function arguments.

Table 326: **XFpga_GetPlConfigData Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFgpa structure

Table 326: **XFpga_GetPlConfigData Arguments** *(cont'd)*

Type	Name	Description
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

Provides PL specific configuration register values.

Prototype

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

Parameters

The following table lists the `XFpga_GetPlConfigReg` function arguments.

Table 327: **XFpga_GetPlConfigReg Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFgpa structure
UINTPTR	ReadbackAddr	Address which is used to store the PL Configuration register data.
u32	ConfigRegAddr	Configuration register address. For more information, see,UG570 - UltraScale Architecture Configuration User Guide.

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 PL programming interface.

Prototype

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

Parameters

The following table lists the `XFpga_InterfaceStatus` function arguments.

Table 328: **XFpga_InterfaceStatus Arguments**

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFgpa structure

Returns

Status of the PL programming interface.

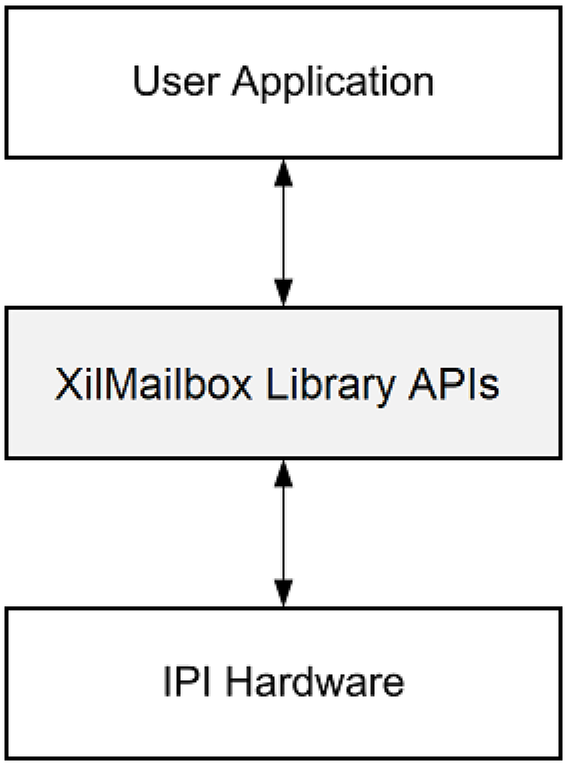
Chapter 11

XilMailbox v1.2

Overview

The XilMailbox library provides the top-level hooks for sending or receiving an inter-processor interrupt (IPI) message using the Zynq® UltraScale+ MPSoC IPI hardware.

Figure 8: **Overview**



For more details on the IPI interrupts, see the Zynq UltraScale+ MPSoC Technical Reference Manual (UG1085).

This library supports the following features:

- Triggering an IPI to a remote agent.
- Sending an IPI message to a remote agent.

- Callbacks for error and recv IPI events.
- Reading an IPI message.

Software Initialization

The following is a list of software initalization events for a given IPI channel:

1. `XMailbox_Initialize()` function initializes a library instance for the given IPI channel.
2. `XMailbox_Send()` function triggers an IPI to a remote agent.
3. `XMailbox_SendData()` function sends an IPI message to a remote agent, message type should be either XILMBOX_MSG_TYPE_REQ (OR) XILMBOX_MSG_TYPE_RESP.
4. `XMailbox_Recv()` function reads an IPI message from a specified source agent, message type should be either XILMBOX_MSG_TYPE_REQ (OR) XILMBOX_MSG_TYPE_RESP.
5. `XMailbox_SetCallBack()` using this function user can register call backs for receive and error events.

Table 329: **Quick Function Reference**

Type	Name	Arguments
u32	XMailbox_Send	`XMailbox` * InstancePtr u32 RemoteId u8 Is_Blocking
u32	XMailbox_SendData	`XMailbox` * InstancePtr u32 RemoteId void * BufferPtr u32 MsgLen u8 BufferType u8 Is_Blocking
u32	XMailbox_Recv	`XMailbox` * InstancePtr u32 SourceId void * BufferPtr u32 MsgLen u8 BufferType
s32	XMailbox_SetCallBack	`XMailbox` * InstancePtr `XMailbox_Handler` HandlerType CallBackFunc CallBackRef
u32	XMailbox_Initialize	`XMailbox` * InstancePtr u8 DeviceId

Table 329: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
u32	XIpiPs_Init	`XMailbox` * InstancePtr u8 DeviceId
u32	XIpiPs_Send	`XMailbox` * InstancePtr u8 Is_Blocking
u32	XIpiPs_SendData	`XMailbox` * InstancePtr void * MsgBufferPtr u32 MsgLen u8 BufferType u8 Is_Blocking
u32	XIpiPs_PollforDone	`XMailbox` * InstancePtr
u32	XIpiPs_RecvData	`XMailbox` * InstancePtr void * MsgBufferPtr u32 MsgLen u8 BufferType
XStatus	XIpiPs_RegisterIrq	void
void	XIpiPs_ErrorIntrHandler	void
void	XIpiPs_IntrHandler	void

Functions

XMailbox_Send

This function triggers an IPI to a destination CPU.

Prototype

```
u32 XMailbox_Send(XMailbox *InstancePtr, u32 RemoteId, u8 Is_Blocking);
```

Parameters

The following table lists the `XMailbox_Send` function arguments.

Table 330: **XMailbox_Send Arguments**

Type	Name	Description
XMailbox *	InstancePtr	Pointer to the XMailbox instance
u32	RemoteId	is the Mask of the CPU to which IPI is to be triggered
u8	Is_Blocking	if set trigger the notification in blocking mode

Returns

- XST_SUCCESS if successful
- XST_FAILURE if unsuccessful

XMailbox_SendData

This function sends an IPI message to a destination CPU.

Prototype

```
u32 XMailbox_SendData(XMailbox *InstancePtr, u32 RemoteId, void *BufferPtr,
u32 MsgLen, u8 BufferType, u8 Is_Blocking);
```

Parameters

The following table lists the `XMailbox_SendData` function arguments.

Table 331: **XMailbox_SendData Arguments**

Type	Name	Description
XMailbox *	InstancePtr	Pointer to the XMailbox instance
u32	RemoteId	is the Mask of the CPU to which IPI is to be triggered
void *	BufferPtr	is the pointer to Buffer which contains the message to be sent
u32	MsgLen	is the length of the buffer/message
u8	BufferType	is the type of buffer (XILMBOX_MSG_TYPE_REQ (OR) XILMBOX_MSG_TYPE_RESP)
u8	Is_Blocking	if set trigger the notification in blocking mode

Returns

- XST_SUCCESS if successful
- XST_FAILURE if unsuccessful

XMailbox_Recv

This function reads an IPI message.

Prototype

```
u32 XMailbox_Recv(XMailbox *InstancePtr, u32 SourceId, void *BufferPtr, u32
MsgLen, u8 BufferType);
```

Parameters

The following table lists the `XMailbox_Recv` function arguments.

Table 332: **XMailbox_Recv Arguments**

Type	Name	Description
`XMailbox` *	InstancePtr	Pointer to the `XMailbox` instance
u32	SourceId	is the Mask for the CPU which has sent the message
void *	BufferPtr	is the pointer to Buffer to which the read message needs to be stored
u32	MsgLen	is the length of the buffer/message
u8	BufferType	is the type of buffer (XILMBOX_MSG_TYPE_REQ or XILMBOX_MSG_TYPE_RESP)

Returns

- XST_SUCCESS if successful
- XST_FAILURE if unsuccessful

XMailbox_SetCallBack

This routine installs an asynchronous callback function for the given HandlerType.

***Note*:** Invoking this function for a handler that already has been installed replaces it with the new handler.

Prototype

```
s32 XMailbox_SetCallBack(XMailbox *InstancePtr, XMailbox_Handler
HandlerType, void *CallBackFuncPtr, void *CallBackRefPtr);
```

Parameters

The following table lists the `XMailbox_SetCallBack` function arguments.

Table 333: **XMailbox_SetCallBack Arguments**

Type	Name	Description
`XMailbox` *	InstancePtr	is a pointer to the `XMailbox` instance.
`XMailbox_Handler`	HandlerType	specifies which callback is to be attached.

Table 333: **XMailbox_SetCallBack Arguments** *(cont'd)*

Type	Name	Description
Commented parameter CallBackFunc does not exist in function XMailbox_SetCallBack.	CallBackFunc	is the address of the callback function.
Commented parameter CallBackRef does not exist in function XMailbox_SetCallBack.	CallBackRef	is a user data item that will be passed to the callback function when it is invoked.

Returns

- XST_SUCCESS when handler is installed.
- XST_INVALID_PARAM when HandlerType is invalid.

XMailbox_Initialize

Initialize the `XMailbox` Instance.

Prototype

```
u32 XMailbox_Initialize(XMailbox *InstancePtr, u8 DeviceId);
```

Parameters

The following table lists the `XMailbox_Initialize` function arguments.

Table 334: **XMailbox_Initialize Arguments**

Type	Name	Description
`XMailbox` *	InstancePtr	is a pointer to the instance to be worked on
u8	DeviceId	is the IPI Instance to be worked on

Returns

XST_SUCCESS if initialization was successful XST_FAILURE in case of failure

XIpiPs_Init

Initialize the ZynqMP Mailbox Instance.

Prototype

```
u32 XIpiPs_Init(XMailbox *InstancePtr, u8 DeviceId);
```

Parameters

The following table lists the `XIpiPs_Init` function arguments.

Table 335: **XIpiPs_Init Arguments**

Type	Name	Description
XMailbox *	InstancePtr	is a pointer to the instance to be worked on
u8	DeviceId	is the IPI Instance to be worked on

Returns

XST_SUCCESS if initialization was successful XST_FAILURE in case of failure

XIpiPs_Send

This function triggers an IPI to a destnation CPU.

Prototype

```
u32 XIpiPs_Send(XMailbox *InstancePtr, u8 Is_Blocking);
```

Parameters

The following table lists the `XIpiPs_Send` function arguments.

Table 336: **XIpiPs_Send Arguments**

Type	Name	Description
XMailbox *	InstancePtr	Pointer to the XMailbox instance.
u8	Is_Blocking	if set trigger the notification in blocking mode

Returns

XST_SUCCESS in case of success XST_FAILURE in case of failure

XIpiPs_SendData

This function sends an IPI message to a destnation CPU.

Prototype

```
u32 XIpiPs_SendData(XMailbox *InstancePtr, void *MsgBufferPtr, u32 MsgLen,
u8 BufferType, u8 Is_Blocking);
```

Parameters

The following table lists the `XIpiPs_SendData` function arguments.

Table 337: **XIpiPs_SendData Arguments**

Type	Name	Description
`XMailbox` *	InstancePtr	Pointer to the `XMailbox` instance
void *	MsgBufferPtr	is the pointer to Buffer which contains the message to be sent
u32	MsgLen	is the length of the buffer/message
u8	BufferType	is the type of buffer
u8	Is_Blocking	if set trigger the notification in blocking mode

Returns

XST_SUCCESS in case of success XST_FAILURE in case of failure

XIpiPs_PollforDone

Poll for an acknowledgement using Observation Register.

Prototype

```
u32 XIpiPs_PollforDone(XMailbox *InstancePtr);
```

Parameters

The following table lists the `XIpiPs_PollforDone` function arguments.

Table 338: **XIpiPs_PollforDone Arguments**

Type	Name	Description
`XMailbox` *	InstancePtr	Pointer to the `XMailbox` instance

Returns

XST_SUCCESS in case of success XST_FAILURE in case of failure

XIpiPs_RecvData

This function reads an IPI message.

Prototype

```
u32 XIpiPs_RecvData(XMailbox *InstancePtr, void *MsgBufferPtr, u32 MsgLen,
u8 BufferType);
```

Parameters

The following table lists the `XIpiPs_RecvData` function arguments.

Table 339: **XIpiPs_RecvData Arguments**

Type	Name	Description
`XMailbox` *	InstancePtr	Pointer to the `XMailbox` instance
void *	MsgBufferPtr	is the pointer to Buffer to which the read message needs to be stored
u32	MsgLen	is the length of the buffer/message
u8	BufferType	is the type of buffer

Returns

- XST_SUCCESS if successful
- XST_FAILURE if unsuccessful

XIpiPs_RegisterIrq

Prototype

```
XStatus XIpiPs_RegisterIrq(XScuGic *IntcInstancePtr, XMailbox *InstancePtr,
u32 IpiIntrId);
```

XIpiPs_ErrorIntrHandler

Prototype

```
void XIpiPs_ErrorIntrHandler(void *XMailboxPtr);
```

XIpiPs_IntrHandler

Prototype

```
void XIpiPs_IntrHandler(void *XMailboxPtr);
```

Enumerations

Enumeration XMailbox_Handler

Contains XMAILBOX Handler Types.

Table 340: **Enumeration XMailbox_Handler Values**

Value	Description
XMAILBOX_RECV_HANDLER	For Recv Handler.
XMAILBOX_ERROR_HANDLER	For Error Handler.

Data Structure Index

The following is a list of data structures:

- XMailbox

XMailbox

Holds the function pointers for the operations that can be performed.

Declaration

```
typedef struct
{
  u32(* XMbox_IPI_Send)(struct XMboxTag *InstancePtr, u8 Is_Blocking),
  u32(* XMbox_IPI_SendData)(struct XMboxTag *InstancePtr, void *BufferPtr,
u32 MsgLen, u8 BufferType, u8 Is_Blocking),
  u32(* XMbox_IPI_Recv)(struct XMboxTag *InstancePtr, void *BufferPtr, u32
MsgLen, u8 BufferType),
  XMailbox_RecvHandler RecvHandler,
  XMailbox_ErrorHandler ErrorHandler,
  void * ErrorRefPtr,
  void * RecvRefPtr,
  XMailbox_Agent Agent
} XMailbox;
```

Table 341: **Structure XMailbox member description**

Member	Description
XMbox_IPI_Send	Triggers an IPI to a destination CPU.
XMbox_IPI_SendData	Sends an IPI message to a destination CPU.
XMbox_IPI_Recv	Reads an IPI message.
RecvHandler	Callback for rx IPI event.
ErrorHandler	Callback for error event.
ErrorRefPtr	To be passed to the error interrupt callback.
RecvRefPtr	To be passed to the receive interrupt callback.
Agent	Used to store IPI Channel information.

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

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright