Standalone Library Documentation

BSP and Libraries Document Collection (UG643)

UG643 (v2023.1) May 16, 2023

AMD Adaptive Computing is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this link for more information.

Table of Contents

Chapter 1: BSP and Libraries Overview **5**

Chapter 2: Xilinx Standard C Libraries **7**

Xilinx Standard C Libraries 7
Arithmetic Operations 8
Input/Output Functions 9

Chapter 3: Standalone Library v8.1 ..... **12**

Hardware Abstraction Layer API 12
MicroBlaze Processor API Reference 30
Arm Processor Common APIs 50
Arm Cortex-R5F Processor APIs 60
Arm Cortex-A9 Processor APIs 76
Arm Cortex-A53 32-bit Processor APIs 103
Arm Cortex-A53 64-bit Processor APIs 114

Chapter 4: LwIP 2.1.3 Library v1.0 ..... **124**

Introduction 124
Using lwIP 125
LwIP Library APIs 134

Chapter 5: XilFlash Library v4.9 ..... **141**

Overview 141
Device Geometry 141
XilFlash Library API 144
Library Parameters in MSS File 152

Chapter 6: XilFFS Library v5.0 ..... **153**

XilFFS Library API Reference 153
Library Parameters in MSS File 156

Chapter 7: XilSecure Library v5.1 ..... **158**

Overview....158
AES-GCM....159
RSA....170
SHA-3....194
Data Structure Index....202

Chapter 8: XilSkey Library v7.4....204
Overview....204
BBRAM PL APIs....209
Zynq UltraScale+ MPSoC BBRAM PS API....211
Zynq eFUSE PS APIs....213
Zynq UltraScale+ MPSoC eFUSE PS APIs....215
eFUSE PL APIs....234
CRC Calculation API....237
User-Configurable Parameters....238
Error Codes....260
Data Structures....270

Chapter 9: XilPM Library v5.0....282
XilPM Zynq UltraScale+ MPSoC APIs....282
XilPM Versal Adaptive SoC APIs....325
XilPM API Version Detail....351
Error Status....357
Error Event Mask....372
Library Parameters in MSS File....372
Data Structure Index....373

Chapter 10: XilFPGA Library v6.4....377
Overview....377
Zynq UltraScale+ MPSoC XilFPGA Library....378
XilFPGA APIs....387

Chapter 11: XilMailbox Library v1.7....398
Overview....398
Data Structure Index....407

Chapter 12: XilSEM Library v1.6....409
Introduction....409
XilSEM Versal adaptive SoC Client APIs....435

Data Structures....468

Chapter 13: XilTimer Library v1.2....476

Overview....476
BSP Configuration Settings....477
XilTimer APIs....477
Data Structures....481

Appendix A: Additional Resources and Legal Notices....483

Finding Additional Documentation....483
Support Resources....484
Revision History....484
Please Read: Important Legal Notices....484

BSP and Libraries Overview

The AMD Vitis™ Unified Software Development Environment provides a variety of software packages, including device drivers, libraries, and board support packages to help you develop a software platform in the baremetal and RTOS based environment. This document describes these software packages in details including API description. The Vitis Unified Software Development Environment also provides the FreeRTOS kernel along with the low level software needed for the kernel to work on a supported processors like MicroBlaze™ , CortexA9, Cortex-R5F, Cortex-A53, and Cortex-A72). 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 v8.1	Describes the Standalone platform, a single-threaded, and a simple operating system (OS) platform that provides the lowest layer of software modules to access the processor-specific functions. Standalone platform functions 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 2.1.3 Library v1.0	Describes the port of the third party networking library, Light Weight IP (lwIP) for embedded processors.
Chapter 5: XilFlash Library v4.9	Provides read/write/erase/lock/unlock features to access a parallel flash device.
Chapter 6: XilFFS Library v5.0	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 v5.1	Provides APIs to access secure hardware on the AMD Zynq™ UltraScale+™ MPSoC.
Chapter 8: XilSkey Library v7.4	Provides a programming mechanism for user-defined eFUSE bits and for programming the KEY into battery-backed RAM (BBRAM) of AMD Zynq™ 7000, 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.
Chapter 9: XilPM Library v5.0	The Zynq UltraScale+ MPSoC and AMD Versal adaptive SoC 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 v6.4	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 and Versal adaptive SoC to run on top of standalone BSPs.

Library Name	Summary
Chapter 11: XilMailbox Library v1.7	Provides the top-level hooks for sending or receiving an inter-processor interrupt (IPI) message using the Zynq UltraScale+ MPSoC and Versal adaptive SoC IP integrator hardware.
Chapter 12: XilSEM Library v1.6	The Xilinx Error Mitigation (XilSEM) library is a pre-configured, pre-verified solution to detect and optionally correct soft errors in Configuration Memory of Versal adaptive SoCs.
Chapter 13: XilTimer Library v1.2	Provides sleep and interval timer functionality, Hardware and Software features that are differentiated using a layered approach.

Chapter 2

Xilinx Standard C 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. These 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 Arm Cortex-A9, Cortex-R5F, Cortext-A53, and Cortex-A72 processors. You can find the header files corresponding to these C standard functions in the `<XILINX_SDK>/gnu/<processor>/<platform>/<toolchain>/<processor-lib>/usr/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 `aarch64-none`, or `gcc-arm-none-eabi`, or `microblazeeb-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 Arm Cortex-A9, Cortex-A53/Cortex-A72 (32-bit mode) processors

```
arm-none-eabi-gcc <C files>
```

- For Arm Cortex-A53/Cortex-A72 (64-bit mode) processors

```
aarch64-none-elf-gcc <C files>
```

- For Cortex-R5F processor

```
armr5-none-eabi-gcc <C files>
```

Memory Management Functions

The MicroBlaze processor and Arm Cortex-A9, Cortex-A53, Cortex-A72, and Cortex-R5F) 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 the 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 corresponds 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 the Vitis IDE 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 embedded libraries contain standard C functions for Input/Output (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 I/O 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 example:

***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.

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

These I/O 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 GNU Compiler Tools in Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400). In addition to the standard C functions, the Vitis IDE processors library provides the following smaller I/O functions:

Quick Function Reference

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

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 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 a 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 into a signed decimal notation.
- l: The int argument is converted into 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 needs 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.
- p: Used for pointer arguments to display the argument as an address in the hexadecimal digits. Applicable only for 64-bit Arm.

Prototype

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

Standalone Library v8.1

Hardware Abstraction Layer API

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

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 1: **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 2: **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 3: **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 4: **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 5: **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 5: **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

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 6: **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 7: **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 8: **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 9: **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 10: **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 11: **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 12: **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 13: **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 14: **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 15: **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 16: **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

Hardware Platform Information

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

Table 17: **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 18: **XGetPlatform_Info Arguments**

Name	Description
None.	

Returns

The information about platform defined in xplatform_info.h

Data types for Software IP Cores

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

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 AMD.

Table 19: **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 20: **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

Software status codes

The xstatus.h file contains the software status codes.These codes are used throughout the AMD 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.

- Cache test: xil_testcache.h contains utility functions to test cache.
- I/O test: The Xil_testio.h file contains 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.

- Memory test: 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 21: **Quick Function Reference**

Type	Name	Arguments
s32	Xil_TestMem32	• u32 * Addr • u32 Words • u32 Pattern • u8 Subtest
s32	Xil_TestMem16	• u16 * Addr • u32 Words • u16 Pattern • u8 Subtest
s32	Xil_TestMem8	• u8 * Addr • u32 Words • u8 Pattern • u8 Subtest
s32	RoateLeft	• u32 Input • u8 Width
s32	RotateRight	
s32	Xil_TestIO8	• u32 Input • u8 Width
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_TestMem32

Perform a destructive 32-bit wide memory test.

***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.

Prototype

```
s32 Xil_TestMem32(u32 *Addr, u32 Words, u32 Pattern, u8 Subtest);
```

Parameters

The following table lists the `Xil_TestMem32` function arguments.

Table 22: **Xil_TestMem32 Arguments**

Name	Description
Addr	Pointer to the region of memory to be tested.
Words	Length of the block.
Pattern	Constant used for the constant pattern test, if 0, 0xDEADBEEF is used.
Subtest	Test type selected. See xil_testmem.h file for possible values.

Returns

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

Xil_TestMem16

Perform a destructive 16-bit wide memory test.

***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.

Prototype

```
s32 Xil_TestMem16(u16 *Addr, u32 Words, u16 Pattern, u8 Subtest);
```

Parameters

The following table lists the `Xil_TestMem16` function arguments.

Table 23: **Xil_TestMem16 Arguments**

Name	Description
Addr	Pointer to the region of memory to be tested.
Words	Length of the block.
Pattern	Constant used for the constant pattern test, if 0, 0xDEADBEEF is used.
Subtest	Test type selected. See xil_testmem.h file for possible values.

Returns

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

Xil_TestMem8

Perform a destructive 8-bit wide memory test.

***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.

Prototype

```
s32 Xil_TestMem8(u8 *Addr, u32 Words, u8 Pattern, u8 Subtest);
```

Parameters

The following table lists the `Xil_TestMem8` function arguments.

Table 24: **Xil_TestMem8 Arguments**

Name	Description
Addr	Pointer to the region of memory to be tested.
Words	Length of the block.
Pattern	Constant used for the constant pattern test, if 0, 0xDEADBEEF is used.

Table 24: **Xil_TestMem8 Arguments** *(cont'd)*

Name	Description
Subtest	Test type selected. See xil_testmem.h file for possible values.

Returns

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

RoateLeft

Rotates the provided value to the left one bit position.

Prototype

```
u32 RotateLeft(u32 Input, u8 Width);
```

Parameters

The following table lists the `RotateLeft` function arguments.

Table 25: **RotateLeft Arguments**

Name	Description
Input	Value to be rotated to the left
Width	Number of bits in the input data

Returns

The resulting unsigned long value of the rotate left

RotateRight

Rotates the provided value to the right one bit position.

Prototype

```
u32 RotateRight(u32 Input, u8 Width);
```

Parameters

The following table lists the `RotateRight` function arguments.

Table 26: **RotateRight Arguments**

Name	Description
Input	Value to be rotated to the right
Width	Number of bits in the input data

Returns

The resulting unsigned long value of the rotate right.

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 27: **Xil_TestIO8 Arguments**

Name	Description
Addr	Pointer to the region of memory to be tested.
Length	Length of the block.
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 28: **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 29: **Xil_TestIO32 Arguments**

Name	Description
Addr	Pointer to the region of memory to be tested.

Table 29: **Xil_TestIO32 Arguments** *(cont'd)*

Name	Description
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 Reference

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 30: **Quick Function Reference**

Type	Name	Arguments
void	microblaze_register_handler	XInterruptHandler Handler void * DataPtr

Table 30: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	microblaze_register_exception_handler	u32 ExceptionId Top void * DataPtr

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 31: **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.

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 32: **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, available in the <install-directory>/src/microblaze folder, contains Microblaze specific exception related APIs and macros.

Application programs can use these APIs for various exception related operations. For example, enable exception, disable exception, register exception hander.

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

Table 33: **Quick Function Reference**

Type	Name	Arguments
void	Xil_ExceptionNullHandler	void * Data
void	Xil_ExceptionInit	void
void	Xil_ExceptionEnable	void
void	Xil_ExceptionDisable	void
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 34: **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);
```

Xil_ExceptionEnable

Enable Exceptions.

Prototype

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

Xil_ExceptionDisable

Disable Exceptions.

Prototype

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

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 35: **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 36: **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 Cache APIs

This contains implementation of cache related driver functions.

The xil_cache.h file contains cache related driver functions (or macros) that can be used to access the device.

The user should refer to the hardware device specification for more details of the device operation. The functions in this header file can be used across all AMD supported processors.

Table 37: **Quick Function Reference**

Type	Name	Arguments
void	Xil_DCacheDisable	void
void	Xil_ICacheDisable	void

Functions

Xil_DCacheDisable

Disable the data cache.

Prototype

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

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Prototype

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

Returns

None.

Definitions

#Define Xil_L1DCacheInvalidate

Description

Invalidate the entire L1 data cache.

If the cacheline is modified (dirty), the modified contents are lost.

***Note*:** Processor must be in real mode.

#Define Xil_L2CacheInvalidate

Description

Invalidate the entire L2 data cache.

If the cacheline is modified (dirty),the modified contents are lost.

***Note*:** Processor must be in real mode.

#Define Xil_L1DCacheInvalidateRange

Description

Invalidate the L1 data cache for the given address range.

If the bytes specified by the address (Addr) are cached by the L1 data cache, the cacheline containing that byte is invalidated.If the cacheline is modified (dirty), the modified contents are lost.

***Note*:** Processor must be in real mode.

Parameters

The following table lists the `Xil_L1DCacheInvalidateRange` function arguments.

Table 38: **Xil_L1DCacheInvalidateRange Arguments**

Name	Description
Addr	is address of range to be invalidated.
Len	is the length in bytes to be invalidated.

#Define Xil_L2CacheInvalidateRange

Description

Invalidate the L1 data cache for the given address range.

If the bytes specified by the address (Addr) are cached by the L1 data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost.

***Note*:** Processor must be in real mode.

Parameters

The following table lists the `Xil_L2CacheInvalidateRange` function arguments.

Table 39: **XilL2CacheInvalidateRange Arguments**

Name	Description
Addr	Address of range to be invalidated.
Len	Length in bytes to be invalidated.

#Define Xil_L1DCacheFlushRange

Description

Flush the L1 data cache for the given address range.

If the bytes specified by the address (Addr) are cached by the data cache, and is modified (dirty), the cacheline will be written to system memory.The cacheline will also be invalidated.

Parameters

The following table lists the `Xil_L1DCacheFlushRange` function arguments.

Table 40: **Xil_L1DCacheFlushRange Arguments**

Name	Description
Addr	the starting address of the range to be flushed.
Len	length in byte to be flushed.

#Define Xil_L2CacheFlushRange

Description

Flush the L2 data cache for the given address range.

If the bytes specified by the address (Addr) are cached by the data cache, and is modified (dirty), the cacheline will be written to system memory. The cacheline will also be invalidated.

Parameters

The following table lists the `Xil_L2CacheFlushRange` function arguments.

Table 41: **Xil_L2CacheFlushRange Arguments**

Name	Description
Addr	the starting address of the range to be flushed.
Len	length in byte to be flushed.

#Define Xil_L1DCacheFlush

Description

Flush the entire L1 data cache.

If any cacheline is dirty, the cacheline will be written to system memory. The entire data cache will be invalidated.

#Define Xil_L2CacheFlush

Description

Flush the entire L2 data cache.

If any cacheline is dirty, the cacheline will be written to system memory. The entire data cache will be invalidated.

#Define Xil_L1ICacheInvalidateRange

Description

Invalidate the instruction cache for the given address range.

Parameters

The following table lists the `Xil_L1ICacheInvalidateRange` function arguments.

Table 42: **Xil_L1ICacheInvalidateRange Arguments**

Name	Description
Addr	Address of ragne to be invalidated.
Len	Length in bytes to be invalidated.

#Define Xil_L1ICacheInvalidate

Description

Invalidate the entire instruction cache.

#Define Xil_L1DCacheEnable

Description

Enable the L1 data cache.

***Note*:** This is processor specific.

#Define Xil_L1DCacheDisable

Description

Disable the L1 data cache.

***Note*:** This is processor specific.

#Define Xil_L1ICacheEnable

Description

Enable the instruction cache.

***Note*:** This is processor specific.

#Define Xil_L1ICacheDisable

Description

Disable the L1 Instruction cache.

***Note*:** This is processor specific.

#Define Xil_DCacheEnable

Description

Enable the data cache.

#Define Xil_ICacheEnable

Description

Enable the instruction cache.

#Define Xil_DCacheInvalidate

Description

Invalidate the entire Data cache.

#Define Xil_DCacheInvalidateRange

Description

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.

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 43: **Xil_DCacheInvalidateRange Arguments**

Name	Description
Addr	Start address of range to be invalidated.
Len	Length of range to be invalidated in bytes.

#Define Xil_DCacheFlush

Description

Flush the entire Data cache.

#Define Xil_DCacheFlushRange

Description

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 that byte is invalidated. If the cacheline is modified (dirty), the written to system memory first before the before the line is invalidated.

Parameters

The following table lists the `Xil_DCacheFlushRange` function arguments.

Table 44: **Xil_DCacheFlushRange Arguments**

Name	Description
Addr	Start address of range to be flushed.
Len	Length of range to be flushed in bytes.

#Define Xil_ICacheInvalidate

Description

Invalidate the entire instruction cache.

#Define Xil_ICacheInvalidateRange

Description

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 cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 45: **Xil_ICacheInvalidateRange Arguments**

Name	Description
Addr	Start address of ragne to be invalidated.
Len	Length of range to be invalidated in bytes.

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.

Definitions

#Define getfslx

Description

Performs a get function on an input FSL of the MicroBlaze processor.

Parameters

The following table lists the `getfslx` function arguments.

Table 46: **getfslx Arguments**

Name	Description
val	variable to sink data from get function
id	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
flags	valid FSL macro flags

#Define putfslx

Description

Performs a put function on an input FSL of the MicroBlaze processor.

Parameters

The following table lists the `putfslx` function arguments.

Table 47: **putfslx Arguments**

Name	Description
val	variable to source data to put function
id	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
flags	valid FSL macro flags

#Define tgetfslx

Description

Performs a test get function on an input FSL of the MicroBlaze processor.

Parameters

The following table lists the `tgetfslx` function arguments.

Table 48: **tgetfslx Arguments**

Name	Description
val	variable to sink data from get function
id	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
flags	valid FSL macro flags

#Define tputfslx

Description

Performs a put function on an input FSL of the MicroBlaze processor.

Parameters

The following table lists the `tputfslx` function arguments.

Table 49: **tputfslx Arguments**

Name	Description
id	FSL identifier
flags	valid FSL macro flags

#Define getdfslx

Description

Performs a getd function on an input FSL of the MicroBlaze processor.

Parameters

The following table lists the `getdfslx` function arguments.

Table 50: **getdfslx Arguments**

Name	Description
val	variable to sink data from getd function
var	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
flags	valid FSL macro flags

#Define putdfslx

Description

Performs a putd function on an input FSL of the MicroBlaze processor.

Parameters

The following table lists the `putdfslx` function arguments.

Table 51: **putdfslx Arguments**

Name	Description
val	variable to source data to putd function
var	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
flags	valid FSL macro flags

#Define tgetdfslx

Description

Performs a test getd function on an input FSL of the MicroBlaze processor;.

Parameters

The following table lists the `tgetdfslx` function arguments.

Table 52: **tgetdfslx Arguments**

Name	Description
val	variable to sink data from getd function
var	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
flags	valid FSL macro flags

#Define tputdfslx

Description

Performs a put function on an input FSL of the MicroBlaze processor.

Parameters

The following table lists the `tputdfslx` function arguments.

Table 53: **tputdfslx Arguments**

Name	Description
var	FSL identifier
flags	valid FSL macro flags

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 54: **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 55: **microblaze_get_pvr Arguments**

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

Returns

0 - SUCCESS -1 - FAILURE

Definitions

#Define MICROBLAZE_PVR_IS_FULL

Description

Return non-zero integer if PVR is of type FULL, 0 if basic.

Parameters

The following table lists the `MICROBLAZE_PVR_IS_FULL` function arguments.

Table 56: **MICROBLAZE_PVR_IS_FULL Arguments**

Name	Description
_pvr	pvr data structure

#Define MICROBLAZE_PVR_USE_BARREL

Description

Return non-zero integer if hardware barrel shifter present.

Parameters

The following table lists the `MICROBLAZE_PVR_USE_BARREL` function arguments.

Table 57: **MICROBLAZE_PVR_USE_BARREL Arguments**

Name	Description
_pvr	pvr data structure

#Define MICROBLAZE_PVR_USE_DIV

Description

Return non-zero integer if hardware divider present.

Parameters

The following table lists the `MICROBLAZE_PVR_USE_DIV` function arguments.

Table 58: **MICROBLAZE_PVR_USE_DIV Arguments**

Name	Description
_pvr	pvr data structure

#Define MICROBLAZE_PVR_USE_HW_MUL

Description

Return non-zero integer if hardware multiplier present.

Parameters

The following table lists the `MICROBLAZE_PVR_USE_HW_MUL` function arguments.

Table 59: **MICROBLAZE_PVR_USE_HW_MUL Arguments**

Name	Description
_pvr	pvr data structure

#Define MICROBLAZE_PVR_USE_FPU

Description

Return non-zero integer if hardware floating point unit (FPU) present.

Parameters

The following table lists the `MICROBLAZE_PVR_USE_FPU` function arguments.

Table 60: **MICROBLAZE_PVR_USE_FPU Arguments**

Name	Description
_pvr	pvr data structure

#Define MICROBLAZE_PVR_USE_ICACHE

Description

Return non-zero integer if I-cache present.

Parameters

The following table lists the `MICROBLAZE_PVR_USE_ICACHE` function arguments.

Table 61: **MICROBLAZE_PVR_USE_ICACHE Arguments**

Name	Description
_pvr	pvr data structure

#Define MICROBLAZE_PVR_USE_DCACHE

Description

Return non-zero integer if D-cache present.

Parameters

The following table lists the `MICROBLAZE_PVR_USE_DCACHE` function arguments.

Table 62: **MICROBLAZE_PVR_USE_DCACHE Arguments**

Name	Description
_pvr	pvr data structure

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 63: **Quick Function Reference**

Type	Name	Arguments
u32	Xil_SetMBFrequency	u32 Val
u32	Xil_GetMBFrequency	void
void	MB_Sleep	MilliSeconds-

Functions

Xil_SetMBFrequency

Sets variable which stores Microblaze frequency value.

***Note*:** It must be called after runtime change in Microblaze frequency, failing to do so would result in to incorrect behavior of sleep routines

Prototype

```
u32 Xil_SetMBFrequency(u32 Val);
```

Parameters

The following table lists the `Xil_SetMBFrequency` function arguments.

Table 64: **Xil_SetMBFrequency Arguments**

Name	Description
Val	- Frequency value to be set

Returns

XST_SUCCESS - If frequency updated successfully XST_INVALID_PARAM - If specified frequency value is not valid

Xil_GetMBFrequency

Returns current Microblaze frequency value.

Prototype

```
u32 Xil_GetMBFrequency();
```

Returns

MBFreq - Current Microblaze frequency value

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 65: **MB_Sleep Arguments**

Name	Description
MilliSeconds-	Delay time in milliseconds.

Returns

None.

Arm Processor Common APIs

Arm Event Counters Functions

This section contain APIs for configuring and controlling the Performance Monitor Events for Arm based processors supported by standalone BSP. For more information about the event counters, see xpm_counter.h file.

It contains APIs to setup an event, return the event counter value, disable event(s), enable events, and reset event counters. It also provides a helper function: Xpm_SleepPerfCounter that is used to implement sleep routines in non-OS environment. It also contains two APIs which are being deprecated. Users are advised not to use them. On usage of these APIs, refer to xpm_conter_example.c file which is available in the standalone/examples folder.

Table 66: **Quick Function Reference**

Type	Name	Arguments
void	Xpm_SetEvents	s32 PmcrCfg
void	Xpm_GetEventCounters	u32 * PmCtrValue
u32	Xpm_DisableEvent	u32 EventCntrId
u32	Xpm_SetUpAnEvent	u32 EventID
u32	Xpm_GetEventCounter	u32 EventCntrId u32 * CntVal
void	Xpm_DisableEventCounters	void
void	Xpm_EnableEventCounters	void
void	Xpm_ResetEventCounters	void
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 67: **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 68: **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.

Prototype

```
u32 Xpm_DisableEvent(u32 EventCntrId);
```

Parameters

The following table lists the `Xpm_DisableEvent` function arguments.

Table 69: **Xpm_DisableEvent Arguments**

Name	Description
EventCntrId	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, refer xpm_counter.h. Once invoked, the API searches for an available counter. After finding one, it sets up the counter to count events for the requested event.

Prototype

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

Parameters

The following table lists the `Xpm_SetUpAnEvent` function arguments.

Table 70: **Xpm_SetUpAnEvent Arguments**

Name	Description
EventID	For valid values, 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 function reads the number of events that has been counted for the requsted event ID. This function is called after a call to Xpm_SetUpAnEvent.

Prototype

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

Parameters

The following table lists the `Xpm_GetEventCounter` function arguments.

Table 71: **Xpm_GetEventCounter Arguments**

Name	Description
EventCntrId	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.
CntVal	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);
```

Returns

None.

Xpm_EnableEventCounters

This function enables the Cortex R5 event counters.

Prototype

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

Returns

None.

Xpm_ResetEventCounters

This function resets the Cortex R5 event counters.

Prototype

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

Returns

None.

Xpm_SleepPerfCounter

This is helper function used by sleep or usleep APIs to generate delay in second or microsecond.

Prototype

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

Parameters

The following table lists the `Xpm_SleepPerfCounter` function arguments.

Table 72: **Xpm_SleepPerfCounter Arguments**

Name	Description
delay	Delay time in second or microsecond
frequency	Number of counts in second or microseconds

Returns

None.

Arm Processor Exception Handling

Arm processors specific exception related APIs for Cortex-A53, Cortex-A9, and Cortex-R5F are used for enabling or disabling IRQ, registering or removing handler for exceptions, or initializing exception vector table with the null handler.

Table 73: **Quick Function Reference**

Type	Name	Arguments
void	Xil_ExceptionRegisterHandler	u32 Exception_id Xil_ExceptionHandler Handler void * Data
void	Xil_ExceptionRemoveHandler	u32 Exception_id
void	Xil_GetExceptionRegisterHandler	u32 Exception_id Xil_ExceptionHandler * Handler void ** Data
void	Xil_ExceptionInit	void
void	Xil_DataAbortHandler	void
void	Xil_PrefetchAbortHandler	void
void	Xil_UndefinedExceptionHandler	void

Functions

Xil_ExceptionRegisterHandler

Registers a handler for a specific exception. This handler is being called when the processor encounters the specified exception.

Prototype

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

Parameters

The following table lists the `Xil_ExceptionRegisterHandler` function arguments.

Table 74: **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	Handler for the exception.
Data	Reference to the data that is 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.

Prototype

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

Parameters

The following table lists the `Xil_ExceptionRemoveHandler` function arguments.

Table 75: **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.

Prototype

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

Parameters

The following table lists the `Xil_GetExceptionRegisterHandler` function arguments.

Table 76: **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	Handler for that exception.

Table 76: **Xil_GetExceptionRegisterHandler Arguments** *(cont'd)*

Name	Description
Data	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).

Prototype

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

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);
```

Returns

None.

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);
```

Returns

None.

Xil_UndefinedExceptionHandler

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

Prototype

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

Returns

None.

Definitions

Define Xil_ExceptionEnableMask

Definition

```
#define Xil_ExceptionEnableMask{                                    \
          register u32 Reg __asm("cpsr"); \
          mtcpsr((Reg) & (~((Mask) & XIL_EXCEPTION_ALL))); \
        }
```

Description

Enable Exceptions.

Note: If bit is 0, exception is enabled. C-Style signature: void `Xil_ExceptionEnableMask(Mask)`

Define Xil_ExceptionEnable

Definition

```
#define Xil_ExceptionEnable

              Xil_ExceptionEnableMask
            (XIL_EXCEPTION_IRQ)
```

Description

Enable the IRQ exception.

Note: None.

Define Xil_ExceptionDisableMask

Definition

```
#define Xil_ExceptionDisableMask{                                       \
          register u32 Reg __asm("cpsr"); \
          mtcpsr((Reg) | ((Mask) & XIL_EXCEPTION_ALL)); \
        }
```

Description

Disable Exceptions.

Note: If bit is 1, exception is disabled. C-Style signature: `Xil_ExceptionDisableMask(Mask)`

Define Xil_ExceptionDisable

Definition

```
#define Xil_ExceptionDisable

              Xil_ExceptionDisableMask
            (XIL_EXCEPTION_IRQ)
```

Description

Disable the IRQ exception.

Note: None.

Define Xil_EnableNestedInterrupts

Definition

```
#define Xil_EnableNestedInterrupts__asm__ __volatile__ ("stmfd   sp!,
{lr}"); \
        __asm__ __volatile__ ("mrs     lr, spsr");  \
        __asm__ __volatile__ ("stmfd   sp!, {lr}"); \
        __asm__ __volatile__ ("msr     cpsr_c, #0x1F"); \
        __asm__ __volatile__ ("stmfd   sp!, {lr}");
```

Description

Enable nested interrupts by clearing the I and F bits in CPSR.

This API is defined for cortex-a9 and cortex-r5.

***Note*:** This macro is supposed to be used from interrupt handlers. In the interrupt handler the interrupts are disabled by default (I and F are 1). To allow nesting of interrupts, this macro should be used. It clears the I and F bits by changing the ARM mode to system mode. Once these bits are cleared and provided the preemption of interrupt conditions are met in the GIC, nesting of interrupts will start happening. Caution: This macro must be used with caution. Before calling this macro, the user must ensure that the source of the current IRQ is appropriately cleared. Otherwise, as soon as we clear the I and F bits, there can be an infinite loop of interrupts with an eventual crash (all the stack space getting consumed).

Define Xil_DisableNestedInterrupts

Definition

```
#define Xil_DisableNestedInterrupts __asm__ __volatile__ ("ldmfd   sp!,
{lr}");   \
        __asm__ __volatile__ ("msr     cpsr_c, #0x92"); \
        __asm__ __volatile__ ("ldmfd   sp!, {lr}"); \
        __asm__ __volatile__ ("msr     spsr_cxsf, lr"); \
        __asm__ __volatile__ ("ldmfd   sp!, {lr}"); \
```

Description

Disable the nested interrupts by setting the I and F bits.

This API is defined for cortex-a9 and cortex-r5.

***Note*:** This macro is meant to be called in the interrupt service routines. This macro cannot be used independently. It can only be used when nesting of interrupts have been enabled by using the macro `Xil_EnableNestedInterrupts()` . In a typical flow, the user first calls the Xil_EnableNestedInterrupts in the ISR at the appropriate point. The user then must call this macro before exiting the interrupt service routine. This macro puts the ARM back in IRQ/FIQ mode and hence sets back the I and F bits.

Arm Cortex-R5F Processor APIs

Arm Cortex-R5F 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 Arm Cortex-R5F processor APIs.

Arm Cortex-R5F Processor Boot Code

The boot.S file contains a minimal set of code for transferring control from the processor reset location of the processor to the start of the application. The boot code performs minimum configuration which is required for an application to run starting from reset state of the processor. 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

Arm Cortex-R5F 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 77: **Quick Function Reference**

Type	Name	Arguments
void	Xil_SetTlbAttributes	INTPTR addr u32 attrib
void	Xil_EnableMPU	void
void	Xil_DisableMPU	void
u32	Xil_SetMPURegion	INTPTR 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	void
u32	Xil_GetNextMPURegion	void
u32	Xil_DisableMPURegionByRegNum	u32 reg_num
u16	Xil_GetMPUFreeRegMask	void
u32	Xil_SetMPURegionByRegNum	u32 reg_num INTPTR addr u64 size u32 attrib
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.

Prototype

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

Parameters

The following table lists the `Xil_SetTlbAttributes` function arguments.

Table 78: **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);
```

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);
```

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 79: **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 80: **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 81: **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);
```

Returns

Number of free regions available to users

Xil_GetNextMPURegion

Returns the next available free MPU region.

Prototype

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

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 82: **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);
```

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 83: **Xil_SetMPURegionByRegNum Arguments**

Name	Description
reg_num	The region number to be enabled
addr	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.

Xil_MemMap

Memory mapping for Cortex-R5F.

If successful, the mapped region will include all of the memory requested, but may include more. Specifically, it will be a power of 2 in size, aligned on a boundary of that size.

Prototype

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

Parameters

The following table lists the `Xil_MemMap` function arguments.

Table 84: **Xil_MemMap Arguments**

Name	Description
Physaddr	is base physical address at which to start mapping. NULL in Physaddr masks possible mapping errors.
size	of region to be mapped.
flags	used to set translation table.

Returns

Physaddr on success, NULL on error. Ambiguous if Physaddr==NULL

Arm Cortex-R5F 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 85: **Quick Function Reference**

Type	Name	Arguments
void	Xil_DCacheEnable	void
void	Xil_DCacheDisable	void
void	Xil_DCacheInvalidate	void
void	Xil_DCacheInvalidateRange	INTPTR adr u32 len
void	Xil_DCacheFlush	void
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	void
void	Xil_ICacheDisable	void
void	Xil_ICacheInvalidate	void
void	Xil_ICacheInvalidateRange	INTPTR adr u32 len
void	Xil_ICacheInvalidateLine	INTPTR adr

Functions

Xil_DCacheEnable

Enable the Data cache.

Prototype

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

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Prototype

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

Returns

None.

Xil_DCacheInvalidate

Invalidate the entire Data cache.

Prototype

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

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 86: **Xil_DCacheInvalidateRange Arguments**

Name	Description
adr	32-bit 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);
```

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 87: **Xil_DCacheFlushRange Arguments**

Name	Description
adr	32-bit 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 88: **Xil_DCacheInvalidateLine Arguments**

Name	Description
adr	32-bit 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 89: **Xil_DCacheFlushLine Arguments**

Name	Description
adr	32-bit 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 90: **Xil_DCacheStoreLine Arguments**

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

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

Prototype

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

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Prototype

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

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

Prototype

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

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 91: **Xil_ICacheInvalidateRange Arguments**

Name	Description
adr	32-bit 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 92: **Xil_ICacheInvalidateLine Arguments**

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

Returns

None.

Arm Cortex-R5F 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 93: **Quick Function Reference**

Type	Name	Arguments
void	XTime_SetTime	XTime Xtime_Global

Table 93: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
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 94: **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 95: **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.

Arm Cortex-R5F 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 Cortex-A9 Processor APIs

Arm 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.

Arm Cortex-A9 Processor Boot Code

The boot code performs minimum configuration which is required for an application to run starting from processor reset state of the processor. 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*:**

The 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 1 MB of memory.

The overview of translation table memory attributes is described below.

For region 0x00000000 - 0x3FFFFFFF, a system where DDR is less than 1 GB, 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 1 MB, 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.

	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

	Memory Range	Definition in Translation Table
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

Arm 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 96: **Quick Function Reference**

Type	Name	Arguments
void	Xil_DCacheEnable	void
void	Xil_DCacheDisable	void
void	Xil_DCacheInvalidate	void
void	Xil_DCacheInvalidateRange	INTPTR adr u32 len
void	Xil_DCacheFlush	void
void	Xil_DCacheFlushRange	INTPTR adr u32 len
void	Xil_ICacheEnable	void
void	Xil_ICacheDisable	void
void	Xil_ICacheInvalidate	void
void	Xil_ICacheInvalidateRange	INTPTR adr u32 len
void	Xil_DCacheInvalidateLine	u32 adr

Table 96: **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	void
void	Xil_L1DCacheDisable	void
void	Xil_L1DCacheInvalidate	void
void	Xil_L1DCacheInvalidateLine	u32 adr
void	Xil_L1DCacheInvalidateRange	u32 adr u32 len
void	Xil_L1DCacheFlush	void
void	Xil_L1DCacheFlushLine	u32 adr
void	Xil_L1DCacheFlushRange	u32 adr u32 len
void	Xil_L1DCacheStoreLine	u32 adr
void	Xil_L1ICacheEnable	void
void	Xil_L1ICacheDisable	void
void	Xil_L1ICacheInvalidate	void
void	Xil_L1ICacheInvalidateLine	u32 adr
void	Xil_L1ICacheInvalidateRange	u32 adr u32 len
void	Xil_L2CacheEnable	void

Table 96: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	Xil_L2CacheDisable	void
void	Xil_L2CacheInvalidate	void
void	Xil_L2CacheInvalidateLine	u32 adr
void	Xil_L2CacheInvalidateRange	u32 adr u32 len
void	Xil_L2CacheFlush	void
void	Xil_L2CacheFlushLine	u32 adr
void	Xil_L2CacheFlushRange	u32 adr u32 len
void	Xil_L2CacheStoreLine	u32 adr

Functions

Xil_DCacheEnable

Enable the Data cache.

Prototype

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

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Prototype

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

Returns

None.

Xil_DCacheInvalidate

Invalidate the entire Data cache.

Prototype

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

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.

In this function, if start address or end address is not aligned to cache-line, particular cache-line containing unaligned start or end address is flush first and then invalidated the others as invalidating the same unaligned cache line may result into loss of data. This issue raises few possibilities.

If the address to be invalidated is not cache-line aligned, the following choices are available:

1. Invalidate the cache line when required and do not bother much for the side effects. Though it sounds good, it can result in hard-to-debug issues. The problem is, if some other variable are allocated in the same cache line and had been recently updated (in cache), the invalidation would result in loss of data.
2. Flush the cache line first. This will ensure that if any other variable present in the same cache line and updated recently are flushed out to memory. Then it can safely be invalidated. Again it sounds good, but this can result in issues. For example, when the invalidation happens in a typical ISR (after a DMA transfer has updated the memory), then flushing the cache line means, losing data that were updated recently before the ISR got invoked.

Linux prefers the second one. To have uniform implementation (across standalone and Linux), the second option is implemented. This being the case, following needs to be taken care of:

1. Whenever possible, the addresses must be cache line aligned. Please nore that, not just start address, even the end address must be cache line aligned. If that is taken care of, this will always work.

2. 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.

This is going to always work if done carefully. However, the concern is, there is no guarantee that invalidate has not needed to be done after DMA is complete. For example, because of some reasons if the first cache line or last cache line (assuming the buffer in question comprises of multiple cache lines) 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.

Prototype

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

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 97: **Xil_DCacheInvalidateRange Arguments**

Name	Description
adr	32-bit 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.

Prototype

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

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.

Prototype

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

Parameters

The following table lists the `Xil_DCacheFlushRange` function arguments.

Table 98: **Xil_DCacheFlushRange Arguments**

Name	Description
adr	32-bit 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.

Prototype

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

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Prototype

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

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

Prototype

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

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.

Prototype

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

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 99: **Xil_ICacheInvalidateRange Arguments**

Name	Description
adr	32-bit 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 100: **Xil_DCacheInvalidateLine Arguments**

Name	Description
adr	32-bit 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 101: **Xil_DCacheFlushLine Arguments**

Name	Description
adr	32-bit 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 102: **Xil_DCacheStoreLine Arguments**

Name	Description
adr	32-bit 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 103: **Xil_ICacheInvalidateLine Arguments**

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

Returns

None.

Xil_L1DCacheEnable

Enable the level 1 Data cache.

Prototype

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

Returns

None.

Xil_L1DCacheDisable

Disable the level 1 Data cache.

Prototype

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

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);
```

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 104: **Xil_L1DCacheInvalidateLine Arguments**

Name	Description
adr	32-bit 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.

Prototype

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

Parameters

The following table lists the `Xil_L1DCacheInvalidateRange` function arguments.

Table 105: **Xil_L1DCacheInvalidateRange Arguments**

Name	Description
adr	32-bit 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);
```

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 106: **Xil_L1DCacheFlushLine Arguments**

Name	Description
adr	32-bit 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.

Prototype

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

Parameters

The following table lists the `Xil_L1DCacheFlushRange` function arguments.

Table 107: **Xil_L1DCacheFlushRange Arguments**

Name	Description
adr	32-bit 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 108: **Xil_L1DCacheStoreLine Arguments**

Name	Description
adr	Address to be stored.

Returns

None.

Xil_L1ICacheEnable

Enable the level 1 instruction cache.

Prototype

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

Returns

None.

Xil_L1ICacheDisable

Disable level 1 the instruction cache.

Prototype

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

Returns

None.

Xil_L1ICacheInvalidate

Invalidate the entire level 1 instruction cache.

Prototype

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

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 109: **Xil_L1ICacheInvalidateLine Arguments**

Name	Description
adr	32-bit 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.

Prototype

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

Parameters

The following table lists the `Xil_L1ICacheInvalidateRange` function arguments.

Table 110: **Xil_L1ICacheInvalidateRange Arguments**

Name	Description
adr	32-bit 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.

Prototype

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

Returns

None.

Xil_L2CacheDisable

Disable the L2 cache.

Prototype

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

Returns

None.

Xil_L2CacheInvalidate

Invalidate the entire level 2 cache.

Prototype

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

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 111: **Xil_L2CacheInvalidateLine Arguments**

Name	Description
adr	32-bit 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.

Prototype

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

Parameters

The following table lists the `Xil_L2CacheInvalidateRange` function arguments.

Table 112: **Xil_L2CacheInvalidateRange Arguments**

Name	Description
adr	32-bit 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.

Prototype

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

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 113: **Xil_L2CacheFlushLine Arguments**

Name	Description
adr	32-bit 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.

Prototype

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

Parameters

The following table lists the `Xil_L2CacheFlushRange` function arguments.

Table 114: **Xil_L2CacheFlushRange Arguments**

Name	Description
adr	32-bit 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 115: **Xil_L2CacheStoreLine Arguments**

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

Returns

None.

Arm 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 116: **Quick Function Reference**

Type	Name	Arguments
void	Xil_SetTlbAttributes	INTPTR Addr u32 attrib
void	Xil_EnableMMU	void
void	Xil_DisableMMU	void
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 117: **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);
```

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);
```

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 118: **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.

Arm 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 119: **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 120: **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 121: **XTime_GetTime Arguments**

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

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 122: **Quick Function Reference**

Type	Name	Arguments
void	XL2cc_EventCtrInit	s32 Event0 s32 Event1
void	XL2cc_EventCtrStart	void
void	XL2cc_EventCtrStop	u32 * EveCtr0 u32 * EveCtr1

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 123: **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.

Prototype

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

Returns

None.

XL2cc_EventCtrStop

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

Prototype

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

Parameters

The following table lists the `XL2cc_EventCtrStop` function arguments.

Table 124: **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.

Arm 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.

Definitions

Define CONFIG_ARM_ERRATA_775420

Definition

```
#define CONFIG_ARM_ERRATA_7754201
```

Description

Errata No: 775420 Description: A data cache maintenance operation which aborts, might lead to deadlock.

Define CONFIG_ARM_ERRATA_794073

Definition

```
#define CONFIG_ARM_ERRATA_7940731
```

Description

Errata No: 794073 Description: Speculative instruction fetches with MMU disabled might not comply with architectural requirements.

Define CONFIG_PL310_ERRATA_588369

Definition

```
#define CONFIG_PL310_ERRATA_5883691
```

Description

PL310 L2 Cache Errata.

Errata No: 588369 Description: Clean & Invalidate maintenance operations do not invalidate clean lines

Define CONFIG_PL310_ERRATA_727915

Definition

```
#define CONFIG_PL310_ERRATA_7279151
```

Description

Errata No: 727915 Description: Background Clean and Invalidate by Way operation can cause data corruption.

Arm 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.

Arm Cortex-A53 32-bit Processor APIs

Arm 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.

Arm Cortex-A53 32-bit Processor Boot Code

The boot.S file contains a minimal set of code for transferring control from the processor reset location to the start of the application. The boot code performs minimum configuration which is required for an application to run starting from processor reset state of the processor. 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

The translation_table.S contains a static page table required by MMU for cortex-A53. This translation table is flat mapped (input address = output address) with default memory attributes defined for zynq ultrascale+ architecture. It utilizes short descriptor translation table format with each section defining 1MB of memory.

For DDR in region 0x00000000 - 0x7FFFFFFF, a system where DDR is less than 2 GB, region after DDR and before PL is marked as undefined/reserved in translation table. In region 0xFFC00000 - 0xFFDFFFFF, it contains CSU and PMU memory which are marked as Device since it is less than 1 MB and falls in a region with device memory.

The overview of translation table memory attributes is described below.

	Memory Range	Definition in Translation Table
DDR	0x00000000 - 0x7FFFFFFF	Normal write-back Cacheable
PL	0x80000000 - 0xBFFFFFFF	Strongly Ordered
QSPI, lower PCIe	0xC0000000 - 0xEFFFFFFF	Device Memory
Reserved	0xF0000000 - 0xF7FFFFFF	Unassigned
STM Coresight	0xF8000000 - 0xF8FFFFFF	Device Memory
GIC	0xF9000000 - 0xF90FFFFF	Device memory
Reserved	0xF9100000 - 0xFCFFFFFF	Unassigned
FPS, LPS slaves	0xFD000000 - 0xFFBFFFFF	Device memory
CSU, PMU	0xFFC00000 - 0xFFDFFFFF	Device Memory
TCM, OCM	0xFFE00000 - 0xFFFFFFFF	Normal write-back cacheable

Arm 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 125: **Quick Function Reference**

Type	Name	Arguments
void	Xil_DCacheEnable	void
void	Xil_DCacheDisable	void
void	Xil_DCacheInvalidate	void
void	Xil_DCacheInvalidateRange	INTPTR adr u32 len

Table 125: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	Xil_DCacheFlush	void
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	void
void	Xil_ICacheDisable	void
void	Xil_ICacheInvalidate	void
void	Xil_ICacheInvalidateRange	INTPTR adr u32 len

Functions

Xil_DCacheEnable

Enable the Data cache.

Prototype

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

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Prototype

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

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);
```

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 to avoid such corruption.

Prototype

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

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 126: **Xil_DCacheInvalidateRange Arguments**

Name	Description
adr	32-bit 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.

Prototype

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

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.

Prototype

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

Parameters

The following table lists the `Xil_DCacheFlushRange` function arguments.

Table 127: **Xil_DCacheFlushRange Arguments**

Name	Description
adr	32-bit 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 128: **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.

***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 129: **Xil_DCacheFlushLine Arguments**

Name	Description
adr	32-bit 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.

***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 130: **Xil_ICacheInvalidateLine Arguments**

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

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

Prototype

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

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Prototype

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

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

Prototype

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

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.

Prototype

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

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 131: **Xil_ICacheInvalidateRange Arguments**

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

Returns

None.

Arm 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.

Table 132: **Quick Function Reference**

Type	Name	Arguments
void	Xil_SetTlbAttributes	UINTPTR Addr u32 attrib
void	Xil_EnableMMU	void
void	Xil_DisableMMU	void

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 133: **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 32-bit mode.

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

Prototype

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

Returns

None.

Xil_DisableMMU

Disable MMU for Cortex A53 processors in 32-bit 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);
```

Returns

None.

Arm Cortex-A53 32-bit Mode Time Functions

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

Table 134: **Quick Function Reference**

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

Functions

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.

Prototype

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

Parameters

The following table lists the `XTime_SetTime` function arguments.

Table 135: **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.

Prototype

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

Parameters

The following table lists the `XTime_GetTime` function arguments.

Table 136: **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.

Arm 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.

Arm Cortex-A53 64-bit Processor APIs

Arm Cortex-A53 64-bit Processor API

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 Arm Cortex-A53 64-bit Processor APIs.

***Note*:** These APIs are applicable for the Cortex-A72 processor as well.

Arm Cortex-A53 64-bit Processor Boot Code

The boot code performs minimum configuration which is required for an application. Cortex-A53 starts by checking current exception level. If the current exception level is EL3 and BSP is built for EL3, it will do initialization required for application execution at EL3. Below is a sequence illustrating what all configuration is performed before control reaches to main function for EL3 execution.

1. Program vector table base for exception handling
2. Set reset vector table base address
3. Program stack pointer for EL3
4. Routing of interrupts to EL3
5. Enable ECC protection
6. Program generic counter frequency
7. Invalidate instruction cache, data cache and TLBs
8. Configure MMU registers and program base address of translation table
9. Transfer control to _start which clears BSS sections and runs global constructor before jumping to main application

If the current exception level is EL1 and BSP is also built for EL1_NONSECURE it will perform initialization required for application execution at EL1 non-secure. For all other combination, the execution will go into infinite loop. Below is a sequence illustrating what all configuration is performed before control reaches to main function for EL1 execution.

1. Program vector table base for exception handling
2. Program stack pointer for EL1
3. Invalidate instruction cache, data cache and TLBs
4. Configure MMU registers and program base address of translation table
5. Transfer control to _start which clears BSS sections and runs global constructor before jumping to main application

Arm 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 137: **Quick Function Reference**

Type	Name	Arguments
void	Xil_DCacheEnable	void
void	Xil_DCacheDisable	void
void	Xil_DCacheInvalidate	void
void	Xil_DCacheInvalidateRange	INTPTR adr INTPTR len
void	Xil_DCacheInvalidateLine	INTPTR adr
void	Xil_DCacheFlush	void
void	Xil_DCacheFlushLine	INTPTR adr
void	Xil_ICacheEnable	void
void	Xil_ICacheDisable	void
void	Xil_ICacheInvalidate	void
void	Xil_ICacheInvalidateRange	INTPTR adr INTPTR len

Table 137: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	Xil_ICacheInvalidateLine	INTPTR adr
void	Xil_ConfigureL1Prefetch	u8 num

Functions

Xil_DCacheEnable

Enable the Data cache.

Prototype

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

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Prototype

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

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);
```

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 138: **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 139: **Xil_DCacheInvalidateLine Arguments**

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

Returns

None.

Xil_DCacheFlush

Flush the Data cache.

Prototype

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

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 140: **Xil_DCacheFlushLine Arguments**

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

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

Prototype

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

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Prototype

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

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

Prototype

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

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.

Prototype

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

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 141: **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 142: **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 143: **Xil_ConfigureL1Prefetch Arguments**

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

Returns

None.

Arm 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 144: **Quick Function Reference**

Type	Name	Arguments
void	Xil_SetTlbAttributes	UINTPTR Addr u64 attrib

Functions

Xil_SetTlbAttributes

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 145: **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.

Arm Cortex-A53 64-bit Mode Time Functions

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

Table 146: **Quick Function Reference**

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

Functions

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.

Prototype

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

Parameters

The following table lists the `XTime_SetTime` function arguments.

Table 147: **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.

Prototype

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

Parameters

The following table lists the `XTime_GetTime` function arguments.

Table 148: **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.

Arm 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.

LwIP 2.1.3 Library v1.0

Introduction

The lwIP is an open source TCP/IP protocol suite available under the BSD license. The lwIP is a standalone stack with no operating system dependencies but can be used along with the operating systems. The lwIP library provides the following two APIs for use by the applications.

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

The lwip213 is built on the open source lwIP library version 2.1.3. The lwip213 library provides adapters for the Ethernetlite (axi_ethernetlite), TEMAC (axi_ethernet), and Gigabit Ethernet controller and MAC (GigE) cores. The library runs on MicroBlaze™, Arm Cortex-A9, Cortex-R5, and Cortex-A53, Cortex-A72, and Cortex-R5F processors. The Ethernetlite and TEMAC cores apply for MicroBlaze systems. The Gigabit Ethernet controller and MAC (GigE) core is applicable only for the following:

- Cortex-A9 system (AMD Zynq 7000 processor devices)
- Cortex-A53 and Cortex-R5 system (AMD Zynq UltraScale+ MPSoC)
- Cortex-A72 and Cortex-R5Fsystem (AMD Versal adaptive SoC)

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
- lwIP wiki: http://lwip.scribblewiki.com
- AMD lwIP designs and application examples: LightWeight IP Application Examples (XAPP1026)
- lwIP examples using RAW and Socket APIs: http://savannah.nongnu.org/projects/lwip/
- FreeRTOS Port for Zynq 7000 is available for download from the [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 lwip213 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 7000) 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 systems and must be included for using lwIP socket API. The FreeRTOS BSP for Zynq 7000, Zynq UltraScale+ MPSoC, and Versal devices is available for download from the the [FreeRTOS][http://www.freertos.org] website.

Setting up the Hardware System

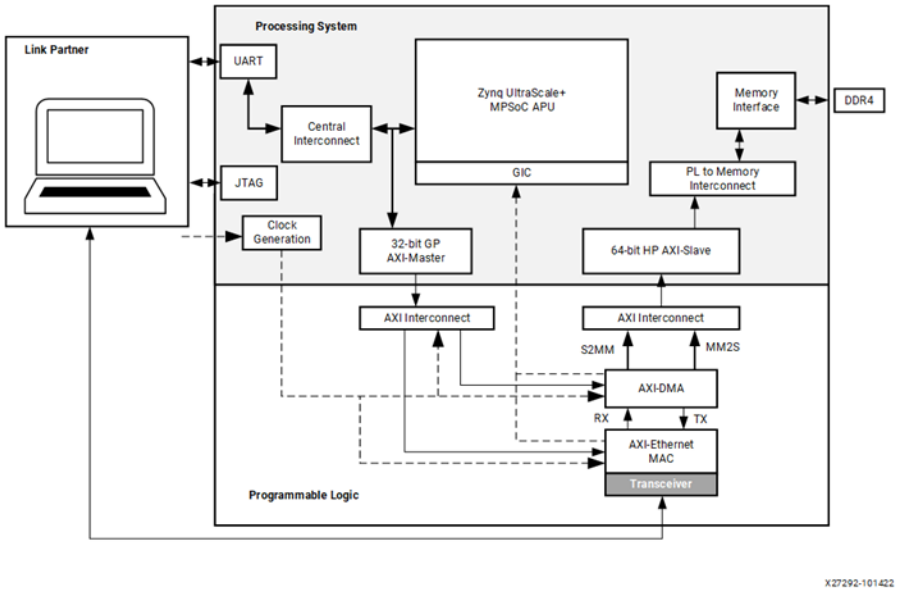
This section describes the hardware configurations supported by lwIP. The key components of the hardware system include:

- **Processor**: MicroBlaze, Cortex-A9, Cortex-A53, or Cortex-R5 processor. The Cortex-A9 processor applies to Zynq systems. The Cortex-A53 and Cortex-R5 processors apply to Zynq UltraScale+ MPSoC systems. The Cortex-A72 and Cortex-R5F processors apply to Versal adaptive SoC systems.
- **MAC**: LwIP supports axi_ethernetlite, axi_ethernet, and Gigabit Ethernet controller and MAC (GigE) cores.

- **Timer**: To maintain TCP timers, lwIP use raw API based applications. It requires periodic calling of certain functions by the application. An application achieve this by registering an interrupt handler with a 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. There is a built-in DMA for GigE-based Zynq, Zynq UltraScale+ MPSoC, and Versal adaptive SoC systems which does not require extra configuration. 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 Zynq Ultrascale+ MPSoC device utilizing the axi_ethernet core with DMA.

Figure 1: **AXI Ethernet subsystem with DMA on Zynq Ultrascale+ MPSoC**



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.

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 **lwip213_v1.0** library from the list of Supported Libraries.
15. Expand the Overview tree and select lwip213_v1.0. The configuration options for the lwip213_v1.0 library are listed.
16. Configure the **lwip213_v1.0** library and click OK.

Configuring lwIP Options

The lwIP library provides configurable parameters. The following 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 lwip213 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 lwip213 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.

Table 149: **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, Zynq UltraScale+ MPSoC, and Versal systems using FreeRTOS, appropriate priority should be set. The default priority of 1 will not give the expected behavior. For FreeRTOS (Zynq 7000, Zynq UltraScale+ MPSoC, and Versal adative SoC 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. 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 AMD provided lwIP adapters are not tested for multiple MACs. Multiple Axi Ethernet's are not supported on Zynq UltraScale+ MPSoCs.	integer	0 = Use Zynq 7000 PS-based or Zynq UltraScale+ MPSoC 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 devices, Zynq UltraScale+ MPSoC, and Versal device, 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 devices, Zynq UltraScale+ MPSoC, and Versal device), 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 devices, Zynq UltraScale+ MPSoC, and Versal ACAP, the TCP and IP receive checksum offloading is always present, so this attribute does not apply. Default is false.
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, Zynq UltraScale+ MPSoC, and Versal ACAP, 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 the AMD 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 devices, 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.
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, for example 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.
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.

Attribute	Default	Type	Description
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 7000 and Zynq Ultrascale+ 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.

Attribute	Default	Type	Description
udp_tx_blocking	FALSE	Boolean	When enabled, the application sends UDP packet blocks till the packet is transmitted.Useful for zero-copy UDP transmission use cases where the application might be using the same buffer to send information out for consecutive UDP packets. This option, when turned on, ensures that the application gets to use the TX buffer only after the buffer is used by the adapter and MAC.

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.
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.
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/lwip213/src/lwip-2.1.3/doc/rawapi.txt` file describes the lwIP Raw API.

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 7000, Zynq UltraScale+ MPSoC, and Versal adatpive SoC) 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 150: **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)
```

xemac_add

The `xemac_add()` function provides a unified interface to add any AMD 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 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);
```

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, Zynq UltraScale+ MPSoC, Versal, and the GigE controller.

Prototype

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

Chapter 5

XilFlash Library v4.9

Overview

The XilFlash library provides read/write/erase/lock/unlock features to access a parallel flash device.

This library implements the functionality for flash memory devices that conform to the "Common Flash Interface" (CFI) standard. CFI allows a single flash library to be used for an entire family of parts and helps us determine the algorithm to utilize during runtime.

***Note*:** All the calls in the library are blocking in nature in that the control is returned back to user only after the current operation is completed successfully or an error is reported.

Library Initialization

The `XFlash_Initialize()` function should be called by the application before any other function in the library. The initialization function checks for the device family and initializes the XFlash instance with the family specific data. The VT table (contains the function pointers to family specific APIs) is setup and family specific initialization routine is called.

***Note*:** The XilFlash library is not supported by Zynq Ultrascale+ MPSoC and Versal devices.

Device Geometry

The device geometry varies for different flash device families.

Following sections describes the geometry of different flash device families:

Intel Flash Device Geometry

Flash memory space is segmented into areas called blocks. The size of each block is based on a power of 2. A region is defined as a contiguous set of blocks of the same size. Some parts have several regions while others have one. The arrangement of blocks and regions is referred to by this module as the part's geometry. Some Intel flash supports multiple banks on the same device. This library supports single and multiple bank flash devices.

AMD Flash Device Geometry

Flash memory space is segmented into areas called banks and further in to regions and blocks. The size of each block is based on a power of 2. A region is defined as a contiguous set of blocks of the same size. Some parts have several regions while others have one. A bank is defined as a contiguous set of blocks. The bank may contain blocks of different size. The arrangement of blocks, regions and banks is referred to by this module as the part's geometry. The cells within the part can be programmed from a logic 1 to a logic 0 and not the other way around. To change a cell back to a logic 1, the entire block containing that cell must be erased. When a block is erased all bytes contain the value 0xFF. The number of times a block can be erased is finite. Eventually the block will wear out and will no longer be capable of erasure. As of this writing, the typical flash block can be erased 100,000 or more times.

Write Operation

The write call can be used to write a minimum of zero bytes and a maximum entire flash. If the Offset Address specified to write is out of flash or if the number of bytes specified from the Offset address exceed flash boundaries an error is reported back to the user. The write is blocking in nature in that the control is returned back to user only after the write operation is completed successfully or an error is reported.

Read Operation

The read call can be used to read a minimum of zero bytes and maximum of entire flash. If the Offset Address specified to write is out of flash boundary an error is reported back to the user. The read function reads memory locations beyond Flash boundary. Care should be taken by the user to make sure that the Number of Bytes + Offset address is within the Flash address boundaries. The read is blocking in nature in that the control is returned back to user only after the read operation is completed successfully or an error is reported.

Erase Operation

The erase operations are provided to erase a Block in the Flash memory. The erase call is blocking in nature in that the control is returned back to user only after the erase operation is completed successfully or an error is reported.

Sector Protection

The Flash Device is divided into Blocks. Each Block can be protected individually from unwarranted writing/erasing. The Block locking can be achieved using `XFlash_Lock()` lock. All the memory locations from the Offset address specified will be locked. The block can be unlocked using XFlash_UnLock() call. All the Blocks which are previously locked will be unlocked. The Lock and Unlock calls are blocking in nature in that the control is returned back to user only after the operation is completed successfully or an error is reported. The AMD flash device requires high voltage on Reset pin to perform lock and unlock operation. User must provide this high voltage (As defined in datasheet) to reset pin before calling lock and unlock API for AMD flash devices. Lock and Unlock features are not tested for AMD flash device.

Device Control

Functionalities specific to a Flash Device Family are implemented as Device Control. The following are the Intel specific device control:

- Retrieve the last error data.
- Get Device geometry.
- Get Device properties.
- Set RYBY pin mode.
- Set the Configuration register (Platform Flash only).

The following are the AMD specific device control:

- Get Device geometry.
- Get Device properties.
- Erase Resume.
- Erase Suspend.
- Enter Extended Mode.
- Exit Extended Mode.
- Get Protection Status of Block Group.
- Erase Chip.

***Note*:** This library needs to know the type of EMC core (AXI or XPS) used to access the cfi flash, to map the correct APIs. This library should be used with the emc driver, v3_01_a and above, so that this information can be automatically obtained from the emc driver.

This library is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads, mutual exclusion, virtual memory, cache control, or HW write protection management must be satisfied by the layer above this library. All writes to flash occur in units of bus-width bytes. If more than one part exists on the data bus, then the parts are written in parallel. Reads from flash are performed in any width up to the width of the data bus. It is assumed that the flash bus controller or local bus supports these types of accesses.

XilFlash Library API

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

Table 151: **Quick Function Reference**

Type	Name	Arguments
int	XFlash_Initialize	XFlash * InstancePtr u32 BaseAddress u8 BusWidth int IsPlatformFlash
int	XFlash_Reset	XFlash * InstancePtr
int	XFlash_DeviceControl	XFlash * InstancePtr u32 Command DeviceCtrlParam * Parameters
int	XFlash_Read	XFlash * InstancePtr u32 Offset u32 Bytes void * DestPtr
int	XFlash_Write	XFlash * InstancePtr u32 Offset u32 Bytes void * SrcPtr
int	XFlash_Erase	XFlash * InstancePtr u32 Offset u32 Bytes
int	XFlash_Lock	XFlash * InstancePtr u32 Offset u32 Bytes

Table 151: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
int	XFlash_Unlock	XFlash * InstancePtr u32 Offset u32 Bytes
int	XFlash_IsReady	XFlash * InstancePtr

Functions

XFlash_Initialize

This function initializes a specific XFlash instance.

The initialization entails:

- Check the Device family type.
- Issuing the CFI query command.
- Get and translate relevant CFI query information.
- Set default options for the instance.
- Setup the VTable.
- Call the family initialize function of the instance.

Initialize the AMD Platform Flash XL to Async mode if the user selects to use the Platform Flash XL in the MLD. The Platform Flash XL is an Intel CFI complaint device.

- XFLASH_PART_NOT_SUPPORTED if the command set algorithm or Layout is not supported by any flash family compiled into the system.
- XFLASH_CFI_QUERY_ERROR if the device would not enter CFI query mode. Either the device(s) do not support CFI, the wrong BaseAddress param was used, an unsupported part layout exists, or a hardware problem exists with the part.

***Note*:** BusWidth is not the width of an individual part. Its the total operating width. For example, if there are two 16-bit parts, with one tied to data lines D0-D15 and other tied to D15-D31, BusWidth would be (32 / 8) = 4. If a single 16-bit flash is in 8-bit mode, then BusWidth should be (8 / 8) = 1.

Prototype

```
int XFlash_Initialize(XFlash *InstancePtr, u32 BaseAddress, u8 BusWidth, int
IsPlatformFlash);
```

Parameters

The following table lists the `XFlash_Initialize` function arguments.

Table 152: **XFlash_Initialize Arguments**

Type	Name	Description
XFlash *	InstancePtr	Pointer to the XFlash instance.
u32	BaseAddress	Base address of the flash memory.
u8	BusWidth	Total width of the flash memory, in bytes
int	IsPlatformFlash	Used to specify if the flash is a platform flash.

Returns

- XST_SUCCESS if successful.

XFlash_Reset

This function resets the flash device and places it in read mode.

***Note*:** None.

Prototype

```
int XFlash_Reset(XFlash *InstancePtr);
```

Parameters

The following table lists the `XFlash_Reset` function arguments.

Table 153: **XFlash_Reset Arguments**

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

Returns

- XST_SUCCESS if successful.
- XFLASH_BUSY if the flash devices were in the middle of an operation and could not be reset.
- XFLASH_ERROR if the device(s) have experienced an internal error during the operation. `XFlash_DeviceControl()` must be used to access the cause of the device specific error. condition.

XFlash_DeviceControl

This function is used to execute device specific commands.

For a list of device specific commands, see the xilflash.h.

***Note*:** None.

Prototype

```
int XFlash_DeviceControl(XFlash *InstancePtr, u32 Command, DeviceCtrlParam
*Parameters);
```

Parameters

The following table lists the `XFlash_DeviceControl` function arguments.

Table 154: **XFlash_DeviceControl Arguments**

Type	Name	Description
XFlash *	InstancePtr	Pointer to the XFlash instance.
u32	Command	Device specific command to issue.
DeviceCtrlParam *	Parameters	Specifies the arguments passed to the device control function.

Returns

- XST_SUCCESS if successful.
- XFLASH_NOT_SUPPORTED if the command is not recognized/supported by the device(s).

XFlash_Read

This function reads the data from the Flash device and copies it into the specified user buffer.

The source and destination addresses can be on any alignment supported by the processor.

The device is polled until an error or the operation completes successfully.

***Note*:** This function allows the transfer of data past the end of the device's address space. If this occurs, then results are undefined.

Prototype

```
int XFlash_Read(XFlash *InstancePtr, u32 Offset, u32 Bytes, void *DestPtr);
```

Parameters

The following table lists the `XFlash_Read` function arguments.

Table 155: **XFlash_Read Arguments**

Type	Name	Description
XFlash *	InstancePtr	Pointer to the XFlash instance.
u32	Offset	Offset into the device(s) address space from which to read.
u32	Bytes	Number of bytes to copy.
void *	DestPtr	Destination address to copy data to.

Returns

- XST_SUCCESS if successful.
- XFLASH_ADDRESS_ERROR if the source address does not start within the addressable areas of the device(s).

XFlash_Write

This function programs the flash device(s) with data specified in the user buffer.

The source and destination address must be aligned to the width of the flash's data bus.

The device is polled until an error or the operation completes successfully.

***Note*:** None.

Prototype

```
int XFlash_Write(XFlash *InstancePtr, u32 Offset, u32 Bytes, void *SrcPtr);
```

Parameters

The following table lists the `XFlash_Write` function arguments.

Table 156: **XFlash_Write Arguments**

Type	Name	Description
XFlash *	InstancePtr	Pointer to the XFlash instance.
u32	Offset	Offset into the device(s) address space from which to begin programming. Must be aligned to the width of the flash's data bus.
u32	Bytes	Number of bytes to program.
void *	SrcPtr	Source address containing data to be programmed. Must be aligned to the width of the flash's data bus. The SrcPtr doesn't have to be aligned to the flash width if the processor supports unaligned access. But, since this library is generic, and some processors(eg. Microblaze) do not support unaligned access; this API requires the SrcPtr to be aligned.

Returns

- XST_SUCCESS if successful.
- XFLASH_ERROR if a write error occurred. This error is usually device specific. Use `XFlash_DeviceControl()` to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially programmed.

XFlash_Erase

This function erases the specified address range in the flash device.

The number of bytes to erase can be any number as long as it is within the bounds of the device(s).

The device is polled until an error or the operation completes successfully.

***Note*:** Due to flash memory design, the range actually erased may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

Prototype

```
int XFlash_Erase(XFlash *InstancePtr, u32 Offset, u32 Bytes);
```

Parameters

The following table lists the `XFlash_Erase` function arguments.

Table 157: **XFlash_Erase Arguments**

Type	Name	Description
XFlash *	InstancePtr	Pointer to the XFlash instance.
u32	Offset	Offset into the device(s) address space from which to begin erasure.
u32	Bytes	Number of bytes to erase.

Returns

- XST_SUCCESS if successful.
- XFLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).

XFlash_Lock

This function Locks the blocks in the specified range of the flash device(s).

The device is polled until an error or the operation completes successfully.

- XFLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).

***Note*:** Due to flash memory design, the range actually locked may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

Prototype

```
int XFlash_Lock(XFlash *InstancePtr, u32 Offset, u32 Bytes);
```

Parameters

The following table lists the `XFlash_Lock` function arguments.

Table 158: **XFlash_Lock Arguments**

Type	Name	Description
XFlash *	InstancePtr	Pointer to the XFlash instance.
u32	Offset	Offset into the device(s) address space from which to begin block locking. The first three bytes of every block is reserved for special purpose. The offset should be at least three bytes from start of the block.
u32	Bytes	Number of bytes to Lock in the Block starting from Offset.

Returns

- XST_SUCCESS if successful.

XFlash_Unlock

This function Unlocks the blocks in the specified range of the flash device(s).

The device is polled until an error or the operation completes successfully.

***Note*:** None.

Prototype

```
int XFlash_Unlock(XFlash *InstancePtr, u32 Offset, u32 Bytes);
```

Parameters

The following table lists the `XFlash_Unlock` function arguments.

Table 159: **XFlash_Unlock Arguments**

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

Table 159: **XFlash_Unlock Arguments** *(cont'd)*

Type	Name	Description
u32	Offset	Offset into the device(s) address space from which to begin block UnLocking. The first three bytes of every block is reserved for special purpose. The offset should be at least three bytes from start of the block.
u32	Bytes	Number of bytes to UnLock in the Block starting from Offset.

Returns

- XST_SUCCESS if successful.
- XFLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).

XFlash_IsReady

This function checks the readiness of the device, which means it has been successfully initialized.

***Note*:** None.

Prototype

```
int XFlash_IsReady(XFlash *InstancePtr);
```

Parameters

The following table lists the `XFlash_IsReady` function arguments.

Table 160: **XFlash_IsReady Arguments**

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

Returns

TRUE if the device has been initialized (but not necessarily started), and FALSE otherwise.

Library Parameters in MSS File

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

```
BEGIN LIBRARY
PARAMETER LIBRARY_NAME = xilflash
PARAMETER LIBRARY_VER = 4.8
PARAMETER PROC_INSTANCE = microblaze_0
PARAMETER ENABLE_INTEL = true
PARAMETER ENABLE_AMD = false
END
```

The table below describes the libgen customization parameters.

Parameter	Default Value	Description
LIBRARY_NAME	xilflash	Specifies the library name.
LIBRARY_VER	4.8	Specifies the library version.
PROC_INSTANCE	microblaze_0	Specifies the processor name.
ENABLE_INTEL	true/false	Enables or disables the Intel flash device family.
ENABLE_AMD	true/false	Enables or disables the AMD flash device family.

XilFFS Library v5.0

XilFFS Library API Reference

The AMD 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 http://elm-chan.org/fsw/ff/00index_e.html

***Note*:** The XilFFS library uses R0.15 w/patch1 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 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 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 v5.1

Overview

The XilSecure library provide APIs to access cryptographic accelerators on the Zynq UltraScale +™ MPSoC devices. The library is designed to run on top of AMD standalone BSPs. It is tested for Arm Cortex™-A53, Arm Cortex-R5F, and MicroBlaze™ processors. 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 the U-Boot or Linux for cryptographic acceleration.

The XilSecure library includes:

***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, XPPU, or TrustZone to limit memory access.

- 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 the DDR memory. 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.

By default, PMUFW does not allow device key for any decryption operation requested through IP Integrator 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.

Parameter	Description
xsecure_environment	Default = FALSE. Set the value to TRUE to allow usage of device key through the IPI response calls.
tpm_support	Default = FALSE. Enables decryption of bitstream to memory and then writes it to PCAP, allows calculation of SHA on the decrypted bitstream in chunks. Valid configuration only for FSBL BSP.

Source Files

The source files for the library can be found at:

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

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_AesInitialize 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 provided GCM tag is 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.*

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 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.

XilSecure AES Zynq UltraScale+ MPSoC APIs

Table 161: **Quick Function Reference**

Type	Name	Arguments
s32	XSecure_AesInitialize	XSecure_Aes * InstancePtr XCsuDma * CsuDmaPtr u32 KeySel u32 * IvPtr u32 * KeyPtr
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

Table 161: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
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
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 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 162: **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
IvPtr	Pointer to the Initialization Vector for decryption
KeyPtr	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 163: **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 164: **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 165: **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 166: **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 167: **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 168: **XSecure_AesEncryptUpdate Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.
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 169: **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 170: **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_AesWaitForDone    Xil_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 171: **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 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.

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.

XilSecure RSA Zynq UltraScale+ MPSoC APIs

Table 172: **Quick Function Reference**

Type	Name	Arguments
u32	XSecure_RsaCfgInitialize	XSecure_Rsa * InstancePtr
u32	XSecure_RsaOperation	XSecure_Rsa * InstancePtr u8 * Input u8 * Result
u8 *	XSecure_RsaGetTPadding	void

Functions

XSecure_RsaCfgInitialize

This function stores the base address of RSA core registers.

Prototype

```
u32 XSecure_RsaCfgInitialize(XSecure_Rsa *InstancePtr);
```

Parameters

The following table lists the `XSecure_RsaCfgInitialize` function arguments.

Table 173: **XSecure_RsaCfgInitialize Arguments**

Name	Description
InstancePtr	Pointer to the `XSecure_Rsa` instance.

Returns

XST_SUCCESS on success.

XSecure_RsaOperation

This function handles the all RSA operations with provided inputs.

Prototype

```
u32 XSecure_RsaOperation(XSecure_Rsa *InstancePtr, u8 *Input, u8 *Result, u8
EncDecFlag, u32 Size);
```

Parameters

The following table lists the `XSecure_RsaOperation` function arguments.

Table 174: **XSecure_RsaOperation Arguments**

Name	Description
InstancePtr	Pointer to the `XSecure_Rsa` instance.
Input	Pointer to the buffer which contains the input data to be decrypted.
Result	Pointer to the buffer where resultant decrypted data to be stored .

Returns

XST_SUCCESS on success.

XSecure_RsaGetTPadding

This function returns PKCS padding as per the silicon version.

Prototype

```
u8 * XSecure_RsaGetTPadding(void);
```

Returns

XSecure_Silicon2_TPadSha3 if Silicon version is not 1.0 XSecure_Silicon1_TPadSha3 if Silicon version is 1.0

Enumerations

Enumeration XSecure_RsaState

Table 175: **Enumeration XSecure_RsaState Values**

Value	Description
XSECURE_RSA_UNINITIALIZED	RSA uninitialized value
XSECURE_RSA_INITIALIZED	RSA initialized value

Definitions

XSECURE_CSU_RSA_STATUS_DONE

Operation Done

Definition

```
#define XSECURE_CSU_RSA_STATUS_DONE(0x1U)
```

XSECURE_CSU_RSA_STATUS_BUSY

RSA busy

Definition

```
#define XSECURE_CSU_RSA_STATUS_BUSY(0x2U)
```

XSECURE_CSU_RSA_STATUS_ERROR

Error

Definition

```
#define XSECURE_CSU_RSA_STATUS_ERROR(0x4U)
```

XSECURE_CSU_RSA_STATUS_PROG_CNT

Progress Counter

Definition

```
#define XSECURE_CSU_RSA_STATUS_PROG_CNT(0xF8U)
```

XSECURE_CSU_RSA_CONTROL_512

RSA 512 Length Code

Definition

```
#define XSECURE_CSU_RSA_CONTROL_512(0x00U)
```

XSECURE_CSU_RSA_CONTROL_576

RSA 576 Length Code

Definition

```
#define XSECURE_CSU_RSA_CONTROL_576(0x10U)
```

XSECURE_CSU_RSA_CONTROL_704

RSA 704 Length Code

Definition

```
#define XSECURE_CSU_RSA_CONTROL_704(0x20U)
```

XSECURE_CSU_RSA_CONTROL_768

RSA 768 Length Code

Definition

```
#define XSECURE_CSU_RSA_CONTROL_768(0x30U)
```

XSECURE_CSU_RSA_CONTROL_992

RSA 992 Length Code

Definition

```
#define XSECURE_CSU_RSA_CONTROL_992(0x40U)
```

XSECURE_CSU_RSA_CONTROL_1024

RSA 1024 Length Code

Definition

```
#define XSECURE_CSU_RSA_CONTROL_1024(0x50U)
```

XSECURE_CSU_RSA_CONTROL_1152

RSA 1152 Length Code

Definition

```
#define XSECURE_CSU_RSA_CONTROL_1152(0x60U)
```

XSECURE_CSU_RSA_CONTROL_1408

RSA 1408 Length Code

Definition

```
#define XSECURE_CSU_RSA_CONTROL_1408(0x70U)
```

XSECURE_CSU_RSA_CONTROL_1536

RSA 1536 Length Code

Definition

```
#define XSECURE_CSU_RSA_CONTROL_1536(0x80U)
```

XSECURE_CSU_RSA_CONTROL_1984

RSA 1984 Length Code

Definition

```
#define XSECURE_CSU_RSA_CONTROL_1984(0x90U)
```

XSECURE_CSU_RSA_CONTROL_2048

RSA 2048 Length Code

Definition

```
#define XSECURE_CSU_RSA_CONTROL_2048(0xA0U)
```

XSECURE_CSU_RSA_CONTROL_3072

RSA 3072 Length Code

Definition

```
#define XSECURE_CSU_RSA_CONTROL_3072(0xB0U)
```

XSECURE_CSU_RSA_CONTROL_4096

RSA 4096 Length Code

Definition

```
#define XSECURE_CSU_RSA_CONTROL_4096(0xC0U)
```

XSECURE_CSU_RSA_CONTROL_DCA

Abort Operation

Definition

```
#define XSECURE_CSU_RSA_CONTROL_DCA(0x08U)
```

XSECURE_CSU_RSA_CONTROL_NOP

No Operation

Definition

```
#define XSECURE_CSU_RSA_CONTROL_NOP(0x00U)
```

XSECURE_CSU_RSA_CONTROL_EXP

Exponentiation Opcode

Definition

```
#define XSECURE_CSU_RSA_CONTROL_EXP(0x01U)
```

XSECURE_CSU_RSA_CONTROL_EXP_PRE

Expo. using R*R mod M

Definition

```
#define XSECURE_CSU_RSA_CONTROL_EXP_PRE(0x05U)
```

XSECURE_CSU_RSA_CONTROL_MASK

Definition

```
#define XSECURE_CSU_RSA_CONTROL_MASK                    (
                XSECURE_CSU_RSA_CONTROL_4096
             + \

                XSECURE_CSU_RSA_CONTROL_EXP_PRE
            )
```

XSECURE_RSA_FAILED

RSA Failed Error Code

Definition

```
#define XSECURE_RSA_FAILED0x1U
```

XSECURE_RSA_DATA_VALUE_ERROR

for RSA private decryption data should be lesser than modulus

Definition

```
#define XSECURE_RSA_DATA_VALUE_ERROR0x2U
```

XSECURE_RSA_ZEROIZE_ERROR

for RSA zeroization Error

Definition

```
#define XSECURE_RSA_ZEROIZE_ERROR0x80U
```

XSECURE_HASH_TYPE_SHA3

SHA-3 hash size

Definition

```
#define XSECURE_HASH_TYPE_SHA3(48U)
```

XSECURE_FSBL_SIG_SIZE

FSBL signature size

Definition

```
#define XSECURE_FSBL_SIG_SIZE(512U)
```

XSECURE_RSA_MAX_BUFF

RSA RAM Write Buffers

Definition

```
#define XSECURE_RSA_MAX_BUFF(6U)
```

XSECURE_RSA_MAX_RD_WR_CNT

No of writes or reads to RSA RAM Buffers

Definition

```
#define XSECURE_RSA_MAX_RD_WR_CNT(22U)
```

XSECURE_RSA_BYTE_MASK

RSA BYTE MASK

Definition

```
#define XSECURE_RSA_BYTE_MASK(0XFFU)
```

XSECURE_RSA_BYTE_SHIFT

RSA BYTE

Definition

```
#define XSECURE_RSA_BYTE_SHIFT(8U)
```

XSECURE_RSA_HWORD_SHIFT

RSA HWORD

Definition

```
#define XSECURE_RSA_HWORD_SHIFT(16U)
```

XSECURE_RSA_SWORD_SHIFT

RSA SWORD

Definition

```
#define XSECURE_RSA_SWORD_SHIFT(24U)
```

XSECURE_RSA_512_KEY_SIZE

RSA 512 key size

Definition

```
#define XSECURE_RSA_512_KEY_SIZE(512U/8U)
```

XSECURE_RSA_576_KEY_SIZE

RSA 576 key size

Definition

```
#define XSECURE_RSA_576_KEY_SIZE(576U/8U)
```

XSECURE_RSA_704_KEY_SIZE

RSA 704 key size

Definition

```
#define XSECURE_RSA_704_KEY_SIZE(704U/8U)
```

XSECURE_RSA_768_KEY_SIZE

RSA 768 key size

Definition

```
#define XSECURE_RSA_768_KEY_SIZE(768U/8U)
```

XSECURE_RSA_992_KEY_SIZE

RSA 992 key size

Definition

```
#define XSECURE_RSA_992_KEY_SIZE(992U/8U)
```

XSECURE_RSA_1024_KEY_SIZE

RSA 1024 key size

Definition

```
#define XSECURE_RSA_1024_KEY_SIZE(1024U/8U)
```

XSECURE_RSA_1152_KEY_SIZE

RSA 1152 key size

Definition

```
#define XSECURE_RSA_1152_KEY_SIZE(1152U/8U)
```

XSECURE_RSA_1408_KEY_SIZE

RSA 1408 key size

Definition

```
#define XSECURE_RSA_1408_KEY_SIZE(1408U/8U)
```

XSECURE_RSA_1536_KEY_SIZE

RSA 1536 key size

Definition

```
#define XSECURE_RSA_1536_KEY_SIZE(1536U/8U)
```

XSECURE_RSA_1984_KEY_SIZE

RSA 1984 key size

Definition

```
#define XSECURE_RSA_1984_KEY_SIZE(1984U/8U)
```

XSECURE_RSA_2048_KEY_SIZE

RSA 2048 key size

Definition

```
#define XSECURE_RSA_2048_KEY_SIZE(2048U/8U)
```

XSECURE_RSA_3072_KEY_SIZE

RSA 3072 key size

Definition

```
#define XSECURE_RSA_3072_KEY_SIZE(3072U/8U)
```

XSECURE_RSA_4096_KEY_SIZE

RSA 4096 key size

Definition

```
#define XSECURE_RSA_4096_KEY_SIZE(4096U/8U)
```

XSECURE_RSA_512_SIZE_WORDS

RSA 512 Size in words

Definition

```
#define XSECURE_RSA_512_SIZE_WORDS(16)
```

XSECURE_RSA_576_SIZE_WORDS

RSA 576 Size in words

Definition

```
#define XSECURE_RSA_576_SIZE_WORDS(18)
```

XSECURE_RSA_704_SIZE_WORDS

RSA 704 Size in words

Definition

```
#define XSECURE_RSA_704_SIZE_WORDS(22)
```

XSECURE_RSA_768_SIZE_WORDS

RSA 768 Size in words

Definition

```
#define XSECURE_RSA_768_SIZE_WORDS(24)
```

XSECURE_RSA_992_SIZE_WORDS

RSA 992 Size in words

Definition

```
#define XSECURE_RSA_992_SIZE_WORDS(31)
```

XSECURE_RSA_1024_SIZE_WORDS

RSA 1024 Size in words

Definition

```
#define XSECURE_RSA_1024_SIZE_WORDS(32)
```

XSECURE_RSA_1152_SIZE_WORDS

RSA 1152 Size in words

Definition

```
#define XSECURE_RSA_1152_SIZE_WORDS(36)
```

XSECURE_RSA_1408_SIZE_WORDS

RSA 1408 Size in words

Definition

```
#define XSECURE_RSA_1408_SIZE_WORDS(44)
```

XSECURE_RSA_1536_SIZE_WORDS

RSA 1536 Size in words

Definition

```
#define XSECURE_RSA_1536_SIZE_WORDS(48)
```

XSECURE_RSA_1984_SIZE_WORDS

RSA 1984 Size in words

Definition

```
#define XSECURE_RSA_1984_SIZE_WORDS(62)
```

XSECURE_RSA_2048_SIZE_WORDS

RSA 2048 Size in words

Definition

```
#define XSECURE_RSA_2048_SIZE_WORDS(64)
```

XSECURE_RSA_3072_SIZE_WORDS

RSA 3072 Size in words

Definition

```
#define XSECURE_RSA_3072_SIZE_WORDS(96)
```

XSECURE_RSA_4096_SIZE_WORDS

RSA 4096 Size in words

Definition

```
#define XSECURE_RSA_4096_SIZE_WORDS(128U)
```

XSECURE_CSU_RSA_RAM_EXPO

bit for RSA RAM Exponent

Definition

```
#define XSECURE_CSU_RSA_RAM_EXPO(0U)
```

XSECURE_CSU_RSA_RAM_MOD

bit for RSA RAM modulus

Definition

```
#define XSECURE_CSU_RSA_RAM_MOD(1U)
```

XSECURE_CSU_RSA_RAM_DIGEST

bit for RSA RAM Digest

Definition

```
#define XSECURE_CSU_RSA_RAM_DIGEST(2U)
```

XSECURE_CSU_RSA_RAM_SPAD

bit for RSA RAM SPAD

Definition

```
#define XSECURE_CSU_RSA_RAM_SPAD(3U)
```

XSECURE_CSU_RSA_RAM_RES_Y

bit for RSA RAM Result(Y)

Definition

```
#define XSECURE_CSU_RSA_RAM_RES_Y(4U)
```

XSECURE_CSU_RSA_RAM_RES_Q

bit for RSA RAM Result(Q)

Definition

```
#define XSECURE_CSU_RSA_RAM_RES_Q(5U)
```

XSECURE_RSA_SIGN_ENC

RSA encryption flag

Definition

```
#define XSECURE_RSA_SIGN_ENC0U
```

XSECURE_RSA_SIGN_DEC

RSA decryption flag

Definition

```
#define XSECURE_RSA_SIGN_DEC1U
```

XilSecure RSA Common APIs

This section provides RSA common APIS for Zynq UltraScale+ MPSoC and Versal adaptive SoC.

Table 176: **Quick Function Reference**

Type	Name	Arguments
int	XSecure_RsaInitialize	XSecure_Rsa * InstancePtr u8 * Mod u8 * ModExt u8 * ModExpo
int	XSecure_RsaInitialize_64Bit	XSecure_Rsa * InstancePtr u64 Mod u64 ModExt u64 ModExpo
int	XSecure_RsaSignVerification	const u8 * Signature const u8 * Hash u32 HashLen
int	XSecure_RsaSignVerification_64Bit	const u64 Signature const u64 Hash u32 HashLen

Table 176: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
int	XSecure_RsaPublicEncrypt	XSecure_Rsa * InstancePtr u8 * Input u32 Size u8 * Result
int	XSecure_RsaPublicEncrypt_64Bit	XSecure_Rsa * InstancePtr u64 Input u32 Size u64 Result
int	XSecure_RsaPrivateDecrypt	XSecure_Rsa * InstancePtr u8 * Input u32 Size u8 * Result
int	XSecure_RsaPrivateDecrypt_64Bit	XSecure_Rsa * InstancePtr u64 Input u32 Size u64 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 partition signature when authenticated boot image is generated by bootgen, else the all of them should be extracted from the key

Prototype

```
int XSecure_RsaInitialize(XSecure_Rsa *InstancePtr, u8 *Mod, u8 *ModExt, u8
*ModExpo);
```

Parameters

The following table lists the `XSecure_RsaInitialize` function arguments.

Table 177: **XSecure_RsaInitialize Arguments**

Name	Description
InstancePtr	- Pointer to the `XSecure_Rsa` instance

Table 177: **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_RSA_INVALID_PARAM - On invalid arguments

XSecure_RsaInitialize_64Bit

This function initializes a a `XSecure_Rsa` structure with the default values located at a 64-bit address required for operating the RSA cryptographic engine.

***Note*:** Modulus, ModExt and ModExpo are part of partition signature when authenticated boot image is generated by bootgen, else the all of them should be extracted from the key

Prototype

```
int XSecure_RsaInitialize_64Bit(XSecure_Rsa *InstancePtr, u64 Mod, u64
ModExt, u64 ModExpo);
```

Parameters

The following table lists the `XSecure_RsaInitialize_64Bit` function arguments.

Table 178: **XSecure_RsaInitialize_64Bit Arguments**

Name	Description
InstancePtr	- Pointer to the `XSecure_Rsa` instance
Mod	- Address of the key Modulus of key size
ModExt	- Address of the pre-calculated exponential (R^2 Mod N) value • 0 - if user doesn't have pre-calculated R^2 Mod N value, control will take care of this calculation internally
ModExpo	- Address of the buffer which contains key exponent

Returns

- XST_SUCCESS - If initialization was successful

- XSECURE_RSA_INVALID_PARAM - On invalid arguments

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

```
int XSecure_RsaSignVerification(const u8 *Signature, const u8 *Hash, u32
HashLen);
```

Parameters

The following table lists the `XSecure_RsaSignVerification` function arguments.

Table 179: **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

Returns

- XST_SUCCESS - If decryption was successful
- XSECURE_RSA_INVALID_PARAM - On invalid arguments
- XST_FAILURE - In case of mismatch

XSecure_RsaSignVerification_64Bit

This function verifies the RSA decrypted data located at a 64-bit address provided is either matching with the provided expected hash by taking care of PKCS padding.

Prototype

```
int XSecure_RsaSignVerification_64Bit(const u64 Signature, const u64 Hash,
u32 HashLen);
```

Parameters

The following table lists the `XSecure_RsaSignVerification_64Bit` function arguments.

Table 180: **XSecure_RsaSignVerification_64Bit Arguments**

Name	Description
Signature	- Address of the buffer which holds the decrypted RSA signature
Hash	- Address of 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

Returns

- XST_SUCCESS - If decryption was successful
- XSECURE_RSA_INVALID_PARAM - On invalid arguments
- 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

```
int XSecure_RsaPublicEncrypt(XSecure_Rsa *InstancePtr, u8 *Input, u32 Size,
u8 *Result);
```

Parameters

The following table lists the `XSecure_RsaPublicEncrypt` function arguments.

Table 181: **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
- XSECURE_RSA_INVALID_PARAM - On invalid arguments
- XSECURE_RSA_STATE_MISMATCH_ERROR - If State mismatch is occurred

XSecure_RsaPublicEncrypt_64Bit

This function handles the RSA encryption for data available at 64-bit address 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

```
int XSecure_RsaPublicEncrypt_64Bit(XSecure_Rsa *InstancePtr, u64 Input, u32
Size, u64 Result);
```

Parameters

The following table lists the `XSecure_RsaPublicEncrypt_64Bit` function arguments.

Table 182: **XSecure_RsaPublicEncrypt_64Bit Arguments**

Name	Description
InstancePtr	Pointer to the `XSecure_Rsa` instance
Input	Address of 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	Address of buffer where resultant decrypted data to be stored

Returns

- XST_SUCCESS - If encryption was successful
- XSECURE_RSA_INVALID_PARAM - On invalid arguments
- XSECURE_RSA_STATE_MISMATCH_ERROR - If State mismatch is occurred

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 here needs to match the key size used in the XSecure_RsaInitialize function

Prototype

```
int XSecure_RsaPrivateDecrypt(XSecure_Rsa *InstancePtr, u8 *Input, u32 Size,
u8 *Result);
```

Parameters

The following table lists the `XSecure_RsaPrivateDecrypt` function arguments.

Table 183: **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
- XSECURE_RSA_STATE_MISMATCH_ERROR - If State mismatch is occurred
- XST_FAILURE - On RSA operation failure

XSecure_RsaPrivateDecrypt_64Bit

This function handles the RSA decryption for data available at 64-bit address with the private 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

```
int XSecure_RsaPrivateDecrypt_64Bit(XSecure_Rsa *InstancePtr, u64 Input, u32
Size, u64 Result);
```

Parameters

The following table lists the `XSecure_RsaPrivateDecrypt_64Bit` function arguments.

Table 184: **XSecure_RsaPrivateDecrypt_64Bit Arguments**

Name	Description
InstancePtr	- Pointer to the `XSecure_Rsa` instance
Input	- Address of 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	- Address of 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
- XSECURE_RSA_STATE_MISMATCH_ERROR - If State mismatch is occurred
- XST_FAILURE - On RSA operation failure

Definitions

XSECURE_RSA_BYTE_PAD_LENGTH

PKCS Byte Padding

Definition

```
#define XSECURE_RSA_BYTE_PAD_LENGTH(3U)
```

XSECURE_RSA_T_PAD_LENGTH

PKCS T Padding

Definition

```
#define XSECURE_RSA_T_PAD_LENGTH(19U)
```

XSECURE_RSA_BYTE_PAD1

PKCS T Padding Byte

Definition

```
#define XSECURE_RSA_BYTE_PAD1(0X00U)
```

XSECURE_RSA_BYTE_PAD2

PKCS T Padding Byte

Definition

```
#define XSECURE_RSA_BYTE_PAD2(0X01U)
```

XSECURE_RSA_BYTE_PAD3

PKCS T Padding Byte

Definition

```
#define XSECURE_RSA_BYTE_PAD3(0XFFU)
```

XSECURE_RSA_INVALID_PARAM

Invalid Argument

Definition

```
#define XSECURE_RSA_INVALID_PARAM(0x82U)
```

XSECURE_RSA_STATE_MISMATCH_ERROR

State mismatch

Definition

```
#define XSECURE_RSA_STATE_MISMATCH_ERROR(0x84U)
```

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.

```
static 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.

XilSecure SHA3 Zynq UltraScale+ MPSoC APIs

Table 185: **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
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 XSecure_Sha3PadType Sha3PadType
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 186: **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 187: **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 188: **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 189: **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 190: **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
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 191: **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 192: **XSecure_Sha3PadSelection Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.
Sha3PadType	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 193: **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 the operation.

Prototype

```
u32 XSecure_Sha3WaitForDone(XSecure_Sha3 *InstancePtr);
```

Parameters

The following table lists the `XSecure_Sha3WaitForDone` function arguments.

Table 194: **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:

```
static u32 SecureSha3Example()
{
        XSecure_Sha3 Secure_Sha3;
        XCsuDma CsuDma;
        XCsuDma_Config *Config;
        u8 Out[SHA3_HASH_LEN_IN_BYTES];
        u32 Status = XST_FAILURE;
        u32 Size = 0U;
```

```
	Size = Xil_Strnlen(Data, SHA3_INPUT_DATA_LEN);
	if (Size != SHA3_INPUT_DATA_LEN) {
		xil_printf("Provided data length is Invalid\n\r");
		Status = XST_FAILURE;
		goto END;
	}

	Config = XCsuDma_LookupConfig(0);
	if (NULL == Config) {
		xil_printf("config failed\n\r");
		Status = XST_FAILURE;
		goto END;
	}

	Status = XCsuDma_CfgInitialize(&CsuDma, Config, Config->BaseAddress);
	if (Status != XST_SUCCESS) {
		Status = XST_FAILURE;
		goto END;
	}

	/*
	 * Initialize the SHA-3 driver so that it's ready to use
	 */
	XSecure_Sha3Initialize(&Secure_Sha3, &CsuDma);

	XSecure_Sha3Digest(&Secure_Sha3, (u8*)Data, Size, Out);

	xil_printf(" Calculated Hash \r\n ");
	SecureSha3PrintHash(Out);

	Status = SecureSha3CompareHash(Out, ExpHash);
END:
	return Status;
}

/
*****************************************************************************/
static u32 SecureSha3CompareHash(u8 *Hash, u8 *ExpectedHash)
{
	u32 Index;
	u32 Status = XST_FAILURE;

	for (Index = 0U; Index < SHA3_HASH_LEN_IN_BYTES; Index++) {
		if (Hash[Index] != ExpectedHash[Index]) {
			xil_printf("Expected Hash \r\n");
			SecureSha3PrintHash(ExpectedHash);
			xil_printf("SHA Example Failed at Hash Comparison \r
\n");
			break;
		}
	}
	if (Index == SHA3_HASH_LEN_IN_BYTES) {
		Status = XST_SUCCESS;
	}

	return Status;
}

/
*****************************************************************************/
static void SecureSha3PrintHash(u8 *Hash)
{
```

```
        u32 Index;

        for (Index = 0U; Index < SHA3_HASH_LEN_IN_BYTES; Index++) {
                xil_printf(" %0x ", Hash[Index]);
        }
        xil_printf(" \r\n ");
}
```

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.

Data Structure Index

The following is a list of data structures:

- XSecure_AuthParam
- XSecure_Rsa

XSecure_AuthParam

Declaration

```
typedef struct
{
  XCsuDma * CsuDmaInstPtr,
  u8 * Data,
  u32 Size,
  u8 * AuthCertPtr,
  u32 SignatureOffset,
  XSecure_Sha3PadType PaddingType,
  u8 AuthIncludingCert
} XSecure_AuthParam;
```

Table 195: **Structure XSecure_AuthParam member description**

Member	Description
CsuDmaInstPtr	CSUDMA instance pointer
Data	Data pointer
Size	Size
AuthCertPtr	Authentication certificate pointer
SignatureOffset	Signature offset
PaddingType	SHA3 padding type
AuthIncludingCert	Authentication including certificate

XSecure_Rsa

The RSA driver instance data structure. A pointer to an instance data structure is passed around by functions to refer to a specific driver instance.

Declaration

```
typedef struct
{
  u32 BaseAddress,
  u8 * Mod,
  u8 * ModExt,
  u8 * ModExpo,
  u8 EncDec,
  u32 SizeInWords,
  XSecure_RsaState RsaState
} XSecure_Rsa;
```

Table 196: **Structure XSecure_Rsa member description**

Member	Description
BaseAddress	Device Base Address
Mod	Modulus
ModExt	Precalc. R sq. mod N
ModExpo	Exponent
EncDec	0 for signature verification and 1 for generation
SizeInWords	
RsaState	RSA key size in words

XilSkey Library v7.4

Overview

The XilSKey library provides APIs for programming and reading eFUSE bits and for programming the battery-backed RAM (BBRAM) of AMD Zynq™ 7000, AMD UltraScale™ , and AMD UltraScale+™ 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+ 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

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:

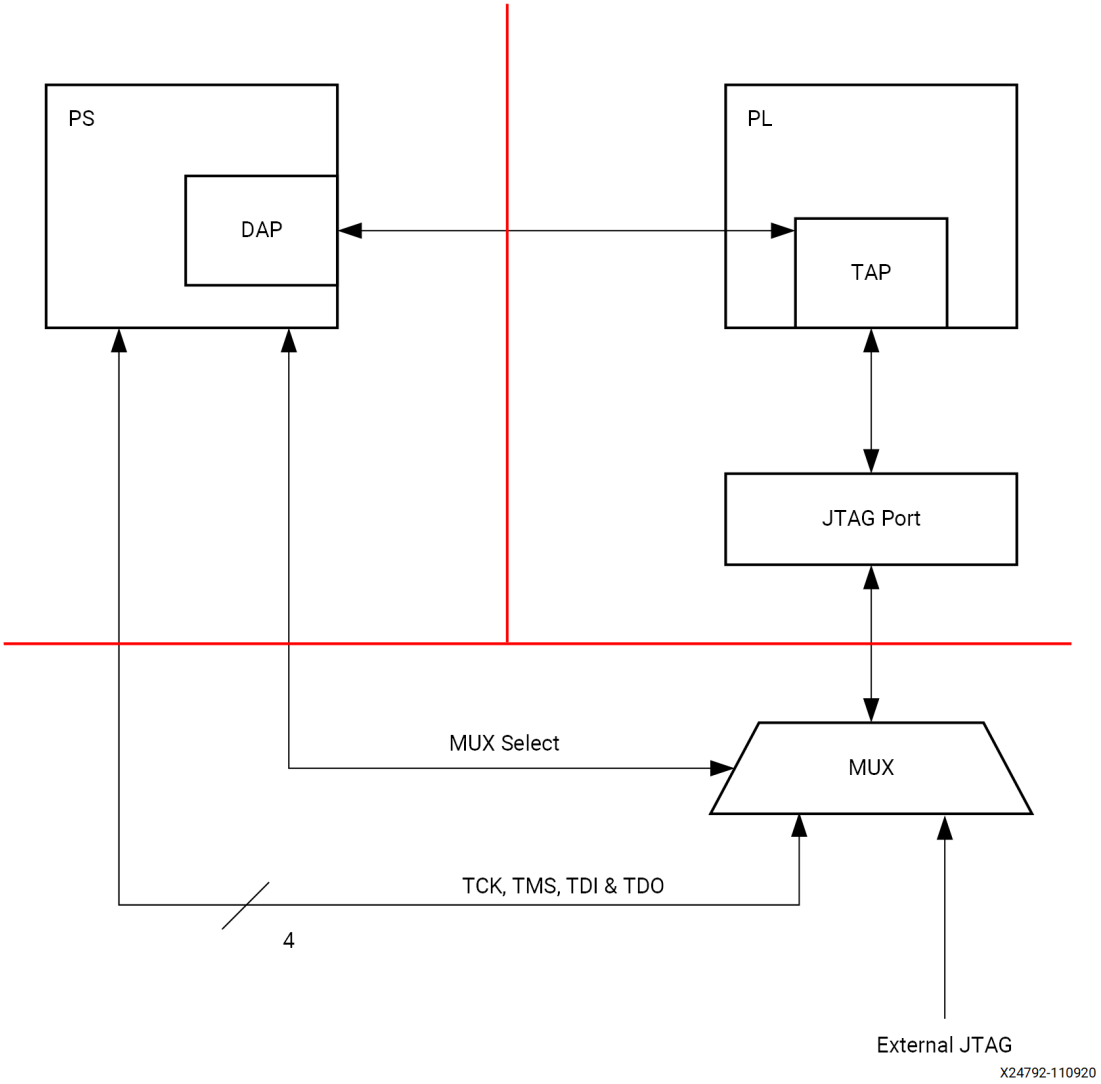
- Include a MUX between the external JTAG and the JTAG operation driven by the MIOs
- Assign a MUX selection PIN To rephrase, to select JTAG for PL EFUSE or PL BBRAM writing, you must define the following:

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.

***Note*:** 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 APIs

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 .
- 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 197: **Quick Function Reference**

Type	Member	Arguments
int	XilSKey_Bbram_Program	XilSKey_Bbram * InstancePtr
int	XilSKey_Bbram_JTAGServerInit	XilSKey_Bbram * InstancePtr

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 198: **XilSKey_Bbram_Program Arguments**

Type	Member	Description
XilSKey_Bbram *	InstancePtr	Pointer to XilSKey_Bbram

Returns

XilSKey_Bbram_JTAGServerInit

This function initializes JTAG server.

Prototype

```
int XilSKey_Bbram_JTAGServerInit(XilSKey_Bbram *InstancePtr);
```

Parameters

The following table lists the `XilSKey_Bbram_JTAGServerInit` function arguments.

Table 199: **XilSKey_Bbram_JTAGServerInit Arguments**

Type	Member	Description
XilSKey_Bbram *	InstancePtr	Pointer to XilSKey_Bbram

Returns

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

- 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 200: **Quick Function Reference**

Type	Member	Arguments
u32	XilSKey_ZynqMp_Bbram_Program	const u32 * AesKey
u32	XilSKey_ZynqMp_Bbram_Zeroise	void

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(const u32 *AesKey);
```

Parameters

The following table lists the `XilSKey_ZynqMp_Bbram_Program` function arguments.

Table 201: **XilSKey_ZynqMp_Bbram_Program Arguments**

Type	Member	Description
const 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);
```

Returns

Zynq eFUSE PS APIs

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 202: **Quick Function Reference**

Type	Member	Arguments
u32	XilSKey_EfusePs_Write	XilSKey_EPs * InstancePtr
u32	XilSKey_EfusePs_Read	XilSKey_EPs * InstancePtr
u32	XilSKey_EfusePs_ReadStatus	XilSKey_EPs * InstancePtr u32 * StatusBits

Functions

XilSKey_EfusePs_Write

PS eFUSE interface functions

This function is used to write to the PS eFUSE.

***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 *InstancePtr);
```

Parameters

The following table lists the `XilSKey_EfusePs_Write` function arguments.

Table 203: **XilSKey_EfusePs_Write Arguments**

Type	Member	Description
XilSKey_EPs *	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 *InstancePtr);
```

Parameters

The following table lists the `XilSKey_EfusePs_Read` function arguments.

Table 204: **XilSKey_EfusePs_Read Arguments**

Type	Member	Description
XilSKey_EPs *	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 205: **XilSKey_EfusePs_ReadStatus Arguments**

Type	Member	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 APIs

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.

This file contains the PS eFUSE API's of Zynq UltraScale+ MPSoC to program/read the eFUSE array.

Table 206: **Quick Function Reference**

Type	Member	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	const u32 AddrHigh const u32 AddrLow
void	XilSKey_ZynqMp_EfusePs_SetTimerValues	void
u32	XilSKey_ZynqMp_EfusePs_ReadRow	u8 Row XskEfusePs_Type EfuseType u32 * RowData
u32	XilSKey_ZynqMp_EfusePs_SetWriteConditions	void

Table 206: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
u32	XilSKey_ZynqMp_EfusePs_WriteAndVerifyBit	u8 Row u8 Column XskEfusePs_Type EfuseType
u32	XilSKey_ZynqMp_EfusePs_Init	void
u32	XilSKey_ZynqMp_EfusePs_CheckForZeros	u8 RowStart u8 RowEnd XskEfusePs_Type EfuseType
u32	XilSKey_ZynqMp_EfusePs_ProgramPufAsUserFuses	const XilSKey_PufEfuse * PufFuse
u32	XilSKey_ZynqMp_EfusePs_ReadPufAsUserFuses	EfuseAccess
u32	XilSKey_ZynqMp_EfusePs_WritePufHelprData	const XilSKey_Puf * InstancePtr
u32	XilSKey_ZynqMp_EfusePs_ReadPufHelprData	u32 * Address
u32	XilSKey_ZynqMp_EfusePs_WritePufChash	const XilSKey_Puf * InstancePtr
u32	XilSKey_ZynqMp_EfusePs_ReadPufChash	u32 * Address u8 ReadOption
u32	XilSKey_ZynqMp_EfusePs_WritePufAux	const XilSKey_Puf * InstancePtr
u32	XilSKey_ZynqMp_EfusePs_ReadPufAux	u32 * Address u8 ReadOption
u32	XilSKey_Write_Puf_EfusePs_SecureBits	const XilSKey_Puf_Secure * WriteSecureBits
u32	XilSKey_Read_Puf_EfusePs_SecureBits	XilSKey_Puf_Secure * SecureBitsRead u8 ReadOption
u32	XilSKey_Puf_Registration	XilSKey_Puf * InstancePtr
u32	XilSKey_Puf_Regeneration	const 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 207: **XilSKey_ZynqMp_EfusePs_CheckAesKeyCrc Arguments**

Type	Member	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 208: **XilSKey_ZynqMp_EfusePs_ReadUserFuse Arguments**

Type	Member	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 209: **XilSKey_ZynqMp_EfusePs_ReadPpk0Hash Arguments**

Type	Member	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 210: **XilSKey_ZynqMp_EfusePs_ReadPpk1Hash Arguments**

Type	Member	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 211: **XilSKey_ZynqMp_EfusePs_ReadSpkId Arguments**

Type	Member	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 212: **XilSKey_ZynqMp_EfusePs_ReadDna Arguments**

Type	Member	Description
u32 *	DnaRead	Pointer to an array of 3 x u32 words which holds the readback DNA.

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 213: **XilSKey_ZynqMp_EfusePs_ReadSecCtrlBits Arguments**

Type	Member	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

This function reloads the cache of eFUSE so that can be directly read from cache.

***Note*:** Not recommended to call this API frequently, if this API is called all the cache memory is reloded by reading eFUSE array, reading eFUSE bit multiple times may diminish the life time.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_CacheLoad(void);
```

Returns

- XST_SUCCESS on successful cache reload
- ErrorCode on failure

XilSKey_ZynqMp_EfusePs_Write

This function is used to program the PS eFUSE of Zynq UltraScale+ MPSoC, 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 214: **XilSKey_ZynqMp_EfusePs_Write Arguments**

Type	Member	Description
XilSKey_ZynqMpEPs *	InstancePtr	Pointer to the XilSKey_ZynqMpEPs.

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

XilSkey_ZynqMpEfuseAccess

This function is used by PMUFW IPI call handler for programming eFUSE.

Register	Read	Write	Size in bytes	Offset
Version	YES	NO	0x0	0x4
DNA	YES	NO	0xc	0xC
User0	YES	YES	0x4	0x20
User1	YES	YES	0x4	0x24
User2	YES	YES	0x4	0x28
User3	YES	YES	0x4	0x2c
User4	YES	YES	0x4	0x30
User5	YES	YES	0x4	0x34
User6	YES	YES	0x4	0x38
User7	YES	YES	0x4	0x3C
Misc user	YES	YES	0x4	0x40
Secure control	YES	YES	0x4	0x58
SPK ID	YES	YES	0x4	0x5C

Register	Read	Write	Size in bytes	Offset
AES key	NO	YES	0x20	0x60
PPK0 hash	YES	YES	0x30	0xA0
PPK1 hash	YES	YES	0x30	0xD0

Prototype

```
u32 XilSkey_ZynqMpEfuseAccess(const u32 AddrHigh, const u32 AddrLow);
```

Parameters

The following table lists the `XilSkey_ZynqMpEfuseAccess` function arguments.

Table 215: **XilSkey_ZynqMpEfuseAccess Arguments**

Type	Member	Description
const u32	AddrHigh	Higher 32-bit address of the XilSKey_Efuse structure.
const u32	AddrLow	Lower 32-bit address of the XilSKey_Efuse structure.

Returns

- XST_SUCCESS - On success
- ErrorCode - on Failure

XilSKey_ZynqMp_EfusePs_SetTimerValues

This function sets timers for programming and reading from eFUSE.

Prototype

```
void XilSKey_ZynqMp_EfusePs_SetTimerValues(void);
```

XilSKey_ZynqMp_EfusePs_ReadRow

This function returns particular row data directly from eFUSE array.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadRow(u8 Row, XskEfusePs_Type EfuseType, u32
*RowData);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadRow` function arguments.

Table 216: **XilSKey_ZynqMp_EfusePs_ReadRow Arguments**

Type	Member	Description
u8	Row	specifies the row number to read.
XskEfusePs_Type	EfuseType	specifies the eFUSE type.
u32 *	RowData	is a pointer to 32-bit variable to hold the data read from provided data

Returns

- XST_SUCCESS - On success
- ErrorCode - on Failure

XilSKey_ZynqMp_EfusePs_SetWriteConditions

This function sets all the required parameters to program eFUSE array.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_SetWriteConditions(void);
```

Returns

- XST_SUCCESS - On success
- ErrorCode - on Failure

XilSKey_ZynqMp_EfusePs_WriteAndVerifyBit

This function programs and verifies the particular bit of eFUSE array

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_WriteAndVerifyBit(u8 Row, u8 Column,
XskEfusePs_Type EfuseType);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_WriteAndVerifyBit` function arguments.

Table 217: **XilSKey_ZynqMp_EfusePs_WriteAndVerifyBit Arguments**

Type	Member	Description
u8	Row	specifies the row number.
u8	Column	specifies the column number.
XskEfusePs_Type	EfuseType	specifies the eFUSE type.

Returns

- XST_SUCCESS - On success
- ErrorCode - on Failure

XilSKey_ZynqMp_EfusePs_Init

This function initializes sysmonpsu driver.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_Init(void);
```

Returns

- XST_SUCCESS - On success
- ErrorCode - on Failure

XilSKey_ZynqMp_EfusePs_CheckForZeros

This function is used verify eFUSE keys for Zeros

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_CheckForZeros(u8 RowStart, u8 RowEnd,
XskEfusePs_Type EfuseType);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_CheckForZeros` function arguments.

Table 218: **XilSKey_ZynqMp_EfusePs_CheckForZeros Arguments**

Type	Member	Description
u8	RowStart	is row number from which verification has to be started.
u8	RowEnd	is row number till which verification has to be ended.
XskEfusePs_Type	EfuseType	is the type of the eFUSE in which these rows reside.

Returns

- XST_SUCCESS if keys are not programmed.
  - Errorcode on failure.

XilSKey_ZynqMp_EfusePs_ProgramPufAsUserFuses

This function programs PUF HD eFuses as general purpose eFuses

***Note*:** For ZynqMp by default PUF eFuses are used for PUF helper data To program PUF eFuses as general purpose eFuses user needs to enable the macro XSK_ACCESS_PUF_USER_EFUSE - For BareMetal support XSK_ACCESS_USER_EFUSE and XSK_ACCESS_PUF_USER_EFUSE - For Linux support

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ProgramPufAsUserFuses(const XilSKey_PufEfuse
*PufFuse);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ProgramPufAsUserFuses` function arguments.

Table 219: **XilSKey_ZynqMp_EfusePs_ProgramPufAsUserFuses Arguments**

Type	Member	Description
const XilSKey_PufEfuse *	PufFuse	is pointer to the XilSKey_PufEfuse structure

Returns

- XST_SUCCESS - On success
- ErrorCode - on Failure

XilSKey_ZynqMp_EfusePs_ReadPufAsUserFuses

This function reads PUF HD eFuses as general purpose eFuses

***Note*:** For ZynqMp by default PUF eFuses are used for PUF helper data To program PUF eFuses as general purpose eFuses user needs to enable the macro XSK_ACCESS_PUF_USER_EFUSE - For BareMetal support XSK_ACCESS_USER_EFUSE and XSK_ACCESS_PUF_USER_EFUSE - For Linux support

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadPufAsUserFuses(const XilSKey_PufEfuse
*PufFuse);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadPufAsUserFuses` function arguments.

Table 220: **XilSKey_ZynqMp_EfusePs_ReadPufAsUserFuses Arguments**

Type	Member	Description
Commented parameter EfuseAccess does not exist in function XilSKey_ZynqMp_EfusePs_ReadPufAsUserFuses.	EfuseAccess	is a pointer to the XilSKey_Efuse structure

Returns

- XST_SUCCESS - On success
- ErrorCode - on Failure

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(const XilSKey_Puf *InstancePtr);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_WritePufHelprData` function arguments.

Table 221: **XilSKey_ZynqMp_EfusePs_WritePufHelprData Arguments**

Type	Member	Description
const 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 222: **XilSKey_ZynqMp_EfusePs_ReadPufHelprData Arguments**

Type	Member	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(const XilSKey_Puf *InstancePtr);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_WritePufChash` function arguments.

Table 223: **XilSKey_ZynqMp_EfusePs_WritePufChash Arguments**

Type	Member	Description
const 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 224: **XilSKey_ZynqMp_EfusePs_ReadPufChash Arguments**

Type	Member	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(const XilSKey_Puf *InstancePtr);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_WritePufAux` function arguments.

Table 225: **XilSKey_ZynqMp_EfusePs_WritePufAux Arguments**

Type	Member	Description
const 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 226: **XilSKey_ZynqMp_EfusePs_ReadPufAux Arguments**

Type	Member	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(const XilSKey_Puf_Secure
*WriteSecureBits);
```

Parameters

The following table lists the `XilSKey_Write_Puf_EfusePs_SecureBits` function arguments.

Table 227: **XilSKey_Write_Puf_EfusePs_SecureBits Arguments**

Type	Member	Description
const 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 228: **XilSKey_Read_Puf_EfusePs_SecureBits Arguments**

Type	Member	Description
XilSKey_Puf_Secure *	SecureBitsRead	- 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 229: **XilSKey_Puf_Registration Arguments**

Type	Member	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(const XilSKey_Puf *InstancePtr);
```

Parameters

The following table lists the `XilSKey_Puf_Regeneration` function arguments.

Table 230: **XilSKey_Puf_Regeneration Arguments**

Type	Member	Description
const 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 APIs

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 231: **Quick Function Reference**

Type	Member	Arguments
u32	XilSKey_EfusePl_SystemInit	`XilSKey_EPl` * InstancePtr
u32	XilSKey_EfusePl_Program	`XilSKey_EPl` * InstancePtr
u32	XilSKey_EfusePl_ReadStatus	`XilSKey_EPl` * InstancePtr u32 * StatusBits

Table 231: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
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 232: **XilSKey_EfusePl_SystemInit Arguments**

Type	Member	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 *InstancePtr);
```

Parameters

The following table lists the `XilSKey_EfusePl_Program` function arguments.

Table 233: **XilSKey_EfusePl_Program Arguments**

Type	Member	Description
XilSKey_EPl *	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 234: **XilSKey_EfusePl_ReadStatus Arguments**

Type	Member	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 235: **XilSKey_EfusePl_ReadKey Arguments**

Type	Member	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 236: **Quick Function Reference**

Type	Member	Arguments
u32	XilSKey_CrcCalculation	const u8 * Key
u32	XilSkey_CrcCalculation_AesKey	const 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(const u8 *Key);
```

Parameters

The following table lists the `XilSKey_CrcCalculation` function arguments.

Table 237: **XilSKey_CrcCalculation Arguments**

Type	Member	Description
const 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(const u8 *Key);
```

Parameters

The following table lists the `XilSkey_CrcCalculation_AesKey` function arguments.

Table 238: **XilSkey_CrcCalculation_AesKey Arguments**

Type	Member	Description
const 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.
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	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 does 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.
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.
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.

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)

The table below lists the MUX selection pin for Zynq BBRAM PL JTAG operations.

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-bit.

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 and User Key Parameters

The table below lists the AES key 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.
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	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 = 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_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 = 00000000000000000524156a63950bcedafeadcdeabaadee34216615aaaabbaaa. 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.
XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_1	Default = 00000000000000000524156a63950bcedafeadcdeabaadee34216615aaaabbaaa. 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 = 00000000000000000524156a63950bcedafeadcdeabaadee34216615aaaabbaaa. 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.

Parameter Name	Description
XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_3	Default = 000000000000000000524156a63950bcedafeadcdeabaadee34216615aaaabbaaa. 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 devices, 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 permanently disable the write to FUSE_AES and check CRC for AES key by programming control bit of FUSE. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_USER_KEY_READ	Default = FALSE TRUE permanently disable the write to 32-bit FUSE_USER and read of FUSE_USER key by programming control bit of FUSE. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_SECURE_READ	Default = FALSE TRUE permanently disable the write to FUSE_Secure block and reading of secure block by programming control bit of FUSE. FALSE does not modify this control bit of eFuse.

Macro Name	Description
XSK_EFUSEPL_DISABLE_FUSE_CNTRL_WRITE	Default = FALSE. TRUE permanently disables the write to FUSE_CNTRL block by programming control bit of FUSE. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_RSA_KEY_READ	Default = FALSE. TRUE permanently disables the write to FUSE_RSA block and reading of FUSE_RSA Hash by programming control bit of FUSE. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_KEY_WRITE	Default = FALSE. TRUE permanently disables the write to FUSE_AES block by programming control bit of FUSE. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_USER_KEY_WRITE	Default = FALSE. TRUE permanently disables the write to FUSE_USER block by programming control bit of FUSE. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_SECURE_WRITE	Default = FALSE. TRUE permanently disables the write to FUSE_SECURE block by programming control bit of FUSE. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_RSA_HASH_WRITE	Default = FALSE. TRUE permanently disables the write to FUSE_RSA authentication key by programming control bit of FUSE.FALSE does not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_128BIT_USER_KEY_WRITE	Default = FALSE. TRUE permanently disables 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 permanently allow encrypted bitstream only. FALSE does 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 enables RSA authentication of bitstream FALSE does not modify this secure bit of eFuse.
XSK_EFUSEPL_DISABLE_JTAG_CHAIN	Default = FALSE. TRUE disables JTAG permanently. FALSE does not modify this secure bit of eFuse.
XSK_EFUSEPL_DISABLE_TEST_ACCESS	Default = FALSE. TRUE disables AMD test access. FALSE does not modify this secure bit of eFuse.
XSK_EFUSEPL_DISABLE_AES_DECRYPTOR	Default = FALSE. TRUE disables decoder completely. FALSE does not modify this secure bit of eFuse.
XSK_EFUSEPL_ENABLE_OBFUSCATION_EFUSEAES	Default = FALSE. TRUE enables 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 devices, 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 enables programming SLR config order 0's eFUSE. FALSE disables programming.
XSK_EFUSEPL_PGM_SLR_CONFIG_ORDER_1	Default = FALSE TRUE enables programming SLR config order 1's eFUSE. FALSE disables programming.
XSK_EFUSEPL_PGM_SLR_CONFIG_ORDER_2	Default = FALSE TRUE enables programming SLR config order 2's eFUSE. FALSE disables programming.
XSK_EFUSEPL_PGM_SLR_CONFIG_ORDER_3	Default = FALSE TRUE enables programming SLR config order 3's eFUSE. FALSE disables 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 reads 32-bit FUSE_USER from eFUSE of all available SLRs and each time updates in XilSKey_EPl 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 reads FUSE_USER from eFUSE of all available SLRs and each time updates in XilSKey_EPl 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 reads 128-bit USER key eFUSE of all available SLRs and each time updates in XilSKey_EPl 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/SSI technology 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/SSI technology 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/SSI technology 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/SSI technology 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 performs CRC check of FUSE_AES with provided CRC value in macro XSK_EFUSEPL_CRC_OF_EXPECTED_AES_KEY. And result of CRC check is updated in the 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). 0 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 permanently disable the CRC check of FUSE_AES. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_AES_WR_LOCK	Default = FALSE TRUE permanently disable the writing to FUSE_AES block. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_ENC_ONLY	Default = FALSE TRUE permanently enable encrypted booting only using the Fuse key. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_BBRAM_DISABLE	Default = FALSE TRUE permanently disable the BBRAM key. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_ERR_DISABLE	Default = FALSE TRUE permanently disables the error messages in JTAG status register. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_JTAG_DISABLE	Default = FALSE TRUE permanently disable JTAG controller. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_DFT_DISABLE	Default = FALSE TRUE permanently disable DFT boot mode. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_PROG_GATE_DISABLE	Default = FALSE TRUE permanently disable PROG_GATE feature in PPD. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_SECURE_LOCK	Default = FALSE TRUE permanently disable reboot into JTAG mode when doing a secure lockdown. FALSE does not modify thi s control bit of eFuse.
XSK_EFUSEPS_RSA_ENABLE	Default = FALSE TRUE permanently enable RSA authentication during boot. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_PPK0_WR_LOCK	Default = FALSE TRUE permanently disable writing to PPK0 efuses. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_PPK0_INVLD	Default = FALSE TRUE permanently revoke PPK0. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_PPK1_WR_LOCK	Default = FALSE TRUE permanently disable writing PPK1 efuses. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_PPK1_INVLD	Default = FALSE TRUE permanently revoke PPK1. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_0	Default = FALSE TRUE permanently disable writing to USER_0 efuses. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_1	Default = FALSE TRUE permanently disable writing to USER_1 efuses. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_2	Default = FALSE TRUE permanently disable writing to USER_2 efuses. FALSE does not modify this control bit of eFuse.

Macro Name	Description
XSK_EFUSEPS_USER_WRLK_3	Default = FALSE TRUE permanently disable writing to USER_3 efuses. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_4	Default = FALSE TRUE permanently disable writing to USER_4 efuses. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_5	Default = FALSE TRUE permanently disable writing to USER_5 efuses. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_6	Default = FALSE TRUE permanently disable writing to USER_6 efuses. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_7	Default = FALSE TRUE permanently disable writing to USER_7 efuses. FALSE does not modify this control bit of eFuse.
XSK_EFUSEPS_LBIST_EN	Default = FALSE TRUE permanently enables logic BIST to be run during boot. FALSE does not modify this control bit of eFUSE.
XSK_EFUSEPS_LPD_SC_EN	Default = FALSE TRUE permanently enables zeroization of registers in Low Power Domain(LPD) during boot. FALSE does not modify this control bit of eFUSE.
XSK_EFUSEPS_FPD_SC_EN	Default = FALSE TRUE permanently enables zeroization of registers in Full Power Domain(FPD) during boot. FALSE does not modify this control bit of eFUSE.
XSK_EFUSEPS_PBR_BOOT_ERR	Default = FALSE TRUE permanently enables the boot halt when there is any PMU error. FALSE does 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 burns the AES key provided in XSK_EFUSEPS_AES_KEY. FALSE ignores the key provide XSK_EFUSEPS_AES_KEY.
XSK_EFUSEPS_AES_KEY	Default = 0000000000000000000000000000000000000000000000000000000000000000 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.

Parameter Name	Description
XSK_EFUSEPS_CHECK_AES_KEY_CRC	Default value is FALSE. TRUE checks 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*:** Ensure 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 burns User0 Fuse provided in XSK_EFUSEPS_USER0_FUSES. FALSE ignores the value provided in XSK_EFUSEPS_USER0_FUSES
XSK_EFUSEPS_WRITE_USER1_FUSE	Default = FALSE TRUE burns User1 Fuse provided in XSK_EFUSEPS_USER1_FUSES. FALSE ignores the value provided in XSK_EFUSEPS_USER1_FUSES
XSK_EFUSEPS_WRITE_USER2_FUSE	Default = FALSE TRUE burns User2 Fuse provided in XSK_EFUSEPS_USER2_FUSES. FALSE ignores the value provided in XSK_EFUSEPS_USER2_FUSES
XSK_EFUSEPS_WRITE_USER3_FUSE	Default = FALSE TRUE burns User3 Fuse provided in XSK_EFUSEPS_USER3_FUSES. FALSE ignores the value provided in XSK_EFUSEPS_USER3_FUSES
XSK_EFUSEPS_WRITE_USER4_FUSE	Default = FALSE TRUE burns User4 Fuse provided in XSK_EFUSEPS_USER4_FUSES. FALSE ignores the value provided in XSK_EFUSEPS_USER4_FUSES
XSK_EFUSEPS_WRITE_USER5_FUSE	Default = FALSE TRUE burns User5 Fuse provided in XSK_EFUSEPS_USER5_FUSES. FALSE ignores the value provided in XSK_EFUSEPS_USER5_FUSES

Parameter Name	Description
XSK_EFUSEPS_WRITE_USER6_FUSE	Default = FALSE TRUE burns User6 Fuse provided in XSK_EFUSEPS_USER6_FUSES. FALSE ignores the value provided in XSK_EFUSEPS_USER6_FUSES
XSK_EFUSEPS_WRITE_USER7_FUSE	Default = FALSE TRUE burns User7 Fuse provided in XSK_EFUSEPS_USER7_FUSES. FALSE ignores 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
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

Parameter Name	Description
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 burns PPK0 sha3 hash provided in XSK_EFUSEPS_PPK0_SHA3_HASH. FALSE ignores the hash provided in XSK_EFUSEPS_PPK0_SHA3_HASH.

Parameter Name	Description
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 = 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000 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 burns PPK1 sha3 hash provided in XSK_EFUSEPS_PPK1_SHA3_HASH. FALSE ignores 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 = 0000000000000000000000000000000000000000000000000000000000000000 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 burns SPKID provided in XSK_EFUSEPS_SPK_ID. FALSE ignores 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. Ensure 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 = 00000000000000000000000000000000000000000000000000000000000000000000000000000AES key (in HEX) that must be programmed into BBRAM.
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 does 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 allows only authenticated application. FALSE authentication is not mandatory.
XSK_PUF_REG_MODE	Default = XSK_PUF_MODE4KPUF 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 does not read secure bits.
XSK_PUF_PROGRAM_SECUREBITS	Default = FALSE TRUE programs PUF secure bits based on the user input provided at XSK_PUF_SYN_INVALID, XSK_PUF_SYN_WRLK and XSK_PUF_REGISTER_DISABLE. FALSE does not program any PUF secure bits.
XSK_PUF_SYN_INVALID	Default = FALSE TRUE permanently invalidates the already programmed syndrome data.FALSE will not modify anything
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.

Macro Name	Description
XSK_PUF_AES_KEY	Default = 00000000000000000000000000000000000000000000000000000000000000000000000000000000 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*:** Key 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 239: **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.
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.

Table 239: **Enumeration XSKEfusePl_ErrorCodes Values** *(cont'd)*

Value	Description
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.
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.

Table 239: **Enumeration XSKEfusePl_ErrorCodes Values** *(cont'd)*

Value	Description
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 240: **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.

Table 240: **Enumeration XSKEfusePs_ErrorCodes Values** *(cont'd)*

Value	Description
XSK_EFUSEPS_ERROR_PARAMETER_NULL	0x10 Passed parameter null.
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_INVALID_PARAM	0x17 Error for invalid parameters.
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_PUF_CANT_BE_USED_FOR_USER_DATA	0x00C0 Error when requested for PUF HD eFuses programming for user data, but as Chash is already programmed which means that PUF HD is already programmed with syndrome data
XSK_EFUSEPS_ERROR_PUF_USER_DATA	0x00C2 Error when requested for PUF HD eFuses programming for user data, data provided for Row 0 of efuse page 2 or page 3 or data provided for Row 63 of efuse page 3 is not valid
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.
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.

Table 240: **Enumeration XSKEfusePs_ErrorCodes Values** *(cont'd)*

Value	Description
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.
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.

Table 240: **Enumeration XSKEfusePs_ErrorCodes Values** *(cont'd)*

Value	Description
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.
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.

Table 240: **Enumeration XSKEfusePs_ErrorCodes Values** *(cont'd)*

Value	Description
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
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	0xEA00 Error When PUF Chash already programmed in eFuse.
XSK_EFUSEPS_ERROR_PUF_AUX_ALREADY_PROGRAMMED	0xEB00 Error When PUF AUX already programmed in eFuse.
XSK_EFUSEPS_ERROR_PPK0_BIT_CANT_REVERT	0xEC00 Already programmed PPK0 Hash bit cannot be reverted
XSK_EFUSEPS_ERROR_PPK1_BIT_CANT_REVERT	0xEC00 Already programmed PPK1 Hash bit cannot be reverted
XSK_EFUSEPS_ERROR_CMPLTD_EFUSE_PRGRM_WITH_ERR	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 241: **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.
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.

Table 241: **Enumeration XskZynqMp_Ps_Bbram_ErrorCodes Values** *(cont'd)*

Value	Description
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.

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.

Procedure

This section provides detailed descriptions of the various procedures.

Zynq eFUSE Writing Procedure Running from DDR as an Application

1. This sequence is same as the existing flow described below.
2. Provide the required inputs in `xilskey_input.h`, then compile the platform project.
3. Take the latest FSBL (ELF), stitch the <output>.elf generated to it (using the bootgen utility), and generate a bootable image.
4. Write the generated binary image into the flash device (for example: QSPI, NAND).
5. 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 Structures

id_codes_t

Declaration

```
typedef struct
{
  XSKEfusePl_Fpga flag,
  u32 id,
  u32 irLen,
  char numSlr,
  int masterSlr
} id_codes_t;
```

Table 242: **Structure id_codes_t member description**

Member	Description
flag	Fpga series of Efuse
id	Device ID
irLen	IR length for device
numSlr	Number of SLRs in device
masterSlr	master SLR number

js_command_impl_struct

Declaration

```
typedef struct
{
  js_lib_command_t lib,
  size_t byte_offset,
  js_state_t start_state,
  js_state_t end_state,
  unsigned int write_bytes,
  unsigned int read_bytes,
  unsigned int partial_bytes,
  int state_bits
} js_command_impl_struct;
```

Table 243: **Structure js_command_impl_struct member description**

Member	Description
lib	Command to execute
byte_offset	Intermediatre command processing state
start_state	Start state
end_state	End state

Table 243: **Structure js_command_impl_struct member description** *(cont'd)*

Member	Description
write_bytes	Write bytes
read_bytes	Read bytes
partial_bytes	Partial bytes
state_bits	State bits

js_command_sequence_struct

JTAG command sequence object

Commands are added to this object for later execution.

Declaration

```
typedef struct
{
  js_node_t * node
} js_command_sequence_struct;
```

Table 244: **Structure js_command_sequence_struct member description**

Member	Description
node	JTAG node

js_lib_command_buffer_struct

Declaration

```
typedef struct
{
  js_lib_command_buffer_t * next,
  unsigned char * buf_free,
  unsigned char * buf_end,
  unsigned char buf_start[1]
} js_lib_command_buffer_struct;
```

Table 245: **Structure js_lib_command_buffer_struct member description**

Member	Description
next	Stores next address
buf_free	First available byte in buffer
buf_end	End of buffer
buf_start	Start of buffer (must be last)

js_lib_command_sequence_struct

JTAG command sequence library object

Declaration

```
typedef struct
{
  js_command_sequence_t base,
  js_lib_command_t * cmd_list,
  unsigned int cmd_count,
  unsigned int cmd_max,
  unsigned int cmd_size,
  js_lib_command_buffer_t * buf_head
} js_lib_command_sequence_struct;
```

Table 246: **Structure js_lib_command_sequence_struct member description**

Member	Description
base	Client visible part
cmd_list	Array of commands
cmd_count	Count of added commands
cmd_max	Command list capacity
cmd_size	Size of command objects
buf_head	Command buffer pool

js_lib_command_struct

Declaration

```
typedef struct
{
  js_command_kind_t kind,
  unsigned int flags,
  js_state_t state,
  size_t count,
  unsigned char * tdi_buf,
  unsigned char * tdo_buf
} js_lib_command_struct;
```

Table 247: **Structure js_lib_command_struct member description**

Member	Description
kind	Command to execute
flags	Flags for command
state	State after command
count	Command specific count
tdi_buf	TDI buffer pointer
tdo_buf	TDO buffer pointer

js_lib_port_struct

Declaration

```
typedef struct
{
  js_port_t base,
  int(* get_property)(js_lib_port_t *port, js_property_kind_t kind,
js_property_value_t *valuep),
  int(* set_property)(js_lib_port_t *port, js_property_kind_t kind,
js_property_value_t value),
  int(* run_command_sequence)(js_lib_command_sequence_t *commands),
  int(* close_port)(js_lib_port_t *port)
} js_lib_port_struct;
```

Table 248: **Structure js_lib_port_struct member description**

Member	Description
base	Base class - must be first.
get_property	See `js_get_property()`
set_property	See `js_set_property()`
run_command_sequence	See `js_run_command_sequence()`
close_port	See `js_close_port()`

js_lib_server_struct

JTAG server library object

Declaration

```
typedef struct
{
  js_server_t base,
  int(* get_port_descr_list)(js_lib_server_t *server, js_port_descr_t
**port_listp),
  int(* open_port)(js_lib_server_t *server, js_port_descr_t *port_descr,
js_lib_port_t **port),
  void(* deinit_server)(js_lib_server_t *server),
  char last_error[100]
} js_lib_server_struct;
```

Table 249: **Structure js_lib_server_struct member description**

Member	Description
base	Public part of object
get_port_descr_list	See `js_get_port_descr_list()`
open_port	See `js_open_port()`
deinit_server	See `js_deinit_server()`
last_error	Error of last command

js_lib_state_info_struct

Declaration

```
typedef struct
{
  const char * name,
  unsigned char state_tms[JS_STATE_MAX],
  unsigned char state_clk[JS_STATE_MAX],
  js_state_t stable_state
} js_lib_state_info_struct;
```

Table 250: **Structure js_lib_state_info_struct member description**

Member	Description
name	Name of state
state_tms	TMS bits to move to other state
state_clk	Clocks to move to other state
stable_state	Nearest stable state

js_node_struct

JTAG node information

Nodes typically represent a TAP controller. A node can also represent a collection of TAP controllers, i.e. the whole scan chain or a device containing multiple TAP controllers. For scan chains containing JTAG multiplexers a node can represent the branches on the multiplexer.

Declaration

```
typedef struct
{
  js_port_t * port,
  int is_tap,
  int is_mux,
  int is_branch,
  int is_active,
  uint32_t idcode,
  int irlen,
  const char * name,
  js_node_t ** parent,
  js_node_t ** child_list,
  unsigned int child_count
} js_node_struct;
```

Table 251: **Structure js_node_struct member description**

Member	Description
port	Associated port
is_tap	Node represents one or more TAP controllerst

Table 251: **Structure js_node_struct member description** *(cont'd)*

Member	Description
is_mux	Node is a multiplexer (child nodes are branches)
is_branch	Node is a branch on a multiplexer
is_active	This branch is part of the scan chain
idcode	IDCODE of the TAP (JS_DUMMY_IDCODE if unknown or NA)
irlen	Instruction register length in bits
name	Name of node
parent	Parent node
child_list	Child nodes
child_count	Child nodes count

js_port_descr_struct

JTAG port description

This is retrieved by calling `js_get_port_descr_list()`, and is needed for `js_open_port()` to indicate which port to use.

Declaration

```
typedef struct
{
  char manufacturer[64],
  char serial[64],
  char port[64],
  char description[128],
  void * handle
} js_port_descr_struct;
```

Table 252: **Structure js_port_descr_struct member description**

Member	Description
manufacturer	manufacturer information
serial	Serial information
port	Port information
description	Description information
handle	Pointer to handle

js_port_impl_struct

Declaration

```
typedef struct
{
  js_lib_port_t lib,
  js_lib_command_sequence_t * cmdseq,
  js_state_t state,
  int irPrePadBits,
  int irPostPadBits,
  int drPrePadBits,
  int drPostPadBits,
  js_node_t root_node_obj
} js_port_impl_struct;
```

Table 253: **Structure js_port_impl_struct member description**

Member	Description
lib	Base class - must be first.
cmdseq	Command sequence after normalization
state	Current JTAG state
irPrePadBits	IR pre-pad bits
irPostPadBits	IR post-pad bits
drPrePadBits	DR pre-pad bits
drPostPadBits	DR post-pad bits
root_node_obj	Root node object

js_port_struct

JTAG port

This is retrieved by calling `js_open_port()`.

Declaration

```
typedef struct
{
  js_server_t * server,
  js_node_t * root_node
} js_port_struct;
```

Table 254: **Structure js_port_struct member description**

Member	Description
server	Server associated with port
root_node	Node representing the whole scan chain

js_server_struct

JTAG server object

Server instances are created by implementation specific create functions, typically named js_create_<implementation-name>.

TODO: Add a way to schedule a command sequence to be repeated at regular intervals. This is useful to poll for events, like breakpoint hit.

Declaration

```
typedef struct
{
  char dummy
} js_server_struct;
```

Table 255: **Structure js_server_struct member description**

Member	Description
dummy	dummy

js_zynq

Declaration

```
typedef struct
{
  js_lib_server_t js,
  js_port_descr_t * port_list,
  unsigned int port_count,
  unsigned int port_max
} js_zynq;
```

Table 256: **Structure js_zynq member description**

Member	Description
js	Base class - must be first.
port_list	Port list
port_count	Port count
port_max	Port max

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 257: **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 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 For ZYNQ and 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 For Zynq and Ultrascale
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 For Zynq and Ultrascale
CtrlWrite	If XTRUE will disable eFUSE write to FUSE_CNTRL block in both Zynq and Ultrascale For 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 only For Ultrascale If XTRUE will disable eFUSE write to FUSE_USER block in Ultrascale
UserKeyWrite	only For Ultrascale If XTRUE will disable eFUSE write to FUSE_SEC block in Ultrascale
SecureWrite	only For Ultrascale If XTRUE will disable eFUSE write to FUSE_RSA block in Ultrascale
RSAWrite	only For Ultrascale
User128BitWrite	If TRUE will disable eFUSE write to 128BIT FUSE_USER block in Ultrascale only For Ultrascale
SecureRead	IF XTRUE will disable eFuse read to FUSE_SEC block and also disables eFuse write to FUSE_SEC block in Ultrascale only For Ultrascale
AESKeyExclusive	If XTRUE will force eFUSE key to be used if booting Secure Image In Zynq Only for Zynq
JtagDisable	If XTRUE then permanently sets the Zynq ARM DAP controller in bypass mode in both zynq and ultrascale. for Zynq and Ultrascale
UseAESOnly	If XTRUE will force to use Secure boot with eFUSE key only for both Zynq and Ultrascale For Zynq and Ultrascale
EncryptOnly	If XTRUE will only allow encrypted bitstreams only For Ultrascale only
IntTestAccessDisable	If XTRUE then sets the disable's Xilinx internal test access in Ultrascale Only for Ultrascale
DecoderDisable	If XTRUE then permanently disables the decryptor in Ultrascale Only for Ultrascale
RSAEnable	Enable RSA authentication in ultrascale only for Ultrascale
FuseObfusEn	Enable Obfuscated feature for decryption of eFUSE AES only for Ultrascale
ProgAESandUserLowKey	Following is the define to select if the user wants to select AES key and User Low Key for Zynq Only for Zynq
ProgUserHighKey	Following is the define to select if the user wants to select User Low Key for Zynq Only for Zynq

Table 257: **Structure XilSKey_EPl member description** *(cont'd)*

Member	Description
ProgAESKeyUltra	Following is the define to select if the user wants to select User key for Ultrascale Only for Ultrascale
ProgUserKeyUltra	Following is the define to select if the user wants to select User key for Ultrascale Only for Ultrascale
ProgRSAKeyUltra	Following is the define to select if the user wants to select RSA key for Ultrascale Only for Ultrascale
ProgUser128BitUltra	Following is the define to select if the user wants to program 128-bit User key for Ultrascale Only for Ultrascale
CheckAESKeyUltra	Following is the define to select if the user wants to read AES key for Ultrascale Only for Ultrascale
ReadUserKeyUltra	Following is the define to select if the user wants to read User key for Ultrascale Only for Ultrascale
ReadRSAKeyUltra	Following is the define to select if the user wants to read RSA key for Ultrascale Only for Ultrascale
ReadUser128BitUltra	Following is the define to select if the user wants to read 128-bit User key for Ultrascale Only for Ultrascale
AESKey	This is the REF_CLK value in Hz < u32 RefClk; This is for the aes_key value for both Zynq and Ultrascale
UserKey	This is for the user_key value for both Zynq and Ultrascale
RSAKeyHash	This is for the rsa_key value for Ultrascale Only for Ultrascale
User128Bit	This is for the User 128-bit key value for Ultrascale Only for Ultrascale
JtagMioTDI	TDI MIO Pin Number for ZYNQ Only for ZYNQ
JtagMioTDO	TDO MIO Pin Number for ZYNQ Only for ZYNQ
JtagMioTCK	TCK MIO Pin Number for ZYNQ Only for ZYNQ
JtagMioTMS	TMS MIO Pin Number for ZYNQ Only for ZYNQ
JtagMioMuxSel	MUX Selection MIO Pin Number for ZYNQ Only for ZYNQ
JtagMuxSelLineDefVal	Value on the MUX Selection line for ZYNQ Only for ZYNQ
JtagGpioID	GPIO device ID Only for Ultrascale
HwmGpioStart	Hardware module Start signal's GPIO pin number Only for Ultrascale
HwmGpioReady	Hardware module Ready signal's GPIO pin number
HwmGpioEnd	Hardware module End signal's GPIO pin number Only for Ultrascale
JtagGpioTDI	TDI AXI GPIO pin number for Ultrascale Only for Ultrascale
JtagGpioTDO	TDO AXI GPIO pin number for Ultrascale Only for Ultrascale
JtagGpioTMS	TMS AXI GPIO pin number for Ultrascale Only for Ultrascale
JtagGpioTCK	TCK AXI GPIO pin number for Ultrascale Only for Ultrascale
GpioInputCh	AXI GPIO Channel number of all Inputs TDO Only for Ultrascale
GpioOutPutCh	AXI GPIO Channel number for all Outputs TDI/TMS/TCK Only for Ultrascale
AESKeyReadback	AES key read only for Zynq
UserKeyReadback	User key read in Ultrascale and Zynq for Ultrascale and Zynq
CrcOfAESKey	Expected AES key's CRC for Ultrascale here we can't read AES key directly Only for Ultrascale
AESKeyMatched	Flag is True is AES's CRC is matched, otherwise False Only for Ultrascale
RSAHashReadback	RSA key read back for Ultrascale Only for Ultrascale

Table 257: **Structure XilSKey_EPl member description** *(cont'd)*

Member	Description
User128BitReadBack	User 128-bit key read back for Ultrascale Only for Ultrascale
SystemInitDone	Internal variable to check if timer, XADC and JTAG are initialized.
FpgaFlag	Stores Fpga series of Efuse
CrcToVerify	CRC of AES key to verify programmed AES key Only for Ultrascale
NumSlr	Number of SLRs to iterate through
MasterSlr	Current SLR to iterate through
SlrConfigOrderIndex	Slr configuration order Index

XilSKey_JtagSlr

< `XilSKey_JtagSlr` structure provides information of JtagSlr's

Declaration

```
typedef struct
{
  u32 NumSlr,
  u32 CurSlr,
  u32 IrLen
} XilSKey_JtagSlr;
```

Table 258: **Structure XilSKey_JtagSlr member description**

Member	Description
NumSlr	Number of SLRs to iterate through
CurSlr	Current SLR to iterate through
IrLen	Device IR length

XilSKey_UsrFuses

`XilSKey_UsrFuses` holds the User FUSES which needs to be actually programmed

Declaration

```
typedef struct
{
  u8 UserFuse[XSK_ZYNQMP_EFUSEPS_USER_FUSE_ROW_LEN_IN_BITS]
} XilSKey_UsrFuses;
```

Table 259: **Structure XilSKey_UsrFuses member description**

Member	Description
UserFuse	UserFuse data to be programmed

XilPM Library v5.0

XilPM Zynq UltraScale+ MPSoC APIs

AMD 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 260: **Quick Function Reference**

Type	Name	Arguments
XStatus	XPm_InitXilpm	XIpiPsu * IpiInst
enum `XPmBootStatus`	XPm_GetBootStatus	void
void	XPm_SuspendFinalize	void
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
XStatus	XPm_RequestSuspend	const enum `XPmNodeId` target const enum `XPmRequestAck` ack const u32 latency const u8 state

Table 260: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
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	u32 type u32 subtype
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
XStatus	XPm_FeatureCheck	const enum XPmApiId featureId u32 * version
XStatus	XPm_IsFunctionSupported	const enum XPmApiId apiId const u32 functionId

Table 260: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	XPm_InitSuspendCb	const enum XPmSuspendReason reason const u32 latency const u32 state const u32 timeout
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 const enum XPmResetAction resetaction
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 clk

Table 260: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
XStatus	XPm_ClockDisable	const enum XPmClock clk
XStatus	XPm_ClockGetStatus	const enum XPmClock clk u32 *const status
XStatus	XPm_ClockSetDivider	const enum XPmClock clk const u32 divider
XStatus	XPm_ClockGetDivider	const enum XPmClock clk u32 *const divider
XStatus	XPm_ClockSetParent	const enum XPmClock clk const enum XPmClock parent
XStatus	XPm_ClockGetParent	const enum XPmClock clk enum XPmClock *const parent
XStatus	XPm_ClockSetRate	const enum XPmClock clk const u32 rate
XStatus	XPm_ClockGetRate	const enum XPmClock clk 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_PinCtrlRequest	const u32 pin
XStatus	XPm_PinCtrlRelease	const u32 pin

Table 260: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
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
XStatus	XPm_DevIoctl	const u32 deviceId const pm_ioctl_id ioctlId const u32 arg1 const u32 arg2 u32 *const response

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 261: **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

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 262: **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 263: **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 264: **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 265: **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 266: **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 267: **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 268: **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 269: **XPm_SystemShutdown Arguments**

Type	Name	Description
u32	type	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
u32	subtype	Restart subtype (SYSTEM or PS_ONLY or SUBSYSTEM)

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 270: **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 271: **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 272: **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 273: **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_FeatureCheck

This function queries information about the version of API ID and queries information about supported IOCTL IDs and supported QUERY IDs.

Note: Returns non zero value in version if API is supported or returns 0U in version if API is not supported.

Prototype

```
XStatus XPm_FeatureCheck(const enum XPmApiId featureId, u32 *version);
```

Parameters

The following table lists the `XPm_FeatureCheck` function arguments.

Table 274: **XPm_FeatureCheck Arguments**

Type	Name	Description
const enum `XPmApiId`	featureId	API ID
u32 *	version	Pointer to store API version.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_IsFunctionSupported

This function queries information about the IOCTL/QUERY function ID is supported or not.

Prototype

```
XStatus XPm_IsFunctionSupported(const enum XPmApiId apiId, const u32
functionId);
```

Parameters

The following table lists the `XPm_IsFunctionSupported` function arguments.

Table 275: **XPm_IsFunctionSupported Arguments**

Type	Name	Description
const enum `XPmApiId`	apiId	API ID
const u32	functionId	IOCTL/QUERY ID

Returns

XST_SUCCESS if IOCTL ID/QUERY ID supported 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 276: **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

Table 276: **XPm_InitSuspendCb Arguments** *(cont'd)*

Type	Name	Description
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.
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 277: **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 278: **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 279: **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 off (powered down),
    - 1 : if CPU is active (powered up),
    - 2 : if CPU is in sleep (powered down),
    - 3 : 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 280: **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: Power value is not actual power consumption of device. It is default dummy power value which is fixed in PMUFW. Temperature type is not supported for ZynqMP.

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 281: **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 micro seconds 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 282: **XPm_ResetAssert Arguments**

Type	Name	Description
const enum `XPmReset`	reset	ID of the reset line
const enum `XPmResetAction`	resetaction	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 283: **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 284: **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 285: **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 286: **XPm_MmioWrite Arguments**

Type	Name	Description
const u32	address	Physical 32-bit address of memory mapped register to write to.

Table 286: **XPm_MmioWrite Arguments** *(cont'd)*

Type	Name	Description
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 287: **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 clk);
```

Parameters

The following table lists the `XPm_ClockEnable` function arguments.

Table 288: **XPm_ClockEnable Arguments**

Type	Name	Description
const enum `XPmClock`	clk	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 clk);
```

Parameters

The following table lists the `XPm_ClockDisable` function arguments.

Table 289: **XPm_ClockDisable Arguments**

Type	Name	Description
const enum `XPmClock`	clk	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 clk, u32 *const status);
```

Parameters

The following table lists the `XPm_ClockGetStatus` function arguments.

Table 290: **XPm_ClockGetStatus Arguments**

Type	Name	Description
const enum XPmClock	clk	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_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 clk, const u32 divider);
```

Parameters

The following table lists the `XPm_ClockSetDivider` function arguments.

Table 291: **XPm_ClockSetDivider Arguments**

Type	Name	Description
const enum XPmClock	clk	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_ClockGetDivider

Call this function to get divider of a clock.

Prototype

```
XStatus XPm_ClockGetDivider(const enum XPmClock clk, u32 *const divider);
```

Parameters

The following table lists the `XPm_ClockGetDivider` function arguments.

Table 292: **XPm_ClockGetDivider Arguments**

Type	Name	Description
const enum XPmClock	clk	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 clk, const enum XPmClock
parent);
```

Parameters

The following table lists the `XPm_ClockSetParent` function arguments.

Table 293: **XPm_ClockSetParent Arguments**

Type	Name	Description
const enum XPmClock	clk	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 clk, enum XPmClock *const
parent);
```

Parameters

The following table lists the `XPm_ClockGetParent` function arguments.

Table 294: **XPm_ClockGetParent Arguments**

Type	Name	Description
const enum XPmClock	clk	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 clk, const u32 rate);
```

Parameters

The following table lists the `XPm_ClockSetRate` function arguments.

Table 295: **XPm_ClockSetRate Arguments**

Type	Name	Description
const enum XPmClock	clk	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 clk, u32 *const rate);
```

Parameters

The following table lists the `XPm_ClockGetRate` function arguments.

Table 296: **XPm_ClockGetRate Arguments**

Type	Name	Description
const enum XPmClock	clk	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 297: **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 298: **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 299: **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 300: **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_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 301: **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 302: **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 303: **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 304: **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 305: **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 306: **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

XPm_DevIoctl

This function performs driver-like IOCTL functions on shared system devices.

Prototype

```
XStatus XPm_DevIoctl(const u32 deviceId, const pm_ioctl_id ioctlId, const
u32 arg1, const u32 arg2, u32 *const response);
```

Parameters

The following table lists the `XPm_DevIoctl` function arguments.

Table 307: **XPm_DevIoctl Arguments**

Type	Name	Description
const u32	deviceId	ID of the device
const pm_ioctl_id	ioctlId	IOCTL function ID
const u32	arg1	Argument 1
const u32	arg2	Argument 2
u32 *const	response	Ioctl response

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Enumerations

Enumeration XPmApiCbId

PM API Callback ID

Table 308: **Enumeration XPmApiCbId Values**

Value	Description
PM_INIT_SUSPEND_CB	Suspend callback
PM_ACKNOWLEDGE_CB	Acknowledge callback
PM_NOTIFY_CB	Notify callback
PM_NOTIFY_STL_NO_OP	Notify STL No OP

Enumeration XPmNodeId

PM Node ID

Table 309: **Enumeration XPmNodeId Values**

Value	Description
NODE_UNKNOWN	0x0
NODE_APU	0x1
NODE_APU_0	0x2
NODE_APU_1	0x3
NODE_APU_2	0x4
NODE_APU_3	0x5
NODE_RPU	0x6
NODE_RPU_0	0x7
NODE_RPU_1	0x8
NODE_PLD	0x9
NODE_FPD	0xA
NODE_OCM_BANK_0	0xB
NODE_OCM_BANK_1	0xC
NODE_OCM_BANK_2	0xD
NODE_OCM_BANK_3	0xE
NODE_TCM_0_A	0xF
NODE_TCM_0_B	0x10
NODE_TCM_1_A	0x11
NODE_TCM_1_B	0x12
NODE_L2	0x13
NODE_GPU_PP_0	0x14

Table 309: **Enumeration XPmNodeId Values** *(cont'd)*

Value	Description
NODE_GPU_PP_1	0x15
NODE_USB_0	0x16
NODE_USB_1	0x17
NODE_TTC_0	0x18
NODE_TTC_1	0x19
NODE_TTC_2	0x1A
NODE_TTC_3	0x1B
NODE_SATA	0x1C
NODE_ETH_0	0x1D
NODE_ETH_1	0x1E
NODE_ETH_2	0x1F
NODE_ETH_3	0x20
NODE_UART_0	0x21
NODE_UART_1	0x22
NODE_SPI_0	0x23
NODE_SPI_1	0x24
NODE_I2C_0	0x25
NODE_I2C_1	0x26
NODE_SD_0	0x27
NODE_SD_1	0x28
NODE_DP	0x29
NODE_GDMA	0x2A
NODE_ADMA	0x2B
NODE_NAND	0x2C
NODE_QSPI	0x2D
NODE_GPIO	0x2E
NODE_CAN_0	0x2F
NODE_CAN_1	0x30
NODE_EXTERN	0x31
NODE_APLL	0x32
NODE_VPLL	0x33
NODE_DPLL	0x34
NODE_RPLL	0x35
NODE_IOPLL	0x36
NODE_DDR	0x37
NODE_IPI_APU	0x38
NODE_IPI_RPU_0	0x39
NODE_GPU	0x3A
NODE_PCIE	0x3B

Table 309: **Enumeration XPmNodeId Values** *(cont'd)*

Value	Description
NODE_PCAP	0x3C
NODE_RTC	0x3D
NODE_LPD	0x3E
NODE_VCU	0x3F
NODE_IPI_RPU_1	0x40
NODE_IPI_PL_0	0x41
NODE_IPI_PL_1	0x42
NODE_IPI_PL_2	0x43
NODE_IPI_PL_3	0x44
NODE_PL	0x45
NODE_ID_MAX	0x46

Enumeration XPmRequestAck

PM Acknowledge Request

Table 310: **Enumeration XPmRequestAck Values**

Value	Description
REQUEST_ACK_NO	No Ack
REQUEST_ACK_BLOCKING	Blocking Ack
REQUEST_ACK_NON_BLOCKING	Non blocking Ack
REQUEST_ACK_CB_CERROR	Callback Error

Enumeration XPmAbortReason

PM Abort Reasons

Table 311: **Enumeration XPmAbortReason Values**

Value	Description
ABORT_REASON_WKUP_EVENT	Wakeup Event
ABORT_REASON_PU_BUSY	Processor Busy
ABORT_REASON_NO_PWRDN	No Powerdown
ABORT_REASON_UNKNOWN	Unknown Reason

Enumeration XPmSuspendReason

PM Suspend Reasons

Table 312: **Enumeration XPmSuspendReason Values**

Value	Description
SUSPEND_REASON_PU_REQ	Processor request
SUSPEND_REASON_ALERT	Alert
SUSPEND_REASON_SYS_SHUTDOWN	System shutdown

Enumeration XPmRamState

PM RAM States

Table 313: **Enumeration XPmRamState Values**

Value	Description
PM_RAM_STATE_OFF	Off
PM_RAM_STATE_RETENTION	Retention
PM_RAM_STATE_ON	On

Definitions

Define PM_VERSION_MAJOR

Definition

```
#define PM_VERSION_MAJOR1
```

Description

PM Version Number macros

Define PM_VERSION_MINOR

Definition

```
#define PM_VERSION_MINOR1
```

Description

PM Version Number macros

Define PM_VERSION

Definition

```
#define PM_VERSION((
                PM_VERSION_MAJOR
             << 16) |
                PM_VERSION_MINOR
            )
```

Description

PM Version Number macros

Define PM_API_BASE_VERSION

Definition

```
#define PM_API_BASE_VERSION(1U)
```

Description

PM API versions

Define PM_API_VERSION_2

Definition

```
#define PM_API_VERSION_2(2U)
```

Description

PM API versions

Define PM_CAP_ACCESS

Definition

```
#define PM_CAP_ACCESS0x1U
```

Description

Capabilities for RAM

Define PM_CAP_CONTEXT

Definition

```
#define PM_CAP_CONTEXT0x2U
```

Description

Capabilities for RAM

Define PM_CAP_WAKEUP

Definition

```
#define PM_CAP_WAKEUP0x4U
```

Description

Capabilities for RAM

Define NODE_STATE_OFF

Definition

```
#define NODE_STATE_OFF0
```

Description

Node State

Define NODE_STATE_ON

Definition

```
#define NODE_STATE_ON1
```

Description

Node State

Define PROC_STATE_FORCEDOFF

Definition

```
#define PROC_STATE_FORCEDOFF0
```

Description

Processor's state

Define PROC_STATE_ACTIVE

Definition

```
#define PROC_STATE_ACTIVE1
```

Description

Processor's state

Define PROC_STATE_SLEEP

Definition

```
#define PROC_STATE_SLEEP2
```

Description

Processor's state

Define PROC_STATE_SUSPENDING

Definition

```
#define PROC_STATE_SUSPENDING3
```

Description

Processor's state

Define MAX_LATENCY

Definition

```
#define MAX_LATENCY(~0U)
```

Description

Maximum Latency/QOS

Define MAX_QOS

Definition

```
#define MAX_QOS100U
```

Description

Maximum Latency/QOS

Define PMF_SHUTDOWN_TYPE_SHUTDOWN

Definition

```
#define PMF_SHUTDOWN_TYPE_SHUTDOWN0U
```

Description

System shutdown/Restart

Define PMF_SHUTDOWN_TYPE_RESET

Definition

```
#define PMF_SHUTDOWN_TYPE_RESET1U
```

Description

System shutdown/Restart

Define PMF_SHUTDOWN_SUBTYPE_SUBSYSTEM

Definition

```
#define PMF_SHUTDOWN_SUBTYPE_SUBSYSTEM0U
```

Description

System shutdown/Restart

Define PMF_SHUTDOWN_SUBTYPE_PS_ONLY

Definition

```
#define PMF_SHUTDOWN_SUBTYPE_PS_ONLY1U
```

Description

System shutdown/Restart

Define PMF_SHUTDOWN_SUBTYPE_SYSTEM

Definition

```
#define PMF_SHUTDOWN_SUBTYPE_SYSTEM2U
```

Description

System shutdown/Restart

Define PM_CLOCK_DIV0_ID

Definition

```
#define PM_CLOCK_DIV0_ID0U
```

Description

Define PM_CLOCK_DIV1_ID

Definition

```
#define PM_CLOCK_DIV1_ID1U
```

Description

Define PM_API_MIN

Definition

```
#define PM_API_MIN
                PM_GET_API_VERSION
```

Description

XilPM Versal Adaptive SoC APIs

AMD Power Management (XilPM) provides Embedded Energy Management Interface (EEMI) APIs for power management on Versal devices. For more details about EEMI, see the Embedded Energy Management Interface (EEMI) API User Guide (UG1200).

The platform and power management functionality used by the APU/RPU applications is provided by the files in the 'XilPM<version>/versal/client' folder, where '<version>' is the version of the XilPM library.

Enumerations

Enumeration XPmAbortReason

PM abort reasons

Table 314: **Enumeration XPmAbortReason Values**

Value	Description
ABORT_REASON_WKUP_EVENT	Wakeup Event
ABORT_REASON_PU_BUSY	Processor Busy
ABORT_REASON_NO_PWRDN	No Powerdown
ABORT_REASON_UNKNOWN	Unknown Reason

Enumeration XPmBootStatus

Boot status enumeration.

Table 315: **Enumeration XPmBootStatus Values**

Value	Description
PM_INITIAL_BOOT	boot is a fresh system startup
PM_RESUME	boot is a resume
PM_BOOT_ERROR	error, boot cause cannot be identified

Enumeration XPmRequestAck

PM Acknowledge Request Types

Table 316: **Enumeration XPmRequestAck Values**

Value	Description
REQUEST_ACK_NO	No Ack
REQUEST_ACK_BLOCKING	Blocking Ack
REQUEST_ACK_NON_BLOCKING	Non blocking Ack
REQUEST_ACK_CB_CERROR	Callback Error

Enumeration XPmCapability

Device capability requirements enumeration.

Table 317: **Enumeration XPmCapability Values**

Value	Description
PM_CAP_ACCESS	Full access
PM_CAP_CONTEXT	Configuration and contents retained
PM_CAP_WAKEUP	Enabled as a wake-up source
PM_CAP_UNUSABLE	Not usable
PM_CAP_SECURE	Secure access type (non-secure/secure)
PM_CAP_COHERENT	Device Coherency
PM_CAP_VIRTUALIZED	Device Virtualization

Enumeration XPmDeviceUsage

Usage status, returned by PmGetNodeStatus

Table 318: **Enumeration XPmDeviceUsage Values**

Value	Description
PM_USAGE_CURRENT_SUBSYSTEM	Current subsystem is using
PM_USAGE_OTHER_SUBSYSTEM	Other subsystem is using

Enumeration XPmResetActions

Reset configuration argument

Table 319: **Enumeration XPmResetActions Values**

Value	Description
PM_RESET_ACTION_RELEASE	Reset action release
PM_RESET_ACTION_ASSERT	Reset action assert
PM_RESET_ACTION_PULSE	Reset action pulse

Enumeration XPmSuspendReason

Suspend reasons

Table 320: **Enumeration XPmSuspendReason Values**

Value	Description
SUSPEND_REASON_PU_REQ	Processor request
SUSPEND_REASON_ALERT	Alert
SUSPEND_REASON_SYS_SHUTDOWN	System shutdown

Enumeration XPmApiCbId_t

PM API callback IDs

Table 321: **Enumeration XPmApiCbId_t Values**

Value	Description
PM_INIT_SUSPEND_CB	Suspend callback
PM_ACKNOWLEDGE_CB	Acknowledge callback
PM_NOTIFY_CB	Notify callback

Enumeration PmPinFunIds

Pin Function IDs

Table 322: **Enumeration PmPinFunIds Values**

Value	Description
PIN_FUNC_SPI0	Pin function ID of SPI0
PIN_FUNC_SPI0_SS	Pin function ID of SPI0_SS
PIN_FUNC_SPI1	Pin function ID of SPI1
PIN_FUNC_SPI1_SS	Pin function ID of SPI1_SS
PIN_FUNC_CAN0	Pin function ID of CAN0
PIN_FUNC_CAN1	Pin function ID of CAN1
PIN_FUNC_I2C0	Pin function ID of I2C0
PIN_FUNC_I2C1	Pin function ID of I2C1
PIN_FUNC_I2C_PMC	Pin function ID of I2C_PMC
PIN_FUNC_TTC0_CLK	Pin function ID of TTC0_CLK
PIN_FUNC_TTC0_WAV	Pin function ID of TTC0_WAV
PIN_FUNC_TTC1_CLK	Pin function ID of TTC1_CLK
PIN_FUNC_TTC1_WAV	Pin function ID of TTC1_WAV
PIN_FUNC_TTC2_CLK	Pin function ID of TTC2_CLK

Table 322: **Enumeration PmPinFunIds Values** *(cont'd)*

Value	Description
PIN_FUNC_TTC2_WAV	Pin function ID of TTC2_WAV
PIN_FUNC_TTC3_CLK	Pin function ID of TTC3_CLK
PIN_FUNC_TTC3_WAV	Pin function ID of TTC3_WAV
PIN_FUNC_WWDT0	Pin function ID of WWDT0
PIN_FUNC_WWDT1	Pin function ID of WWDT1
PIN_FUNC_SYSMON_I2C0	Pin function ID of SYSMON_I2C0
PIN_FUNC_SYSMON_I2C0_ALERT	Pin function ID of SYSMON_I2C0_AL
PIN_FUNC_UART0	Pin function ID of UART0
PIN_FUNC_UART0_CTRL	Pin function ID of UART0_CTRL
PIN_FUNC_UART1	Pin function ID of UART1
PIN_FUNC_UART1_CTRL	Pin function ID of UART1_CTRL
PIN_FUNC_GPIO0	Pin function ID of GPIO0
PIN_FUNC_GPIO1	Pin function ID of GPIO1
PIN_FUNC_GPIO2	Pin function ID of GPIO2
PIN_FUNC_EMIO0	Pin function ID of EMIO0
PIN_FUNC_GEM0	Pin function ID of GEM0
PIN_FUNC_GEM1	Pin function ID of GEM1
PIN_FUNC_TRACE0	Pin function ID of TRACE0
PIN_FUNC_TRACE0_CLK	Pin function ID of TRACE0_CLK
PIN_FUNC_MDIO0	Pin function ID of MDIO0
PIN_FUNC_MDIO1	Pin function ID of MDIO1
PIN_FUNC_GEM_TSU0	Pin function ID of GEM_TSU0
PIN_FUNC_PCIE0	Pin function ID of PCIE0
PIN_FUNC_SMAP0	Pin function ID of SMAP0
PIN_FUNC_USB0	Pin function ID of USB0
PIN_FUNC_SD0	Pin function ID of SD0
PIN_FUNC_SD0_PC	Pin function ID of SD0_PC
PIN_FUNC_SD0_CD	Pin function ID of SD0_CD
PIN_FUNC_SD0_WP	Pin function ID of SD0_WP
PIN_FUNC_SD1	Pin function ID of SD1
PIN_FUNC_SD1_PC	Pin function ID of SD1_PC
PIN_FUNC_SD1_CD	Pin function ID of SD1_CD
PIN_FUNC_SD1_WP	Pin function ID of SD1_WP
PIN_FUNC_OSPI0	Pin function ID of OSPI0
PIN_FUNC_OSPI0_SS	Pin function ID of OSPI0_SS
PIN_FUNC_QSPI0	Pin function ID of QSPI0
PIN_FUNC_QSPI0_FBCLK	Pin function ID of QSPI0_FBCLK
PIN_FUNC_QSPI0_SS	Pin function ID of QSPI0_SS
PIN_FUNC_TEST_CLK	Pin function ID of TEST_CLK

Table 322: **Enumeration PmPinFunIds Values** *(cont'd)*

Value	Description
PIN_FUNC_TEST_SCAN	Pin function ID of TEST_SCAN
PIN_FUNC_TAMPER_TRIGGER	Pin function ID of TAMPER_TRIGGER
MAX_FUNCTION	Max Pin function

Enumeration pm_pinctrl_config_param

Pin Control Configuration

Table 323: **Enumeration pm_pinctrl_config_param Values**

Value	Description
PINCTRL_CONFIG_SLEW_RATE	Pin config slew rate
PINCTRL_CONFIG_BIAS_STATUS	Pin config bias status
PINCTRL_CONFIG_PULL_CTRL	Pin config pull control
PINCTRL_CONFIG_SCHMITT_CMOS	Pin config schmitt CMOS
PINCTRL_CONFIG_DRIVE_STRENGTH	Pin config drive strength
PINCTRL_CONFIG_VOLTAGE_STATUS	Pin config voltage status
PINCTRL_CONFIG_TRI_STATE	Pin config tri state
PINCTRL_CONFIG_MAX	Max Pin config

Enumeration pm_pinctrl_slew_rate

Pin Control Slew Rate

Table 324: **Enumeration pm_pinctrl_slew_rate Values**

Value	Description
PINCTRL_SLEW_RATE_FAST	Fast slew rate
PINCTRL_SLEW_RATE_SLOW	Slow slew rate

Enumeration pm_pinctrl_bias_status

Pin Control Bias Status

Table 325: **Enumeration pm_pinctrl_bias_status Values**

Value	Description
PINCTRL_BIAS_DISABLE	Bias disable
PINCTRL_BIAS_ENABLE	Bias enable

Enumeration pm_pinctrl_pull_ctrl

Pin Control Pull Control

Table 326: **Enumeration pm_pinctrl_pull_ctrl Values**

Value	Description
PINCTRL_BIAS_PULL_DOWN	Bias pull-down
PINCTRL_BIAS_PULL_UP	Bias pull-up

Enumeration pm_pinctrl_schmitt_cmos

Pin Control Input Type

Table 327: **Enumeration pm_pinctrl_schmitt_cmos Values**

Value	Description
PINCTRL_INPUT_TYPE_CMOS	Input type CMOS
PINCTRL_INPUT_TYPE_SCHMITT	Input type SCHMITT

Enumeration pm_pinctrl_drive_strength

Pin Control Drive Strength

Table 328: **Enumeration pm_pinctrl_drive_strength Values**

Value	Description
PINCTRL_DRIVE_STRENGTH_TRISTATE	tri-state
PINCTRL_DRIVE_STRENGTH_4MA	4mA
PINCTRL_DRIVE_STRENGTH_8MA	8mA
PINCTRL_DRIVE_STRENGTH_12MA	12mA
PINCTRL_DRIVE_STRENGTH_MAX	Max value

Enumeration pm_pinctrl_tri_state

Pin Control Tri State

Table 329: **Enumeration pm_pinctrl_tri_state Values**

Value	Description
PINCTRL_TRI_STATE_DISABLE	Tri state disable
PINCTRL_TRI_STATE_ENABLE	Tri state enable

Enumeration XPm_PllConfigParams

PLL parameters

Table 330: **Enumeration XPm_PllConfigParams Values**

Value	Description
PM_PLL_PARAM_ID_DIV2	PLL param ID DIV2
PM_PLL_PARAM_ID_FBDIV	PLL param ID FBDIV
PM_PLL_PARAM_ID_DATA	PLL param ID DATA
PM_PLL_PARAM_ID_PRE_SRC	PLL param ID PRE_SRC
PM_PLL_PARAM_ID_POST_SRC	PLL param ID POST_SRC
PM_PLL_PARAM_ID_LOCK_DLY	PLL param ID LOCK_DLY
PM_PLL_PARAM_ID_LOCK_CNT	PLL param ID LOCK_CNT
PM_PLL_PARAM_ID_LFHF	PLL param ID LFHF
PM_PLL_PARAM_ID_CP	PLL param ID CP
PM_PLL_PARAM_ID_RES	PLL param ID RES
PM_PLL_PARAM_MAX	PLL param ID max

Enumeration XPmPllMode

PLL modes

Table 331: **Enumeration XPmPllMode Values**

Value	Description
PM_PLL_MODE_INTEGER	PLL mode integer
PM_PLL_MODE_FRACTIONAL	PLL mode fractional
PM_PLL_MODE_RESET	PLL mode reset

Enumeration XPmInitFunctions

PM init node functions

Table 332: **Enumeration XPmInitFunctions Values**

Value	Description
FUNC_INIT_START	Function ID INIT_START
FUNC_INIT_FINISH	Function ID INIT_FINISH
FUNC_SCAN_CLEAR	Function ID SCAN_CLEAR
FUNC_BISR	Function ID BISR
FUNC_LBIST	Function ID LBIST
FUNC_MEM_INIT	Function ID MEM_INIT

Table 332: **Enumeration XPmInitFunctions Values** *(cont'd)*

Value	Description
FUNC_MBIST_CLEAR	Function ID MBIST_CLEAR
FUNC_HOUSECLEAN_PL	Function ID HOUSECLEAN_PL
FUNC_HOUSECLEAN_COMPLETE	Function ID HOUSECLEAN_COMPLETE
FUNC_MIO_FLUSH	Function ID MIO FLUSH
FUNC_MEM_CTRLR_MAP	Function ID MEM_CTRLR_MAP
FUNC_MAX_COUNT_PMINIT	Function ID MAX

Enumeration XPmNotifyEvent

PM notify events

Table 333: **Enumeration XPmNotifyEvent Values**

Value	Description
EVENT_STATE_CHANGE	State change event
EVENT_ZERO_USERS	Zero user event
EVENT_CPU_IDLE_FORCE_PWRDWN	CPU idle event during force power down

Definitions

Define PM_VERSION_MAJOR

Definition

```
#define PM_VERSION_MAJOR1UL
```

Description

PM Version Number

Define PM_VERSION_MINOR

Definition

```
#define PM_VERSION_MINOR0UL
```

Description

PM Version Number

Define PM_VERSION

Definition

```
#define PM_VERSION((
                PM_VERSION_MAJOR
             << 16) |
                PM_VERSION_MINOR
            )
```

Description

PM Version Number

Define ABORT_REASON_MIN

Definition

```
#define ABORT_REASON_MIN

                ABORT_REASON_WKUP_EVENT
```

Description

PM Abort Reasons

Define ABORT_REASON_MAX

Definition

```
#define ABORT_REASON_MAX

                ABORT_REASON_UNKNOWN
```

Description

PM Abort Reasons

Define XPM_MAX_CAPABILITY

Definition

```
#define XPM_MAX_CAPABILITY((u32)
                PM_CAP_ACCESS
              | (u32)
                PM_CAP_CONTEXT
              | (u32)
                PM_CAP_WAKEUP
            )
```

Description

Requirement limits

Define XPM_MAX_LATENCY

Definition

```
#define XPM_MAX_LATENCY(0xFFFFU)
```

Description

Requirement limits

Define XPM_MAX_QOS

Definition

```
#define XPM_MAX_QOS(100U)
```

Description

Requirement limits

Define XPM_MIN_CAPABILITY

Definition

```
#define XPM_MIN_CAPABILITY(0U)
```

Description

Requirement limits

Define XPM_MIN_LATENCY

Definition

```
#define XPM_MIN_LATENCY(0U)
```

Description

Requirement limits

Define XPM_MIN_QOS

Definition

```
#define XPM_MIN_QOS(0U)
```

Description

Requirement limits

Define XPM_DEF_CAPABILITY

Definition

```
#define XPM_DEF_CAPABILITY
                XPM_MAX_CAPABILITY
```

Description

Requirement limits

Define XPM_DEF_LATENCY

Definition

```
#define XPM_DEF_LATENCY
                XPM_MAX_LATENCY
```

Description

Requirement limits

Define XPM_DEF_QOS

Definition

```
#define XPM_DEF_QOS
                XPM_MAX_QOS
```

Description

Requirement limits

Define NODE_STATE_OFF

Definition

```
#define NODE_STATE_OFF(0U)
```

Description

Device node status

Define NODE_STATE_ON

Definition

```
#define NODE_STATE_ON(1U)
```

Description

Device node status

Define PROC_STATE_SLEEP

Definition

```
#define PROC_STATE_SLEEP
                NODE_STATE_OFF
```

Description

Processor node status

Define PROC_STATE_ACTIVE

Definition

```
#define PROC_STATE_ACTIVE
                NODE_STATE_ON
```

Description

Processor node status

Define PROC_STATE_FORCEDOFF

Definition

```
#define PROC_STATE_FORCEDOFF(7U)
```

Description

Processor node status

Define PROC_STATE_SUSPENDING

Definition

```
#define PROC_STATE_SUSPENDING(8U)
```

Description

Processor node status

Define PM_SHUTDOWN_TYPE_SHUTDOWN

Definition

```
#define PM_SHUTDOWN_TYPE_SHUTDOWN(0U)
```

Description

System shutdown macros

Define PM_SHUTDOWN_TYPE_RESET

Definition

```
#define PM_SHUTDOWN_TYPE_RESET(1U)
```

Description

System shutdown macros

Define PM_SHUTDOWN_SUBTYPE_RST_SUBSYSTEM

Definition

```
#define PM_SHUTDOWN_SUBTYPE_RST_SUBSYSTEM(0U)
```

Description

System shutdown macros

Define PM_SHUTDOWN_SUBTYPE_RST_PS_ONLY

Definition

```
#define PM_SHUTDOWN_SUBTYPE_RST_PS_ONLY(1U)
```

Description

System shutdown macros

Define PM_SHUTDOWN_SUBTYPE_RST_SYSTEM

Definition

```
#define PM_SHUTDOWN_SUBTYPE_RST_SYSTEM(2U)
```

Description

System shutdown macros

Define PM_SUSPEND_STATE_CPU_IDLE

Definition

```
#define PM_SUSPEND_STATE_CPU_IDLE0x0U
```

Description

State arguments of the self suspend

Define PM_SUSPEND_STATE_SUSPEND_TO_RAM

Definition

```
#define PM_SUSPEND_STATE_SUSPEND_TO_RAM0xFU
```

Description

State arguments of the self suspend

Define XPM_RPU_MODE_LOCKSTEP

Definition

```
#define XPM_RPU_MODE_LOCKSTEP0U
```

Description

RPU operation mode

Define XPM_RPU_MODE_SPLIT

Definition

```
#define XPM_RPU_MODE_SPLIT1U
```

Description

RPU operation mode

Define XPM_RPU_BOOTMEM_LOVEC

Definition

```
#define XPM_RPU_BOOTMEM_LOVEC(0U)
```

Description

RPU Boot memory

Define XPM_RPU_BOOTMEM_HIVEC

Definition

```
#define XPM_RPU_BOOTMEM_HIVEC(1U)
```

Description

RPU Boot memory

Define XPM_RPU_TCM_SPLIT

Definition

```
#define XPM_RPU_TCM_SPLIT0U
```

Description

RPU TCM mode

Define XPM_RPU_TCM_COMB

Definition

```
#define XPM_RPU_TCM_COMB1U
```

Description

RPU TCM mode

Define XPM_BOOT_HEALTH_STATUS_MASK

Definition

```
#define XPM_BOOT_HEALTH_STATUS_MASK(0x1U)
```

Description

Boot health status mask

Define XPM_TAPDELAY_QSPI

Definition

```
#define XPM_TAPDELAY_QSPI(2U)
```

Description

Tap delay signal type

Define XPM_TAPDELAY_BYPASS_DISABLE

Definition

```
#define XPM_TAPDELAY_BYPASS_DISABLE(0U)
```

Description

Tap delay bypass

Define XPM_TAPDELAY_BYPASS_ENABLE

Definition

```
#define XPM_TAPDELAY_BYPASS_ENABLE(1U)
```

Description

Tap delay bypass

Define XPM_OSPI_MUX_SEL_DMA

Definition

```
#define XPM_OSPI_MUX_SEL_DMA(0U)
```

Description

Ospi AXI Mux select

Define XPM_OSPI_MUX_SEL_LINEAR

Definition

```
#define XPM_OSPI_MUX_SEL_LINEAR(1U)
```

Description

Ospi AXI Mux select

Define XPM_OSPI_MUX_GET_MODE

Definition

```
#define XPM_OSPI_MUX_GET_MODE(2U)
```

Description

Ospi AXI Mux select

Define XPM_TAPDELAY_INPUT

Definition

```
#define XPM_TAPDELAY_INPUT(0U)
```

Description

Tap delay type

Define XPM_TAPDELAY_OUTPUT

Definition

```
#define XPM_TAPDELAY_OUTPUT(1U)
```

Description

Tap delay type

Define XPM_DLL_RESET_ASSERT

Definition

```
#define XPM_DLL_RESET_ASSERT(0U)
```

Description

Dll reset type

Define XPM_DLL_RESET_RELEASE

Definition

```
#define XPM_DLL_RESET_RELEASE(1U)
```

Description

Dll reset type

Define XPM_DLL_RESET_PULSE

Definition

```
#define XPM_DLL_RESET_PULSE(2U)
```

Description

Dll reset type

Define XPM_RESET_REASON_EXT_POR

Definition

```
#define XPM_RESET_REASON_EXT_POR(0U)
```

Description

Reset Reason

Define XPM_RESET_REASON_SW_POR

Definition

```
#define XPM_RESET_REASON_SW_POR(1U)
```

Description

Reset Reason

Define XPM_RESET_REASON_SLR_POR

Definition

```
#define XPM_RESET_REASON_SLR_POR(2U)
```

Description

Reset Reason

Define XPM_RESET_REASON_ERR_POR

Definition

```
#define XPM_RESET_REASON_ERR_POR(3U)
```

Description

Reset Reason

Define XPM_RESET_REASON_DAP_SRST

Definition

```
#define XPM_RESET_REASON_DAP_SRST(7U)
```

Description

Reset Reason

Define XPM_RESET_REASON_ERR_SRST

Definition

```
#define XPM_RESET_REASON_ERR_SRST(8U)
```

Description

Reset Reason

Define XPM_RESET_REASON_SW_SRST

Definition

```
#define XPM_RESET_REASON_SW_SRST(9U)
```

Description

Reset Reason

Define XPM_RESET_REASON_SLR_SRST

Definition

```
#define XPM_RESET_REASON_SLR_SRST(10U)
```

Description

Reset Reason

Define XPM_RESET_REASON_INVALID

Definition

```
#define XPM_RESET_REASON_INVALID(0xFFU)
```

Description

Reset Reason

Define XPM_PROBE_COUNTER_TYPE_LAR_LSR

Definition

```
#define XPM_PROBE_COUNTER_TYPE_LAR_LSR(0U)
```

Description

Probe Counter Type

Define XPM_PROBE_COUNTER_TYPE_MAIN_CTL

Definition

```
#define XPM_PROBE_COUNTER_TYPE_MAIN_CTL(1U)
```

Description

Probe Counter Type

Define XPM_PROBE_COUNTER_TYPE_CFG_CTL

Definition

```
#define XPM_PROBE_COUNTER_TYPE_CFG_CTL(2U)
```

Description

Probe Counter Type

Define XPM_PROBE_COUNTER_TYPE_STATE_PERIOD

Definition

```
#define XPM_PROBE_COUNTER_TYPE_STATE_PERIOD(3U)
```

Description

Probe Counter Type

Define XPM_PROBE_COUNTER_TYPE_PORT_SEL

Definition

```
#define XPM_PROBE_COUNTER_TYPE_PORT_SEL(4U)
```

Description

Probe Counter Type

Define XPM_PROBE_COUNTER_TYPE_SRC

Definition

```
#define XPM_PROBE_COUNTER_TYPE_SRC(5U)
```

Description

Probe Counter Type

Define XPM_PROBE_COUNTER_TYPE_VAL

Definition

```
#define XPM_PROBE_COUNTER_TYPE_VAL(6U)
```

Description

Probe Counter Type

Define XST_API_BASE_VERSION

Definition

```
#define XST_API_BASE_VERSION(1U)
```

Description

PM API versions

Define XST_API_QUERY_DATA_VERSION

Definition

```
#define XST_API_QUERY_DATA_VERSION(2U)
```

Description

PM API versions

Define XST_API_REG_NOTIFIER_VERSION

Definition

```
#define XST_API_REG_NOTIFIER_VERSION(2U)
```

Description

PM API versions

Define XST_API_PM_IOCTL_VERSION

Definition

```
#define XST_API_PM_IOCTL_VERSION(2U)
```

Description

PM API versions

Define XST_API_PM_FEATURE_CHECK_VERSION

Definition

```
#define XST_API_PM_FEATURE_CHECK_VERSION(2U)
```

Description

PM API versions

Define XST_API_SELF_SUSPEND_VERSION

Definition

```
#define XST_API_SELF_SUSPEND_VERSION(2U)
```

Description

PM API versions

Define XST_API_FORCE_POWERDOWN_VERSION

Definition

```
#define XST_API_FORCE_POWERDOWN_VERSION(2U)
```

Description

PM API versions

Define XST_API_REQUEST_NODE_VERSION

Definition

```
#define XST_API_REQUEST_NODE_VERSION(2U)
```

Description

PM API versions

Define XST_API_RELEASE_NODE_VERSION

Definition

```
#define XST_API_RELEASE_NODE_VERSION(2U)
```

Description

PM API versions

Define XST_API_GET_OP_CHAR_VERSION

Definition

```
#define XST_API_GET_OP_CHAR_VERSION(2U)
```

Description

PM API versions

Define AIE_OPS_COL_RST

Definition

```
#define AIE_OPS_COL_RSTBIT(0U)
```

Description

AIE Run time Operations

Define AIE_OPS_SHIM_RST

Definition

```
#define AIE_OPS_SHIM_RSTBIT(1U)
```

Description

AIE Run time Operations

Define AIE_OPS_ENB_COL_CLK_BUFF

Definition

```
#define AIE_OPS_ENB_COL_CLK_BUFFBIT(2U)
```

Description

AIE Run time Operations

Define AIE_OPS_ZEROIZATION

Definition

```
#define AIE_OPS_ZEROIZATIONBIT(3U)
```

Description

AIE Run time Operations

Define AIE_OPS_DIS_COL_CLK_BUFF

Definition

```
#define AIE_OPS_DIS_COL_CLK_BUFFBIT(4U)
```

Description

AIE Run time Operations

Define AIE_OPS_ENB_AXI_MM_ERR_EVENT

Definition

```
#define AIE_OPS_ENB_AXI_MM_ERR_EVENTBIT(5U)
```

Description

AIE Run time Operations

Define AIE_OPS_SET_L2_CTRL_NPI_INTR

Definition

```
#define AIE_OPS_SET_L2_CTRL_NPI_INTRBIT(6U)
```

Description

AIE Run time Operations

Define AIE_OPS_MAX

Definition

```
#define AIE_OPS_MAX                     (
              AIE_OPS_COL_RST
             |        \

              AIE_OPS_SHIM_RST
             |        \

              AIE_OPS_ENB_COL_CLK_BUFF
             |    \

              AIE_OPS_ZEROIZATION
             |        \

              AIE_OPS_DIS_COL_CLK_BUFF
             |    \
```

```
                AIE_OPS_ENB_AXI_MM_ERR_EVENT
              |     \

                AIE_OPS_SET_L2_CTRL_NPI_INTR
            )
```

Description

AIE Run time Operations

XilPM API Version Detail

XilPM EEMI API Version Detail

This section provide details of EEMI API version and the history for PM APIs of XilPM library.

NAME	ID	Platform	Version	Description
PM_GET_API_VERSION	0x1	Versal adaptive SoC, Zynq UltraScale+ MPSoC	1	This API is used to request the version number of the API.
PM_SET_CONFIGURATION	0x2	Zynq UltraScale+ MPSoC	1	This API is used to configure the power management framework.
PM_GET_NODE_STATUS	0x3	Versal adaptive SoC, Zynq UltraScale+ MPSoC	1	This API is used to obtain information about current status of a device.
PM_GET_OP_CHARACTERISTIC	0x4	Versal adaptive SoC, Zynq UltraScale+ MPSoC	2	V1 -This API is used to gets operating characteristics of a device. V2 - Added support of bitmask functionality, user can check the supported "type" first before performing the actual functionality.
PM_REGISTER_NOTIFIER	0x5	Versal adaptive SoC, Zynq UltraScale+ MPSoC	2	V1 -This API is used to register a subsystem to be notified about the device event . V2 - Added support of event management functionality . ***Note*:** V2 API is only supported in Versal and V1 API is only supported in Zynq UltraScale+ MPSoC.
PM_REQUEST_SUSPEND	0x6	Versal adaptive SoC, Zynq UltraScale+ MPSoC	1	This API is used to send suspend request to another subsystem.
PM_SELF_SUSPEND	0x7	Versal adaptive SoC, Zynq UltraScale+ MPSoC	2	V1 - This API is used to suspend a child subsystem. V2 - Added support of cpu idle functionality during force powerdown . ***Note*:** V2 API is only supported in Versal and V1 API is only supported in Zynq UltraScale+ MPSoC.

NAME	ID	Platform	Version	Description
PM_FORCE_POWERDOWN	0x8	Versal adaptive SoC and Zynq UltraScale+ MPSoC	2	V1 -This API is used to Powerdown other processor or node. V2 - Added support of cpu idle functionality during force powerdown. ***Note*:** V2 API is only supported in Versal and V1 API is only supported in Zynq UltraScale+ MPSoC.
PM_ABORT_SUSPEND	0x9	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to aborting suspend of a child subsystem.
PM_REQUEST_WAKEUP	0xA	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to start-up and wake-up a child subsystem.
PM_SET_WAKEUP_SOURCE	0xB	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to set wakeup source.
PM_SYSTEM_SHUTDOWN	0xC	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to shutdown or restart the system .
PM_REQUEST_NODE	0xD	Versal adaptive SoC and Zynq UltraScale+ MPSoC	2	V1 -This API is used to request the usage of a device. V2 - Added support of security policy handling during request device. ***Note*:** V2 API is only supported in Versal and V1 API is only supported in Zynq UltraScale+ MPSoC.
PM_RELEASE_NODE	0xE	Versal adaptive SoC and Zynq UltraScale+ MPSoC	2	V1 -This API is used to release the usage of a device. V2 - Added support of security policy handling during request device. ***Note*:** V2 API is only supported in Versal and V1 API is only supported in Zynq UltraScale+ MPSoC.
PM_SET_REQUIREMENT	0xF	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to announce a change in requirement for a specific slave node which is currently in use.
PM_SET_MAX_LATENCY	0x10	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to set maximum allowed latency for the device.
PM_RESET_ASSERT	0x11	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to reset or de-reset a device.
PM_RESET_GET_STATUS	0x12	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to reads the device reset state.
PM_MMIO_WRITE	0x13	Zynq UltraScale+ MPSoC	1	This API is used to writes a value into a register. ***Note*:** This API is deprecated for Versal adaptive SoC.

NAME	ID	Platform	Version	Description
PM_MMIO_READ	0x14	Zynq UltraScale+ MPSoC	1	This API is used to reads a value from a register. ***Note*:** This API is deprecated for Versal adaptive SoC.
PM_INIT_FINALIZE	0x15	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to initializes subsystem and releases unused devices.
PM_GET_CHIPID	0x18	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to request the version and ID code of a chip.
PM_PINCTRL_REQUEST	0x1C	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to requests the pin.
PM_PINCTRL_RELEASE	0x1D	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to releases the pin
PM_PINCTRL_GET_FUNCTION	0x1E	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to reads the pin function.
PM_PINCTRL_SET_FUNCTION	0x1F	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to sets the pin function.
PM_PINCTRL_CONFIG_PARAM_GET	0x20	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to reads the pin parameter value.
PM_PINCTRL_CONFIG_PARAM_SET	0x21	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to sets the pin parameter value.
PM_IOCTL	0x22	Versal adaptive SoC and Zynq UltraScale+ MPSoC	2	V1 -This API is used to performs driver-like IOCTL functions on shared system devices. V2 - Added support of bitmask functionality, user can check the supported ID first before performing the actual functionality .
PM_QUERY_DATA	0x23	Versal adaptive SoC and Zynq UltraScale+ MPSoC	2	V1 -This API is used to queries information about the platform resources. V2 - Added support of bitmask functionality, user can check the supported ID first before performing the actual functionality.
PM_CLOCK_ENABLE	0x24	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to enables the clock.
PM_CLOCK_DISABLE	0x25	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to disables the clock.
PM_CLOCK_GETSTATE	0x26	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to reads the clock state.
PM_CLOCK_SETDIVIDER	0x27	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to sets the divider value of the clock.
PM_CLOCK_GETDIVIDER	0x28	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to reads the clock divider.

NAME	ID	Platform	Version	Description
PM_CLOCK_SETRATE	0x29	Versal adaptive SoC	1	This API is used to sets the rate of the clock.
PM_CLOCK_GETRATE	0x2A	Versal adaptive SoC	1	This API is used to gets the rate of the clock.
PM_CLOCK_SETPARENT	0x2B	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to sets the parent of the clock.
PM_CLOCK_GETPARENT	0x2C	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to reads the clock parent.
PM_PLL_SET_PARAM	0x30	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to sets the parameter of PLL clock.
PM_PLL_GET_PARAM	0x31	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to reads the parameter of PLL clock.
PM_PLL_SET_MODE	0x32	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to sets the mode of PLL clock.
PM_PLL_GET_MODE	0x33	Versal adaptive SoC and Zynq UltraScale+ MPSoC	1	This API is used to reads the mode of PLL clock.
PM_REGISTER_ACCESS	0x34	Zynq UltraScale+ MPSoC	1	This API is used for register read/write access data . ***Note*:** This API is deprecated for Versal adaptive SoC.
PM_EFUSE_ACCESS	0x35	Zynq UltraScale+ MPSoC	1	This API is used to provides access to efuse memory.
PM_FEATURE_CHECK	0x3F	Versal adaptive SoC and Zynq UltraScale+ MPSoC	2	V1 -This API is used to returns supported version of the given API. V2 - Added support of bitmask payload functionality.

XilPM IOCTL IDs Detail

This section provides the details of the IOCTL IDs which are supported across the different platforms and their brief descriptions.

Name	ID	Platform	Description
IOCTL_GET_RPU_OPER_MODE	0	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get RPU mode
IOCTL_SET_RPU_OPER_MODE	1	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Set RPU mode
IOCTL_RPU_BOOT_ADDR_CONFIG	2	Versal adaptive SoC and Zynq UltraScale+ MPSoC	RPU boot address config
IOCTL_TCM_COMB_CONFIG	3	Versal adaptive SoC and Zynq UltraScale+ MPSoC	TCM config
IOCTL_SET_TAPDELAY_BYPASS	4	Versal adaptive SoC and Zynq UltraScale+ MPSoC	TAP delay bypass
IOCTL_SET_SGMII_MODE	5	Zynq UltraScale+ MPSoC	SGMII mode

Name	ID	Platform	Description
IOCTL_SD_DLL_RESET	6	Versal adaptive SoC and Zynq UltraScale+ MPSoC	SD DLL reset
IOCTL_SET_SD_TAPDELAY	7	Versal adaptive SoC and Zynq UltraScale+ MPSoC	SD TAP delay
IOCTL_SET_PLL_FRAC_MODE	8	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Set PLL frac mode
IOCTL_GET_PLL_FRAC_MODE	9	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get PLL frac mode
IOCTL_SET_PLL_FRAC_DATA	10	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Set PLL frac data
IOCTL_GET_PLL_FRAC_DATA	11	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get PLL frac data
IOCTL_WRITE_GGS	12	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Write GGS
IOCTL_READ_GGS	13	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Read GGS
IOCTL_WRITE_PGGS	14	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Write PGGS
IOCTL_READ_PGGS	15	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Read PGGS
IOCTL_ULPI_RESET	16	Zynq UltraScale+ MPSoC	ULPI reset
IOCTL_SET_BOOT_HEALTH_STATUS	17	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Set boot status
IOCTL_AFI	18	Zynq UltraScale+ MPSoC	AFI
IOCTL_PROBE_COUNTER_READ	19	Versal adaptive SoC	Probe counter read
IOCTL_PROBE_COUNTER_WRITE	20	Versal adaptive SoC	Probe counter write
IOCTL_OSPI_MUX_SELECT	21	Versal adaptive SoC	OSPI mux select
IOCTL_USB_SET_STATE	22	Versal adaptive SoC	USB set state
IOCTL_GET_LAST_RESET_REASON	23	Versal adaptive SoC	Get last reset reason
IOCTL_AIE_ISR_CLEAR	24	Versal adaptive SoC	AIE ISR clear
IOCTL_REGISTER_SGI	25	None	Register SGI to ATF
IOCTL_SET_FEATURE_CONFIG	26	Zynq UltraScale+ MPSoC	Set runtime feature config
IOCTL_GET_FEATURE_CONFIG	27	Zynq UltraScale+ MPSoC	Get runtime feature config
IOCTL_READ_REG	28	Versal adaptive SoC	Read a 32-bit register
IOCTL_MASK_WRITE_REG	29	Versal adaptive SoC	RMW a 32-bit register
IOCTL_SET_SD_CONFIG	30	Zynq UltraScale+ MPSoC	Set SD config register value
IOCTL_SET_GEM_CONFIG	31	Zynq UltraScale+ MPSoC	Set GEM config register value
IOCTL_SET_USB_CONFIG	32	Zynq UltraScale+ MPSoC	Set USB config register value
IOCTL_AIE_OPS	33	Versal adaptive SoC	AIE1/AIEML Run Time Operations
IOCTL_GET_QOS	34	Versal adaptive SoC	Get Device QoS value

Name	ID	Platform	Description
IOCTL_GET_APU_OPER_MODE	35	Versal adaptive SoC	Get APU operation mode
IOCTL_SET_APU_OPER_MODE	36	Versal adaptive SoC	Set APU operation mode

XilPM QUERY IDs Detail

This section provides the details of the QUERY IDs which are supported across the different platforms and their brief descriptions.

Name	ID	Platform	Description
XPM_QID_INVALID	0	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Invalid Query ID
XPM_QID_CLOCK_GET_NAME	1	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get clock name
XPM_QID_CLOCK_GET_TOPOLOGY	2	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get clock topology
XPM_QID_CLOCK_GET_FIXEDFACTOR_PARAMS	3	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get clock fixedfactor parameter
XPM_QID_CLOCK_GET_MUXSOURCES	4	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get clock mux sources
XPM_QID_CLOCK_GET_ATTRIBUTES	5	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get clock attributes
XPM_QID_PINCTRL_GET_NUM_PINS	6	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get total pins
XPM_QID_PINCTRL_GET_NUM_FUNCTIONS	7	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get total pin functions
XPM_QID_PINCTRL_GET_NUM_FUNCTION_GROUPS	8	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get total pin function groups
XPM_QID_PINCTRL_GET_FUNCTION_NAME	9	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get pin function name
XPM_QID_PINCTRL_GET_FUNCTION_GROUPS	10	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get pin function groups
XPM_QID_PINCTRL_GET_PIN_GROUPS	11	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get pin groups
XPM_QID_CLOCK_GET_NUM_CLOCKS	12	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get number of clocks
XPM_QID_CLOCK_GET_MAX_DIVISOR	13	Versal adaptive SoC and Zynq UltraScale+ MPSoC	Get max clock divisor
XPM_QID_PLD_GET_PARENT	14	Versal adaptive SoC	Get PLD parent

XilPM GET_OP_CHAR IDs Detail

This section provides the details of the GET_OP_CHAR IDs which are supported across the different platforms and their brief descriptions.

Name	ID	Platform	Description
PM_OPCHAR_TYPE_POWER	1	Zynq UltraScale+ MPSoC	Operating characteristic ID power
PM_OPCHAR_TYPE_TEMP	2	Versal adaptive SoC	Operating characteristic ID temperature
PM_OPCHAR_TYPE_LATENCY	3	Versal adaptive SoC, Zynq UltraScale+ MPSoC	Operating characteristic ID latency

Error Status

This section lists the Power management specific return error statuses.

Enumerations

Enumeration XPmBootStatus

Boot Status

Table 334: **Enumeration XPmBootStatus Values**

Value	Description
PM_INITIAL_BOOT	boot is a fresh system startup
PM_RESUME	boot is a resume
PM_BOOT_ERROR	error, boot cause cannot be identified

Enumeration XPmResetAction

PM Reset Action types

Table 335: **Enumeration XPmResetAction Values**

Value	Description
XILPM_RESET_ACTION_RELEASE	Reset action release
XILPM_RESET_ACTION_ASSERT	Reset action assert
XILPM_RESET_ACTION_PULSE	Reset action pulse

Enumeration XPmReset

PM Reset Line IDs

Table 336: **Enumeration XPmReset Values**

Value	Description
XILPM_RESET_PCIE_CFG	Reset ID PCIE_CFG
XILPM_RESET_PCIE_BRIDGE	Reset ID PCIE_BRIDGE
XILPM_RESET_PCIE_CTRL	Reset ID PCIE_CTRL
XILPM_RESET_DP	Reset ID DP
XILPM_RESET_SWDT_CRF	Reset ID SWDT_CRF
XILPM_RESET_AFI_FM5	Reset ID AFI_FM5
XILPM_RESET_AFI_FM4	Reset ID AFI_FM4
XILPM_RESET_AFI_FM3	Reset ID AFI_FM3
XILPM_RESET_AFI_FM2	Reset ID AFI_FM2
XILPM_RESET_AFI_FM1	Reset ID AFI_FM1
XILPM_RESET_AFI_FM0	Reset ID AFI_FM0
XILPM_RESET_GDMA	Reset ID GDMA
XILPM_RESET_GPU_PP1	Reset ID GPU_PP1
XILPM_RESET_GPU_PP0	Reset ID GPU_PP0
XILPM_RESET_GPU	Reset ID GPU
XILPM_RESET_GT	Reset ID GT
XILPM_RESET_SATA	Reset ID SATA
XILPM_RESET_ACPU3_PWRON	Reset ID ACPU3_PWRON
XILPM_RESET_ACPU2_PWRON	Reset ID ACPU2_PWRON
XILPM_RESET_ACPU1_PWRON	Reset ID ACPU1_PWRON
XILPM_RESET_ACPU0_PWRON	Reset ID ACPU0_PWRON
XILPM_RESET_APU_L2	Reset ID APU_L2
XILPM_RESET_ACPU3	Reset ID ACPU3
XILPM_RESET_ACPU2	Reset ID ACPU2
XILPM_RESET_ACPU1	Reset ID ACPU1
XILPM_RESET_ACPU0	Reset ID ACPU0
XILPM_RESET_DDR	Reset ID DDR
XILPM_RESET_APM_FPD	Reset ID APM_FPD
XILPM_RESET_SOFT	Reset ID SOFT
XILPM_RESET_GEM0	Reset ID GEM0
XILPM_RESET_GEM1	Reset ID GEM1
XILPM_RESET_GEM2	Reset ID GEM2
XILPM_RESET_GEM3	Reset ID GEM3
XILPM_RESET_QSPI	Reset ID QSPI
XILPM_RESET_UART0	Reset ID UART0
XILPM_RESET_UART1	Reset ID UART1
XILPM_RESET_SPI0	Reset ID SPI0
XILPM_RESET_SPI1	Reset ID SPI1
XILPM_RESET_SDIO0	Reset ID SDIO0

Table 336: **Enumeration XPmReset Values** *(cont'd)*

Value	Description
XILPM_RESET_SDIO1	Reset ID SDIO1
XILPM_RESET_CAN0	Reset ID CAN0
XILPM_RESET_CAN1	Reset ID CAN1
XILPM_RESET_I2C0	Reset ID I2C0
XILPM_RESET_I2C1	Reset ID I2C1
XILPM_RESET_TTC0	Reset ID TTC0
XILPM_RESET_TTC1	Reset ID TTC1
XILPM_RESET_TTC2	Reset ID TTC2
XILPM_RESET_TTC3	Reset ID TTC3
XILPM_RESET_SWDT_CRL	Reset ID SWDT_CRL
XILPM_RESET_NAND	Reset ID NAND
XILPM_RESET_ADMA	Reset ID ADMA
XILPM_RESET_GPIO	Reset ID GPIO
XILPM_RESET_IOU_CC	Reset ID IOU_CC
XILPM_RESET_TIMESTAMP	Reset ID TIMESTAMP
XILPM_RESET_RPU_R50	Reset ID RPU_R50
XILPM_RESET_RPU_R51	Reset ID RPU_R51
XILPM_RESET_RPU_AMBA	Reset ID RPU_AMBA
XILPM_RESET_OCM	Reset ID OCM
XILPM_RESET_RPU_PGE	Reset ID RPU_PGE
XILPM_RESET_USB0_CORERESET	Reset ID USB0_CORERESE
XILPM_RESET_USB1_CORERESET	Reset ID USB1_CORERESE
XILPM_RESET_USB0_HIBERRESET	Reset ID USB0_HIBERRES
XILPM_RESET_USB1_HIBERRESET	Reset ID USB1_HIBERRES
XILPM_RESET_USB0_APB	Reset ID USB0_APB
XILPM_RESET_USB1_APB	Reset ID USB1_APB
XILPM_RESET_IPI	Reset ID IPI
XILPM_RESET_APM_LPD	Reset ID APM_LPD
XILPM_RESET_RTC	Reset ID RTC
XILPM_RESET_SYSMON	Reset ID SYSMON
XILPM_RESET_AFI_FM6	Reset ID AFI_FM6
XILPM_RESET_LPD_SWDT	Reset ID LPD_SWDT
XILPM_RESET_FPD	Reset ID FPD
XILPM_RESET_RPU_DBG1	Reset ID RPU_DBG1
XILPM_RESET_RPU_DBG0	Reset ID RPU_DBG0
XILPM_RESET_DBG_LPD	Reset ID DBG_LPD
XILPM_RESET_DBG_FPD	Reset ID DBG_FPD
XILPM_RESET_APLL	Reset ID APLL
XILPM_RESET_DPLL	Reset ID DPLL

Table 336: **Enumeration XPmReset Values** *(cont'd)*

Value	Description
XILPM_RESET_VPLL	Reset ID VPLL
XILPM_RESET_IOPLL	Reset ID IOPLL
XILPM_RESET_RPLL	Reset ID RPLL
XILPM_RESET_GPO3_PL_0	Reset ID GPO3_PL_0
XILPM_RESET_GPO3_PL_1	Reset ID GPO3_PL_1
XILPM_RESET_GPO3_PL_2	Reset ID GPO3_PL_2
XILPM_RESET_GPO3_PL_3	Reset ID GPO3_PL_3
XILPM_RESET_GPO3_PL_4	Reset ID GPO3_PL_4
XILPM_RESET_GPO3_PL_5	Reset ID GPO3_PL_5
XILPM_RESET_GPO3_PL_6	Reset ID GPO3_PL_6
XILPM_RESET_GPO3_PL_7	Reset ID GPO3_PL_7
XILPM_RESET_GPO3_PL_8	Reset ID GPO3_PL_8
XILPM_RESET_GPO3_PL_9	Reset ID GPO3_PL_9
XILPM_RESET_GPO3_PL_10	Reset ID GPO3_PL_10
XILPM_RESET_GPO3_PL_11	Reset ID GPO3_PL_11
XILPM_RESET_GPO3_PL_12	Reset ID GPO3_PL_12
XILPM_RESET_GPO3_PL_13	Reset ID GPO3_PL_13
XILPM_RESET_GPO3_PL_14	Reset ID GPO3_PL_14
XILPM_RESET_GPO3_PL_15	Reset ID GPO3_PL_15
XILPM_RESET_GPO3_PL_16	Reset ID GPO3_PL_16
XILPM_RESET_GPO3_PL_17	Reset ID GPO3_PL_17
XILPM_RESET_GPO3_PL_18	Reset ID GPO3_PL_18
XILPM_RESET_GPO3_PL_19	Reset ID GPO3_PL_19
XILPM_RESET_GPO3_PL_20	Reset ID GPO3_PL_20
XILPM_RESET_GPO3_PL_21	Reset ID GPO3_PL_21
XILPM_RESET_GPO3_PL_22	Reset ID GPO3_PL_22
XILPM_RESET_GPO3_PL_23	Reset ID GPO3_PL_23
XILPM_RESET_GPO3_PL_24	Reset ID GPO3_PL_24
XILPM_RESET_GPO3_PL_25	Reset ID GPO3_PL_25
XILPM_RESET_GPO3_PL_26	Reset ID GPO3_PL_26
XILPM_RESET_GPO3_PL_27	Reset ID GPO3_PL_27
XILPM_RESET_GPO3_PL_28	Reset ID GPO3_PL_28
XILPM_RESET_GPO3_PL_29	Reset ID GPO3_PL_29
XILPM_RESET_GPO3_PL_30	Reset ID GPO3_PL_30
XILPM_RESET_GPO3_PL_31	Reset ID GPO3_PL_31
XILPM_RESET_RPU_LS	Reset ID RPU_LS
XILPM_RESET_PS_ONLY	Reset ID PS_ONLY
XILPM_RESET_PL	Reset ID PL
XILPM_RESET_GPIO5_EMIO_92	Reset ID GPIO5_EMIO_92

Table 336: **Enumeration XPmReset Values** *(cont'd)*

Value	Description
XILPM_RESET_GPIO5_EMIO_93	Reset ID GPIO5_EMIO_93
XILPM_RESET_GPIO5_EMIO_94	Reset ID GPIO5_EMIO_94
XILPM_RESET_GPIO5_EMIO_95	Reset ID GPIO5_EMIO_95

Enumeration XPmNotifyEvent

PM Notify Events Enum

Table 337: **Enumeration XPmNotifyEvent Values**

Value	Description
EVENT_STATE_CHANGE	State change event
EVENT_ZERO_USERS	Zero user event
EVENT_CPU_IDLE_FORCE_PWRDWN	CPU idle event during force power down

Enumeration XPmClock

PM Clock IDs

Table 338: **Enumeration XPmClock Values**

Value	Description
PM_CLOCK_IOPLL	Clock ID IOPLL
PM_CLOCK_RPLL	Clock ID RPLL
PM_CLOCK_APLL	Clock ID APLL
PM_CLOCK_DPLL	Clock ID DPLL
PM_CLOCK_VPLL	Clock ID VPLL
PM_CLOCK_IOPLL_TO_FPD	Clock ID IOPLL_TO_FPD
PM_CLOCK_RPLL_TO_FPD	Clock ID RPLL_TO_FPD
PM_CLOCK_APLL_TO_LPD	Clock ID APLL_TO_LPD
PM_CLOCK_DPLL_TO_LPD	Clock ID DPLL_TO_LPD
PM_CLOCK_VPLL_TO_LPD	Clock ID VPLL_TO_LPD
PM_CLOCK_ACPU	Clock ID ACPU
PM_CLOCK_ACPU_HALF	Clock ID ACPU_HALF
PM_CLOCK_DBG_FPD	Clock ID DBG_FPD
PM_CLOCK_DBG_LPD	Clock ID DBG_LPD
PM_CLOCK_DBG_TRACE	Clock ID DBG_TRACE
PM_CLOCK_DBG_TSTMP	Clock ID DBG_TSTMP
PM_CLOCK_DP_VIDEO_REF	Clock ID DP_VIDEO_REF
PM_CLOCK_DP_AUDIO_REF	Clock ID DP_AUDIO_REF

Table 338: **Enumeration XPmClock Values** *(cont'd)*

Value	Description
PM_CLOCK_DP_STC_REF	Clock ID DP_STC_REF
PM_CLOCK_GDMA_REF	Clock ID GDMA_REF
PM_CLOCK_DPDMA_REF	Clock ID DPDMA_REF
PM_CLOCK_DDR_REF	Clock ID DDR_REF
PM_CLOCK_SATA_REF	Clock ID SATA_REF
PM_CLOCK_PCIE_REF	Clock ID PCIE_REF
PM_CLOCK_GPU_REF	Clock ID GPU_REF
PM_CLOCK_GPU_PP0_REF	Clock ID GPU_PP0_REF
PM_CLOCK_GPU_PP1_REF	Clock ID GPU_PP1_REF
PM_CLOCK_TOPSW_MAIN	Clock ID TOPSW_MAIN
PM_CLOCK_TOPSW_LSBUS	Clock ID TOPSW_LSBUS
PM_CLOCK_GTGREF0_REF	Clock ID GTGREF0_REF
PM_CLOCK_LPD_SWITCH	Clock ID LPD_SWITCH
PM_CLOCK_LPD_LSBUS	Clock ID LPD_LSBUS
PM_CLOCK_USB0_BUS_REF	Clock ID USB0_BUS_REF
PM_CLOCK_USB1_BUS_REF	Clock ID USB1_BUS_REF
PM_CLOCK_USB3_DUAL_REF	Clock ID USB3_DUAL_REF
PM_CLOCK_USB0	Clock ID USB0
PM_CLOCK_USB1	Clock ID USB1
PM_CLOCK_CPU_R5	Clock ID CPU_R5
PM_CLOCK_CPU_R5_CORE	Clock ID CPU_R5_CORE
PM_CLOCK_CSU_SPB	Clock ID CSU_SPB
PM_CLOCK_CSU_PLL	Clock ID CSU_PLL
PM_CLOCK_PCAP	Clock ID PCAP
PM_CLOCK_IOU_SWITCH	Clock ID IOU_SWITCH
PM_CLOCK_GEM_TSU_REF	Clock ID GEM_TSU_REF
PM_CLOCK_GEM_TSU	Clock ID GEM_TSU
PM_CLOCK_GEM0_TX	Clock ID GEM0_TX
PM_CLOCK_GEM1_TX	Clock ID GEM1_TX
PM_CLOCK_GEM2_TX	Clock ID GEM2_TX
PM_CLOCK_GEM3_TX	Clock ID GEM3_TX
PM_CLOCK_GEM0_RX	Clock ID GEM0_RX
PM_CLOCK_GEM1_RX	Clock ID GEM1_RX
PM_CLOCK_GEM2_RX	Clock ID GEM2_RX
PM_CLOCK_GEM3_RX	Clock ID GEM3_RX
PM_CLOCK_QSPI_REF	Clock ID QSPI_REF
PM_CLOCK_SDIO0_REF	Clock ID SDIO0_REF
PM_CLOCK_SDIO1_REF	Clock ID SDIO1_REF
PM_CLOCK_UART0_REF	Clock ID UART0_REF

Table 338: **Enumeration XPmClock Values** *(cont'd)*

Value	Description
PM_CLOCK_UART1_REF	Clock ID UART1_REF
PM_CLOCK_SPI0_REF	Clock ID SPI0_REF
PM_CLOCK_SPI1_REF	Clock ID SPI1_REF
PM_CLOCK_NAND_REF	Clock ID NAND_REF
PM_CLOCK_I2C0_REF	Clock ID I2C0_REF
PM_CLOCK_I2C1_REF	Clock ID I2C1_REF
PM_CLOCK_CAN0_REF	Clock ID CAN0_REF
PM_CLOCK_CAN1_REF	Clock ID CAN1_REF
PM_CLOCK_CAN0	Clock ID CAN0
PM_CLOCK_CAN1	Clock ID CAN1
PM_CLOCK_DLL_REF	Clock ID DLL_REF
PM_CLOCK_ADMA_REF	Clock ID ADMA_REF
PM_CLOCK_TIMESTAMP_REF	Clock ID TIMESTAMP_REF
PM_CLOCK_AMS_REF	Clock ID AMS_REF
PM_CLOCK_PL0_REF	Clock ID PL0_REF
PM_CLOCK_PL1_REF	Clock ID PL1_REF
PM_CLOCK_PL2_REF	Clock ID PL2_REF
PM_CLOCK_PL3_REF	Clock ID PL3_REF
PM_CLOCK_WDT	Clock ID WDT
PM_CLOCK_IOPLL_INT	Clock ID IOPLL_INT
PM_CLOCK_IOPLL_PRE_SRC	Clock ID IOPLL_PRE_SRC
PM_CLOCK_IOPLL_HALF	Clock ID IOPLL_HALF
PM_CLOCK_IOPLL_INT_MUX	Clock ID IOPLL_INT_MUX
PM_CLOCK_IOPLL_POST_SRC	Clock ID IOPLL_POST_SRC
PM_CLOCK_RPLL_INT	Clock ID RPLL_INT
PM_CLOCK_RPLL_PRE_SRC	Clock ID RPLL_PRE_SRC
PM_CLOCK_RPLL_HALF	Clock ID RPLL_HALF
PM_CLOCK_RPLL_INT_MUX	Clock ID RPLL_INT_MUX
PM_CLOCK_RPLL_POST_SRC	Clock ID RPLL_POST_SRC
PM_CLOCK_APLL_INT	Clock ID APLL_INT
PM_CLOCK_APLL_PRE_SRC	Clock ID APLL_PRE_SRC
PM_CLOCK_APLL_HALF	Clock ID APLL_HALF
PM_CLOCK_APLL_INT_MUX	Clock ID APLL_INT_MUX
PM_CLOCK_APLL_POST_SRC	Clock ID APLL_POST_SRC
PM_CLOCK_DPLL_INT	Clock ID DPLL_INT
PM_CLOCK_DPLL_PRE_SRC	Clock ID DPLL_PRE_SRC
PM_CLOCK_DPLL_HALF	Clock ID DPLL_HALF
PM_CLOCK_DPLL_INT_MUX	Clock ID DPLL_INT_MUX
PM_CLOCK_DPLL_POST_SRC	Clock ID DPLL_POST_SRC

Table 338: **Enumeration XPmClock Values** *(cont'd)*

Value	Description
PM_CLOCK_VPLL_INT	Clock ID VPLL_INT
PM_CLOCK_VPLL_PRE_SRC	Clock ID VPLL_PRE_SRC
PM_CLOCK_VPLL_HALF	Clock ID VPLL_HALF
PM_CLOCK_VPLL_INT_MUX	Clock ID VPLL_INT_MUX
PM_CLOCK_VPLL_POST_SRC	Clock ID VPLL_POST_SRC
PM_CLOCK_CAN0_MIO	Clock ID CAN0_MIO
PM_CLOCK_CAN1_MIO	Clock ID CAN1_MIO
PM_CLOCK_ACPU_FULL	Clock ID ACPU_FULL
PM_CLOCK_GEM0_REF	Clock ID GEM0_REF
PM_CLOCK_GEM1_REF	Clock ID GEM1_REF
PM_CLOCK_GEM2_REF	Clock ID GEM2_REF
PM_CLOCK_GEM3_REF	Clock ID GEM3_REF
PM_CLOCK_GEM0_REF_UNGATED	Clock ID GEM0_REF_UNGATED
PM_CLOCK_GEM1_REF_UNGATED	Clock ID GEM1_REF_UNGATED
PM_CLOCK_GEM2_REF_UNGATED	Clock ID GEM2_REF_UNGATED
PM_CLOCK_GEM3_REF_UNGATED	Clock ID GEM3_REF_UNGATED
PM_CLOCK_EXT_PSS_REF	Clock ID EXT_PSS_REF
PM_CLOCK_EXT_VIDEO	Clock ID EXT_VIDEO
PM_CLOCK_EXT_PSS_ALT_REF	Clock ID EXT_PSS_ALT_REF
PM_CLOCK_EXT_AUX_REF	Clock ID EXT_AUX_REF
PM_CLOCK_EXT_GT_CRX_REF	Clock ID EXT_GT_CRX_REF
PM_CLOCK_EXT_SWDT0	Clock ID EXT_SWDT0
PM_CLOCK_EXT_SWDT1	Clock ID EXT_SWDT1
PM_CLOCK_EXT_GEM0_TX_EMIO	Clock ID EXT_GEM0_TX_EMIO
PM_CLOCK_EXT_GEM1_TX_EMIO	Clock ID EXT_GEM1_TX_EMIO
PM_CLOCK_EXT_GEM2_TX_EMIO	Clock ID EXT_GEM2_TX_EMIO
PM_CLOCK_EXT_GEM3_TX_EMIO	Clock ID EXT_GEM3_TX_EMIO
PM_CLOCK_EXT_GEM0_RX_EMIO	Clock ID EXT_GEM0_RX_EMIO
PM_CLOCK_EXT_GEM1_RX_EMIO	Clock ID EXT_GEM1_RX_EMIO
PM_CLOCK_EXT_GEM2_RX_EMIO	Clock ID EXT_GEM2_RX_EMIO
PM_CLOCK_EXT_GEM3_RX_EMIO	Clock ID EXT_GEM3_RX_EMIO
PM_CLOCK_EXT_MIO50_OR_MIO51	Clock ID EXT_MIO50_OR_MIO51
PM_CLOCK_EXT_MIO0	Clock ID EXT_MIO0
PM_CLOCK_EXT_MIO1	Clock ID EXT_MIO1
PM_CLOCK_EXT_MIO2	Clock ID EXT_MIO2
PM_CLOCK_EXT_MIO3	Clock ID EXT_MIO3
PM_CLOCK_EXT_MIO4	Clock ID EXT_MIO4
PM_CLOCK_EXT_MIO5	Clock ID EXT_MIO5
PM_CLOCK_EXT_MIO6	Clock ID EXT_MIO6

Table 338: **Enumeration XPmClock Values** *(cont'd)*

Value	Description
PM_CLOCK_EXT_MIO7	Clock ID EXT_MIO7
PM_CLOCK_EXT_MIO8	Clock ID EXT_MIO8
PM_CLOCK_EXT_MIO9	Clock ID EXT_MIO9
PM_CLOCK_EXT_MIO10	Clock ID EXT_MIO10
PM_CLOCK_EXT_MIO11	Clock ID EXT_MIO11
PM_CLOCK_EXT_MIO12	Clock ID EXT_MIO12
PM_CLOCK_EXT_MIO13	Clock ID EXT_MIO13
PM_CLOCK_EXT_MIO14	Clock ID EXT_MIO14
PM_CLOCK_EXT_MIO15	Clock ID EXT_MIO15
PM_CLOCK_EXT_MIO16	Clock ID EXT_MIO16
PM_CLOCK_EXT_MIO17	Clock ID EXT_MIO17
PM_CLOCK_EXT_MIO18	Clock ID EXT_MIO18
PM_CLOCK_EXT_MIO19	Clock ID EXT_MIO19
PM_CLOCK_EXT_MIO20	Clock ID EXT_MIO20
PM_CLOCK_EXT_MIO21	Clock ID EXT_MIO21
PM_CLOCK_EXT_MIO22	Clock ID EXT_MIO22
PM_CLOCK_EXT_MIO23	Clock ID EXT_MIO23
PM_CLOCK_EXT_MIO24	Clock ID EXT_MIO24
PM_CLOCK_EXT_MIO25	Clock ID EXT_MIO25
PM_CLOCK_EXT_MIO26	Clock ID EXT_MIO26
PM_CLOCK_EXT_MIO27	Clock ID EXT_MIO27
PM_CLOCK_EXT_MIO28	Clock ID EXT_MIO28
PM_CLOCK_EXT_MIO29	Clock ID EXT_MIO29
PM_CLOCK_EXT_MIO30	Clock ID EXT_MIO30
PM_CLOCK_EXT_MIO31	Clock ID EXT_MIO31
PM_CLOCK_EXT_MIO32	Clock ID EXT_MIO32
PM_CLOCK_EXT_MIO33	Clock ID EXT_MIO33
PM_CLOCK_EXT_MIO34	Clock ID EXT_MIO34
PM_CLOCK_EXT_MIO35	Clock ID EXT_MIO35
PM_CLOCK_EXT_MIO36	Clock ID EXT_MIO36
PM_CLOCK_EXT_MIO37	Clock ID EXT_MIO37
PM_CLOCK_EXT_MIO38	Clock ID EXT_MIO38
PM_CLOCK_EXT_MIO39	Clock ID EXT_MIO39
PM_CLOCK_EXT_MIO40	Clock ID EXT_MIO40
PM_CLOCK_EXT_MIO41	Clock ID EXT_MIO41
PM_CLOCK_EXT_MIO42	Clock ID EXT_MIO42
PM_CLOCK_EXT_MIO43	Clock ID EXT_MIO43
PM_CLOCK_EXT_MIO44	Clock ID EXT_MIO44
PM_CLOCK_EXT_MIO45	Clock ID EXT_MIO45

Table 338: **Enumeration XPmClock Values** *(cont'd)*

Value	Description
PM_CLOCK_EXT_MIO46	Clock ID EXT_MIO46
PM_CLOCK_EXT_MIO47	Clock ID EXT_MIO47
PM_CLOCK_EXT_MIO48	Clock ID EXT_MIO48
PM_CLOCK_EXT_MIO49	Clock ID EXT_MIO49
PM_CLOCK_EXT_MIO50	Clock ID EXT_MIO50
PM_CLOCK_EXT_MIO51	Clock ID EXT_MIO51
PM_CLOCK_EXT_MIO52	Clock ID EXT_MIO52
PM_CLOCK_EXT_MIO53	Clock ID EXT_MIO53
PM_CLOCK_EXT_MIO54	Clock ID EXT_MIO54
PM_CLOCK_EXT_MIO55	Clock ID EXT_MIO55
PM_CLOCK_EXT_MIO56	Clock ID EXT_MIO56
PM_CLOCK_EXT_MIO57	Clock ID EXT_MIO57
PM_CLOCK_EXT_MIO58	Clock ID EXT_MIO58
PM_CLOCK_EXT_MIO59	Clock ID EXT_MIO59
PM_CLOCK_EXT_MIO60	Clock ID EXT_MIO60
PM_CLOCK_EXT_MIO61	Clock ID EXT_MIO61
PM_CLOCK_EXT_MIO62	Clock ID EXT_MIO62
PM_CLOCK_EXT_MIO63	Clock ID EXT_MIO63
PM_CLOCK_EXT_MIO64	Clock ID EXT_MIO64
PM_CLOCK_EXT_MIO65	Clock ID EXT_MIO65
PM_CLOCK_EXT_MIO66	Clock ID EXT_MIO66
PM_CLOCK_EXT_MIO67	Clock ID EXT_MIO67
PM_CLOCK_EXT_MIO68	Clock ID EXT_MIO68
PM_CLOCK_EXT_MIO69	Clock ID EXT_MIO69
PM_CLOCK_EXT_MIO70	Clock ID EXT_MIO70
PM_CLOCK_EXT_MIO71	Clock ID EXT_MIO71
PM_CLOCK_EXT_MIO72	Clock ID EXT_MIO72
PM_CLOCK_EXT_MIO73	Clock ID EXT_MIO73
PM_CLOCK_EXT_MIO74	Clock ID EXT_MIO74
PM_CLOCK_EXT_MIO75	Clock ID EXT_MIO75
PM_CLOCK_EXT_MIO76	Clock ID EXT_MIO76
PM_CLOCK_EXT_MIO77	Clock ID EXT_MIO77

Enumeration XPmPllParam

PLL parameters

Table 339: **Enumeration XPmPllParam Values**

Value	Description
PM_PLL_PARAM_ID_DIV2	PLL param ID DIV2
PM_PLL_PARAM_ID_FBDIV	PLL param ID FBDIV
PM_PLL_PARAM_ID_DATA	PLL param ID DATA
PM_PLL_PARAM_ID_PRE_SRC	PLL param ID PRE_SRC
PM_PLL_PARAM_ID_POST_SRC	PLL param ID POST_SRC
PM_PLL_PARAM_ID_LOCK_DLY	PLL param ID LOCK_DLY
PM_PLL_PARAM_ID_LOCK_CNT	PLL param ID LOCK_CNT
PM_PLL_PARAM_ID_LFHF	PLL param ID LFHF
PM_PLL_PARAM_ID_CP	PLL param ID CP
PM_PLL_PARAM_ID_RES	PLL param ID RES

Enumeration XPmPllMode

PLL Modes

Table 340: **Enumeration XPmPllMode Values**

Value	Description
PM_PLL_MODE_INTEGER	PLL mode integer
PM_PLL_MODE_FRACTIONAL	PLL mode fractional
PM_PLL_MODE_RESET	PLL mode reset

Enumeration XPmPinFn

Pin Function IDs

Table 341: **Enumeration XPmPinFn Values**

Value	Description
PINCTRL_FUNC_CAN0	Pin Function CAN0
PINCTRL_FUNC_CAN1	Pin Function CAN1
PINCTRL_FUNC_ETHERNET0	Pin Function ETHERNET0
PINCTRL_FUNC_ETHERNET1	Pin Function ETHERNET1
PINCTRL_FUNC_ETHERNET2	Pin Function ETHERNET2
PINCTRL_FUNC_ETHERNET3	Pin Function ETHERNET3
PINCTRL_FUNC_GEMTSU0	Pin Function GEMTSU0
PINCTRL_FUNC_GPIO0	Pin Function GPIO0
PINCTRL_FUNC_I2C0	Pin Function I2C0
PINCTRL_FUNC_I2C1	Pin Function I2C1
PINCTRL_FUNC_MDIO0	Pin Function MDIO0

Table 341: **Enumeration XPmPinFn Values** *(cont'd)*

Value	Description
PINCTRL_FUNC_MDIO1	Pin Function MDIO1
PINCTRL_FUNC_MDIO2	Pin Function MDIO2
PINCTRL_FUNC_MDIO3	Pin Function MDIO3
PINCTRL_FUNC_QSPI0	Pin Function QSPI0
PINCTRL_FUNC_QSPI_FBCLK	Pin Function QSPI_FBCLK
PINCTRL_FUNC_QSPI_SS	Pin Function QSPI_SS
PINCTRL_FUNC_SPI0	Pin Function SPI0
PINCTRL_FUNC_SPI1	Pin Function SPI1
PINCTRL_FUNC_SPI0_SS	Pin Function SPI0_SS
PINCTRL_FUNC_SPI1_SS	Pin Function SPI1_SS
PINCTRL_FUNC_SDIO0	Pin Function SDIO0
PINCTRL_FUNC_SDIO0_PC	Pin Function SDIO0_PC
PINCTRL_FUNC_SDIO0_CD	Pin Function SDIO0_CD
PINCTRL_FUNC_SDIO0_WP	Pin Function SDIO0_WP
PINCTRL_FUNC_SDIO1	Pin Function SDIO1
PINCTRL_FUNC_SDIO1_PC	Pin Function SDIO1_PC
PINCTRL_FUNC_SDIO1_CD	Pin Function SDIO1_CD
PINCTRL_FUNC_SDIO1_WP	Pin Function SDIO1_WP
PINCTRL_FUNC_NAND0	Pin Function NAND0
PINCTRL_FUNC_NAND0_CE	Pin Function NAND0_CE
PINCTRL_FUNC_NAND0_RB	Pin Function NAND0_RB
PINCTRL_FUNC_NAND0_DQS	Pin Function NAND0_DQS
PINCTRL_FUNC_TTC0_CLK	Pin Function TTC0_CLK
PINCTRL_FUNC_TTC0_WAV	Pin Function TTC0_WAV
PINCTRL_FUNC_TTC1_CLK	Pin Function TTC1_CLK
PINCTRL_FUNC_TTC1_WAV	Pin Function TTC1_WAV
PINCTRL_FUNC_TTC2_CLK	Pin Function TTC2_CLK
PINCTRL_FUNC_TTC2_WAV	Pin Function TTC2_WAV
PINCTRL_FUNC_TTC3_CLK	Pin Function TTC3_CLK
PINCTRL_FUNC_TTC3_WAV	Pin Function TTC3_WAV
PINCTRL_FUNC_UART0	Pin Function UART0
PINCTRL_FUNC_UART1	Pin Function UART1
PINCTRL_FUNC_USB0	Pin Function USB0
PINCTRL_FUNC_USB1	Pin Function USB1
PINCTRL_FUNC_SWDT0_CLK	Pin Function SWDT0_CLK
PINCTRL_FUNC_SWDT0_RST	Pin Function SWDT0_RST
PINCTRL_FUNC_SWDT1_CLK	Pin Function SWDT1_CLK
PINCTRL_FUNC_SWDT1_RST	Pin Function SWDT1_RST
PINCTRL_FUNC_PMU0	Pin Function PMU0

Table 341: **Enumeration XPmPinFn Values** *(cont'd)*

Value	Description
PINCTRL_FUNC_PCIE0	Pin Function PCIE0
PINCTRL_FUNC_CSU0	Pin Function CSU0
PINCTRL_FUNC_DPAUX0	Pin Function DPAUX0
PINCTRL_FUNC_PJTAG0	Pin Function PJTAG0
PINCTRL_FUNC_TRACE0	Pin Function TRACE0
PINCTRL_FUNC_TRACE0_CLK	Pin Function TRACE0_CLK
PINCTRL_FUNC_TESTSCAN0	Pin Function TESTSCAN0

Enumeration XPmPinParam

PIN Control Parameters

Table 342: **Enumeration XPmPinParam Values**

Value	Description
PINCTRL_CONFIG_SLEW_RATE	Pin config slew rate
PINCTRL_CONFIG_BIAS_STATUS	Pin config bias status
PINCTRL_CONFIG_PULL_CTRL	Pin config pull control
PINCTRL_CONFIG_SCHMITT_CMOS	Pin config schmitt CMOS
PINCTRL_CONFIG_DRIVE_STRENGTH	Pin config drive strength
PINCTRL_CONFIG_VOLTAGE_STATUS	Pin config voltage status
PINCTRL_CONFIG_MIO_TRI_STATE	Pin config tri state for MIO pins

Enumeration pm_feature_id

IOCTL IDs

Table 343: **Enumeration pm_feature_id Values**

Value	Description
XPM_FEATURE_INVALID	Invalid ID
XPM_FEATURE_OVERTEMP_STATUS	Over temperature status
XPM_FEATURE_OVERTEMP_VALUE	Over temperature limit
XPM_FEATURE_EXTWDT_STATUS	External watchdog status
XPM_FEATURE_EXTWDT_VALUE	External watchdog interval

Enumeration XPm_SdConfigType

Config types for SD configs at run time

Table 344: **Enumeration XPm_SdConfigType Values**

Value	Description
SD_CONFIG_INVALID	Invalid SD config
SD_CONFIG_EMMC_SEL	To set SD_EMMC_SEL in CTRL_REG_SD and SD_SLOTTYPE
SD_CONFIG_BASECLK	To set SD_BASECLK in SD_CONFIG_REG1
SD_CONFIG_8BIT	To set SD_8BIT in SD_CONFIG_REG2
SD_CONFIG_FIXED	To set fixed config registers

Enumeration XPm_GemConfigType

Config types for GEM configs at run time

Table 345: **Enumeration XPm_GemConfigType Values**

Value	Description
GEM_CONFIG_INVALID	Invalid GEM config
GEM_CONFIG_SGMII_MODE	To set GEM_SGMII_MODE in GEM_CLK_CTRL register
GEM_CONFIG_FIXED	To set fixed config registers

Enumeration XPm_UsbConfigType

Table 346: **Enumeration XPm_UsbConfigType Values**

Value	Description
USB_CONFIG_INVALID	Invalid USB config
USB_CONFIG_FIXED	To set fixed config registers

Definitions

Define XST_PM_INTERNAL

Definition

```
#define XST_PM_INTERNAL2000L
```

Description

Power management specific return error status 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

Error Event Mask

Library Parameters in MSS File

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

The list of flags is as follows:

- rpu0_as_power_management_master
- rpu1_as_power_management_master

- apu_as_power_management_master
- rpu0_as_reset_management_master
- rpu1_as_reset_management_master
- apu_as_reset_management_master

The flags have two values:

- TRUE (default value): Master(APU/RPU0/RPU1) is enabled as power/reset management master
- FALSE: Master(APU/RPU0/RPU1) is disabled as power/reset management master

Note: Flag rpu1_as_power_management_master and rpu1_as_reset_management_master are only counted if the RPU is in split mode.

Data Structure Index

The following is a list of data structures:

- XPm_DeviceStatus
- XPm_NodeStatus
- XPm_Notifier
- pm_acknowledge
- pm_init_suspend

pm_acknowledge

Declaration

```
typedef struct
{
  volatile bool received,
  enum XPmNodeId node,
  XStatus status,
  u32 opp
} pm_acknowledge;
```

Table 347: **Structure pm_acknowledge member description**

Member	Description
received	Has acknowledge argument been received?
node	Node argument about which the acknowledge is
status	Acknowledged status

Table 347: **Structure pm_acknowledge member description** *(cont'd)*

Member	Description
opp	Operating point of node in question

pm_init_suspend

Declaration

```
typedef struct
{
  volatile bool received,
  enum XPmSuspendReason reason,
  u32 latency,
  u32 state,
  u32 timeout
} pm_init_suspend;
```

Table 348: **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_DeviceStatus

Contains the device status information.

Declaration

```
typedef struct
{
  u32 Status,
  u32 Requirement,
  u32 Usage
} XPm_DeviceStatus;
```

Table 349: **Structure XPm_DeviceStatus member description**

Member	Description
Status	Device power state
Requirement	Requirements placed on the device by the caller
Usage	Usage info (which subsystem is using the device)

XPm_NodeStatus

XPm_NodeStatus - struct containing node status information

Declaration

```
typedef struct
{
  u32 status,
  u32 requirements,
  u32 usage
} XPm_NodeStatus;
```

Table 350: **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),
  enum XPmNodeId node,
  enum XPmNotifyEvent event,
  u32 flags,
  volatile u32 oppoint,
  volatile u32 received,
  struct XPm_Ntfier * next
} XPm_Notifier;
```

Table 351: **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.

Table 351: **Structure XPm_Notifier member description** *(cont'd)*

Member	Description
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 v6.4

Overview

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

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

Supported Features

AMD Zynq™ UltraScale+™ MPSoC platform

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

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

Versal™ Adaptive SoC

The following features are supported in the Versal platform:

- Loading PL reconfiguration or DFX PDI (PDI with PL and NoC configuration data)
- Loading PL reconfiguration or DFX PDI with Device Key Encryption enabled
- Loading PL reconfiguration or DFX PDI with Authentication enabled

- Loading PL reconfiguration or DFX PDI with Authentication and Device Key Encryption enabled

Zynq UltraScale+ MPSoC XilFPGA Library

The library when used for Zynq UltraScale+ MPSoC runs on top of standalone BSPs.

It is tested for Arm Cortex-A53, Arm Cortex-R5F, and MicroBlaze. In the most common use case, you should run this library on the PMU MicroBlaze with PMU firmware to serve requests from either Linux or U-Boot for bitstream programming.

XilFPGA library Interface modules

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

Processor Configuration Access Port (PCAP)

The processor configuration access port (PCAP) is used to configure the programmable logic (PL) through the PS.

CSU DMA driver

The CSU DMA driver is used to transfer the actual bitstream file for the PS to PL after PCAP initialization.

XilSecure Library

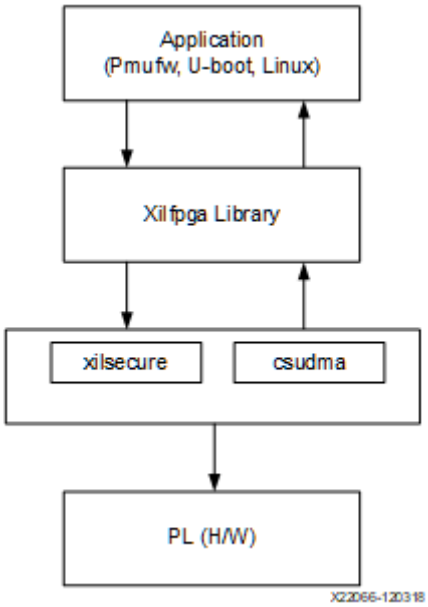
The XilSecure library provides APIs to access secure hardware on the Zynq UltraScale+ MPSoC devices.

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.

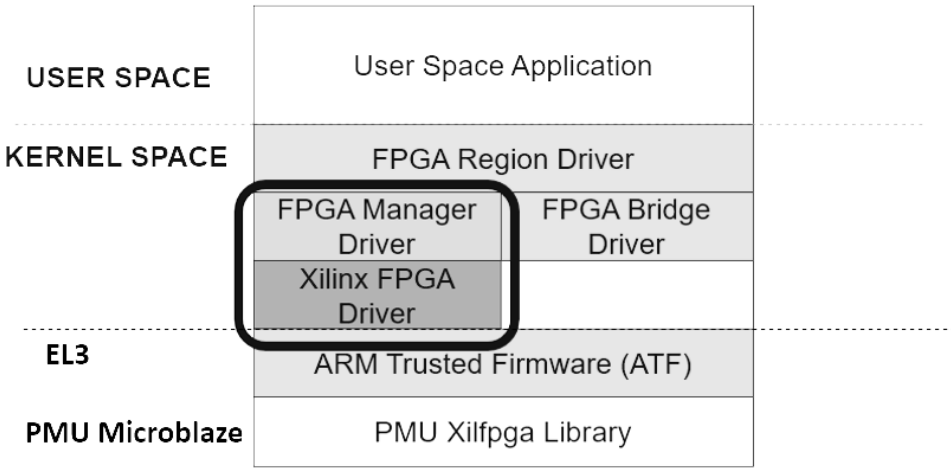
Figure 3: **XilFPGA Design Summary**



Flow Diagram

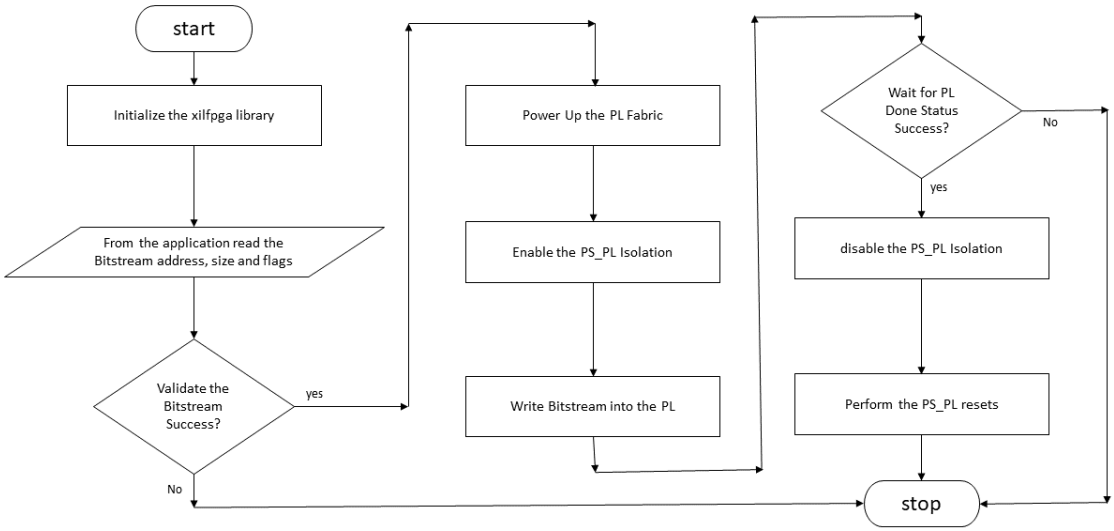
The following figure illustrates the Bitstream loading flow on the Linux operating system.

Figure 4: **Bitstream loading on Linux**



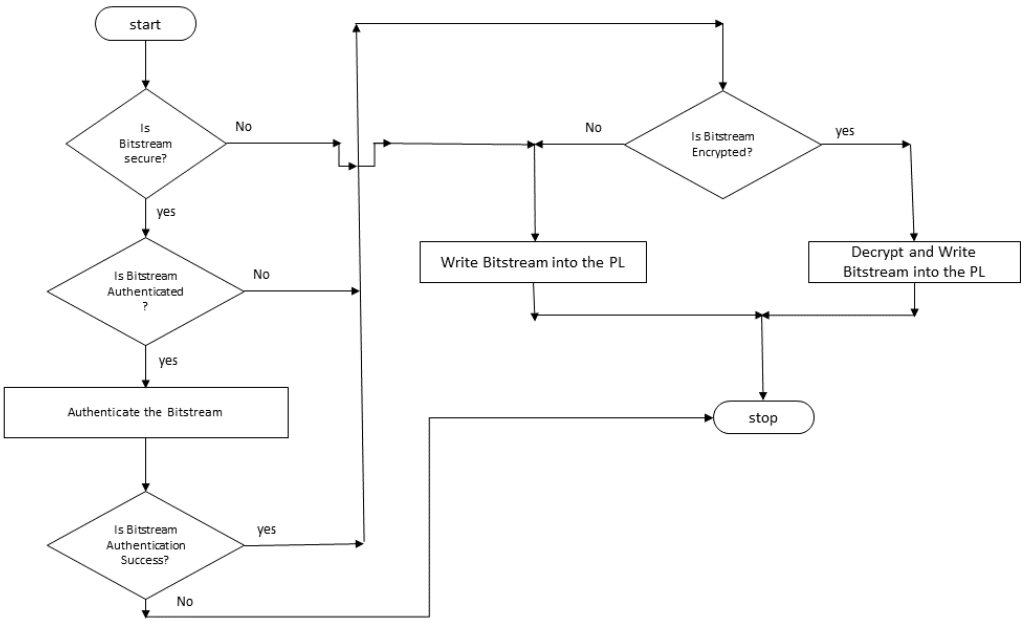
The following figure illustrates the XilFPGA PL configuration sequence.

Figure 5: **XilFPGA PL Configuration Sequence**



The following figure illustrates the Bitstream write sequence.

Figure 6: **Bitstream write Sequence**



XilFPGA BSP Configuration Settings

Xilfpga provides the following user configuration BSP settings.

Table 352: **User Configuration BSP Settings**

Parameter Name	Type	Default Value	Description
secure_mode	bool	TRUE	Enables secure Bitstream loading support.
debug_mode	bool	FALSE	Enables the Debug messages in the library.
ocm_address	int	0xfffc0000	Address used for the Bitstream authentication.
base_address	int	0x80000	Holds the Bitstream Image address. This flag is valid only for Cortex-A53 or 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.
secure_environment	bool	FALSE	Enable the secure PL configuration using the IPI. This flag is valid only for Cortex-A53 or 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.

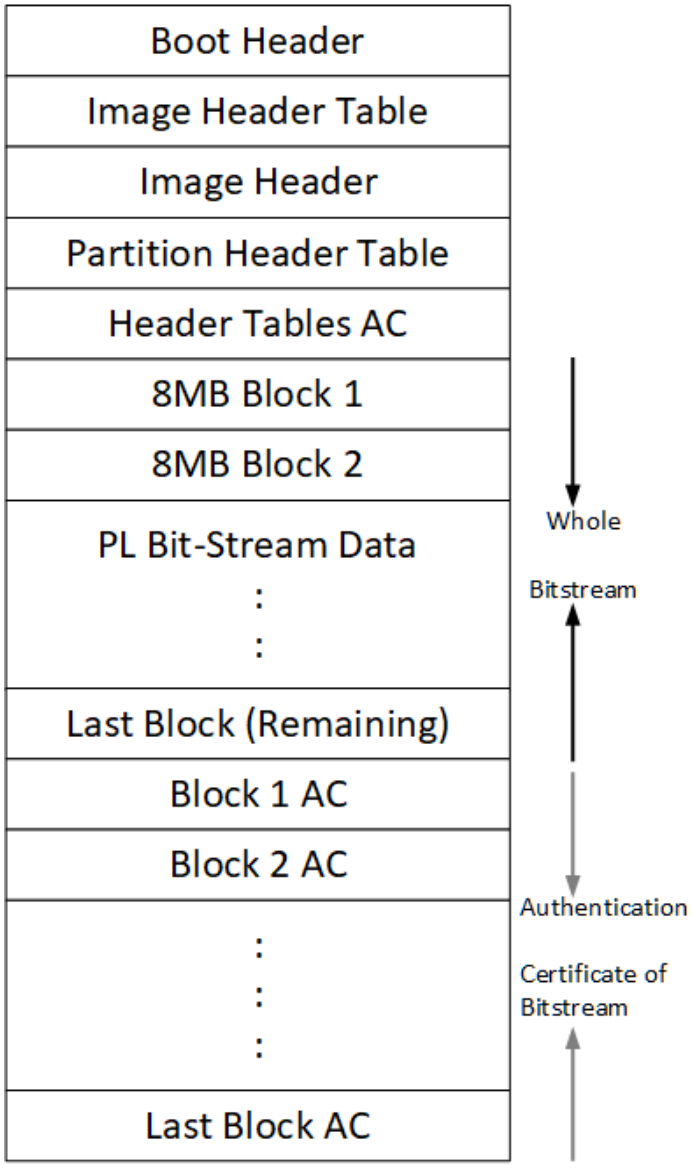
- The second method uses external DDR for authentication prior to sending the data to the decryptor, there by requiring trust in the external DDR. For details, see Authenticated and Encrypted Bitstream Loading Using DDR.

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.

Loading an Authenticated and Encrypted Bitstream using OCM

To authenticate the Bitstream partition securely, XilFPGA uses the FSBL section's OCM memory to copy the bitstream in chunks from DDR.

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 56KB of every 8MB block.
2. XilFPGA copies a 56KB chunk from the first 8MB block to Read Buffer.
3. XilFPGA calculates hash on 56KB 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).

Loading an Authenticated and Encrypted Bitstreamusing DDR Memory Controller

The software workflow for authenticating Bitstream is as follows:

1. XilFPGA identifies DDR secure Bitstream image base address.
2. XilFPGA calculates hash for the first 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.

Versal Adaptive SoC XilFPGA Library

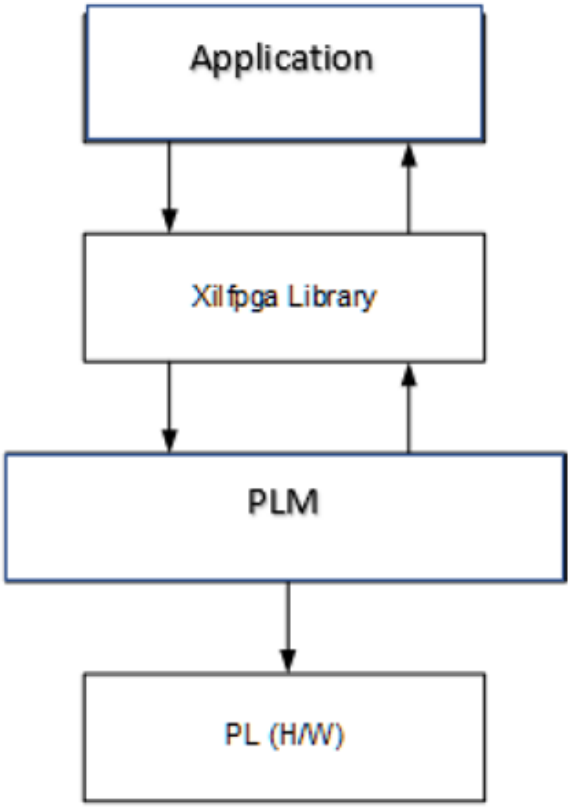
The library, when used for Versal devices, runs on top of standalone BSPs.

It is tested for Arm Cortex-A72 and Arm Cortex-R5F. The most common use-case is that the user can run this on either Arm Cortex-A72 or Arm Cortex-R5F and requests PLM to load the bitstream (PDI) on to the PL. In Versal devices, the bitstream is always generated in the PDI file format.

Design Summary

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

Figure 8: **XilFPGA Design Summary**



BSP Configuration Settings

XilFPGA provides the following user configuration BSP settings.

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

XilFPGA APIs

This section provides detailed descriptions of the XilFPGA library APIs.

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

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

XilFPGA Top Layer

The functionality exist in this layers is completely interface agnostic. It provides a unique interface to load the Bitstream across multiple platforms.

Interface Specific Layer

This layer is responsible for providing the interface-specific errors. In case of Zynq UltraScale+ MPSoC, it provides the errors related to PCAP interface.

XilFPGA Lower Layer

This layer is responsible for providing the error related to the lower level drivers used by interface layer.

XilFPGA APIs for Versal Adative SoC and Zynq UltraScale+ MPSoC

This file contains the definitions of Bitstream loading functions.

Table 353: **Quick Function Reference**

Type	Name	Arguments
u32	XFpga_BitStream_Load	`XFpga` * InstancePtr UINTPTR BitstreamImageAddr UINTPTR KeyAddr u32 Size u32 Flags
u32	XFpga_ValidateImage	`XFpga` * InstancePtr UINTPTR BitstreamImageAddr UINTPTR KeyAddr u32 Size u32 Flags
u32	XFpga_PL_Preconfig	`XFpga` * InstancePtr

Table 353: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
u32	XFpga_Write_PI	`XFpga` * InstancePtr UINTPTR BitstreamImageAddr UINTPTR KeyAddr u32 Size u32 Flags
u32	XFpga_PL_PostConfig	`XFpga` * InstancePtr
u32	XFpga_GetVersion	u32 * Version
u32	XFpga_GetFeatureList	`XFpga` * InstancePtr
u32	XFpga_Initialize	void

Functions

XFpga_BitStream_Load

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

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

Prototype

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

Parameters

The following table lists the `XFpga_BitStream_Load` function arguments.

Table 354: **XFpga_BitStream_Load Arguments**

Type	Name	Description
`XFpga` *	InstancePtr	Pointer to the XFpga structure.
UINTPTR	BitstreamImageAddr	Linear memory bitstream image base address
UINTPTR	KeyAddr	AES key address which is used for decryption.
u32	Size	Used to store size of bitstream image.

Table 354: **XFpga_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_ValidateImage

This function is used to validate the bitstream image.

Prototype

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

Parameters

The following table lists the `XFpga_ValidateImage` function arguments.

Table 355: **XFpga_ValidateImage Arguments**

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

Returns

Codes as mentioned in xilfpga.h

XFpga_PL_Preconfig

This function prepares the FPGA to receive configuration data.

Prototype

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

Parameters

The following table lists the `XFpga_PL_Preconfig` function arguments.

Table 356: **XFpga_PL_Preconfig Arguments**

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

Returns

Codes as mentioned in xilfpga.h

XFpga_Write_PI

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

Prototype

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

Parameters

The following table lists the `XFpga_Write_Pl` function arguments.

Table 357: **XFpga_Write_PI Arguments**

Type	Name	Description
`XFpga`	InstancePtr	Pointer to the XFpga structure
UINTPTR	BitstreamImageAddr	Linear memory bitstream image base address
UINTPTR	KeyAddr	Aes key address which is used for decryption.
u32	Size	Used to store size of bitstream image.

Table 357: **XFpga_Write_PI 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_PL_PostConfig

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

Prototype

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

Parameters

The following table lists the `XFpga_PL_PostConfig` function arguments.

Table 358: **XFpga_PL_PostConfig Arguments**

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

Returns

Codes as mentioned in xilfpga.h

XFpga_GetVersion

This function is used to read xilfpga library version info.

***Note*:** This API is not supported for the Versal platform.

Prototype

```
u32 XFpga_GetVersion(u32 *Version);
```

Parameters

The following table lists the `XFpga_GetVersion` function arguments.

Table 359: **XFpga_GetVersion Arguments**

Type	Name	Description
u32 *	Version	xilfpga library version to read

Returns

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

XFpga_GetFeatureList

This function is used to Get xilfpga component supported feature list.

***Note*:**

- This API is not supported for the Versal platform.

Prototype

```
u32 XFpga_GetFeatureList(XFpga *InstancePtr, u32 *FeatureList);
```

Parameters

The following table lists the `XFpga_GetFeatureList` function arguments.

Table 360: **XFpga_GetFeatureList Arguments**

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

Returns

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

XFpga_Initialize

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

Prototype

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

XilFPGA APIs for Zynq UltraScale+ MPSoC

Table 361: **Quick Function Reference**

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

Functions

XFpga_GetPlConfigData

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

***Note*:** This API is not supported for the Versal platform.

Prototype

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

Parameters

The following table lists the `XFpga_GetPlConfigData` function arguments.

Table 362: **XFpga_GetPlConfigData Arguments**

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

Returns

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

XFpga_GetPlConfigReg

This function provides PL specific configuration register values.

***Note*:** This API is not supported for the Versal platform.

Prototype

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

Parameters

The following table lists the `XFpga_GetPlConfigReg` function arguments.

Table 363: **XFpga_GetPlConfigReg Arguments**

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

Returns

- XFPGA_SUCCESS if, successful

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

XFpga_InterfaceStatus

This function provides the status of the PL programming interface.

***Note*:** This API is not supported for the Versal platform.

Prototype

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

Parameters

The following table lists the `XFpga_InterfaceStatus` function arguments.

Table 364: **XFpga_InterfaceStatus Arguments**

Type	Name	Description
`XFpga`	InstancePtr	Pointer to the XFpga structure

Returns

Status of the PL programming interface

XilMailbox Library v1.7

Overview

This file contains the definitions for Zynq UltraScale+ MPSoC and Versal devices IP Integrator implementation.

This file contains the definitions for Mailbox library top level functions. This file contains the definitions for Mailbox library top level functions.

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. 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.

For a full description of IPI features, please see the hardware spec. This library supports the following features:For a full description of IPI features, please see the hardware spec. This library supports the following features:

Table 365: **Quick Function Reference**

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

Table 365: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
u32	XIpiPs_RecvData	XMailbox * InstancePtr void * MsgBufferPtr u32 MsgLen u8 BufferType
XStatus	XIpiPs_RegisterIrq	XScuGic * IntcInstancePtr XMailbox * InstancePtr u32 IpiIntrId
void	XIpiPs_ErrorIntrHandler	void * XMailboxPtr
void	XIpiPs_IntrHandler	void * XMailboxPtr
u32	XMailbox_Initialize	XMailbox * InstancePtr u8 DeviceId
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 void * CallBackFuncPtr void * CallBackRefPtr

Functions

XIpiPs_Init

Initialize the Zynq UltraScale+ MPSoC Mailbox Instance.

Prototype

```
u32 XIpiPs_Init(XMailbox *InstancePtr, u8 DeviceId);
```

Parameters

The following table lists the `XIpiPs_Init` function arguments.

Table 366: **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 367: **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 368: **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 369: **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 370: **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

This function registers an irq.

Prototype

```
XStatus XIpiPs_RegisterIrq(XScuGic *IntcInstancePtr, XMailbox *InstancePtr,
u32 IpiIntrId);
```

Parameters

The following table lists the `XIpiPs_RegisterIrq` function arguments.

Table 371: **XIpiPs_RegisterIrq Arguments**

Type	Name	Description
XScuGic *	IntcInstancePtr	Pointer to the scugic instance
`XMailbox` *	InstancePtr	Pointer to the `XMailbox` instance
u32	IpiIntrId	is the interrupt id of the IPI

Returns

- XST_SUCCESS if successful
- XST_FAILURE if unsuccessful

XIpiPs_ErrorIntrHandler

This function implements the interrupt handler for errors.

Prototype

```
void XIpiPs_ErrorIntrHandler(void *XMailboxPtr);
```

Parameters

The following table lists the `XIpiPs_ErrorIntrHandler` function arguments.

Table 372: **XIpiPs_ErrorIntrHandler Arguments**

Type	Name	Description
void *	XMailboxPtr	Pointer to the `XMailbox` instance

Returns

None

XIpiPs_IntrHandler

This function implements the interrupt handler.

Prototype

```
void XIpiPs_IntrHandler(void *XMailboxPtr);
```

Parameters

The following table lists the `XIpiPs_IntrHandler` function arguments.

Table 373: **XIpiPs_IntrHandler Arguments**

Type	Name	Description
void *	XMailboxPtr	Pointer to the `XMailbox` instance

Returns

None

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 374: **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

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 375: **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 376: **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 377: **XMailbox_Recv Arguments**

Type	Name	Description
`XMailbox` *	InstancePtr	Pointer to the `XMailbox` instance

Table 377: **XMailbox_Recv Arguments** *(cont'd)*

Type	Name	Description
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.

Prototype

```
s32 XMailbox_SetCallBack(XMailbox *InstancePtr, XMailbox_Handler
HandlerType, void *CallBackFuncPtr, void *CallBackRefPtr);
```

Parameters

The following table lists the `XMailbox_SetCallBack` function arguments.

Table 378: **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.
void *	CallBackFuncPtr	is the address of the callback function.
void *	CallBackRefPtr	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_FAILURE when HandlerType is invalid.

Enumerations

Enumeration XMailbox_Handler

This typedef contains XMAILBOX Handler Types.

Table 379: **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_Agent

XMailbox

Data structure used to refer XilMailbox.

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 380: **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.

Table 380: **Structure XMailbox member description** *(cont'd)*

Member	Description
RecvHandler	Recieve handler.
ErrorHandler	Callback for RX IPI event.
ErrorRefPtr	To be passed to the error interrupt callback.
RecvRefPtr	To be passed to the receive interrupt callback.
Agent	Agent to store IPI channel information.

XMailbox_Agent

Data structure used to refer Xilmailbox agents.

Declaration

```
typedef struct
{
  XIpiPsu IpiInst,
  XScuGic GicInst,
  u32 SourceId,
  u32 RemoteId
} XMailbox_Agent;
```

Table 381: **Structure XMailbox_Agent member description**

Member	Description
IpiInst	IPI instance.
GicInst	Interrupt instance.
SourceId	Source ID.
RemoteId	Remote ID.

Chapter 12

XilSEM Library v1.6

Introduction

The Xilinx Soft Error Mitigation (XilSEM) library is a pre-configured, pre-verified solution to detect and optionally correct soft errors in Configuration Memory of Versal ACAPs. A soft error is caused by ionizing radiation and is extremely uncommon in commercial terrestrial operating environments. While a soft error does not damage the device, it carries a small statistical possibility of transiently altering the device behavior.

The XilSEM library does not prevent soft errors; however, it provides a method to better manage the possible system-level effect. Proper management of a soft error can increase reliability and availability, and reduce system maintenance and downtime. In most applications, soft errors can be ignored. In applications where a soft error cannot be ignored, this library documentation provides information to configure the XilSEM library through the CIPS IP core and interact with the XilSEM library at runtime.

The XilSEM library is part of the Platform Loader and Manager (PLM) which is loaded into and runs on the Platform Management Controller (PMC).

When a soft error occurs, one or more bits of information are corrupted. The information affected might be in the device Configuration Memory (which determines the intended behavior of the design) or might be in design memory elements (which determine the state of the design). In the Versal architecture, Configuration Memory includes Configuration RAM and NPI Registers. Configuration RAM is associated with the Programmable Logic (PL) and is used to configure the function of the design loaded into the PL. This includes function block behavior and function block connectivity. This memory is physically distributed across the PL and is the larger category of Configuration Memory by bit count. Only a fraction of these bits are essential to the correct operation of the associated resources for the currently loaded design.

NPI Registers are associated with other function blocks which can be configured independently from the PL. The Versal Architecture integrated shell is a notable application of function blocks configured in this manner, making it possible for the integrated shell to be operational with the PL in an unconfigured state. This memory is physically distributed throughout the device and is the smaller category of Configuration Memory by bit count. Only a fraction of these bits are essential to the correct operation of the associated resources for the currently loaded design.

Prior to configuring the XilSEM library for use through the CIPS IP core, assess whether the XilSEM library helps meet the soft error mitigation goals of your system's deployment and what mitigation approaches should be included, if any. There are two main considerations:

- Understanding the soft error mitigation requirements for your system
- What design and system actions need to be taken if a soft error occurs

If the effects of soft errors are a concern for your system's deployment, each component in the system design will usually need a soft error rate (SER) budget distributed from a system-level SER budget. To make estimates for the contribution from a Versal ACAP, use the Xilinx SEU Estimator tool, which is tentatively planned for availability after the UG116 publication of production qualification data for Xilinx 7 nm technology. To estimate SER for a deployment, at the minimum you need:

- Target device selection(s) to be deployed
- Estimated number of devices in deployment

In addition to providing a device level estimate, the Xilinx SEU Estimator also estimates the number of soft errors statistically likely to occur across the deployment for a selected operation time interval. After an estimate has been generated, it must be used to evaluate if the SER requirement for the deployment is met. Other design approaches might need to be implemented to reduce the system-level SER. These approaches include addition of mitigation solutions in the system, which might be suitable for implementation in any devices in the system which can offer such features. Versal ACAPs offer many supporting features, including the XilSEM library. The XilSEM library can be configured to detect or detect and correct soft errors. Therefore, consideration must be given to the system-level response, if any, that must be taken in response to a soft error in the Versal ACAP. If no action is taken when a soft error is detected in the Versal ACAP, the benefits of using the XilSEM library should be assessed and understood. While this is a valid use case, the effects of soft errors in the Versal ACAP should be anticipated and this approach should be reviewed to ensure it supports the system-level soft error mitigation strategy.

If there is no SER target at the system level, then it is unclear what system changes (and tradeoffs that come with them) need to be made to mitigate soft errors, including whether a deployment benefits from use of the XilSEM library.

XilSEM Features

Following are the features of the XilSEM library:

- Integration of silicon features to fully leverage and improve upon the inherent protections for Configuration Memory.
- Scan-based detection of soft errors in Configuration RAM using ECC and CRC techniques.
- Algorithmic correction of soft errors in Configuration RAM using ECC techniques.

- Scan-based detection of soft errors in NPI Registers using SHA techniques.
- Error injection to support evaluation of system integration and response.

XilSEM library supports the following devices:

- VC1902, VM1802, VP1202, VM1402, and VM1502
- VP1802 (SSI technology support available from 2022.2 release onwards)
- VP1502, VP1702 (support is available from 2023.1 release onwards)

Unsupported Features

The following features are not supported by the XilSEM library:

- The XilSEM library does not operate on soft errors outside of the configuration RAM and NPI Registers. Soft error mitigation in other device resources, if necessary, based on design requirements, must be addressed at the user design level through measures such as redundancy or error detection and correction codes.
- The XilSEM library initializes and manages the integrated silicon features for soft error mitigation and when included in a design, do not modify settings of associated features in the PMC.
- Design simulations that include the XilSEM library are supported. However, it is not possible to observe the library and integrated silicon features in simulation. Hardware-based evaluation is required.
- The XilSEM library is not explicitly designed for use under high irradiation environments, including but not limited to the space or artificially generated environments. Therefore, Xilinx offers only limited guidance specific to the application of the XilSEM library in such environments.
- Debug prints are not enabled in the XilSEM library. For debug information, use the XSem_CmdCfrGetStatus and XSem_CmdNpiGetStatus. See XilSEM Library Versal Adaptive SoC Client APIs for more information. You will be notified via optional IPI and GPO. For more information, see Listening for SEU Detection.
- XilSEM support for SSIT with DFX designs is not available. The XilSEM library is delivered pre-compiled as a pre-verified solution; source code is not provided. User modification of the XilSEM library is not supported.

Documentation

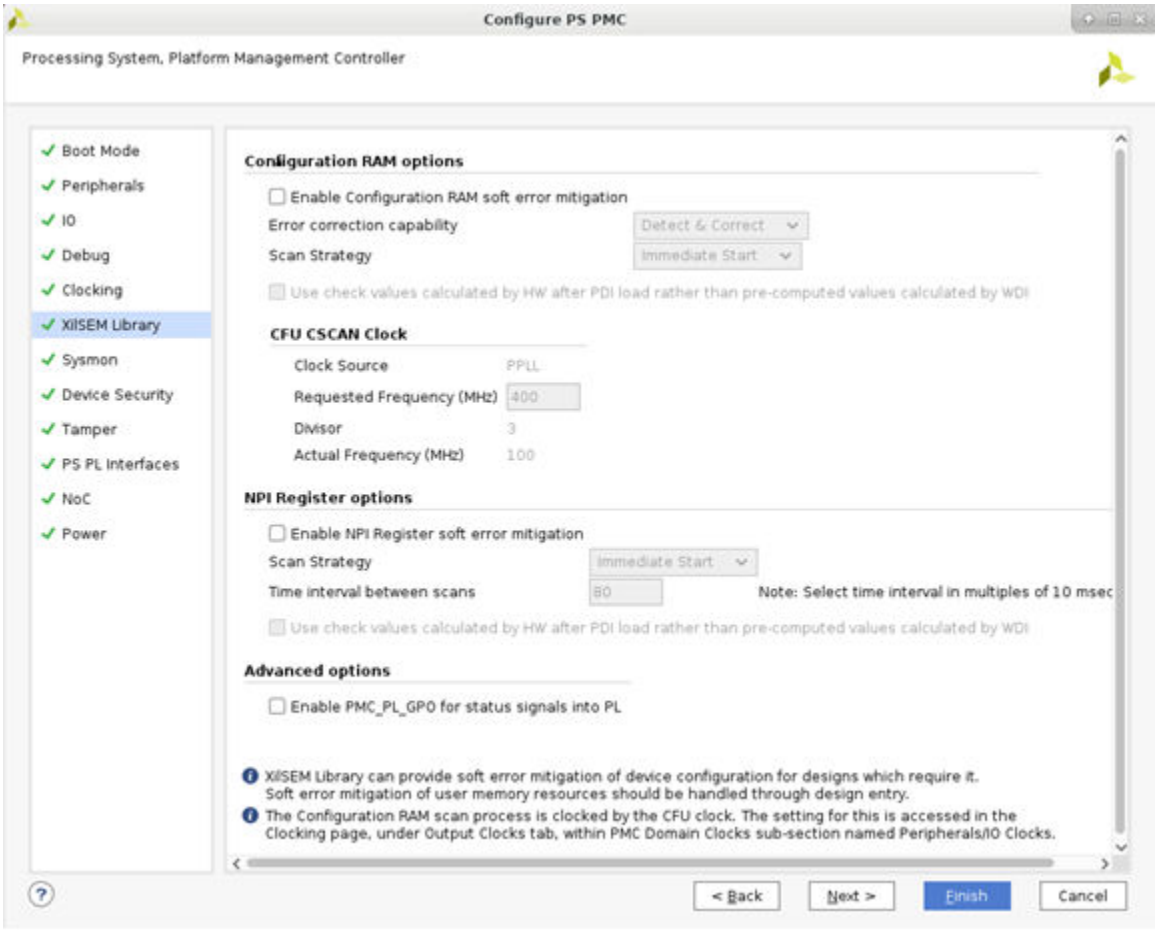
This user guide is the main documentation for the XilSEM library. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page or by using the Xilinx Documentation Navigator. Download the Xilinx Documentation Navigator from the Downloads page. For more information about this tool and the features available, open the online help after installation.

To help in the design and debug process when using the XilSEM library, the Xilinx Support web page contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support. Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product. Answer Records are created and maintained daily ensuring that you have access to the most accurate information available. The Master Answer Record for the XilSEM library is AR76513. Answer Records for the XilSEM library can be located by using the Search Support box on the main Xilinx support web page.

Configuration

The XilSEM library configuration options available through the Control Interfaces and Processing System (CIPS) Customize IP dialog box in the Vivado Design Suite are shown in the following figure.

Figure 9: **XilSEM Library Configuration within CIPS Customize IP Dialog Box in Vivado Design Suite**



From within CIPS, the fundamental feature of the XilSEM library can be enabled. This feature is the soft error mitigation of the configuration RAM through a scan-based process automated by integrated logic and managed by the XilSEM library, which reads back the configuration RAM and uses ECC and CRC to detect and correct soft errors. Some of the features of CRAM Scan can be configured through properties as shown in the following table.

Table 382: **XilSEM CRAM Scan properties**

Property Name	Supported Values	Default Value	Description
CONFIG.PS_PMC_CONFIG {SEM_MEM_ENABLE_SCAN_AFTER 1}	1 (Immediate start) 0 (Deferred start)	1 (Immediate start)	Immediate start: Enables automatic start of the configuration RAM scanning after boot. Deferred start: Start of the scan during mission

The XilSEM library provides an optional feature for the soft error mitigation of NPI Registers, provided the supplemental hardware resources required for this feature are accessible. This feature is a scan-based process automated by integrated logic and managed by the XilSEM library, which reads back NPI Registers and uses SHA to detect errors. Some of the features of the NPI Scan can be configured thorough properties as shown in Table: XilSEM NPI Properties.

The following are the different configuration options for CRAM scan.

- Scan strategy:
  - Immediate start: Enables automatic start of CRAM scanning after boot.
  - Deferred start:Start of the CRAM scan during mission based on your request.
- ECC Method:
  - HW ECC: The value calculated during the first scan will be used as golden ECC values.
  - SW ECC: Tool generated values are used as golden ECC for CRAM scan.

NPI Register scan requires XilSEM library use of PMC cryptographic acceleration. The following are some of the scenarios in which the XilSEM library cannot offer this optional feature:

- User operation of the SHA block in the PMC for purposes of the user design
- Device is programmed for `no end-user accessible` cryptographic functions

Use the following two mechanism to identify if cryptographic acceleration block is disabled in your device.

- Read device Manual
- During run time, XilSEM on PLM updates bit-31 in `SEM_NPI_SCAN_STATUS` register. The following bit status indicates if cryptographic acceleration block is present.
  - 0: cryptographic acceleration is present

- 1: cryptographic acceleration is not present

- If you try to perform NPI scan requests when cryptographic acceleration block is disabled, you will get error notification. CRAM scan has no relevance to cryptographic acceleration block and is functional.

The following are the different configuration options for CRAM scan.

- Scan strategy:
  - Immediate start: Enables automatic start of NPI scanning after boot.
  - Deferred start: Start of the NPI scan during mission based on your request.
- SHA Method:
  - HW SHA: The value calculated during the first scan will be used as golden SHA.
  - SW SHA: Tool generated values are used as golden SHA for NPI scan.

Table 383: **XilSEM NPI Properties**

Property Name	Supported Values	Default Value	Description
CONFIG.PS_PMC_CONFIG {SEM_NPI_ENABLE_SCAN_AFTER 0}	0 (Immediate start) 1 (Deferred start)	0 (Immediate start)	Immediate start: Enables automatic start of NPI scanning after boot. Deferred start: Start of the NPI scan during mission based on your request.
BITGEN.GENERATESWSHA	False (HW SHA) True (SW SHA)	True (SW SHA)	HW SHA: The value calculated during the first scan will be used as golden SHA. SW SHA: Tool generated values are used as golden SHA for NPI scan.
CONFIG.PS_PMC_CONFIG {SEM_TIME_INTERVAL_BETWEEN_SCANS 80 }	>=80 msec && <=1000 msec	80 msec	The interval in milliseconds at which the NPI scan is repeated. Recommended to use 80 msec.

Additional XilSEM library properties as referenced below are available.

Table 384: **Additional XilSEM Library Properties**

Property Name	Supported Values	Default Value	Description
CONFIG.PS_PMC_CONFIG {PMC_GPO_ENABLE 0}	0 (Disabled) 1 (Enabled)	0 (Disabled)	Disabled: PMC_PL_GPO will not be triggered for status and error information. Enabled: PMC_PL_GPO will be triggered for status and error information.

Operation

Initializing the SEU Mitigation

To initialize the SEU mitigation, it is first necessary to successfully initialize the inter-processor interrupt (IPI). The following reference code, which can be part of an RPU application, illustrates how to initialize the IPI:

```
#define TARGET_IPI_INT_DEVICE_ID
                    (XPAR_XIPIPSU_0_DEVICE_ID)

/* Load Config for Processor IPI Channel */
IpiCfgPtr = XIpiPsu_LookupConfig(TARGET_IPI_INT_DEVICE_ID);
if (NULL == IpiCfgPtr) {
    xil_printf("[%s] ERROR: IPI LookupConfig failed\n\r", \
            __func__, Status);
    goto END;
}

/* Initialize the IPI Instance pointer */
Status = XIpiPsu_CfgInitialize(&IpiInst, IpiCfgPtr,
        IpiCfgPtr-BaseAddress);
if (XST_SUCCESS != Status) {
    xil_printf("[%s] ERROR: IPI instance initialize failed\n\r", \
            __func__, Status);
    goto END;
}
```

Although Configuration RAM scan can be configured to begin automatically, in XilSEM library configurations where this option is not selected (deferred start-up), you are responsible for manually starting the Configuration RAM scan. The following reference code, which can be part of an RPU application, illustrates how to accomplish Configuration RAM scan start-up.

***Note*:** When you choose the "Deferred start-up" option, do not perform error injection or partial bit stream loading without this initialization.

```
XStatus Status = XST_FAILURE;
XSemIpiResp IpiResp={0};
Status = XSem_CmdCfrInit(&IpiInst, &IpiResp);
if ((XST_SUCCESS == Status) && (CMD_ACK_CFR_INIT == IpiResp.RespMsg1) &&
(XST_SUCCESS == IpiResp.RespMsg2)) {
       xil_printf("Success: CRAM Initialization Successful\n\r");
} else {
xil_printf("Failure: CRAM Initialization Failed\n\r");
Status = XST_FAILURE;
}
```

Similarly, NPI Register scan can be configured to begin automatically, and in XilSEM library configurations where this option is not selected, you are responsible for manually starting the NPI Register scan. The following reference code, which can be part of an RPU application, illustrates how to accomplish NPI Register scan start-up.

***Note*:** An NPI scan cannot be performed in SHA3 disabled devices. If SHA3 is not present, XilSEM updates the status in NPI scan status register. You can use the XSem_CmdNpiGetStatus API to know NPI scan status.

```
XStatus Status = XST_FAILURE;
XSemIpiResp IpiResp={0};
Status = XSem_CmdNpiStartScan(&IpiInst, &IpiResp);
if ((XST_SUCCESS == Status) && (CMD_ACK_NPI_STARTSCAN == IpiResp.RespMsg1)
&& (XST_SUCCESS == IpiResp.RespMsg2)) {
       xil_printf("Success: NPI Initialization Successful\n\r");
} else {
       xil_printf("Failure: NPI Initialization Failed\n\r");
       Status = XST_FAILURE;
}
```

Listening for SEU Detection

The XilSEM library can maintain SEU mitigation operation without any need for the user design to listen for SEU detections. However, it may be desired maintain an event log, or take design or system level actions in response to an event.

Whenever an error is detected, be it correctable or uncorrectable, XilSEM has a mechanism to notify the errors over IPI. This equally applies for errors detected during NPI scanning and CRAM scanning. To receive notification of error detections in the configuration RAM or NPI registers, the following steps are required of an RPU application:

- GIC initialization
- IPI initialization
  - IPI configuration and connection with GIC
  - IPI callback registration with IPI interrupt handler (CRAM and NPI)
- Register error event notification (CRAM and NPI)
- Check global variables that hold notified event information (CRAM and NPI)

For GIC initialization, use the following code snippet:

```
#define INTC_DEVICE_ID (XPAR_SCUGIC_SINGLE_DEVICE_ID)
s32 GicSetupInterruptSystem(XScuGic *GicInst)
{
    s32 Status;

    XScuGic_Config *GicCfgPtr = XScuGic_LookupConfig(INTC_DEVICE_ID);
    if (NULL == GicCfgPtr) {
        xil_printf("XScuGic_LookupConfig() failed\r\n");
        goto END;
    }

    Status = XScuGic_CfgInitialize(GicInst, GicCfgPtr, \
            GicCfgPtr->CpuBaseAddress);
    if (XST_SUCCESS != Status) {
        xil_printf("XScuGic_CfgInitialize() failed with error: %d\r\n",\
```

```
                Status);
        goto END;
    }

    /*
     * Connect the interrupt controller interrupt Handler to the
     * hardware interrupt handling logic in the processor.
     */
#if defined (__aarch64__)
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_FIQ_INT,
#elif defined (__arm__)
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,
#endif
         (Xil_ExceptionHandler)XScuGic_InterruptHandler, GicInst);
    Xil_ExceptionEnable();

END:
    return Status;
}
```

For IPI configuration and connection with GIC, use the following code snippet:

```
#define IPI_TEST_CHANNEL_ID
                    (XPAR_XIPIPSU_0_DEVICE_ID)
#define IPI_INT_ID
       (XPAR_XIPIPSU_0_INT_ID)

/* Allocate one callback pointer for each bit in the register */
static IpiCallback IpiCallbacks[11];

static ssize_t ipimask2idx(u32 m)
{
    return __builtin_ctz(m);
}

/**
 * IpiIrqHandler() - Interrupt handler of IPI peripheral
 * @InstancePtr    Pointer to the IPI data structure
 */
static void IpiIrqHandler(XIpiPsu *InstancePtr)
{
    u32 Mask;

    /* Read status to determine the source CPU (who generated IPI) */
    Mask = XIpiPsu_GetInterruptStatus(InstancePtr);

    /* Handle all IPIs whose bits are set in the mask */
    while (Mask) {
        u32 IpiMask = Mask & (-Mask);
        ssize_t idx = ipimask2idx(IpiMask);

        /* If the callback for this IPI is registered execute it */
        if (idx >= 0 && IpiCallbacks[idx])
            IpiCallbacks[idx](InstancePtr);

        /* Clear the interrupt status of this IPI source */
        XIpiPsu_ClearInterruptStatus(InstancePtr, IpiMask);

        /* Clear this IPI in the Mask */
        Mask &= ~IpiMask;
    }
}
```

```
static XStatus IpiConfigure(XIpiPsu * IpiInst, XScuGic * GicInst)
{
    int Status = XST_FAILURE;
    XIpiPsu_Config *IpiCfgPtr;

    if (NULL == IpiInst) {
        goto END;
    }

    if (NULL == GicInst) {
        xil_printf("%s ERROR GIC Instance is NULL\n", __func__);
        goto END;
    }

    /* Look Up the config data */
    IpiCfgPtr = XIpiPsu_LookupConfig(IPI_TEST_CHANNEL_ID);
    if (NULL == IpiCfgPtr) {
        Status = XST_FAILURE;
        xil_printf("%s ERROR in getting CfgPtr\n", __func__);
        goto END;
    }
    /* Init with the Cfg Data */
    Status = XIpiPsu_CfgInitialize(IpiInst, IpiCfgPtr, \
            IpiCfgPtr->BaseAddress);
    if (XST_SUCCESS != Status) {
        xil_printf("%s ERROR #%d in configuring IPI\n", __func__,
                Status);
        goto END;
    }

    /* Clear Any existing Interrupts */
    XIpiPsu_ClearInterruptStatus(IpiInst, XIPIPSU_ALL_MASK);

    Status = XScuGic_Connect(GicInst, IPI_INT_ID,
            (Xil_ExceptionHandler)IpiIrqHandler, IpiInst);
    if (XST_SUCCESS != Status) {
        xil_printf("%s ERROR #%d in GIC connect\n", __func__, Status);
        goto END;
    }
    /* Enable IPI interrupt at GIC */
    XScuGic_Enable(GicInst, IPI_INT_ID);

END:
    return Status;
}
```

For IPI callback registration with IPI handler, see the following code snippet:

```
XStatus IpiRegisterCallback(XIpiPsu *const IpiInst, const u32 SrcMask,
        IpiCallback Callback)
{
    ssize_t idx;

    if (!Callback)
        return XST_INVALID_PARAM;

    /* Get index into IpiChannels array */
    idx = ipimask2idx(SrcMask);
    if (idx < 0)
        return XST_INVALID_PARAM;
```

```
    /* Check if callback is already registered, return failure if it is */
    if (IpiCallbacks[idx])
        return XST_FAILURE;

    /* Entry is free, register callback */
    IpiCallbacks[idx] = Callback;

    /* Enable reception of IPI from the SrcMask/CPU */
    XIpiPsu_InterruptEnable(IpiInst, SrcMask);

    return XST_SUCCESS;
}
```

For IPI callback to receive event messages for CRAM, see the following code snippet:

```
#define SRC_IPI_MASK
                    (XPAR_XIPIPS_TARGET_PSV_PMC_0_CH0_MASK)

/*Global variables to hold the event count when notified*/
u8 EventCnt_UnCorEcc = 0U;
u8 EventCnt_Crc = 0U;
u8 EventCnt_CorEcc = 0U;
u8 EventCnt_IntErr = 0U;

void XSem_IpiCallback(XIpiPsu *const InstancePtr)
{
    int Status;
    u32 Payload[PAYLOAD_ARG_CNT] = {0};

    Status = XIpiPsu_ReadMessage(XSem_IpiGetInst(),SRC_IPI_MASK,Payload,\
            PAYLOAD_ARG_CNT, XIPIPSU_BUF_TYPE_MSG);
    if (Status != XST_SUCCESS) {
        xil_printf("ERROR #%d while reading IPI buffer\n", Status);
        return;
    }

    if ((XSEM_EVENT_ERROR == Payload[0]) && \
            (XSEM_NOTIFY_CRAM == Payload[1])) {
        if (XSEM_EVENT_CRAM_UNCOR_ECC_ERR == Payload[2]) {
            EventCnt_UnCorEcc++;
        } else if (XSEM_EVENT_CRAM_CRC_ERR == Payload[2]) {
            EventCnt_Crc++;
        } else if (XSEM_EVENT_CRAM_INT_ERR == Payload[2]) {
            EventCnt_IntErr++;
        } else if (XSEM_EVENT_CRAM_COR_ECC_ERR == Payload[2]) {
            EventCnt_CorEcc++;
        } else {
            xil_printf("%s Some other callback received: %d:%d:%d\n",
                    __func__, Payload[0], \
                    Payload[1], Payload[2]);
        }
    } else {
        xil_printf("%s Some other callback received: %d\n", \
                __func__, Payload[0]);
    }
}
```

Note: In the above code snippets, global counters are incremented when an event has been notified. It is up to you to define and implement any desired design and system response.

For IPI callback to receive event messages for NPI, see the following code snippet:

```
/*Global variables to hold the event count when notified*/
u8 NPI_CRC_EventCnt = 0U;
u8 NPI_INT_EventCnt = 0U;

void XSem_IpiCallback(XIpiPsu *const InstancePtr)
{
    int Status;
    u32 Payload[PAYLOAD_ARG_CNT] = {0};

    Status = XIpiPsu_ReadMessage(&IpiInst, SRC_IPI_MASK, Payload,
PAYLOAD_ARG_CNT,
            XIPIPSU_BUF_TYPE_MSG);
    if (Status != XST_SUCCESS) {
        xil_printf("ERROR #%d while reading IPI buffer\n", Status);
        return;
    }

    if ((XSEM_EVENT_ERROR == Payload[0]) && (XSEM_NOTIFY_NPI == Payload[1]))
{
        if (XSEM_EVENT_NPI_CRC_ERR == Payload[2]) {
            NPI_CRC_EventCnt++;
        } else if (XSEM_EVENT_NPI_INT_ERR == Payload[2]) {
            NPI_INT_EventCnt++;
        } else {
            xil_printf("%s Some other callback received: %d:%d:%d\n",
                    __func__, Payload[0], Payload[1], Payload[2]);
        }
    } else {
        xil_printf("%s Some other callback received: %d\n", __func__,
Payload[0]);
    }
}
```

For IPI initialization (includes IPI configuration and connection with GIC, IPI callback registration with IPI handler), see the following code snippet:

```
XStatus IpiInit(XIpiPsu * InstancePtr, XScuGic * GicInst)
{
    int Status;

    Status = IpiConfigure(InstancePtr, GicInst);
    if (XST_SUCCESS != Status) {
        xil_printf("IpiConfigure() failed with error: %d\r\n",
                Status);
    }

    Status = IpiRegisterCallback(InstancePtr, SRC_IPI_MASK, \
            XSem_IpiCallback);
    return Status;
}
```

***Note*:** CRAM and NPI should run one at time.

For initializing GIC, XilSEM IPI instance and registering ISR handler to process XilSEM notifications from PLM, use the below code snippet:

```
XStatus XSem_IpiInitApi (void)
{
    XStatus Status = XST_FAILURE;

    /* GIC Initialize */
    Status = GicSetupInterruptSystem(&GicInst);
    if (Status != XST_SUCCESS) {
        xil_printf("GicSetupInterruptSystem failed with error: %d\r\n",\
                Status);
        goto END;
    }

    Status = IpiInit(&IpiInst, &GicInst);
    if (XST_SUCCESS != Status) {
        xil_printf("[%s] IPI Init Error: Status 0x%x\r\n", \
                __func__, Status);
        goto END;
    }

END:
    return Status;
}
```

For register error event notification with CRAM, see the following code snippet:

```
XSem_Notifier Notifier = {
        .Module = XSEM_NOTIFY_CRAM,
        .Event = XSEM_EVENT_CRAM_UNCOR_ECC_ERR | XSEM_EVENT_CRAM_CRC_ERR | \
         XSEM_EVENT_CRAM_INT_ERR | XSEM_EVENT_CRAM_COR_ECC_ERR,
    .Flag = 1U,
};
int Status;
Status = XSem_RegisterEvent(&IpiInst, &Notifier);
if (XST_SUCCESS == Status) {
    xil_printf("Success: Event registration \n\r");
} else {
    xil_printf("Error: Event registration failed \n\r");
    goto END;
}
```

For register error event notification with NPI, see the following code snippet:

```
XSem_Notifier Notifier = {
        .Module = XSEM_NOTIFY_NPI,
        .Event = XSEM_EVENT_NPI_CRC_ERR | XSEM_EVENT_NPI_INT_ERR,
    .Flag = 1U,
};

int Status;
Status = XSem_RegisterEvent(&IpiInst, &Notifier);
if (XST_SUCCESS == Status) {
    xil_printf("Success: Event registration \n\r");
} else {
    xil_printf("Error: Event registration failed \n\r");
    goto END;
}
```

To check global variables that holds the count of the notified event for CRAM, use the following code snippet:

```
if(EventCnt_UnCorEcc > 0){
    xil_printf("Uncorrectable error has been detected in CRAM\n\r");
}else if(EventCnt_Crc > 0){
    xil_printf("CRC error has been detected in CRAM\n\r");
} else if(EventCnt_CorEcc > 0){
    xil_printf("Correctable error has been detected and corrected in CRAM\n
\r");
} else if(EventCnt_IntErr > 0){
    xil_printf("Internal error has occurred in CRAM\n\r");
}
```

To check global variables that holds the count of the notified event for NPI, use the following code snippet:

```
if(NPI_CRC_EventCnt > 0){
    xil_printf("CRC error has been detected in NPI\n\r");
}else if(NPI_INT_EventCnt > 0){
    xil_printf("Internal error has occurred in NPI\n\r");
}
```

Note: The IPI notification happens only when you register IPI notifications in your application. It is recommended that before the applications sets in to mission mode, you shall read XilSEM status using the the XSem_CmdCfrGetStatus and XSem_CmdNpiGetStatus APIs.

From 2022.2 release onwards, XilSEM error notification support is extended for A72 Linux and bare-metal application users.

- A72 bare-metal application users can receive XilSEM error notifications through XilPM client. interface XPm_RegisterNotifier with event ID as XilSEM events (XIL_EVENT_ERROR_MASK_XSEM_CRAM_CE (0x00000020U), XIL_EVENT_ERROR_MASK_XSEM_CRAM_UE (0x00000040U), and XIL_EVENT_ERROR_MASK_XSEM_NPI_UE (0x00000080U)) and node ID as XIL_NODETYPE_EVENT_ERROR_SW_ERR (0x28110000U).
- A72 Linux users can receive XilSEM error notifications through XilSEM EDAC driver.

Note: From 2023.1 release onwards, Linux XilSEM EDAC driver supports sysfs interface to perform XilSEM scan operations initialize, start, stop scan, error inject, read ECC, and configuration values. For more information, see this page.

For more information, refer Event Management Framework in Versal Adaptive SoC System Software Developers Guide (UG1304).

An additional method exists for the user design to listen for SEU detections. Unlike the IP integrator method which is used for RPU user software, this method is used for PL user logic. It involves PMC_PL_GPO outputs of the PMC routed into the PL, supplying quick access to key event information. To access this feature, it must be enabled during XilSEM library configuration.

- Bit 0: CRAM correctable error

The GPO is set to High when the error is detected. The GPO is cleared when the error is corrected. If you have disabled correction in the CIPS configuration, the GPO is set and remains High until CRAM scan is re-initialized.

- Bit 1: CRAM/NPI uncorrectable error

  The GPO is set to High when the error is detected.

- Bit 2: CRAM/NPI/XilSEM internal error
- Bit 3: Reserved

The following diagram provides a XilSEM-centric view of information conduits to user software and user logic implemented in a Versal adaptive SoC.

Figure 10: **Flow of Information on a Versal adaptive SoC using a XilSEM Subsystem**

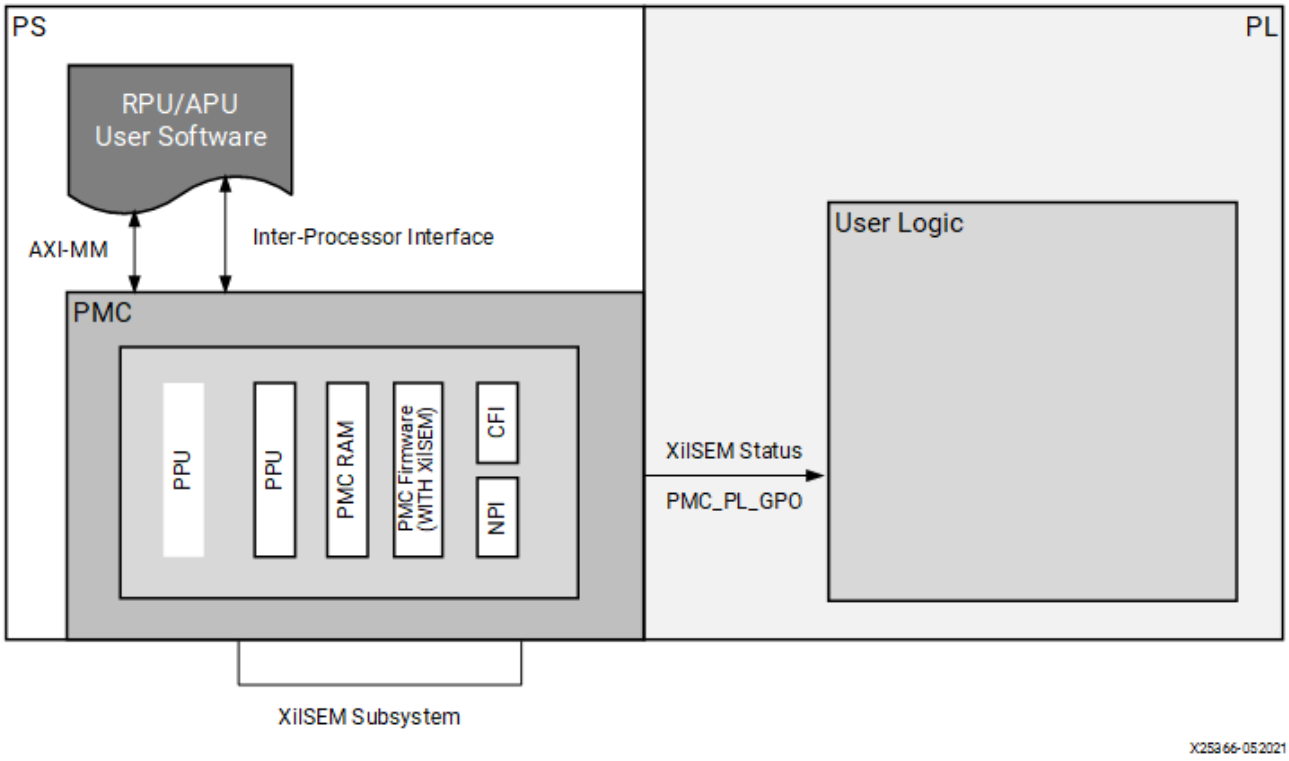


Figure 11: **Software Flow for CRAM Scan**

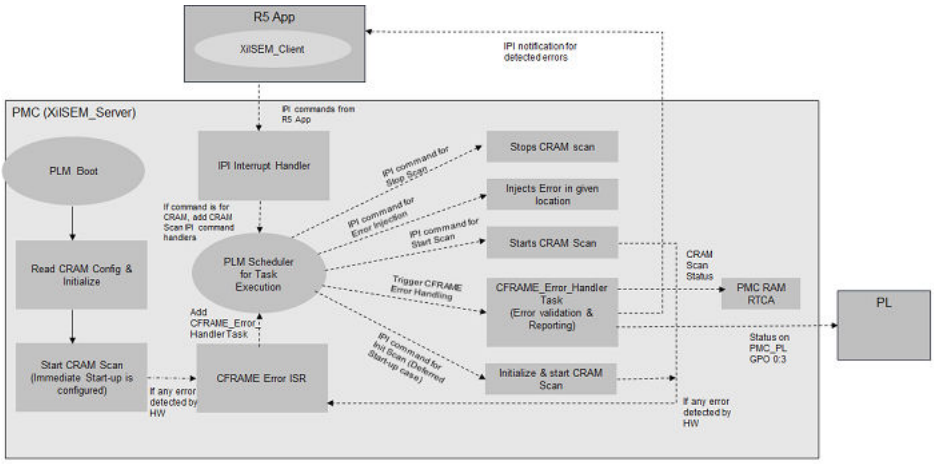
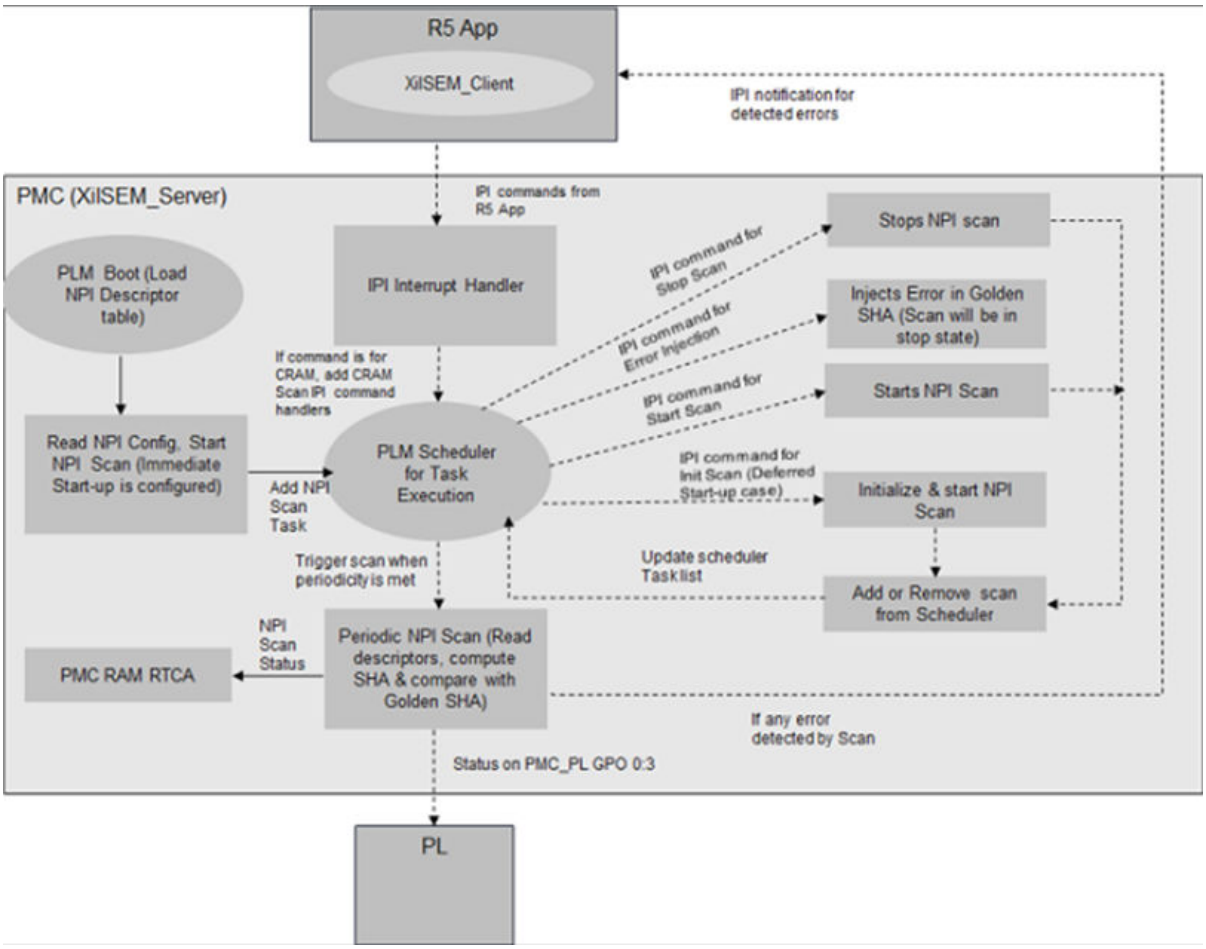


Figure 12: **Software Flow for NPI Scan**



Selection of system integration methods necessarily depends on the system requirements, particularly with regard to control and status of a XilSEM subsystem. The following table shows a variety of use cases that can be supported with the available integration methods.

Table 385: **Supported Use-cases with the Available Integration Method**

Use Cases	System Integration Methods	
	XilSEM Client on R5	PMC GPO to PL (Output)
Background operation without notification (requires use of immediate start options for Configuration RAM scan and NPI Register scan)		
Background with critical event notification to PL design (requires use of immediate start options for Configuration RAM scan and NPI Register scan)		Available
Interactive with event and detailed status notification by IPI	Available	
Interactive with event and detailed status notification by IPI with critical event notification to PL design	Available	Available

Injecting Errors for Test

Injecting an Error in Configuration RAM

Here are steps and code snippets illustrating how to inject an error in the configuration RAM for test purposes.

1. Stop Scan

```
/*To stop scan*/
XSemIpiResp IpiResp = {0};
XStatus Status = XST_FAILURE;
Status = XSem_CmdCfrStopScan(&IpiInst, &IpiResp);
if ((XST_SUCCESS == Status) && \
        (CMD_ACK_CFR_STOP_SCAN == IpiResp.RespMsg1) && \
        (XST_SUCCESS == IpiResp.RespMsg2)) {
    xil_printf("[%s] Success: Stop\n\r", __func__);
} else {
    xil_printf("[%s] Error: Stop Status 0x%x Ack 0x%x, Ret 0x%x", \
        __func__, Status, IpiResp.RespMsg1, IpiResp.RespMsg2);
    goto END;
}
```

2. Inject Error

   ***Note*:** The error injection shall be done with valid CFRAME address and valid row.

   - For valid range of addresses, refer to the LAST_FRAME_TOP (CFRAME_REG) and LAST_FRAME_BOT (CFRAME_REG) registers.
   - For valid row, refer to the CFU_ROW_RANGE (CFU_APB) register.

To inject uncorrectable error, ensure to inject in alternate bit positions in the same qword. To inject CRC error, ensure to inject in alternate bit positions of the same qword in 3 or more than 3 positions. Refer to the xsem_cram_example.c file in the examples directory. For register information, see *Versal ACAP Register Reference* (AM012).

The safe location to perform error injection is QWORD 12 which has ECC bits. The error injection will not change the design behavior.

When an uncorrectable/CRC error or correctable error (correction is disabled in the configuration) is detected by XilSEM, the CRAM scan on PLM will be stopped and the user will be notified over IPI. In such case, ensure to reprogram/ restart to recover the error.

```
/* Inject 1-bit correctable error */
XSemIpiResp IpiResp = {0};
XStatus Status = XST_FAILURE;
XSemCfrErrInjData ErrData = {0x7, 0x0, 0x0, 0x0};
Status = XSem_CmdCfrNjctErr(&IpiInst, &ErrData, &IpiResp);
if ((XST_SUCCESS == Status) && \
        (CMD_ACK_CFR_NJCT_ERR == IpiResp.RespMsg1) && \
        (XST_SUCCESS == IpiResp.RespMsg2)) {
    xil_printf("[%s] Success: Inject\n\r", __func__);
} else {
    xil_printf("[%s] Error: Inject Status 0x%x Ack 0x%x, Ret 0x%x",\
        __func__, Status, IpiResp.RespMsg1, IpiResp.RespMsg2);
    goto END;
}
```

3. Start scan to detect/correct injected error

```
/*To start scan*/
Status = XSem_CmdCfrStartScan(&IpiInst, &IpiResp);
if ((XST_SUCCESS == Status) && \
        (CMD_ACK_CFR_START_SCAN == IpiResp.RespMsg1) && \
        (XST_SUCCESS == IpiResp.RespMsg2)) {
    xil_printf("[%s] Success: Start\n\r", __func__);
} else {
    xil_printf("[%s] Error: Start Status 0x%x Ack 0x%x, Ret 0x%x", \
        __func__, Status, IpiResp.RespMsg1, IpiResp.RespMsg2);
    goto END;
}
```

Wait for the XilSEM library to detect and optionally correct the error (if configured to do so) and report the error. The following code illustrates how to validate if the error injection was successful.

```
/*To validate injection*/
while (XST_SUCCESS != XSem_ChkCfrStatus(0x1, 0x800U, 11U)) {
    usleep(100);
}
xil_printf("Cram Status [0x%08x]\n\r", CfrStatusInfo.Status);
xil_printf("Cor Ecc Cnt [0x%08x]\n\r", CfrStatusInfo.ErrCorCnt);
xil_printf("Error address details are as\n\r");
for (Index = 0U ; Index < MAX_CRAMERR_REGISTER_CNT; Index++) {
    xil_printf("\nErrAddrL%d [0x%08x]\n\r", \
                Index, CfrStatusInfo.ErrAddrL[Index]);
    xil_printf("ErrAddrH%d [0x%08x]\n\n\r", \
                Index, CfrStatusInfo.ErrAddrH[Index]);
}
```

```
/*Function to check CFR status*/
static XStatus XSem_ChkCfrStatus(u32 ExpectedStatus, u32 Mask, u32 Shift)
{
XSemCfrStatus CfrStatusInfo = {0};
XStatus Status = XST_FAILURE;
u32 Temp = 0U;
Status = XSem_CmdCfrGetStatus(&CfrStatusInfo);
if (XST_SUCCESS == Status) {
    Temp = DataMaskShift(CfrStatusInfo.Status, Mask, Shift);
    if (Tesmp != ExpectedStatus) {
        Status = XST_FAILURE;
    }
}
return Status;
}
```

Injecting an Error in NPI Registers

Here are steps and code snippets illustrating how to inject an error in NPI registers for test purposes.

1. Stop scan

```
/*To stop scan*/
Status = XSem_CmdNpiStopScan(&IpiInst, &IpiResp);
if ((XST_SUCCESS == Status) &&
        (CMD_ACK_NPI_STOPSCAN == IpiResp.RespMsg1) &&
        (XST_SUCCESS == IpiResp.RespMsg2)) {
    xil_printf("[%s] Success: Stop\n\r", __func__);
} else {
    xil_printf("[%s] Error: Stop Status 0x%x Ack 0x%x, Ret 0x%x\n\r", \
            __func__, Status, IpiResp.RespMsg1, IpiResp.RespMsg2);
    goto END;
}
```

2. Inject error

```
/* Inject error in first used SHA */
Status = XSem_CmdNpiInjectError(&IpiInst, &IpiResp);
if ((XST_SUCCESS == Status) &&
        (CMD_ACK_NPI_ERRINJECT == IpiResp.RespMsg1) &&
        (XST_SUCCESS == IpiResp.RespMsg2)) {
    xil_printf("[%s] Success: Inject\n\r", __func__);
} else {
    xil_printf("[%s] Error: Inject Status 0x%x Ack 0x%x, Ret 0x%x\n\r", \
            __func__, Status, IpiResp.RespMsg1, IpiResp.RespMsg2);
    goto END;
}
```

3. Start scan to detect injected error

```
/*To start scan*/
Status = XSem_CmdNpiStartScan(&IpiInst, &IpiResp);
if ((XST_SUCCESS == Status) &&
        (CMD_ACK_NPI_STARTSCAN == IpiResp.RespMsg1) &&
        (XST_SUCCESS == IpiResp.RespMsg2)) {
    xil_printf("[%s] Success: Start\n\r", __func__);
```

```
} else {
    xil_printf("[%s] Error: Start Status 0x%x Ack 0x%x, Ret 0x%x\n\r", \
               __func__, Status, IpiResp.RespMsg1, IpiResp.RespMsg2);
    goto END;
}
```

Wait for the XilSEM library to detect and report the error. The following code illustrates how to validate if the error injection was successful.

```
/*To validate injection*/
TimeoutCount = 4U;
while (TimeoutCount != 0U) {
    Status = XSem_CmdNpiGetStatus(&NpiStatus);
    if (XST_SUCCESS != Status) {
        xil_printf("[%s] ERROR: NPI Status read failure\n\r", \
                   __func__, Status);
        goto END;
    }
    /* Read NPI_SCAN_ERROR status bit */
    TempA_32 = ((NpiStatus.Status & 0x00020000U) >> 17U);
    if (TempA_32 == 1U) {
        goto END;
    }
    TimeoutCount--;
    /* Small delay before polling again */
    usleep(25000);
}
xil_printf("[%s] ERROR: Timeout occurred waiting for error\n\r",
__func__);
Status = XST_FAILURE;
END:
    return Status;
```

Note:

- All the macros used in the code snippets are defined in xsem_client_api.h
- The above code snippets are demonstrated through examples in the XilSEM library using xsem_cram_example.c and xsem_npi_example.c. You can import these examples into the Vitis IDE to get more implementation details.
- When an error is injected in NPI golden SHA value, this condition is treated as uncorrectable error. XilSEM on PLM stops the scan.

XilSEM Operation During Partial Bitstream Loading

When you make a request for partial bitstream loading, XilSEM performs the following actions.

1. Checks if any CFRAME error is present
2. Corrects the error (provided correction is enabled in CIPS Customize IP window)
3. Notifies the error if the correction is not enabled in CIPS IP in Vivado Design Suite GUI
4. Suspends both CRAM scan and NPI scan
5. Reinitializes both CRAM and NPI scan operations, once the bitstream loading is complete.

Specifications

Standards

No standards compliance or certification testing is defined. The XilSEM library, running on Versal adaptive SoC, is exposed to a beam of accelerated particles as part of an extensive hardware validation process.

Performance

Performance metrics for the XilSEM library are derived from silicon specifications and direct measurement and are for budgetary purposes only. Actual performance might vary.

Table 386: **Performance Metrics for the Configuration RAM**

Device and Conditions	Initialization	Complete Scan Time	Correctable ECC Error Handling	Uncorrectable CRC Error Handling	Other Uncorrectable Error Handling
XCVC1902 PMC = 320 MHz CFU = 400 MHz	SW ECC: 18.1 ms HW ECC: 36.2 ms	13.6 ms (with 149682 CRAM frames)	55 µs	16 µs	49 µs
XCVM1802 PMC = 320 MHz CFU = 400 MHz	SW ECC: 18.1 ms HW ECC: 36.2 ms	13.6 ms (with 149682 CRAM frames)	55 µs	16 µs	49 µs

Table 387: **Performance Metrics for NPI Registers**

Device and Conditions	Initialization	Complete Scan Time	Uncorrectable SHA Error Handling	Other Uncorrectable Error Handling
XCVC1902 PMC = 320 MHz CFU = 400 MHz	SW SHA: 13.2 ms HW SHA: 13.2 ms	13.3 ms (with 900+ NPI slaves)	12.47 ms	342.8 s (DMA to SHA transfer timeout)
XCVM1802 PMC = 320 MHz CFU = 420 MHz	SW SHA: 14.7 ms HW SHA: 14.7 ms	14.7 ms (with 900+ NPI slaves)	13.7 ms	353.5 s (DMA to SHA transfer timeout)

Table 388: **Performance Metrics for the CRAM scan with different clock frequencies (XCVC1902)**

Device and Conditions (XCVC1902)	CRAM Scan Frequency	Initialization	Complete Scan Time (Total Frames: 149682)	Correctable ECC Error Handling	Uncorrectable CRC Error Handling	Other Uncorrectable Error Handling
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 3	399.96 MHz	HWECC: 27.044 ms SWECC: 13.619 ms	13.84 ms	59 µs	16 µs	44 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 4	299.96 MHz	HWECC: 36.024 ms SWECC: 18.145 ms	18.46 ms	63 µs	16 µs	48 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 5	239.96 MHz	HWECC: 45.022 ms SWECC: 22.591 ms	23.08 ms	68 µs	17 µs	50 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 6	199.98 MHz	HWECC: 53.672 ms SWECC: 22.087 ms	27.69 ms	70 µs	18 µs	53 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 7	171.41 MHz	HWECC: 63.078 ms SWECC: 31.845 ms	32.3 ms	75 µs	19 µs	57 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 8	149.98 MHz	HWECC: 72.117 ms SWECC: 36.378 ms	36.92 ms	79 µs	19 µs	59 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 9	133.32 MHz	HW ECC: 81.276 ms SWECC: 40.918 ms	41.54 ms	83 µs	20 µs	63 µs

Table 388: **Performance Metrics for the CRAM scan with different clock frequencies (XCVC1902)** *(cont'd)*

Device and Conditions (XCVC1902)	CRAM Scan Frequency	Initialization	Complete Scan Time (Total Frames: 149682)	Correctable ECC Error Handling	Uncorrectable CRC Error Handling	Other Uncorrectable Error Handling
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 10	119.98 MHz	HWECC: 90.203 ms SWECC: 45.473 ms	46.15 ms	87 µs	21 µs	66 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 11	109.08 MHz	HWECC: 99.289 ms SWECC: 50.033 ms	50.77 ms	89 µs	22 µs	68 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 12	99.99 MHz	HWECC: 108.278 ms SWECC: 54.536 ms	55.38 ms	95 µs	22 µs	72 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 13	92.29 MHz	HWECC: 117.260 ms SWECC: 59.059 ms	60.00 ms	97 µs	23 µs	73 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 14	85.7 MHz	HWECC: 125.280 ms SWECC: 63.617 ms	64.61 ms	101 µs	24 µs	78 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 15	79.99 MHz	HWECC: 135.362 ms SWECC: 68.207 ms	69.23 ms	108 µs	25 µs	81 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 16	74.99 MHz	HWECC: 143.257 ms SWECC: 72.739 ms	73.85 ms	109 µs	25 µs	81 µs

Table 388: **Performance Metrics for the CRAM scan with different clock frequencies (XCVC1902)** *(cont'd)*

Device and Conditions (XCVC1902)	CRAM Scan Frequency	Initialization	Complete Scan Time (Total Frames: 149682)	Correctable ECC Error Handling	Uncorrectable CRC Error Handling	Other Uncorrectable Error Handling
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 17	70.58 MHz	HWECC: 153.441 ms SWECC: 77.297 ms	78.46 ms	112 µs	23 µs	85 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 18	66.66 MHz	HWECC: 162.451 ms SWECC: 81.809 ms	83.08 ms	118 µs	27 µs	89 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 19	63.15 MHz	HWECC: 169.972 ms SWECC: 86.382 ms	87.69 ms	122 µs	28 µs	93 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 20	59.99 MHz	HWECC: 178.952 ms SWECC: 90.844 ms	92.31 ms	127 µs	28 µs	96 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 21	57.13 MHz	HWECC: 189.504 ms SWECC: 95.403 ms	96.92 ms	129 µs	30 µs	98 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 25	47.99 MHz	HWECC: 226.884 ms SWECC: 113.603 ms	115.39 ms	147 µs	33 µs	111 µs
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 30	39.99 MHz	HWECC: 268.404 ms SWECC: 136.369 ms	138.469 ms	170 µs	37 µs	129 µs

Table 388: **Performance Metrics for the CRAM scan with different clock frequencies (XCVC1902)** *(cont'd)*

Device and Conditions (XCVC1902)	CRAM Scan Frequency	Initialization	Complete Scan Time (Total Frames: 149682)	Correctable ECC Error Handling	Uncorrectable CRC Error Handling	Other Uncorrectable Error Handling
PMC = 320 MHz FBDIV = 72 Input_clock = 33.33 MHz CLKOUTDIV = 2 DIVISOR0 = 32	37.49 MHz	HWECC: 286.474 ms SWECC: 145.374 ms	147.7 ms	172 µs	38 µs	130 µs

Error detection latency is the major component of the total error mitigation latency. Error detection latency is a function of the device size and the underlying clock signals driving the processes involved, as these determine the Complete Scan time. It is also a function of the type of error and the relative position of the error with respect to the position of the scan process, at the time the error occurs. The error detection latency can be bounded as follows:

- Maximum error detection latency for detection by ECC is one Complete Scan Time
  - This represents a highly unlikely case when an error at a given location occurs directly “behind” the scan process.
  - It will take one Complete Scan Time for the scan process to return to the error location at which time it will detect it.
- Maximum error detection latency for detection by CRC or SHA is 2.0 × Complete Scan Time
  - This represents an extremely unlikely case when an error occurs directly “behind” the scan process and is located at the scan start location (where the checksum accumulation begins at each scan).
  - It will take one Complete Scan Time for the scan process to complete the current checksum accumulation (which will pass) and then a second Complete Scan Time to complete a checksum accumulation which includes the error (which will fail).

When a CFRAME error is detected, a task is added to PLM scheduler for error validation, correction and notification. The time for correction and notification depends on other requests which PLM has already been processing. The time also depends on the secure data size and secure operation. The following table lists the timing for different secure operations and power management tasks.

Table 389: **PLM requests processing time**

Example PLM Requests/Tasks[1]	Processing Time (Approximate)
Optional NPI Scan (varies by resource utilization)	15 ms
Secure Data Request Authentication (RSA) Xsecure_RsaPublicEncrypt_64bit	1.87 ms
Secure Data Request Authentication (RSA) Xsecure_RsaPrivateDecrypt_64bit	91.13 ms
Secure Data Request Authentication (ECDSA-P384) Xsecure_EllipticVerifySign_64bit	6.31 ms
Secure Data Request Authentication (ECDSA-P521) Xsecure_EllipticVerifySign_64bit	14.56 ms
Secure Data Request, SHA, 100 Kb (varies by size)	0.17 ms
Secure Data Request, AES, 100 Kb (varies by size)	0.77 ms
Power Management Requests (estimated)	1 ms

Reliability

As part of the extensive hardware validation process for the XilSEM library, reliability metrics are experimentally obtained and summarized in the table below.

Throughput

The throughput metrics of the XilSEM library are not specified.

Power

The power metrics of the XilSEM library are not specified. However, the Xilinx Power Estimator (XPE) can be used to estimate XilSEM library power, which is mostly contributed by the configuration RAM scan process operated on the CFU clock domain. The CFU clock default frequency set by CIPS is high, to maximize bandwidth of the configuration RAM. Depending on design requirements, it may be desirable to operate at lower CFU clock frequencies. CIPS currently supports user access to edit many clock generator settings, including the CFU clock. You can take advantage of this compile time tradeoff by lowering the CFU clock setting. This change, unless later overridden during design operation, will impact any CFU activity and that has broader implications than the XilSEM library. For example, other activities which use the CFU include (but are not limited to):

- PL Housekeeping at Boot
- Readback Capture and Verify

[1] DFX activity requires XilSEM scan to be stopped (same as Legacy SEM IP core) to avoid scan interference with each other. Clock management request estimates are pending, time is assumed short.

- DFX / Partial Reconfig Using this compile time tradeoff is simple and can work well, especially with smaller devices where the bandwidth to size ratio of the configuration RAM might have ample margin versus design requirements to support trading it for power reduction. However, it is critically important to evaluate the broader impact of this tradeoff beyond XilSEM library power.

XilSEM Versal adaptive SoC Client APIs

This file provides XilSEM client interface to send IPI requests from the user application to XilSEM server on PLM. This provides APIs to Init, Start, Stop, Error Injection, Event notification registration, get Status details for both CRAM and NPI.

***Note*:**

Table 390: **Quick Function Reference**

Type	Member	Arguments
XStatus	XSem_RegisterEvent	XIpiPsu * IpiInst `XSem_Notifier` * Notifier
XStatus	XSem_CmdCfrInit	XIpiPsu * IpiInst `XSemIpiResp` * Resp
XStatus	XSem_CmdCfrStartScan	XIpiPsu * IpiInst `XSemIpiResp` * Resp
XStatus	XSem_CmdCfrStopScan	XIpiPsu * IpiInst `XSemIpiResp` * Resp
XStatus	XSem_CmdCfrNjctErr	XIpiPsu * IpiInst \ `XSemCfrErrInjData` * ErrDetail \ `XSemIpiResp` * Resp
XStatus	XSem_CmdCfrGetStatus	`XSemCfrStatus` * CfrStatusInfo
XStatus	XSem_CmdCfrReadFrameEcc	XIpiPsu * IpiInst u32 CframeAddr u32 RowLoc `XSemIpiResp` * Resp
XStatus	XSem_CmdNpiStartScan	XIpiPsu * IpiInst `XSemIpiResp` * Resp

Table 390: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
XStatus	XSem_CmdNpiStopScan	XIpiPsu * IpiInst XSemIpiResp * Resp
XStatus	XSem_CmdNpiInjectError	XIpiPsu * IpiInst XSemIpiResp * Resp
XStatus	XSem_CmdNpiGetGldnSha	XIpiPsu * IpiInst XSemIpiResp * Resp XSem_DescriptorData * DescData
XStatus	XSem_CmdNpiGetStatus	XSemNpiStatus * NpiStatusInfo
XStatus	XSem_CmdGetConfig	XIpiPsu * IpiInst XSemIpiResp * Resp
u32	XSem_CmdCfrGetCrc	u32 RowIndex
void	XSem_CmdCfrGetTotalFrames	u32 RowIndex u32 * FrameCntPtr
XStatus	XSem_Ssit_CmdCfrInit	XIpiPsu * IpiInst XSemIpiResp * Resp u32 TargetSlr
XStatus	XSem_Ssit_CmdCfrStartScan	XIpiPsu * IpiInst XSemIpiResp * Resp u32 TargetSlr
XStatus	XSem_Ssit_CmdCfrStopScan	XIpiPsu * IpiInst XSemIpiResp * Resp u32 TargetSlr
XStatus	XSem_Ssit_CmdCfrNjctErr	XIpiPsu * IpiInst \ XSemCfrErrInjData * ErrDetail \ XSemIpiResp * Resp u32 TargetSlr

Table 390: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
XStatus	XSem_Ssit_CmdCfrReadFrameEcc	XIpiPsu * IpiInst u32 CframeAddr u32 RowLoc XSemIpiResp * Resp u32 TargetSlr
XStatus	XSem_Ssit_CmdGetStatus	XIpiPsu * IpiInst XSemIpiResp * Resp u32 TargetSlr XSemStatus * StatusInfo
XStatus	XSem_Ssit_CmdNpiStartScan	XIpiPsu * IpiInst XSemIpiResp * Resp u32 TargetSlr
XStatus	XSem_Ssit_CmdNpiStopScan	XIpiPsu * IpiInst XSemIpiResp * Resp u32 TargetSlr
XStatus	XSem_Ssit_CmdNpiInjectError	XIpiPsu * IpiInst XSemIpiResp * Resp u32 TargetSlr
XStatus	XSem_Ssit_CmdGetConfig	XIpiPsu * IpiInst XSemIpiResp * Resp u32 TargetSlr
XStatus	XSem_Ssit_CmdCfrGetCrc	XIpiPsu * IpiInst u32 RowIndex XSemIpiResp * Resp u32 TargetSlr

Functions

XSem_RegisterEvent

This function is used to register/un-register event notification with XilSEM Server. Primarily this function sends an IPI request to PLM to invoke SEM Event Notifier registration, waits for PLM to process the request and check the status. Since SLR slave devices do not support IPI, registering events to mater registers events on all slave SLRs.

***Note*:** The caller shall initialize the notifier object before invoking the XSem_RegisterEvent function.

Prototype

```
XStatus XSem_RegisterEvent(XIpiPsu *IpiInst, XSem_Notifier *Notifier);
```

Parameters

The following table lists the `XSem_RegisterEvent` function arguments.

Table 391: **XSem_RegisterEvent Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
`XSem_Notifier` *	Notifier	Pointer of the notifier object to be associated with the requested notification • Notifier->Module: The SEM module from which notification is required • Notifier->Event: Event(s) belonging to the Module for which notifications are required • Notifier->Flag: Flags to enable or disable notifications

Returns

This API returns the success or failure.

- XST_FAILURE: On event registration/un-registration failure
- XST_SUCCESS: On event registration/un-registration success

XSem_CmdCfrInit

This function is used to initialize CRAM scan from user application. Primarily this function sends an IPI request to PLM to start CRAM Scan Initialization, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_CmdCfrInit(XIpiPsu *IpiInst, XSemIpiResp *Resp);
```

Parameters

The following table lists the `XSem_CmdCfrInit` function arguments.

Table 392: **XSem_CmdCfrInit Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance

Table 392: **XSem_CmdCfrInit Arguments** *(cont'd)*

Type	Member	Description
XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of CRAM Initialization(0x10301) • Resp->RespMsg2: Status of CRAM Initialization 0x01000000U - ECC/CRC error detected during calibration in case of SWECC 0X00000080U - Calibration timeout 0X00002000U - Internal error

Returns

This API returns the success or failure.

- XST_FAILURE: On CRAM Initialization failure
- XST_SUCCESS: On CRAM Initialization success

XSem_CmdCfrStartScan

This function is used to start CRAM scan from user application. Primarily this function sends an IPI request to PLM to invoke SEM CRAM StartScan, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_CmdCfrStartScan(XIpiPsu *IpiInst, XSemIpiResp *Resp);
```

Parameters

The following table lists the `XSem_CmdCfrStartScan` function arguments.

Table 393: **XSem_CmdCfrStartScan Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of CRAM start scan(0x10302) • Resp->RespMsg2: Status of CRAM start scan 0x2000 – Null pointer error 0x00F00000 – Active crc/uncor error 0x00500000 – CRAM init not done 0x00600000 – Start scan failed

Returns

This API returns the success or failure.

- XST_FAILURE: On CRAM start scan failure
- XST_SUCCESS: On CRAM start scan success

XSem_CmdCfrStopScan

This function is used to stop CRAM scan from user application. Primarily this function sends an IPI request to PLM to invoke SEM CRAM StopScan, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_CmdCfrStopScan(XIpiPsu *IpiInst, XSemIpiResp *Resp);
```

Parameters

The following table lists the `XSem_CmdCfrStopScan` function arguments.

Table 394: **XSem_CmdCfrStopScan Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of CRAM stop scan(0x10303) • Resp->RespMsg2: Status of CRAM stop scan 0x00500000 – CRAM init not done 0x00700000 – Stop scan failed

Returns

This API returns the success or failure.

- XST_FAILURE: On CRAM stop scan failure
- XST_SUCCESS: On CRAM stop scan success

XSem_CmdCfrNjctErr

This function is used to inject an error at a valid location in CRAM from user application. Primarily this function sends an IPI request to PLM to perform error injection in CRAM with user provided arguments in *ErrDetail, waits for PLM to process the request and reads the response message.

Note:

- Total number of frames in a row is not same for all rows.

- XSem_CmdCfrGetTotalFrames API is provided to know the total number of frames in a row for each block. Output param (FrameCntPtr) of XSem_CmdCfrGetTotalFrames API is updated with total number of frames of each block type for the input row. If a particular block in a row has 0 frames, then error injection shall not be performed. Range of Frame number: 0 to (FrameCntPtr[n] - 1) where n is block type with range 0 to 6. ECC bits. The error injection will not change the design behaviour.

Prototype

```
XStatus XSem_CmdCfrNjctErr(XIpiPsu *IpiInst, \ XSemCfrErrInjData *ErrDetail,
\ XSemIpiResp *Resp);
```

Parameters

The following table lists the `XSem_CmdCfrNjctErr` function arguments.

Table 395: **XSem_CmdCfrNjctErr Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
\ XSemCfrErrInjData *	ErrDetail	Structure Pointer with Error Injection details • ErrDetail->Row : Row Number (Min: 0 , Max: (value at CFU_ROW_RANGE)-1) • ErrDetail->Efar : Frame Address ◦ Frame Number [0:19] (Refer note) ◦ Block Type [20:22] • ErrDetail->Qword : Quad Word(Min: 0, Max: 24) • ErrDetail->Bit : Bit Position(Min: 0, Max: 127)
\ XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of CRAM error injection(0x10304) • Resp->RespMsg2: Status of CRAM error injection 0x00002000 – Null pointer error 0x00500000 – CRAM init not done 0x00800000 – Invalid row 0x00900000 – Invalid qword 0x00A00000 – Invalid bit 0x00B00000 – Invalid frame address 0x00C00000 – Unexpected bits flipped 0x00D00000 – Masked bit 0x00E00000 – Invalid block type 0x00F00000 – Active crc/uncor ecc error in CRAM

Returns

This API returns the success or failure.

- XST_FAILURE: On CRAM error injection failure
- XST_SUCCESS: On CRAM error injection success

XSem_CmdCfrGetStatus

This function is used to read all CRAM Status registers from PMC RAM and send to user application.

Prototype

```
XStatus XSem_CmdCfrGetStatus(XSemCfrStatus *CfrStatusInfo);
```

Parameters

The following table lists the `XSem_CmdCfrGetStatus` function arguments.

Table 396: **XSem_CmdCfrGetStatus Arguments**

Type	Member	Description
XSemCfrStatus *	CfrStatusInfo	Structure Pointer with CRAM Status details • CfrStatusInfo->Status: Provides details about CRAM scan ◦ Bit [31-25]: Reserved ◦ Bit [24:20]: CRAM Error codes - 00001: Unexpected CRC error when CRAM is not in observation state - 00010: Unexpected ECC error when CRAM is not in Observation or Initialization state - 00011: Safety write error in SEU handler - 00100: ECC/CRC ISR not found in any Row - 00101: CRAM Initialization is not done - 00110: CRAM Start Scan failure - 00111: CRAM Stop Scan failure - 01000: Invalid Row for Error Injection - 01001: Invalid QWord for Error Injection - 01010: Invalid Bit for Error Injection - 01011: Invalid Frame Address for Error Injection - 01100: Unexpected Bit flip during Error Injection - 01101: Masked Bit during Injection - 01110: Invalid Block Type for Error Injection - 01111: CRC or Uncorrectable Error or correctable error(when correction is disabled) is active in CRAM - 10000: ECC or CRC Error detected during CRAM Calibration in case of SWECC ◦ Bit [19-18]: Reserved ◦ Bit [17]: 0: CRAM scan is enabled in design 1: CRAM scan is disabled in design ◦ Bit [16]: 0: CRAM scan is not initialized 1: CRAM Initialization is completed ◦ Bit [15-14]: CRAM Correctable ECC error status - 00: No Correctable error encountered - 01: Correctable error is detected and corrected - 10: Correctable error is detected but not corrected (Correction is disabled) - 11: Reserved ◦ Bit [13]: 0: No error in CRAM scan 1: CRAM scan has internal error (Null pointer access/Safety write error) In this error condition, scan will be stopped and an event will be sent to R5. ◦ Bit [12]: 0: No error in error decoding 1: Invalid Error Location is reported In this error condition, scan will be stopped and an event will be sent to R5. ◦ Bit [11]: 0: No correctable error detected 1: Correctable ECC error detected In this condition, an event will be sent to R5. If correction is disabled, then scan will be stopped. Else, scan will continue to run.

Returns

This API returns the success or failure.

- XST_FAILURE: If NULL pointer reference of CfrStatusInfo
- XST_SUCCESS: On successful read from PMC RAM

XSem_CmdCfrReadFrameEcc

This function is used to Read frame ECC of a particular Frame. Primarily this function sends an IPI request to PLM to invoke SEM CRAM SendFrameEcc, waits for PLM to process the request and reads the response message.

Note:

- Total number of frames in a row is not same for all rows.
- XSem_CmdCfrGetTotalFrames API is provided to know the total number of frames in a row for each block. Output param (FrameCntPtr) of XSem_CmdCfrGetTotalFrames API is updated with total number of frames of each block type for the input row. If a particular block in a row has 0 frames, then error injection shall not be performed. Range of Frame number: 0 to (FrameCntPtr[n] - 1) where n is block type with range 0 to 6. ECC bits. The error injection will not change the design behaviour.

Prototype

```
XStatus XSem_CmdCfrReadFrameEcc(XIpiPsu *IpiInst, u32 CframeAddr, u32
RowLoc, XSemIpiResp *Resp);
```

Parameters

The following table lists the `XSem_CmdCfrReadFrameEcc` function arguments.

Table 397: **XSem_CmdCfrReadFrameEcc Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
u32	CframeAddr	Frame Address • Frame Number [0:19] (Refer note) • Block Type [20:22]
u32	RowLoc	Row index(Min: 0 , Max: CFU_ROW_RANGE -1)

Table 397: **XSem_CmdCfrReadFrameEcc Arguments** *(cont'd)*

Type	Member	Description
XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of CRAM Send Frame ECC(0x3030A) • Resp->RespMsg2: Segment 0 ECC value • Resp->RespMsg3: Segment 1 ECC value • Resp->RespMsg4: Status of CRAM stop scan

Returns

This API returns the success or failure.

- XST_FAILURE: On CRAM Read Frame ECC failure
- XST_SUCCESS: On CRAM Read Frame ECC success

XSem_CmdNpiStartScan

This function is used to start NPI scan from user application. Primarily this function sends an IPI request to PLM to invoke SEM NPI StartScan, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_CmdNpiStartScan(XIpiPsu *IpiInst, XSemIpiResp *Resp);
```

Parameters

The following table lists the `XSem_CmdNpiStartScan` function arguments.

Table 398: **XSem_CmdNpiStartScan Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of NPI start scan(0x10305) • Resp->RespMsg2: Status of NPI start scan (0: Success, 1: Failure)

Returns

This API returns the success or failure.

- XST_FAILURE: On NPI start scan failure

- XST_SUCCESS: On NPI start scan success

XSem_CmdNpiStopScan

This function is used to stop NPI scan from user application. Primarily this function sends an IPI request to PLM to invoke SEM NPI StopScan, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_CmdNpiStopScan(XIpiPsu *IpiInst, XSemIpiResp *Resp);
```

Parameters

The following table lists the `XSem_CmdNpiStopScan` function arguments.

Table 399: **XSem_CmdNpiStopScan Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of NPI stop scan(0x10306) • Resp->RespMsg2: Status of NPI stop scan (0: Success, 1: Failure)

Returns

This API returns the success or failure.

- XST_FAILURE: On NPI stop scan failure
- XST_SUCCESS: On NPI stop scan success

XSem_CmdNpiInjectError

This function is used to inject SHA error in NPI descriptor list (in the first NPI descriptor) from user application. Primarily this function sends an IPI request to PLM to invoke SEM NPI ErrorInject, waits for PLM to process the request and reads the response message.

***Note*:** The caller shall invoke this XSem_CmdNpiInjectError function again to correct the injected error in NPI descriptor.

Prototype

```
XStatus XSem_CmdNpiInjectError(XIpiPsu *IpiInst, XSemIpiResp *Resp);
```

Parameters

The following table lists the `XSem_CmdNpiInjectError` function arguments.

Table 400: **XSem_CmdNpiInjectError Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of NPI error injection(0x10307) • Resp->RespMsg2: Status of NPI error injection (0: Success, 1: Failure)

Returns

This API returns the success or failure.

- XST_FAILURE: On NPI error injection failure
- XST_SUCCESS: On NPI error injection success

XSem_CmdNpiGetGldnSha

This function is used to get golden SHA.

Prototype

```
XStatus XSem_CmdNpiGetGldnSha(XIpiPsu *IpiInst, XSemIpiResp *Resp,
XSem_DescriptorData *DescData);
```

Parameters

The following table lists the `XSem_CmdNpiGetGldnSha` function arguments.

Table 401: **XSem_CmdNpiGetGldnSha Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of NPI get golden SHA(0x10310) • Resp->RespMsg2: Status of NPI get golden SHA
XSem_DescriptorData *	DescData	Structure pointer to hold total descriptor count, golden SHA and information related to descriptors

Returns

This API returns the success or failure.

- XST_FAILURE: On NPI golden SHA retrieve failure
- XST_SUCCESS: On NPI golden SHA retrieve success

XSem_CmdNpiGetStatus

This function is used to read all NPI Status registers from PMC RAM and send to user application.

Prototype

```
XStatus XSem_CmdNpiGetStatus(XSemNpiStatus *NpiStatusInfo);
```

Parameters

The following table lists the `XSem_CmdNpiGetStatus` function arguments.

Table 402: **XSem_CmdNpiGetStatus Arguments**

Type	Member	Description
XSemNpiStatus *	NpiStatusInfo	Structure Pointer with NPI Status details • NpiStatusInfo->Status: Provides details about NPI scan ◦ Bit [31]: 0 - No error, SHA-3 engine is present 1 - Cryptographic acceleration blocks are disabled for export compliance. No support for NPI scan, an event will be sent to R5 ◦ Bit [30]: 0 - NPI scan is running for scheduled interval 1 - Indicates NPI scan failed to run on scheduled interval. If NPI scan is not executed as per the configured periodicity, the error will notified to R5 user. The scan will continue to run ◦ Bit [29]: 0- No descriptor missed during scanning 1- Indicates NPI scan failed to scan all descriptors completely (excluding arbitration failures). This will be notified to R5 user and the scan will continue to run. ◦ Bit [28]: 0 - NPI scan executing within budget time 1 - Indicates NPI scan has exceeded maximum budget execution time of 20ms. This will be notified to R5 user and the scan will continue to run. ◦ Bit [27]: 0 - No error in SLR to SLR communication 1 - Indicates failure in SSIT internal communication channel This bit is applicable for SSIT devices. ◦ Bit [26]: Reserved ◦ Bit [25]: 0 - No error in PMC_PL_GPO 1 - Indicates GPO Initialization or write failed. This is HW failure. In this condition, the scan will be stopped, and notification will be sent to R5 ◦ Bit [24]: 0 - No error SHA-3 engine 1 - Indicates SHA engine failed to function during initialization or start or DMA transfer. This is HW failure. In this condition, the scan will be stopped, and notification will be sent to R5 ◦ Bit [23]: 0 - No error in register writes 1 - Indicates the register write and read back failure occurred during the scan. This is HW failure. In this condition, the scan will be stopped, and notification will be sent to R5 ◦ Bit [22]: Reserved ◦ Bit [21]: 0 - No error in DDR calibration 1 - Indicates NPI DDRMC Main Slave Arbitration Timeout occurred during the scan. If the DDRMC calibration is not done, the descriptor will be skipped and scan will continue to run for next descriptor ◦ Bit [20]: 0 - No error in descriptor format 1 - Indicates NPI Descriptor has invalid format. This failure indicates that there is some corruption in the XilSEM NPI descriptor data. The scan will be stopped, and notification will be sent to R5 ◦ Bit [19]: 0 - No error in NPI Descriptor SHA header 1 - Indicates NPI Descriptor SHA Header mismatch occurred during the scan. This failure indicates that there is some corruption in the XilSEM NPI descriptor data. The scan will be stopped, and notification will be sent to R5 ◦ Bit [18]: 0 - NPI descriptors are present in the memory 1 - Indicates the absence of NPI Descriptor (Zero descriptors) This failure indicates that there is some corruption in the XilSEM NPI descriptor data. The scan will be stopped, and notification will be sent to R5 ◦ Bit [17]: 0 - No error in SHA comparison during run time 1- Indicates SHA comparison failure occurred during run time. This failure indicates that there is some bit flip in the NPI registers. The scan will be stopped and an event will be sent to R5 ◦ Bit [16]: 0 - No error in SHA comparison during first scan 1-

Returns

This API returns the success or failure.

- XST_FAILURE: If NULL pointer reference of NpiStatusInfo
- XST_SUCCESS: On successful read from PMC RAM

XSem_CmdGetConfig

This function is used to read CRAM & NPI configuration. Primarily this function sends an IPI request to PLM to invoke SEM Get configuration command, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_CmdGetConfig(XIpiPsu *IpiInst, XSemIpiResp *Resp);
```

Parameters

The following table lists the `XSem_CmdGetConfig` function arguments.

Table 403: **XSem_CmdGetConfig Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance

Table 403: **XSem_CmdGetConfig Arguments** *(cont'd)*

Type	Member	Description
XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of Get Configuration(0x30309) • Resp->RespMsg2: CRAM Attribute register details ◦ Bit [31:16]: Not Implemented ◦ Bit [15:9]: Reserved ◦ Bit [8]: Reserved ◦ Bit [7]: Reserved ◦ Bit [6:5]: Indicates when to start CRAM scan - 00: Do not automatically start scan - 01: Enable scan automatically after device configuration. - 10: Reserved - 11: Reserved ◦ Bit [4]: Reserved ◦ Bit [3]: Indicates HwECC/SwECC - 0: Uses hardware calculated ECC. - 1: Uses software calculated ECC that comes from tools and part of CDO ◦ Bit [2]: Indicates Correctable error is to be corrected/not - 0: Disables error correction capability - 1: Enables error correction capability ◦ Bit [1:0]: Define the mode of the scan (Enable/Disable scan) - 00: Disable Configuration RAM scan - 01: RESERVED - 10: Enable Configuration RAM scan - 11: RESERVED • Resp->RespMsg3: NPI Attribute register details ◦ Bit [31:24]: Not implemented ◦ Bit [23:18]: Reserved ◦ Bit [17:8]: The scheduled time in milliseconds that the NPI scan will be periodically performed. Default Setting: 0x064 = 100ms ◦ Bit [7:6]: Reserved ◦ Bit [5:4]: Indicates when to start NPI scan - 00: Do not automatically start scan - 01: Enable scan automatically after device configuration. - 10: Reserved - 11: Reserved ◦ Bit [3]: Reserved ◦ Bit [2]: Indicates HwSHA/SwSHA. 0: Use hardware calculated SHA. 1: Use software calculated SHA.

Returns

This API returns the success or failure.

- XST_FAILURE: On Get Configuration failure
- XST_SUCCESS: On Get Configuration success

XSem_CmdCfrGetCrc

This function is used to read CFRAME golden CRC for a row.

***Note*:**

- Total number of rows is not same for all platforms.
- The number maximum rows (CFU_ROW_RANGE) can be obtained by reading the address CFU_ROW_RANGE(0XF12B006C).

Prototype

```
u32 XSem_CmdCfrGetCrc(u32 RowIndex);
```

Parameters

The following table lists the `XSem_CmdCfrGetCrc` function arguments.

Table 404: **XSem_CmdCfrGetCrc Arguments**

Type	Member	Description
u32	RowIndex	Row index for which CRC to be read (Min: 0 , Max: CFU_ROW_RANGE -1)

Returns

This API returns the Golden CRC for a given Row.

XSem_CmdCfrGetTotalFrames

This function is used to read total frames in a row.

***Note*:**

- Total number of frames in a row is not same for all rows.
- XSem_CmdCfrGetTotalFrames API is provided to know the total number of frames in a row for each block. Output param (FrameCntPtr) of XSem_CmdCfrGetTotalFrames API is updated with total number of frames of each block type for the input row. If a particular block in a row has 0 frames, then error injection shall not be performed. Range of Frame number: 0 to (FrameCntPtr[n] - 1) where n is block type with range 0 to 6.

- The safe location to perform error injection is QWORD 12 which has ECC bits. The error injection will not change the design behaviour.

Prototype

```
void XSem_CmdCfrGetTotalFrames(u32 RowIndex, u32 *FrameCntPtr);
```

Parameters

The following table lists the `XSem_CmdCfrGetTotalFrames` function arguments.

Table 405: **XSem_CmdCfrGetTotalFrames Arguments**

Type	Member	Description
u32	RowIndex	Row index for which total number of frames is to be read (Min: 0 , Max: CFU_ROW_RANGE -1)
u32 *	FrameCntPtr	Pointer to store Total frames • FrameCntPtr[0] : Type_0 total frames • FrameCntPtr[1] : Type_1 total frames • FrameCntPtr[2] : Type_2 total frames • FrameCntPtr[3] : Type_3 total frames • FrameCntPtr[4] : Type_4 total frames • FrameCntPtr[5] : Type_5 total frames • FrameCntPtr[6] : Type_6 total frames

Returns

XSem_Ssit_CmdCfrInit

This function is used to initialize CRAM scan on targeted SLR from user application. Primarily this function sends an IPI request to PLM to start CRAM Scan Initialization, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_Ssit_CmdCfrInit(XIpiPsu *IpiInst, XSemIpiResp *Resp, u32
TargetSlr);
```

Parameters

The following table lists the `XSem_Ssit_CmdCfrInit` function arguments.

Table 406: **XSem_Ssit_CmdCfrInit Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance

XSemIpiResp *	Resp	Structure Pointer of IPI response
		• Resp->RespMsg1: Acknowledgment ID of CRAM Initialization(0x10301)
		• Resp->RespMsg2:
		  ◦ if Broadcast Status of Cfr Init in Master
		  ◦ else : Status of Cfr Init in Target SLR
		• RespMsg3, 4 and 5 are updated only in case of broadcast
		• Resp->RespMsg3: Status of CfrInit in Slave 1
		• Resp->RespMsg4: Status of CfrInit in Slave 2
		• Resp->RespMsg5: Status of CfrInit in Slave 3 Status of CRAM initialization: 0x01000000U - ECC/CRC error detected during calibration in case of SWECC 0X00000080U - Calibration timeout 0X00002000U - Internal error

u32	TargetSlr	Target SLR index on which command is to be executed • 0x0 : Target is master only • 0x1 : Target is slave 0 only • 0x2 : Target is slave 1 only • 0x3 : Target is slave 2 only • 0xF : Broadcast for all devices

Returns

This API returns the success or failure.

- XST_FAILURE: On CRAM Initialization failure
- XST_SUCCESS: On CRAM Initialization success

XSem_Ssit_CmdCfrStartScan

This function is used to start CRAM scan on targeted SLR from user application. Primarily this function sends an IPI request to PLM to invoke SEM CRAM StartScan, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_Ssit_CmdCfrStartScan(XIpiPsu *IpiInst, XSemIpiResp *Resp, u32
TargetSlr);
```

Parameters

The following table lists the `XSem_Ssit_CmdCfrStartScan` function arguments.

Table 407: **XSem_Ssit_CmdCfrStartScan Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance

XSemIpiResp *	Resp	Structure Pointer of IPI response
		• Resp->RespMsg1: Acknowledgment ID of CRAM start scan(0x10302)
		• Resp->RespMsg2:
		  ◦ if Broadcast : Status of CfrStartScan in Master
		  ◦ else : Status of CfrStartScan in Target SLR
		• RespMsg3, 4 and 5 are updated only in case of broadcast
		• Resp->RespMsg3: Status of CfrStartScan in Slave 1
		• Resp->RespMsg4: Status of CfrStartScan in Slave 2
		• Resp->RespMsg5: Status of CfrStartScan in Slave 3 Status of CRAM start scan: 0x2000 – Null pointer error 0x00F00000 – Active crc/uncor error 0x00500000 – CRAM init not done 0x00600000 – Start scan failed

u32	TargetSlr	Target SLR index on which command is to be executed • 0x0 : Target is master only • 0x1 : Target is slave 0 only • 0x2 : Target is slave 1 only • 0x3 : Target is slave 2 only • 0xF : Broadcast for all devices

Returns

This API returns the success or failure.

- XST_FAILURE: On CRAM start scan failure
- XST_SUCCESS: On CRAM start scan success

XSem_Ssit_CmdCfrStopScan

This function is used to stop CRAM scan on targeted SLR from user application. Primarily this function sends an IPI request to PLM to invoke SEM CRAM StopScan, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_Ssit_CmdCfrStopScan(XIpiPsu *IpiInst, XSemIpiResp *Resp, u32
TargetSlr);
```

Parameters

The following table lists the `XSem_Ssit_CmdCfrStopScan` function arguments.

Table 408: **XSem_Ssit_CmdCfrStopScan Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance

XSemIpiResp *	Resp	Structure Pointer of IPI response
		• Resp->RespMsg1: Acknowledgment ID of CRAM stop scan(0x10303)
		• Resp->RespMsg2:
		  ◦ if Broadcast : Status of CfrStoptScan in Master
		  ◦ else : Status of CfrStoptScan in Target SLR
		• RespMsg3, 4 and 5 are updated only in case of broadcast
		• Resp->RespMsg3: Status of CfrStoptScan in Slave 1
		• Resp->RespMsg4: Status of CfrStoptScan in Slave 2
		• Resp->RespMsg5: Status of CfrStoptScan in Slave 3 Status of CRAM stop scan: 0x00500000 – CRAM init not done 0x00700000 – Stop scan failed

u32	TargetSlr	Target SLR index on which command is to be executed • 0x0 : Target is master only • 0x1 : Target is slave 0 only • 0x2 : Target is slave 1 only • 0x3 : Target is slave 2 only • 0xF : Broadcast for all devices

Returns

This API returns the success or failure.

- XST_FAILURE: On CRAM stop scan failure
- XST_SUCCESS: On CRAM stop scan success

XSem_Ssit_CmdCfrNjctErr

This function is used to inject an error at a valid location in CRAM on targeted SLR from user application. Primarily this function sends an IPI request to PLM to perform error injection in CRAM with user provided arguments in *ErrDetail, waits for PLM to process the request and reads the response message.

Note:

- Total number of frames in a row is not same for all rows.
- XSem_CmdCfrGetTotalFrames API is provided to know the total number of frames in a row for each block. Output param (FrameCntPtr) of XSem_CmdCfrGetTotalFrames API is updated with total number of frames of each block type for the input row. If a particular block in a row has 0 frames, then error injection shall not be performed. Range of Frame number: 0 to (FrameCntPtr[n] - 1) where n is block type with range 0 to 6. ECC bits. The error injection will not change the design behaviour.

Prototype

```
XStatus XSem_Ssit_CmdCfrNjctErr(XIpiPsu *IpiInst, \ XSemCfrErrInjData
*ErrDetail, \ XSemIpiResp *Resp, u32 TargetSlr);
```

Parameters

The following table lists the `XSem_Ssit_CmdCfrNjctErr` function arguments.

Table 409: **XSem_Ssit_CmdCfrNjctErr Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
\ XSemCfrErrInjData *	ErrDetail	Structure Pointer with Error Injection details • ErrDetail->Row : Row Number (Min: 0 , Max: (value at CFU_ROW_RANGE)-1) • ErrDetail->Efar : Frame Address ◦ Frame Number [0:19] (Refer note) ◦ Block Type [20:22] • ErrDetail->Qword : Quad Word(Min: 0, Max: 24) • ErrDetail->Bit : Bit Position(Min: 0, Max: 127)
\ XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of CRAM error injection(0x10304) • Resp->RespMsg2: Status of CRAM error injection

Table 409: **XSem_Ssit_CmdCfrNjctErr Arguments** *(cont'd)*

Type	Member	Description
u32	TargetSlr	Target SLR index on which command is to be executed • 0x0 : Target is master only • 0x1 : Target is slave 0 only • 0x2 : Target is slave 1 only • 0x3 : Target is slave 2 only • 0xF : Broadcast not supported for this API Status of CRAM error injection: 0x00002000 – Null pointer error 0x00500000 – CRAM init not done 0x00800000 – Invalid row 0x00900000 – Invalid qword 0x00A00000 – Invalid bit 0x00B00000 – Invalid frame address 0x00C00000 – Unexpected bits flipped 0x00D00000 – Masked bit 0x00E00000 – Invalid block type 0x00F00000 – Active crc/uncor ecc error in CRAM

Returns

This API returns the success or failure.

- XST_FAILURE: On CRAM error injection failure
- XST_SUCCESS: On CRAM error injection success

XSem_Ssit_CmdCfrReadFrameEcc

This function is used to Read frame ECC of a particular Frame. Primarily this function sends an IPI request to PLM to invoke SEM CRAM SendFrameEcc on targeted SLR, waits for PLM to process the request and reads the response message.

Note:

- Total number of frames in a row is not same for all rows.
- XSem_CmdCfrGetTotalFrames API is provided to know the total number of frames in a row for each block. Output param (FrameCntPtr) of XSem_CmdCfrGetTotalFrames API is updated with total number of frames of each block type for the input row. If a particular block in a row has 0 frames, then error injection shall not be performed. Range of Frame number: 0 to (FrameCntPtr[n] - 1) where n is block type with range 0 to 6. ECC bits. The error injection will not change the design behaviour.

Prototype

```
XStatus XSem_Ssit_CmdCfrReadFrameEcc(XIpiPsu *IpiInst, u32 CframeAddr, u32
RowLoc, XSemIpiResp *Resp, u32 TargetSlr);
```

Parameters

The following table lists the `XSem_Ssit_CmdCfrReadFrameEcc` function arguments.

Table 410: **XSem_Ssit_CmdCfrReadFrameEcc Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
u32	CframeAddr	Frame Address • Frame Number [0:19] (Refer note) • Block Type [20:22]
u32	RowLoc	Row index(Min: 0 , Max: CFU_ROW_RANGE -1)
XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of CRAM Send Frame ECC(0x3030A) • Resp->RespMsg2: Segment 0 ECC value • Resp->RespMsg3: Segment 1 ECC value • Resp->RespMsg4: Status of CRAM stop scan
u32	TargetSlr	Target SLR index on which command is to be executed • 0x0 : Target is master only • 0x1 : Target is slave 0 only • 0x2 : Target is slave 1 only • 0x3 : Target is slave 2 only • 0xF : Broadcast not supported for this API

Returns

This API returns the success or failure.

- XST_FAILURE: On CRAM Read Frame ECC failure
- XST_SUCCESS: On CRAM Read Frame ECC success

XSem_Ssit_CmdGetStatus

This function is used to get the SEM status register values from all SLRs in SSIT device.

Prototype

```
XStatus XSem_Ssit_CmdGetStatus(XIpiPsu *IpiInst, XSemIpiResp *Resp, u32
TargetSlr, XSemStatus *StatusInfo);
```

Parameters

The following table lists the `XSem_Ssit_CmdGetStatus` function arguments.

Table 411: **XSem_Ssit_CmdGetStatus Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of get Cfr status(0x1030D) • Resp->RespMsg2: SLR Index in which the command is executed
u32	TargetSlr	Target SLR index on which command is to be executed • 0x0 : Target is master only • 0x1 : Target is slave 0 only • 0x2 : Target is slave 1 only • 0x3 : Target is slave 2 only

Table 411: **XSem_Ssit_CmdGetStatus Arguments** *(cont'd)*

Type	Member	Description
XSemStatus *	StatusInfo	Structure Pointer with SEM Status details • StatusInfo->NpiStatus: Provides details about NPI scan ◦ Bit [31]: 0 - No error, SHA-3 engine is present 1 - Cryptographic acceleration blocks are disabled for export compliance. No support for NPI scan, an event will be sent to R5 ◦ Bit [30]: 0 - NPI scan is running for scheduled interval 1 - Indicates NPI scan failed to run on scheduled interval. If NPI scan is not executed as per the configured periodicity, the error will notified to R5 user. The scan will continue to run ◦ Bit [29]: 0- No descriptor missed during scanning 1- Indicates NPI scan failed to scan all descriptors completely (excluding arbitration failures). This will be notified to R5 user and the scan will continue to run. ◦ Bit [28]: 0 - NPI scan executing within budget time 1 - Indicates NPI scan has exceeded maximum budget execution time of 20ms. This will be notified to R5 user and the scan will continue to run. ◦ Bit [27]: 0 - No error in SLR to SLR communication 1 - Indicates failure in SSIT internal communication channel This bit is applicable for SSIT devices. ◦ Bit [26]: Reserved ◦ Bit [25]: 0 - No error in PMC_PL_GPO 1 - Indicates GPO Initialization or write failed. This is HW failure. In this condition, the scan will be stopped, and notification will be sent to R5 ◦ Bit [24]: 0 - No error SHA-3 engine 1 - Indicates SHA engine failed to function during initialization or start or DMA transfer. This is HW failure. In this condition, the scan will be stopped, and notification will be sent to R5 ◦ Bit [23]: 0 - No error in register writes 1 - Indicates the register write and read back failure occurred during the scan. This is HW failure. In this condition, the scan will be stopped, and notification will be sent to R5 ◦ Bit [22]: Reserved ◦ Bit [21]: 0 - No error in DDR calibration 1 - Indicates NPI DDRMC Main Slave Arbitration Timeout occurred during the scan. If the DDRMC calibration is not done, the descriptor will be skipped and scan will continue to run for next descriptor ◦ Bit [20]: 0 - No error in descriptor format 1 - Indicates NPI Descriptor has invalid format. This failure indicates that there is some corruption in the XilSEM NPI descriptor data. The scan will be stopped, and notification will be sent to R5 ◦ Bit [19]: 0 - No error in NPI Descriptor SHA header 1 - Indicates NPI Descriptor SHA Header mismatch occurred during the scan. This failure indicates that there is some corruption in the XilSEM NPI descriptor data. The scan will be stopped, and notification will be sent to R5 ◦ Bit [18]: 0 - NPI descriptors are present in the memory 1 - Indicates the absence of NPI Descriptor (Zero descriptors) This failure indicates that there is some corruption in the XilSEM NPI descriptor data. The scan will be stopped, and notification will be sent to R5 ◦ Bit [17]: 0 - No error in SHA comparison during run time 1- Indicates SHA comparison failure occurred during run time. This failure indicates that there is some bit flip in the NPI registers. The scan will be stopped and an event will be sent to R5 ◦ Bit [16]: 0 - No error in SHA comparison during first scan 1-

Returns

This API returns the success or failure.

- XST_FAILURE: If NULL pointer reference of CfrStatusInfo
- XST_SUCCESS: On successful read from PMC RAM

XSem_Ssit_CmdNpiStartScan

This function is used to start NPI scan from user application. Primarily this function sends an IPI request to PLM to invoke SEM NPI StartScan on targeted SLR PLM, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_Ssit_CmdNpiStartScan(XIpiPsu *IpiInst, XSemIpiResp *Resp, u32
TargetSlr);
```

Parameters

The following table lists the `XSem_Ssit_CmdNpiStartScan` function arguments.

Table 412: **XSem_Ssit_CmdNpiStartScan Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance

XSemIpiResp *	Resp	Structure Pointer of IPI response
		• Resp->RespMsg1: Acknowledgment ID of NPI start scan(0x10305)
		• Resp->RespMsg2:
		  ◦ if Broadcast : Status of NpiStartScan in Master
		  ◦ else : Status of NpiStartScan in Target SLR
		• RespMsg3, 4 and 5 are updated only in case of broadcast
		• Resp->RespMsg3: Status of NpiStarttScan in Slave 1
		• Resp->RespMsg4: Status of NpiStarttScan in Slave 2
		• Resp->RespMsg5: Status of NpiStarttScan in Slave 3 Status of start scan: 0: Success, 1: Failure

u32	TargetSlr	Target SLR index on which command is to be executed • 0x0 : Target is master only • 0x1 : Target is slave 0 only • 0x2 : Target is slave 1 only • 0x3 : Target is slave 2 only • 0xF : Broadcast for all devices

Returns

This API returns the success or failure.

- XST_FAILURE: On NPI start scan failure
- XST_SUCCESS: On NPI start scan success

XSem_Ssit_CmdNpiStopScan

This function is used to stop NPI scan from user application. Primarily this function sends an IPI request to master PLM to invoke SEM NPI StopScan on targeted SLR PLM and waits for master PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_Ssit_CmdNpiStopScan(XIpiPsu *IpiInst, XSemIpiResp *Resp, u32
TargetSlr);
```

Parameters

The following table lists the `XSem_Ssit_CmdNpiStopScan` function arguments.

Table 413: **XSem_Ssit_CmdNpiStopScan Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance

XSemIpiResp *	Resp	Structure Pointer of IPI response
		• Resp->RespMsg1: Acknowledgment ID of NPI stop scan(0x10306)
		• Resp->RespMsg2:
		  ◦ if Broadcast : Status of NpiStoptScan in Master
		  ◦ else : Status of NpiStoptScan in Target SLR
		• RespMsg3, 4 and 5 are updated only in case of broadcast
		• Resp->RespMsg3: Status of NpiStoptScan in Slave 1
		• Resp->RespMsg4: Status of NpiStoptScan in Slave 2
		• Resp->RespMsg5: Status of NpiStoptScan in Slave 3 Status of stop scan: 0: Success, 1: Failure

u32	TargetSlr	Target SLR index on which command is to be executed • 0x0 : Target is master only • 0x1 : Target is slave 0 only • 0x2 : Target is slave 1 only • 0x3 : Target is slave 2 only • 0xF : Broadcast for all devices

Returns

This API returns the success or failure.

- XST_FAILURE: On NPI stop scan failure
- XST_SUCCESS: On NPI stop scan success

XSem_Ssit_CmdNpiInjectError

This function is used to inject SHA error in NPI descriptor list (in the first NPI descriptor) from user application on targeted SLR . Primarily this function sends an IPI request to PLM to invoke SEM NPI ErrorInject, waits for PLM to process the request and reads the response message of the given SLR.

***Note*:** The caller shall invoke this XSem_CmdNpiInjectError function again to correct the injected error in NPI descriptor.

Prototype

```
XStatus XSem_Ssit_CmdNpiInjectError(XIpiPsu *IpiInst, XSemIpiResp *Resp, u32
TargetSlr);
```

Parameters

The following table lists the `XSem_Ssit_CmdNpiInjectError` function arguments.

Table 414: **XSem_Ssit_CmdNpiInjectError Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of NPI error injection(0x10307) • Resp->RespMsg2: Status of NPI error injection
u32	TargetSlr	Target SLR index on which command is to be executed • 0x0 : Target is master only • 0x1 : Target is slave 0 only • 0x2 : Target is slave 1 only • 0x3 : Target is slave 2 only • 0xF : Broadcast not supported for this API Status of NPI scan error injection: 0: Success, 1: Failure

Returns

This API returns the success or failure.

- XST_FAILURE: On NPI error injection failure
- XST_SUCCESS: On NPI error injection success

XSem_Ssit_CmdGetConfig

This function is used to read CRAM & NPI configuration. Primarily this function sends an IPI request to PLM to invoke SEM Get configuration command on targeted SLR , waits for PLM to process the request and reads the response message for a given SLR.

- Resp->RespMsg2: CRAM Attribute register details
  - Bit [31:16]: Not Implemented
  - Bit [15:9]: Reserved
  - Bit [8]: Reserved
  - Bit [7]: Reserved
  - Bit [6:5]: Indicates when to start CRAM scan
    - 00: Do not automatically start scan
    - 01: Enable scan automatically after device configuration.
    - 10: Reserved
    - 11: Reserved
  - Bit [4]: Reserved
  - Bit [3]: Indicates HwECC/SwECC
    - 0: Uses hardware calculated ECC.
    - 1: Uses software calculated ECC that comes from tools and part of CDO
  - Bit [2]: Indicates Correctable error is to be corrected/not
    - 0: Disables error correction capability
    - 1: Enables error correction capability
  - Bit [1:0]: Define the mode of the scan (Enable/Disable scan)
    - 00: Disable Configuration RAM scan
    - 01: RESERVED
    - 10: Enable Configuration RAM scan
    - 11: RESERVED

- Resp->RespMsg3: NPI Attribute register details
  - Bit [31:24]: Not implemented
  - Bit [23:18]: Reserved
  - Bit [17:8]: The scheduled time in milliseconds that the NPI scan will be periodically performed. Default Setting: 0x064 = 100ms
  - Bit [7:6]: Reserved
  - Bit [5:4]: Indicates when to start NPI scan
    - 00: Do not automatically start scan
    - 01: Enable scan automatically after device configuration.
    - 10: Reserved
    - 11: Reserved
  - Bit [3]: Reserved
  - Bit [2]: Indicates HwSHA/SwSHA 0: Use hardware calculated SHA. 1: Use software calculated SHA.
  - Bit [1:0]: Reserved
- Resp->RespMsg4: Status of Get Configuration command

Prototype

```
XStatus XSem_Ssit_CmdGetConfig(XIpiPsu *IpiInst, XSemIpiResp *Resp, u32
TargetSlr);
```

Parameters

The following table lists the `XSem_Ssit_CmdGetConfig` function arguments.

Table 415: **XSem_Ssit_CmdGetConfig Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of Get Configuration(0x30309)

Table 415: **XSem_Ssit_CmdGetConfig Arguments** *(cont'd)*

Type	Member	Description
u32	TargetSlr	Target SLR index on which command is to be executed • 0x0 : Target is master only • 0x1 : Target is slave 0 only • 0x2 : Target is slave 1 only • 0x3 : Target is slave 2 only • 0xF : Broadcast not supported for this API

Returns

This API returns the success or failure.

- XST_FAILURE: On Get Configuration failure
- XST_SUCCESS: On Get Configuration success

XSem_Ssit_CmdCfrGetCrc

This function is used to read CFRAME golden CRC for a row in a target SLR.

Prototype

```
XStatus XSem_Ssit_CmdCfrGetCrc(XIpiPsu *IpiInst, u32 RowIndex, XSemIpiResp
*Resp, u32 TargetSlr);
```

Parameters

The following table lists the `XSem_Ssit_CmdCfrGetCrc` function arguments.

Table 416: **XSem_Ssit_CmdCfrGetCrc Arguments**

Type	Member	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
u32	RowIndex	Row index for which CRC to be read (Min: 0 , Max: CFU_ROW_RANGE -1)

Table 416: **XSem_Ssit_CmdCfrGetCrc Arguments** *(cont'd)*

Type	Member	Description
XSemIpiResp *	Resp	Structure Pointer of IPI response • Resp->RespMsg1: Acknowledgment ID of CFR get CRC (0x1030C) • Resp->RespMsg2: Status of CFR get CRC • Resp->RespMsg3: CRC register word 0 • Resp->RespMsg4: CRC register word 1 • Resp->RespMsg5: CRC register word 2 • Resp->RespMsg6: CRC register word 3
u32	TargetSlr	Target SLR index on which command is to be executed • 0x0 : Target is master only • 0x1 : Target is slave 0 only • 0x2 : Target is slave 1 only • 0x3 : Target is slave 2 only • 0xF : Broadcast not supported for this API

Returns

This API returns the success or failure.

- XST_FAILURE: On Get CRC failure
- XST_SUCCESS: On Get CRC success

Data Structures

XSem_DescriptorData

Structure contains descriptor information used during SEM NPI scan

Declaration

```
typedef struct
{
  u32 DescriptorCount,
  XSem_DescriptorInfo DescriptorInfo[NPI_MAX_DESCRIPTORS]
} XSem_DescriptorData;
```

Table 417: **Structure XSem_DescriptorData member description**

Member	Description
DescriptorCount	Current Total Descriptor Count
DescriptorInfo	Descriptor information: Contains descriptor attributes and golden SHA value

XSem_DescriptorInfo

Structure to hold each descriptor information

Descriptor Attribute: Contains detailed information related to descriptor

- Bit[31:16]: Base Address - Slave Base address in the descriptor for which attributes are present. Base address is only applicable for DDRMC_MAIN and GT slave address, for other descriptors, this value should be ignored.
- Bit[15:8]: Exclusive Lock Type - Bit[8] if set, signifies that descriptor contains GT slaves. Bit[9] if set, signifies that descriptor contains DDRMC slave registers. Bit[10:15] are reserved for future use
- Bit[7:0]: Slave skip count index - Points the corresponding byte in SEM NPI slave skip count registers that need to be updated on arbitration failure(Value is zero if arbitration is not required).

Descriptor Golden SHA: This word contains the Golden SHA value that is used to compare with hardware calculated SHA value.

Declaration

```
typedef struct
{
  u32 DescriptorAttrib,
  u32 DescriptorGldnSha
} XSem_DescriptorInfo;
```

Table 418: **Structure XSem_DescriptorInfo member description**

Member	Description
DescriptorAttrib	Descriptor attributes
DescriptorGldnSha	Descriptor Golden SHA value

XSem_Notifier

`XSem_Notifier` : This structure contains details of event notifications to be registered with the XilSEM server

Declaration

```
typedef struct
{
  u32 Module,
  u32 Event,
  u32 Flag
} XSem_Notifier;
```

Table 419: **Structure XSem_Notifier member description**

Member	Description
Module	The module information to receive notifications. Module can be either CRAM or NPI. • For CRAM: use XSEM_NOTIFY_CRAM • For NPI: use XSEM_NOTIFY_NPI
Event	The event types specify the specific event registration. For CRAM module, events are: • Uncorrectable ECC error: use XSEM_EVENT_CRAM_UNCOR_ECC_ERR • Uncorrectable CRC error: use XSEM_EVENT_CRAM_CRC_ERR • Internal error: use XSEM_EVENT_CRAM_INT_ERR • Correctable ECC error: use XSEM_EVENT_CRAM_COR_ECC_ERR For NPI module, events are: • Uncorrectable CRC error: use XSEM_EVENT_NPI_CRC_ERR • Unsupported Descriptor Format: use XSEM_EVENT_NPI_DESC_FMT_ERR • Descriptors absent for Scan: use XSEM_EVENT_NPI_DESC_ABSNT_ERR • SHA Indicator mismatch: use XSEM_EVENT_NPI_SHA_IND_ERR • SHA engine error: use XSEM_EVENT_NPI_SHA_ENGINE_ERR • Periodic Scan Missed: use XSEM_EVENT_NPI_PSCAN_MISSED_ERR • Cryptographic Accelerator Disabled: use XSEM_EVENT_NPI_CRYPTO_EXPORT_SET_ERR • Safety Write Failure: useXSEM_EVENT_NPI_SFTY_WR_ERR • GPIO Error event: use XSEM_EVENT_NPI_GPIO_ERR • Self Diagnosis failed: use XSEM_EVENT_NPI_SELF_DIAG_FAIL
Flag	Event flags to enable or disable notification. • To enable event notification: use XSEM_EVENT_ENABLE • To disable event notification: use XSEM_EVENT_DISABLE

XSem_XmpuCfg

XMPU Config Database

Declaration

```
typedef struct
{
  u32 DdrmcNocAddr,
  u32 XmpuRegionNum,
  u32 XmpuConfig,
  u32 XmpuStartAddrLo,
  u32 XmpuStartAddrUp,
  u32 XmpuEndAddrLo,
  u32 XmpuEndAddrUp,
  u32 XmpuMaster
} XSem_XmpuCfg;
```

Table 420: **Structure XSem_XmpuCfg member description**

Member	Description
DdrmcNocAddr	DDRMC NOC address
XmpuRegionNum	XMPU region number
XmpuConfig	XMPU entry config
XmpuStartAddrLo	XMPU start address lower portion
XmpuStartAddrUp	XMPU start address upper portion
XmpuEndAddrLo	XMPU end address lower portion
XmpuEndAddrUp	XMPU end address upper portion
XmpuMaster	XMPU master ID and mask

XSemCfrErrInjData

`XSemCfrErrInjData` - CRAM Error Injection structure to hold the Error Injection Location details for CRAM:

- Frame address
- Quad Word in a Frame, Range from 0 to 24
- Bit Position in a Quad Word, Range from 0 to 127
- Row Number, Range can be found from CFU_APB_CFU_ROW_RANGE register

Declaration

```
typedef struct
{
  u32 Efar,
  u32 Qword,
  u32 Bit,
  u32 Row
} XSemCfrErrInjData;
```

Table 421: **Structure XSemCfrErrInjData member description**

Member	Description
Efar	Frame Address
Qword	Quad Word 0...24
Bit	Bit Position 0...127
Row	Row Number

XSemCfrStatus

`XSemCfrStatus` - CRAM Status structure to store the data read from PMC RAM registers This structure provides:

- CRAM scan state information
- The low address of last 7 corrected error details if correction is enabled in design
- The high address of last 7 corrected error details if correction is enabled in design
- CRAM corrected error bits count value

Declaration

```
typedef struct
{
  u32 Status,
  u32 ErrAddrL[MAX_CRAMERR_REGISTER_CNT],
  u32 ErrAddrH[MAX_CRAMERR_REGISTER_CNT],
  u32 ErrCorCnt
} XSemCfrStatus;
```

Table 422: **Structure XSemCfrStatus member description**

Member	Description
Status	CRAM Status
ErrAddrL	Error Low register L0...L6
ErrAddrH	Error High register H0...H6
ErrCorCnt	Count of correctable errors

XSemIpiResp

XSemIpiResp - IPI Response Data structure

Declaration

```
typedef struct
{
  u32 RespMsg1,
  u32 RespMsg2,
  u32 RespMsg3,
  u32 RespMsg4,
  u32 RespMsg5,
  u32 RespMsg6,
  u32 RespMsg7
} XSemIpiResp;
```

Table 423: **Structure XSemIpiResp member description**

Member	Description
RespMsg1	Response word 1 (Ack Header)
RespMsg2	Response word 2
RespMsg3	Response word 3
RespMsg4	Response word 4
RespMsg5	Response word 5
RespMsg6	Response word 6
RespMsg7	Response word 7 (Reserved for CRC)

XSemNpiStatus

XSemNpiStatus - NPI Status structure to store the data read from PMC RAM registers. This structure provides:

- NPI scan status information (Refer XSem_CmdNpiGetStatus API)
- NPI descriptor slave skip counter value if arbitration fails
- NPI scan counter value
- NPI heartbeat counter value
- NPI scan error information if SHA mismatch is detected

Declaration

```
typedef struct
{
  u32 Status,
  u32 SlvSkipCnt[MAX_NPI_SLV_SKIP_CNT],
  u32 ScanCnt,
  u32 HbCnt,
  u32 ErrInfo[MAX_NPI_ERR_INFO_CNT]
} XSemNpiStatus;
```

Table 424: **Structure XSemNpiStatus member description**

Member	Description
Status	NPI scan status
SlvSkipCnt	Slave Skip Count: Contains the number of times NPI scan failed to get arbitration for GT/DDRMC descriptor
ScanCnt	NPI Scan Count: Increments each time NPI scan successfully completes a full scan (Including the first scan)
HbCnt	Heart Beat Count: Increments for each slave group scanned in the descriptor and each time SHA is calculated
ErrInfo	Error Information: Is updated with error details in case of SHA mismatch failure. -ErrInfo[0]: Calculated SHA value of the descriptor for which mismatch is observed. -ErrInfo[1]: Bit[31:16] Reserved. Bit[15:8] The descriptor index number for which the SHA failure is observed. Bit[7:0] The skip count index of the descriptor where SHA failure is observed (This value is zero if arbitration is not applicable for the descriptor)

XSemStatus

`XSemStatus` - SEM (CRAM & NPI) Status structure to store the data read from PMC RAM registers of any SLR This structure provides:

- NPI scan status information (Refer XSem_CmdNpiGetStatus API)
- NPI descriptor slave skip counter value if arbitration fails
- NPI scan counter value
- NPI heartbeat counter value
- NPI scan error information if SHA mismatch is detected
- CRAM scan state information
- The low address of last 7 corrected error details if correction is enabled in design
- The high address of last 7 corrected error details if correction is enabled in design
- CRAM corrected error bits count value

Declaration

```
typedef struct
{
  u32 NpiStatus,
  u32 SlvSkipCnt[MAX_NPI_SLV_SKIP_CNT],
  u32 ScanCnt,
  u32 HbCnt,
  u32 ErrInfo[MAX_NPI_ERR_INFO_CNT],
  u32 CramStatus,
  u32 ErrAddrL[MAX_CRAMERR_REGISTER_CNT],
  u32 ErrAddrH[MAX_CRAMERR_REGISTER_CNT],
  u32 ErrCorCnt
} XSemStatus;
```

Table 425: **Structure XSemStatus member description**

Member	Description
NpiStatus	NPI scan status
SlvSkipCnt	Slave Skip Count: Contains the number of times NPI scan failed to get arbitration for GT/DDRMC descriptor
ScanCnt	NPI Scan Count: Increments each time NPI scan successfully completes a full scan (Including the first scan)
HbCnt	Heart Beat Count: Increments for each slave group scanned in the descriptor and each time SHA is calculated
ErrInfo	Error Information: Is updated with error details in case of SHA mismatch failure. -ErrInfo[0]: Calculated SHA value of the descriptor for which mismatch is observed. -ErrInfo[1]: Bit[31:16] Reserved. Bit[15:8] The descriptor index number for which the SHA failure is observed. Bit[7:0] The skip count index of the descriptor where SHA failure is observed (This value is zero if arbitration is not applicable for the descriptor)
CramStatus	CRAM Status
ErrAddrL	Error Low register L0...L6
ErrAddrH	Error High register H0...H6
ErrCorCnt	Count of correctable errors

XilTimer Library v1.2

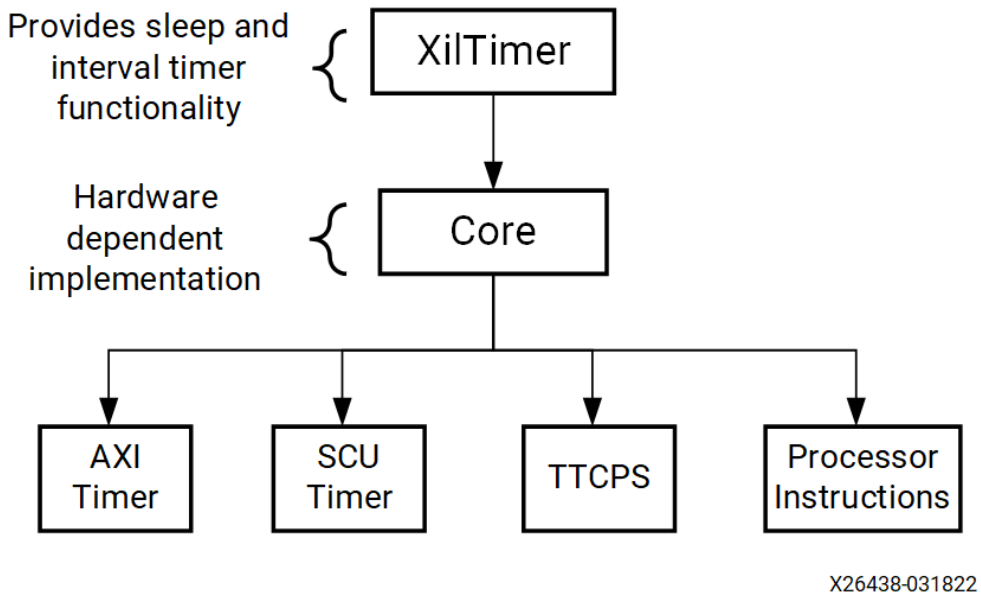
Overview

The XilTimer library provides sleep and interval timer functionality where the Hardware and Software features are differentiated using a layered approach. The top level layer is platform-agnostic where the core layer (low level) is hardware dependent.

If none of the above-mentioned IPs are present in a given design, then the sleep functionality will be executed using processor instructions. For the sleep and tick functionality, you can select a specific timer using the library software configuration wizard. The following figure illustrates the XilTimer functionality.

***Note*:** Currently, this library supports TTC, scutimer, and axi_timer IPs.

Figure 13: **XilTimer Library**



BSP Configuration Settings

For using sleep functionality, you can use sleep, msleep, or usleep APIs that provide delay in multiples of seconds, milliseconds, and microseconds.

For configuring interval timer functionality, execute the following steps.

1. Configure the interval period in XTimer_SetInterval API
2. Register the internal timer handler in XTimer_SetHandler API. This API is executed when the interval time expires.

XilTimer provides the following user configuration BSP settings.

***Note*:** By default, XilTimer library is disabled. You can enable it through the library software configuration wizard.

Table : **BSP Configuration Settings**

Parameter Name	Type	Description
sleep timer	peripheral_instance	This parameter is used to select specific timer for sleep functionality
interval_timer	peripheral_instance	This parameter is used to select specific timer for interval timer functionality
en_interval_timer	bool	This parameter enables Interval Time (Default value is false)

XilTimer APIs

This section contains the sleep API's and definitions for ttcps sleep implementation.

Table 426: **Quick Function Reference**

Type	Member	Arguments
void	XilTimer_Sleep	void
void	sleep	unsigned int seconds
void	msleep	unsigned long mseconds
void	usleep	unsigned long useconds

Table 426: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
void	XTimer_SetHandler	`XTimer_TickHandler` FuncPtr void * CallBackRef u8 Priority
void	XTimer_SetInterval	unsigned long delay
void	XTimer_ClearTickInterrupt	void

Functions

XilTimer_Sleep

This API contains common delay implementation using library APIs.

Prototype

```
void XilTimer_Sleep(unsigned long delay, XTimer_DelayType DelayType);
```

Returns

None

sleep

This API gives delay in seconds.

Prototype

```
void sleep(unsigned int seconds);
```

Parameters

The following table lists the `sleep` function arguments.

Table 427: **sleep Arguments**

Type	Member	Description
unsigned int	seconds	Delay time in seconds

Returns

none

msleep

This API gives delay in msec

Note: none

Prototype

```
void msleep(unsigned long mseconds);
```

Parameters

The following table lists the `msleep` function arguments.

Table 428: **msleep Arguments**

Type	Member	Description
unsigned long	mseconds	Delay time in milliseconds

Returns

none

usleep

This API gives delay in usec

Note: none

Prototype

```
void usleep(unsigned long useconds);
```

Parameters

The following table lists the `usleep` function arguments.

Table 429: **usleep Arguments**

Type	Member	Description
unsigned long	useconds	Delay time in microseconds

Returns

none

XTimer_SetHandler

This routine installs an asynchronous callback function for the given FuncPtr.

Prototype

```
void XTimer_SetHandler(XTimer_TickHandler FuncPtr, void *CallBackRef, u8
Priority);
```

Parameters

The following table lists the `XTimer_SetHandler` function arguments.

Table 430: **XTimer_SetHandler Arguments**

Type	Member	Description
`XTimer_TickHandler`	FuncPtr	is the address of the callback function.
void *	CallBackRef	is a user data item that will be passed to the callback function when it is invoked.
u8	Priority	- Priority for the interrupt

Returns

None

XTimer_SetInterval

This API sets the elapse interval for the timer instance.

Note: none

Prototype

```
void XTimer_SetInterval(unsigned long delay);
```

Parameters

The following table lists the `XTimer_SetInterval` function arguments.

Table 431: **XTimer_SetInterval Arguments**

Type	Member	Description
unsigned long	delay	Delay time in milliseconds

Returns

none

XTimer_ClearTickInterrupt

This API clears the interrupt status of the tick timer instance.

***Note*:** none

Prototype

```
void XTimer_ClearTickInterrupt(void);
```

Returns

none

Data Structures

XTimerTag

Structure to the XTimer instance.

Declaration

```
typedef struct
{
  void(* XTimer_ModifyInterval)(struct XTimerTag *InstancePtr, u32 delay,
XTimer_DelayType Delaytype),
  void(* XTimer_TickIntrHandler)(struct XTimerTag *InstancePtr, u8 Priority),
  void(* XTimer_TickInterval)(struct XTimerTag *InstancePtr, u32 Delay),
  void(* XSleepTimer_Stop)(struct XTimerTag *InstancePtr),
  void(* XTickTimer_Stop)(struct XTimerTag *InstancePtr),
  void(* XTickTimer_ClearInterrupt)(struct XTimerTag *InstancePtr),
  XTimer_TickHandler Handler,
  void * CallBackRef,
  XTmrCtr AxiTimer_SleepInst,
  XTmrCtr AxiTimer_TickInst,
  XTtcPs TtcPs_SleepInst,
  XTtcPs TtcPs_TickInst,
  XScuTimer ScuTimer_SleepInst,
  XScuTimer ScuTimer_TickInst
} XTimerTag;
```

Table 432: **Structure XTimerTag member description**

Member	Description
XTimer_ModifyInterval	Modifies the timer interval

Table 432: **Structure XTimerTag member description** *(cont'd)*

Member	Description
XTimer_TickIntrHandler	Tick interrupt handler
XTimer_TickInterval	Configures the tick interval
XSleepTimer_Stop	Stops the sleep timer
XTickTimer_Stop	Stops the tick timer
XTickTimer_ClearInterrupt	Clears the Tick timer interrupt status
Handler	Callback function
CallBackRef	Callback reference for handler
AxiTimer_SleepInst	Sleep Instance for AxiTimer
AxiTimer_TickInst	Tick Instance for AxiTimer
TtcPs_SleepInst	Sleep Instance for TTCPS
TtcPs_TickInst	Tick Instance for TTCPS
ScuTimer_SleepInst	Sleep Instance for Scutimer
ScuTimer_TickInst	Tick Instance for Scutimer

Appendix A

Additional Resources and Legal Notices

Finding Additional Documentation

Documentation Portal

The AMD Adaptive Computing Documentation Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Documentation Portal, go to https://docs.xilinx.com.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

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

Design Hubs

AMD 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.
- Go to the Design Hubs webpage.

***Note*:** For more information on DocNav, see the Documentation Navigator webpage.

Support Resources

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

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
05/16/2023 Version 2023.1	
Entire document.	Updated all libraries for 2023.1 version.

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

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

Copyright