Xilinx Standalone Library Documentation

OS and Libraries Document Collection

UG643 (2018.3) December 5, 2018

XILINX.

Table of Contents

Chapter 1: Xilinx OS and Libraries Overview

Xilinx Standard C Libraries

Chapter 2: Xilinx Standard C Libraries

Standalone Library (v6.8)

Chapter 3: Xilinx Hardware Abstraction Layer API

Chapter 5: Cortex R5 Processor API

Chapter 6: ARM Processor Common API

Chapter 7: Cortex A9 Processor API

Chapter 8: Cortex A53 32-bit Processor API

Chapter 9: Cortex A53 64-bit Processor API

XilMFS Library (v2.3)

Chapter 10: Overview

Chapter 11: XilMFS Library API

Chapter 12: Utility Functions

Chapter 13: Library Parameters in MSS File

LwIP 2.0.2 Library (v1_2)

Chapter 14: Introduction

Chapter 15: Using lwIP

Chapter 16: LwIP Library APIs

XilFlash Library (v4.5)

Chapter 17: Overview

Chapter 18: XilFlash Library API

Chapter 19: Library Parameters in MSS File

XilSecure Library (v3.2)

Chapter 25: Overview

Chapter 26: AES-GCM

Chapter 27: RSA

Chapter 28: SHA-3

Chapter 29: SHA-2

XilRSA Library (v1.5)

Chapter 30: Overview

Chapter 31: XilRSA APIs

XilSKey Library (v6.6)

Chapter 32: Overview

XilPM Library (v2.4)

XilFPGA Library (v4.2)

Chapter 44: Overview

Chapter 1

Xilinx OS and Libraries Overview

The Software Development Kit (SDK) provides a variety of Xilinx® software packages, including drivers, libraries, board support packages, and complete operating systems to help you develop a software platform. This document collection provides information on these.

Complete documentation for other operating systems can be found in the their respective reference guides. Device drivers are documented along with the corresponding peripheral documentation. The documentation is listed in the following table; click the name to open the document.

Document ID	Document Name	Summary
UG645	Xilinx Standard C Libraries	Describes the software libraries available for the embedded processors.
UG647	Standalone Library Reference v6.8	Describes the Standalone platform, a single-threaded, simple operating system (OS) platform that provides the lowest layer of software modules used to access processor-specific functions. Some typical functions offered by the Standalone platform include setting up the interrupts and exceptions systems, configuring caches, and other hardware specific functions. The Hardware Abstraction Layer (HAL) is described in this document.

Document ID	Document Name	Summary
UG649	LibXil Memory File System (MFS) Library v2.3	Describes a simple, memory-based file system that can reside in RAM, ROM, or Flash memory.
UG650	LwIP 1.4.1 Library v1_2	Describes the SDK port of the third party networking library, Light Weight IP (lwIP) for embedded processors.
UG651	XilFlash Library v4.5	Describes the functionality provided in the flash programming library. This library provides access to flash memory devices that conform to the Common Flash Interface (CFI) standard. Intel and AMD CFI devices for some specific part layouts are currently supported.
UG652	XilIsf Library v5.12	Describes the In System Flash hardware library, which enables higher-layer software (such as an application) to communicate with the Isf. XilIsf supports the Xilinx In-System Flash and external Serial Flash memories from Atmel (AT45XXXD), Spansion(S25FLXX), Winbond W25QXX, and Micron N25QXX.
UG1032	XilFFS Library v4.0	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. A link to the source of file system is provided in the PDF where the file system description can be found.

Document ID	Document Name	Summary
UG1225	XilPM Library v2.4	The Zynq UltraScale+ MPSoC power management framework is a set of power management options, based upon an implementation of the extensible energy management interface (EEMI). The power management framework allows software components running across different processing units (PUs) on a chip or device to issue or respond to requests for power management.
UG1189	XilSecure Library v3.2	The XilSecure library provides APIs to access secure hardware on the Zynq UltraScale+ MPSoC devices.
UG1190	XilRSA Library v1.5	The XilRSA library provides APIs to use RSA encryption and decryption algorithms and SHA algorithms.
UG1191	XilSKey Library v6.6	The XilSKey library provides a programming mechanism for user-defined eFUSE bits and for programming the KEY into battery-backed RAM (BBRAM) of Zynq® SoC, provides programming mechanisms for eFUSE bits of UltraScale™ devices. The library also provides programming mechanisms for eFUSE bits and BBRAM key of the Zynq® UltraScale+™ MPSoC devices.
UG1229	XilFPGA Library v4.2	The XilFPGA library provides an interface to the Linux or bare-metal users for configuring the programmable logic (PL) over PCAP from PS. The library is designed for Zynq UltraScale+ MPSoC devices to run on top of Xilinx standalone BSPs.

About the Libraries

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

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

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

Library Organization

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

Note

> “LibXil Drivers” are the device drivers included in the software platform to provide an interface to the peripherals in the system. These drivers are provided along with Xilinx SDK and are configured by Libgen. This document collection contains a chapter that briefly discusses the concept of device drivers and the way they integrate with the board support package in Xilinx SDK.

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

Xilinx Standard C Libraries

Chapter 2

Xilinx Standard C Libraries

Overview

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

Standard C Library (libc.a)

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

- `<XILINX_SDK>` is the Xilinx SDK installation path
- `<processor>` is ARM or MicroBlaze
- `<platform>` is Solaris (sol), Windows (nt), or Linux (lin)
- `<processor-lib>` is `arm-xilinx-eabi` or `microblaze-xilinx-elf`

The `lib.c` directories and functions are:

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

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

- For MicroBlaze processors:

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

- For Cortex A9 processors:

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

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

Xilinx C Library (libxil.a)

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

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

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

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

Memory Management Functions

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

Note

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

Arithmetic Operations

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

MicroBlaze Processor

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

Integer Arithmetic

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

Floating Point Arithmetic

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

Thread Safety

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

Modules

- Input/Output Functions

Input/Output Functions

Overview

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

Note

> The C standard I/O routines such as printf, scanf, vfprintf are, by default, line buffered. To change the buffering scheme to no buffering, you must call setvbuf appropriately. For example:
>
> ```
> setvbuf (stdout, NULL, _IONBF, 0);
> ```

These Input/Output routines require that a newline is terminated with both a CR and LF. Ensure that your terminal CR/LF behavior corresponds to this requirement.
For more information on setting the standard input and standard output devices for a system, see *Embedded System Tools Reference Manual* (UG1043). In addition to the standard C functions, the SDK processors library provides the following smaller I/O functions:

Functions

- void print (char ∗)
- void putnum (int)
- void xil_printf (const ∗char ctrl1,...)

Function Documentation

void print (char ∗)

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

void putnum (int)

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

void xil_printf (const ∗char *ctrl1,* ...)

xil_printf() is a light-weight implementation of `printf`. It is much smaller in size (only 1 Kb). It does not have support for floating point numbers. `xil_printf()` also does not support printing of long (such as 64-bit) numbers.
About format string support:
The format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %, and ends with a conversion specifier.
In between there can be (in order) zero or more flags, an optional minimum field width and an optional precision.
Supported flag characters are:
The character % is followed by zero or more of the following flags:

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

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

- d The int argument is converted to signed decimal notation.
- `l` The int argument is converted to a signed long notation.

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

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

Standalone (v6.8)

Chapter 3

Xilinx Hardware Abstraction Layer API

Overview

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

Modules

- Assert APIs
- IO interfacing APIs
- Definitions for available xilinx platforms
- Data types for Xilinx Software IP Cores
- Customized APIs for memory operations
- Xilinx software status codes
- Test utilities for memory and caches

Assert APIs

Overview

The `xil_assert.h` file contains the assert related functions.

Macros

- #define Xil_AssertVoid(Expression)
- #define Xil_AssertNonvoid(Expression)
- #define Xil_AssertVoidAlways()
- #define Xil_AssertNonvoidAlways()

Typedefs

- typedef void(∗ Xil_AssertCallback) (const char8 ∗File, s32 Line)

Functions

- void Xil_Assert (const char8 ∗File, s32 Line)
- void XNullHandler (void ∗NullParameter)
- void Xil_AssertSetCallback (Xil_AssertCallback Routine)

Variables

- u32 Xil_AssertStatus
- s32 Xil_AssertWait

Macro Definition Documentation

#define Xil_AssertVoid(*Expression*)

This assert macro is to be used for void functions. This in conjunction with the Xil_AssertWait boolean can be used to accomodate tests so that asserts which fail allow execution to continue.

Parameters

Expression	expression to be evaluated. If it evaluates to false, the assert occurs.

Returns

Returns void unless the Xil_AssertWait variable is true, in which case no return is made and an infinite loop is entered.

#define Xil_AssertNonvoid(*Expression*)

This assert macro is to be used for functions that do return a value. This in conjunction with the Xil_AssertWait boolean can be used to accomodate tests so that asserts which fail allow execution to continue.

Parameters

Expression	expression to be evaluated. If it evaluates to false, the assert occurs.

Returns

Returns 0 unless the Xil_AssertWait variable is true, in which case no return is made and an infinite loop is entered.

#define Xil_AssertVoidAlways( )

Always assert. This assert macro is to be used for void functions. Use for instances where an assert should always occur.

Returns

Returns void unless the Xil_AssertWait variable is true, in which case no return is made and an infinite loop is entered.

#define Xil_AssertNonvoidAlways( )

Always assert. This assert macro is to be used for functions that do return a value. Use for instances where an assert should always occur.

Returns

Returns void unless the Xil_AssertWait variable is true, in which case no return is made and an infinite loop is entered.

Typedef Documentation

typedef void(∗ Xil_AssertCallback) (const char8 ∗File, s32 Line)

This data type defines a callback to be invoked when an assert occurs. The callback is invoked only when asserts are enabled

Function Documentation

void Xil_Assert (const char8 ∗ *File,* s32 *Line*)

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.

Parameters

file	filename of the source
line	linenumber within File

Returns

None.

Note

None.

void XNullHandler (void * *NullParameter*)

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.

Parameters

NullParameter	arbitrary void pointer and not used.

Returns

None.

Note

None.

void Xil_AssertSetCallback (Xil_AssertCallback *Routine*)

Set up a callback function to be invoked when an assert occurs. If a callback is already installed, then it will be replaced.

Parameters

routine	callback to be invoked when an assert is taken

Returns

None.

Note

This function has no effect if NDEBUG is set

Variable Documentation

u32 Xil_AssertStatus

This variable allows testing to be done easier with asserts. An assert sets this variable such that a driver can evaluate this variable to determine if an assert occurred.

s32 Xil_AssertWait

This variable allows the assert functionality to be changed for testing such that it does not wait infinitely. Use the debugger to disable the waiting during testing of asserts.

IO interfacing APIs

Overview

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

Functions

- u16 Xil_EndianSwap16 (u16 Data)
- u32 Xil_EndianSwap32 (u32 Data)
- static INLINE u8 Xil_In8 (UINTPTR Addr)
- static INLINE u16 Xil_In16 (UINTPTR Addr)
- static INLINE u32 Xil_In32 (UINTPTR Addr)
- static INLINE u64 Xil_In64 (UINTPTR Addr)
- static INLINE void Xil_Out8 (UINTPTR Addr, u8 Value)
- static INLINE void Xil_Out16 (UINTPTR Addr, u16 Value)
- static INLINE void Xil_Out32 (UINTPTR Addr, u32 Value)
- static INLINE void Xil_Out64 (UINTPTR Addr, u64 Value)
- static INLINE u16 Xil_In16LE (UINTPTR Addr)
- static INLINE u32 Xil_In32LE (UINTPTR Addr)
- static INLINE void Xil_Out16LE (UINTPTR Addr, u16 Value)
- static INLINE void Xil_Out32LE (UINTPTR Addr, u32 Value)
- static INLINE u16 Xil_In16BE (UINTPTR Addr)
- static INLINE u32 Xil_In32BE (UINTPTR Addr)
- static INLINE void Xil_Out16BE (UINTPTR Addr, u16 Value)
- static INLINE void Xil_Out32BE (UINTPTR Addr, u32 Value)

Function Documentation

u16 Xil_EndianSwap16 (u16 *Data*)

Perform a 16-bit endian converion.

Parameters

Data	16 bit value to be converted

Returns

converted value.

u32 Xil_EndianSwap32 (u32 *Data*)

Perform a 32-bit endian converion.

Parameters

Data	32 bit value to be converted

Returns

converted value.

static INLINE u8 Xil_In8 (UINTPTR *Addr*) `[static]`

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

Parameters

Addr	contains the address to perform the input operation at.

Returns

The Value read from the specified input address.

Note

None.

static INLINE u16 Xil_In16 (UINTPTR *Addr*) [static]

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

Parameters

Addr	contains the address to perform the input operation at.

Returns

The Value read from the specified input address.

Note

None.

static INLINE u32 Xil_In32 (UINTPTR *Addr*) [static]

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

Parameters

Addr	contains the address to perform the input operation at.

Returns

The Value read from the specified input address.

Note

None.

static INLINE u64 Xil_In64 (UINTPTR *Addr*) [static]

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

Parameters

Addr	contains the address to perform the input operation at.

Returns

The Value read from the specified input address.

Note

None.

static INLINE void Xil_Out8 (UINTPTR *Addr*, u8 *Value*) `[static]`

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

Parameters

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

Returns

None.

Note

None.

static INLINE void Xil_Out16 (UINTPTR *Addr*, u16 *Value*) `[static]`

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

Parameters

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

Returns

None.

Note

None.

static INLINE void Xil_Out32 (UINTPTR *Addr*, u32 *Value*) `[static]`

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

Parameters

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

Returns

None.

Note

None.

static INLINE void Xil_Out64 (UINTPTR *Addr,* u64 *Value*) `[static]`

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

Parameters

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

Returns

None.

Note

None.

static INLINE u16 Xil_In16LE (UINTPTR *Addr*) `[static]`

Perform a little-endian input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters

Addr	contains the address at which to perform the input operation.

Returns

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is big-endian, the return value is the byte-swapped value read from the address.

static INLINE u32 Xil_In32LE (UINTPTR *Addr*) [static]

Perform a little-endian input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters

Addr	contains the address at which to perform the input operation.

Returns

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is big-endian, the return value is the byte-swapped value read from the address.

static INLINE void Xil_Out16LE (UINTPTR *Addr,* u16 *Value*) [static]

Perform a little-endian output operation for a 16-bit memory location by writing the specified value to the specified address.

Parameters

Addr	contains the address at which to perform the output operation.
Value	contains the value to be output at the specified address. The value has the same endianness as that of the processor. For example, if the processor is big-endian, the byteswapped value is written to the address.

static INLINE void Xil_Out32LE (UINTPTR *Addr,* u32 *Value*) [static]

Perform a little-endian output operation for a 32-bit memory location by writing the specified value to the specified address.

Parameters

Addr	contains the address at which to perform the output operation.
Value	contains the value to be output at the specified address. The value has the same endianness as that of the processor. For example, if the processor is big-endian, the byteswapped value is written to the address.

static INLINE u16 Xil_In16BE (UINTPTR *Addr*) `[static]`

Perform an big-endian input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters

Addr	contains the address at which to perform the input operation.

Returns

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is little-endian, the return value is the byte-swapped value read from the address.

static INLINE u32 Xil_In32BE (UINTPTR *Addr*) `[static]`

Perform a big-endian input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters

Addr	contains the address at which to perform the input operation.

Returns

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is little-endian, the return value is the byte-swapped value read from the address.

static INLINE void Xil_Out16BE (UINTPTR *Addr,* u16 *Value*) `[static]`

Perform a big-endian output operation for a 16-bit memory location by writing the specified value to the specified address.

Parameters

Addr	contains the address at which to perform the output operation.
Value	contains the value to be output at the specified address. The value has the same endianness as that of the processor. For example, if the processor is little-endian, the byteswapped value is written to the address.

static INLINE void Xil_Out32BE (UINTPTR *Addr,* u32 *Value*) [static]

Perform a big-endian output operation for a 32-bit memory location by writing the specified value to the specified address.

Parameters

Addr	contains the address at which to perform the output operation.
Value	contains the value to be output at the specified address. The value has the same endianness as that of the processor. For example, if the processor is little-endian, the byteswapped value is written to the address.

Definitions for available xilinx platforms

Overview

The `xplatform_info.h` file contains definitions for various available Xilinx® platforms.

Functions

- u32 XGetPlatform_Info ()
- u32 XGetPSVersion_Info ()
- u32 XGet_Zynq_UltraMp_Platform_info ()

Function Documentation

u32 XGetPlatform_Info ( )

This API is used to provide information about platform.

Parameters

None.	

Returns

The information about platform defined in xplatform_info.h

u32 XGetPSVersion_Info ( )

This API is used to provide information about PS Silicon version.

Parameters

None.	

Returns

The information about PS Silicon version.

u32 XGet_Zynq_UltraMp_Platform_info ( )

This API is used to provide information about zynq ultrascale MP platform.

Parameters

None.	

Returns

The information about zynq ultrascale MP platform defined in xplatform_info.h

Data types for Xilinx Software IP Cores

Overview

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

Macros

- #define XIL_COMPONENT_IS_READY
- #define XIL_COMPONENT_IS_STARTED

New types

New simple types.

- typedef uint8_t u8
- typedef uint16_t **u16**
- typedef uint32_t **u32**
- typedef char char8
- typedef int8_t **s8**
- typedef int16_t **s16**

- typedef int32_t **s32**
- typedef int64_t **s64**
- typedef uint64_t **u64**
- typedef int **sint32**
- typedef intptr_t **INTPTR**
- typedef uintptr_t **UINTPTR**
- typedef ptrdiff_t **PTRDIFF**
- typedef long **LONG**
- typedef unsigned long **ULONG**
- typedef void(∗ XInterruptHandler) (void ∗InstancePtr)
- typedef void(∗ XExceptionHandler) (void ∗InstancePtr)
- #define **__XUINT64__**
- #define XUINT64_MSW(x)
- #define XUINT64_LSW(x)
- #define **ULONG64_HI_MASK**
- #define **ULONG64_LO_MASK**
- #define UPPER_32_BITS(n)
- #define LOWER_32_BITS(n)

Macro Definition Documentation

#define XIL_COMPONENT_IS_READY

component has been initialized

#define XIL_COMPONENT_IS_STARTED

component has been started

#define XUINT64_MSW(*x*)

Return the most significant half of the 64 bit data type.

Parameters

x	is the 64 bit word.

Returns

The upper 32 bits of the 64 bit word.

#define XUINT64_LSW(*x*)

Return the least significant half of the 64 bit data type.

Parameters

x	is the 64 bit word.

Returns

The lower 32 bits of the 64 bit word.

#define UPPER_32_BITS(*n*)

return bits 32-63 of a number

Parameters

n	: the number we're accessing

Returns

bits 32-63 of number

Note

A basic shift-right of a 64- or 32-bit quantity. Use this to suppress the "right shift count >= width of type" warning when that quantity is 32-bits.

#define LOWER_32_BITS(*n*)

return bits 0-31 of a number

Parameters

n	: the number we're accessing

Returns

bits 0-31 of number

Typedef Documentation

typedef uint8_t u8

guarded against xbasic_types.h.

typedef char char8

xbasic_types.h does not typedef s* or u64

typedef void(* XInterruptHandler) (void *InstancePtr)

This data type defines an interrupt handler for a device. The argument points to the instance of the component

typedef void(* XExceptionHandler) (void *InstancePtr)

This data type defines an exception handler for a processor. The argument points to the instance of the component

Customized APIs for memory operations

Overview

The `xil_mem.h` file contains prototypes for function related to memory operations. These APIs are applicable for all processors supported by Xilinx®.

Functions

- void Xil_MemCpy (void *dst, const void *src, u32 cnt)

Function Documentation

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

This function copies memory from once location to other.

Parameters

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

Xilinx software status codes

Overview

The `xstatus.h` file contains Xilinx® software status codes. Status codes have their own data type called `int`. These codes are used throughout the Xilinx device drivers.

Test utilities for memory and caches

Overview

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.

- **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 are descriptions of Memory test subtests:

    - XIL_TESTMEM_ALLMEMTESTS: Runs all of the subtests.
    - XIL_TESTMEM_INCREMENT: Incrementing Value Test. This test starts at `XIL_TESTMEM_INIT_VALUE` and uses the incrementing value as the test value for memory.
    - XIL_TESTMEM_WALKONES: Walking Ones Test. This test uses a walking `1` as the test value for memory.

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

    - XIL_TESTMEM_WALKZEROS: 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: 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: 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`.

⚠ **WARNING:** *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.

Functions

- s32 Xil_TestIO8 (u8 ∗Addr, s32 Length, u8 Value)
- s32 Xil_TestIO16 (u16 ∗Addr, s32 Length, u16 Value, s32 Kind, s32 Swap)
- s32 Xil_TestIO32 (u32 ∗Addr, s32 Length, u32 Value, s32 Kind, s32 Swap)
- 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)

Memory subtests

- #define XIL_TESTMEM_ALLMEMTESTS
- #define **XIL_TESTMEM_INCREMENT**
- #define **XIL_TESTMEM_WALKONES**
- #define **XIL_TESTMEM_WALKZEROS**
- #define **XIL_TESTMEM_INVERSEADDR**
- #define **XIL_TESTMEM_FIXEDPATTERN**
- #define **XIL_TESTMEM_MAXTEST**

Macro Definition Documentation

#define XIL_TESTMEM_ALLMEMTESTS

See the detailed description of the subtests in the file description.

Function Documentation

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

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.

Parameters

Addr	a pointer to the region of memory to be tested.
Length	Length of the block.
Value	constant used for writting the memory.

Returns

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

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

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.

Parameters

Addr	a pointer to the region of memory to be tested.
Length	Length of the block.
Value	constant used for writting 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

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

Perform a destructive 32-bit wide register IO test. Each location is tested by sequentially writing a 32-bit wide regsiter, 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.

Parameters

Addr	a pointer to the region of memory to be tested.
Length	Length of the block.
Value	constant used for writting 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

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

Perform a destructive 32-bit wide memory test.

Parameters

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 for possible values.

Returns

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

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 boundry 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.

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

Perform a destructive 16-bit wide memory test.

Parameters

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	type of test selected. See xil_testmem.h for possible values.

Returns

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

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 boundry 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.

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

Perform a destructive 8-bit wide memory test.

Parameters

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	type of test selected. See xil_testmem.h for possible values.

Returns

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

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 boundry 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.

Chapter 4

Microblaze Processor API

Overview

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

Modules

- Microblaze Pseudo-asm Macros and Interrupt handling APIs
- Microblaze exception APIs
- Microblaze Processor Cache APIs
- MicroBlaze Processor FSL Macros
- Microblaze PVR access routines and macros
- Sleep Routines for Microblaze

Microblaze Pseudo-asm Macros and Interrupt handling APIs

Overview

Standalone 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

Functions

- void microblaze_register_handler (XInterruptHandler Handler, void ∗DataPtr)
- void microblaze_register_exception_handler (u32 ExceptionId, Xil_ExceptionHandler Handler, void ∗DataPtr)

Microblaze pseudo-asm macros

The following is a summary of the MicroBlaze processor pseudo-asm macros.

- #define mfgpr(rn)
- #define mfmsr()
- #define mfear()
- #define **mfeare**()
- #define mfesr()
- #define mffsr()

Macro Definition Documentation

#define mfgpr(*rn*)

Return value from the general purpose register (GPR) rn.

Parameters

rn	General purpose register to be read.

#define mfmsr( )

Return the current value of the MSR.

Parameters

None	

#define mfear( )

Return the current value of the Exception Address Register (EAR).

Parameters

None	

#define mfesr( )

Return the current value of the Exception Status Register (ESR).

Parameters

None	

#define mffsr( )

Return the current value of the Floating Point Status (FPS).

Parameters

None	

Function Documentation

void microblaze_register_handler (XInterruptHandler *Handler,* void ∗ *DataPtr*)

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.

Parameters

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

Returns

None.

void microblaze_register_exception_handler (u32 *ExceptionId,* Xil_ExceptionHandler *Handler,* void ∗ *DataPtr*)

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.

Parameters

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.

Note

None.

Microblaze exception APIs

Overview

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

Data Structures

- struct MB_ExceptionVectorTableEntry

Typedefs

- typedef void(∗ Xil_ExceptionHandler) (void ∗Data)
- typedef void(∗ XInterruptHandler) (void ∗InstancePtr)

Functions

- 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)

Data Structure Documentation

struct MB_ExceptionVectorTableEntry

Currently HAL is an augmented part of standalone BSP, so the old definition of MB_ExceptionVectorTableEntry is used here.

Typedef Documentation

typedef void(∗ Xil_ExceptionHandler) (void ∗Data)

This typedef is the exception handler function.

typedef void(∗ XInterruptHandler) (void ∗InstancePtr)

This data type defines an interrupt handler for a device. The argument points to the instance of the component

Function Documentation

void Xil_ExceptionInit (void)

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

Parameters

None.	

Returns

None.

void Xil_ExceptionEnable (void)

Enable Exceptions.

Returns

None.

void Xil_ExceptionDisable (void)

Disable Exceptions.

Parameters

None.	

Returns

None.

void Xil_ExceptionRegisterHandler (u32 *Id*, Xil_ExceptionHandler *Handler*, void ∗ *Data*)

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.

Parameters

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.

void Xil_ExceptionRemoveHandler (u32 *Id*)

Removes the handler for a specific exception Id. The stub handler is then registered for this exception Id.

Parameters

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

Microblaze Processor Cache APIs

Overview

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.

Note

Macros

- void Xil_L1DCacheInvalidate()
- void Xil_L2CacheInvalidate()
- void Xil_L1DCacheInvalidateRange(Addr, Len)
- void Xil_L2CacheInvalidateRange(Addr, Len)
- void Xil_L1DCacheFlushRange(Addr, Len)
- void Xil_L2CacheFlushRange(Addr, Len)

- void Xil_L1DCacheFlush()
- void Xil_L2CacheFlush()
- void Xil_L1ICacheInvalidateRange(Addr, Len)
- void Xil_L1ICacheInvalidate()
- void Xil_L1DCacheEnable()
- void Xil_L1DCacheDisable()
- void Xil_L1ICacheEnable()
- void Xil_L1ICacheDisable()
- void Xil_DCacheEnable()
- void Xil_ICacheEnable()

Functions

- void Xil_DCacheDisable (void)
- void Xil_ICacheDisable (void)

Macro Definition Documentation

void Xil_L1DCacheInvalidate( )

Invalidate the entire L1 data cache. If the cacheline is modified (dirty), the modified contents are lost.

Parameters

None.	

Returns

None.

Note

Processor must be in real mode.

void Xil_L2CacheInvalidate( )

Invalidate the entire L2 data cache. If the cacheline is modified (dirty),the modified contents are lost.

Parameters

None.	

Returns

None.

Note

Processor must be in real mode.

void Xil_L1DCacheInvalidateRange(*Addr*, *Len*)

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.

Parameters

Addr	is address of ragne to be invalidated.
Len	is the length in bytes to be invalidated.

Returns

None.

Note

Processor must be in real mode.

void Xil_L2CacheInvalidateRange(*Addr*, *Len*)

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.

Parameters

Addr	address of ragne to be invalidated.
Len	length in bytes to be invalidated.

Returns

None.

Note

Processor must be in real mode.

void Xil_L1DCacheFlushRange(*Addr*, *Len*)

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

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

Returns

None.

void Xil_L2CacheFlushRange(*Addr*, *Len*)

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

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

Returns

None.

void Xil_L1DCacheFlush( )

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.

Returns

None.

void Xil_L2CacheFlush( )

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.

Returns

None.

void Xil_L1ICacheInvalidateRange(*Addr*, *Len*)

Invalidate the instruction cache for the given address range.

Parameters

Addr	is address of ragne to be invalidated.
Len	is the length in bytes to be invalidated.

Returns

None.

void Xil_L1ICacheInvalidate( )

Invalidate the entire instruction cache.

Parameters

None	

Returns

None.

void Xil_L1DCacheEnable( )

Enable the L1 data cache.

Returns

None.

void Xil_L1DCacheDisable( )

Disable the L1 data cache.

Returns

None.

Note

This is processor specific.

void Xil_L1ICacheEnable( )

Enable the instruction cache.

Returns

None.

Note

This is processor specific.

void Xil_L1ICacheDisable( )

Disable the L1 Instruction cache.

Returns

None.

Note

This is processor specific.

void Xil_DCacheEnable( )

Enable the data cache.

Parameters

None	

Returns

None.

void Xil_ICacheEnable( )

Enable the instruction cache.

Parameters

None	

Returns

None.

Note

Function Documentation

void Xil_DCacheDisable (void)

Disable the data cache.

Parameters

None	

Returns

None.

void Xil_ICacheDisable (void)

Disable the instruction cache.

Parameters

None	

Returns

None.

MicroBlaze Processor FSL Macros

Overview

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

Macros

- #define getfslx(val, id, flags)
- #define putfslx(val, id, flags)
- #define tgetfslx(val, id, flags)
- #define tputfslx(id, flags)
- #define getdfslx(val, var, flags)
- #define putdfslx(val, var, flags)
- #define tgetdfslx(val, var, flags)
- #define tputdfslx(var, flags)

Macro Definition Documentation

#define getfslx(*val, id, flags*)

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

Parameters

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(*val, id, flags*)

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

Parameters

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(*val, id, flags*)

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

Parameters

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(*id, flags*)

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

Parameters

id	FSL identifier
flags	valid FSL macro flags

#define getdfslx(*val, var, flags*)

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

Parameters

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(*val, var, flags*)

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

Parameters

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(*val, var, flags*)

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

Parameters

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(*var, flags*)

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

Parameters

var	FSL identifier
flags	valid FSL macro flags

Microblaze PVR access routines and macros

Overview

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.

Macros

- #define MICROBLAZE_PVR_IS_FULL(_pvr)
- #define MICROBLAZE_PVR_USE_BARREL(_pvr)
- #define MICROBLAZE_PVR_USE_DIV(_pvr)
- #define MICROBLAZE_PVR_USE_HW_MUL(_pvr)
- #define MICROBLAZE_PVR_USE_FPU(_pvr)
- #define MICROBLAZE_PVR_USE_ICACHE(_pvr)
- #define MICROBLAZE_PVR_USE_DCACHE(_pvr)
- #define MICROBLAZE_PVR_MICROBLAZE_VERSION(_pvr)
- #define MICROBLAZE_PVR_USER1(_pvr)
- #define MICROBLAZE_PVR_USER2(_pvr)
- #define MICROBLAZE_PVR_D_LMB(_pvr)
- #define MICROBLAZE_PVR_D_PLB(_pvr)
- #define MICROBLAZE_PVR_I_LMB(_pvr)
- #define MICROBLAZE_PVR_I_PLB(_pvr)
- #define MICROBLAZE_PVR_INTERRUPT_IS_EDGE(_pvr)
- #define MICROBLAZE_PVR_EDGE_IS_POSITIVE(_pvr)
- #define MICROBLAZE_PVR_INTERCONNECT(_pvr)
- #define MICROBLAZE_PVR_USE_MUL64(_pvr)
- #define MICROBLAZE_PVR_OPCODE_0x0_ILLEGAL(_pvr)

- #define MICROBLAZE_PVR_UNALIGNED_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_ILL_OPCODE_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_IPLB_BUS_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_DPLB_BUS_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_DIV_ZERO_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_FPU_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_FSL_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_DEBUG_ENABLED(_pvr)
- #define MICROBLAZE_PVR_NUMBER_OF_PC_BRK(_pvr)
- #define MICROBLAZE_PVR_NUMBER_OF_RD_ADDR_BRK(_pvr)
- #define MICROBLAZE_PVR_NUMBER_OF_WR_ADDR_BRK(_pvr)
- #define MICROBLAZE_PVR_FSL_LINKS(_pvr)
- #define MICROBLAZE_PVR_ICACHE_ADDR_TAG_BITS(_pvr)
- #define MICROBLAZE_PVR_ICACHE_ALLOW_WR(_pvr)
- #define MICROBLAZE_PVR_ICACHE_LINE_LEN(_pvr)
- #define MICROBLAZE_PVR_ICACHE_BYTE_SIZE(_pvr)
- #define MICROBLAZE_PVR_DCACHE_ADDR_TAG_BITS(_pvr)
- #define MICROBLAZE_PVR_DCACHE_ALLOW_WR(_pvr)
- #define MICROBLAZE_PVR_DCACHE_LINE_LEN(_pvr)
- #define MICROBLAZE_PVR_DCACHE_BYTE_SIZE(_pvr)
- #define MICROBLAZE_PVR_ICACHE_BASEADDR(_pvr)
- #define MICROBLAZE_PVR_ICACHE_HIGHADDR(_pvr)
- #define MICROBLAZE_PVR_DCACHE_BASEADDR(_pvr)
- #define MICROBLAZE_PVR_DCACHE_HIGHADDR(_pvr)
- #define MICROBLAZE_PVR_TARGET_FAMILY(_pvr)
- #define MICROBLAZE_PVR_MSR_RESET_VALUE(_pvr)
- #define MICROBLAZE_PVR_MMU_TYPE(_pvr)

Functions

- int microblaze_get_pvr (pvr_t *pvr)

Macro Definition Documentation

#define MICROBLAZE_PVR_IS_FULL(*_pvr*)

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

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USE_BARREL(*_pvr*)

Return non-zero integer if hardware barrel shifter present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USE_DIV(*_pvr*)

Return non-zero integer if hardware divider present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USE_HW_MUL(*_pvr*)

Return non-zero integer if hardware multiplier present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USE_FPU(*_pvr*)

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

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USE_ICACHE(*_pvr*)

Return non-zero integer if I-cache present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USE_DCACHE(*_pvr*)

Return non-zero integer if D-cache present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_MICROBLAZE_VERSION(*_pvr*)

Return MicroBlaze processor version encoding. Refer to the MicroBlaze Processor Reference Guide (UG081) for mappings from encodings to actual hardware versions.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USER1(*_pvr*)

Return the USER1 field stored in the PVR.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USER2(*_pvr*)

Return the USER2 field stored in the PVR.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_D_LMB(*_pvr*)

Return non-zero integer if Data Side PLB interface is present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_D_PLB(*_pvr*)

Return non-zero integer if Data Side PLB interface is present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_I_LMB(*_pvr*)

Return non-zero integer if Instruction Side Local Memory Bus (LMB) interface present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_I_PLB(*_pvr*)

Return non-zero integer if Instruction Side PLB interface present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_INTERRUPT_IS_EDGE(*_pvr*)

Return non-zero integer if interrupts are configured as edge-triggered.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_EDGE_IS_POSITIVE(*_pvr*)

Return non-zero integer if interrupts are configured as positive edge triggered.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_INTERCONNECT(*_pvr*)

Return non-zero if MicroBlaze processor has PLB interconnect; otherwise return zero.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USE_MUL64(*_pvr*)

Return non-zero integer if MicroBlaze processor supports 64-bit products for multiplies.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_OPCODE_0x0_ILLEGAL(*_pvr*)

Return non-zero integer if opcode 0x0 is treated as an illegal opcode. multiplies.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_UNALIGNED_EXCEPTION(*_pvr*)

Return non-zero integer if unaligned exceptions are supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_ILL_OPCODE_EXCEPTION(*_pvr*)

Return non-zero integer if illegal opcode exceptions are supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_IPLB_BUS_EXCEPTION(*_pvr*)

Return non-zero integer if I-PLB exceptions are supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DPLB_BUS_EXCEPTION(*_pvr*)

Return non-zero integer if I-PLB exceptions are supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DIV_ZERO_EXCEPTION(*_pvr*)

Return non-zero integer if divide by zero exceptions are supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_FPU_EXCEPTION(*_pvr*)

Return non-zero integer if FPU exceptions are supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_FSL_EXCEPTION(*_pvr*)

Return non-zero integer if FSL exceptions are present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DEBUG_ENABLED(*_pvr*)

Return non-zero integer if debug is enabled.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_NUMBER_OF_PC_BRK(*_pvr*)

Return the number of hardware PC breakpoints available.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_NUMBER_OF_RD_ADDR_BRK(*_pvr*)

Return the number of read address hardware watchpoints supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_NUMBER_OF_WR_ADDR_BRK(*_pvr*)

Return the number of write address hardware watchpoints supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_FSL_LINKS(*_pvr*)

Return the number of FSL links present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_ICACHE_ADDR_TAG_BITS(*_pvr*)

Return the number of address tag bits for the I-cache.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_ICACHE_ALLOW_WR(*_pvr*)

Return non-zero if writes to I-caches are allowed.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_ICACHE_LINE_LEN(*_pvr*)

Return the length of each I-cache line in bytes.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_ICACHE_BYTE_SIZE(*_pvr*)

Return the size of the D-cache in bytes.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DCACHE_ADDR_TAG_BITS(*_pvr*)

Return the number of address tag bits for the D-cache.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DCACHE_ALLOW_WR(*_pvr*)

Return non-zero if writes to D-cache are allowed.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DCACHE_LINE_LEN(*_pvr*)

Return the length of each line in the D-cache in bytes.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DCACHE_BYTE_SIZE(*_pvr*)

Return the size of the D-cache in bytes.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_ICACHE_BASEADDR(*_pvr*)

Return the base address of the I-cache.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_ICACHE_HIGHADDR(*_pvr*)

Return the high address of the I-cache.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DCACHE_BASEADDR(*_pvr*)

Return the base address of the D-cache.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DCACHE_HIGHADDR(*_pvr*)

Return the high address of the D-cache.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_TARGET_FAMILY(*_pvr*)

Return the encoded target family identifier.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_MSR_RESET_VALUE(*_pvr*)

Refer to the MicroBlaze Processor Reference Guide (UG081) for mappings from encodings to target family name strings.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_MMU_TYPE(*_pvr*)

Returns the value of C_USE_MMU. Refer to the MicroBlaze Processor Reference Guide (UG081) for mappings from MMU type values to MMU function.

Parameters

_pvr	pvr data structure

Function Documentation

int microblaze_get_pvr (pvr_t * *pvr*)

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

Parameters

pvr-	address of PVR data structure to be populated

Returns

0 - SUCCESS -1 - FAILURE

Sleep Routines for Microblaze

Overview

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

Note

microblaze_sleep.h may contain architecture-dependent items.

Functions

- void MB_Sleep (u32 MilliSeconds) __attribute__((__deprecated__))

Function Documentation

void MB_Sleep (u32 *MilliSeconds*)

Provides delay for requested duration..

Parameters

MilliSeconds-	Delay time in milliseconds.

Returns

None.

Note

Instruction cache should be enabled for this to work.

Chapter 5

Cortex R5 Processor API

Overview

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

Modules

- Cortex R5 Processor Boot Code
- Cortex R5 Processor MPU specific APIs
- Cortex R5 Processor Cache Functions
- Cortex R5 Time Functions
- Cortex R5 Event Counters Functions
- Cortex R5 Processor Specific Include Files

Cortex R5 Processor Boot Code

Overview

The `boot.S` file contains a minimal set of code for transferring control from the processor's 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's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

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

7. Enable Floating point unit
8. Transfer control to _start which clears BSS sections and jumping to main application

Cortex R5 Processor MPU specific APIs

Overview

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	Note
DDR	0x00000000 - 0x7FFFFFFF	Normal write-back Cacheable	For a system where DDR is less than 2GB, region after DDR and before PL is marked as undefined in translation table
PL	0x80000000 - 0xBFFFFFFF	Strongly Ordered	
QSPI	0xC0000000 - 0xDFFFFFFF	Device Memory	
PCIe	0xE0000000 - 0xEFFFFFFF	Device Memory	
STM_CORESIGHT	0xF8000000 - 0xF8FFFFFF	Device Memory	

	Memory Range	Attributes of MPURegion	Note
RPU_R5_GIC	0xF9000000 - 0xF90FFFFF	Device Memory	
FPS	0xFD000000 - 0xFDFFFFFF	Device Memory	
LPS	0xFE000000 - 0xFFFFFFFF	Device Memory	0xFE000000 - 0xFEFFFFFF upper LPS slaves, 0xFF000000 - 0xFFFFFFFF lower LPS slaves
OCM	0xFFFC0000 - 0xFFFFFFFF	Normal write-back Cacheable	

Functions

- void Xil_SetTlbAttributes (INTPTR Addr, u32 attrib)
- void Xil_EnableMPU (void)
- void Xil_DisableMPU (void)
- void Xil_SetMPURegion (INTPTR addr, u64 size, u32 attrib)

Function Documentation

void Xil_SetTlbAttributes (INTPTR *addr,* u32 *attrib*)

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

Parameters

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

Returns

None.

void Xil_EnableMPU (void)

Enable MPU for Cortex R5 processor. This function invalidates I cache and flush the D Caches, and then enables the MPU.

Parameters

None.	

Returns

None.

void Xil_DisableMPU (void)

Disable MPU for Cortex R5 processors. This function invalidates I cache and flush the D Caches, and then disabes the MPU.

Parameters

None.	

Returns

None.

void Xil_SetMPURegion (INTPTR *addr,* u64 *size,* u32 *attrib*)

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

Parameters

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.

Cortex R5 Processor Cache Functions

Overview

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.

Functions

- 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)

Function Documentation

void Xil_DCacheEnable (void)

Enable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheDisable (void)

Disable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheInvalidate (void)

Invalidate the entire Data cache.

Parameters

None.	

Returns

None.

void Xil_DCacheInvalidateRange (INTPTR *adr,* u32 *len*)

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

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

Returns

None.

void Xil_DCacheFlush (void)

Flush the entire Data cache.

Parameters

None.	

Returns

None.

void Xil_DCacheFlushRange (INTPTR *adr,* u32 *len*)

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.

Parameters

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

Returns

None.

void Xil_DCacheInvalidateLine (INTPTR *adr*)

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.

Parameters

adr	32bit address of the data to be flushed.

Returns

None.

Note

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

void Xil_DCacheFlushLine (INTPTR *adr*)

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.

Parameters

adr	32bit address of the data to be flushed.

Returns

None.

Note

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

void Xil_DCacheStoreLine (INTPTR *adr*)

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).

Parameters

adr	32bit address of the data to be stored

Returns

None.

Note

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

void Xil_ICacheEnable (void)

Enable the instruction cache.

Parameters

None.	

Returns

None.

void Xil_ICacheDisable (void)

Disable the instruction cache.

Parameters

None.	

Returns

None.

void Xil_ICacheInvalidate (void)

Invalidate the entire instruction cache.

Parameters

None.	

Returns

None.

void Xil_ICacheInvalidateRange (INTPTR *adr,* u32 *len*)

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.

Parameters

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

Returns

None.

void Xil_ICacheInvalidateLine (INTPTR *adr*)

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.

Parameters

adr	32bit address of the instruction to be invalidated.

Returns

None.

Note

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

Cortex R5 Time Functions

Overview

The `xtime_l.c` file and corresponding `xtime_l.h` include file provide access to the 32-bit counter in TTC. The `sleep.c`, `usleep.c` file and the corresponding `sleep.h` include file implement sleep functions. Sleep functions are implemented as busy loops.

Functions

- void XTime_StartTimer (void)
- void XTime_SetTime (XTime Xtime_Global)
- void XTime_GetTime (XTime ∗Xtime_Global)

Function Documentation

void XTime_StartTimer (void)

Starts the TTC timer 3 counter 0 if present and if it is not already running with desired parameters for sleep functionalities.

Parameters

None.	

Returns

None.

Note

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

void XTime_SetTime (XTime *Xtime_Global*)

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.

Parameters

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

Returns

None.

Note

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

void XTime_GetTime (XTime ∗ *Xtime_Global*)

Get the time from the timer counter register.

Parameters

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

Returns

None.

Cortex R5 Event Counters Functions

Overview

Cortex R5 event counter functions can be utilized to configure and control the Cortex-R5 performance monitor events. Cortex-R5 Performance Monitor has 6 event counters which can be used to count a variety of events described in Coretx-R5 TRM. xpm_counter.h defines configurations XPM_CNTRCFGx which can be used to program the event counters to count a set of events.

Note

It doesn't handle the Cortex-R5 cycle counter, as the cycle counter is being used for time keeping.

Functions

- void Xpm_SetEvents (s32 PmcrCfg)
- void Xpm_GetEventCounters (u32 ∗PmCtrValue)

Function Documentation

void Xpm_SetEvents (s32 *PmcrCfg*)

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

Parameters

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.

void Xpm_GetEventCounters (u32 ∗ *PmCtrValue*)

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

Parameters

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.

Cortex R5 Processor Specific Include Files

Overview

The `xpseudo_asm.h` file includes `xreg_cortexr5.h` and `xpseudo_asm_gcc.h`.
The `xreg_cortexr5.h` include file contains the register numbers and the register bits for the ARM Cortex-R5 processor.
The `xpseudo_asm_gcc.h` file 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.

Chapter 6

ARM Processor Common API

Overview

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

Modules

- ARM Processor Exception Handling

ARM Processor Exception Handling

Overview

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

Macros

- #define Xil_ExceptionEnableMask(Mask)
- #define Xil_ExceptionEnable()
- #define Xil_ExceptionDisableMask(Mask)
- #define Xil_ExceptionDisable()
- #define Xil_EnableNestedInterrupts()
- #define Xil_DisableNestedInterrupts()

Typedefs

- typedef void(∗ Xil_ExceptionHandler) (void ∗data)

Functions

- void Xil_ExceptionRegisterHandler (u32 Exception_id, Xil_ExceptionHandler Handler, void ∗Data)
- void Xil_ExceptionRemoveHandler (u32 Exception_id)

- void Xil_ExceptionInit (void)
- void Xil_DataAbortHandler (void *CallBackRef)
- void Xil_PrefetchAbortHandler (void *CallBackRef)
- void Xil_UndefinedExceptionHandler (void *CallBackRef)

Macro Definition Documentation

#define Xil_ExceptionEnableMask(*Mask*)

Enable Exceptions.

Parameters

Mask	for exceptions to be enabled.

Returns

None.

Note

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

#define Xil_ExceptionEnable( )

Enable the IRQ exception.

Returns

None.

Note

None.

#define Xil_ExceptionDisableMask(*Mask*)

Disable Exceptions.

Parameters

Mask	for exceptions to be enabled.

Returns

None.

Note

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

#define Xil_ExceptionDisable( )

Disable the IRQ exception.

Returns

None.

Note

None.

#define Xil_EnableNestedInterrupts( )

Enable nested interrupts by clearing the I and F bits in CPSR. This API is defined for cortex-a9 and cortex-r5.

Returns

None.

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( )

Disable the nested interrupts by setting the I and F bits. This API is defined for cortex-a9 and cortex-r5.

Returns

None.

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.

Typedef Documentation

typedef void(∗ Xil_ExceptionHandler) (void ∗data)

This typedef is the exception handler function.

Function Documentation

void Xil_ExceptionRegisterHandler (u32 *Exception_id,* Xil_ExceptionHandler *Handler,* void ∗ *Data*)

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

Parameters

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

Returns

None.

Note

None.

void Xil_ExceptionRemoveHandler (u32 *Exception_id*)

Removes the Handler for a specific exception Id. The stub Handler is then registered for this exception Id.

Parameters

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.

Note

None.

void Xil_ExceptionInit (void)

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).

Parameters

None.	

Returns

None.

Note

None.

void Xil_DataAbortHandler (void * *CallBackRef*)

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

Parameters

None	

Returns

None.

Note

None.

void Xil_PrefetchAbortHandler (void * *CallBackRef*)

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

Parameters

None	

Returns

None.

Note

None.

void Xil_UndefinedExceptionHandler (void * *CallBackRef*)

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

Parameters

None	

Returns

None.

Note

None.

Chapter 7

Cortex A9 Processor API

Overview

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

Modules

- Cortex A9 Processor Boot Code
- Cortex A9 Processor Cache Functions
- Cortex A9 Processor MMU Functions
- Cortex A9 Time Functions
- Cortex A9 Event Counter Function
- PL310 L2 Event Counters Functions
- Cortex A9 Processor and pl310 Errata Support
- Cortex A9 Processor Specific Include Files

Cortex A9 Processor Boot Code

Overview

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's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling

2. Invalidate instruction cache, data cache and TLBs

3. Program stack pointer for various modes (IRQ, FIQ, supervisor, undefine, abort, system)

4. Configure MMU with short descriptor translation table format and program base address of translation table

5. Enable data cache, instruction cache and MMU

6. Enable Floating point unit
7. Transfer control to _start which clears BSS sections, initializes global timer and runs global constructor before jumping to main application

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

	Memory Range	**Definition in Translation Table**	**Note**
DDR	0x00000000 - 0x3FFFFFFF	Normal write-back Cacheable	For a system where DDR is less than 1GB, region after DDR and before PL is marked as undefined/reserved in translation table
PL	0x40000000 - 0xBFFFFFFF	Strongly Ordered	
Reserved	0xC0000000 - 0xDFFFFFFF	Unassigned	
Memory mapped devices	0xE0000000 - 0xE02FFFFF	Device Memory	
Reserved	0xE0300000 - 0xE0FFFFFF	Unassigned	
NAND, NOR	0xE1000000 - 0xE3FFFFFF	Device memory	
SRAM	0xE4000000 - 0xE5FFFFFF	Normal write-back Cacheable	
Reserved	0xE6000000 - 0xF7FFFFFF	Unassigned	
AMBA APB Peripherals	0xF8000000 - 0xF8FFFFFF	Device Memory	0xF8000C00 - 0xF8000FFF, 0xF8010000 -0xF88FFFFF and 0xF8F03000 to 0xF8FFFFFF are reserved but due to granual size of 1MB, it is not possible to define separate regions for them

	Memory Range	Definition in Translation Table	Note
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	0xFFF00000 to 0xFFFB0000 is reserved but due to 1MB granual size, it is not possible to define separate region for it

Cortex A9 Processor Cache Functions

Overview

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.

Functions

- 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)
- 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)
- 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)

Function Documentation

void Xil_DCacheEnable (void)

Enable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheDisable (void)

Disable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheInvalidate (void)

Invalidate the entire Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheInvalidateRange (INTPTR *adr,* u32 *len*)

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, loosing 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, follwoing 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.

Parameters

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

Returns

None.

Note

None.

void Xil_DCacheFlush (void)

Flush the entire Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheFlushRange (INTPTR *adr,* u32 *len*)

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.

Parameters

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

Returns

None.

Note

None.

void Xil_ICacheEnable (void)

Enable the instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheDisable (void)

Disable the instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheInvalidate (void)

Invalidate the entire instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheInvalidateRange (INTPTR *adr,* u32 *len*)

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.

Parameters

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

Returns

None.

Note

None.

void Xil_DCacheInvalidateLine (u32 *adr*)

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.

Parameters

adr	32bit address of the data to be flushed.

Returns

None.

Note

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

void Xil_DCacheFlushLine (u32 *adr*)

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.

Parameters

adr	32bit address of the data to be flushed.

Returns

None.

Note

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

void Xil_DCacheStoreLine (u32 *adr*)

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).

Parameters

adr	32bit address of the data to be stored.

Returns

None.

Note

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

void Xil_ICacheInvalidateLine (u32 *adr*)

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.

Parameters

adr	32bit address of the instruction to be invalidated.

Returns

None.

Note

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

void Xil_L1DCacheEnable (void)

Enable the level 1 Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L1DCacheDisable (void)

Disable the level 1 Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L1DCacheInvalidate (void)

Invalidate the level 1 Data cache.

Parameters

None.	

Returns

None.

Note

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

void Xil_L1DCacheInvalidateLine (u32 *adr*)

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.

Parameters

adr	32bit address of the data to be invalidated.

Returns

None.

Note

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

void Xil_L1DCacheInvalidateRange (u32 *adr,* u32 *len*)

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.

Parameters

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

Returns

None.

Note

None.

void Xil_L1DCacheFlush (void)

Flush the level 1 Data cache.

Parameters

None.	

Returns

None.

Note

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

void Xil_L1DCacheFlushLine (u32 *adr*)

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.

Parameters

adr	32bit address of the data to be flushed.

Returns

None.

Note

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

void Xil_L1DCacheFlushRange (u32 *adr,* u32 *len*)

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.

Parameters

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

Returns

None.

Note

None.

void Xil_L1DCacheStoreLine (u32 *adr*)

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).

Parameters

Address	to be stored.

Returns

None.

Note

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

void Xil_L1ICacheEnable (void)

Enable the level 1 instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L1ICacheDisable (void)

Disable level 1 the instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L1ICacheInvalidate (void)

Invalidate the entire level 1 instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L1ICacheInvalidateLine (u32 *adr*)

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.

Parameters

adr	32bit address of the instruction to be invalidated.

Returns

None.

Note

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

void Xil_L1ICacheInvalidateRange (u32 *adr,* u32 *len*)

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.

Parameters

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

Returns

None.

Note

None.

void Xil_L2CacheEnable (void)

Enable the L2 cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L2CacheDisable (void)

Disable the L2 cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L2CacheInvalidate (void)

Invalidate the entire level 2 cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L2CacheInvalidateLine (u32 *adr*)

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.

Parameters

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

Returns

None.

Note

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

void Xil_L2CacheInvalidateRange (u32 *adr,* u32 *len*)

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.

Parameters

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

Returns

None.

Note

None.

void Xil_L2CacheFlush (void)

Flush the entire level 2 cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L2CacheFlushLine (u32 *adr*)

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.

Parameters

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

Returns

None.

Note

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

void Xil_L2CacheFlushRange (u32 *adr,* u32 *len*)

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.

Parameters

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

Returns

None.

Note

None.

void Xil_L2CacheStoreLine (u32 *adr*)

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).

Parameters

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

Returns

None.

Note

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

Cortex A9 Processor MMU Functions

Overview

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

Functions

- void Xil_SetTlbAttributes (INTPTR Addr, u32 attrib)
- void Xil_EnableMMU (void)
- void Xil_DisableMMU (void)

Function Documentation

void Xil_SetTlbAttributes (INTPTR *Addr,* u32 *attrib*)

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

Parameters

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.

Note

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

void Xil_EnableMMU (void)

Enable MMU for cortex A9 processor. This function invalidates the instruction and data caches, and then enables MMU.

Parameters

None.	

Returns

None.

void Xil_DisableMMU (void)

Disable MMU for Cortex A9 processors. This function invalidates the TLBs, Branch Predictor Array and flushed the D Caches before disabling the MMU.

Parameters

None.	

Returns

None.

Note

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

Cortex A9 Time Functions

Overview

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.

Functions

- void XTime_SetTime (XTime Xtime_Global)
- void XTime_GetTime (XTime ∗Xtime_Global)

Function Documentation

void XTime_SetTime (XTime *Xtime_Global*)

Set the time in the Global Timer Counter Register.

Parameters

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

Returns

None.

Note

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

void XTime_GetTime (XTime ∗ *Xtime_Global*)

Get the time from the Global Timer Counter Register.

Parameters

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

Returns

None.

Note

None.

Cortex A9 Event Counter Function

Overview

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

Note

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

Functions

- void Xpm_SetEvents (s32 PmcrCfg)
- void Xpm_GetEventCounters (u32 *PmCtrValue)

Function Documentation

void Xpm_SetEvents (s32 *PmcrCfg*)

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

Parameters

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.

Note

None.

void Xpm_GetEventCounters (u32 * *PmCtrValue*)

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

Parameters

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.

Note

None.

PL310 L2 Event Counters Functions

Overview

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.

Functions

- void XL2cc_EventCtrInit (s32 Event0, s32 Event1)
- void XL2cc_EventCtrStart (void)
- void XL2cc_EventCtrStop (u32 *EveCtr0, u32 *EveCtr1)

Function Documentation

void XL2cc_EventCtrInit (s32 *Event0*, s32 *Event1*)

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

Parameters

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

Returns

None.

Note

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

void XL2cc_EventCtrStart (void)

This function starts the event counters in L2 Cache controller.

Parameters

None.	

Returns

None.

Note

None.

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

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

Parameters

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.

Note

None.

Cortex A9 Processor and pl310 Errata Support

Overview

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.

errata_definitions

The errata conditions handled in the standalone BSP are listed below

- #define **ENABLE_ARM_ERRATA**
- #define CONFIG_ARM_ERRATA_742230
- #define CONFIG_ARM_ERRATA_743622
- #define CONFIG_ARM_ERRATA_775420
- #define CONFIG_ARM_ERRATA_794073
- #define CONFIG_PL310_ERRATA_588369
- #define CONFIG_PL310_ERRATA_727915
- #define CONFIG_PL310_ERRATA_753970

Macro Definition Documentation

#define CONFIG_ARM_ERRATA_742230

Errata No: 742230 Description: DMB operation may be faulty

#define CONFIG_ARM_ERRATA_743622

Errata No: 743622 Description: Faulty hazard checking in the Store Buffer may lead to data corruption.

#define CONFIG_ARM_ERRATA_775420

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

#define CONFIG_ARM_ERRATA_794073

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

#define CONFIG_PL310_ERRATA_588369

PL310 L2 Cache Errata Errata No: 588369 Description: Clean & Invalidate maintenance operations do not invalidate clean lines

#define CONFIG_PL310_ERRATA_727915

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

#define CONFIG_PL310_ERRATA_753970

Errata No: 753970 Description: Cache sync operation may be faulty

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.

Chapter 8

Cortex A53 32-bit Processor API

Overview

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.

Modules

- Cortex A53 32-bit Processor Boot Code
- Cortex A53 32-bit Processor Cache Functions
- Cortex A53 32-bit Processor MMU Handling
- Cortex A53 32-bit Mode Time Functions
- Cortex A53 32-bit Processor Specific Include Files

Cortex A53 32-bit Processor Boot Code

Overview

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's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

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

The `translation_table.S` file 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. The overview of translation table memory attributes is described below.

	Memory Range	**Definition in Translation Table**	**Note**
DDR	0x00000000 - 0x7FFFFFFF	Normal write-back Cacheable	For a system where DDR is less than 2GB, region after DDR and before PL is marked as undefined/reserved in translation table
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	This region contains CSU and PMU memory which are marked as Device since it is less than 1MB and falls in a region with device memory
TCM, OCM	0xFFE00000 - 0xFFFFFFFF	Normal write-back cacheable	

Cortex A53 32-bit Processor Cache Functions

Overview

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.

Functions

- void Xil_DCacheEnable (void)
- void Xil_DCacheDisable (void)
- void Xil_DCacheInvalidate (void)
- void Xil_DCacheInvalidateRange (INTPTR adr, u32 len)
- void Xil_DCacheInvalidateLine (u32 adr)
- void Xil_DCacheFlush (void)
- void Xil_DCacheFlushLine (u32 adr)
- void Xil_ICacheEnable (void)
- void Xil_ICacheDisable (void)
- void Xil_ICacheInvalidate (void)
- void Xil_ICacheInvalidateRange (INTPTR adr, u32 len)
- void Xil_ICacheInvalidateLine (u32 adr)

Function Documentation

void Xil_DCacheEnable (void)

Enable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheDisable (void)

Disable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheInvalidate (void)

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

Parameters

None.	

Returns

None.

Note

In Cortex-A53, functionality to simply invalide 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.

void Xil_DCacheInvalidateRange (INTPTR *adr*, u32 *len*)

Invalidate the Data cache for the given address range. The cachelines present in the adderss range are cleaned and invalidated.

Parameters

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

Returns

None.

Note

In Cortex-A53, functionality to simply invalide 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.

void Xil_DCacheInvalidateLine (u32 *adr*)

Invalidate a Data cache line. The cacheline is cleaned and invalidated.

Parameters

adr	32 bit address of the data to be invalidated.

Returns

None.

Note

In Cortex-A53, functionality to simply invalide 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.

void Xil_DCacheFlush (void)

Flush the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheFlushLine (u32 *adr*)

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.

Parameters

adr	32bit address of the data to be flushed.

Returns

None.

Note

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

void Xil_ICacheEnable (void)

Enable the instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheDisable (void)

Disable the instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheInvalidate (void)

Invalidate the entire instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheInvalidateRange (INTPTR *adr,* u32 *len*)

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.

Parameters

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

Returns

None.

Note

None.

void Xil_ICacheInvalidateLine (u32 *adr*)

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.

Parameters

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

Returns

None.

Note

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

Cortex A53 32-bit Processor MMU Handling

Overview

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

Functions

- void Xil_SetTlbAttributes (INTPTR Addr, u32 attrib)
- void Xil_EnableMMU (void)
- void Xil_DisableMMU (void)

Function Documentation

void Xil_SetTlbAttributes (INTPTR *Addr*, u32 *attrib*)

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

Parameters

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.

Note

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

void Xil_EnableMMU (void)

Enable MMU for Cortex-A53 processor in 32bit mode. This function invalidates the instruction and data caches before enabling MMU.

Parameters

None.	

Returns

None.

void Xil_DisableMMU (void)

Disable MMU for Cortex A53 processors in 32bit mode. This function invalidates the TLBs, Branch Predictor Array and flushed the data cache before disabling the MMU.

Parameters

None.	

Returns

None.

Note

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

Cortex A53 32-bit Mode Time Functions

Overview

The `xtime_l.c` file and corresponding `xtime_l.h` include file provide access to the 64-bit generic counter in Cortex-A53. The `sleep.c`, `usleep.c` file and the corresponding `sleep.h` include file implement sleep functions. Sleep functions are implemented as busy loops.

Functions

- void XTime_StartTimer (void)
- void XTime_SetTime (XTime Xtime_Global)
- void XTime_GetTime (XTime ∗Xtime_Global)

Function Documentation

void XTime_StartTimer (void)

Start the 64-bit physical timer counter.

Parameters

None.	

Returns

None.

Note

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

void XTime_SetTime (XTime *Xtime_Global*)

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.

Parameters

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

Returns

None.

Note

None.

void XTime_GetTime (XTime * *Xtime_Global*)

Get the time from the physical timer counter register.

Parameters

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

Returns

None.

Note

None.

Cortex A53 32-bit Processor Specific Include Files

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. These inline assembler instructions can be used from drivers and user applications written in C.

Cortex A53 64-bit Processor API

Overview

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

Modules

- Cortex A53 64-bit Processor Boot Code
- Cortex A53 64-bit Processor Cache Functions
- Cortex A53 64-bit Processor MMU Handling
- Cortex A53 64-bit Mode Time Functions
- Cortex A53 64-bit Processor Specific Include Files

Cortex A53 64-bit Processor Boot Code

Overview

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's reset state. Cortex-A53 starts execution from EL3 and currently application is also run from EL3. 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. 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

Cortex A53 64-bit Processor Cache Functions

Overview

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.

Functions

- 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)
- void Xil_ICacheInvalidateLine (INTPTR adr)

Function Documentation

void Xil_DCacheEnable (void)

Enable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheDisable (void)

Disable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheInvalidate (void)

Invalidate the Data cache. The contents present in the cache are cleaned and invalidated.

Parameters

None.	

Returns

None.

Note

In Cortex-A53, functionality to simply invalide 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.

void Xil_DCacheInvalidateRange (INTPTR *adr*, INTPTR *len*)

Invalidate the Data cache for the given address range. The cachelines present in the adderss range are cleaned and invalidated.

Parameters

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

Returns

None.

Note

In Cortex-A53, functionality to simply invalide 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.

void Xil_DCacheInvalidateLine (INTPTR *adr*)

Invalidate a Data cache line. The cacheline is cleaned and invalidated.

Parameters

adr	64bit address of the data to be flushed.

Returns

None.

Note

In Cortex-A53, functionality to simply invalide 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.

void Xil_DCacheFlush (void)

Flush the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheFlushLine (INTPTR *adr*)

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.

Parameters

adr	64bit address of the data to be flushed.

Returns

None.

Note

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

void Xil_ICacheEnable (void)

Enable the instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheDisable (void)

Disable the instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheInvalidate (void)

Invalidate the entire instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheInvalidateRange (INTPTR *adr,* INTPTR *len*)

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.

Parameters

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

Returns

None.

Note

None.

void Xil_ICacheInvalidateLine (INTPTR *adr*)

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.

Parameters

adr	64bit address of the instruction to be invalidated.

Returns

None.

Note

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

Cortex A53 64-bit Processor MMU Handling

Overview

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

Functions

- void Xil_SetTlbAttributes (INTPTR Addr, u64 attrib)

Function Documentation

void Xil_SetTlbAttributes (INTPTR *Addr,* u64 *attrib*)

brief It sets the memory attributes for a section, in the translation table. If the address (defined by Addr) is less than 4GB, the memory attribute(attrib) is set for a section of 2MB memory. If the address (defined by Addr) is greater than 4GB, the memory attribute (attrib) is set for a section of 1GB memory.

Parameters

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.

Note

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

Cortex A53 64-bit Mode Time Functions

Overview

The `xtime_l.c` file and corresponding `xtime_l.h` include file provide access to the 64-bit generic counter in Cortex-A53. The `sleep.c`, `usleep.c` file and the corresponding `sleep.h` include file implement sleep functions. Sleep functions are implemented as busy loops.

Functions

- void XTime_StartTimer (void)
- void XTime_SetTime (XTime Xtime_Global)
- void XTime_GetTime (XTime ∗Xtime_Global)

Function Documentation

void XTime_StartTimer (void)

Start the 64-bit physical timer counter.

Parameters

None.	

Returns

None.

Note

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

void XTime_SetTime (XTime *Xtime_Global*)

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.

Parameters

Xtime_Global	64bit value to be written to the physical timer counter register.

Returns

None.

Note

None.

void XTime_GetTime (XTime * *Xtime_Global*)

Get the time from the physical timer counter register.

Parameters

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

Returns

None.

Note

None.

Cortex A53 64-bit Processor Specific Include Files

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. These inline assembler instructions can be used from drivers and user applications written in C.

XilMFS Library v2.3

Chapter 10

Overview

The XilMFS library provides the capability to manage program memory in the form of file handles. You can create directories and have files within each directory. The file system can be accessed from the high-level C language through function calls specific to the file system.

Chapter 11

XilMFS Library API

Overview

This chapter provides a linked summary and detailed descriptions of the XilMSF library APIs.

Functions

- void mfs_init_fs (int numbytes, char ∗address, int init_type)
- void mfs_init_genimage (int numbytes, char ∗address, int init_type)
- int mfs_change_dir (const char ∗newdir)
- int mfs_delete_file (char ∗filename)
- int mfs_create_dir (char ∗newdir)
- int mfs_delete_dir (char ∗newdir)
- int mfs_rename_file (char ∗from_file, char ∗to_file)
- int mfs_exists_file (char ∗filename)
- int mfs_get_current_dir_name (char ∗dirname)
- int mfs_get_usage (int ∗num_blocks_used, int ∗num_blocks_free)
- int mfs_dir_open (const char ∗dirname)
- int mfs_dir_close (int fd)
- int mfs_dir_read (int fd, char ∗∗filename, int ∗filesize, int ∗filetype)
- int mfs_file_open (const char ∗filename, int mode)
- int mfs_file_read (int fd, char ∗buf, int buflen)
- int mfs_file_write (int fd, const char ∗buf, int buflen)
- int mfs_file_close (int fd)
- long mfs_file_lseek (int fd, long offset, int whence)
- int mfs_ls ()
- int mfs_ls_r (int recurse)
- int mfs_cat (char ∗filename)
- int mfs_copy_stdin_to_file (char ∗filename)
- int mfs_file_copy (char ∗from_file, char ∗to_file)

Function Documentation

void mfs_init_fs (int *numbytes*, char ∗ *address*, int *init_type*)

Initialize the file system.
This function must be called before any file system operations. Use `mfs_init_genimage()` instead of this function for initializing with file images generated by `mfsgen`.

Parameters

numbytes	Number of bytes allocated or reserved for this file system.
address	Starting address of the memory block. address must be word aligned (4 byte boundary).
init_type	• MFSINIT_NEW creates a new, empty file system for read/write • MFSINIT_IMAGE initializes a file system whose data has been previously loaded into memory at the base address. • MFSINIT_ROM_IMAGE initializes a Read-Only file system whose data has been previously loaded into memory at the base address.

void mfs_init_genimage (int *numbytes*, char ∗ *address*, int *init_type*)

Initialize the file system with a file image generated by `mfsgen`.
This function must be called before any file system operations. Use `mfs_init_fs()` instead of this function for other initialization.

Parameters

numbytes	Number of bytes allocated or reserved for this file sy stem.
address	Starting address of the memory block. address must be word aligned (4 byte boundary).
init_type	• MFSINIT_IMAGE initializes a file system whose data has been previously loaded into memory at the base address. • MFSINIT_ROM_IMAGE initializes a Read-Only file system whose data has been previously loaded into memory at the base address.

int mfs_change_dir (const char ∗ *newdir*)

Modify `global mfs_current_dir` to index of `newdir` if it exists.
`mfs_current_dir` is not modified otherwise.

Parameters

newdir	is the name of the new directory

Returns

1 for success and 0 for failure

int mfs_delete_file (char ∗ *filename*)

Delete a file from directory.

WARNING: *This function does not completely free up the directory space used by the file. Repeated calls to create and delete files can cause the file system to run out of space.*

Parameters

filename	Name of the file to be deleted. Delete the data blocks corresponding to the file and then delete the file entry from its directory.

Returns

1 on success, 0 on failure

Note

Delete will not work on a directory unless the directory is empty.

int mfs_create_dir (char ∗ *newdir*)

Create a new empty directory called newdir inside the current directory.

Parameters

newdir	is the name of the directory

Returns

index of new directory in the file system on success. 0 on failure

int mfs_delete_dir (char * *newdir*)

Delete the directory named newdir if it exists, and is empty.

Parameters

newdir	is the name of the directory

Returns

Index of new directory in the file system on success. 0 on failure.

int mfs_rename_file (char * *from_file,* char * *to_file*)

Rename `from_file` to `to_file`.
Rename works for directories as well as files. Function fails if to_file already exists. works for dirs as well as files cannot rename to something that already exists

Parameters

from_file	
to_file	

Returns

1 on success, 0 on failure

int mfs_exists_file (char * *filename*)

check if a file exists

Parameters

filename	is the name of the file

Returns

0 if filename is not a file in the current directory
1 if filename is a file in the current directory
2 if filename is a directory in the current directory

int mfs_get_current_dir_name (char * *dirname*)

get the name of the current directory

Parameters

dirname	= pre_allocated buffer of at least MFS_MAX_FILENAME_SIZE+1 chars The directory name is copied to this buffer

Returns

1 if success, 0 if failure

int mfs_get_usage (int * *num_blocks_used,* int * *num_blocks_free*)

get the number of used blocks and the number of free blocks in the file system through pointers

Parameters

num_blocks_used	
num_blocks_free	the return value is 1 (for success) and 0 for failure to obtain the numbers

int mfs_dir_open (const char * *dirname*)

open a directory for reading each subsequent call to mfs_dir_read() returns one directory entry until end of directory

Parameters

dirname	is the name of the directory to open

Returns

index of dir in array mfs_open_files or -1

int mfs_dir_close (int *fd*)

close a directory - same as closing a file

Parameters

fd	is the descriptor of the directory to close

Returns

1 on success, 0 otherwise

int mfs_dir_read (int *fd*, char ∗∗ *filename*, int ∗ *filesize*, int ∗ *filetype*)

read values from the next valid directory entry The last 3 parameters are output values

Parameters

fd	is the file descriptor for an open directory file
filename	is a pointer to the filename within the MFS itself
filesize	is the size in bytes for a regular file or the number of entries in a directory
filetype	is MFS_BLOCK_TYPE_FILE or MFS_BLOCK_TYPE_DIR

Returns

1 for success and 0 for failure or end of dir

int mfs_file_open (const char ∗ *filename*, int *mode*)

open a file

Parameters

filename	is the name of the file to open
mode	is MFS_MODE_READ or MFS_MODE_WRITE or MFS_MODE_CREATE this function should be used for FILEs and not DIRs no error checking (is this FILE and not DIR?) is done for MFS_MODE_READ MFS_MODE_CREATE automatically creates a FILE and not a DIR MFS_MODE_WRITE fails if the specified file is a DIR

Returns

index of file in array mfs_open_files or -1

int mfs_file_read (int *fd*, char ∗ *buf*, int *buflen*)

read characters to a file

Parameters

fd	is a descriptor for the file from which the characters are read
buf	is a pre allocated buffer that will contain the read characters
buflen	is the number of characters from buf to be read fd should be a valid index in mfs_open_files array Works only if fd points to a file and not a dir buf should be a pointer to a pre-allocated buffer of size buflen or more buflen chars are read and placed in buf if fewer than buflen chars are available then only that many chars are read

Returns

num bytes read or 0 for error=no bytes read

int mfs_file_write (int *fd,* const char ∗ *buf,* int *buflen*)

write characters to a file

Parameters

fd	is a descriptor for the file to which the characters are written
buf	is a buffer containing the characters to be written out
buflen	is the number of characters from buf to be written out fd should be a valid index in mfs_open_files array buf should be a pointer to a pre-allocated buffer of size buflen or more buflen chars are read from buf and written to 1 or more blocks of the file

Returns

1 for success or 0 for error=unable to write to file

int mfs_file_close (int *fd*)

close an open file and recover the file table entry in mfs_open_files corresponding to the fd if the fd is not valid, return 0 fd is not valid if the index in mfs_open_files is out of range, or if the corresponding entry is not an open file

Parameters

fd	is the file descriptor for the file to be closed

Returns

1 on success, 0 otherwise

long mfs_file_lseek (int *fd,* long *offset,* int *whence*)

seek to a given offset within the file

Parameters

fd	should be a valid file descriptor for an open file
whence	is one of MFS_SEEK_SET, MFS_SEEK_CUR or MFS_SEEK_END
offset	is the offset from the beginning, end or current position as specified by the whence parameter if MFS_SEEK_END is specified, the offset can be either 0 or negative otherwise offset should be positive or 0 it is an error to seek before beginning of file or after the end of file

Returns

-1 on failure, value of the offset from the beginning of the file, on success

int mfs_ls ( )

list contents of current directory

Returns

1 on success and 0 on failure

int mfs_ls_r (int *recurse*)

recursive directory listing list the contents of current directory if any of the entries in the current directory is itself a directory, immediately enter that directory and call mfs_ls_r() once again

Parameters

recurse	If parameter recurse is non zero continue recursing else stop recursing recurse=0 lists just the current directory recurse = -1 allows unlimited recursion recurse = n stops recursing at a depth of n

Returns

1 on success and 0 on failure

int mfs_cat (char ∗ *filename*)

print the file to stdout

Parameters

filename	- file to print

Returns

1 on success, 0 on failure

int mfs_copy_stdin_to_file (char ∗ *filename*)

copy from stdin to named file

Parameters

filename	- file to print

Returns

1 on success, 0 on failure

int mfs_file_copy (char ∗ *from_file,* char ∗ *to_file*)

copy from_file to to_file to_file is created new copy fails if to_file exists already copy fails is from_file or to_file cannot be opened

Parameters

from_file	
to_file	

Returns

1 on success, 0 on failure

Chapter 12

Utility Functions

This chapter provides a summary and detailed descriptions of the utility functions that can be used along with the MFS.
These functions are defined in `mfs_filesys_util.c` and are declared in `xilmfs.h`. /∗∗

Chapter 13

Library Parameters in MSS File

A memory file system can be integrated with a system using the following snippet in the Microprocessor Software Specification (MSS) file.

```
BEGIN LIBRARY
parameter LIBRARY_NAME = xilmfs
parameter LIBRARY_VER = 2.3
parameter numbytes= 50000
parameter base_address = 0xffe00000
parameter init_type = MFSINIT_NEW
parameter need_utils = false END
```

The memory file system must be instantiated with the name xilmfs. The following table lists the libgen customization parameters.

Parameter	Description
LIBRARY_NAME	Specifies the library name. Default is `xilmfs`
LIBRARY_VER	Specifies the library version. Default is 2.3
numbytes	Number of bytes allocated for file system.
base_address	Starting address for file system memory.
init_type	Options are: MFSINIT_NEW (default) creates a new, empty file system. MFSINIT_ROM_IMAGE creates a file system based on a pre-loaded memory image loaded in memory of size numbytes at starting address base_address. This memory is considered read-only and modification of the file system is not allowed. MFS_INIT_IMAGE is similar to the previous option except that the file system can be modified, and the memory is readable and writable.

Parameter	Description
need_utils	true or false (default=false) If true, this causes `stdio.h` to be included from `mfs_config.h`. The functions described in Utility Functions require that you have defined `stdin` or `stdout`. Setting the need_utils to true causes stdio.h to be included.

WARNING: *The underlying software and hardware platforms must support stdin and stdout peripherals for these utility functions to compile and link correctly.*

LwIP 2.0.2 Library v1_2

Chapter 14

Introduction

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

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

The lwip202_v1_2 is an SDK library that is built on the open source lwIP library version 2.0.2. The lwip202_v1_2 library provides adapters for the Ethernetlite (axi_ethernetlite), the TEMAC (axi_ethernet), and the Gigabit Ethernet controller and MAC (GigE) cores. The library can run on MicroBlaze™, ARM Cortex-A9, ARM Cortex-A53, and ARM Cortex-R5 processors. The Ethernetlite and TEMAC cores apply for MicroBlaze systems. The Gigabit Ethernet controller and MAC (GigE) core is applicable only for ARM Cortex-A9 system (Zynq®-7000 processor devices) and ARM Cortex-A53 & ARM Cortex-R5 system (Zynq® UltraScale+™ MPSoC).

Features

The lwIP provides support for the following protocols:

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

References

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

Chapter 15

Using lwIP

Overview

The following sections detail the hardware and software steps for using lwIP for networking. The key steps are:

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 lwip202_v1_2 to be a part of the software platform. For operating with lwIP socket API, the Xilkernel library or FreeRTOS BSP is a prerequisite. See the Note below.

Note

> The Xilkernel library is available only for MicroBlaze systems. For Cortex-A9 based systems (Zynq) and Cortex-A53 or Cortex-R5 based systems (Zynq® UltraScale™+ MPSoC), there is no support for Xilkernel. Instead, use FreeRTOS. A FreeRTOS BSP is available for Zynq systems and must be included for using lwIP socket API. The FreeRTOS BSP for Zynq is available for download from the the [FreeRTOS][freertos] website.

Setting up the Hardware System

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

- Processor: Either a MicroBlaze™ or a Cortex-A9 or a Cortex-A53 or a Cortex-R5 processor. The Cortex-A9 processor applies to Zynq systems. The Cortex-A53 and Cortex-R5 processors apply to Zynq UltraScale+ MPSoC systems.
- MAC: LwIP supports axi_ethernetlite, axi_ethernet, and Gigabit Ethernet controller and MAC (GigE) cores.
- Timer: to maintain TCP timers, lwIP raw API based applications require that certain functions are called at periodic intervals by the application. An application can do this by registering an interrupt handler with a timer.
- DMA: For axi_ethernet based systems, the axi_ethernet cores can be configured with a soft DMA engine (AXI DMA and MCDMA) or a FIFO interface. For GigE-based Zynq and Zynq UltraScale+ MPSoC systems, there is a built-in DMA and so no extra configuration is needed. Same applies to axi_ethernetlite based systems, which have their built-in buffer management provisions.

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

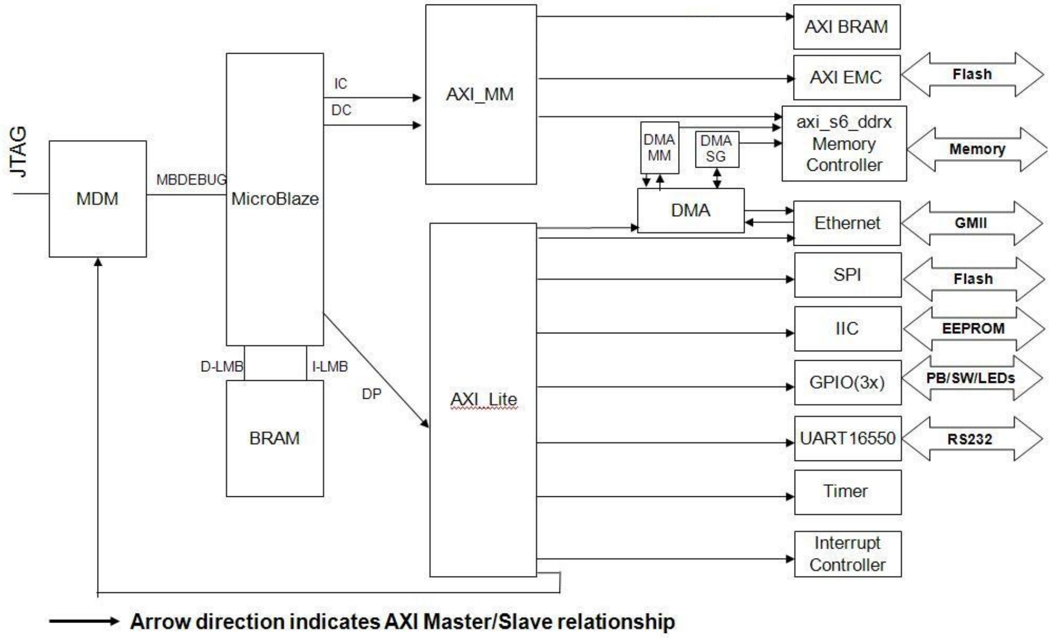

Figure 15.1: System Architecture using axi_ethernet core with DMA

Setting up the Software System

To use lwIP in a software application, you must first compile the lwIP library as a part of the software application. To move the hardware design to SDK, you must first export it from the Hardware tools.

1. Select Project > Export Hardware Design to SDK.
   The **Export to SDK** dialog box appears.

2. Click **Export & Launch SDK**.
   Vivado® exports the design to SDK. SDK opens and prompts you to create a workspace.

3. Compile the lwIP library:

   (a) Select **File** > **New** > **Xilinx Board Support Package**.
   The **New Board Support Package** wizard appears.

   (b) Specify the project name and select a location for it.

   (c) Select the BSP.
   XilKernel is not supported for Zynq and Zynq UltraScale+ MPSoC devices. FreeRTOS must be used for Zynq. The FreeRTOS BSP for Zynq is available for download from the [FreeRTOS][freertos] website. For more information, see the help documentation provided provided with the port to use the FreeRTOS BSP.

   (d) Click Finish.
   The Board Support Package Settings window opens.

(e) Select the lwip202 library with version 1_2 .
On the left side of the SDK window, lwip202_v1_2 appears in the list of libraries to be compiled.

(f) Select lwip202 in the **Project Explorer** view.
The configuration options for lwIP are listed.

(g) Configure the lwIP and click OK.
The board support package automatically builds with lwIP included in it.

Configuring lwIP Options

The lwIP library provides configurable parameters. The values for these parameters can be changed in SDK. There are two major categories of configurable options:

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

Customizing lwIP API Mode

The lwip202_v1_2 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 lwip202_v1_2 also provides the ability to set the priority on TCP/IP and other lwIP application threads.

The following table describes the lwIP library API mode options.

Attribute	Description	Type	Default
api_mode {RAW_API | SOCKET_API}	The lwIP library mode of operation	enum	RAW_API

Attribute	Description	Type	Default
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. **Note** For GigE based Zynq-7000 and Zynq UltraScale+ MPSoC systems using FreeRTOS, appropriate priority should be set. The default priority of 1 will not give the expected behaviour. For FreeRTOS (Zynq-7000 and Zynq UltraScale+ MPSoC systems), all internal lwIP tasks (except the main TCP/IP task) are created with the priority level set for this attribute. The TCP/IP task is given a higher priority than other tasks for improved performance. The typical TCP/IP task priority is 1 more than the priority set for this attribute for FreeRTOS.	integer	1

Attribute	Description	Type	Default
use_axieth_on_zynq	In the event that the AxiEthernet soft IP is used on a Zynq-7000 device or a Zynq UltraScale+ MPSoC device. This option ensures that the GigE on the Zynq-7000 PS (EmacPs) is not enabled and the device uses the AxiEthernet soft IP for Ethernet traffic. **Note** The existing Xilinx-provided lwIP adapters are not tested for multiple MACs. Multiple Axi Ethernet's are not supported on Zynq UltraScale+ MPSOC devices.	integer	0 = Use Zynq-7000 PS-based or ZynMP PS-based GigE controller 1= User AxiEthernet

Configuring Xilinx Adapter Options

The Xilinx adapters for EMAC/GigE cores are configurable.

Ethernetlite Adapter Options

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

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

TEMAC Adapter Options

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

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

Attribute	Type	Description
tcp_ip_rx_checksum_ofload	boolean	Offload TCP and IP Receive checksum calculation (hardware support required). Applicable only for AXI systems. For GigE in Zynq and Zynq UltraScale+ MPSoC devices, the TCP and IP receive checksum offloading is always present, so this attribute does not apply. Default is false.
tcp_ip_tx_checksum_ofload	boolean	Offload TCP and IP Transmit checksum calculation (hardware support required). Applicable only for AXI systems. For GigE in Zynq and Zynq UltraScale+ MPSoC devices, the TCP and IP transmit checksum offloading is always present, so this attribute does not apply. Default is false.
phy_link_speed	CONFIG_LINKSPEED_ AUTODETECT	Link speed as auto-negotiated by the PHY. lwIP configures the TEMAC/GigE for this speed setting. This setting must be correct for the TEMAC/GigE to transmit or receive packets. The CONFIG_LINKSPEED_ AUTODETECT setting attempts to detect the correct linkspeed by reading the PHY registers; however, this is PHY dependent, and has been tested with the Marvell and TI PHYs present on Xilinx development boards. For other PHYs, select the correct speed. Default is enum.

Attribute	Type	Description
temac_use_jumbo_ frames_experimental	boolean	Use TEMAC jumbo frames (with a size up to 9k bytes). If this option is selected, jumbo frames are allowed to be transmitted and received by the TEMAC. For GigE in Zynq there is no support for jumbo frames, so this attribute does not apply. Default is false.

Configuring Memory Options

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

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

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

Note

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

Configuring Packet Buffer (Pbuf) Memory Options

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

Attribute	Default	Type	Description
pbuf_pool_size	256	Integer	Number of buffers in pbuf pool. For high performance systems, you might consider increasing the pbuf pool size to a higher value, such as 512.
pbuf_pool_bufsize	1700	Integer	Size of each pbuf in pbuf pool. For systems that support jumbo frames, you might consider using a pbuf pool buffer size that is more than the maximum jumbo frame size.
pbuf_link_hlen	16	Integer	Number of bytes that should be allocated for a link level header.

Configuring ARP Options

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

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

Configuring IP Options

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

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

Configuring ICMP Options

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

Attribute	Default	Type	Description
icmp_ttl	255	Integer	ICMP TTL value.

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

Configuring IGMP Options

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

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

Configuring UDP Options

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

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

Configuring TCP Options

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

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

Attribute	Default	Type	Description
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

Chapter 16

LwIP Library APIs

Overview

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 `$XILINX_SDK/sw/ThirdParty/sw_services/` lwip202_v1_2 `/src/lwip-2.0.2/doc/rawapi.txt` file describes the lwIP Raw API.

LwIP Performance

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

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

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

RAW API Example

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

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

  /* the MAC address of the board.
   * This should be unique per board/PHY */
  unsigned char mac_ethernet_address[] =
    {0x00, 0x0a, 0x35, 0x00, 0x01, 0x02};

  lwip_init();

  /* Add network interface to the netif_list,
    * and set it as default */
  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);

  /* now enable interrupts */
  platform_enable_interrupts();

  /* specify that the network if is up */
  netif_set_up(netif);

  /* start the application, setup callbacks */
  start_application();

  /* receive and process packets */
  while (1) {
    xemacif_input(netif);
    /* application specific functionality */
    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 processor systems) based applications, once the control reaches application `main` routine, a task (can be termed as main_thread) with an entry point function as main_thread() is created before starting the scheduler. After the FreeRTOS scheduler starts, the control reaches `main_thread()`, where the lwIP internal initialization happens. The application then creates one more thread (network_thread) to initialize the MAC layer.

The following pseudo-code illustrates a typical socket mode program structure.

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

  /* the MAC address of the board.
    * This should be unique per board/PHY */
  unsigned char mac_ethernet_address[] =
    {0x00, 0x0a, 0x35, 0x00, 0x01, 0x02};

  netif = &server_netif;

  /* initialize IP addresses to be used */
  IP4_ADDR(&ipaddr,192,168,1,10);
  IP4_ADDR(&netmask,255,255,255,0);
  IP4_ADDR(&gw,192,168,1,1);

  /* Add network interface to the netif_list,
    * and set it as default */
  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);

  /* specify that the network if is up */
  netif_set_up(netif);

  /* start packet receive thread
   - required for lwIP operation */
  sys_thread_new(“xemacif_input_thread”, xemacif_input_thread,
    netif,
    THREAD_STACKSIZE, DEFAULT_THREAD_PRIO);

  /* now we can start application threads */
  /* start webserver thread (e.g.) */
  sys_thread_new(“httpd” web_application_thread, 0,
      THREAD_STACKSIZE DEFAULT_THREAD_PRIO);
}

int main_thread()
{
  /* initialize lwIP before calling sys_thread_new */
  lwip_init();

  /* any thread using lwIP should be created using
    * sys_thread_new() */
  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.

Functions

- 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)
- void xemacif_input_thread (struct netif ∗netif)
- int xemacif_input (struct netif ∗netif)
- void xemacpsif_resetrx_on_no_rxdata (struct netif ∗netif)

- void lwip_init (void)

void lwip_init (void)

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.

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*)

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

void xemacif_input_thread (struct netif * *netif*)

Note

For Socket mode only.

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:

```
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`.

int xemacif_input (struct netif * *netif*)

Note

For RAW mode only.

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

```
while (1) {
          /* receive packets */
          xemacif_input(netif);

          /* do application specific processing */
     }
```

Note

The program is notified of the received data through callbacks.

void xemacpsif_resetrx_on_no_rxdata (struct netif ∗ *netif*)

Note

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

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.

XilFlash Library v4.5

Chapter 17

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.

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 write 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.

Chapter 18

XilFlash Library API

Overview

This chapter provides a linked summary and detailed descriptions of the LibXil Flash library APIs.

Functions

- 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)
- int XFlash_Unlock (XFlash ∗InstancePtr, u32 Offset, u32 Bytes)
- int XFlash_IsReady (XFlash ∗InstancePtr)

Function Documentation

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

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 Xilinx 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.

Parameters

InstancePtr	Pointer to the XFlash instance.
BaseAddress	Base address of the flash memory.
BusWidth	Total width of the flash memory, in bytes.
IsPlatformFlash	Used to specify if the flash is a platform flash.

Returns

- XST_SUCCESS if successful.
- 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.

int XFlash_Reset (XFlash * *InstancePtr*)

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

Parameters

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.

Note

None.

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

This function is used to execute device specific commands.
For a list of device specific commands, see the xilflash.h.

Parameters

InstancePtr	Pointer to the XFlash instance.
Command	Device specific command to issue.
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).

Note

None.

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

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.

Parameters

InstancePtr	Pointer to the XFlash instance.
Offset	Offset into the device(s) address space from which to read.
Bytes	Number of bytes to copy.
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).

Note

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

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

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.

Parameters

InstancePtr	Pointer to the XFlash instance.
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.
Bytes	Number of bytes to program.
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.

Note

None.

int XFlash_Erase (XFlash ∗ *InstancePtr*, u32 *Offset*, u32 *Bytes*)

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.

Parameters

InstancePtr	Pointer to the XFlash instance.
Offset	Offset into the device(s) address space from which to begin erasure.
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).

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.

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

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.

Parameters

InstancePtr	Pointer to the XFlash instance.
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 atleast three bytes from start of the block.
Bytes	Number of bytes to Lock 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).

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.

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

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.

Parameters

InstancePtr	Pointer to the XFlash instance.
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 atleast three bytes from start of the block.
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).

Note

None.

int XFlash_IsReady (XFlash * *InstancePtr*)

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

Parameters

InstancePtr	Pointer to the XFlash instance.

Returns

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

Note

None.

Chapter 19

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 = 4.5
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	4.5	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.

Xillsf Library v5.12

Chapter 20

Overview

The LibXil Isf library:

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

Additional information:

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

Supported Devices

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

Device Series	Manufacturer
AT45DB011D AT45DB021D AT45DB041D AT45DB081D AT45DB161D AT45DB321D AT45DB642D	Atmel
W25Q16 W25Q32 W25Q64 W25Q80 W25Q128 W25X10 W25X20 W25X40 W25X80 W25X16 W25X32 W25X64	Winbond
S25FL004 S25FL008 S25FL016 S25FL032 S25FL064 S25FL128 S25FL129 S25FL256 S25FL512 S70FL01G	Spansion
SST25WF080	SST

Device Series	Manufacturer
N25Q032 N25Q064 N25Q128 N25Q256 N25Q512 N25Q00AA MT25Q01 MT25Q02 MT25Q512 MT25QL02G MT25QU02G	Micron
MX66L1G45G MX66U1G45G	Macronix
IS25WP256D IS25LP256D IS25WP064A IS25LP064A IS25WP032D IS25LP032D IS25WP016D IS25LP016D IS25WP080D IS25LP080D	ISSI

Note

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

References

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

Chapter 21

Xilisf Library API

Overview

This chapter provides a linked summary and detailed descriptions of the Xilisf library APIs.

Functions

- int XIsf_Initialize (XIsf ∗InstancePtr, XIsf_Iface ∗SpiInstPtr, u8 SlaveSelect, u8 ∗WritePtr)
- int XIsf_GetStatus (XIsf ∗InstancePtr, u8 ∗ReadPtr)
- int XIsf_GetStatusReg2 (XIsf ∗InstancePtr, u8 ∗ReadPtr)
- int XIsf_GetDeviceInfo (XIsf ∗InstancePtr, u8 ∗ReadPtr)
- int XIsf_Write (XIsf ∗InstancePtr, XIsf_WriteOperation Operation, void ∗OpParamPtr)
- int XIsf_Read (XIsf ∗InstancePtr, XIsf_ReadOperation Operation, void ∗OpParamPtr)
- int XIsf_Erase (XIsf ∗InstancePtr, XIsf_EraseOperation Operation, u32 Address)
- int XIsf_MicronFlashEnter4BAddMode (XIsf ∗InstancePtr)
- int XIsf_MicronFlashExit4BAddMode (XIsf ∗InstancePtr)
- int XIsf_SectorProtect (XIsf ∗InstancePtr, XIsf_SpOperation Operation, u8 ∗BufferPtr)
- int XIsf_Ioctl (XIsf ∗InstancePtr, XIsf_IoctlOperation Operation)
- int XIsf_WriteEnable (XIsf ∗InstancePtr, u8 WriteEnable)
- void XIsf_RegisterInterface (XIsf ∗InstancePtr)
- int XIsf_SetSpiConfiguration (XIsf ∗InstancePtr, XIsf_Iface ∗SpiInstPtr, u32 Options, u8 PreScaler)
- void XIsf_SetStatusHandler (XIsf ∗InstancePtr, XIsf_Iface ∗XIfaceInstancePtr, XIsf_StatusHandler XilIsf_Handler)
- void XIsf_IfaceHandler (void ∗CallBackRef, u32 StatusEvent, unsigned int ByteCount)

Function Documentation

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

This API when called initializes the SPI interface with default settings.
With custom settings, user should call XIsf_SetSpiConfiguration() and then call this API. The geometry of the underlying Serial Flash is determined by reading the Joint Electron Device Engineering Council (JEDEC) Device Information and the Status Register of the Serial Flash.

Parameters

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

Returns

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

Note

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

int XIsf_GetStatus (XIsf ∗ *InstancePtr*, u8 ∗ *ReadPtr*)

This API reads the Serial Flash Status Register.

Parameters

InstancePtr	Pointer to the XIsf instance.
ReadPtr	Pointer to the memory where the Status Register content is copied.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

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

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

This API reads the Serial Flash Status Register 2.

Parameters

InstancePtr	Pointer to the XIsf instance.
ReadPtr	Pointer to the memory where the Status Register content is copied.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

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

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

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

Parameters

InstancePtr	Pointer to the XIsf instance.
ReadPtr	Pointer to the buffer where the Device information is copied.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

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

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

This API writes the data to the Serial Flash.

Parameters

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

Operations

- Normal Write(XISF_WRITE), Dual Input Fast Program (XISF_DUAL_IP_PAGE_WRITE), Dual Input Extended Fast Program(XISF_DUAL_IP_EXT_PAGE_WRITE), Quad Input Fast Program(XISF_QUAD_IP_PAGE_WRITE), Quad Input Extended Fast Program (XISF_QUAD_IP_EXT_PAGE_WRITE):
  - The OpParamPtr must be of type struct XIsf_WriteParam.
  - OpParamPtr->Address is the start address in the Serial Flash.
  - OpParamPtr->WritePtr is a pointer to the data to be written to the Serial Flash.

    - OpParamPtr->NumBytes is the number of bytes to be written to Serial Flash.
    - This operation is supported for Atmel, Intel, STM, Winbond and Spansion Serial Flash.
- Auto Page Write (XISF_AUTO_PAGE_WRITE):
    - The OpParamPtr must be of 32 bit unsigned integer variable.
    - This is the address of page number in the Serial Flash which is to be refreshed.
    - This operation is only supported for Atmel Serial Flash.
- Buffer Write (XISF_BUFFER_WRITE):
    - The OpParamPtr must be of type struct XIsf_BufferToFlashWriteParam.
    - OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
    - OpParamPtr->WritePtr is a pointer to the data to be written to the Serial Flash SRAM Buffer.
    - OpParamPtr->ByteOffset is byte offset in the buffer from where the data is to be written.
    - OpParamPtr->NumBytes is number of bytes to be written to the Buffer. This operation is supported only for Atmel Serial Flash.
- Buffer To Memory Write With Erase (XISF_BUF_TO_PAGE_WRITE_WITH_ERASE)/ Buffer To Memory Write Without Erase (XISF_BUF_TO_PAGE_WRITE_WITHOUT_ERASE):
    - The OpParamPtr must be of type struct XIsf_BufferToFlashWriteParam.
    - OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
    - OpParamPtr->Address is starting address in the Serial Flash memory from where the data is to be written. These operations are only supported for Atmel Serial Flash.
- Write Status Register (XISF_WRITE_STATUS_REG):
    - The OpParamPtr must be of type of 8 bit unsigned integer variable. This is the value to be written to the Status Register.
    - This operation is only supported for Intel, STM Winbond and Spansion Serial Flash.
- Write Status Register2 (XISF_WRITE_STATUS_REG2):
    - The OpParamPtr must be of type (u8 $*$) and should point to two 8 bit unsigned integer values. This is the value to be written to the 16 bit Status Register. This operation is only supported in Winbond (W25Q) Serial Flash.
- One Time Programmable Area Write(XISF_OTP_WRITE):
    - The OpParamPtr must be of type struct XIsf_WriteParam.
    - OpParamPtr->Address is the address in the SRAM Buffer of the Serial Flash to which the data is to be written.
    - OpParamPtr->WritePtr is a pointer to the data to be written to the Serial Flash.
    - OpParamPtr->NumBytes should be set to 1 when performing OTPWrite operation. This operation is only supported for Intel Serial Flash.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

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

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

This API reads the data from the Serial Flash.

Parameters

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

Operations

- Normal Read (XISF_READ), Fast Read (XISF_FAST_READ), One Time Programmable Area Read(XISF_OTP_READ), Dual Output Fast Read (XISF_CMD_DUAL_OP_FAST_READ), Dual Input/Output Fast Read (XISF_CMD_DUAL_IO_FAST_READ), Quad Output Fast Read (XISF_CMD_QUAD_OP_FAST_READ) and Quad Input/Output Fast Read (XISF_CMD_QUAD_IO_FAST_READ):
  - The OpParamPtr must be of type struct XIsf_ReadParam.
  - OpParamPtr->Address is start address in the Serial Flash.
  - OpParamPtr->ReadPtr is a pointer to the memory where the data read from the Serial Flash is stored.
  - OpParamPtr->NumBytes is number of bytes to read.
  - OpParamPtr->NumDummyBytes is the number of dummy bytes to be transmitted for the Read command. This parameter is only used in case of Dual and Quad reads.
  - Normal Read and Fast Read operations are supported for Atmel, Intel, STM, Winbond and Spansion Serial Flash.

  - Dual and quad reads are supported for Winbond (W25QXX), Numonyx(N25QXX) and Spansion (S25FL129) quad flash.
  - OTP Read operation is only supported in Intel Serial Flash.

- Page To Buffer Transfer (XISF_PAGE_TO_BUF_TRANS):
  - The OpParamPtr must be of type struct XIsf_FlashToBufTransferParam .
  - OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
  - OpParamPtr->Address is start address in the Serial Flash. This operation is only supported in Atmel Serial Flash.
- Buffer Read (XISF_BUFFER_READ) and Fast Buffer Read(XISF_FAST_BUFFER_READ):
  - The OpParamPtr must be of type struct XIsf_BufferReadParam.
  - OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
  - OpParamPtr->ReadPtr is pointer to the memory where data read from the SRAM buffer is to be stored.
  - OpParamPtr->ByteOffset is byte offset in the SRAM buffer from where the first byte is read.
  - OpParamPtr->NumBytes is the number of bytes to be read from the Buffer. These operations are supported only in Atmel Serial Flash.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

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

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

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

Parameters

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

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

- The erased bytes will read as 0xFF.
- For Intel, STM, Winbond or Spansion Serial Flash the user application must call XIsf_WriteEnable() API by passing XISF_WRITE_ENABLE as an argument before calling XIsf_Erase() API.
- Atmel Serial Flash support Page/Block/Sector Erase
- operations.
- Intel, Winbond, Numonyx (N25QXX) and Spansion Serial Flash support Sector/Block/Bulk Erase operations.
- STM (M25PXX) Serial Flash support Sector/Bulk Erase operations.

int XIsf_MicronFlashEnter4BAddMode (XIsf * *InstancePtr*)

This API enters the Micron flash device into 4 bytes addressing mode.
As per the Micron spec, before issuing the command to enter into 4 byte addr mode, a write enable command is issued.

Parameters

InstancePtr	Pointer to the XIsf instance.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if it fails.

Note

Applicable only for Micron flash devices

int XIsf_MicronFlashExit4BAddMode (XIsf * *InstancePtr*)

This API exits the Micron flash device from 4 bytes addressing mode.
As per the Micron spec, before issuing this command a write enable command is first issued.

Parameters

InstancePtr	Pointer to the XIsf instance.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if it fails.

Note

Applicable only for Micron flash devices

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

This API is used for performing Sector Protect related operations.

Parameters

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

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

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

int XIsf_Ioctl (XIsf ∗ *InstancePtr,* XIsf_IoctlOperation *Operation*)

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

Parameters

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

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

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

int XIsf_WriteEnable (XIsf ∗ *InstancePtr,* u8 *WriteEnable*)

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

Parameters

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

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

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

void XIsf_RegisterInterface (XIsf $*$ *InstancePtr*)

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

Parameters

InstancePtr	Pointer to the XIsf instance.

Returns

None

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

This API sets the configuration of SPI.
This will set the options and clock prescaler (if applicable).

Parameters

InstancePtr	Pointer to the XIsf instance.
SpiInstPtr	Pointer to XIsf_Iface instance to be worked on.
Options	Specified options to be set.
PreScaler	Value of the clock prescaler to set.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

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

void XIsf_SetStatusHandler (XIsf ∗ *InstancePtr,* XIsf_Iface ∗ *XIfaceInstancePtr,* XIsf_StatusHandler *XiIsf_Handler*)

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

Parameters

InstancePtr	Pointer to the XIsf Instance.
QspiInstancePtr	Pointer to the XIsf_Iface instance to be worked on.
XiIsf_Handler	Status handler for the application.

Returns

None

Note

None.

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

This API is the handler which performs processing for the QSPI driver.
It is called from an interrupt context such that the amount of processing performed should be minimized. It is called when a transfer of QSPI data completes or an error occurs.
This handler provides an example of how to handle QSPI interrupts but is application specific.

Parameters

CallBackRef	Reference passed to the handler.
StatusEvent	Status of the QSPI .
ByteCount	Number of bytes transferred.

Returns

None

Note

None.

Chapter 22

Library Parameters in MSS File

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

```
BEGIN LIBRARY
PARAMETER LIBRARY_NAME = xilisf
PARAMETER LIBRARY_VER = 5.12
PARAMETER serial_flash_family = 1
PARAMETER serial_flash_interface = 1
END
```

The table below describes the libgen customization parameters.

Parameter	Default Value	Description
LIBRARY_NAME	xilisf	Specifies the library name.
LIBRARY_VER	5.12	Specifies the library version.
serial_flash_family	1	Specifies the serial flash family. Supported numerical values are: 1 = Xilinx In-system Flash or Atmel Serial Flash 2 = Intel (Numonyx) S33 Serial Flash 3 = STM (Numonyx) M25PXX/N25QXX Serial Flash 4 = Winbond Serial Flash 5 = Spansion Serial Flash/Micron Serial Flash/Cypress Serial Flash 6 = SST Serial Flash
Serial_flash_interface	1	Specifies the serial flash interface. Supported numerical values are: 1 = AXI QSPI Interface 2 = SPI PS Interface 3 = QSPI PS Interface or QSPI PSU Interface

Note

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

XilFFS Library v4.0

Chapter 23

Overview

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

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

Note

> The XilFFS library uses `Revision R0.13b` of the generic FAT filesystem module.

Library Files

The table below lists the file system files.

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

The table below lists the glue layer files.

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

Selecting a File System with an SD Interface

To select a file system with an SD interface:

1. Launch Xilinx SDK. Xilinx SDK prompts you to create a workspace.
2. Select **File** > **New** > **Xilinx Board Support Package**. The **New Board Support Package** wizard appears.
3. Specify a project name.
4. Select **Standalone** from the **Board Support Package OS** drop-down list. The **Board Support Package Settings** wizard appears.
5. Select the **xilffs** library from the list of **Supported Libraries**.
6. Expand the **Overview** tree and select **xilffs**. The configuration options for xilffs are listed.
7. 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.
8. 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. Launch Xilinx SDK. Xilinx SDK prompts you to create a workspace.
2. Select **File** > **New** > **Xilinx Board Support Package**. The b>New Board Support Package appears.
3. Specify a project name.
4. **Standalone** from the **Board Support Package OS** drop-down list. The **Board Support Package Settings** wizard appears.
5. Select the **xilffs** library from the list of **Supported Libraries**.
6. Expand the **Overview** tree and select xilffs. The configuration options for xilffs are listed.
7. Configure the xilffs by setting the `fs_interface = 2` to select `RAM`.

8. 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.

9. Use any lwip application that requires a RAM based file system - TCP/UDP performance test apps or tftp or webserver examples.

10. Build the bsp and the application to use the RAM based file system.

Chapter 24

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 = 4.0
PARAMETER fs_interface = 1
PARAMETER read_only = false
PARAMETER use_lfn = false
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
END
```

The table below describes the libgen customization parameters.

Parameter	Default Value	Description
LIBRARY_NAME	xilffs	Specifies the library name.
LIBRARY_VER	4.0	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	False	Enables the long file name (LFN) support, if true.
enable_multi_partitio	False	Enables the multi partition support, if true.

Parameter	Default Value	Description
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.
enable_exfat	false	Enables support for exFAT file system.
word_access	True	Enables word access for misaligned memory access platform.

XilSecure Library v3.2

Chapter 25

Overview

The XilSecure library provides APIs to access secure hardware on the Zynq® UltraScale+™ MPSoC devices and also provides a software implementation for SHA-2 hash generation.
The XilSecure library includes:

- SHA-3/384 engine for 384 bit hash calculation
- AES engine for symmetric key encryption and decryption
- RSA engine for signature generation, signature verification, encryption and decryption.

  Note

  > The above libraries are grouped into the Configuration and Security Unit (CSU) on the Zynq UltraScale+ MPSoC device.

- SHA-2/256 algorithm for calculating 256 bit hash

  Note

  > The SHA-2 hash generation is a software algorithm which generates SHA2 hash on provided data. SHA-2 support is deprecated. Use SHA-3 instead of SHA-2.

Source Files

The following is a list of source files shipped as a part of the XilSecure library:

- `xsecure_hw.h`: This file contains the hardware interface for all the three modules.
- `xsecure_sha.h`: This file contains the driver interface for SHA-3 module.
- `xsecure_sha.c`: This file contains the implementation of the driver interface for SHA-3 module.
- `xsecure_rsa.h`: This file contains the driver interface for RSA module.
- `xsecure_rsa.c`: This file contains the implementation of the driver interface for RSA module.
- `xsecure_aes.h`: This file contains the driver interface for AES module.
- `xsecure_aes.c` : This file contains the implementation of the driver interface for AES module
- `xsecure_sha2.h`: This file contains the interface for SHA2 hash algorithm.

- `xsecure_sha2_a53_32b.a`: Pre-compiled file which has SHA2 implementation for A53 32bit.
- `xsecure_sha2_a53_64b.a`: Pre-compiled file which has SHA2 implementation for A53 64 bit.
- `xsecure_sha2_a53_r5.a`: Pre-compiled file which has SHA2 implementation for r5.
- `xsecure_sha2_pmu.a`: Pre-compiled file which has SHA2 implementation for PMU.

Chapter 26

AES-GCM

Overview

This block uses AES-GCM algorithm to encrypt or decrypt the provided data. It 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
- Decryption of Zynq® Ultrascale+™ MPSoC boot image partition, where boot image is generated using `bootgen`.
    - Support for Key rolling
    - Operational key support
- Authentication using GCM tag.
- Key loading based on key selection, key can be either the user provided key loaded into the KUP key or the device key used in the boot.

For either encryption or decryption AES should be initialized first, the XSecure_AesInitialize() API initializes the AES's instance with provided parameters as described.

AES Encryption Function Usage

When all the data to be encrypted is available, the `XSecure_AesEncryptData()` can be used with appropriate parameters as described. When all the data is not available use the following functions in following order.

1. `XSecure_AesEncryptInit()`
2. `XSecure_AesEncryptUpdate()` - This API 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 with appropriate parameters as described. When all the data is not available use the following functions in following order.

1. `XSecure_AesDecryptInit()`

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

The GCM-TAG matching will also be verified and appropriate status will be returned.

> **WARNING:** *Warning when using the KUP key for encryption/decryption of the data, you should take care of the key storage. Key should be placed in a secure memory region for example, internal memory. Not doing so may result in security vulnerability.*

Modules

- AES-GCM API Example Usage

Functions

- s32 XSecure_AesInitialize (XSecure_Aes *InstancePtr, XCsuDma *CsuDmaPtr, u32 KeySel, u32 *Iv, u32 *Key)
- void 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, u8 *GcmTagAddr)
- s32 XSecure_AesDecrypt (XSecure_Aes *InstancePtr, u8 *Dst, const u8 *Src, u32 Length)
- void XSecure_AesEncryptInit (XSecure_Aes *InstancePtr, u8 *EncData, u32 Size)
- void XSecure_AesEncryptUpdate (XSecure_Aes *InstancePtr, const u8 *Data, u32 Size)
- void XSecure_AesEncryptData (XSecure_Aes *InstancePtr, u8 *Dst, const u8 *Src, u32 Len)
- void XSecure_AesReset (XSecure_Aes *InstancePtr)
- void XSecure_AesWaitForDone (XSecure_Aes *InstancePtr)

Function Documentation

s32 XSecure_AesInitialize (XSecure_Aes * *InstancePtr*, XCsuDma * *CsuDmaPtr*, u32 *KeySel*, u32 * *Iv*, u32 * *Key*)

This function initializes the instance pointer.

Parameters

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

Returns

XST_SUCCESS if initialization was successful.

Note

All the inputs are accepted in little endian format, but AES engine accepts the data in big endianess, this will be taken care while passing data to AES engine.

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

This function initializes the AES engine for decryption.

Parameters

InstancePtr	Pointer to the XSecure_Aes instance.
DecData	Pointer in which decrypted data will be stored.
Size	Expected size of the data in bytes.
GcmTagAddr	Pointer to the GCM tag which needs to be verified during decryption of the data.

Returns

None

Note

If data is encrypted using XSecure_AesEncrypt API then GCM tag address will be at the end of encrypted data. EncData + Size will be the GCM tag address. Chunking will not be handled over here.

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

This function is used to update the AES engine for decryption with provided data.

Parameters

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.

Returns

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

- XSECURE_CSU_AES_GCM_TAG_MISMATCH: If GCM tag is mismatched
- XST_SUCCESS: If GCM tag is matching.

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.

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

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

Parameters

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.

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.

Note

When XSecure_AesEncryptData() API is used for encryption In same buffer GCM tag also be stored, but Size should be mentioned only for data.

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

This function will handle the AES-GCM Decryption.

Parameters

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.

Note

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

void XSecure_AesEncryptInit (XSecure_Aes ∗ *InstancePtr*, u8 ∗ *EncData*, u32 *Size*)

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

Parameters

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.

Returns

None

Note

If all the data to be encrypted is available at single location One can use XSecure_AesEncryptData() directly.

void XSecure_AesEncryptUpdate (XSecure_Aes ∗ *InstancePtr*, const u8 ∗ *Data*, u32 *Size*)

This function is used to update the AES engine with provided data for encryption.

Parameters

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.

Returns

None

Note

When Size of the data equals to size of the remaining data to be processed 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 mentioned at encryption initialization (XSecure_AesEncryptInit()). If all the data to be encrypted is available at single location Please call XSecure_AesEncryptData() directly.

void XSecure_AesEncryptData (XSecure_Aes ∗ *InstancePtr,* u8 ∗ *Dst,* const u8 ∗ *Src,* u32 *Len*)

This Function encrypts the data provided by using hardware AES engine.

Parameters

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

Returns

None

Note

If data to be encrypted is not available at one place one can call XSecure_AesEncryptInit() and update the AES engine with data to be encrypted by calling XSecure_AesEncryptUpdate() API multiple times as required.

void XSecure_AesReset (XSecure_Aes ∗ *InstancePtr*)

This function resets the AES engine.

Parameters

InstancePtr	is a pointer to the XSecure_Aes instance.

Returns

None

void XSecure_AesWaitForDone (XSecure_Aes * *InstancePtr*)

This function waits for AES completion.

Parameters

InstancePtr	Pointer to the XSecure_Aes instance.

Returns

None

AES-GCM API Example Usage

The `xilsecure_aes_example.c` file illustrates AES usage with decryption of a Zynq® UltraScale+™ MPSoC boot image placed at a predefined location in memory. You can select the key type (device key or user-selected KUP key). The example assumes that the boot image is present at `0x0400000` (DDR); consequently, the image must be loaded at that address through JTAG. The example decrypts the boot image and returns `XST_SUCCESS` or `XST_FAILURE` based on whether the GCM tag was successfully matched.
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 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 Device key or user provided key. If more than one block is found then the key and 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.

The following is a snippet from the `xilsecure_aes_example.c` file.

```
int SecureAesExample(void)
{
  u8 *Dst = (u8 *)(UINTPTR)DestinationAddr;
  XCsuDma_Config *Config;

  int Status;
  u32 PhOffset;
  u32 PartitionOffset;
  u32 PartitionLen;

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

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

  /* Read partition offset and length from partition header */
  PhOffset = XSecure_In32((UINTPTR)(ImageOffset +
        XSECURE_BHDR_PH_OFFSET));
  PartitionOffset = (XSecure_In32((UINTPTR)(ImageOffset + PhOffset +
      XSECURE_PH_PARTITION_OFFSET)) * XSECURE_WORD_MUL);
  PartitionLen = (XSecure_In32((UINTPTR)(ImageOffset + PhOffset +
      XSECURE_PH_PARTITION_LEN_OFFSET)) * XSECURE_WORD_MUL);

  *(csu_iv + XSECURE_IV_INDEX) =  (*(csu_iv + XSECURE_IV_INDEX)) +
      (XSecure_In32((UINTPTR)(ImageOffset + PhOffset +
        XSECURE_PH_PARTITION_IV_OFFSET)) &
        XSECURE_PH_PARTITION_IV_MASK);
  /*
   * Initialize the Aes driver so that it’s ready to use
   */
  XSecure_AesInitialize(&Secure_Aes, &CsuDma, XSECURE_CSU_AES_KEY_SRC_KUP,
                               (u32 *)csu_iv, (u32 *)csu_key);

  Status = XSecure_AesDecrypt(&Secure_Aes, Dst,
      (u8 *)(UINTPTR)(ImageOffset + PartitionOffset),
            PartitionLen);


  if(Status != XST_SUCCESS)
  {
    return XST_FAILURE;
  }

  return XST_SUCCESS;
}
```

Note

The `xilsecure_aes_example.c` example file is available in the `<library-install-path>\examples` folder. Where `<library-install-path>` is the XilSecure library installation path.

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

```
static s32 SecureAesExample(void)
{
  XCsuDma_Config *Config;
  s32 Status;
  u32 Index;
  u8 DecData[XSECURE_DATA_SIZE]__attribute__ ((aligned (64)));
  u8 EncData[XSECURE_DATA_SIZE + XSECURE_SECURE_GCM_TAG_SIZE]
          __attribute__ ((aligned (64)));

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

  Status = XCsuDma_CfgInitialize(&CsuDma, 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, &CsuDma,
        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;
}
```

Chapter 27

RSA

Overview

The xsecure_rsa.h file contains hardware interface related information for RSA hardware accelerator. This block performs the modulus math based on Rivest-Shamir-Adelman (RSA)-4096 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 needs pre-calculated value of `R^2 mod N`, which is generated by bootgen and is present in the signature along with modulus and exponent. If you do not have the pre-calculated exponential value pass NULL, the controller will take care of exponential value.

Note

- From RSA key modulus, exponent should be extracted. If image is created using bootgen all the fields are available in the boot image.
- For matching, PKCS v1.5 padding scheme has to be applied in the manner while comparing the data hash with decrypted hash.

Modules

- RSA API Example Usage

Functions

- s32 XSecure_RsaInitialize (XSecure_Rsa *InstancePtr, u8 *Mod, u8 *ModExt, u8 *ModExpo)
- s32 XSecure_RsaDecrypt (XSecure_Rsa *InstancePtr, u8 *EncText, u8 *Result)
- u32 XSecure_RsaSignVerification (u8 *Signature, u8 *Hash, u32 HashLen)
- s32 XSecure_RsaPublicEncrypt (XSecure_Rsa *InstancePtr, u8 *Input, u32 Size, u8 *Result)
- s32 XSecure_RsaPrivateDecrypt (XSecure_Rsa *InstancePtr, u8 *Input, u32 Size, u8 *Result)

Function Documentation

s32 XSecure_RsaInitialize (XSecure_Rsa * *InstancePtr,* u8 * *Mod,* u8 * *ModExt,* u8 * *ModExpo*)

This function initializes a specific Xsecure_Rsa instance so that it is ready to be used.

Parameters

InstancePtr	Pointer to the XSecure_Rsa instance.
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.

Note

`Modulus`, `ModExt` and `ModExpo` are part of prtition signature when authenticated boot image is generated by bootgen, else the all of them should be extracted from the key.

s32 XSecure_RsaDecrypt (XSecure_Rsa * *InstancePtr,* u8 * *EncText,* u8 * *Result*)

This function handles the RSA decryption from end to end.

Parameters

InstancePtr	Pointer to the XSecure_Rsa instance.
EncText	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 if decryption was successful.

Note

This API will be deprecated soon. Instead of this please use XSecure_RsaPublicEncrypt() API. This API can only support 4096 key Size.

u32 XSecure_RsaSignVerification (u8 ∗ *Signature,* u8 ∗ *Hash,* u32 *HashLen*)

This function verifies the RSA decrypted data provided is either matching with the provided expected hash by taking care of PKCS padding.

Parameters

Signature	Pointer to the buffer which holds the decrypted RSA signature
Hash	Pointer to the buffer which has hash calculated on the data to be authenticated.
HashLen	Length of Hash used. • For SHA3 it should be 48 bytes • For SHA2 it should be 32 bytes

Returns

XST_SUCCESS if decryption was successful.

s32 XSecure_RsaPublicEncrypt (XSecure_Rsa ∗ *InstancePtr,* u8 ∗ *Input,* u32 *Size,* u8 ∗ *Result*)

This function handles the RSA signature encryption with public key components provide at XSecure_RsaInitialize() API.

Parameters

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 and • XSECURE_RSA_2048_KEY_SIZE
Result	Pointer to the buffer where resultant decrypted data to be stored .

Returns

XST_SUCCESS if encryption was successful.

Note

Modulus of API XSecure_RsaInitialize() should also be same size of key size mentioned in this API and exponent should be 32 bit size.

s32 XSecure_RsaPrivateDecrypt (XSecure_Rsa * *InstancePtr*, u8 * *Input*, u32 *Size*, u8 * *Result*)

This function handles the RSA signature decryption with private key components provide at XSecure_RsaInitialize() API.

Parameters

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 and XSECURE_RSA_2048_KEY_SIZE*
Result	Pointer to the buffer where resultant decrypted data to be stored .

Returns

XST_SUCCESS if decryption was successful. else returns an error code

- XSECURE_RSA_DATA_VALUE_ERROR - if input data is greater than modulus
    - XST_FAILURE - on RSA operation failure

Note

Modulus and Exponent in XSecure_RsaInitialize() API should also be same as key size mentioned in this API.

RSA API Example Usage

The `xilsecure_rsa_example.c` file illustrates the usage of XilSecure APIs by authenticating the FSBL partition of the Zynq® UltraScale+™ MPSoC boot image. The boot image signature is encrypted using RSA-4096 algorithm. Resulting digest is matched with SHA3 Hash calculated on the FSBL.

The authenticated boot image should be loaded in memory through JTAG and address of the boot image should be passed to the function. By default, the example assumes that the authenticated image is present at location 0x04000000 (DDR), which can be changed as required.

The following is a snippet from the `xilsecure_rsa_example.c` file.

```
u32 SecureRsaExample(void)
{
  u32 Status;

  /*
   * Download the boot image elf at a DDR location, Read the boot header
   * assign Src pointer to the location of FSBL image in it. Ensure
   * that linker script does not map the example elf to the same
   * location as this standalone example
   */
  u32 FsblOffset = XSecure_In32((UINTPTR)(ImageOffset + HeaderSrcOffset));

  xil_printf(" Fsbl Offset in the image is  %0x ",FsblOffset);
  xil_printf(" \r\n ");
```

```
u32 FsblLocation = ImageOffset + FsblOffset;

xil_printf(" Fsbl Location is %0x ",FsblLocation);
xil_printf(" \r\n ");

u32 TotalFsblLength = XSecure_In32((UINTPTR)(ImageOffset +
        HeaderFsblTotalLenOffset));

u32 AcLocation = FsblLocation + TotalFsblLength - XSECURE_AUTH_CERT_MIN_SIZE;

xil_printf(" Authentication Certificate Location is %0x ",AcLocation);
xil_printf(" \r\n ");

u8 BIHash[XSECURE_HASH_TYPE_SHA3] __attribute__ ((aligned (4)));
u8 * SpkModular = (u8 *)XNULL;
u8 * SpkModularEx = (u8 *)XNULL;
u32 SpkExp = 0;
u8 * AcPtr = (u8 *)(UINTPTR)AcLocation;
u32 ErrorCode = XST_SUCCESS;
u32 FsblTotalLen = TotalFsblLength - XSECURE_FSBL_SIG_SIZE;

xil_printf(" Fsbl Total Length(Total - BI Signature) %0x ",
    (u32)FsblTotalLen);
xil_printf(" \r\n ");

AcPtr += (XSECURE_RSA_AC_ALIGN + XSECURE_PPK_SIZE);
SpkModular = (u8 *)AcPtr;
AcPtr += XSECURE_FSBL_SIG_SIZE;
SpkModularEx = (u8 *)AcPtr;
AcPtr += XSECURE_FSBL_SIG_SIZE;
SpkExp = *((u32 *)AcPtr);
AcPtr += XSECURE_RSA_AC_ALIGN;

AcPtr += (XSECURE_SPK_SIG_SIZE + XSECURE_BHDR_SIG_SIZE);
xil_printf(" Boot Image Signature Location is %0x ",(u32)(UINTPTR)AcPtr);
xil_printf(" \r\n ");

/*
 * Set up CSU DMA instance for SHA-3 transfers
 */
XCsuDma_Config *Config;

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

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

/*
 * Initialize the SHA-3 driver so that it's ready to use
 * Look up the configuration in the config table and then initialize it.
 */

XSecure_Sha3Initialize(&Secure_Sha3, &CsuDma);
/* As we are authenticating FSBL here SHA3 KECCAK should be used */
Status = XSecure_Sha3PadSelection(&Secure_Sha3, XSECURE_CSU_KECCAK_SHA3);
if (Status != XST_SUCCESS) {
  goto ENDF;
}
XSecure_Sha3Start(&Secure_Sha3);

XSecure_Sha3Update(&Secure_Sha3, (u8 *)(UINTPTR)FsblLocation,
        FsblTotalLen);
  XSecure_Sha3Finish(&Secure_Sha3, (u8 *)BIHash);

  /*
   * Initialize the Rsa driver so that it's ready to use
   * Look up the configuration in the config table and then initialize it.
   */
  XSecure_RsaInitialize(&Secure_Rsa, SpkModular, SpkModularEx,
          (u8 *)&SpkExp);

  /*
   * Decrypt Boot Image Signature.
   */
  if(XST_SUCCESS != XSecure_RsaDecrypt(&Secure_Rsa, AcPtr,
            XSecure_RsaSha3Array))
  {
    ErrorCode = XSECURE_IMAGE_VERIF_ERROR;
    goto ENDF;
  }

  xil_printf("\r\n Calculated Boot image Hash \r\n ");
  int i= 0;
  for(i=0; i < 384/8; i++)
  {
    xil_printf(" %0x ", BIHash[i]);
  }
  xil_printf(" \r\n ");

  xil_printf("\r\n Hash From Signature \r\n ");
  int ii= 128;
  for(ii = 464; ii < 512; ii++)
  {
    xil_printf(" %0x ", XSecure_RsaSha3Array[ii]);
  }
  xil_printf(" \r\n ");

  /*
   * Authenticate FSBL Signature.
   */
  if(XSecure_RsaSignVerification(XSecure_RsaSha3Array, BIHash,
                         XSECURE_HASH_TYPE_SHA3) != 0)
  {
    ErrorCode = XSECURE_IMAGE_VERIF_ERROR;
  }

ENDF:
  return ErrorCode;
}
```

Note

> The `xilsecure_rsa_example.c` example file is available in the `<library-install-path>\examples` folder. Where `<library-install-path>` is the XilSecure library installation path.

The following example illustrates the usage of RSA APIs to encrypt the data using public key and to decrypt the data using private key.

Note

> Application should take care of the padding.

```
u32 SecureRsaExample(void)
{
  u32 Index;
  /* RSA signature decrypt with private key */
  /*
   * Initialize the Rsa driver with private key components
   * so that it's ready to use
   */
  XSecure_RsaInitialize(&Secure_Rsa, Modulus, NULL, PrivateExp);


  if(XST_SUCCESS != XSecure_RsaPrivateDecrypt(&Secure_Rsa, Data,
            Size, Signature)) {
    xil_printf("Failed at RSA signature decryption\n\r");
    return XST_FAILURE;
  }

  xil_printf("\r\n Decrypted Signature with private key\r\n ");

  for(Index = 0; Index < Size; Index++) {
    xil_printf(" %02x ", Signature[Index]);
  }
  xil_printf(" \r\n ");

  /* Verification if Data is expected */
  for(Index = 0; Index < Size; Index++) {
    if (Signature[Index] != ExpectedSign[Index]) {
      xil_printf("\r\nError at verification of RSA signature"
          " Decryption\n\r");
      return XST_FAILURE;
    }
  }

  /* RSA signature encrypt with Public key components */

  /*
   * Initialize the Rsa driver with public key components
   * so that it's ready to use
   */

  XSecure_RsaInitialize(&Secure_Rsa, Modulus, NULL, (u8 *)&PublicExp);

  if(XST_SUCCESS != XSecure_RsaPublicEncrypt(&Secure_Rsa, Signature,
            Size, EncryptSignatureOut)) {
    xil_printf("\r\nFailed at RSA signature encryption\n\r");
    return XST_FAILURE;
  }
  xil_printf("\r\n Encrypted Signature with public key\r\n ");

  for(Index = 0; Index < Size; Index++) {
    xil_printf(" %02x ", EncryptSignatureOut[Index]);
  }

  /* Verification if Data is expected */
  for(Index = 0; Index < Size; Index++) {
    if (EncryptSignatureOut[Index] != Data[Index]) {
      xil_printf("\r\nError at verification of RSA signature"
          " encryption\n\r");
      return XST_FAILURE;
    }
  }


  return XST_SUCCESS;
}
```

Chapter 28

SHA-3

Overview

This block uses the NIST-approved SHA-3 algorithm to generate 384 bit hash on the input data. Because the SHA-3 hardware only accepts 104 byte blocks as minimum input size, the input data is padded with a 10∗1 sequence to complete the final byte block. The padding is handled internally by the driver API.

Note

> By default, the library calculates hash using NIST SHA3 padding, but supports choosing KECCAK SHA3 padding.

Initialization & Configuration

The SHA-3 driver instance can be initialized using the `XSecure_Sha3Initialize()` function.
A pointer to CsuDma instance has to be passed in initialization as CSU DMA will be used for data transfers to SHA module.

SHA-3 Functions Usage

When all the data is available on which sha3 hash must be calculated, the `XSecure_Sha3Digest()` can be used with appropriate parameters, as described. When all the data is not available on which sha3 hash must be calculated, use the sha3 functions in the following order:

1. `XSecure_Sha3Start()`
2. `XSecure_Sha3Update()` - This API can be called multiple times till input data is completed.
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.

Modules

- SHA-3 API Example Usage

Functions

- s32 XSecure_Sha3Initialize (XSecure_Sha3 ∗InstancePtr, XCsuDma ∗CsuDmaPtr)
- void XSecure_Sha3Start (XSecure_Sha3 ∗InstancePtr)
- void XSecure_Sha3Update (XSecure_Sha3 ∗InstancePtr, const u8 ∗Data, const u32 Size)
- void XSecure_Sha3Finish (XSecure_Sha3 ∗InstancePtr, u8 ∗Hash)
- void 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 Sha3Type)

Function Documentation

s32 XSecure_Sha3Initialize (XSecure_Sha3 ∗ *InstancePtr*, XCsuDma ∗ *CsuDmaPtr*)

This function initializes a specific Xsecure_Sha3 instance so that it is ready to be used.

Parameters

InstancePtr	Pointer to the XSecure_Sha3 instance.
CsuDmaPtr	Pointer to the XCsuDma instance.

Returns

XST_SUCCESS if initialization was successful

Note

The base address is initialized directly with value from xsecure_hw.h By default uses NIST SHA3 padding, to change to KECCAK padding call XSecure_Sha3PadSelection() after XSecure_Sha3Initialize().

void XSecure_Sha3Start (XSecure_Sha3 ∗ *InstancePtr*)

This function configures the SSS and starts the SHA-3 engine.

Parameters

InstancePtr	Pointer to the XSecure_Sha3 instance.

Returns

None

void XSecure_Sha3Update (XSecure_Sha3 * *InstancePtr*, const u8 * *Data*, const u32 *Size*)

This function updates hash for new input data block.

Parameters

InstancePtr	Pointer to the XSecure_Sha3 instance.
Data	Pointer to the input data for hashing.
Size	Size of the input data in bytes.

Returns

None

void XSecure_Sha3Finish (XSecure_Sha3 * *InstancePtr*, u8 * *Hash*)

This function sends the last data and padding when blocksize is not multiple of 104 bytes.

Parameters

InstancePtr	Pointer to the XSecure_Sha3 instance.
Hash	Pointer to location where resulting hash will be written

Returns

None

void XSecure_Sha3Digest (XSecure_Sha3 * *InstancePtr*, const u8 * *In*, const u32 *Size*, u8 * *Out*)

This function calculates the SHA-3 digest on the given input data.

Parameters

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

None

void XSecure_Sha3_ReadHash (XSecure_Sha3 * *InstancePtr,* u8 * *Hash*)

Reads the SHA3 hash of the data. It can be called intermediately of updates also to read hashs.

Parameters

InstancePtr	Pointer to the XSecure_Sha3 instance.
Hash	Pointer to a buffer in which read hash will be stored.

Returns

None

Note

None

s32 XSecure_Sha3PadSelection (XSecure_Sha3 * *InstancePtr,* XSecure_Sha3PadType *Sha3Type*)

This function provides an option to select the SHA-3 padding type to be used while calculating the hash.

Parameters

InstancePtr	Pointer to the XSecure_Sha3 instance.
Sha3Type	Type of the sha3 padding to be used. • For NIST SHA-3 padding - XSECURE_CSU_NIST_SHA3 • For KECCAK SHA-3 padding - XSECURE_CSU_KECCAK_SHA3

Returns

By default provides support for NIST SHA-3, if wants to change for Keccak SHA-3 this function should be called after XSecure_Sha3Initialize()

SHA-3 API Example Usage

The xilsecure_sha_example.c file is a simple example application that demonstrates the usage of SHA-3 device to calculate 384 bit hash on Hello World string. A more typical use case of calculating the hash of boot image as a step in authentication process using the SHA-3 device has been illustrated in the xilsecure_rsa_example.c.
The contents of the xilsecure_sha_example.c file are shown below:

```
int SecureHelloWorldExample()
{
  u8 HelloWorld[4] = {’h’,’e’,’l’,’l’};
  u32 Size = sizeof(HelloWorld);
  u8 Out[384/8];
  XCsuDma_Config *Config;

  int Status;

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

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

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

  XSecure_Sha3Digest(&Secure_Sha3, HelloWorld, Size, Out);

  xil_printf(" Calculated Digest \r\n ");
  int i= 0;
  for(i=0; i< (384/8); i++)
  {
    xil_printf(" %0x ", Out[i]);
  }
  xil_printf(" \r\n ");

  return XST_SUCCESS;
}
```

Note

The `xilsecure_sha_example.c` and `xilsecure_rsa_example.c` example files are available in the `<library-install-path>\examples` folder. Where `<library-install-path>` is the XilSecure library installation path.

Chapter 29

SHA-2

Overview

This is an algorithm which generates 256 bit hash on the input data.

Note

SHA-2 will be deprecated from release 2019.1 onwards. Xilinx recommends you to use SHA-3.

SHA-2 Function Usage

When all the data is available on which sha2 hash must be calculated, the `sha_256()` can be used with appropriate parameters, as described. When all the data is not available on which sha2 must be calculated, use the sha2 functions in the following order:

1. `sha2_starts()`
2. `sha2_update()` - This API can be called multiple times till input data is completed.
3. `sha2_finish()` - Provides the final hash of the data.

To get intermediate hash values after each `sha2_update()`, you can call `sha2_hash()` after the `sha2_update()` call.

Modules

- SHA-2 Example Usage

Functions

- void sha_256 (const unsigned char ∗in, const unsigned int size, unsigned char ∗out)
- void sha2_starts (sha2_context ∗ctx)
- void sha2_update (sha2_context ∗ctx, unsigned char ∗input, unsigned int ilen)
- void sha2_finish (sha2_context ∗ctx, unsigned char ∗output)
- void sha2_hash (sha2_context ∗ctx, unsigned char ∗output)

Function Documentation

void sha_256 (const unsigned char ∗ *in*, const unsigned int *size*, unsigned char ∗ *out*)

This function calculates the hash for the input data using SHA-256 algorithm. This function internally calls the sha2_init, updates and finishes functions and updates the result.

Parameters

In	Char pointer which contains the input data.
Size	Length of the input data
Out	Pointer to location where resulting hash will be written.

Returns

None

void sha2_starts (sha2_context ∗ *ctx*)

This function initializes the SHA2 context.

Parameters

ctx	Pointer to sha2_context structure that stores status and buffer.

Returns

None

void sha2_update (sha2_context ∗ *ctx*, unsigned char ∗ *input*, unsigned int *ilen*)

This function adds the input data to SHA256 calculation.

Parameters

ctx	Pointer to sha2_context structure that stores status and buffer.
input	Pointer to the data to add.
Out	Length of the input data.

Returns

None

void sha2_finish (sha2_context ∗ *ctx,* unsigned char ∗ *output*)

This function finishes the SHA calculation.

Parameters

ctx	Pointer to sha2_context structure that stores status and buffer.
output	Pointer to the calculated hash data.

Returns

None

void sha2_hash (sha2_context ∗ *ctx,* unsigned char ∗ *output*)

This function reads the SHA2 hash, it can be called intermediately of updates to read the SHA2 hash.

Parameters

ctx	Pointer to sha2_context structure that stores status and buffer.
output	Pointer to the calculated hash data.

Returns

None

SHA-2 Example Usage

The `xilsecure_sha2_example.c` file contains the implementation of the interface functions for SHA driver. When all the data is available on which sha2 must be calculated, the sha_256() function can be used with appropriate parameters, as described. But, when all the data is not available on which sha2 must be calculated, use the sha2 functions in the following order:

- sha2_update() can be called multiple times till input data is completed.
- sha2_context is updated by the library only; do not change the values of the context.

The contents of the `xilsecure_sha2_example.c` file are shown below:

```
u32 XSecure_Sha2_Hash_Gn()
{
  sha2_context Sha2;
  u8 Output_Hash[32];
  u8 IntermediateHash[32];
  u8 Cal_Hash[32];
  u32 Index;
  u32 Size = XSECURE_DATA_SIZE;
  u32 Status;
  /* Generating SHA2 hash */
  sha2_starts(&Sha2);
  sha2_update(&Sha2, (u8 *)Data, Size - 1);

  /* If required we can read intermediate hash */
  sha2_hash(&Sha2, IntermediateHash);
  xil_printf("Intermediate SHA2 Hash is: ");
  for (Index = 0; Index < 32; Index++) {
    xil_printf("%02x", IntermediateHash[Index]);
  }
  xil_printf("\n");

  sha2_finish(&Sha2, Output_Hash);

  xil_printf("Generated SHA2 Hash is: ");
  for (Index = 0; Index < 32; Index++) {
    xil_printf("%02x", Output_Hash[Index]);
  }
  xil_printf("\n");

  /* Convert expected Hash value into hexa */
  Status = XSecure_ConvertStringToHexBE(XSECURE_EXPECTED_SHA2_HASH,
              Cal_Hash, 64);
  if (Status != XST_SUCCESS) {
    xil_printf("Error: While converting expected "
        "string of SHA2 hash to hexa\n\r");
    return XST_FAILURE;
  }

  /* Compare generated hash with expected hash value */
  for (Index = 0; Index < 32; Index++) {
    if (Cal_Hash[Index] != Output_Hash[Index]) {
      xil_printf("Error: SHA2 Hash generated through "
      "XilSecure library does not match with "
      "expected hash value\n\r");
      return XST_FAILURE;
    }
  }

  return XST_SUCCESS;
}
```

Note

The xilsecure_sha2_example.c example file is available in the <library-install-path>\examples folder. Where <library-install-path> is the XilSecure library installation path.

XilRSA Library v1.5

Overview

The XilRSA library provides APIs to use RSA encryption and decryption algorithms and SHA algorithms for Zynq®-7000 SoC devices.

Note

The RSA-2048 bit is used for RSA and the SHA-256 bit is used for hash.

For an example on usage of this library, refer to the RSA Authentication application and its documentation.

Source Files

The following is a list of source files shipped as a part of the XilRSA library:

- librsa.a: Pre-compiled file which contains the implementation.
- xilrsa.h: This file contains the APIs for SHA2 and RSA-20148..

Usage of SHA-256 Functions

When all the data is available on which sha2 must be calculated, the sha_256() function can be used with appropriate parameters, as described. When all the data is not available on which sha2 must be calculated, use the sha2 functions in the following order:

1. sha2_update() can be called multiple times till input data is completed.
2. sha2_context is updated by the library only; do not change the values of the context.

SHA2 API Example Usage

```
sha2_context ctx;
sha2_starts(&ctx);
sha2_update(&ctx, (unsigned char *)in, size);
sha2_finish(&ctx, out);
```

Following is the source code of the sha2_context class.

```
typedef struct
{
  unsigned int state[8];
  unsigned char buffer[SHA_BLKBYTES];
  unsigned long long bytes;
} sha2_context;
```

Chapter 31

XilRSA APIs

Overview

This section provides detailed descriptions of the XilRSA library APIs.

Macros

- #define RSA_NUMBER RSA_DIGIT *

Functions

- void rsa2048_pubexp (RSA_NUMBER a, RSA_NUMBER x, unsigned long e, RSA_NUMBER m, RSA_NUMBER rrm)
- void sha_256 (const unsigned char *in, const unsigned int size, unsigned char *out)
- void sha2_starts (sha2_context *ctx)
- void sha2_update (sha2_context *ctx, unsigned char *input, unsigned int ilen)
- void sha2_finish (sha2_context *ctx, unsigned char *output)

Macro Definition Documentation

#define RSA_NUMBER RSA_DIGIT *

RSA_NUMBER is a pointer of type unsigned long.

Function Documentation

void rsa2048_pubexp (RSA_NUMBER *a*, RSA_NUMBER *x*, unsigned long *e*, RSA_NUMBER *m*, RSA_NUMBER *rrm*)

This function is used to perform RSA encryption using 2048 bit public key.

Parameters

a	A pointer to the buffer of size 256 bytes in which ouput data will be placed.
x	A pointer to the input data buffer of size 256 bytes.
e	A pointer to the public key exponent of size 64 bytes.
m	A pointer to the public key modulus of size 256 bytes
rrm	A pointer to the public key modulus extension of size 256 bytes.

Returns

None

Note

All the required public key components should be extracted by user from RSA key pair or point to the key components in authentication certificate in boot image which is created by bootgen.

void sha_256 (const unsigned char ∗ *in*, const unsigned int *size*, unsigned char ∗ *out*)

This function calculates the hash for the input data using SHA-256 algorithm. This function internally calls the sha2_init, updates and finishes functions and updates the result.

Parameters

In	Char pointer which contains the input data.
Size	Length of the input data
Out	Pointer to location where resulting hash will be written.

Returns

None

void sha2_starts (sha2_context ∗ *ctx*)

This function initializes the SHA2 context.

Parameters

ctx	Pointer to sha2_context structure that stores status and buffer.

Returns

None

void sha2_update (sha2_context $*$ *ctx,* unsigned char $*$ *input,* unsigned int *ilen*)

This function adds the input data to SHA256 calculation.

Parameters

ctx	Pointer to sha2_context structure that stores status and buffer.
input	Pointer to the data to add.
Out	Length of the input data.

Returns

None

void sha2_finish (sha2_context $*$ *ctx,* unsigned char $*$ *output*)

This function finishes the SHA calculation.

Parameters

ctx	Pointer to sha2_context structure that stores status and buffer.
output	Pointer to the calculated hash data.

Returns

None

XilSKey Library v6.6

Chapter 32

Overview

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

- In Zynq-7000 devices:
  - PS eFUSE holds the RSA primary key hash bits and user feature bits, which can enable or disable some Zynq-7000 processor features.
  - PL eFUSE holds the AES key, the user key and some of the feature bits.
  - PL BBRAM holds the AES key.
- In 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:
  - PS eFUSE holds the AES key, the user fuses, PPK0 and PPK1 hash, SPK ID and some user feature bits which can be used to enable or disable some Zynq UltraScale+ MPSoC features.
  - BBRAM holds the AES key.

Hardware Setup

This section describes the hardware setup required for programming PL BBRAM or PL eFUSE.

Hardware setup for Zynq PL

This chapter describes the hardware setup required for programming BBRAM or eFUSE of Zynq PL devices. PL eFUSE or PL BBRAM is accessed through PS via MIO pins which are used for communication PL eFUSE or PL BBRAM through JTAG signals, these can be changed depending on the hardware setup.
A hardware setup which dedicates four MIO pins for JTAG signals should be used and the MIO pins should be mentioned in application header file (xilskey_input.h). There should be a method to download this example and have the MIO pins connected to JTAG before running this application. You can change the listed pins at your discretion.

MUX Usage Requirements

To write the PL eFUSE or PL BBRAM using a driver you must:

- Use four MIO lines (TCK,TMS,TDO,TDI)
- Connect the MIO lines to a JTAG port

If you want to switch between the external JTAG and JTAG operation driven by the MIOs, you must:

- Include a MUX between the external JTAG and the JTAG operation driven by the MIOs
- Assign a MUX selection PIN

To rephrase, to select JTAG for PL EFUSE or PL BBRAM writing, you must define the following:

- The MIOs used for JTAG operations (TCK,TMS,TDI,TDO).
- The MIO used for the MUX Select Line.
- The Value on the MUX Select line, to select JTAG for PL eFUSE or PL BBRAM writing.

The following graphic illustrates the correct MUX usage.

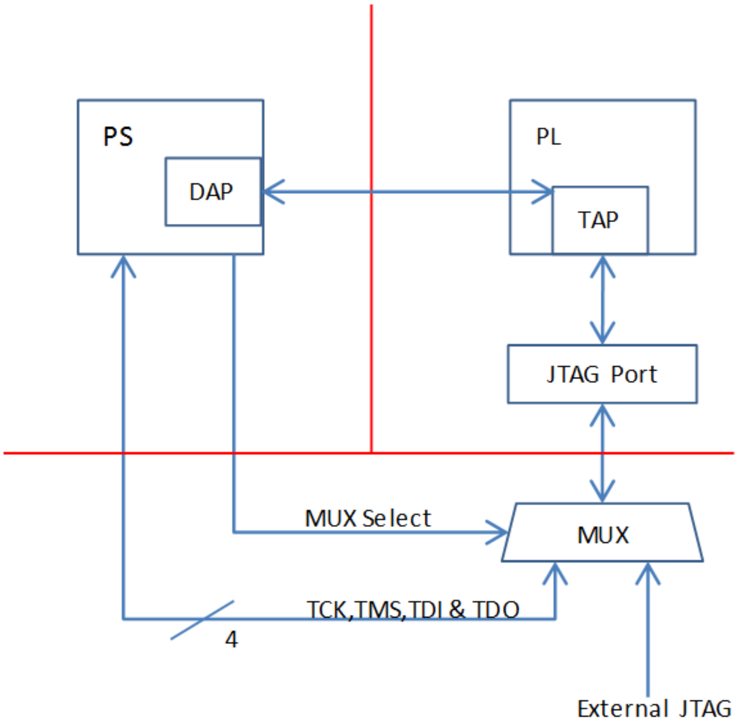

Figure 32.1: MUX Usage

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 chapter 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. 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.

WARNING: *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.

Chapter 33

BBRAM PL API

Overview

This chapter provides a linked summary and detailed descriptions of the battery-backed RAM (BBRAM) APIs of Zynq® PL and UltraScale™ devices.

Example Usage

- Zynq BBRAM PL example usage:
  - The Zynq BBRAM PL example application should contain the `xilskey_bbram_example.c` and `xilskey_input.h` files.
  - You should provide user configurable parameters in the `xilskey_input.h` file. For more information, refer Zynq User-Configurable PL BBRAM Parameters.
- UltraScale BBRAM example usage:
  - The UltraScale BBRAM example application should contain the `xilskey_bbram_ultrascale_input.h` and `xilskey_bbram_ultrascale_example.c` files.
  - You should provide user configurable parameters in the `xilskey_bbram_ultrascale_input.h` file. For more information, refer UltraScale or UltraScale+ User-Configurable BBRAM PL Parameters.

Note

It is assumed that you have set up your hardware prior to working on the example application. For more information, refer Hardware Setup.

Functions

- int XilSKey_Bbram_Program (XilSKey_Bbram ∗InstancePtr)

Function Documentation

int XilSKey_Bbram_Program (XilSKey_Bbram ∗ *InstancePtr*)

This function implements the BBRAM algorithm for programming and verifying key.
The program and verify will only work together in and in that order.

Parameters

InstancePtr	Pointer to XilSKey_Bbram

Returns

- XST_FAILURE - In case of failure
- XST_SUCCESS - In case of Success

Note

This function will program BBRAM of Ultrascale and Zynq as well.

Chapter 34

Zynq UltraScale+ MPSoC BBRAM PS API

Overview

This chapter provides a linked summary and detailed descriptions of the battery-backed RAM (BBRAM) APIs for Zynq® UltraScale+™ MPSoC devices.

Example Usage

- The Zynq UltraScale+ MPSoc example application should contain the `xilskey_bbramps_zynqmp_example.c` file.
- User configurable key can be modified in the same file (`xilskey_bbramps_zynqmp_example.c`), at the XSK_ZYNQMP_BBRAMPS_AES_KEY macro.

Functions

- u32 XilSKey_ZynqMp_Bbram_Program (u32 *AesKey)
- void XilSKey_ZynqMp_Bbram_Zeroise ()

Function Documentation

u32 XilSKey_ZynqMp_Bbram_Program (u32 * *AesKey*)

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.

Parameters

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.

void XilSKey_ZynqMp_Bbram_Zeroise ( )

This function zeroize's Bbram Key.

Parameters

None.	

Returns

None.

Note

BBRAM key will be zeroized.

Chapter 35

Zynq eFUSE PS API

Overview

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.

Functions

- u32 XilSKey_EfusePs_Write (XilSKey_EPs *PsInstancePtr)
- u32 XilSKey_EfusePs_Read (XilSKey_EPs *PsInstancePtr)
- u32 XilSKey_EfusePs_ReadStatus (XilSKey_EPs *InstancePtr, u32 *StatusBits)

Function Documentation

u32 XilSKey_EfusePs_Write (XilSKey_EPs * *InstancePtr*)

PS eFUSE interface functions.
PS eFUSE interface functions.

Parameters

InstancePtr	Pointer to the PsEfuseHandle which describes which PS eFUSE bit should be burned.

Returns

- XST_SUCCESS.
- Incase 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.

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

u32 XilSKey_EfusePs_Read (XilSKey_EPs * *InstancePtr*)

This function is used to read the PS eFUSE.

Parameters

InstancePtr	Pointer to the PsEfuseHandle which describes which PS eFUSE should be burned.

Returns

- XST_SUCCESS no errors occured.
- Incase 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.

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

u32 XilSKey_EfusePs_ReadStatus (XilSKey_EPs * *InstancePtr*, u32 * *StatusBits*)

This function is used to read the PS efuse status register.

Parameters

InstancePtr	Pointer to the PS eFUSE instance.
StatusBits	Buffer to store the status register read.

Returns

- XST_SUCCESS.
- XST_FAILURE

Note

This API unlocks the controller and reads the Zynq PS eFUSE status register.

Chapter 36

Zynq UltraScale+ MPSoC eFUSE PS API

Overview

This chapter provides a linked summary and detailed descriptions of the Zynq MPSoC UltraScale+ eFUSE PS APIs.

Example Usage

- For programming eFUSE other than 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 and programming PUF, the 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.

Data Structures

- struct XilSKey_Efuse

Functions

- 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_Write (XilSKey_ZynqMpEPs ∗InstancePtr)
- u32 XilSkey_ZynqMpEfuseAccess (const u32 AddrHigh, const u32 AddrLow)
- u32 XilSKey_ZynqMp_EfusePs_WritePufHelprData (XilSKey_Puf ∗InstancePtr)

- u32 XilSKey_ZynqMp_EfusePs_ReadPufHelprData (u32 ∗Address)
- u32 XilSKey_ZynqMp_EfusePs_WritePufChash (XilSKey_Puf ∗InstancePtr)
- u32 XilSKey_ZynqMp_EfusePs_ReadPufChash (u32 ∗Address, u8 ReadOption)
- u32 XilSKey_ZynqMp_EfusePs_WritePufAux (XilSKey_Puf ∗InstancePtr)
- u32 XilSKey_ZynqMp_EfusePs_ReadPufAux (u32 ∗Address, u8 ReadOption)
- u32 XilSKey_Write_Puf_EfusePs_SecureBits (XilSKey_Puf_Secure ∗WriteSecureBits)
- u32 XilSKey_Read_Puf_EfusePs_SecureBits (XilSKey_Puf_Secure ∗SecureBitsRead, u8 ReadOption)
- u32 XilSKey_Puf_Debug2 (XilSKey_Puf ∗InstancePtr)
- u32 XilSKey_Puf_Registration (XilSKey_Puf ∗InstancePtr)
- u32 XilSKey_Puf_Regeneration (XilSKey_Puf ∗InstancePtr)

Data Structure Documentation

struct XilSKey_Efuse

XilSKey_Efuse is the eFUSE access structure.
Using this structure, user can request the eFUSE access via IPI response.

Data Fields

u64	Src	Address of Data buffer.
u32	Size	Size in words.
u32	Offset	offset
u32	Flag	Flag - 0 : to read efuse and Flag - 1 : to write efuse.

Function Documentation

u32 XilSKey_ZynqMp_EfusePs_CheckAesKeyCrc (u32 *CrcValue*)

This function performs CRC check of AES key.

Parameters

CrcValue	A 32 bit CRC value of an expected AES key.

Returns

- XST_SUCCESS on successful CRC check.
- ErrorCode on failure

Note

For Calculating CRC of AES key use XilSKey_CrcCalculation() API or XilSkey_CrcCalculation_AesKey() API.

u32 XilSKey_ZynqMp_EfusePs_ReadUserFuse (u32 * *UseFusePtr*, u8 *UserFuse_Num*, u8 *ReadOption*)

This function is used to read user fuse from efuse or cache based on user's read option.

Parameters

UseFusePtr	Pointer to an array which holds the readback user fuse in.
UserFuse_Num	A variable which holds the user fuse number. Range is (User fuses: 0 to 7)
ReadOption	A variable which has to be provided by user based on this input reading is happend from cache or from efuse array. • 0 Reads from cache • 1 Reads from efuse array

Returns

- XST_SUCCESS on successful read
- ErrorCode on failure

u32 XilSKey_ZynqMp_EfusePs_ReadPpk0Hash (u32 * *Ppk0Hash*, u8 *ReadOption*)

This function is used to read PPK0 hash from efuse or cache based on user's read option.

Parameters

Ppk0Hash	A pointer to an array which holds the readback PPK0 hash in.
ReadOption	A variable which has to be provided by user based on this input reading is happend from cache or from efuse array. • 0 Reads from cache • 1 Reads from efuse array

Returns

- XST_SUCCESS on successful read
- ErrorCode on failure

u32 XilSKey_ZynqMp_EfusePs_ReadPpk1Hash (u32 * *Ppk1Hash*, u8 *ReadOption*)

This function is used to read PPK1 hash from efuse or cache based on user's read option.

Parameters

Ppk1Hash	Pointer to an array which holds the readback PPK1 hash in.
ReadOption	A variable which has to be provided by user based on this input reading is happend from cache or from efuse array. • 0 Reads from cache • 1 Reads from efuse array

Returns

- XST_SUCCESS on successful read
- ErrorCode on failure

u32 XilSKey_ZynqMp_EfusePs_ReadSpkId (u32 ∗ *SpkId,* u8 *ReadOption*)

This function is used to read SPKID from efuse or cache based on user's read option.

Parameters

SpkId	Pointer to a 32 bit variable which holds SPK ID.
ReadOption	A variable which has to be provided by user based on this input reading is happend from cache or from efuse array. • 0 Reads from cache • 1 Reads from efuse array

Returns

- XST_SUCCESS on successful read
- ErrorCode on failure

void XilSKey_ZynqMp_EfusePs_ReadDna (u32 ∗ *DnaRead*)

This function is used to read DNA from efuse.

Parameters

DnaRead	Pointer to 32 bit variable which holds the readback DNA in.

Returns

None.

u32 XilSKey_ZynqMp_EfusePs_ReadSecCtrlBits (XilSKey_SecCtrlBits * *ReadBackSecCtrlBits,* u8 *ReadOption*)

This function is used to read the PS efuse secure control bits from cache or eFUSE based on user input provided.

Parameters

ReadBackSecCtrlBits	Pointer to the XilSKey_SecCtrlBits which holds the read secure control bits.
ReadOption	Variable which has to be provided by user based on this input reading is happend from cache or from efuse array. • 0 Reads from cache • 1 Reads from efuse array

Returns

- XST_SUCCESS if reads successfully
- XST_FAILURE if reading is failed

Note

Cache reload is required for obtaining updated values for ReadOption 0.

u32 XilSKey_ZynqMp_EfusePs_Write (XilSKey_ZynqMpEPs * *InstancePtr*)

This function is used to program the PS efuse of ZynqMP, based on user inputs.

Parameters

InstancePtr	Pointer to the XilSKey_ZynqMpEPs.

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

Note

On successful efuse programming cache is been reloaded, so programmed efuse bits can directly read from efuse cache.

u32 XilSkey_ZynqMpEfuseAccess (const u32 *AddrHigh*, const u32 *AddrLow*)

This function provides access to efuse memory.

Parameters

AddrHigh	Higher 32-bit address of the XilSKey_Efuse structure.
AddrLow	Lower 32-bit address of the XilSKey_Efuse structure.

Returns

- XST_SUCCESS - On success
- ErrorCode - on Failure

Note

This function is for programming efuse through IPI response it has corresponding IPI response call in PMUFW.

u32 XilSKey_ZynqMp_EfusePs_WritePufHelprData (XilSKey_Puf ∗ *InstancePtr*)

This function programs the PS efuse's with puf helper data of ZynqMp.

Parameters

InstancePtr	Pointer to the XilSKey_Puf instance.

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

Note

To generate PufSyndromeData please use XilSKey_Puf_Registration API

u32 XilSKey_ZynqMp_EfusePs_ReadPufHelprData (u32 ∗ *Address*)

This function reads the puf helper data from eFUSE.

Parameters

Address	Pointer to data array which holds the Puf helper data read from ZynqMp efuse.

Returns

- XST_SUCCESS if reads successfully.
- Errorcode on failure.

Note

This function only reads from eFUSE non-volatile memory. There is no option to read from Cache.

u32 XilSKey_ZynqMp_EfusePs_WritePufChash (XilSKey_Puf ∗ *InstancePtr*)

This API programs eFUSE with CHash value.

Parameters

InstancePtr	Pointer to the XilSKey_Puf instance.

Returns

- XST_SUCCESS if chash is programmed successfully.
- Errorcode on failure

Note

To generate CHash value please use XilSKey_Puf_Registration API

u32 XilSKey_ZynqMp_EfusePs_ReadPufChash (u32 ∗ *Address*, u8 *ReadOption*)

This API reads efuse puf CHash Data from efuse array or cache based on the user read option.

Parameters

Address	Pointer which holds the read back value of chash
ReadOption	A u8 variable which has to be provided by user based on this input reading is happend from cache or from efuse array. • 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

Note

Cache reload is required for obtaining updated values for ReadOption 0.

u32 XilSKey_ZynqMp_EfusePs_WritePufAux (XilSKey_Puf ∗ *InstancePtr*)

This API programs efuse puf Auxilary Data.

Parameters

InstancePtr	Pointer to the XilSKey_Puf instance.

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

Note

To generate Auxilary data please use the below API u32 XilSKey_Puf_Registration(XilSKey_Puf ∗InstancePtr)

u32 XilSKey_ZynqMp_EfusePs_ReadPufAux (u32 ∗ *Address*, u8 *ReadOption*)

This API reads efuse puf Auxilary Data from efuse array or cache based on user read option.

Parameters

Address	Pointer which holds the read back value of Auxilary
ReadOption	A u8 variable which has to be provided by user based on this input reading is happend from cache or from efuse array. • 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

Note

Cache reload is required for obtaining updated values for ReadOption 0.

u32 XilSKey_Write_Puf_EfusePs_SecureBits (XilSKey_Puf_Secure * *WriteSecureBits*)

This function programs the eFUSE PUF secure bits.

Parameters

WriteSecureBits	Pointer to the XilSKey_Puf_Secure structure

Returns

XST_SUCCESS - On success ErrorCode - on Failure

u32 XilSKey_Read_Puf_EfusePs_SecureBits (XilSKey_Puf_Secure * *SecureBitsRead,* u8 *ReadOption*)

This function is used to read the PS efuse PUF secure bits from cache or from eFUSE array based on user selection.

Parameters

SecureBits	Pointer to the XilSKey_Puf_Secure which holds the read eFUSE secure bits of PUF.
ReadOption	A u8 variable which has to be provided by user based on this input reading is happened from cache or from efuse array. • 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from efuse array

Returns

XST_SUCCESS - On success ErrorCode - on Failure

u32 XilSKey_Puf_Debug2 (XilSKey_Puf * *InstancePtr*)

PUF Debug 2 operation.

Parameters

InstancePtr	Pointer to the XilSKey_Puf instance.

Returns

- XST_SUCCESS if debug 2 mode was successful.
- ERROR if registration was unsuccessful.

Note

Updates the Debug 2 mode result @ InstancePtr->Debug2Data

u32 XilSKey_Puf_Registration (XilSKey_Puf ∗ *InstancePtr*)

PUF Registration/Re-registration.

Parameters

InstancePtr	Pointer to the XilSKey_Puf instance.

Returns

- XST_SUCCESS if registration/re-registration was successful.
- ERROR if registration was unsuccessful

Note

Updates the syndrome data @ InstancePtr->SyndromeData

u32 XilSKey_Puf_Regeneration (XilSKey_Puf ∗ *InstancePtr*)

PUF Re-generation.

Parameters

InstancePtr	Pointer to the XilSKey_Puf instance.

Returns

- XST_SUCCESS if regeneration was successful.
- ERROR if regeneration was unsuccessful

Chapter 37

eFUSE PL API

Overview

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.

Functions

- u32 XilSKey_EfusePl_Program (XilSKey_EPl ∗PlInstancePtr)
- u32 XilSKey_EfusePl_ReadStatus (XilSKey_EPl ∗InstancePtr, u32 ∗StatusBits)
- u32 XilSKey_EfusePl_ReadKey (XilSKey_EPl ∗InstancePtr)

Function Documentation

u32 XilSKey_EfusePl_Program (XilSKey_EPl ∗ *InstancePtr*)

Programs PL eFUSE with input data given through InstancePtr.

Parameters

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

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.

u32 XilSKey_EfusePl_ReadStatus (XilSKey_EPl * *InstancePtr,* u32 * *StatusBits*)

Reads the PL efuse status bits and gets all secure and control bits.

Parameters

InstancePtr	Pointer to PL eFUSE instance.
StatusBits	Buffer to store the status bits read.

Returns

- XST_FAILURE - In case of failure
- XST_SUCCESS - In case of Success

u32 XilSKey_EfusePl_ReadKey (XilSKey_EPl * *InstancePtr*)

Reads the PL efuse keys and stores them in the corresponding arrays in instance structure.

Parameters

InstancePtr	Pointer to PL eFUSE instance.

Returns

- XST_FAILURE - In case of failure
- XST_SUCCESS - In case of Success

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()

Chapter 38

CRC Calculation API

Overview

This chapter provides a linked summary and detailed descriptions of the CRC calculation APIs.
For UltraScale and Zynq UltraScale+ MPSoC devices, programmed AES cannot be read back. The programmed AES key can only be confirmed by doing CRC check of AES key.

Functions

- u32 XilSKey_CrcCalculation (u8 ∗Key)
- u32 XilSkey_CrcCalculation_AesKey (u8 ∗Key)

Function Documentation

u32 XilSKey_CrcCalculation (u8 ∗ *Key*)

Calculates CRC value of provided key, this API expects key in string format.

Parameters

Key	Pointer to the string contains AES key in hexa decimal 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

Note

This API calculates CRC of AES key for Ultrascale Microblaze's PL eFuse and ZynqMp UltraScale PS eFuse. If length of the string provided is lesser than 64, API appends the string with zeros.

u32 XilSkey_CrcCalculation_AesKey (u8 * *Key*)

Calculates CRC value of the provided key.
Key should be provided in hexa buffer.

Parameters

Key	Pointer to an array of size 32 which contains AES key in hexa decimal.

Returns

Crc of provided AES key value.

Note

This API calculates CRC of AES key for Ultrascale Microblaze's PL eFuse and ZynqMp Ultrascale's PS eFuse. This API calculates CRC on AES key provided in hexa format. To calculate CRC on the AES key in string format please use XilSKey_CrcCalculation. To call this API one can directly pass array of AES key which exists in an instance. Example for storing key into Buffer: If Key is "123456" buffer should be {0x12 0x34 0x56}

Chapter 39

User-Configurable Parameters

Overview

This chapter provides detailed descriptions of the various user configurable parameters.

Modules

- Zynq User-Configurable PS eFUSE Parameters
- Zynq User-Configurable PL eFUSE Parameters
- Zynq User-Configurable PL BBRAM Parameters
- UltraScale or UltraScale+ User-Configurable BBRAM PL Parameters
- UltraScale or UltraScale+ User-Configurable PL eFUSE Parameters
- Zynq UltraScale+ MPSoC User-Configurable PS eFUSE Parameters
- Zynq UltraScale+ MPSoC User-Configurable PS BBRAM Parameters
- Zynq UltraScale+ MPSoC User-Configurable PS PUF 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.

Macro Name	Description
XSK_EFUSEPS_RSA_KEY_HASH_VALUE	Default = 0000000000000000000000000000000000 000000000000000000000000000000 The specified value is converted to a hexadecimal buffer and written into the PS eFUSE array when the write API is used. This value should be the Primary Public Key (PPK) hash provided in string format. The buffer must be 64 characters long: valid characters are 0-9, a-f, and A-F. Any other character is considered an invalid string and will not burn RSA hash. When the `Xilskey_EfusePs_Write()` API is used, the RSA hash is written, and the XSK_EFUSEPS_ENABLE_RSA_KEY_HASH must have a value of TRUE.
XSK_EFUSEPS_DISABLE_DFT_JTAG	Default = FALSE TRUE disables DFT JTAG permanently. FALSE will not modify the eFuse PS DFT JTAG disable bit.
XSK_EFUSEPS_DISABLE_DFT_MODE	Default = FALSE TRUE disables DFT mode permanently. FALSE will not modify the eFuse PS DFT mode disable bit.

Zynq User-Configurable PL eFUSE Parameters

Overview

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.

Macro Name	Description
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.

Modules

- MIO Pins for Zynq PL eFUSE JTAG Operations
- MUX Selection Pin for Zynq PL eFUSE JTAG Operations
- MUX Parameter for Zynq PL eFUSE JTAG Operations
- AES and User Key Parameters

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	(18)
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	(21)	This pin toggles between the external JTAG or MIO driving JTAG operations.

MUX Parameter for Zynq PL eFUSE JTAG Operations

The table below lists the MUX parameter.

Parameter Name	Description
XSK_EFUSEPL_MIO_MUX_SEL_DEFAULT_VAL	Default = LOW. LOW writes zero on the MUX select line before PL_eFUSE writing. HIGH writes one on the MUX select line before PL_eFUSE writing.

AES and User Key Parameters

The table below lists the AES and user key parameters.

Parameter Name	Description
XSK_EFUSEPL_PROGRAM_AES_AND_USER_LOW_KEY	Default = FALSE. TRUE burns the AES and User Low hash key, which are given in the XSK_EFUSEPL_AES_KEY and the XSK_EFUSEPL_USER_LOW_KEY respectively. FALSE ignores the provided values. You cannot write the AES Key and the User Low Key separately.
XSK_EFUSEPL_PROGRAM_USER_HIGH_KEY	Default =FALSE. TRUE burns the User High hash key, given in XSK_EFUSEPL_PROGRAM_USER_HIGH_KEY. FALSE ignores the provided values.
XSK_EFUSEPL_AES_KEY	Default = 0000000000000000000000000000000000000000000000000000000000000000 This value converted to hex buffer and written into the PL eFUSE array when write API is used. This value should be the AES Key, given in string format. It must be 64 characters long. Valid characters are 0-9, a-f, A-F. Any other character is considered an invalid string and will not burn AES Key. To write AES Key, XSK_EFUSEPL_PROGRAM_AES_AND_USER_LOW_KEY must have a value of TRUE.
XSK_EFUSEPL_USER_LOW_KEY	Default = 00 This value is converted to a hexadecimal buffer and written into the PL eFUSE array when the write API is used. This value is the User Low Key given in string format. It must be two characters long; valid characters are 0-9,a-f, and A-F. Any other character is considered as an invalid string and will not burn the User Low Key. To write the User Low Key, XSK_EFUSEPL_PROGRAM_AES_AND_USER_LOW_KEY must have a value of TRUE.

Parameter Name	Description
XSK_EFUSEPL_USER_HIGH_KEY	Default = 000000 The default value is converted to a hexadecimal buffer and written into the PL eFUSE array when the write API is used. This value is the User High Key given in string format. The buffer must be six characters long: valid characters are 0-9, a-f, A-F. Any other character is considered to be an invalid string and does not burn User High Key. To write the User High Key, the XSK_EFUSEPL_PROGRAM_USER_HIGH_KEY must have a value of TRUE.

Zynq User-Configurable PL BBRAM Parameters

Overview

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)

Modules

- MUX Parameter for Zynq BBRAM PL JTAG Operations
- AES and User Key Parameters

MUX Parameter for Zynq BBRAM PL JTAG Operations

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

Parameter Name	Description
XSK_BBRAM_MIO_MUX_SEL_DEFAULT_VAL	Default = LOW. LOW writes zero on the MUX select line before PL_eFUSE writing. HIGH writes one on the MUX select line before PL_eFUSE writing.

AES and User Key Parameters

The table below lists the AES and user key parameters.

Parameter Name	Description
XSK_BBRAM_AES_KEY	Default = XX. AES key (in HEX) that must be programmed into BBRAM.
XSK_BBRAM_AES_KEY_SIZE_IN_BITS	Default = 256. Size of AES key. Must be 256 bits.

UltraScale or UltraScale+ User-Configurable BBRAM PL Parameters

Overview

Following parameters need to be configured.
Based on your inputs, BBRAM is programmed with the provided AES key.

Modules

- AES Keys and Related Parameters
- DPA Protection for BBRAM key
- GPIO Pins Used for PL Master JTAG and HWM Signals
- GPIO Channels

AES Keys and Related Parameters

The following table shows AES key related parameters.

Parameter Name	Description
XSK_BBRAM_PGM_OBFUSCATED_KEY	Default = FALSE By default XSK_BBRAM_PGM_OBFUSCATED_KEY is FALSE, BBRAM is programmed with a non-obfuscated key provided in XSK_BBRAM_AES_KEY and DPA protection can be either in enabled/disabled state. TRUE programs the BBRAM with key provided in XSK_BBRAM_OBFUSCATED_KEY and DPA protection cannot be enabled.
XSK_BBRAM_OBFUSCATED_KEY	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 should have TRUE value.
XSK_BBRAM_AES_KEY	Default = 0000000000000000524156a63950bcedafeadcdeabaadee34216615aaaabbaaa The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM,when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM. **Note** For programming BBRAM with the key, XSK_BBRAM_PGM_OBFUSCATED_KEY should have a 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 Pins Used for PL Master JTAG and HWM Signals

In Ultrascale the following GPIO pins are used for connecting MASTER_JTAG pins to access BBRAM. These can be changed depending on your hardware.The table below shows the GPIO pins used for PL MASTER JTAG signals.

Master JTAG Signal	Default PIN Number
XSK_BBRAM_AXI_GPIO_JTAG_TDO	0
XSK_BBRAM_AXI_GPIO_JTAG_TDI	0
XSK_BBRAM_AXI_GPIO_JTAG_TMS	1
XSK_BBRAM_AXI_GPIO_JTAG_TCK	2

GPIO Channels

The following table shows GPIO channel number.

Parameter	Default Channel Number	Master JTAG Signal Connected
XSK_BBRAM_GPIO_INPUT_CH	2	TDO
XSK_BBRAM_GPIO_OUTPUT_CH	1	TDI, TMS, TCK

Note

> All inputs and outputs of GPIO can 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

Overview

The table below lists the user-configurable PL eFUSE parameters for UltraScale™ devices.

Macro Name	Description
XSK_EFUSEPL_DISABLE_AES_KEY_READ	Default = FALSE TRUE will permanently disable the write to FUSE_AES and check CRC for AES key by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_USER_KEY_READ	Default = FALSE TRUE will permanently disable the write to 32 bit FUSE_USER and read of FUSE_USER key by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_SECURE_READ	Default = FALSE TRUE will permanently disable the write to FUSE_Secure block and reading of secure block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_FUSE_CNTRL_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_CNTRL block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.

Macro Name	Description
XSK_EFUSEPL_DISABLE_RSA_KEY_READ	Default = FALSE. TRUE will permanently disable the write to FUSE_RSA block and reading of FUSE_RSA Hash by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_KEY_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_AES block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_USER_KEY_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_USER block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_SECURE_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_SECURE block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_RSA_HASH_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_RSA authentication key by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_128BIT_USER_KEY _WRITE	Default = FALSE. TRUE will permanently disable the write to 128 bit FUSE_USER by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_ALLOW_ENCRYPTED_ONLY	Default = FALSE. TRUE will permanently allow encrypted bitstream only. FALSE will not modify this Secure bit of eFuse.
XSK_EFUSEPL_FORCE_USE_FUSE_AES_ONLY	Default = FALSE. TRUE then allows only FUSE's AES key as source of encryption FALSE then allows FPGA to configure an unencrypted bitstream or bitstream encrypted using key stored BBRAM or eFuse.
XSK_EFUSEPL_ENABLE_RSA_AUTH	Default = FALSE. TRUE will enable RSA authentication of bitstream FALSE will not modify this secure bit of eFuse.

Macro Name	Description
XSK_EFUSEPL_DISABLE_JTAG_CHAIN	Default = FALSE. TRUE will disable JTAG permanently. FALSE will not modify this secure bit of eFuse.
XSK_EFUSEPL_DISABLE_TEST_ACCESS	Default = FALSE. TRUE will disables Xilinx test access. FALSE will not modify this secure bit of eFuse.
XSK_EFUSEPL_DISABLE_AES_DECRYPTOR	Default = FALSE. TRUE will disables decoder completely. FALSE will not modify this secure bit of eFuse.
XSK_EFUSEPL_ENABLE_OBFUSCATION_EFUSEAES	Default = FALSE. TRUE will enable obfuscation feature for eFUSE AES key.

Modules

- GPIO Pins Used for PL Master JTAG Signal
- GPIO Channels
- AES Keys and Related Parameters

GPIO Pins Used for PL Master JTAG Signal

In Ultrascale the following GPIO pins are used for connecting MASTER_JTAG pins to access BBRAM. These can be changed depending on your hardware.The table below shows the GPIO pins used for PL MASTER JTAG signals.

Master JTAG Signal	Default PIN Number
XSK_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 can 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.

AES Keys and Related Parameters

The following table shows AES key related parameters.

Parameter Name	Description
XSK_EFUSEPL_PROGRAM_AES_KEY_ULTRA	Default = FALSE TRUE will burn the AES key given in XSK_EFUSEPL_AES_KEY. FALSE will ignore the values given.
XSK_EFUSEPL_PROGRAM_USER_KEY_ULTRA	Default = FALSE TRUE will burn 32 bit User key given in XSK_EFUSEPL_USER_KEY. FALSE will ignore the values given.
XSK_EFUSEPL_PROGRAM_RSA_HASH_ULTRA	Default = FALSE TRUE will burn RSA hash given in XSK_EFUSEPL_RSA_KEY_HASH_VALUE. FALSE will ignore the values given.
XSK_EFUSEPL_PROGRAM_USER_KEY_128BIT	Default = FALSE TRUE will burn 128 bit User key given in XSK_EFUSEPL_USER_KEY_128BIT_0, XSK_EFUSEPL_USER_KEY_128BIT_1, XSK_EFUSEPL_USER_KEY_128BIT_2, XSK_EFUSEPL_USER_KEY_128BIT_3 FALSE will ignore the values given.

Parameter Name	Description
XSK_EFUSEPL_CHECK_AES_KEY_ULTRA	Default = FALSE TRUE will perform CRC check of FUSE_AES with provided CRC value in macro XSK_EFUSEPL_CRC_OF_EXPECTED_AES_KEY. And result of CRC check will be updated in XilSKey_EPl instance parameter AESKeyMatched with either TRUE or FALSE. FALSE CRC check of FUSE_AES will not be performed.
XSK_EFUSEPL_READ_USER_KEY_ULTRA	Default = FALSE TRUE will read 32 bit FUSE_USER from UltraScale eFUSE and updates in XilSKey_EPl instance parameter UserKeyReadback. FALSE 32-bit FUSE_USER key read will not be performed.
XSK_EFUSEPL_READ_RSA_HASH_ULTRA	Default = FALSE TRUE will read FUSE_USER from UltraScale eFUSE and updates in XilSKey_EPl instance parameter RSAHashReadback. FALSE FUSE_RSA_HASH read will not be performed.
XSK_EFUSEPL_READ_USER_KEY128_BIT	Default = FALSE TRUE will read 128 bit USER key from UltraScale eFUSE and updates in XilSKey_EPl instance parameter User128BitReadBack. FALSE 128 bit USER key read will not be performed.
XSK_EFUSEPL_AES_KEY	Default = 0000000000000000000000000000000000000000000000000000000000000000 The value mentioned in this will be converted to hex buffer and written into the PL eFUSE array when write API used. This value should be the PPK(Primary Public Key) hash 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 that for writing the AES Key, XSK_EFUSEPL_PROGRAM_AES_KEY_ULTRA should have TRUE value.

Parameter Name	Description
XSK_EFUSEPL_USER_KEY	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the PL eFUSE array when write API used. This value should be the User Key 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 User Key. Note that, for writing the User Key, XSK_EFUSEPL_PROGRAM_USER_KEY_ULTRA should have TRUE value.
XSK_EFUSEPL_RSA_KEY_HASH_VALUE	Default = 000000000000000000000000000000000 0000000000000000000000000000000000000000 00000000000000000000000 he value mentioned in this will be converted to hex buffer and written into the PL eFUSE array when write API used. This value should be the RSA Key hash given in string format. It should be 96 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn RSA hash value. Note that, for writing the RSA hash, XSK_EFUSEPL_PROGRAM_RSA_HASH_ULTRA should have TRUE value.
XSK_EFUSEPL_CRC_OF_EXPECTED_AES_KEY	Default = XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS CRC value of FUSE_AES with all Zeros. Expected FUSE_AES key's CRC value 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 or reverse polynomial 0x82F63B78. 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_

Zynq UltraScale+ MPSoC User-Configurable PS eFUSE Parameters

Overview

The table below lists the user-configurable PS eFUSE parameters for Zynq UltraScale+ MPSoC devices.

Macro Name	Description
XSK_EFUSEPS_AES_RD_LOCK	Default = FALSE TRUE will permanently disable the CRC check of FUSE_AES. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_AES_WR_LOCK	Default = FALSE TRUE will permanently disable the writing to FUSE_AES block. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_ENC_ONLY	Default = FALSE TRUE will permanently enable encrypted booting only using the Fuse key. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_BBRAM_DISABLE	Default = FALSE TRUE will permanently disable the BBRAM key. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_ERR_DISABLE	Default = FALSE TRUE will permanently disables the error messages in JTAG status register. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_JTAG_DISABLE	Default = FALSE TRUE will permanently disable JTAG controller. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_DFT_DISABLE	Default = FALSE TRUE will permanently disable DFT boot mode. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PROG_GATE_DISABLE	Default = FALSE TRUE will permanently disable PROG_GATE feature in PPD. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_SECURE_LOCK	Default = FALSE TRUE will permanently disable reboot into JTAG mode when doing a secure lockdown. FALSE will not modify thi s control bit of eFuse.

Macro Name	Description
XSK_EFUSEPS_RSA_ENABLE	Default = FALSE TRUE will permanently enable RSA authentication during boot. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PPK0_WR_LOCK	Default = FALSE TRUE will permanently disable writing to PPK0 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PPK0_INVLD	Default = FALSE TRUE will permanently revoke PPK0. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PPK1_WR_LOCK	Default = FALSE TRUE will permanently disable writing PPK1 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PPK1_INVLD	Default = FALSE TRUE will permanently revoke PPK1. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_0	Default = FALSE TRUE will permanently disable writing to USER_0 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_1	Default = FALSE TRUE will permanently disable writing to USER_1 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_2	Default = FALSE TRUE will permanently disable writing to USER_2 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_3	Default = FALSE TRUE will permanently disable writing to USER_3 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_4	Default = FALSE TRUE will permanently disable writing to USER_4 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_5	Default = FALSE TRUE will permanently disable writing to USER_5 efuses. FALSE will not modify this control bit of eFuse.

Macro Name	Description
XSK_EFUSEPS_USER_WRLK_6	Default = FALSE TRUE will permanently disable writing to USER_6 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_7	Default = FALSE TRUE will permanently disable writing to USER_7 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_LBIST_EN	Default = FALSE TRUE will permanently enables logic BIST to be run during boot. FALSE will not modify this control bit of eFUSE.
XSK_EFUSEPS_LPD_SC_EN	Default = FALSE TRUE will permanently enables zeroization of registers in Low Power Domain(LPD) during boot. FALSE will not modify this control bit of eFUSE.
XSK_EFUSEPS_FPD_SC_EN	Default = FALSE TRUE will permanently enables zeroization of registers in Full Power Domain(FPD) during boot. FALSE will not modify this control bit of eFUSE.
XSK_EFUSEPS_PBR_BOOT_ERR	Default = FALSE TRUE will permanently enables the boot halt when there is any PMU error. FALSE will not modify this control bit of eFUSE.

Modules

- AES Keys and Related Parameters
- User Keys and Related Parameters
- PPK0 Keys and Related Parameters
- PPK1 Keys and Related Parameters
- SPK ID and Related Parameters

AES Keys and Related Parameters

The following table shows AES key related parameters.

Parameter Name	Description
XSK_EFUSEPS_WRITE_AES_KEY	Default = TRUE TRUE will burn the AES key provided in XSK_EFUSEPS_AES_KEY. FALSE will ignore the key provide XSK_EFUSEPS_AES_KEY.

Parameter Name	Description
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.

User Keys and Related Parameters

Single bit programming is allowed for all the USER fuses.
If user requests to revert already programmed bit. Library throws an error. If user fuses is non-zero also 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 = TRUE TRUE will burn User0 Fuse provided in XSK_EFUSEPS_USER0_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER0_FUSES
XSK_EFUSEPS_WRITE_USER1_FUSE	Default = TRUE TRUE will burn User1 Fuse provided in XSK_EFUSEPS_USER1_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER1_FUSES
XSK_EFUSEPS_WRITE_USER2_FUSE	Default = TRUE TRUE will burn User2 Fuse provided in XSK_EFUSEPS_USER2_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER2_FUSES
XSK_EFUSEPS_WRITE_USER3_FUSE	Default = TRUE TRUE will burn User3 Fuse provided in XSK_EFUSEPS_USER3_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER3_FUSES

Parameter Name	Description
XSK_EFUSEPS_WRITE_USER4_FUSE	Default = TRUE TRUE will burn User4 Fuse provided in XSK_EFUSEPS_USER4_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER4_FUSES
XSK_EFUSEPS_WRITE_USER5_FUSE	Default = TRUE TRUE will burn User5 Fuse provided in XSK_EFUSEPS_USER5_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER5_FUSES
XSK_EFUSEPS_WRITE_USER6_FUSE	Default = TRUE TRUE will burn User6 Fuse provided in XSK_EFUSEPS_USER6_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER6_FUSES
XSK_EFUSEPS_WRITE_USER7_FUSE	Default = TRUE TRUE will burn User7 Fuse provided in XSK_EFUSEPS_USER7_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER7_FUSES
XSK_EFUSEPS_USER0_FUSES	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID. **Note** For writing the User0 Fuse, XSK_EFUSEPS_WRITE_USER0_FUSE should have TRUE value

Parameter Name	Description
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

Parameter Name	Description
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 = TRUE TRUE will burn PPK0 sha3 hash provided in XSK_EFUSEPS_PPK0_SHA3_HASH. FALSE will ignore the hash provided in XSK_EFUSEPS_PPK0_SHA3_HASH.
XSK_EFUSEPS_PPK0_IS_SHA3	Default = TRUE TRUE XSK_EFUSEPS_PPK0_SHA3_HASH should be of string length 96 it specifies that PPK0 is used to program SHA3 hash. FALSE XSK_EFUSEPS_PPK0_SHA3_HASH should be of string length 64 it specifies that PPK0 is used to program SHA2 hash.

Parameter Name	Description
XSK_EFUSEPS_PPK0_HASH	Default = 0000000000000000000000000000000000 00000000000000000000000000000000000000000 000000000000000000000 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 = TRUE TRUE will burn PPK1 sha3 hash provided in XSK_EFUSEPS_PPK1_SHA3_HASH. FALSE will ignore the hash provided in XSK_EFUSEPS_PPK1_SHA3_HASH.
XSK_EFUSEPS_PPK1_IS_SHA3	Default = FALSE 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 = TRUE TRUE will burn SPKID provided in XSK_EFUSEPS_SPK_ID. FALSE will ignore the hash provided in XSK_EFUSEPS_SPK_ID.
XSK_EFUSEPS_SPK_ID	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID. **Note** For writing the SPK ID, XSK_EFUSEPS_WRITE_SPKID should have TRUE value.

Note

PPK hash should be unmodified hash generated by bootgen. Single bit programming is allowed for User FUSEs (0 to 7), if you specify a value that tries to set a bit that was previously programmed to 1 back to 0, you will get an error. you have to provide already programmed bits also along with new requests.

Zynq UltraScale+ MPSoC User-Configurable PS BBRAM Parameters

The table below lists the AES and user key parameters.

Parameter Name	Description
XSK_ZYNQMP_BBRAMPS_AES_KEY	Default = 0000000000000000000000000000000000 000000000000000000000000000000 AES 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 will not program data into eFUSE.
XSK_PUF_IF_CONTRACT_MANUFACTURER	Default = FALSE This should be enabled when application is hand over to contract manufacturer. TRUE will allow only authenticated application. FALSE authentication is not mandatory.
XSK_PUF_REG_MODE	Default = XSK_PUF_MODE4K PUF registration is performed in 4K mode. For only understanding it is provided in this file, but user is not supposed to modify this.

Macro Name	Description
XSK_PUF_READ_SECUREBITS	Default = FALSE TRUE will read status of the puf secure bits from eFUSE and will be displayed on UART. FALSE will not read secure bits.
XSK_PUF_PROGRAM_SECUREBITS	Default = FALSE TRUE will program PUF secure bits based on the user input provided at XSK_PUF_SYN_INVALID, XSK_PUF_SYN_WRLK and XSK_PUF_REGISTER_DISABLE. FALSE will not program any PUF secure bits.
XSK_PUF_SYN_INVALID	Default = FALSE TRUE will permanently invalidate the already programmed syndrome data. FALSE will not modify anything
XSK_PUF_SYN_WRLK	Default = FALSE TRUE will permanently disable programming syndrome data into eFUSE. FALSE will not modify anything.
XSK_PUF_REGISTER_DISABLE	Default = FALSE TRUE permanently does not allow PUF syndrome data registration. FALSE will not modify anything.
XSK_PUF_RESERVED	Default = FALSE TRUE programs this reserved eFUSE bit. FALSE will not modify anything.
XSK_PUF_AES_KEY	Default = 0000000000000000000000000000000000000000000000000000000000000000 The value mentioned in this will be converted to hex buffer and encrypts this with PUF helper data and generates a black key and written into the Zynq UltraScale+ MPSoC PS eFUSE array when XSK_PUF_PROGRAM_EFUSE macro is TRUE. This value should be given in string format. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn AES Key. Note Provided here should be red key and application calculates the black key and programs into eFUSE if XSK_PUF_PROGRAM_EFUSE macro is TRUE. To avoid programming eFUSE results can be displayed on UART com port by making XSK_PUF_INFO_ON_UART to TRUE.

Macro Name	Description
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.

Chapter 40

Error Codes

Overview

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

Modules

- PL eFUSE Error Codes
- PS eFUSE Error Codes
- Zynq UltraScale+ MPSoC BBRAM PS Error Codes

PL eFUSE Error Codes

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 br>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.

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.

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

XSK_EFUSEPS_ERROR_NONE 0
No error.

XSK_EFUSEPS_ERROR_ADDRESS_XIL_RESTRICTED 0x01
Address is restricted.

XSK_EFUSEPS_ERROR_READ_TMEPERATURE_OUT_OF_RANGE 0x02
Temperature obtained from XADC is out of range.

XSK_EFUSEPS_ERROR_READ_VCCPAUX_VOLTAGE_OUT_OF_RANGE 0x03
VCCAUX obtained from XADC is out of range.

XSK_EFUSEPS_ERROR_READ_VCCPINT_VOLTAGE_OUT_OF_RANGE 0x04
VCCINT obtained from XADC is out of range.

XSK_EFUSEPS_ERROR_WRITE_TEMPERATURE_OUT_OF_RANGE 0x05
Temperature obtained from XADC is out of range.

XSK_EFUSEPS_ERROR_WRITE_VCCPAUX_VOLTAGE_OUT_OF_RANGE 0x06
VCCAUX obtained from XADC is out of range.

XSK_EFUSEPS_ERROR_WRITE_VCCPINT_VOLTAGE_OUT_OF_RANGE 0x07
VCCINT obtained from XADC is out of range.

XSK_EFUSEPS_ERROR_VERIFICATION 0x08
Verification error.

XSK_EFUSEPS_ERROR_RSA_HASH_ALREADY_PROGRAMMED 0x09
RSA hash was already programmed.

XSK_EFUSEPS_ERROR_CONTROLLER_MODE 0x0A
Controller mode error

XSK_EFUSEPS_ERROR_REF_CLOCK 0x0B
Reference clock not between 20 to 60MHz

XSK_EFUSEPS_ERROR_READ_MODE 0x0C
Not supported read mode

XSK_EFUSEPS_ERROR_XADC_CONFIG 0x0D
XADC configuration error.

XSK_EFUSEPS_ERROR_XADC_INITIALIZE 0x0E
XADC initialization error.

XSK_EFUSEPS_ERROR_XADC_SELF_TEST 0x0F
XADC self-test failed.

XSK_EFUSEPS_ERROR_PARAMETER_NULL Utils Error Codes. 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_PROGRAMMING_TBIT_PATTERN 0x16
Error in programming TBITS.

XSK_EFUSEPS_ERROR_BEFORE_PROGRAMMING 0x0080
Error occurred before programming.

XSK_EFUSEPS_ERROR_PROGRAMMING 0x00A0
Error in programming eFUSE.

XSK_EFUSEPS_ERROR_READ 0x00B0
Error in reading.

XSK_EFUSEPS_ERROR_BYTES_REQUEST 0x00C0
Error in requested byte count.

XSK_EFUSEPS_ERROR_RESRVD_BITS_PRGRMG 0x00D0
Error in programming reserved bits.

XSK_EFUSEPS_ERROR_ADDR_ACCESS 0x00E0
Error in accessing requested address.

XSK_EFUSEPS_ERROR_PS_STRUCT_NULL XSKEfuse_Write/Read()common error codes. 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.

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_CACHE_LOAD 0xB000
Error in re-loading CACHE.

XSK_EFUSEPS_ERROR_WRITE_USER0_FUSE 0xC000
Error in programming USER 0 Fuses.

XSK_EFUSEPS_ERROR_WRITE_USER1_FUSE 0xC100
Error in programming USER 1 Fuses.

XSK_EFUSEPS_ERROR_WRITE_USER2_FUSE 0xC200
Error in programming USER 2 Fuses.

XSK_EFUSEPS_ERROR_WRITE_USER3_FUSE 0xC300
Error in programming USER 3 Fuses.

XSK_EFUSEPS_ERROR_WRITE_USER4_FUSE 0xC400
Error in programming USER 4 Fuses.

XSK_EFUSEPS_ERROR_WRITE_USER5_FUSE 0xC500
Error in programming USER 5 Fuses.

XSK_EFUSEPS_ERROR_WRITE_USER6_FUSE 0xC600
Error in programming USER 6 Fuses.

XSK_EFUSEPS_ERROR_WRITE_USER7_FUSE 0xC700
Error in programming USER 7 Fuses.

XSK_EFUSEPS_ERROR_WRTIE_USER0_LK_BIT 0xC800
Error in programming USER 0 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USER1_LK_BIT 0xC900
Error in programming USER 1 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USER2_LK_BIT 0xCA00
Error in programming USER 2 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USER3_LK_BIT 0xCB00
Error in programming USER 3 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USER4_LK_BIT 0xCC00
Error in programming USER 4 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USER5_LK_BIT 0xCD00
Error in programming USER 5 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USER6_LK_BIT 0xCE00
Error in programming USER 6 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USER7_LK_BIT 0xCF00
Error in programming USER 7 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_PROG_GATE0_DIS_BIT 0xD000
Error in programming PROG_GATE0 disabling bit.

XSK_EFUSEPS_ERROR_WRTIE_PROG_GATE1_DIS_BIT 0xD100
Error in programming PROG_GATE1 disabling bit.

XSK_EFUSEPS_ERROR_WRTIE_PROG_GATE2_DIS_BIT 0xD200
Error in programming PROG_GATE2 disabling bit.

XSK_EFUSEPS_ERROR_WRTIE_SEC_LOCK_BIT 0xD300
Error in programming SEC_LOCK bit.

XSK_EFUSEPS_ERROR_WRTIE_PPK0_WR_LK_BIT 0xD400
Error in programming PPK0 write lock bit.

XSK_EFUSEPS_ERROR_WRTIE_PPK0_RVK_BIT 0xD500
Error in programming PPK0 revoke bit.

XSK_EFUSEPS_ERROR_WRTIE_PPK1_WR_LK_BIT 0xD600
Error in programming PPK1 write lock bit.

XSK_EFUSEPS_ERROR_WRTIE_PPK1_RVK_BIT 0xD700
Error in programming PPK0 revoke bit.

XSK_EFUSEPS_ERROR_WRITE_PUF_SYN_INVLD 0xD800
Error while programming the PUF syndrome invalidate bit.

XSK_EFUSEPS_ERROR_WRITE_PUF_SYN_WRLK 0xD900
Error while programming Syndrome write lock bit.

XSK_EFUSEPS_ERROR_WRITE_PUF_SYN_REG_DIS 0xDA00
Error while programming PUF syndrome register disable bit.

XSK_EFUSEPS_ERROR_WRITE_PUF_RESERVED_BIT 0xDB00
Error while programming PUF reserved bit.

XSK_EFUSEPS_ERROR_WRITE_LBIST_EN_BIT 0xDC00
Error while programming LBIST enable bit.

XSK_EFUSEPS_ERROR_WRITE_LPD_SC_EN_BIT 0xDD00
Error while programming LPD SC enable bit.

XSK_EFUSEPS_ERROR_WRITE_FPD_SC_EN_BIT 0xDE00
Error while programming FPD SC enable bit.

XSK_EFUSEPS_ERROR_WRITE_PBR_BOOT_ERR_BIT 0xDF00
Error while programming PBR boot error bit.

XSK_EFUSEPS_ERROR_PUF_INVALID_REG_MODE 0xE000
Error when PUF registration is requested with invalid registration mode.

XSK_EFUSEPS_ERROR_PUF_REG_WO_AUTH 0xE100
Error when write not allowed without authentication enabled.

XSK_EFUSEPS_ERROR_PUF_REG_DISABLED 0xE200
Error when trying to do PUF registration and when PUF registration is disabled.

XSK_EFUSEPS_ERROR_PUF_INVALID_REQUEST 0xE300
Error when an invalid mode is requested.

XSK_EFUSEPS_ERROR_PUF_DATA_ALREADY_PROGRAMMED 0xE400
Error when PUF is already programmed in eFUSE.

XSK_EFUSEPS_ERROR_PUF_DATA_OVERFLOW 0xE500
Error when an over flow occurs.

XSK_EFUSEPS_ERROR_CMPLTD_EFUSE_PRGRM_WITH_ERR 0x10000
eFUSE programming is completed with temp and vol read errors.

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.

Zynq UltraScale+ MPSoC BBRAM PS Error Codes

XSK_ZYNQMP_BBRAMPS_ERROR_NONE 0
No error.

XSK_ZYNQMP_BBRAMPS_ERROR_IN_PRGRMG_ENABLE 0x01
If this error is occurred programming is not possible.

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.

Chapter 41

Status Codes

For Zynq® and UltraScale™, the status in the `xilskey_efuse_example.c` file is conveyed through a UART or reboot status register in the following format: `0xYYYYZZZZ`, where:

- YYYY represents the PS eFUSE Status.
- ZZZZ represents the PL eFUSE Status.

The table below lists the status codes.

Status Code Values	**Description**
0x0000ZZZZ	Represents PS eFUSE is successful and PL eFUSE process returned with error.
0xYYYY0000	Represents PL eFUSE is successful and PS eFUSE process returned with error.
0xFFFF0000	Represents PS eFUSE is not initiated and PL eFUSE is successful.
0x0000FFFF	Represents PL eFUSE is not initiated and PS eFUSE is successful.
0xFFFFZZZZ	Represents PS eFUSE is not initiated and PL eFUSE is process returned with error.
0xYYYYFFFF	Represents PL eFUSE is not initiated and PS eFUSE is process returned with error.

For Zynq UltraScale+ MPSoC, the status in the `xilskey_bbramps_zynqmp_example.c`, `xilskey_puf_registration.c` and `xilskey_efuseps_zynqmp_example.c` files is conveyed as 32 bit error code. Where Zero represents that no error has occurred and if the value is other than Zero, a 32 bit error code is returned.

Chapter 42

Procedures

This chapter provides detailed descriptions of the various procedures.

Zynq eFUSE Writing Procedure Running from DDR as an Application

This sequence is same as the existing flow described below.

1. Provide the required inputs in `xilskey_input.h`, then compile the SDK project.
2. Take the latest FSBL (ELF), stitch the `<output>.elf` generated to it (using the bootgen utility), and generate a bootable image.
3. Write the generated binary image into the flash device (for example: QSPI, NAND).
4. To burn the eFUSE key bits, execute the image.

Zynq eFUSE Driver Compilation Procedure for OCM

The procedure is as follows:

1. Open the linker script (`lscript.ld`) in the SDK project.
2. Map all the chapters 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 chapter 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

   ```
   $Outfilename
   ```

3. Update memory has to be done using the tcl command updatemem.

   ```
   updatemem -meminfo $file.mmi -data $Outfilename.elf -bit $design.bit
   -proc design_1_i/microblaze_0 -out $Final.bit
   ```

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

   ```
   write_mem_info $Outfilename
   ```

3. Update memory has to be done using the tcl command updatemem:

   ```
   updatemem -meminfo $file.mmi -data $Outfilename.elf -bit $design.bit
   -proc design_1_i/microblaze_0 -out $Final.bit
   ```

4. Program the board using `$Final.bit bitstream`.
5. Output can be seen in UART terminal.

XilPM Library v2.4

Chapter 43

XilPM APIs

Overview

Xilinx Power Management(XilPM) provides Embedded Energy Management Interface (EEMI) APIs for power management on Zynq® UltraScale+™ MPSoC. For more details about power management on Zynq Ultrascale+ MPSoC, see the Zynq UltraScale+ MPSoC Power Management User Guide (UG1199). For more details about EEMI, see the Embedded Energy Management Interface (EEMI) API User Guide(UG1200).

Modules

- Error Status

Data Structures

- struct XPm_Notifier
- struct XPm_NodeStatus
- struct XPmClockSel2ClkIn
- struct XPmClockMux
- struct XPmClockModel

Enumerations

- enum XPmApiId
- enum XPmApiCbId
- enum XPmNodeId
- enum XPmRequestAck
- enum XPmAbortReason
- enum XPmSuspendReason
- enum XPmRamState
- enum XPmOpCharType
- enum XPmBootStatus
- enum XPmResetAction
- enum XPmReset
- enum XPmNotifyEvent
- enum XPmClock

Functions

- XStatus XPm_InitXilpm (XIpiPsu ∗IpiInst)
- void XPm_SuspendFinalize (void)
- enum XPmBootStatus XPm_GetBootStatus (void)
- XStatus XPm_RequestSuspend (const enum XPmNodeId target, const enum XPmRequestAck ack, const u32 latency, const u8 state)
- XStatus XPm_SelfSuspend (const enum XPmNodeId nid, const u32 latency, const u8 state, const u64 address)
- XStatus XPm_ForcePowerDown (const enum XPmNodeId target, const enum XPmRequestAck ack)
- XStatus XPm_AbortSuspend (const enum XPmAbortReason reason)
- XStatus XPm_RequestWakeUp (const enum XPmNodeId target, const bool setAddress, const u64 address, const enum XPmRequestAck ack)
- XStatus XPm_SetWakeUpSource (const enum XPmNodeId target, const enum XPmNodeId wkup_node, const u8 enable)
- XStatus XPm_SystemShutdown (u32 type, u32 subtype)
- XStatus XPm_SetConfiguration (const u32 address)
- XStatus XPm_InitFinalize (void)
- 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 u32 event, const u32 oppoint)
- XStatus XPm_RequestNode (const enum XPmNodeId node, const u32 capabilities, const u32 qos, const enum XPmRequestAck ack)
- XStatus XPm_ReleaseNode (const enum XPmNodeId node)
- XStatus XPm_SetRequirement (const enum XPmNodeId nid, const u32 capabilities, const u32 qos, const enum XPmRequestAck ack)
- XStatus XPm_SetMaxLatency (const enum XPmNodeId node, const u32 latency)
- XStatus XPm_GetApiVersion (u32 ∗version)
- XStatus XPm_GetNodeStatus (const enum XPmNodeId node, XPm_NodeStatus ∗const nodestatus)
- XStatus XPm_RegisterNotifier (XPm_Notifier ∗const notifier)
- XStatus XPm_UnregisterNotifier (XPm_Notifier ∗const notifier)
- XStatus XPm_GetOpCharacteristic (const enum XPmNodeId node, const enum XPmOpCharType type, u32 ∗const result)
- XStatus XPm_ResetAssert (const enum XPmReset reset, const enum XPmResetAction assert)
- XStatus XPm_ResetGetStatus (const enum XPmReset reset, u32 ∗status)
- XStatus XPm_MmioWrite (const u32 address, const u32 mask, const u32 value)
- XStatus XPm_MmioRead (const u32 address, u32 ∗const value)
- XStatus XPm_ClockEnable (const enum XPmClock clock)
- XStatus XPm_ClockDisable (const enum XPmClock clock)
- XStatus XPm_ClockGetStatus (const enum XPmClock clock, u32 ∗const status)
- XStatus XPm_ClockGetDivider (const enum XPmClock clock, u32 ∗const divider)
- XStatus XPm_ClockSetParent (const enum XPmClock clock, const enum XPmClock parent)
- XStatus XPm_ClockGetParent (const enum XPmClock clock, enum XPmClock ∗const parent)

- XStatus XPm_ClockSetRate (const enum XPmClock clock, const u32 rate)
- XStatus XPm_ClockGetRate (const enum XPmClock clock, u32 ∗const rate)
- XStatus XPm_PllSetParameter (const enum XPmNodeId node, const enum XPmPllParam parameter, const u32 value)
- XStatus XPm_PllGetParameter (const enum XPmNodeId node, const enum XPmPllParam parameter, u32 ∗const value)
- XStatus XPm_PllSetMode (const enum XPmNodeId node, const enum XPmPllMode mode)
- XStatus XPm_PllGetMode (const enum XPmNodeId node, enum XPmPllMode ∗const mode)
- XStatus XPm_PinCtrlRequest (const u32 pin)
- XStatus XPm_PinCtrlRelease (const u32 pin)
- XStatus XPm_PinCtrlSetFunction (const u32 pin, const enum XPmPinFn fn)
- XStatus XPm_PinCtrlGetFunction (const u32 pin, enum XPmPinFn ∗const fn)
- XStatus XPm_PinCtrlSetParameter (const u32 pin, const enum XPmPinParam param, const u32 value)
- XStatus XPm_PinCtrlGetParameter (const u32 pin, const enum XPmPinParam param, u32 ∗const value)

PM Version Number macros

- #define **PM_VERSION_MAJOR** 1
- #define **PM_VERSION_MINOR** 1
- #define **PM_VERSION** ((PM_VERSION_MAJOR << 16) | PM_VERSION_MINOR)

Capabilities for RAM

- #define **PM_CAP_ACCESS** 0x1U
- #define **PM_CAP_CONTEXT** 0x2U
- #define **PM_CAP_WAKEUP** 0x4U

Node default states macros

- #define **NODE_STATE_OFF** 0
- #define **NODE_STATE_ON** 1

Processor's states macros

- #define **PROC_STATE_FORCEDOFF** 0
- #define **PROC_STATE_ACTIVE** 1
- #define **PROC_STATE_SLEEP** 2
- #define **PROC_STATE_SUSPENDING** 3

Maximum Latency/QOS macros

- #define **MAX_LATENCY** (~0U)
- #define **MAX_QOS** 100U

System shutdown/Restart macros

- #define **PMF_SHUTDOWN_TYPE_SHUTDOWN** 0U
- #define **PMF_SHUTDOWN_TYPE_RESET** 1U
- #define **PMF_SHUTDOWN_SUBTYPE_SUBSYSTEM** 0U
- #define **PMF_SHUTDOWN_SUBTYPE_PS_ONLY** 1U
- #define **PMF_SHUTDOWN_SUBTYPE_SYSTEM** 2U

PM API Min and Max macros

- #define **PM_API_MIN** PM_GET_API_VERSION

Payload Packets

Assigning of argument values into array elements. pause and pm_dbg are used for debugging and should be removed in final version.

- #define **PACK_PAYLOAD**(pl, arg0, arg1, arg2, arg3, arg4, arg5)
- #define **PACK_PAYLOAD0**(pl, api_id) PACK_PAYLOAD(pl, api_id, 0, 0, 0, 0, 0)
- #define **PACK_PAYLOAD1**(pl, api_id, arg1) PACK_PAYLOAD(pl, api_id, arg1, 0, 0, 0, 0)
- #define **PACK_PAYLOAD2**(pl, api_id, arg1, arg2) PACK_PAYLOAD(pl, api_id, arg1, arg2, 0, 0, 0)
- #define **PACK_PAYLOAD3**(pl, api_id, arg1, arg2, arg3) PACK_PAYLOAD(pl, api_id, arg1, arg2, arg3, 0, 0)
- #define **PACK_PAYLOAD4**(pl, api_id, arg1, arg2, arg3, arg4) PACK_PAYLOAD(pl, api_id, arg1, arg2, arg3, arg4, 0)
- #define **PACK_PAYLOAD5**(pl, api_id, arg1, arg2, arg3, arg4, arg5) PACK_PAYLOAD(pl, api_id, arg1, arg2, arg3, arg4, arg5)

Data Structure Documentation

struct XPm_Notifier

XPm_Notifier - Notifier structure registered with a callback by app

Data Fields

- void(∗const callback)(XPm_Notifier ∗const notifier)
- enum XPmNodeId node
- enum XPmNotifyEvent event
- u32 flags
- volatile u32 oppoint
- volatile u32 received
- XPm_Notifier ∗ next

Field Documentation

void(∗const callback) (XPm_Notifier ∗const notifier) 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)

enum XPmNodeId node Node argument (the node to receive notifications about)

enum XPmNotifyEvent event Event argument (the event type to receive notifications about)

u32 flags Flags

volatile u32 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.

volatile u32 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.

XPm_Notifier∗ next Pointer to next notifier in linked list. Must not be modified while the notifier is registered. User shall not ever modify this value.

struct XPm_NodeStatus

XPm_NodeStatus - struct containing node status information

Data Fields

- u32 status
- u32 requirements
- u32 usage

Field Documentation

u32 status Node power state

u32 requirements Current requirements asserted on the node (slaves only)

u32 usage Usage information (which master is currently using the slave)

struct XPmClockSel2ClkIn

Pair of multiplexer select value and selected clock input

Data Fields

- enum XPmClock clkIn
- const u8 select

Field Documentation

enum XPmClock clkIn ID of the clock that is selected with the 'select' value

const u8 select Select value of the clock multiplexer

struct XPmClockMux

MUX select values to clock input mapping

Data Fields

- const XPmClockSel2ClkIn *const inputs
- const u8 size
- const u8 bits
- const u8 shift

Field Documentation

const XPmClockSel2ClkIn* const inputs Mux select to pll mapping at the input of the multiplexer

const u8 size Size of the inputs array

const u8 bits Number of bits of mux select

const u8 shift Number of bits to shift 'bits' in order to get mux select mask

struct XPmClockModel

Clock model

Data Fields

- enum XPmClock id
- const XPmClockMux *const mux
- const u8 type
- const XPmClockModel *const next

Field Documentation

enum XPmClock id Clock ID

const XPmClockMux* const mux Pointer to the mux model

const u8 type Type specifying the available divisors

const XPmClockModel* const next Next clock in the list

Enumeration Type Documentation

enum XPmApiId

APIs for Miscellaneous functions, suspending of PUs, managing PM slaves and Direct control.

enum XPmApiCbId

PM API Callback Id Enum

enum XPmNodeId

PM Node ID Enum

enum XPmRequestAck

PM Acknowledge Request Types

enum XPmAbortReason

PM Abort Reasons Enum

enum XPmSuspendReason

PM Suspend Reasons Enum

enum XPmRamState

PM RAM States Enum

enum XPmOpCharType

PM Operating Characteristic types Enum

enum XPmBootStatus

Boot Status Enum

enum XPmResetAction

PM Reset Action types

enum XPmReset

PM Reset Line IDs

enum XPmNotifyEvent

PM Notify Events Enum

enum XPmClock

PM Clock IDs

Function Documentation

XStatus XPm_InitXilpm (XIpiPsu * *IpiInst*)

Initialize xilpm library.

Parameters

IpiInst	Pointer to IPI driver instance

Returns

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

Note

None

void XPm_SuspendFinalize (void)

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.

enum XPmBootStatus XPm_GetBootStatus (void)

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.

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.

Note

None

XStatus XPm_RequestSuspend (const enum XPmNodeId *target*, const enum XPmRequestAck *ack*, const u32 *latency*, const u8 *state*)

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.

Parameters

target	Node ID of the PU node to be suspended
ack	Requested acknowledge type. REQUEST_ACK_BLOCKING is not supported.
latency	Maximum wake-up latency requirement in us(micro sec)
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

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.

XStatus XPm_SelfSuspend (const enum XPmNodeId *nid*, const u32 *latency*, const u8 *state*, const u64 *address*)

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.

Parameters

nid	Node ID of the CPU node to be suspended.
latency	Maximum wake-up latency requirement in us(microsecs)
state	Instead of specifying a maximum latency, a CPU can also explicitly request a certain power state.
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

Note

This is a blocking call, it will return only once PMU has responded

XStatus XPm_ForcePowerDown (const enum XPmNodeId *target,* const enum XPmRequestAck *ack*)

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.

Parameters

target	Node ID of the PU node or power island/domain to be powered down.
ack	Requested acknowledge type

Returns

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

Note

Force power down may not be requested by a PU for itself.

XStatus XPm_AbortSuspend (const enum XPmAbortReason *reason*)

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.

Parameters

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

Note

Calling PU expects the PMU to abort the initiated suspend procedure. This is a non-blocking call without any acknowledge.

XStatus XPm_RequestWakeUp (const enum XPmNodeId *target*, const bool *setAddress*, const u64 *address*, const enum XPmRequestAck *ack*)

This function can be used to request power up of a CPU node within the same PU, or to power up another PU.

Parameters

target	Node ID of the CPU or PU to be powered/woken up.
setAddress	Specifies whether the start address argument is being passed. • 0 : do not set start address • 1 : set start address
address	Address from which to resume when woken up. Will only be used if set_address is 1.
ack	Requested acknowledge type

Returns

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

Note

If acknowledge is requested, the calling PU will be notified by the power management controller once the wake-up is completed.

XStatus XPm_SetWakeUpSource (const enum XPmNodeId *target*, const enum XPmNodeId *wkup_node*, const u8 *enable*)

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.

Parameters

target	Node ID of the target to be woken up.
wkup_node	Node ID of the wakeup device.
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

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.

XStatus XPm_SystemShutdown (u32 *type*, u32 *subtype*)

This function can be used by a privileged PU to shut down or restart the complete device.

Parameters

restart	Should the system be restarted automatically? • PM_SHUTDOWN : no restart requested, system will be powered off permanently • PM_RESTART : restart is requested, system will go through a full reset

Returns

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

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.

XStatus XPm_SetConfiguration (const u32 *address*)

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.

Parameters

address	Start address of the configuration object

Returns

XST_SUCCESS if successful, otherwise an error code

Note

The provided address must be in 32-bit address space which is accessible by the PMU.

XStatus XPm_InitFinalize (void)

This function is called to notify the power management controller about the completed power management initialization.

Returns

XST_SUCCESS if successful, otherwise an error code

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.

void XPm_InitSuspendCb (const enum XPmSuspendReason *reason*, const u32 *latency*, const u32 *state*, const u32 *timeout*)

Callback function to be implemented in each PU, allowing the power management controller to request that the PU suspend itself.

Parameters

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
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.
state	Targeted sleep/suspend state.
timeout	Timeout in ms, specifying how much time a PU has to initiate its suspend procedure before it's being considered unresponsive.

Returns

None

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.

void XPm_AcknowledgeCb (const enum XPmNodeId *node*, const XStatus *status*, const u32 *oppoint*)

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.

Parameters

node	ID of the component or sub-system in question.
status	Status of the operation: • OK: the operation completed successfully • ERR: the requested operation failed
oppoint	Operating point of the node in question

Returns

None

Note

None

void XPm_NotifyCb (const enum XPmNodeId *node,* const u32 *event,* const u32 *oppoint*)

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.

Parameters

node	ID of the node the event notification is related to.
event	ID of the event
oppoint	Current operating state of the node.

Returns

None

Note

None

XStatus XPm_RequestNode (const enum XPmNodeId *node,* const u32 *capabilities,* const u32 *qos,* const enum XPmRequestAck *ack*)

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.

Parameters

node	Node ID of the PM slave requested
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
qos	Quality of Service (0-100) required
ack	Requested acknowledge type

Returns

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

Note

None

XStatus XPm_ReleaseNode (const enum XPmNodeId *node*)

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.

Parameters

node	Node ID of the PM slave.

Returns

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

Note

None

XStatus XPm_SetRequirement (const enum XPmNodeId *nid,* const u32 *capabilities,* const u32 *qos,* const enum XPmRequestAck *ack*)

This function is used by a PU to announce a change in requirements for a specific slave node which is currently in use.

Parameters

nid	Node ID of the PM slave.
capabilities	Slave-specific capabilities required.
qos	Quality of Service (0-100) required.
ack	Requested acknowledge type

Returns

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

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).

XStatus XPm_SetMaxLatency (const enum XPmNodeId *node*, const u32 *latency*)

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.

Parameters

node	Node ID of the PM slave.
latency	Maximum wake-up latency required.

Returns

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

Note

Setting maximum wake-up latency can constrain the set of possible power states a resource can be put into.

XStatus XPm_GetApiVersion (u32 ∗ *version*)

This function is used to request the version number of the API running on the power management controller.

Parameters

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

Note

None

XStatus XPm_GetNodeStatus (const enum XPmNodeId *node*, XPm_NodeStatus *const *nodestatus*)

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.

Parameters

node	ID of the component or sub-system in question.
nodestatus	Used to return the complete status of the node.

- status - The current power state of the requested node.
  - For CPU nodes:
    - 0 : if CPU is powered down,
    - 1 : if CPU is active (powered up),
    - 2 : if CPU is suspending (powered up)
  - For power islands and power domains:
    - 0 : if island is powered down,
    - 1 : if island is powered up
  - For PM slaves:
    - 0 : if slave is powered down,
    - 1 : if slave is powered up,
    - 2 : if slave is in retention
- requirement - Slave nodes only: Returns current requirements the requesting PU has requested of the node.
- usage - Slave nodes only: Returns current usage status of the node:
  - 0 : node is not used by any PU,
  - 1 : node is used by caller exclusively,
  - 2 : node is used by other PU(s) only,
  - 3 : node is used by caller and by other PU(s)

Returns

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

Note

None

XStatus XPm_RegisterNotifier (XPm_Notifier *const *notifier*)

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.

Parameters

notifier	Pointer to the notifier object to be associated with the requested notification. The notifier object contains the following data related to the notification:

- 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.

Returns

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

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.

XStatus XPm_UnregisterNotifier (XPm_Notifier *const *notifier*)

A PU calls this function to unregister for the previously requested notifications.

Parameters

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

Note

None

XStatus XPm_GetOpCharacteristic (const enum XPmNodeId *node*, const enum XPmOpCharType *type*, u32 *const *result*)

Call this function to request the power management controller to return information about an operating characteristic of a component.

Parameters

node	ID of the component or sub-system in question.
type	Type of operating characteristic requested: • power (current power consumption), • latency (current latency in us to return to active state), • temperature (current temperature),
result	Used to return the requested operating characteristic.

Returns

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

Note

None

XStatus XPm_ResetAssert (const enum XPmReset *reset*, const enum XPmResetAction *assert*)

This function is used to assert or release reset for a particular reset line. Alternatively a reset pulse can be requested as well.

Parameters

reset	ID of the reset line
assert	Identifies action: • PM_RESET_ACTION_RELEASE : release reset, • PM_RESET_ACTION_ASSERT : assert reset, • PM_RESET_ACTION_PULSE : pulse reset,

Returns

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

Note

None

XStatus XPm_ResetGetStatus (const enum XPmReset *reset,* u32 ∗ *status*)

Call this function to get the current status of the selected reset line.

Parameters

reset	Reset line
status	Status of specified reset (true - asserted, false - released)

Returns

Returns 1/XST_FAILURE for 'asserted' or 0/XST_SUCCESS for 'released'.

Note

None

XStatus XPm_MmioWrite (const u32 *address,* const u32 *mask,* const u32 *value*)

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.

Parameters

address	Physical 32-bit address of memory mapped register to write to.
mask	32-bit value used to limit write to specific bits in the register.
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

Note

If the access isn't permitted this function returns an error code.

XStatus XPm_MmioRead (const u32 *address*, u32 ∗const *value*)

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.

Parameters

address	Physical 32-bit address of memory mapped register to read from.
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

Note

If the access isn't permitted this function returns an error code.

XStatus XPm_ClockEnable (const enum XPmClock *clock*)

Call this function to enable (activate) a clock.

Parameters

clock	Identifier of the target clock to be enabled

Returns

Status of performing the operation as returned by the PMU-FW

Note

If the access isn't permitted this function returns an error code.

XStatus XPm_ClockDisable (const enum XPmClock *clock*)

Call this function to disable (gate) a clock.

Parameters

clock	Identifier of the target clock to be disabled

Returns

Status of performing the operation as returned by the PMU-FW

Note

If the access isn't permitted this function returns an error code.

XStatus XPm_ClockGetStatus (const enum XPmClock *clock*, u32 ∗const *status*)

Call this function to get status of a clock gate state.

Parameters

clock	Identifier of the target clock
status	Location to store clock gate state (1=enabled, 0=disabled)

Returns

Status of performing the operation as returned by the PMU-FW

XStatus XPm_ClockGetDivider (const enum XPmClock *clock*, u32 ∗const *divider*)

Call this function to get divider of a clock.

Parameters

clock	Identifier of the target clock
divider	Location to store the divider value

Returns

XST_INVALID_PARAM or status of performing the operation as returned by the PMU-FW

XStatus XPm_ClockSetParent (const enum XPmClock *clock*, const enum XPmClock *parent*)

Call this function to set parent for a clock.

Parameters

clock	Identifier of the target clock
parent	Identifier of the target parent clock

Returns

XST_INVALID_PARAM or status of performing the operation as returned by the PMU-FW.

Note

If the access isn't permitted this function returns an error code.

XStatus XPm_ClockGetParent (const enum XPmClock *clock*, enum XPmClock ∗const *parent*)

Call this function to get parent of a clock.

Parameters

clock	Identifier of the target clock
parent	Location to store clock parent ID

Returns

XST_INVALID_PARAM or status of performing the operation as returned by the PMU-FW.

XStatus XPm_ClockSetRate (const enum XPmClock *clock*, const u32 *rate*)

Call this function to set rate of a clock.

Parameters

clock	Identifier of the target clock
rate	Clock frequency (rate) to be set

Returns

Status of performing the operation as returned by the PMU-FW

Note

Interface exists but the functionality is not supported in the current release.

XStatus XPm_ClockGetRate (const enum XPmClock *clock*, u32 ∗const *rate*)

Call this function to get rate of a clock.

Parameters

clock	Identifier of the target clock
rate	Location where the rate should be stored

Returns

Status of performing the operation as returned by the PMU-FW

Note

Interface exists but the functionality is not supported in the current release.

XStatus XPm_PllSetParameter (const enum XPmNodeId *node*, const enum XPmPllParam *parameter*, const u32 *value*)

Call this function to set a PLL parameter.

Parameters

node	PLL node identifier
parameter	PLL parameter identifier
value	Value of the PLL parameter

Returns

Status of performing the operation as returned by the PMU-FW

Note

If the access isn't permitted this function returns an error code.

XStatus XPm_PllGetParameter (const enum XPmNodeId *node*, const enum XPmPllParam *parameter*, u32 ∗const *value*)

Call this function to get a PLL parameter.

Parameters

node	PLL node identifier
parameter	PLL parameter identifier
value	Location to store value of the PLL parameter

Returns

Status of performing the operation as returned by the PMU-FW

XStatus XPm_PllSetMode (const enum XPmNodeId *node*, const enum XPmPllMode *mode*)

Call this function to set a PLL mode.

Parameters

node	PLL node identifier
mode	PLL mode to be set

Returns

Status of performing the operation as returned by the PMU-FW

Note

If the access isn't permitted this function returns an error code.

XStatus XPm_PllGetMode (const enum XPmNodeId *node*, enum XPmPllMode ∗const *mode*)

Call this function to get a PLL mode.

Parameters

node	PLL node identifier
mode	Location to store the PLL mode

Returns

Status of performing the operation as returned by the PMU-FW

XStatus XPm_PinCtrlRequest (const u32 *pin*)

Call this function to request a pin control.

Parameters

pin	PIN identifier (index from range 0-77)

Returns

Status of performing the operation as returned by the PMU-FW

XStatus XPm_PinCtrlRelease (const u32 *pin*)

Call this function to release a pin control.

Parameters

pin	PIN identifier (index from range 0-77)

Returns

Status of performing the operation as returned by the PMU-FW

XStatus XPm_PinCtrlSetFunction (const u32 *pin,* const enum XPmPinFn *fn*)

Call this function to set a pin function.

Parameters

pin	Pin identifier
fn	Pin function to be set

Returns

Status of performing the operation as returned by the PMU-FW

Note

If the access isn't permitted this function returns an error code.

XStatus XPm_PinCtrlGetFunction (const u32 *pin*, enum XPmPinFn ∗const *fn*)

Call this function to get currently configured pin function.

Parameters

pin	PLL node identifier
fn	Location to store the pin function

Returns

Status of performing the operation as returned by the PMU-FW

XStatus XPm_PinCtrlSetParameter (const u32 *pin*, const enum XPmPinParam *param*, const u32 *value*)

Call this function to set a pin parameter.

Parameters

pin	Pin identifier
param	Pin parameter identifier
value	Value of the pin parameter to set

Returns

Status of performing the operation as returned by the PMU-FW

Note

If the access isn't permitted this function returns an error code.

XStatus XPm_PinCtrlGetParameter (const u32 *pin*, const enum XPmPinParam *param*, u32 ∗const *value*)

Call this function to get currently configured value of pin parameter.

Parameters

pin	Pin identifier
param	Pin parameter identifier
value	Location to store value of the pin parameter

Returns

Status of performing the operation as returned by the PMU-FW

Error Status

Overview

This section lists the Power management specific return error statuses.

Macros

- #define XST_PM_INTERNAL 2000L
- #define XST_PM_CONFLICT 2001L
- #define XST_PM_NO_ACCESS 2002L
- #define XST_PM_INVALID_NODE 2003L
- #define XST_PM_DOUBLE_REQ 2004L
- #define XST_PM_ABORT_SUSPEND 2005L
- #define XST_PM_TIMEOUT 2006L
- #define XST_PM_NODE_USED 2007L

Macro Definition Documentation

#define XST_PM_INTERNAL 2000L

An internal error occurred while performing the requested operation

#define XST_PM_CONFLICT 2001L

Conflicting requirements have been asserted when more than one processing cluster is using the same PM slave

#define XST_PM_NO_ACCESS 2002L

The processing cluster does not have access to the requested node or operation

#define XST_PM_INVALID_NODE 2003L

The API function does not apply to the node passed as argument

#define XST_PM_DOUBLE_REQ 2004L

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 2005L

The target processing cluster has aborted suspend

#define XST_PM_TIMEOUT 2006L

A timeout occurred while performing the requested operation

#define XST_PM_NODE_USED 2007L

Slave request cannot be granted since node is non-shareable and used

XilFPGA Library v4.2

Chapter 44

Overview

The XilFPGA library provides an interface to the Linux or bare-metal users for configuring the programmable logic (PL) over PCAP from PS.
The library is designed for Zynq® UltraScale+™ MPSoC to run on top of Xilinx standalone BSPs. It is tested for A53, R5 and MicroBlaze. In the most common use case, we expect users to run this library on PMU MicroBlaze with PMUFW to serve requests from Linux for Bitstream programming.

Supported Features

The following features are supported in Zynq® UltraScale+™ MPSoC platform.

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

XilFPGA library Interface modules

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

Processor Configuration Access Port (PCAP)

The processor configuration access port (PCAP) is used to configure the programmable logic (PL) through the PS.

CSU DMA driver

The CSU DMA driver is used to transfer the actual bitstream file for the PS to PL after PCAP initialization.

XilSecure Library

The XilSecure library provides APIs to access secure hardware on the Zynq UltraScale+ MPSoC devices.

Note

The current version of library supports only Zynq UltraScale MPSoC devices.

Design Summary

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

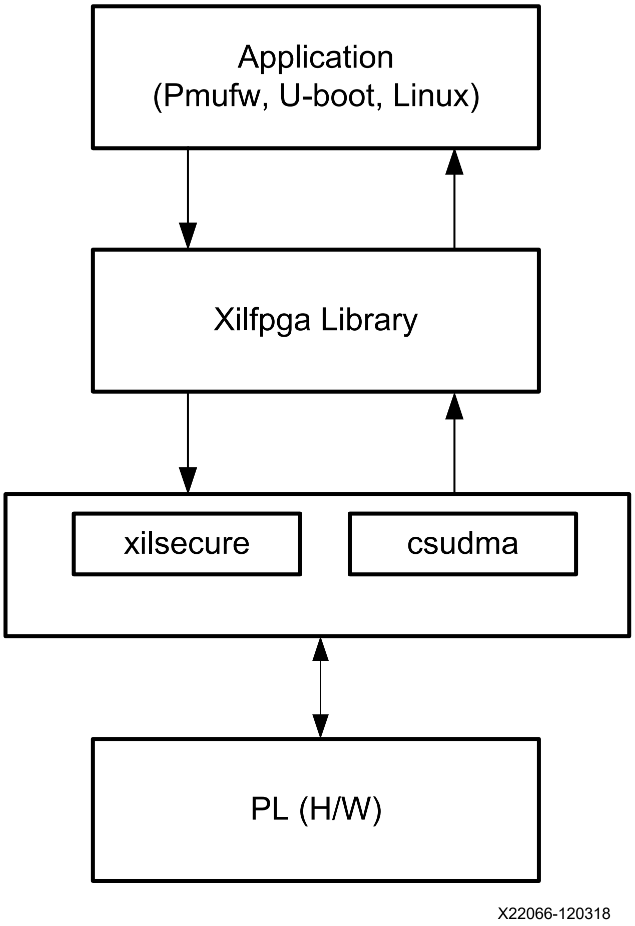

Figure 44.1: XilFPGA Design Summary

Flow Diagram

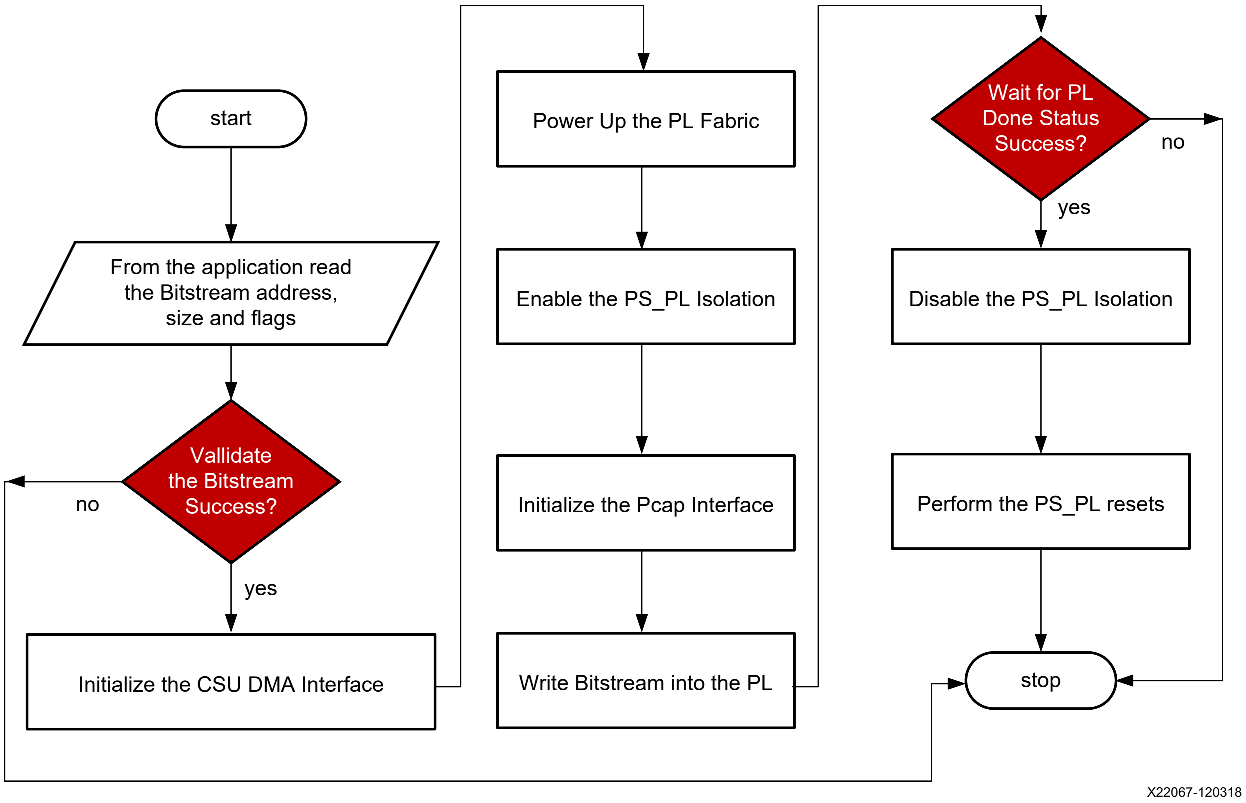
start
From the application read the Bitstream address, size and flags
Vallidate the Bitstream Success?
no
yes
Initialize the CSU DMA Interface
Power Up the PL Fabric
Enable the PS_PL Isolation
Initialize the Pcap Interface
Write Bitstream into the PL
Wait for PL Done Status Success?
no
yes
Disable the PS_PL Isolation
Perform the PS_PL resets
stop
X22067-120318

Figure 44.2: XilFPGA Library Workflow

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. Launch Xilinx SDK. Xilinx SDK prompts you to create a workspace.
2. Select **File** > **New** > **Xilinx Board Support Package**. The **New Board Support Package** wizard appears.
3. Specify a project name.
4. Select **Standalone** from the **Board Support Package OS** drop-down list. The **Board Support Package Settings** wizard appears.
5. Select the **xilfpga** library from the list of **Supported Libraries**.
6. Expand the **Overview** tree and select **xilfpga**. The configuration options for xilfpga are listed.
7. Configure the xilfpga by providing the base address of the Bit-stream file (DDR address) and the size (in bytes).
8. Click **OK**. The board support package automatically builds with XilFPGA library included in it.
9. Double-click the **system.mss** file to open it in the **Editor** view.
10. Scroll-down and locate the **Libraries** chapter.
11. Click **Import Examples** adjacent to the XilFPGA 4.2 entry.

Enabling Secure Mode

To support encrypted and authenticated bitstream loading, you must enable secure mode in PMUFW.

1. Launch Xilinx SDK. Xilinx SDK prompts you to create a workspace.
2. Select **File** > **New** > **Application Project**. The **New Application Project** wizard appears.
3. Specify a project name.
4. Select **Standalone** from the **OS Platform** drop-down list.
5. Select a supported hardware platform.
6. Select **psu_pmu_0** from the **Processor** drop-down list.
7. Click Next. The **Templates** page appears.
8. Select **ZynqMP PMU Firmware** from the **Available Templates** list.
9. Click **Finish**. A PMUFW application project is created with the required BSPs.
10. Double-click the **system.mss** file to open it in the **Editor** view.

11. Click the **Modify this BSP's Settings** button. The **Board Support Package Settings** dialog box appears.
12. Select **xilfpga**. Various settings related to the library appears.
13. Select **secure_mode** and modify its value to **true** .
14. 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

For Bitstream, the size of the file is too large to be contained inside the device and external memory must be used. The use of external memory could pose access security issues. The following section describes how Bitstream is authenticated securely using external memory.

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.

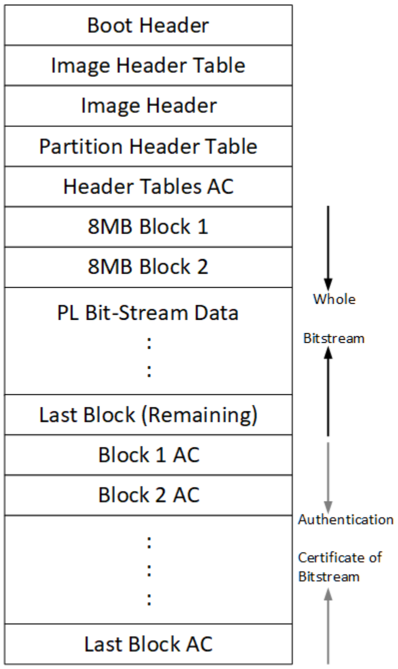

Figure 44.3: Bitstream Blocks

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

Authenticated Bitstream Loading Using OCM

To authenticate the Bitstream partition securely, XilFPGA uses the FSBL chapter's OCM memory to copy the bitstream in chunks from DDR.
The software workflow for authenticating Bitstream is as follows:

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

   Note

   > The chunk that XilFPGA copies can be of any size. A 56KB chunk is taken for better performance.

5. XilFPGA authenticates the Bitstream.
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 cannot use the warm restart when the FSBL OCM memory is used to authenticate the Bitstream.

Authenticated Bitstream Loading Using DDR

XilFPGA uses DDR to authenticate as In-sufficient OCM memory (OCM memory occupies with ATF and FSBL).
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
4. If Authentication is successful, XilFPGA transmits data to PCAP via DMA (for unencrypted Bitstream) or AES (if encryption is enabled).
5. Repeats steps 1 through 4 for all the blocks of Bitstream.

Chapter 45

XilFPGA APIs

Overview

This chapter provides detailed descriptions of the XilFPGA library APIs.

Functions

- u32 XFpga_PL_BitStream_Load (UINTPTR BitstreamImageAddr, UINTPTR AddrPtr, u32 flags)
- u32 XFpga_InterfaceStatus (void)
- u32 XFpga_PL_PostConfig (XFpga_Info ∗PLInfoPtr)
- u32 XFpga_PL_ValidateImage (XFpga_Info ∗PLInfoPtr)
- u32 XFpga_GetPlConfigReg (u32 ConfigReg, UINTPTR Address)
- u32 XFpga_GetPlConfigData (XFpga_Info ∗PLInfoPtr)

Function Documentation

u32 XFpga_PL_BitStream_Load (UINTPTR *BitstreamImageAddr,* UINTPTR *AddrPtr,* u32 *flags*)

The API is used to load the bitstream file into the PL region.
This function performs:

- Power-up the PL fabric.
- Performs PL-PS Isolation.
- Initialize PL configuration Interface
- Write a Bitstream into the PL
- Wait for the PL Done Status.
- Restore PS-PL Isolation (Power-up PL fabric).

Parameters

BitstreamImageAddr	Linear memory Bitstream image base address
AddrPtr	Aes key address which is used for Decryption.
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.

u32 XFpga_InterfaceStatus (void)

This function provides the STATUS of PL programming interface.

Parameters

None	

Returns

Status of the PL programming interface.

u32 XFpga_PL_PostConfig (XFpga_Info ∗ *PLInfoPtr*)

This function set FPGA to operating state after writing.

Parameters

PLInfoPtr	Pointer to XFgpa_Info structure

Returns

Codes as mentioned in xilfpga.h

u32 XFpga_PL_ValidateImage (XFpga_Info ∗ *PLInfoPtr*)

This function is used to validate the Bitstream Image.

Parameters

PLInfoPtr	Pointer to XFgpa_Info structure

Returns

Codes as mentioned in xilfpga.h

u32 XFpga_GetPlConfigReg (u32 *ConfigReg,* UINTPTR *Address*)

This function provides PL specific configuration register values.

Parameters

ConfigReg	Constant which represents the configuration register value to be returned.
Address	DMA linear buffer address.

Returns

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

u32 XFpga_GetPlConfigData (XFpga_Info ∗ *PLInfoPtr*)

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

Parameters

PLInfoPtr	Pointer to XFgpa_Info structure

Returns

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

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

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

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Please Read: Important Legal Notices

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

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