Lattice Sentry Root-of-Trust Reference Design for MachXO3D

User Guide

FPGA-RD-02203-1.0

July 2020

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS and with all faults, and all risk associated with such information is entirely with Buyer. Buyer shall not rely on any data and performance specifications or parameters provided herein. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. No Lattice products should be used in conjunction with mission- or safety-critical or any other application in which the failure of Lattice's product could create a situation where personal injury, death, severe property or environmental damage may occur. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice.

Contents

Figures

Tables

Acronyms in This Document

A list of acronyms used in this document.

Acronym	Definition
AMBA	Advanced Microcontroller Bus Architecture used by the RISC-V to communicate with peripherals.
BMC	Baseboard Management Controller
BSP	Board Support Package, the layer of software containing hardware-specific drivers and libraries to function in a particular hardware environment.
CoT	Chain of Trust
CPU	Central Processing Unit
ECDSA	Elliptic Curve Digital Signature Algorithm
EFB	Embedded Function Block, a hard block in Lattice FPGA device.
ESB	Embedded Security Block, a hardened security block in MachXO3D device.
GPIO	General Purpose Input Output.
GUI	Graphic User Interface
HAL	Hardware Abstraction Layer, a software interface to hide the detail of the hardware design and provide general services to the upper layer.
I^2C	Inter Integrated Circuit
PFR	Platform Firmware Resiliency
QSPI	Quad Serial Peripheral Interface
OOB	Out of Band
PCH	Platform Controller Hub
PLD	Programmable Logic Device
RISC-V	Reduced Instruction Set Computer – Five, a free and open instruction set architecture (ISA) enabling a new era of processor innovation through open standard collaboration.
RoT	Root of Trust
RTL	Register Transfer Level
Rx	Receiver
SDK	System Design and Develop Kit. A set of software development tools that allows the creation of applications for software package on the Lattice embedded platform.
SHA	Secure Hash Algorithm
SoC	System on Chip
SPI	Serial Peripheral Interface
Tx	Transmitter
UART	Universal Asynchronous Receiver-Transmitter
UFM	User Flash Memory

1. Introduction

1.1. Purpose

Lattice MachXO3D device is a new low-density FPGA with enhanced security features and on-chip dual boot flash. The enhanced bitstream security and user-mode security functions enable MachXO3D device to be used as a Root-of-Trust hardware solution in a complex system. With Lattice MachXO3D device, you can implement a Platform Firmware Resiliency (PFR) solution in your system, as described in NIST Special Publication 800-193.

The purpose of this document is to introduce the design methodology of the Lattice MachXO3D PFR solution using the Lattice Propel toolsets, which can largely reduce the design complexity.

1.2. Audience

The intended audience for this document includes embedded system designers and embedded software developers. The technical guidelines assume readers have expertise in embedded system design and FPGA technologies. In addition, readers are recommended to read NIST 800-193 Platform Firmware Resiliency Guidelines before reading this document.

The content in this document is a MachXO3D PFR solution design guide of recommended flows using Lattice Propel tools. It introduces a recommended design guide but not a constraint to experienced users.

1.3. Document Structure

The remainder of this document is with the following major sections:

- Platform Firmware Resiliency System (PFR) Root of Trust (RoT) Introduction section — introduces the Lattice MachXO3D PFR RoT (Root of Trust) solution, including system architecture, functionality overview, and principles supporting firmware resiliency.
- PFR System Architecture and Runtime Flow section — Describes the Lattice MachXO3D PFR RoT firmware architecture, runtime flow, particularly the system configuration, protection, detection and recovery mechanism.
- PFR IP API Reference and PFR Component API Reference sections — List the API reference for the PFR IP and component.
- PFR System Design (from Lattice Propel) section — Shows the design flow through Lattice Propel toolsets, including template design, customization, and simulation.
- PFR System Validation Guide section — A system validation guide by applying Lattice PFR utilities.

2. Platform Firmware Resiliency System (PFR) Root of Trust (RoT) Introduction

2.1. PFR

NIST 800-193 Platform Firmware Resiliency (PFR) Guidelines describe the principles of supporting platform resiliency. As stated in NIST 800-193, the security guidelines are based on the following three principles:

Protection: Mechanisms for ensuring that Platform Firmware code and critical data remain in a state of integrity and are protected from corruption, such as the process for ensuring the authenticity and integrity of firmware updates.

Detection: Mechanisms for detecting when Platform Firmware code and critical data have been corrupted, or otherwise changed from an authorized state.

Recovery: Mechanisms for restoring Platform Firmware code and critical data to a state of integrity in the event that any such firmware code or critical data are detected to have been corrupted, or when forced to recover through an authorized mechanism. Recovery is limited to the ability to recover firmware code and critical data.

2.2. RoT

The security mechanisms are founded in Roots of Trust (RoT). An RoT is an element that forms the basis of providing one or more security-specific functions, such as measurement, storage, reporting, recovery, verification, and update. An RoT must be designed to always behave in the expected manner because its proper functioning is essential to providing its security-specific functions and because its misbehavior cannot be detected. An RoT is typically just the first element in a Chain of Trust (CoT) and can serve as an anchor in such a chain to deliver more complex functionality.

The foundational guidelines on the Roots of Trust (RoT) that support the subsequent guidelines for Protection, Detection, and Recovery. These guidelines are organized based on the logical component responsible for each of those security properties:

The **Root of Trust for Update (RTU)** is responsible for authenticating firmware updates and critical data changes to support platform protection capabilities.

The **Root of Trust for Detection (RTD)** is responsible for firmware and critical data corruption detection capabilities.

The **Root of Trust for Recovery (RTRec)** is responsible for recovery of firmware and critical data when corruption is detected.

2.3. Lattice RoT Mechanism

Lattice MachXO3D FPGA can serve as the Root of Trust and provide the following services:

Image Authentication: On system power-up or reset, MachXO3D device holds the protected devices in reset while it authenticates their boot images in SPI flash. After each signature authentication passes, MachXO3D device releases each resets, and those devices can boot from their authenticated SPI flash image. Image authentication can also be requested at any time through the I^2C Out of Band (OOB) communication path.

Image Recovery: If a flash image becomes corrupted for any reason, it fails to be authenticated. The MachXO3D device can restore it to a known good state by copying from an authenticated backup image.

SPI Flash Monitoring and Protection: The MachXO3D device can monitor multiple SPI/QSPI buses for unauthorized activity and block unauthorized accesses using external SPI quick switches. The monitors can be configured to watch for specific SPI flash commands and address ranges defined by the system designer and designate them as authorized (whitelisted) or unauthorized (blacklisted).

Event Logging: MachX3OD device logs security events, such as unauthorized flash accesses and notifies the BMC.

I^2C Monitoring: The MachXO3D device can monitor an I^2C bus for unauthorized activity and block unauthorized transactions by disabling the I2C bus. The monitor can be configured with multiple whitelist or blacklist filters to watch for specific byte or bit patterns defined by the system designer and designate them as authorized or unauthorized I^2C transactions.

2.4. System Architecture

Figure 2.1 shows the architecture of a Lattice MachXO3D FPGA working as an RoT device. The system design consists of a RISC-V processor connected to a set of peripherals through the AMBA bus. Software running on the processor controls the general and PFR solution peripherals and handles all the events at runtime to perform the system functionalities.

General Peripherals include the MachXO3D hard GPIO, UART, JTAG, and I^{2}C Slave. These modules perform the basic board level controls and communications. PFR solution Peripherals include EFB, ESB, QSPI Streamer/Monitor, I^{2}C Monitor and Customer PLD interface, which perform the main PFR functionalities. Customers can add or remove the peripherals using the Lattice Propel tools according to their requirement. For the details of customization, refer to the Lattice Sentry QSPI Streamer section.

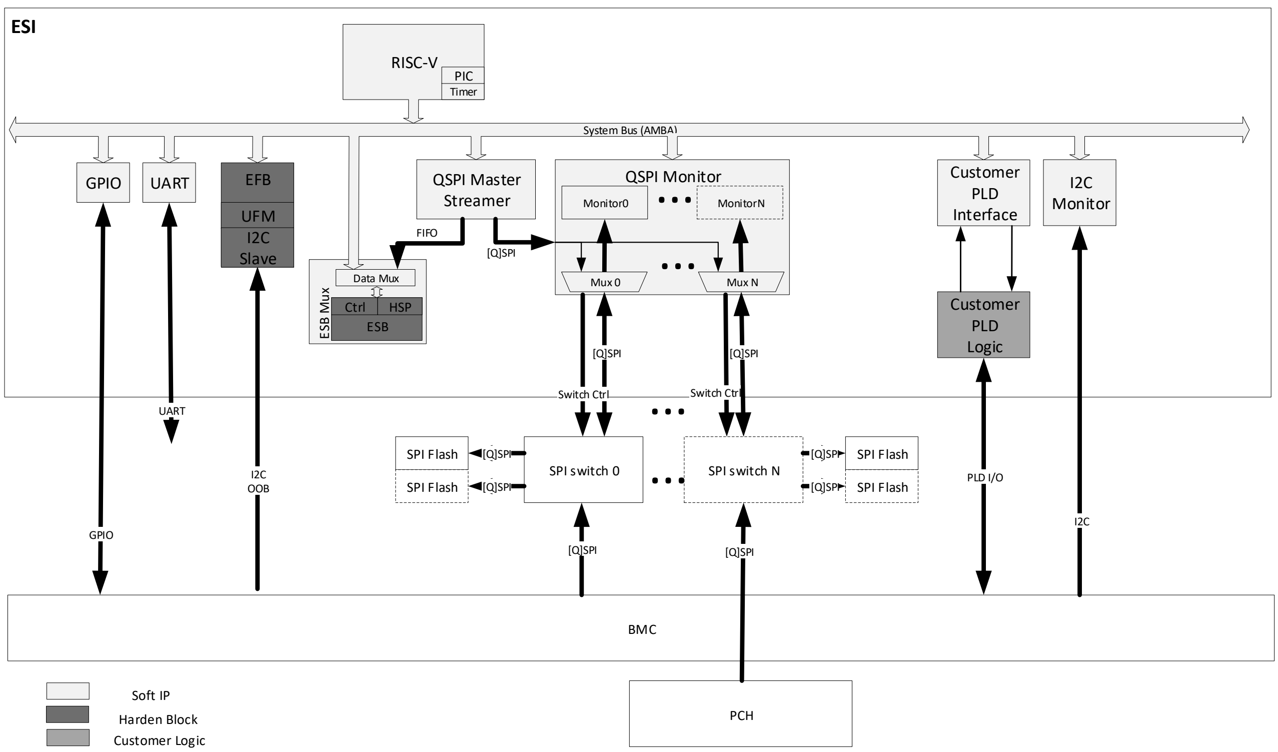

Figure 2.1. Lattice PFR System Architecture

2.5. Functionality Overview

2.5.1. RISC-V Processor

The RISC-V Processor is a configurable CPU soft IP based on the open source Vex RISC-V core, which integrates JTAG debugger, PIC and Timer. The RISC-V core supports RV32I instruction set and 5-stage pipelines to fulfill the performance requirement for PFR system. JTAG debugger, PIC, and Timer can be enabled or disabled based on the system requirement.

2.5.2. Lattice Sentry QSPI Master Streamer

Lattice Sentry QSPI Master Streamer is a configurable SPI master that supports SPI and QSPI slaves. It contains FIFOs for Tx and Rx data, which enable it to support very long SPI transactions (more than 32 bits). It also provides an external Rx FIFO interface (8-bit) that can be connected to the ESB for image authentication.

QSPI Streamer incorporates an SPI FIFO Master that provides significant performance improvement by supporting data read and write transactions of programmable length, allowing an entire SPI flash device to be read in one SPI transaction. The external Rx FIFO interface also enables direct transmission of input data from the SPI slave to another block, such as the ESB without tying up the CPU or system bus.

For details on QSPI streamer, refer to Lattice Sentry QSPI Master Streamer IP Core – Lattice Propel Builder User Guide (FPGA-IPUG-02109). For the system software developer, refer to the PFR IP API Reference section for more details on the API reference.

2.5.3. Lattice Sentry QSPI Monitor

The QSPI Monitor is a configurable security module which can monitor one or more SPI or QSPI buses for unauthorized activity and block transactions by controlling the chip select signal and external quick switch devices. In addition to monitoring, it can connect external SPI/QSPI buses to the QSPI Master Streamer through a programmable mux/demux block.

The QSPI Monitor watches the external buses for allowed flash commands and flash addresses. It provides fine grain control over the set of allowed commands, and supports up to four configurable address spaces which can be independently monitored for erase, program, and read commands. Address spaces can be whitelisted for erase or program commands or blacklisted for read commands. Both 24-bit and 32-bit flash addresses are supported.

For details on the QSPI Monitor, refer to Lattice Sentry QSPI Monitor IP Core for MachXO3D – Lattice Propel Builder User Guide (FPGA-IPUG-02110). For the system software developer, refer to the PFR IP API Reference section for more details on the API reference.

2.5.4. Lattice Sentry I2C Monitor

The I^2C Monitor is a configurable security module which can monitor traffic on an I^2C bus to identify unauthorized activity, based on set of up to 20 programmable filters. If unauthorized activity is detected, the I^2C bus is disabled and firmware is notified so that an event can be logged.

For details of the I^2C monitor, refer to Lattice Sentry I^2C Monitor IP Core for MachXO3D – Lattice Propel Builder User Guide (FPGA-IPUG-02108). For the system software developer, refer to the PFR IP API Reference section for more details on the API reference.

2.5.5. Lattice Sentry ESB Mux

The Embedded Security Block (ESB) is a hardened block which provides a set of security services for MachXO3D device.

The ESB has two interfaces for sending and receiving data: a register interface, and a High Speed Data Port (HSP) which is a FIFO-style interface.

The ESB Mux provides a thin layer around the ESB which provides separate interface ports for AMBA and HSP, and an internal mux to select between them. The mux is controlled by a new control register which is mapped into unused ESB address space. This register is always available through the AMBA interface, regardless of whether the mux is set to AMBA or HSP.

For details on the ESB Mux, refer to Lattice Sentry Embedded Security Block Mux IP Core for MachXO3D – Lattice Propel Builder User Guide (FPGA-IPUG-02107). For the system software developer, refer to the PFR IP API Reference section for more details on the API reference.

2.5.6. Lattice Sentry PLD Interface

The PLD Interface is a register-based interface which is used by firmware to send and receive messages between code executing on the RISC-V and the customer control PLD logic. It can be used to request system control actions, to check status, or to send customized messages. It is the customer's responsibility to connect the PLD logic to the defined interface and implement the actions associated with messages sent by firmware. The design of the actual Customer PLD logic is system-dependent and is implemented by the customer for the particular system.

For details on Customer PLD, refer to Lattice Sentry PLD Interface IP Core for MachXO3D User Guide (FPGA-IPUG-02106). For the system software developer, refer to the PFR IP API Reference section for more details on the API reference.

2.5.7. Embedded Function Block

The Embedded Function Block (EFB) is a hardened block in MachXO3D device, which is used to access User Flash Memory (UFM) and I^2C slave device.

In Lattice PFR solution, the UFM is used to store the Manifest, event logs, and a backup Uboot image, while the I^2C slave device is used to communicate with the BMC or PCH devices.

2.5.8. General Peripherals

Besides the PFR solution peripherals, some general peripherals are also integrated into the system for board-level control or communication, including GPIO, UART, etc. You can add or remove these modules based on your own system requirement.

3. PFR System Architecture and Runtime Flow

3.1. Firmware Architecture

The PFR solution of Lattice MachXO3D FPGA has firmware running on the processor to handle the system dependent information and runtime events.

Figure 3.1 shows the architecture of the firmware of the PFR 2.0 RISC–V solution. The Lattice PFR solution firmware is composed of four layers.

- Sitting on the top is the APP layer, which is the demo application to demonstrate all the features on Protection, Detection and Recovery that FPR spec defined.
- The Component layer is functional module based for dedicated solutions. For PFR solution, this layer contains OOB Communication module, Log/Manifest Management module, and Security Management module to implement the corresponding features.
- BSP/Driver and HAL layers are automatically generated during the system design. All the system-dependent information is applied statically into the source code. The BSP/Driver layer is for all the general IPs, while the HAL layer is for the RISC-V processor IP that capsulates all the platform dependent information.

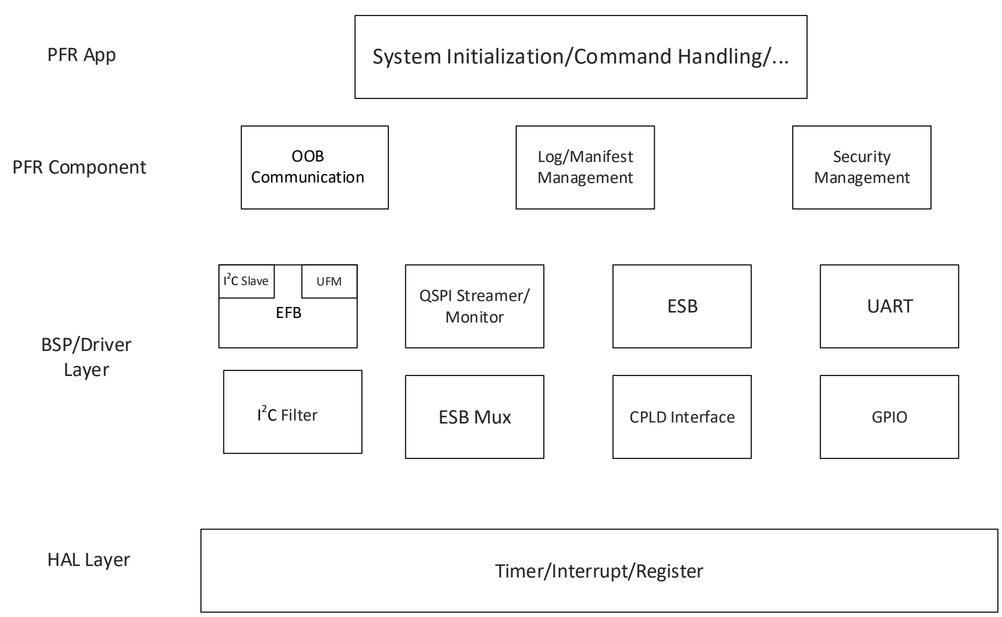

Figure 3.1. Software Architecture of Lattice PFR Solution

3.2. Runtime Flow

The runtime flow of the firmware is shown in Figure 3.2, which can be stated in five steps:

1. Configuration Handler: Read and parse the system Manifest and configure the system accordingly. Refer to the Configuration section for more details.
2. Boot-up Protection Handler: Authenticate the firmware on the SPI flash before BMC/PCH boot up. Refer to the Boot Up Protection section for more details.
3. Recovery Handler: Recover the firmware on the SPI flash if the image is corrupted. Refer to the Recovery section for more detail.
4. Invalid SPI/I^{2}C Event Detection and Protection: Monitor and detect the system SPI/I^{2}C events and avoid invalid behaviors. Refer to the Detection section for more details.

5. Logging and Reporting Handler: Log events which occur and report to the BMC/PCH when requested. Refer to the Logs and Reporting section for more details.

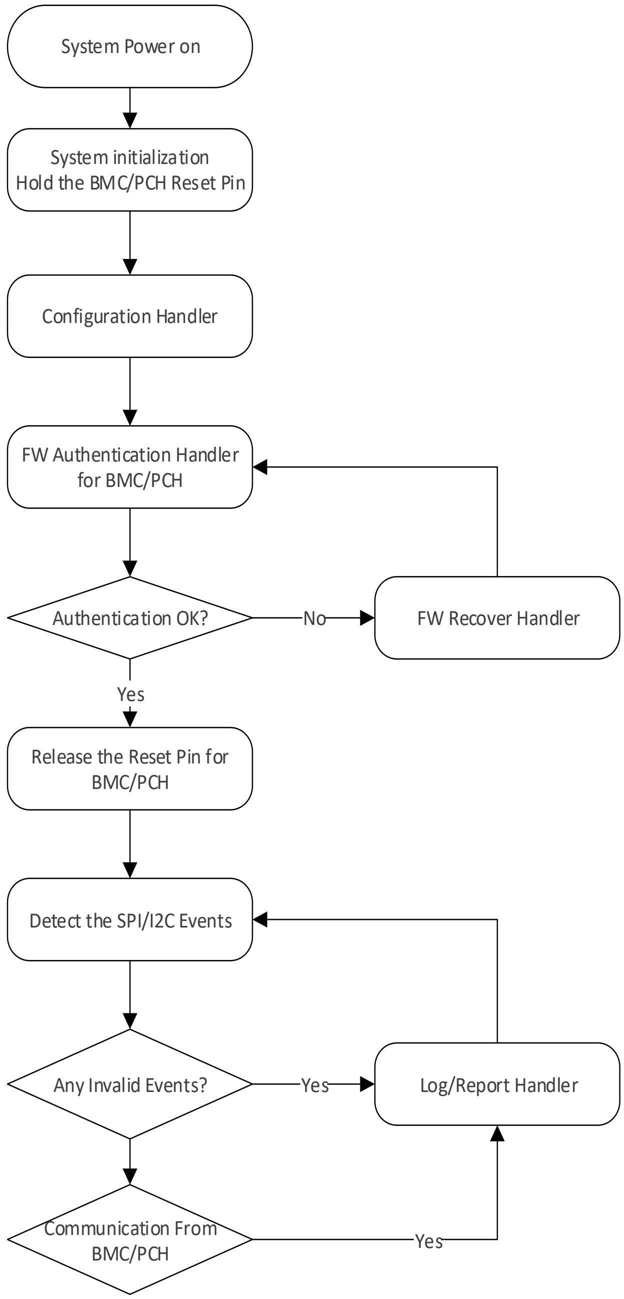

Figure 3.2. Lattice PFR Runtime Flow

3.3. Configuration

System dependent information is configured as a manifest, which is stored in the UFM of Lattice MachXO3D FPGA device. The system manifest is a data structure which provides crucial information (flash layout, signature, keys…) for each firmware stored, authenticated and monitored on a SPI flash in the system.

Use of the manifest in the RoT device can make it easier to maintain a common code functionality for authentication and recovery across different platform designs.

3.3.1. Configuration Flow

During runtime, the system software reads the manifest in the UFM and parse the critical data for firmware authentication, recovery and detection. Figure 3.3 shows basic flow of the Lattice PFR 2.0 configuration handler.

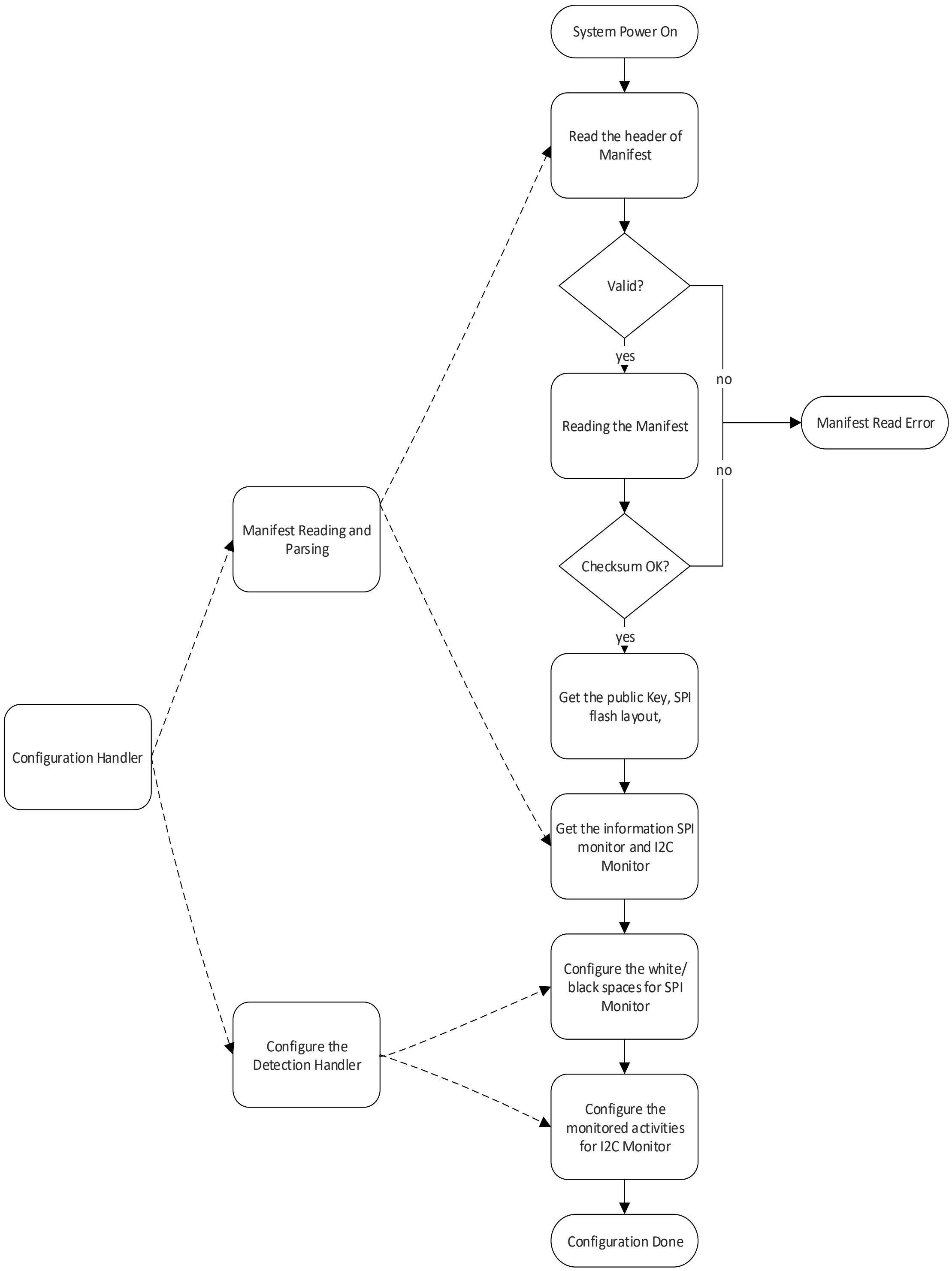

Figure 3.3. Lattice PFR 2. 0 Configuration Handler Flow

3.3.2. MachXO3D PFR Manifest Manager

Lattice Propel provides a Manifest Manager tool to manage the manifest for your own system. The Manifest is then stored in UFM2 of the MachXO3D device.

You can follow steps below to create, modify the manifest for their system.

1. Open Lattice Propel SDK. Click LatticeTools -> Lattice Sentry Manifest Manager to run manifest manager. See Figure 3.4.
2. Click the Open button and choose the .mem file. Manifest Manager loads the .mem file and parses its manifest information, as shown in the three tabs, Image Data, Flash Data and I^{2}C Filter Data (Figure 3.5).
3. Click the Generate button to create the .mem file for UFM2 initialization. The .jed file is programmed into UFM2.

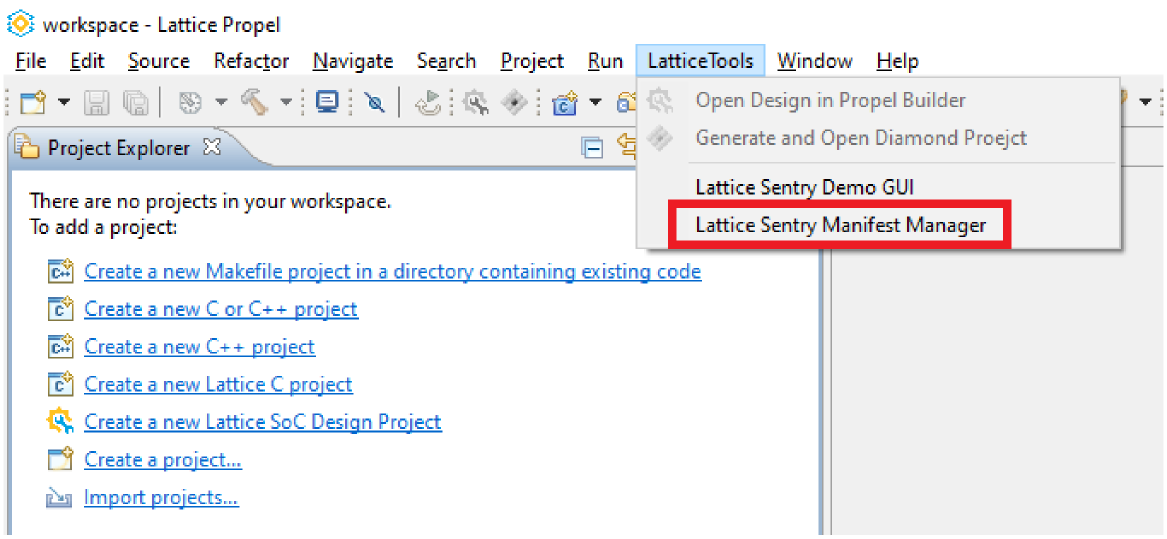

Figure 3.4. Launch Manifest Manager in Lattice Propel SDK

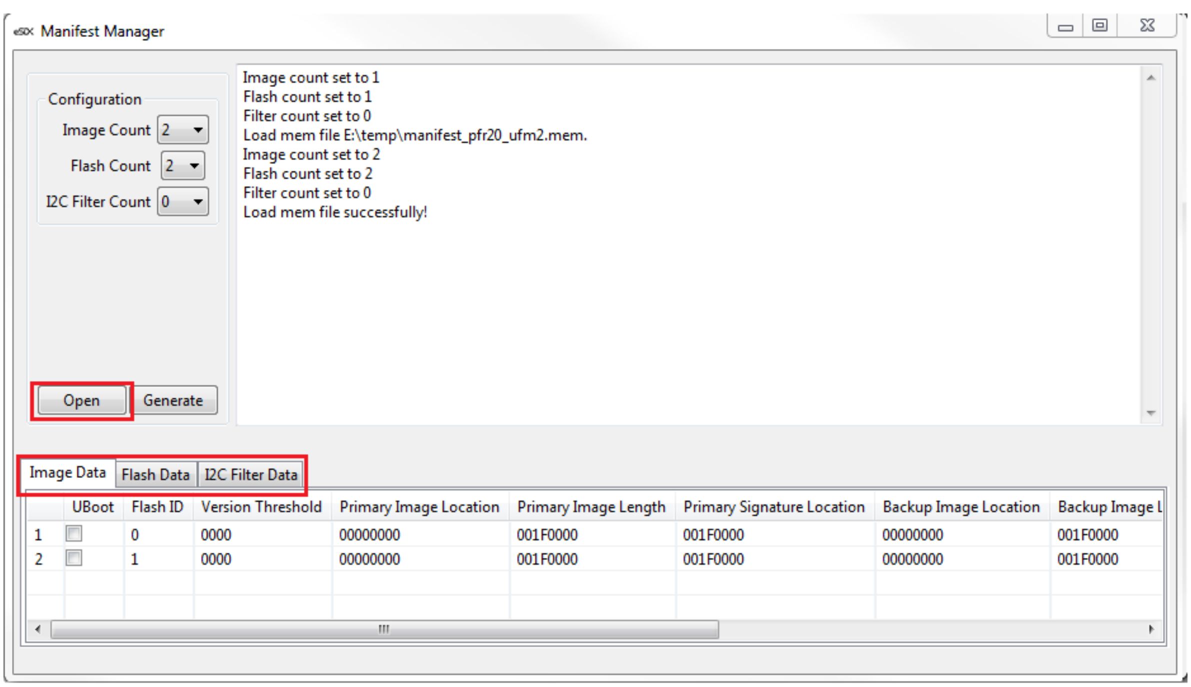

Figure 3.5. Manifest Manager Window

3.4. Boot Up Protection

Before the system boots up, the MachXO3D RoT ensures that the system firmware is valid. If not, the RoT performs recovery.

Figure 3.6 shows the basic flow of authentication for the firmware on the SPI flash. Basically, the authentication consists of two steps. First, perform ECDSA verification using the firmware data and signature stored on the SPI flash with the public key in the Manifest. The second step is to perform a version check to avoid firmware roll back.

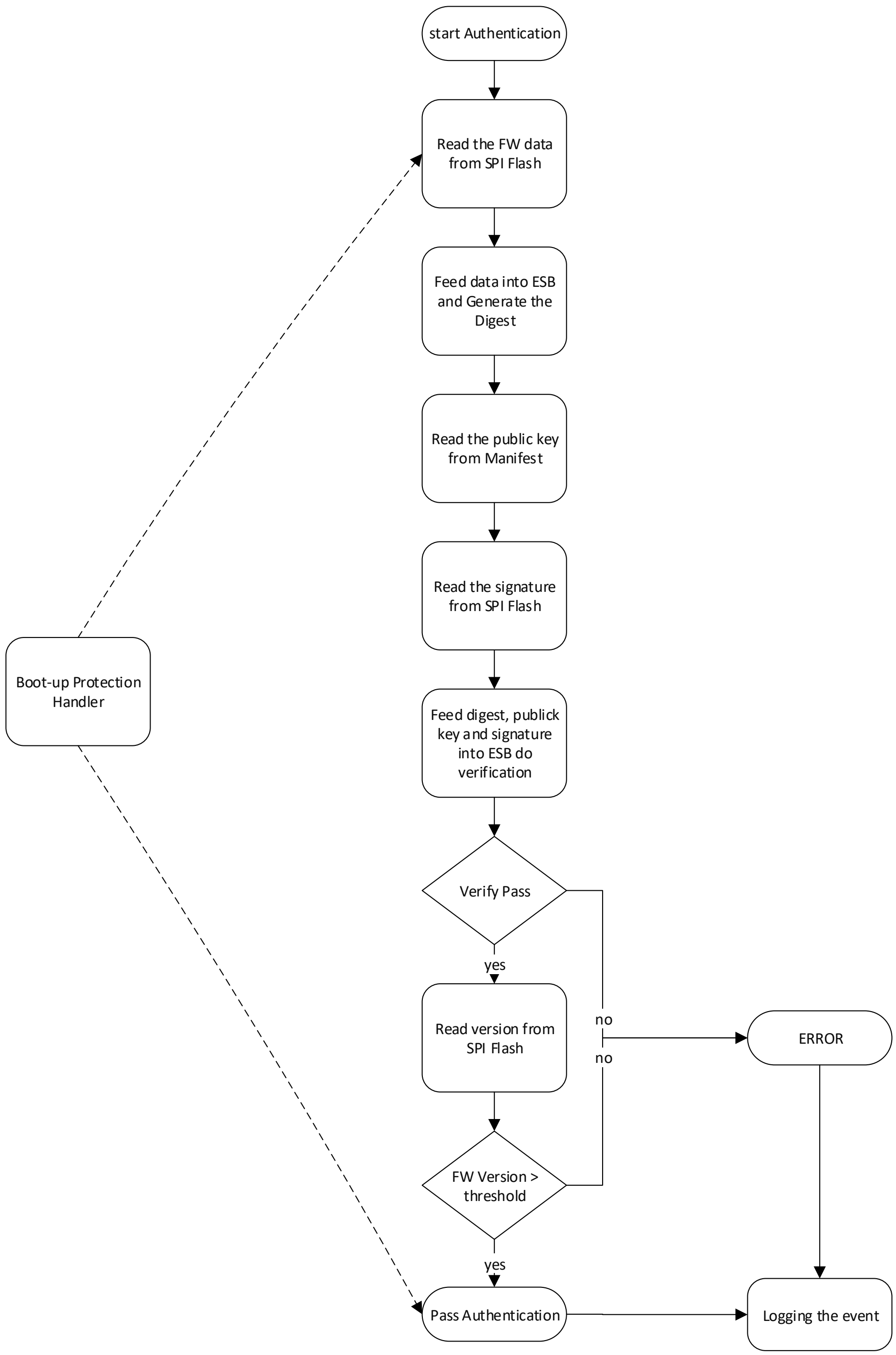

Figure 3.6. PFR Boot-up Protection Handler

3.5. Recovery

Recovery mechanism aims to keep the firmware and critical data in a valid and authorized state in case that they are detected to have been corrupted. Generally, two cases trigger the recovery mechanism, one detects the Firmware has been corrupted, the other is that the BMC/PCH triggers the recovery on purpose. After recovery, authentication is recommended to ensure the integrity of the firmware and data in the recovered flash.

Figure 3.7 shows the basic flow of the recovery process.

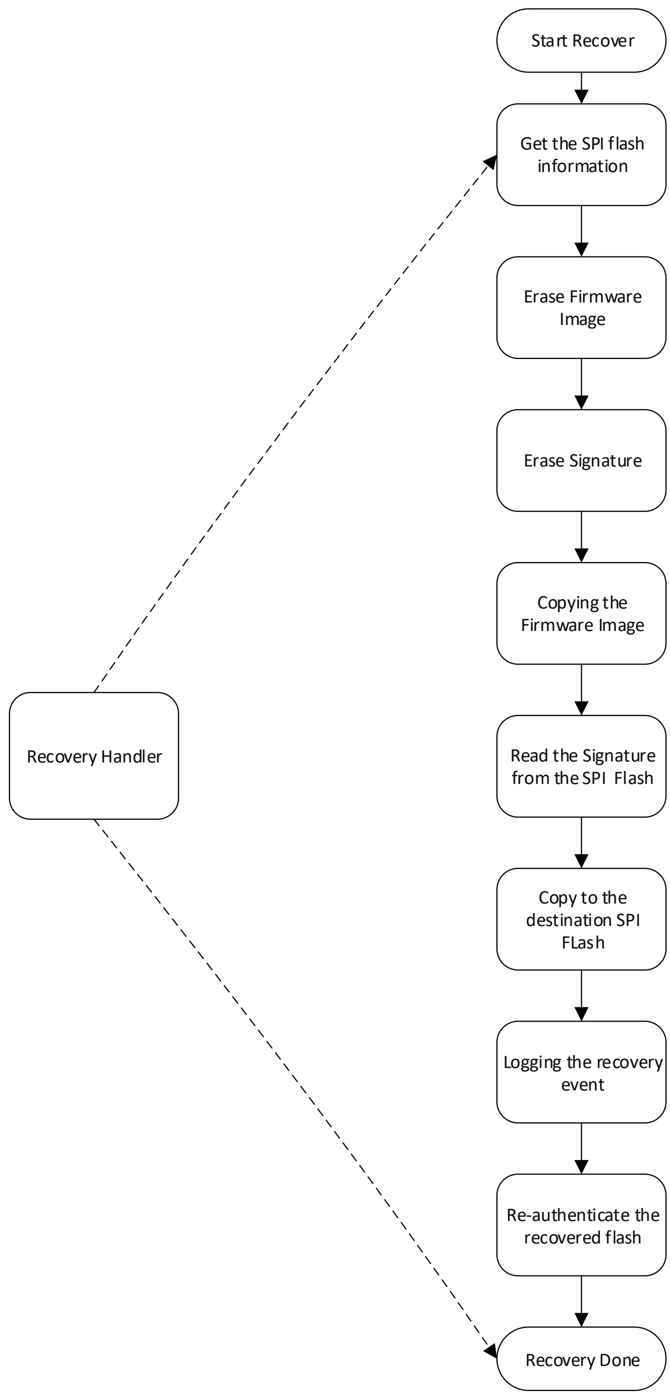

Figure 3.7. PFR Recovery Handler

3.6. Detection

The detection mechanism can detect unauthorized changes to device firmware and critical data before it is executed or consumed by the device. In Lattice MachXO3D PFR solution (Figure 3.8), two kinds of events can be monitored, SPI flash access and I²C read/write.

Firmware and critical data can be stored on the SPI flashes of the system. At different locations of the flash, the access authority level may be different. Three authority levels are defined in Lattice PFR solution, which are called white, grey and black list. For each monitored spaces of the flash, an authority level is defined and configured in manifest accordingly.

Table 3.1. Authority Level Definition

Authority Level	Definition
White	Read, Erase, and Write are all allowed.
Grey	Only Read is allowed. Neither Erase nor Write operation is permitted.
Black	None of the Read, Erase or Write operation is permitted. The operation is disturbed when any of the Read, Erase, or Write operation is detected on the SPI bus.

The I²C bus may be used for communications among on-board devices. Some critical data can be exchanged. The Lattice MachXO3D PFR solution can be configured to define a set of transactions which are monitored on the I²C bus at runtime. If there are any illegal transactions detected, an interrupt or a flag is issued to notify the processor. This information is logged and reported to the BMC/PCH.

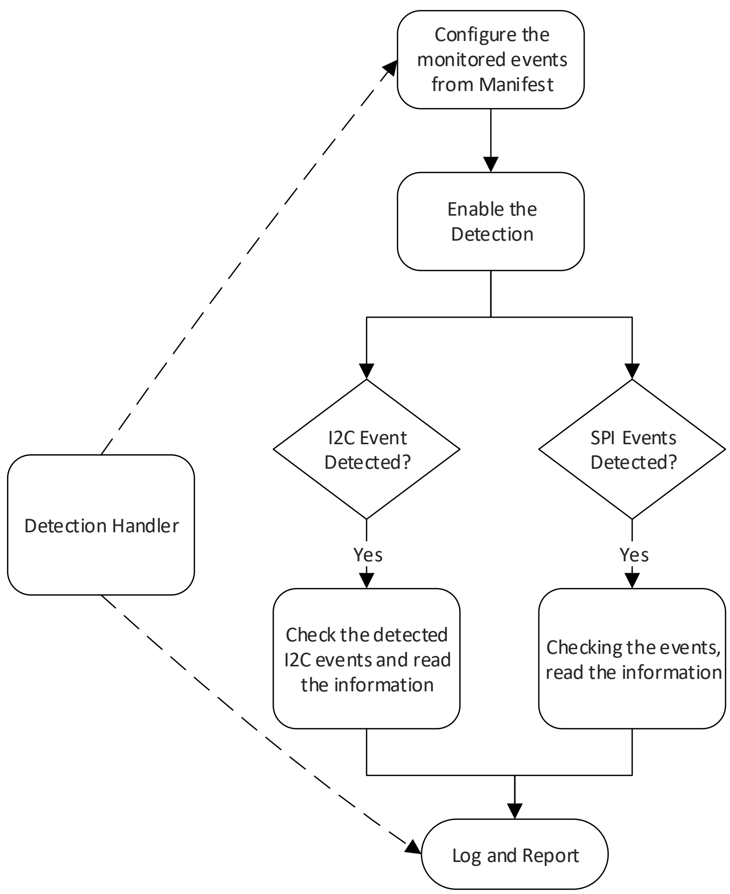

Figure 3.8. PFR Detection Handler

3.7. Logs and Reporting

Logs of the different events are written to the UFM3 of the Lattice MachXO3D device, starting from page 1. Each page of UFM3 holds a single log entry. Byte 0 is the log index and indicates the page where the log is stored, as well as an indicator of available memory. Byte 15 is used to indicate if a log has been read (RD).

The BMC/PCH can read the logs from RoT device and know the events in the system via the I^{2}C OOB channel. Table 3.2 shows the detailed definition of the log format.

Table 3.2. Lattice PFR Log Format Definition

Log Entry Type	Data Byte 0	Data Byte 1	Data Byte 2	Data Byte 3	Date Byte 4	Data Byte 5	Data Byte 6	Data Byte 7	Data Byte 8	Data Byte 9	Data Byte 10	Data Byte 11	Data Byte 12	Data Byte 13	Data Byte 14	Data Byte 15
Authentication	Log Index	0x0	Img ID	Pri/ Sec	Pass/ Fail	0x00	0x00	0x00	Timestamp in Seconds (32-bit)				—	—	—	RD
SPI Exception	Log Index	0x01	Flash ID	SPI CMD	SPI Address				Timestamp in Seconds (32-bit)				—			RD
I^{2}C Exception	Log Index	0x02	I2C ID	Filter ID	0x00	0x00	0x00	0x00	Timestamp in Seconds (32-bit)				—	—	—	RD
Recovery	Log Index	0x04	Img ID	0-Pri=> BU 1-BU=> Pri	0x00	0x00	0x00	0x00	Timestamp in Seconds (32-bit)				—	—	—	RD
Recovery UBoot	Log Index	0x05	Img ID	1-Pri 2-BU	0x00	0x00	0x00	0x00	Timestamp in Seconds (32-bit)				—	—	—	RD

4. PFR IP API Reference

The PFR IPs are critical parts of the Lattice PFR solution. You need to make use of the APIs to initialize, configure, and control the IPs to perform the functions.

The following sections provide reference to the APIs for each PFR IP, which is released in the corresponding IP package by Lattice.

4.1. Lattice Sentry QSPI Monitor

qspi_mon_init	
unsigned char qspi_mon_init(struct spi_mon_instance *this_spi_monitor, unsigned int base_address)	
Parameter	**Description**
this_spi_monitor	The pointer to current QSPI monitor instance.
base_address	Base address of the QSPI monitor module, Propel SDK automatically parses the address map of the SoC system and passes the information to software.
Returns	**Description**
unsigned char	0: Succeeded in initializing the QSPI monitor module. 1: Failed to initialize the QSPI monitor module.
Description	
This function is used to Initializes QSPI monitor instance. This function is supposed to be called when the platform is initializing. This function should be called before calling any QSPI monitor related functions.	

qspi_mon_flash_update	
unsigned char qspi_mon_flash_update(struct spi_mon_instance *this_spi_monitor, unsigned int flash_id, unsigned int flash_select, unsigned int master_select)	
Parameter	**Description**
this_spi_monitor	The pointer to current QSPI monitor instance.
flash_id	The value of the flash id number.
flash_select	The value of flash to select: 0x10: Select Flash A. 0x20: Select Flash B.
master_select	The value of master to select: 0: SPI Monitor 1: Internal Master
Returns	**Description**
unsigned char	0: Succeeded in selecting the new flash. 1: Failed to select the new flash.
Description	
This function is used to select flash that QSPI master accesses to.	

qspi_mon_ws_update	
unsigned char qspi_mon_ws_update(struct spi_mon_instance *this_spi_monitor, unsigned int flash_id, unsigned int mon_cntl, unsigned int dummy_num, struct spi_monitor_space *flash_mon_sp)	
Parameter	**Description**
this_spi_monitor	The pointer to current QSPI monitor instance.
flash_id	The value of the flash ID number.
mon_cntl	The monitor control value to set to the QSPI monitor.
dummy_num	The value of dummy byte number that sets to the QSPI monitor.
flash_mon_sp	The pointer to the flash monitor space that set to the QSPI monitor.
Returns	**Description**
unsigned char	0: Succeeded in updating the QSPI monitor space. 1: Failed to update the QSPI monitor space.
Description	
This function is used to update white space and control setting for the QSPI monitor.	

qspi_mon_exception_get	
unsigned char qspi_mon_exception_get(struct spi_mon_instance *this_spi_monitor, unsigned int flash_id, unsigned int *command, unsigned int *address)	
Parameter	**Description**
this_spi_monitor	The pointer to current QSPI monitor instance.
flash_id	The value of the flash ID number.
command	The pointer to the buffer to store the exception SPI command.
address	The pointer to the buffer to store the exception SPI address.
Returns	Description
unsigned char	0: Succeeded in getting the exception. 1: Failed to get the exception.
Description	
This function is used to get the command and SPI access address of the exception from the QSPI monitor.	

4.2. Lattice Sentry QSPI Streamer

spi_streamer_init	
unsigned char spi_streamer_init(struct spi_streamer_instance *this_spi, unsigned int base_addr, unsigned int spi_mode, unsigned int sck_div)	
Parameter	**Description**
this_spi	The pointer to the instance of current QSPI streamer device.
base_addr	Base address of the QSPI streamer module. Propel SDK parses the address map of the SoC system and passes the information to software.
spi_mode	The value of QSPI mode to select. 0x00: QSPI mode 0 0x03: QSPI mode 3
sck_div	The value of the clock division.
Returns	**Description**
unsigned char	0: Succeeded in initializing the QSPI streamer. 1: Failed to initialize the QSPI streamer.
Description	
This function is used to Initialize QSPI streamer module. This function is supposed to be called when the platform is initializing. This function should be called before calling any QSPI streamer related functions.	

spi_write	
unsigned char spi_write(struct spi_streamer_instance *this_spi, unsigned int addr, unsigned int length, unsigned char *buff, unsigned char addr4B)	
Parameter	**Description**
this_spi	The pointer to the instance of current QSPI streamer device.
addr	The start address of the SPI device to write to.
length	The number of data in byte that is written to the SPI device.
buff	The pointer to the data buffer that is written to the SPI device.
addr4B	The value of the addressing mode to select. 0: 3-byte address mode 1: 4-byte address mode
Returns	**Description**
unsigned char	0: Succeeded in writing the specified data to the SPI device. 1: Failed to write the specified data to the SPI device.
Description	
This function is used to write the specified length of data in the buffer to the SPI device from the specified address. Refer to spi_read() for the data reading details.	

spi_read	
unsigned char spi_read(struct spi_streamer_instance *this_spi, unsigned int addr, unsigned int length, unsigned char *buff, unsigned char addr4B)	
Parameter	**Description**
this_spi	The pointer to the instance of current QSPI streamer device.
addr	The start address of SPI device to read from.
length	The length of data in byte that is read from the SPI device.
buff	The pointer to the data buff that stores the data read from the SPI device.
addr4B	The value of mode to select. 0: 3-byte address mode 1: 4-byte address mode
Returns	**Description**
unsigned char	0: Succeeded in reading the specified data from the SPI device. 1: Failed to read the specified data from the SPI device.
Description	
This function is used to read the specified length of data from the SPI device. Refer to spi_write() for the data writing details.	

spi_write_txfifo	
unsigned char spi_write_txfifo(struct spi_streamer_instance *this_spi, unsigned int addr, unsigned int length)	
Parameter	**Description**
this_spi	The pointer to the instance of current QSPI streamer device.
addr	The start address of the SPI device to write to.
length	The number of data in byte that is written to the SPI device.
Returns	**Description**
unsigned char	0: Succeeded in writing the specified data to the SPI device. 1: Failed to write the specified data to the SPI device.
Description	
This function is used to write the specified length of data in the TX FIFO to the SPI device from the specified address.	

spi_read_txfifo	
unsigned char spi_read_txfifo(struct spi_streamer_instance *this_spi, unsigned int addr, unsigned int length)	
Parameter	**Description**
this_spi	The pointer to the instance of current QSPI streamer device.
addr	The start address of SPI device to read from.
length	The length of data in byte that is read from the SPI device.
Returns	**Description**
unsigned char	0: Succeeded in reading the specified data from the SPI device. 1: Failed to read the specified data from the SPI device.
Description	
This function is used to read the specified length of data from the SPI device and store the data into the TX FIFO of the QSPI streamer module.	

spi_read_esb	
unsigned char spi_read_esb(void *this_spi_streamer, unsigned int addr, unsigned int length, unsigned char addr4B)	
Parameter	**Description**
this_spi	The pointer to the instance of current QSPI streamer device.
addr	The start address of SPI device to read from.
length	The length of data in byte that is read from the SPI device.
addr4B	The value of mode to select. 0: 3-byte address mode 1: 4-byte address mode
Returns	**Description**
unsigned char	0: Succeeded in reading the specified data from the SPI device. 1: Failed to read the specified data from the SPI device.
Description	
This function is used to read the specified length of data from the SPI device and feed to the ESB module for processing. For details on general data read, refer to spi_read().	

spi_erase_4k	
unsigned char spi_erase_4k(struct spi_streamer_instance *this_spi, unsigned int addr, unsigned char addr4B)	
Parameter	**Description**
this_spi	The pointer to the instance of current QSPI streamer device.
addr	The start address of the SPI device to erase
addr4B	The value of mode to select. 0: 3-byte address mode 1: 4-byte address mode
Returns	**Description**
unsigned char	0: Succeeded in erasing the 4K data. 1: Failed to erase the 4K data.
Description	
This function is used to erase a 4K memory of the SPI device from the specified address.	

4.3. Lattice Sentry I²C Monitor

i2c_mon_init	
unsigned char i2c_mon_init(struct i2c_mon_instance *this_i2cmon, unsigned int base_addr)	
Parameter	**Description**
`this_i2cmon`	The pointer to the instance of the current I²C monitor.
`base_addr`	Base address of the I²C monitor module. Propel SDK automatically parses the address map of the SoC system and pass the information to software.
Returns	**Description**
`unsigned char`	0: Succeeded in initializing the I²C monitor. 1: Failed to initialize the I²C monitor.
Description	
This function is used to initialize the I²C monitor module. This function is supposed to be called when the platform is initializing. This function should be called before calling any I²C monitor related functions.	

i2c_mon_conf	
unsigned char i2c_mon_conf(struct i2c_mon_instance *this_i2cmon, struct i2c_mon_entry *entry_data, unsigned int entry_num)	
Parameter	**Description**
this_i2cmon	The pointer to the instance of the current I^2C monitor.
entry_data	The pointer to the entry data that is configured to the I^2C monitor device.
entry_num	The number of the monitor entry. Maximum number is 20.
Returns	**Description**
unsigned char	0: Succeeded in configuring the I^2C monitor. 1: Failed to configure the I^2C monitor.
Description	
This function is used to configure the I^2C monitor device by setting the number of entry and the entry data.	

i2c_mon_enable	
unsigned char i2c_mon_enable(struct i2c_mon_instance *this_i2cmon, unsigned int mon_en)	
Parameter	**Description**
this_i2cmon	The pointer to the instance of the current I^2C monitor.
mon_en	The value of enable the i2c monitor. 0: Disable the monitor. 1: Enable the monitor.
Returns	**Description**
unsigned char	0: Succeeded in enabling/disabling the I^2C monitor. 1: Failed to enable/disable the I^2C monitor.
Description	
This function is used to enable or disable the I^2Cmonitor to start or stop the monitoring of the I^2C bus.	

i2c_mon_bus_stop	
unsigned char i2c_mon_bus_stop(struct i2c_mon_instance *this_i2cmon, unsigned char bus_stop)	
Parameter	**Description**
this_i2cmon	The pointer to the instance of the current I^2C monitor.
bus_stop	The value of the bus stop flag of the I^2C monitor. 0: Release the I^2C bus 1: Stop the I^2C bus
Returns	**Description**
unsigned char	0: Stopped or released the monitored I^2C bus. 1: Failed to stop or release the monitored I^2C bus.
Description	
This function is used to stop or release the monitored I^2C bus.	

i2c_mon_event_get	
unsigned char i2c_mon_event_get(struct i2c_mon_instance *this_i2cmon, unsigned char *event_cnt, unsigned int *dct_evt)	
Parameter	**Description**
this_i2cmon	The pointer to the instance of the current I^2C monitor.
event_cnt	The pointer to the buffer to store the detected events count of the I^2C monitor.
dct_evt	The pointer to the buffer to store the detected event number of the I^2C monitor.
Returns	**Description**
unsigned char	0: Succeeded in getting the detected I^2C events. 1: Failed to get the detected I^2C events.
Description	
This function is used to get the number of event and the count of detected events from the I^2C monitor.	

i2c_mon_isr	
void i2c_mon_isr(void *ctx)	
Parameter	**Description**
ctx	The pointer to the context of the I^2C monitor device.
Returns	**Description**
void	—
Description	
This function is used to process I^2C monitor interrupt. The function can be registered via calling pic_isr_register ().	

4.4. Lattice Sentry ESB Mux

esb_init	
unsigned char esb_init(struct esb_instance *this_esb, unsigned int base_addr);	
Parameter	**Description**
this_esb	The pointer to the instance of the current ESB device.
base_addr	Base address of the ESB module, Propel SDK automatically parses the address map of the SoC system and passes the information to the software.
Returns	**Description**
unsigned char	0: Succeeded in initializing the ESB module. 1: Failed to initialize the ESB module.
Description	
This function is supposed to be called when the platform is initialized. This function should be called before calling any ESB related functions.	

esb_mux_por_sel	
unsigned char esb_mux_port_sel(struct esb_instance *this_esb, unsigned int sel_port)	
Parameter	**Description**
this_esb	The pointer to the instance of the current ESB device.
sel_port	Select the ESB mux to high speed port (HSP) or WISHBONE bus port.
Returns	**Description**
unsigned char	0: Succeeded in selecting the specified port for ESB module. 1: Failed to select the specified port for ESB module.
Description	
This function is used to select the ESB mux to the specified data port. There are two data ports for the ESB module: one is the HSP high-speed port, the other is the WISHBONE bus port.	

esb_switch_idle	
unsigned char esb_switch_idle(struct esb_instance *this_esb)	
Parameter	**Description**
this_esb	The pointer to the instance of the current ESB device.
Returns	**Description**
unsigned char	0: Succeeded in switching the ESB module to idle state. 1: Failed to switch the ESB module to idle state.
Description	
This function is used to switch the ESB module into idle state.	

esb_trng32bits_get	
unsigned char esb_trng32bits_get(struct esb_instance *this_esb, unsigned int *trn_value)	
Parameter	**Description**
this_esb	The pointer to the instance of the current ESB device.
trn_value	The pointer to the data buffer to store the 32-bit long random number generated by the ESB module.
Returns	**Description**
unsigned char	0: Succeeded in getting the random number. 1: Failed to get the random number.
Description	
This function is used to generate a 32-bit long random number by the ESB module.	

esb_nonce_get	
unsigned char esb_nonce_get(struct esb_instance *this_esb, unsigned char p_trn[16])	
Parameter	**Description**
this_esb	The pointer to the instance of the current ESB device.
p_trn	The data buffer to store the 15-byte random number generated by the ESB block and one byte checksum.
Returns	**Description**
unsigned char	0: Succeeded in getting the random number. 1: Failed to get the random number.
Description	
This function is used to get the random number generated by the ESB module.	

esb_trng256bits_get	
unsigned char esb_trng256bits_get(struct esb_instance *this_esb, unsigned char p_trn[32])	
Parameter	**Description**
this_esb	The pointer to the instance of the current ESB device.
p_trn	The data array to store the 256-bit random number generated by the ESB module.
Returns	**Description**
unsigned char	0: Succeeded in getting the random number. 1: Failed to get the random number.
Description	
This function is used to generate a 256-bit long random number.	

esb_pubkey_derive	
unsigned char esb_pubkey_derive(struct esb_instance *this_esb, EccPoint * p_publicKey, unsigned char p_privateKey[NUM_ECC_DIGITS])	
Parameter	**Description**
this_esb	The pointer to the instance of the current ESB device.
p_publicKey	The pointer to data buffer to store the generated public key.
p_privateKey	The private key input to the ESB module.
Returns	**Description**
unsigned char	0: Succeeded in deriving the public key. 1: Failed to derive the public key.
Description	
This function is used to derive the public key.	

esb_ecdh_get	
unsigned char esb_ecdh_get(struct esb_instance *this_esb, unsigned char p_secret[NUM_ECC_DIGITS], EccPoint * p_publicKey, unsigned char p_privateKey[NUM_ECC_DIGITS])	
Parameter	**Description**
this_esb	The pointer to the instance of the current ESB device.
p_secret	The data array to store the shared secret generated by ECDH.
p_publicKey	The public key to for ECDH.
p_privateKey	The private key for ECDH.
Returns	**Description**
unsigned char	0: Succeeded in getting the ECDH shared secret. 1: Failed to get the ECDH shared secret.
Description	
This function is used to generate the shared secret with ECDH.	

esb_aes	
unsigned char esb_aes(struct esb_instance *this_esb, unsigned char *key, unsigned char *bufferIn, unsigned char *bufferOut, unsigned int decrypt)	
Parameter	**Description**
this_esb	The pointer to the instance of the current ESB device.
key	The 128-bit long public key to do the AES encryption or decryption.
bufferIn	16-byte long data to do the AES encryption or decryption.
bufferOut	The 16-byte long result of the AES encryption or decryption for the input data.
decrypt	The flag to indicate to do encryption or decryption. 0: To do encryption 1: To do decryption
Returns	**Description**
unsigned char	0: Succeeded in doing the AES for the input data. 1: Failed to do the AES for the input data
Description	
This function is used to do the AES encryption or decryption for the input data with the specified public key.	

esb_sha256	
unsigned char esb_sha256(struct esb_instance *this_esb, struct sha256_ctx *ctx)	
Parameter	**Description**
this_esb	The pointer to the instance of the current ESB device.
ctx	The pointer to the context to do the SHA256.
Returns	**Description**
unsigned char	0: Succeeded in generating the digest via SHA-256 hash function. 1: Failed to generate the digest via SHA-256 hash function.
Description	
This function is used to generate a 256-bit long digest for the data specified in the context via the SHA-256 hash function.	

esb_esdsa_verify	
unsigned char esb_esdsa_verify(struct esb_instance *this_esb, unsigned int digest[], unsigned int pub_key[], unsigned int signature[], unsigned char *auth_pass)	
Parameter	**Description**
this_esb	The pointer to the instance of the current ESB device.
digest	The digest that feeds to the ESB module to do the ECDSA authentication.
pub_key	The public key that feeds to the ESB module to do the ECDSA authentication.
signature	The signature that feeds to the ESB module to do the ECDSA authentication.
auth_pass	The pointer to the data buffer to hold the authentication result: 1: Authentication passed. 0: Authentication failed.
Returns	**Description**
unsigned char	0: Succeeded in doing the ECDSA verification. 1: Failed to do the ECDSA verification.
Description	
This function is used to do the ECDSA authentication.	

get_nonce	
unsigned char get_nonce(struct esb_instance *this_esb, unsigned char p_trn[16])	
Parameter	**Description**
this_esb	The pointer to the instance of the current ESB device.
p_trn	The data buffer to store the 15-byte random number generated by the ESB block and one byte checksum.
Returns	**Description**
unsigned char	0: Succeeded in getting the random number. 1: Failed to get the random number.
Description	
This function is used to get the random number generated by the ESB module.	

4.5. Lattice Sentry PLD Interface

cstm_pld_init	
unsigned char cstm_pld_init(struct cstm_pld_instance *this_cstm_pld, unsigned int base_addr)	
Parameter	**Description**
this_cstm_pld	The pointer to the current customer PLD instance.
base_addr	The base address of the customer PLD module. Propel SDK automatically parses the address map of the SoC system and passes the information to software.
Returns	**Description**
unsigned char	0: Succeeded in initializing the customer PLD module. 1: Failed to initialize the customer PLD module.
Description	
This function is used to initialize the customer PLD module.	

cstm_pld_int_set	
unsigned char cstm_pld_int_set(struct cstm_pld_instance *this_cstm_pld, unsigned int ints)	
Parameter	**Description**
this_cstm_pld	The pointer to the current customer PLD instance.
ints	The interrupts bit set to notify the PLD logic.
Returns	**Description**
unsigned char	0: Succeeded in setting the interrupt bits. 1: Failed to set the interrupt bits.
Description	
This function is used to set the specified interrupts bit to notify the customer PLD logic.	

cstm_pld_int_status_get	
unsigned char cstm_pld_int_status_get(struct cstm_pld_instance *this_cstm_pld, unsigned int *ints)	
Parameter	**Description**
this_cstm_pld	The pointer to the current customer PLD instance.
ints	The pointer to data buffer to hold the interrupt status.
Returns	**Description**
unsigned char	0: Succeeded in getting the interrupt status. 1: Failed to get the interrupt status.
Description	
This function is used to get the interrupt status of customer PLD module.	

cstm_pld_msg_receive	
unsigned char cstm_pld_msg_receive(struct cstm_pld_instance *this_cstm_pld, unsigned char *msg)	
Parameter	**Description**
this_cstm_pld	The pointer to the current customer PLD instance.
msg	The pointer to buffer to hold the message that is received from the customer PLD logic.
Returns	**Description**
unsigned char	0: Succeeded in receiving the message. 1: Failed to receive the message.
Description	
This function is used to receive the message from the customer PLD logic.	

cstm_pld_msg_send	
unsigned char cstm_pld_msg_send(struct cstm_pld_instance *this_cstm_pld, unsigned char *msg)	
Parameter	**Description**
this_cstm_pld	The pointer to the current customer PLD instance.
msg	The pointer to the message that is to be sent to the customer PLD logic.
Returns	**Description**
unsigned char	0: Succeeded in sending the message to the customer PLD logic. 1: Failed to send the message to the customer PLD logic.
Description	
This function is used to send the message to the customer PLD logic.	

cstm_pld_isr	
void cstm_pld_isr(void *ctx)	
Parameter	**Description**
ctx	The pointer to context that is passed to the interrupt service routine.
Returns	**Description**
void	—
Description	
This function is called when there is interrupts from the customer PLD module. The function can be registered via calling pic_isr_register ().	

5. PFR Component API Reference

The component layer of the Lattice PFR solution provides basic function for protection, detection, and recovery.

The following section provides the API reference on how to manage the manifest, OOB channel, high-level security and log. Based on the provided component layer APIs, you can develop your own PFR software easily.

5.1. Manifest Management

load_manifest_flash	
unsigned char load_manifest_flash(struct st_manifest_t *manifest)	
Parameter	**Description**
manifest	The pointer to the manifest of the system.
Returns	**Description**
unsigned char	Returns 0 if no error.
Description	
This function is used to load the manifest into internal flash.	

mfst_oob_read	
unsigned char mfst_oob_read(struct st_manifest_t *manifest, volatile struct st_i2cCtx_t *this_i2c_efb, struct esb_instance *this_esb)	
Parameter	**Description**
manifest	The pointer to the manifest of the system.
this_i2c_efb	The pointer to the instance of the current I^2C device used for the OOB channel.
this_esb	The pointer to the instance of the current ESB device.
Returns	**Description**
unsigned char	Returns 0 if no error.
Description	
This function is used to read manifest from UFM and send data to BMC over the OOB channel.	

mfst_ufm_read	
unsigned char mfst_ufm_read(struct st_manifest_t *manifest, struct spi_mon_instance *SPImonitor)	
Parameter	**Description**
manifest	The pointer to the manifest of the system.
SPImonitor	The pointer to the instance of the current SPI monitor device.
Returns	**Description**
unsigned char	Returns 0 if no error.
Description	
This function is used to read manifest from UFM.	

mfst_ufm_write	
unsigned char mfst_ufm_write(struct st_manifest_t *manifest, volatile struct st_i2cCtx_t *this_i2c_efb)	
Parameter	**Description**
manifest	The pointer to the manifest of the system.
this_i2c_efb	The pointer to the instance of the current I^{2}C device used for the OOB channel.
Returns	**Description**
unsigned char	Returns 0 if no error.
Description	
This function is used to update manifest in UFM.	

mfst_image_update	
unsigned char mfst_image_update(struct st_manifest_t *manifest, volatile struct st_i2cCtx_t *this_i2c_efb);	
Parameter	**Description**
manifest	The pointer to the manifest of the system.
this_i2c_efb	The pointer to the instance of the current I^{2}C device used for the OOB channel.
Returns	**Description**
unsigned char	Returns 0 if no error.
Description	
This function is used to update the image information in manifest.	

mfst_sign_update	
unsigned char mfst_sign_update(struct st_manifest_t *manifest, volatile struct st_i2cCtx_t *this_i2c_efb)	
Parameter	**Description**
manifest	The pointer to the manifest of the system.
this_i2c_efb	The pointer to the instance of the current I^{2}C device used for the OOB channel.
Returns	**Description**
unsigned char	Returns 0 if no error.
Description	
This function is used to update the signature information in manifest.	

mfst_ver_update	
unsigned char mfst_ver_update(struct st_manifest_t *manifest, volatile struct st_i2cCtx_t *this_i2c_efb)	
Parameter	**Description**
manifest	The pointer to the manifest of the system.
this_i2c_efb	The pointer to the instance of the current I^{2}C device used for the OOB channel.
Returns	**Description**
unsigned char	Returns 0 if no error.
Description	
This function is used to update the version information in manifest.	

mfst_ver_thrhd_update	
unsigned char mfst_ver_thrhd_update(struct st_manifest_t *manifest, volatile struct st_i2cCtx_t *this_i2c_efb)	
Parameter	**Description**
manifest	The pointer to the manifest of the system.
this_i2c_efb	The pointer to the instance of the current I^2C device used for the OOB channel.
Returns	**Description**
unsigned char	Returns 0 if no error.
Description	
This function is used to update version threshold in manifest.	

mfst_pkey_update	
unsigned char mfst_pkey_update(struct st_manifest_t *manifest, volatile struct st_i2cCtx_t *this_i2c_efb)	
Parameter	**Description**
manifest	The pointer to the manifest of the system.
this_i2c_efb	The pointer to the instance of the current I^2C device used for the OOB channel.
Returns	**Description**
unsigned char	Returns 0 if no error.
Description	
This function is used to update the public key in manifest.	

mfst_wsa_update	
unsigned char mfst_wsa_update(struct st_manifest_t *manifest, volatile struct st_i2cCtx_t *this_i2c_efb, struct spi_mon_instance *SPImonitor)	
Parameter	**Description**
manifest	The pointer to the manifest of the system.
this_i2c_efb	The pointer to the instance of the current I^2C device used for the OOB channel.
SPImonitor	The pointer to the instance of the current SPI monitor device.
Returns	**Description**
unsigned char	Returns 0 if no error.
Description	
This function is used to update the white space address in manifest.	

5.2. OOB Management

oob_cmd_get	
unsigned char oob_cmd_get(volatile struct st_i2cCtx_t *this_i2c_efb, unsigned char num)	
Parameter	**Description**
this_i2c_efb	The pointer to the instance of the current I^2C device used for the OOB channel.
num	The offset of the OOB command in the data buffer.
Returns	**Description**
unsigned char	Returns the value of the OOB command.
Description	
This function is used to get the OOB command sent from the BMC.	

oob_param_num_get	
unsigned char oob_param_num_get(volatile struct st_i2cCtx_t *this_i2c_efb)	
Parameter	**Description**
this_i2c_efb	The pointer to the instance of the current I^2C device used for the OOB channel.
Returns	**Description**
unsigned char	Returns the parameter length of the current OOB command.
Description	
This function is used to get the length of the current OOB command.	

oob_status_get	
unsigned char oob_status_get (volatile struct st_i2cCtx_t *this_i2c_efb)	
Parameter	**Description**
this_i2c_efb	The pointer to the instance of the current I^2C device used for the OOB channel.
Returns	**Description**
unsigned char	Returns the current status of the OOB channel.
Description	
This function is used to get the status of the OOB channel.	

oob_status_set	
void oob_status_set(volatile struct st_i2cCtx_t *this_i2c_efb, unsigned char status)	
Parameter	**Description**
this_i2c_efb	The pointer to the instance of the current I^2C device.
status	The status of OOB user wants to set.
Returns	**Description**
void	—
Description	
This function is used to set the status of OOB channel.	

oob_status_buffer_set	
void oob_status_buffer_set(volatile struct st_i2cCtx_t *this_i2c_efb, unsigned char status)	
Parameter	**Description**
this_i2c_efb	The pointer to the instance of the current I^2C device used for the OOB channel.
status	The status of OOB you want to get.
Returns	**Description**
void	—
Description	
This function is used to set the status buffer of the OOB channel.	

oob_data_read	
unsigned char oob_data_read(volatile struct st_i2cCtx_t *this_i2c_efb, struct esb_instance *this_esb)	
Parameter	**Description**
this_i2c_efb	The pointer to the instance of the current I^2C device used for the OOB channel.
this_esb	The pointer to the instance of the current ESB device.
Returns	**Description**
unsigned char	0 : Succeeded in reading the data from the OOB channel. 1: Failed to read the data from the OOB channel.
Description	
This function is used to read data from OOB channel.	

oob_data_write	
void oob_data_write(volatile struct st_i2cCtx_t *this_i2c_efb, unsigned char *buff, unsigned char size, unsigned char checksum)	
Parameter	**Description**
this_i2c_efb	The pointer to the instance of the current I^2C device used for OOB channel.
buff	The pointer to the data buffer you want to write to the OOB channel.
size	The length of data in byte you want to write.
checksum	The check sum of the data written to the OOB channel.
Returns	**Description**
void	—
Description	
This function is used to send data via OOB channel.	

5.3. Security Manager

Select_flash	
int select_flash(struct spi_mon_instance *SPImonitor, unsigned int flash_id, unsigned int flash_select, unsigned int master_select);	
Parameter	**Description**
SPImonitor	The pointer to the QSPI monitor device.
flash_id	The value of the flash ID you want to select.
flash_select	The primary of secondary flash you want to select.
master_select	The SPI master you want to select. 0: QSPI Monitor. 1: Internal QSPI master.
Returns	**Description**
int	1: Succeeded in selecting the SPI flash. -1: Failed to select the SPI flash.
Description	
This function is used to select the SPI flash you want to access.	

authenticate_image	
int authenticate_image(struct st_manifest_t *manifest, struct spi_mon_instance *SPImonitor, struct spi_streamer_instance *qspi_master_streamer_inst, struct esb_instance *esb_inst, unsigned int image_id, unsigned int flash_sel);	
Parameter	**Description**
manifest	The pointer to the current manifest.
SPImonitor	The pointer to the QSPI monitor device.
qspi_master_streamer_inst	The pointer to the QSPI streamer device.
esb_inst	The pointer to the ESB device.
image_id	The image ID that used to get the image related information from the manifest.
flash_sel	The primary or the secondary SPI flash where you wants to do the authentication.
Returns	**Description**
int	1: Succeeded in authenticating the specified image. –1: Failed to authenticate the specified image.
Description	
This function is used to authenticate the specified image stored on the SPI flash.	

recover_image	
int recover_image(struct st_manifest_t *manifest, struct spi_mon_instance *SPImonitor, struct spi_streamer_instance *qspi_master_streamer_inst, unsigned int image_id, unsigned int buflash2priflash);	
Parameter	**Description**
manifest	The pointer to the current manifest.
SPImonitor	The pointer to the QSPI monitor device.
qspi_master_streamer_inst	The pointer to the QSPI streamer device.
image_id	The image ID that used to get the image related information from the manifest.
buflash2priflash	The flash to indicate the direction of the recovery. 0 means recovery from primary to secondary.
Returns	**Description**
int	1: Succeeded in recovering the specified image. –1: Failed to recover the specified image.
Description	
This function is used to recover the image from the specified source to the specified destination.	

recover_uboot	
int recover_uboot(struct st_manifest_t *manifest, struct spi_mon_instance *SPImonitor, struct spi_streamer_instance *qspi_master_streamer_inst, unsigned int image_id, unsigned int pri_sec);	
Parameter	**Description**
manifest	The pointer to the current manifest.
SPImonitor	The pointer to the QSPI monitor device.
qspi_master_streamer_inst	The pointer to the QSPI streamer device
image_id	The image ID that used to get the image related information from the manifest.
pri_sec	The value to specify recovery destination. 1: To recover primary SPI flash device. 2: To recover secondary SPI flash device.
Returns	**Description**
int	1: Succeeded in recovering the SPI flash with the Uboot image. –1: Failed to recovery the SPI flash with the Uboot image.
Description	
This function is used to recover the flash image with the Uboot image.	

5.4. Log Management

log_write	
int log_write(struct st_manifest_t *manifest, unsigned char *data)	
Parameter	**Description**
manifest	The pointer to the current manifest of the system.
data	The pointer to the data buffer that stores the log.
Returns	**Description**
int	0: Succeeded in writing the log. –1: Failed to write the log.
Description	
This function is used to write one slot of log data into the UFM.	

log_read	
unsigned int log_read(struct st_manifest_t *manifest, volatile struct st_i2cCtx_t *this_i2c_efb, unsigned char *pException, struct esb_instance *this_esb);	
Parameter	**Description**
manifest	The pointer to the manifest of the current system.
this_i2c_efb	The pointer to the I^2C slave device that is used as the communication channel.
pException	The pointer to the flag for exception.
this_esb	The pointer to the ESB device.
Returns	**Description**
unsigned int	Return the available address for the next log.
Description	
This function is used to read the log from the UFM and send it to BMC via the OOB channel.	

log_ack	
int log_ack(struct st_manifest_t *manifest, unsigned int page);	
Parameter	**Description**
manifest	The pointer to the current manifest of the system.
page	The value of log entry.
Returns	**Description**
int	0: Succeeded in writing the log. –1: Failed to write the log.
Description	
This function is used to acknowledge that the previous log has been received.	

log_clear	
int log_clear(struct st_manifest_t *manifest);	
Parameter	**Description**
manifest	The pointer to the current manifest of the system.
Returns	**Description**
int	0: Succeeded to clear the log. No other return value.
Description	
This function is used to write one slots of log data into the UFM.	

6. PFR System Design (from Lattice Propel)

Lattice Propel is a platform for embedded system design, development, and validation. Lattice Propel provides a PFR Solution Template to simplify customer PFR solution design.

6.1. PFR Solution Template

The PFR Solution Template provides a baseline PFR implementation with all required features enabled. You can follow Lattice Propel tool flow to create or modify a standard PFR design.

The diagram below (Figure 6.1) shows the general design flow based on Propel tool sets. Choose PFR Template during the Select Solutions Templates step. After that, follow the Propel user guide to create the entire design step by step.

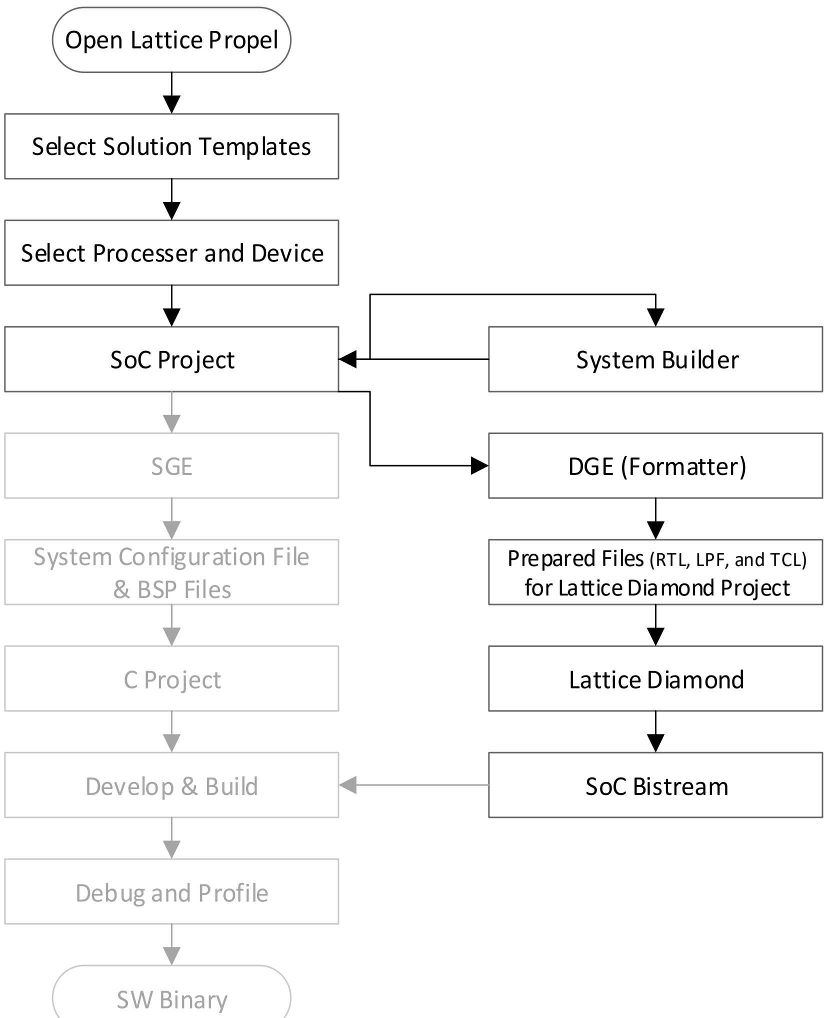

Figure 6.1. Lattice Propel Template Flow

6.2. PFR System Design Customization

You can customize your hardware and software designs on top of the PFR Solution Template to meet your specific requirements.

When creating a new PFR system design, to build a customized design, you can:

- after creating the SoC project, customize the SoC design in System Builder.
- after creating a project in Lattice Diamond:
    - add/edit RTL source files to bring in customer logic;
    - edit the LPF file for I/O mapping and constrain settings.
- After the software project is created, edit the source files in Propel SDK.

Further changes can be made to the existing PFR system design which is created through the Propel tool sets. Note when an SoC design is changed in the System Builder, it is necessary to build the hardware project in Propel SDK to regenerate the BSP. After that, a new software project needs to be created with the updated BSP.

6.2.1. Customer PLD Customization

As stated in the Embedded Function Block section, a Customer PLD module is provided to allow you to integrate the control logic into the PFR solution. In the Lattice PFR Solution Template, a simple customer PLD design is provided (Figure 6.2) to demonstrate a typical usage as monitoring and controlling customized I/O pads.

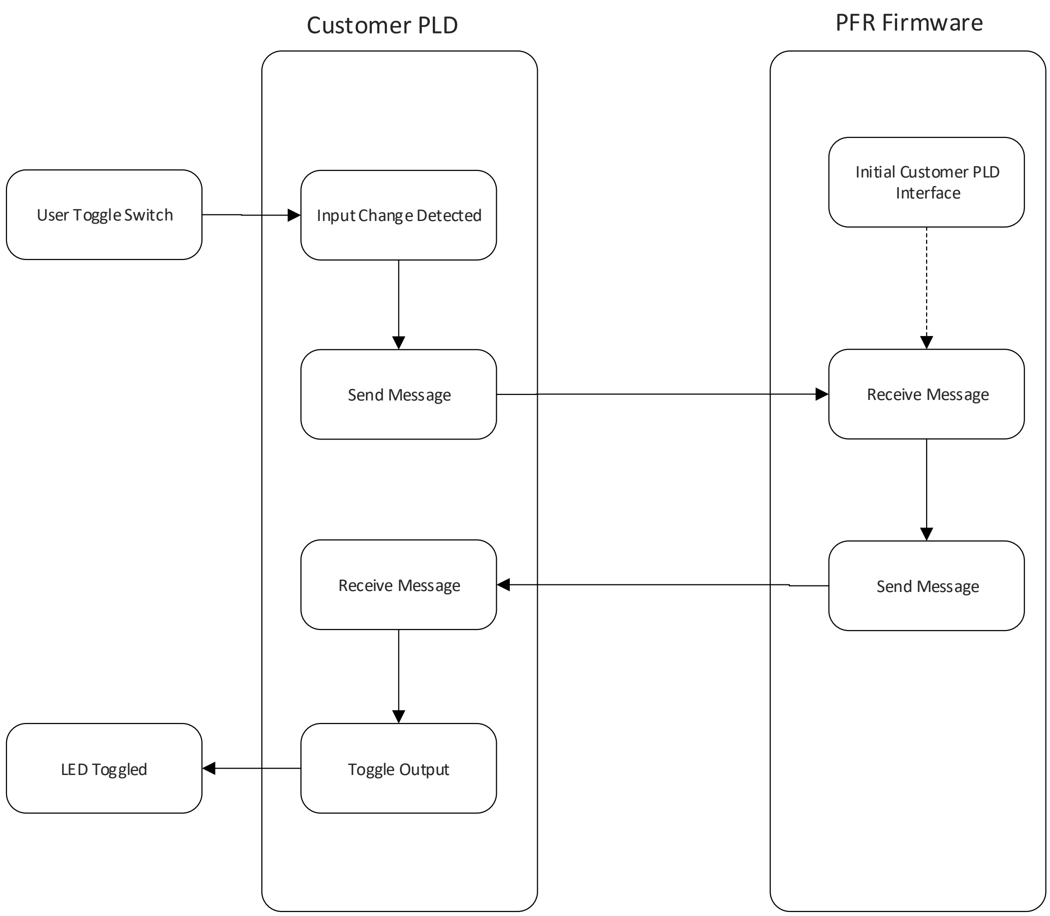

Figure 6.2. Customer PLD Work Flow

You can edit the template project to customize the functionality of customer PLD as well as the firmware accordingly.

6.3. System-level Simulation

After a new PFR design is created via the Lattice Propel platform, system-level simulation tool is provided to verify the functionalities of the system. This section only covers PFR-specific topics. For details on how to launch the simulation, refer to the System Simulation Flow of Lattice Propel 1.0 User Guide (FPGA-UG-02110) for how to launch the simulation.

There are several pre-developed test cases available for a quick evaluation. Once PFR template is generated, a folder named *sim* is created as well. Read readme.txt inside this *sim* folder for detailed information.

6.3.1. Simulation Details

As shown Figure 6.3, in Baseboard Management Controller (BMC) and PCH, as well as several SPI flashes, are instantiated in the simulation platform. UART slaves are also there for debugging purpose. uart_bmc is the special one used to simulate "PFR Demo Tool" in Propel SDK.

The uart_bmc module reads the I^{2}C command in stimulus_bmc.txt. After that, this uart_bmc module sends to/communicate with BMC via UART. BMC decodes and communicates with DUT via I^{2}C. Once DUT gets valid commands, it acts accordingly.

PCH has similar workflow, but it mainly communicates with SPI flash in the simulation platform.

Figure 6.3 shows the overview of the simulation platform.

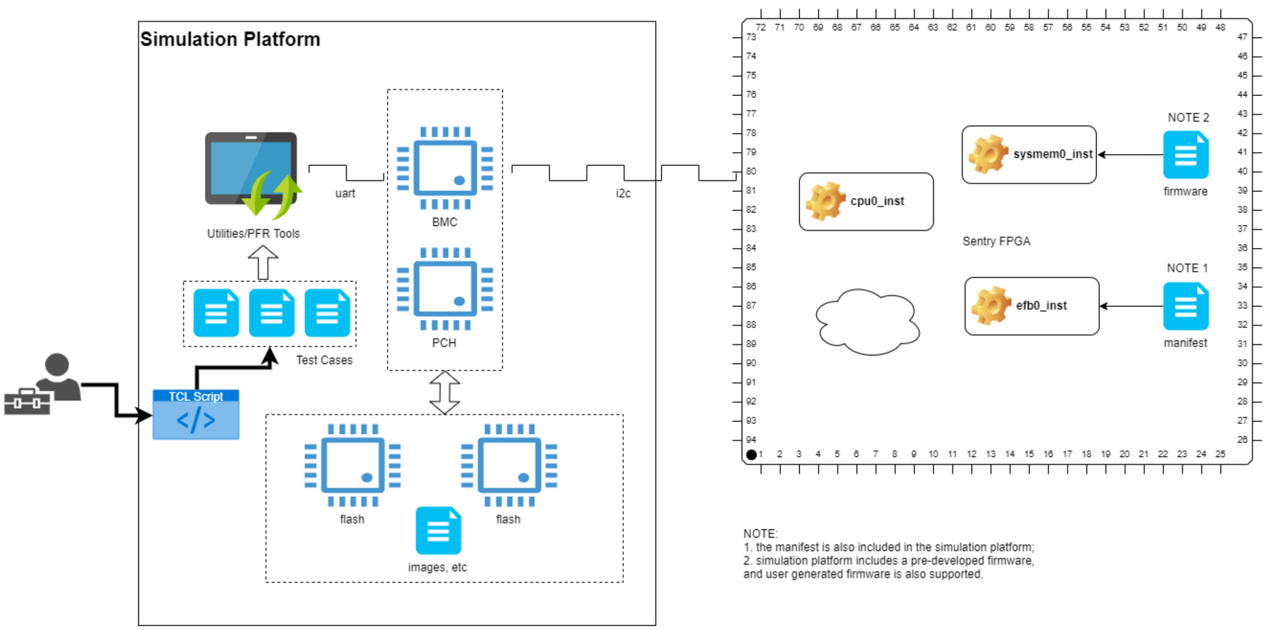

Figure 6.3. PFR System Simulation Platform Overview

7. PFR System Validation Guide

7.1. PFR Utilities

A set of utilities in Lattice Propel can let you validate the functionalities for the PFR system. With these utilities, you can perform system-level validation for your own PFR solutions.

7.1.1. PFR Demo Tool GUI

The PFR Demo Tool GUI is a tool which can communicate between a PC with Windows platform and the MachXO3D device through UART to I²C bridge on the Lattice MachXO3D PFR Demo Board. It also provides SPI access to verify the monitoring and protection of the SPI Flash. The PFR Demo GUI is integrated in Lattice Propel platform.

To use PFR Demo Tool:

1. Connect mini-USB cable from PC to the mini-USB connector J6 of the MachXO3D PFR Demo Board.
2. From your PC desktop, invoke Lattice Propel. Choose LatticeTools -> Lattice PFR Demo Tool to invoke Lattice PFR Demo Tool. See Figure 7.1.

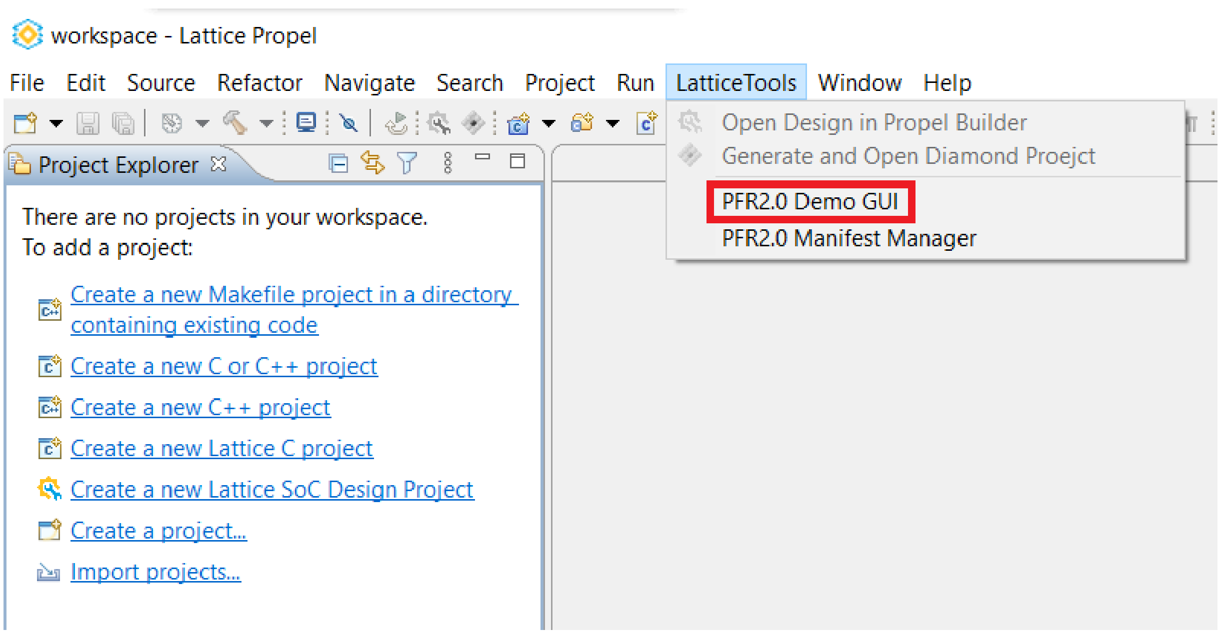

Figure 7.1. Launch Lattice PFR Demo Tool

3. The available COM ports are listed in Console Output. Clicking the Scan Ports button can update the available ports. See Figure 7.2.
4. Two COM ports are associated with the MachXO3D PFR Demo Board. The COM port with smaller number is for BMC, while the COM port with larger number is for PCH. Select the associated COM port for both BMC and PCH channel. See Figure 7.2.

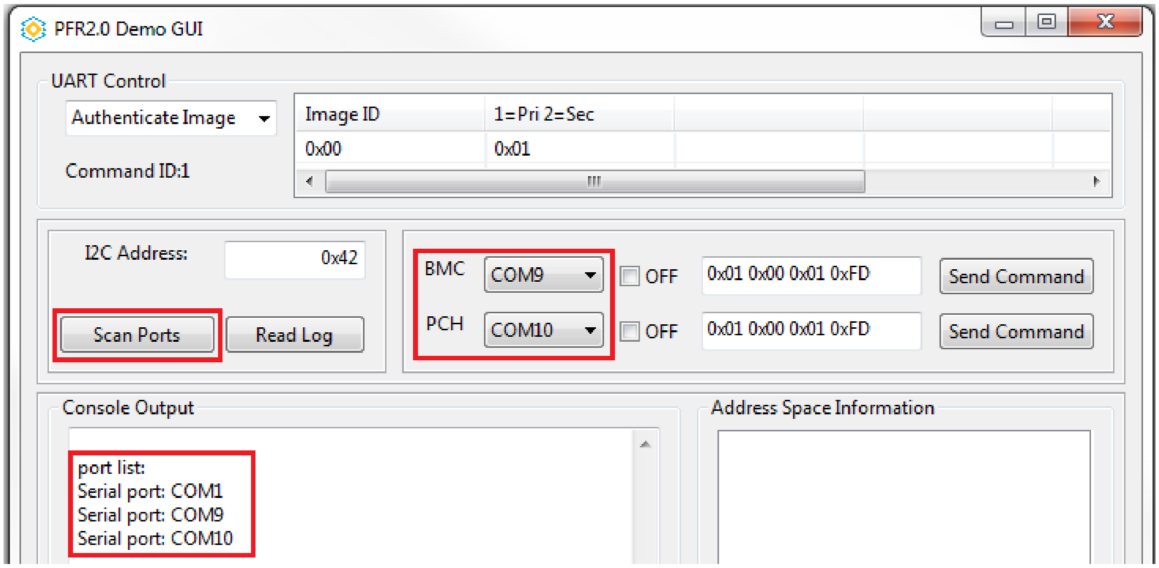

Figure 7.2 COM Port Scan of the Lattice PFR Demo Tool

5. Clicking the OFF check box for BMC to open the port and establish the connection between GUI and BMC. If the BMC port can be opened successfully, the OFF check box is changed to ON. See Figure 7.3. All logs are listed in the Console Output area. For PCH, the operation is similar.

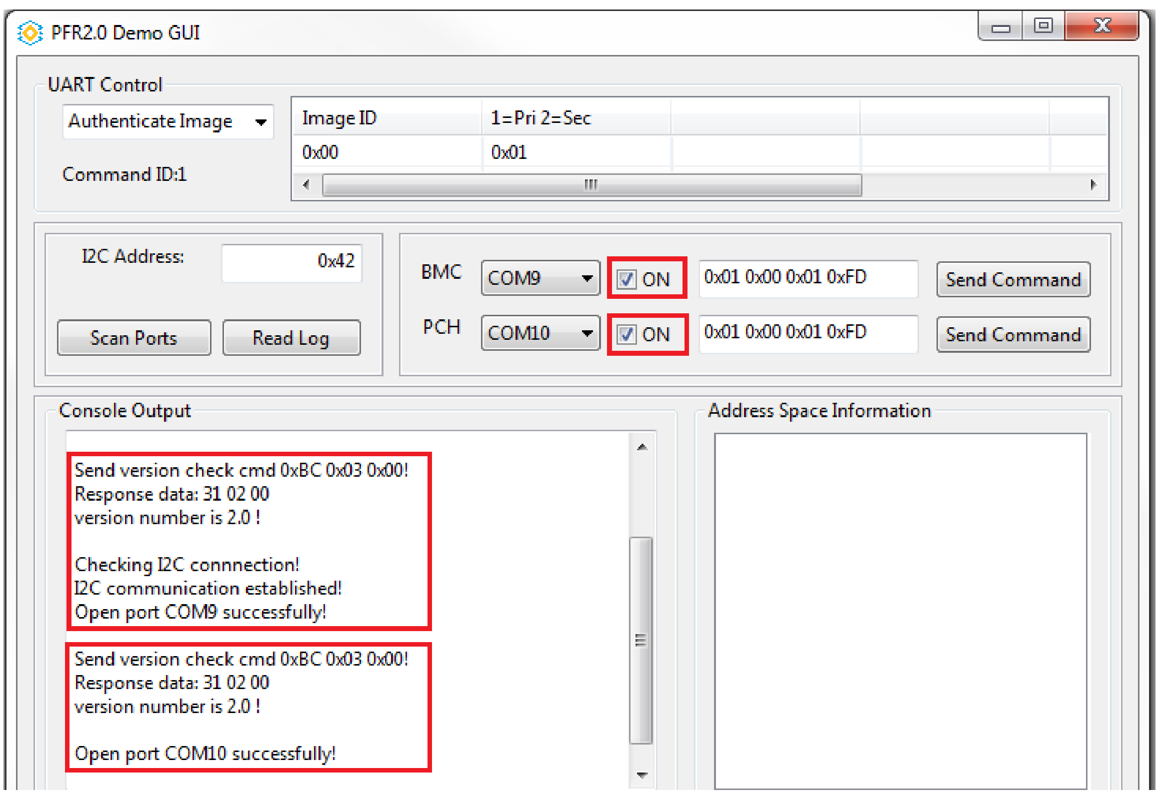

Figure 7.3 **Enable Lattice PFR Demo Tool**

6. Click the Clear button to clear the message log in the Console Output window.
7. In the UART Control section, you can select a command and change the parameters for the corresponding command. The message for this command is generated automatically.

8. Clicking *Send Command* can send selected command and receive the response. All logs are shown in the Console Output window. See Figure 7.4.

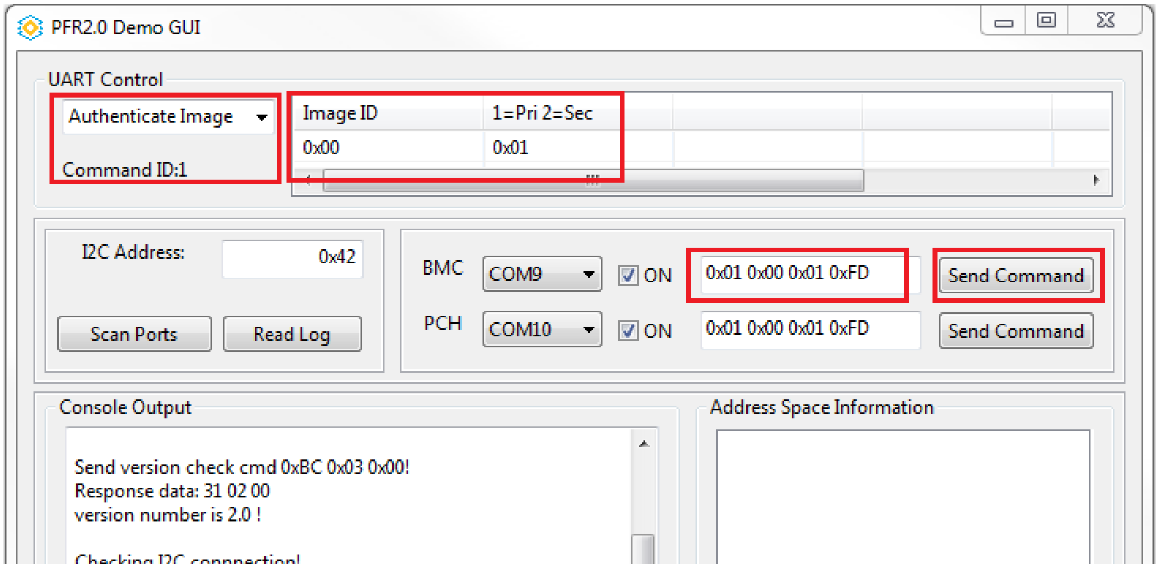

Figure 7.4. Send Command of Lattice PFR Demo Tool

9. Clicking *Read Log* reads one log entry at a time. Logs are available for Authentication, Recovery, and SPI Exceptions. When the Current and Last Index values are the same, there are no more log entries. See Figure 7.5.

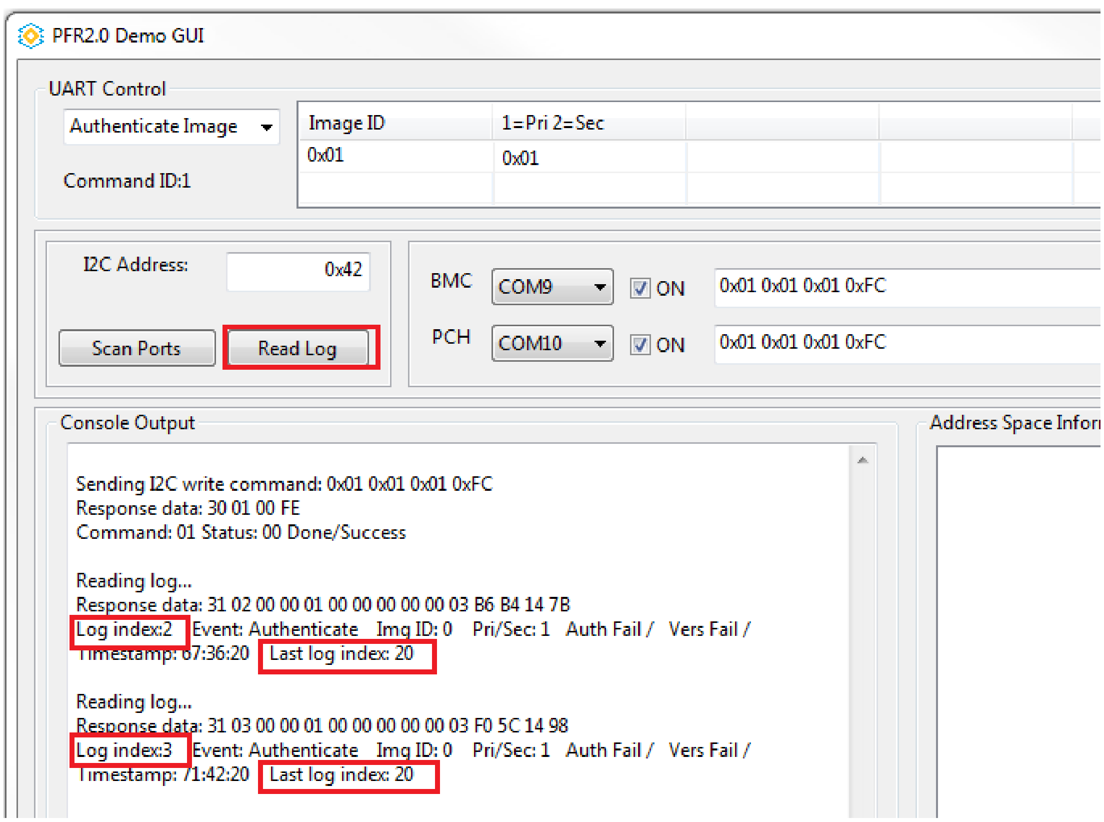

Figure 7.5 Logging of Lattice PFR Demo Tool

10. Clicking *Read Address Space* retrieves the information of the manifest from UFM2 in MachXO3D device. In the Address Space Information area, the Flash0 tab is for the BMC port and the Flash1 tab is for the PCH port. See Figure 7.6.

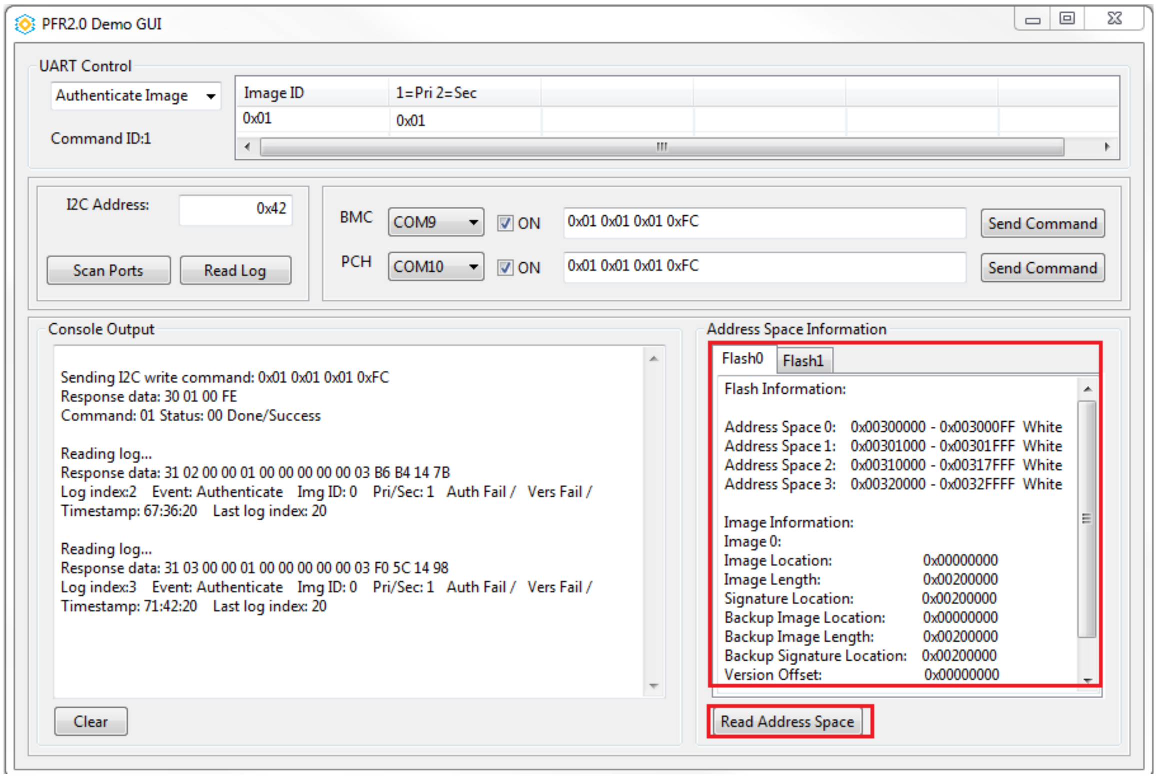

Figure 7.6 Read Address Space of Lattice PFR Demo Tool

For the detail definition of the commands, refer to the Write Commands and Read Commands sections of the MachXO3D Platform Firmware Resiliency Out-of-Band I^{2}C Command Protocol User Guide (FPGA-UG-02032).

7.2. Key Feature Validation Method

Lattice Propel provides several methods which can be used to validate the PFR functionalities at different levels. When you design a PFR solution using Lattice Propel, functions from basic register access to system-level can all be validated in the simulation environment. At board-level validation, key features for PFR system, including authentication, protection, and recovery are necessary. Lattice Propel provides tool set to validate the basic features on demo board.

7.2.1. Function Simulation

Follow steps below, you can form Functional Simulation at multiple levels:

1. Register access testing for all available registers. Special registers, such as write-only registers, are not covered at this stage, in order to make sure the correctness of SOC connection, address map, and basic quality of RTLs of SOC and IP.
2. Functional simulation for all available IP BSP to ensure each standalone IP works as expected.
3. Build up the system-level simulation environment which is aligned with maximum real application hardware environment, and then use firmware directly as stimulus to do the system-level simulation.

For Step 1 above, write and readback scenario are used as the starting point.

For Step 2 above, the functionality of each IP plus BSP is the key focus.

Meanwhile, for Step 1 and Step 2, each transaction on the system bus (AHBLITE and APB buses) is traced from end to end with address map checking. The content of each transaction is also checked.

Step 3 mainly verifies the functionality of the system-level usage defined in firmware.

An internal UVM-based simulation platform has been developed to support verification of all levels. Each level of verification can be enabled/customized using a unified configuration interface.

An external user can have a customized simulation environment which can be run using Active-HDL.

Lattice Propel provides a utility, Lattice PFR Demo Tool, which allows you to operate all PFR I^2C commands to implement and validate the PFR Key functionality.

7.2.2. Authentication

As stated in the Boot Up Protection section, the PFR system authenticates BMC/PCH image at boot-up stage. For function validation, you can use a command to perform image authentication manually.
The command should be selected with correct arguments in the Lattice PFR Demo Tool.
To force authentication for the Primary image in Flash0, select the command 'Authenticate Image' and modify the value in the right command parameter table (Figure 7.7), then it generates the whole command *0x01 0x00 0x01 0xFD*. Click the *Send Command*. You can see a Console Output message (Figure 7.7), if it was executed successfully.

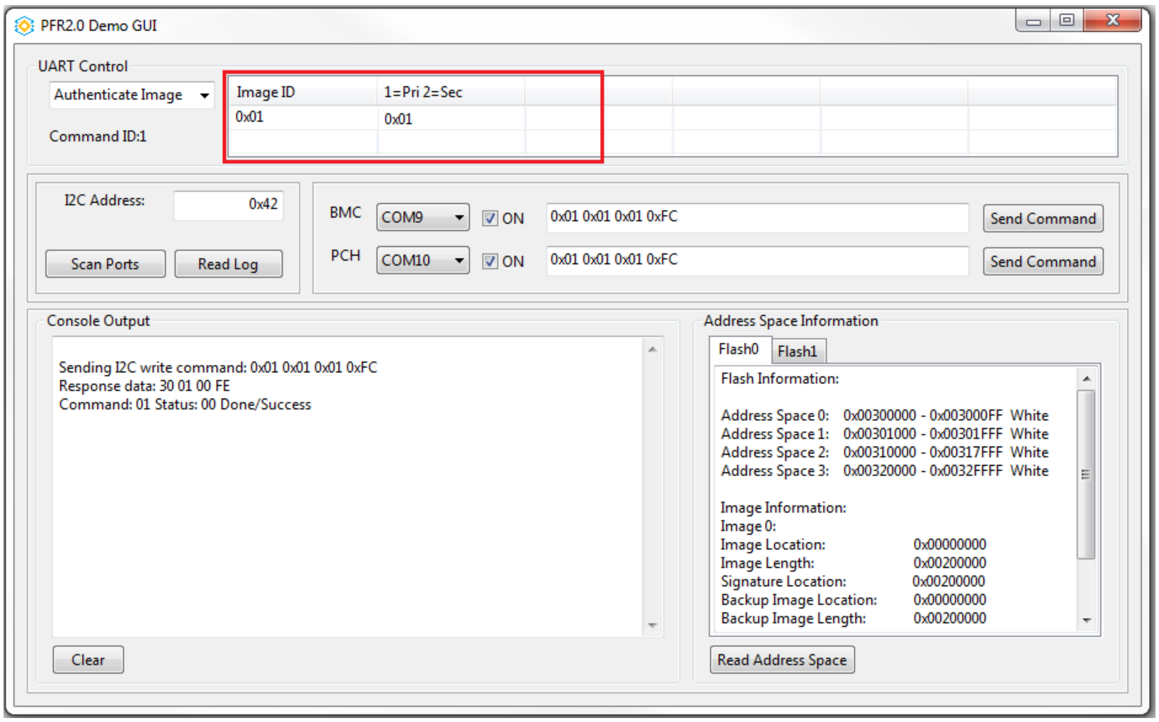

Figure 7.7. BMC Image Authentication for Flash 0

Authenticate Image (0x01 0x00 0x01 0xFD) – to authenticate Primary image in Flash0
Authenticate Image (0x01 0x00 0x02 0xFC) – to authenticate Secondary image in Flash0
Authenticate Image (0x01 0x01 0x01 0xFC) – to authenticate Primary image in Flash1
Authenticate Image (0x01 0x01 0x02 0xFB) – to authenticate Secondary image in Flash1

Next, check all of the security logs by clicking *Read Log*, and the latest log should be "Event: Authenticate Img ID: 0 Pri/Sec: 1 Auth Pass / Vers Pass /", which is corresponded to the previous command *0x01 0x00 0x01 0xFD*, as shown in Figure 7.8.

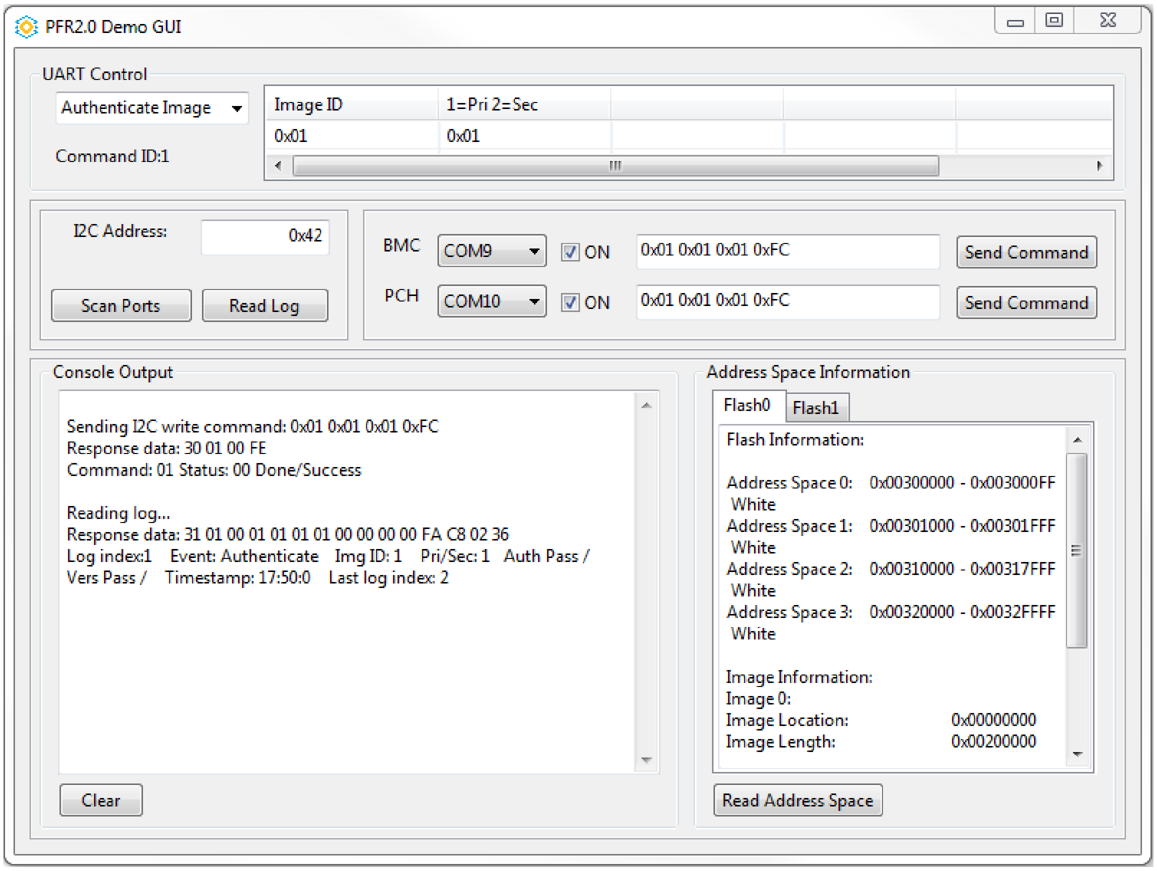

Figure 7.8. Get logs for image authentications

7.2.3. Protection

Click *Read Address Space* to get the Address Space information for Flash0 and Flash1. All White Spaces are also listed, as shown in Figure 7.8, which was configured in Manifest file as default.

7.2.3.1. Legal Operation (Operate on White Space)

Read 16 bytes starting from 0x00300000 in Flash0 (White Space), program a value (0x5A) to 0x00300003, and read back the bytes again.

Flash Page Read (0xF3 0x00 0x30 0x00 0x00) – to read 16 bytes started from 0x00300000 in Flash0. The read back data is all 0xff, as Figure 7.9 shows.

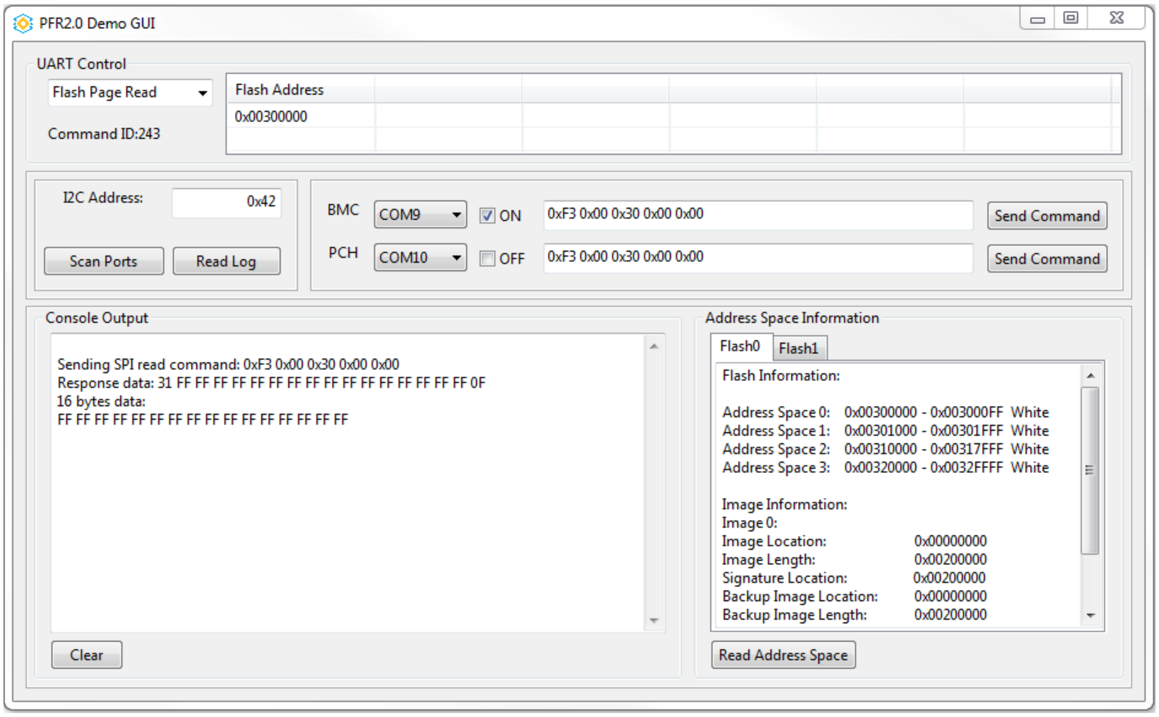

Figure 7.9. Initial value of 0x00300000~0x0030000F

Flash Sector Erase (0xF0 0x00 0x30 0x00 0x00 0x01) – to erase the sector started from 0x00300000 in Flash0.
Flash Byte Write (0xF4 0x00 0x30 0x00 0x03 0x5A) – to write a value (0x5A) to 0x00300003 in Flash0.
Flash Page Read (0xF3 0x00 0x30 0x00 0x00) – to read 16 Bytes started from 0x00300000 in Flash0 with above steps,

As Figure 7.10 shows, the address 0x00300003 was programmed with 0x5A successfully, for 0x00300003 is in White Address List space 0.

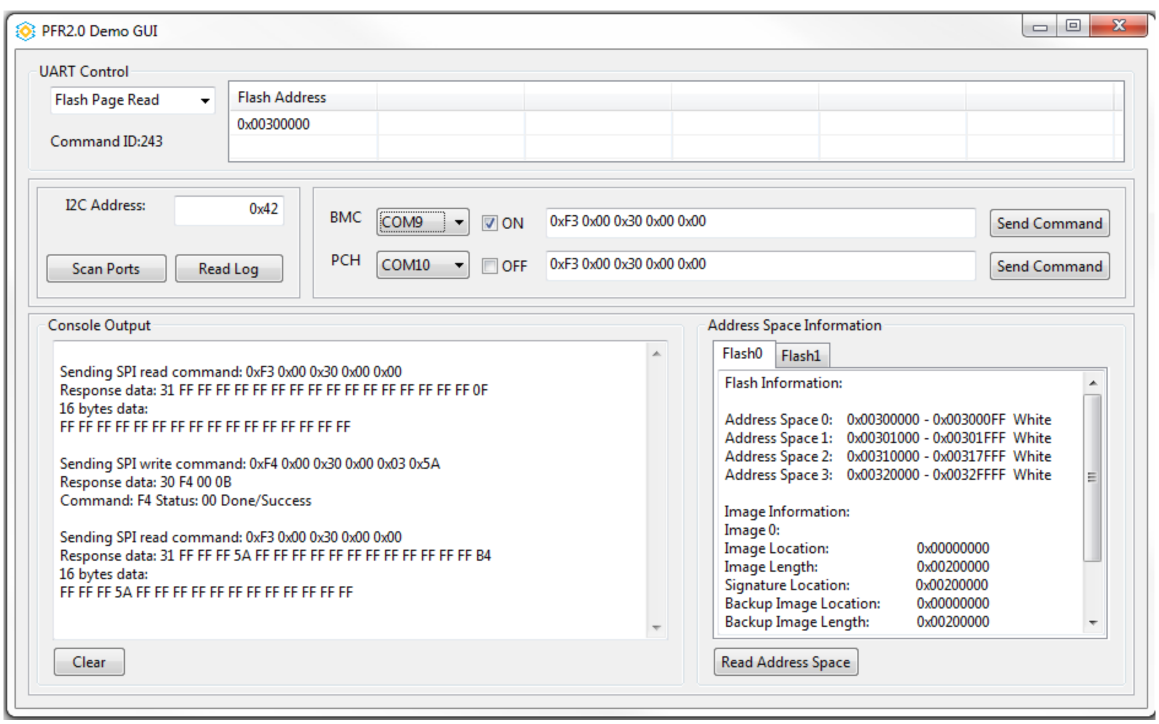

Figure 7.10. Value of 0x00300000~0x0030000F After write

7.2.3.2. Illegal Operation (operate on Black Space)

Reading 16 bytes started from 0x00300100 in Flash0, program a value (0x5A) to 0x00300103, and read back the bytes again. Follow steps below:
Enable SPI Filter (0x16 0x00 0x01 0xE8) – to enable all commands for filtering on BMC SPI port
Flash Page Read (0xF3 0x00 0x30 0x01 0x00) – to read 16 Bytes started from 0x00300100 in Flash0
Flash Byte Write (0xF4 0x00 0x30 0x01 0x03 0x5A) – to write a value (0x5A) to 0x00300103 in Flash0
Flash Page Read (0xF3 0x00 0x30 0x01 0x00) – to read 16 Bytes started from 0x00300100 in Flash0.
After running above steps, Figure 7.11 shows that the address 0x00300103 is still 0xFF, was not programmed with 0x5A successfully. 0x00300103 is out of White Address List data space 0, so it cannot be programmed.

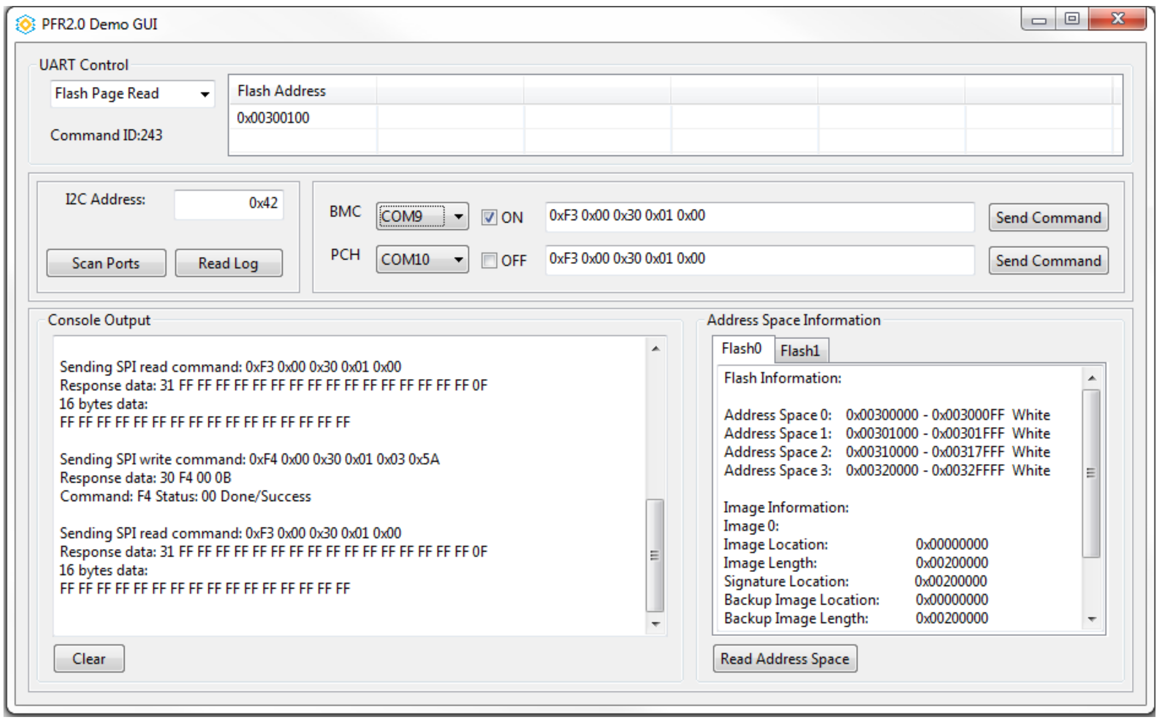

Figure 7.11. Value of 0x00300100~0x0030010F after write

Using the Read log operation, an SPI Exception Event is printed in detail by Lattice PFR Demo Tool, as shown in Figure 7.12. The illegal command is captured as the Flash Byte Write to BMC Flash0.

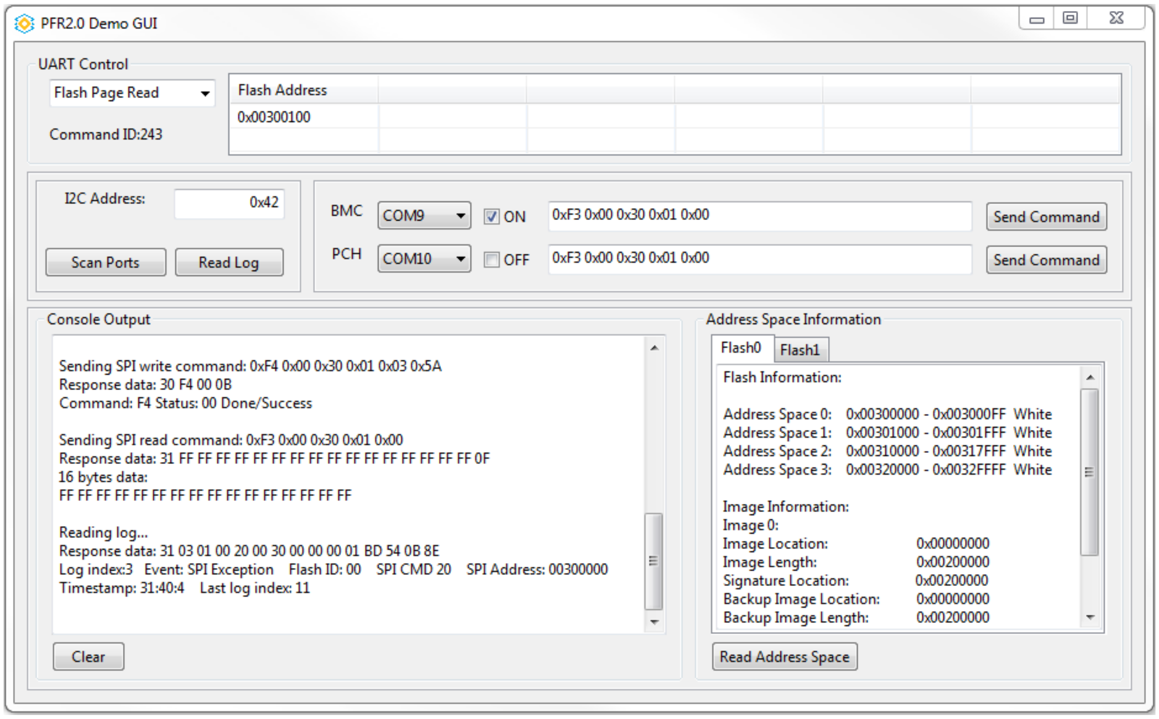

Figure 7.12. Logs of Illegal Operation

7.2.4. Recovery

Image recovery is demonstrated by manually destroying the image and recovering it from a known good image.

7.2.4.1. Manual Image Destroy

Disable all commands filtering for BMC. Then erase the sector starting from 0x00100000 in Flash0 to destroy Primary image in Flash0. Authenticate Primary image after destroying the Primary image. Authentication should fail, as Figure 7.13 shows. Follow steps below:

Enable SPI Filter (0x16 0x00 0x00 0xE9) – to disable all commands for filtering on BMC SPI port

Flash Sector Erase (0xF0 0x00 0x10 0x00 0x00 0x01) – to erase the sector started from 0x00100000 in Flash0

Authenticate Image (0x01 0x00 0x01 0xFD) – to authenticate Primary image in Flash0

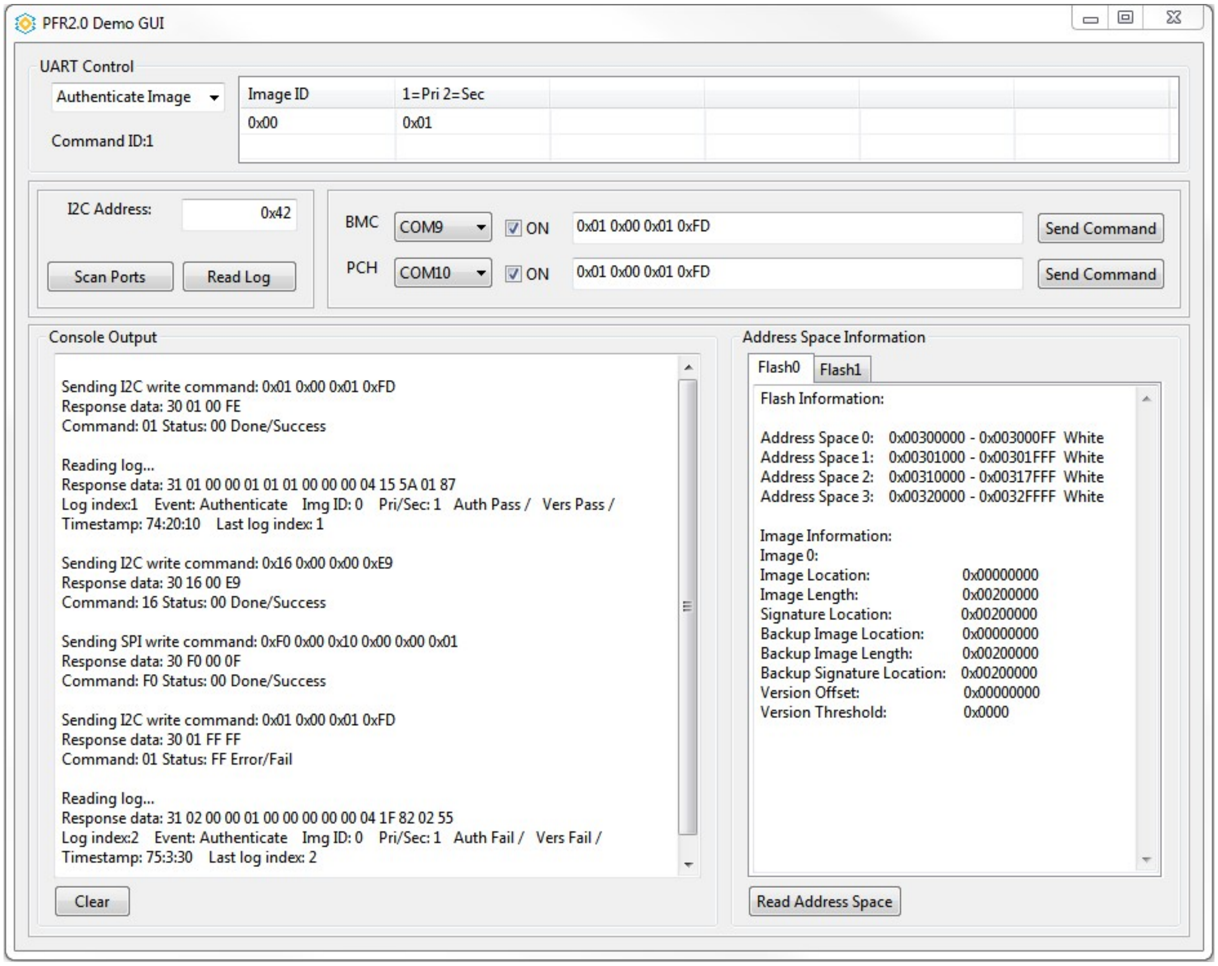

Figure 7.13. Authentication Failed with Destroyed Image

7.2.4.2. Manual Image Recovery

Select the command *Recovery Image* and modify the value in the right command parameter table (Figure 7.14). It generates the whole command *0x02 0x00 0x01 0xFC*. Click *Send Command*. If successful, the console output appears with messages, as shown in Figure 7.14.

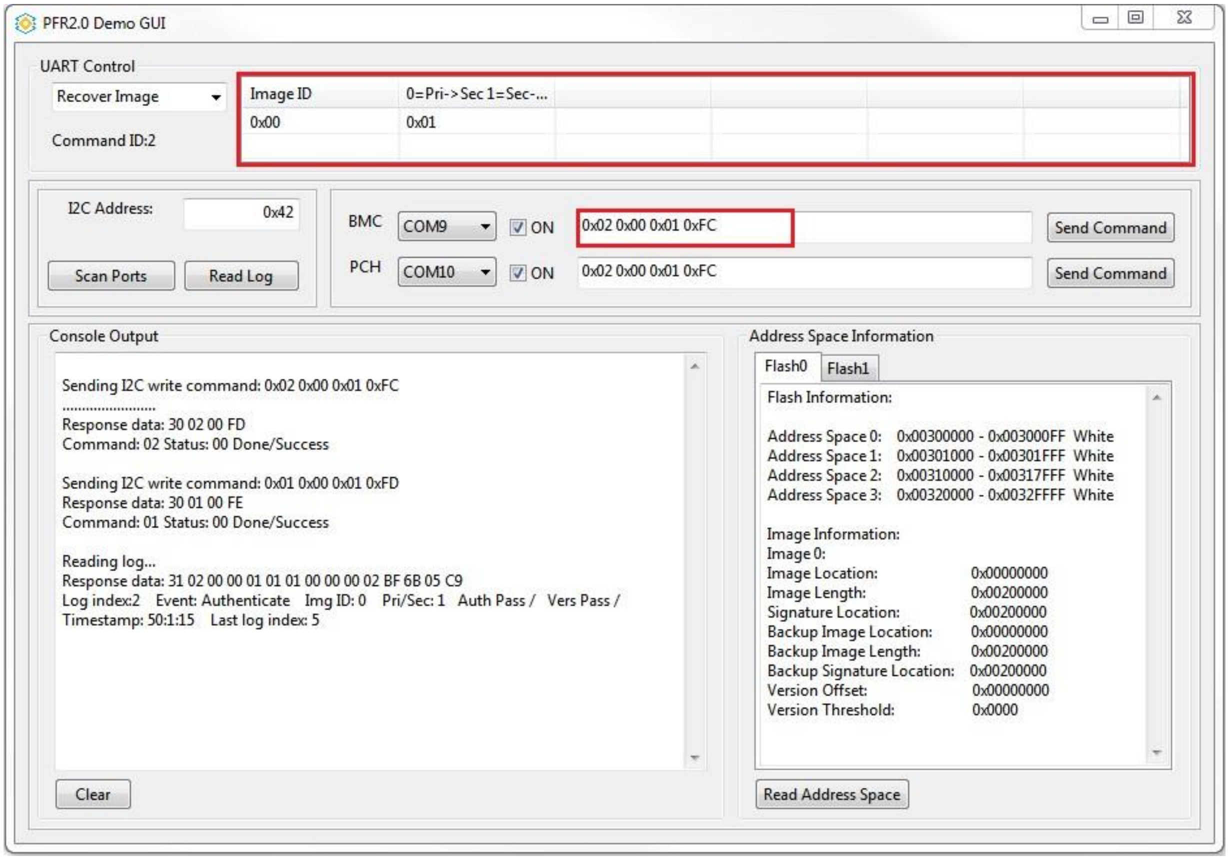

Figure 7.14. Authenticate Primary Image after Recovery Done

Recover Image (0x02 0x00 0x01 0xFC) – to recover BMC image to Primary with Secondary (good image) in Flash0.
Authenticate Image (0x01 0x00 0x01 0xFD) – to authenticate Primary image in Flash0.

Reference

- Lattice Sentry PLD Interface IP Core for MachXO3D - Lattice Propel Builder (FPGA-IPUG-02106)
- Lattice Sentry Embedded Security Block Mux IP Core for MachXO3D - Lattice Propel Builder (FPGA-IPUG-02107)
- Lattice Sentry I^2C Monitor IP Core for MachXO3D - Lattice Propel Builder (FPGA-IPUG-02108)
- Lattice Sentry QSPI Master Streamer IP Core for MachXO3D - Lattice Propel Builder (FPGA-IPUG-02109)
- Lattice Sentry QSPI Monitor IP Core for MachXO3D - Lattice Propel Builder (FPGA-IPUG-02110)
- Lattice Propel 1.0 User Guide (FPGA-UG-02110)

Revision History

Revision A, July 2020

Section	Change Summary
All	Initial release.

www.latticesemi.com