Lattice Propel 1.1 Root-of-Trust Reference Design

User Guide

FPGA-RD-02243-1.1

March 2022

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, with all faults and associated risk the responsibility entirely of the 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
FW	Firmware
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
MCTP	Management Component Transport Protocol
PFR	Platform Firmware Resiliency
QSPI	Quad Serial Peripheral Interface
OOB	Out of Band
PCH	Platform Controller Hub
PFR	Platform Firmware Resiliency
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
RTRec	Root of Trust for Recovery
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.
SFB	SoC Function Block
SHA	Secure Hash Algorithm
SMBus	System Management Bus
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 Mach-NX 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 the Mach-NX device to be used as a Root-of-Trust hardware solution in a complex system. With Lattice Mach-NX 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 Sentry PFR solution on the Mach-NX device 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.

Contents in this document are the Mach-NX 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 Mach-NX PFR Root of Trust (RoT) solution, including system architecture, functionality overview, and principles supporting firmware resiliency.
- PFR System Architecture and Runtime Flow section — Describes the Lattice Mach-NX 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 PFR 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). A 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. A RoT device must be designed to always behave in the expected manner. Proper function of the device is essential to providing security-specific functions. If this device is unchecked, faulty behavior cannot be detected. A RoT is typically the first element in a Chain of Trust (CoT) and can serve as an anchor for the chain to deliver more complex functionality.

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

- The Root of Trust for Update (RTU) is responsible for authenticating firmware updates and critical data changes to support platform protection.
- The Root of Trust for Detection (RTD) is responsible for firmware and critical data corruption detection.
- 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 Mach-NX FPGA can serve as the Root of Trust and can provide the following services:

- Image Authentication: On system power-up or reset, Mach-NX device holds the protected devices in reset while it authenticates their boot images in SPI flash. After each signature authentication passes, Mach-NX device releases each reset, and those devices can boot from their authenticated SPI flash image. Image authentication can also be requested at any time through the Out of Band (OOB) communication path.
- Image Recovery: If a flash image becomes corrupted for any reason, it fails to be authenticated. The Mach-NX device can restore it to a known good state by copying from an authenticated backup image.
- SPI Flash Monitoring and Protection: The Mach-NX device can monitor multiple SPI/QSPI buses for unauthorized activity and block unauthorized accesses using external quick switches. The monitors can be configured to check for specific SPI flash commands and address ranges defined by the system designer and designate them as authorized (whitelisted) or unauthorized (blacklisted).
- Event Logging: Mach-NX device logs security events, such as unauthorized flash accesses and notifies the BMC.
- SMBus Filtering: The Mach-NX device can monitor a SMBus for unauthorized activity and filter the unauthorized transactions. The monitor can be configured with multiple whitelist or blacklist filters to watch for specific commands defined by the system designer and designate them as authorized or unauthorized SMBus transactions.

2.4. System Architecture

Figure 2.1 shows the architecture of a Lattice Mach-NX FPGA working as a RoT device. The system design consists of the SFB module, which integrates 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 in SFB module include the Mach-NX hard GPIO, UART, JTAG, and SMBus Mailbox, as shown in Figure 2.1. These modules perform the basic board-level controls and communications. PFR solution Peripherals include Secure Enclave, QSPI Streamer/Monitor, SMBus Filter and Customer PLD interface, which perform the main PFR functionalities. You can add or remove the peripherals using the Lattice Propel tools upon your design requirement. For details of customization, refer to the PFR System Design (from Lattice Propel) section.

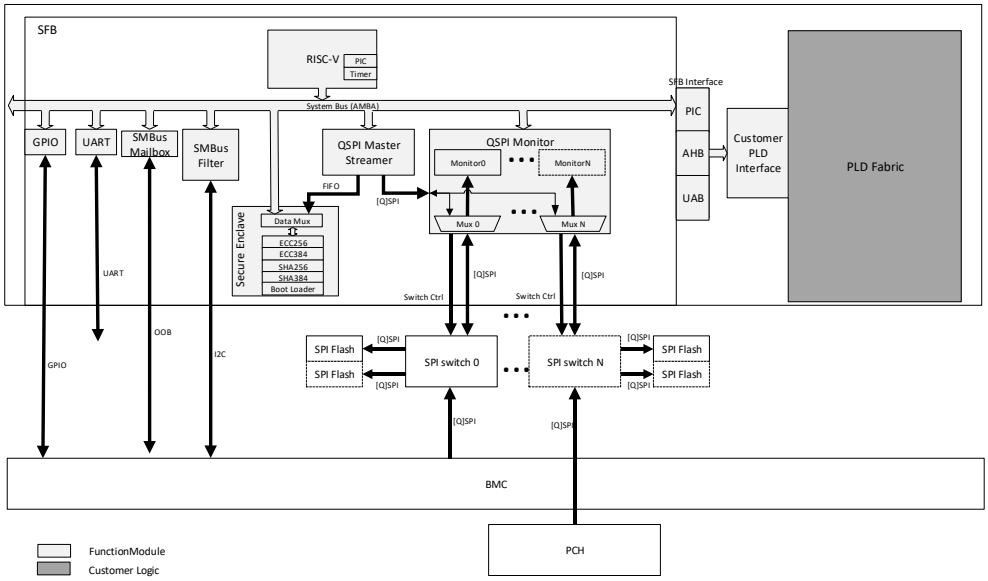

Figure 2.1. Lattice PFR System Architecture

2.5. Functionality Overview

2.5.1. Mach-NX SoC Function Block

SoC Function Block (SFB) is a hard module in Mach-NX device mainly designed for Lattice Sentry PFR solution. It contains RISC-V processor, PFR solution-specific function modules, and other general modules for communication with BMC and PCH.

2.5.1.1. RISC-V Processor

The RISC-V Processor provides the main control function in Mach-NX SFB block. The processor 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.

2.5.1.2. Lattice Sentry Secure Enclave

The Secure Enclave is a security block that provides a set of security services for Mach-NX device, including ECC256, ECC384, SHA256, and SHA384 crypto functions. The module has two interfaces for sending and receiving data: a register interface, and a High Speed Data Port (HSP) which is a FIFO-style interface.

Besides the security services, the Secure Enclave also has a boot loader function which performs the secure boot for the whole system.

For the system software developer, refer to the PFR IP API Reference section for more details on the API reference.

2.5.1.3. Lattice Sentry QSPI Master Streamer

Lattice Sentry QSPI Master Streamer is a configurable SPI master that supports single, dual and quad modes. It contains FIFOs for Tx and Rx data, which supports long SPI transactions (more than 32 bits). It also provides an external 8-bit Rx FIFO interface that can be connected to the Secure Enclave for image authentication.

QSPI Streamer incorporates a 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 enables direct transmission of input data from the SPI slave to another block, such as the Secure Enclave which frees up the CPU or system bus.

For the system software developer, refer to the PFR IP API Reference section for more details on the API reference.

2.5.1.4. 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 checks the external buses for allowed flash commands and flash addresses. This block 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 command, or be blacklisted for read commands. Both 24-bit and 32-bit flash addressing are supported.

For the system software developer, refer to the PFR IP API Reference section for more details on the API reference.

2.5.1.5. Lattice Sentry System Management Bus (SMBus) Filter

The SMBus filter is a configurable security module which can monitor traffic on the SMBus to identify unauthorized activity, based on set of up to 256 programmable filters. If unauthorized activity is detected, the SMBus is disabled and PFR firmware is notified so that an event can be logged.

For the system software developer, refer to the PFR IP API Reference section for more details on the API reference.

2.5.1.6. General Peripherals

Besides the PFR solution peripherals, SFB also integrates some general peripherals for board-level control or communication, including GPIO, UART, SMBus Mailbox. You can use one or more of these modules based on the system requirement.

2.5.2. Mach-NX SFB Interface

2.5.2.1. Customer PLD Interface

The Customer PLD Interface is a register-based interface which is used to send and receive messages between the PFR firmware and the customer control PLD logic. It can be used to request system control actions, to check status, or to send customized messages. You may want 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 the system software developer, refer to the PFR IP API Reference section for more details on the API reference.

2.5.2.2. UFM Access Module (UAB)

The UFM Access Module (UAB) is a functional block inside the SFB interface for accessing the internal flash memory of Mach-NX device. Through the UAB block, PFR solution firmware can access the manifest of the system and runtime log event data.

3. PFR System Architecture and Runtime Flow

3.1. Firmware Architecture

The Lattice PFR solution of Mach-NX device 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 3.0 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 PFR 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.

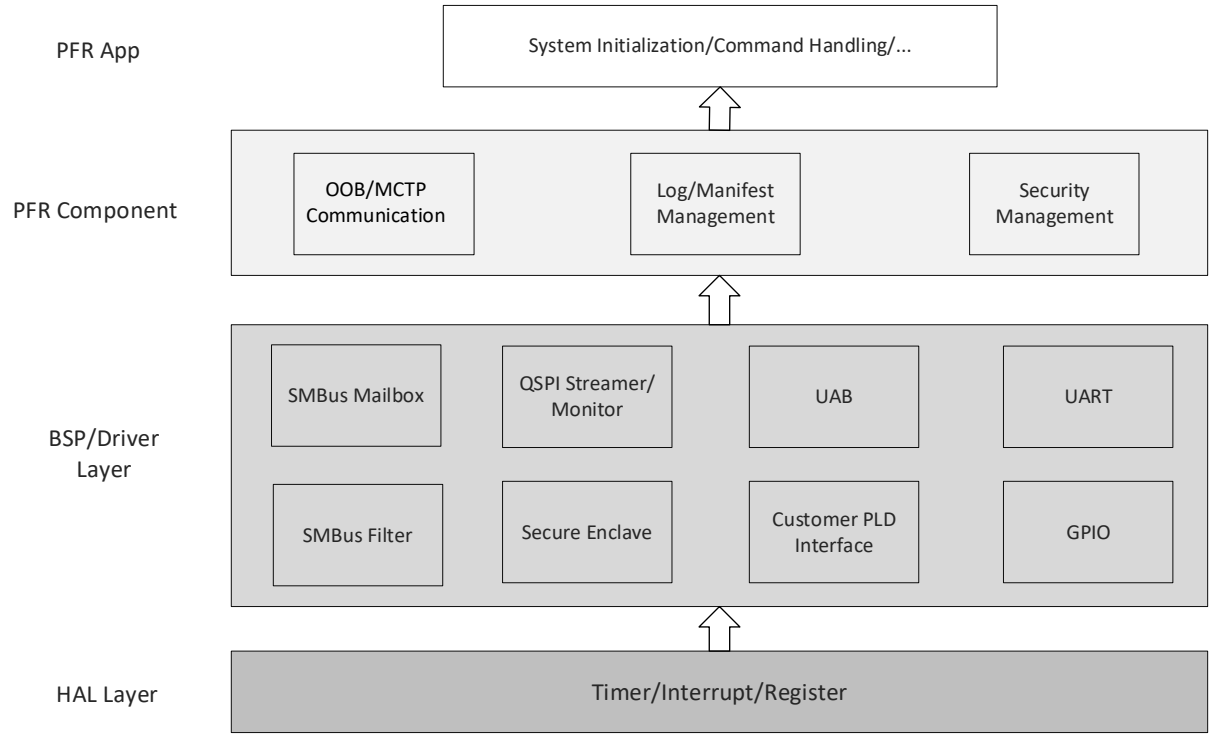

Figure 3.1. Software Architecture of Lattice PFR Solution

3.2. Bootloader

The Bootloader performs the secure boot function after the system is power on and is responsible for loading customer firmware from the external flash. The boot up flow is shown in Figure 3.2.

During the boot up flow, Bootloader will parse the flash configuration data in UFM3 of Mach-NX device, for the detail of the flash configuration in UFM3, please refer to the Flash Address Tool section.

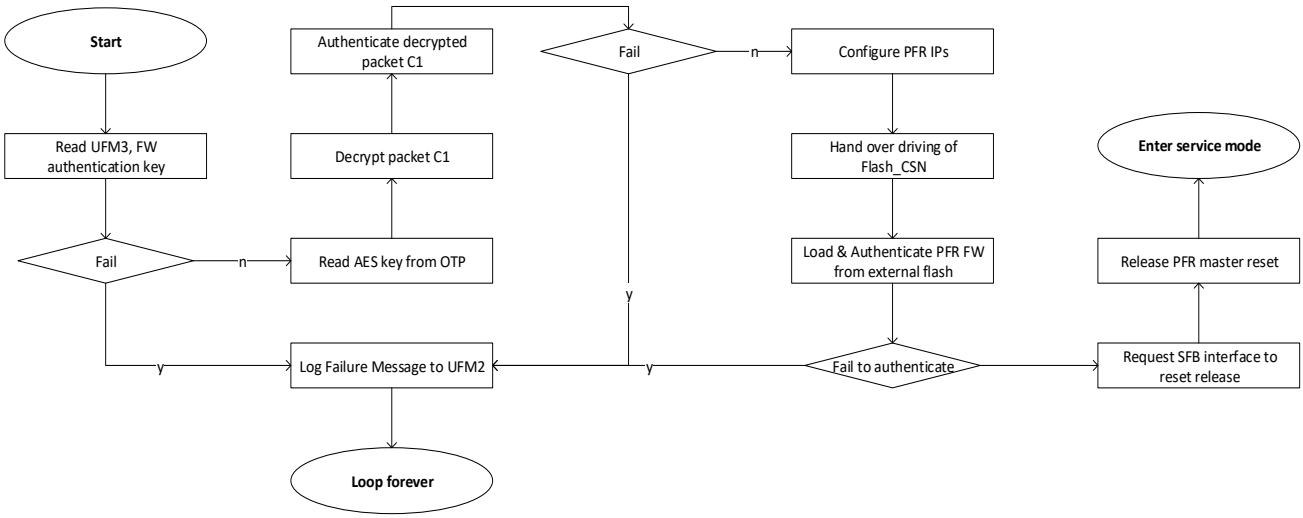

Figure 3.2. Customer PFR Firmware Boot Up Flow

3.3. Runtime Flow

The firmware runtime flow comprises the following major steps (as shown in Figure 3.3):

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 details.
4. Invalid SPI/SMBus Event Detection and Protection: Monitor and detect the system SPI/SMBus events to 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.

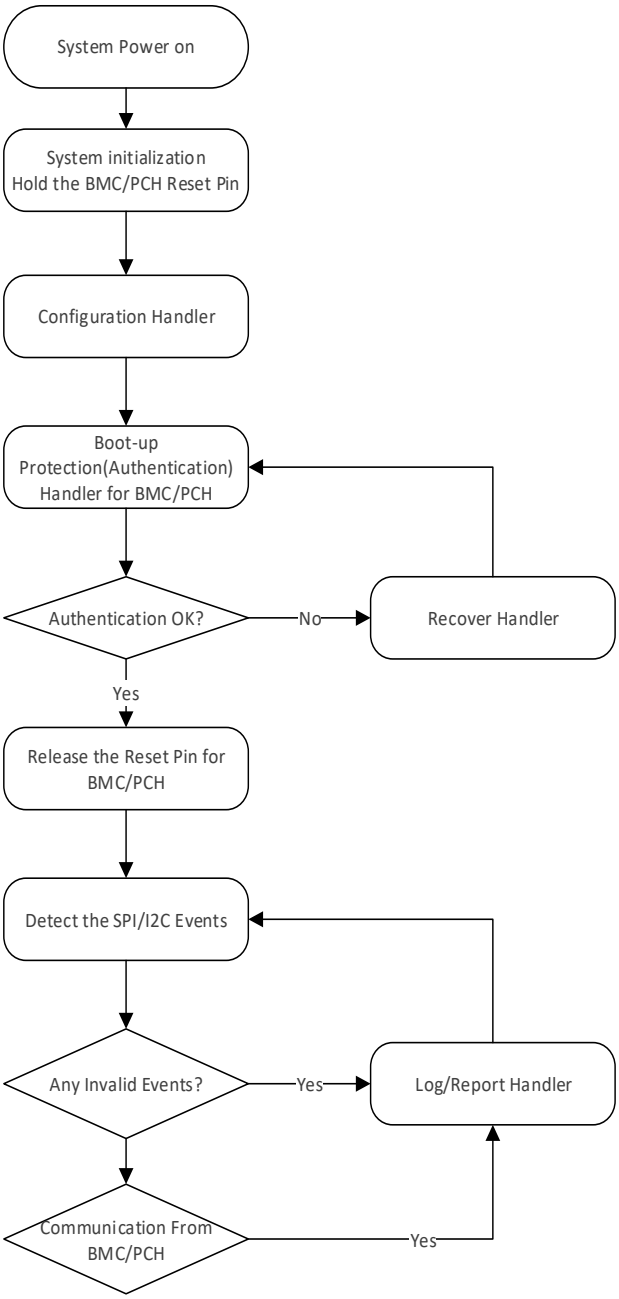

Figure 3.3. Lattice PFR Runtime Flow

3.4. Configuration

System dependent information is configured as a manifest, which is stored in the UFM of Lattice Mach-NX FPGA device. The system manifest is a data structure which provides crucial information such as flash layout, signature, and keys, for each firmware to store, authenticate and monitor on the SPI flash in the system.

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

During the runtime, the system software reads the manifest in the UFM and parses the critical data for firmware authentication, recovery and detection. Figure 3.4 shows configuration flow of Lattice PFR 3.0 Configuration Handler.

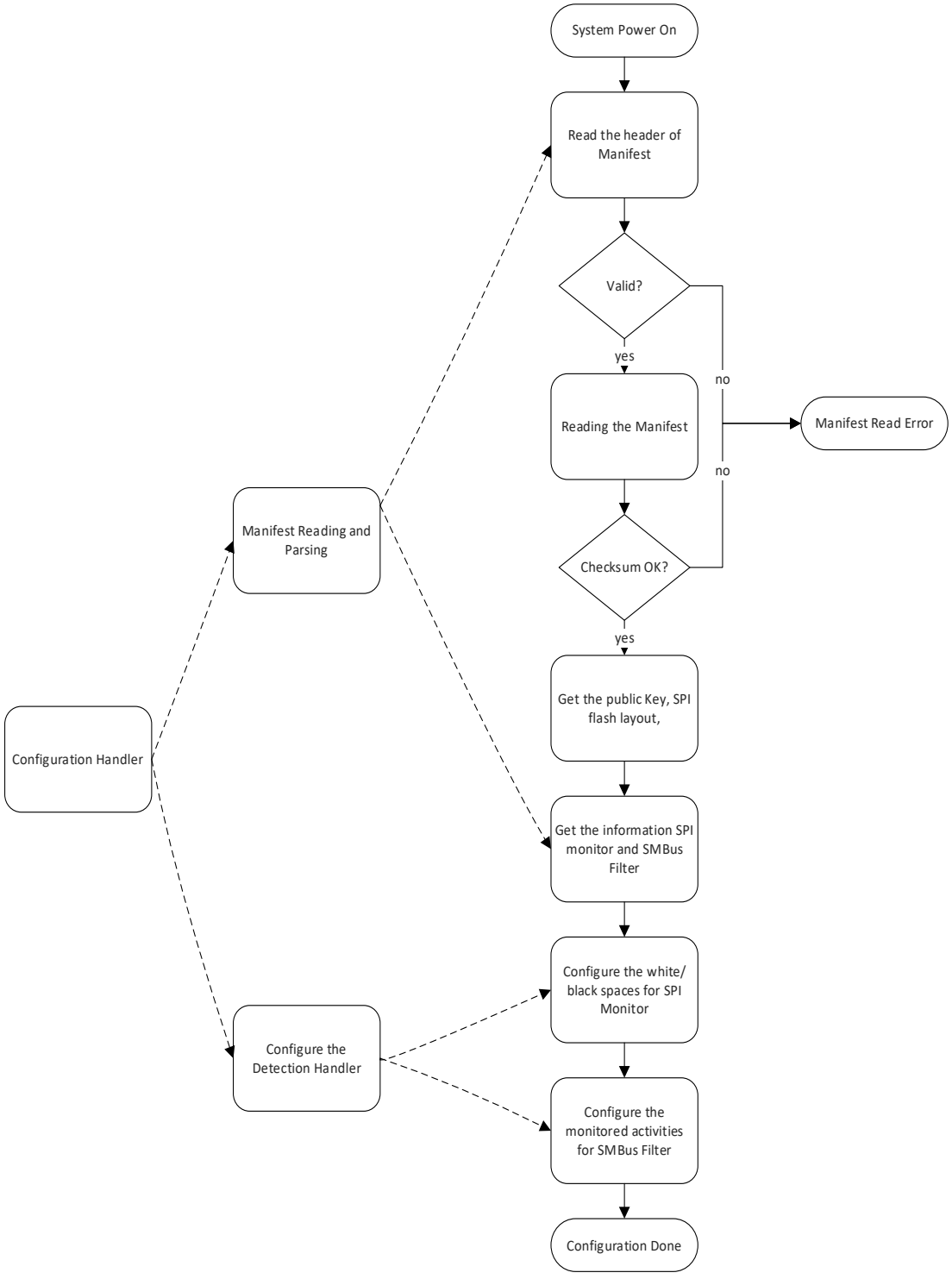

Figure 3.4. Lattice PFR 3.0 Configuration Flow

3.4.1. Mach-NX PFR Manifest Manager

Lattice Propel provides a Manifest Manager tool to manage the manifest for your own system. The Manifest is stored in UFM0 of the Mach-NX device.

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

1. Open Lattice Propel SDK. Click LatticeTools -> Lattice Sentry Tools for Mach-NX -> Lattice Sentry Manifest Manager to run manifest manager. See Figure 3.5.
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 SMBus Filter Data (Figure 3.6).
3. Click the Generate button to create the .mem file for UFM0 initialization. The .jed file is programmed into UFM0.

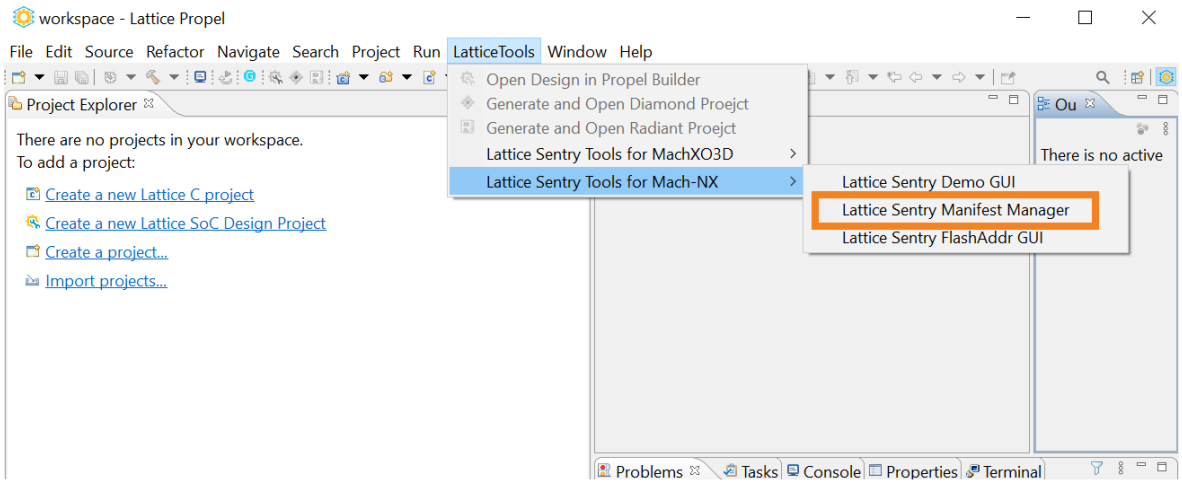

Figure 3.5. Launch Manifest Manager in Lattice Propel SDK

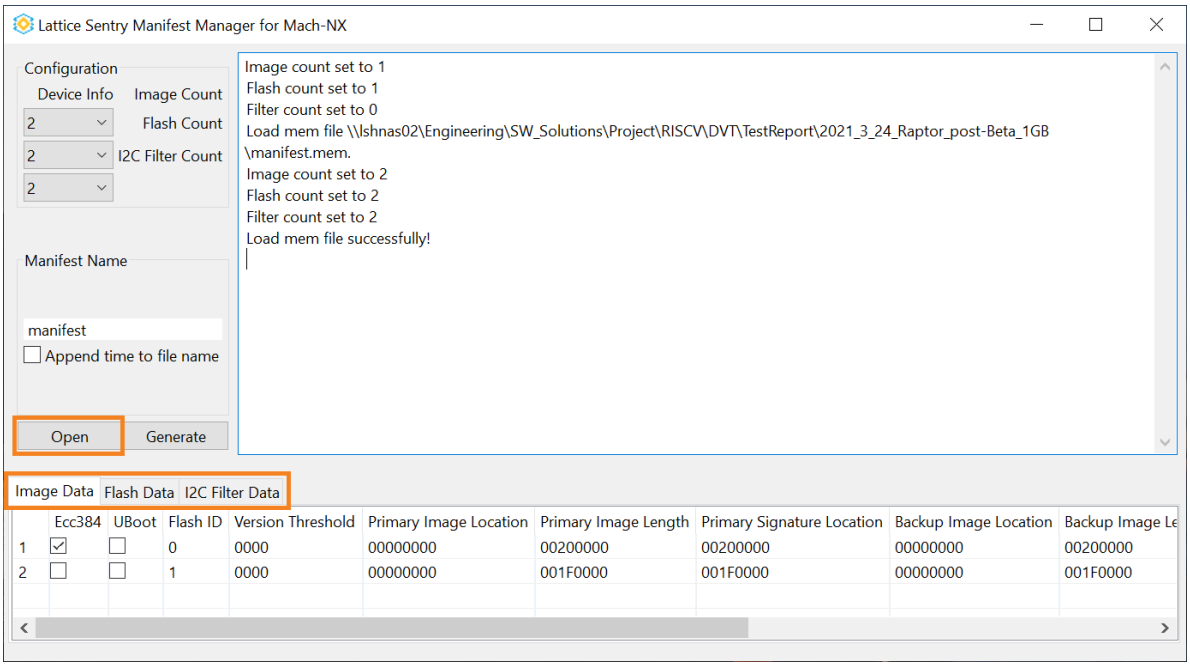

Figure 3.6. Manifest Manager Window

3.4.2. Flash Address Tool

Lattice Propel provides a Flash Address tool to configure the system flash storage related information which can be used during the system secure boot. The flash configuration data is stored in UFM3 of the Mach-NX device.

You can follow steps below to create and modify the flash configuration data for your system.

1. In Lattice Propel SDK, choose LatticeTools -> Lattice Sentry Tools for Mach-NX -> Lattice Sentry FlashAddr GUI. See Figure 3.7.
2. Click the Open button and choose the “config.jed” file from the SoC project (Figure 3.8), which is generated by Lattice Propel builder. Flash Address tool loads the “config.jed” file and parses the information. You need to input the specific data for “SoC Firmware Address” and “SFB Config Address” to match your own system for successful boot up. You also need to provide the “Recovery Target Image Address” for BMC to use in case of a boot up failure.
3. Click the Save button to create the .jed file for UFM3 initialization. The .jed file can then be programmed into UFM3.

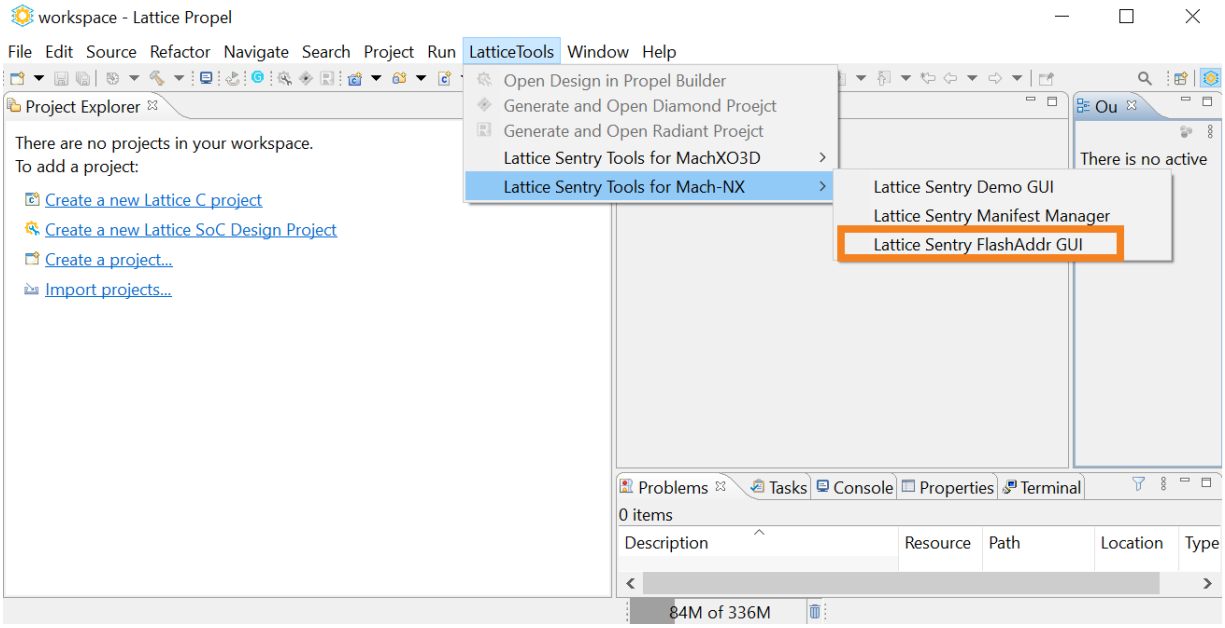

Figure 3.7. Launch Lattice Sentry Flash Address GUI

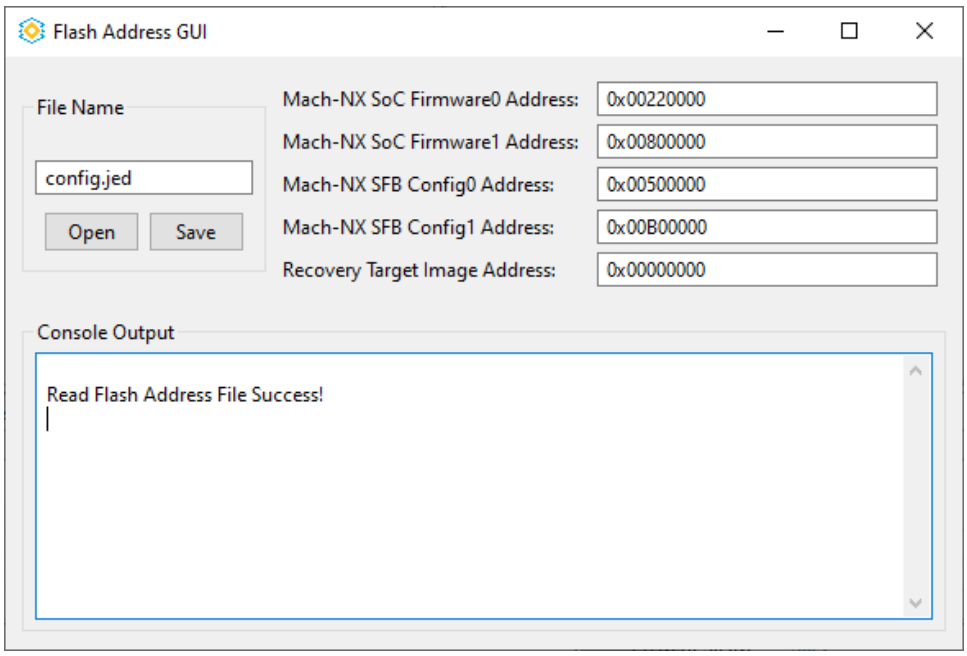

Figure 3.8. Configuration in Flash Address Tool

3.5. Boot Up Protection

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

Figure 3.9 shows the boot-up protection flow for authenticating the firmware on the SPI flash. 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.

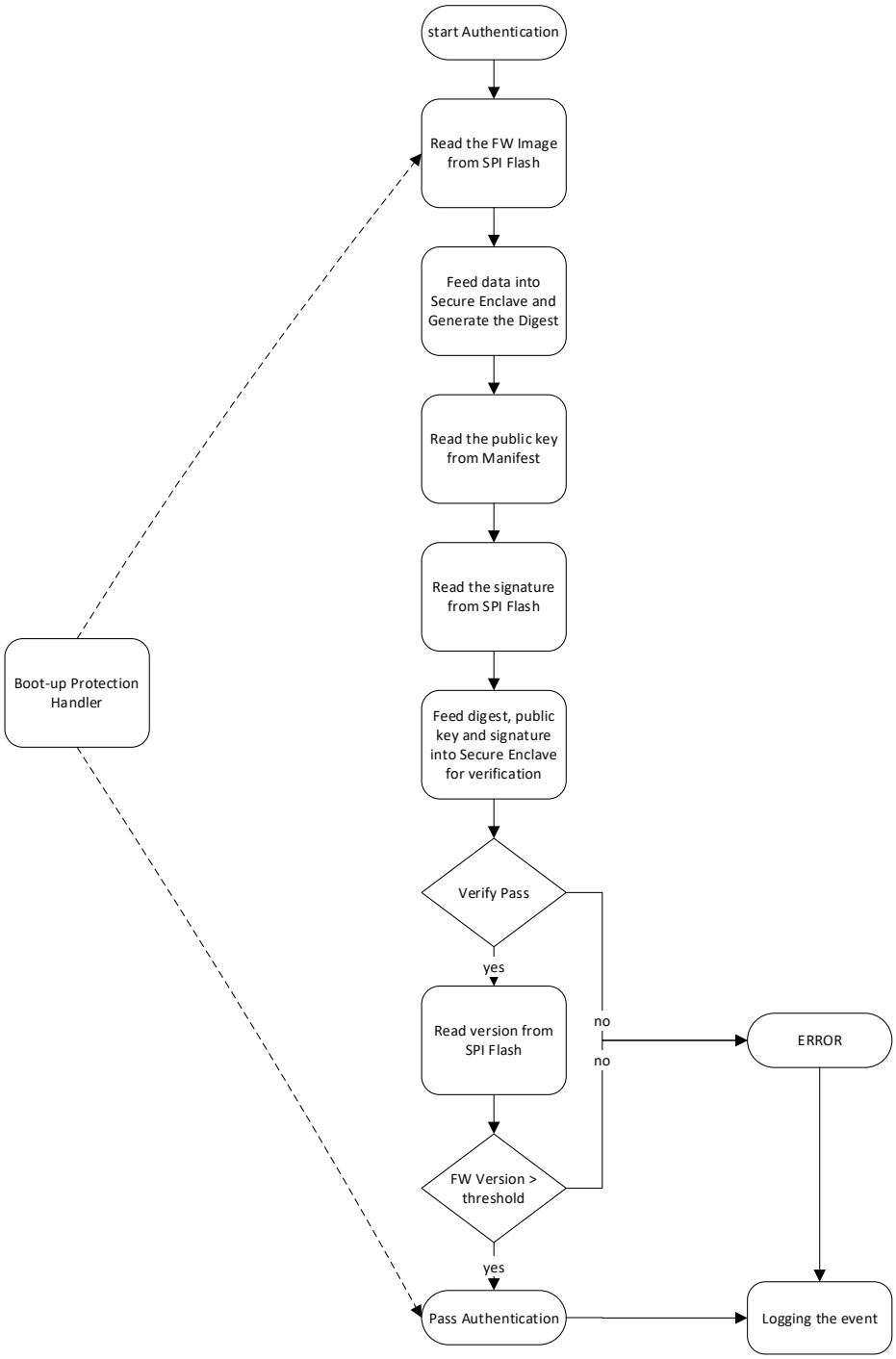

Figure 3.9. PFR Boot-up Protection Handler

3.6. Recovery

Recovery mechanism aims to keep the firmware and critical data in a valid and authorized state in case the firmware and the critical data are detected to have been corrupted. Generally, two circumstances can trigger the recovery mechanism: one is when RoT has detected the firmware has been corrupted, the other is the BMC/PCH initiates the recovery progress. After recovery, authentication is recommended to ensure the integrity of the firmware and data in the recovered flash.

Figure 3.10 shows the recovery process flow.

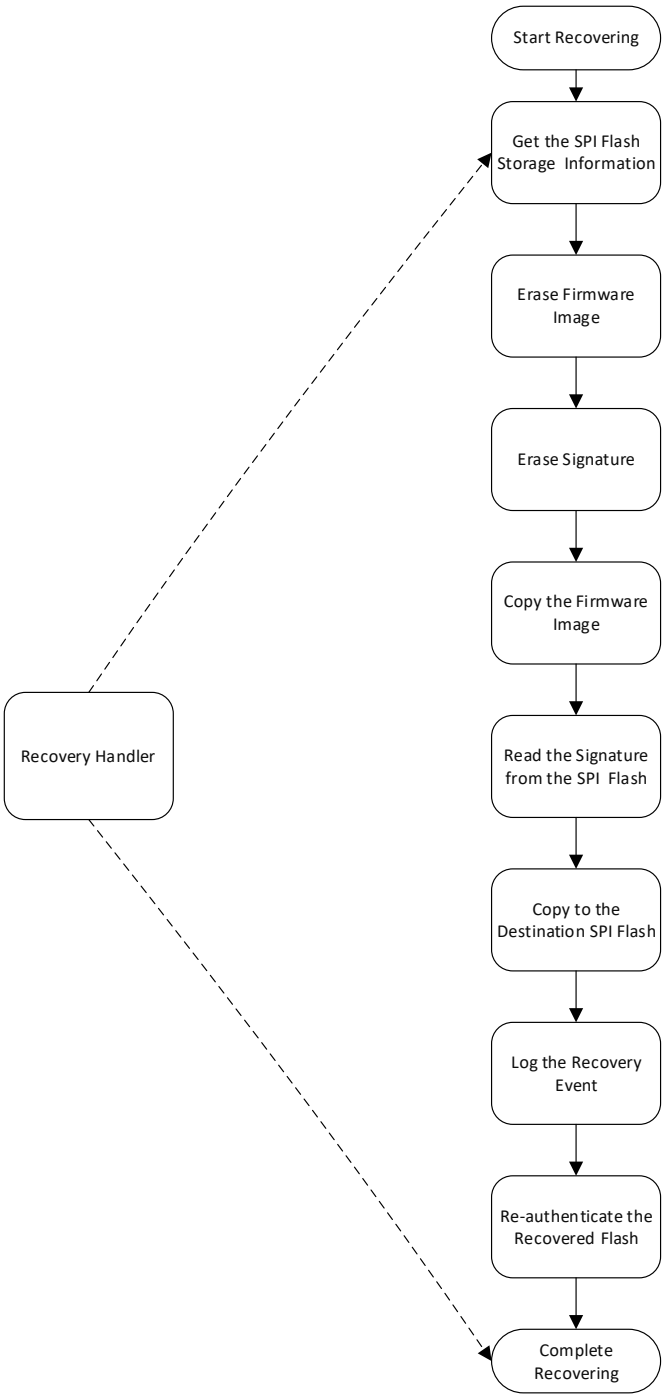

Figure 3.10. PFR Recovery Handler

3.7. Detection

The detection mechanism can detect unauthorized changes to device firmware and critical data before the firmware is executed or the data is consumed by the device. In Lattice Mach-NX PFR solution (Figure 3.11), two kinds of events can be monitored, SPI flash access and SMBus access.

Firmware and critical data can be stored on the SPI flashes of the system. Different locations of the flash can have different authority levels. The three authority levels defined in the Lattice PFR solution are called White, Grey and Black lists (Table 3.1). For each monitored spaces of the flash, one authority level is defined and configured in the 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	Read, Erase or Write operations are not permitted. The transaction is blocked when any of the Read, Erase, or Write operation is detected on the SPI bus.

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

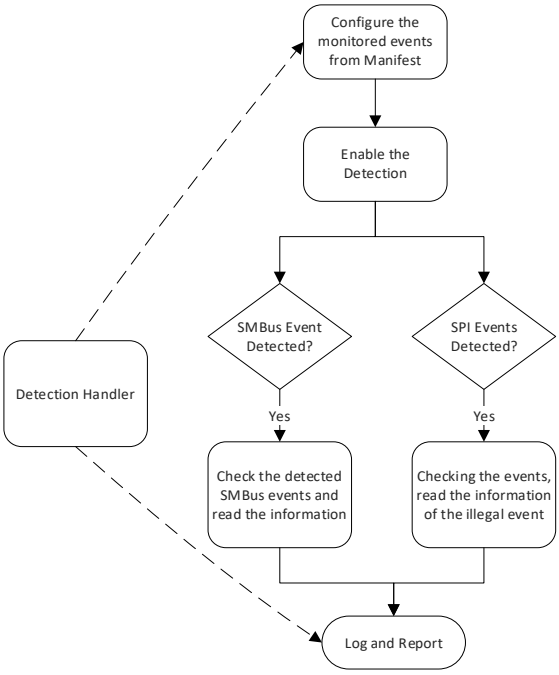

Figure 3.11. PFR Detection Handler

3.8. Logs and Reporting

Logged events are written to the UFM2 of the Lattice Mach-NX device, starting from page 1. Each page of UFM2 holds a single log entry. Byte 0 is the log index and indicates the page where the log is stored. Byte 15 is used to indicate if a log has been read (RD).

The BMC/PCH can read the log from RoT device via the SMBus 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
SMBus Exception	Log Index	0x02	SMBus 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 can use 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 the 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 via the sys_platform.h.
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 the 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 the current QSPI monitor instance.
flash_id	The value of the flash ID number.
mon_cntl	The monitor control value that is configured for the QSPI monitor.
dummy_num	The value of dummy byte number that is configured in the QSPI monitor.
flash_mon_sp	The pointer to the flash monitoring spaces that is configured for 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 the 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 the 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 via the sys_platform.h.
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 the current QSPI streamer device.
addr	The start address of the SPI flash to write to.
length	The number of data in bytes 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 the SPI flash 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 flash 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 flash 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.	

qspi_quad_read	
unsigned char qspi_quad_read(void *this_spi, 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 the SPI flash to read from.
length	The length of data for the current read.
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 data from flash. 1: Failed to read the data.
Description	
This function is used read the specified length of data from the flash in quad mode.	

qspi_quad_write	
unsigned char spi_quad_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 flash to write to.
length	The length of data for the current write.
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 writing the data to flash. 1: Failed to write the data to flash.
Description	
This function is used to write the specified length of data to the flash in quad mode.	

qspi_quad_read_crypto	
unsigned char qspi_quad_read_crypto (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 the SPI flash to read from.
length	The length of data for the current write.
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 data from flash. 1: Failed to read the data from flash.
Description	
This function is used to read the data from flash and feed into the secure enclave.	

4.3. Lattice Sentry SMBus Filter

smbus_filter_init	
unsigned char smbus_filter_init (struct smbus_filter_instance *this_smbus_filter, unsigned int base_addr);	
Parameter	**Description**
SMBus filter	The pointer to the instance of the current SMBus filter.
base_addr	Base address of the SMBus Filter 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 SMBus filter. 1: Failed to initialize the SMBus filter.
Description	
This function is used to initialize the SMBus filter module. This function is supposed to be called when the platform is being initialized. This function should be called before calling any SMBus filter related functions.	

smbus_filter_set_whitelist	
void smbus_filter_set_whitelist(struct smbus_filter_manifest *sm_filter_manifest, struct smbus_filter_instance *this_smbus_filter, unsigned char list_id)	
Parameter	**Description**
sm_filter_manifest	The pointer to the smbus configuration data in the manifest.
this_smbus_filter	The pointer to the instance of the current SMBus filter.
list_id	The list ID to be configured for the SMBus filter.
Returns	**Description**
unsigned char	0: Succeeded in configuring the SMBus filter. 1: Failed to configure the SMBus filter.
Description	
This function is used to configure the SMBus filter device by setting the number of entry and the entry data.	

smbus_filter_event_get	
unsigned char smbus_filter_event_get(struct smbus_filter_instance *this_filter, unsigned char *addr_status, unsigned int *cmd_status);	
Parameter	**Description**
this_filter	The pointer to the instance of the current SMBus filter.
addr_status	The pointer to the buffer to store the detected slave address.
cmd_status	The pointer to the buffer to store the detected command.
Returns	**Description**
unsigned char	0: Succeeded in getting the detected SMBus filter events. 1: Failed to get the detected SMBus filter events.
Description	
This function is used to get the slave address and SMBus command of the detected event.	

i2c_mon_isr	
void smbus_filter_isr(void *ctx)	
Parameter	**Description**
ctx	The pointer to the context of the SMBus filter device.
Returns	**Description**
void	—
Description	
This function is used to process SMBus filter interrupt. The function can be registered via calling pic_isr_register ().	

4.4. Lattice Sentry Secure Enclave

4.4.1. Crypto256 Interface

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. The ESB module only can start new operation in 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.	

4.4.2. Crypto384 Interface

crypto384_init	
unsigned int crypto_init(struct crypto_instance *this_crypto, unsigned int base_addr)	
Parameter	**Description**
this_crypto	The pointer to the instance of the current crypto384 device.
base_addr	Base address of the Crypto384 module, the SFB module has a pre-assigned address for each module and passed by header file to the software via Propel.
Returns	**Description**
unsigned char	0: Succeeded in initializing the Crypto384 module. 1: Failed to initialize the Crypto384 module
Description	
This function is supposed to be called when the platform is initialized. This function should be called before calling any Crypto384 related functions.	

crypto_sha384	
unsigned int crypto_sha384(struct crypto_instance *this_crypto, struct sha384_ctx* ctx, unsigned char mode)	
Parameter	**Description**
this_ crypto	The pointer to the instance of the current Crypto384 device.
ctx	The pointer to the context to do the SHA384.
mode	The SHA384 mode to do the general SHA384 or CDI HAMC SHA384.
Returns	**Description**
unsigned char	0: Succeeded in generating the digest via SHA-384 hash function. 1: Failed to generate the digest via SHA-384 hash function.
Description	
This function is used to generate a 384-bit long digest for the data specified in the context via the SHA-384 hash function.	

crypto_firmware_sha384	
unsigned int crypto_sha384(struct crypto_instance *this_crypto, struct sha384_ctx* ctx)	
Parameter	**Description**
this_ crypto	The pointer to the instance of the current Crypto384 device.
ctx	The pointer to the context to do the SHA384.
Returns	**Description**
unsigned char	0: Succeeded in generating the CDI HMAC SHA-384 digest for firmware image. 1: Failed to generate the CDI HMAC SHA-384 digest for firmware image.
Description	
This function is used to generate a 384-bit long digest for the firmware image specified in the context via the CDI HMAC SHA-384 hash function.	

crypto_hmac_sha384	
unsigned int crypto_hmac_sha384(struct crypto_instance *this_crypto, unsigned char *hmac_key, struct sha384_ctx* ctx)	
Parameter	**Description**
this_ crypto	The pointer to the instance of the current Crypto384 device.
hmac_key	The pointer to buffer holding the HMAC key.
ctx	The pointer to the context to do the SHA384.
Returns	**Description**
unsigned char	0: Succeeded in generating the MAC code via SHA-384 hash function. 1: Failed to generate the MAC code via SHA-384 hash function.
Description	
This function is used to generate a 384-bit MAC code for the data specified in the context via the SHA-384 hash function and the HMAC key provided.	

crypto_keypair_derive	
unsigned char crypto_keypair_derive(struct crypto_instance *this_crypto, struct ecc384_point * p_publicKey, unsigned char p_privateKey[NUM_ECC_DIGITS_384])	
Parameter	**Description**
this_ crypto	The pointer to the instance of the current Crypto384 device.
p_publicKey	The pointer to the structure to store the public key generated.
p_privateKey	The pointer to the array to store the private key generated.
Returns	**Description**
unsigned char	0: Succeeded in generating the ECC384 key pair. 1: Failed to generate the ECC384 key pair.
Description	
This function is used to generate a key pair of ECC384.	

crypto_pubkey_derive	
unsigned char crypto_pubkey_derive(struct crypto_instance *this_crypto, struct ecc384_point * p_publicKey, unsigned char p_privateKey[NUM_ECC_DIGITS_384]);	
Parameter	**Description**
this_crypto	The pointer to the instance of the current Crypto384 device.
p_publicKey	The pointer to the structure to store the public key generated.
p_privateKey	The pointer to the array storing the private key.
Returns	**Description**
unsigned char	0: Succeeded in generating the ECC384 public key. 1: Failed to generate the ECC384 public key.
Description	
This function is used to generate an ECC384 public key from the provided private key.	

crypto_ecdh_get	
unsigned char crypto_ecdh_get(struct crypto_instance *this_crypto, unsigned char p_secret[NUM_ECC_DIGITS_384], struct ecc384_point * p_publicKey, unsigned char p_privateKey[NUM_ECC_DIGITS_384]);	
Parameter	**Description**
this_crypto	The pointer to the instance of the current Crypto384 device.
p_secret	The pointer to the array to store the shared secret key generated.
p_publicKey	The pointer to the structure of the public key caller provides.
p_privateKey	The pointer to the array of the private key caller provides.
Returns	**Description**
unsigned char	0: Succeeded in getting the shared secret key via ECDH. 1: Failed to get the shared secret key via ECDH.
Description	
This function is used to generate a shared secret key via ECDH based on provided ECC384 public key and private key.	

crypto384_ecdsa_sign	
unsigned char crypto_ecdsa_sign(struct crypto_instance *this_crypto, unsigned int digest[], unsigned int private_key[], unsigned int nonce[], unsigned int signature[]);	
Parameter	**Description**
this_crypto	The pointer to the instance of the current Crypto384 device.
digest	The pointer to the array storing the digest.
private_key	The pointer to the array storing the private key.
nonce	The pointer to the array storing the random number.
signature	The pointer to the array used to store the signature generated.
Returns	**Description**
unsigned char	0: Succeeded in generating the signature via ECDSA. 1: Failed to generate the signature via ECDSA.
Description	
This function is used to generate the ECDSA signature for the input digest and private key.	

crypto_ecdsa_verify	
unsigned char crypto_ecdsa_verify(struct crypto_instance *this_crypto, unsigned int digest[], unsigned int pub_key[], unsigned int signature[], unsigned char *auth_pass)	
Parameter	**Description**
this_crypto	The pointer to the instance of the current Crypto384 device.
digest	The pointer to the array storing the digest.
pub_key	The pointer to the array storing the public key.
signature	The pointer to the array storing the signature.
auth_pass	The pointer to the buffer to store the ECDSA verification result.
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 verification for the input digest, signature and public key.	

crypto_ecies_encryptex	
unsigned char crypto_ecies_encryptex(struct crypto_instance *this_crypto, unsigned char p_secret[NUM_ECC_DIGITS_384], unsigned char *plain_text, unsigned char length, unsigned char *auth_tag, unsigned char *cipher_text)	
Parameter	**Description**
this_crypto	The pointer to the instance of the current Crypto384 device.
p_secret	The pointer to the array storing the shared secret key.
plain_text	The pointer to buffer storing the plain text that needs to be encrypted.
length	The length of the plan text in byte.
auth_tag	The pointer to the buffer to store the authentication tag.
cipher_text	The pointer to the buffer to store the encrypted text.
Returns	**Description**
unsigned char	0: Succeeded in doing the ECIES encryption. 1: Failed to do the ECIES encryption.
Description	
This function is used to do the ECIES encryption for the plain text.	

crypto_ecies_decryptex	
unsigned char crypto_ecies_decryptex(struct crypto_instance *this_crypto, unsigned char p_secret[NUM_ECC_DIGITS_384], unsigned char *auth_tag, unsigned char *cipher_text, unsigned char length, unsigned char cipher_status, unsigned char *plain_data)	
Parameter	**Description**
this_crypto	The pointer to the instance of the current Crypto384 device.
p_secret	The pointer to the array storing the shared secret key.
auth_tag	The pointer to the buffer storing the authentication tag.
cipher_text	The pointer to buffer storing the cipher text that needs to be decrypted.
length	The length of the plan text in byte.
cipher_status	The pointer to the buffer to store the cipher status.
plain_data	The pointer to the buffer to store the plain text decrypted.
Returns	**Description**
unsigned char	0: Succeeded in doing the ECIES decryption. 1: Failed to do the ECIES decryption.
Description	
This function is used to do the ECIES decryption for the input cipher text and authentication tag.	

crypto_ecies_decryptex	
unsigned char crypto_jtag_cntl(struct crypto_instance *this_crypto, unsigned int ctrl);	
Parameter	**Description**
this_crypto	The pointer to the instance of the current Crypto384 device.
p_secret	The pointer to the array storing the shared secret key.
auth_tag	The pointer to the buffer storing the authentication tag.
cipher_text	The pointer to buffer storing the cipher text that needs to be decrypted.
length	The length of the plan text in byte.
cipher_status	The pointer to the buffer to store the cipher status.
plain_data	The pointer to the buffer to store the plain text decrypted.
Returns	**Description**
unsigned char	0: Succeeded in doing the ECIES decryption. 1: Failed to do the ECIES decryption.
Description	
This function is used to do the ECIES decryption for the input cipher text and authentication tag.	

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 ().	

4.6. UFM Access Block (UAB)

uab_init	
unsigned char uab_init(struct uab_instance *this_uab, unsigned int base_addr)	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
base_addr	The base address of the UAB 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 UAB module. 1: Failed to initialize the UAB module.
Description	
This function is used to initialize the UAB module.	

uab_done_set	
unsigned char uab_done_set(struct uab_instance *this_uab, uint32_t cfg, uint32_t auth)	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
cfg	Specify the configuration sector. 0: CFG0 1: CFG1
auth	Specify the DONE bit or AUTH DONE bit to be set. 0: DONE 1: AUTH Done
Returns	**Description**
unsigned char	0: Succeeded in setting the DONE bit. 1: Failed to set the DONE bit.
Description	
This function is used to set the DONE or AUTH DONE bit for the specified configuration sector. After in-system-program the configuration sector, the DONE bit or AUTH Done bit needs be set, otherwise Config Engine cannot boot up the bit-stream successfully.	

uab_auth_eanble_write	
unsigned char uab_auth_enable_write(struct uab_instance *this_uab, uint32_t enable)	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
enable	The value to set the authentication enable bit 0: HMAC_SHA 1: ECDSA
Returns	**Description**
unsigned char	0: Succeeded in setting the authentication enable bit. 1: Failed to set the authentication enable bit.
Description	
This function is used to set the authentication enable bit. Once updating the public key, the authentication enable bit will also be erased and need to be set by using this function.	

uab_usercode_read	
unsigned char uab_usercode_read(struct uab_instance *this_uab, unsigned char usercode[])	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
usercode	The data buffer to store the user code read back.
Returns	**Description**
unsigned char	0: Succeeded in reading back the user code. 1: Failed to read back the user code.
Description	
This function is used to read back the user code from the UAB module.	

uab_pubkey_read	
unsigned char uab_pubkey_read(struct uab_instance *this_uab, unsigned char pubkey[64])	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
pubkey[]	The data buffer to store the public key read back from UAB module.
Returns	**Description**
unsigned char	0: Succeeded in reading back the public key. 1: Failed to read back the public key.
Description	
This function is used to read the public key from the UAB module.	

uab_pubkey_write	
unsigned char uab_pubkey_write(struct uab_instance *this_uab, unsigned char pubkey[64])	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
pubkey[]	Data buffer storing the public key to be written to UAB module.
Returns	**Description**
unsigned char	0: Succeeded in writing the public key. 1: Failed to write the public key.
Description	
This function is used to write the public key into the UAB module.	

uab_usec_read	
unsigned char uab_usec_read(struct uab_instance *this_uab, unsigned short *usec)	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
usec	Pointer to the buffer to store the USEC data read back.
Returns	**Description**
unsigned char	0: Succeeded in reading back the USEC data. 1: Failed to read back the USEC data.
Description	
This function is used to read back the USEC data from the UAB module.	

uab_usec_write	
unsigned char uab_usec_write(struct uab_instance *this_uab, unsigned short usec)	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
usec	Data buffer storing the USEC to be written to UAB module.
Returns	**Description**
unsigned char	0: Succeeded in writing the USEC. 1: Failed to write the USEC.
Description	
This function is used to write the USEC data into the UAB module.	

uab_csec_read	
unsigned char uab_csec_read(struct uab_instance *this_uab, unsigned int *csec)	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
csec	Data buffer storing the CSEC data read back from UAB module.
Returns	**Description**
unsigned char	0: Succeeded in reading back the CSEC data. 1: Failed to read back the CSEC data.
Description	
This function is used to read back the CSEC data from the UAB module.	

uab_csec_write	
unsigned char uab_csec_write(struct uab_instance *this_uab, unsigned int csec)	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
csec	Data buffer storing the CSEC to be written to UAB module.
Returns	**Description**
unsigned char	0: Succeeded in writing the CSEC data. 1: Failed to write the CSEC data.
Description	
This function is used to write the CSEC data into the UAB module.	

uab_feabit_read	
unsigned char uab_feabit_read(struct uab_instance *this_uab, unsigned int *feabit)	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
feabit	Data buffer storing the feature bits read back from UAB module.
Returns	**Description**
unsigned char	0: Succeeded in reading back the feature bits. 1: Failed to read back the feature bits.
Description	
This function is used to read back the feature bits from the UAB module.	

uab_feabit_write	
unsigned char uab_feabit_write(struct uab_instance *this_uab, unsigned int feabit)	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
feabit	Feature bits value to be written to UAB module.
Returns	**Description**
unsigned char	0: Succeeded in writing the feature bits. 1: Failed to write the feature bits.
Description	
This function is used to write the feature bits into the UAB module.	

uab_cr0_read	
unsigned char uab_cr0_read(struct uab_instance *this_uab, unsigned int *cr0_value)	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
cr0_value	Data buffer storing the control register 0 read back from UAB module.
Returns	**Description**
unsigned char	0: Succeeded in reading back the control register 0. 1: Failed to read back the control register 0.
Description	
This function is used to read back the control register 0 from the UAB module.	

uab_cr0_write	
unsigned char uab_cr0_write(struct uab_instance *this_uab, unsigned int cr0_value)	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
cr0_value	The value to be written to the Control Register 0.
Returns	**Description**
unsigned char	0: Succeeded in writing the Control Register 0. 1: Failed to write the Control Register 0.
Description	
This function is used to write the Control Register 0 into the UAB module.	

uab_udss_write	
unsigned char uab_udss_write(struct uab_instance *this_uab, unsigned int ufm, unsigned char udss_val)	
Parameter	**Description**
this_uab	The pointer to the current UAB instance.
ufm	Specify the user flash sector.
udss_val	The value to be written to the UDSS section for each sector.
Returns	**Description**
unsigned char	0: Succeeded in writing the UDSS value. 1: Failed to write the UDSS value.
Description	
This function is used to write the UDSS value for the specified user flash sector.	

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, MCTP protocol, 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 I²C 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 the 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 and then parse the information into internal data structure.	

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 I²C 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 I²C 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 I²C 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 I²C 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 I²C 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 I²C 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 I²C 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. MCTP Processing

mctp_init	
void mctp_init(struct mctp *mctp, mctp_rx_fn fn, void *data)	
Parameter	**Description**
mctp	The pointer to the current mctp component.
fn	The function pointer to the callback function which handles the vendor specific commands.
date	The pointer to the argument of the callback function.
Returns	**Description**
void	—
Description	
This function is used to Initialize MCTP structure. This function is supposed to be called when the platform is being initialized.	

mctp_register_bus	
void mctp_register_bus(struct mctp *mctp, struct mctp_binding *binding, unsigned char eid)	
Parameter	**Description**
mctp	The pointer to the current mctp component.
binding	The pointer to the bus instance that the MCTP protocol is running on.
eid	The Endpoint ID values for the MCTP local bus.
Returns	**Description**
void	—
Description	
This function is used to register a binding bus that the MCTP protocol is running on. This function is supposed to be called when the platform is being initialized.	

mctp_message_rx	
int mctp_message_rx(struct mctp_binding *binding, struct mctp_pktbuf *pkt)	
Parameter	**Description**
binding	The pointer to the instance of the binding bus.
pkt	The pointer to the MCTP packet.
Returns	**Description**
int	1: Succeeded in parsing the MCTP packet. 0: Failed to parse the MCTP packet.
Description	
This function is used to parse the received MCTP packets.	

mctp_message_tx	
int mctp_message_tx(struct mctp *mctp, unsigned char_t eid, void *msg, unsigned int msg_len)	
Parameter	**Description**
mctp	The pointer to the current MCTP component.
eid	The Endpoint ID values for the target MCTP bus.
msg	The pointer to the message that is to be sent to the binding bus.
msg_len	The number of message in bytes that is to be sent to the binding bus.
Returns	**Description**
int	Returns 0 if no error.
Description	
This function is used to send the specified length of message in the buffer to a peer device.	

mctp_pktbuf_init	
void mctp_pktbuf_init(struct mctp_binding *binding, struct mctp_pktbuf *buf, unsigned int len)	
Parameter	**Description**
binding	The pointer to the instance of the binding bus.
buf	The pointer to the MCTP packet.
len	The length of the data in the packet buffer.
Returns	**Description**
void	—
Description	
This function is used to Initialize the mctp packet with the specified length.	

mctp_pktbuf_hdr	
struct mctp_hdr *mctp_pktbuf_hdr(struct mctp_pktbuf *pkt)	
Parameter	**Description**
pkt	The pointer to the MCTP packet.
Returns	**Description**
struct mctp_hdr *	Return the address of the packet header.
Description	
This function is used to get the address of the packet header.	

mctp_pktbuf_size	
unsigned char mctp_pktbuf_size(struct mctp_pktbuf *pkt)	
Parameter	**Description**
pkt	The pointer to the mctp packet.
Returns	**Description**
unsigned char	Returns the value of the size of packet buff.
Description	
This function is used to get the size of packet buff.	

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.	

cfg_isp	
void cfg_isp(struct st_pfr_instance *pfr_inst, unsigned int fromAddr, unsigned char is_signed)	
Parameter	**Description**
pfr_inst	The pointer to the current PFR instance.
fromAddr	The flash address where firmware can load the Jedec file and download into the CFG.
is_signed	1: The Jedec file is signed. 0: The Jedec file is not signed.
Returns	**Description**
void	—
Description	
This function is used to load the Jedec file from the specified flash address and download the Jedec file into the CFG space and set the done bit and authentication done bit accordingly.	

fw_authdone_set	
int fw_authdone_set(struct st_pfr_instance *pfr_inst, unsigned int start_address)	
Parameter	**Description**
pfr_inst	The pointer to the current PFR instance
start_address	The flash address where the new firmware image is located
Returns	**Description**
int	0: Succeeded in setting the done-bit for the specified firmware image. –1: Failed to set the done-bit for the firmware image.
Description	
This function is used to set the done-bit for the new firmware image. Otherwise, the system cannot boot up successfully with the new firmware image.	

ufm3_update	
unsigned char ufm3_update(struct uab_instance *uab_inst, unsigned int start_address)	
Parameter	**Description**
pfr_inst	The pointer to the current PFR instance.
start_address	The flash address where the new ufm3 data is located.
Returns	**Description**
unsigned int	0: Succeeded in updating the data for ufm3. 1: Failed to update the ufm3 data.
Description	
This function is used to update the ufm3 data into ufm2. And Mach-NX device authenticates the data and makes update into UFM3 when booting up.	

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 in clearing the log. No other return value.
Description	
This function is used to write one slot 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.

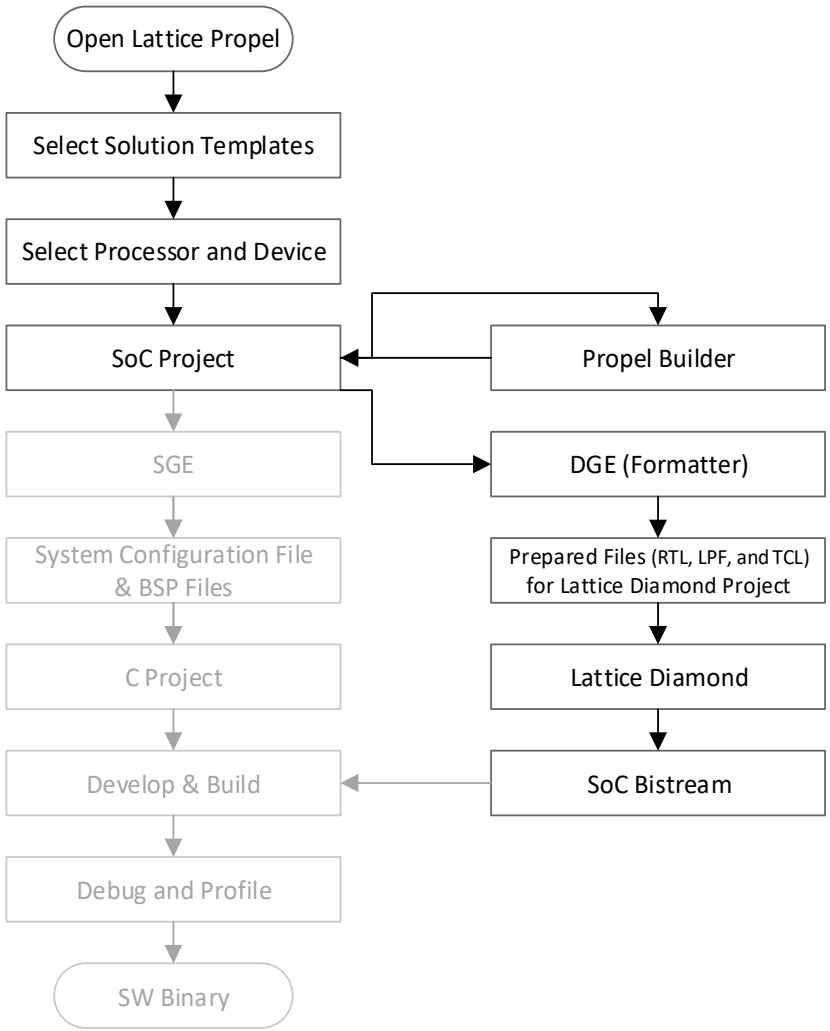

Figure 6.1. Lattice Propel Template Flow

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 Customer PLD Interface 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.

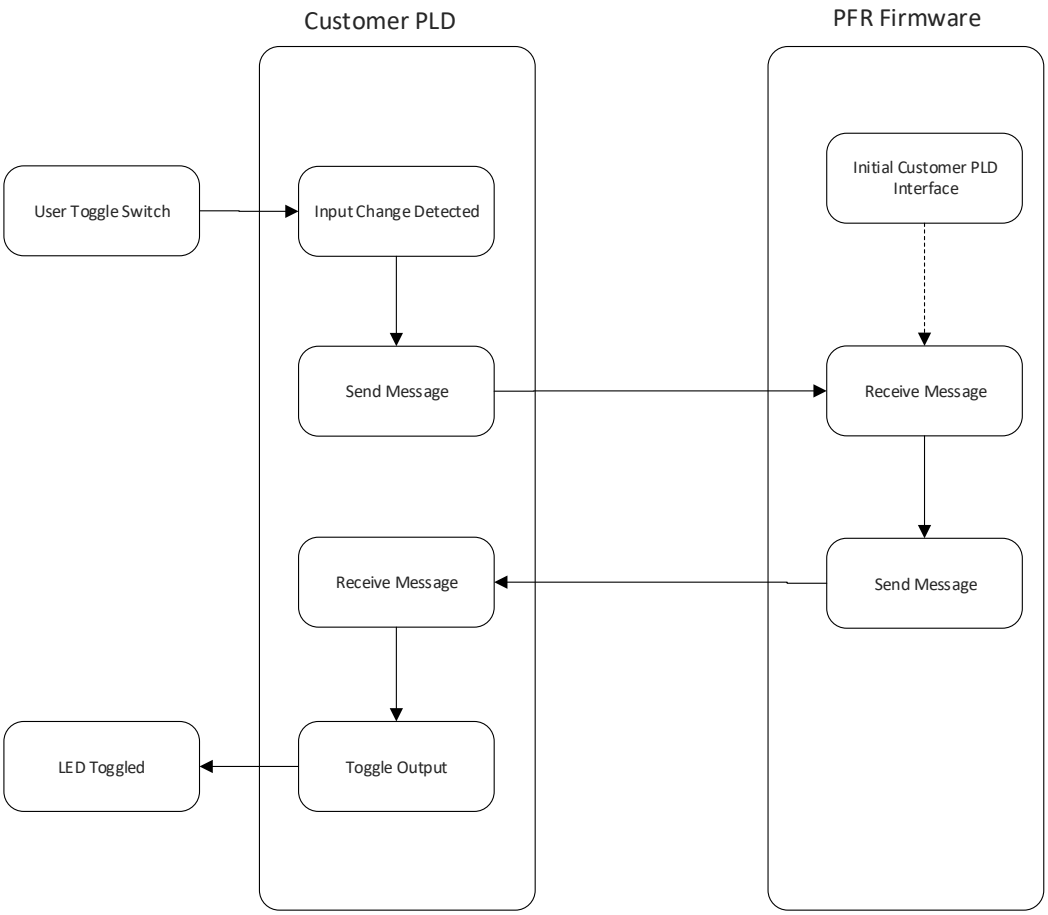

Figure 6.2. Customer PLD Workflow

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

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. Lattice Sentry Demo GUI Tool

The Lattice Sentry Demo GUI is a tool which can communicate between a PC with Windows platform and the Mach-NX device through UART to I^2C bridge on the Lattice Sentry Demo Board for Mach-NX part. This tool also provides SPI access to verify the monitoring and protection of the SPI Flash. The Lattice Sentry Demo GUI is integrated in Lattice Propel platform.

To use Lattice Sentry Demo GUI Tool:

1. Connect mini-USB cable from PC to the mini-USB connector J11 of the Lattice Sentry Demo Board for Mach-NX.
2. From your PC desktop, invoke Lattice Propel. Choose LatticeTools > Lattice Sentry Tools for Mach-NX > Lattice Sentry Demo GUI to invoke Lattice Sentry Demo Tool. See Figure 7.1.

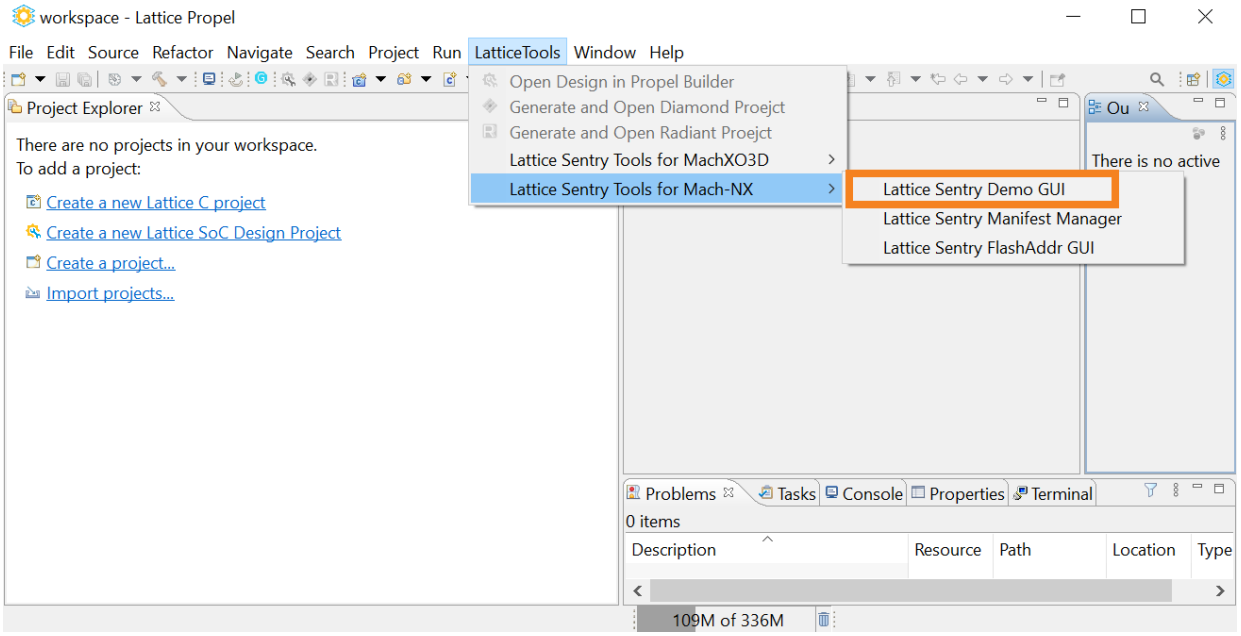

Figure 7.1. Launch Lattice Sentry Demo GUI Tool

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 Lattice Sentry Demo Board for Mach-NX. 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.

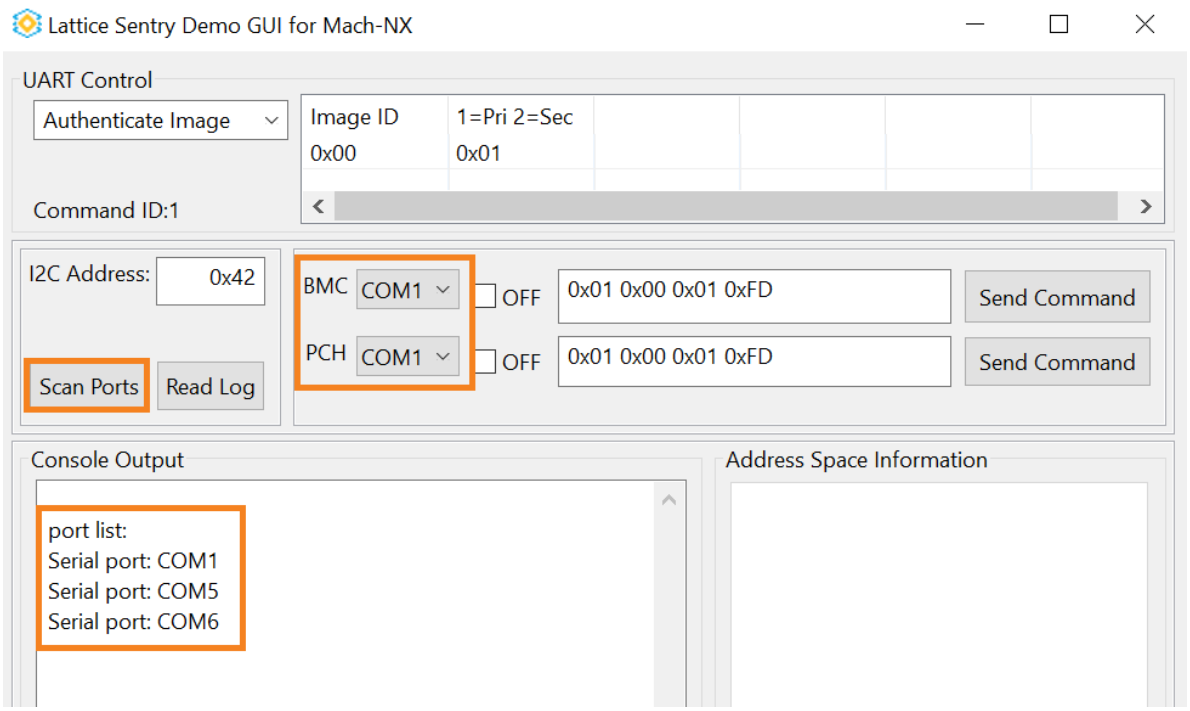

Figure 7.2 COM Port Scan of the Lattice Sentry Demo GUI Tool

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.

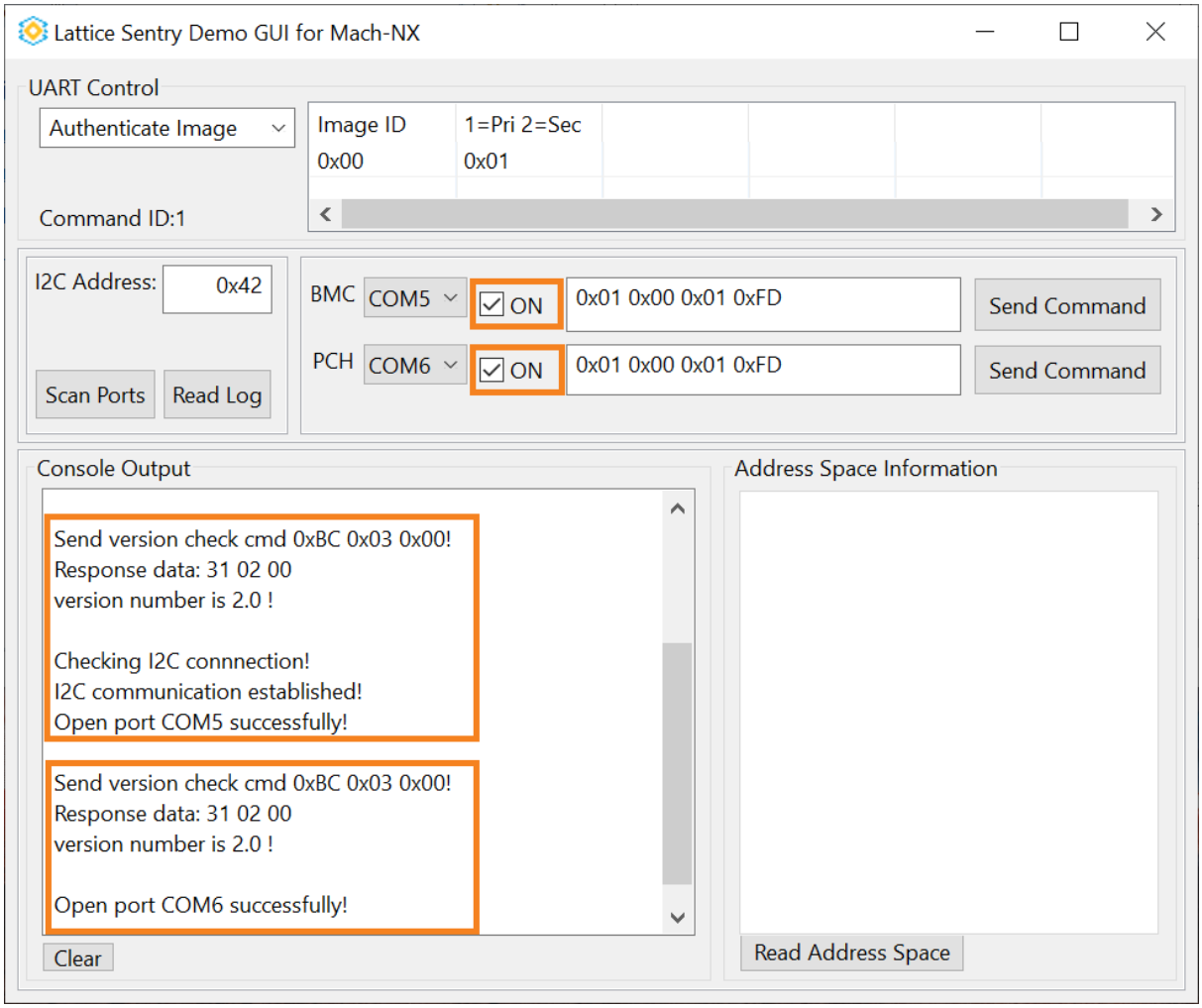

Figure 7.3 Enable Lattice Sentry Demo GUI Tool

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.

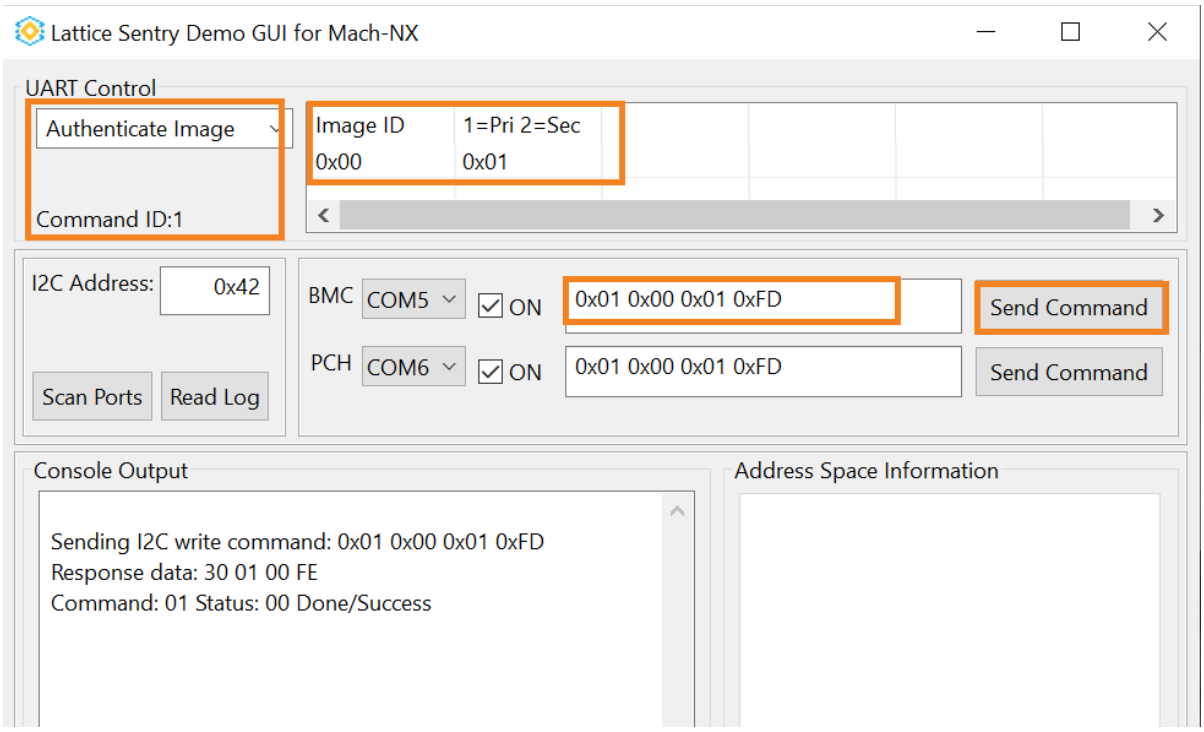

Figure 7.4. Send Command of Lattice Sentry Demo GUI Tool

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.

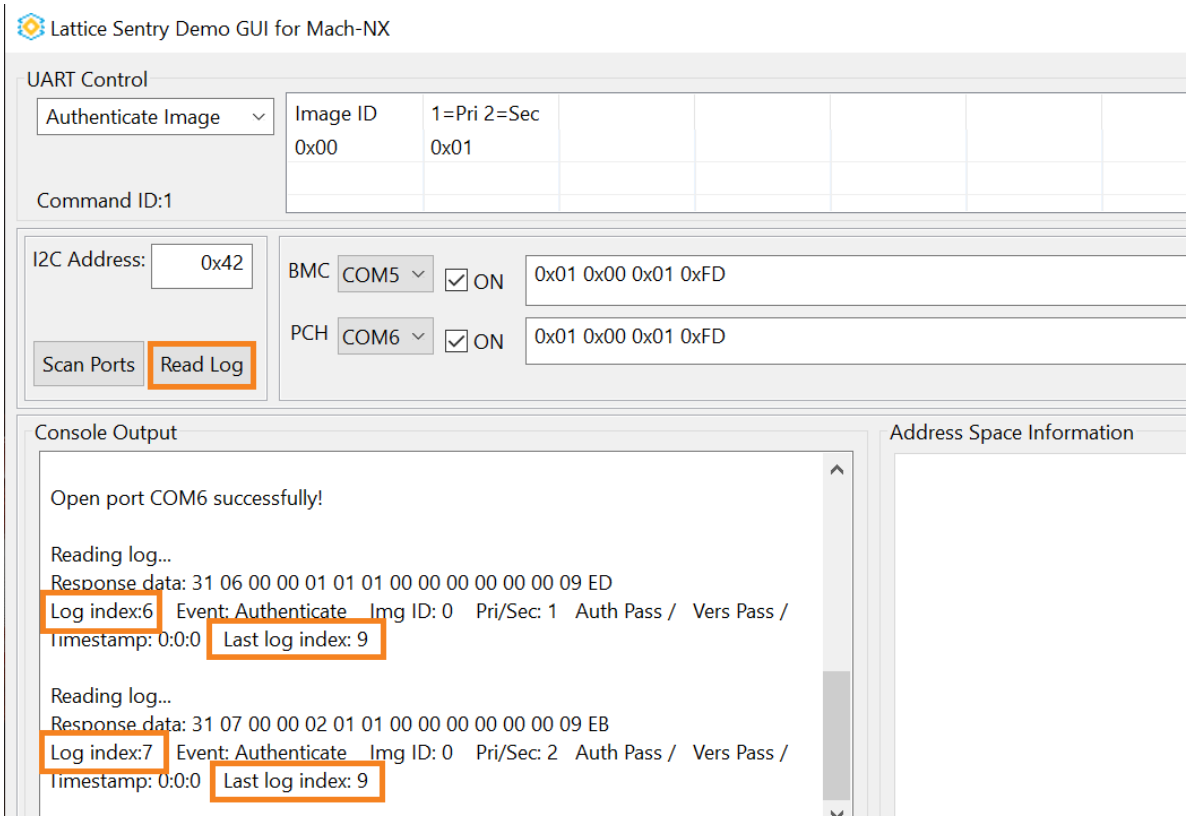

Figure 7.5 Logging of Lattice Sentry Demo GUI Tool

10. Clicking *Read Address Space* retrieves the information of the manifest from UFM0 in Mach-NX 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.

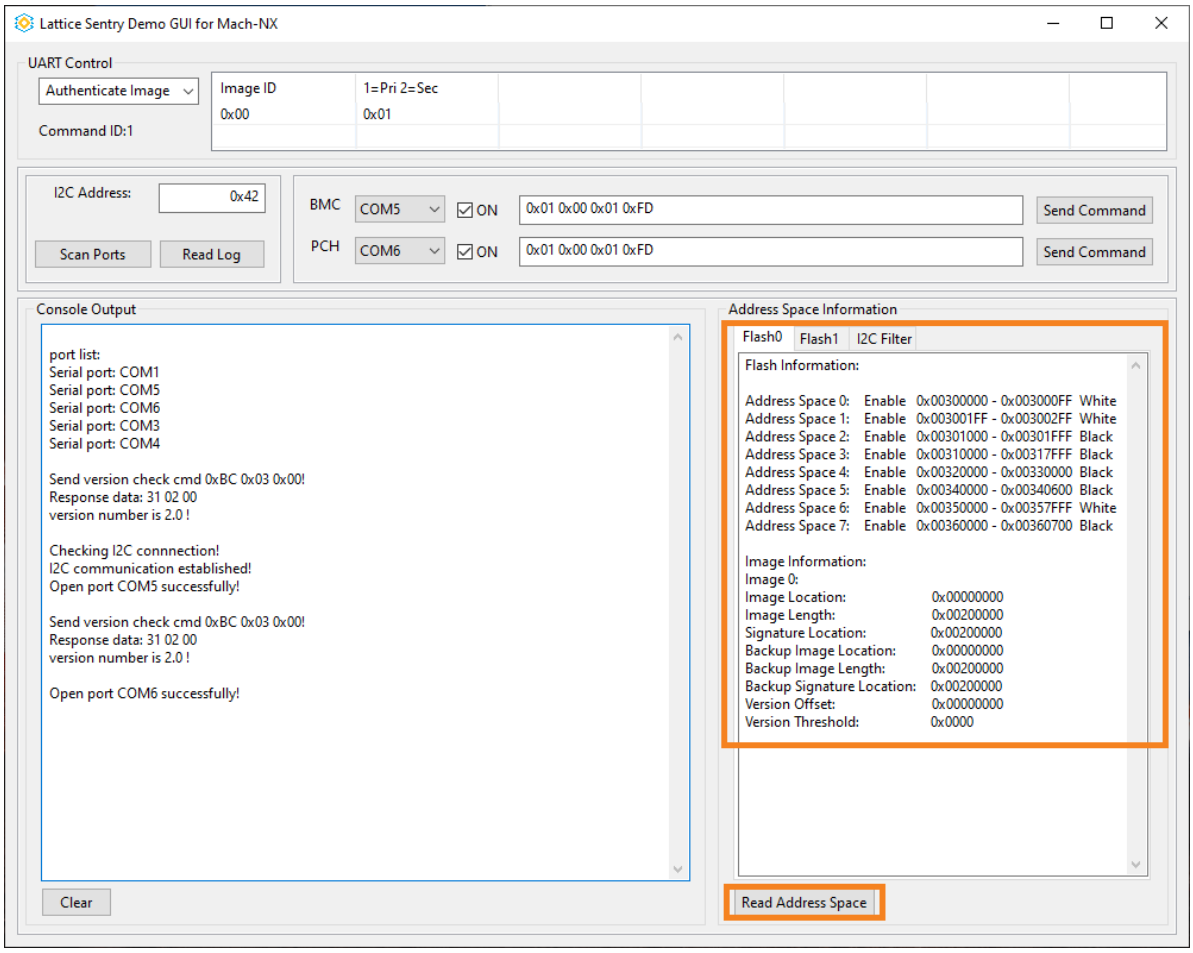

Figure 7.6 Read Address Space of Lattice Sentry Demo GUI Tool

For the detail definition of the commands, refer to the Write Commands and Read Commands sections of the Mach-NX Platform Firmware Resiliency Out-of-Band I^2C 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 Sentry Demo GUI 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.

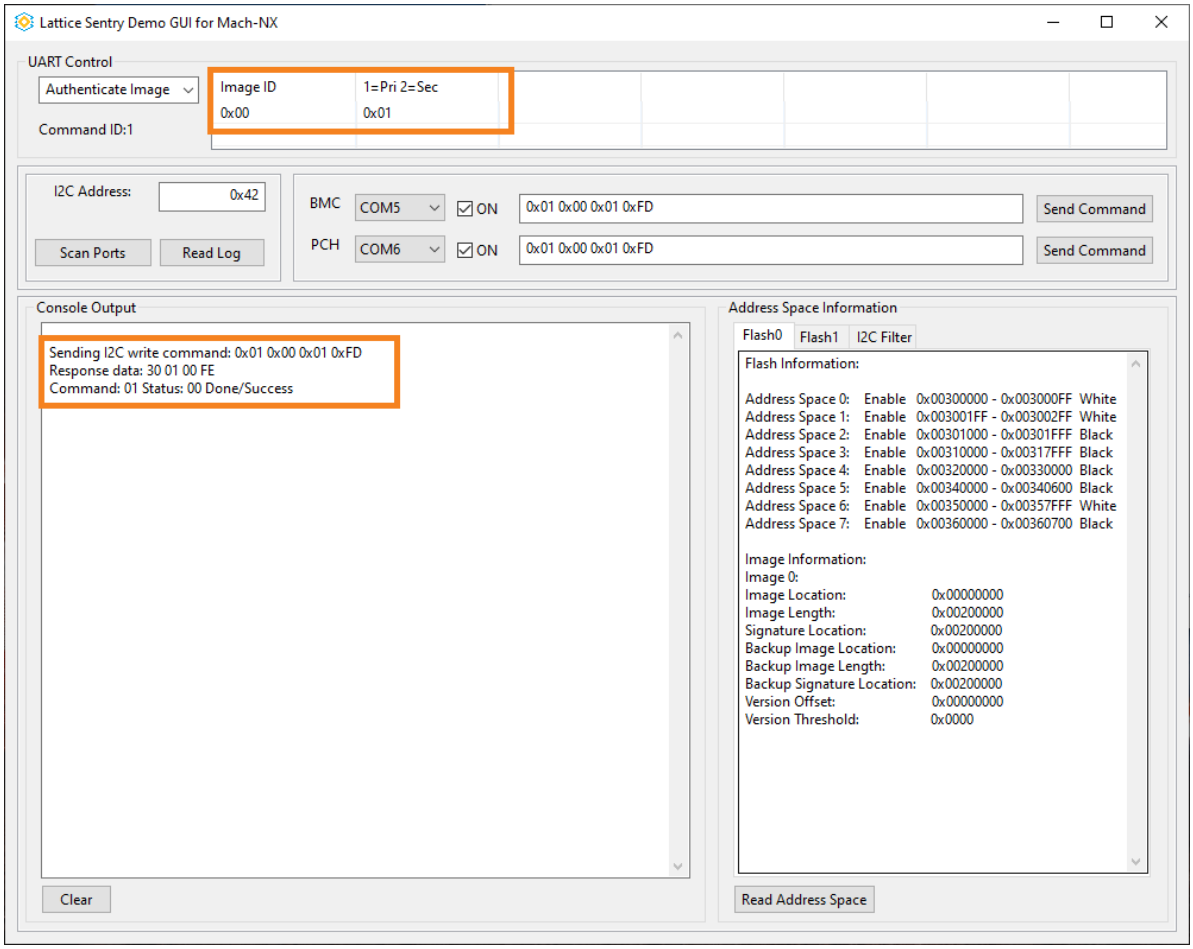

Figure 7.7. BMC Image Authentication for Flash 0

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 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.

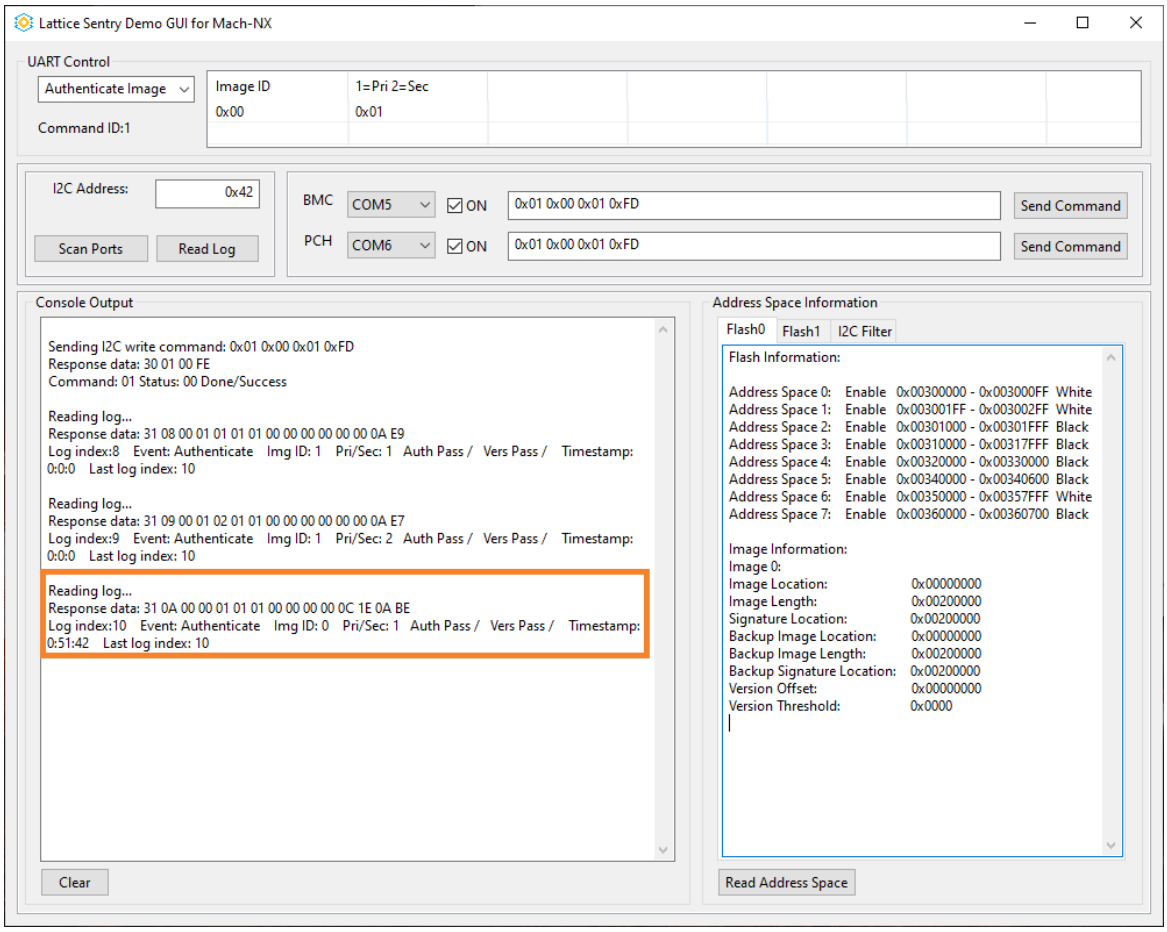

Figure 7.8. Get Logs for Image Authentications

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.

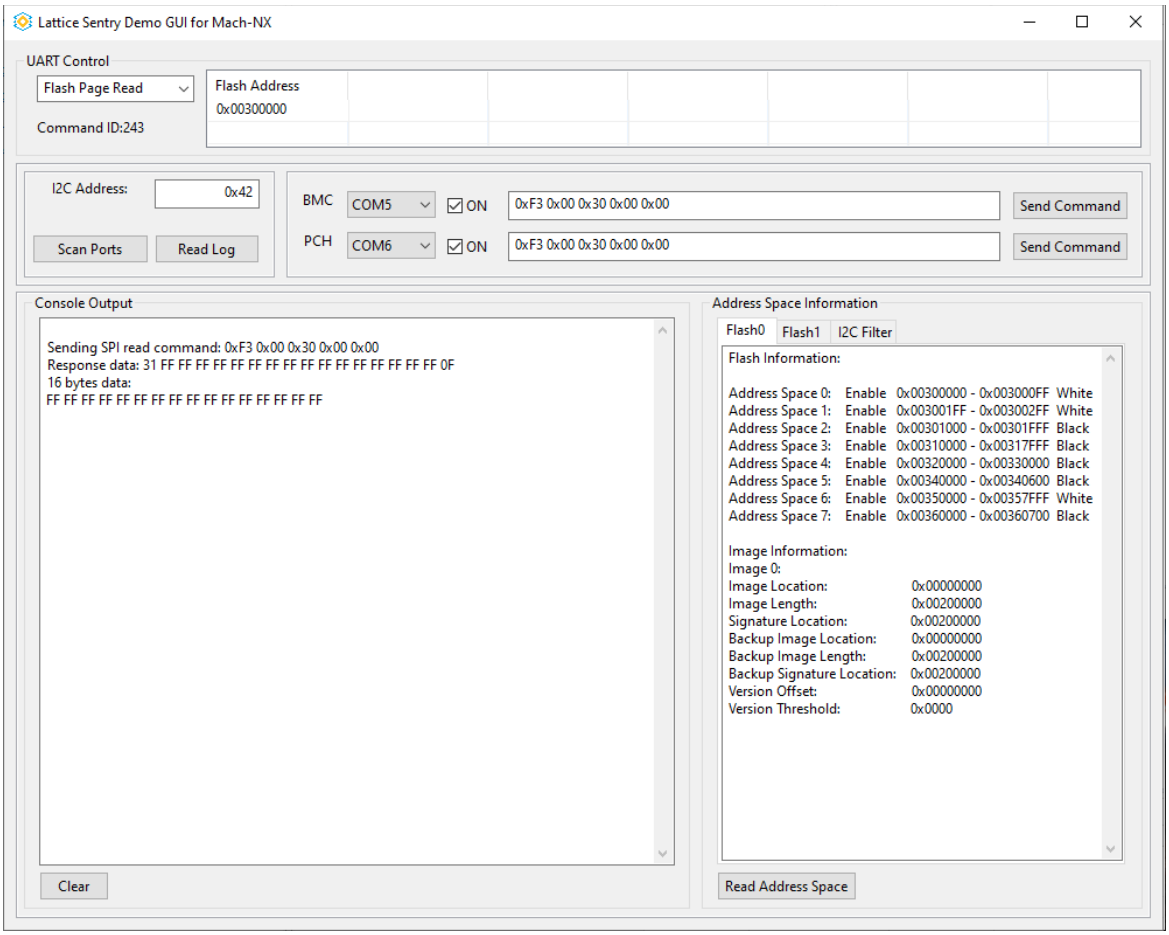

Figure 7.9. Initial Value of 0x00300000~0x0030000F

Disable SPI Filter (0x16 0x00 0x00 0xE9) – to disable all commands for filtering on BMC SPI port.
Flash Sector Erase (0xF0 0x00 0x30 0x00 0x00 0x01) – to erase the sector started from 0x00300000 in Flash0.
Enable SPI Filter (0x16 0x00 0x01 0xE8) – to enable all commands for filtering on BMC SPI port.
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.

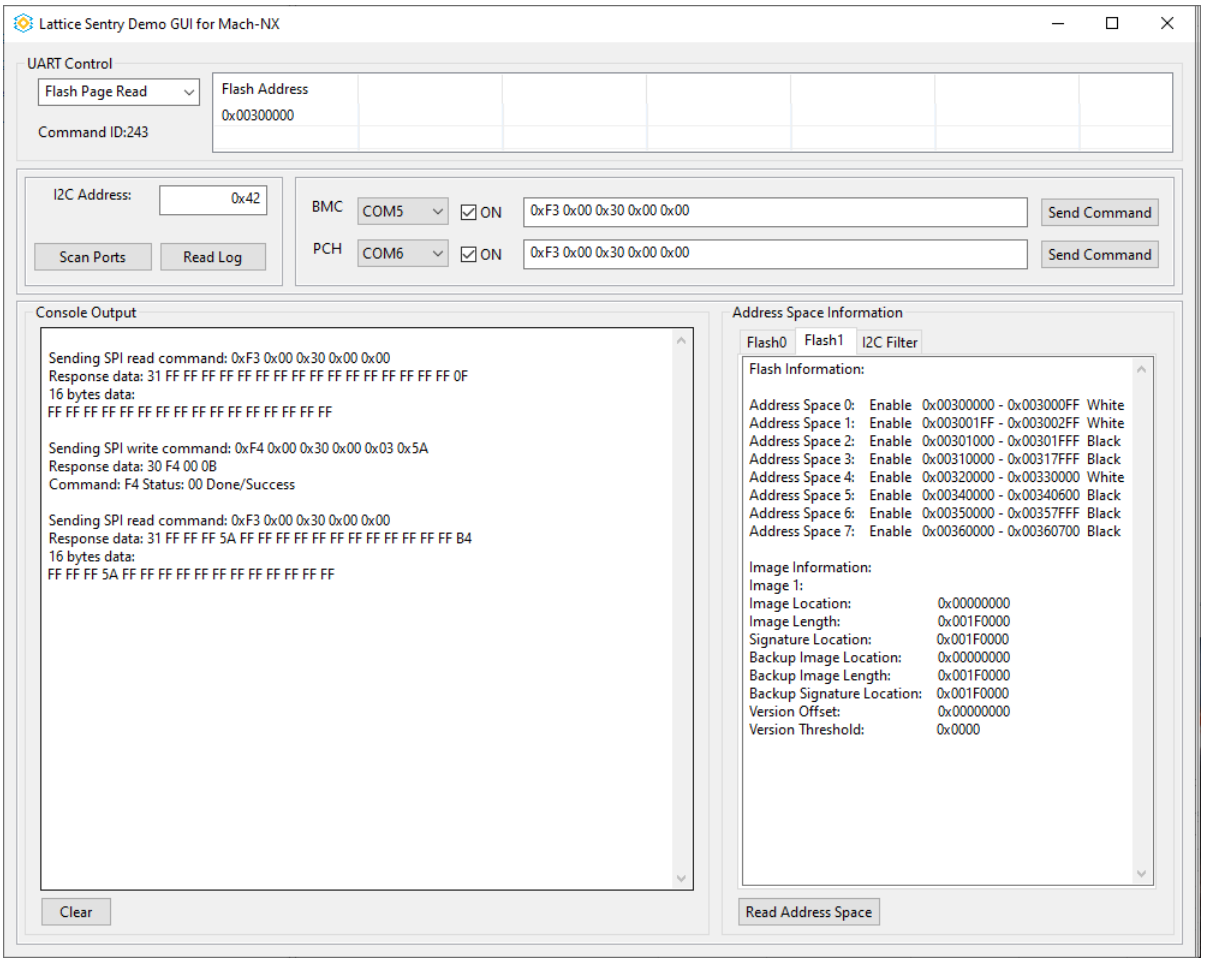

Figure 7.10. Value of 0x00300000~0x0030000F after Write

7.2.3.2. Illegal Operation (operate on Black Space)

Reading 16 bytes started from 0x00310000 in Flash0, program a value (0xAA) to 0x00310003, and read back the bytes again. Follow steps below:

Flash Page Read (0xF3 0x00 0x31 0x00 0x00) – to read 16 Bytes started from 0x00310000 in Flash0

Flash Byte Write (0xF4 0x00 0x31 0x00 0x03 0xAA) – to write a value (0xAA) to 0x00310003 in Flash0

Flash Page Read (0xF3 0x00 0x31 0x00 0x00) – to read 16 Bytes started from 0x00310000 in Flash0.

After running above steps, Figure 7.11 shows that the read address 0x00310000 is blocked and the return values are all 0x00. 0x00310003 is Black Address Space 3 (0x00310000~0x00317FFF), so it cannot be programmed.

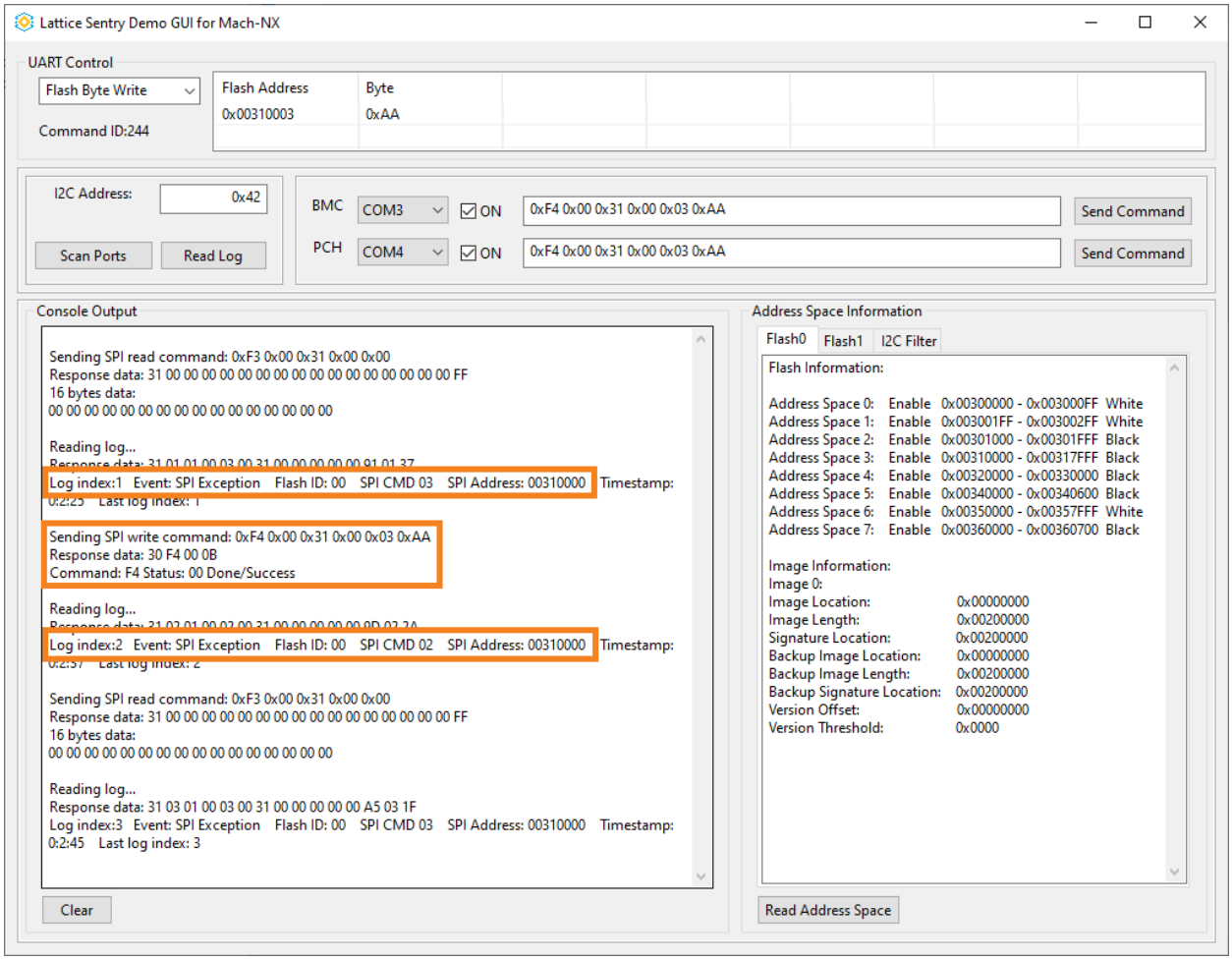

Figure 7.11. Value of 0x00310000~0x0031000F after Write

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

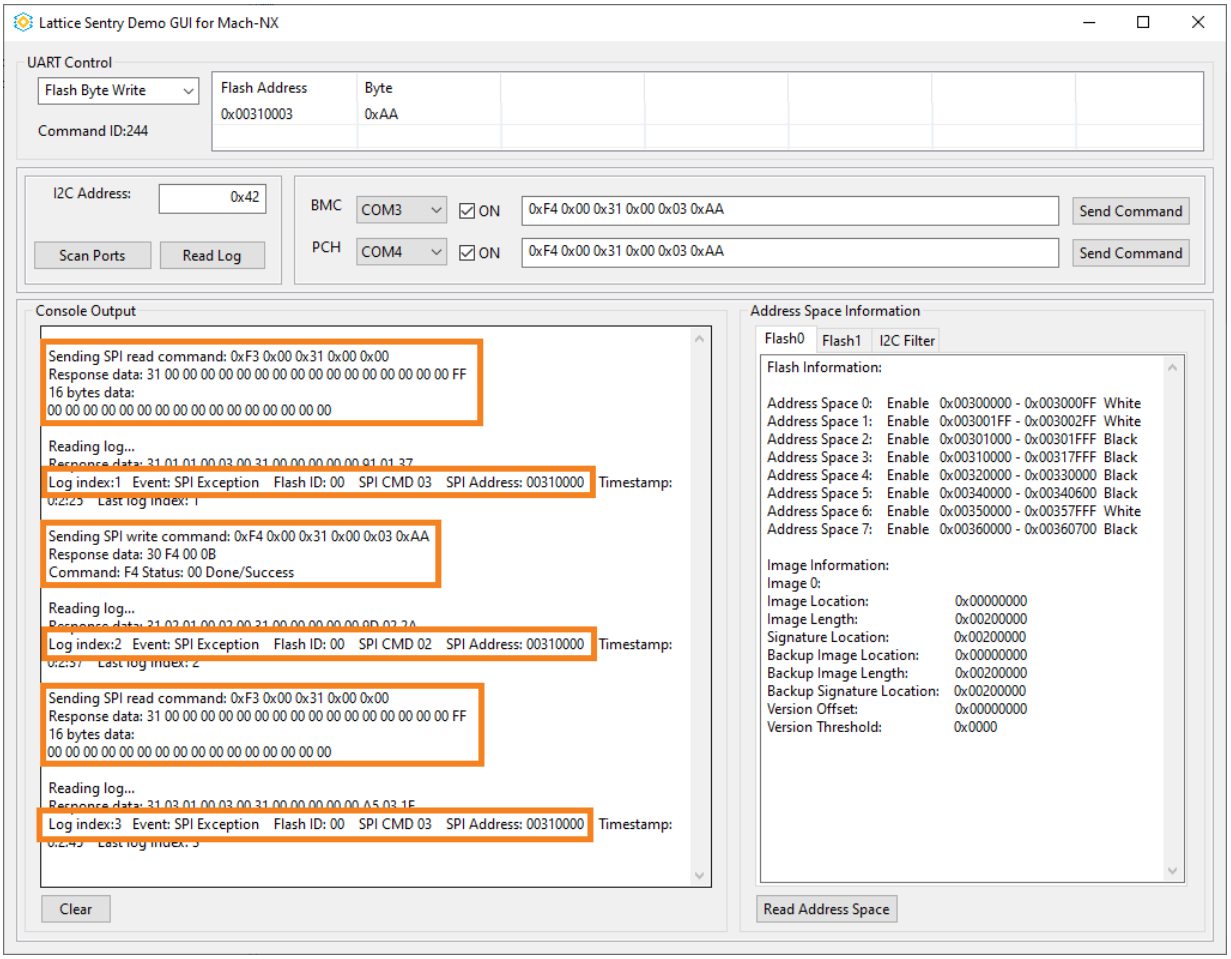

Figure 7.12. Logs of Illegal Operation

7.2.4. Recovery

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

7.2.4.1. Manual Image Corruption

Disable all commands filtering for BMC. Then erase the sector starting from 0x00100000 in Flash0 to corrupt Primary image in Flash0. Authenticate Primary image after corrupting the Primary image. Authentication should fail, as Figure 7.13 shows. Follow steps below:
Authenticate Image (0x01 0x00 0x01 0xFD) – to authenticate Primary image in Flash0.
Disable 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.

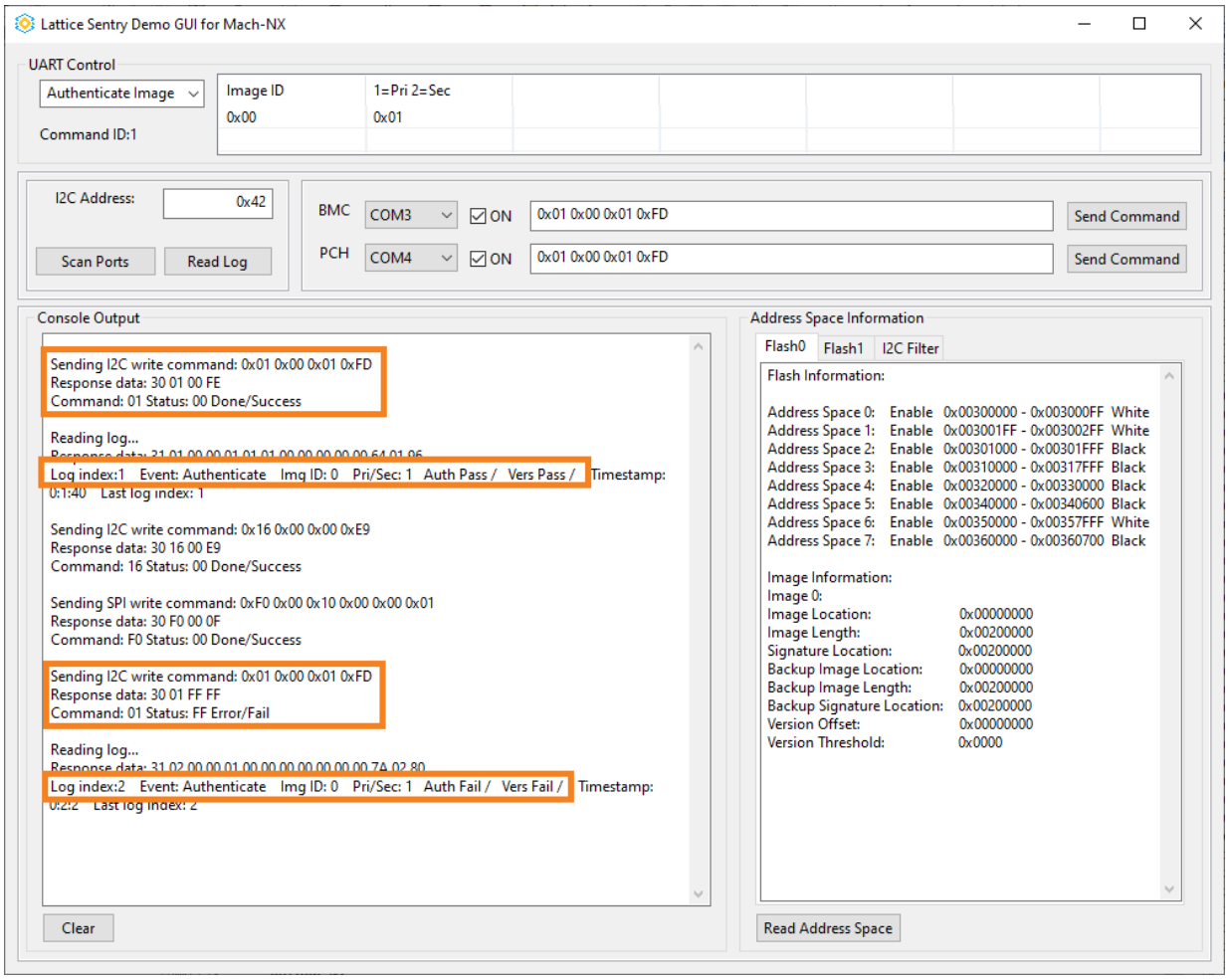

Figure 7.13. Authentication Failed with Corrupted Image

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.

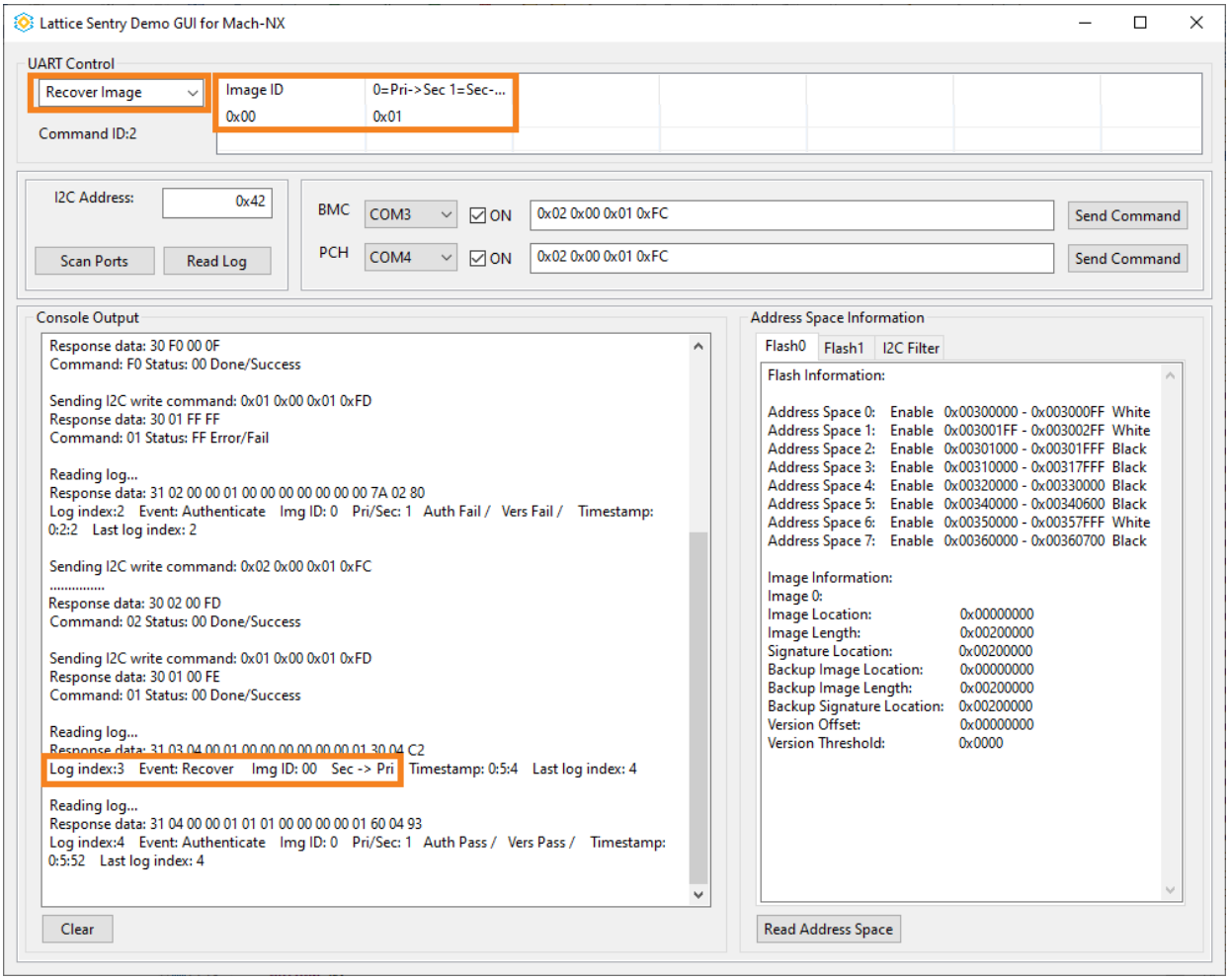

Figure 7.14. Authenticate Primary Image after Recovery Done

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.

References

- Lattice Sentry PLD Interface IP Core (FPGA-IPUG-02106)
- SFB Interface IP Core (FPGA-IPUG-02151)
- Lattice Sentry SMBus Mailbox IP Core - Lattice Propel Builder (FPGA-IPUG-02165)
- Lattice Sentry I2C Filter IP Core - Lattice Propel Builder (FPGA-IPUG-02166)

Revision History

Revision 1.1, March 2022

Section	Change Summary
PFR IP API Reference	In the UFM Access Block (UAB) section: • newly added uab_done_set, uab_auth_eanble_write, uab_usercode_read, uab_pubkey_read, uab_pubkey_write, uab_usec_read, uab_usec_write, uab_csec_read, uab_csec_write, uab_feabit_read, uab_feabit_write, uab_cr0_read, uab_cr0_write, and uab_udss_write blocks.

Revision 1.0, January 2022

Section	Change Summary
All	Initial general-purpose production release.

www.latticesemi.com