PFE Health Monitoring User Manual

Rev. 1.4.0 — 29 June 2023 User manual

Revision History

Revision history

Revision number	Date	Description
1.4.0	28.07.2023	The Error Injection details added.
1.3.0	2023-06-29	The Driver reset in multi-instance scenarios limitation updated as drivers provide mitigation in form of the graceful/ungraceful reset support.
1.2.0	2023-05-15	Added reporting to DEM in PFE MCAL driver.
1.1.0	2023-04-06	Added events multiplicities. The Out-of-buffers event made informal only.
1.0.0	2023-01-23	Initial version.

1 Acronyms and Definitions

Table 1. Acronyms and Definitions

Term	Definition
3-tuple	Set of source and destination IP addresses, and L4 protocol number.
5-tuple	Set of source and destination IP addresses, L4 protocol number, and L4 source and destination ports.
API	Application Programming Interface.
CLASS	Primary processing engine of the PFE. It uses PFE Firmware.
Classifier	See CLASS.
Classification Algorithm	A method how PFE firmware processes Ethernet frames. Refers to implemented algorithms, for example L2 Bridge, L3 Router, or L2L3 VLAN Bridge.
BMU	Buffer Management Unit.
DSCP/DS	Differentiated Services Code Point/Differentiated Services field in IPv4 or IPv6 header as defined by RFC 2474.
ECC	Error Correction Code.
Egress Timestamp Report	A special metadata generated by PFE and delivered to the requesting host. The report carries the egress timestamp of a previously transmitted Ethernet frame.
EMAC	Ethernet Media Access Controller.
ERM	Error Reporting Module.
FCCU	Fault Collection and Control Unit.
FCI	Fast Control Interface. Software library that provides API for configuration and management of PFE Firmware and PFE Hardware. For details refer to the *FCI API Reference* [1].
FW	Firmware. A software component running as part of a hardware subsystem.
HIF	Host Interface. A Physical Interface passing frames between PFE and a Host. The HIF consists of multiple channels (HIF0, HIF1, and so on). Each channel can interface a different Host.
HM	Health Monitor or Health Monitoring. Ability of the PFE HW and SW to recognize malfunctions and recover from the faulty state.
Host	Execution environment within the SoC where the PFE Driver is running.
HSE	Hardware Security Engine. A subsystem the PFE interacts with, but it is not part of the PFE.
HW	Hardware.
IHC	Inter-Host Communication. A virtual communication channel in PFE allowing communication between PFE Driver instances.
IP	Internet Protocol. Generic abbreviation encompassing IPv4 and IPv6.
IPSec	IP security protocol. Also refers to a premium PFE feature *IPsec Offload*. The feature is provided by the premium version of the PFE Firmware.
IPv4	IP version 4.
IPv6	IP version 6.
Kernel, Linux Kernel	https://docs.kernel.org/
QOS	Quality of Service
L2, L3, L4	Layers of the ISO/OSI model.
LLCE	Low Latency Communication Engine. A subsystem the PFE interacts with, but it is not part of the PFE.
MCAL	Microcontroller Abstraction Layer as defined by AUTOSAR.
PCP	Priority Code Point field in vlan tag as defined by IEEE 802.1Q.

Table 1. Acronyms and Definitions...*continued*

Term	Definition
PE	Processing Engine. A processor core within the PFE that runs the PFE Firmware.
PFE	Packet Forwarding Engine. Hardware-based accelerator for classification and forwarding of Ethernet traffic between selected interfaces. It requires firmware for its operation.
PFE Driver	The software that is handling the PFE and interfaces the rest of software stack. It does not run in the PFE but on the host CPU.
PFE Firmware	A necessary software component that runs inside the PFE. It is provided by NXP in a binary form. For details refer to the *PFE Firmware User Manual* [3].
PHY	Ethernet physical layer transceiver.
Physical Interface	A hardware interface of PFE. Generic term that includes all PFE EMACs (for connection to Ethernet networks) all PFE HIF channels (for connection to Hosts).
Premium Feature	The premium feature is not part of the standard delivery and it must be ordered separately from NXP. The PFE Firmware containing the premium feature is called Premium PFE Firmware.
PTP	Precision Time Protocol defined in the IEEE 1588 standard.
QNX	A commercial real-time operating system.
S32G	S32G SoC that contains the PFE.
SoC	System on a Chip. In the context of the PFE refers to the S32G device.
SW	Software.
TTL	Time To Live field of IPv4 header or Hop Limit field IPv6 header.
UTIL	Auxiliary Processing Engine of the PFE. Its firmware is needed only for certain features.
VLAN	Virtual LAN as defined by IEEE 802.1Q.

2 References

[1] *FCI API Reference*, available within the PFE Driver Release Package from FlexNet (nxp.com). The package can be obtained via the web portal under the section **Automotive SW - S32G - PFE Driver + Standard Firmware**.

[2] *S32G Reference Manual*, available via NXP representatives.

[3] *S32G PFE Firmware User Manual*, available as part of the S32G2 PFE Firmware release package from FlexNet (nxp.com). The package can be obtained via the web portal under the section **Automotive SW - S32G - PFE Driver + Standard Firmware**.

[4] *User Manual for S32 MCU Driver*, available within the MCU module delivered in the S32 Real-Time Drivers release that can be obtained from FlexNet (nxp.com).

3 Preface

This document describes purpose, mechanisms, and API related to the PFE Health Monitoring.

4 Introduction

The term “PFE Health Monitoring” refers to the ability of PFE to recognize and eliminate malfunctions. It covers all aspects of PFE operation in normal and faulty scenarios. Normal scenario refers to a nominal state when the PFE HW and SW works according to

specifications. The faulty scenario then describes situation when the PFE HW, or SW, or both experience conditions that are preventing reliable operation.

The faulty scenarios impact subsystems and applications leveraging services provided by the PFE. To keep the services available and reliable, any malfunction must be detected and potentially corrected. Following chapters aim to provide an insight into how such goal can be achieved.

5 Architecture

5.1 Health Monitoring Principle

The PFE HW and SW implement mechanisms for detection and reporting of various events. These events carry information about the current PFE state. PFE can be in normal or faulty state.

In normal state, PFE works according to expectations. All HW and SW error detection mechanisms are enabled and the PFE HW and SW is continuously monitored.

If an event arises, the PFE enters a faulty state. The faulty state is reported to a higher application layer, which can then execute an informative or recovery action. Recovery action returns PFE to its normal state.

5.2 Reference Scenario

Following figure depicts a reference deployment regarding the PFE Health Monitoring. In the scenario, multiple PFE Driver instances are reporting PFE health status via FCI API. At the same time, the FCCU is reporting additional PFE health status via interrupts. A Control Application is collecting events from all relevant event sources. If a malfunction is detected, the control application executes a Recovery Action.

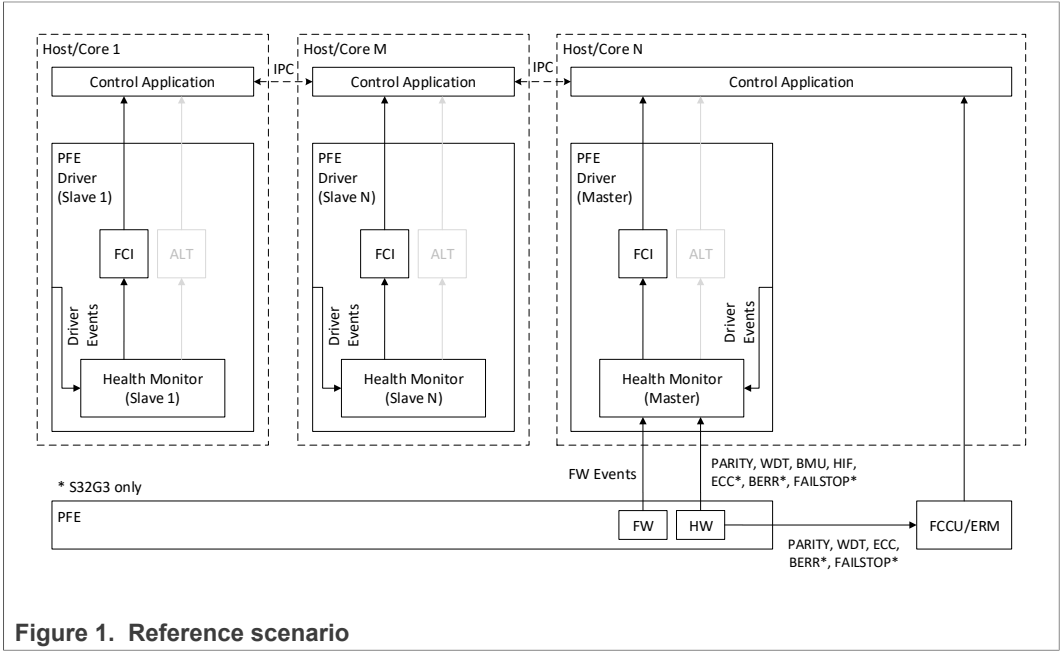

Figure 1. Reference scenario

Note: *Beside FCI, the Control Application can receive the events via an alternative (ALT) API. See the Events API chapter.*

5.3 Actors

The PFE Health Monitoring involves multiple actors:

- PFE HW
- PFE FW
- PFE SW
  - PFE Driver (master)
  - PFE Driver (slave)
  - PFE Driver (MINIHIF, attached)
  - PFE Driver (MINIHIF, detached)
- FCCU and ERM
- Control Application
- Integrator

5.3.1 PFE HW

The PFE HW refers to the PFE HW subsystem integrated within the S32G SoC. It includes PFE together with its EMACs. The PFE HW can detect and report following event classes:

- Parity events
- Watchdog events
- HW block events (HIF, BMU)
- ECC events
- Bus error events (S32G3 only)
- Fail-stop events (S32G3 only)

5.3.2 PFE FW

The PFE firmware is a software component that is executed inside the PFE by dedicated processing engines (PEs). The firmware implements software-defined features of the PFE. It is provided by NXP in a binary form.

The PFE firmware detects and reports following event classes:

- PE stall events
- PE exception events
- Firmware runtime error events
- Fail-stop events (S32G3 only)

5.3.3 PFE SW

PFE software represents the PFE driver component. There are four implementations of the PFE driver component: Linux Driver, MCAL Driver, QNX Driver, and the LLCE aka MINIHIF Driver.

Multiple instances of a PFE driver can simultaneously run on a single SoC. In such scenario, the master-slave principle is applied. Each PFE driver implementation[1] can be instantiated as a master or slave variant. The deployment within the SoC allows one master instance and arbitrary number of slave instances.

1 Except for MINIHIF detached, which is always treated as a slave variant.

The PFE SW enables, detects, collects, and reports all types of PFE Health-related events. The PFE SW integrates the *Health Monitor Module* in form of a software subcomponent realizing the required tasks. Besides HW events, the PFE SW also detects and reports various SW runtime events. The SW runtime events can be characterized as all events originating in the PFE SW that potentially affect the PFE or system operation in a negative way.

5.3.3.1 PFE Driver (master)

This is the PFE master driver variant. Ensures detection and reporting of following event classes:

- HW
  - Parity events
  - Watchdog events
  - HW block events (HIF, BMU)
  - ECC events (S32G3 only)
  - Bus error events (S32G3 only)
  - Fail-stop events (S32G3 only)
  - Firmware events
- SW
  - Driver runtime events

5.3.3.2 PFE Driver (slave)

This is the PFE slave driver variant. Detects and reports following event classes:

- HW
  - None
- SW
  - Driver runtime events

5.3.3.3 PFE Driver (MINIHIF, attached)

A PFE driver instance acting as an owner of the PFE Driver (MINIHIF, detached) component. It is either PFE Driver (master) or PFE Driver (slave) instance. Detects and reports events according to the variant.

5.3.3.4 PFE Driver (MINIHIF, detached)

The detached MINIHIF component is a special, reduced variant of PFE driver. It has been adapted to run within the LLCE firmware. Requires an attached master or slave PFE driver instance managing PFE HW resources that are not accessible from this detached component. The detached part currently does not participate in PFE Health Monitoring as an event source but can be affected by recovery actions. See the Recovery Actions topic.

5.3.4 FCCU and ERM

FCCU together with ERM provide SoC-level means to detect and report certain PFE HW events. FCCU/ERM detects and report following event classes:

- HW
  - Parity events

  - Watchdog events
  - ECC events
  - Fail-stop events (S32G3 only)
  - Bus error events (S32G3 only)
- SW
  - None

5.3.5 Control Application

The control application has been defined as an independent, distributed, application-level PFE management component. The control application has access to PFE Health via APIs exported by each PFE driver instance deployed within the SoC. It also has access to the FCCU and ERM. Based on the events reported by all sources, the control application is expected to take systemwide actions leading to recovery from the faulty state.

5.3.6 Integrator

Integrator is aware of the whole system composition:

- Knows all involved PFE driver instances and variants.
- Knows hardware and software environments executing the PFE SW.
- Knows system dependencies and implications affecting the PFE Health Monitoring.

The Integrator designs the Control Application regarding the knowledge and the PFE Health Monitoring documentation.

6 Events

The PFE Health Monitoring utilizes two event detection and reporting agents: the PFE driver and the FCCU. Due to the nature of the PFE HW and the SoC some events are detected only by the PFE driver, some only by the FCCU, and some by both. Furthermore, some events are detectable only on specific SoC versions. This chapter describes all available events and event classes with respect to the SoC versions. Note that every event (or event class) that requires a recovery action has the appropriate action listed. For detailed description of recovery actions, see the Recovery Actions topic.

6.1 Events detected by PFE SW

Following table summarizes all events that can be detected by the PFE driver. Events are uniquely identified by their IDs. Sets or ranges of IDs are defined to form event classes helping to reduce the table complexity. Events are reported simultaneously via 2 means:

1. Human-readable text form. The event ID and optionally detailed description/ syndrome are printed into a standard system log file or terminal.
2. Automation-friendly API. See the Events API topic.

Every event has assigned multiplicity and a recovery action. Multiplicity indicates how many times the given event can occur during PFE Driver lifecycle. The recovery action describes required action leading to recovery from the faulty state. Recovery action n/a means that no action is required. See the Recovery Actions topic.

Table 2. Events detected by PFE SW

ID	Event class	Source	S32G2	S32G3	Multiplicity	Recovery
0	Reserved	n/a	*	*	0	n/a
1	SW Runtime Error	SW	*	*	0..*	Driver reset
2	PFE ECC Multi-bit Error (uncorrectable)	HW		*	0..1	Soft reset
10-27	PFE Watchdog	HW	*	*	0..1	Soft reset
30, 33	EMAC ECC (correctable)	HW	*	*	0..*	n/a
31, 34	EMAC ECC (uncorrectable)	HW	*	*	0..*	Hard reset
32, 35	EMAC ECC Address Error	HW	*	*	0..*	Hard reset
36-39	EMAC Parity	HW	*	*	0..*	Hard reset
40-44	EMAC Watchdog	HW	*	*	0..*	Hard reset
60-79	Bus Error	HW		*	0..*	Soft reset
100-130	PFE Parity	HW	*	*	0..1	Soft reset
140-145	Fail-stop	HW		*	0..1	Soft reset
150	Fail-stop	FW		*	0..1	Soft reset
151	Fail-stop	SW		*	0..1	Soft reset
170	Out of Buffers	HW	*	*	0..*	n/a
171-172	BMU Error	HW	*	*	0..*	Soft reset
180	PE Stall	FW	*	*	0..9	Soft reset
181	PE Exception	FW	*	*	0..9	Soft reset
182	FW Error	FW	*	*	0..*	Soft reset
190-192	HIF Error	HW	*	*	0..1	Soft reset

6.2 Events detected by FCCU

This table lists event classes, that can be detected by the FCCU module. The event classes are identified via the FCCU-specific NCF number.

Every event has assigned multiplicity and a recovery action. Multiplicity indicates how many times the given event can occur during PFE Driver lifecycle. The recovery action describes required action leading to recovery from the faulty state. Recovery action n/a means that no action is required. See the Recovery Actions topic.

Table 3. Events detected by FCCU

ID	Event (class)	Source	S32G2	S32G3	Multiplicity	Recovery
NCF 101	PFE ECC (uncorrectable)	HW	*	*	0..*	Soft reset
NCF 100	PFE ECC (uncorrectable)	HW		*	0..*	Soft reset
NCF 101	PFE ECC Address Error	HW	*	*	0..*	Soft reset
NCF 100	PFE Parity	HW	*	*	0..*	Soft reset
NCF 100	PFE Watchdog	HW	*	*	0..*	Soft reset
NCF 99	EMAC ECC (uncorrectable)	HW	*	*	0..*	Hard reset
NCF 99	EMAC Parity	HW	*	*	0..*	Hard reset
NCF 99	EMAC Watchdog	HW	*	*	0..*	Hard reset
NCF 100	Bus Error	HW		*	0..*	Soft reset
NCF 100	Fail-stop	HW		*	0..1	Soft reset

Except the ID, the FCCU does not provide any other event details. The ID is sufficient to implement a recommended PFE recovery procedure. For diagnostic purposes, it is beneficial to get a detailed event description. The FCCU software (interrupt handler) can optionally gather the details from dedicated location. For PFE ECC events, the syndromes can be found in PFE-related ERM registers. Diagnostic details of the remaining events are provided in human-readable form by the PFE driver.

7 Events API

Events detected by PFE driver are collected and provided to user/applications simultaneously via 3 channels:

1. Human-readable text form. The event ID and optionally detailed description/ syndrome are printed into a standard system log file or terminal.
2. Automation-friendly API: The FCI API.
3. The AUTOSAR DEM module (PFE MCAL driver only).

7.1 FCI API

The primary Health Monitoring API, that is consistent among all PFE SW, is provided by the FCI. The FCI API is described in the *FCI API Reference* [1] document. The PFE Health Monitoring-related FCI API consists of the FCI event `FPP_CMD_HEALTH_MONITOR_EVENT` and the associated data structure `fpp_health_monitor_cmd_t`. The FCI event together with its argument structure are delivered to FCI client. The FCI client is a software application running in the same execution environment as the PFE driver. The delivery to the client has unsolicited form and fulfills following principles:

- Events detected by driver, before an FCI client is available, are stored in internal event database.
- The internal events database has limited space. New events detected by the driver that can't fit into the database are dropped and shall not be provided via the FCI API. User is notified via system log.
- Events pending in the internal events database are delivered to an FCI client when the client is registered and started listening for FCI events.
- Events detected by the driver while an FCI client is available are instantly delivered to that client.

The argument data structure sent to FCI client describes the Health Monitoring event being reported and contains following fields:

- **ID**
  This value represents the event ID as defined in chapter Events part Events detected by PFE SW.
- **Type**
  This value represents the event type. The encoding is:

Table 4. Type values

Type	Description
0	Information
1	Warning
2	Error

- **Source**
  Diagnostic information describing source of the event. The encoding is:

Table 5. Source values

Source	Description
0	Unknown
1	Driver
2	Watchdog
3	EMAC0

Table 5. Source values...*continued*

Source	Description
4	EMAC1
5	EMAC2
6	BUS
7	PFE Parity
8	HW Fail-stop
9	FW Fail-stop
10	SW Fail-stop
11	PFE ECC
12	CLASS
13	UTIL
14	TMU
15	HIF
16	BMU

- **Description**
  A string containing detailed event description. Providing the description is an optional feature controlled on per-driver basis by the `PFE_CFG_HM_STRINGS_ENABLED` option. When the option is turned on, the description is filled. Otherwise, it contains an empty string.

7.2 DEM

The PFE MCAL driver can optionally use the AUTOSAR DEM module to report HM events. The AUTOSAR configuration tool includes a reference to DEM events for each event class from Table 2. Each event class can be independently enabled and configured. If such an event is reported, the Control Application can execute a recovery action immediately, or it can use the FCI API to get more information about the event.

8 Recovery Actions

Recovery actions are intended to recover PFE from faulty state. The faulty state is indicated by events listed in the chapter Events. Every event (or event class) has an appropriate recovery action assigned in the chapter Events. Executing the appropriate recovery action eliminates functional degradation caused by the faulty state.

8.1 Soft Reset

This recovery procedure refers to a soft reset of the PFE HW. The sequence involves shutting down the PFE, issuing the soft reset via dedicated PFE register, and bringing the PFE back up. All these steps are executed by PFE master driver startup/shutdown phases. Therefore, to issue a PFE soft reset, following steps are required:

1. Preferably gracefully terminate all applications leveraging the PFE.
2. Shut down all PFE slave driver instances.
3. Reset the PFE master driver instance.
4. Start all PFE slave drivers again.

8.2 Hard Reset

The hard reset represents SoC-assisted PFE HW reset. The procedure is utilizing the *Software Reset Partition* manipulation. To execute the PFE hard reset, following steps must be executed:

1. Preferably gracefully terminate all applications leveraging the PFE.
2. Shut down all PFE slave driver instances.
3. Shut down the PFE master driver instance.
4. Perform the PFE HW Reset.[2]
5. Start all PFE drivers again.

8.3 Driver Reset

Driver reset is recommended when a software error has been reported. The reset sequence consists of following steps:

1. Preferably gracefully terminate all applications leveraging the affected PFE driver.
2. Shut down the running PFE driver instance.[3]
3. Start the PFE driver instance again.

Driver shutdown and start are implementation-specific tasks. Specifics per variant are listed in chapters Driver Shutdown and Driver Startup.

Warning: *The driver reset suffers from the Driver reset in multi-instance scenarios and The detached PFE MINIHIF driver cannot be reset limitations.*

8.4 Driver Shutdown

This chapter depicts specifics of driver shutdown per implementation.

Warning: *The shutdown suffers from the Driver reset in multi-instance scenarios limitation.*

PFE Linux Driver Shutdown

Linux driver is shut down by successfully removing the driver module from the kernel.

PFE MCAL Driver Shutdown

MCAL driver is shut down by successful call of the vendor-specific `Eth_43_PFE_DeInit()` function.

PFE MINIHIF Driver (attached) Shutdown

The attached part of the PFE MINIHIF driver currently follows the PFE MCAL Driver Shutdown principles.

2 This step is necessary only when the master driver is MCAL PFE driver, or if the reset is disabled in device tree of Linux PFE driver. In all remaining cases the PFE master driver resets the PFE hardware during the driver startup.

3 This step is optional for MCAL PFE Driver. Makes sense only if a deferred Start is needed.

PFE MINIHIF Driver (detached) Shutdown

The detached MINIHIF component is shut down by successful call of the `pfe_minihif_drv_deinit()` function.

Warning: *The shutdown suffers from the The detached PFE MINIHIF driver cannot be reset limitation.*

PFE QNX Driver Shutdown

QNX driver is shut down by removing the driver library from the *io-pkt* process or by terminating the *io-pkt* process itself.

8.5 Driver Startup

This chapter depicts specifics of driver startup per implementation.

PFE Linux Driver Startup

Linux driver is started by inserting the driver module into the running kernel.

PFE MCAL Driver Startup

MCAL driver is started by calling the standard `Eth_43_PFE_Init()` function. Calling the `Eth_43_PFE_Init()` function of an already running MCAL driver instance causes that the driver instance is reset.

PFE MINIHIF Driver (attached) Startup

The attached part of the PFE MINIHIF driver currently follows the PFE MCAL Driver Startup principles.

PFE MINIHIF Driver (detached) Startup

The detached driver component is started by calling the `pfe_minihif_drv_init()` function.

PFE QNX Driver Startup

QNX driver is started by mounting the driver library into the *io-pkt* process.

8.6 PFE HW Reset

Following steps reset the PFE hardware into its initial state:

1. Execute the PFE Software Reset Partition turn-off sequence (see the *S32G Reference Manual*[2]).
2. Execute the PFE Software Reset Partition turn-on sequence (see the *S32G Reference Manual*[2]).

Note: *The software resettable partition, that corresponds to PFE within the S32G SoC, is the Partition 2.*

Note: *In the AUTOSAR MCAL environment, the MCU driver implements both the partition turn-off and turn-on sequences. See the User Manual for S32 MCU Driver [4].*

9 Error Injection

For testing purposes, a majority of Events can be invoked manually via custom software. This is called Error Injection. The following chapters describe how to inject various Events.

Limitations of Error Injection

- Every Error Injection example expects a running PFE Master Driver.
- It is recommended to always execute only one Error Injection, test the response, and then reset the driver. Successful execution of multiple Error Injections in a row (without driver reset) is not guaranteed.
- Some Events can be injected only on S32G3 SoC.
- Some Events cannot be injected.

Auxiliary Functions

Error Injection utilizes manipulation of memory content and registers. In the subsequent chapters, the following auxiliary functions are often used:

- `hal_read32()` Read 32 bit value from a physical memory/register.
- `hal_write32()` Write 32 bit value to a physical memory/register.

It is expected that users implement these auxiliary functions as they seem fit. Example of a simple implementation:

```
#define hal_read32(addr)        (*(volatile uint32_t *)(addr))
#define hal_write32(val, addr)  (*(volatile uint32_t *)(addr) = ((uint32_t)(val)))
```

9.1 SW Runtime Error (1)

Software runtime errors are reported in an event of a driver failure. The driver reports this error with every call of `NXP_LOG_ERROR`. To inject the error, use the following mechanism:

- QNX: To inject this error, update the driver code to call `NXP_LOG_ERROR("SW Runtime Error Injected.")`. Make sure that the call is invoked after the Health Monitor is initialized.
- Linux: To inject this error, update the driver code to call `NXP_LOG_ERROR("SW Runtime Error Injected.")`. Make sure that the call is invoked after the Health Monitor is initialized.
- MCAL: Call Eth_43_PFE_GetL2BridgeStats with invalid index.

9.2 PFE ECC Multi-bit Error (uncorrectable) (2, NCF 100, NCF 101)

SoC	PFE SW Events to be injected	FCCU Events to be injected
S32G2	n/a	NCF 101
S32G3	2	NCF 100, NCF 101

Note: *Correct configuration of ERM is required to propagate the NCF 101 event to FCCU.*

Execute the following code snippet:

```
hal_write32(0x00000003, 0x44054148);
hal_write32(0x40000000, 0x44054004);
hal_write32(0x00000001, 0x44054000);
hal_read32(0x46000000);
```

9.3 PFE Watchdog (10-27, NCF 100)

SoC	PFE SW Events to be injected	FCCU Events to be injected
S32G2	18, 19	NCF 100
S32G3	10, 11	NCF 100

Execute the following code snippet:

```
hal_write32(0x00000000, 0x4609408c);
```

9.4 EMAC ECC (correctable) (30, 33)

SoC	PFE SW Events to be injected	FCCU Events to be injected
S32G2	30	n/a
S32G3	30	n/a

Execute the following code snippet:

```
reg = hal_read32(0x460A0000);
hal_write32(reg & (~0x00000001), 0x460a0000);

hal_write32(0x00000103, 0x460a0c08);
while ((hal_read32(0x460a0c08) & 0x00000100) != 0)
{
    ;
}

hal_write32(0x00010003, 0x460a0c08);

for (int i = 0; i < 2; i++)
{
    hal_write32(0xabcdef01, 0x460a0c10);
    reg = hal_read32(0x460a0c08);
    hal_write32(reg | 0x00000800, 0x460a0c08);
    while ((hal_read32(0x460a0c0c) & 0x00000001) != 0)
    {
        ;
    }
}

for (int i = 0; i < 2; i++)
{
    reg = hal_read32(0x460a0c08);
    hal_write32((reg & (~0x00000800)) | 0x00000400, 0x460a0c08);
    while ((hal_read32(0x460a0c0c) & 0x00000001) != 0)
    {
        ;
    }
    hal_read32(0x460A0c10);
}
```

9.5 EMAC ECC (uncorrectable) (31, 34, NCF 99)

SoC	PFE SW Events to be injected	FCCU Events to be injected
S32G2	31	NCF 99
S32G3	31	NCF 99

Execute the following code snippet:

```
reg = hal_read32(0x460a0000);
hal_write32(reg & (~0x00000001), 0x460a0000);

hal_write32(0x00000103, 0x460a0c08);
while ((hal_read32(0x460A0c08) & 0x00000100) != 0)
{
    ;
}

hal_write32(0x00030003, 0x460a0c08);

for (int i = 0; i < 2; i++)
{
    hal_write32(0xabcdef01, 0x460a0c10);
    reg = hal_read32(0x460a0c08);
    hal_write32(reg | 0x00000800, 0x460a0c08);
    while ((hal_read32(0x460a0c0c) & 0x00000001) != 0)
    {
        ;
    }
}

for (int i = 0; i < 2; i++)
{
    reg = hal_read32(0x460a0c08);
    hal_write32((reg & (~0x00000800)) | 0x00000400, 0x460a0c08);
    while ((hal_read32(0x460a0c0c) & 0x00000001) != 0)
    {
        ;
    }
    hal_read32(0x460a0c10);
}
```

9.6 EMAC ECC Address Error (32, 35)

SoC	PFE SW Events to be injected	FCCU Events to be injected
S32G2	32	n/a
S32G3	32	n/a

Note: *Additional event IDs may be detected as side effect.*

Execute the following code snippet:

```
reg = hal_read32(0x460A0000);
hal_write32(reg & (~0x00000001), 0x460a0000);

hal_write32(0x00000103, 0x460a0c08);
while ((hal_read32(0x460a0c08) & 0x00000100) != 0)
{
    ;
```

```
}

hal_write32(0x00070003, 0x460a0c08);

for (int i = 0; i < 2; i++)
{
    hal_write32(0xabcdef01, 0x460a0c10);
    reg = hal_read32(0x460a0c08);
    hal_write32(reg | 0x00000800, 0x460a0c08);
    while ((hal_read32(0x460a0c0c) & 0x00000001) != 0)
    {
        ;
    }
}

for (int i = 0; i < 2; i++)
{
    reg = hal_read32(0x460a0c08);
    hal_write32((reg & (~0x00000800)) | 0x00000400, 0x460a0c08);
    while ((hal_read32(0x460a0c0c) & 0x00000001) != 0)
    {
        ;
    }
    hal_read32(0x460a0c10);
}
```

9.7 EMAC Parity (36-39, NCF 99)

SoC	PFE SW Events to be injected	FCCU Events to be injected
S32G2	37	NCF 99
S32G3	37	NCF 99

Execute the following code snippet:

```
reg = hal_read32(0x460a0ce0);
hal_write32(reg | 0x00000080, 0x460a0ce0);

hal_write32(0x00000103, 0x460a0c08);
while ((hal_read32(0x460a0c08) & 0x00000100) != 0)
{
    ;
}

for (i = 0; i < 2; i++)
{
    hal_write32(0xabcdef01, 0x460a0c10);
    reg = hal_read32(0x460a0c08);
    hal_write32(reg | 0x00003800, 0x460a0c08);
    while ((hal_read32(0x460a0c0c) & 0x00000001) != 0)
    {
        ;
    }
}

for (i = 0; i < 2; i++)
{
    reg = hal_read32(0x460a0c08);
    hal_write32((reg & (~0x00000800)) | 0x00003400, 0x460a0c08);
    while ((hal_read32(0x460a0c0c) & 0x00000001) != 0)
    {
        ;
    }
    hal_read32(0x460a0c10);
}
```

9.8 EMAC Watchdog (40-44, NCF 99)

SoC	PFE SW Events to be injected	FCCU Events to be injected
S32G2	42, 44	NCF 99
S32G3	42, 44	NCF 99

Execute the following code snippet:

```
hal_write32(0x00000000, 0x460a0148);
hal_write32(0x00000801, 0x460a0148);
```

9.9 Bus Error (60-79, NCF 100)

SoC	PFE SW Events to be injected	FCCU Events to be injected
S32G2	n/a	n/a
S32G3	64, 73	NCF 100

Note: *Additional event IDs may be detected as side effect.*

1. Execute the following code snippet:

```
hal_write32(0x00000109, 0x460940e8);
hal_write32(0x00001234, 0x460940ec);
hal_write32(0x00001234, 0x460940f0);
hal_write32(0x00000001, 0x4609404c);
```

2. Send packet through EMAC0.

9.10 PFE Parity (100-130, NCF 100)

SoC	PFE SW Events to be injected	FCCU Events to be injected
S32G2	n/a	n/a
S32G3	120, 121, 124	n/a

Note: *Additional event IDs may be detected as side effect. On S32G2 SoC this event can be detected, but cannot be injected.*

1. Execute the following code snippet:

```
hal_write32(0x00000109, 0x460940e8);
hal_write32(0x00001234, 0x460940ec);
hal_write32(0x00001234, 0x460940f0);
hal_write32(0x00000001, 0x4609404c);
```

2. Send packet through EMAC0.

9.11 Fail-stop (140-145, NCF 100)

SoC	PFE SW Events to be injected	FCCU Events to be injected
S32G2	n/a	n/a
S32G3	141	NCF 100

Note: *Additional event IDs may be detected as side effect.*

Execute the following code snippet:

```
hal_write32(0x00000003, 0x460940c0);
hal_write32(0x0000003F, 0x460940b4);
hal_write32(0x00000000, 0x46094088);
```

9.12 Fail-stop (150, NCF 100)

SoC	PFE SW Events to be injected	FCCU Events to be injected
S32G2	n/a	n/a
S32G3	150	NCF 100

Note: *Additional event IDs may be detected as side effect.*

Execute the following code snippet:

```
hal_write32(0x00000001, 0x460940c4);
```

9.13 Fail-stop (151, NCF 100)

SoC	PFE SW Events to be injected	FCCU Events to be injected
S32G2	n/a	n/a
S32G3	151	NCF 100

Note: *Additional event IDs may be detected as side effect.*

```
hal_write32(0x00000001, 0x460940dc);
```

9.14 Out of Buffers (170)

SoC	PFE SW Events to be injected	FCCU Events to be injected
S32G2	170	n/a
S32G3	170	n/a

Execute the following code snippet:

```
```

```
while(hal_read32(0x46088048) != 0)
{
    hal_read32(0x46088030);
}
```

9.15 BMU Error (171 - 172)

SoC	PFE SW Events to be injected	FCCU Events to be injected
S32G2	171	n/a
S32G3	171	n/a

Execute the following code snippet:

```
hal_write32(0x00000000, 0x46088034);
hal_write32(0x00000000, 0x46088034);
```

9.16 PE Stall (180)

Error injection requires PFE firmware change. Contact an NXP representative.

9.17 PE Exception (181)

Error injection requires PFE firmware change. Contact an NXP representative.

9.18 FW Error (182)

Error injection requires PFE firmware change. Contact an NXP representative.

9.19 HIF Error (190-192)

Error injection not available.

10 Limitations

This chapter describes known limitations affecting the PFE Health Monitoring.

10.1 Driver reset in multi-instance scenarios

The PFE driver software reset suffers from this limitation in multi-instance scenarios. The essence is that the PFE master driver instance cannot be currently reset (see Driver Reset topic) without affecting the other running PFE driver instances. Following constraint applies:

- When the running PFE master driver is shut down then all remaining running PFE driver instances are rendered non-functional. Functionality is recovered by resetting all PFE driver instances after the master is restarted.

10.2 The detached PFE MINIHIF driver cannot be reset

The detached part of the PFE MINIHIF driver currently does not implement support for the component shutdown and restart. To reset the detached PFE MINIHIF driver, the whole execution environment (for example, LLCE CPU) running the component must be shut down and restarted.

11 Legal information

11.1 Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

11.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at http://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Suitability for use in automotive applications — This NXP product has been qualified for use in automotive applications. If this product is used by customer in the development of, or for incorporation into, products or services (a) used in safety critical applications or (b) in which failure could lead to death, personal injury, or severe physical or environmental damage (such products and services hereinafter referred to as "Critical Applications"), then customer makes the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, safety, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. As such, customer assumes all risk related to use of any products in Critical Applications and NXP and its suppliers shall not be liable for any such use by customer. Accordingly, customer will indemnify and hold NXP harmless from any claims, liabilities, damages and associated costs and expenses (including attorneys' fees) that NXP may incur related to customer's incorporation of any product in a Critical Application.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

11.3 Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents

1 Acronyms and Definitions 2
2 References 3
3 Preface 3
4 Introduction 3
5 Architecture 4
5.1 Health Monitoring Principle 4
5.2 Reference Scenario 4
5.3 Actors 5
5.3.1 PFE HW 5
5.3.2 PFE FW 5
5.3.3 PFE SW 5
5.3.3.1 PFE Driver (master) 6
5.3.3.2 PFE Driver (slave) 6
5.3.3.3 PFE Driver (MINIHIF, attached) 6
5.3.3.4 PFE Driver (MINIHIF, detached) 6
5.3.4 FCCU and ERM 6
5.3.5 Control Application 7
5.3.6 Integrator 7
6 Events 7
6.1 Events detected by PFE SW 7
6.2 Events detected by FCCU 8
7 Events API 9
7.1 FCI API 9
7.2 DEM 10
8 Recovery Actions 10
8.1 Soft Reset 10
8.2 Hard Reset 11
8.3 Driver Reset 11
8.4 Driver Shutdown 11
8.5 Driver Startup 12
8.6 PFE HW Reset 12
9 Error Injection 13
9.1 SW Runtime Error (1) 13
9.2 PFE ECC Multi-bit Error (uncorrectable) (2, NCF 100, NCF 101) 13
9.3 PFE Watchdog (10-27, NCF 100) 14
9.4 EMAC ECC (correctable) (30, 33) 14
9.5 EMAC ECC (uncorrectable) (31, 34, NCF 99) 15
9.6 EMAC ECC Address Error (32, 35) 15
9.7 EMAC Parity (36-39, NCF 99) 16
9.8 EMAC Watchdog (40-44, NCF 99) 17
9.9 Bus Error (60-79, NCF 100) 17
9.10 PFE Parity (100-130, NCF 100) 17
9.11 Fail-stop (140-145, NCF 100) 18
9.12 Fail-stop (150, NCF 100) 18
9.13 Fail-stop (151, NCF 100) 18
9.14 Out of Buffers (170) 18
9.15 BMU Error (171 - 172) 19
9.16 PE Stall (180) 19
9.17 PE Exception (181) 19
9.18 FW Error (182) 19
9.19 HIF Error (190-192) 19
10 Limitations 19
10.1 Driver reset in multi-instance scenarios 19
10.2 The detached PFE MINIHIF driver cannot be reset 20
11 Legal information 21

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 29 June 2023
Document identifier: UM11885