

Xilinx Standalone Library Documentation

XilSEM Library v1.4

UG1355 (v2021.2) October 13, 2021



Revision History

The following table shows the revision history for this document.

Section	Revision Summary
10/13/2021 Version 1.4	
Chapter 1: Introduction	Added XiISEM Operation During Partial Bitstream Loading . Updated Initializing the SEU Mitigation , Listening for SEU Detection , and Performance .
06/16/2021 Version 1.3	
Initial release.	N/A

Table of Contents

Revision History.....	2
Chapter 1: Introduction.....	4
Features.....	5
Unsupported Features.....	6
Documentation.....	6
Configuration.....	7
Operation.....	10
Specification.....	21
Chapter 2: XiISEM Library Versal Client APIs.....	25
Functions.....	26
Chapter 3: Data Structure Index.....	36
XSem_Notifier.....	36
XSem_XmpuCfg.....	37
XSemCfrErrInjData.....	38
XSemCfrStatus.....	38
XSemIpiResp.....	39
XSemNpiStatus.....	40
Appendix A: Additional Resources and Legal Notices.....	41
Xilinx Resources.....	41
Documentation Navigator and Design Hubs.....	41
Please Read: Important Legal Notices.....	42

Introduction

The Xilinx[®] Soft Error Mitigation (XilSEM) library is a pre-configured, pre-verified solution to detect and optionally correct soft errors in Configuration Memory of Versal[™] ACAPs. A soft error is caused by ionizing radiation and is extremely uncommon in commercial terrestrial operating environments. While a soft error does not damage the device, it carries a small statistical possibility of transiently altering the device behavior.

The XilSEM library does not prevent soft errors; however, it provides a method to better manage the possible system-level effect. Proper management of a soft error can increase reliability and availability, and reduce system maintenance and downtime. In most applications, soft errors can be ignored. In applications where a soft error cannot be ignored, this library documentation provides information to configure the XilSEM library through the CIPS IP core and interact with the XilSEM library at run time.

The XilSEM library is part of the Platform Loader and Manager (PLM) which is loaded into and runs on the Platform Management Controller (PMC). When a soft error occurs, one or more bits of information are corrupted. The information affected might be in the device Configuration Memory (which determines the intended behavior of the design) or might be in design memory elements (which determine the state of the design). In the Versal architecture, Configuration Memory includes Configuration RAM and NPI Registers. Configuration RAM is associated with the Programmable Logic (PL) and is used to configure the function of the design loaded into the PL. This includes function block behavior and function block connectivity. This memory is physically distributed across the PL and is the larger category of Configuration Memory by bit count. Only a fraction of these bits are essential to the correct operation of the associated resources for the currently loaded design.

NPI Registers are associated with other function blocks which can be configured independently from the PL. The Versal ACAP architecture integrated shell is a notable application of function blocks configured in this manner, making it possible for the integrated shell to be operational with the PL in an unconfigured state. This memory is physically distributed throughout the device and is the smaller category of Configuration Memory by bit count. Only a fraction of these bits are essential to the correct operation of the associated resources for the currently loaded design.

Prior to configuring the XilSEM library for use through the CIPS IP core, assess whether the XilSEM library helps meet the soft error mitigation goals of your system's deployment and what mitigation approaches should be included, if any. There are two main considerations:

- Understanding the soft error mitigation requirements for your system

- What design and system actions need to be taken if a soft error occurs

If the effects of soft errors are a concern for your system's deployment, each component in the system design will usually need a soft error rate (SER) budget distributed from a system-level SER budget. To make estimates for the contribution from a Versal ACAP, use the Xilinx SEU Estimator tool, which is tentatively planned for availability after the UG116 publication of production qualification data for Xilinx 7 nm technology. To estimate SER for a deployment, at the minimum you need:

- Target device selection(s) to be deployed
- Estimated number of devices in deployment

In addition to providing a device level estimate, the Xilinx SEU Estimator also estimates the number of soft errors statistically likely to occur across the deployment for a selected operation time interval. After an estimate has been generated, it must be used to evaluate if the SER requirement for the deployment is met. Other design approaches might need to be implemented to reduce the system-level SER. These approaches include addition of mitigation solutions in the system, which might be suitable for implementation in any devices in the system which can offer such features. Versal ACAPs offer many supporting features, including the XiISEM library.

Consideration must be given to the system-level response, if any, that must be taken in response to a soft error in the Versal ACAP. If no action is taken when a soft error is detected in the Versal ACAP, the benefits of using the XiISEM library should be assessed and understood. While this is a valid use case, the effects of soft errors in the Versal ACAP should be anticipated and this approach should be reviewed to ensure it supports the system-level soft error mitigation strategy.

If there is no SER target at the system level, then it is unclear what system changes (and tradeoffs that come with them) need to be made to mitigate soft errors, including whether a deployment benefits from use of the XiISEM library.

Features

Following are the features of the XiISEM library:

- Integration of silicon features to fully leverage and improve upon the inherent protections for Configuration Memory.
- Scan-based detection of soft errors in Configuration RAM using ECC and CRC techniques.
- Algorithmic correction of soft errors in Configuration RAM using ECC techniques.
- Scan-based detection of soft errors in NPI Registers using SHA techniques.
- Error injection to support evaluation of system integration and response.

Unsupported Features

The following features are not supported by the XilSEM library:

- The XilSEM library does not operate on soft errors outside of Configuration RAM and NPI Registers. Soft error mitigation in other device resources, if necessary, based on design requirements, must be addressed at the user design level through measures such as redundancy or error detection and correction codes.
- The XilSEM library initializes and manages the integrated silicon features for soft error mitigation and when included in a design, do not modify settings of associated features in the PMC.
- Design simulations that include the XilSEM library are supported. However, it is not possible to observe the library and integrated silicon features in simulation. Hardware-based evaluation is required.
- The XilSEM library is not currently specified for use under high irradiation, including but not limited to space or artificially generated environments. Therefore, Xilinx does not support or answer questions specific to the application of the XilSEM library in such environments.
- Debug prints are not enabled in the XilSEM library. For debug information, use the [XSem_CmdCfrGetStatus](#) and [XSem_CmdNpiGetStatus](#). See [Chapter 2: XilSEM Library Versal Client APIs](#) for more information. You will be notified via optional IPI and GPO. For more information, see [Listening for SEU Detection](#).

Documentation

This user guide is the main documentation for the XilSEM library. This guide, along with documentation related to all products that aid in the design process, can be found on the [Xilinx Support web page](#) or by using the Xilinx Documentation Navigator. Download the Xilinx Documentation Navigator from the [Downloads page](#). For more information about this tool and the features available, open the online help after installation.

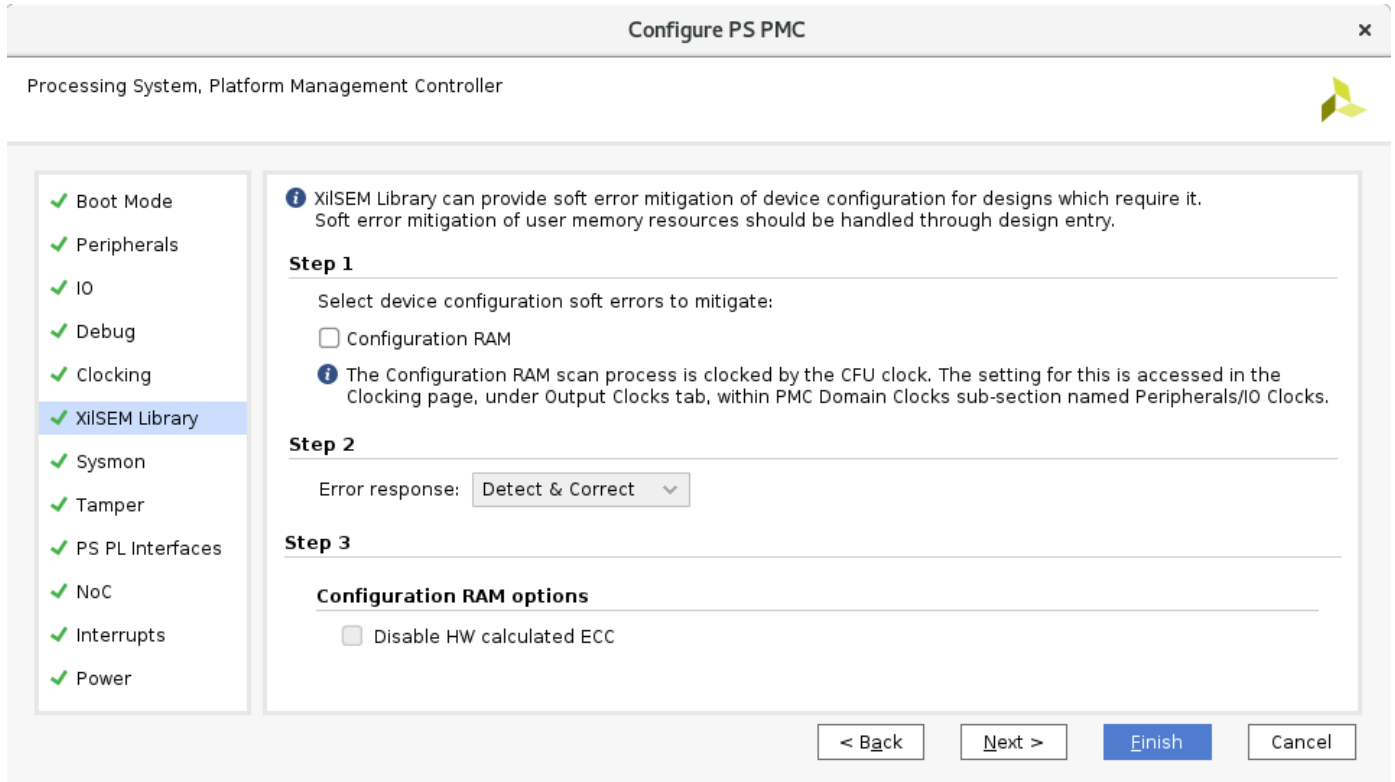
To help in the design and debug process when using the XilSEM library, the [Xilinx Support web page](#) contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support.

Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product. Answer Records are created and maintained daily ensuring that you have access to the most accurate information available. The Master Answer Record for the XilSEM library is [AR76513](#). Answer Records for the XilSEM library can be located by using the Search Support box on the main [Xilinx support web page](#).

Configuration

The XiISEM library configuration options available through the CIPS v3.0 Customize IP dialog box in the Vivado Design Suite are shown in the following figure.

Figure 1: XiISEM Library Configuration within CIPS v3.0 Customize IP Dialog Box in Vivado Design Suite



Configure PS PMC

Processing System, Platform Management Controller

- ✓ Boot Mode
- ✓ Peripherals
- ✓ IO
- ✓ Debug
- ✓ Clocking
- ✓ **XiISEM Library**
- ✓ Sysmon
- ✓ Tamper
- ✓ PS PL Interfaces
- ✓ NoC
- ✓ Interrupts
- ✓ Power

XiISEM Library can provide soft error mitigation of device configuration for designs which require it. Soft error mitigation of user memory resources should be handled through design entry.

Step 1

Select device configuration soft errors to mitigate:

☐ Configuration RAM

Step 2

Error response: **Detect & Correct**

Step 3

Configuration RAM options

☐ Disable HW calculated ECC

< Back Next > **Finish** Cancel

From within CIPS, the fundamental feature of the XiISEM library can be enabled. This feature is the soft error mitigation of Configuration RAM through a scan-based process automated by integrated logic and managed by the XiISEM library, which reads back Configuration RAM and uses ECC and CRC to detect and correct soft errors. If this feature is enabled in CIPS, the default behavior is to begin operation automatically after boot, calculate golden checksums using hardware resources, and then enable the process for error detection and correction. These options can also be accessed through properties applied to the design.

Table 1: XilSEM CRAM Properties

Property Name	Supported Values	Default Value	Description
CONFIG.PS_PMC_CONFIG {SEM_MEM_SCAN 0}	0 (Disabled) 1 (Enabled)	0 (Disabled)	Disabled: No Configuration RAM scan is enabled. Enabled: Configuration RAM scan is enabled.
CONFIG.PS_PMC_CONFIG {SEM_MEM_ENABLE_SCAN_AFTER 0}	0 (Immediate start) 1 (Deferred start)	0 (Immediate start)	Immediate start: Enables automatic start of Configuration RAM scanning after boot. Deferred start: Start of the scan during mission based on user request.
CONFIG.PS_PMC_CONFIG {SEM_MEM_GOLDEN_ECC_SW 0}	0 (HW ECC) 1 (SW ECC)	0 (HW ECC)	HW ECC: Hardware computes the golden ECC values during calibration. SW ECC: Tool generated values are used as golden ECC values for Configuration RAM scan.
CONFIG.PS_PMC_CONFIG {SEM_ERROR_HANDLE_OPTIONS {Detect & Correct}}	{Detect only} {Detect & correct}	{Detect & correct}	Detect only: No correction is done for all correctable errors. All errors are reported to the user. Detect & correct: Correction is enabled for all correctable errors. Correctable and uncorrectable errors are reported to the user.

Advanced features of the XilSEM library can be accessed through properties applied to the design. The XilSEM library provides an optional feature for the soft error mitigation of NPI Registers, provided the supplemental hardware resources required for this feature are accessible. This feature is a scan-based process automated by integrated logic and managed by the XilSEM library, which reads back NPI Registers and uses SHA to detect errors.

NPI Register scan requires XilSEM library use of PMC cryptographic acceleration. The following are some of the scenarios in which the XilSEM library cannot offer this optional feature:

- User operation of the SHA block in the PMC for purposes of the user design
- Device is programmed for “no end-user accessible” cryptographic functions

Table 2: XilSEM NPI Properties

Property Name	Supported Values	Default Value	Description
CONFIG.PS_PMC_CONFIG {SEM_NPI_SCAN 0}	0 (Disabled) 1 (Enabled)	0 (Disabled)	Disabled: No NPI scan is enabled. Enabled: NPI scan is enabled.

Table 2: XilSEM NPI Properties (cont'd)

Property Name	Supported Values	Default Value	Description
CONFIG.PS_PMC_CONFIG {SEM_NPI_ENABLE_SCAN_AFTER 0}	0 (Immediate start) 1 (Deferred start)	0 (Immediate start)	Immediate start: Enables automatic start of NPI scanning after boot. Deferred start: Start of the NPI scan during mission based on your request.
BITGEN.GENERATESWSHA	False (HW SHA) True (SW SHA)	True (SW SHA)	HW SHA: The value calculated during the first scan will be used as golden SHA. SW SHA: Tool generated values are used as golden SHA for NPI scan.
CONFIG.PS_PMC_CONFIG {SEM_TIME_INTERVAL_BETWEEN_SCANS 100}	>=100 msec && <=1000 msec	100 msec	The interval in milliseconds at which the NPI scan is repeated. Recommended to use 100 msec.

Additional XilSEM library properties as referenced below are available.

Table 3: Additional XilSEM Library Properties

Property Name	Supported Values	Default Value	Description
CONFIG.PS_PMC_CONFIG {PMC_GPO_ENABLE 0}	0 (Disabled) 1 (Enabled)	0 (Disabled)	Disabled: PMC_PL_GPO will not be triggered for status and error information. Enabled: PMC_PL_GPO will be triggered for status and error information.

Configuring XilSEM

CIPS does not support a GUI interface to configure XilSEM parameters. So, you need to set the XilSEM library parameters using the Vivado Tcl/TK command. Here is a sample command to enable XilSEM with the hardware ECC/SHA and immediate start-up:

```
set_property -dict [list CONFIG.PS_PMC_CONFIG {SEM_MEM_SCAN 1
SEM_MEM_ENABLE_ALL_TEST_FEATURE 0 SEM_MEM_GOLDEN_ECC_SW 0
SEM_MEM_ENABLE_SCAN_AFTER 1 SEM_NPI_SCAN 1 SEM_NPI_ENABLE_SCAN_AFTER 1 } ]
[get_bd_cells versal_cips_0]
set_property bitstream.general.clearecc no [current_bd_design]
set_property bitstream.general.compress true [current_bd_design]
validate_bd_design
```

Operation

Initializing the SEU Mitigation

To initialize the SEU mitigation, it is first necessary to successfully initialize the inter-processor interrupt (IPI). The following reference code, which can be part of an RPU or APU application, illustrates how to initialize the IPI:

```
#define TARGET_IPI_INT_DEVICE_ID
(XPAR_XIPIPSU_0_DEVICE_ID)

/* Load Config for Processor IPI Channel */
IpiCfgPtr = XIpiPsu_LookupConfig(TARGET_IPI_INT_DEVICE_ID);
if (NULL == IpiCfgPtr) {
    xil_printf("[%s] ERROR: IPI LookupConfig failed\n\r", \
        __func__, Status);
    goto END;
}

/* Initialize the IPI Instance pointer */
Status = XIpiPsu_CfgInitialize(&IpiInst, IpiCfgPtr,
    IpiCfgPtr->BaseAddress);
if (XST_SUCCESS != Status) {
    xil_printf("[%s] ERROR: IPI instance initialize failed\n\r", \
        __func__, Status);
    goto END;
}
```

Although Configuration RAM scan can be configured to begin automatically, in XiISEM library configurations where this option is not selected, the user is responsible for manually starting the Configuration RAM scan. The following reference code, which can be part of an RPU or APU application, illustrates how to accomplish Configuration RAM scan start-up.

```
XStatus Status = XST_FAILURE;
XSemIpiResp IpiResp={0};
Status = XSem_CmdCfrInit(&IpiInst, &IpiResp);
if ((XST_SUCCESS == Status) && (CMD_ACK_CFR_INIT == IpiResp.RespMsg1) &&
    (XST_SUCCESS == IpiResp.RespMsg2)) {
    xil_printf("Success: CRAM Initialization Successful\n\r");
} else {
    xil_printf("Failure: CRAM Initialization Failed\n\r");
    Status = XST_FAILURE;
}
```

Similarly, NPI Register scan can be configured to begin automatically, and in XiISEM library configurations where this option is not selected, you are responsible for manually starting the NPI Register scan. The following reference code, which can be part of an RPU or APU application, illustrates how to accomplish NPI Register scan start-up.

Note: An NPI scan cannot be performed in SHA3 disabled devices. If SHA3 is not present, XilSEM updates the status in NPI scan status register. You can use the [XSem_CmdNpiGetStatus](#) API to know NPI scan status.

```
XStatus Status = XST_FAILURE;
XSemIpiResp IpiResp={0};
Status = XSem_CmdNpiStartScan(&IpiInst, &IpiResp);
if ((XST_SUCCESS == Status) && (CMD_ACK_NPI_STARTSCAN == IpiResp.RespMsg1)
&& (XST_SUCCESS == IpiResp.RespMsg2)) {
    xil_printf("Success: NPI Initialization Successful\n\r");
} else {
    xil_printf("Failure: NPI Initialization Failed\n\r");
    Status = XST_FAILURE;
}
```

Listening for SEU Detection

The XilSEM library can maintain SEU mitigation operation without any need for the user design to listen for SEU detections. However, it may be desired maintain an event log, or take design or system level actions in response to an event.

Whenever an error is detected, be it correctable or uncorrectable, XilSEM has a mechanism to notify the errors over IPI. This equally applies for errors detected during NPI scanning and CRAM scanning. To receive notification of error detections in Configuration RAM or NPI Registers, the following steps are required of an RPU or APU application:

- GIC initialization
- IPI initialization
 - IPI configuration and connection with GIC
 - IPI callback registration with IPI interrupt handler (CRAM and NPI)
- Register error event notification (CRAM and NPI)
- Check global variables that hold notified event information (CRAM and NPI)

For GIC initialization, use the following code snippet:

```
#define INTC_DEVICE_ID (XPAR_SCUGIC_SINGLE_DEVICE_ID)
s32 GicSetupInterruptSystem(XScuGic *GicInst)
{
    s32 Status;

    XScuGic_Config *GicCfgPtr = XScuGic_LookupConfig(INTC_DEVICE_ID);
    if (NULL == GicCfgPtr) {
        xil_printf("XScuGic_LookupConfig() failed\n\r");
        goto END;
    }

    Status = XScuGic_CfgInitialize(GicInst, GicCfgPtr, \
        GicCfgPtr->CpuBaseAddress);
    if (XST_SUCCESS != Status) {
        xil_printf("XScuGic_CfgInitialize() failed with error: %d\n\r", \
```

```

        Status);
        goto END;
    }

    /*
     * Connect the interrupt controller interrupt Handler to the
     * hardware interrupt handling logic in the processor.
     */
#ifdef (__aarch64__)
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_FIQ_INT,
#ifdef (__arm__)
        Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,
#endif
        (Xil_ExceptionHandler)XScuGic_InterruptHandler, GicInst);
    Xil_ExceptionEnable();

END:
    return Status;
}

```

For IPI configuration and connection with GIC, use the following code snippet:

```

#define IPI_TEST_CHANNEL_ID
    (XPAR_XIPIPSU_0_DEVICE_ID)
#define IPI_INT_ID
    (XPAR_XIPIPSU_0_INT_ID)

/* Allocate one callback pointer for each bit in the register */
static IpiCallback IpiCallbacks[11];

static ssize_t ipimask2idx(u32 m)
{
    return __builtin_ctz(m);
}

/**
 * IpiIrqHandler() - Interrupt handler of IPI peripheral
 * @InstancePtr    Pointer to the IPI data structure
 */
static void IpiIrqHandler(XIpiPsu *InstancePtr)
{
    u32 Mask;

    /* Read status to determine the source CPU (who generated IPI) */
    Mask = XIpiPsu_GetInterruptStatus(InstancePtr);

    /* Handle all IPIs whose bits are set in the mask */
    while (Mask) {
        u32 IpiMask = Mask & (-Mask);
        ssize_t idx = ipimask2idx(IpiMask);

        /* If the callback for this IPI is registered execute it */
        if (idx >= 0 && IpiCallbacks[idx])
            IpiCallbacks[idx](InstancePtr);

        /* Clear the interrupt status of this IPI source */
        XIpiPsu_ClearInterruptStatus(InstancePtr, IpiMask);

        /* Clear this IPI in the Mask */
        Mask &= ~IpiMask;
    }
}

```

```
static XStatus IpiConfigure(XIpiPsu * IpiInst, XScuGic * GicInst)
{
    int Status = XST_FAILURE;
    XIpiPsu_Config *IpiCfgPtr;

    if (NULL == IpiInst) {
        goto END;
    }

    if (NULL == GicInst) {
        xil_printf("%s ERROR GIC Instance is NULL\n", __func__);
        goto END;
    }

    /* Look Up the config data */
    IpiCfgPtr = XIpiPsu_LookupConfig(IPI_TEST_CHANNEL_ID);
    if (NULL == IpiCfgPtr) {
        Status = XST_FAILURE;
        xil_printf("%s ERROR in getting CfgPtr\n", __func__);
        goto END;
    }
    /* Init with the Cfg Data */
    Status = XIpiPsu_CfgInitialize(IpiInst, IpiCfgPtr, \
        IpiCfgPtr->BaseAddress);
    if (XST_SUCCESS != Status) {
        xil_printf("%s ERROR #%d in configuring IPI\n", __func__,
            Status);
        goto END;
    }

    /* Clear Any existing Interrupts */
    XIpiPsu_ClearInterruptStatus(IpiInst, XIPIPSU_ALL_MASK);

    Status = XScuGic_Connect(GicInst, IPI_INT_ID,
        (Xil_ExceptionHandler)IpiIrqHandler, IpiInst);
    if (XST_SUCCESS != Status) {
        xil_printf("%s ERROR #%d in GIC connect\n", __func__, Status);
        goto END;
    }
    /* Enable IPI interrupt at GIC */
    XScuGic_Enable(GicInst, IPI_INT_ID);

END:
    return Status;
}
```

For IPI callback registration with IPI handler, see the following code snippet:

```
XStatus IpiRegisterCallback(XIpiPsu *const IpiInst, const u32 SrcMask,
    IpiCallback Callback)
{
    ssize_t idx;

    if (!Callback)
        return XST_INVALID_PARAM;

    /* Get index into IpiChannels array */
    idx = ipimask2idx(SrcMask);
    if (idx < 0)
        return XST_INVALID_PARAM;
```

```

/* Check if callback is already registered, return failure if it is */
if (IpiCallbacks[idx])
    return XST_FAILURE;

/* Entry is free, register callback */
IpiCallbacks[idx] = Callback;

/* Enable reception of IPI from the SrcMask/CPU */
XIpiPsu_InterruptEnable(IpiInst, SrcMask);

return XST_SUCCESS;
}

```

For IPI callback to receive event messages for CRAM, see the following code snippet:

```

#define SRC_IPI_MASK
    (XPAR_XIPIPS_TARGET_PSV_PMC_0_CH0_MASK)

/*Global variables to hold the event count when notified*/
u8 EventCnt_UnCorEcc = 0U;
u8 EventCnt_Crc = 0U;
u8 EventCnt_CorEcc = 0U;
u8 EventCnt_IntErr = 0U;

void XSem_IpiCallback(XIpiPsu *const InstancePtr)
{
    int Status;
    u32 Payload[PAYLOAD_ARG_CNT] = {0};

    Status = XIpiPsu_ReadMessage(XSem_IpiGetInst(), SRC_IPI_MASK, Payload, \
        PAYLOAD_ARG_CNT, XIPIPSU_BUF_TYPE_MSG);
    if (Status != XST_SUCCESS) {
        xil_printf("ERROR #%d while reading IPI buffer\n", Status);
        return;
    }

    if ((XSEM_EVENT_ERROR == Payload[0]) && \
        (XSEM_NOTIFY_CRAM == Payload[1])) {
        if (XSEM_EVENT_CRAM_UNCOR_ECC_ERR == Payload[2]) {
            EventCnt_UnCorEcc++;
        } else if (XSEM_EVENT_CRAM_CRC_ERR == Payload[2]) {
            EventCnt_Crc++;
        } else if (XSEM_EVENT_CRAM_INT_ERR == Payload[2]) {
            EventCnt_IntErr++;
        } else if (XSEM_EVENT_CRAM_COR_ECC_ERR == Payload[2]) {
            EventCnt_CorEcc++;
        } else {
            xil_printf("%s Some other callback received: %d:%d:%d\n", \
                __func__, Payload[0], \
                Payload[1], Payload[2]);
        }
    } else {
        xil_printf("%s Some other callback received: %d\n", \
            __func__, Payload[0]);
    }
}

```

Note: In the above code snippets, global counters are incremented when an event has been notified. It is up to the user to define and implement any desired design and system response.

For IPI callback to receive event messages for NPI, see the following code snippet:

```
/*Global variables to hold the event count when notified*/
u8 NPI_CRC_EventCnt = 0U;
u8 NPI_INT_EventCnt = 0U;

void XSem_IpiCallback(XIpiPsu *const InstancePtr)
{
    int Status;
    u32 Payload[PAYLOAD_ARG_CNT] = {0};

    Status = XIpiPsu_ReadMessage(&IpiInst, SRC_IPI_MASK, Payload,
PAYLOAD_ARG_CNT,
    XIPIPSU_BUF_TYPE_MSG);
    if (Status != XST_SUCCESS) {
        xil_printf("ERROR #%d while reading IPI buffer\n", Status);
        return;
    }

    if ((XSEM_EVENT_ERROR == Payload[0]) && (XSEM_NOTIFY_NPI ==
Payload[1])) {
        if (XSEM_EVENT_NPI_CRC_ERR == Payload[2]) {
            NPI_CRC_EventCnt++;
        } else if (XSEM_EVENT_NPI_INT_ERR == Payload[2]) {
            NPI_INT_EventCnt++;
        } else {
            xil_printf("%s Some other callback received: %d:%d:%d\n",
                __func__, Payload[0], Payload[1], Payload[2]);
        }
    } else {
        xil_printf("%s Some other callback received: %d\n", __func__,
Payload[0]);
    }
}
```

For IPI initialization (includes IPI configuration and connection with GIC, IPI callback registration with IPI handler), see the following code snippet:

```
XStatus IpiInit(XIpiPsu * InstancePtr, XScuGic * GicInst)
{
    int Status;

    Status = IpiConfigure(InstancePtr, GicInst);
    if (XST_SUCCESS != Status) {
        xil_printf("IpiConfigure() failed with error: %d\r\n",
            Status);
    }

    Status = IpiRegisterCallback(InstancePtr, SRC_IPI_MASK, \
        XSem_IpiCallback);
    return Status;
}
```

Note: CRAM and NPI should run one at time.

For initializing GIC, XiSEM IPI instance and registering ISR handler to process XiSEM notifications from PLM, use the below code snippet:

```
XStatus XSem_IpiInitApi (void)
{
    XStatus Status = XST_FAILURE;

    /* GIC Initialize */
    Status = GicSetupInterruptSystem(&GicInst);
    if (Status != XST_SUCCESS) {
        xil_printf("GicSetupInterruptSystem failed with error: %d\r\n", \
            Status);
        goto END;
    }

    Status = IpiInit(&IpiInst, &GicInst);
    if (XST_SUCCESS != Status) {
        xil_printf("[%s] IPI Init Error: Status 0x%x\r\n", \
            __func__, Status);
        goto END;
    }

END:
    return Status;
}
```

For register error event notification with CRAM, see the following code snippet:

```
XSem_Notifier Notifier = {
    .Module = XSEM_NOTIFY_CRAM,
    .Event = XSEM_EVENT_CRAM_UNCOR_ECC_ERR | XSEM_EVENT_CRAM_CRC_ERR | \
        XSEM_EVENT_CRAM_INT_ERR | XSEM_EVENT_CRAM_COR_ECC_ERR,
    .Flag = 1U,
};

int Status;
Status = XSem_RegisterEvent(&IpiInst, &Notifier);
if (XST_SUCCESS == Status) {
    xil_printf("Success: Event registration \n\r");
} else {
    xil_printf("Error: Event registration failed \n\r");
    goto END;
}
```

For register error event notification with NPI, see the following code snippet:

```
XSem_Notifier Notifier = {
    .Module = XSEM_NOTIFY_NPI,
    .Event = XSEM_EVENT_NPI_CRC_ERR | XSEM_EVENT_NPI_INT_ERR,
    .Flag = 1U,
};

int Status;
Status = XSem_RegisterEvent(&IpiInst, &Notifier);
if (XST_SUCCESS == Status) {
    xil_printf("Success: Event registration \n\r");
} else {
    xil_printf("Error: Event registration failed \n\r");
    goto END;
}
```


To check global variables that holds the count of the notified event for CRAM, use the following code snippet:

```
if(EventCnt_UnCorEcc > 0){
    xil_printf("Uncorrectable error has been detected in CRAM\n\r");
}else if(EventCnt_Crc > 0){
    xil_printf("CRC error has been detected in CRAM\n\r");
} else if(EventCnt_CorEcc > 0){
    xil_printf("Correctable error has been detected and corrected in CRAM\n\r");
} else if(EventCnt_IntErr > 0){
    xil_printf("Internal error has occurred in CRAM\n\r");
}
```

To check global variables that holds the count of the notified event for NPI, use the following code snippet:

```
if(NPI_CRC_EventCnt > 0){
    xil_printf("CRC error has been detected in NPI\n\r");
}else if(NPI_INT_EventCnt > 0){
    xil_printf("Internal error has occurred in NPI\n\r");
}
```

Note: The IPI notification will happen only when you register IPI notifications in your application. It is recommended that before the applications sets in to mission mode, you shall read XiISEM status using the [XSem_CmdCfrGetStatus](#) and [XSem_CmdNpiGetStatus](#) APIs.

An additional method exists for the user design to listen for SEU detections. Unlike the IPI method which is used for RPU / APU user software, this method is used for PL user logic. It involves PMC_PL_GPO outputs of the PMC routed into the PL, supplying quick access to key event information. To access this feature, it must be enabled during XiISEM library configuration.

- Bit 0: CRAM correctable error

The GPO is set to High when the error is detected. The GPO is cleared when the error is corrected. If you have disabled correction in the CIPS configuration, the GPO is set and remains High until CRAM scan is re-initialized.

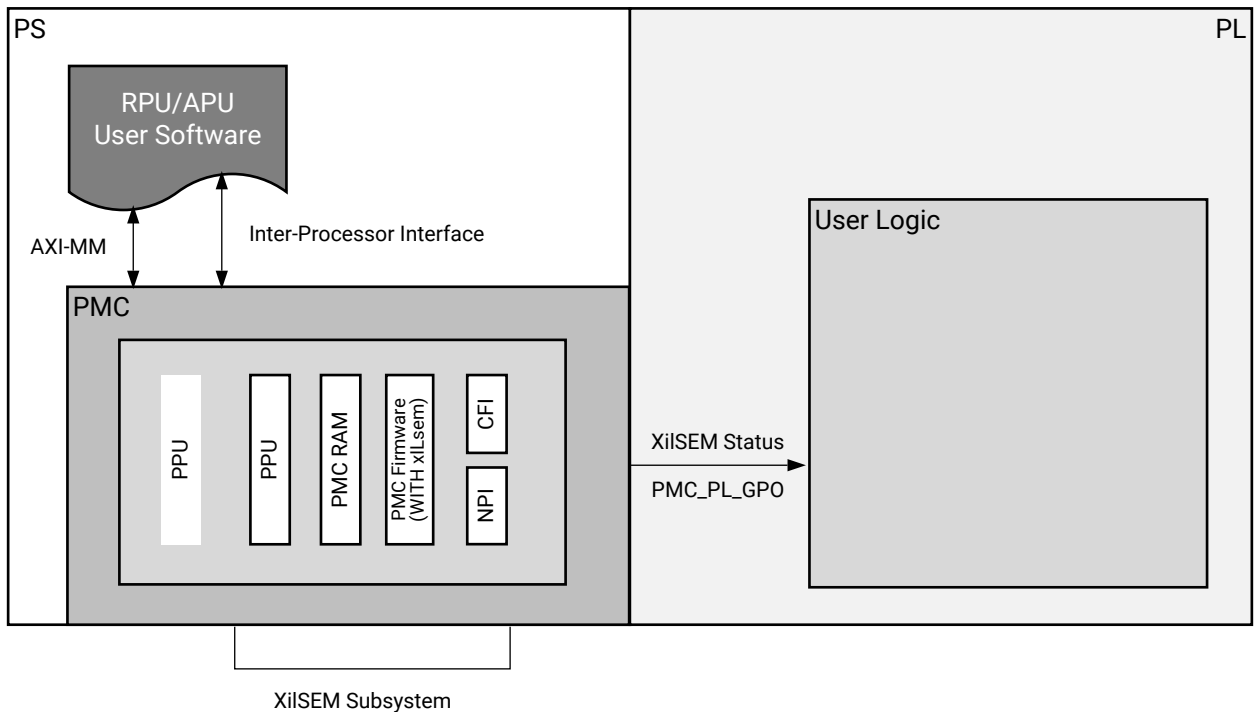
- Bit 1: CRAM/NPI uncorrectable error

The GPO is set to High when the error is detected.

- Bit 2: CRAM/NPI/XiISEM internal error
- Bit 3: Reserved

The following diagram provides a XiISEM-centric view of information conduits to user software and user logic implemented in a Versal ACAP.

Figure 2: Flow of Information on a Versal ACAP using a XilSEM Subsystem



X25366-052021

Injecting Errors for Test

Injecting an Error in Configuration RAM

Here are steps and code snippets illustrating how to inject an error in Configuration RAM for test purposes.

1. Stop Scan

```
/*To stop scan*/
XSemIpiResp IpiResp = {0};
XStatus Status = XST_FAILURE;
Status = XSem_CmdCfrStopScan(&IpiInst, &IpiResp);
if ((XST_SUCCESS == Status) && \
    (CMD_ACK_CFR_STOP_SCAN == IpiResp.RespMsg1) && \
    (XST_SUCCESS == IpiResp.RespMsg2)) {
    xil_printf("[%s] Success: Stop\n\r", __func__);
} else {
    xil_printf("[%s] Error: Stop Status 0x%x Ack 0x%x, Ret 0x%x", \
        __func__, Status, IpiResp.RespMsg1, IpiResp.RespMsg2);
    goto END;
}
```

2. Inject Error

Note: The error injection shall be done with valid CFRAME address and valid row.

- For valid range of addresses, refer to the LAST_FRAME_TOP (CFRAME_REG) and LAST_FRAME_BOT (CFRAME_REG) registers.
- For valid row, refer to the CFU_ROW_RANGE (CFU_CSR) register.

For register information, see *Versal ACAP Register Reference* (AM012).

```
/* Inject 1-bit correctable error */
XSemIpiResp IpiResp = {0};
XStatus Status = XST_FAILURE;
XSemCfrErrInjData ErrData = {0x7, 0x0, 0x0, 0x0};
Status = XSem_CmdCfrNjctErr(&IpiInst, &ErrData, &IpiResp);
if ((XST_SUCCESS == Status) && \
    (CMD_ACK_CFR_NJCT_ERR == IpiResp.RespMsg1) && \
    (XST_SUCCESS == IpiResp.RespMsg2)) {
    xil_printf("[%s] Success: Inject\n\r", __func__);
} else {
    xil_printf("[%s] Error: Inject Status 0x%x Ack 0x%x, Ret 0x%x", \
        __func__, Status, IpiResp.RespMsg1, IpiResp.RespMsg2);
    goto END;
}
```

3. Start scan to detect/correct injected error

```
/*To start scan*/
Status = XSem_CmdCfrStartScan(&IpiInst, &IpiResp);
if ((XST_SUCCESS == Status) && \
    (CMD_ACK_CFR_START_SCAN == IpiResp.RespMsg1) && \
    (XST_SUCCESS == IpiResp.RespMsg2)) {
    xil_printf("[%s] Success: Start\n\r", __func__);
} else {
    xil_printf("[%s] Error: Start Status 0x%x Ack 0x%x, Ret 0x%x", \
        __func__, Status, IpiResp.RespMsg1, IpiResp.RespMsg2);
    goto END;
}
```

Wait for the XilSEM library to detect and optionally correct the error (if configured to do so) and report the error. The following code illustrates how to validate if the error injection was successful.

```
/*To validate injection*/
while (XST_SUCCESS != XSem_ChkCfrStatus(0x1, 0x800U, 11U)) {
    usleep(100);
}
xil_printf("Cram Status [0x%08x]\n\r", CfrStatusInfo.Status);
xil_printf("Cor Ecc Cnt [0x%08x]\n\r", CfrStatusInfo.ErrCorCnt);
xil_printf("Error address details are as\n\r");
for (Index = 0U; Index < MAX_CRAMERR_REGISTER_CNT; Index++) {
    xil_printf("\nErrAddrL%d [0x%08x]\n\r", \
        Index, CfrStatusInfo.ErrAddrL[Index]);
    xil_printf("ErrAddrH%d [0x%08x]\n\r", \
        Index, CfrStatusInfo.ErrAddrH[Index]);
}
/*Function to check CFR status*/
static XStatus XSem_ChkCfrStatus(u32 ExpectedStatus, u32 Mask, u32 Shift)
{
    XSemCfrStatus CfrStatusInfo = {0};
    XStatus Status = XST_FAILURE;
    u32 Temp = 0U;
```

```

Status = XSem_CmdCfrGetStatus(&CfrStatusInfo);
if (XST_SUCCESS == Status) {
    Temp = DataMaskShift(CfrStatusInfo.Status, Mask, Shift);
    if (Temp != ExpectedStatus) {
        Status = XST_FAILURE;
    }
}
return Status;
}

```

Injecting an Error in NPI Registers

Here are steps and code snippets illustrating how to inject an error in NPI registers for test purposes.

1. Stop scan

```

/*To stop scan*/
Status = XSem_CmdNpiStopScan(&IpiInst, &IpiResp);
if ((XST_SUCCESS == Status) &&
    (CMD_ACK_NPI_STOPSCAN == IpiResp.RespMsg1) &&
    (XST_SUCCESS == IpiResp.RespMsg2)) {
    xil_printf("[%s] Success: Stop\n\r", __func__);
} else {
    xil_printf("[%s] Error: Stop Status 0x%x Ack 0x%x, Ret 0x%x\n\r", \
        __func__, Status, IpiResp.RespMsg1, IpiResp.RespMsg2);
    goto END;
}

```

2. Inject error

```

/* Inject error in first used SHA */
Status = XSem_CmdNpiInjectError(&IpiInst, &IpiResp);
if ((XST_SUCCESS == Status) &&
    (CMD_ACK_NPI_ERRINJECT == IpiResp.RespMsg1) &&
    (XST_SUCCESS == IpiResp.RespMsg2)) {
    xil_printf("[%s] Success: Inject\n\r", __func__);
} else {
    xil_printf("[%s] Error: Inject Status 0x%x Ack 0x%x, Ret 0x%x\n\r", \
        __func__, Status, IpiResp.RespMsg1, IpiResp.RespMsg2);
    goto END;
}

```

3. Start scan to detect injected error

```

/*To start scan*/
Status = XSem_CmdNpiStartScan(&IpiInst, &IpiResp);
if ((XST_SUCCESS == Status) &&
    (CMD_ACK_NPI_STARTSCAN == IpiResp.RespMsg1) &&
    (XST_SUCCESS == IpiResp.RespMsg2)) {
    xil_printf("[%s] Success: Start\n\r", __func__);
} else {
    xil_printf("[%s] Error: Start Status 0x%x Ack 0x%x, Ret 0x%x\n\r", \
        __func__, Status, IpiResp.RespMsg1, IpiResp.RespMsg2);
    goto END;
}

```

Wait for the XilSEM library to detect and report the error. The following code illustrates how to validate if the error injection was successful.

```
/*To validate injection*/
TimeoutCount = 4U;
while (TimeoutCount != 0U) {
    Status = XSem_CmdNpiGetStatus(&NpiStatus);
    if (XST_SUCCESS != Status) {
        xil_printf("[%s] ERROR: NPI Status read failure\n\r", \
            __func__, Status);
        goto END;
    }
    /* Read NPI_SCAN_ERROR status bit */
    TempA_32 = ((NpiStatus.Status & 0x00020000U) >> 17U);
    if (TempA_32 == 1U) {
        goto END;
    }
    TimeoutCount--;
    /* Small delay before polling again */
    usleep(25000);
}
xil_printf("[%s] ERROR: Timeout occurred waiting for error\n\r", __func__);
Status = XST_FAILURE;
END:
return Status;
```

Note: .

- All the macros used in the code snippets are defined in `xsem_client_api.h`
- The above code snippets are demonstrated through examples in the XilSEM library using `xsem_cram_example.c` and `xsem_npi_example.c`. You can import these examples into the Vitis IDE to get more implementation details.

XilSEM Operation During Partial Bitstream Loading

When you make a request for partial bitstream loading, both CRAM scan and NPI scan are suspended. XilSEM reinitializes both CRAM and NPI scan operations, once the bitstream loading is complete.

Specification

Standards

No standards compliance or certification testing is defined. The XilSEM library, running on Versal ACAPs, is exposed to a beam of accelerated particles as part of an extensive hardware validation process.

Performance

Performance metrics for the XiISEM library are derived from silicon specifications and direct measurement and are for budgetary purposes only. Actual performance might vary.

Table 4: Performance Metrics for the Configuration RAM

Device and Conditions	Initialization	Complete Scan Time	Correctable ECC Error Handling	Uncorrectable CRC Error Handling	Other Uncorrectable Error Handling
XCVC1902 PMC = 320 MHz CFU = 400 MHz	SW ECC: 18.1 ms HW ECC: 36.2 ms	13.6 ms (with 149682 CRAM frames)	55 us	16 us	49 us
XCVM1802 PMC = 320 MHz CFU = 400 MHz	SW ECC: 18.1 ms HW ECC: 36.2 ms	13.6 ms (with 149682 CRAM frames)	55 us	16 us	49 us

Table 5: Performance Metrics for NPI Registers

Device and Conditions	Initialization	Complete Scan Time	Uncorrectable SHA Error Handling	Other Uncorrectable Error Handling
XCVC1902 PMC = 320 MHz CFU = 400 MHz	SW SHA: 13.2 ms HW SHA: 13.2 ms	13.3 ms (with 900+ NPI slaves)	12.47 ms	342.8 s (DMA to SHA transfer timeout)
XCVM1802 PMC = 320 MHz CFU = 420 MHz	SW SHA: 14.7 ms HW SHA: 14.7 ms	14.7 ms (with 900+ NPI slaves)	13.7 ms	353.5 s (DMA to SHA transfer timeout)

Error detection latency is the major component of the total error mitigation latency. Error detection latency is a function of the device size and the underlying clock signals driving the processes involved, as these determine the Complete Scan time. It is also a function of the type of error and the relative position of the error with respect to the position of the scan process, at the time the error occurs. The error detection latency can be bounded as follows:

- Maximum error detection latency for detection by ECC is one Complete Scan Time
 - This represents a highly unlikely case when an error at a given location occurs directly “behind” the scan process.
 - It will take one Complete Scan Time for the scan process to return to the error location at which time it will detect it.

- Maximum error detection latency for detection by CRC or SHA is $2.0 \times \text{Complete Scan Time}$
 - This represents an extremely unlikely case when an error occurs directly “behind” the scan process and is located at the scan start location (where the checksum accumulation begins at each scan).
 - It will take one Complete Scan Time for the scan process to complete the current checksum accumulation (which will pass) and then a second Complete Scan Time to complete a checksum accumulation which includes the error (which will fail).

Reliability

As part of the extensive hardware validation process for the XilSEM library, reliability metrics are experimentally obtained and summarized in the table below.

Table 6: Reliability

Versal ACAP Series	XilSEM Failure Rate
AI Core, Prime, Premium (monolithic devices)	Pending Characterization

Throughput

The throughput metrics of the XilSEM library are not specified.

Power

The power metrics of the XilSEM library are not specified. However, the Xilinx Power Estimator (XPE) can be used to estimate XilSEM library power, which is mostly contributed by the Configuration RAM scan process operated on the CFU clock domain. The CFU clock default frequency set by CIPS is high, to maximize bandwidth of the Configuration RAM. Depending on design requirements, it may be desirable to operate at lower CFU clock frequencies.

CIPS currently supports user access to edit many clock generator settings, including the CFU clock. Users can take advantage of this compile time tradeoff by lowering the CFU clock setting. This change, unless later overridden during design operation, will impact any CFU activity and that has broader implications than the XilSEM library. For example, other activities which use the CFU include (but are not limited to):

- PL Housekeeping at Boot
- Readback Capture and Verify
- DFX / Partial Reconfig

Using this compile time tradeoff is simple and can work well, especially with smaller devices where the bandwidth to size ratio of Configuration RAM might have ample margin vs. design requirements to support trading it for power reduction. However, it is critically important to evaluate the broader impact of this tradeoff beyond XilSEM library power.

XilSEM Library Versal Client APIs

This file provides XilSEM library client interface to send IPI requests from the user application to XilSEM library server on PLM. This provide APIs to Start, Stop, Error Injection, Event notification registration, Status details for both CRAM and NPI.

Table 7: Quick Function Reference

Type	Name	Arguments
XStatus	XSem_CmdCfrInit	XIpiPsu * IpiInst XSemIpiResp * Resp
XStatus	XSem_CmdCfrStartScan	XIpiPsu * IpiInst XSemIpiResp * Resp
XStatus	XSem_CmdCfrStopScan	XIpiPsu * IpiInst XSemIpiResp * Resp
XStatus	XSem_CmdCfrNjctErr	XIpiPsu * IpiInst XSemCfrErrInjData * ErrDetail XSemIpiResp * Resp
XStatus	XSem_CmdCfrGetStatus	XSemCfrStatus * CfrStatusInfo
XStatus	XSem_CmdNpiStartScan	XIpiPsu * IpiInst XSemIpiResp * Resp
XStatus	XSem_CmdNpiStopScan	XIpiPsu * IpiInst XSemIpiResp * Resp
XStatus	XSem_CmdNpiInjectError	XIpiPsu * IpiInst XSemIpiResp * Resp
XStatus	XSem_CmdNpiGetStatus	XSemNpiStatus * NpiStatusInfo
XStatus	XSem_RegisterEvent	XIpiPsu * IpiInst XSem_Notifier * Notifier

Functions

XSem_CmdCfrInit

This function is used to initialize CRAM scan from user application. Primarily this function sends an IPI request to PLM to start CRAM Scan Initialization, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_CmdCfrInit(XIpiPsu *IpiInst, XSemIpiResp *Resp);
```

Parameters

The following table lists the `XSem_CmdCfrInit` function arguments.

Table 8: XSem_CmdCfrInit Arguments

Type	Name	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemIpiResp *	Resp	Structure Pointer of IPI response. <ul style="list-style-type: none"> Resp->RespMsg1: Acknowledgment ID of CRAM Initialization Resp->RespMsg2: Status of CRAM Initialization

Returns

This API returns the success or failure.

- XST_FAILURE: On CRAM Initialization failure
- XST_SUCCESS: On CRAM Initialization success

XSem_CmdCfrStartScan

This function is used to start CRAM scan from user application. Primarily this function sends an IPI request to PLM to invoke SEM CRAM StartScan, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_CmdCfrStartScan(XIpiPsu *IpiInst, XSemIpiResp *Resp);
```

Parameters

The following table lists the `XSem_CmdCfrStartScan` function arguments.

Table 9: XSem_CmdCfrStartScan Arguments

Type	Name	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemIpiResp *	Resp	Structure Pointer of IPI response. <ul style="list-style-type: none"> Resp->RespMsg1: Acknowledgment ID of CRAM start scan Resp->RespMsg2: Status of CRAM start scan

Returns

This API returns the success or failure.

- XST_FAILURE: On CRAM start scan failure
- XST_SUCCESS: On CRAM start scan success

XSem_CmdCfrStopScan

This function is used to stop CRAM scan from user application. Primarily this function sends an IPI request to PLM to invoke SEM CRAM StopScan, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_CmdCfrStopScan(XIpiPsu *IpiInst, XSemIpiResp *Resp);
```

Parameters

The following table lists the `XSem_CmdCfrStopScan` function arguments.

Table 10: XSem_CmdCfrStopScan Arguments

Type	Name	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemIpiResp *	Resp	Structure Pointer of IPI response. <ul style="list-style-type: none"> Resp->RespMsg1: Acknowledgment ID of CRAM stop scan Resp->RespMsg2: Status of CRAM stop scan

Returns

This API returns the success or failure.

- XST_FAILURE: On CRAM stop scan failure
- XST_SUCCESS: On CRAM stop scan success

XSem_CmdCfrNjctErr

This function is used to inject an error at a valid location in CRAM from user application. Primarily this function sends an IPI request to PLM to perform error injection in CRAM with user provided arguments in *ErrDetail, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_CmdCfrNjctErr(XIpiPsu *IpiInst, XSemCfrErrInjData *ErrDetail,
XSemIpiResp *Resp);
```

Parameters

The following table lists the XSem_CmdCfrNjctErr function arguments.

Table 11: XSem_CmdCfrNjctErr Arguments

Type	Name	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemCfrErrInjData *	ErrDetail	Structure Pointer with Error Injection details
XSemIpiResp *	Resp	Structure Pointer of IPI response. <ul style="list-style-type: none"> • Resp->RespMsg1: Acknowledgment ID of CRAM error injection • Resp->RespMsg2: Status of CRAM error injection

Returns

This API returns the success or failure.

- XST_FAILURE: On CRAM error injection failure
- XST_SUCCESS: On CRAM error injection success

XSem_CmdCfrGetStatus

This function is used to read all CRAM Status registers from PMC RAM and send to user application.

Prototype

```
XStatus XSem_CmdCfrGetStatus(XSemCfrStatus *CfrStatusInfo);
```

Parameters

The following table lists the `XSem_CmdCfrGetStatus` function arguments.

Table 12: XSem_CmdCfrGetStatus Arguments

Type	Name	Description
XSemCfrStatus *	CfrStatusInfo	<p>Structure Pointer with CRAM Status details</p> <ul style="list-style-type: none"> CfrStatusInfo->Status: Provides details about CRAM scan <ul style="list-style-type: none"> Bit [31-25]: Reserved Bit [24:20]: CRAM Error codes <ul style="list-style-type: none"> 00001: Unexpected CRC error when CRAM is not in observation state 00010: Unexpected ECC error when CRAM is not in Observation or Initialization state 00011: Safety write error in SEU handler 00100: ECC/CRC ISR not found in any row 00101: CRAM Initialization is not done 00110: CRAM Start Scan failure 00111: CRAM Stop Scan failure 01000: Invalid Row for Error Injection 01001: Invalid QWord for Error Injection 01010: Invalid Bit for Error Injection 01011: Invalid Frame Address for Error Injection 01100: Unexpected Bit flip during Error Injection 01101: Masked Bit during Injection 01110: Invalid Block Type for Error Injection 01111: CRC or Uncorrectable Error is active in CRAM 10000: ECC or CRC Error detected during CRAM Calibration in case of SWECC Bit [19-17]: Reserved Bit [16]: CRAM Initialization is completed Bit [15-14]: CRAM Correctable ECC error status <ul style="list-style-type: none"> 00: No Correctable error encountered 01: Correctable error detected and corrected 10: Correctable error detected but not corrected (Correction is disabled) 11: Reserved Bit [13]: CRAM Scan internal error Bit [12]: CRAM Invalid Error Location detected Bit [11]: CRAM Correctable ECC error detected Bit [10]: CRAM CRC error detected Bit [09]: CRAM Uncorrectable ECC error detected Bit [08]: CRAM ECC error during Initialization Bit [07]: CRAM Calibration Timeout error Bit [06]: CRAM Fatal/Error State Bit [05]: CRAM Error Injection State Bit [04]: CRAM Idle State Bit [03]: CRAM Correction State Bit [02]: CRAM Observation State Bit [01]: CRAM Initialization State Bit [00]: CRAM Scan is included in design CfrStatusInfo->ErrAddrL: This stores the low address of the last 7 corrected error details if correction is enabled in design. CfrStatusInfo->ErrAddrH: This stores the high address of the last 7 corrected error details if correction is enabled in design. CfrStatusInfo->ErrCorCnt: Counter value of Correctable Error Bits

Returns

This API returns the success or failure.

- XST_FAILURE: If NULL pointer reference of CfrStatusInfo
- XST_SUCCESS: On successful read from PMC RAM

XSem_CmdNpiStartScan

This function is used to start NPI scan from user application. Primarily this function sends an IPI request to PLM to invoke SEM NPI StartScan, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_CmdNpiStartScan(XIpiPsu *IpiInst, XSemIpiResp *Resp);
```

Parameters

The following table lists the XSem_CmdNpiStartScan function arguments.

Table 13: XSem_CmdNpiStartScan Arguments

Type	Name	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemIpiResp *	Resp	Structure Pointer of IPI response. <ul style="list-style-type: none"> • Resp->RespMsg1: Acknowledgment ID of NPI start scan • Resp->RespMsg2: Status of NPI start scan

Returns

This API returns the success or failure.

- XST_FAILURE: On NPI start scan failure
- XST_SUCCESS: On NPI start scan success

XSem_CmdNpiStopScan

This function is used to stop NPI scan from user application. Primarily this function sends an IPI request to PLM to invoke SEM NPI StopScan, waits for PLM to process the request and reads the response message.

Prototype

```
XStatus XSem_CmdNpiStopScan(XIpiPsu *IpiInst, XSemIpiResp *Resp);
```

Parameters

The following table lists the `XSem_CmdNpiStopScan` function arguments.

Table 14: XSem_CmdNpiStopScan Arguments

Type	Name	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSemIpiResp *	Resp	Structure Pointer of IPI response. <ul style="list-style-type: none"> Resp->RespMsg1: Acknowledgment ID of NPI stop scan Resp->RespMsg2: Status of NPI stop scan

Returns

This API returns the success or failure.

- XST_FAILURE: On NPI stop scan failure
- XST_SUCCESS: On NPI stop scan success

XSem_CmdNpiInjectError

This function is used to inject SHA error in NPI descriptor list (in the first NPI descriptor) from user application. Primarily this function sends an IPI request to PLM to invoke SEM NPI ErrorInject, waits for PLM to process the request and reads the response message.

Note: The caller shall invoke this `XSem_CmdNpiInjectError` function again to correct the injected error in NPI descriptor.

Prototype

```
XStatus XSem_CmdNpiInjectError(XIpiPsu *IpiInst, XSemIpiResp *Resp);
```

Parameters

The following table lists the `XSem_CmdNpiInjectError` function arguments.

Table 15: XSem_CmdNpiInjectError Arguments

Type	Name	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance

Table 15: XSem_CmdNpiInjectError Arguments (cont'd)

Type	Name	Description
<code>XSemIpiResp *</code>	Resp	Structure Pointer of IPI response. <ul style="list-style-type: none"> Resp->RespMsg1: Acknowledgment ID of NPI error injection Resp->RespMsg2: Status of NPI error injection

Returns

This API returns the success or failure.

- XST_FAILURE: On NPI error injection failure
- XST_SUCCESS: On NPI error injection success

XSem_CmdNpiGetStatus

This function is used to read all NPI Status registers from PMC RAM and send to user application.

Prototype

```
XStatus XSem_CmdNpiGetStatus(XSemNpiStatus *NpiStatusInfo);
```

Parameters

The following table lists the XSem_CmdNpiGetStatus function arguments.

Table 16: XSem_CmdNpiGetStatus Arguments

Type	Name	Description
<code>XSemNpiStatus *</code>	<code>NpiStatusInfo</code>	<p>Structure Pointer with NPI Status details</p> <ul style="list-style-type: none"> <code>NpiStatusInfo->Status</code>: Provides details about NPI scan <ul style="list-style-type: none"> Bit [31]: Cryptographic acceleration blocks are disabled for export compliance Bit [30]: NPI periodic scan missed error Bit [29-26]: Reserved Bit [25]: NPI GPIO write failure Bit [24]: NPI SHA engine failure Bit [23]: NPI Safety register write failure Bit [22]: NPI GT slave arbitration failure Bit [21]: NPI DDRMC Main slave arbitration failure Bit [20]: NPI Slave Address is not valid Bit [19]: NPI Descriptor SHA header is not valid Bit [18]: NPI Descriptor header is not valid Bit [17]: NPI SHA mismatch error is detected during scan Bit [16]: NPI SHA mismatch error is detected in first scan Bit [15-12]: Reserved Bit [11]: NPI periodic scan is enabled Bit [10]: NPI scan is suspended Bit [09]: NPI completed the first scan Bit [08]: NPI Scan is included in design Bit [07-06]: Reserved Bit [05]: NPI Internal Error State Bit [04]: NPI SHA mismatch Error State Bit [03]: NPI SHA Error Injection State Bit [02]: NPI Scan State Bit [01]: NPI Initialization State Bit [00]: NPI Idle State <code>NpiStatusInfo->SlvSkipCnt</code>: Provides NPI descriptor slave skip counter value if arbitration failure. This is 8 words result to accommodate 32 1-Byte skip counters for individual slaves arbitration failures. Slaves can be DDRMC Main, GT for which arbitration is required before performing scanning. <code>NpiStatusInfo->ScanCnt</code>: NPI scan counter value. This counter represents number of periodic scan cycle completion. <code>NpiStatusInfo->HbCnt</code>: NPI heartbeat counter value. This counter represents number of scanned descriptor slaves. <code>NpiStatusInfo->ErrInfo</code>: NPI scan error information if SHA mismatch is detected. This is 2 word information. <ul style="list-style-type: none"> Word 0: Node ID of descriptor for which SHA mismatch is detected Word 1 Bit [15-8]: NPI descriptor index number Word 1 Bit [7-0]: NPI Slave Skip count Index

Returns

This API returns the success or failure.

- XST_FAILURE: If NULL pointer reference of NpiStatusInfo
- XST_SUCCESS: On successful read from PMC RAM

XSem_RegisterEvent

This function is used to register/un-register event notification with XiSEM Server. Primarily this function sends an IPI request to PLM to invoke SEM Event Notifier registration, waits for PLM to process the request and check the status.

Note: The caller shall initialize the notifier object before invoking the XSem_RegisterEvent function.

Prototype

```
XStatus XSem_RegisterEvent(XIpiPsu *IpiInst, XSem_Notifier *Notifier);
```

Parameters

The following table lists the XSem_RegisterEvent function arguments.

Table 17: XSem_RegisterEvent Arguments

Type	Name	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance
XSem_Notifier *	Notifier	Pointer of the notifier object to be associated with the requested notification

Returns

This API returns the success or failure.

- XST_FAILURE: On event registration/un-registration failure
- XST_SUCCESS: On event registration/un-registration success

Data Structure Index

The following is a list of data structures:

- [XSemCfrErrInjData](#)
- [XSemCfrStatus](#)
- [XSemIpiResp](#)
- [XSemNpiStatus](#)
- [XSem_Notifier](#)
- [XSem_XmpuCfg](#)

XSem_Notifier

This structure contains details of event notification to be registered with the XiISEM server.

Declaration

```
typedef struct
{
    u32 Module,
    u32 Event,
    u32 Flag
} XSem_Notifier;
```

Table 18: Structure XSem_Notifier member description

Member	Description
Module	<p>The module information to receive notifications. Module can be either CRAM or NPI.</p> <ul style="list-style-type: none"> • For CRAM: use XSEM_NOTIFY_CRAM • For NPI: use XSEM_NOTIFY_NPI

Table 18: Structure XSem_Notifier member description (cont'd)

Member	Description
Event	<p>The event types specify the specific event registration. For CRAM module, events are:</p> <ul style="list-style-type: none"> Uncorrectable ECC error: use XSEM_EVENT_CRAM_UNCOR_ECC_ERR Uncorrectable CRC error: use XSEM_EVENT_CRAM_CRC_ERR Internal error: use XSEM_EVENT_CRAM_INT_ERR Correctable ECC error: use XSEM_EVENT_CRAM_COR_ECC_ERR <p>For NPI module, events are:</p> <ul style="list-style-type: none"> Uncorrectable CRC error: use XSEM_EVENT_NPI_CRC_ERR Internal error: use XSEM_EVENT_NPI_INT_ERR
Flag	<p>Event flags to enable or disable notification.</p> <ul style="list-style-type: none"> To enable event notification: use XSEM_EVENT_ENABLE To disable event notification: use XSEM_EVENT_DISABLE

XSem_XmpuCfg

XMPU Config Database

Declaration

```
typedef struct
{
    u32 DdrmcNocAddr,
    u32 XmpuRegionNum,
    u32 XmpuConfig,
    u32 XmpuStartAddrLo,
    u32 XmpuStartAddrUp,
    u32 XmpuEndAddrLo,
    u32 XmpuEndAddrUp,
    u32 XmpuMaster
} XSem_XmpuCfg;
```

Table 19: Structure XSem_XmpuCfg member description

Member	Description
DdrmcNocAddr	DDRMC NOC address
XmpuRegionNum	XMPU region number
XmpuConfig	XMPU entry config
XmpuStartAddrLo	XMPU start address lower portion
XmpuStartAddrUp	XMPU start address upper portion

Table 19: Structure XSem_XmpuCfg member description (cont'd)

Member	Description
XmpuEndAddrLo	XMPU end address lower portion
XmpuEndAddrUp	XMPU end address upper portion
XmpuMaster	XMPU master ID and mask

XSemCfrErrInjData

CRAM Error Injection structure to hold the Error Injection Location details for CRAM.

- Frame address
- Quad Word in a Frame, Range from 0 to 24
- Bit Position in a Quad Word, Range from 0 to 127
- Row Number, Range can be found from CFU_APB_CFU_ROW_RANGE register

Declaration

```
typedef struct
{
    u32 Efar,
    u32 Qword,
    u32 Bit,
    u32 Row
} XSemCfrErrInjData;
```

Table 20: Structure XSemCfrErrInjData member description

Member	Description
Efar	Frame Address
Qword	Quad Word 0...24
Bit	Bit Position 0...127
Row	Row Number

XSemCfrStatus

CRAM Status structure to store the data read from PMC RAM registers. This structure provides:

- CRAM scan state information
- The low address of last 7 corrected error details if correction is enabled in design
- The high address of last 7 corrected error details if correction is enabled in design

- CRAM corrected error bits count value

Declaration

```
typedef struct
{
    u32 Status,
    u32 ErrAddrL[MAX_CRAMERR_REGISTER_CNT],
    u32 ErrAddrH[MAX_CRAMERR_REGISTER_CNT],
    u32 ErrCorCnt
} XSemCfrStatus;
```

Table 21: Structure XSemCfrStatus member description

Member	Description
Status	CRAM Status
ErrAddrL	Error Low register L0...L6
ErrAddrH	Error High register H0...H6
ErrCorCnt	Count of correctable errors

XSemIpiResp

IPI Response Data structure.

Declaration

```
typedef struct
{
    u32 RespMsg1,
    u32 RespMsg2,
    u32 RespMsg3,
    u32 RespMsg4,
    u32 RespMsg5,
    u32 RespMsg6,
    u32 RespMsg7
} XSemIpiResp;
```

Table 22: Structure XSemIpiResp member description

Member	Description
RespMsg1	Response word 1 (Ack Header)
RespMsg2	Response word 2
RespMsg3	Response word 3
RespMsg4	Response word 4
RespMsg5	Response word 5
RespMsg6	Response word 6
RespMsg7	Response word 7 (Reserved for CRC)

XSemNpiStatus

NPI Status structure to store the data read from PMC RAM registers. This structure provides:

- NPI scan status information
- NPI descriptor slave skip counter value if arbitration failure
- NPI scan counter value
- NPI heartbeat counter value
- NPI scan error information if SHA mismatch is detected

Declaration

```
typedef struct
{
    u32 Status,
    u32 SlvSkipCnt[MAX_NPI_SLV_SKIP_CNT],
    u32 ScanCnt,
    u32 HbCnt,
    u32 ErrInfo[MAX_NPI_ERR_INFO_CNT]
} XSemNpiStatus;
```

Table 23: Structure XSemNpiStatus member description

Member	Description
Status	NPI Status
SlvSkipCnt	Slave Skip Count
ScanCnt	Scan Count
HbCnt	Heart Beat Count
ErrInfo	Error Information when SHA mismatch occurs

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

Documentation Navigator and Design Hubs

Xilinx[®] Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado[®] IDE, select **Help** → **Documentation and Tutorials**.
- On Windows, select **Start** → **All Programs** → **Xilinx Design Tools** → **DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

Note: For more information on DocNav, see the [Documentation Navigator](#) page on the Xilinx website.

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2020-2021 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.