

Application Note

F29x Error Handling and Debug Guide



Prarthan Bhatt

ABSTRACT

This application note focuses on providing overview of the error handling architecture and guidance on how to efficiently debug error events. F29x device architecture provides systematic error handling management focusing on functional safety for various end applications. Error Aggregator Module (EAM) and Error Signaling Module (ESM) provides error aggregation, logging and configurable response for all error events throughout the device. The application note provides guidance on debugging the error source using the tools and error logging provided by EAM and ESM.

Table of Contents

1 Introduction.....	2
2 Error Handling Architecture Overview.....	2
3 Example Overview.....	3
4 Error Aggregator Overview.....	3
4.1 Error Aggregation.....	3
4.2 Error Logging.....	5
4.3 Error Debugging Using EAM Module.....	5
5 Error Signaling Module Overview.....	9
5.1 ESM Error Event Output Configuration and Status Information.....	10
5.2 ESM Error Events Debugging.....	13
5.3 Miscellaneous Debug Tips for ESM.....	14
6 BootROM EAM and ESM Error Status.....	15
7 FAQ's:.....	16
8 Summary.....	17
9 References.....	18

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

For functional safety critical development, manage both systematic and random faults. F29x device architecture has built-in hardware safety mechanisms that provides systematic management for all the error events across the device. To begin, understand the error handling architecture at high level and then deep dive on how to interpret the error logs and configure response for each error event.

This application note first provides an overview of the error handling architecture, then explains the EAM and ESM features and tools in detail with an example and concludes with frequently asked questions and error debugging tips.

Table 1-1 lists the terms and abbreviations used in this application note.

Table 1-1. Terms and Abbreviations Used and Their Explanations

Terms or Abbreviations	Explanation
Systematic Error	Systematic faults result from an inadequacy in the design, development or manufacturing process and typically stem from gaps in the development process. Refer to safety standard like ISO 26262 for more information
Random Error	A random error is a hardware fault that occurs unpredictably during a component's operational lifetime. Unlike systematic errors, which are deterministic and result from design flaws, random errors are statistical and are managed through probabilistic analysis.
System Address	Each MCU has a unique memory map that defines the address range for each component. A system address is an address that falls within this map and corresponds to a specific resource
EAM	Error Aggregator Module
ESM	Error Signaling Module
NMI	Non-maskable Interrupt
CCS	Code Composer Studio

2 Error Handling Architecture Overview

The Figure 2-1 shows at a high level how the error handling is done. Error is detected at source which can be a peripheral, module, memory, interconnect or processing unit and this propagates to EAM for error aggregation and then passed on to ESM for user configurable error response within the device. Critical device error events that need error aggregation and logging go through EAM whereas all other device error events are passed on directly from the error source to ESM and are listed in the Error Events table in the ESM Chapter in [F29x Technical Reference Manual](#).

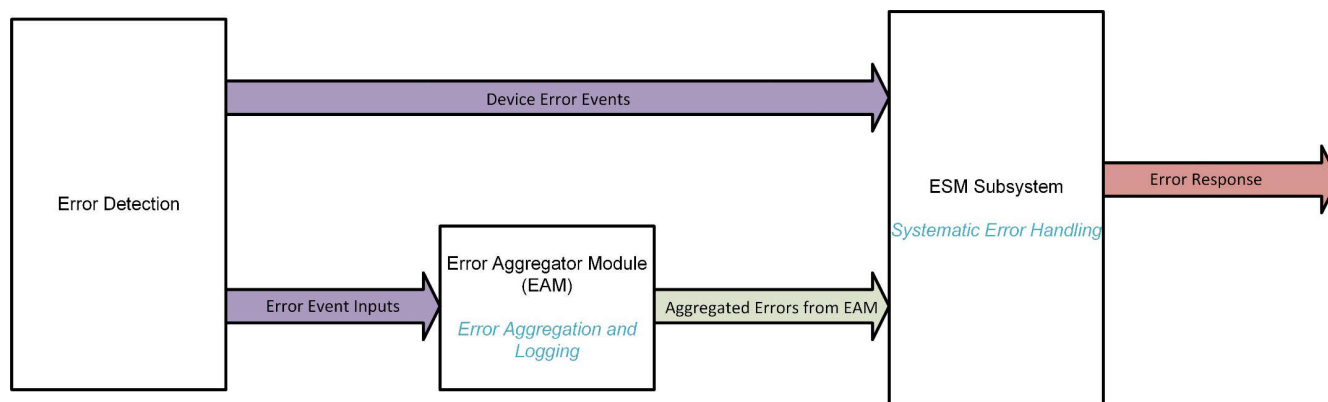


Figure 2-1. Device Error Handling Architecture

3 Example Overview

The EAM and ESM functionality is understood in the below sections with an example taking ESM and EAM components separately one at a time. This application note uses the [F29 SDK ESM multicore example](#) from F29x SDK which is multicore example (CPU1 and CPU3). CPU3 application code has write to M0RAM location which causes CPU3 DW bus security violation error, example showcases how to handle this error using EAM and ESM.

This application note takes this example and shows how an error can be debugged using tools.

4 Error Aggregator Overview

Error aggregator module (EAM) is an interface between **ESM** and **critical modules** that generate errors like C29x CPU, PIPE, RTDMA, Memory controllers, Peripheral bridges and Read Interfaces. EAM provides error logging and aggregation that is necessary for similar type of errors to reduce the number of errors passed to ESM.

The device contains below EAM module's (where x is from 1 to 3 and y is from 1 to 2) :

1. CPUx PR Error Aggregator - Aggregates errors occurred during CPU program fetch access
2. CPUx DR1 Error Aggregator - Aggregates errors occurred during CPU Data Read access on DR1 port
3. CPUx DR2 Error Aggregator - Aggregates errors occurred during CPU Data Read access on DR2 port
4. CPUx DW Error Aggregator - Aggregates errors occurred during CPU data write access
5. CPUx INT Error Aggregator - Aggregates interrupt related errors from CPU and associated PIPE module
6. RTDMAy DR Error Aggregator - Aggregates errors occurred during RTDMA data read access
7. RTDMAy DW Error Aggregator - Aggregates errors occurred during RTDMA data write access
8. SSU Error Aggregator - Aggregates errors sent out by SSU module

For detailed view of the EAM modules refer to the Error Aggregator chapter in [F29x Technical Reference Manual](#).

The following sections showcases error aggregation, error logging and interpretation of the error flag registers with an example.

4.1 Error Aggregation

C29x CPU has 4 buses – CPU DR1 (Data Read bus 1), DR2 (Data Read bus 2), DW (Data Write bus) and PR (Program fetch/read bus). Error originating from each bus is detected and captured/logged separately in respective EAM error flag registers for isolating the error source.

In addition to error aggregation, the error events are also segregated into **low priority** errors and **high priority** errors. In this example, the error events across all four buses are first segregated into two categories – low priority and high priority based on **severity** of error and then aggregated. The aggregated outputs – low priority and high priority error events are then passed to ESM. Error priority is **pre-defined** in the device depending on severity, refer to Error Aggregator Chapter in [F29x Technical Reference Manual](#) to know more about error priority for all errors captured in EAM.

Each error has error type value and fixed pre-defined priority assigned to this as shown in the table below taking CPU PR bus as an example.

An example shown in [Table 4-1](#) is for CPU PR bus. Single bit (Correctable error) and WARNPSP errors are classified as low priority errors and all other errors are classified as high priority errors. All high priority error type within CPU PR EAM have single aggregated output similarly for all low priority errors within CPU PR EAM there is one aggregated output.

Table 4-1. EAM CPU PR Error Type Priority

Error Type Value	CPUx PR Error	RAM, ROM, FRI – PR Error	Priority
0x01	Instruction fetch security violation. Instruction packet crossed LINK, STACK, ZONE boundary. Linear code crossed LINK, STACK, ZONE boundary. Regular Branch and Calls crossed STACK, ZONE boundary.	Reserved	High
0x02	Secure entry error	Reserved	High
0x04	Secure exit error	Reserved	High
0x08	MAX PSP error	Reserved	High
0x10	Access timeout error	Reserved	High
0x20	Access ACK error	Access ACK error	High
0x40	Uncorrectable error	Uncorrectable error	High
0x80	Correctable error	Reserved	Low
0x100	WARN PSP error	Reserved	Low
0x200	Software breakpoint error	Reserved	High
0x400	Illegal instruction error	Reserved	High
0x800	Instruction timeout error	Reserved	High

All high-priority errors from all CPU buses - CPU PR, DR1, DR2 and DW are also combined as CPU HPERR (high priority error) and sent to ESM. Similarly, all low-priority errors from CPU PR, DR1, DR2 and DW are combined as CPU LPERR (Low priority error) and sent to ESM. This is shown in [Figure 4-1](#).

Advantage for the aggregation is that this reduces the number of error events passed to ESM and the corresponding error response configuration in ESM for these error events (captured in the table above) to two error events (high priority and low priority) across all CPU buses. This is redundant to configure each error originating from each CPU bus separately especially when there are several such error events in the device. Hence the error across all the CPU buses are aggregated and provided to ESM. Along with aggregation of errors, segregation is also important since this enables users to configure ESM to generate appropriate action for low priority and high priority errors separately.

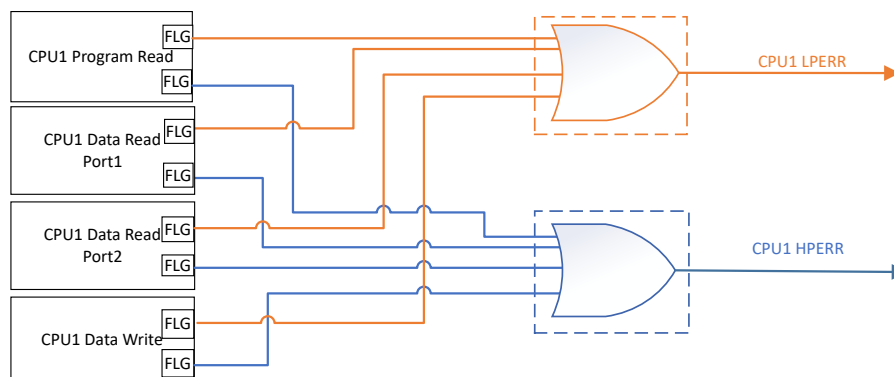


Figure 4-1. CPU1 EAM Modules Error Aggregation

Note

The EAM error type priority is not to be confused with ESM output priority. To know more about ESM output priority refer to ESM chapter in [F29x Technical Reference Manual](#).

4.2 Error Logging

The EAM module provides comprehensive error logging capabilities needed for user to debug the source of all critical errors in the device.

All errors are logged in EAM registers with the following information:

1. Error Type – Multibit value that maps to specific error type (for example Access acknowledge error, Uncorrectable error etc.). The value is pre-defined for each type, for example – 0x20 for Access ACK error on CPU PR error as shown in the table in Error Aggregation section above.
2. Error Address – System Address at which the error occurred used to debug the error origin. There are separate high priority and low priority error address registers.
3. Program Counter (Applicable only for CPU EAM modules) – Program counter address captured helps identify the source of error. The PC address is particularly useful to identify code corresponding to the program counter address that respective CPU was executing which caused the error.

4.3 Error Debugging Using EAM Module

To make error debugging easier, error handling functions are integrated in the Code Composer Studio (CCS) as shown in [Figure 4-2](#).

Using CCS Scripts Menu, user can find the error status captured. All the below steps mentioned below for error debug are done with the Error_Agg_Check_Status() hotmenu function in the GEL file executable from the Scripts Menu as shown in the image [Figure 4-2](#).

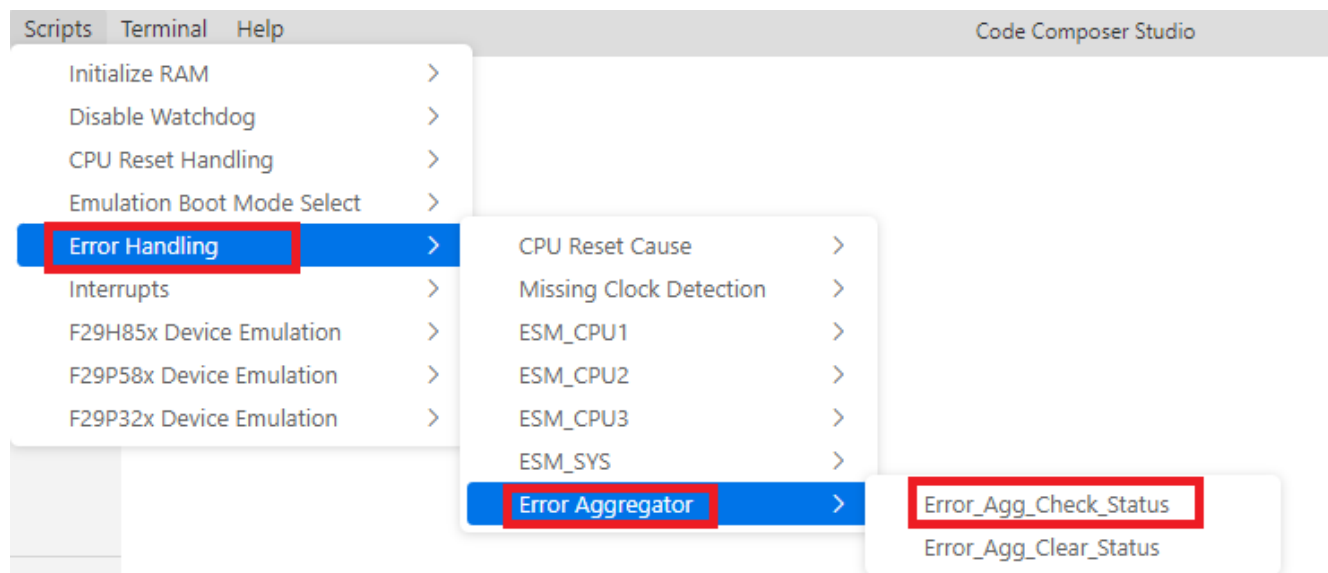


Figure 4-2. Error Aggregator GEL File Function

1. Check the error type register value for each EAM module (CPU PR/DR1/DR2/DW, RTDMA, SSU, CPU INT, Ethercat) if this has value other than 0x0 to identify if any error has occurred or not.
2. If particular EAM module error type register has a value other than 0x0 then find the corresponding error aggregator **low priority error address** for low priority error and **high priority error address** for high priority error.
3. If particular EAM module error type register has a value other than 0x0 then find the error aggregator **program counter** address – only in case of CPU EAM modules.

4.3.1 EAM Error Debugging

1. Run the Error_Agg_Check_Status() GEL file hotmenu function, [Figure 4-3](#) is the CCS GEL output for the ESM Multicore example (esm_ex1_cpu1_cpu3) from the [F29 SDK](#). The Error_Agg_Check_Status() function does the following for the output:
 - a. Error_Agg_Check_Status() maps the error type value to the error. In this example as seen in the GEL output, the error is CPU3 DW Security Violation error since in the example CPU3 code writes to M0RAM location which is not allowed. This was intentionally done to create an error scenario to showcase the error debug example.
 - b. High priority error address is 0x20000000 and Program counter is 0x10402C14 as seen from the GEL output below, this is fetched from the corresponding CPU3_DW high priority error address and program counter registers.

```

---
GEL Output x  Debug Output
C29xx_CPU1: Error Aggregator Status :
C29xx_CPU1: - CPU3_DW Errors (HP Error Addr = 0x20000000, LP Error Addr = 0x00000000, PC = 0x10402C14)
C29xx_CPU1: - SECURITY_VIO
C29xx_CPU1: Error Aggregator Status Done

```

Figure 4-3. Error Aggregator Check Status GEL Output Log

2. The Error_Agg_Check_Status() GEL function needs to be run before clearing the ESM/EAM flags from either CPU1 or CPU3 as shown below. To achieve this the breakpoint was placed before ESM/EAM flag clear function execution as shown in the NMI ISR – ESM/EAM Clear Flags figure below so that the EAM register can be read and decoded by GEL function before they are cleared.

Interrupt_clearEsmEaFlags() is the reference driverlib function provided as part of F29 SDK that clears all the EAM and ESM flags which is used in this example's CPU1 and CPU3 NMI ISR's (as seen in NMI ISR – ESM/EAM Clear Flags) and also present in driverlib default NMI handler. In case, the Interrupt_clearEsmEaFlags() function was already executed then user can look at the nmiStatus struct (memory location used to store the ESM/EAM error flag register values) to find the error information in the CCS watch window as shown in [Figure 4-5](#).

Note

The clearing of EAM and ESM flags is an important step in NMI (Non-Maskable Interrupt) ISR since that avoids NMIWD (NMI Watchdog) to timeout and trigger system reset (XRSn).

```

C esm_cpu1_cpu3_multi_c29x1.c x  C esm_cpu1_cpu3_multi_c29x3.c
esm_cpu1_cpu3_multi_c29x1 > C esm_cpu1_cpu3_multi_c29x1.c > myNMI_CPU1_ISR
140
141 void myNMI_CPU1_ISR(void)
142 {
143     cpu1nmigen = true;
144
145     //
146     // Clear the raw status and deassert the level interrupt.
147     //
148     Interrupt_clearEsmEaFlags(nmiStatus: &nmiStatus);
149

```

Figure 4-4. NMI ISR – ESM/EAM Clear Flags



Figure 4-5. NMI Status Capture log

3. Following is the register view output from CCS for the same example.
 - a. The register output high priority error address, program counter address and error type value matches with the GEL function file output shown in Error Aggregator Check Status GEL Output Log - [Figure 4-3](#).

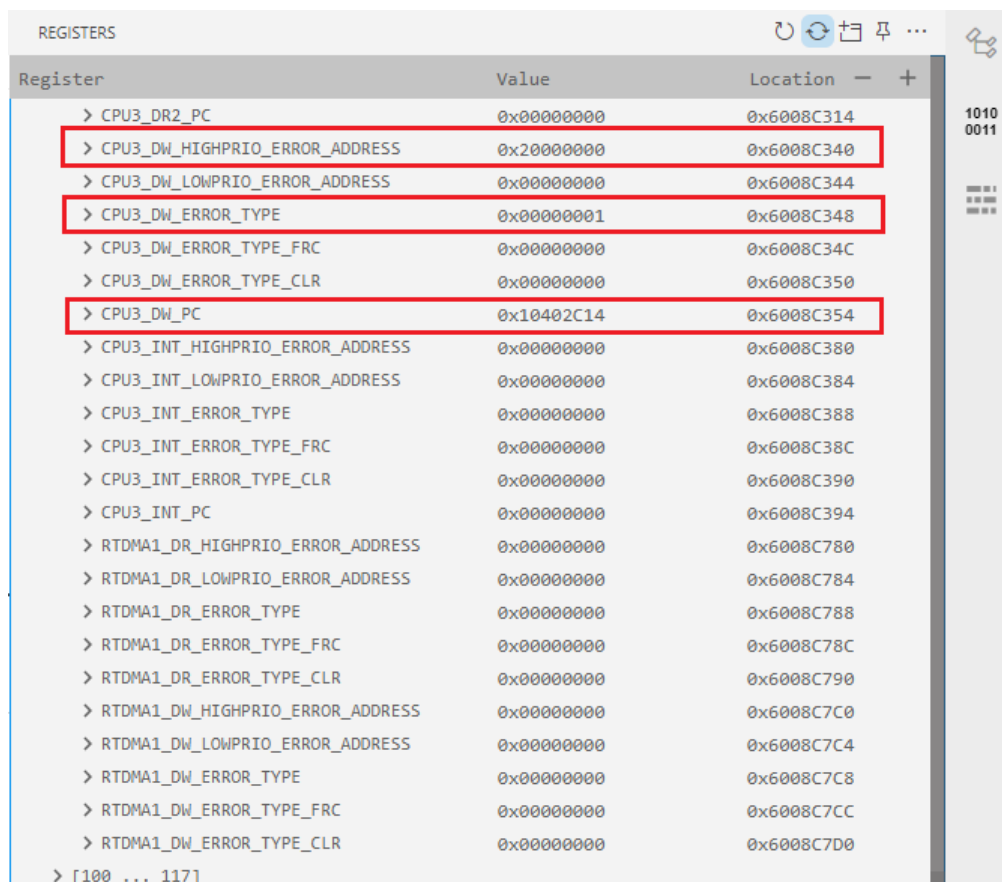


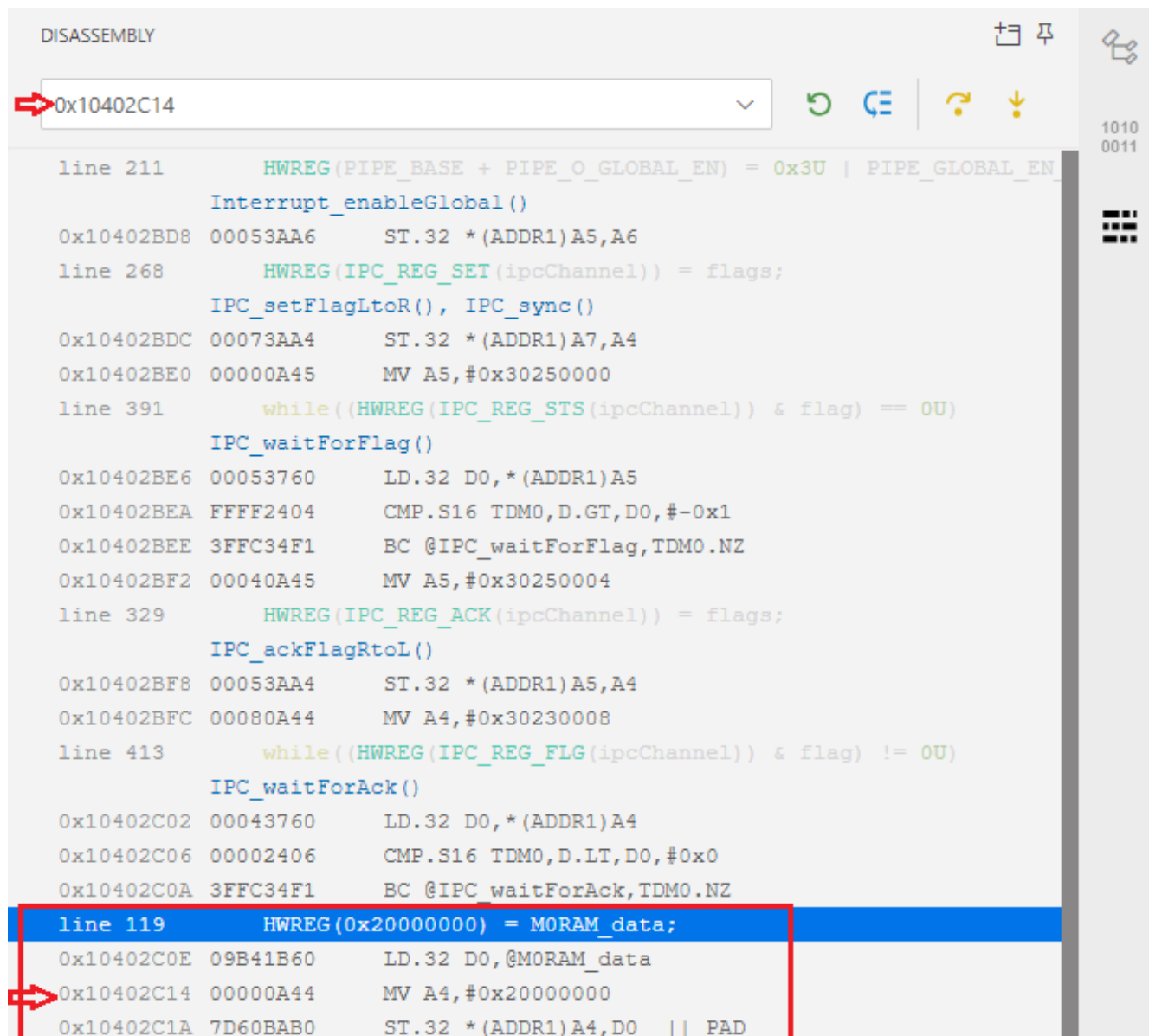
Figure 4-6. CCS Register View of Error Aggregator Registers

4.3.2 Interpreting Error Address and Program Counter Values

As explained in section 3.2, error address and program counter addresses can be used for debugging source of the error. This section showcases interpretation of error address and program counter taking esm multicore example (esm_ex1_cpu1_cpu3) from the [F29 SDK](#).

The program counter (PC) address can be copied to the CCS disassembly view to find the source code where the error occurred. For this example, when looking at the corresponding PC address CCS disassembly view for the EAM captured PC address (0x10402C14) as shown in [Figure 4-7](#), points to the data write operation to location 0x20000000 (M0 RAM) also logged in the High Priority Address register in EAM. Hence with the PC address and error address information, user can pin-point the issue to a specific CPU3 source code write operation to memory location that caused the error.

The error occurred on CPU3 DW (data write) bus which also matches the expected behavior from the code perspective since the CPU3 application code has write operation for the M0RAM_data to M0RAM which is not allowed from CPU3 code. CPU3 only has read data permission for M0RAM hence write operation in this case caused security violation error on CPU3 DW bus.



```

DISASSEMBLY

0x10402C14
line 211      HWREG(PIPE_BASE + PIPE_O_GLOBAL_EN) = 0x3U | PIPE_GLOBAL_EN;
              Interrupt_enableGlobal()
0x10402BD8 00053AA6      ST.32 *(ADDR1)A5,A6
line 268      HWREG(IPC_REG_SET(ipcChannel)) = flags;
              IPC_setFlagLtoR(), IPC_sync()
0x10402BDC 00073AA4      ST.32 *(ADDR1)A7,A4
0x10402BE0 00000A45      MV A5,#0x30250000
line 391      while((HWREG(IPC_REG_STS(ipcChannel)) & flag) == 0U)
              IPC_waitForFlag()
0x10402BE6 00053760      LD.32 D0,*(ADDR1)A5
0x10402BEA FFFF2404      CMP.S16 TDM0,D.GT,D0,#-0x1
0x10402BEE 3FFC34F1      BC @IPC_waitForFlag,TDM0.NZ
0x10402BF2 00040A45      MV A5,#0x30250004
line 329      HWREG(IPC_REG_ACK(ipcChannel)) = flags;
              IPC_ackFlagRtoL()
0x10402BF8 00053AA4      ST.32 *(ADDR1)A5,A4
0x10402BFC 00080A44      MV A4,#0x30230008
line 413      while((HWREG(IPC_REG_FLG(ipcChannel)) & flag) != 0U)
              IPC_waitForAck()
0x10402C02 00043760      LD.32 D0,*(ADDR1)A4
0x10402C06 00002406      CMP.S16 TDM0,D.LT,D0,#0x0
0x10402C0A 3FFC34F1      BC @IPC_waitForAck,TDM0.NZ
line 119      HWREG(0x20000000) = M0RAM_data;
0x10402C0E 09B41B60      LD.32 D0,@M0RAM_data
0x10402C14 00000A44      MV A4,#0x20000000
0x10402C1A 7D60BAB0      ST.32 *(ADDR1)A4,D0 || PAD

```

Figure 4-7. Disassembly View of Program Counter

Table 4-2. M0RAM Access from CPU3

Memory	Interleaved	CPU1	CPU2	CPU3	HSM	RTDMA1	RTDMA2
M0 RAM	Yes	0WS data (read and write)	0WS data (read-only)	3WS data (read-only)	-	-	-

5 Error Signaling Module Overview

The Error Signaling Module (ESM) provides systematic consolidation of responses to error events throughout the device into one location which is crucial for several safety critical applications.

ESM Subsystem contains following modules:

1. ESM CPU1 - Dedicated ESM module for output to CPU1
2. ESM CPU2 - Dedicated ESM module for output to CPU2
3. ESM CPU3 - Dedicated ESM module for output to CPU3
4. System ESM - Dedicated ESM module for system level outputs (mainly the ERRORSTS pin output, device reset, and integration to other modules using XBAR event outputs)

Figure 5-1 describes how the ESM subsystem integrates at the device level in detail, for more info refer to ESM chapter in the F29x TRM.

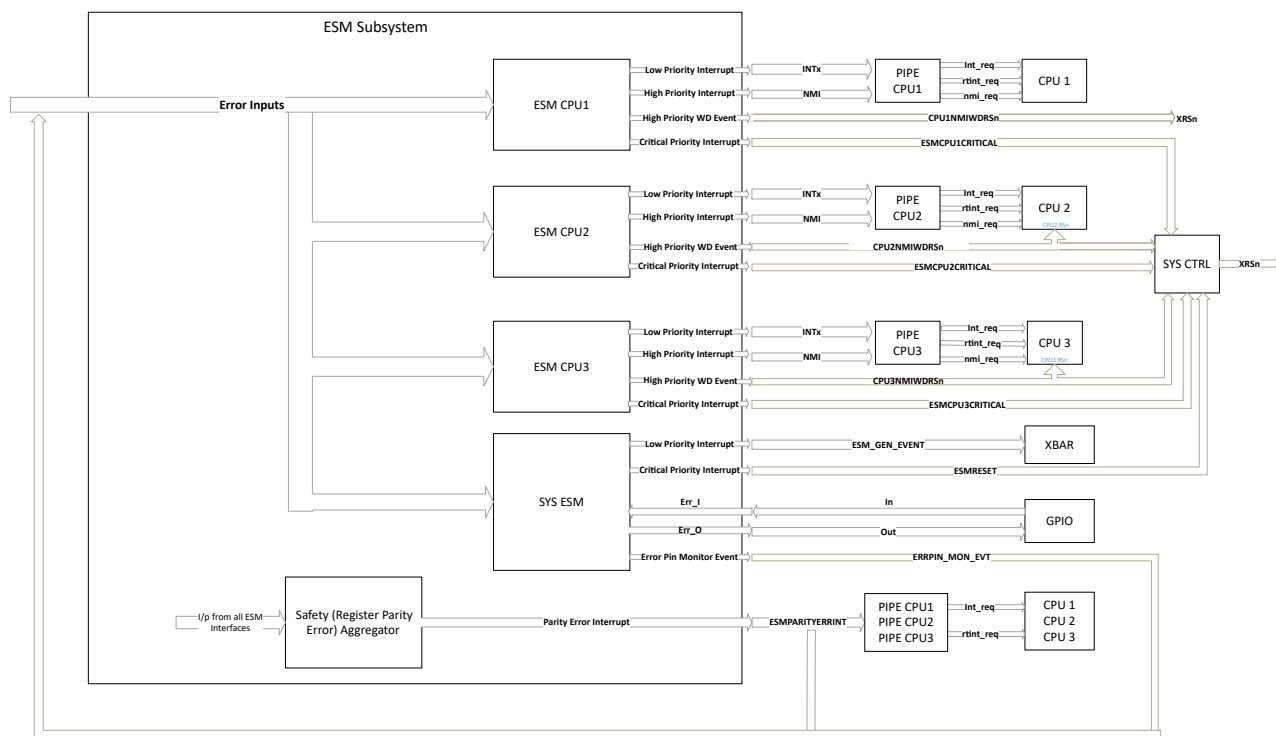


Figure 5-1. ESM Subsystem Integration Block Diagram

The ESM provides features to classify errors by severity and to provide programmable error response. Error Signaling Module provides a way to indicate error pin response, selectable interrupt priority response, or Non-Maskable interrupt (NMI) to CPU depending on severity of error encountered. The user is responsible to determine what error response is to be taken for each error event so that this is consistent with the system safety concept.

1. Interrupt to specific CPU:
 - a. Interrupt (INT or RTINT from PIPE to CPU) (Low priority interrupt output of ESM) - Generally selected for correctable or low severity errors encountered in the device or can be implemented for diagnostics

- outside the CPU. An interrupt allows events external to the CPU to generate a program sequence context transfer to an interrupt handler where software has an opportunity to manage the fault.
- b. Non-Maskable Interrupt (NMI) (High priority interrupt output of ESM) - Generally selected for uncorrectable or critical errors encountered in the device where error response is required to transfer context to NMI ISR and software has an opportunity to manage the fault and abort the operation safely.
2. Error Signaling Pin:
 - a. Error pin (ERRORSTS) action for external monitor like PMIC (Power management integrated circuits) to act for cases where required response is to generate an external error response.
 3. Resets
 - a. Respective CPU Reset (CPURSn): ESM is capable of generating reset to individual CPU to bring system in safe state upon detection of error in MCU.
 - b. Device Reset (XRSn): Upon detection of error, trigger device reset (XRSn) to bring MCU in safe state.

The section below gives a brief overview of **configurations** for the ESM CPU and System ESM modules for above outputs in ESM subsystem for more details refer to the ESM chapter in [F29x Technical Reference Manual](#) and device integration.

Overview and key points to know about error events in ESM:

1. Error events are common to all ESM module (ESM CPU1/2/3 and System ESM)
2. Each ESM module has separate **configuration** and **status** registers hence all ESM modules can work independently of each other allowing flexibility for different use cases. For example, an error event on the occurrence can be configured to output an interrupt to CPU1 from ESM CPU1 and **not** configured to output an interrupt to CPU3 from ESM CPU3 module.
3. Error events are divided in further groups of 32. F29x devices have total of 256 error events hence there are total 8 Groups.

Note

All Group0 Error Events are mapped to trigger NMI by default. Group0 error events are high priority aggregated CPU error outputs from EAM (Error Aggregator Module).

5.1 ESM Error Event Output Configuration and Status Information

Figure below shows how the respective ESM block can be configured to affect the available output from respective ESM modules. To see further how these outputs are connected to device peripherals refer to ESM subsystem device integration diagram in [F29x Technical Reference Manual](#) ESM chapter.

Status Registers listed below are useful in identifying which error event is active and enabled to influence the output from ESM CPU and Sys ESM module:

1. RAW Status/Set Register (RAW_j) - This indicates if the error event is active where j stands for error event index (j= 0 to 255).
2. Interrupt Enable Status/Clear Register (STS_j) – This indicates if the error event is active and enabled to influence either low priority or high priority interrupt where j stands for error event index (j= 0 to 255).

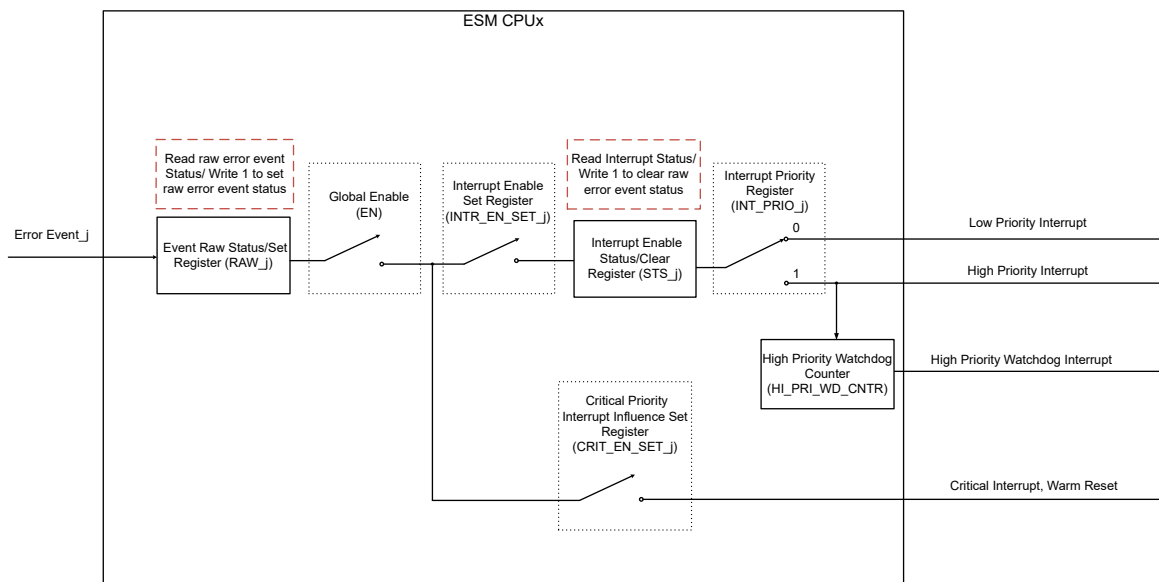


Figure 5-2. ESM CPU Detailed Configuration and Status Info View

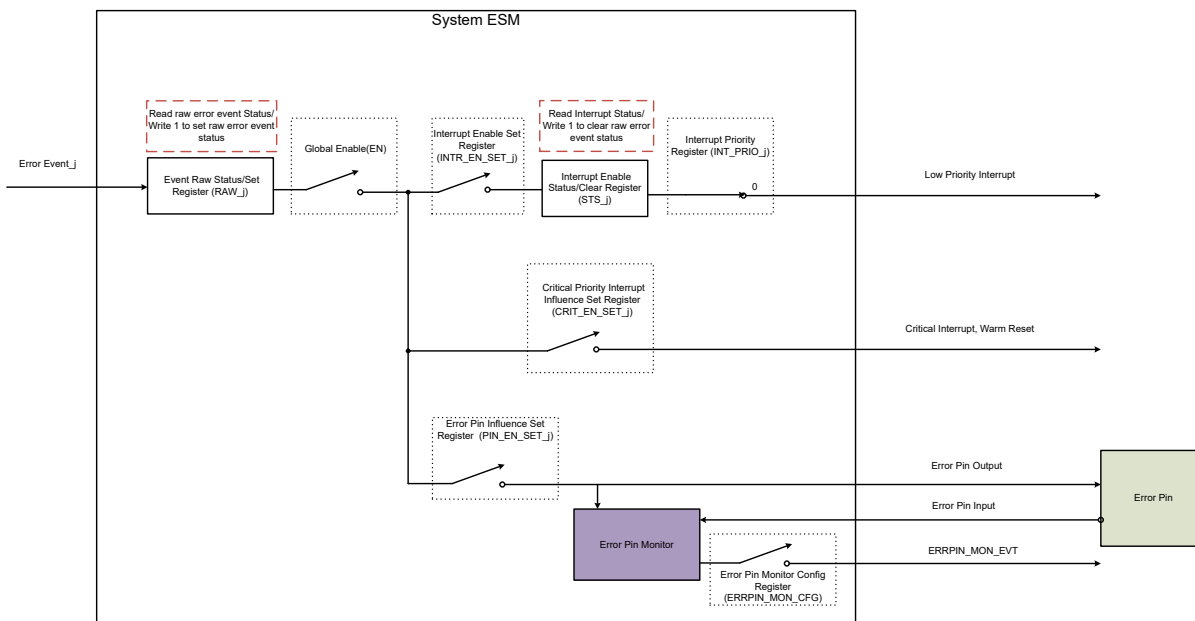


Figure 5-3. System ESM Detailed Configuration and Status Info View

5.1.1 Sysconfig ESM Configuration

Sysconfig supports configuring the individual error events for desired output from respective ESM module as described in [Section 5.1](#).

The [Figure 5-4](#) for example showcases how ESM CPU1 can be configured for error event - Error Aggregator CPU1 HPERR for high priority NMI output. Global parameters settings in respective ESM CPU Sysconfig module can be used to configure high priority watchdog enable, watchdog counter pre-load value as well as define interrupt handler configuration for low priority interrupt and NMI outputs.

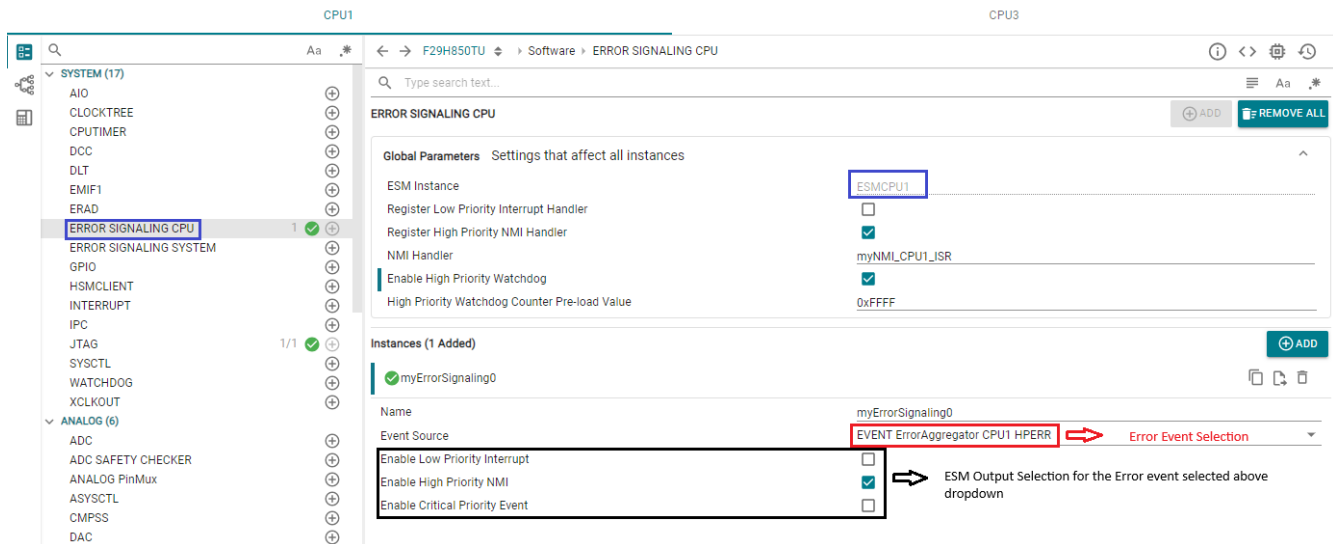


Figure 5-4. ESM CPU Sysconfig Module

The Figure 5-5 for example showcases how System ESM can be configured for error event - Error Aggregator CPU1 HPERR to enable influence on error pin. The error status pin (ERRORSTS) configuration like polarity, output pin mode configuration, and so on can also be done using the global parameters section in System ESM Sysconfig module.

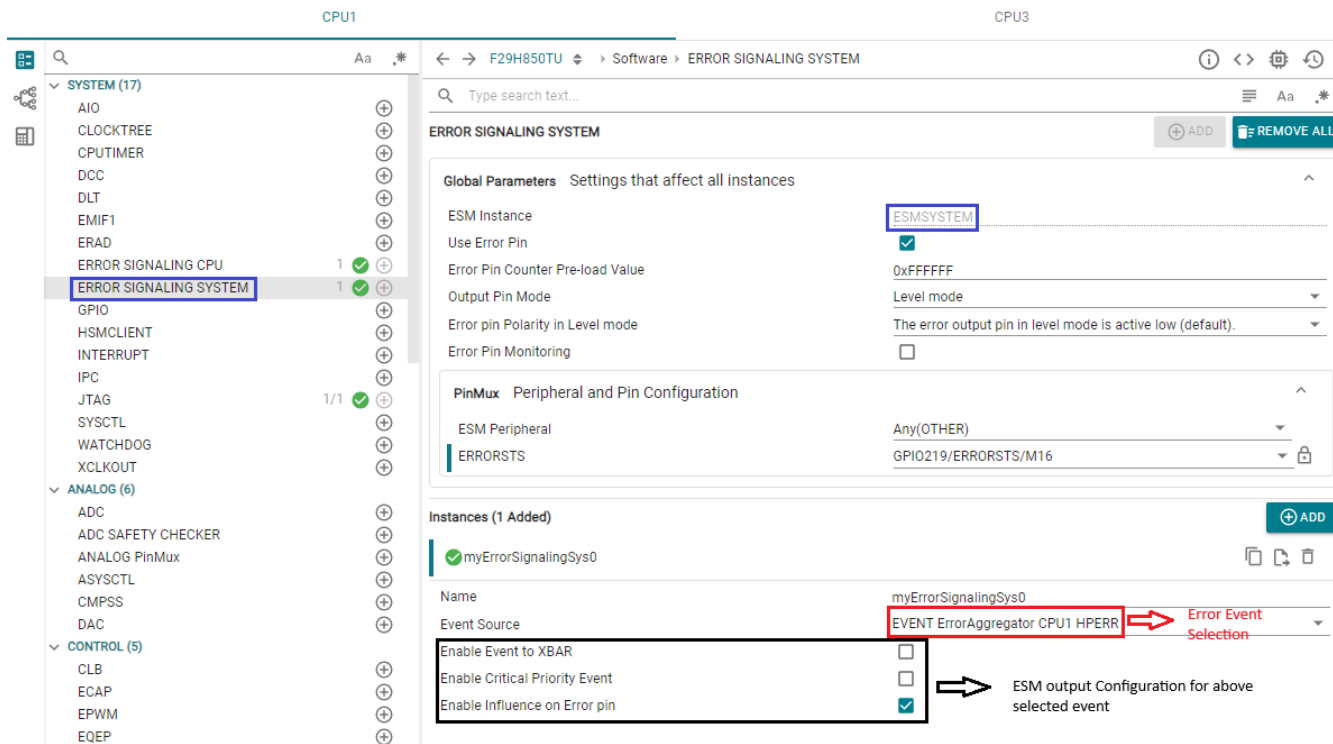


Figure 5-5. System ESM Sysconfig Module

5.2 ESM Error Events Debugging

Follow the below steps for error events debug :

1. Run the following scripts `ESM_CPU_Check_Status()` GEL file hot menu function from CCS to check status of the error events as shown in [Figure 5-6](#). This function output indicates error events separated in two categories:
 - a. **Active/Pending** Error Events – Indicates error events that are active/pending
 - i. When an error event is active means the raw status of the error event is set, the function checks for RAW status register (`RAW_j`) for each event stated in F29x TRM ESM error events table.
 - b. **Active, Pending, and Enabled** Error Events – Indicates error events that are active/pending and enabled.
 - i. When an error event is active, pending and enabled means that the RAW status is set for the error event as well as the Interrupt Enable Set Register is also set by the user to trigger interrupt output from respective ESM module.

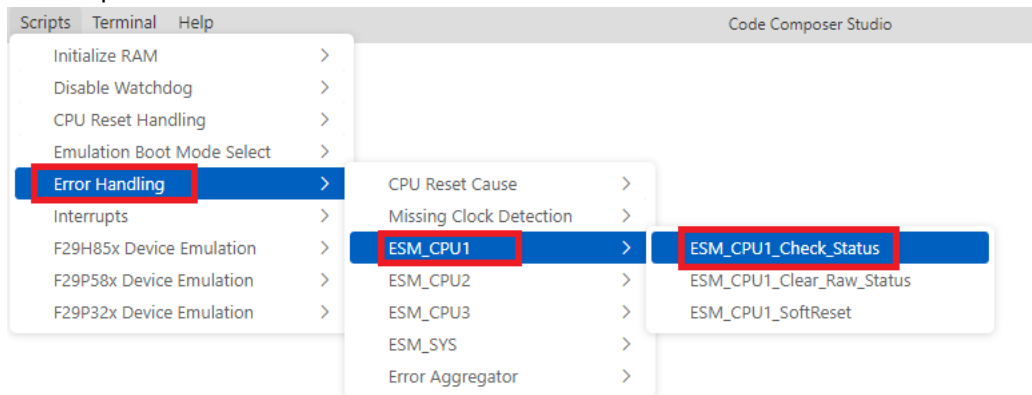
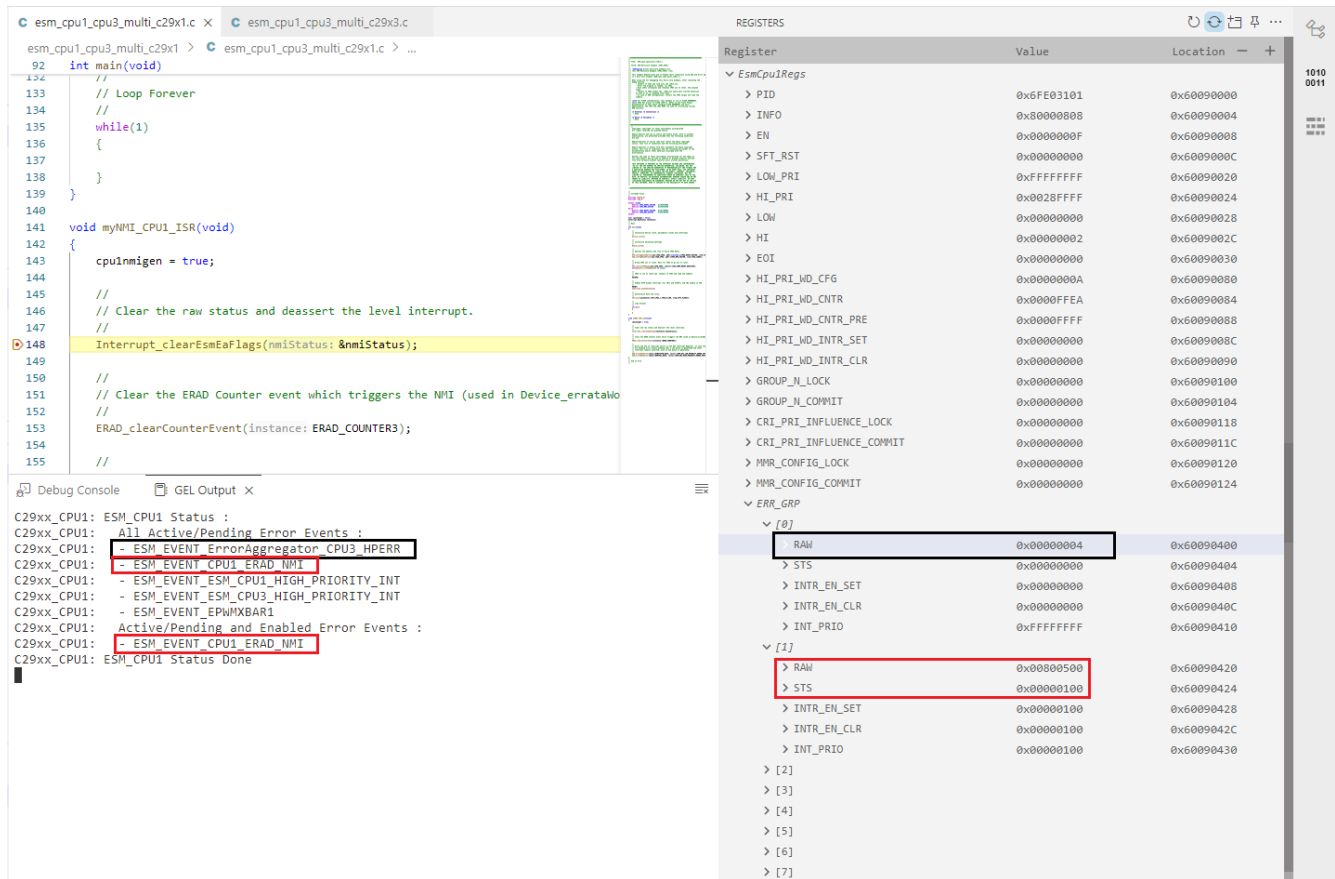


Figure 5-6. ESM Error Status GEL Function

2. Example output from the GEL output and correlation to the ESM registers is shown in the figure below. This is continuation of the same ESM Multicore example taken from F29 SDK.
 - a. [Figure 5-7](#) shows the CPU1_ERAD_NMI error event is both active and enabled. The ESM CPU1 - Interrupt priority register (`INT_PRIO`) is also set for the CPU1_ERAD_NMI error event to trigger NMI both for CPU1 (in ESM CPU1) and CPU3 (in ESM CPU3).
 - b. In addition to this there are other error events like ErrorAggregator_CPU3_HPERR (Error aggregator CPU3 high priority error from security violation error on CPU3 DW bus as explained in sections above), EPWMXBAR1, CPU1 High Priority interrupt and CPU3 High priority interrupt output also active which is decoded using the ESM RAW Status register (`RAW_j`) value. CPU1 and CPU3 high priority interrupt outputs are CPU1 and CPU3 NMI output flags whereas EPWM XBAR and ERAD NMI events are used in the NMI errata workaround implementation hence active as expected, check details in the F29x device errata.



The screenshot displays the TI Studio IDE with the ESM Error Event Status GEL Output. The left pane shows the C29xx_CPU1: ESM_CPU1 Status output, which lists active and pending error events. The right pane shows the EsmCpu1Regs register window, highlighting the RAW and STS registers.

Active/Pending Error Events:

- ESM_EVENT_ErrorAggregator_CPU3_HPERR
- ESM_EVENT_CPU1_ERAD_NMI

Active/Pending and Enabled Error Events:

- ESM_EVENT_CPU1_ERAD_NMI

Registers:

Register	Value	Location
RAW	0x00000004	0x60090400
STS	0x00000000	0x60090404

Figure 5-7. ESM Error Event Status GEL Output

- Similar to EAM register flags, check for the GEL output before RAW status register is cleared in NMI ISR or check the same structure (nmiStatus) as shown before where they are saved for debug later. Clearing ESM RAW status (RAW_j) register flags is necessary to avoid the NMIWD timeout in case particular event is configured to trigger NMI when active.

5.3 Miscellaneous Debug Tips for ESM

- Check RESC (Reset Cause) register to verify if the ESM High Priority Watchdog Interrupt (NMIWD) caused NMIWDRSn to occur. If System ESM is configured for Critical Priority Interrupt output for any error event check ESMRESET bit in RESC register.
- Check ESM HI_PRI Register which shows the highest priority outstanding high priority interrupt. The lowest event number has the highest priority, value of 0xFFFF indicates that there are no high priority interrupts active/pending.
- Check ESM LOW_PRI Register which shows highest priority outstanding low priority interrupt. The lowest event number has the highest priority, value of 0xFFFF indicates that there are no low priority interrupts active/pending.

6 BootROM EAM and ESM Error Status

When the application is not able to clear the error before a NMIWD (High Priority Watchdog) timeout, then a reset is triggered from ESM CPU1 instance. In this case, bootROM which runs after the device reset (XRSn) clears errors to avoid a back-to-back NMIWD rest loop and stores the error information and status to M0 RAM (refer to Table below) for further debug.

BootROM clears the following status:

1. ESM Group0 RAW Status for ESM CPU1 and System ESM instances of ESM-Subsystem
2. All CPUx error aggregator type registers

Also saves the following in M0 RAM for user to debug the source of error:

1. ESM RAW Status for Group0 only
2. Error Aggregator CPU1 - PR, DR1/2, DW, and INT instances error information including high-priority error address, low-priority error address, error type, and program counter registers

Table 6-1. BootROM Error Status Information

Description	Address
ESM RAW Status	0x2000_0868
CPU1 PR Error Aggregator High Priority Error address	0x2000_086C
CPU1 PR Error Aggregator Low Priority Error address	0x2000_0870
CPU1 PR Error Aggregator Error Type	0x2000_0874
CPU1 PR Error Aggregator PC value	0x2000_0878
CPU1 DR1 Error Aggregator High Priority Error address	0x2000_087C
CPU1 DR1 Error Aggregator Low Priority Error address	0x2000_0880
CPU1 DR1 Error Aggregator Error Type	0x2000_0884
CPU1 DR1 Error Aggregator PC value	0x2000_0888
CPU1 DR2 Error Aggregator High Priority Error address	0x2000_088C
CPU1 DR2 Error Aggregator Low Priority Error address	0x2000_0890
CPU1 DR2 Error Aggregator Error Type	0x2000_0894
CPU1 DR2 Error Aggregator PC value	0x2000_0898
CPU1 DW Error Aggregator High Priority Error address	0x2000_089C
CPU1 DW Error Aggregator Low Priority Error address	0x2000_08A0
CPU1 DW Error Aggregator Error Type	0x2000_08A4
CPU1 DW Error Aggregator PC value	0x2000_08A8
CPU1 INT Error Aggregator High Priority Error address	0x2000_08AC
CPU1 INT Error Aggregator Low Priority Error address	0x2000_08B0
CPU1 INT Error Aggregator Error Type	0x2000_08B4
CPU1 INT Error Aggregator PC value	0x2000_08B8

7 FAQ's:

1. Does the user setup and configure NMI ISR ?

Answer – Yes, this is recommended that users' setup NMI ISR during device initialization so that when high priority error occurs (for example Group0 error events) which trigger NMI to respective CPU can then gracefully clear the Error flags in EAM and ESM as per guidance in F29x TRM and ESM multicore F29 SDK example. Failure to clear ESM raw status flag causes NMIWD timeout to trigger XRSn (Device reset). User can check NMIWD bit in RESC (Reset Cause) register to confirm the same.

2. What if there was no NMI ISR setup by application and error event caused NMI based on default settings from Group 0 ESM error events ?

Answer - If there was no user/application NMI ISR setup then CPU goes to default NMI handler in BootROM where CPU clears and saves the error status and flags in M0 RAM address for debug. Refer to BootROM TRM chapter for more information.

3. What happens if NMI ISR is configured but the error flags are not cleared ?

Answer – The ESM NMIWD timeout occurs and causes High priority watchdog interrupt output from respective CPU. ESM CPU1 High priority watchdog interrupt output is connected to cause XRSn while ESM CPU2/CPU3 causes respective CPURSn as shown in ESM Subsystem Integration View diagram.

4. What if ESM is not configured to generate NMI to respective CPU for high priority CPU EAM errors (passed to ESM as Group 0 error event) ?

Answer – When ESM Group0 error event is active passed on through EAM module and is not configured to generate NMI, CPU goes in fault state. CPU EAM module high priority errors when active should be configured to always trigger NMI to respective CPU.

5. Does ESM/EAM flags clear after XRSn (Device Reset) or CPURSn (CPU Reset) ?

Answer – No, error type register (which is referred to as EAM error flags in this document) and ESM RAW status register (RAW_j) (which is referred to as ESM error flags) are only reset by PORESETn. These flags retain the value even after XRSn and CPURSn since application needs this error flag information for debug in case of reset triggered by ESM as response to error event occurrence not handled by application software.

6. Why does CPU keeps entering NMI ISR even after clearing ESM Raw status register for all events that caused NMI ?

Answer – After clearing the ESM RAW status register (RAW_j) flags make sure the EOI Register is also written to with appropriate key for corresponding error ESM interrupt output. This step of writing EOI is basically user acknowledging the ESM interrupt output which de-asserts the ESM output. If EOI register is not written to then ESM interrupt output remains asserted even if ESM error event flag is cleared.

7. Is it possible to generate NMI to CPU1 only and not to CPU3 and vice-versa based on a specific error event occurrence ?

Answer – Yes, ESM is highly configurable and since there are separate ESM tiles dedicated for each CPU it is possible to setup configuration during initialization to generate NMI or any other error response to either CPU from its respective ESM CPU tile and disable error response or configure different error response for particular error event from another ESM CPU tile.

8 Summary

ESM and EAM provide users systematic way to handle errors in the device upon occurrence. Application note highlights how the error event propagates from the source where this is detected to how the error response is provided from ESM to the device. Lastly, this document lists details on debug tools which can be used to identify the source of error.

9 References

1. Texas Instruments, [F29H85x and F29P58x Real-Time Microcontrollers](#), technical reference manual
2. Texas Instruments, [F29H85x-SDK: ESM Example Between CPU1 and CPU3](#)
3. Texas Instruments, [CCSTUDIO: Code Composer Studio™ integrated development environment \(IDE\)](#)
4. Texas Instruments, [C2000™ SysConfig](#)

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to [TI's Terms of Sale](#) or other applicable terms available either on [ti.com](https://www.ti.com) or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2025, Texas Instruments Incorporated