Reliability, Availability, and Serviceability(RAS) on ARM64

Wei Fu

AGENDA

- What is RAS?
  - Introduction: Definition, Importance, History on X86
  - Overview of RAS functions
- ARMv8 CPU requirements for RAS
  - CPU core, GICv2/GICv3
  - **RAS Extension(ARMv8.2)**
- SDEI(Software Delegated Exception Interface)
- APEI(ACPI Platform Error Interfaces)
  - BERT and CPER, HEST and GHESv2, EINJ/ERST
- SW components for RAS(in example)
  - All SW components
  - How it works: BERT, HEST
- Current status and Planning

What is RAS?

There are three key aspects to robust systems:

Reliability: Continuity, Computation needs be **correct and reliable**.

Availability: Readiness, System needs to **remain available as long as possible**.

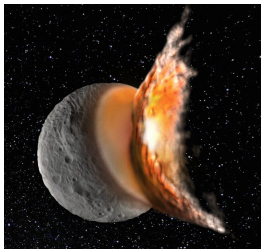
Serviceability: Ability to undergo modifications and repairs, System should provide information to administrator to aid in system servicing.

The RAS architecture mostly describes **data corruption faults**, which mostly occur **in memory and on data links**, can also be used for handling other types of **physical faults** found in systems.

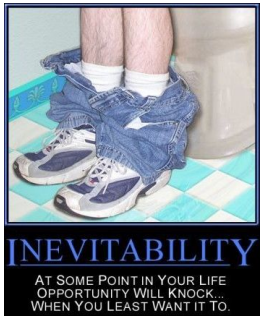
In another word, The RAS architecture primarily cares about errors produced from hardware .

How important is RAS ?

Enterprise systems often provide mission-critical services:



Impacts
System failure or data corruption impacts the customer's business and reputation



Inevitability
Although faults are rare, enterprise systems can be very large. So failures are inevitable.

So we have to maintain system very well, and Operating Expense (OPEX) for maintenance is inevitable. **How to make the cost down?**

OPEX
Operating Expense (OPEX) for maintenance is reduced by **replacing only failed parts**, and **scheduled maintenance** is cheaper than unscheduled service outages.

What if we don't have RAS?

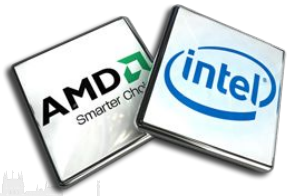
We DO have some other solutions:

To Be Successful in Business,
You Need a Little Luck.
--Richard Branson

LEG
Server

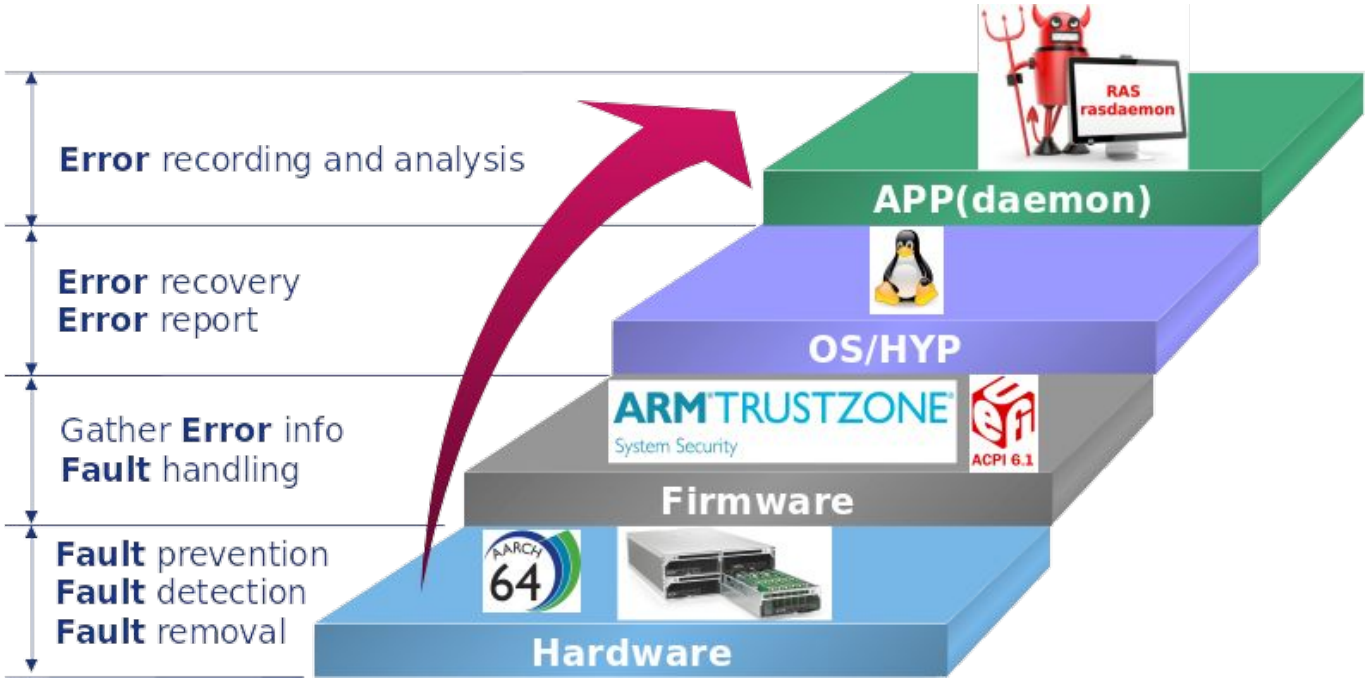
RAS on X86 History

- Machine Check Architecture (**MCA**)
  - An **x86** mechanism in which the CPU reports **hardware errors** to the operating system
  - model-specific registers (**MSRs**)
    - set up machine checking
    - recorde detected errors
    - the information they contain is CPU specific
  - Machine Check Exception (**MCE**)
    - signals the detection of an uncorrected machine-check error
    - handler collect information about error from MSRs
    - mcelog



- **EDAC** (Error Detection and Correction)
  - designed to report and possibly act on hardware errors
  - inspect the hardware directly (system-specific handling and decoding.)
  - only support memory controller and PCI/AGP errors
- PCI-E: Advanced Error Reporting (**AER**)
- Other Hardware features
  - **ECC** in memory controllers and I/O RAMs

Overview of RAS functions



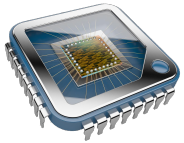
LEG
Server

ARMv8 CPU requirements for RAS

- CPU
  - ARMv8-A architecture (ARMv8.2)
  - EL2, EL3, or both
    - Virtualization extension or Security extensions or both

- Generic Interrupt Controller (GICv2/GICv3)
  - Interrupt routing modes
  - Private and shared interrupts(PPI/SPI)
  - Ability to set an interrupt pending event signaling and delegation
  - Interrupt groups/priority

- **RAS Extension**
  - **ESB (Error Synchronization Barrier) instructions**
  - **RAS Extension registers**
  - **corrupted data poisoning**



Overview of RAS Extension

The RAS extension deals primarily with errors produced from hardware faults.

ESB

Synchronization barrier **instruction**, used to synchronize **Unrecoverable errors**, to make sure containable errors architecturally consumed by the PE and not silently propagated.

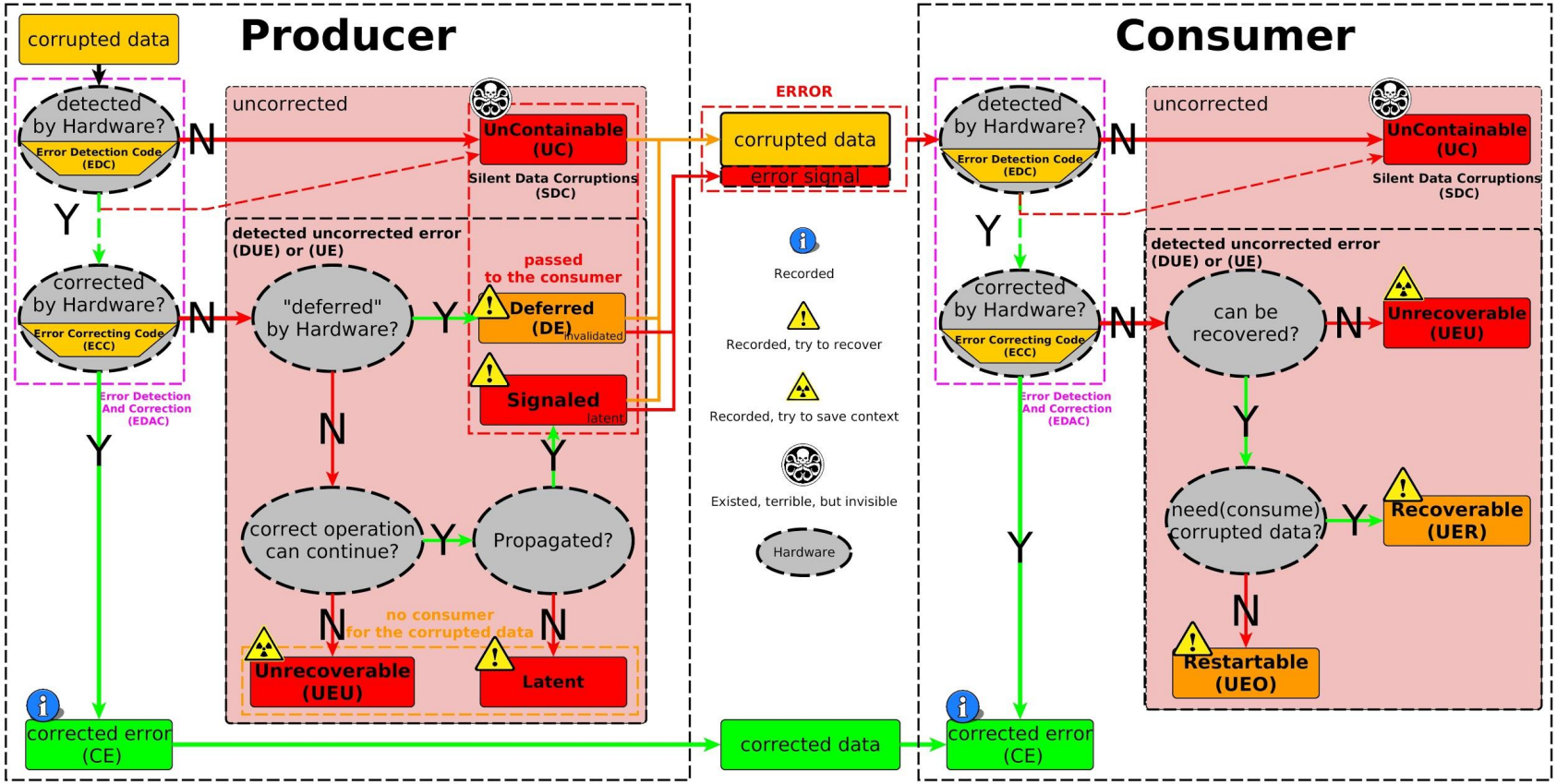
Registers Group

- Processor Feature
- Trapping accesses to Error system
- Error records (Single/Group)
- Deferred Interrupt Status(work with ESB)

POISON

signals corrupted data (hardware mechanism)

Taxonomy of Error in Diagram



ESB instruction

ESB(Error Synchronization Barrier) can be used to isolate Unrecoverable errors. Software can determine that:

- The error was reported as Unrecoverable.
- The preferred return address of the **SEI** is an ESB instruction.

The software between that ESB and the previous ESB can be isolated. ESB might update

- DISR_EL1 / DISR (Deferred Interrupt Status Register)
  - SEIs Generated by instructions occurring in program order before the ESB, are either taken before or at the ESB or pended in the DISR_EL1 / DISR register (depending on whether the SEI exception is masked).
- VDISR_EL2 / VDISR(Virtual Deferred Interrupt Status Register)
  - for virtualization

RAS Extension registers

System register view

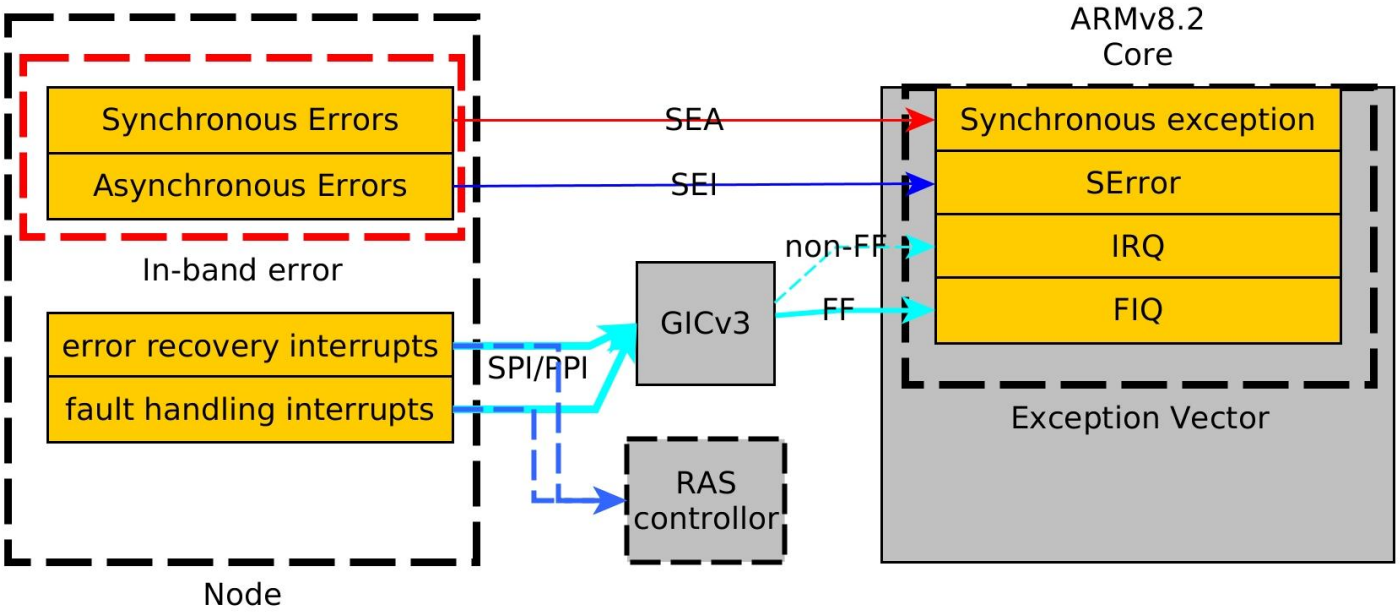
- Processor/Memory Model **Feature** Register
- Error **Record** Register
  - ID, Select Register
  - Record Register(ERX*_EL1)
    - Feature
    - Control
    - Record Primary Syndrome
    - Record Address Register
    - Record Miscellaneous Registers
- Hypervisor **Configuration** Register
- **Virtual** SError Exception Syndrome Register
- Secure Configuration Register

Memory-mapped view

- Peripheral/Component ID Register
- Error Record ID Register
- Record Register(ERR<n>*)
  - Feature
  - Control
  - Record Primary Syndrome
  - Record Address Register
  - Record Miscellaneous Registers
  - **Group Status Registers**
- **Interrupt** Register
  - Error Interrupt Configuration Register (Fault-Handling/Recovery)
  - Generic Error Interrupt Configuration Register
  - Error Interrupt Status Register
- Device Affinity Register
- Device Architecture Register

RAS Extension

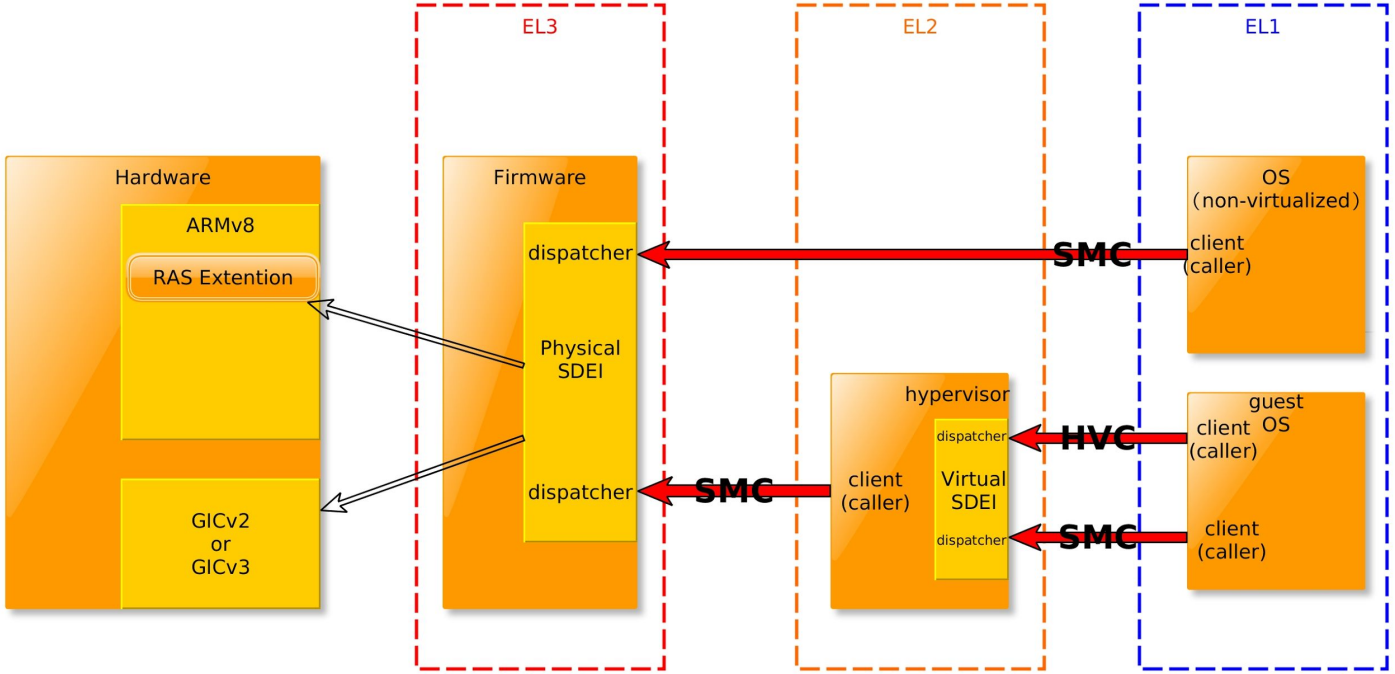
How to route interrupts and Exception abort



LEG
Server

SDEI: Software Delegated Exception Interface

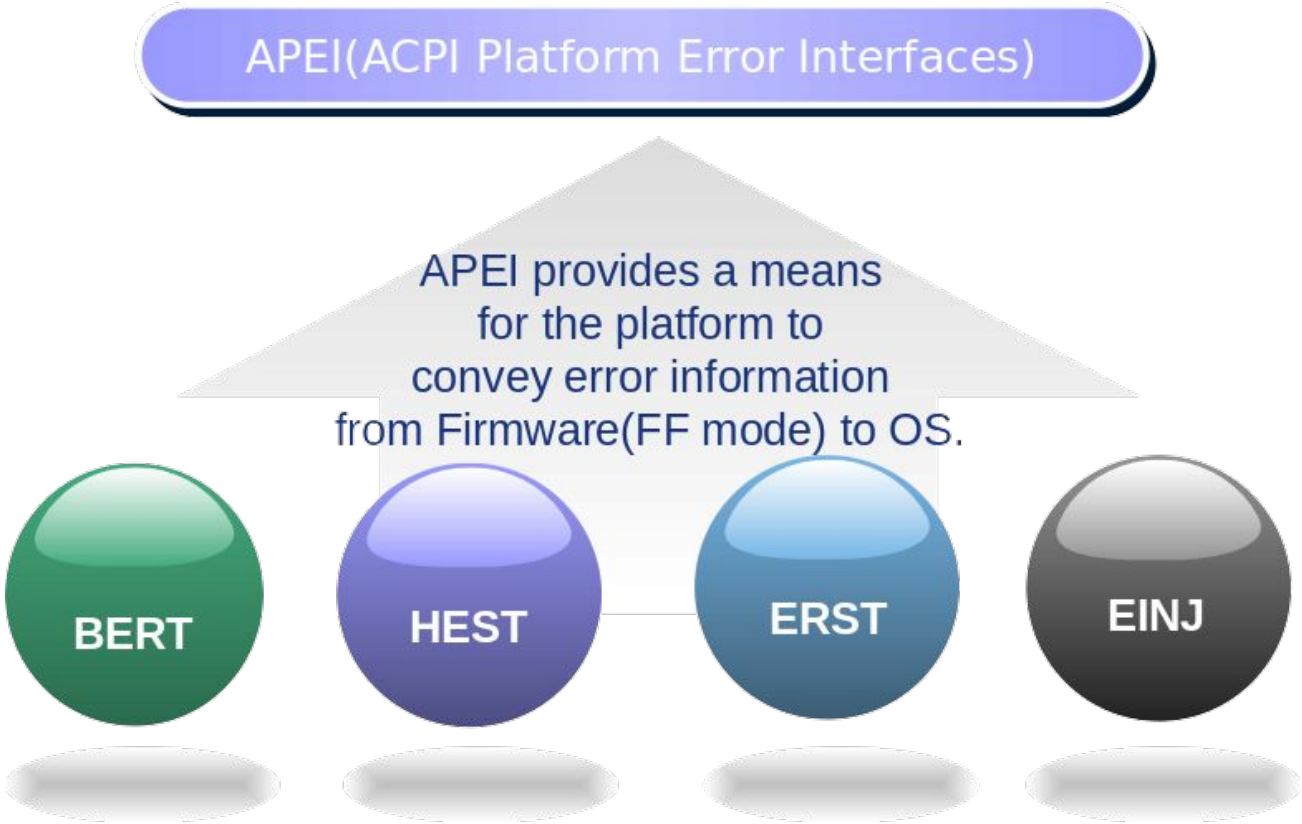
A PSCI-like interface for **registering and servicing system events** from system firmware(But it's NOT only for RAS)



SDEI: Some Interface Functions

Event control	info & other control
SDEI_EVENT_REGISTER	SDEI_VERSION
SDEI_EVENT_ENABLE/DISABLE	SDEI_FEATURES
SDEI_EVENT_CONTEXT	
SDEI_EVENT_COMPLETE	SDEI_INTERRUPT_BIND/RELEASE
SDEI_EVENT_COMPLETE_AND_RESUME	SDEI_PE_UNMASK
SDEI_EVENT_UNREGISTER	SDEI_PE_DATA_RESET
SDEI_EVENT_STATUS	SDEI_SYSTEM_DATA_RESET
SDEI_EVENT_GET_INFO	
SDEI_EVENT_ROUTING_SET	
SDEI_EVENT_SIGNAL	

APEI(ACPI Platform Error Interfaces)



BERT: Boot Error Record Table

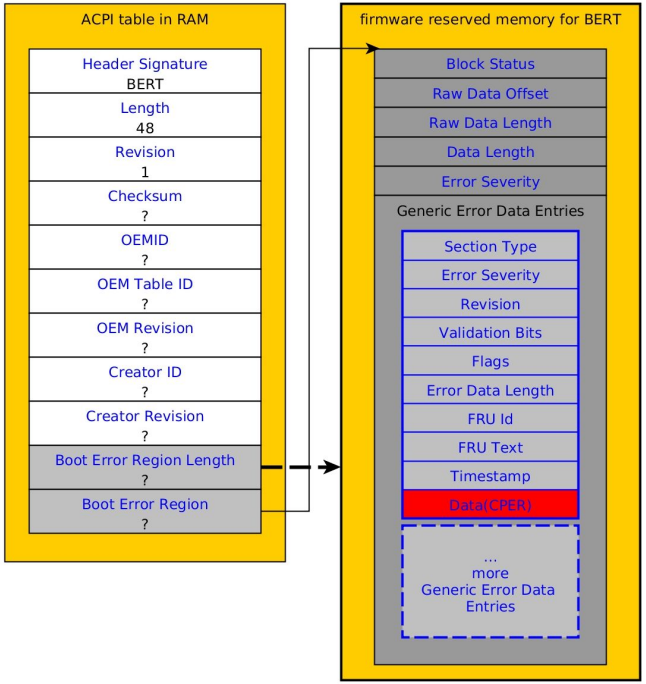
Scenarios: Record errors in emergency (OS crash/reset)

Mechanism: Report unhandled errors that occurred **in the previous boot**

Key info:

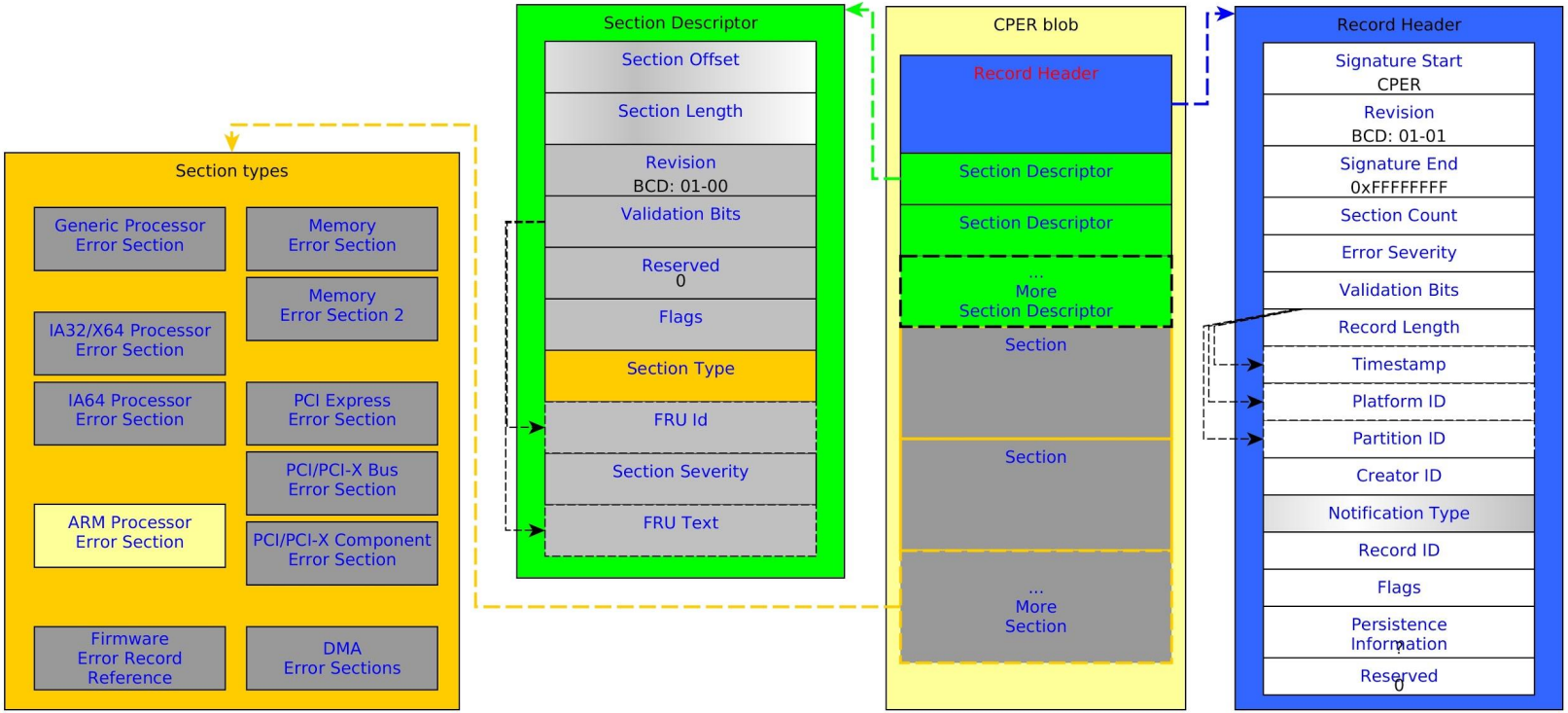
WHERE are the error records

SIZE of the error record region



CPER : Common Platform Error Record

The description is in the Appendix of UEFI Specification. With the help of CPER and FF(Firmware-First mode), OS can get all kinds of error we could think of.



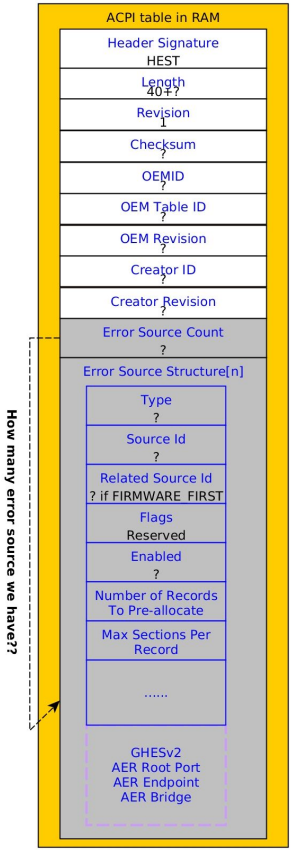
HEST: Hardware Error Source Table

Scenarios: Record errors in runtime (OS still can work)

Mechanism: describes a standardized mechanism platforms may use to describe their error sources by Error

Key info:

- **HOW** to get trigger
- **WHERE** are the error records
- **HOW** to release records' mem

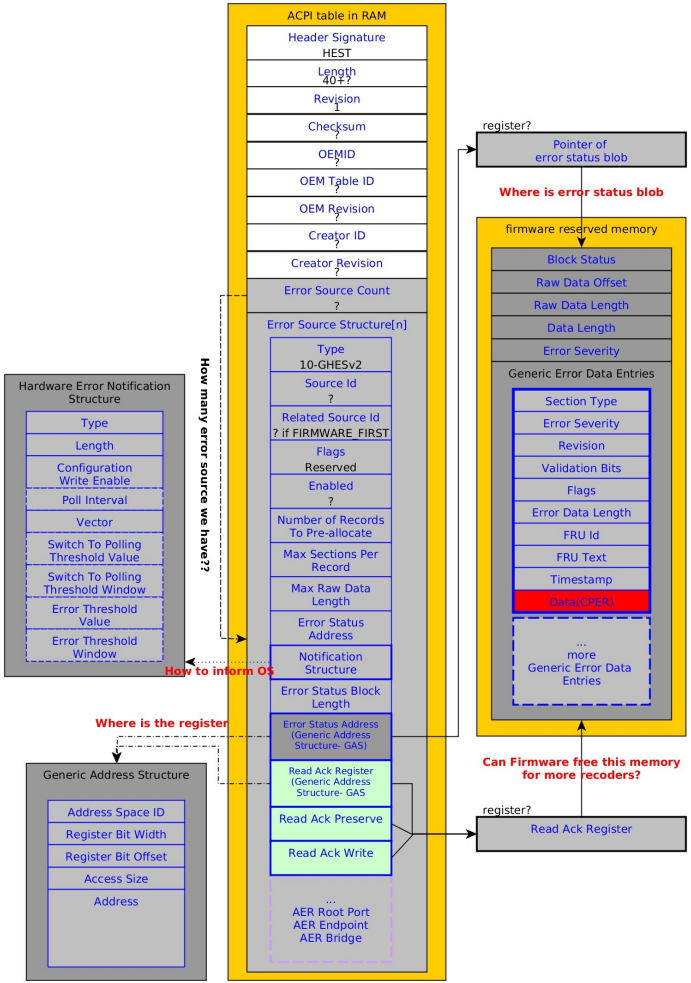


HEST: Hardware Error Source Table

For **IA-32** :
MCE/CMC/NMI

For **PCI**:
AER Root
Port/Endpoint/Bridge

For **generic hardware**:
GHES(Generic Hardware Error Source) V1/V2



For **ARM64** : **GHES v2**

HOW to get trigger:
Notification Structure

WHERE are the error records:
Error Status Address
(GAS : Generic Address Structure)

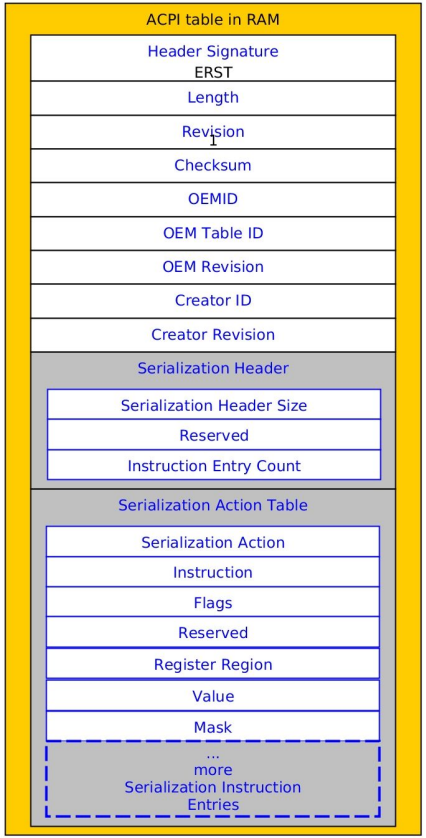
HOW to release records' mem:
Read Ack Register

ERST: Error Record Serialization Table

Scenarios: Record and Retrieve errors in persistent storage

Mechanism : Operation abstract, provides details necessary to communicate with on-board persistent storage for error recording

Plan B: IPMI , MTD, block device, ABRT...

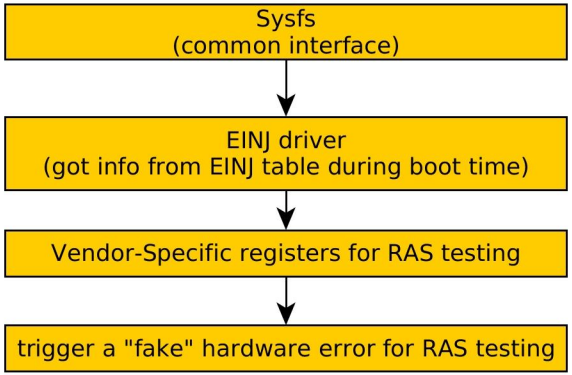


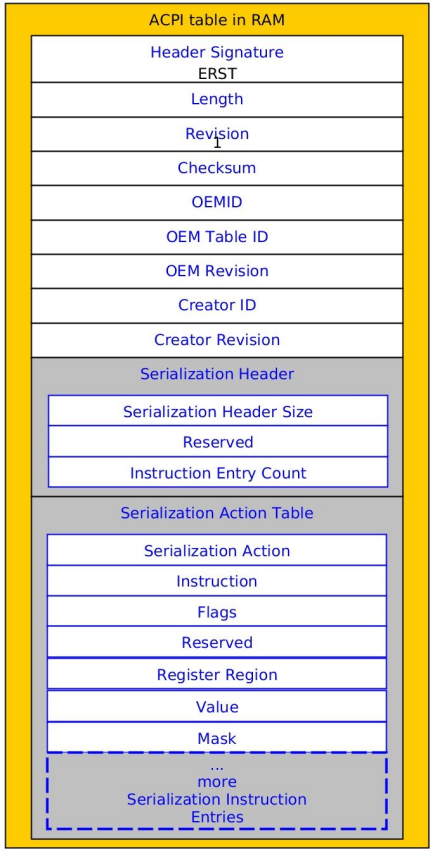
EINJ: Error Injection Table

Scenarios: Test OSPM error handling stack

Mechanism: Operation abstract, provides a generic interface which OSPM can inject hardware errors to the platform without requiring platform specific software.

Possible use case:





SW components for RAS

Firmware:

ARM TF(ARM Trusted Firmware)with RAS support

UEFI(Unified Extensible Firmware Interface): tianocore-edk2

ACPI tables(with APEI tables)

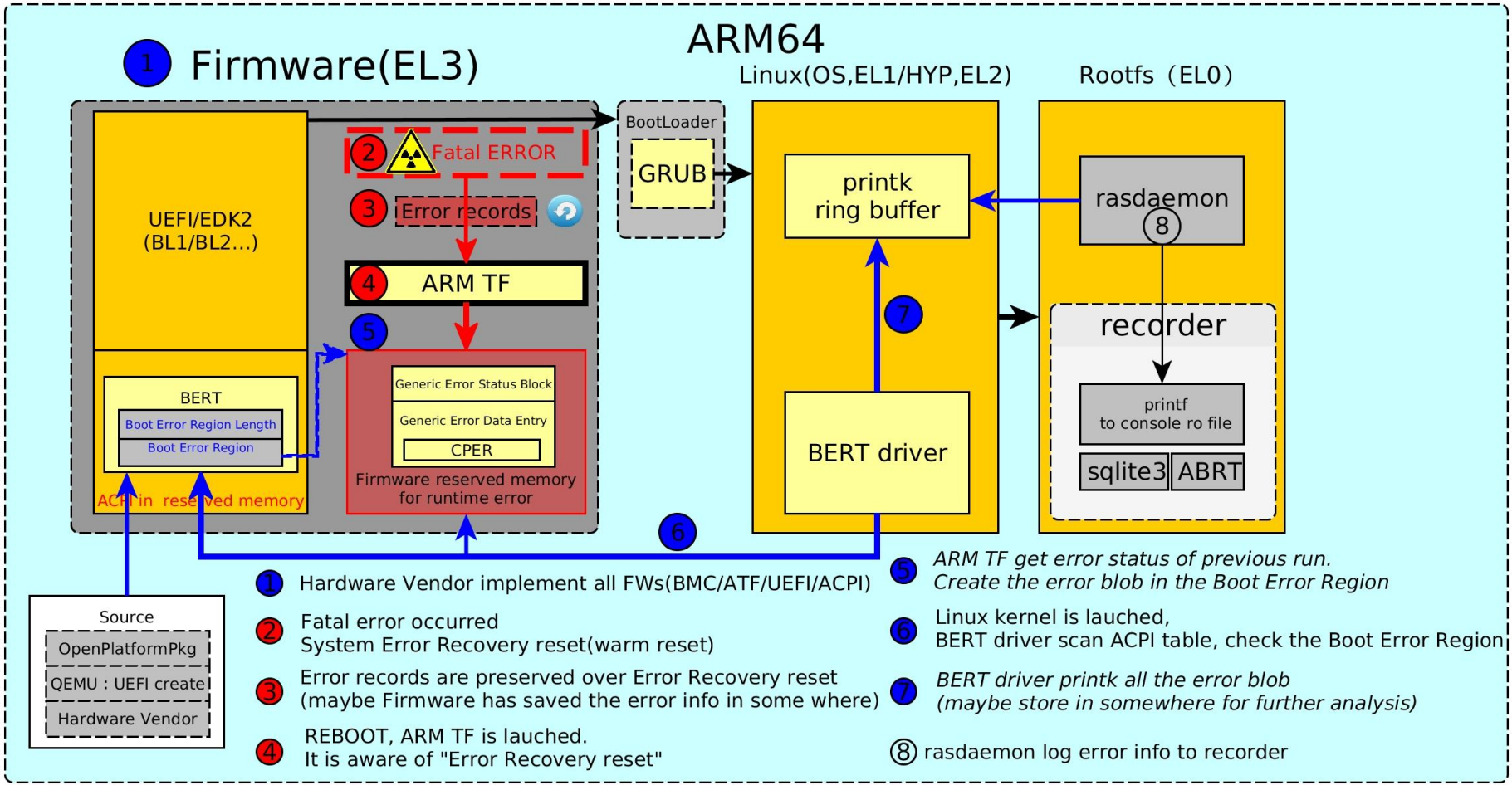
OS:

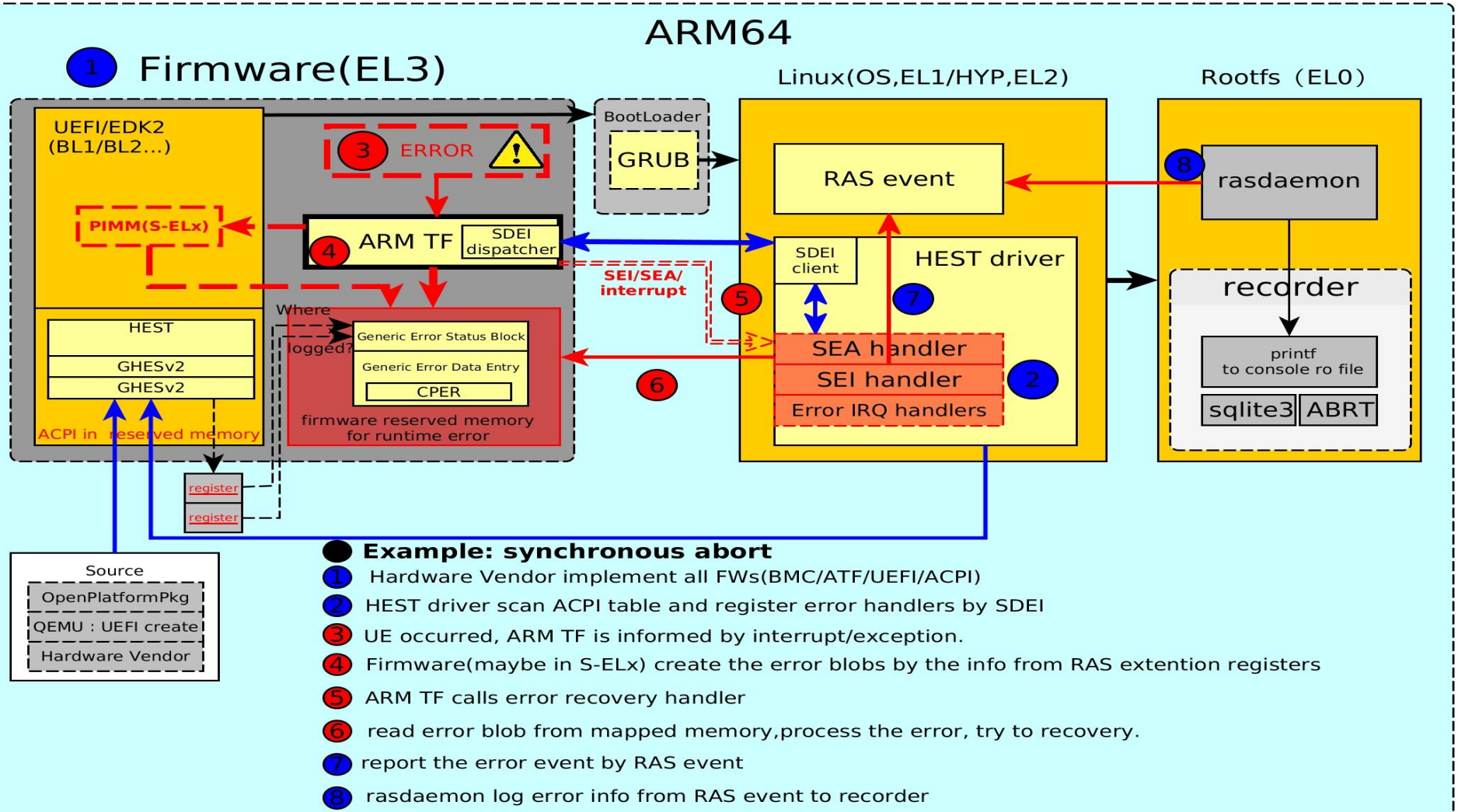
Linux kernel(with APEI drivers)

Userspace:

rasdaemon

How it works: BERT, HEST




ARM64
1 Firmware(EL3)
Linux(OS,EL1/HYP,EL2)
Rootfs（EL0）
UEFI/EDK2
(BL1/BL2...)
PIMM(S-ELx)
3 ERROR
4 ARM TF
SDEI dispatcher
BootLoader
GRUB
RAS event
8 rasdaemon
SDEI client
HEST driver
SEI/SEA/ interrupt
5
7
recorder
printf to console ro file
sqlite3
ABRT
HEST
GHESv2
GHESv2
ACPI in reserved memory
Where logged?
Generic Error Status Block
Generic Error Data Entry
CPER
firmware reserved memory for runtime error
SEA handler
SEI handler
Error IRQ handlers
2
6
register
register
Source
OpenPlatformPkg
QEMU : UEFI create
Hardware Vendor
Example: synchronous abort
1 Hardware Vendor implement all FWs(BMC/ATF/UEFI/ACPI)
2 HEST driver scan ACPI table and register error handlers by SDEI
3 UE occurred, ARM TF is informed by interrupt/exception.
4 Firmware(maybe in S-ELx) create the error blobs by the info from RAS extention registers
5 ARM TF calls error recovery handler
6 read error blob from mapped memory,process the error, try to recovery.
7 report the error event by RAS event
8 rasdaemon log error info from RAS event to recorder

Current status

- Hardware: ARMv8.2 spec includes RAS extension
- Firmware:
  - RAS extension doc has described some different software sequences based on different scenarios, but the spec itself is underdevelopment
  - SDEI is underdevelopment

- OS(Linux): APEI on ARM64 can be enabled in kernel.
  - BERT works
  - GHESv2 of HEST is upstreaming(v11) by Qualcomm engineers
  - ERST and EINJ are untested because of the lack of FW

- Daemon(Application)
  - rasdaemon can be launched on ARM64

What we are missing and planning

- Hardware: need a hardware or a simulator (ARMv8.2, including RAS extension)

- Firmware:
  - RAS extension and SDEI spec
  - ARM TF with RAS extension support and SDEI dispatcher
  - REST and EINJ implementation

- OS(Linux):
  - SDEI client and dispatcher(virtual SDEI)
  - GHESv2 support(upstreaming)
  - RAS event (How to pass/keep info?)

- Daemon(Application)
  - rasdaemon on ARM64 for RAS event

Acknowledgments

Al Stone (Red Hat)

Charles Garcia-Tobin(ARM)

John Feeney(Red Hat)

Jon Masters(Red Hat)

Zhigao Li(Huawei)

Thank You

#BUD17

For further information: www.linaro.org

BUD17 keynotes and videos on: connect.linaro.org