uniserver

D4.5 HEI and Error Handlers Implementation

Contract number	688540
Project website	http://www.uniserver2020.eu
Contractual deadline	Project Month 15 (M17): 30th April 2017
Actual Delivery Date	2 May 2017
Dissemination level	Public
Report Version	1.1
Main Authors	Peter Lawthers (APM)
Reviewers	Konstantinos Tovletoglou (QUB), Manolis Kaliorakis (UoA), George Papadimitriou (UoA), Athanasios Chatzidimitriou (UoA), Dimitris Gizopoulos (UoA), G. Karakonstantis (QUB), Lev Muchanov,(QUB), Christos Antonopoulos (UTH), Pedro Trancoso (UCY), Zacharias Hadjilambrou (UCY)
Keywords	Hardware Exposure Interface, Error Handlers, X-Gene Platform

Notice: The research leading to these results has received funding from the European Community's Horizon 2020 Programme for Research and Technical development under grant agreement no. 688540.

Disclaimer
This deliverable has been prepared by the responsible Work Package of the Project in accordance with the Consortium Agreement and the Grant Agreement Nr 688540. It solely reflects the opinion of the parties to such agreements on a collective basis in the context of the project and to the extent foreseen in such agreements.

Acknowledgements
The work presented in this document has been conducted in the context of the EU Horizon 2020. UniServer is a 36-month project that started on February 1st, 2016 and is funded by the European Commission. The partners in the project are:

The Queen's University of Belfast (QUB)
The University of Cyprus (UCY)
The University of Athens (UoA)
Applied Micro Circuits Corporation Deutschland Gmbh (APM)
ARM Holdings UK (ARM)
IBM Ireland Limited (IBM)
University of Thessaly (UTH)
WorldSensing (WSE)
Meritorious Audit Limited (MER)
Sparsity (SPA)

More information
Public UniServer reports and other information pertaining to the project are available through the UniServer public Web site under http://www.uniserver2020.eu.

Change Log

Version	Description of change
0.1	Initial draft
0.2	Updated with firmware changes and debug hints
0.3	Incorporated review comments
1.0	Final version
1.1	Concurrent buffering updates, formatting

Table of Contents

Index of Figures

Index of Tables

Executive Summary

This document describes the Application Programming Interface (API) for the X-Gene processor family Hardware Exposure Interface (HEI) and Error Handling specifications as developed in task T4.1 within Work Package WP4 of the UniServer Project Description of Action (DoA). This is in fulfilment of deliverable D4.5, HEI and Error Handlers Implementation.

The HEI API provides an event driven interface for notifications of CPU and peripheral errors. This interface will be used to enable the development of firmware and software modules to monitor the hardware behavior, alerts, and error conditions and provide the mechanism for System Failure Avoidance while the design margins of the hardware platform will be explored for energy-efficiency or performance boost.

1. Introduction

This document describes the Application Programming Interface (API) for the X-Gene processor family Hardware Exposure Interface (HEI) and Error Handling specifications as developed in task T4.1 within Work Package WP4 of the UniServer Project Description of Action (DoA). This is deliverable D4.5. The described specification applies to the UniServer microserver board defined in D3.1 and finalized as per D3.4. This hardware interface will be used to enable the development of firmware and software modules to monitor the hardware behavior, alerts, and error conditions and provide the mechanism for System Failure Avoidance while the design margins of the hardware platform will be explored for energy-efficiency or performance boost.

1.1. Conventions and Terminologies

Throughout this document, unless otherwise specified, X-Gene is used to denote all the SoC's belonging to the X-Gene family, including X-Gene 1, X-Gene 2 and X-Gene 3.

This document uses the following terms and abbreviations as described in the following table.

Term	Definitions
ACPI	Advanced Configuration and Power Interface
API	Applications Programming Interface
BackUcErr	Background Uncorrectable error DDR double bit error detected during a background scrubbing operation
BMC	Baseband Management Controller
Byte	An 8-bit quantity
CErr	Correctable Error
Clr	Clear
CPPC	Collaborative Processor Performance Control (CPPC), a new interface for CPU performance control between the OS and the platform defined in ACPI 5.0 specification
CPU	Central Processor Unit. A CPU might contain one or more CPU cores.
CSW	Central Switch
DataTag (error)	
DemandUcErr	Demand Uncorrectable error.. DDR double bit error detected during a requested operation
DIMM	Dual In-line Memory Module
DRAM	Dynamic random-access memory, a type of random-access computer memory
DSDT	Differentiated System Descriptor Table
ECC	Error Checking and Correction
EH	Error Handling
GUID	Global Unique Identifier
HEI	Hardware Exposure Interface
I2C	Inter-integrated Circuit, a 2-wire, multi-master, serial bus
ICF	Instruction Cache and Fetch
ICFESR	Instruction Cache and Fetch Error Status Register
Intr	Interrupt
I/O	Input/Output
IOB	I/O Bridge

L2C	L2 cache
L2ESR	L2C Error Status Register
L3C	L3 cache
LSU	Load Store Unit
MCB	Memory Controller Bridge
MCU	Memory Controller Unit
MMU	Memory Management Unit
MultCErr	Multiple Correctable Errors occurred
MultiHit	Multiple Cache Hit error. During an Instruction or Data Cache read lookup not just one but multiple hits occurred in the Way Select macro. Considered an UcErr
MultUnCErr	Multiple Uncorrectable Errors occurred
PCIe	Peripheral Component Interconnect Express (PCIe or PCI-E), a serial expansion bus standard for connecting a computer to one or more peripheral devices.
PMD	Processor Module. An X-Gene PMD comprises two CPU cores sharing the same L2 cache
PMPro	Power Management Processor
SATA	Serial ATA (SATA, abbreviated from Serial AT Attachment), a computer bus interface that connects host bus adapters to mass storage devices such as hard disk drives, optical drives, and solid-state drives.
SBF	Standby Fabric
SDB	Stored Data Buffer
SlimPro	Scalable Lightweight Intelligent Management Processor
SoC	System-on-chip
UcEvict	Uncorrectable Eviction Error. An L3 cache line with an error is evicted thus dropped writeback. Considered an UcErr.
UcErr	Uncorrectable Error
VRD	Voltage Regulator-Down
Word	A 16-bit quantity

Table 1: Terminologies

1.2. Overview

The API is based on the register mappings described in the D4.1, Hardware Exposure Interface (HEI) and Error Handlers Specification, version 1.2. The interface described therein exposes a set of sensors and registers that allow software to closely monitor and control the processor.

It is important to note that the HEI registers are only accessible via I2C. They are not accessible via a memory mapped interface. Helper routines are provided in the API to ease application development.

The API provides a demand notification mechanism whereby software can register for a series of events that will be triggered when certain conditions are met. The software can be a Linux kernel module or an application running in user space on Linux. The software can specify a *handler* (also known as a callback) that will be invoked when the specified event occurs.

For example, the API provides for notification when the SOC Hot threshold is reached (see Table 6, *Sensor Registers*, in the D4.1 HEI document for specifics of the threshold values). The notification is delivered asynchronously, without the application having to poll.

The underlying firmware implementation in the X-Gene processor provides a set of mask registers so that the software will only be notified of events in which it has expressed interest. This reduces the overhead of delivering events that the software does not care about.

2. Event Monitoring

The API provides an interface to the alert sources specified in D4.1, HEI and Error Handlers Specification. It does not provide wrappers for the sensor registers described in that document. The following event types are provided:

- Thermal and Power events
- PMD/CPU Errors (including memory controller errors)
- Cache errors
- Memory Errors
- PCIe Errors
- SATA Errors
- Other I/O Errors
- ACPI state change

Each event that is delivered from firmware in the processor to software must be handled before subsequent events of the same type are delivered. In this manner, events are said to be *blocking*, and must be explicitly cleared by software. This is analogous to an interrupt handler clearing the source of an interrupt in an Interrupt Service Routine (ISR).

The API provides a mechanism for kernel modules and user space applications to register a *callback handler* that will be invoked upon receipt of an event. The SlimPro firmware provides a mechanism to mask out event generation when no one has registered an interest, reducing unnecessary interrupts and improving the utilization of system resources.

2.1. Firmware

Internal to the processor complex is a dedicated portion of firmware (executing on the SlimPro core) that continually monitors the processor complex. When it detects an error situation, it informs the operating system by delivering an asynchronous message via an I2C mailbox interrupt.

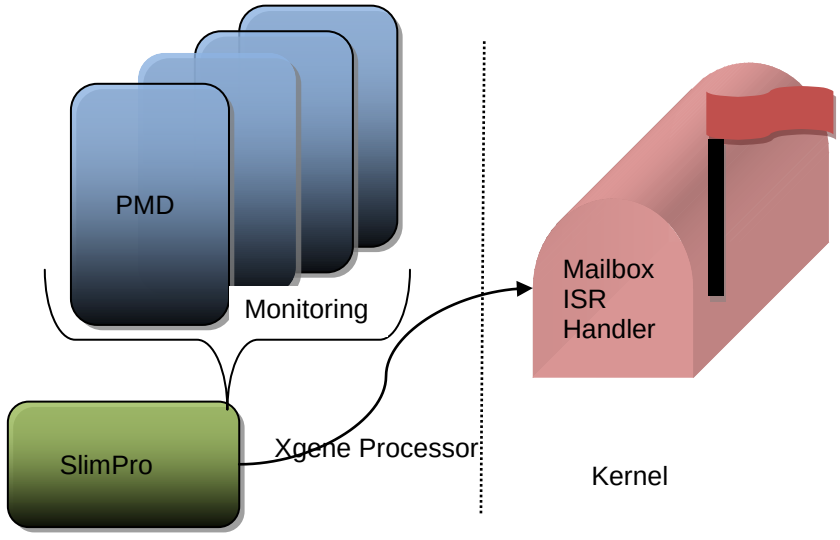

Figure 1: HW/SW communication

After receiving and acknowledging the interrupt, the kernel processes the event in the context of a kernel thread (invoked via `schedule_work()`).

2.2. I2C Layering

The HEI module requires an I2C interface to send and receive data from the SlimPro processor inside the X-Gene processor complex. To simplify the implementation, there are two interfaces; the first is a synchronous I2C interface for reading and writing the register map. The second interface is asynchronous, and is used for receiving event data.

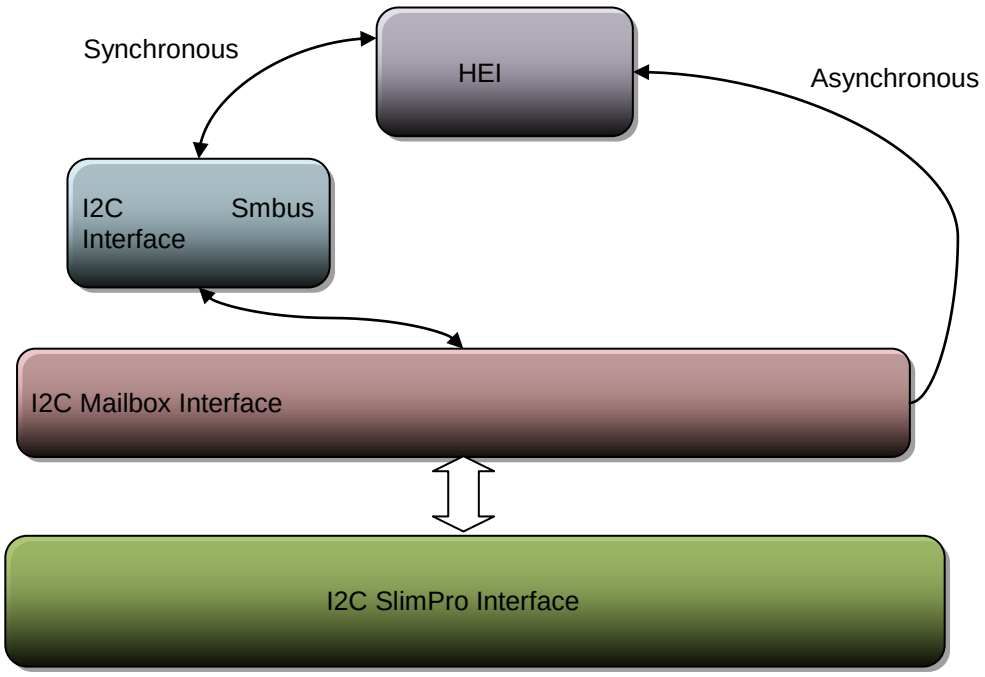

Figure 2: HEI I2C layering

The HEI layers itself on top of the X-Gene SlimPro mailbox driver, `i2c-xgene-slimpro.c` which provides a synchronous I2C smbus interface. The HEI uses this interface to read and write the X-Gene register map.

2.3. Event Delivery Handling

Asynchronous notifications from the hardware are delivered via an I2C interface using the mailbox interface. The SlimPro firmware creates a mailbox message of type SLIMPRO_USER_MSG, and then rings a doorbell. The Tianocore firmware has been enhanced to create an ACPI device (`APMC0D2A`) with a dedicated doorbell (doorbell 5) in the Differentiated System Descriptor Table (DSDT).

Messages are delivered from the firmware, placed into a kernel fifo, and then a kernel thread is scheduled to handle the message via `schedule_work()`. The kernel thread verifies that it is the intended recipient of the message by checking the type and then proceeds to handle the message.

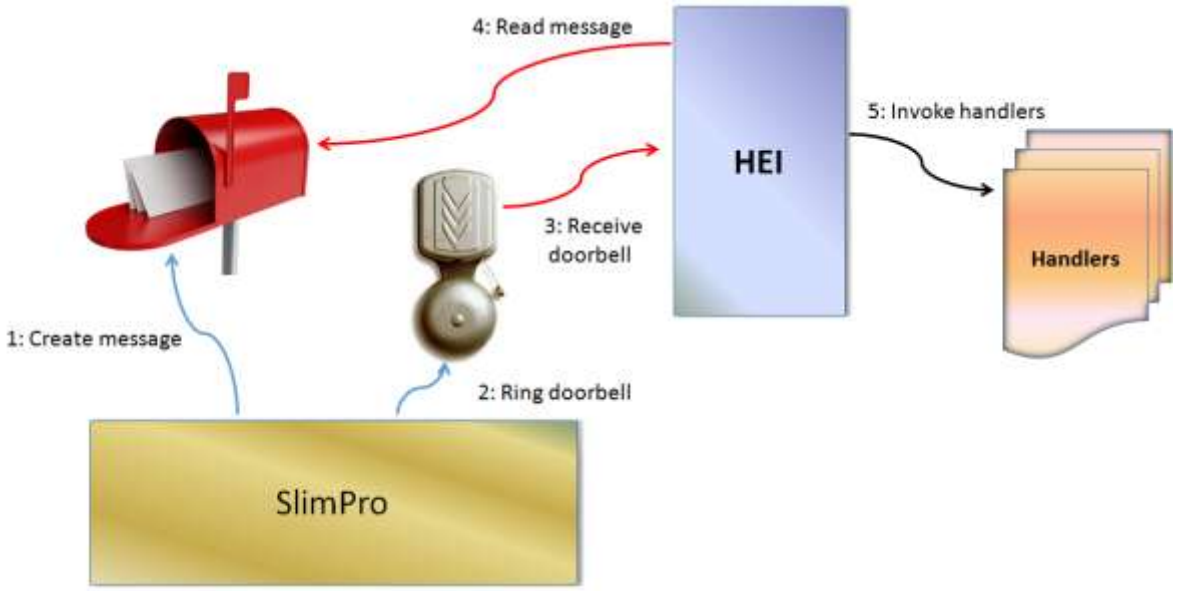

Figure 3: Event flow from firmware to kernel

The HEI module reads the event registers and determines the events that are pending. Each event type may have zero or more callback handlers associated with it. For each callback handler, the HEI module will invoke the handler if it has expressed an interest in the specific event. If a handler has registered for more than one event and more than one event is active, then the handler will be passed a mask of all active events. It will not be invoked twice, once for each event.

2.4. Concurrency

The API is designed to provide a high level of concurrency. As such, there is minimal synchronization between users of the API. This reduces contention and improves performance, but cooperating users of the API must be aware of possible interactions.

The primary area of conflict can be where a multi-step operation is required, such as selecting a PMD and reading the results. If one user of the API writes to one of the selection registers (for example, the PMD selection register at 0x80), it should ensure that no other user writes to the same register before the results in the PMD registers 0x81-0x84 are read. The same is true of the MCU (0x90) register.

Stated differently, it is up to the user of the API to ensure that multiple threads cooperate when modifying one of the selection registers. The API provides no transaction semantics.

2.4.1. Buffer Management

The firmware has limited buffering capability. When multiple callbacks wish to consume the same event, care must be taken so that all consumers see the same data. The API provides a buffering scheme so that all users will see the same data. This is particularly important for events that require a “selection” write, such as PMD and MCU events.

Each firmware interrupt is assigned a monotonically increasing 64-bit generation number. This can be thought of as an interrupt instantiation. The generation number implicitly tracks the firmware buffers. The generation number is a parameter in the HEI callback.

For example, if the firmware sends an interrupt saying there is a PMD and MCU event, then the registered event callback routine would first request a buffer from the HEI, and supply a buffer "read" routine to be

invoked if there is no buffer (a separate, callback). The particular event instantiation is specified by the generation number (this is similar to how NFS handles inode number reuse during delete-reuse cycles).

The buffer callback would read the I2C registers for the PMD and MCU events, and put the data in the buffer. The buffer will be released when the last callback has returned the buffer.

See the API prototypes section for details on the API.

2.5. Kernel Modules

There are two kernel modules. The first is the `xgene_hei` module, which implements the linkage between the SlimPro firmware and the Linux kernel. This module maintains the list of callback handlers and manages the firmware. The source is located in the Linux tree under `drivers/misc/xgene_hei.c` .

The second module is intended to replace the existing HWMON driver, and is called `xgene_hei_hwmon`. The source is located in the Linux tree under `drivers/misc/xgene_hei_hwmon.c` . This module registers for the same set of events as monitored by HWMON and also EDAC. This includes PMD, memory, cache (level 3), PCIE, SATA, and temperature. Once an event is received, the event is logged to syslog. The format of the logging is open to discussion; currently it logs a message for every specific bit that is set in an event. During the course of the project log messages formatting will be revised. There is nothing that precludes the logging being moved to user space.

2.5.1. Module parameters

Each module has a debug parameter that controls the verbosity of debug messages. Messages are logged using the Linux kernel tracing facility, which is typically mounted under `/sys/kernel/debug/tracing`. The log itself is the trace file. The verbosity can be controlled by increasing (or decreasing) the debug parameter when the module is loaded. At run time, the parameter is exposed in the `/sys/module/<module_name>/parameters/debug` file.

For example, to disable all logging for the xgene_hei module,

```
echo 0 > /sys/module/xgene_hei/parameters/debug
```

In addition to the debug parameter, `xgene_hei_hwmon` has an additional parameter that controls the logging of messages to syslog. To prevent a system from spamming the log, all log messages can be **ratelimited**. However, this can also mean that a log message will not be seen during testing. The default is for messages to **not** be ratelimited.

To enable ratelimiting to avoid spamming syslog

```
echo 1 > /sys/modules/xgene_hei_hwmon/parameters/enable_ratelimit
```

By default, log messages are generated in both raw and detailed format. The raw format presents the values read from the firmware, while the detailed format provides a textual description. To change (or disable logging entirely), one can write to the `logfmt` module parameter. For example, to only log the raw data:

```
#define LOGFMT_NONE     0x0
#define LOGFMT_RAW      0x1
#define LOGFMT_DETAIL   0x2
echo 1 > /sys/modules/xgene_hei_hwmon/parameters/logfmt
```

2.5.2. Logging

All events received from the firmware are logged by the `xgene_hei_hwmon` driver. Each log message is accompanied by a monotonically increasing number to make it easier to distinguish log messages. Each log message event is a single string, terminated by a newline. The format (grammar) of the log messages is as follows:

2.5.2.1 * Raw format

```
*  HEI alert_<number>  [type_string]: 0x<register>=0x<value>
*              [ ,0x<register>=0x<value> ]
```

2.5.2.2 * Detailed format:

```
*   HEI_alert_<number> soc_hot: temp=<value>
*   HEI_alert_<number> soc_vr_hot: temp=<value>
*   HEI_alert_<number> pmd_vr_hot: mask=0x<value> temp=<value>
*   HEI_alert_<number> dimm_vr_hot: mask=0x<value> temp=<value>
*   HEI_alert_<number> dimm_hot: dimm_chan_<value>=<value>
*                      [, dimm_chan_<value>=<value> ]
*   HEI_alert_<number> pmd_error: pmd=<value> [, cpu=<value> ]
*                      [, error_string ] [, error_string ]
*   HEI_alert_<number> mem_error: mcu=<value> [, error_string ]
*                      [, error_string ]
*   HEI_alert_<number> l3c_error: [ error_string ]
*                      [, error_string ]
*   HEI_alert_<number> pci_error:
*                      pci=0x<value> device=0x<value> function=0x<value>
*                      [, error_string ]
*   HEI_alert_<number> sata_error: sata_chan=<value> [, error_string ]
*                      [, error_string ]
*   HEI_alert_<number> acpi_change: [ error_string ] [, error_string ]
*   HEI_alert_<number> other_error: [ error_string ] [, error_string ]
*
```

A sample of the logging output is given below:

```
May 17 17:58:23 tigershark kernel: HEI_alert_193 l3c_error: <reg_0x75=0x41
reg_0x76=0x4 reg_0x77=0x0 reg_0x78=0x0 > tag error, retry single bit tag error,
ecc group=4
May 17 17:58:29 tigershark kernel: HEI_alert_194 soc_hot: <reg_0x10=0x2e >
temp=46
May 17 17:58:29 tigershark kernel: HEI_alert_195 pmd_error: <pmd=0: reg_0x81=0x4
reg_0x82=0x0 reg_0x83=0x0 reg_0x84=0x0 > pmd=0 cpu=0 L1 LSU correctable
error,<pmd=1: reg_0x81=0x0 reg_0x82=0x302 reg_0x83=0x2 reg_0x84=0x0 > pmd=1
cpu=0 L2 uncorrectable error, L2 data ECC error
May 17 17:58:29 tigershark kernel: HEI_alert_196 mem_error: <mcu=0: reg_0x91=0x1
reg_0x92=0x0 reg_0x93=0x0 > correctable error, <mcu=1: reg_0x91=0x2
reg_0x92=0xdead reg_0x93=0xbeef > demand uncorrectable error, row=0xdead,
column=0xbeef, <mcu=2: reg_0x91=0x0 reg_0x92=0x0 reg_0x93=0x0 > <mcu=4:
reg_0x91=0x0 reg_0x92=0x0 reg_0x93=0x0 >
```

Figure 4: Sample logging from xgene_hei_hwmon

2.6. API Callbacks

2.6.1. Kernel Level

When the kernel level handler receives a callback from the HEI module, it reads the register map to gather information about the type of event. After it has collected the information, it ACKs the event, allowing subsequent events to be processed. It then logs the event using the normal system logging mechanism (a **ratelimited** `printk()`).

Since the HEI replacement for HWMON has all the logging capability, it also provides the interface to user space. There are two separate interfaces; an `ioctl`() based interface for command and control, and a `netlink` socket interface for receipt of the actual event data.

The rationale for splitting up the interface is that because the `ioctl`() interface is file-descriptor based, this means that when the process exits, the kernel will be notified via the normal file close callbacks. While a

socket is in fact represented by a file descriptor, there is no clean mechanism in the netlink socket interface to be notified when a socket is closed. The kernel must be informed so that it can clean up any pending callbacks and to clear any event registrations with the hardware that were claimed by the user process.

The pseudo-code for a kernel level event callback would be as follows:

```
Request buffer from HEI, providing buffer fill callback routine
      Buffer fill callback invoked if buffer empty
Acknowledge (clear) events
Process events
```

For more information, see `xghei_hwmon_cb()` in `drivers/misc/xgene_hei_hwmon.c`

2.6.2. Event Buffer

The xgene_hei_hwmon driver uses a buffer (`struct xghei_evtbuf`) to track the data for events. This buffer has space for all possible events, and is defined in `include/misc/xgene_hei.h` in the kernel tree.

2.6.3. User Level

To provide event callouts to user space, the HEI kernel level handler creates a `netlink` socket via `genl_register_family()` and also creates a device file in the device namespace (created dynamically via `uevent`) for command and control.

When an event occurs, the flow is exactly the same as with a kernel based module. As with the kernel level handlers, the user level handlers gather all the information about the event, and then ACKs the event in the hardware before sending the event to user space. The rationale for this is so that there is no delay in subsequent event notifications caused by the latency of user space being scheduled, running, and finally clearing the event in the hardware.

User space is passed a message buffer that contains all pertinent information about the event. It can, if it so chooses, read the register map or other sensors for additional information, but the bulk of the information is present in the actual callback notification.

2.6.3.1 Structure Access

The structure handed to user space is a composite of all possible events, as follows:

```
struct xghei_uspace_event {
        /*
         * Pointers in each structure are *offsets*
         * into data[0], not absolute, relative to
         * the start (&xghei_uspace_event->data);
         *
         * To access the data, either use the relative offset
         * directly, or add the absolute address of the
         * address of data[0]
         */
        struct xghei_uspace_socevt      soc_evt;
        struct xghei_uspace_vrevt       soc_vr_evt;
        struct xghei_uspace_vrevt       pmd_vr_evt;
        struct xghei_uspace_vrevt       dimm_vr_evt;
        struct xghei_uspace_dimmevt     dimm_evt;
        struct xghei_uspace_pmdevt      pmd_evt;
        struct xghei_uspace_mcuevt      mcu_evt;
        struct xghei_uspace_l3cevt      l3c_evt;
        struct xghei_uspace_pcievt      pci_evt;
        struct xghei_uspace_sataevt     sata_evt;
        struct xghei_uspace_acpievt     acpi_evt;
        /*
         * Actual event data follows in-band
         */
        uint8_t         data[0];
};
```

Figure 5: User event structure

For example, to access the SOC temperature information when handling an SOC temperature event, the buffer will contain the temperature reading from the SOC. If the HEI kernel level callout handler was unable to read the data, the buffer will then contain the kernel error code.

```
struct xghei_uspace_socevt {
      uint16_t soc_temp;
      int32_t  soc_temperr;
};
```

Figure 6: Sub-structure example, SOC user event data structure

2.6.3.2 Array Access

The structure passed to userland has no pointers; all arrays are referenced by offset. The xghei library takes care of adjusting the base addresses so that fields that can be interpreted as arrays are absolute.

For example, the structure that captures MCU events is as follows:

```
struct xghei_uspace_mcuevt {
        uint32_t        num_mcu;
        uint32_t        num_mcu_err_reg;
        uint16_t        mcu_select;
        uint16_t        mcu_pad;
        int32_t         mcu_sel_error;
        uint64_t        mcu_reg_sel_err;

        /*
         * The following are actually two dimmensional arrays
         *      mcu_reg[num_mcu][num_mcu_err_reg];
         *      mcu_reg_err[num_mcu][num_mcu_err_reg];
         * Use
         *      xghei_mcu_reg(xghei_uspace_pmdevt *, pmd)
```

```
                *       xghei_mcu_reg_err(xghei_uspace_pmdevt *, pmd)
                * to access each row (second dimension)
                */
               uint64_t         mcu_reg;
               uint64_t         mcu_reg_err;
};
```

Figure 7: MCU event structure

Eliminating pointers from the structures passed between user and kernel space means the same code can execute on both 32 and 64 bit user space binaries.

To read the array that tracks the errors accompanying the reading of the selection register, the code would be:

```
int32_t *mcu_reg_sel_err;

mcu_reg_sel_err = (int32_t *)mcu_evt->mcu_reg_sel_err;
```

Figure 8: Array access example

To access the two dimensional arrays that record the data for each MCU, there are two helper routines:

```
static inline uint16_t *
xghei_mcu_reg(struct xghei_uspace_mcuevt *mcu_evt, uint32_t mcu)

static inline int32_t *
xghei_mcu_reg_err(struct xghei_uspace_mcuevt *mcu_evt, uint32_t mcu)
```

Figure 9: User space array helper routines

The helper routines should be used to access the actual data for each mcu. For example,

```
uint16_t *mcu_reg;
int32_t *mcu_reg_err, *mcu_reg_sel_err;

for (mcu = 0; mcu < mcu_evt->num_mcu; mcu++) {
      mcu_reg = xghei_mcu_reg(mcu_evt, mcu);
      mcu_reg_err = xghei_mcu_reg_err(mcu_evt, mcu);

<.......>
      }
}
```

Figure 10: Two dimensional array access example

}

2.7. User Library

To receive an event in user space, applications can link with a dynamic library that wraps the HEI. The library provides an asynchronous event notification via a `netlink` socket to a POSIX Thread that is created within the application.

To provide the illusion of true asynchronous event delivery, the library creates a POSIX Thread that blocks on a `netlink` socket read. When an event is delivered, the thread reads the data from the socket, and then invokes the callback handler that was specified by the application. Therefore, applications using the library must also link with the pthreads and netlink libraries.

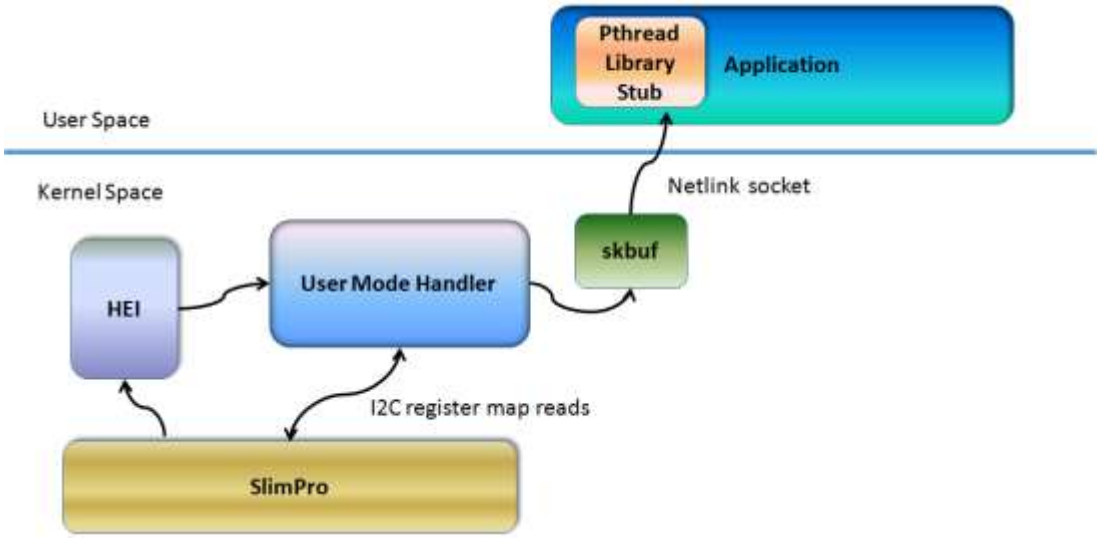

Figure 11: User space callouts

To simplify the reading and writing of the I2C register map, the API also includes functions to read and write the SlimPro register map. For example, to read the SOC temperature sensor, consider the following code fragment:

```
/*
 * From table 6, sensor registers
 */
#define XGENE_SOC_TEMP          0x10

uint16_t val;
uint64_t handle = <HEI HANDLE>

error = xghei_regread16(handle, XGENE_SOC_TEMP, &val);
```

Figure 12: Reading SlimPro register map

2.7.1. Code Example

To receive a callback when the SOC_HOT threshold is crossed, an application can register and specify the `XGHEI_SOC_HOT` event mask.

The following is a simple test program to verify the operation of the SOC_HOT callback. In this example, the application has no additional context other than the event handle, and only reads the current temperature during the callback.


```
/*
 * Test program to receive overtemps errors
 */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdint.h>
#include <string.h>
#include <uapi/linux/xgene_hei.h>


/*
 * From table 6, sensor registers
 */
#define XGENE_SOC_TEMP          0x10

struct userctx {
        uint64_t handle;
};

char *Progname;

void usage(void) {
        printf("%s: [-d debug]\n", Progname);
        printf("Receive callbacks for SOC\n");
}

/*
 * Test callback. The 'unused' parameter corresponds to
 * the generation number that is used in kernel callbacks
 */
static void hei_callback(void *context, uint64_t alert_mask,
        uint64_t unused, struct xghei_uspace_event *evt)
{
        struct userctx *ctx = context;
        struct xghei_uspace_socevt *soc_evt = &evt->soc_evt;
        int error;
        uint16_t val;

        printf("Received callback with context %p handle 0x%lx mask 0x%lx\n",
                context, (unsigned long)ctx->handle, alert_mask);
        printf("SOC temp %d err %d\n", soc_evt->soc_temp, soc_evt->soc_temperr);

        /*
         * Compare with current temp
         */
        error = xghei_regread16(ctx->handle, XGENE_SOC_TEMP, &val);
        if (error == 0)
                printf("SOC currently at %hd\n", val);
        else
                printf("Can not read sensor data, error %d\n", error);

}
```

Figure 13: SOC callback code

The body of the program registers for the callback, and then does nothing until the user presses a key.

```
int
main(int argc, char *argv[])
{
        int     error, opt;
        uint64_t mask;
        struct userctx *ctx = NULL;
        char buf[4];

        Progname = argv[0];

        while ((opt = getopt(argc, argv, "d")) != -1) {
                switch (opt) {
                case 'd':
                        xghei_setdebug(4);
                        break;
                default:
                        usage();
                        error = EINVAL;
                        goto out;
                }
        }

        ctx = malloc(sizeof(struct userctx));
        if (ctx == NULL) {
                printf("Can not alloc mem for user ctx\n");
                error = ENOMEM;
                goto out;
        }
        memset(ctx, 0, sizeof(struct userctx));

        printf("Registering for SOC_HOT\n");

        mask = XGHEI_SOC_HOT;
        error = xghei_register_alert_cb(ctx, mask, hei_callback, &ctx->handle);
        if (error) {
                printf("Can not register, error %d\n", error);
                goto out;
        }
        printf("Received handle 0x%lx for context %p\n",
                (unsigned long)ctx->handle, ctx);

        /*
         * Wait for events
         */
        printf("Press return to exit ");
        (void)fgets(buf, sizeof(buf), stdin);

        error = xghei_unregister_alert_cb(ctx->handle);
        if (error) {
                printf("Can not unregister, error %d\n", error);
                goto out;
        }

out:
        if (ctx)
                free(ctx);
        return error;
}
```

Figure 14: Main body of SOC HOT test program

2.7.2. Demonstration of the HEI

The above program can be compiled and linked with the user level HEI API shared library. As documented in D4.1, the HEI exposes a number of test injection methods. Writing to the debug register 0xF4 will trigger a variety of errors (see the D4.1 HEI documentation for complete details). The SOC overtemp is value 0x01. When the HEI hwmon driver is loaded, writing to this register as follows:

```
i2cset -y 1 0x2f 0xf4 0x01
```

will produce the following output, illustrating the correct output of the HEI from the firmware, through the Linux stack, and all the way to user space.

```
[root@tigershark hei]# ./sochot
Registering for SOC_HOT
Received handle 0x33780030 for context 0x33780010
Press return to exit
Received callback with context 0x33780010 handle 0x33780030 mask 0x2
SOC temp 42 err 0
SOC currently at 42
```

Figure 15: HEI logging output example

2.7.2.1 Compiling

Assume that the HEI shared library has been installed in `/usr/local/lib/xgene`. The library makes use of netlink sockets and POSIX Threads, so additional libraries are required during the link phase. If the test program is named `soc_hot.c`, the command to compile and link would be as follows:

```
gcc -Wall sochot.c -o sochot -l nl-3 -l nl-genl-3 -l pthread \
    -L /usr/local/lib/xgene -l xghei
```

To execute the code, the dynamic linker must be told where the library lives:

```
export LD_LIBRARY_PATH=/usr/local/lib/xgene
```

3. QEMU

For initial development, the platform was the virtual machine emulator QEMU running on CentOS 7.3 on an APM Mustang board. The ARM virtual machine 2.8 of QEMU was extended to emulate an APM Mustang board by adding entries to the ACPI Differentiated System Description Table (DSDT). The emulated DSDT was taken from a running Mustang board using the `iasl` utility on `/sys/firmware/acpi/tables/DSDT`.

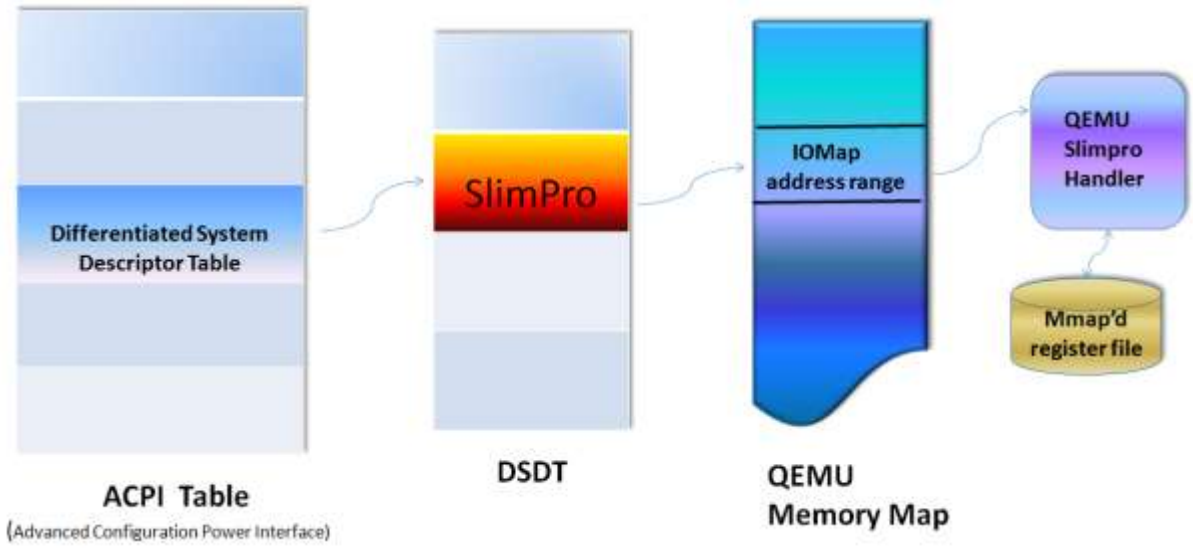

Figure 16: QEMU ACPI linkage

In the target VM, any IO access to the SlimPro IOMap address range would trap into QEMU. This provides a straightforward mechanism to emulate the HEI register map and SlimPro functionality. All I2C register map reads and writes are supported, but there is only one bank of registers (not one bank per core) for the sake of simplicity.

3.1. QEMU Testing

The register map is exported as an `mmap'd` file on the host. An application on the host (or even the Linux `hexedit` utility) can modify the register map to emulate any type of error. Given that most errors require many registers to be set, a test injection tool was developed that allowed for certain types of errors to be injected. Once the register map has been modified, a new command in the QEMU monitor (accessed via `CTL-A` from the command line window where QEMU was started) allows for a mailbox interrupt to be generated to the target VM.

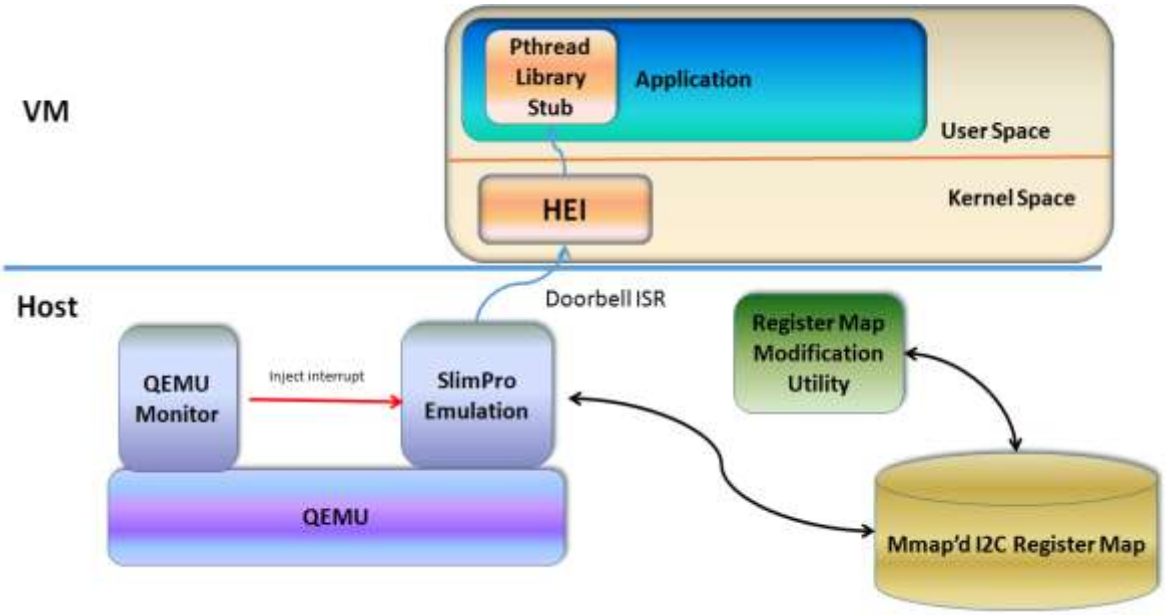

Figure 17: QEMU testing

4. SlimPro Modifications

SlimPro has been enhanced to support multiple outstanding events. The events can be acknowledged by the Linux kernel in any order. The depth of the buffer that tracks outstanding events is 16, which means that 16 different events can be tracked at any given time. If the firmware detects that a duplicate event occurs while the Linux kernel is still processing a previous event, the duplicate event is dropped.

Events can be posted to the buffer at any time. The firmware has an event loop that polls the various event sources (MCU, PMD, LC3, etc.). When the event sources have been polled, it then walks through the buffer and generates an alert for each event that the Linux kernel has indicated it wants to receive (via a Mask Register). Events that are not “interesting” to the kernel are discarded.

An event notification is generated by ringing a dedicated platform doorbell (doorbell 5, as described in the DSDT ACPI table). The Linux kernel receives the message, and then reads the corresponding registers (as documented in the D4.1 HEI spec) to determine the cause of the event.

The firmware has also been enhanced to allow the injection of errors to test the functionality of the API. The specific errors and registers are documented in the D4.1 HEI document.

4.1. Buffering

There are 16 buffers in the firmware used for storing events prior to delivery to Linux. The firmware buffer is reused as follows:

1. When the event is read out of the buffer and written to the I2C registers during the polling loop
2. For PMD 'selection' events, when the I2C register for PMD events is read, 0x81-0x84
3. For MCU 'selection' events, when the I2C register for MCU events is read, 0x91-0x93

For events that require a ‘selection’ write (MCU and PMD) to select which object to retrieve information about, it is possible that the buffer will be immediately re-used and overwritten once it is read.

There is no connection in the firmware between event delivery and the re-use of buffers. To ensure that all consumers of events, particularly of MCU and PMD events see the same data, it is recommended that the buffering capabilities provided by the API and exported by the `xgene_hei_hwmon` driver are used.

4.2. Interrupt Coalescing

If multiple events are pending in the firmware, only one interrupt will be generated. This is to reduce the overhead of generating and responding to interrupts. The linux driver is expected to read all the information for all events that are signalled upon receipt of the interrupt.

5. API Prototypes

There are two modes of operation. In kernel space, notification is provided while running in the context of a kernel thread. In user space, the application must link against the HEI dynamic library, and notification is provided in the context of a POSIX Thread. The function prototypes are the same in kernel space and user space.

The library defines are located in `<uapi/linux/xgene_hei.h>`.

5.1. Alert Notification APIs

The HEI Alert Notification APIs include the following interfaces:

5.1.1. xghei_evtfunc_t

Name

`xghei_cbfunc_t` – Call-back function to be called when notifying an alert

Synopsis

```
typedef void (*xghei_cbfunc_t)(void *context, uint64_t alert_mask,
uint64_t gen, struct xghei_uspace_event *evt)
```

Description

This function pointer is to be provided to the X-Gene HEI Alert Notification API on registration. On the occurrence of an alert, the HEI Alert Notification API will call the function passing in the appropriate context `context` associated with specific alert registration. The `alert_mask` contains a bitmask of the alert sources previously registered for that is part of this alert notification.

5.1.1.1 User Space

When running in user space, the event has already been acknowledged by the kernel drivers. The application is not responsible for clearing the event in the hardware. All pertinent information is gathered by the kernel driver in the `xghei_uspace_event` structure prior to issuing the callout.

The callout is asynchronous with respect to the execution of the application program. To provide this illusion of user-space interrupts, a POSIX Thread is spawned internal to the HEI library to listen for events. When an event is detected, the thread will invoke the routine provided at registration time.

5.1.1.2 Kernel Space

The callback happens in the context of a kernel thread after receipt of the interrupt from the hardware. The kernel module must clear the event as soon as possible to prevent blocking further notifications. The `xghei_uspace_event` pointer is NULL for kernel modules.

The `gen` parameter corresponds to the interrupt instantiation; it is monotonically increasing and is incremented upon receipt of every interrupt from the firmware. For example, if there are two back-to-back PMD events, the generation number would be used to distinguish between them.

The generation number is used to obtain the buffer holding the event data for this interrupt.

Return value

None

5.1.2. xghei_register_alert_cb

Name
`xghei_register_alert_cb` – register for specific alert sources in the system.

Synopsys
```
int xghei_register_alert_cb(void *context, uint64_t alert_mask,
 xghei_cbfunc_t cbfunc, uint64_t *handle)
```

Description
The `xghei_register_alert_cb` call is used to register for X-Gene alert notifications. Context represents an opaque handle in the application. The `alert_mask` is a 64-bit mask as described section 5.2. indicating which alert sources the application is interested in.

The application can register for all alert sources or a subset of the alert sources as specified in the `alert_mask`. When an HEI alert happens, the API will trigger a call-back to the supplied call-back function using the registered context. There is no restriction on the number of registrations for a single application process; however, for practical purpose, depending on how the application is designed, it should self-limit the number of registration instances to a few. For example, one instance to handle power or thermal related alerts and another instance to handle errors.

Registering with an alert_mask set to zero is legal; the assumption is that the alert mask will be updated at a later time via `xghei_set_alert_mask()`.

Return value
This function returns zero on success or a non-zero Linux error code. An opaque handle for the application to use in subsequent accesses to other APIs is returned in the `handle` parameter. An error would occur if there is no supplied context or callback. In the case the API decides that the supplied context is already used, the supplied `alert_mask` replaces the existing mask.

5.1.3. xgene_unregister_alert_notification

Name
`xghei_unregister_alert_cb` – unregister from HEI alert notification

Synopsis
```
int xghei_unregister_alert_notification(uint64_t handle);
```

Description
The application calls this function to unregister itself from the HEI notification.

Return value
This function returns 0 on success, or a non-zero error code. For example, an attempt to unregister while callouts are still active will return `EBUSY`. In this case, when the callout handler clears the last event that it is processing, then all event registrations for the handle will be removed.

5.1.4. xghei_set_alert_mask

Name
`xghei_set_alert_mask` - Change the alert_mask previously set with the initial alert notification registration or a previous `xghei_set_alert_mask`.

Synopsis
```
int xghei_set_alert_mask(uint64_t handle, uint64_t alert_mask)
```

Description
The application calls this function to change the alert sources. If the `alert_mask` is 0, this would temporarily disable alert notifications to this instance of alert registration.

Return value
This call returns 0 on success or a non-zero error code if an error occurred. An error would occur if the provided handle is invalid.

5.1.5. xghei_get_alert_mask

Name
`xghei_get_alert_mask` – query for which alert sources the application has registered for.

Synopsis
```
int xghei_get_alert_mask (uint64_t handle, uint64_t *alert_mask)
```

Description
The application calls this function to retrieve the bitmask of alert sources it currently registers for.

Return value
This call returns 0 if successful or a non-zero error code if an error occurred. An error would occur if the handle is invalid or the provided pointer to the alert_mask is NULL.

5.1.6. xghei_clear_alert

Name
`xghei_clear_alert` – clear the alert after processing is done

Synopsis
```
int xghei_clear_alert(uint64_t handle)
```

Description
The kernel module calls this function to acknowledge the receipt of the alert from the X-Gene HEI. In user space, API has already cleared the event and applications do not need to invoke this function. This function is not exported to user space.

Because there is a single (global) alert from hardware while there might be multiple registrations for the alert, the alert itself will not be cleared until the last client acknowledges it. Before that happens, no alert would be generated.

There are two ways that an alert that has been notified to a client is cleared: when the client application explicitly calls this function, and when the application unregisters from receiving alerts. Therefore, it is critical that once the alert is processed, the application must ensure that it calls this function as soon as possible.

Return value
This call returns 0 if successful or a non-zero error code if an error occurred.

5.1.7. xghei_regread16

Name
`xghei_regread16` – read a 16 bit I2C register

Synopsis
```
int xghei_regread16(uint64_t handle, int regnum, uint16_t *data)
```

Description

Read the 16 bit I2C register specified by `regnum` into the location specified by `data`. This helper function provides a mechanism for users to manipulate the Xgene SlimPro register map.

This helper routine is provided to make it easier to read/write the HEI registers. The HEI registers are only accessible via I2C.

Return Value
This calls returns 0 if successful or a non-zero error code if an error occurred.

5.1.8. xghei_regwrite16

Name
`xghei_regwrite16` Write a 16 bit I2C register

Synopsis
```
int xghei_regwrite16(uint64_t handle, int regnum, uint16_t data)
```

Description
Write the 16 bit I2C register specified by `regnum` from the location specified by `data`.

This helper routine is provided to make it easier to read/write the HEI registers. The HEI registers are only accessible via I2C.

Return Value
This call returns 0 if successful or a non-zero error code if an error occurred.

5.1.9. xghei_buf_fill_t

Name
`xghei_buf_fill_t` Event buffer fill callback

Synopsis
```
typedef void (*xghei_buf_fill_t)(uint64_t handle, uint64_t event_mask,
struct xghei_evtbuf *evtbuf);
```

Description
Fill in the event buffer `evtbuf` with data from the firmware for all events specified in `event_mask`. The event buffer is the kernel variant of the user level `struct xghei_uspace_event` and is described in `include/misc/xgene_hei.h` in the kernel tree.

Return Value
None

5.1.10. xghei_get_evtbuf

Name
`xghei_get_evtbuf` Get an event buffer

Synopsis
```
int xghei_get_evtbuf(uint64_t gen, uint64_t handle, xghei_buf_fill_t
fill_func,struct xghei_evtbuf **evtbuf);
```

Description

Get the event buffer holding the data for the events corresponding to the interrupt `gen`. If no buffer exists, invoke the callback routine `fill_func()` to fill it.

Buffers are referenced counted, and must be released with a call to `xghei_put_evtbuf()`.

Return Value
This call returns 0 if successful or a non-zero error code if an error occurred.

5.1.11. xghei_put_evtbuf

Name
`xghei_put_evtbuf` Release an event buffer

Synopsis
```
void t xghei_put_evtbuf(uint64_t gen);
```

Description
Event buffers are reference counted. Every buffer that is obtained via `xghei_get_evtbuf()` must be released with a corresponding call to `xghei_put_evtbuf()`.

Buffers are identified by the generation number `gen`.

Return Value
None.

5.2. Alert Mask definitions

In the above interfaces, the alert_mask is a `uint64_t` bitmask. Each alert bit definition is mentioned below. For example, to set the alert mask to be notified of all PMD and memory errors, the mask would be constructed as follows:

```
uint64_t mask = XGHEI_PMD_ERROR | XGHEI_MEMORY_ERROR;
```

```
#ifndef BIT_ULL
#define BIT_ULL(nr)             (1ULL << (nr))
#endif

/*
 * The masks for the HEI are constrained to a 64-bit number.
 *
 * It is recommended to use the bit mask rather than the ordinal. All
 * APIs expect a bit mask.
 */
#define _XGHEI_SOC_OVER_ORDINAL         0
#define XGHEI_SOC_OVER                  BIT_ULL(_XGHEI_SOC_OVER_ORDINAL)
#define _XGHEI_SOC_HOT_ORDINAL          1
#define XGHEI_SOC_HOT                   BIT_ULL(_XGHEI_SOC_HOT_ORDINAL)
#define _XGHEI_SOC_VR_HOT_ORDINAL       16
#define XGHEI_SOC_VR_HOT                BIT_ULL(_XGHEI_SOC_VR_HOT_ORDINAL)
#define _XGHEI_PMD_VR_HOT_ORDINAL       17
#define XGHEI_PMD_VR_HOT                BIT_ULL(_XGHEI_PMD_VR_HOT_ORDINAL)
#define _XGHEI_DIMM_VR_HOT_ORDINAL     18
#define XGHEI_DIMM_VR_HOT              BIT_ULL(_XGHEI_DIMM_VR_HOT_ORDINAL)
#define _XGHEI_DIMM_HOT_ORDINAL         32
#define XGHEI_DIMM_HOT                  BIT_ULL(_XGHEI_DIMM_HOT_ORDINAL)
#define _XGHEI_PMD_ERROR_ORDINAL        48
#define XGHEI_PMD_ERROR                 BIT_ULL(_XGHEI_PMD_ERROR_ORDINAL)
```



```
#define _XGHEI_MEMORY_ERROR_ORDINAL     49
#define XGHEI_MEMORY_ERROR              BIT_ULL(_XGHEI_MEMORY_ERROR_ORDINAL)
#define _XGHEI_L3C_ERROR_ORDINAL        50
#define XGHEI_L3C_ERROR                 BIT_ULL(_XGHEI_L3C_ERROR_ORDINAL)
#define _XGHEI_PCIE_ERROR_ORDINAL       51
#define XGHEI_PCIE_ERROR                BIT_ULL(_XGHEI_PCIE_ERROR_ORDINAL)
#define _XGHEI_SATA_ERROR_ORDINAL       52
#define XGHEI_SATA_ERROR                BIT_ULL(_XGHEI_SATA_ERROR_ORDINAL)
#define _XGHEI_OTHER_ERROR_ORDINAL      53
#define XGHEI_OTHER_ERROR               BIT_ULL(_XGHEI_OTHER_ERROR_ORDINAL)
#define _XGHEI_ACPI_STATE_CHANGE_ORDINAL 54
#define XGHEI_ACPI_STATE_CHANGE
BIT_ULL(_XGHEI_ACPI_STATE_CHANGE_ORDINAL)
#define XGHEI_MAX_ALERT         (sizeof(uint64_t) * 8)
```

5.3. Performance considerations

The following provides an estimate of the performance overhead of using the Hardware Exposure Interface framework for error detection. Estimates are based on X-Gene 2 system timing.

- Sampling rate of voltage and power sensors: this will depend on different hardware design, but it would take approximately 400ms to update a sensor
- Time it takes to write an HEI register via kernel /dev/i2c interface: 0.10 seconds
- Time it takes to read an HEI register via kernel /dev/i2c interface: 0.05 seconds
- Time it takes to receive an alert notification and determine the nature of the alert: this would involve reading the GPI Data Set register (0x60) to determine which data set contains the errors.
- The best case is for data set #0-#2 where the application just needs to issue one more read. The estimated total time to handle this alert would be about 0.10 seconds plus overhead

The worst case is for data set #3 (register 0x64) in the case of PMD/CPU or ACPI state change alerts. The application would need to read GPI Status register for PMD at 0x70 to determine which PMD, write PMD selection register at 0x80 and then read the data from the PMD Error Registers at 0x81-0x84. The estimated total time in this case would be 3 reads (0.15 seconds) plus one write (0.10 seconds) or 0.25 seconds in total.

