RAS specifications

Robert Mustacchi

Version 0.1.0

Table of Contents

1. Purpose

The purpose of this document is to define requirements, suggested implementations, and future visions for system reliability, availability, and serviceability. The primary focus of this document is on current x86 platforms. Design and requirements will be focused on x86 platforms and leverage existing x86-specific terminology. While this document often uses Intel specific terms, it should be equally apply to AMD systems. Further, this document should be broadly applicable to other architectures and platforms.

Reliability and availability of x86 based platforms has clearly been a major focus of vendors and there is already a wide body of features and options already available. While these components are crucial, serviceability and debugability require more attention than has traditionally been given for operating at a larger scale.

The document first describes the guiding principles and values as well as the general design principles ascribed by Joyent. The next part of the document surveys how existing systems are used and then is organized based on the major components that exist in the system and how such manageability and serviceability is expected to function. Finally, the document covers specific proposals and open problems.

2. Guiding Principles and Values

- Transparent

We want platforms to be transparent in the information that they provide. Information should not be hidden from systems software. A firmware module should not interpose and consume events or change their meaning.

- Self-Describing

We want systems to be self-describing. This means that the platform can programatically describe what is plugged into, where it is, and how this correlates with information that humans use when servicing the system.

- Consistent

The way that systems are designed here should be consistent. This means that the self-description provided should be the same. The way that errors are emitted and the content of that information should not change across different instances of the same system. This information may change between generations of systems, but even different systems of the same generation should strive to be as consistent as possible.

- Operating System First

While there is a trend towards a firmware first methodology in RAS, ultimately all information, when possible, should be delivered to the operating system. This does not mean that the platform firmware is not a part of the process of interacting with and dealing with RAS, but rather that the ultimate decision maker should be the operating system. The OS has more knowledge about the

following issues:

1. The impact of a given error or event
2. The intent of the operator of the system

- Standards Where Possible

We want what we build upon to be standardized wherever possible. Not only does this allow more tooling to be built around these RAS capabilities, but it also allows the vendors that we're working with to turn around and offer these capabilities to their other customers. While some systems may take more advantage of these capabilities than others, a vendor should not be limited due to some specific proprietary aspect of this.

- Error Localization

When an error is detected in the system, the system must be able to translate this to a discrete component. For example, if a correctable memory error were to occur, it is insufficient to say that it occurred at a given physical address. The error must have sufficient detail as to make it clear as to which DIMM and bank this occurred on. This applies to different components in the system.

3. General Design Notes

3.1. Post-Mortem

At Joyent, an important goal is that we can root cause every failure that occurs in the field. As such, whenever a failure occurs, it is important to capture as much information about the state at the time of the failure as possible. The goal is that from a single failure, it should be possible to either root cause or make significant progress in understanding why a failure occurred. This is especially important for transient or rare failures. The going in assumption should always be that a failure will never be reproduced in a lab setting.

Take for example software. When a program crashes, a core file will be generated which can be inspected after the fact. The same is true for the operating system. When a fatal error occurs, the operating system will produce a crash dump. This crash dump will contain a copy of the state of the system at the time the crash occurred.

While such mechanisms may be harder for hardware systems, creating captures of core memory or register state is very useful for understanding failure and why such issues occurred.

Wherever it is possible and practical, hardware systems should support similar capabilities.

3.2. In-Band versus Out-of-Band Management

The system must support both in-band and out-of-band management. Both are equally important to the operation of the system. Out-of-band management is essential for understanding the system, particularly when the system itself is not functioning for whatever reason.

However, it is not sufficient. In general the operating system needs to be able to leverage and drive

the same mechanisms that are driven through out-of-band management. The operating system and systems software in general needs in-band management this so that it can trigger events and obtain information to better inform operators. Ultimately the systems software can be used to better express operator intent, making it powerful.

For example, if systems software has decided to offline a hard drive due to a predictive failure, then systems software needs the ability to toggle the chassis service and/or the drive bay LED.

The more information that the OS has, the more of an ability it has to make informed decisions and better correlate multiple, disparate error sources. This can allow better predictive failure, automated responses, and more detailed impact assessment for operators.

3.3. Alerting and Thresholds

The system on the whole, whether through firmware or systems software, needs the ability to have alerts on the health of components in the system. Many such alarms rely on crossing a specific threshold.

Take for example, a temperature based alarm. If a temperature crosses a specific threshold, then an alarm is triggered. That threshold itself must be configurable.

When systems firmware triggers these alerts and thresholds, the means of delivery of such alerts need to be configurable. However, a required means of delivery is to both systems software and to the IPMI system event log via the service processor. Operators may want the platform to be able to deliver event notifications to external systems even when the operating system isn't running.

3.4. Testing

Vendors must be able to provide tools that can run on a normal production system to inject errors so that the platform's reaction to such errors can be explicitly demonstrated. Specialized debug systems or builds of firmware will only be accepted for cases where it is impossible to test otherwise. However, all of the differences between versions must be documented and code differences, provided.

4. Role of Existing Systems Management Frameworks

The purpose of this section is to introduce existing systems management frameworks and give a high-level overview of how they fit into the broader specifications.

4.1. SMBIOS

The System Management BIOS (SMBIOS) is a specification maintained by the DMTF that provides information related to the management of the system. SMBIOS provides a series of tables that describe the system. These tables are placed in main memory by platform firmware and is discovered either by UEFI services or by searching BIOS memory.

SMBIOS is used to map between devices that are plugged into the system, the manufacturing data about them, and provides information about where in the chassis or system board that device is.

Importantly, SMBIOS provides a snapshot of information about the running system at boot time. Unfortunately, SMBIOS does not have a good means of updating itself. This means that it should not be used for any dynamic information that can be changed while the system is running. For example, at this time the CPUs, as constructed on a standard system, are not hot-pluggable. As such, it is appropriate for SMBIOS to include information about them. However, if you take the example of a hot-swappable power supply, then information about that power supply that can change should not be included, for example a serial number.

4.2. IPMI

The Intelligent Platform Management Interface (IPMI) provides a framework for managing a system and provides a substantial amount of information via both in band and out of band management. The in band management features are especially important as they allow one to discover information about the system ranging from FRUs to sensors.

Generally, IPMI is implemented on a baseband management controller (BMC) or another lights out system. Importantly this means that IPMI controllers have access to the system in a separate way from the primary operating system and can function regardless of whether or not the operating system is running.

The operating system generally can access IPMI information over the IPMI-defined KCS bus.

Throughout this document we will refer to the unit that provides IPMI or similar services as a service processor (SP). This is being used as a generic term so as not to constrain it to an IPMI-specific interface.

4.3. ACPI

The Advanced Configuration and Power Interface (ACPI) specification provides a means for x86 and ARM based systems to enumerate and discover hardware devices that are present in the system beyond the original IBM PC BIOS calls. ACPI provides tables of information and supports running dynamic methods provided by the platform in response to events. It is used for runtime power management and as a means for driving various firmware actions.

In addition, ACPI also encodes information about various physical aspects of devices and the chassis. For example, ACPI is used to map ports on a USB root controller together and provide information about the type of port that exists at the other end.

4.4. SES

SCSI Enclosure Services (SES) is a specification for talking to an enclosure processor. This enclosure processor is generally used as part of a backplane and can be used to manage sets of SATA and SAS disks plugged into a storage backplane. This can determine information about which devices are present and toggle LEDs related to those devices.

In addition, SES can represent information related to a storage enclosure such as power supplies and fans.

4.5. SMBus and PMBus

The system management bus (SMBus) is a bus that is similar to i2c and is used to connect various peripherals together. For example, DIMMs are connected to SMBus and can provide information about the system through the JEDEC specified data pages over the interface.

Historically, the SMBus on a system has been the world of systems firmware. While it is not expected that the operating system will take over the SMBus, it is expected that the operating system will be able to take advantage of it to get additional information.

The power management bus (PMBus) is a set of specifications that live on top of SMBus. It is used to control and get diagnostic information about different power devices.

4.6. Redfish

Redfish is an emerging technology designed to function as a new interface for systems management and replace many of the functions that IPMI is used for. While it is expected that some of the out of band management functionality will be provided via redfish, today Redfish is missing various in-band programmatic interfaces. Without those, it is not a viable replacement for how IPMI is being used for in-band uses.

5. Components

5.1. CPUs

5.1.1. Error Detection

CPUs should be able to detect and optionally correct errors that occur in the CPU. Today, many of these events are highlighted through specific error architectures. On Intel x86 CPUs this is the Machine Check Architecture ([glos-MCA]). Examples of these events include, but are far from limited to:

- Cache Errors
- Data Bus Errors
- Internal State Errors
- Temperature Threshold Errors

All CPUs should support the current x86 MCA framework and emit events to the error banks as configured. The current MCA framework and events is documented in the Intel 64 and Intel IA-32 Architectures Software Developer's Manual [sdm].

On some systems there is the ability for systems firmware to be notified by the EMCAv2 architecture through an SMM trap. Firmware is allowed to configure and manage this; however, in

all such cases it **must not** consume the event. The event must be delivered to the operating system.

The systems firmware should not make the determination that a given MCA event can or cannot be handled by the operating system. For example, if an uncorrectable memory error is delivered to the operating system, then the systems firmware must not intervene and assert that this cannot be handled. It is up to the operating system to determine what page that memory error occurred on and be able to take the appropriate action. For example, the operating system may decide, based on policy, that if the memory error occurred in a user process that it should kill that process. Otherwise, if it occurred in a page owned by the kernel, the operating system may opt to take down the system.

For any level of error that occurred in this way, the service processor (SP) must log such information in the system event log with enough details to localize the error to a specific device and cause. It should be possible for the operating system to sync up with the SP and determine events that were generated or occurred while it was not running, because, for example, the system was being booted by systems firmware.

If an error is considered fatal to the platform in such a way that it cannot be handled, then the error must be delivered to the SP. The SP should determine whether or not it is possible to deliver an NMI to the operating system to get a crash dump. If it does, then it must provide some means of the operating system being able to get access to this information. The SP or some other part of the system, such as the management engine restarting the system should always be the last resort.

There exist classes of CPU errors which cannot be delivered by the MCA architecture because the CPU is in such a bad state that it is not possible for it to operate. While it may be possible for such an error to be delivered to another socket, given that the system is already in an undefined state, it is not recommended that this occur. Instead, the system should deliver it to the SP which should take care of resetting the system and ensure that the information is recorded and in an easy to obtain way from the operating system after the fact.

When such an error occurs it is tantamount that the SP gather as much information about the system as it can and store it as a form of crash dump.

5.1.2. Identification

Today, CPUs are identified in two different ways. They're identified by computers through the values returned in CPUID which on current generation systems are based on the xapic and x2apic identifier. When working in the context of the operating system, humans use these logical IDs as well.

Technicians on the other hand look at CPUs in terms of FRUs, which is a socket. Technicians utilize the silk screen labeling. It is expected that the system board provides silkscreens for the CPUs. If the CPUs are being described in any kind of out of band management ala Redfish or IPMI, it is expected that it will use identical labels.

Both of these means of identifying a CPU are important; however, we need a programmatic way to map between these identifiers from the running operating system. This is important, because CPU MCA events will come in and refer to a specific logical (CPUID based ID) for a CPU. If a CPU needs to be replaced in the field, the technician needs to understand what the corresponding socket is.

Today there does not exist a standardized way to do this. To facilitate this, we need to introduce a new means of accessing this information. Basic CPU information is already available in the SMBIOS specification [1: See Processor Information (Type 4), Section 7.5 of [smb-spec]]. We propose to extend this information to provide a means for mapping between the silkscreen socket information on the system board and the information that is provided by the processor via CPUID. For more information, see Proposal 1: CPU to Socket Silkscreen Mapping.

Automatic Replacement

When a fault occurs on a CPU and it is replaced, we'd like to be able to know that the CPU has actually been replaced by a new unit without an operator's manual intervention. To do this, we need something such as a serial number that can be associated with a CPU. This serial number like information does not need to be generally accessible as the original portion that was available via CPUID in the Pentium III was. Technologies such as Intel PPIN can often serve as a source for a synthetic identifier. For more information on how this is used, see RFD 137.

5.1.3. Firmware

CPUs have firmware that comes in the form of microcode. It is expected that all CPUs that are in use will support some form of online microcode update. While it is nice for BIOS and other firmware updates to start the CPUs with a given microcode, that cannot be relied upon and existing OS based microcode updating tools must be able to function long after the operating system and processor have been started.

5.1.4. Sensors

There are many different sensors that exist for the CPU. Some of them, such as temperature sensors, are exposed as PCI devices that exist on a CPU socket, core, or logical processor basis. Others are available as MSRs. Sensors which exist through IPMI must be correctly attributed to the CPUs. They should not be attributed to the system board or other peripherals.

5.2. DIMMs

This section covers all devices that are present in current DIMM slots as seen by the processor. This should be the same whether they are traditional volatile memory devices or they are one of the many forthcoming non-volatile memory devices.

5.2.1. Error Detection

One of the primary concerns around memory devices is error detection. These memory devices are capable of generating correctable and uncorrectable errors. Today the errors on all such DIMMs are delivered through the memory controller to the processor as an MCA event.

All such generated events must be delivered to the processor such that the operating system can record them and take action on them. This should be the case for both correctable and uncorrectable errors on DIMMs. In no circumstances should a correctable or uncorrectable error be transformed into an NMI or another level of error, absent some subsequent failure in the system.

Systems firmware should record these events in the system event log in the SP; however, they should not interpose on them or modify them in such a way that the operating system cannot notice them. If the error rate reaches a certain threshold, then systems firmware may perform hysteresis on writing to its system event log; however, if this is performed, the amount recorded must be noted. When reading these events, both the physical address as well as the DIMM and bank should be recorded.

The aforementioned hysteresis should only apply to the act of systems firmware writing in its log. Systems firmware **must not** interpose or apply this hysteresis for events delivered to the operating system.

Error Localization

An important aspect of DIMM errors is the ability to localize these errors. To that end, it is expected that the operating system will have the ability to translate the addresses to the particular device. On Intel based systems, it is expected that the operating system will use information from the CPU's memory controller (IMC on current Intel systems and UMC on current AMD systems) to direct and manage these errors.

5.2.2. DIMM Channel Disablement

Some platforms will opt to disable a DIMM channel for a particular reason during the boot process. This may be because an error occurred during the discovery phase, an error occurred during the training, or errors occurred during memory test. In all such cases, the platform must log a message to the system event log to notify the operator that such an event occurred with detailed information about both the physical address as well as the channel, DIMM, and bank information.

Secondly, the running operating system must be able to discover and distinguish the case where a DIMM channel was disabled by the platform from a case where the DIMM is not plugged in. While it may be possible to rely on the memory controller for the basic state information, that is insufficient. The operating system must be able to determine **why** the channel was disabled.

Similarly, if a subset of a DIMM, for example, a given rank, was disabled during training or some other part of the process, that information should be recorded and observable.

For more details on the challenges with this, please see Issue 1: DIMM Disablement.

On the whole, it is preferable that the system be able to boot up with a reduced DIMM count than the system refuse to start until a faulty DIMM has been removed or replaced.

5.2.3. Identification

The next critical problem that we face with DIMMs is being able to get information about the DIMM. This ranges from the serial number to its capacity.

The system should provide information about DIMMs in SMBIOS by creating type 17 entries in SMBIOS. This captures a reasonable amount of data about the DIMM. In addition, JEDEC has standardized various SPD (Serial Presence Detect) data for DIMMs. This is generally the source that much of the data from SMBIOS comes from.

There should be one SMBIOS type 17 record for every DIMM slot on the system. Note that the number of DIMM slots on the system may be less than the upper bound on implemented channels on the system. There must not be an SMBIOS type 17 entry for such a DIMM slot when there is no corresponding physical slot.

SMBIOS provides a Location Tag which is useful for being able to determine where a given DIMM physically is. However, this is not sufficient. Critically we are missing a programmatic way of mapping between an SMBIOS type 17 entry or SPD data for a DIMM and the corresponding information from the memory controller.

SPD data today provides no form of location information — which makes sense as it is coming from the EEPROM of the DIMM. Dynamic SPD data (such as sensors) can be correlated to an SMBIOS type 17 entry through a DIMM's serial number.

This lack of mapping between the memory controller level data and the SMBIOS/SPD level data is problematic. For a tentative solution to this, please see Proposal 2: DIMM Identification.

5.2.4. Firmware

It is not expected that any DIMMs today have firmware that is required to be upgraded. However, for upcoming upcoming non-volatile DIMMs, a means of identifying and performing firmware upgrade on the devices from the running operating system is required. It is more important that in-band firmware information can be communicated, with the eventual expectation that full in-band upload and download of firmware can be performed.

5.2.5. Sensors

DIMMs expose sensors through their SPD data in the temperature page. In addition, the memory controller may have additional sensors that exist in the system. The SPD sensors should be accessible by the system over SMBus. If sensors are being added at an IPMI layer, then the entity IDs and other information must make it clear which DIMM it corresponds to and this should be able to be correlated to SMBIOS, memory controller, and SPD data. Further, in IPMI sensors for DIMMs must exist under the DIMM records themselves.

5.3. System Board

The System Board (often times referred to as the mother board) is a nexus for all of the different parts of the system. It accepts power, has sockets for the CPU, DIMMs, and expansion devices. There are many headers on it which allow it to be connected to additional devices.

5.3.1. Silkscreen labels

All connectors and slots on the system board must be labelled with a silkscreen. The labelling should be clear and it should be obvious as to which component it is referring to.

In particular, when referring to expansion slots such as PCI express, it should be designed in such a way as a full card does not obstruct the silkscreen, if possible.

It should be possible to determine which silkscreen refers to which components programatically

from the running system. This may occur through SMBIOS, IPMI, ACPI, or some other systems management interface.

5.3.2. Slot and Header Population

An important thing that the system needs to be able to determine is when slots or various headers are in use and if so, what they are. This covers the following components:

- CPU Sockets
- DIMM Slots
- PCIe Expansion Slots
- Fan Headers
- USB Headers
- Misc. I/O headers used for Serial, etc.

In all cases, systems software should be able to determine which slots are in use and which slots are not. CPUs, DIMMs, and PCIe devices and their slots are all covered in their own respective sections

For the remaining headers, while there is information that the system can transmit methods things such as the SMBIOS type 8 port connector information, it can be hard to make use of those records to understand what has and hasn't been wired up. For example, a USB header may be in use to connect to a set of auxiliary USB ports that are not a part of the system board. For example, front USB ports on the chassis. An open question is how do we map those potentially in-use headers. See [issue-port-header] for more information.

5.3.3. Built-in Ports

Another challenge that we have is the ability to map several of the built-in ports to labels that exist. For example, systems today have a combination of USB, Ethernet, VGA, Serial, and more ports sitting on them. These ports are built-in to the system board and exposed, generally, through the rear of the chassis or on the system board inside of the chassis. Importantly, unlike other ports, these are not built into the chassis and connected to a header. Instead, they are part of the system board FRU.

These ports provide a variant of Issue 3: Header/Port mapping where by we need to have a well defined way to understand which ports correspond to which logical devices. While SMBIOS provides type 41 Onboard Devices Extended Information [2: See section 7.42 of [smb-spec]], it does not provide a good way for us to think of those items which are ports. In addition, for those items which do not employ a PCI-like bus, device, and function, there is no way to map those items to operating systems visible devices.

An important goal for this is to be able to determine whether a given USB port is internal to the chassis or external. This will be used by operators to create different policies around trusting devices plugged into different ports.

5.3.4. BIOS Management

The system board is often home to the system BIOS or UEFI implementation. Traditionally, the only way to manage these interfaces and observe them has been through being able to enter the BIOS while the system is booting. This is a major serviceability impediment.

Instead the following information needs to be available through at least out of band management and preferably through in band management as well:

- Current BIOS settings, including boot order
- Current firmware revisions
- Ability to install new BIOS firmware revisions and prepare them for a future boot
- Ability to toggle between primary and backup firmware images
- Ability to set BIOS settings.

All of these equally apply to UEFI based systems as well as BIOS based systems.

5.3.5. Sensors

The system board should have the ability to enumerate all of the sensors that are unique to it. These should generally be exposed through IPMI. However, if they are not exposed via IPMI, then the platform should provide some means of enumerating these sensors and how to find them in some dynamic fashion, whether that's through ACPI, discovery through SMBus, or some other system specific method. These must be publicly documented.

It is important that such sensors be enumerated correctly. Sensors that are not a part of the system board itself or refer to it, should not be enumerated for it. For example, a DIMM temperature sensor should not be enumerated under the system board, but logically must refer to its corresponding DIMM such that we can tell which DIMM slot this refers to.

5.3.6. SMBIOS

There are two different SMBIOS information types that are important to be able to program on the running system. These are the System Information (type 1) [3: See Section 7.2 of [smb-spec]] and Base Board (type 2) [4: See Section 7.3 of [smb-spec]]. It should be possible to overwrite the various string fields. The serial number information should be accurate.

5.3.7. Other Firmware

It is possible that the system board has other devices that have firmware on them. For example, many system boards may deploy CPLDs or other devices to help drive and manage them. As with other firmware, the revisions of all such components should be made consumable to the operating system and if possible the firmware should be modifiable by the operating system. It is acceptable that this work goes through another component on the system to manage this. For example, if it has to be communicated to the Intel ME, service processor, or some other controller such as the innovation engine.

5.4. Chassis

The chassis is the first thing that an operator having to service a system will have to interact with. When thinking about the chassis, one has to consider that operators will be dealing with racks upon racks of gear and that means that small differences between systems can be easily overlooked unless strongly stated.

5.4.1. Exterior Labeling

While we've made a focus about silkscreens from the perspective of a system board and mapping CPUs and DIMMs, the chassis has its own labeling that we need to consider.

In particular the following components should be labelled:

1. Expansion slots should be labelled with information that corresponds to the internal slots.
2. Removable entities such as power supplies should have labels affixed next to the bays that are visible and clear both when the bay is and isn't populated.
3. USB ports should be labeled in a way that corresponds to labels that the system can understand.
4. An RJ-45 port that is dedicated to lights out management should be explicitly labelled as such.
5. System identification information such as part numbers and serial numbers should be present on a non-removable part of the system. If possible, this should be accessible while the system is in its normal service position in the rack.

Importantly, this labeling shares many aspects of the same set of problems as Issue 3: Header/Port mapping and this labeling will likely end up being a part of that.

5.4.2. Interior Chassis Labelling

The interior lid of the chassis should have diagrams that explain how to perform common service operations. This is useful as a way to remind data center technicians of the steps that need to be taken when operating on a system. However, this should not be construed as a replacement for a full service manual for the system.

5.4.3. Required Tools

A technician should be able to service a majority of the chassis without requiring tools. This implies the following:

1. The chassis should be able to open and close without requiring tools (an optional tool-based lock is permitted).
2. All hot swappable components (fans, PSUs, disks) must be serviceable without tools. Tools may still be required to insert or remove a disk from a tray; however, none should be required when performing the actual swap.
3. Any metallic or plastic based components in the chassis which are designed to direct airflow or move components must be modifiable without tools.
4. Components that require screws, such as the CPUs or the system board, should be kept to a

minimum.

The technician should be able to complete all work with either a #1 or #2 Phillip's head screwdriver. If another type of screw is required, such as a small hex wrench, then the chassis must have a predefined place such that a small tool can be stored and provided with the chassis.

5.4.4. Identification LEDs

A system should have the ability to have an identification LED that is controlled through IPMI as the chassis indicator. At a minimum the identifier should have two modes:

- Off
- Identification (switching between and on and off pattern at a fixed frequency)

Ideally the identification LED should support additional modes and be able to operate in the following fashion:

- Off
- Solid color indicating health
- Alternate solid color indicating service required
- Blink functionality for either color

5.4.5. Firmware

If the chassis has any firmware that belongs to its FRU then that must be reported through to the platform in some way. It should be clear that said firmware belongs to the chassis FRU.

5.5. CPU based PCIe root ports

The PCIe root ports are a part of the CPU that is often configured explicitly by the BIOS. In Intel parlance this is often called the IIO while on AMD it is often called the NBIO. Unlike the rest of the CPU, the PCIe root ports do not deliver events through the MCA architecture, but instead through PCI express advanced error reporting (AER).

The PCIe root ports external design specifications and PCIe generally define different classes of errors which are broken into categories based on whether they are corrected, recoverable, or fatal.

Any error that is generated by a PCIe device should be forwarded to the operating system and in almost all cases should not result in a fatal error being received by the operating system. Even if a card has been surprise removed, systems software should make the determination as to whether or not the system should be taken down.

When fatal errors are encountered, the platform firmware must make sure that the contents of the error are firmly written down in the system event log in the event that the operating system is not able to properly record information as a crash.

An NMI should not be injected into the operating system unless there is no other option and there is no chance that the system can continue coherently from such a point.

Systems vendors must enumerate all events that will cause an NMI to be injected into the running system by the PCIe root ports.

5.6. PCH

The platform controller hub (PCH) is a series of devices that provide general support for features on the system. These run from USB controllers, to Ethernet, to SMBus, and more. These devices are generally represented as a series of PCIe devices that exist; however, they may exist in other forms.

It is expected that every device accurately support the PCIe AER specifications and be able to deliver AERs when events happen. If a device is being used by the platform, but the OS has not itself set up a driver for this and accessed it, then the platform firmware should still be able to receive AERs and forward them onto the OS to notify it that a given component is having issues.

If a non-PCIe based component is being leveraged and has an error, the systems vendor must define how the operating system will be notified about this. If this is a recoverable or corrected error that does not require operating system intervention, then the platform must still notify the OS that this has occurred.

If the error is instead fatal, the firmware of the system must record detailed error information to non-volatile storage. If the firmware must take down the running system as a result of this, then it is obligated to have a mechanism for informing the operating system through in-band signalling that such an event has occurred on the next run.

5.6.1. Firmware Upgrades

One challenge with the PCH is that it houses two different additional processors which each house their own firmware: the Management Engine (ME) and the Innovation Engine (IE).

The system must provide methods such that systems software can determine the firmware revisions of these devices. In addition, while there are security and complexity concerns for in-band firmware upgrades of these components, some form of firmware upgrade must be provided. It is required that such an upgrade be delivered while the system is in service and deferred until the next system reset.

5.7. PCI and PCIe Devices

The most common form of expansion card used in systems today is a PCI express device. Many of the devices on the PCH (even if they use a DMI link to the device) are exposed as PCI and PCI express devices. Connectivity to storage and the network is almost always provided through PCI express devices.

5.7.1. Location and Identification

All PCIe devices must be identifiable as existing in a given location in the system. For devices that are part of the CPU and PCH this is optional, though it is recommended that it is made clear in some form that this is where they reside.

For devices which do not support hotplug (usually found in traditional expansion slots), this information should be obtainable through a SMBIOS type 9, System Slot [5: See section 7.10 of [smb-spec]], entry. Specific PCI devices should be mapped to the location string through the bus, device, and function of the device.

For U.2, NGSFF, and Intel ruler form factor PCIe devices or other form factors that are being used for hot-pluggable NVMe devices, then the platform must provide a way to map this information to the bay on the chassis that these are accessible on. It is recommended that each bay have an SMBIOS entry so that the platform can enumerate how many hot pluggable PCIe bays exist. For more information on such requirements, please see the Backplanes section.

5.7.2. Firmware

PCIe devices may contain their own firmware. If so, these devices should conform to the notes in the general discussion on Firmware. A means of native firmware upgrade is required.

NVMe Firmware Upgrade

NVMe devices have additional requirements around firmware upgrade. All NVMe devices must implement the following optional commands:

1. Firmware Commit [6: See section 5.11 of [nvme]].
2. Firmware Image Download [7: See section 5.12 of [nvme]].

NVMe devices should support more than one slot for firmware.

5.7.3. Hotplug

Devices that are not standard expansion cards should support hotplug and surprise removal. This generally means all U.2, NGSFF, and other more recent NVMe based devices. However, the rules that follow apply to any device. Here, when we say hotplug, we refer to the idea of a surprise removal. That is, a device which does not follow the requirements for orderly removal as discussed in [pci-hp].

When such a device is placed in the proper slot, the system must report this by setting the 'Hot-plug surprise' and the 'Hot-Plug Capable' bit of the Slot Capabilities Register [8: See section 7.8.9 and table 7-18 of [pcie]] to one.

PCIe hotplug should follow the normal specification and allow an interrupt to be posted to the operating system when such events occur. Such a notification must not be hidden from the operating system by the platform firmware.

Devices should ensure that any acknowledged activities are still completed, even in the face of surprise removal. For example, a NVMe device that has acknowledged a write and then been removed from the system, must still ensure that it is flushed to stable storage.

5.7.4. Sensors

Devices should provide access to sensors to gauge device temperature. If active cooling is employed,

the device must provide a way to understand the health of said cooling devices. Further, if there are failures or any thresholds triggered, those must be presented in some form to the device driver as discussed in the following sections.

This is not limited to just temperature and cooling. This should also cover electrical health (such as voltage, current, wattage, etc.) and other internal aspects of the device.

5.7.5. Errors

While the PCIe AER capability is optional, it is required for all discrete expansion cards. Systems software will configure all devices to generate AERs.

Internal Device-specific Errors

It is possible that devices may have internal conditions that can be triggered. For example, the device has a cache that has a fatal ECC error. Devices must expose this information in some form. The operating system should receive notifications of these events through one of the device's standard MSI/MSI-X interrupt vectors.

It must be clearly specified how the operating system can recover from such a state. For example, by power cycling the device. In addition, as much information about the internal error should be provided in some fashion to the operating system to ease eventual RCA procedures.

5.7.6. Reset Capabilities

Where possible, devices should support a per-function reset capability. If a device only supports a single function and does not support virtual functions, then this capability is not required and power cycling should be able to occur at a device level.

5.8. Backplanes

The backplane of a system is often a discrete FRU that exists in the system that is used to connect all of the storage (whether SATA, SAS, NVMe) in the system to a set of ports on either the system board or distinct expansion cards. The backplane is wired to a number of 'bays'. Each bay can house one device, though some bays can house multiple devices. The bays are usually built into the chassis.

Today, most backplanes come in two different forms. They are either considered passive devices or active devices. In the case of passive devices, they exist primarily to pass through the electrical wiring of storage devices. Active backplanes have components that actively participate and may modify the transmitted data. For example, this is performed by SAS expanders and PCIe switches.

5.8.1. LEDs

The backplane should support multiple LEDs on a per-bay basis which include:

- An activity LED
- A service LED

The service LED should be able to indicate several different states:

- Blinking for identification
- Multiple solid colors to distinguish between different health states such as operational or service required.

Importantly all LEDs **must** function regardless of physical presence of the device in a backplane slot.

The current LED state should be retrievable from the system. LED settings do not have to persist across power cycles. However, non-activity LED settings should not change because of device insertion or removal.

LED Control Methods

The methods for controlling the LEDs on a given platform must be well defined.

For SATA/SAS devices behind SAS expanders, the LEDs must be controllable through SES.

For SATA/SAS devices which are directly attached through a passive backplane, the controller must be able to control this whether a SAS HBA or the on-board AHCI controller.

For NVMe style PCIe bays, the bays should support independent control through the PCIe slot control register [9: See section 7.8.10 of [pcie]]; however, if this would not allow for per-bay control, then instead alternative means of toggling these LEDs through IPMI should be documented.

5.8.2. Power Control

Each bay and slot must have independent power control. This power control should be available through the basic protocol methods wherever possible and also available through IPMI or similar management.

5.8.3. Identification

Identification of devices can be tricky due to the use of both passive and active backplane technologies which indicate how much is known or not by the platform. Regardless, a system should be devised such that it is possible to determine the following information about a given bay:

- Is a device physically plugged into the bay?
- What is the vital product data (VPD) for device plugged into the bay?

The operating system must be able to query this information. There are several different schemes and ways that this is done today:

- SCSI Enclosure Services (SES)

SES provides a programmatic way to describe the bays in a given backplane or chassis and gives basic information about them such as the SAS WWN.

One unfortunate complication with SES is that it is usually only placed on systems when a SAS expander is also used. While sometimes a SAS expander is a necessary evil, it can also often get in the way.

- Static Mappings

Another way that this is done is by knowing which phy lanes are wired into which slots on the backplane. While this is an option, it is not the preferred method due to the fact that a wiring mistake between the HBA and the backplane or having the HBA placed in a different PCIe expansion slot can end up causing issues with this enumeration.

Identification Challenges with NVMe

Importantly, the rise of U.2, NGSFF, and other form factor NVMe devices mean that we have new challenges in the backplane for identifying devices.

With the rise of the NVMe-MI specification, system vendors should be able to leverage this information and expose it with the relevant bay mappings, allowing a system to be able to fully flush out the topology information. Unfortunately, this is not standardized today.

Further, even when PCIe switches are employed, it is important that the manipulation of the devices from a PCIe perspective is independent. One should ensure that if the LEDs are advertised in the slot capabilities register [10: See section 7.8.9 of [pcie]], then each bay is independent.

Identification Challenges with AHCI

On occasion, systems will have SATA devices wired up to to a backplane and directly connected to an AHCI controller. In these cases, the ahci controller should allow for controlling the LEDs through the AHCI Enclosure Management specification [11: See section 12 of [ahci]]. If possible, the controller should be rigged up such that it can receive messages and be used to transmit the VPD data from the device in the bay.

However, even if this is in place, then we still need a means of being able to map between which bay is which, as most of the commonly implemented methods are not self-describing. While they can be solved with the static mappings described above, it would be better if a self-describing system was used for this.

5.8.4. Firmware

Whether the backplane is active or passive, there will likely be firmware involved. Even passive backplanes may have a CPLD to help drive the LEDs on the system.

As with other firmware modules, whether upgradeable or not, the system should expose what the revision of the firmware is and the specific FRU that it belongs to.

5.9. SAS Expanders

Systems that have SAS expanders must conform to the following expectations:

- The FRU that the expander is a part of should be well defined
- The SAS expander must support firmware download and enumeration through standard SAS commands

Further, it should be possible to obtain statistics about the state, count, statistics, and more of all of the phys that the expander is attached to. Further, if active SFF cables with transceiver information are present, then it should be possible to retrieve that information from either end.

SAS expanders are often part of the backplane in systems and will be the component that implements and provides SCSI enclosure services (SES) which allow for LED control and identification of devices.

5.10. SATA and SAS Drives

SATA and SAS drives, whether based on rotating or solid state media are a mainstay of the systems that we are building today. However, they both have unique failure modes due to either the mechanical or electrical nature of the medium.

5.10.1. Error Reporting

Regardless of medium, the drives should report standard SATA and SAS errors in response to commands as per the corresponding command sets.

When internal events and errors occur, they should be logged in a device-specific manner. These event and error logs should be exposed through either a standard log page or through a vendor-specific interface.

When vendor specific interfaces are used, it is required that one be able to write open source tools to collect said log information, even if the tooling to interpret it is not open source (though it is recommended that vendors pursue that path whenever possible).

Flash Wear Leveling

One particular challenge is the endurance of drives. Solid State drives should report their wear leveling metrics through the standard SATA and SAS log pages. If more details are available through a vendor-specific interface, then information related to that should be provided.

5.10.2. Firmware Upgrade

SATA

Firmware upgrade must be possible for SATA devices through the ATA8-ACS DOWNLOAD MICROCODE command. Devices must support the word 0 (feature) subcommand 03h (Download with offsets and save microcode for immediate and future use). Ideally, future revisions of SATA devices will have support for an alternate download procedure ala the SAS versions described below. Further drives should try to comply to the general firmware upgrade guidelines.

Where possible, vendors should also supply the ability to obtain the current running binary firmware image from the drive.

SAS

Firmware upgrade must be possible for SAS devices through the SCSI SPC WRITE BUFFER

command. Drives must support mode 07h (Download microcode with offsets, save, and activate). Drives should support a combination of modes 0Dh (Download microcode with offsets, select activation events, save, and defer activate), 0Eh (Download microcode with offsets, save, and defer activate), and 0Fh (Activate deferred microcode).

Where possible, vendors should also supply the ability to obtain the current running firmware image from the drive through either a standard or vendor-specific SCSI command.

5.11. Transceivers

Transceivers in the system are expected to implement and conform to the SFF specifications for obtaining information about the transceiver. All transceivers are expected to implement page 0xa0 of the appropriate specification. This means SFF 8472 for SFP, SFP+, and SFP28 devices, SFF-8436 for QSFP devices, and SFF-8636 for QSFP28 parts.

Any device that accepts such a transceiver, whether a networking device, a host bus adapter, or another component must provide a means of querying this i2c information. It is not expected that this information be on the SMBus, PMBus, or another platform-wide i2c bus.

It is recommended that devices supporting SFF-8472 implement page 0xa2 and that devices implementing SFF-8636 implement all of the optional sensor information such that the component's health can be better understood.

While we expect all such transceivers to be of a high quality and available, if a transceiver fails, the only requirements are that the containing component is able to indicate the failure. If the component is able to offer additional, detailed failure information then that should be made available to the component. NICs should be able to clearly indicate when they're no longer able to communicate with the transceiver through normal link state change events and then based on firmware indicate whether or not the transceiver is physically present or not.

Dealing with the availability of a given transceiver should not be done at the transceiver level, but at the device level that is leveraging the transceivers. For example, multiple NICs should be employed and availability provided at that layer of the stack whether through layer two methods such as LACP or layer three routing protocols such as BGP and OSPF. Similarly, if transceivers are being used to connect external storage HBAs, then multipathing of some form should be employed.

5.11.1. Identification

All transceivers should have the manufacturing information filled in. This includes the vendor and serial number information.

5.11.2. Firmware

If the transceiver possesses a UFM of some form, then it is required that the UFM information be exposed in some way by the transceiver that the operating system can obtain. In addition, the operating system should be able to upload and download new firmware to devices.

5.12. Chassis Power Supplies

Different chassis have different forms of support for power supplies. In some cases the power supplies in question are hot-swappable and in other cases they are built-in to the chassis itself. The power supplies themselves have a wide range of different information that is relevant. For example, these ranges from the supported ranges of voltages, currents, and watts at a high level, down to different probes that exist in different parts of the power supply.

The platform must be able to determine the following information about power supplies:

- How many power supplies exist in the system?
- Are the power supplies separate FRUs?
- Are the power supplies hot-swappable?
- For removable power supplies, how many slots exist in the system?
- For removable power supplies, how many power supplies are physically present?
- How many power supplies are plugged in?
- What input current and voltage ranges is the power supply designed for?
- What input current and voltage is the power supply running at?
- What output current, voltage, and wattage ranges is the power supply designed for?
- What output current, voltage, and wattage ranges is the power supply operating at?
- What more detailed power information is available?
- What is the manufacturing vital product data?

Today, a combination of SMBIOS and IPMI should be used to supply information about the power supplies in the system, their current state, and how many are present.

When the system has static power supplies that are either not their own FRU, or are their own FRU, but not hot-swappable, then the platform should correctly fill out the SMBIOS Type 39, System Power Supply [12: See section 7.40 from of [smb-spec]], information. The Power Unit Group information (offset 04h) must be accurate and represent which power supplies provide redundancy. In addition, due to the static nature of the power supplies in such a configuration, all of the power supply location, manufacturer, and other VPD information most be present.

In cases where the system has dynamic, hot-swappable power supplies, the SMBIOS information should be used to represent the number of such power supply slots that exist in the system and the capability of being hot-swappable by setting bit zero of the Power Supply Characteristics member at offset 0Eh in the SMBIOS type 39, System Power Supply, record. Further, such devices should provide location information. The platform should not provide basic manufacturing information via SMBIOS and instead should provide it through IPMI FRU records.

In addition, it is recommended that the platform expose PSU information over the power management bus (PMBus), allowing for deeper introspection by the runtime operating system. As dynamic properties should not be filled into SMBIOS if the power supply is hot-swappable, this implies that the various probes and cooling devices that are part of a hot-swappable power supply should not be enumerated in SMBIOS. Again, the rationale for this is that the SMBIOS information is

static and therefore cannot be updated to reflect the changes in the system's information.

In all cases, there must be a means to map between the IPMI records and corresponding SMBIOS entries. One way to do this is to use the slot label information.

5.12.1. Firmware

Modern power supplies are increasing in complexity. If a power supply has firmware, regardless of its upgradability, then that information must be exposed through either PMBus or an IPMI platform record.

A plan for supporting the firmware upgrade that is executable from a native environment is required.

5.12.2. LEDs

Some power supplies have LEDs that are present on the units. When such LEDs exist, the ability to toggle them and control the LEDs should be provided over some interface to the operating system. The LEDs should consist of the following different modes:

- Normal Behavior
- Blinking pattern for identification
- A color to indicate the device has been faulted

Ideally these should be exposed over IPMI or a similar systems management interface that can be used by the operating system in conjunction with the systems firmware such that the operating system can override or change the LEDs with firmware's knowledge.

5.12.3. Upstream Identification

Unlike other components of the chassis, a power supply exists as part of a broader fabric. A power supply may be plugged into a wall outlet or extension cord, but it is more commonly plugged into a power distribution unit (PDU) in a data center. A server should be able to determine which PDU a given power supply is plugged into and vice-versa. For more information on this, see Issue 2: PSU/PDU Discovery.

5.13. Fans

Active cooling is often a necessary and crucial part of the system. For any given fan in the system, we have a few different questions that we want to be able to answer:

- Is it present?
- Is it hot-pluggable?
- Where is it?
- Is it spinning?
- If so, how fast?

- What type of device is it?

The expectation is that the platform provides a way to explicitly determine the following items:

- How many fan slots exist?
- How many fan slots are populated?
- Which fan slots are serviceable?
- Which fan slots are hot-pluggable?
- What type of fans are used in what slots?
- What speeds are all fans spinning at?

One challenge with fan population rules is that many system boards have more headers for fans than a given chassis uses. This may cause it not to be possible to determine the number of expected fans in the system. In such cases, there must be enough identifying information present in the system, such that the running operating system or other management tools can determine whether or not a given class of system or a specific instance of it, should have a specific number of fans.

Unlike other devices, most fans do not have read only memory (such as an EEPROM) which can be used to deliver manufacturer information such as the name of the manufacturer, the serial number, or the part number. Therefore, when populating records for fans, it is understood that this information may not be available.

5.13.1. IPMI and SMBIOS

Every fan in the system should have a corresponding tachometer sensor provided by IPMI. Conventionally, these sensors should have entity IDs that try to relate it to the sets of fans that are expected to exist.

To break this into the two different phases we propose the following high level principles:

1. SMBIOS information should list what the platform expects to be present as part of its design.
2. IPMI should reflect what is actually present and have a way to map to a corresponding SMBIOS entry.
3. If a fan is not provided by the platform (such as on a PSU), then SMBIOS should not provide information about that; however, IPMI should still provide entity information to indicate what device it is a part of and provide server information where possible.

To make this more concrete, SMBIOS would be used to answer the question of how many fan slots exist in the chassis and label them. As well as indicate whether the fan slots are hot-pluggable or not. SMBIOS should not populate any of the information about RPM or other aspects of the system. There may be fewer SMBIOS slots than headers on the system board.

Instead, IPMI should be populated with sensor records that correspond to the different physical fan headers. In addition, the IPMI records should provide information to map the fan sensors back to a corresponding labeled, SMBIOS entry.

Through IPMI senors, it should be possible to distinguish for each fan, whether or not the fan is

plugged in and whether or not it is spinning in any way.

Finally, to cover cases like option 3, IPMI should make it clear what entity the fan belongs to. For example, a fan on a power supply or a fan that is part of an active CPU heat sink should make it clear that they map directly to those devices.

5.14. Service Processor

The service processor is a secondary processor that exists in the system that is used to help manage the running system and provide management to it when the main system is not necessarily running or able to run.

5.14.1. Management Interface Support

The service processor must support the IPMI 2.0 specification and should support a revision of the Redfish specification. The SP should support running IPMI services through the KCS bus. If that is not provided, then another form of in-band management must be documented by the vendor and implemented in the system.

If there are vendor-specific commands that are required for correct operation, those must be properly documented.

5.14.2. Network Support

The service processor must support being attached to the network through a dedicated Ethernet port. The device should support either static or dynamic configuration through something such as DHCP. If DHCP is used, the SP should support announcing itself through a DHCP client identifier in a way that makes it clear who the manufacturer of the service processor is and the class of system it is a part of.

Historically many dedicated and shared ports have had issues with negotiating speeds while the chassis was powered off or in the middle of a power cycle. The dedicated management port must always support negotiating full-duplex gigabit support through auto-negotiation as per IEEE 802.3 [ieee] regardless of the chassis's power status.

5.14.3. Firmware Upgrade

As per the general [firmware] section, the service processor should support some way of performing firmware upgrade while the main system is operable. This should preferably be driven through native code mechanisms.

5.14.4. Power Control

There are conditions where the service processor may need to be restarted due to conditions such as memory leaks, firmware upgrades, or as part of other troubleshooting steps. Such a power control should be available to the operating system.

5.15. Firmware

Firmware in systems is one of the more challenging parts of system serviceability and reliability. The chief issue with firmware is that it is often unobservable software. While the term firmware is used here, this also covers other kinds of devices such as PROMs that serve a similar function on devices.

We classify firmware into two different buckets:

1. Upgradable firmware modules
2. Static firmware modules

The first group represent those that can be serviced by the running operating system at run time of the system. The second group suggest firmware modules which cannot be serviced at runtime. These second group of firmware modules may still be serviceable by a technician or by replacing a FRU; however, from a practical perspective, that makes them unchangeable. A disk drive is an example of the first set, while a YubiKey is an example of the second set.

All firmware modules present in the system must support querying the current revision of the in use firmware as well as any other installed versions. If a given firmware module supports multiple slots (such as a primary and backup firmware module), then all versions must be discoverable.

Devices in the upgradeable set should have a means of having the operating system install new firmware onto the devices. Similarly, having the ability to capture the current image in use on the device is a desirable, but optional requirement.

5.15.1. Desired Upgrade Mindset

One of the particular challenges with firmware upgrades is the fact that interruptions in the upgrade can cause problems with devices. With that in mind, what we'd like to see is that all devices adopt something similar in spirit to a hybrid of the NVMe and SAS specifications.

Particularly, the model that we'd like to see is that for any given piece of firmware on the device, there exist at least two distinct slots which can hold independent copies or versions of a given piece of firmware. Finally, the device should support the ability to record a firmware revision to take effect on the next power on or reset.

These two features combined make it very easy for systems software (or any other service) to upgrade the firmware without interfering with any currently running service and account for failures that occur due to bugs with the systems software or other incidents such as a power loss. The ability to download and then set firmware to be updated at the next power cycle, provides a much more flexible and powerful system for administrators. This allows firmware upgrades to be rolled out at the same time; however, be applied to the running systems based on operator-defined timing. Further, it means that the heavy lift and time involved in validating the firmware or performing other device-specific work has already been done before the system cycles.

5.15.2. Per-Device Uniqueness

While it is recognized that this process will ultimately be specific to individual devices. The more

that this different logic can be made similar to one another, the better. If at a minimum vendors can have the same logical method of flashing the firmware, even if it involves device specific methods, then that will be a net improvement on the serviceability of the system.

5.15.3. Change Log

When a new firmware image is available, a detailed change log of what has changed between versions is required. If a given image contains sub-components, for example, a BIOS update that embeds a CPU microcode update, then the details of what has changed in those versions is also required.

5.15.4. Source Code

Traditionally, firmware has been delivered as closed source binary blobs. Where possible, the source code for said firmware should be made available to allow operators to try and better understand how the firmware is operating. This will greatly aide in the debugging process between the operator and the vendor. It is not expected that an environment for building this will necessarily be provided nor is it expected that the operators will want to create their own firmware revisions.

5.15.5. Debugability

Where possible, firmware should provide means for understanding what is happening in production. It is recommended that some set of statistics or a circular buffer of data be possible to retrieve by the operating system or service processor.

If possible, the firmware should keep detailed statistics about the device to help us understand and explore any notions of why a failure has occurred, various internal error rates, or other aspects that describe the health of the device that might impact it reliability, availability, and should be used to help indicate impending service needs.

While interpreting said data may need vendor-specific, proprietary tools, the ability to capture the data must be able to be implemented in a separate and open-source fashion that can be incorporated as part of the broader platform. For example, for a SAS based hard drive, the data should be consumable via some kind of standard or vendor-specific SCSI command. When debugging production issues, all components are presumed guilty until proven innocent. This includes firmware. The goal of this debugging information should be to exonerate (or indict) the device.

5.16. Manuals

Manuals should be provided for all components in the system that describe the following information at a minimum:

1. The set of standards and specifications they adhere to. Whether from an electrical, programmatic, or legal perspective.
2. Detailed instructions on how to service the part or any other components in the system.
3. For devices with programmatic interfaces, a full programing specification should be included.

4. Manuals should be made available as searchable PDFs with table of contents metadata populated.

6. Specific Requirements

This section will be filled out in full detail at a later date.

7. Proposals

This section covers various requests for enhancements to the system. These enhancements should all be possible today through updated firmware. These are all just proposals, nothing more. Other means of solving these problems are perfectly acceptable.

The proposals are:

- Proposal 1: CPU to Socket Silkscreen Mapping
- Proposal 2: DIMM Identification

7.1. Proposal 1: CPU to Socket Silkscreen Mapping

The purpose of this proposal is to have a self-describing means to map a given CPU core to the socket as identified by the silkscreen on the system board. Today, most of these mappings are being performed for other devices through SMBIOS. For example, PCIe slots are labeled through a location tag in type 9 [13: See section 7.10 of [smb-spec]]. DIMMs are labeled through a location tag in type 17 [14: See section 7.18 of [smb-spec]].

Today, CPUs already have an entry in SMBIOS [15: See section 7.5 of [smb-spec]]. This type does have a location tag that indicates the socket; however, it does not include sufficient information to be able to map it to the set of logical CPUs the socket contains.

To that end, we propose that vendors add a vendor specific SMBIOS table entry to provide this mapping information, while it is proposed for addition to the SMBIOS standard. The core idea is to leverage the APIC IDs and provide a group of them to map to the existing handle for the processor. The following table is laid out in the spirit of the SMBIOS tables [16: See the 'SMBIOS structures', section 6, of [smb-spec]].

Offset	Name	Length	Value	Description
0	Type	BYTE	VARIES	OEM Specific Value
1	Length	BYTE	Varies	Varies, must cover at least one ID and the total length should cover all IDs
2	Handle	WORD	Varies	-

4	CPU Handle	WORD	Varies	SMBIOS handle of the CPU (Type 4) this entry corresponds to.
6	ID Type	BYTE	Varies	See subsequent table
7	ID Count	BYTE	Varies	Encodes the number of identifiers that follow. Must be at least 1.
8	IDs	Varies	Varies	Encodes processor IDs. Values vary based on ID Type.

By default, a given entry should include a list of the x2apic IDs as returned by the %edx register from the CPUID instruction Extended Topology Enumeration Leaf (0Bh) <sdm>> for every logical processor in the socket. This data is equivalent to performing a RDMSR instruction of the x2apic ID of the processor when it is executing in x2apic mode. At a minimum, one x2apic ID for every physical core should be encoded. That way, even if one physical core is broken or is not enumerating, it should still be possible to map another core of the same physical package to a socket.

Once a single logical processor of a core has been mapped to a socket, all logical processors in that core can be mapped together based on the cluster and package ID of the processor. For more information on these fields and their meaning, see Section 8.9.1 Hierarchical Mapping of Shared Resources of Intel Volume 3 [sdm]. In addition, if only a single core of a package is present in the SMBIOS record, then the other cores and logical processor can be mapped together in a similar fashion.

To facilitate older processor or future evolutions of the ID format, the table structure does not assume the use of the x2apic IDs and instead the type of ID is encoded. The defined ID types should be as follows:

Byte Value	**Meaning**
00h	Reserved
01h	Other
02h	Unknown
03h	x2apic ID
04h	apic ID
05h-255h	Reserved

The values 01h and 02h are only defined to match the SMBIOS standard. Per this specification, the value here should always be 03h. If the system is running with the x2apic disabled for some reason, then the use of the value 04h is allowed. In such a case, then the 8-bit local APIC IDs should be encoded. This id should be retrieved by reading the memory mapped address containing the ID as described in Section 10.4.6 Local APIC ID of Intel Volume 3 [sdm].

The following is an example of how this data would be encoded for a socket whose CPU handle was 46h and has four logical processors with an x2apic ids of 00h, 01h, 02h, and 03h.

Offset	Name	Value
00	Type	81h (This is a representative OEM / SYSTEM specific value)
01	Length	18h (This will vary for other systems)
02	Handle	2329h (This is a representative value)
04	CPU Handle	0046h
06	ID Type	03h
07	ID Count	04h
08	ID 0	00000000h
0c	ID 1	00000001h
10	ID 2	00000002h
14	ID 3	00000003h

If the table encoded local APIC IDs instead of x2apic IDs, then the previous example would instead be encoded as:

Offset	Name	Value
0	Type	81h (This is a representative OEM / SYSTEM specific value)
1	Length	0ch (This will vary for other systems)
2	Handle	2329h (This is a representative value)
4	CPU Handle	0046h
6	ID Type	04h
7	ID Count	04h
8	ID 0	00h
9	ID 1	01h
a	ID 2	02h
b	ID 3	03h

7.2. Proposal 2: DIMM Identification

The purpose of this proposal is to have a self-describing means to map a given DIMM as identified by a processor's memory controller to the information provided in SMBIOS type 17 records about the memory devices.

A DIMM is identified in SMBIOS in a few ways. It has a bunch of strings that describe where it is

physically which are designed to match the silkscreen on the boards. It also has a notion of a 'Set'. This set information is meant to try and describe groups of DIMMs that have to be in similar groups. However, 'set' information is rarely populated.

From the processor and memory controller perspective a DIMM is identified in a different way entirely. Instead, you're informed of the following information:

1. The Memory Controller
2. The Channel the DIMM is in
3. Which DIMM in the channel it is

The memory controller is usually interacted with in a socket-aware fashion. To identify which set of devices this maps to, there are two different approaches we can use:

1. Specify the PCI bus, device, and function of the memory controller.
2. Specify the socket.

At the moment, we have opted to use option one at this time to try and minimize our dependence on Proposal 2: DIMM Identification which would be required for option 2 and would need to change in the face of designs where the memory controller was not a part of the CPU die as it currently is.

To marry these up, we are tentatively proposing that an additional SYSTEM- or OEM- specific entry be added. This should have the following fields:

Offset	Name	Length	Value	Description
0	Type	BYTE	VARIES	OEM Specific Value
1	Length	BYTE	0bh	-
2	Handle	WORD	Varies	-
4	Memdev Handle	Word	Varies	SMBIOS handle of the corresponding memory device (type 17)
6	Bus Number	Byte	Varies	PCI bus number of the corresponding memory controller
7	Device/Function Number	Byte	Varies	PCI device and function of the corresponding memory controller
8	Controller	BYTE	Varies	Numerical ID of the Memory Controller

9	Channel	BYTE	Varies	Numerical ID of the Channel for the indicated Controller
a	Module	BYTE	VARIES	Numerical ID of the DIMM within the channel

One entry of this type should exist for every populated memory device on the system. In other words, it is not expected to exist if there is no DIMM present in a given slot, while an entry for type 17 will likely exist to indicate the existence of the slot.

For the three bytes of identification information, Controller, Channel, and Module, all values should be zero-indexed.

A system that has two sockets and therefore two memory controllers, with one DIMM per channel and three channels would have entries for the following tuples of Controller, Channel, and Module:

- 0, 0, 0 - Controller zero, Channel zero, DIMM slot zero
- 0, 1, 0 - Controller zero, Channel one, DIMM slot zero
- 0, 2, 0 - Controller zero, Channel two, DIMM slot zero
- 1, 0, 0 - Controller one, Channel zero, DIMM slot zero
- 1, 1, 0 - Controller one, Channel one, DIMM slot zero
- 1, 2, 0 - Controller one, Channel two, DIMM slot zero

On the other hand a system that had a single memory controller with two channels and two DIMMs per channel, all of which were populated would have the following entries:

- 0, 0, 0 - Controller Zero, Channel Zero, DIMM Slot Zero
- 0, 0, 1 - Controller Zero, Channel Zero, DIMM Slot One
- 0, 1, 0 - Controller Zero, Channel One, DIMM Slot Zero
- 0, 1, 1 - Controller Zero, Channel One, DIMM Slot One

8. Open Issues

This section covers major open design quesions that we have. The following sections describe the problems and give ideas to what a possible path formward might be; however, substantailly more work is required.

The Issues are:

8.1. Issue 1: DIMM Disablement

One of the current challenges with systems management is the case of when DIMMs are disabled at either a rank or channel level. The crux of the issue is that the OS needs to be able to tell the difference between a DIMM being disabled by the platform due to errors and the DIMM being removed. While both result in memory not being available, the action that operators will take as a result will change.

It's not clear if all of this can occur through the memory controller and IMC. The IMC is able to communicate population and disablement of channels and separately whether or not a channel is enabled or disabled.

While it would be ideal if this information was in sync with SMBIOS information, it is known that in some cases that the SMBIOS information is not well defined for a given DIMM if the DIMM channel has been disabled.

While all of this can tell us the current state of the DIMM, none of this can tell us **why** any of this occurred. It is not clear what a good way to get this information would be. In ACPI a table exists for recording memory device information the PMTT (Platform Memory Topology Table), which has information about the DIMM and can map it back to an SMBIOS related entry.

Unfortunately, based on analysis of a couple different systems, none of them have ever bothered to implement the PMTT table. Therefore it is unlikely that this makes sense as something to work with or extend.

It's possible that there may be a way to associate a FRU or SDR record for the DIMM with the cause over IPMI; however, the right way to move forward with this is unclear and is left open as something for us to be able to consider. One useful property is that once systems firmware has handed over control of the system then this data will be read-only and does not have to change in any way.

While it is tempting to use the SMBIOS memory error information handle that exist for a DIMM, this is not often used by the system and it doesn't necessarily give us total information as it doesn't provide a way of telling us exactly what has failed and why it was failed.

8.2. Issue 2: PSU/PDU Discovery

An open challenge that we have in the data center is a means of mapping between power supplies (PSUs) and their corresponding power distribution units (PDUs).

The goal here is something very similar to the link layer discovery protocol (LLDP) that is used to map between network ports and the corresponding switches (or any other system a NIC has been plugged into). As part of the exchanges, they are able to transmit a few different pieces of information such as system UUIDs, host and port names, etc.

We need a mechanism that isn't too dissimilar between the PSU and PDU. The set of data that is transmitted and retrieved needs to be controllable by the operating system itself.

The final complication of this is that this cannot be implemented by requiring additional cabling in

the system. The system needs to be able to function over the exiting power cables and transmit that information over it. The basis for which, already does exist in systems.

8.3. Issue 3: Header/Port mapping

Another open challenge that we have is between the ability to map between logical devices that the operating system sees, the headers on the system board, and the ports that exist in the chassis that operators use.

Unfortunately, the information that SMBIOS contains falls a little bit short today. For example, the challenge that we have is to map between the USB ports that exist in the chassis with those that are seen logically by the USB controller. Take as an example, the following location information:

```
ID    SIZE TYPE
10    18   SMB_TYPE_PORT (type 8) (port connector)

  Location Tag: J3A1

  Internal Reference Designator: J3A1
  External Reference Designator: USB2
  Internal Connector Type: 0 (none)
  External Connector Type: 18 (USB)
  Port Type: 16 (USB)
```

Here we have a way of mapping between the header and the port. However, we don't have a way of mapping these back to the logical entries that the operating system sees. Figuring out how to perform this mapping, is an open question.

There are a lot of different ways to approach this and that varies on the component in question. For example, with USB ports, we may be able to go through and leverage the existing ACPI information as a way to build this mapping.

However, for other devices we still need to figure out how we should perform similar mappings and where these additional and auxiliary mappings should exist. Whether it be SMBIOS, ACPI, IPMI, or some other service that's available at run-time.

9. Revision History

Version	Release Date	Changes
0.1.0	30 April 2018	• 'Silkscreen labels' is denoted with incorrect header. • Added 'Automatic Replacement' section to '5.1.2. Identification'. • Various typos fixed.

0.0.1	23 March 2018	• Initial Revision

References

- [acpi] Advanced Configuration and Power Interface (ACPI) Specification http://www.uefi.org/sites/default/files/resources/ACPI%206_2_A_Sept29.pdf Version 6.2 Errata A September 2017
- [ahci] Serial ATA Advanced Host Controller Interface 1.3.1. https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/serial-ata-ahci-spec-rev1-3-1.pdf
- [ieee] IEEE Standard for Ethernet 802.3-2015. IEEE Computer Society.
- [ipmi] Intelligent Platform Management Interface Specification Second Generation v2.0 https://www.intel.com/content/www/us/en/servers/ipmi/ipmi-second-gen-interface-spec-v2-rev1-1.html Document Revision 1.1 October 1, 2013
- [nvme] NVM Express http://nvmexpress.org/wp-content/uploads/NVM-Express-1_3a-20171024_ratified.pdf Revision 1.3a October 24, 2017
- [pcie] PCI Express Base Specification Revision 3.0. November 10, 2010.
- [pcie-hp] PCI Hot-Plug Specification. Revision 1.1 June 20, 2001
- [pmbus] Power System Management Protocol Specification http://www.pmbus.org/Specifications/CurrentSpecifications Revision 1.3.1 13 March 2015.
- [rfd-137] RFD 137: CPU Autoreplacement and ID Synthesis https://github.com/joyent/rfd/tree/master/rfd/0137
- [smbus] System Management Bus (SMBus) Specification http://www.smbus.org/specs/SMBus_3_0_20141220.pdf Version 3.0 20 Dec 2014
- [sdm] Intel 64 and Intel IA-32 Architectures Software Developer's Manual. https://software.intel.com/en-us/articles/intel-sdm. March 2017.
- [smb-spec] System Management BIOS (SMBIOS) Reference Specification. https://www.dmtf.org/sites/default/files/standards/documents/DSP0134_3.1.1.pdf Version: 3.1.1. 2017-01-12

Glossary

ACPI

Advanced Configuration and Power Interface. A specification that describes an extensible means for a platform to discover hardware components and make changes to the running system. Specifications available online <[acpi]>.

ACS

ATA Command Set. The primary set of commands that are required for ATA devices.

AER

Advanced Error Reporting. An optional form of error reporting provided by PCI express devices that allows for more fine-grained error reporting and response.

AHCI

Advanced Host Controller Interface. A specification for a SATA HBA controller. An AHCI chipset is commonly found on most current x86 platform controller hubs.

API

Application programmer interface.

APIC

Advanced Programmable Interrupt Controller. A class of interrupt controllers used on Intel x86 platforms. This is documented in Chapter 10 Advanced Programmable Interrupt Controller (APIC) in Intel Volume 3 [sdm]. See also, x2apic.

ASIC

Application Specific Integrated Circuit. An integrated circuit, commonly thought of as a chip, that has been created for a specific purpose. For example, a CPU or GPU. While an ASIC may be able to run programs on top of it, the logic of the ASIC itself is fixed, unlike a CPLD or FPGA.

BIOS

Basic Input/Output System. The BIOS is a basic operating environment in IBM compatible PCs which still is used today. This environment provides an interface to the platform firmware and specifies basic components and programming environments. The BIOS has almost completely been replaced by UEFI.

BGP

Border Gateway Protocol. A protocol that is used to exchange routing information on the Internet.

BMC

Baseboard Management Controller. A component in the IPMI specification that is a separate processor that can be used to manage the system.

CPLD

Complex Programmable Logic Device. A device that has a series of logic gates that can be programmed and reprogrammed. Often used to implement a small chip that does not make sense to produce as an ASIC.

CPU

Central Processing Unit.

DHCP

Dynamic Host Configuration Protocol. A protocol used to assign devices on an IPv4 network a dynamic address, automatically. A variant exists for IPv6 systems.

DIMM

Dual in-line memory module. DIMMs are used to provide volatile memory to the system.

DMI

Direct Media Interconnect. An Intel specific interconnect that is used to connect components

between the PCH and CPU.

DMTF

Data center Management Task Force. A standards body responsible for specifications such as SMBIOS and Redfish.

DDR

Double Data Rate. A data transfer technique used in DIMMs.

EEPROM

Electrically Erasable Programmable Read-Only Memory. A class of non-volatile memory that exists on devices, but optionally allows for bytes to be rewritten by the system.

EMCAv2

Extended Machine Check Architecture v2. An extension to the x86 MCA architecture that allows for firmware to intercept, consume, and enhance, MCA events.

FPGA

Field Programmable Gate Array. A reprogrammable logic device that is often used as a makeshift chip. Uses a different form of programmable logic than CPLDs.

FRU

Field Replaceable Unit. A distinct component of a physical system that can be replaced independently of other components by a technician in the field. Common examples are a hot-swappable power supply, a PCIe expansion card like a networking card, or a CPU.

GPU

Graphical Processing Unit.

HBA

Host Bus Adapter. A device which is used to bridge two different environments and send commands to devices on that bus. The most common HBAs are disk controllers. For example, a SAS HBA provides a mean for systems software to send SAS commands to devices on the SAS fabric.

i2c

Integer-Integrated Circuit. A bus format that is used for low-cost, simple, low-speed interconnects. i2c forms the basis for the SMBus system and various components in the chassis leverage an i2c style interface.

IEEE

Institute of Electrical and Electronics Engineers. A professional association. Maintains the Ethernet specifications.

IIO

Integrated I/O. A unit on an Intel CPU that interfaces with I/O expansion units such as PCIe. The AMD equivalent is the NBIO.

IMC

Integrated Memory Controller. Refers to the on-CPU memory controller used on Intel platforms. The AMD equivalent is the UMC.

IPMI

Intelligent Platform Management Interface. A specification that defines lights out management capabilities that may be delivered by a service processor or other firmware. Specifications available online [ipmi].

JEDEC

An international standards organization. Most commonly used in computing for the various DDR memory standards.

KCS

Keyboard Controller Style. A type of interface that may be used by the operating system to issue and retrieve IPMI commands.

LLDP

Link Layer Discover Protocol. A protocol maintained by the IEEE for network device discovery.

MCA

Machine Check Architecture. A part of the x86 architecture. It is the framework that is used to deliver notifications about classes of hardware errors to the operating system.

Motherboard

Often shortened to mobo. An alternate and somewhat more common name for system board.

MSR

Model Specific Register. A non-architectural register on the CPU whose actions vary based on CPU model.

NBIO

A unit on an AMD CPU that serves as a PCIe root port. The Intel equivalent is the IIO.

NIC

Network Interface Card. A peripheral card that provides networking capabilities, generally over Ethernet.

NMI

Non-Maskable Interrupt. Refers to a class of interrupts that the processor and operating system cannot disable. Often used to signal a fatal error condition in the system. Operators can inject NMIs through IPMI.

NVMe

Non-Volatile Memory Express. A specification for interfacing with non-volatile memory devices. Generally, over PCIe. Specifications available online [nvme].

OSPF

Open Shortest Path First. A protocol for exchanging Internet routing information.

PCH

Platform controller hub. A chip that exists on the system board that provides a number of common functions such as USB and Ethernet controllers.

PCI

Peripheral Component Interface. Refers to a standard, both electrically and programatically for devices. It has been evolved into the PCI Express (PCIe) specification. From a software perspective, most expansion cards today interface with the system via PCI/PCIe.

PCIe

PCI Express. The current revision of the PCI bus. Where the PCI bus was previously a parallel bus, the PCIe bus is a serial bus. The PCIe bus is backwards compatible with PCI based devices from a software programming perspective. Almost all modern expansion cards are based on PCIe.

PDU

Power Distribution Unit. A device that is used to provide power to several devices. It can be thought of as an intelligent power strip.

PMBus

Power Management bus. A specification that describes a command set for power related devices to both query and manage them. PMBus leverages SMBus as a transport layer. Specifications are available online [pmbus].

PMTT

Platform Memory Topology Table. An ACPI table that describes memory information. See section 5.2.21.12 Memory Topology Table (PMTT) of ACPI v 6.2 [*] for more information.

PROM

Programmable Read-Only Memory. A device that has readable non-volatile memory that can generally only be programmed once.

PSU

Power Supply Unit. A device which accepts either AC or DC power and transforms it into various power lines that are required for the system board, drives, expansion cards, and other components in the platform.

QSFP

Quad small form factor pluggable transceiver. An iteration of the SFP/SFP+ specifications that combines four transceivers in one, allowing for increased data rates. These are commonly used for 40 Gb/s Ethernet.

QSFP28

Quad small form factor pluggable 28 transceiver. An iteration of the SFP28 and QSFP specifications that combine four SFP28 transceivers, allowing for increased data rates. These are commonly used for 100 Gb/s Ethernet.

RAS

Reliability, Availability, and Serviceability. Reliability refers to the ability to detect errors. Availability is the ability to still operate in the face of failure. Serviceability refers to capabilities that reduce the effort required to service a component.

RCA

Root Cause Analysis. Usually refers to the act of determining the underlying cause behind the specific failure of a component in the system.

Redfish

An API standard developed by the DMTF that focuses around systems management. Specifications available online.

SAS

Serial Attached SCSI. SAS devices are a drive interface that has been used by modern hard drives and solid state drives. It is an evolution of the SCSI command set; however, the physical interfaces have changed substantially from the original SCSI implementations.

SATA

Serial ATA. A specification used by both hard drives and solid state drives. It derives its command set from the old ATA specifications. However, the modern physical layer and the way that commands are transferred has changed substantially.

SCSI

Originally a type of hard drive interface and command set. These days the command set has survived, but the physical layer has changed to SAS.

SDR

Sensor Data Record. Information about a sensor that is provided by IPMI based systems.

SFF

Small Form Factor. The SFF is a set of specifications that are provided by the Small Form Factor Committee. They create various standards that cover mechanical, electrical, and informational.

SFP

Small Form Factor Pluggable transceiver. A specification for a pluggable transceiver used in networking devices.

SFP+

Enhanced Small Form Factor Pluggable transceiver. A specification for an enhanced transceiver that can accommodate speeds used by 10 Gb/s Ethernet.

SFP28

Small From Factor Pluggable 28 transceiver. A revision of SFP+ devices that support both 10 Gb/s and 25 Gb/s Ethernet.

SPC

SCSI Primary Commands. The set of standards commands that all SCSI devices must implement.

SES

SCSI Enclosure Services. A series of specifications that describe a means to get information about a storage enclosure that is available as a SCSI target.

SMBIOS

The System Management BIOS specification. A DMTF standard that defines read-only data structures about the system. Specifications available online [smbios].

SMBus

System Management Bus. A standard that describes how peripherals are interconnected on a shared bus and how devices are discovered. It is derived from i2c. This interface is generally used as part of the low level interconnects between the system and devices. Specifications are available online [smbus].

SMM

System Management Mode. A mode of execution for x86 CPUs that is more privileged than that of the operating system. SMM is often used for various power management, error handling, and other platform-specific issues.

SP

Service Processor. A device that provides platform management capabilities even when the system is powered off or unavailable. Often a component of lights out management. See also, IPMI.

SPD

Serial Presence Detect. Refers to a standardized method of accessing information on DIMMs such as their EEPROMs. This information is available over SMBus.

System Board

The primary electrical board that contains soldered on chips, physical slots for CPUs, DIMMs, PCIe devices, and more, and wiring to interconnect all of the components. Often called the motherboard.

UEFI

Unified Extensible Firmware Interface. A series of specifications for systems firmware that are used to boot systems. A replacement for the BIOS. Also provides runtime services to the operation system, allowing for additional management capabilities.

UFM

Upgradable Firmware Module. A class of firmware that can be upgradable in the field.

UMC

Unified Memory Controller. Refers to the on-CPU memory controller used on AMD platforms. The Intel equivalent is the IMC.

UUID

Universally Unique Identifier. An ID that should be globally unique.

VPD

Vital Product Data. Refers to information available on a device such as the part and serial numbers. Usually this information is available in EEPROMs burned into the devices.

WWN

World Wide Name. A unique identifier that is assigned to a device. Commonly used in SAS devices both for identification and addressing.

x86

The Intel computing architecture used by both Intel and AMD.

xapic and x2apic

Extended advanced programmable interrupt controller and its second revision. These are interrupt controllers used on x86 systems that have a standardized programming interface and capabilities. This is documented in Chapter 10 Advanced Programmable Interrupt Controller (APIC) in Intel Volume 3 [sdm].