SILICON ENABLING IN A MODULAR ARCHITECTURE

Contributors

Isaac Oram
Intel Corporation

Tim Lewis
Phoenix Technologies

Vincent Zimmer
Intel Corporation

The Unified Extensible Interface UEFI Platform Initialization (PI) specifications allow for modular silicon enabling using defined building blocks. This includes system initialization, boot, and the unique requirements of the pre-boot space.

The adoption of UEFI specifications in the BIOS industry has reached critical mass in recent years. UEFI is now a component of firmware, operating systems, add-in devices, and other industry standards. Most products in the Intel® Architectures Personal Computer ecosystem are based on designs derived from the original EFI specifications and their current counterparts: the UEFI specification, and extensions such as the Intel® Platform Innovation Framework for EFI (Framework) and UEFI Platform Initialization (PI) specifications. These specifications have become the cornerstone for Intel silicon enabling in the Intel Architectures Personal Computer ecosystem.

This article will explore the use of the UEFI Platform Initialization (PI) specifications as a framework for silicon enabling. We will examine the building block elements provided for by the specification as well as the platform boot process, unique modes, and common uses. Additionally, we will examine drivers for different processors, memory and graphics controllers, and support chips.

Introduction

Personal computing has undergone a quiet revolution in the last five years, which was primarily driven by silicon enabling. As the industry evolved from personal computers (PCs) into many distinct product lines built on several architectures, an extensible and component-oriented industry standard firmware architecture was introduced and generally accepted into the BIOS environment. This revolution started for a number of reasons, but ultimately was driven by the need to initialize and utilize increasingly complex silicon and products in a cost-effective manner.

Definition: Silicon enabling can be defined as the activities required for OEM to deliver products to market utilizing a particular set of silicon products. This goal can be met through delivery of silicon specifications, reference code, sample code, binary modules, default settings, and the like.

"Definition: Silicon enabling can be defined as the activities required for OEM to deliver products to market utilizing a particular set of silicon products. This goal can be met through delivery of silicon specifications, reference code, sample code, binary modules, default settings, and the like."

Industry standards typically arise from a distinct need. Even with an identified need an industry standard relies on more than simply being documented in order to be successful. The "generally accepted extensible industry standard

architecture" in this case is the UEFI Forum's Platform Initialization (PI) architecture and is effectively realized as a standard through five vectors:

- It is documented in the UEFI Platform Initialization Specification. This provides a binary interoperable component architecture that is predominately applicable for silicon initialization prior to loading an operating system (OS). It is commonly thought of as a BIOS construction architecture specification.
- It is publicly instantiated in two open source software development environments, the EFI Development Kit (EDK) and EDK II; a common location for these kits is the website www.tianocore.org. These provide widely available and readily reusable implementations that module developers can utilize as a reference for building products that conform to the various UEFI and PI specifications.
- It is instantiated in BIOS products as a de facto standard implementation through a concept known as the Intel Green H, which is fundamentally a specific version of interface header files (corresponding to industry standards), library implementations, and some core modules. This provides a means to deliver source code into multiple BIOS codebases with minimal integration required.
- It is testable utilizing the UEFI published Self-Certification Tests (SCT). The SCT exercise the interfaces defined in the various UEFI specifications.
- It is used by leading hardware, software, and systems companies and thus is effectively required in the marketplace.

A BIOS software standard has the special constraints of having to support many different activities, most germane to this discussion are hardware initialization and system debug. These special constraints necessitate both source code and binary module compatibility and interoperability. These are required in order to effectively realize a component-oriented architecture used by a multitude of companies across the industry that support the Intel Architectures Personal Computer ecosystem. In the case of PI, binary compatibility is supported through the specifications and tests. Source compatibility is supported through the EDK and Intel Green H. Widespread requirement is realized through the support of BIOS, hardware, and software vendors throughout the industry.

"A BIOS software standard has the special constraints of having to support many different activities, most germane to this discussion are hardware initialization and system debug."

"In the case of PI, binary compatibility is supported through the specifications and tests."

In order to explore the modern silicon enabling environment, it is necessary to discuss the PI definition itself, the extensive silicon initialization performed within this infrastructure, the opportunities that are available, examples of the relevance, and the future opportunities not yet realized.

Platform Initialization Architecture

What is a BIOS? The term BIOS stands for basic input/output system. BIOS is a class of firmware that runs on the in-band processors or CPUs of a system in order to initialize the platform hardware complex and pass control to an

operating system. For purposes of the security architecture mentioned in the earlier chapter, it is the "firmware" layer.

"The original PC/XT had an 8-KB BIOS that initialized the system and passed control to DOS."*

The original PC/XT had an 8-KB BIOS that initialized the system and passed control to DOS*. This was 1982. Since that time, the BIOS domain has evolved significantly, including the transition of the industry to the Unified Extensible Firmware Interface (UEFI).

We will refer to the original BIOS as the *conventional BIOS* and the UEFI-based boot code as *UEFI*. Conventional BIOS and UEFI (UEFI Specification) based-systems must carry out several roles. First, each has the concept of platform initialization. This phase is the code that commences execution immediately after a platform restart (S3, S5, and so on). In a conventional BIOS, this is a vendor-specific flow and construction, but is sometimes referred to as the *stackless assembly or boot-block code*. In UEFI, there is a separate set of standards referred to as the Platform Initialization (PI) standards (UEFI PI Specification). In a PI-based platform initialization, the SEC and PEI phases commence this early execution.

"In UEFI, there is a separate set of standards referred to as the Platform Initialization (PI) standards (UEFI PI Specification)."

The temporal evolution of a UEFI PI-based boot is shown in Figure 1.

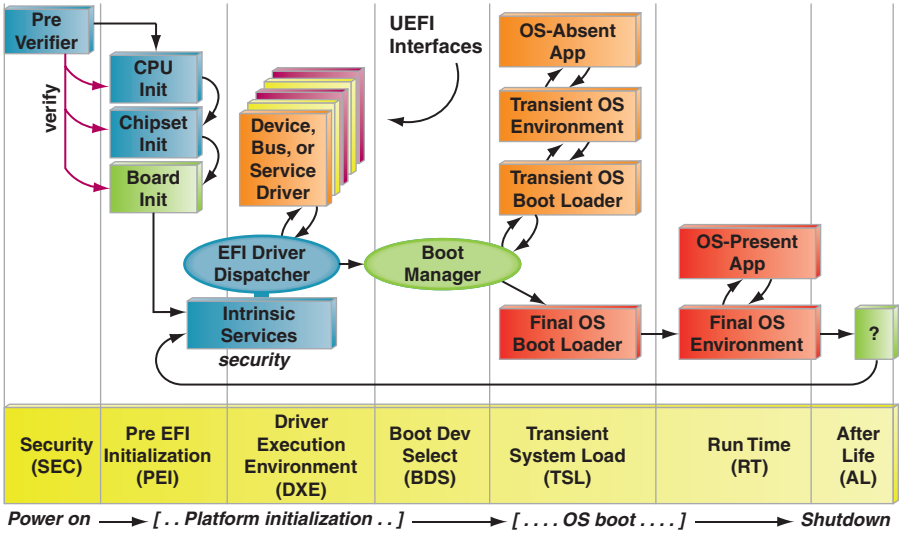

Figure 1: UEFI PI boot flow
(Source: Intel Corporation, 2011)

Afterward, each platform needs to discover I/O buses, dispatch option ROMs from host-bus adapter cards, and so on. In conventional BIOS, this I/O enumeration happens in the power-on-self test (POST) phase. There is no real standard for POST on a conventional BIOS. For UEFI PI-based firmware, though, this phase of execution occurs in the Driver Execution Environment (DXE).

DXE also serves as the UEFI core for purposes of supporting UEFI-based operating systems.

After BIOS POST and DXE, though, the "standard" part of the interface to the platform appears. For a PC/AT BIOS, the standard includes the de facto interrupt-callable interface (for example, Int13h for disk, Int10h for video) that executes in 16-bit real mode on the x86 architecture. For UEFI, this option ROM and loader interoperability includes the UEFI boot services and protocols (for example, EFI_BLOCK_IO_PROTOCOL as analog to BIOS int13h) and is described by the UEFI specification. It is during this phase of execution where third party content can appear from disk or adapters that did not necessarily ship with the platform manufacturer's (PM) system board.

This taxonomy above is critical because the transition from an execution regime provided by the PM into a space where third party codes can run has implications on the construction of a trusted platform.

"the transition from an execution regime provided by the PM into a space where third party codes can run has implications on the construction of a trusted platform."

Figure 2 shows the generic BIOS initialization flow. This flow includes the initialization of the platform CPU, memory, and I/O devices during POST. The POST flow again is analogous to the DXE flow in Figure 1.

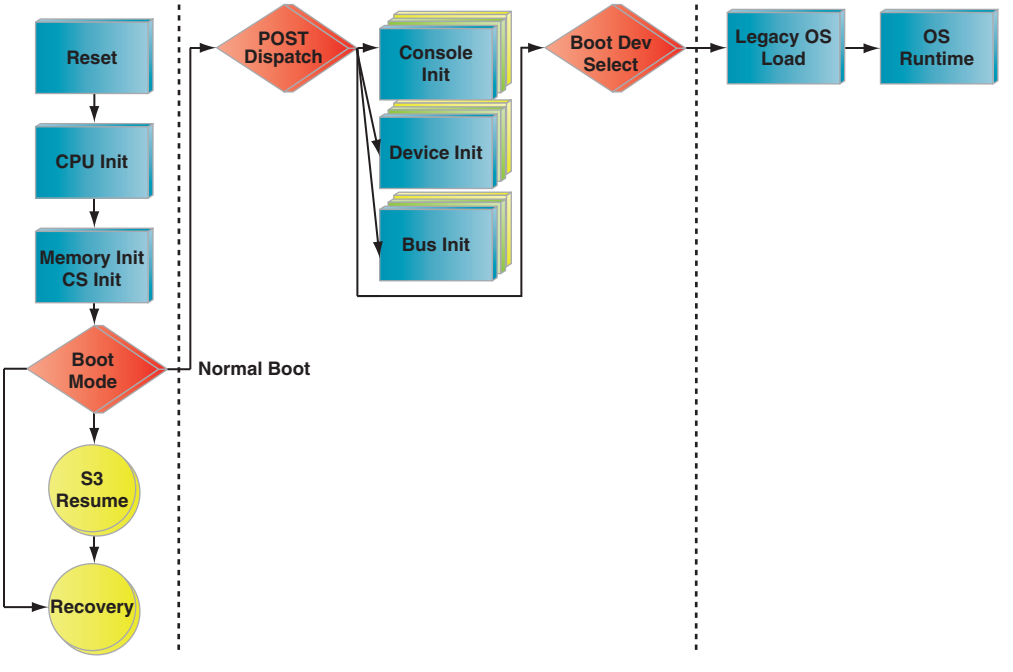

Figure 2: High-level BIOS flow
(Source: Intel Corporation, 2011)

UEFI Firmware

Above the hardware layer of the security architecture is the firmware, as shown in Figure 3.

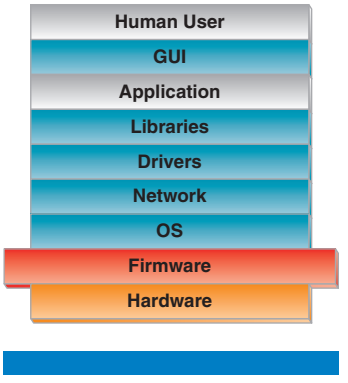

Figure 3: Firmware layer of the security architecture (Source: Intel Corporation, 2011)

Since there are several code-base implementations of platform firmware today and possibly in the future, this discussion will be held in the context of the Platform Initialization (PI) standards and some representative codebases thereof.

As a backgrounder, the Platform Initialization Working Group of the UEFI Forum delivers the Platform Initialization Architecture Specifications, based on Intel PEI and DXE specifications. The Platform Initialization Architecture Specifications are independent of the UEFI 2.0 Specification. The PI Architecture platforms can still boot today's operating systems. AMD, AMI, Apple, Dell, HP, IBM, Insyde, Intel, Lenovo, Microsoft, and Phoenix own the specifications. The goal of the Platform Initialization Working Group is to allow silicon vendors who create "reference code" today to package this reference code as modules that snap into PI Architecture firmware implementations. Figure 4 illustrates the PI scope.

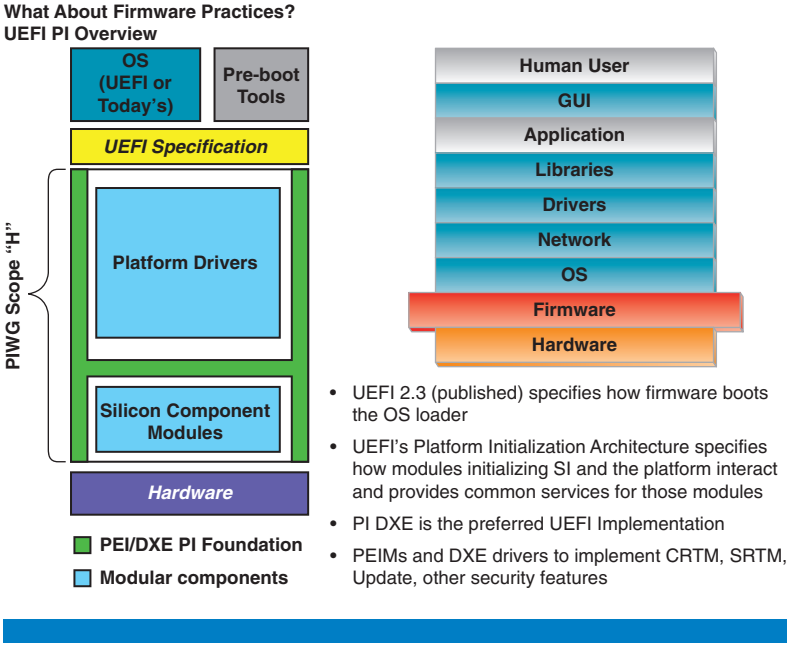

Figure 4 UEFI PI (Source: Intel Corporation, 2011)

UEFI and PI specifications only describe normative material around interface definition and mechanism, but no informative content, such as why and how. The latter is intended to be the purview of design guides.

More information on the UEFI PI Specifications can be found at the UEFI Web site.[1]

"In contrast to the monolithic nature of a BIOS, the UEFI PI allows hardware agility via software extensibility by exposing a driver model."

In contrast to the monolithic nature of a BIOS, the UEFI PI allows hardware agility via software extensibility by exposing a driver model, such as dependency-expression–based PEI Modules (PEIMs) and DXE drivers. Extensibility points can serve as a point of attack for malware. This type of

malware is essentially undetectable by OS-hosted protection technologies, such as antivirus software, since the malware could execute well before the OS. Consequently, the security integrity foundation of the pre-OS boot environment provided by UEFI and other pre-OS extensibility must be solid, while still providing enough flexibility to support hardware agility.

The PI phase is intended to only be extensible by the platform manufacturer (PM), not third party (in contrast to UEFI and its option ROM/loader/driver model). The installation and behavior of code in this early PI flow is said to act under the authority of the platform manufacturer; this will be referred to as "PM_AUTH" below.

Preservation of this PI as PM-extensible-only-intent in both construction and survivability in the field is a goal of the system design.

Figure 5 describes the UEFI PI boot flow, including annotation of the PM-extensible-only PI code and the third party extensible UEFI.

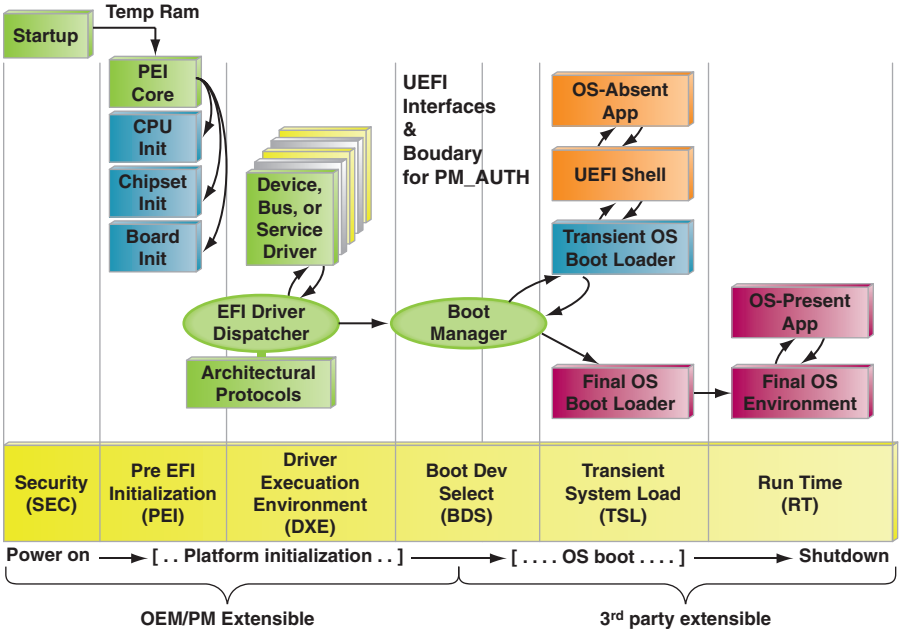

Figure 5: UEFI PI boot flow
(Source: Intel Corporation, 2011)

Silicon Initialization

The UEFI Platform Initialization or Intel® Platform Innovation Framework for EFI (Framework) specifications provide a software environment that allows a silicon vendor to deliver nearly all required core silicon initialization. For the purposes of this discussion, we are not distinguishing between the two sets of specifications, as they largely cover similar infrastructure for component development and interoperability.

"Software remains the most efficient method of initializing and supporting this increasingly complex hardware."

Silicon initialization remains required due to the costs and complexities associated with initialization in hardware and the relative inflexibility of hardware as opposed to software. In the last 15 years, we have seen processors grow from <10 to >800 million transistors.[2] Accompanying this growth was a range of increasingly complex features and capabilities. Software remains the most efficient method of initializing and supporting this increasingly complex hardware.

For the purposes of this discussion, we are focused on the critical components (core silicon) that are a part of every Intel architecture system: processors, memory controllers, graphics controllers, storage controllers, system bus controllers, IO controllers, and the like. There are a host of additional devices that comprise a modern Intel architecture system, but their needs are primarily met by the UEFI Specification. Additionally, these devices tend to be initialized later in the boot process with more reliance on other industry standards, such as USB, PCIE, and ACPI. In some cases, BIOS plays a minimal role in their initialization.

"Core silicon initialization relies on a phased approach."

Where Does Silicon Initialization Occur?

Core silicon initialization relies on a phased approach. At initial system reset, a very limited set of hardware resources is available; devices are inaccessible, memory is not available, and so forth. This leads BIOS to change the basic operating environment for software as hardware is initialized and resources expand. In PI:

- The system commences at reset (the first instruction fetched by the host bootstrap processor) in the SEC phase, where it makes memory available by initializing the processor cache and then transitions to PEI phase.
- The system starts PEI with a small amount of stack and heap available, and initializes enough hardware to have permanent memory available and then we transition to DXE phase.
- The system starts DXE with permanent memory, initializes core silicon, and then transitions to BDS phase.
- The system starts BDS with core silicon initialized and proceeds to initialize hardware required to boot an OS (input, output, and storage devices). At a high level, BDS corresponds to "executing the UEFI driver model" for the purposes of booting an OS.

There are special sub-phases in the flow:

- PEI pre-mem
- PEI post-mem
- SMM
- CSM

SEC	PEI (Pre-memory)	PEI (Post-Memory)	DXE	SMM	BDS	CSM
Native Processor Mode	Debug Infrastructure	Final Stack/ Heap	MP Support	Code Infrastructure	PCI	Legacy OS interfaces
Initial Stack/Heap	Memory	Full Config (S3)	SMM	Protection mechanisms	USB	
Microcode Update	Processor/ Chipset interfaces	Flash Update (Capsule)	Power/Terminal Management	Flash Write	SATA	
	Long Initialization devices	Cache Configuration	PCI		Graphics	
					Storage	

Table 1: Mapping of core SI to phases

As Table 1 shows, silicon initialization is spread throughout the boot process. The reasons are generally based on cost and complexity. As the boot progresses, the cost and complexity of initializing a set of silicon functionality goes down as more infrastructure becomes available. This is not to say that SMM initialization is not complex, but it is significantly less complex after permanent memory and DXE services are available. The cost stems from the need to execute directly from uncompressed FLASH memory prior to memory availability. The end result is that various core silicon initialization activities included in the table can be accomplished reasonably cheaply by PI modules and typically comprise greater than 90 percent of the silicon initialization required.

"As the boot progresses, the cost and complexity of initializing a set of silicon functionality goes down as more infrastructure becomes available."

How Is Silicon Initialization Implemented?

In the Intel case, silicon initialization takes the form of a set of packages that correspond roughly to silicon products. These silicon reference packages may support a single product, multiple products, a subset of the features associated with a single product, or even a capability that spans multiple products. Examples include the platform power management (PPM) reference package, which has often supported multiple generations of silicon, the platform controller hub (PCH) reference package, which typically supports a single generation of silicon, and the integrated clock controller (ICC) reference package, which is only applicable when the PCH is used in a specific configuration. The complexity in the object model arises from the complexity of the overall platform as well as the variability in the use of the reference package. An example of the latter is that PPM reference package is often enabled later in the product development cycle, so packaging it separately from the "always necessary" processor code gives customers flexibility in developing their product.

"The complexity in the object model arises from the complexity of the overall platform as well as the variability in the use of the reference package."

Each package effectively contains dynamic linked libraries (DLLs) for the different BIOS environments, such as pre-memory (PEIM), UEFI boot

services and runtime services (DXE), system management mode (SMM), and so on. These DLLs are usually delivered in source form, in a silicon reference package typically containing:

1. Source code
2. Custom interfaces as well as their documentation
3. Build files for use in an open source EDK
4. Sample code
5. Static libraries
6. Design and integration documentation

In order to reduce the number of supported configurations, the silicon reference packages are developed and tested against a specific version of industry standard interfaces and useful libraries, the Intel Green H. By defining the Intel Green H as a specific set of files, it is reasonable to deliver source code that has the attributes of being easily integrated and reusable without modification. By agreeing on such basics as the header files containing the services tables, common protocol structure definitions, and the like, a host of source code portability problems are avoided.

"By agreeing on such basics as the header files containing the services tables, common protocol structure definitions, and the like, a host of source code portability problems are avoided."

While PI provides a rich set of basic services, and in some cases abstracts more complex services, it is not sufficient to cover all possible silicon features with industry standard interfaces. In order to address this, reference packages provide custom interfaces. These interfaces typically take the form of variables, HOB, PPI, and protocols, as well as dependency expressions, callbacks, and other UEFI and PI services. With this toolbox available, silicon vendors can provide rich services to abstract silicon initialization. In the current silicon reference packages, it is common for there to be a "policy" protocol allowing the consumer to pass in board and design specific parameters. In other cases, interfaces, such as the I/O trap protocol, are provided. These services allow even the most complex silicon feature implementation to be shared between the producer and the consumer.

In total, with a silicon reference package that delivers the code and the Intel Green H giving it a "socket" to plug into, the consumer can build the modules using a widely available EDK and then focus their efforts on integrating into their BIOS build environment and enabling the various features of the silicon reference package in their BIOS. This allows the silicon vendor to deliver silicon initialization that is reused in widely divergent BIOS codebases that have unique requirements and attributes.

"This allows the silicon vendor to deliver silicon initialization that is reused in widely divergent BIOS codebases that have unique requirements and attributes."

Silicon Enabling: Changing the Role of BIOS

The advent of the UEFI PI standards represents a watershed moment for the industry.

Implement Once

PI has allowed core silicon initialization to be developed by the silicon provider and used everywhere that silicon is used. Previously, core silicon initialization was done by many different companies based on the limited documentation available for said silicon (SI). The downside of this set of behaviors is that the various implementations of said SI initialization by the SI consumers may deviate from the intent of the SI producers. Also, the validation effort applied internally by the SI producer towards the SI producer's implementation of the initialization code does not directly map to the results of the SI consumer; the latter will have its own SI initialization implementation and possibly different system board design.

Deployment

With many of the SI consumers aligning system board designs closely with the SI producers because of sensitive analog signaling and layout considerations, the point of variability more becomes the SI initialization code. As noted above, the SI producer creates SI initialization code to validate the SI internally. If the SI producer and SI consumer both support the UEFI PI specification for their system board firmware, the SI producer can release its SI initialization modules at the same time the physical SI components are released. This allows for a deployment model wherein there is no time delay between "hardware and firmware" with respect to deployment.

"allows for a deployment model wherein there is no time delay between "hardware and firmware" with respect to deployment."

Integration

The advent of systems supporting the UEFI and UEFI PI specifications is changing the relationship and responsibilities of the players in the PC firmware space.

In the "old" days (before UEFI and PI), the silicon vendor produced reference source code that showed the important initialization and configuration steps for initializing their specific chip or chips. The BIOS vendor took that source code, modified it for their build system, and hooked up each of the reference code fragments to their code base. Rolling out bug fixes to all customers was time-consuming because each one required an integration step that was unique to the target code base. Quality assurance for the BIOS focused on functional test or homegrown API testing, because there were few well-defined APIs. The only standard means of extending the BIOS was the option ROM, which was essentially unchanged for 20 years.

As time went by, many silicon vendors began to create a "core" source code package for their chips, as well as a "plug-in" layer for each of the BIOS code bases that they supported. This eased some of the problems for the silicon vendor, because now the single change they made to the "core" would work in all of the target code bases. But it created a problem for the BIOS vendors. Each silicon vendor was creating a unique "plug-in" style, with its own quirks, often favoring the BIOS code base that the silicon vendor used internally.

"Now each silicon vendor could package the support for their chips into drivers."

With UEFI and then, later, PI, there was a jump forward. Now each silicon vendor could package the support for their chips into drivers. The specifications defined how they were launched, how they published interfaces and how they discovered interfaces. Because of the standardization, there is no longer a separate "plug-in" layer for different BIOS vendor code bases. This has led to a decoupling of the code producer (silicon vendor) from the code consumer (BIOS vendor or OEM).

There have been a few consequences of this new model:

1. Write Once, Run Anywhere Drivers. If the BIOS complies with these specifications, a driver can be inserted in any BIOS and it should work.
2. Silicon-Vendor Produced Drivers. With UEFI, over time, the responsibility for all source code related to silicon support is shifting to the silicon vendor. Previously, despite the large amount of reference code given out in the "old" days, the BIOS vendor was still responsible for filling in the gaps.
3. Increased Testability. UEFI provides well-defined APIs for all services required for booting the system. The robustness of the implementation for these APIs is critical for insuring interoperability. Taking advantage of the well-defined APIs, the UEFI Testing Working Group has produced the Self-Certification Test (SCT) for each generation of the specifications.
4. Monolithic To Modular BIOS. More and more, the BIOS functions as a platform running a collection of drivers and applications, rather than a single body of code with a few strange appendages.
5. Drivers Offer Built-In Customization. In the "old" model, customization for a specific platform or product was handled by the OEM or BIOS vendor much later in the development cycle and often involved an insertion of hooks and flags directly into vendor-provided code. Now, the silicon vendors are inserting customizability into their drivers, using PI's Platform Configuration Database (PCD), policy protocols, and EFI variables.

"Once the inner workings of the BIOS world were exposed via a public specification, many new ideas came forward."

Once the inner workings of the BIOS world were exposed via a public specification, many new ideas came forward. The UEFI BIOS is gaining new capabilities because UEFI lowers the barrier to implementing new ideas that work on every PC.

Interoperability in Practice

The UEFI PI specification does not give you everything you need to build a PC platform. Many of the small silicon components—embedded controllers, super I/O controllers, flash devices—are represented in the specification but are not fully developed. However, the PI specification provides a solid foundation on which production-quality firmware can be built.

A good of example of this can be found in the PI specification's support for the System Management Bus (SMBus). Originally developed for supporting battery-management subsystems, this two-wire multi-master bus interface has

evolved into a standard motherboard side-band bus for sensor and platform management.[3] Since its introduction in 1995, it has become an integral part of other industry standards, including PCI, IPMI[4], DASH[5] and ASF[6].

The specification describes two APIs that abstract the SMBus host controller's low-level features: EFI_PEI_SMBUS2_PPI (for the PEI phase) and the EFI_SMBUS_HC_PROTOCOL (for the DXE phase). These APIs allow commands to be sent and received on the SMBus address without specific knowledge of the hardware interface.

```
typedef struct _EFI_SMBUS_HC_PROTOCOL {
   EFI_SMBUS_HC_EXECUTE_OPERATION          Execute;
   EFI_SMBUS_HC_PROTOCOL_ARP_DEVICE        ArpDevice;
   EFI_SMBUS_HC_PROTOCOL_GET_ARP_MAP       GetArpMap;
   EFI_SMBUS_HC_PROTOCOL_NOTIFY            Notify;
}  EFI_SMBUS_HC_PROTOCOL;
```

Code 1. SMBus Host Controller Protocol.
(Source: UEFI Forum, Inc.)

This protocol interface structure has function pointers to different abstracted functions. *Execute* will send a command to a targeted SMBus device over the SMBus. *ArpDevice* and *GetArpMap* will handle the SMBus address-resolution protocol to assign unique addresses to SMBus devices and report the results. *Notify* allows other drivers to register for a callback when SMBus devices send event notifications.

For example, the following code fragment sends the Get UDID (directed) command and then prints the error message or else the device's Unique Device Identifier (UDID).

```
EFI_STATUS s;
EFI_SMBUS_DEVICE_ADDRESS DeviceAddr;
SMBUS_GET_UDID_DIRECTED_RESP GetUdidResp;
s = SmbusHc–>Execute (
            SmbusHc,
            SMBUS_ADDR_DEV_DEFAULT_WRITE,
            DeviceAddr | 1,
            EfiSmbusReadBlock,
            TRUE,
            &DataSize,
            &GetUdidResp);
if (EFI_ERROR (s)) {
  printf (“Failed (%r). Skipped\n”, s);
} else {
  DumpUdid (&GetUdidResp.Udid);
}
```

Code 2. SMBus Host Controller Protocol Execute() Example
(Source: Phoenix Technologies Ltd.)

"The caller is not required to know anything about how the SMBus Host Controller is implemented in hardware or, indeed, whether or not hardware is present at all."

The caller is not required to know anything about how the SMBus Host Controller is implemented in hardware or, indeed, whether or not hardware is present at all.

Where's the SMBus Bus?

So the PI specification provides access to the host controller. But where's the bus? Where's the SMBus device driver? In the UEFI driver model, support for an industry standard bus is usually broken into three drivers:

1. *Host Controller Device Driver.* This driver abstracts a specific type of host controller and produces a Host Controller Protocol to allow the attributes and features to be discovered and manipulated.
2. *Bus Driver.* This driver implements the requirements of the industry standard bus. It uses the Host Controller Protocol to enumerate the bus, finding all child devices, creating handles for them and installing an instance of the I/O Protocol on each.
3. *Device Driver.* This driver implements the requirements of a specific device on the bus. It uses the I/O Protocol to discover device features, send commands, and produce new protocols used by other drivers and applications.

This model is seen in the UEFI Specification with the PCI bus. The chipset drivers produce the PCI Root Bridge I/O Protocol (*Host Controller*). The PCI bus driver produces the PCI I/O Protocol (*I/O*). The graphics device drivers (for example) use the PCI I/O Protocol and produces the Graphics Output Protocol.

"The USB bus driver uses the USB Host Controller Protocol and produces the USB I/O Protocol (I/O)*."*

The USB bus follows a similar model. The PCI-XHCI drivers produce the USB Host Controller2 Protocol (*Host Controller*). The USB bus driver uses the USB Host Controller Protocol and produces the USB I/O Protocol (*I/O*). The USB keyboard driver (for example) uses the USB I/O Protocol and produces the Simple Text Input Protocol.

However, when we look back at what the PI specification describes, we can see that it provides just the basics: the Host Controller protocol. But where is the bus driver? Where is the I/O protocol? The subsequent sections talk about the methods that layer on top of the host controller itself so a consumer can talk to the necessary device.

The SMBus Driver Model

Phoenix's SecureCore Tiano* builds on the strong foundation provided by the PI specification to create a full UEFI-style driver model. This allows SMBus device drivers to be started, stopped, and enumerated. It also allows DASH, ASF, and IPMI to be layered on top in a well-understood fashion.

For example, Figure 6 shows the ASF architecture layered on top of a driver producing the PI specification's SMBus Host Controller protocol.

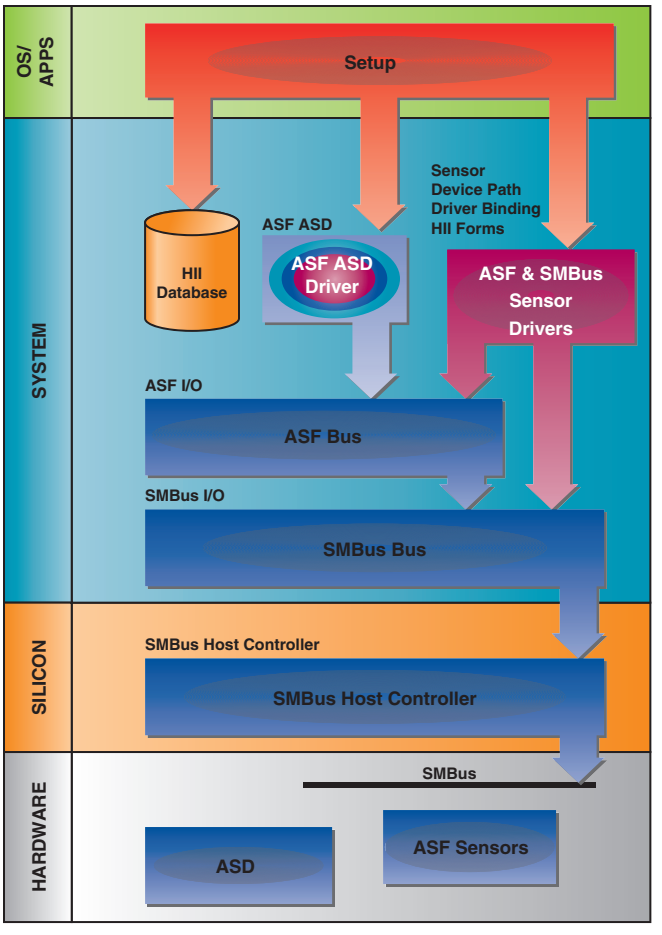

Figure 6: ASF/SMBus UEFI driver model
(Source: Phoenix Technologies Ltd., 2011)

The SMBus Host Controller driver still occupies the critical role, abstracting communication across the SMBus. But a new driver, the SMBus Bus driver, enumerates all the devices on the bus, creates child handles for each of the discovered devices and installs an instance of the SMBus I/O protocol on each. This follows the UEFI driver model shown earlier for PCI and USB.

```
typedef struct _SCT_SMBUS_IO_PROTOCOL {
  UINT32 Size;

  SMBUS_ADDR Addr;

  SCT_SMBUS_IO_IDENTIFY Identify;
  SCT_SMBUS_IO_EXECUTE Execute;
} SCT_SMBUS_IO_PROTOCOL, *PSCT_SMBUS_IO_PROTOCOL;
```

Code 3. Secure Core Tiano (SCT) I/O Protocol
(Source: Phoenix Technologies Ltd., 2011)

The *Execute* member function is a pass-through function to the SMBus Host Controller protocol for the host controller to which this device is attached. The new member function *Identify* is used to return the Unique Device Identifier (UDID), which provides the vendor identifier and device identifier (similar to PCI configuration space fields of the same name) and some capabilities flags.

This UDID is used by the *Supported()* and *Start()* member functions of the Driver Binding protocol produced by the ASF and SMBus device drivers.

"The device drivers can, in turn, produce additional protocols and even setup pages."

The device drivers can, in turn, produce additional protocols and even setup pages. For example, sensor drivers can provide real-time temperature, voltage, or fan-speed values to the user.

ASF-Beyond SMBus

The Alert Standard Format is one of the remote platform management standards created by the Desktop Management Task Force. On the local PC platform, it uses the SMBus to communication between different ASF-compliant devices, such as NICs (or Alert Signaling Devices/ASDs), serial EEPROMs, and sensors. Each of these is an SMBus device, and more.

In Figure 6, the ASF bus driver examines each instance of the SMBus I/O protocol created previously to see if it is also an ASF-compliant device. It does this by examining the capabilities flags in the SMBus device's UDID. From the *Supported()* function of the ASF Bus driver:

```
SMBUS_UDID Udid;
s = SmbusIo–>Identify (SmbusIo, &Udid);
if (EFI_ERROR (s)) {
  DPRINTF_ERROR ("Could Not Return Device UDID.\n");
  goto Done;
}

if ((Udid.Interface & SMBUS_UDID_INTERFACE_ASF) == 0) {
  s = EFI_UNSUPPORTED;
}

...install an instance of the ASF I/O protocol...
s = EFI_SUCCESS;
Done:
return s;
```

Code 4. ASF device identification
(Source: Phoenix Technologies, Ltd., 2011)

In this case, the protocol is installed on the same device handle, since it is referring to the same device. The ASF I/O protocol converts ASF-style commands into SMBus commands and sends them to the ASF device.

Going one step further, the Alert Signaling Devices (or ASD) drivers or ASF sensor drivers in Figure 6 examine each device handle to see whether it supports ASF I/O and its UDID matches a specific device and vendor identifier, or responds to a specific command.

Building on the UEFI PI Specification

The same idea described for ASF is used for other industry specifications, such as DASH and IPMI. Each adds a layer on top of the SMBus I/O, such as DASH's MCTP and PLDM or IPMI's system interface.

The UEFI PI specification describes the base level APIs necessary to create a fully functional SMBus driver stack. Using this, device drivers can be made to support individual SMBus devices and additional standards, such as ASF, DASH, and IPMI. Phoenix's SecureCore Tiano* has employed this similar model for other small silicon devices, including embedded controllers, Super I/O controllers, and flash devices.

"The UEFI PI specification describes the base level APIs necessary to create a fully functional SMBus driver stack."

Making the SMBus devices into UEFI driver-model devices also allows the developer to take advantage of the wealth of debugging and development tools available, including the UEFI Shell.[7]

Future Directions

Going forward, the PI architecture allows for coordinated release of silicon and silicon initialization. Given the UEFI PI-defined interfaces and binary image format, the full advantages of implementations of the architecture can be realized. Specifically, the module implementations are decoupled from the code base implementation instance; the only criteria is a level of UEFI PI specification compatibility. The modules can be built as self-describing binaries that allow for SI consumers to do source or binary debug. And ultimately, the modules can be released only as binaries depending upon the support agreement or other business criteria. The UEFI PI also allows for automated integration and testing, wherein a new module can only demand testing related to its functionality, not a full testing regression regime.

"The modules can be built as self-describing binaries that allow for SI consumers to do source or binary debug."

Finally, PI allows for more of an OS-like model. Just as there are well-defined buses and sockets for SI, PI provides the same pin-outs and sockets for "firmware." Removing the time and effort to produce independent SI initialization modules will move the bar to providing more vendor-specific value added features on the platform that are end-customer visible, such as accelerated boot time or a rich graphical user interface. The advantages of UEFI PI become even more powerful for both the small and highly-integrated system-on-a-chip (SOC). The intent of UEFI PI specifications is to cover enough interfaces to "build real systems" but not so much as to make every implementation look the same or remove the opportunity to innovate and differentiate.

UEFI PI will continue to offer a robust execution environment for system board manufacturers to innovate and provide assurance around the implementation of UEFI and UEFI PI features.

"the UEFI PI provides a menu of items from the SI producer, BIOS vendor, and others who can provide content."

And finally, the UEFI PI provides a menu of items from the SI producer, BIOS vendor, and others who can provide content. And it is from this menu that the system board vendors can choose elements that they need to build basic platform capabilities, freeing up their development resources to build differentiated added value while covering basic tasks from these menu items.

References

[1] ***http://www.uefi.org/specs/***

[2] ***http://www.intel.com/technology/timeline.pdf***

[3] System Management Bus (SMBus) Specification, Version 2.0 (August 3, 2000) Copyright © 1994, 1995, 1998, 2000 Duracell, Inc., Energizer Power Systems, Inc., Fujitsu, Ltd., Intel Corporation, Linear Technology Inc., Maxim Integrated Products, Mitsubishi Electric Semiconductor Company, PowerSmart, Inc., Toshiba Battery Co. Ltd., Unitrode Corporation, USAR Systems, Inc.

[4] Intelligent Platform Management Interface Specification, Version 2.0 (February 12, 2004, June 12 2009 Markup), Copyright © 2009 Intel Corporation, Hewlett-Packard Company, NEC Corporation, Dell Inc.

[5] Management Component Transport Protocol (MCTP) SMBus/ I2C Transport Binding Specification, Version 1.0.0 (July 28, 2009), Copyright © 2009 Distributed Management Task Force, Inc.

[6] Alert Standard Format Specification, Version 2.0 (April 23, 2003), Copyright © 2000-2002 Distributed Management Task Force, Inc.

[7] *Harnessing the UEFI Shell: Moving the Platform Beyond DOS*, M Rothman, T Lewis, V Zimmer, R Hale (Intel Press, 2010)

Authors' Biographies

Isaac Oram is a platform firmware architect in the PC Client Group at Intel Corporation with 14 years of experience in research and development for notebook, desktop, server, and tablet product firmware.

Tim Lewis is the Chief BIOS Architect for Phoenix Technologies Ltd., responsible for the architecture of Phoenix's SecureCore Tiano* firmware for x86 and ARM processors. With over 24 years of firmware experience, Tim has worked on everything from compiler design to low-power-ASICs to firmware and Windows* application design. He represents Phoenix on the UEFI board of directors and to the ACPI 5.0 promoters. Tim has a B.A. in History from San Jose State and an M. Div. from Western Seminary.

Vincent J. Zimmer is a Principal Engineer in the Software and Services Group at Intel Corporation and has over 18 years experience in embedded software

development and design, including BIOS, firmware, and RAID development. Vincent received an Intel Achievement Award and holds over 200 patents. He has a Bachelor of Science in Electrical Engineering degree from Cornell University, Ithaca, New York, and a Master of Science in Computer Science degree from the University of Washington, Seattle.
http://www.twitter.com/VincentZimmer and vincent.zimmer@gmail.com