Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

EFI Architecture

The Extensible Firmware Interface (EFI) describes a programmatic interface to the motherboard, chipset, CPU, and other components

May 10, 2007
URL:http://www.drdobbs.com/embedded-systems/efi-architecture/199500688

Vincent Zimmer *is a Principal Engineer in the Software and Solutions Group at Intel Corporation. Vincent was awarded an Intel Achievement Award for his development of the EFI Framework Architecture.* ***Michael Rothman*** *is a Senior Staff Engineer in the Software and Solutions Group at Intel.* ***Robert Hale*** *is a Principal Engineer in the Desktop Enterprise Group at Intel. This article was excerpted from their book Beyond BIOS: Implementing the Unified Extensible Firmware Interface with Intel's Framework.*

I believe in standards. Everyone should have one.

--George Morrow

The Extensible Firmware Interface (EFI) describes a programmatic interface to the platform. The platform includes the motherboard, chipset, central processing unit (CPU), and other components. EFI allows for pre-operating system (pre-OS) agents. Pre-OS agents are OS loaders, diagnostics, and other applications that the system needs for applications to execute and interoperate, including EFI drivers and applications. EFI represents a pure interface specification against which the drivers and applications interact, and this chapter highlights some of the architectural aspects of the interface. These architectural aspects include a set of objects and interfaces described by the EFI Specification.

The cornerstones for understanding EFI applications and drivers are several EFI concepts that are defined in the EFI 1.1 Specification. Assuming you are new to EFI, the following introduction explains a few of the key EFI concepts in a helpful framework to keep in mind as you study the specification:

- Objects managed by EFI-based firmware. Used to manage system state, including I/O devices, memory, and events.
- The EFI System Table. The primary data structure with data information tables and function calls to interface with the systems.
- Handle database and protocols. The means by which callable interfaces are registered.
- EFI images. The executable content format by which code is deployed.
- Events. The means by which software can be signaled in response to some other activity

- Device paths. A data structure that describes the hardware location of an entity, such as the bus, spindle, partition, and file name of an EFI image on a formatted disk.

Objects Managed by EFI-based Firmware

Several different types of objects can be managed through the services provided by EFI. Some EFI drivers may need to access environment variables, but most do not. Rarely do EFI drivers require the use of a monotonic counter, watchdog timer, or real-time clock. The EFI System Table is the most important data structure, because it provides access to all EFI-provided the services and to all the additional data structures that describe the configuration of the platform.

EFI System Table

The EFI System Table is the most important data structure in EFI. A pointer to the EFI System Table is passed into each driver and application as part of its entry-point handoff. From this one data structure, an EFI executable image can gain access to system configuration information and a rich collection of EFI services that includes the following:

- EFI Boot Services.
- EFI Runtime Services.
- Protocol services.

The EFI Boot Services and EFI Runtime Services are accessed through the EFI Boot Services Table and the EFI Runtime Services Table, respectively. Both of these tables are data fields in the EFI System Table. The number and type of services that each table makes available is fixed for each revision of the EFI specification. The EFI Boot Services and EFI Runtime Services are defined in the EFI 1.10 Specification. Chapter 4 of the EFI 1.10 Specification describes the common uses that EFI drivers make of these services.

Protocol services are groups of related functions and data fields that are named by a Globally Unique Identifier (GUID), a 16-bit, statistically-unique entity defined in Appendix A of the EFI 1.10 Specification. Typically, protocol services are used to provide software abstractions for devices such as consoles, disks, and networks, but they can be used to extend the number of generic services that are available in the platform. Protocols are the mechanism for extending the functionality of EFI firmware over time. The EFI 1.10 Specification defines over 30 different protocols, and various implementations of EFI firmware and EFI drivers may produce additional protocols to extend the functionality of a platform.

Handle Database

The handle database is composed of objects called handles and protocols. Handles are a collection of one or more protocols, and protocols are data structures that are named by a GUID. The data structure for a protocol may be empty, may contain data fields, may contain services, or may contain both services and data fields. During EFI initialization, the system firmware, EFI drivers, and EFI applications create handles and attach one or more protocols to the handles. Information in the handle database is global and can be accessed by any executable EFI image.

The handle database is the central repository for the objects that are maintained by EFI-based firmware. The handle database is a list of EFI handles, and each EFI handle is identified by a unique handle number that is maintained by the system firmware. A handle number provides a database "key" to an

entry in the handle database. Each entry in the handle database is a collection of one or more protocols. The types of protocols, named by a GUID, that are attached to an EFI handle determine the handle type. An EFI handle may represent components such as the following:

- Executable images such as EFI drivers and EFI applications
- Devices such as network controllers and hard drive partitions
- EFI services such as EFI Decompression and the EBC Virtual Machine

Figure 1 shows a portion of the handle database. In addition to the handles and protocols, a list of objects is associated with each protocol. This list is used to track which agents are consuming which protocols. This information is critical to the operation of EFI drivers, because this information is what allows EFI drivers to be safely loaded, started, stopped, and unloaded without any resource conflicts.

[Click image to view at full size]

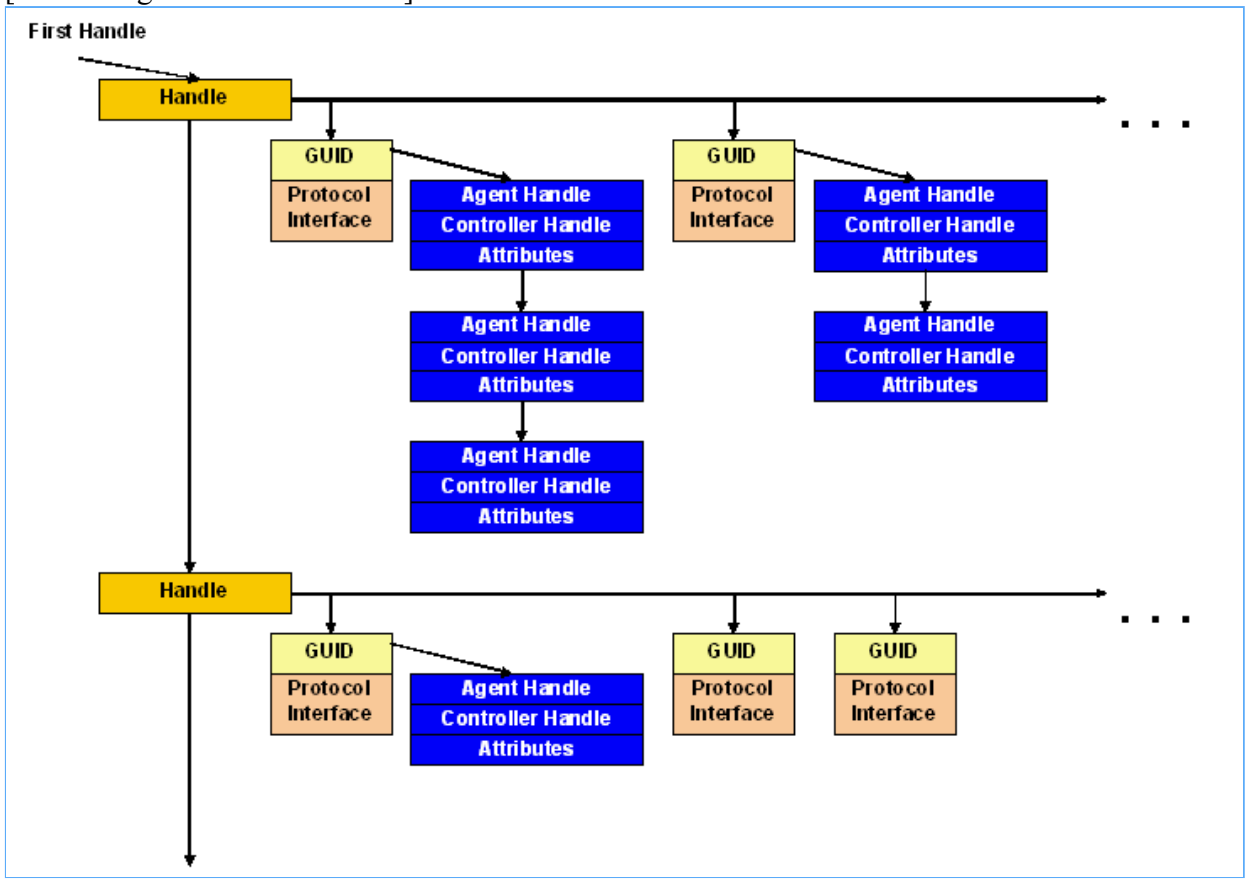

Figure 1: Handle Database.

Figure 2 shows the different types of handles that can be present in the handle database and the relationships between the various handle types. All handles reside in the same handle database and the types of protocols that are associated with each handle differentiate the handle type. Like file system handles in an operating system context, the handles are unique for the session, but the values can be arbitrary. Also, like the handle returned from an fopen

function in a C library, the value does not necessarily serve a useful purpose in a different process or during a subsequent restart in the same process. The handle is just a transitory value to manage state.

[Click image to view at full size]

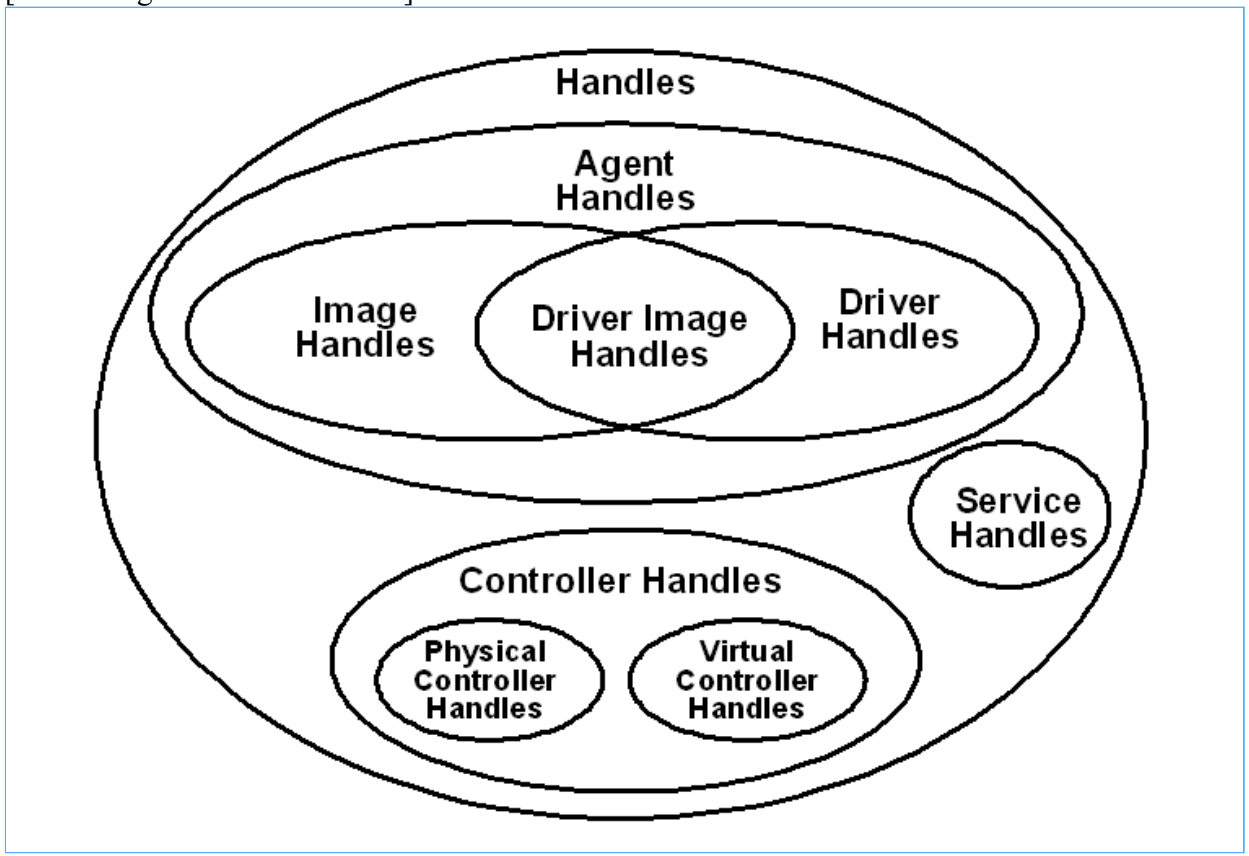

Figure 2: Handle Types.

Protocols

The extensible nature of EFI is built, to a large degree, around protocols. EFI drivers are sometimes confused with EFI protocols. Although they are closely related, they are distinctly different. An EFI driver is an executable EFI image that installs a variety of protocols of various handles to accomplish its job.

An EFI protocol is a block of function pointers and data structures or APIs that have been defined by a specification. At a minimum, the specification must define a GUID. This number is the protocol's real name; boot services like LocateProtocol uses this number to find his protocol in the handle database. The protocol often includes a set of procedures and/or data structures, called the protocol interface structure. The following code sequence is an example of a protocol definition from section 9.6 of the EFI 1.10 Specification. Notice how it defines two function definitions and one data field.

Sample GUID

```
#define EFI_COMPONENT_NAME_PROTOCOL_GUID \
{0x107a772c,0xd5e1,0x11d4,0x9a,0x46,0x0,0x90,
 0x27,0x3f,0xc1,0x4d}
```

Protocol Interface Structure

```
typedef struct _EFI_COMPONENT_NAME_PROTOCOL {
 EFI_COMPONENT_NAME_GET_DRIVER_NAME
  GetDriverName;
 EFI_COMPONENT_NAME_GET_CONTROLLER_NAME
  GetControllerName;
 CHAR8
  *SupportedLanguages;
} EFI_COMPONENT_NAME_PROTOCOL;
```

Figure 3 shows a single handle and protocol from the handle database that is produced by an EFI driver. The protocol is composed of a GUID and a protocol interface structure. Many times, the EFI driver that produces a protocol interface maintains additional private data fields. The protocol interface structure itself simply contains pointers to the protocol function. The protocol functions are actually contained within the EFI driver. An EFI driver might produce one protocol or many protocols depending on the driver's complexity.

[Click image to view at full size]

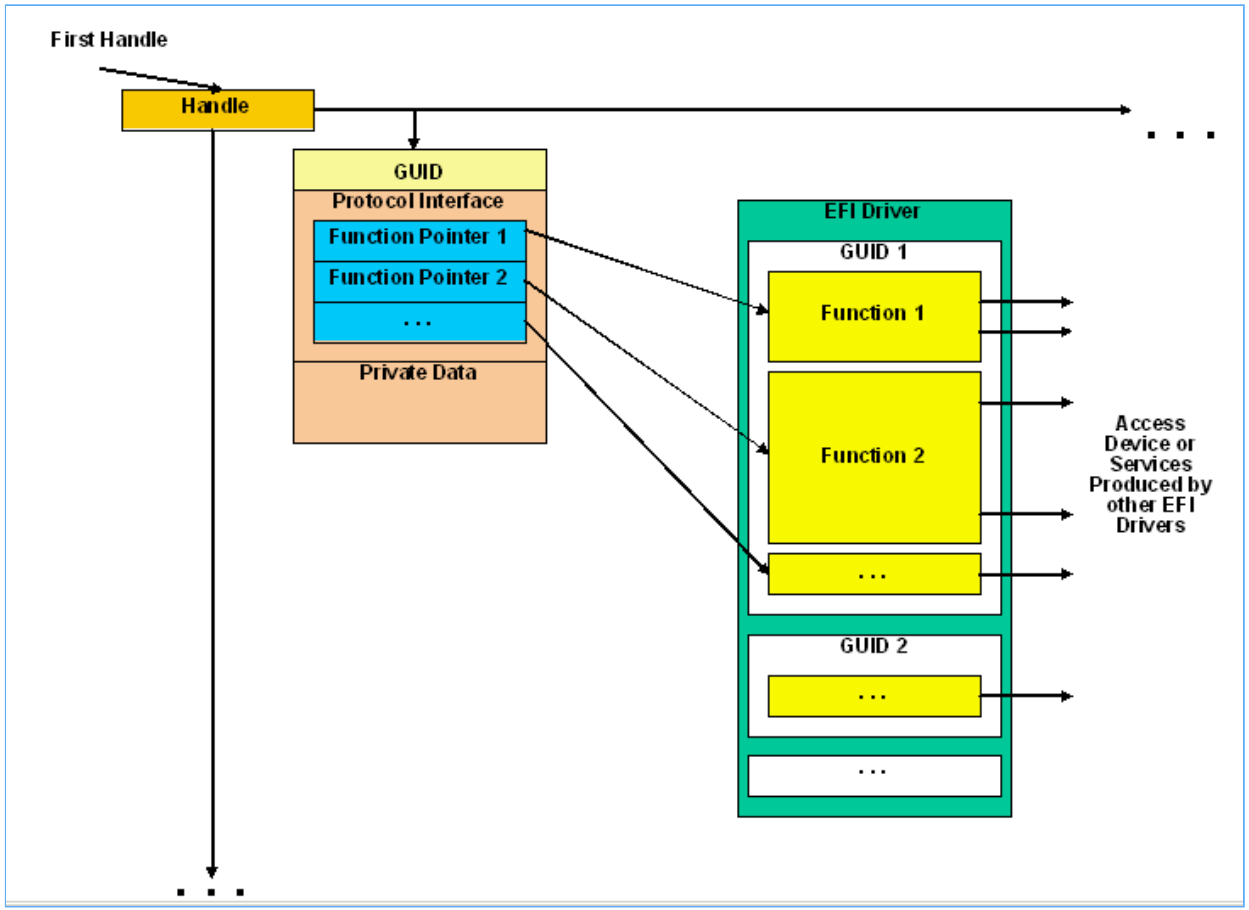

Figure 3: Construction of a Protocol.

Not all protocols are defined in the EFI 1.10 Specification. The EFI Developer Kit (EDK) includes many protocols that are not part of the EFI 1.10 Specification. These protocols provide the wider range of functionality that might be needed in any particular implementation, but they are not defined in the EFI 1.10 Specification because they do not present an external interface that is required to support booting an OS or writing an EFI driver. The creation of new protocols is how EFI-based systems can be extended over time as new devices, buses, and technologies are introduced. For example, some protocols that are in the EDK but not in the EFI 1.10 Specification are:

- Varstore. Interface to abstract storage of EFI persistent binary objects.
- ConIn. Service to provide a character console input.
- ConOut. Service to provide a character console output.
- StdErr. Service to provide a character console output for error messaging.
- PrimaryConIn. Console input with primary view.
- VgaMiniPort. Service that provides Video Graphics Array output.
- UsbAtapi. Service to abstract block access on USB bus.

The EFI Application Toolkit also contains a number of EFI protocols that may be found on some platforms, such as,:

- PPP Deamon. Point-to-Point Protocol driver.
- Ramdisk. File system instance on a Random Access Memory buffer.
- TCP/IP. Transmission Control Protocol/Internet Protocol.

The OS loader and drivers should not depend on these types of protocols because they are not guaranteed to be present in every EFI-compliant system. OS loaders and drivers should depend only on protocols that are defined in the EFI 1.10 Specification and protocols that are required by platform design guides such as Design Implementation Guide for 64-bit Server.

The extensible nature of EFI allows the developers of each platform to design and add special protocols. Using these protocols, they can expand the capabilities of EFI and provide access to proprietary devices and interfaces in congruity with the rest of the EFI architecture.

Because a protocol is "named" by a GUID, no other protocols should have that same identification number. Care must be taken when creating a new protocol to define a new GUID for it. EFI fundamentally assumes that a specific GUID exposes a specific protocol interface. Cutting and pasting an existing GUID or hand-modifying an existing GUID creates the opportunity for a duplicate GUID to be introduced. A system containing a duplicate GUID inadvertently could find the new protocol and think that it is another protocol, crashing the system as a result. For these types of bugs, finding the root cause is also very difficult. The GUID allows for naming APIs without having to worry about namespace collision. In systems such as PC/AT BIOS, services were added as an enumeration. For example, the venerable Int15h interface would pass the service type in AX. Since no central repository or specification managed the evolution of Int15h services, several vendors defined similar service numbers, thus making interoperability with operating systems and pre-OS applications difficult. Through the judicious use of GUIDs to name APIs and an association to develop the specification, EFI balances the need for API evolution with interoperability.

Working with Protocols

Any EFI code can operate with protocols during boot time. However, after **ExitBootServices()** is called, the handle database is no longer available. Several EFI boot time services work with EFI protocols.

Multiple Protocol Instances

A handle may have many protocols attached to it. However, it may have only one protocol of each type. In other words, a handle may not have more than one instance of the exact same protocol. Otherwise, it would make requests for a particular protocol on a handle nondeterministic.

However, drivers may create multiple instances of a particular protocol and attach each instance to a different handle. The PCI I/O Protocol fits this scenario, where the PCI bus driver installs a PCI I/O Protocol instance for each PCI device. Each instance of the PCI I/O Protocol is configured with data values that are unique to that PCI device, including the location and size of the EFI Option ROM (OpROM) image.

Also, each driver can install customized versions of the same protocol as long as they do not use the same handle. For example, each EFI driver installs the Component Name Protocol on its driver image handle, yet when the **EFI_COMPONENT_NAME_PROTOCOL.GetDriverName()** function is called, each handle returns the unique name of the driver that owns that image handle. The **EFI_COMPONENT_NAME_PROTOCOL.GetDriverName()** function on the USB bus driver handle returns "USB bus driver" for the English language, but on the PXE driver handle it returns "PXE base code driver."

Tag GUID

A protocol may be nothing more than a GUID. In such cases, the GUID is called a tag GUID. Such protocols can serve useful purposes such as marking a device handle as special in some way or allowing other EFI images to easily find the device handle by querying the system for the device handles with that protocol GUID attached. The EDK uses the HOT_PLUG_DEVICE_GUID in this way to mark device handles that represent devices from a hot-plug bus such as USB.

EFI Images

All EFI images contain a PE/COFF header that defines the format of the executable code as required by the Microsoft Portable Executable and Common Object File Format Specification (Microsoft 1997). The target for this code can be an IA-32 processor, an Itanium processor, or a processor agnostic, generic EFI Byte Code. The header defines the processor type and the image type. Presently there are three processor types and the following three image types defined:

- EFI applications. Images that have their memory and state reclaimed upon exit.
- EFI Boot Service drivers. Images that have their memory and state preserved throughout the pre-operating system flow. Their memory is reclaimed upon invocation of **ExitBootServices()** by the OS loader.
- EFI Runtime drivers. Images whose memory and state persist throughout the evolution of the machine. These images coexist with and can be invoked by an EFI-aware operating system.

The value of the EFI Image format is that various parties can create binary executables that interoperate. For example, the operating system loader for Microsoft Windows and Linux for an EFI-aware OS build is simple an EFI application. In addition, third parties can create EFI drivers to abstract their particular hardware, such as a networking interface host bus adapter (HBA) or other device. EFI images are loaded and relocated into memory with the Boot Service **gBS->LoadImage()**. Several supported storage locations for EFI images are available, including the following:

- Expansion ROMs on a PCI card.
- System ROM or system flash.
- A media device such as a hard disk, floppy, CD-ROM, or DVD.
- A LAN boot server.

In general, EFI images are not compiled and linked at a specific address. Instead, the EFI image contains relocation fix-ups so the EFI image can be placed anywhere in system memory. The Boot **Service gBS->LoadImage()** does the following:

- Allocates memory for the image being loaded.
- Automatically applies the relocation fix-ups to the image.
- Creates a new image handle in the handle database, which installs an instance of the EFI_LOADED_IMAGE_PROTOCOL.

This instance of the EFI_LOADED_IMAGE_PROTOCOL contains information about the EFI image that was loaded. Because this information is published in the handle database, it is available to all EFI components.

After an EFI image is loaded with **gBS->LoadImage()**, it can be started with a call to **gBS->StartImage()**. The header for an EFI image contains the address of the entry point that is called by **gBS->StartImage()**. The entry point always receives the following two parameters:

- The image handle of the EFI image being started.
- A pointer to the EFI System Table.

These two items allow the EFI image to do the following:

- Access all of the EFI services that are available in the platform.
- Retrieve information about where the EFI image was loaded from and where in memory the image was placed.

The operations that the EFI image performs in its entry point vary depending on the type of EFI image. Table 1 shows the various EFI image types and the relationships between the different levels of images.

[Click image to view at full size]

Type of Image	Description
Application	An EFI image of type EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION. This image is executed and automatically unloaded when the image exits or returns from its entry point.
OS loader	A special type of application that normally does not return or exit. Instead, it calls the EFI Boot Service **gBS->ExitBootServices()** to transfer control of the platform from the firmware to an operating system.
Drivern	An EFI image of type EFI_IMAGE_SUBSYSTEM_BOOT_SERVICE_DRIVER or EFI_IMAGE_SUBSYSTEM_RUNTIME_DRIVER. If this image returns EFI_SUCCESS, then the image is not unloaded. If the image returns an error code other than EFI_SUCCESS, then the image is automatically unloaded from system memory. The ability to stay resident in system memory is what differentiates a driver from an application. Because drivers can stay resident in memory, they can provide services to other drivers, applications, or an opera.
Service driver	A driver that produces one or more protocols on one or more new service handles and returns EFI_SUCESS from its entry point.
Initializing driver	A driver that does not create any handles and does not add any protocols to the handle database. Instead, this type of driver performs some initialization operations and returns an error code so the driver is unloaded from system memory.
Root bridge driver	A driver that creates one or physical controller handles that contain a Device Path Protocol and a protocol that is a software abstraction for the I/O services provided by a root bus produced by a core chipset. The most common root bridge driver is one that creates handles for the PCI root bridges in the platform that support the Device Path Protocol and the PCI Root Bridge I/O Protocol.
EFI 1.02 driver	A driver that follows the EFI 1.02 Specification. This type of driver does not use the EFI Driver Model. These types of drivers are not discussed in detail in this document. Instead, this document presents recommendations on converting EFI 1.02 drivers to drivers that follow the EFI Driver Model.
EFI Driver Model driver	A driver that follows the EFI Driver Model that is described in detail in the EFI 1.10 Specification. This type of driver is fundamentally different from service drivers, initializing drivers, root bridge drivers, and EFI 1.02 drivers because a driver that follows the EFI Driver Model is not allowed to touch hardware or produce device-related services in the driver entry point. Instead, the driver entry point of a driver that follows the EFI Driver Model is allowed only to register a set of services.
Device driver	A driver that follows the EFI Driver Model. This type of driver produces one or more driver handles or driver image handles by installing one or more instances of the Driver Binding Protocol into the handle database. This type of driver does not create any child handles when the **Start()** service of the Driver Binding Protocol is called. Instead, it only adds additional I/O protocols to existing controller handles.
Bus driver	A driver that follows the EFI Driver Model. This type of driver produces one or more driver handles or driver image handles by installing one or more instances of the Driver Binding Protocol in the handle database. This type of driver creates new child handles when the **Start()** service of the Driver Binding Protocol is called. It also adds I/O protocols to these newly created child handles.
Hybrid driver	A driver that follows the EFI Driver Model and shares characteristics with both device drivers and bus drivers. This distinction means that the **Start()** service of the Driver Binding Protocol will add I/O protocols to existing handles and it will create child handles.

Table 1: Description of Image Types.

Applications

An EFI application starts execution at its entry point, then continues execution until it reaches a return from its entry point or it calls the **Exit()** boot service function. When done, the image is unloaded from memory. Some examples of common EFI applications include the EFI shell, EFI shell commands, flash utilities, and diagnostic utilities. It is perfectly acceptable to invoke EFI applications from inside other EFI applications.

OS Loader

A special type of EFI application, called an OS boot loader, calls the **ExitBootServices()** function when the OS loader has set up enough of the OS infrastructure to be ready to assume ownership of the system resources. At **ExitBootServices()**, the EFI core frees all of its boot time services and drivers, leaving only the run-time services and drivers.

Drivers

EFI drivers differ from EFI applications in that the driver stays resident in memory unless an error is returned from the driver's entry point. The EFI core firmware, the boot manager, or other EFI applications may load drivers.

EFI 1.02 Drivers

Several types of EFI drivers exist, having evolved with subsequent levels of the specification. In EFI 1.02, drivers were constructed without a defined driver model. The EFI 1.10 Specification provides a driver model that replaces the way drivers were built in EFI 1.02 but that still maintains backward compatibility with EFI 1.02 drivers. EFI 1.02 immediately started the driver inside the entry point. Following this method meant that the driver searched immediately for supported devices, installed the necessary I/O protocols, and started the timers that were needed to poll the devices. However, this method did not give the system control over the driver loading and connection policies, so the EFI Driver Model was introduced in section 1.6 of the EFI 1.10 Specification to resolve these issues.

The Floating-Point Software Assist (FPSWA) driver is a common example of an EFI 1.02 driver; other EFI 1.02 drivers can be found in the EFI Application Toolkit 1.02.12.38. For compatibility, EFI 1.02 drivers can be converted to EFI 1.10 drivers that follow the EFI Driver Model.

Boot Service and Runtime Drivers

Boot-time drivers are loaded into area of memory that are marked as EfiBootServicesCode, and the drivers allocate their data structures from memory marked as EfiBootServicesData. These memory types are converted to available memory after **gBS->ExitBootServices()** is called.

Runtime drivers are loaded in memory marked as **EfiRuntimeServicesCode**, and they allocate their data structures from memory marked as **EfiRuntimeServicesData**. These types of memory are preserved after **gBS->ExitBootServices()** is called, thereby enabling the runtime driver to provide services to an operating system while the operating system is running. Runtime drivers must publish an alternative calling mechanism, because the EFI handle database does not persist into OS runtime. The most common examples of EFI runtime drivers are the Floating-Point Software Assist driver (FPSWA.efi) and the network Universal Network Driver Interface (UNDI) driver. Other than these examples, runtime drivers are not very common. In addition, the implementation and validation of runtime drivers is much more difficult than boot service drivers because EFI supports the translation of runtime services and runtime drivers from a physical addressing mode to a virtual addressing mode. With this translation, the operating system can make virtual calls to the runtime code. The OS typically runs in virtual mode, so it must transition into physical mode to make the call. Transitions into physical

mode for modern, multiprocessor operating systems are expensive because they entail flushing translation look-up blocks (TLB), rendezvousing coordinating all CPUs, and other tasks. As such, EFI runtime offers an efficient invocation mechanism because no transition is required.

Events and Task Priority Levels

Events are another type of object that is managed through EFI services. An event can be created and destroyed, and an event can be either in the waiting state or the signaled state. An EFI image can do any of the following:

- Create an event.
- Destroy an event.
- Check to see if an event is in the signaled state.
- Wait for an event to be in the signaled state.
- Request that an event be moved from the waiting state to the signaled state.

Because EFI does not support interrupts, it can present a challenge to driver writers who are accustomed to an interrupt-driven driver model. Instead, EFI supports polled drivers. The most common use of events by an EFI driver is the use of timer events that allow drivers to periodically poll a device. Table 2 shows the different types of events that are supported in EFI and the relationships between those events.

[Click image to view at full size]

Type of Events	Description
Wait event	An event whose notification function is executed whenever the event is checked or waited upon.
Signal event	An event whose notification function is scheduled for execution whenever the event goes from the waiting state to the signaled state.
Exit Boot Services event	A special type of signal event that is moved from the waiting state to the signaled state when the EFI Boot Service **ExitBootServices()** is called. This call is the point in time when ownership of the platform is transferred from the firmware to an operating system. The event's notification function is scheduled for execution when **ExitBootServices()** is called.
Set Virtual Address Map event	A special type of signal event that is moved from the waiting state to the signaled state when the EFI Runtime Service **SetVirtualAddressMap()** is called. This call is the point in time when the operating system is making a request for the runtime components of EFI to be converted from a physical addressing mode to a virtual addressing mode. The operating system provides the map of virtual addresses to use. The event's notification function is scheduled for execution when **SetVirtualAddressMap()** is called.
Timer event	A type of signal event that is moved from the waiting state to the signaled state when at least a specified amount of time has elapsed. Both periodic and one-shot timers are supported. The event's notification function is scheduled for execution when a specific amount of time has elapsed.
Periodic timer event	A type of timer event that is moved from the waiting state to the signaled state at a specified frequency. The event's notification function is scheduled for execution when a specific amount of time has elapsed.
One-shot timer event	A type of timer event that is moved from the waiting state to the signaled state after the specified timer period has elapsed. The event's notification function is scheduled for execution when a specific amount of time has elapsed.

Table 2: Description of Event Types.

The following three elements are associated with every event:

- The Task Priority Level (TPL) of the event.
- A notification function.
- A notification context.

The notification function for a wait event is executed when the state of the event is checked or when the event is being waited upon. The notification function of a signal event is executed whenever the event transitions from the waiting state to the signaled state. The notification context is passed into the notification function each time the notification function is executed. The TPL is the priority at which the notification function is executed. Table 3 lists the

four TPL levels that are defined today. Additional TPL's could be added later. An example of a compatible addition to the TPL list could include a series of "Interrupt TPL's" between TPL_NOTIFY and TPL_HIGH_LEVEL in order to provide interrupt-driven I/O support within EFI.

[Click image to view at full size]

Task Priority Level	Description
TPL_APPLICATION	The priority level at which EFI images are executed.
TPL_CALLBACK	The priority level for most notification functions.
TPL_NOTIFY	The priority level at which most I/O operations are performed.
TPL_HIGH_LEVEL	The priority level for the one timer interrupt supported in EFI.

Table 3: Task Priority Levels Defined in EFI.

TPLs serve the following two purposes:

- To define the priority in which notification functions are executed.
- To create locks.

For priority definition, you use this mechanism only when more than one event is in the signaled state at the same time. In these cases, the application executes the notification function that has been registered with the higher priority first. Also, notification functions at higher priorities can interrupt the execution of notification functions executing at a lower priority.

For creating locks, code running in normal context and code in an interrupt context can access the same data structure because EFI does support a single-timer interrupt. This access can cause problems and unexpected results if the updates to a shared data structure are not atomic. An EFI application or EFI driver that wants to guarantee exclusive access to a shared data structure can temporarily raise the task priority level to prevent simultaneous access from both normal context and interrupt context. The application can create a lock by temporarily raising the task priority level to TPL_HIGH_LEVEL. This level blocks even the one-timer interrupt, but you must take care to minimize the amount of time that the system is at TPL_HIGH_LEVEL. Since all timer-based events are blocked during this time, any driver that requires periodic access to a device is prevented from accessing its device. A TPL is similar to the IRQL in Microsoft Windows and the SPL in various Unix implementations. A TPL describes a prioritization scheme for access control to resources.