



UEFI Driver Development Guide for All Hardware Device Classes

Nov 2011

Version 1.0

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright 2011 Intel Corporation. All rights reserved.

Table of Contents

UEFI Driver Development Guide for All Hardware Device Classes	1
Requirements	1
Proper management of DMA addresses.....	2
Requirements, recommendations, and optional elements	2
Reminders, tips, do's and don'ts.....	7
For more information.....	8

List of Tables

Table 1.	Required UEFI protocols for UEFI device drivers	2
Table 2.	Additional implementation requirements for UEFI device drivers.....	3
Table 3.	Recommended or required UEFI protocols for all UEFI device drivers.....	3
Table 4.	Additional implementation recommendations for all UEFI device drivers.....	5
Table 5.	Optional UEFI protocols for UEFI device drivers that are important	5
Table 6.	Platform requirements for supporting UEFI device drivers.....	6

UEFI Driver Development Guide for All Hardware Device Classes

This document lists required, recommended, and optional UEFI protocols and elements for all classes of hardware device drivers. It also provides brief notes on design strategies and implementation for each protocol.

This document is a "short list" -- a reference list. More information about required and recommended UEFI protocols, the driver binding model, and UEFI boot services is available in the Intel® UEFI Driver Writer's Guide. (The driver writer's guide is expected to be available later this year on www.tianocore.org) UEFI driver code samples and templates are available in the Intel® UEFI Developer's Kit 2010 (Intel® UDK 2010, or Intel® UDK2010). Complete information on all required protocols and other elements for UEFI drivers is provided in the current UEFI specification.

Requirements

A UEFI driver is required for any PC hardware device needed for the boot process to complete. Hardware devices can be categorized into the following:

- Credential providers: Fingerprint sensors, retinal scanners, smartcard readers, and so on
- Network boot devices: UNDI, SMP, MMP
- Graphic output devices: Simple text, graphics output
- USB devices and USB host controllers: Add-in card manufacturer PCI controller with USB ports or OEM plug-in USB devices that are bootable
- Boot devices (non-SCSI): Block I/O, network devices, simple file systems, load file, load file 2
- SCSI devices: SCSI pass thru, SCSI bus, and individual SCSI devices
- Console devices: Simple input provider, simple input ex, simple pointer -- mice, serial I/O protocol (remote consoles)

Note that independent hardware vendors (IHVs) can choose not to implement all of the required elements of the UEFI specification -- for example all elements might not be implemented on a specialized system configuration that does not support all the services and functionality implied by the required elements. Also, some elements are required depending on a specific platform's features. Some elements are required depending on the features that a specific driver requires. Other elements are recommended based on coding experience, for reasons of portability, and/or for other considerations. It is recommended that you implement all required and recommended elements in your drivers.

Proper management of DMA addresses

For some drivers, the CPU address and the PCI address do not have to be the same. The addresses tend to be the same on Intel® Architecture-based platforms. However, features such as Intel® Virtualization Technology for Directed I/O and other CPU architectures do not require a 1:1 mapping between these two address spaces. When managing pointers in the common buffer, it is critical for the driver to understand whether the address is a CPU address or a PCI address, and manage the DMA address accordingly.

Drivers that do not have to have the same CPU address and PCI address include:

- USB host controller drivers
- Network boot device drivers
- Graphics controller device drivers

Requirements, recommendations, and optional elements

The following table lists the two protocols required by the UEFI specification for all UEFI device drivers.

Table 1. Required UEFI protocols for UEFI device drivers

Required protocol	Description
<i>EFI_DRIVER_BINDING_PROTOCOL</i>	<p>Provides functions for starting and stopping the driver, as well as a function for determining whether the driver can manage a particular controller.</p> <p>Device drivers are required to implement the Driver Binding Protocol.</p> <p>Device drivers must ignore the <i>RemainingDevicePath</i> parameter that is passed into the <i>Supported()</i> and <i>Start()</i> services of the Driver Binding Protocol.</p> <p>All drivers that follow the UEFI driver model must support the <i>Stop()</i> service.</p>

The following table lists additional implementation requirements for UEFI device drivers.

Table 2. Additional implementation requirements for UEFI device drivers

Additional implementation requirements	Description
The driver must use the UEFI system table.	Provides access to UEFI boot services, UEFI runtime services, consoles, firmware vendor information, and the system configuration tables.
The driver must use the UEFI boot services.	All functions defined as boot services.
The driver must use the UEFI runtime services.	All functions defined as runtime services.
The driver must manage one controller handle, but should be able to more controller handles.	Even if a driver writer is convinced that the driver will manage only a single controller, the driver should be designed to manage multiple controllers. The overhead for this functionality is low, and it will make the driver more portable.
The driver must consume one or more I/O-related protocols from the controller handle.	The type of I/O-related protocols consumed depends on the type of device being managed.
The driver must produce one or more I/O-related protocols on the same controller handle.	The type of I/O-related protocols produced depends on the type of device being managed.
The driver must not produce any child handles.	This feature is the main distinction between device drivers and bus/hybrid drivers.

The following table lists protocols that are optional according to the specification, but which are strongly recommended based on coding experience and best practices. These protocols should be supported by all device drivers.

Table 3. Recommended or required UEFI protocols for all UEFI device drivers

Recommended / required protocol	Description and notes
<i>EFI_LOADED_IMAGE_PROTOCOL</i>	<p>This protocol is produced by the UEFI core as part of the LoadImage()/StartImage() calls when the UEFI Driver is loaded. A UEFI device driver must consume this protocol if there are multiple images.</p> <p>This protocol contains information about the UEFI image that was loaded. You may use this protocol on any image handle to obtain information about the loaded image.</p> <p>If the driver loads an image, for good coding practices, the driver should also unload the image when done.</p> <p>Note that it is recommended that the <i>EFI_LOADED_IMAGE_PROTOCOL.Unload()</i> service be implemented during driver development, driver debug, and system integration. It is strongly recommended that this service remain in drivers for add-in adapters to help debug interaction issues during system integration. The unload service allows the driver to be dynamically unloaded.</p>

**Table 3. Recommended or required UEFI protocols for all UEFI device drivers —
continued**

Recommended / required protocol	Description and notes
<p><i>EFI_COMPONENT_NAME2_PROTOCOL</i> and <i>EFI_COMPONENT_NAME_PROTOCOL</i></p>	<p>The Component Name2 Protocol replaces the older Component Name Protocol.</p> <p>The Component Name Protocols provide functions for retrieving a human-readable name of a driver and the controllers that a driver is managing.</p> <p>The platform determines whether it will support the older Component Name Protocol or the current Component Name2 Protocol, or both. Because of this, it is strongly recommended that you implement both protocols in your driver.</p>
<p><i>EFI_DRIVER_DIAGNOSTICS2_PROTOCOL</i> and <i>EFI_DRIVER_DIAGNOSTICS_PROTOCOL</i></p>	<p>Provides diagnostics services for the controllers that UEFI drivers are managing. Note that time-consuming diagnostics should be deferred until the Driver Diagnostics Protocols are invoked.</p> <p>If the driver will allow the UEFI shell command <i>drvdiag</i> to perform a cursory check of the connections managed by the driver, then the driver must implement the Driver Diagnostics2 Protocol. These protocols are the only mechanism available to a driver when the driver wants to alert the user to a problem that was detected with a controller.</p> <p>The platform determines whether it will support the older Driver Diagnostics Protocol or the current Driver Diagnostics2 Protocol, or both, or neither. Because of this, it is strongly recommended that you implement both protocols in your driver.</p>
<p>Human Interface Infrastructure (HII) protocols</p>	<p>HII protocols are required by drivers that support user entry for configuration information. Drivers should not use other methods to display information to the user or request information from the user.</p> <p>If you implement HII protocols, you must also implement the Driver Health Protocol.</p>
<p><i>EFI_DRIVER_HEALTH_PROTOCOL</i></p>	<p>This protocol is required if HII is implemented.</p> <p>This protocol produces a collection of services that allow the health status for a controller to be retrieved. Health status could be: healthy, repair required, reboot required, or failed. The device state could require extended time to repair.</p> <p>The Driver Health Protocol is required if the driver needs to produce warning or error messages for the user, or needs to perform a repair operation that is not part of the normal initializing sequence, and the repair operation requires an extended period of time.</p> <p>This protocol is also required if the driver requires the user to make software and/or hardware configuration changes before the boot devices that the driver manages can be used.</p>

The following table lists additional implementation elements that are strongly recommended for all UEFI device drivers.

Table 4. Additional implementation recommendations for all UEFI device drivers

Additional implementation recommendations	Description and notes
Implement an Exit Boot Services event in the driver's entry point	<p>This event is recommended for all UEFI device drivers because it helps improve the hand-off of control to the operating system.</p> <p>If a driver enabled bus mastering for DMA (direct memory access), make sure to disable bus mastering before the operating system calls Exit Boot Services.</p> <p>The Exit Boot Services event is required only if the driver is required to place the devices it manages in a specific state just before control is handed to an operating system.</p>
Use private data structures	<p>All UEFI drivers that follow the UEFI driver model should allocate data structures via the UEFI memory services for each controller. Those data structures should contain all the information that the driver requires to manage each individual controller. This simplifies the process of updating the driver to manage more than one device.</p>

The following table lists protocols that are optional but applicable to all devices classes based on individual device capabilities. These protocols are important from coding experience or are often requested by customers.

Table 5. Optional UEFI protocols for UEFI device drivers that are important

Optional but important protocol	Description and/or notes
<i>EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL</i>	Specifies the device path that was used when a PE/COFF image was loaded through the UEFI boot service LoadImage().
<i>EFI_DEVICE_PATH_PROTOCOL</i>	Provides the location of the device.
<i>EFI_DECOMPRESS_PROTOCOL</i>	Provides interfaces to decompress an image that was compressed using the UEFI compression algorithm.
<i>EFI_DEVICE_PATH_UTILITIES_PROTOCOL</i>	Provides interfaces to create and manipulate UEFI device paths and UEFI device path nodes.
<i>EFI_FIRMWARE_UPDATE_PROTOCOL</i>	<p>Provides an abstraction for a device to provide firmware management support.</p> <p>This protocol makes it easier for Information Technology (IT) departments to manage devices, including performing firmware updates.</p>

The following table does not refer to driver requirements specifically. In other words, these are not protocols that the driver produces or consumes; they are not capabilities of the driver. Instead, these are protocols and some additional requirements that are required *in the platform* when the platform supports a specific feature for UEFI devices.

Table 6. Platform requirements for supporting UEFI device drivers

If the platform includes or supports...	Required protocols and other required elements
Debugging	<p><i>EFI_DEBUG_SUPPORT_PROTOCOL</i></p> <p><i>EFI_DEBUGPORT_PROTOCOL</i></p> <p>UEFI image information table</p>
Overriding the default driver to the controller matching algorithm provided by the UEFI driver binding model	<p><i>EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL</i></p>
Higher priority than the Bus Specific Driver Override Protocol	<p>The <i>EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL</i> must be produced on the same handle as the <i>EFI_DRIVER_BINDING_PROTOCOL</i>.</p>
Authentication of UEFI images, and the platform potentially supports more than one OS loader	<p>If the platform requires secure booting, you should implement the methods described in the UEFI specification for authenticating UEFI variables.</p>
Digital signatures	<p>If the platform requires secure booting, the driver must digitally sign the image(s). The driver must embed the digital signature in the PE/COFF image as described in the UEFI specification, in the section titled "Embedded Signatures."</p>
A driver written in EBC	<p>EBC interpreter. The interpreter supports option ROMs on add-in devices when the platform already has a driver written in EBC or when the platform includes a bus that supports add-in devices that might have an EBC driver on it.</p> <p>Note: If an EBC interpreter is implemented, then it must produce the EBC Protocol interface.</p>
A bus that supports option ROMs for one or more UEFI drivers	<p><i>EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL</i></p> <p>This protocol is produced by the PCI Bus Driver, and consumed by the DXE Core as part of the driver-to-controller connection process.</p> <p>In general, this protocol applies only to bus types that support option ROMs for drivers on their child devices. This protocol lets the driver in the PCI device know which option ROM has the higher priority. This protocol provides a mechanism for bus drivers to override the default driver selection performed by the ConnectController() boot service.</p> <p>At this time, the only bus type that is required to produce this protocol is a PCI bus, and the "container" for drivers is the PCI option ROM.</p> <p>The PCI bus driver is required to produce the Bus Specific Driver Override Protocol for PCI devices that have an attached PCI option ROM if the PCI option ROM contains one or more loadable UEFI drivers. If a PCI option ROM is not present, or if the PCI option ROM does not contain any loadable UEFI drivers, then a Bus Specific Driver Override Protocol will not be produced for that PCI device.</p>

Reminders, tips, do's and don'ts

Some common problems and coding issues can be avoided by remembering a few key points and requirements for writing UEFI 2.3.1 or later device drivers:

- Do not call non-UEFI 2.3.1 or later protocols or legacy BIOS interrupt functions from UEFI 2.x drivers.
- Make code as portable as possible. Do not rely on implementation-specific protocols.
- Make sure protocols are installed before calling them. Use return codes (not just output parameters) to verify that each necessary protocol is installed.
- As per the UEFI specification, if a service returns an error, the output parameters are undefined. When handling errors, check the return code instead of just checking the output parameters.
- UEFI drivers must not attempt to configure other platform hardware.
- Be careful when using periodic timers. Using timers incorrectly can significantly slow the boot process, as well as slow the performance of the system browser.
- The BrowserCallback function should be called only by a callback handler. Do not allow other functions to call that function.
- Make sure the console is installed before the driver tries to use it. Always consider the possibility of a headless system and NULL pointers in the system table.
- Update the HII forms pack only when something changes. The firmware does not check to see if anything has changed between boots or updates.
- Avoid direct user interaction. Publish protocols only for interaction with the firmware, and use the new Driver Health Protocol for any required user input, such as for device configuration and most repair operations.
- In the setup browser, do not directly invoke pop-up windows using EDK I or EDK II routines. All interaction with the user for the setup browser should be conducted via HII functionality.

When debugging:

- Do not assume legacy ports are available for output. Instead, use standard output protocols (gST->StdErr).
- Check platform attributes before enabling EfiPciIoAttributeOperationSupported.
- Do not enable unsupported PCI attributes. Use only the PCI I/O Protocol to adjust attributes.
- Do not use older EDK macros to enable devices. EDK includes some macros that were intended only for the chipset to initialize PCI devices. Do not use these macros to enable devices in a UEFI driver. These macros may not properly set your device attributes.

For more information

For more information about driver requirements, refer to:

UEFI specification. Information about UEFI device types and status codes can be found in the *Unified Extensible Firmware Interface*, version 2.3.1 or later, The UEFI Forum, 2010, www.uefi.org. A summary of UEFI services and GUIDs can be found in the Doxygen-generated help documents for the MdePkg in the UDK 2010 releases.

www.tianocore.org Information on coding standards for UEFI implementations as well as other UEFI documentation is available on www.tianocore.org

UEFI Driver Writers Guide. Refer to the driver writer's guide for key descriptions of how to implement the requirements, as well as recommendations for writing drivers. This guide is expected to be available soon on www.tianocore.org

UEFI Developer's Kit 2010 (UDK2010). This open-source kit contains EDK II (second generation EFI development kit) validated common-core sample code. The open-source UDK2010 is a stable build of the EDKII project, and has been validated on a variety of Intel platforms, operating systems, and application software. The open-source UDK2010 is available for download at www.tianocore.org

Intel® UEFI Developer's Kit 2010 (Intel® UDK 2010, also called the Intel® UDK2010), Intel Corporation, 2010. The licensed Intel UDK2010 is a full instantiation of a UEFI-conformant BIOS. The licensed Intel UDK2010 includes open-source components (downloadable from www.tianocore.org), as well as closed-source components that are available only to Tiano direct licensees. A stable build of the EDK II project, the licensed Intel UDK2010 includes the platform package implementation for each Intel platform upon which it has been validated. The licensed Intel UDK2010 has also been validated on a variety of operating systems and application software. In addition to the full instantiation of the BIOS, the licensed Intel UDK2010 includes additional hardware-specific driver code samples, libraries, and source code. Intel UDK 2010 licenses are no longer being offered.