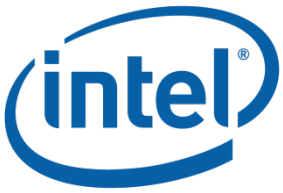

UEFI Driver Development Guide for Graphics Controller Device Classes

Nov 2011

Version 1.0

Table of Contents

List of Tables

UEFI Driver Development Guide for Graphics Controller Device Classes

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

This document is a "short list" -- a reference list based on the UEFI 2.3 specification. More information about required and recommended protocols, the driver binding model, and boot services is available in the Intel® UEFI Driver Writers 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.

Overview

Most graphics controllers are PCI controllers. The graphics drivers managing those controllers are also PCI drivers. However, while most graphics controllers are PCI controllers, graphics controllers can make use of other buses, such as USB buses. Graphics drivers have these characteristics:

- UEFI graphics drivers follow the UEFI driver model.
- Depending on the adapter that the driver manages, a graphics driver can be categorized as either a single or a multiple output adapter.
- The graphics driver must create child handles for each output.
- Graphics drivers must create child handles for some of the graphics output ports, and attach the Graphics Output Protocol (GOP Protocol), EDID Discovered Protocol, and EDID Active Protocol to each active handle that the driver produced.
- Graphics drivers are chip-specific because of the requirement to initialize and manage the graphics device.

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:

- Graphic output devices: Simple text, graphics output
- 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.

Requirements, recommendations, and optional elements

The following table provides a brief overview of the protocols that must be supported by graphics controller device drivers.

Table 1. Required UEFI protocols for graphics controller device drivers

Required protocol	Description
EFI_DRIVER_BINDING_PROTOCOL	Provides services that determine whether the driver supports a given graphics controller, as well as functions for starting and stopping the controller. Device drivers must ignore the *RemainingDevicePath* parameter that is passed into the *Supported()* and *Start()* services of the Driver Binding Protocol. All drivers that follow the UEFI driver model must support the *Stop()* service.
EFI_DEVICE_PATH_PROTOCOL	Provides the location of the device.
EFI_GRAPHICS_OUTPUT_PROTOCOL	Provides support for graphics console devices. This protocol does not require VGA hardware, and is not restricted to PCI devices. The driver must use a unique Graphics Output Protocol to represent each video frame buffer in the system; each buffer is driven out to one or more video output devices. Note that graphics output may be required as part of the startup of an operating system.
EFI_EDID_DISCOVERED_PROTOCOL	Contains the EDID information that was retrieved from the video output device. This information may differ from the EDID Active Protocol since the EDID Active Protocol will take into account any interaction with the EDID Override Protocol that was consumed by this driver. The EDID Discovered Protocol must exist on each handle that represents a video output and must only represent a single video output device.
EFI_EDID_ACTIVE_PROTOCOL	Provides information to the system about a video output device. The information will be retrieved from either the EDID Discovered Protocol or the EDID Override Protocol. The minimum valid size of EDID information is 128 bytes.

The following table lists additional implementation requirements for graphics controller device drivers.

Table 2. Additional implementation requirements for graphics controller 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 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 graphics controller device drivers

Recommended / required protocol	Description and notes
EFI_LOADED_IMAGE_PROTOCOL	This protocol is produced by the UEFI core as part of the LoadImage()/StartImage() calls when the UEFI Driver is loaded. A graphics controller device driver must consume this protocol if there are multiple images. 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. If the driver loads an image, for good coding practices, the driver should also unload the image when done. Note that it is recommended that the *EFI_LOADED_IMAGE_PROTOCOL.Unload()* 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.
EFI_COMPONENT_NAME2_PROTOCOL and *EFI_COMPONENT_NAME_PROTOCOL*	The Component Name2 Protocol replaces the older Component Name Protocol. The Component Name Protocols provide functions for retrieving a human-readable name of a driver and the controllers that a driver is managing. 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.
EFI_DRIVER_DIAGNOSTICS2_PROTOCOL and *EFI_DRIVER_DIAGNOSTICS_PROTOCOL*	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. If the driver will allow the UEFI shell command *drvdiag* 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. 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.
EFI_FIRMWARE_UPDATE_PROTOCOL	Provides an abstraction for a device to provide firmware management support. This protocol makes it easier for Information Technology (IT) departments to manage devices, including performing firmware updates.

The following table lists additional implementation elements that are strongly recommended for graphics controller device drivers.

Table 4. Additional implementation recommendations for graphics controller device drivers

Additional implementation recommendations	Description and notes
Implement an Exit Boot Services event in the driver's entry point	This event is recommended for all UEFI device drivers because it helps improve the hand-off of control to the operating system. 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. 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.
Use private data structures	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.
Optimize the driver for scrolling	Scrolling is one of the most common operations to occur on a pre-boot graphics adapter due to the common use of text base consoles.

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

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

Optional but important protocol	Description and/or notes
EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL	Specifies the device path that was used when a PE/COFF image was loaded through the UEFI boot service LoadImage().
EFI_DEVICE_PATH_UTILITIES_PROTOCOL	Provides interfaces to create and manipulate UEFI device paths and UEFI device path nodes.
EFI_DRIVER_BINDING_PROTOCOL Start() function *RemainingDevicePath* parameter	In general, if it is possible to support the Start() function's RemainingDevicePath parameter, the driver should do so in order to support the rapid boot capability in the UEFI driver model.
Human Interface Infrastructure (HII) protocols	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. If you implement HII protocols, you must also implement the Driver Health Protocol.
EFI_FORM_BROWSER2_PROTOCOL	Allows direction of the configuration driver to use either the HII database or use the passed-in packet of data. If you implement HII protocols, you must implement the Form Browser2 Protocol.

Table 5. Optional UEFI protocols for graphics controller device drivers that are important — *continued*

Optional but important protocol	Description and/or notes
EFI_DRIVER_HEALTH_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. This 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. If you implement HII protocols, you must implement the Driver Health Protocol.

The platform firmware determines the role of the graphics driver by changing how the *Start()* function is called, implementing HII functionality and/or the Simple Text Output Protocol on top of the Graphics Output Protocol, and producing the EDID Override protocol.

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 graphics controller devices.

Table 6. Platform requirements for supporting graphics controller device drivers

If the platform includes or supports...	Required protocols and other required elements
EFI_EDID_OVERRIDE_PROTOCOL	If the platform firmware produces the EDID Override Protocol, the graphics driver must use the data produced by that protocol when populating the EDID Active Protocol.
User-configurable settings	Human Interface infrastructure (HII) protocols *EFI_DRIVER_HEALTH_PROTOCOL EFI_FORM_BROWSER2_PROTOCOL*
Graphics console devices	*EFI_SIMPLE_TEXT_OUTPUT PROTOCOL* If the platform supports graphics console devices, the platform must contain a driver that consumes the Graphics Output Protocol and produces the Simple Text Output Protocol, even if the Graphics Output Protocol is produced by an external driver.
Simple text output	*EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL* It is recommended that the platform firmware be responsible for producing simple text. HII functionality provides a well-formed set of data and forms for configuring UEFI drivers. The Simple Text Output Protocol remains the delivery method to get that data onto the screen for the user to see. Platform firmware must implement the Simple Text Output Protocol along with the GOP protocol produced by the graphics driver. This allows for optimization in terms of language display requirements. For example, if the firmware is running in Mandarin mode, the firmware will know that and load the appropriate UNICODE character set. (The graphics driver has no easy way to adjust itself for this type of issue.)

Table 6. Platform requirements for supporting graphics controller device drivers — *continued*

If the platform includes or supports...	Required protocols and other required elements
Debugging	*EFI_DEBUG_SUPPORT_PROTOCOL* *EFI_DEBUGPORT_PROTOCOL* EFI image information table
Overriding the default driver to the controller matching algorithm provided by the UEFI driver binding model	*EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL*
Higher priority than the Bus Specific Driver Override Protocol	The *EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL* must be produced on the same handle as the *EFI_DRIVER_BINDING_PROTOCOL*.
Compression of the firmware image	*EFI_DECOMPRESS_PROTOCOL* This protocol provides interfaces to decompress an image that was compressed using the UEFI compression algorithm. If the firmware image is compressed, this protocol is required. The protocol should be implemented by platform firmware, not by the graphics driver.
Authentication of UEFI images, and the platform potentially supports more than one OS loader	If the platform requires secure booting, you should implement the methods described in the UEFI specification for authentication of UEFI variables.
Digital signatures	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."
A driver written in EBC	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. Note: If an EBC interpreter is implemented, then it must produce the EBC Protocol interface.
Pointer device (such as a touch screen) that are part of console support	*EFI_SIMPLE_POINTER_PROTOCOL* This protocol must be implemented as part of the graphics console support. However, typically, this protocol would not be implemented in the graphics driver. Instead, it is recommended that you write a separate driver to manage such devices.

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 graphics controller 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.
- Per spec, if a service returns an error, the output parameters are undefined. When
- 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.