UEFI Driver Development Guide for Network Boot Devices

Nov 2011

Version 1.0

Table of Contents

List of Tables

List of Figures

UEFI Driver Development Guide for Network Boot Devices

This development guide lists required, recommended, and optional UEFI protocols and elements for network boot device drivers, such as UNDI (universal network driver interface), SNP (Simple Network Protocol), and MNP (Managed Network Protocol) device drivers. This guide 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, UEFI driver model, and UEFI boot services is available in the Intel® UEFI Driver Writer’s Guide, which is expected to be available later this year on www.tianocore.org. UEFI driver code samples and templates are available in the Intel UDK2010 Developer's Kit. Complete information on all required protocols and other elements for UEFI drivers is provided in the current UEFI specification.

Overview

Network drivers are bus drivers that are used for add-in network cards, serially attached network cards (USB), cardbus network cards, and LAN-on-mainboard network devices. Most network controllers are PCI or USB devices, and the network drivers managing them are also PCI or USB drivers.

Network device drivers could be implemented as UNDI, SNP or MNP device drivers. The architecture of network device drivers and network stack is shown in Figure 1 (next page). It is recommended that network device driver is implemented as UNDI device driver, so as to reuse the SNP and MNP component in system Firmware.

Network device drivers are unique compared to all others peripheral drivers:

- They can be runtime drivers. This is not a requirement of a LAN driver, but network drivers are the only class of driver that is documented in the UEFI specification with that option.
- They can have non-protocol interfaces. The Network Interface Identifier Protocol defines the entrypoint to the UNDI structure, but UNDI is not a protocol. When a c is used, the CDB has the abstractions for the driver to use to call back into the hardware.

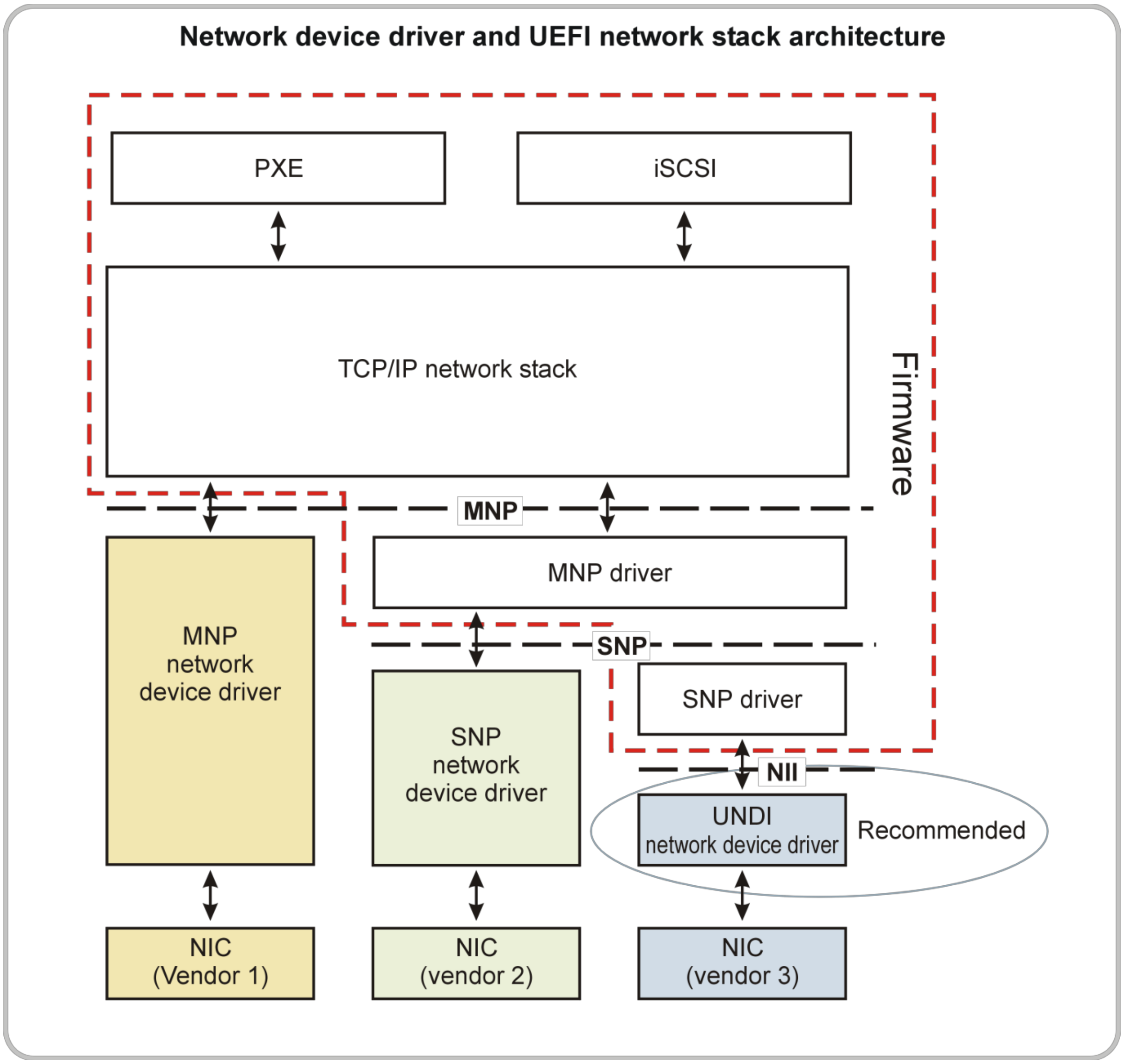

Figure 1. Network device driver and UEFI network stack architecture

Requirements

A UEFI driver is required for any network boot device needed for the boot process to complete.

Note that independent hardware vendors (IHVs) can choose not to implement all 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 network drivers, the CPU address and the PCI address do not have to be the same. They 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 briefly describes the required protocols for network boot device drivers.

Table 1. Required UEFI protocols for UEFI UNDI network boot device drivers

Required protocol	Description
EFI_DRIVER_BINDING_PROTOCOL	Provides functions for starting and stopping the driver, as well as a function for determining whether the driver can manage a particular controller. All network boot device drivers are required to implement the Driver Binding Protocol. However, the protocol for network boot device drivers is simpler than many other device drivers. For network drivers, the protocol produces the Device Path protocol and whichever one of the two interfaces the network driver has chosen as the interface. Network boot device drivers must ignore the *RemainingDevicePath* parameter that is passed into the *Supported()* and *Start()* services of the Driver Binding Protocol. All network boot device drivers that follow the UEFI driver model must support the *Stop()* service.
EFI_DEVICE_PATH_PROTOCOL	Provides the location of the device.
EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL	UNDI is required in order to boot from a network device. The Network Interface Identifier Protocol (NII) is used to get type and revision information about the underlying network interface. This protocol is required only if the underlying network interface is 16-bit UNDI, 32/64-bit S/W UNDI, or H/W UNDI. An instance of the Network Interface Identifier protocol must be created for each physical external network interface that is controlled by the !PXE structure. The !PXE structure is defined in the 32/64-bit UNDI Specification.

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

Table 2. Required UEFI protocols for UEFI SNP network boot device drivers

Required protocol	Description
EFI_DRIVER_BINDING_PROTOCOL	Provides functions for starting and stopping the driver, as well as a function for determining whether the driver can manage a particular controller. All network boot device drivers are required to implement the Driver Binding Protocol. However, the protocol for network boot device drivers is simpler than for many other device drivers. For network drivers, the protocol produces the Device Path protocol and whichever one of the two interfaces the network driver has chosen as the interface. Network boot device drivers must ignore the *RemainingDevicePath* parameter that is passed into the *Supported()* and *Start()* services of the Driver Binding Protocol. All network boot device drivers that follow the UEFI driver model must support the *Stop()* service.
EFI_DEVICE_PATH_PROTOCOL	Provides the location of the device.
EFI_SIMPLE_NETWORK_PROTOCOL	Provides a packet-level interface to a network adapter for a bootable device. The UEFI specification defines multiple interfaces, but this document focuses on boot devices. Booting from a network requires the Simple Network Protocol.

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 network boot device drivers.

Table 3. Required UEFI protocols for UEFI MNP network boot device drivers

Required protocol	Description
EFI_DRIVER_BINDING_PROTOCOL	Provides functions for starting and stopping the driver, as well as a function for determining whether the driver can manage a particular controller. All network boot device drivers are required to implement the Driver Binding Protocol. However, the protocol for network boot device drivers is simpler than for many other device drivers. For network drivers, the protocol produces the Device Path protocol and whichever one of the two interfaces the network driver has chosen as the interface. Network boot device drivers must ignore the *RemainingDevicePath* parameter that is passed into the *Supported()* and *Start()* services of the Driver Binding Protocol. All network boot device drivers that follow the UEFI driver model must support the *Stop()* service.

Table 3. Required UEFI protocols for UEFI MNP network boot device drivers — *continued*

Required protocol	Description
EFI_DEVICE_PATH_PROTOCOL	Provides the location of the device.
EFI_MANAGED_NETWORK_SERVICE_BINDING_PROTOCOL	The MNP provides raw asynchronous network packet I/O services. The services make it possible for multiple-event-driven drivers and applications to access and use the system network interfaces at the same time.
EFI_VLAN_CONFIG_PROTOCOL	Provides manageability interface for VLAN configuration.

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

Table 4. Additional implementation requirements for UEFI network boot 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 implement an Exit Boot Services event in Start() function of 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.
The driver must convert pointers for dynamically allocated memory into runtime pointers.	If the driver includes dynamically allocated memory, pointers to those locations must be converted into runtime pointers (virtual address pointers). Use the UEFI runtime service *ConvertPointer()* routine to handle this conversion. The conversion must be done bottom-up since the virtual pointers are not valid until the SetVirutalAddressMap event returns to the OS. The OS is obligated not to call into a runtime UEFI driver until the system is in virtual mode.
UNDI network drivers must track the status of buffers used for data transfer.	UNDI drivers transfer data in the system through memory buffers. To do this, the UNDI driver often allocates many buffers for data transfer. If those buffers are not tracked properly, it is possible to loose them in the shuffle, and they will not be returned to the system memory management. This can cause a memory leak. When a buffer is being used (taken off the waiting queue and made active) there is a chance of losing the pointer to that buffer in the process, which again, will cause a memory leak. When transmitting data, the UNDI driver must track which buffers have been completed, and return those buffer addresses from the GetStatus API. This allows the top level stack to reuse or de-allocate the buffer and avoid leaking memory.

Table 4. Additional implementation requirements for UEFI network boot device drivers — *continued*

Additional implementation requirements	Description
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 network boot device drivers.

Table 5. Recommended or required UEFI protocols for all UEFI network boot device drivers

Recommended / required protocol	Description and notes
EFI_LOADED_IMAGE_PROTOCOL.Unload()	This protocol is produced by the UEFI core as part of the LoadImage()/StartImage() calls when the UEFI Driver is loaded. A network boot 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. These 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.

Table 5. Recommended or required UEFI protocols for all UEFI network boot device drivers — *continued*

Recommended / required protocol	Description and notes
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 of 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.
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_DRIVER_HEALTH_PROTOCOL	This protocol is required if HII is implemented. 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. 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. 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.

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

Table 6. Additional implementation recommendations for all UEFI network boot device drivers

Additional implementation recommendations	Description and notes
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.

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

Table 7. Optional UEFI protocols for UEFI network boot device drivers that are important

Optional but important protocols	Description and/or notes
EFI_VLAN_CONFIG_PROTOCOL	This protocol may be implemented if VLAN is supported in hardware. VLAN provides a manageability interface for VLAN configuration.
EFI_FIRMWARE_UPDATE_PROTOCOL	Provides an abstraction for a device to provide firmware management support.

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 network boot devices.

Table 8. Platform requirements for supporting UEFI network boot device drivers

If the platform includes or supports...	Required protocols and other required elements
Validating a boot image received through a network device	*EFI_SIMPLE_NETWORK_PROTOCOL* *EFI_PXE_BASE_CODE_PROTOCOL* *EFI_BIS_PROTOCOL* UNDI interface, which is used by the driver to communicate with the PXE protocols provided by the platform firmware. Note that an external driver may produce the UNDI interface.
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*.
Authentication of UEFI images, and the platform potentially supports more than one OS loader	If the platform requires secure boot, you should implement the methods described in the UEFI specification for UEFI authenticated variables.
Digital signatures	If the platform requires secure boot, 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.

Table 8. Platform requirements for supporting UEFI network boot device drivers — *continued*

If the platform includes or supports...	Required protocols and other required elements
A bus that supports option ROMs for one or more UEFI drivers	*EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL* This protocol is produced by the PCI Bus Driver, and consumed by the DXE Core as part of the driver-to-controller connection process. This protocol lets the driver in the PCI device know which option ROM has the higher priority. If a bus supports option ROMs for one or more UEFI drivers, then the bus driver for that bus type must implement the Bus Specific Driver Override Protocol. (This protocol is not produced by the device driver.) This protocol provides a mechanism for bus drivers to override the default driver selection performed by the ConnectController() boot service. All of the drivers returned by this protocol have a higher precedence than drivers found in the general EFI Driver Binding search algorithm. All drivers returned by this protocol have a lower precedence than those drivers returned by the EFI Platform Driver Override Protocol.

Reminders, tips, do's and don'ts

Dynamic media detection support was added to the UNDI interface in UEFI 2.3 specification. To support this feature, the following definitions in the UEFI specification should be supported in all UNDI implementations:

- *PXE_OPFLAGS_GET_MEDIA_STATUS*
- *PXE_STATFLAGS_GET_STATUS_NO_MEDIA_SUPPORTED*
- *PXE_STATFLAGS_GET_STATUS_NO_MEDIA*

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 drivers for network boot devices:

- 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.