VOLUME 2: Platform Initialization Specification

Driver Execution Environment Core Interface

Version 1.1 Errata B

Revision History

Revision	Revision History	Date
1.0	Initial public release.	8/21/06
1.0 errata	Mantis tickets: • M47 dxe_dispatcher_load_image_behavior • M48 Make spec more consistent GUID & filename. • M155 FV_FILE and FV_ONLY: Change subtype number back to th:e original one. • M171Remove 10 us lower bound restriction for the TickPeriod in the Metronome • M178 Remove references to tail in file header and made file checksum for the data • M183 Vol 1-Vol 5: Make spec more consistent. • M192 Change PAD files to have an undefined GUID file name and update all FV	10/29/07
1.1	Mantis tickets: • M39 (Updates PCI Hostbridge & PCI Platform) • M41 (Duplicate 167) • M42 Add the definition of theDXE CIS Capsule AP & Variable AP • M43 (SMbios) • M46 (SMM error codes) • M163 (Add Volume 4--SMM • M167 (Vol2: adds the DXE Boot Services Protocols--new Chapter 12) • M179 (S3 boot script) • M180 (PMI ECR) • M195 (Remove PMI references from SMM CIS) • M196 (disposable-section type to the FFS)	11/05/07
1.1 correction	Restore (missing) MP protocol	03/12/08

1.1 Errata	• 230 Updated to Volume 4, section 4.2, ReportStatusCode • 231Parameter/description updates for Volume 4, section 4.3, ReadSaveState() & WriteSaveState(), Parameters • 232 SMM I/O Protocol Updates • 233 Volume 4, Section 5.2 & 5.3 Updates • 234 Volume 4, Section 5.5 Misc. Errata • 235Volume 4, Chapter 8 Should Be Integrated Into Volume 3, Section 2.1.4.1, 2.1.5.1 and 3.2.5 • 236 Volume 4, Section 9.5.1, 9.6, 9.7, 9.8 and 9.9 Formatting • 238 CpuSaveStateFormat deprecated in Vol4 of SMM PI1.1 draft • 239 rename EFI_SMM_HANDLER_ENTRY_POINT to be EFI_SMM_HANDLER_ENTRY_POINT2 in Vol4 SMM of PI1.1 • 240 PI1.1 Vol4 typos • 244 Replace EFI_FIRMWARE_VOLUME_INFO_PPI with EFI_PEI_FIRMWARE_VOLUME_INFO_PPI • 250 PEI_SPECIFICATION_MINOR_REVISION should be 10 • 251 Firmware File Type Table (Volume 3, 2.1.4.1, Table 1) Should Not Contain Section Information • 252 Volume 3, Table 2 (2.1.5.1) does not contain EFI_SECTION_DISPOSABLE • 253 EFI_SECTION_PIC has incorrect typedef • 254 ReInstallPpi() has incorrect prototype • 255 NotifyPpi() has the incorrect prototype • 256 CreateHob() has incorrect prototype • 257 PEI Specification, Section 4.2.1 and Section 4.2.2 should be peers of 4.1, 4.3, etc. • 258 CreateHob() refers to non-existent specification. • 259 FfsFindNextFile() Parameters Are Incorrect • 260 FfsFindSectionData() has incorrect parameter description • 261 AllocatePages() (PEI) refers to a non-existent specification and non-existent function. • 262 FfsGetVolumeInfo() missing return status codes • 263 EFI_PEI_NOTIFY_DESCRIPTOR and EFI_PEI_PPI_DESCRIPTOR prototypes are incorrect • 264 EFI_PEI_SERVICES: Remove references to "future installed services" from prototype • 265 EFI_FV_BLOCK_MAP definition does not exist • 267 Invalid References To the PI Firmware Storage Specification • 268 GUIDED_SECTION_EXTRACTION_PROTOCOL missing 'EFI_' prefix • 269 References to EFI_FIRMWARE_VOLUME_PROTOCOL should be EFI_FIRMWARE_VOLUME2_PROTOCOL • 272 Various fixes for Communicate() in PI 1.1, Volume 4 • 273 EFI_SMM_CONTROL2_PROTOCOL Errata • 274 Miscellanous SMST Errata from Volume 4, Section 3.2 • 275 Chapter heading for DXE ReportStatusCode function • 276 EFI_STATUS_CODE_RUNTIME_PROTOCOL_GUID has extra ';' • 277 Remove references to "Framework" and "Framework-based" in Volume 5	04/25/08

1.1 Errata	Mantis tickets • 204 Stack HOB update 1.1errata • 225 Correct references from EFI_FIRMWARE_VOLUME_PROTOCOL to EFI_FIRMWARE_VOLUME2_PROTOCOL • 226 Remove references to Framework • 227 Correct protocol name GUIDED_SECTION_EXTRACTION_PROTOCOL • 228 insert"typedef" missing from some typedefs in Volume 3 • 243 Define interface "EFI_PEI_FV_PPI" declaration in PI1.0 FfsFindNextVolume() • 285 Time quality of service in S3 boot script poll operation • 287 Correct MP spec, PIVOLUME 2:Chapter 13.3 and 13.4 - return error language • 290 PI Errata • 305 Remove Datahub reference • 336 SMM Control Protocol update • 345 PI Errata • 353 PI Errata • 360 S3RestoreConfig description is missing • 363 PI Volume 1 Errata • 367 PCI Hot Plug Init errata • 369 Volume 4 Errata • 380 SMM Development errata • 381 Errata on EFI_SMM_SAVE_STATE_IO_INFO	01/13/09
1.1 Errata	• 247 Clarification regarding use of dependency expression section types with firmware volume image files • 399 SMBIOS Protocol Errata • 405 PIWG Volume 5 incorrectly refers to EFI_PCI_OVERRIDE_PROTOCOL • 422 TEMPORARY_RAM_SUPPORT_PPI is misnamed • 428 Volume 5 PCI issue • 430 Clarify behavior w/ the FV extended header	02/23/09
1.1 Errata	• 407 Add LMA Pseudo-Register to SMM Save State Protocol • 455 Clarify InstallPeiMemory() • 465 Correct PMI Interface • 466 Add EXTENDED_SAL_PROC definition, etc • 467 Vol2 & Vol3 Errata	05/22/09

1.1 errata	• 345 PI1.0 errata • 468 Issues on proposed PI1.2 ACPI System Description Table Protocol • 492 Add Resource HOB Protectability Attributes • 494 Vol. 2 Appendix A Clean up • 495 Vol 1: update HOB reference • 380 • 501 Clean Up SetMemoryAttributes() language Per Mantis 489 (from USWG) • 502 Disk info • 503 typo • 504 remove support for fixed address resources • 509 PCI errata – execution phase • 510 PCI errata - platform policy • 511 PIC TE Image clarification/errata • 520 PI Errata • 521 Add help text for EFI_PCD_PROTOCOL for GetNextTokenSpace • 525 Itanium ESAL, MCA/INIT/PMI errata • 526 PI SMM errata • 529 PCD issues in Volume 3 of the PI1.2 Specification • 541 Volume 5 Typo • 543 Clarification around usage of FV Extended header • 550 Naming conflicts w/ PI SMM	12/16/09
1.1 Errata B	• 363 PI volume 1 errata • 365 UEFI Capsule HOB • 381 PI1.1 Errata on EFI_SMM_SAVE_STATE_IO_INFO • 482 One other naming inconsistency in the PCD PPI declaration • 483 PCD Protocol / PPI function name synchronization..... • 496 Boot mode description • 497 Status Code additions • 548 Boot firmware volume clarification • 552 MP services • 553 Update text to PEI • 554 update return code from PEI AllocatePages • 555 Inconsistency in the S3 protocol • 561 Minor update to PCD->SetPointer • 571 duplicate definition of EFI_AP_PROCEDURE in DXE MP (volume2) and SMM (volume 4) • 581 EFI_HOB_TYPE_LOAD_PEIM ambiguity • 591ACPI Protocol Name collision • 592 More SMM name conflicts • 593 A couple of ISA I/O clarifications • 595 SMM driver entry point clarification • 596 Clarify ESAL return codes • 602 SEC->PEI hand-off update • 604 EFI_NOT_SUPPORTED versus EFI_UNSUPPORTED	(2/24/10) 5/27/10

1.1 Errata B	• 628 ACPI SDT protocol errata • 629 Typos in PCD GetSize()	5/27/10

Specification Volumes

The **Platform Initialization Specification** is divided into volumes to enable logical organization, future growth, and printing convenience. The **Platform Initialization Specification** consists of the following volumes:

VOLUME 1: Pre-EFI Initialization Core Interface

VOLUME 2: Driver Execution Environment Core Interface

VOLUME 3: Shared Architectural Elements

VOLUME 4: System Management Mode

VOLUME 5: Standards

Each volume should be viewed in the context of all other volumes, and readers are strongly encouraged to consult the entire specification when researching areas of interest. Additionally, a single-file version of the **Platform Initialization Specification** is available to aid search functions through the entire specification.

Contents

Figures

Tables

1 Introduction

1.1 Overview

This specification defines the core code and services that are required for an implementation of the driver execution environment (DXE) phase of the Unified Extensible Firmware Interface (UEFI) Foundation. This DXE core interface specification (CIS) does the following:

- Describes the basic components of the DXE phase.
- Provides code definitions for services and functions that are architecturally required by the *Unified Extensible Firmware Interface Specification* (UEFI 2.0 specification).
- Presents a set of backward-compatible extensions to the UEFI 2.0 specification.
- Describes the machine preparation that is required for subsequent phases of firmware execution.

See “Organization of the DXE CIS” for more information.

1.2 Organization of the DXE CIS

This DXE core interface specification (CIS) is organized as shown in Table 1. Because the DXE Foundation is just one component of a PI Architecture-based firmware solution, there are a number of additional specifications that are referred to throughout this document.

Table 1. Organization of the DXE CIS

Book	Description
"Overview" on page 7	Describes the major components of DXE, including the boot manager, firmware core, protocols, and requirements.
"Boot Manager" on page 13	Describes the boot manager, which is used to load UEFI drivers, UEFI applications, and UEFI OS loaders.
"UEFI System Table" on page 15	Describes the DXE Service table.
"Services - Boot Services" on page 27	Describes specific event types for DXE Foundation.
"Runtime Capabilities" on page 31	Contains definitions of a runtime protocol for status code support.
"Services - DXE Services" on page 33	Contains definitions for the fundamental services that are present in a DXE-compliant system before an OS is booted.
"Protocols - Device Path Protocol" on page 73	Defines the device path extensions required by the DXE Foundation.
"DXE Foundation" on page 75	Describes the DXE Foundation that consumes HOBs, Firmware Volumes, and DXE Architectural Protocols to produce an UEFI System Table, UEFI Boot Services, UEFI Runtime Services, and the DXE Services.
"DXE Dispatcher" on page 91	Describes the DXE Dispatcher that is responsible for loading and executing DXE drivers from Firmware Volumes.
"DXE Drivers" on page 115	Describes the different classes of DXE drivers that may be stored in Firmware Volumes.
"DXE Architectural Protocols" on page 117	Describes the Architectural Protocols that are produced by DXE drivers. They are also consumed by the DXE Foundation to produce the UEFI Boot Services, UEFI Runtime Services, and DXE Services.
"DXE Runtime Protocols" on page 187	Lists success, error, and warning codes returned by DXE and UEFI interfaces.
"Dependency Expression Grammar" on page 193	Describes the BNF grammar for a tool that can convert a text file containing a dependency expression into a dependency section of a DXE driver stored in a Firmware Volume.

1.3 Target Audience

This document is intended for the following readers:

- IHVs and OEMs who will be implementing DXE drivers that are stored in firmware volumes.
- BIOS developers, either those who create general-purpose BIOS and other firmware products or those who modify these products for use in various vendor architecture–based products.

1.4 Conventions Used in this Document

This document uses the typographic and illustrative conventions described below.

1.4.1 Data Structure Descriptions

Supported processors are “little endian” machines. This distinction means that the low-order byte of a multibyte data item in memory is at the lowest address, while the high-order byte is at the highest address. Some supported processors may be configured for both “little endian” and “big endian” operation. All implementations designed to conform to this specification will use “little endian” operation.

In some memory layout descriptions, certain fields are marked *reserved*. Software must initialize such fields to zero and ignore them when read. On an update operation, software must preserve any reserved field.

The data structures described in this document generally have the following format:

STRUCTURE NAME: The formal name of the data structure.

Summary: A brief description of the data structure.

Prototype: A “C-style” type declaration for the data structure.

Parameters: A brief description of each field in the data structure prototype.

Description: A description of the functionality provided by the data structure, including any limitations and caveats of which the caller should be aware.

Related Definitions: The type declarations and constants that are used only by this data structure.

1.4.2 Protocol Descriptions

The protocols described in this document generally have the following format:

Protocol Name: The formal name of the protocol interface.

Summary: A brief description of the protocol interface.

GUID: The 128-bit Globally Unique Identifier (GUID) for the protocol interface.

Protocol Interface Structure: A “C-style” data structure definition containing the procedures and data fields produced by this protocol interface.

Parameters: A brief description of each field in the protocol interface structure.

Description: A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.

Related Definitions: The type declarations and constants that are used in the protocol interface structure or any of its procedures.

1.4.3 Procedure Descriptions

The procedures described in this document generally have the following format:

ProcedureName(): The formal name of the procedure.

Summary: A brief description of the procedure.

Prototype: A "C-style" procedure header defining the calling sequence.

Parameters: A brief description of each field in the procedure prototype.

Description: A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.

Related Definitions: The type declarations and constants that are used only by this procedure.

Status Codes Returned: A description of any codes returned by the interface. The procedure is required to implement any status codes listed in this table. Additional error codes may be returned, but they will not be tested by standard compliance tests, and any software that uses the procedure cannot depend on any of the extended error codes that an implementation may provide.

1.4.4 Instruction Descriptions

A dependency expression instruction description generally has the following format:

InstructionName The formal name of the instruction.

Syntax: A brief description of the instruction.

Description: A description of the functionality provided by the instruction accompanied by a table that details the instruction encoding.

Operation: Details the operations performed on operands.

Behaviors and Restrictions: An item-by-item description of the behavior of each operand involved in the instruction and any restrictions that apply to the operands or the instruction.

1.4.5 Pseudo-Code Conventions

Pseudo code is presented to describe algorithms in a more concise form. None of the algorithms in this document are intended to be compiled directly. The code is presented at a level corresponding to the surrounding text.

In describing variables, a *list* is an unordered collection of homogeneous objects. A *queue* is an ordered list of homogeneous objects. Unless otherwise noted, the ordering is assumed to be First In First Out (FIFO).

Pseudo code is presented in a C-like format, using C conventions where appropriate. The coding style, particularly the indentation style, is used for readability and does not necessarily comply with an implementation of the *Unified Extensible Firmware Interface Specification* (UEFI 2.0 specification).

1.4.6 Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain text	The normal text typeface is used for the vast majority of the descriptive text in a specification.
Plain text (blue)	In the online help version of this specification, any plain text that is underlined and in blue indicates an active link to the cross-reference. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification.
Bold	In text, a **Bold** typeface identifies a processor register name. In other instances, a **Bold** typeface can be used as a running head within a paragraph.
Italic	In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.
`BOLD Monospace`	Computer code, example code segments, and all prototype code segments use a `BOLD Monospace` typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.
`Bold Monospace`	In the online help version of this specification, words in a `Bold Monospace` typeface that is underlined and in blue indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification. Also, these inactive links in the PDF may instead have a `Bold Monospace` appearance that is underlined but in dark red. Again, these links are not active in the PDF of the specification.
`Italic Monospace`	In code or in text, words in *`Italic Monospace`* indicate placeholder names for variable information that must be supplied (i.e., arguments).
`Plain Monospace`	In code, words in a `Plain Monospace` typeface that is a dark red color but is not bold or italicized indicate pseudo code or example code. These code segments typically occur in one or more separate paragraphs.

1.5 Requirements

This document is an architectural specification that is part of the Platform Initialization Architecture (PI Architecture) family of specifications defined and published by the Unified EFI Forum. The primary intent of the PI Architecture is to present an interoperability surface for firmware components that may originate from different providers. As such, the burden to conform to this

specification falls both on the producer and the consumer of facilities described as part of the specification.

In general, it is incumbent on the producer implementation to ensure that any facility that a conforming consumer firmware component might attempt to use is present in the implementation. Equally, it is incumbent on a developer of a firmware component to ensure that its implementation relies only on facilities that are defined as part of the PI Architecture. Maximum interoperability is assured when collections of conforming components are designed to use only the required facilities defined in the PI Architecture family of specifications.

As this document is an architectural specification, care has been taken to specify architecture in ways that allow maximum flexibility in implementation for both producer and consumer. However, there are certain requirements on which elements of this specification must be implemented to ensure a consistent and predictable environment for the operation of code designed to work with the architectural interfaces described here.

For the purposes of describing these requirements, the specification includes facilities that are required, such as interfaces and data structures, as well as facilities that are marked as optional.

In general, for an implementation to be conformant with this specification, the implementation must include functional elements that match in all respects the complete description of the required facility descriptions presented as part of the specification. Any part of the specification that is not explicitly marked as “optional” is considered a required facility.

Where parts of the specification are marked as “optional,” an implementation may choose to provide matching elements or leave them out. If an element is provided by an implementation for a facility, then it must match in all respects the corresponding complete description.

In practical terms, this means that for any facility covered in the specification, any instance of an implementation may only claim to conform if it follows the normative descriptions completely and exactly. This does not preclude an implementation that provides additional functionality, over and above that described in the specification. Furthermore, it does not preclude an implementation from leaving out facilities that are marked as optional in the specification.

By corollary, modular components of firmware designed to function within an implementation that conforms to the PI Architecture are conformant only if they depend only on facilities described in this and related PI Architecture specifications. In other words, any modular component that is free of any external dependency that falls outside of the scope of the PI Architecture specifications is conformant. A modular component is not conformant if it relies for correct and complete operation upon a reference to an interface or data structure that is neither part of its own image nor described in any PI Architecture specifications.

It is possible to make a partial implementation of the specification where some of the required facilities are not present. Such an implementation is non-conforming, and other firmware components that are themselves conforming might not function correctly with it. Correct operation of non-conforming implementations is explicitly out of scope for the PI Architecture and this specification.

2
Overview

2.1 Driver Execution Environment (DXE) Phase

The Driver Execution Environment (DXE) phase is where most of the system initialization is performed. Pre-EFI Initialization (PEI), the phase prior to DXE, is responsible for initializing permanent memory in the platform so that the DXE phase can be loaded and executed. The state of the system at the end of the PEI phase is passed to the DXE phase through a list of position-independent data structures called *Hand-Off Blocks* (HOBs). HOBs are described in detail in the *Platform Initialization Hand-Off Block Specification*.

There are several components in the DXE phase:

- "DXE Foundation"
- "DXE Dispatcher"
- A set of "DXE Drivers"

The DXE Foundation produces a set of Boot Services, Runtime Services, and DXE Services. The DXE Dispatcher is responsible for discovering and executing DXE drivers in the correct order. The DXE drivers are responsible for initializing the processor, chipset, and platform components as well as providing software abstractions for system services, console devices, and boot devices. These components work together to initialize the platform and provide the services required to boot an operating system. The DXE phase and Boot Device Selection (BDS) phases work together to establish consoles and attempt the booting of operating systems. The DXE phase is terminated when an operating system is successfully booted. The DXE Foundation is composed of boot services code, so no code from the DXE Foundation itself is allowed to persist into the OS runtime environment. Only the runtime data structures allocated by the DXE Foundation and services and data structured produced by runtime DXE drivers are allowed to persist into the OS runtime environment.

Figure 1 shows the phases that a platform with PI Architecture firmware will execute.

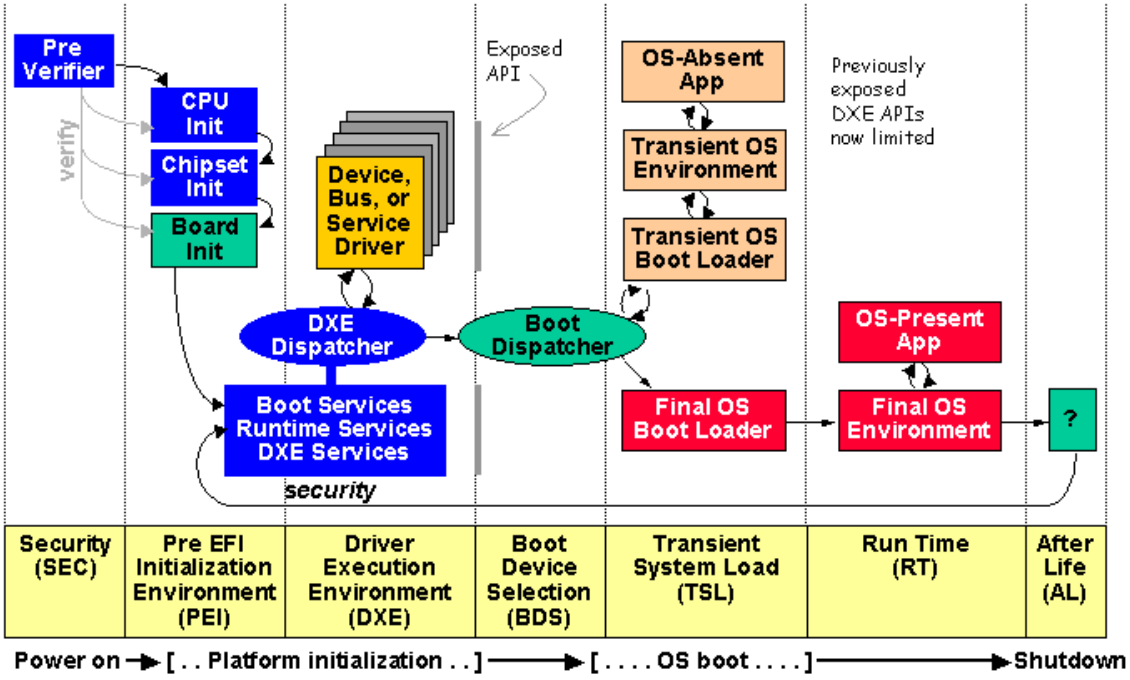

Figure 1. PI Architecture Firmware Phases

In a PI Architecture firmware implementation, the phase executed prior to DXE is PEI. This specification covers the transition from the PEI to the DXE phase, the DXE phase, and the DXE phase's interaction with the BDS phase. The DXE phase does not require a PEI phase to be executed. The only requirement for the DXE phase to execute is the presence of a valid HOB list. There are many different implementations that can produce a valid HOB list for the DXE phase to execute. The PEI phase in a PI Architecture firmware implementation is just one of many possible implementations.

2.2 UEFI System Table

2.2.1 Overview

The UEFI System Table is passed to every executable component in the DXE phase. The UEFI System Table contains a pointer to the following:

- "UEFI Boot Services Table"
- "UEFI Runtime Services Table"

It also contains pointers to the console devices and their associated I/O protocols. In addition, the UEFI System Table contains a pointer to the UEFI Configuration Table, and this table contains a list of GUID/pointer pairs. The UEFI Configuration Table may include tables such as the "DXE Services Dependencies" on page 87, HOB list, ACPI table, SMBIOS table, and SAL System table.

The UEFI Boot Services Table contains services to access the contents of the handle database. The handle database is where protocol interfaces produced by drivers are registered. Other drivers can use the UEFI Boot Services to look up these services produced by other drivers.

All of the services available in the DXE phase may be accessed through a pointer to the UEFI System Table.

2.2.2 UEFI Boot Services Table

Table 2 provides a summary of the services that are available through the UEFI Boot Services Table. These services are described in detail in the UEFI 2.0 specification. This DXE CIS makes a few minor, backward-compatible extensions to these services.

Table 2. UEFI Boot Services

UEFI Boot Services	Description
Task Priority	Provides services to increase or decrease the current task priority level. This can be used to implement simple locks and to disable the timer interrupt for short periods of time. These services depend on the "CPU Architectural Protocol" on page 121.
Memory	Provides services to allocate and free pages in 4 KB increments and allocate and free pool on byte boundaries. It also provides a service to retrieve a map of all the current physical memory usage in the platform.
Event and Timer	Provides services to create events, signal events, check the status of events, wait for events, and close events. One class of events is timer events, and that class supports periodic timers with variable frequencies and one-shot timers with variable durations. These services depend on the "CPU Architectural Protocol" on page 121, the "Timer Architectural Protocol" on page 150, the "Metronome Architectural Protocol" on page 136, and the "Watchdog Timer Architectural Protocol" on page 159.
Protocol Handler	Provides services to add and remove handles from the handle database. It also provides services to add and remove protocols from the handles in the handle database. Additional services are available that allow any component to lookup handles in the handle database, and open and close protocols in the handle database.
Image	Provides services to load, start, exit, and unload images using the PE/COFF image format. These services use the services of the "Security Architectural Protocol" on page 146 if it is present.
Driver Support	Provides services to connect and disconnect drivers to devices in the platform. These services are used by the BDS phase to either connect all drivers to all devices, or to connect only the minimum number of drivers to devices required to establish the consoles and boot an operating system. The minimal connect strategy is one possible mechanism to reduce boot time.

2.2.3 UEFI Runtime Services Table

Table 3 provides a summary of the services that are available through the UEFI Runtime Services Table. These services are described in detail in the UEFI 2.0 specification. One additional runtime service, Status Code Services, is described in this specification.

Table 3. UEFI Runtime Services

UEFI Runtime Services	Description
Variable	Provides services to look up, add, and remove environment variables from nonvolatile storage. These services depend on the Variable Architectural Protocol and the Variable Write Architectural Protocol.
Real Time Clock	Provides services to get and set the current time and date. It also provides services to get and set the time and date of an optional wake-up timer. These services depend on the Real Time Clock Architectural Protocol.
Reset	Provides services to shut down or reset the platform. These services depend on the Reset Architectural Protocol.
Virtual Memory	Provides services that allow the runtime DXE components to be converted from a physical memory map to a virtual memory map. These services can only be called once in physical mode. Once the physical to virtual conversion has been performed, these services cannot be called again. These services depend on the Runtime Architectural Protocol.

2.2.4 DXE Services Table

Table 4 provides a summary of the services that are available through the DXE Services Table. These are new services that are available in boot service time and are required only by the DXE Foundation and DXE drivers.

Table 4. DXE Services

DXE Services	Description
Global Coherency Domain	Provides services to manage I/O resources, memory-mapped I/O resources, and system memory resources in the platform. These services are used to dynamically add and remove these resources from the processor's global coherency domain.
Dispatcher	Provides services to manage DXE drivers that are being dispatched by the DXE Dispatcher.

2.3 DXE Foundation

The DXE Foundation is a boot service image that is responsible for producing the following:

- UEFI Boot Services
- UEFI Runtime Services
- DXE Services

The DXE Foundation consumes a HOB list and the services of the DXE Architectural Protocols to produce the full complement of UEFI Boot Services, UEFI Runtime Services, and DXE Services. The HOB list is described in detail in the *Platform Initialization Hand-Off Block Specification.*

The DXE Foundation is an implementation of UEFI. The DXE Foundation defined in this specification is backward compatible with the UEFI 2.0 specification. As a result, both the DXE Foundation and DXE drivers share many of the attributes of UEFI images. Because this specification makes extensions to the standard UEFI interfaces, DXE images will not be functional

on UEFI systems that are not compliant with this DXE CIS. However, UEFI images must be functional on all UEFI-compliant systems including those that are compliant with the DXE CIS.

2.4 DXE Dispatcher

The DXE Dispatcher is one component of the DXE Foundation. This component is required to discover DXE drivers stored in firmware volumes and execute them in the proper order. The proper order is determine by a combination of an a priori file that is optionally stored in the firmware volume and the dependency expressions that are part of the DXE drivers. The dependency expression tells the DXE Dispatcher the set of services that a particular DXE driver requires to be present for the DXE driver to execute. The DXE Dispatcher does not allow a DXE driver to execute until all of the DXE driver's dependencies have been satisfied. After all of the DXE drivers have been loaded and executed by the DXE Dispatcher, control is handed to the BDS Architectural Protocol that is responsible for implementing a boot policy that is compliant with the UEFI Boot Manager described in the UEFI 2.0 specification.

2.5 DXE Drivers

The DXE drivers are required to initialize the processor, chipset, and platform. They are also required to produce the DXE Architectural Protocols and any additional protocol services required to produce I/O abstractions for consoles and boot devices.

2.6 DXE Architectural Protocols

Table 5 provides a summary of the DXE Architectural Protocols. The DXE Foundation is abstracted from the platform through the DXE Architectural Protocols. The DXE Architectural Protocols manifest the platform-specific components of the DXE Foundation. DXE drivers that are loaded and executed by the DXE Dispatcher component of the DXE Foundation must produce these protocols.

Table 5. DXE Architectural Protocols

DXE Architectural Protocols	Description
Security Architectural	Allows the DXE Foundation to authenticate files stored in firmware volumes before they are used.
CPU Architectural	Provides services to manage caches, manage interrupts, retrieve the processor's frequency, and query any processor-based timers.
Metronome Architectural	Provides the services required to perform very short calibrated stalls.
Timer Architectural	Provides the services required to install and enable the heartbeat timer interrupt required by the timer services in the DXE Foundation.
BDS Architectural	Provides an entry point that the DXE Foundation calls once after all of the DXE drivers have been dispatched from all of the firmware volumes. This entry point is the transition from the DXE phase to the Boot Device Selection (BDS) phase, and it is responsible for establishing consoles and enabling the boot devices required to boot an OS.
Watchdog Timer Architectural	Provides the services required to enable and disable a watchdog timer in the platform.
Runtime Architectural	Provides the services required to convert all runtime services and runtime drivers from physical mappings to virtual mappings.
Variable Architectural	Provides the services to retrieve environment variables and set volatile environment variables.
Variable Write Architectural Protocol	Provides the services to set nonvolatile environment variables.
Monotonic Counter Architectural	Provides the services required by the DXE Foundation to manage a 64-bit monotonic counter.
Reset Architectural	Provides the services required to reset or shutdown the platform.
Real Time Clock Architectural	Provides the services to retrieve and set the current time and date as well as the time and date of an optional wake-up timer.
Capsule Architectural Protocol	Provides the services to retrieve and set the current time and date as well as the time and date of an optional wake-up timer.

2.7 Runtime Protocol

Table 6 provides a summary of the runtime protocol for status codes.

Table 6. Status Codes Runtime Protocol

Status Code Runtime Protocol:	Provides the services to send status codes from the DXE Foundation or DXE drivers to a log or device.

3
Boot Manager

3.1 Boot Manager

The Boot Manager in DXE executes after all the DXE drivers whose dependencies have been satisfied have been dispatched by the DXE Dispatcher. At that time, control is handed to the Boot Device Selection (BDS) phase of execution. The BDS phase is responsible for implementing the platform boot policy. System firmware that is compliant with this specification must implement the boot policy specified in the Boot Manager chapter of the UEFI 2.0 specification. This boot policy provides flexibility that allows system vendors to customize the user experience during this phase of execution.

The Boot Manager must also support booting from a short-form device path that starts with the first node being a firmware volume device path. The boot manager must use the GUID in the firmware volume device node to match it to a firmware volume in the system. The GUID in the firmware volume device path is compared with the firmware volume name GUID. If a match is made, then the firmware volume device path can be appended to the device path of the matching firmware volume and normal boot behavior can then be used.

The BDS phase is implemented as part of the BDS Architectural Protocol. The DXE Foundation will hand control to the BDS Architectural Protocol after all of the DXE drivers whose dependencies have been satisfied have been loaded and executed by the DXE Dispatcher. The BDS phase is responsible for the following:

- Initializing console devices
- Loading device drivers
- Attempting to load and execute boot selections

If the BDS phase cannot make forward progress, it will reinvoke the DXE Dispatcher to see if the dependencies of any additional DXE drivers have been satisfied since the last time the DXE Dispatcher was invoked.

4
UEFI System Table

4.1 DXE Services Table

DXE_SERVICES

Summary

Contains a table header and pointers to all of the DXE-specific services.

Related Definitions

```
#define DXE_SERVICES_SIGNATURE  0x565245535f455844
#define DXE_SERVICES_REVISION   ((1<<16) | (00)

typedef struct {
  EFI_TABLE_HEADER                Hdr;

  //
  // Global Coherency Domain Services
  //
  EFI_ADD_MEMORY_SPACE            AddMemorySpace;
  EFI_ALLOCATE_MEMORY_SPACE       AllocateMemorySpace;
  EFI_FREE_MEMORY_SPACE           FreeMemorySpace;
  EFI_REMOVE_MEMORY_SPACE         RemoveMemorySpace;
  EFI_GET_MEMORY_SPACE_DESCRIPTOR GetMemorySpaceDescriptor;
  EFI_SET_MEMORY_SPACE_ATTRIBUTES SetMemorySpaceAttributes;
  EFI_GET_MEMORY_SPACE_MAP        GetMemorySpaceMap;
  EFI_ADD_IO_SPACE                AddIoSpace;
  EFI_ALLOCATE_IO_SPACE           AllocateIoSpace;
  EFI_FREE_IO_SPACE               FreeIoSpace;
  EFI_REMOVE_IO_SPACE             RemoveIoSpace;
  EFI_GET_IO_SPACE_DESCRIPTOR     GetIoSpaceDescriptor;
  EFI_GET_IO_SPACE_MAP            GetIoSpaceMap;

  //
  // Dispatcher Services
  //
  EFI_DISPATCH                    Dispatch;
  EFI_SCHEDULE                    Schedule;
  EFI_TRUST                       Trust;

  //
```

```
  // Service to process a single firmware volume found in a
capsule
  //
  EFI_PROCESS_FIRMWARE_VOLUME       ProcessFirmwareVolume;
} DXE_SERVICES;
```

Parameters

Hdr

The table header for the DXE Services Table. This header contains the **DXE_SERVICES_SIGNATURE** and **DXE_SERVICES_REVISION** values along with the size of the **DXE_SERVICES_TABLE** structure and a 32-bit CRC to verify that the contents of the DXE Services Table are valid.

AddMemorySpace

Adds reserved memory, system memory, or memory-mapped I/O resources to the global coherency domain of the processor. See the **AddMemorySpace()** function description in this document.

AllocateMemorySpace

Allocates nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor. See the **AllocateMemorySpace()** function description in this document.

FreeMemorySpace

Frees nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor. See the **FreeMemorySpace()** function description in this document.

RemoveMemorySpace

Removes reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor. See the **RemoveMemorySpace()** function description in this document.

GetMemorySpaceDescriptor

Retrieves the descriptor for a memory region containing a specified address. See the **GetMemorySpaceDescriptor()** function description in this document.

SetMemorySpaceAttributes

Modifies the attributes for a memory region in the global coherency domain of the processor. See the **SetMemorySpaceAttributes()** function description in this document.

GetMemorySpaceMap

Returns a map of the memory resources in the global coherency domain of the processor. See the **GetMemorySpaceMap()** function description in this document.

AddIoSpace

Adds reserved I/O or I/O resources to the global coherency domain of the processor. See the **AddIoSpace()** function description in this document.

AllocateIoSpace

Allocates nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor. See the **AllocateIoSpace()** function description in this document.

FreeIoSpace

Frees nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor. See the **FreeIoSpace()** function description in this document.

RemoveIoSpace

Removes reserved I/O or I/O resources from the global coherency domain of the processor. See the **RemoveIoSpace()** function description in this document.

GetIoSpaceDescriptor

Retrieves the descriptor for an I/O region containing a specified address. See the **GetIoSpaceDescriptor()** function description in this document.

GetIoSpaceMap

Returns a map of the I/O resources in the global coherency domain of the processor. See the **GetIoSpaceMap()** function description in this document.

Dispatch

Loads and executed DXE drivers from firmware volumes. See the **Dispatch()** function description in this document.

Schedule

Clears the Schedule on Request (SOR) flag for a component that is stored in a firmware volume. See the **Schedule()** function description in this document.

Trust

Promotes a file stored in a firmware volume from the untrusted to the trusted state. See the **Trust()** function description in this document.

ProcessFirmwareVolume

Creates a firmware volume handle for a firmware volume that is present in system memory. See the **ProcessFirmwareVolume()** function description in this document.

Description

The UEFI DXE Services Table contains a table header and pointers to all of the DXE-specific services. Except for the table header, all elements in the DXE Services Tables are prototypes of function pointers to functions as defined in "Services - DXE Services" on page 33.

4.2 UEFI Image Entry Point Examples

4.2.1 UEFI Application Example

The following example shows the UEFI image entry point for an UEFI application. This application makes use of the UEFI System Table, UEFI Boot Services Table, UEFI Runtime Services Table, and DXE Services Table.

```
EFI_GUID  gEfiDxeServicesTableGuid = DXE_SERVICES_TABLE_GUID;

EFI_SYSTEM_TABLE            *gST;
EFI_BOOT_SERVICES           *gBS;
EFI_RUNTIME_SERVICES        *gRT;
DXE_SERVICES                *gDS;

EfiApplicationEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  UINTN                             Index;
  BOOLEAN                           Result;
  EFI_STATUS                        Status;
  EFI_TIME                          *Time;
  UINTN                             NumberOfDescriptors;
  EFI_GCD_MEMORY_SPACE_DESCRIPTOR   MemorySpaceDescriptor;

  gST = SystemTable;
  gBS = gST->BootServices;
  gRT = gST->RuntimeServices;

  gDS = NULL;
  for (Index = 0; Index < gST->NumberOfTableEntries; Index++) {
    Result = EfiCompareGuid (
               &gEfiDxeServicesTableGuid,
               &(gST->ConfigurationTable[Index].VendorGuid)
               );
    if (Result) {
      gDS = gST->ConfigurationTable[Index].VendorTable;
    }
  }
  if (gDS == NULL) {
    return EFI_NOT_FOUND;
  }

  //
  // Use UEFI System Table to print "Hello World" to the active console
  // output device.
  //
```

```
Status = gST->ConOut->OutputString (gST->ConOut, L”Hello World\n\r”);
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Use UEFI Boot Services Table to allocate a buffer to store the
// current time and date.
//
Status = gBS->AllocatePool (
                EfiBootServicesData,
                sizeof (EFI_TIME),
                (VOID **)&Time
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Use the UEFI Runtime Services Table to get the current
// time and date.
//
Status = gRT->GetTime (&Time, NULL)
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Use UEFI Boot Services to free the buffer that was used to store
// the current time and date.
//
Status = gBS->FreePool (Time);
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Use the DXE Services Table to get the current GCD Memory Space Map
//
Status = gDS->GetMemorySpaceMap (
                &NumberOfDescriptors,
                &MemorySpaceMap
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Use UEFI Boot Services to free the buffer that was used to store
// the GCD Memory Space Map.
//
```

```
  Status = gBS->FreePool (MemorySpaceMap);
  if (EFI_ERROR (Status)) {
    return Status;
  }

  return Status;
}
```

4.2.2 Non-UEFI Driver Model Example (Resident in Memory)

The following example shows the UEFI image entry point for an UEFI driver that does not follow the *UEFI Driver Model*. Because this driver returns **EFI_SUCCESS**, it will stay resident in memory after it exits.

```
EFI_GUID  gEfiDxeServicesTableGuid = DXE_SERVICES_TABLE_GUID;

EFI_SYSTEM_TABLE            *gST;
EFI_BOOT_SERVICES           *gBS;
EFI_RUNTIME_SERVICES        *gRT;
DXE_SERVICES                *gDS;

EfiDriverEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  UINTN    Index;
  BOOLEAN  Result;

  gST = SystemTable;
  gBS = gST->BootServices;
  gRT = gST->RuntimeServices;

  gDS = NULL;
  for (Index = 0; Index < gST->NumberOfTableEntries; Index++) {
    Result = EfiCompareGuid (
               &gEfiDxeServicesTableGuid,
               &(gST->ConfigurationTable[Index].VendorGuid)
               );
    if (Result) {
      gDS = gST->ConfigurationTable[Index].VendorTable;
    }
  }
  if (gDS == NULL) {
    return EFI_REQUEST_UNLOAD_IMAGE;
  }

  //
```

```
  // Implement driver initialization here.
  //

  return EFI_SUCCESS;
}
```

4.2.3 Non-UEFI Driver Model Example (Nonresident in Memory)

The following example shows the UEFI image entry point for an UEFI driver that also does not follow the *UEFI Driver Model*. Because this driver returns the error code **EFI_REQUEST_UNLOAD_IMAGE**, it will not stay resident in memory after it exits.

```
EFI_GUID  gEfiDxeServicesTableGuid = DXE_SERVICES_TABLE_GUID;

EFI_SYSTEM_TABLE            *gST;
EFI_BOOT_SERVICES           *gBS;
EFI_RUNTIME_SERVICES        *gRT;
DXE_SERVICES                *gDS;

EfiDriverEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  UINTN    Index;
  BOOLEAN  Result;

  gST = SystemTable;
  gBS = gST->BootServices;
  gRT = gST->RuntimeServices;

  gDS = NULL;
  for (Index = 0; Index < gST->NumberOfTableEntries; Index++) {
    Result = EfiCompareGuid (
               &gEfiDxeServicesTableGuid,
               &(gST->ConfigurationTable[Index].VendorGuid)
               );
    if (Result) {
      gDS = gST->ConfigurationTable[Index].VendorTable;
    }
  }
  if (gDS == NULL) {
    return EFI_REQUEST_UNLOAD_IMAGE;
  }

  //
  // Implement driver initialization here.
  //
```

```
  return EFI_REQUEST_UNLOAD_IMAGE;
}
```

4.2.4 UEFI Driver Model Example

The following is an *UEFI Driver Model* example that shows the driver initialization routine for the ABC device controller that is on the XYZ bus. The **EFI_DRIVER_BINDING_PROTOCOL** is defined in Chapter 9 of the UEFI 2.0 specification. The function prototypes for the **AbcSupported()**, **AbcStart()**, and **AbcStop()** functions are defined in Section 9.1 of the UEFI 2.0 specification. This function saves the driver's image handle and a pointer to the UEFI Boot Services Table in global variables, so that the other functions in the same driver can have access to these values. It then creates an instance of the **EFI_DRIVER_BINDING_PROTOCOL** and installs it onto the driver's image handle.

```
extern EFI_GUID                     gEfiDriverBindingProtocolGuid;
EFI_BOOT_SERVICES                   *gBS;
static EFI_DRIVER_BINDING_PROTOCOL  mAbcDriverBinding = {
  AbcSupported,
  AbcStart,
  AbcStop,
  0x10,
  NULL,
  NULL
};

AbcEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  EFI_STATUS  Status;

  gBS = SystemTable->BootServices;

  mAbcDriverBinding->ImageHandle         = ImageHandle;
  mAbcDriverBinding->DriverBindingHandle = ImageHandle;

  Status = gBS->InstallMultipleProtocolInterfaces(
                 &mAbcDriverBinding->DriverBindingHandle,
                 &gEfiDriverBindingProtocolGuid, &mAbcDriverBinding,
                 NULL
                 );
  return Status;
}
```

4.2.5 UEFI Driver Model Example (Unloadable)

The following is the same *UEFI Driver Model* example as in the UEFI Driver Model Example, except that it also includes the code required to allow the driver to be unloaded through the boot

service **Unload()**. Any protocols installed or memory allocated in **AbcEntryPoint()** must be uninstalled or freed in the **AbcUnload()**. The **AbcUnload()** function first checks to see how many controllers this driver is currently managing. If the number of controllers is greater than zero, then this driver cannot be unloaded at this time, so an error is returned.

```
extern EFI_GUID                     gEfiLoadedImageProtocolGuid;
extern EFI_GUID                     gEfiDriverBindingProtocolGuid;
EFI_BOOT_SERVICES                   *gBS;
static EFI_DRIVER_BINDING_PROTOCOL  mAbcDriverBinding = {
  AbcSupported,
  AbcStart,
  AbcStop,
  1,
  NULL,
  NULL
};

EFI_STATUS
AbcUnload (
  IN EFI_HANDLE  ImageHandle
  );

AbcEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  EFI_STATUS                 Status;
  EFI_LOADED_IMAGE_PROTOCOL  *LoadedImage;

  gBS = SystemTable->BootServices;

  Status = gBS->OpenProtocol (
                ImageHandle,
                &gEfiLoadedImageProtocolGuid,
                &LoadedImage,
                ImageHandle,
                NULL,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  LoadedImage->Unload = AbcUnload;

  mAbcDriverBinding->ImageHandle         = ImageHandle;
  mAbcDriverBinding->DriverBindingHandle = ImageHandle;

  Status = gBS->InstallMultipleProtocolInterfaces(
```

```
                  &mAbcDriverBinding->DriverBindingHandle,
                  &gEfiDriverBindingProtocolGuid, &mAbcDriverBinding,
                  NULL
                  );

  return Status;
}

EFI_STATUS
AbcUnload (
  IN EFI_HANDLE  ImageHandle
  )

{
  EFI_STATUS  Status;
  UINTN       Count;

  Status = LibGetManagedControllerHandles (ImageHandle, &Count, NULL);
  if (EFI_ERROR (Status)) {
    return Status;
  }

  if (Count > 0) {
    return EFI_ACCESS_DENIED;
  }

  Status = gBS->UninstallMultipleProtocolInterfaces (
                  ImageHandle,
                  &gEfiDriverBindingProtocolGuid, &mAbcDriverBinding,
                  NULL
                  );
  return Status;
}
```

4.2.6 UEFI Driver Model Example (Multiple Instances)

The following is the same as the first UEFI Driver Model example, except that it produces three **EFI_DRIVER_BINDING_PROTOCOL** instances. The first one is installed onto the driver's image handle. The other two are installed onto newly created handles.

```
extern EFI_GUID                     gEfiDriverBindingProtocolGuid;
EFI_BOOT_SERVICES                   *gBS;

static EFI_DRIVER_BINDING_PROTOCOL  mAbcDriverBindingA = {
  AbcSupportedA,
  AbcStartA,
  AbcStopA,
  1,
```

```
  NULL,
  NULL
};

static EFI_DRIVER_BINDING_PROTOCOL  mAbcDriverBindingB = {
  AbcSupportedB,
  AbcStartB,
  AbcStopB,
  1,
  NULL,
  NULL
};

static EFI_DRIVER_BINDING_PROTOCOL  mAbcDriverBindingC = {
  AbcSupportedC,
  AbcStartC,
  AbcStopC,
  1,
  NULL,
  NULL
};

AbcEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  EFI_STATUS  Status;

  gBS = SystemTable->BootServices;

  //
  // Install mAbcDriverBindingA onto ImageHandle
  //
  mAbcDriverBindingA->ImageHandle         = ImageHandle;
  mAbcDriverBindingA->DriverBindingHandle = ImageHandle;



  Status = gBS->InstallMultipleProtocolInterfaces(
                 &mAbcDriverBindingA->DriverBindingHandle,
                 &gEfiDriverBindingProtocolGuid, &mAbcDriverBindingA,
                 NULL
                 );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Install mAbcDriverBindingB onto a newly created handle
```

```
  //
  mAbcDriverBindingB->ImageHandle         = ImageHandle;
  mAbcDriverBindingB->DriverBindingHandle = NULL;

  Status = gBS->InstallMultipleProtocolInterfaces(
                 &mAbcDriverBindingB->DriverBindingHandle,
                 &gEfiDriverBindingProtocolGuid, &mAbcDriverBindingB,
                 NULL
                 );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Install mAbcDriverBindingC onto a newly created handle
  //
  mAbcDriverBindingC->ImageHandle         = ImageHandle;
  mAbcDriverBindingC->DriverBindingHandle = NULL;

  Status = gBS->InstallMultipleProtocolInterfaces(
                 &mAbcDriverBindingC->DriverBindingHandle,
                 &gEfiDriverBindingProtocolGuid, &mAbcDriverBindingC,
                 NULL
                 );

  return Status;
}
```

5
Services - Boot Services

5.1 Extensions to UEFI Boot Service Event Usage

5.1.1 CreateEvent

CreateEventEx() in UEFI 2.0 allows for registration of events named by GUID's. The DXE foundation defines the following:

```
#define EFI_EVENT_LEGACY_BOOT_GUID
{0x2a571201, 0x4966, 0x47f6, 0x8b, 0x86, 0xf3, 0x1e, 0x41, 0xf3,
0x2f, 0x10}
```

This event is to be used with **CreateEventEx()** in order to be notified when the UEFI boot manager is about to boot a legacy boot option. Notification of events of this type is sent just before Int19h is invoked.

5.1.2 Pre-Defined Event Groups

This section describes the pre-defined event groups used by the PI specification.

EFI_EVENT_GROUP_DXE_DISPATCH_GUID

This event group is notified by the system when the DXE dispatcher finished one round of driver dispatch. This allows the SMM dispatcher get chance to dispatch SMM driver which will depend on UEFI protocols.

Related Definitions

```
#define EFI_EVENT_GROUP_DXE_DISPATCH_GUID \
  { 0x7081e22f, 0xcac6, 0x4053, 0x94, 0x68, 0x67, 0x57, \
  0x82, 0xcf, 0x88, 0xe5 }
```

5.1.3 Additions to LoadImage()

Summary

Loads an UEFI image into memory. This function has been extended from the **LoadImage()** Boot Service defined in the UEFI 2.0 specification. The DXE foundation extends this to support an additional image type, allowing UEFI images to be loaded from files stored in firmware volumes. It also validates the image using the services of the Security Architectural Protocol.

Prototype

```
EFI_STATUS
LoadImage (
  IN BOOLEAN           BootPolicy,
  IN EFI_HANDLE        ParentImageHandle,
```

```
IN EFI_DEVICE_PATH   *FilePath,
IN VOID              *SourceBuffer  OPTIONAL ,
IN UINTN             SourceSize,
OUT EFI_HANDLE       *ImageHandle
);
```

Parameters

BootPolicy

If **TRUE**, indicates that the request originates from the boot manager, and that the boot manager is attempting to load *FilePath* as a boot selection. Ignored if *SourceBuffer* is not **NULL**.

ParentImageHandle

The caller's image handle. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function description in the UEFI 2.0 specification. This field is used to initialize the *ParentHandle* field of the **LOADED_IMAGE** protocol for the image that is being loaded.

FilePath

The specific file path from which the image is loaded. Type **EFI_DEVICE_PATH** is defined in the **LocateDevicePath()** function description in the UEFI 2.0 specification.

SourceBuffer

If not **NULL**, a pointer to the memory location containing a copy of the image to be loaded.

SourceSize

The size in bytes of *SourceBuffer*. Ignored if *SourceBuffer* is **NULL**.

ImageHandle

Pointer to the returned image handle that is created when the image is successfully loaded. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function description in the UEFI 2.0 specification.

Description

The **LoadImage()** function loads an UEFI image into memory and returns a handle to the image. The supported subsystem values in the PE image header are listed in "Related Definitions" below. The image is loaded in one of two ways. If *SourceBuffer* is not **NULL**, the function is a memory-to-memory load in which *SourceBuffer* points to the image to be loaded and *SourceSize* indicates the image's size in bytes. *FilePath* specifies where the image specified by *SourceBuffer* and *SourceSize* was loaded. In this case, the caller has copied the image into *SourceBuffer* and can free the buffer once loading is complete.

If *SourceBuffer* is **NULL**, the function is a file copy operation that uses the **EFI_FIRMWARE_VOLUME2_PROTOCOL**, followed by the **SIMPLE_FILE_SYSTEM_PROTOCOL** and then the **LOAD_FILE_PROTOCOL** to access the file referred to by *FilePath*. In this case, the *BootPolicy* flag is passed to the **LOAD_FILE.LoadFile()** function and is used to load the default image responsible for booting

when the *FilePath* only indicates the device. For more information see the discussion of the Load File Protocol in Chapter 11 of the UEFI 2.0 specification.

Regardless of the type of load (memory-to-memory or file copy), the function relocates the code in the image while loading it.

The image is also validated using the **FileAuthenticationState()** service of the Security Architectural Protocol (SAP). If the SAP returns the status **EFI_SUCCESS**, then the load operation is completed normally. If the SAP returns the status **EFI_SECURITY_VIOLATION**, then the load operation is completed normally, and the **EFI_SECURITY_VIOLATION** status is returned. In this case, the caller is not allowed to start the image until some platform specific policy is executed to protect the system while executing untrusted code. If the SAP returns the status **EFI_ACCESS_DENIED**, then the image should never be trusted. In this case, the image is unloaded from memory, and **EFI_ACCESS_DENIED** is returned.

Once the image is loaded, firmware creates and returns an **EFI_HANDLE** that identifies the image and supports the **LOADED_IMAGE_PROTOCOL**. The caller may fill in the image's "load options" data, or add additional protocol support to the handle before passing control to the newly loaded image by calling **StartImage()**. Also, once the image is loaded, the caller either starts it by calling **StartImage()** or unloads it by calling **UnloadImage()**.

Related Definitions

```
//*******************************************************
// Supported subsystem values
//*******************************************************

#define EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION          10
#define EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER  11
#define EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER       12
#define EFI_IMAGE_SUBSYSTEM_SAL_RUNTIME_DRIVER       13
```

Table 7 describes the fields in the above definition.

Table 7. Supported Subsystem Values

Supported Subsystem Values	Description
EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION	The image is loaded into memory of type **EfiLoaderCode**, and the memory is freed when the application exits.
EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER	The image is loaded into memory of type **EfiBootServicesCode**. If the image exits with an error code, then the memory for the image is free. If the image exits with **EFI_SUCCESS**, then the memory for the image is not freed.
EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER	The image is loaded into memory of type **EfiRuntimeServicesCode**. If the image exits with an error code, then the memory for the image is free. If the image exits with **EFI_SUCCESS**, then the memory for the image is not freed. Images of this type are automatically converted from physical addresses to virtual address when the Runtime Service **SetVirtualAddressMap()** is called.

EFI_IMAGE_SUBSYSTEM_SAL_RUNTIME_DRIVER	The image is loaded into memory of type **EfiRuntimeServicesCode**. If the image exits with an error code, then the memory for the image is free. If the image exits with **EFI_SUCCESS**, then the memory for the image is not freed. Images of this type are not converted from physical to virtual addresses when the Runtime Service **SetVirtualAddressMap()** is called.

Status Codes Returned

EFI_SUCCESS	The image was loaded into memory.
EFI_SECURITY_VIOLATION	The image was loaded into memory, but the current security policy dictates that the image should not be executed at this time.
EFI_ACCESS_DENIED	The image was not loaded into memory because the current security policy dictates that the image should never be executed.
EFI_NOT_FOUND	The *FilePath* was not found.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.
EFI_UNSUPPORTED	The image type is not supported, or the device path cannot be parsed to locate the proper protocol for loading the file.
EFI_OUT_OF_RESOURCES	Image was not loaded due to insufficient resources.
EFI_LOAD_ERROR	Image was not loaded because the image format was corrupt or not understood.
EFI_DEVICE_ERROR	Image was not loaded because the device returned a read error.

6
Runtime Capabilities

6.1 Additional Runtime Protocol

6.1.1 Status Code Services

Table 8 lists the runtime protocol that are used to report status codes. This protocol provides a runtime protocol that can be bound by other runtime drivers for reporting status information.

Table 8. Status Code Runtime Protocol

Name	Type	Description
ReportStatusCode	Runtime	Reports status codes at boot services time and runtime.

7
Services - DXE Services

7.1 Introduction

This chapter describes the services in the DXE Services Table. These services include the following:

- Global Coherency Domain (GCD) Services
- Dispatcher Services

The GCD Services are used to manage the system memory, memory-mapped I/O, and I/O resources present in a platform. The Dispatcher Services are used to invoke the DXE Dispatcher and modify the state of a DXE driver that is being tracked by the DXE Dispatcher.

7.2 Global Coherency Domain Services

7.2.1 Global Coherency Domain (GCD) Services Overview

The Global Coherency Domain (GCD) Services are used to manage the memory and I/O resources visible to the boot processor. These resources are managed in two different maps:

- GCD memory space map
- GCD I/O space map

If memory or I/O resources are added, removed, allocated, or freed, then the GCD memory space map and GCD I/O space map are updated. GCD Services are also provided to retrieve the contents of these two resource maps.

The GCD Services can be broken up into two groups. The first manages the memory resources visible to the boot processor, and the second manages the I/O resources visible to the boot processor. Not all processor types support I/O resources, so the management of I/O resources may not be required. However, since system memory resources and memory-mapped I/O resources are required to execute the DXE environment, the management of memory resources is always required.

7.2.2 GCD Memory Resources

The Global Coherency Domain (GCD) Services used to manage memory resources include the following:

- **`AddMemorySpace()`**
- **`AllocateMemorySpace()`**
- **`FreeMemorySpace()`**
- **`RemoveMemorySpace()`**
- **`SetMemorySpaceAttributes()`**

The GCD Services used to retrieve the GCD memory space map include the following:

- **GetMemorySpaceDescriptor()**
- **GetMemorySpaceMap()**

The GCD memory space map is initialized from the HOB list that is passed to the entry point of the DXE Foundation. One HOB type describes the number of address lines that are used to access memory resources. This information is used to initialize the state of the GCD memory space map. Any memory regions outside this initial region are not available to any of the GCD Services that are used to manage memory resources. The GCD memory space map is designed to describe the memory address space with as many as 64 address lines. Each region in the GCD memory space map can begin and end on a byte boundary. There are additional HOB types that describe the location of system memory, the location memory mapped I/O, the location of firmware devices, the location of firmware volumes, the location of reserved regions, and the location of system memory regions that were allocated prior to the execution of the DXE Foundation. The DXE Foundation must parse the contents of the HOB list to guarantee that memory regions reserved prior to the execution of the DXE Foundation are honored. As a result, the GCD memory space map must reflect the memory regions described in the HOB list. The GCD memory space map provides the DXE Foundation with the information required to initialize the memory services such as **AllocatePages()**, **FreePages()**, **AllocatePool()**, **FreePool()**, and **GetMemoryMap()**. See the UEFI 2.0 specification for definitions of these services.

A memory region described by the GCD memory space map can be in one of several different states:

- Nonexistent memory
- System memory
- Memory-mapped I/O
- Reserved memory

These memory regions can be allocated and freed by DXE drivers executing in the DXE environment. In addition, a DXE driver can attempt to adjust the caching attributes of a memory region. Figure 2 shows the possible state transitions for each byte of memory in the GCD memory space map. The transitions are labeled with the GCD Service that can move the byte from one state to another. The GCD services are required to merge similar memory regions that are adjacent to each other into a single memory descriptor, which reduces the number of entries in the GCD memory space map.

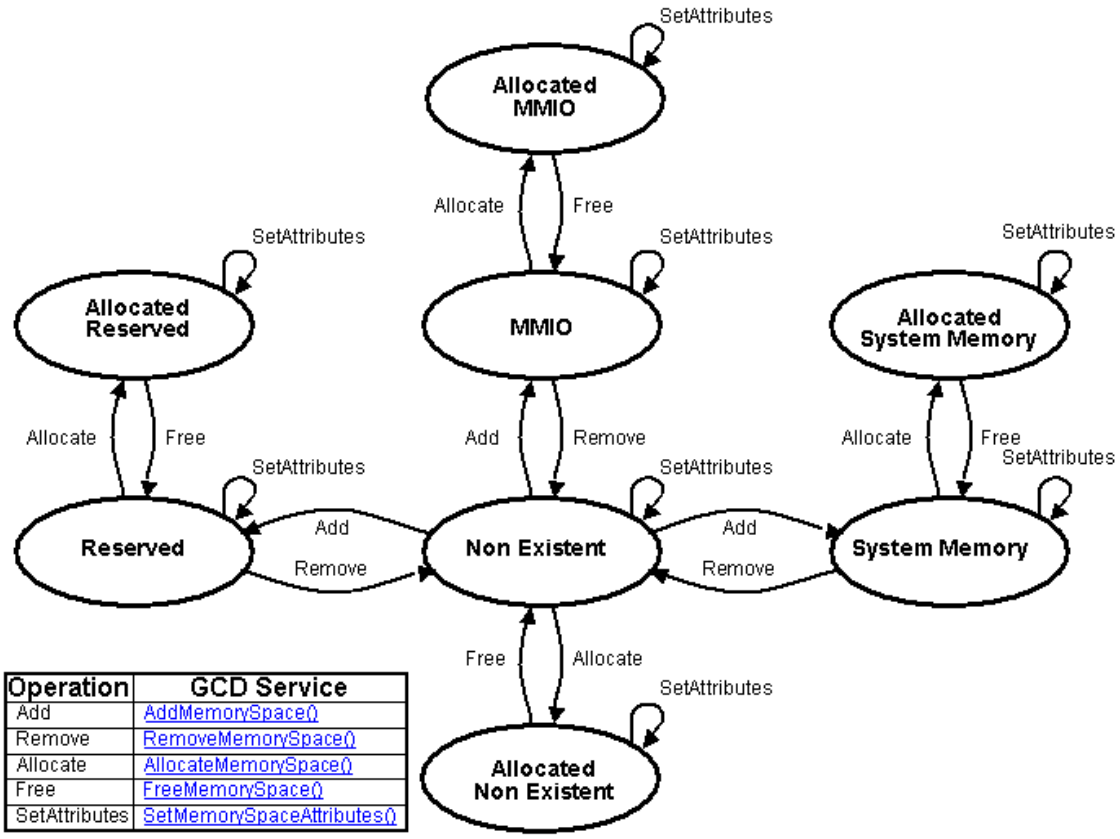

Figure 2. GCD Memory State Transitions

7.2.3 GCD I/O Resources

The Global Coherency Domain (GCD) Services used to manage I/O resources include the following:

- **`AddIoSpace()`**
- **`AllocateIoSpace()`**
- **`FreeIoSpace()`**
- **`RemoveIoSpace()`**

The GCD Services used to retrieve the GCD I/O space map include the following:

- **`GetIoSpaceDescriptor()`**
- **`GetIoSpaceMap()`**

The GCD I/O space map is initialized from the HOB list that is passed to the entry point of the DXE Foundation. One HOB type describes the number of address lines that are used to access I/O resources. This information is used to initialize the state of the GCD I/O space map. Any I/O regions outside this initial region are not available to any of the GCD Services that are used to manage I/O resources. The GCD I/O space map is designed to describe the I/O address space with

as many as 64 address lines. Each region in the GCD I/O space map can being and end on a byte boundary.

An I/O region described by the GCD I/O space map can be in several different states. These include nonexistent I/O, I/O, and reserved I/O. These I/O regions can be allocated and freed by DXE drivers executing in the DXE environment. Figure 3 shows the possible state transitions for each byte of I/O in the GCD I/O space map. The transitions are labeled with the GCD Service that can move the byte from one state to another. The GCD Services are required to merge similar I/O regions that are adjacent to each other into a single I/O descriptor, which reduces the number of entries in the GCD I/O space map.

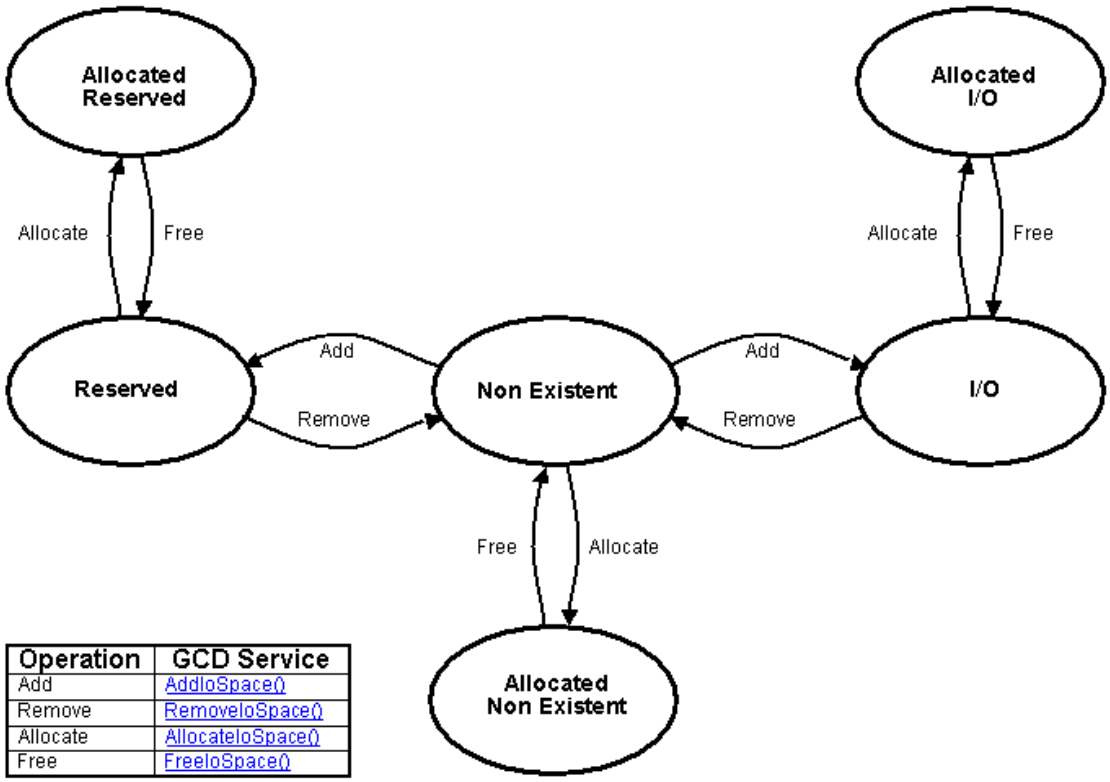

Figure 3. GCD I/O State Transitions

7.2.4 Global Coherency Domain Services

The functions that make up Global Coherency Domain (GCD) Services are used during preboot to add, remove, allocate, free, and provide maps of the system memory, memory-mapped I/O, and I/O resources in a platform. These services, used in conjunction with the Memory Allocation Services, provide the ability to manage all the memory and I/O resources in a platform. Table 9 lists the Global Coherency Domain Services.

Table 9. Global Coherency Domain Boot Type Services

Name	Description
AddMemorySpace	This service adds reserved memory, system memory, or memory-mapped I/O resources to the global coherency domain of the processor.
AllocateMemorySpace	This service allocates nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.
FreeMemorySpace	This service frees nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.
RemoveMemorySpace	This service removes reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.
GetMemorySpaceDescriptor	This service retrieves the descriptor for a memory region containing a specified address.
SetMemorySpaceAttributes	This service modifies the attributes for a memory region in the global coherency domain of the processor.
GetMemorySpaceMap	Returns a map of the memory resources in the global coherency domain of the processor.
AddIoSpace	This service adds reserved I/O, or I/O resources to the global coherency domain of the processor.
AllocateIoSpace	This service allocates nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor.
FreeIoSpace	This service frees nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor.
RemoveIoSpace	This service removes reserved I/O, or I/O resources from the global coherency domain of the processor.
GetIoSpaceDescriptor	This service retrieves the descriptor for an I/O region containing a specified address.
GetIoSpaceMap	Returns a map of the I/O resources in the global coherency domain of the processor.

AddMemorySpace()

Summary

This service adds reserved memory, system memory, or memory-mapped I/O resources to the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ADD_MEMORY_SPACE) (
  IN EFI_GCD_MEMORY_TYPE     GcdMemoryType,
  IN EFI_PHYSICAL_ADDRESS    BaseAddress,
  IN UINT64                  Length,
  IN UINT64                  Capabilities
  );
```

Parameters

GcdMemoryType

The type of memory resource being added. Type **EFI_GCD_MEMORY_TYPE** is defined in "Related Definitions" below. The only types allowed are **EfiGcdMemoryTypeReserved**, **EfiGcdMemoryTypeSystemMemory**, and **EfiGcdMemoryTypeMemoryMappedIo**.

BaseAddress

The physical address that is the start address of the memory resource being added. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The size, in bytes, of the memory resource that is being added.

Capabilities

The bit mask of attributes that the memory resource region supports. The bit mask of available attributes is defined in the **GetMemoryMap()** function description in the UEFI 2.0 specification.

Description

The **AddMemorySpace()** function converts unallocated non-existent memory ranges to a range of reserved memory, a range of system memory, or a range of memory mapped I/O. *BaseAddress* and *Length* specify the memory range, and *GcdMemoryType* specifies the memory type. The bit mask of all supported attributes for the memory range being added is specified by *Capabilities*. If the memory range is successfully added, then **EFI_SUCCESS** is returned.

If the memory range specified by *BaseAddress* and *Length* is of type **EfiGcdMemoryTypeSystemMemory**, then the memory range may be automatically allocated for use by the UEFI memory services. If the addition of the memory range specified by

BaseAddress and *Length* results in a GCD memory space map containing one or more 4 KB regions of unallocated **EfiGcdMemoryTypeSystemMemory** aligned on 4 KB boundaries, then those regions will always be converted to ranges of allocated **EfiGcdMemoryTypeSystemMemory**. This extra conversion will never be performed for fragments of memory that do not meet the above criteria.

If the GCD memory space map contains adjacent memory regions that only differ in their base address and length fields, then those adjacent memory regions must be merged into a single memory descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If *GcdMemoryType* is not **EfiGcdMemoryTypeReserved**, **EfiGcdMemoryTypeSystemMemory**, or **EfiGcdMemoryTypeMemoryMappedIo**, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the memory range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If any portion of the memory range specified by *BaseAddress* and *Length* is not of type **EfiGcdMemoryTypeNonExistent**, then **EFI_ACCESS_DENIED** is returned.

If any portion of the memory range specified by *BaseAddress* and *Length* was allocated in a prior call to **AllocateMemorySpace()**, then **EFI_ACCESS_DENIED** is returned.

If there are not enough system resources available to add the memory resource to the global coherency domain of the processor, then **EFI_OUT_OF_RESOURCES** is returned.

Related Definitions

```
//*******************************************************
// EFI_GCD_MEMORY_TYPE
//*******************************************************
typedef enum {
  EfiGcdMemoryTypeNonExistent,
  EfiGcdMemoryTypeReserved,
  EfiGcdMemoryTypeSystemMemory,
  EfiGcdMemoryTypeMemoryMappedIo,
  EfiGcdMemoryTypeMaximum
} EFI_GCD_MEMORY_TYPE;
```

EfiGcdMemoryTypeNonExistent

A memory region that is visible to the boot processor. However, there are no system components that are currently decoding this memory region.

EfiGcdMemoryTypeReserved

A memory region that is visible to the boot processor. This memory region is being decoded by a system component, but the memory region is not considered to be either system memory or memory-mapped I/O.

EfiGcdMemoryTypeSystemMemory

A memory region that is visible to the boot processor. A memory controller is currently decoding this memory region and the memory controller is producing a tested system memory region that is available to the memory services.

EfiGcdMemoryTypeMemoryMappedIo

A memory region that is visible to the boot processor. This memory region is currently being decoded by a component as memory-mapped I/O that can be used to access I/O devices in the platform.

Status Codes Returned

EFI_SUCCESS	The memory resource was added to the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*GcdMemoryType* is invalid.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_OUT_OF_RESOURCES	There are not enough system resources to add the memory resource to the global coherency domain of the processor.
EFI_UNSUPPORTED	The processor does not support one or more bytes of the memory resource range specified by *BaseAddress* and *Length*.
EFI_ACCESS_DENIED	One or more bytes of the memory resource range specified by *BaseAddress* and *Length* conflicts with a memory resource range that was previously added to the global coherency domain of the processor.
EFI_ACCESS_DENIED	One or more bytes of the memory resource range specified by *BaseAddress* and *Length* was allocated in a prior call to **AllocateMemorySpace()**.

AllocateMemorySpace()

Summary

This service allocates nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ALLOCATE_MEMORY_SPACE) (
  IN      EFI_GCD_ALLOCATE_TYPE    GcdAllocateType,
  IN      EFI_GCD_MEMORY_TYPE      GcdMemoryType,
  IN      UINTN                    Alignment,
  IN      UINT64                   Length,
  IN OUT  EFI_PHYSICAL_ADDRESS     *BaseAddress,
  IN      EFI_HANDLE               ImageHandle,
  IN      EFI_HANDLE               DeviceHandle   OPTIONAL
  );
```

Parameters

GcdAllocateType

The type of allocation to perform. Type **EFI_GCD_ALLOCATE_TYPE** is defined in "Related Definitions" below.

GcdMemoryType

The type of memory resource being allocated. Type **EFI_GCD_MEMORY_TYPE** is defined in **AddMemorySpace()**. The only types allowed are **EfiGcdMemoryTypeNonExistent**, **EfiGcdMemoryTypeReserved**, **EfiGcdMemoryTypeSystemMemory**, and **EfiGcdMemoryTypeMemoryMappedIo**.

Alignment

The log base 2 of the boundary that *BaseAddress* must be aligned on output. For example, a value of 0 means that *BaseAddress* can be aligned on any byte boundary, and a value of 12 means that *BaseAddress* must be aligned on a 4 KB boundary.

Length

The size in bytes of the memory resource range that is being allocated.

BaseAddress

A pointer to a physical address. On input, the way in which the address is used depends on the value of *Type*. See "Description" below for more information. On output the address is set to the base of the memory resource range that was allocated. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

ImageHandle

The image handle of the agent that is allocating the memory resource. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

DeviceHandle

The device handle for which the memory resource is being allocated. If the memory resource is not being allocated for a device that has an associated device handle, then this parameter is optional and may be **NULL**. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Description

The **AllocateMemorySpace()** function searches for a memory range of type *GcdMemoryType* and converts the discovered memory range from the unallocated state to the allocated state. The parameters *GcdAllocateType*, *Alignment*, *Length*, and *BaseAddress* specify the manner in which the GCD memory space map is searched. If a memory range is found that meets the search criteria, then the base address of the memory range is returned in *BaseAddress*, and **EFI_SUCCESS** is returned. *ImageHandle* and *DeviceHandle* are used to convert the memory range from the unallocated state to the allocated state. *ImageHandle* identifies the image that is calling **AllocateMemorySpace()**, and *DeviceHandle* identifies the device that *ImageHandle* is managing that requires the memory range. *DeviceHandle* is optional, because the device that *ImageHandle* is managing might not have an associated device handle. If a memory range meeting the search criteria cannot be found, then **EFI_NOT_FOUND** is returned.

If *GcdAllocateType* is **EfiGcdAllocateAnySearchBottomUp**, then the GCD memory space map is searched from the lowest address up to the highest address looking for unallocated memory ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdMemoryType*.

If *GcdAllocateType* is **EfiGcdAllocateAnySearchTopDown**, then the GCD memory space map is searched from the highest address down to the lowest address looking for unallocated memory ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdMemoryType*.

If *GcdAllocateType* is **EfiGcdAllocateMaxAddressSearchBottomUp**, then the GCD memory space map is searched from the lowest address up to *BaseAddress* looking for unallocated memory ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdMemoryType*.

If *GcdAllocateType* is **EfiGcdAllocateMaxAddressSearchTopDown**, then the GCD memory space map is searched from *BaseAddress* down to the lowest address looking for unallocated memory ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdMemoryType*.

If *GcdAllocateType* is **EfiGcdAllocateAddress**, then the GCD memory space map is checked to see if the memory range starting at *BaseAddress* for *Length* bytes is of type *GcdMemoryType*, unallocated, and begins on a the boundary specified by *Alignment*.

If the GCD memory space map contains adjacent memory regions that only differ in their base address and length fields, then those adjacent memory regions must be merged into a single memory descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If *BaseAddress* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *ImageHandle* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *GcdMemoryType* is not **EfiGcdMemoryTypeNonExistent**, **EfiGcdMemoryTypeReserved**, **EfiGcdMemoryTypeSystem Memory**, or **EfiGcdMemoryTypeMemoryMappedIo**, then **EFI_INVALID_PARAMETER** is returned.

If *GcdAlocateType* is less than zero, or *GcdAllocateType* is greater than or equal to *EfiGcdMaxAllocateType* then **EFI_INVALID_PARAMETER** is returned.

If there are not enough system resources available to allocate the memory range, then **EFI_OUT_OF_RESOURCES** is returned.

Related Definitions

```
//*******************************************************
// EFI_GCD_ALLOCATE_TYPE
//*******************************************************
typedef enum {
  EfiGcdAllocateAnySearchBottomUp,
  EfiGcdAllocateMaxAddressSearchBottomUp,
  EfiGcdAllocateAddress,
  EfiGcdAllocateAnySearchTopDown,
  EfiGcdAllocateMaxAddressSearchTopDown,
  EfiGcdMaxAllocateType
} EFI_GCD_ALLOCATE_TYPE;
```

Status Codes Returned

EFI_SUCCESS	The memory resource was allocated from the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*GcdAllocateType* is invalid.
EFI_INVALID_PARAMETER	*GcdMemoryType* is invalid.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_INVALID_PARAMETER	*BaseAddress* is **NULL**.
EFI_INVALID_PARAMETER	*ImageHandle* is **NULL**.
EFI_OUT_OF_RESOURCES	There are not enough system resources to allocate the memory resource from the global coherency domain of the processor.
EFI_NOT_FOUND	The memory resource request could not be satisfied.

FreeMemorySpace()

Summary

This service frees nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_FREE_MEMORY_SPACE) (
  IN EFI_PHYSICAL_ADDRESS    BaseAddress,
  IN UINT64                  Length
  );
```

Parameters

BaseAddress

The physical address that is the start address of the memory resource being freed. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The size in bytes of the memory resource range that is being freed.

Description

The **FreeMemorySpace()** function converts the memory range specified by *BaseAddress* and *Length* from the allocated state to the unallocated state. If this conversion is successful, then **EFI_SUCCESS** is returned.

If the GCD memory space map contains adjacent memory regions that only differ in their base address and length fields, then those adjacent memory regions must be merged into a single memory descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the memory range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If one or more bytes of the memory range specified by *BaseAddress* and *Length* were not allocated on previous calls to **AllocateMemorySpace()**, then **EFI_NOT_FOUND** is returned.

If there are not enough system resources available to free the memory range, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The memory resource was freed from the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*Length* is zero.

EFI_UNSUPPORTED	The processor does not support one or more bytes of the memory resource range specified by *BaseAddress* and *Length*.
EFI_NOT_FOUND	The memory resource range specified by *BaseAddress* and *Length* was not allocated with previous calls to **AllocateMemorySpace()**.
EFI_OUT_OF_RESOURCES	There are not enough system resources to free the memory resource from the global coherency domain of the processor.

RemoveMemorySpace()

Summary

This service removes reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_REMOVE_MEMORY_SPACE) (
  IN EFI_PHYSICAL_ADDRESS   BaseAddress,
  IN UINT64                 Length
  );
```

Parameters

BaseAddress

The physical address that is the start address of the memory resource being removed. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The size in bytes of the memory resource that is being removed.

Description

The **RemoveMemorySpace()** function converts the memory range specified by *BaseAddress* and *Length* to the memory type **EfiGcdMemoryTypeNonExistent**. If this conversion is successful, then **EFI_SUCCESS** is returned.

If the GCD memory space map contains adjacent memory regions that only differ in their base address and length fields, then those adjacent memory regions must be merged into a single memory descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the memory range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If one or more bytes of the memory range specified by *BaseAddress* and *Length* were not added to the GCD memory space map with previous calls to **AddMemorySpace()**, then **EFI_NOT_FOUND** is returned.

If one or more bytes of the memory range specified by *BaseAddress* and *Length* were allocated from the GCD memory space map with previous calls to **AllocateMemorySpace()**, then **EFI_ACCESS_DENIED** is returned.

If there are not enough system resources available to remove the memory range, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The memory resource was removed from the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_UNSUPPORTED	The processor does not support one or more bytes of the memory resource range specified by *BaseAddress* and *Length*.
EFI_NOT_FOUND	One or more bytes of the memory resource range specified by *BaseAddress* and *Length* was not added with previous calls to **AddMemorySpace()**.
EFI_ACCESS_DENIED	One or more bytes of the memory resource range specified by *BaseAddress* and *Length* has been allocated with **AllocateMemorySpace()**.
EFI_OUT_OF_RESOURCES	There are not enough system resources to remove the memory resource from the global coherency domain of the processor.

GetMemorySpaceDescriptor()

Summary

This service retrieves the descriptor for a memory region containing a specified address.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_MEMORY_SPACE_DESCRIPTOR) (
  IN  EFI_PHYSICAL_ADDRESS              BaseAddress,
  OUT EFI_GCD_MEMORY_SPACE_DESCRIPTOR   *Descriptor
  );
```

Parameters

BaseAddress

The physical address that is the start address of a memory region. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Descriptor

A pointer to a caller allocated descriptor. On return, the descriptor describes the memory region containing *BaseAddress*. Type **EFI_GCD_MEMORY_SPACE_DESCRIPTOR** is defined in "Related Definitions" below.

Description

The **GetMemorySpaceDescriptor()** function retrieves the descriptor for the memory region that contains the address specified by *BaseAddress*. If a memory region containing *BaseAddress* is found, then the descriptor for that memory region is returned in the caller allocated structure *Descriptor*, and **EFI_SUCCESS** is returned.

If *Descriptor* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If a memory region containing *BaseAddress* is not present in the GCD memory space map, then **EFI_NOT_FOUND** is returned.

Related Definitions

```
//*******************************************************
// EFI_GCD_MEMORY_SPACE_DESCRIPTOR
//*******************************************************
typedef struct {
  EFI_PHYSICAL_ADDRESS  BaseAddress;
  UINT64                Length;
  UINT64                Capabilities;
  UINT64                Attributes;
  EFI_GCD_MEMORY_TYPE   GcdMemoryType;
  EFI_HANDLE            ImageHandle;
```

```
  EFI_HANDLE              DeviceHandle;
} EFI_GCD_MEMORY_SPACE_DESCRIPTOR;
```

Parameters

BaseAddress

The physical address of the first byte in the memory region. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The number of bytes in the memory region.

Capabilities

The bit mask of attributes that the memory region is capable of supporting. The bit mask of available attributes is defined in the **GetMemoryMap()** function description in the UEFI 2.0 specification.

Attributes

The bit mask of attributes that the memory region is currently using. The bit mask of available attributes is defined in **GetMemoryMap()**.

GcdMemoryType

Type of the memory region. Type **EFI_GCD_MEMORY_TYPE** is defined in the **AddMemorySpace()** function description.

ImageHandle

The image handle of the agent that allocated the memory resource described by *PhysicalStart* and *NumberOfBytes*. If this field is **NULL**, then the memory resource is not currently allocated. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

DeviceHandle

The device handle for which the memory resource has been allocated. If *ImageHandle* is **NULL**, then the memory resource is not currently allocated. If this field is **NULL**, then the memory resource is not associated with a device that is described by a device handle. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Status Codes Returned

EFI_SUCCESS	The descriptor for the memory resource region containing *BaseAddress* was returned in *Descriptor*.
EFI_INVALID_PARAMETER	*Descriptor* is **NULL**.
EFI_NOT_FOUND	A memory resource range containing *BaseAddress* was not found.

SetMemorySpaceAttributes()

Summary

This service modifies the attributes for a memory region in the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SET_MEMORY_SPACE_ATTRIBUTES) (
  IN EFI_PHYSICAL_ADDRESS          BaseAddress,
  IN UINT64                        Length,
  IN UINT64                        Attributes
  );
```

Parameters

BaseAddress

The physical address that is the start address of a memory region. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The size in bytes of the memory region.

Attributes

The bit mask of attributes to set for the memory region. The bit mask of available attributes is defined in the **GetMemoryMap()** function description in the UEFI 2.0 specification.

Description

The **SetMemorySpaceAttributes()** function modifies the attributes for the memory region specified by *BaseAddress* and *Length* from their current attributes to the attributes specified by *Attributes*. If this modification of attributes succeeds, then **EFI_SUCCESS** is returned.

If the GCD memory space map contains adjacent memory regions that only differ in their base address and length fields, then those adjacent memory regions must be merged into a single memory descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the memory range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If the attributes specified by *Attributes* are not supported for the memory region specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned. The *Attributes* bit mask must be a proper subset of the capabilities bit mask for the specified memory region. The capabilities bit mask is specified when a memory region is added with **AddMemorySpace()** and can be retrieved with **GetMemorySpaceDescriptor()** or **GetMemorySpaceMap()**.

If the attributes for one or more bytes of the memory range specified by *BaseAddress* and *Length* cannot be modified because the current system policy does not allow them to be modified, then **EFI_ACCESS_DENIED** is returned.

If there are not enough system resources available to modify the attributes of the memory range, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The attributes were set for the memory region.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_UNSUPPORTED	The processor does not support one or more bytes of the memory resource range specified by *BaseAddress* and *Length*.
EFI_UNSUPPORTED	The bit mask of attributes is not support for the memory resource range specified by *BaseAddress* and *Length*.
EFI_ACCESS_DENIED	The attributes for the memory resource range specified by *BaseAddress* and *Length* cannot be modified.
EFI_OUT_OF_RESOURCES	There are not enough system resources to modify the attributes of the memory resource range.

GetMemorySpaceMap()

Summary

Returns a map of the memory resources in the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_MEMORY_SPACE_MAP) (
  OUT UINTN                                *NumberOfDescriptors,
  OUT EFI_GCD_MEMORY_SPACE_DESCRIPTOR      **MemorySpaceMap
  );
```

Parameters

NumberOfDescriptors

A pointer to number of descriptors returned in the *MemorySpaceMap* buffer. This parameter is ignored on input, and is set to the number of descriptors in the *MemorySpaceMap* buffer on output.

MemorySpaceMap

A pointer to the array of **EFI_GCD_MEMORY_SPACE_DESCRIPTOR**s. Type **EFI_GCD_MEMORY_SPACE_DESCRIPTOR** is defined in **GetMemorySpaceDescriptor()**. This buffer is allocated with **AllocatePool()**, so it is the caller's responsibility to free this buffer with a call to **FreePool()**. The number of descriptors in *MemorySpaceMap* is returned in *NumberOfDescriptors*. See the UEFI 2.0 specification for definitions of **AllocatePool()** and **FreePool()**.

Description

The **GetMemorySpaceMap()** function retrieves the entire GCD memory space map. If there are no errors retrieving the GCD memory space map, then the number of descriptors in the GCD memory space map is returned in *NumberOfDescriptors*, the array of descriptors from the GCD memory space map is allocated with **AllocatePool()**, the descriptors are transferred into *MemorySpaceMap*, and **EFI_SUCCESS** is returned.

If *NumberOfDescriptors* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *MemorySpaceMap* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If there are not enough resources to allocate *MemorySpaceMap*, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The memory space map was returned in the *MemorySpaceMap* buffer, and the number of descriptors in *MemorySpaceMap* was returned in *NumberOfDescriptors*.
EFI_INVALID_PARAMETER	*NumberOfDescriptors* is **NULL**.

EFI_INVALID_PARAMETER	*MemorySpaceMap* is **NULL**.
EFI_OUT_OF_RESOURCES	There are not enough resources to allocate *MemorySpaceMap*.

AddIoSpace()

Summary

This service adds reserved I/O, or I/O resources to the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ADD_IO_SPACE) (
  IN EFI_GCD_IO_TYPE         GcdIoType,
  IN EFI_PHYSICAL_ADDRESS    BaseAddress,
  IN UINT64                  Length
  );
```

Parameters

GcdIoType

The type of I/O resource being added. Type **EFI_GCD_IO_TYPE** is defined in "Related Definitions" below. The only types allowed are **EfiGcdIoTypeReserved** and **EfiGcdIoTypeIo**.

BaseAddress

The physical address that is the start address of the I/O resource being added. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The size in bytes of the I/O resource that is being added.

Description

The **AddIoSpace()** function converts unallocated non-existent I/O ranges to a range of reserved I/O, or a range of I/O. *BaseAddress* and *Length* specify the I/O range, and *GcdIoType* specifies the I/O type. If the I/O range is successfully added, then **EFI_SUCCESS** is returned.

If the GCD I/O space map contains adjacent I/O regions that only differ in their base address and length fields, then those adjacent I/O regions must be merged into a single I/O descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If *GcdIoType* is not **EfiGcdIoTypeReserved** or **EfiGcdIoTypeIo**, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the I/O range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If any portion of the I/O range specified by *BaseAddress* and *Length* is not of type **EfiGcdIoTypeNonExistent**, then **EFI_ACCESS_DENIED** is returned.

If any portion of the I/O range specified by *BaseAddress* and *Length* was allocated in a prior call to **AllocateIoSpace()**, then **EFI_ACCESS_DENIED** is returned.

If there are not enough system resources available to add the I/O resource to the global coherency domain of the processor, then **EFI_OUT_OF_RESOURCES** is returned.

Related Definitions

```
//*******************************************************
// EFI_GCD_IO_TYPE
//*******************************************************
typedef enum {
  EfiGcdIoTypeNonExistent,
  EfiGcdIoTypeReserved,
  EfiGcdIoTypeIo,
  EfiGcdIoTypeMaximum
} EFI_GCD_IO_TYPE;
```

EfiGcdIoTypeNonExistent

An I/O region that is visible to the boot processor. However, there are no system components that are currently decoding this I/O region.

EfiGcdIoTypeReserved

An I/O region that is visible to the boot processor. This I/O region is currently being decoded by a system component, but the I/O region cannot be used to access I/O devices.

EfiGcdIoTypeIo

An I/O region that is visible to the boot processor. This I/O region is currently being decoded by a system component that is producing I/O ports that can be used to access I/O devices.

Status Codes Returned

EFI_SUCCESS	The I/O resource was added to the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*GcdIoType* is invalid.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_OUT_OF_RESOURCES	There are not enough system resources to add the I/O resource to the global coherency domain of the processor.
EFI_UNSUPPORTED	The processor does not support one or more bytes of the I/O resource range specified by *BaseAddress* and *Length*.
EFI_ACCESS_DENIED	One or more bytes of the I/O resource range specified by *BaseAddress* and *Length* conflicts with an I/O resource range that was previously added to the global coherency domain of the processor.
EFI_ACCESS_DENIED	One or more bytes of the I/O resource range specified by *BaseAddress* and *Length* was allocated in a prior call to **AllocateIoSpace()**.

AllocateIoSpace()

Summary

This service allocates nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ALLOCATE_IO_SPACE) (
  IN     EFI_GCD_ALLOCATE_TYPE    AllocateType,
  IN     EFI_GCD_IO_TYPE          GcdIoType,
  IN     UINTN                    Alignment,
  IN     UINT64                   Length,
  IN OUT EFI_PHYSICAL_ADDRESS     *BaseAddress,
  IN     EFI_HANDLE               ImageHandle,
  IN     EFI_HANDLE               DeviceHandle   OPTIONAL
  );
```

Parameters

GcdAllocateType

The type of allocation to perform. Type **EFI_GCD_ALLOCATE_TYPE** is defined in **AllocateMemorySpace()**.

GcdIoType

The type of I/O resource being allocated. Type **EFI_GCD_IO_TYPE** is defined in **AddIoSpace()**. The only types allowed are **EfiGcdIoTypeNonExistent**, **EfiGcdIoTypeReserved**, and **EfiGcdIoTypeIo**.

Alignment

The log base 2 of the boundary that *BaseAddress* must be aligned on output. For example, a value of 0 means that *BaseAddress* can be aligned on any byte boundary, and a value of 12 means that *BaseAddress* must be aligned on a 4 KB boundary.

Length

The size in bytes of the I/O resource range that is being allocated.

BaseAddress

A pointer to a physical address. On input, the way in which the address is used depends on the value of *Type*. See "Description" below for more information. On output the address is set to the base of the I/O resource range that was allocated. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

ImageHandle

The image handle of the agent that is allocating the I/O resource. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *v*.

DeviceHandle

The device handle for which the I/O resource is being allocated. If the I/O resource is not being allocated for a device that has an associated device handle, then this parameter is optional and may be **NULL**. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Description

The **AllocateIoSpace()** function searches for an I/O range of type *GcdIoType* and converts the discovered I/O range from the unallocated state to the allocated state. The parameters *GcdAllocateType*, *Alignment*, *Length*, and *BaseAddress* specify the manner in which the GCD I/O space map is searched. If an I/O range is found that meets the search criteria, then the base address of the I/O range is returned in *BaseAddress*, and **EFI_SUCCESS** is returned. *ImageHandle* and *DeviceHandle* are used to convert the I/O range from the unallocated state to the allocated state. *ImageHandle* identifies the image that is calling **AllocateIoSpace()**, and *DeviceHandle* identifies the device that *ImageHandle* is managing that requires the I/O range. *DeviceHandle* is optional, because the device that *ImageHandle* is managing might not have an associated device handle. If an I/O range meeting the search criteria cannot be found, then **EFI_NOT_FOUND** is returned.

If *GcdAllocateType* is **EfiGcdAllocateAnySearchBottomUp**, then the GCD I/O space map is searched from the lowest address up to the highest address looking for unallocated I/O ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdIoType*.

If *GcdAllocateType* is **EfiGcdAllocateAnySearchTopDown**, then the GCD I/O space map is searched from the highest address down to the lowest address looking for unallocated I/O ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdIoType*.

If *GcdAllocateType* is **EfiGcdAllocateMaxAddressSearchBottomUp**, then the GCD I/O space map is searched from the lowest address up to *BaseAddress* looking for unallocated I/O ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdIoType*.

If *GcdAllocateType* is **EfiGcdAllocateMaxAddressSearchTopDown**, then the GCD I/O space map is searched from *BaseAddress* down to the lowest address looking for unallocated I/O ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdIoType*.

If *GcdAllocateType* is **EfiGcdAllocateAddress**, then the GCD I/O space map is checked to see if the I/O range starting at *BaseAddress* for *Length* bytes is of type *GcdIoType*, unallocated, and begins on a the boundary specified by *Alignment*.

If the GCD I/O space map contains adjacent I/O regions that only differ in their base address and length fields, then those adjacent I/O regions must be merged into a single I/O descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If *BaseAddress* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *ImageHandle* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *GcdIoType* is not **EfiGcdIoTypeNonExistent**, **EfiGcdIoTypeReserved**, or **EfiGcdIoTypeIo**, then **EFI_INVALID_PARAMETER** is returned.

If *GcdAlocateType* is less than zero, or *GcdAllocateType* is greater than or equal to *EfiGcdMaxAllocateType* then **EFI_INVALID_PARAMETER** is returned.

If there are not enough system resources available to allocate the I/O range, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The I/O resource was allocated from the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*GcdAllocateType* is invalid.
EFI_INVALID_PARAMETER	*GcdIoType* is invalid.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_INVALID_PARAMETER	*BaseAddress* is **NULL**.
EFI_INVALID_PARAMETER	*ImageHandle* is **NULL**.
EFI_OUT_OF_RESOURCES	There are not enough system resources to allocate the I/O resource from the global coherency domain of the processor.
EFI_NOT_FOUND	The I/O resource request could not be satisfied.

FreeIoSpace()

Summary

This service frees nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_FREE_IO_SPACE) (
  IN EFI_PHYSICAL_ADDRESS   BaseAddress,
  IN UINT64                 Length
  );
```

Parameters

BaseAddress

The physical address that is the start address of the I/O resource being freed. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The size in bytes of the I/O resource range that is being freed.

Description

The **FreeIoSpace()** function converts the I/O range specified by *BaseAddress* and *Length* from the allocated state to the unallocated state. If this conversion is successful, then **EFI_SUCCESS** is returned.

If the GCD I/O space map contains adjacent I/O regions that only differ in their base address and length fields, then those adjacent I/O regions must be merged into a single I/O descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the I/O range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If one or more bytes of the I/O range specified by *BaseAddress* and *Length* were not allocated on previous calls to **AllocateIoSpace()**, then **EFI_NOT_FOUND** is returned.

If there are not enough system resources available to free the I/O range, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The I/O resource was freed from the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_UNSUPPORTED	The processor does not support one or more bytes of the I/O resource range specified by *BaseAddress* and *Length*.

EFI_NOT_FOUND	The I/O resource range specified by *BaseAddress* and *Length* was not allocated with previous calls to **AllocateIoSpace()**.
EFI_OUT_OF_RESOURCES	There are not enough system resources to free the I/O resource from the global coherency domain of the processor.

RemoveIoSpace()

Summary

This service removes reserved I/O, or I/O resources from the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_REMOVE_IO_SPACE) (
  IN EFI_PHYSICAL_ADDRESS    BaseAddress,
  IN UINT64                  Length
  );
```

Parameters

BaseAddress

A pointer to a physical address that is the start address of the I/O resource being removed. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

Length

The size in bytes of the I/O resource that is being removed.

Description

The **RemoveIoSpace()** function converts the I/O range specified by *BaseAddress* and *Length* to the I/O type **EfiGcdIoTypeNonExistent**. If this conversion is successful, then **EFI_SUCCESS** is returned.

If the GCD I/O space map contains adjacent I/O regions that only differ in their base address and length fields, then those adjacent I/O regions must be merged into a single I/O descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the I/O range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If one or more bytes of the I/O range specified by *BaseAddress* and *Length* were not added to the GCD I/O space map with previous calls to **AddIoSpace()**, then **EFI_NOT_FOUND** is returned.

If one or more bytes of the I/O range specified by *BaseAddress* and *Length* were allocated from the GCD I/O space map with previous calls to **AllocateIoSpace()**, then **EFI_ACCESS_DENIED** is returned.

If there are not enough system resources available to remove the I/O range, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The I/O resource was removed from the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_UNSUPPORTED	The processor does not support one or more bytes of the I/O resource range specified by *BaseAddress* and *Length*.
EFI_NOT_FOUND	One or more bytes of the I/O resource range specified by *BaseAddress* and *Length* was not added with previous calls to **AddIoSpace()**.
EFI_ACCESS_DENIED	One or more bytes of the I/O resource range specified by *BaseAddress* and *Length* has been allocated with **AllocateIoSpace()**.
EFI_OUT_OF_RESOURCES	There are not enough system resources to remove the I/O resource from the global coherency domain of the processor.

GetIoSpaceDescriptor()

Summary

This service retrieves the descriptor for an I/O region containing a specified address.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_IO_SPACE_DESCRIPTOR) (
  IN  EFI_PHYSICAL_ADDRESS          BaseAddress,
  OUT EFI_GCD_IO_SPACE_DESCRIPTOR   *Descriptor
  );
```

Parameters

BaseAddress

The physical address that is the start address of an I/O region. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

Descriptor

A pointer to a caller allocated descriptor. On return, the descriptor describes the I/O region containing *BaseAddress*. Type **EFI_GCD_IO_SPACE_DESCRIPTOR** is defined in "Related Definitions" below.

Description

The **GetIoSpaceDescriptor()** function retrieves the descriptor for the I/O region that contains the address specified by *BaseAddress*. If an I/O region containing *BaseAddress* is found, then the descriptor for that I/O region is returned in the caller allocated structure *Descriptor*, and **EFI_SUCCESS** is returned.

If *Descriptor* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If an I/O region containing *BaseAddress* is not present in the GCD I/O space map, then **EFI_NOT_FOUND** is returned.

Related Definitions

```
//*******************************************************
// EFI_GCD_IO_SPACE_DESCRIPTOR
//*******************************************************
typedef struct {
  EFI_PHYSICAL_ADDRESS  BaseAddress;
  UINT64                Length;
  EFI_GCD_IO_TYPE       GcdIoType;
  EFI_HANDLE            ImageHandle;
  EFI_HANDLE            DeviceHandle;
} EFI_GCD_IO_SPACE_DESCRIPTOR;
```

Parameters

BaseAddress

Physical address of the first byte in the I/O region. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

Number of bytes in the I/O region.

GcdIoType

Type of the I/O region. Type **EFI_GCD_IO_TYPE** is defined in the **AddIoSpace()** function description.

ImageHandle

The image handle of the agent that allocated the I/O resource described by *PhysicalStart* and *NumberOfBytes*. If this field is **NULL**, then the I/O resource is not currently allocated. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

DeviceHandle

The device handle for which the I/O resource has been allocated. If *ImageHandle* is **NULL**, then the I/O resource is not currently allocated. If this field is **NULL**, then the I/O resource is not associated with a device that is described by a device handle. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Status Codes Returned

EFI_SUCCESS	The descriptor for the I/O resource region containing *BaseAddress* was returned in *Descriptor*.
EFI_INVALID_PARAMETER	*Descriptor* is **NULL**.
EFI_NOT_FOUND	An I/O resource range containing *BaseAddress* was not found.

GetIoSpaceMap()

Summary

Returns a map of the I/O resources in the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_IO_SPACE_MAP) (
  OUT UINTN                            *NumberOfDescriptors,
  OUT EFI_GCD_IO_SPACE_DESCRIPTOR      **IoSpaceMap
  );
```

Parameters

NumberOfDescriptors

A pointer to number of descriptors returned in the *IoSpaceMap* buffer. This parameter is ignored on input, and is set to the number of descriptors in the *IoSpaceMap* buffer on output.

IoSpaceMap

A pointer to the array of **EFI_GCD_IO_SPACE_DESCRIPTOR**s. Type **EFI_GCD_IO_SPACE_DESCRIPTOR** is defined in **GetIoSpaceDescriptor()**. This buffer is allocated with **AllocatePool()**, so it is the caller's responsibility to free this buffer with a call to **FreePool()**. The number of descriptors in *IoSpaceMap* is returned in *NumberOfDescriptors*.

Description

The **GetIoSpaceMap()** function retrieves the entire GCD I/O space map. If there are no errors retrieving the GCD I/O space map, then the number of descriptors in the GCD I/O space map is returned in *NumberOfDescriptors*, the array of descriptors from the GCD I/O space map is allocated with **AllocatePool()**, the descriptors are transferred into *IoSpaceMap*, and **EFI_SUCCESS** is returned.

If *NumberOfDescriptors* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *IoSpaceMap* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If there are not enough resources to allocate *IoSpaceMap*, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The I/O space map was returned in the *IoSpaceMap* buffer, and the number of descriptors in *IoSpaceMap* was returned in *NumberOfDescriptors*.
EFI_INVALID_PARAMETER	*NumberOfDescriptors* is **NULL**.
EFI_INVALID_PARAMETER	*IoSpaceMap* is **NULL**.

EFI_OUT_OF_RESOURCES	There are not enough resources to allocate *IoSpaceMap*.

7.3 Dispatcher Services

7.3.1 Dispatcher Services

The functions that make up the Dispatcher Services are used during preboot to schedule drivers for execution. A driver may optionally have the Schedule On Request (SOR) flag set in the driver's dependency expression. Drivers with this bit set will not be loaded and invoked until they are explicitly requested to do so. Files loaded from firmware volumes may be placed in the untrusted state by the Security Architectural Protocol. The services in this section provide this ability to clear the SOR flag in a DXE driver's dependency expression and the ability to promote a file from a firmware volume from the untrusted to the trusted state. Table 10 lists the Dispatcher Services.

Table 10. Dispatcher Boot Type Services

Name	Description
Dispatch	Loads and executed DXE drivers from firmware volumes.
Schedule	Clears the Schedule on Request (SOR) flag for a component that is stored in a firmware volume.
Trust	Changes the state of a file stored in a firmware volume from the untrusted state to the trusted state.
ProcessFirmwareVolume	Creates a firmware volume handle for a firmware volume that is present in system memory.

Dispatch()

Summary

Loads and executes DXE drivers from firmware volumes.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DISPATCH) (
  VOID
  );
```

Description

The **Dispatch()** function searches for DXE drivers in firmware volumes that have been installed since the last time the **Dispatch()** service was called. It then evaluates the dependency expressions of all the DXE drivers and loads and executes those DXE drivers whose dependency expression evaluate to **TRUE**. This service must interact with the Security Architectural Protocol to authenticate DXE drivers before they are executed. This process is continued until no more DXE drivers can be executed. If one or more DXE drivers are executed, then **EFI_SUCCESS** is returned. If no DXE drivers are executed, **EFI_NOT_FOUND** is returned.

If an attempt is made to invoke the DXE Dispatcher recursively, then no action is performed by the **Dispatch()** service, and **EFI_ALREADY_STARTED** is returned. In this case, because the DXE Dispatcher is already running, it is not necessary to invoke it again. All the DXE drivers that can be dispatched will be dispatched.

Status Codes Returned

EFI_SUCCESS	One or more DXE driver were dispatched.
EFI_NOT_FOUND	No DXE drivers were dispatched.
EFI_ALREADY_STARTED	An attempt is being made to start the DXE Dispatcher recursively. Thus no action was taken.

Schedule()

Summary

Clears the Schedule on Request (SOR) flag for a component that is stored in a firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SCHEDULE) (
  IN EFI_HANDLE          FirmwareVolumeHandle,
  IN CONST EFI_GUID      *FileName
  );
```

Parameters

FirmwareVolumeHandle

The handle of the firmware volume that contains the file specified by *FileName*. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

FileName

A pointer to the name of the file in a firmware volume. This is the file that should have its SOR bit cleared. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Description

The **Schedule()** function searches the dispatcher queues for the driver specified by *FirmwareVolumeHandle* and *FileName*. If this driver cannot be found, then **EFI_NOT_FOUND** is returned. If the driver is found, and its Schedule On Request (SOR) flag is not set in its dependency expression, then **EFI_NOT_FOUND** is returned. If the driver is found, and its SOR bit is set in its dependency expression, then the SOR flag is cleared, and **EFI_SUCCESS** is returned. After the SOR flag is cleared, the driver will be dispatched if the remaining portions of its dependency expression are satisfied. This service does not automatically invoke the DXE Dispatcher. Instead, the **Dispatch()** service must be used to invoke the DXE Dispatcher.

Status Codes Returned

EFI_SUCCESS	The DXE driver was found and its SOR bit was cleared.
EFI_NOT_FOUND	The DXE driver does not exist, or the DXE driver exists and its SOR bit is not set.

Trust()

Summary

Promotes a file stored in a firmware volume from the untrusted to the trusted state. Only the Security Architectural Protocol can place a file in the untrusted state. A platform specific component may choose to use this service to promote a previously untrusted file to the trusted state.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_TRUST) (
  IN EFI_HANDLE          FirmwareVolumeHandle,
  IN CONST EFI_GUID      *FileName
  );
```

Parameters

FirmwareVolumeHandle

The handle of the firmware volume that contains the file specified by *FileName*. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

FileName

A pointer to the name of the file in a firmware volume. This is the file that should be promoted from the untrusted state to the trusted state. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Description

The **Trust()** function promotes the file specified by *FirmwareVolumeHandle* and *FileName* from the untrusted state to the trusted state. If this file is not found in the queue of untrusted files, then **EFI_NOT_FOUND** is returned. If the driver is found, and its state is changed to trusted and **EFI_SUCCESS** is returned. This service does not automatically invoke the DXE Dispatcher. Instead, the **Dispatch()** service must be used to invoke the DXE Dispatcher.

Status Codes Returned

EFI_SUCCESS	The file was found in the untrusted state, and it was promoted to the trusted state.
EFI_NOT_FOUND	The file was not found in the untrusted state.

ProcessFirmwareVolume()

Summary

Creates a firmware volume handle for a firmware volume that is present in system memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PROCESS_FIRMWARE_VOLUME) (
  IN  CONST VOID  *FirmwareVolumeHeader,
  IN  UINTN       Size,
  OUT EFI_HANDLE  *FirmwareVolumeHandle
  );
```

Parameters

FirmwareVolumeHeader

A pointer to the header of the firmware volume.

Size

The size, in bytes, of the firmware volume.

FirmwareVolumeHandle

On output, a pointer to the created handle. This service will install the **EFI_FIRMWARE_VOLUME2_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL** for the of the firmware volume that is described by *FirmwareVolumeHeader* and *Size*. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Description

The **ProcessFirmwareVolume()** function examines the contents of the buffer specified by *FirmwareVolumeHeader* and *Size*. If the buffer contains a valid firmware volume, then a new handle is created, and the **EFI_FIRMWARE_VOLUME2_PROTOCOL** and a memory-mapped **EFI_DEVICE_PATH_PROTOCOL** are installed onto the new handle. The new handle is returned in *FirmwareVolumeHandle*.

Status Codes Returned

EFI_SUCCESS	The **EFI_FIRMWARE_VOLUME2_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL** were installed onto *FirmwareVolumeHandle* for the firmware volume described by *FirmwareVolumeHeader* and *Size*.
EFI_VOLUME_CORRUPTED	The firmware volume described by *FirmwareVolumeHeader* and *Size* is corrupted.

EFI_OUT_OF_RESOURCES	There are not enough system resources available to produce the **EFI_FIRMWARE_VOLUME2_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL** for the firmware volume described by *FirmwareVolumeHeader* and *Size*.

8
Protocols - Device Path Protocol

8.1 Introduction

This section adds two device path node types that describe files stored in firmware volumes:

- Firmware File Media Device Path
- Firmware Volume Media Device Path

These device path nodes are used by a DXE-aware updated UEFI Boot Service **`LoadImage()`** to load UEFI images from firmware volumes. This new capability is used by the DXE Dispatcher to load DXE drivers from firmware volumes.

8.2 Firmware Volume Media Device Path

This type is used by systems implementing the PI architecture specifications to describe a firmware volume.

Table 11. Firmware Volume Media Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 4 – Media Device Path
Sub-Type	1	1	Sub Type 7 – Firmware Volume Media Device Path
Length	2	2	Length of this structure in bytes. Length is 20 bytes.
Firmware Volume Name	4	16	Firmware volume name. Type EFI_GUID.

Table 12. Firmware Volume Device Node Text Representation

Device Node Type/Subtype/Other	Description
Type: 4 (Media Device Path) Sub-Type: 7 (Firmware Volume)	**`Fv(fv-guid)`** The **`fv-guid`** is a GUID.

8.3 Firmware File Media Device Path

This type is used by systems implementing the PI architecture specifications to describe a firmware file in a firmware volume.

Table 13. Firmware File Media Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 4 – Media Device Path
Sub-Type	1	1	Sub Type 6 – Firmware File Media Device Path
Length	2	2	Length of this structure in bytes. Length is 20 bytes.
Firmware File Name	4	16	Firmware file name. Type EFI_GUID.

Table 14. Firmware Volume File Device Node Text Representation

Device Node Type/Subtype/Other	Description
Type: 4 (Media Device Path) Sub-Type: 6 (Firmware File)	**`FvFile(fvfile-guid)`** The **`fvfile-guid`** is a GUID.

9
DXE Foundation

9.1 Introduction

The DXE Foundation is designed to be completely portable with no processor, chipset, or platform dependencies. This lack of dependencies is accomplished by designing in several features:

- The DXE Foundation depends only upon a HOB list for its initial state.

  This means that the DXE Foundation does not depend on any services from a previous phase, so all the prior phases can be unloaded once the HOB list is passed to the DXE Foundation.
- The DXE Foundation does not contain any hard-coded addresses.

  This means that the DXE Foundation can be loaded anywhere in physical memory, and it can function correctly no matter where physical memory or where Firmware Volumes (FVs) are located in the processor's physical address space.
- The DXE Foundation does not contain any processor-specific, chipset-specific, or platform-specific information.

  Instead, the DXE Foundation is abstracted from the system hardware through a set of DXE Architectural Protocol interfaces. These architectural protocol interfaces are produced by a set of DXE drivers that are invoked by the DXE Dispatcher.

The DXE Foundation must produce the UEFI System Table and its associated set of UEFI Boot Services and UEFI Runtime Services. The DXE Foundation also contains the DXE Dispatcher whose main purpose is to discover and execute DXE drivers stored in FVs. The execution order of DXE drivers is determined by a combination of the optional a priori file and the set of dependency expressions that are associated with the DXE drivers. The FV file format allows dependency expressions to be packaged with the executable DXE driver image. DXE drivers utilize a PE/COFF image format, so the DXE Dispatcher must also contain a PE/COFF loader to load and execute DXE drivers.

9.2 Hand-Off Block (HOB) List

The Hand-Off Block (HOB) list contains all the information that the DXE Foundation requires to produce its memory-based services. The HOB list contains the following:

- Information on the boot mode and the memory that was allocated in the previous phase.
- A description of the system memory that was initialized by the previous phase along with information about the firmware devices that were discovered in the previous phase.

The firmware device information includes the system memory locations of the firmware devices and system memory locations of the firmware volumes that are contained within those firmware devices. The firmware volumes may contain DXE drivers, and the DXE Dispatcher is responsible for loading and executing the DXE drivers that are discovered in those firmware volumes.

The I/O resources and memory-mapped I/O resources that were discovered in the previous phase.

The HOB list must be treated as a read-only data structure. It conveys the state of the system at the time the DXE Foundation is started. The DXE Foundation and DXE drivers should never modify the contents of the HOB list.

Figure 4 shows an example HOB list. The first HOB list entry is always the Phase Handoff Information Table (PHIT) HOB that contains the boot mode and a description of the memory regions used by the previous phase. The rest of the HOB list entries can appear in any order. This example shows the various HOB types that are supported. The most important ones to the DXE Foundation are the HOBs that describe system memory and the firmware volumes. A HOB list is terminated by an end of list HOB. There is one additional HOB type that is not shown. This is a GUIDed HOB that allows a module from the previous phase to pass private data to a DXE driver. Only the DXE driver that recognizes the GUID value in the GUIDed HOB will be able to understand the data in the GUIDed HOB. The DXE Foundation does not consume any GUIDed HOBs. The HOB entries are all designed to be position independent. This allows the DXE Foundation to relocate the HOB list to a different location if the DXE Foundation does not like where the previous phase placed the HOB list in memory.

See "HOB Translations" on page 88 for more information on HOB types.

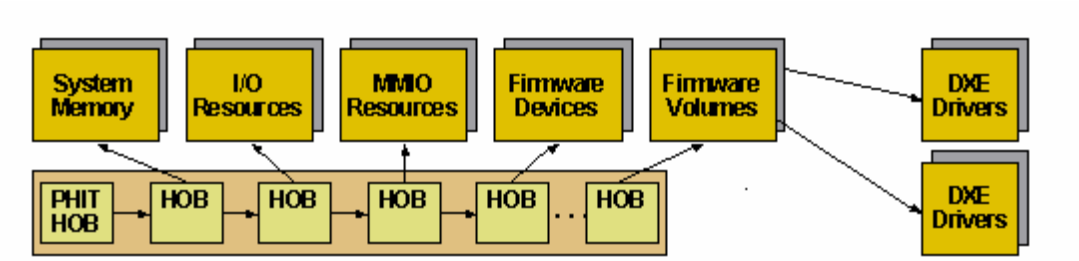

Figure 4. HOB List

9.3 DXE Foundation Data Structures

The DXE Foundation produces the UEFI System Table, and the UEFI System Table is consumed by every DXE driver and executable image invoked by the DXE Dispatcher and BDS. It contains all the information required for these components to utilize the services provided by the DXE Foundation and the services provided by any previously loaded DXE driver. Figure 5 shows the various components that are available through the UEFI System Table.

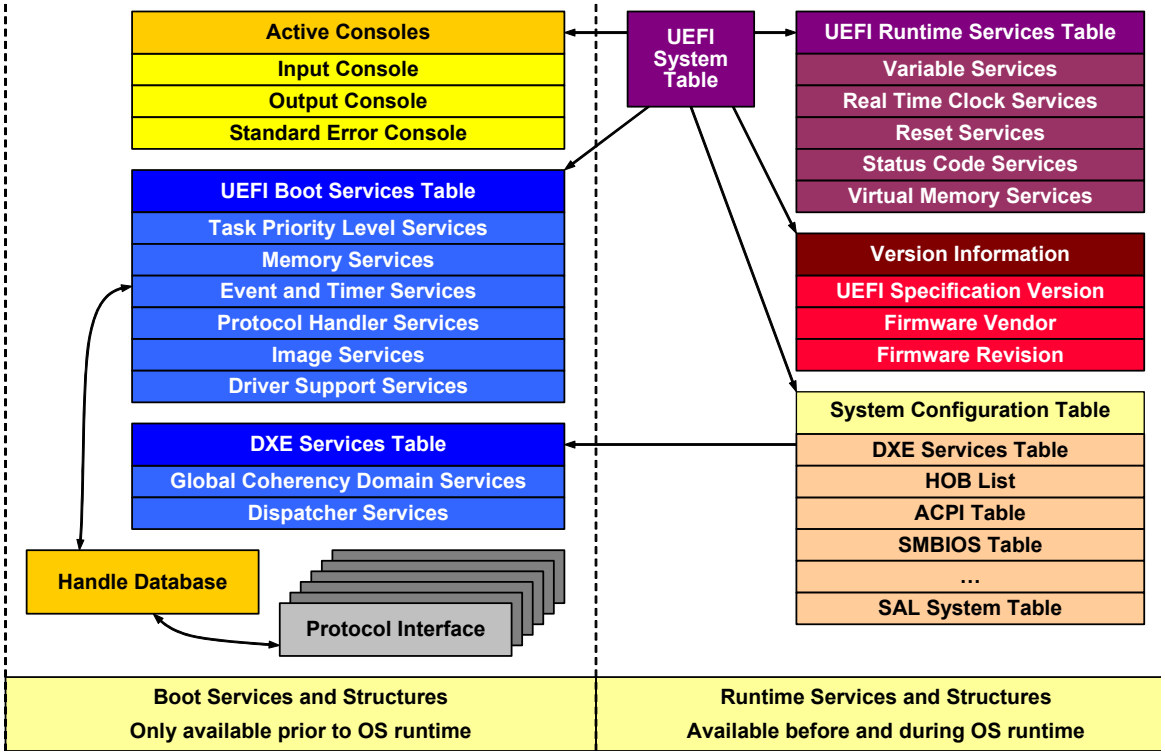

Figure 5. UEFI System Table and Related Components

The DXE Foundation produces the UEFI Boot Services, UEFI Runtime Services, and DXE Services with the aide of the DXE Architectural Protocols. The UEFI System Table also provides access to all the active console devices in the platform and the set of UEFI Configuration Tables. The UEFI Configuration Tables are an extensible list of tables that describe the configuration of the platform. Today, this includes pointers to tables such as DXE Services, the HOB list, ACPI table, SMBIOS table, and the SAL System Table. This list may be expanded in the future as new table types are defined. Also, through the use of the Protocol Handle Services in the UEFI Boot Services Table, any executable image can access the handle database and any of the protocol interfaces that have been registered by DXE drivers.

When the transition to the OS runtime is performed, the handle database, active consoles, UEFI Boot Services, DXE Services, and services provided by boot service DXE drivers are terminated. This frees up memory for use by the OS. This only leaves the UEFI System Table, UEFI Runtime

Services Table, and the UEFI Configuration Tables available in the OS runtime environment. There is also the option of converting all of the UEFI Runtime Services from a physical address space to an OS-specific virtual address space. This address space conversion may be performed only once.

9.4 Required DXE Foundation Components

Figure 6 shows the components that a DXE Foundation must contain. A detailed description of these component follows.

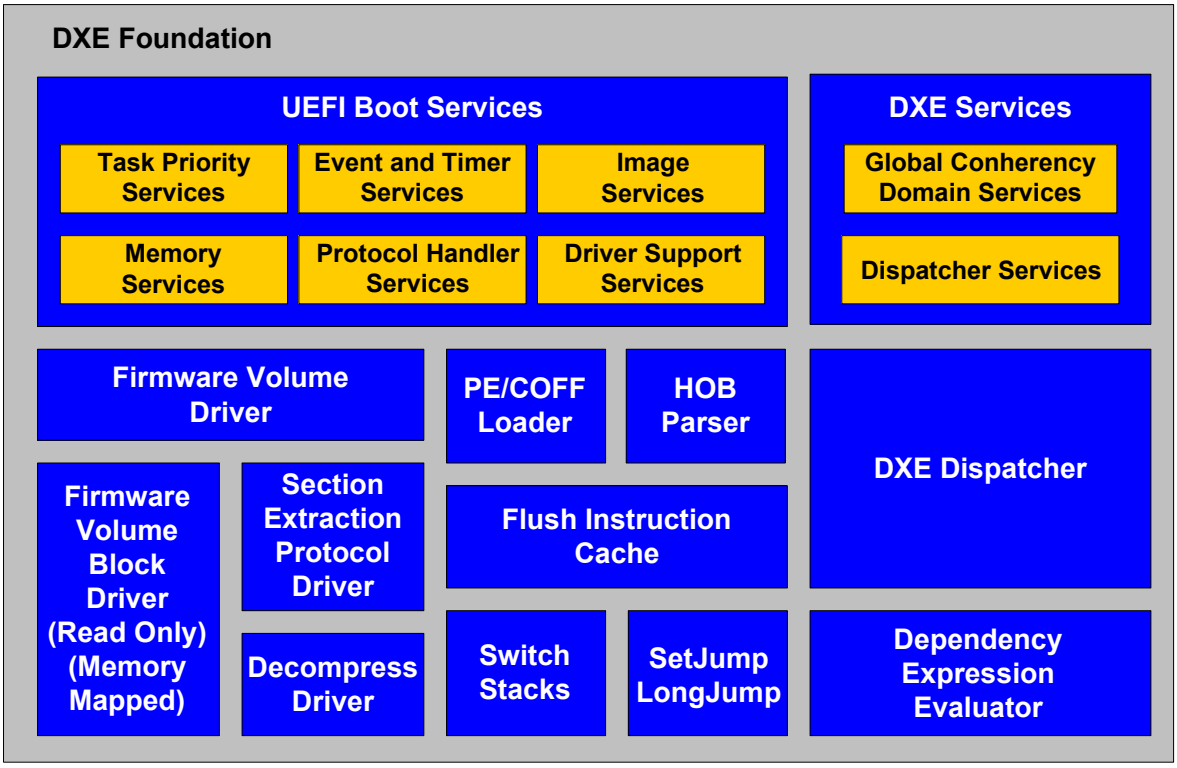

Figure 6. DXE Foundation Components

A DXE Foundation must have the following components:

- An implementation of the UEFI Boot Services. UEFI Boot Services Dependencies describes which services can be made available based on the HOB list alone and which services depend on the presence of architectural protocols.
- An implementation of the DXE Services. DXE Services Dependencies describes which services can be made available based on the HOB list alone and which services depend on the presence of architectural protocols.
- A HOB Parser that consumes the HOB list specified by `HobStart` and initializes the UEFI memory map, GCD memory space map, and GCD I/O space map. See section if for details on the translation from HOBs to the maps maintained by the DXE Foundation

- An implementation if a DXE Dispatcher that includes a dependency expression evaluator. See "DXE Dispatcher" on page 91 for a detailed description of this component.
- A Firmware Volume driver that produces the **EFI_FIRMWARE_VOLUME2_PROTOCOL** for every firmware volume described in the HOB list. This component is used by the DXE Dispatcher to search for *a priori* files and DXE drivers in firmware volumes. See the *Platform Initialization Specification*, Volume 3, for the definition of the Firmware Volume Protocol.
- An instance of the **EFI_DECOMPRESS_PROTOCOL**. See the UEFI 2.0 specification for the detailed requirements for this component. This component is required by the DXE Dispatcher to read compressed sections from DXE drivers stored in firmware volumes. It is expected that most DXE drivers will utilize compressed sections to reduce the size of firmware volumes.
- The DXE Dispatcher uses the Boot Service **StartImage()** to invoke a DXE driver. The Boot Services **StartImage()** and **Exit()** work together to hand control to a DXE driver and return control to the DXE Foundation. Since the Boot Service **Exit()** can be called for anywhere inside a DXE driver, the Boot Service **Exit()** is required to rebalance the stack, so it is in the same state it was in when the Boot Service **Start()** was called. This is typically implemented using the processor-specific functions called **SetJump()** and **LongJump()**. Since the DXE Foundation must use the Boot Services **StartImage()** and **Exit()** to invoke DXE drivers, the routines **SetJump()** and **LongJump()** are required by the DXE Foundation.
- A PE/COFF loader that supports PE32+ image types. This PE/COFF loader is used to implement the UEFI Boot Service **LoadImage()**. The DXE Dispatcher uses the Boot Service **LoadImage()** to load DXE drivers into system memory. If the processor that the DXE Foundation is compiled for requires an instruction cache when an image is loaded into system memory, then an instruction cache flush routine is also required in the DXE Foundation.
- The phase that executed prior to DXE will initialize a stack for the DXE Foundation to use. This stack is described in the HOB list. If the size of this stack does not meet the DXE Foundation's minimum stack size requirement or the stack is not located in memory region that is suitable to the DXE Foundation, then the DXE Foundation will have to allocate a new stack that does meet the minimum size and location requirements. As a result, the DXE Foundation must contain a stack switching routine for the processor type that the DXE Foundation is compiled.

9.5 Handing Control to DXE Dispatcher

The DXE Foundation must complete the following tasks before handing control to the DXE Dispatcher. The order that these tasks are performed is implementation dependent.

- Use the HOB list to initialize the GCD memory space map, the GCD I/O space map, and UEFI memory map.
- Allocate the UEFI Boot Services Table from **EFI_BOOT_SERVICES_MEMORY** and initialize the services that only require system memory to function correctly. The remaining UEFI Boot Services must be filled in with a service that returns **EFI_NOT_AVAILABLE_YET**.
- Allocate the DXE Services Table from **EFI_BOOT_SERVICES_MEMORY** and initialize the services that only require system memory to function correctly. The remaining DXE Services must be filled in with a service that returns **EFI_NOT_AVAILABLE_YET**.
- Allocate the UEFI Runtime Services Table from **EFI_RUNTIME_SERVICES_MEMORY** and initialize all the services to a service that returns **EFI_NOT_AVAILABLE_YET**.
- Allocate the UEFI System Table from **EFI_RUNTIME_SERVICES_MEMORY** and initialize all the fields.
- Build an image handle and **EFI_LOADED_IMAGE_PROTOCOL** instance for the DXE Foundation itself and add it to the handle database.
- If the HOB list is not in a suitable location in memory, then relocate the HOB list to a more suitable location.
- Add the DXE Services Table to the UEFI Configuration Table.
- Add the HOB list to the UEFI Configuration Table.
- Create a notification event for each of the DXE Architectural Protocols. These events will be signaled when a DXE driver installs a DXE Architectural Protocol in the handle database. The DXE Foundation must have a notification function associated with each of these events, so the full complement of UEFI Boot Services, UEFI Runtime Services, and DXE Services can be produced. Each of the notification functions should compute the 32-bit CRC of the UEFI Boot Services Table, UEFI Runtime Services Table, and the DXE Services Table if the **CalculateCrc32()** Boot Services is available.
- Initialize the Decompress Protocol driver that must be available before the DXE Dispatcher can process compressed sections.
- Produce firmware volume handles for the one or more firmware volumes that are described in the HOB list.

Once these tasks have been completed, the DXE Foundation is ready to load and execute DXE drivers stored in firmware volumes. This execution is done by handing control to the DXE Dispatcher. Once the DXE Dispatcher has finished dispatching all the DXE drivers that it can, control is then passed to the BDS Architectural Protocol. If for some reason, any of the DXE Architectural Protocols have not been produced by the DXE drivers, then the system is in an unusable state and the DXE Foundation must halt. Otherwise, control is handed to the BDS Architectural Protocol. The BDS Architectural Protocol is responsible for transferring control to an operating system or system utility.

9.6 DXE Foundation Entry Point

9.6.1 DXE_ENTRY_POINT

The only parameter passed to the DXE Foundation is a pointer to the HOB list. The DXE Foundation and all the DXE drivers must treat the HOB list as read-only data.

The function **DXE_ENTRY_POINT** is the main entry point to the DXE Foundation.

DXE_ENTRY_POINT

Summary

This function is the main entry point to the DXE Foundation.

Prototype

```
typedef
VOID
(EFIAPI *DXE_ENTRY_POINT) (
  IN  CONST VOID  *HobStart
  );
```

Parameters

HobStart

A pointer to the HOB list.

Description

This function is the entry point to the DXE Foundation. The PEI phase, which executes just before DXE, is responsible for loading and invoking the DXE Foundation in system memory. The only parameter that is passed to the DXE Foundation is *HobStart*. This parameter is a pointer to the HOB list that describes the system state at the hand-off to the DXE Foundation. At a minimum, this system state must include the following:

- PHIT HOB
- CPU HOB
- Description of system memory
- Description of one or more firmware volumes

The DXE Foundation is also guaranteed that only one processor is running and that the processor is running with interrupts disabled. The implementation of the DXE Foundation must not make any assumptions about where the DXE Foundation will be loaded or where the stack is located. In general, the DXE Foundation should make as few assumptions about the state of the system as possible. This lack of assumptions will allow the DXE Foundation to be portable to the widest variety of system architectures.

9.7 Dependencies

9.7.1 UEFI Boot Services Dependencies

Table 15 lists all the UEFI Boot Services and the components upon which each of these services depend. The topics that follow describe what responsibilities the DXE Foundation has in producing the services that depend on the presence of DXE Architectural Protocols.

Table 15. Boot Service Dependencies

Name	Dependency
CreateEvent	HOB list
CloseEvent	HOB list
SignalEvent	HOB list
WaitForEvent	HOB list
CheckEvent	HOB list
SetTimer	Timer Architectural Protocol
RaiseTPL	CPU Architectural Protocol
RestoreTPL	CPU Architectural Protocol
AllocatePages	HOB list
FreePages	HOB list
GetMemoryMap	HOB list
AllocatePool	HOB list
FreePool	HOB list
InstallProtocolInterface	HOB list
UninstallProtocolInterface	HOB list
ReinstallProtocolInterface	HOB list
RegisterProtocolNotify	HOB list
LocateHandle	HOB list
HandleProtocol	HOB list
LocateDevicePath	HOB list
OpenProtocol	HOB list
CloseProtocol	HOB list
OpenProtocolInformation	HOB list
ConnectController	HOB list
DisconnectController	HOB list
ProtocolsPerHandle	HOB list
LocateHandleBuffer	HOB list
LocateProtocol	HOB list
InstallMultipleProtocolInterfaces	HOB list
UninstallMultipleProtocolInterfaces	HOB list
LoadImage	HOB list

Name	Dependency
StartImage	HOB list
UnloadImage	HOB list
EFI_IMAGE_ENTRY_POINT	HOB list
Exit	HOB list
ExitBootServices	HOB list
SetWatchDogTimer	Watchdog Architectural Protocol
Stall	Metronome Architectural Protocol Timer Architectural Protocol
CopyMem	HOB list
SetMem	HOB list
GetNextMonotonicCount	Monotonic Counter Architectural Protocol
InstallConfigurationTable	HOB list
CalculateCrc32	Runtime Architectural Protocol

9.7.1.1 SetTimer()

When the DXE Foundation is notified that the **EFI_TIMER_ARCH_PROTOCOL** has been installed, then the Boot Service **SetTimer()** can be made available. The DXE Foundation can use the services of the **EFI_TIMER_ARCH_PROTOCOL** to initialize and hook a heartbeat timer interrupt for the DXE Foundation. The DXE Foundation can use this heartbeat timer interrupt to determine when to signal on-shot and periodic timer events. This service may be called before the **EFI_TIMER_ARCH_PROTOCOL** is installed. However, since a heartbeat timer is not running yet, time is essentially frozen at zero. This means that no periodic or one-shot timer events will fire until the **EFI_TIMER_ARCH_PROTOCOL** is installed.

9.7.1.2 RaiseTPL()

The DXE Foundation must produce the Boot Service **RaiseTPL()** when the memory-based services are initialized. The DXE Foundation is guaranteed to be handed control of the platform with interrupts disabled. Until the DXE Foundation installs a heartbeat timer interrupt and turns on interrupts, this Boot Service can be a very simple function that always succeeds. When the DXE Foundation is notified that the **EFI_CPU_ARCH_PROTOCOL** has been installed, then the full version of the Boot Service **RaiseTPL()** can be made available. When an attempt is made to raise the TPL level to **EFI_TPL_HIGH_LEVEL** or higher, then the DXE Foundation should use the services of the **EFI_CPU_ARCH_PROTOCOL** to disable interrupts.

9.7.1.3 RestoreTPL()

The DXE Foundation must produce the Boot Service **RestoreTPL()** when the memory-based services are initialized. The DXE Foundation is guaranteed to be handed control of the platform with interrupts disabled. Until the DXE Foundation installs a heartbeat timer interrupt and turns on interrupts, this Boot Service can be a very simple function that always succeeds. When the DXE Foundation is notified that the **EFI_CPU_ARCH_PROTOCOL** has been installed, then the full version of the Boot Service **RestoreTPL()** can be made available. When an attempt is made to restore the TPL level to level below **EFI_TPL_HIGH_LEVEL**, then the DXE Foundation should use the services of the **EFI_CPU_ARCH_PROTOCOL** to enable interrupts.

9.7.1.4 SetWatchdogTimer()

When the DXE Foundation is notified that the **EFI_WATCHDOG_ARCH_PROTOCOL** has been installed, then the Boot Service **SetWatchdogTimer()** can be made available. The DXE Foundation can use the services of the **EFI_WATCHDOG_TIMER_ARCH_PROTOCOL** to set the amount of time before the system's watchdog timer will expire.

9.7.1.5 Stall()

When the DXE Foundation is notified that the **EFI_METRONOME_ARCH_PROTOCOL** has been installed, the DXE Foundation can produce a very simple version of the Boot Service **Stall()**. The granularity of the Boot Service **Stall()** will be based on the period of the **EFI_METRONOME_ARCH_PROTOCOL**.

When the DXE Foundation is notified that the **EFI_TIMER_ARCH_PROTOCOL** has been installed, the DXE Foundation can possibly produce a more accurate version of the Boot Service **Stall()**. This all depends on the periods of the **EFI_METRONOME_ARCH_PROTOCOL** and the period of the **EFI_TIMER_ARCH_PROTOCOL**. The DXE Foundation should produce the Boot Service **Stall()** using the most accurate time base available.

9.7.1.6 GetNextMonotonicCount()

When the DXE Foundation is notified that the **EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL** has been installed, then the Boot Service **GetNextMonotonicCount()** is available. The DXE driver that produces the **EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL** is responsible for directly updating the *GetNextMonotonicCount* field of the UEFI Boot Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Boot Services Table.

9.7.1.7 CalculateCrc32()

When the DXE Foundation is notified that the **EFI_RUNTIME_ARCH_PROTOCOL** has been installed, then the Boot Service **CalculateCrc32()** is available. The DXE driver that produces the **EFI_RUNTIME_ARCH_PROTOCOL** is responsible for directly updating the *CalculateCrc32* field of the UEFI Boot Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Boot Services Table.

9.7.2 UEFI Runtime Services Dependencies

Table 16 lists all the UEFI Runtime Services and the components upon which each of these services depend. The topics that follow describe what responsibilities the DXE Foundation has in producing the services that depend on the presence of DXE Architectural Protocols.

Table 16. Runtime Service Dependencies

Name	Dependency
GetVariable	Variable Architectural Protocol
GetNextVariableName	Variable Architectural Protocol
SetVariable	Variable Architectural Protocol / Variable Write Architectural Protocol
GetTime	Real Time Clock Architectural Protocol
SetTime	Real Time Clock Architectural Protocol

GetWakeupTime	Real Time Clock Architectural Protocol
SetWakeupTime	Real Time Clock Architectural Protocol
SetVirtualAddressMap	Runtime Architectural Protocol
ConvertPointer	Runtime Architectural Protocol
ResetSystem	Reset Architectural Protocol
GetNextHighMonotonicCount	Monotonic Counter Architectural Protocol
UpdateCapsule	Capsule Header Protocol
QueryCapsuleCapabilities	Capsule Header Protocol

9.7.2.1 GetVariable()

When the DXE Foundation is notified that the **EFI_VARIABLE_ARCH_PROTOCOL** has been installed, then the Runtime Service **GetVariable()** is available. The DXE driver that produces the **EFI_VARIABLE_ARCH_PROTOCOL** is responsible for directly updating the *GetVariable* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.2 GetNextVariableName()

When the DXE Foundation is notified that the **EFI_VARIABLE_ARCH_PROTOCOL** has been installed, then the Runtime Service **GetNextVariableName()** is available. The DXE driver that produces the **EFI_VARIABLE_ARCH_PROTOCOL** is responsible for directly updating the *GetNextVariableName* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.3 SetVariable()

When the DXE Foundation is notified that the **EFI_VARIABLE_ARCH_PROTOCOL** has been installed, then the Runtime Service **SetVariable()** is available. The DXE driver that produces the **EFI_VARIABLE_ARCH_PROTOCOL** is responsible for directly updating the *SetVariable* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table. The **EFI_VARIABLE_ARCH_PROTOCOL** is required to provide read-only access to all environment variables and write access to volatile environment variables.

When the DXE Foundation is notified that the **EFI_VARIABLE_WRITE_ARCH_PROTOCOL** has been installed, then write access to nonvolatile environment variables will also be available. If an attempt is made to call this function for a nonvolatile environment variable prior to the installation of **EFI_VARIABLE_WRITE_ARCH_PROTOCOL**, then **EFI_NOT_AVAILABLE_YET** must be returned. This allows for flexibility in the design and implementation of the variables services in a platform such that read access to environment variables can be provided very early in the DXE phase and write access to nonvolatile environment variables can be provided later in the DXE phase.

9.7.2.4 GetTime()

When the DXE Foundation is notified that the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** has been installed, then the Runtime Service **GetTime()** is available. The DXE driver that produces the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** is responsible for directly updating the

GetTime field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.5 SetTime()

When the DXE Foundation is notified that the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** has been installed, then the Runtime Service **SetTime()** is available. The DXE driver that produces the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** is responsible for directly updating the *SetTime* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.6 GetWakeupTime()

When the DXE Foundation is notified that the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** has been installed, then the Runtime Service **GetWakeupTime()** is available. The DXE driver that produces the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** is responsible for directly updating the *GetWakeupTime* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.7 SetWakeupTime()

When the DXE Foundation is notified that the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** has been installed, then the Runtime Service **SetWakeupTime()** is available. The DXE driver that produces the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** is responsible for directly updating the *SetWakeupTime* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.8 SetVirtualAddressMap()

When the DXE Foundation is notified that the **EFI_RUNTIME_ARCH_PROTOCOL** has been installed, then the Runtime Service **SetVirtualAddressMap()** is available. The DXE driver that produces the **EFI_RUNTIME_ARCH_PROTOCOL** is responsible for directly updating the *SetVirtualAddressMap* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.9 ConvertPointer()

When the DXE Foundation is notified that the **EFI_RUNTIME_ARCH_PROTOCOL** has been installed, then the Runtime Service **ConvertPointer()** is available. The DXE driver that produces the **EFI_RUNTIME_ARCH_PROTOCOL** is responsible for directly updating the *ConvertPointer* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.10 ResetSystem()

When the DXE Foundation is notified that the **EFI_RESET_ARCH_PROTOCOL** has been installed, then the Runtime Service **ResetSystem()** is available. The DXE driver that produces the **EFI_RESET_ARCH_PROTOCOL** is responsible for directly updating the *Reset* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.11 GetNextHighMonotonicCount()

When the DXE Foundation is notified that the **EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL** has been installed, then the Runtime Service **GetNextHighMonotonicCount()** is available. The DXE driver that produces the **EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL** is responsible for directly updating the *GetNextHighMonotonicCount* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.3 DXE Services Dependencies

Table 17 lists all the DXE Services and the components upon which each of these services depend. The topics that follow describe what responsibilities the DXE Foundation has in producing the services that depend on the presence of DXE Architectural Protocols.

Table 17. DXE Service Dependencies

Name	Dependency
AddMemorySpace	HOB list
AllocateMemorySpace	HOB list
FreeMemorySpace	HOB list
RemoveMemorySpace	HOB list
GetMemorySpaceDescriptor	CPU Architectural Protocol
SetMemorySpaceAttributes	CPU Architectural Protocol
GetMemorySpaceMap	CPU Architectural Protocol
AddIoSpace	HOB list
AllocateIoSpace	HOB list
FreeIoSpace	HOB list
RemoveIoSpace	HOB list
GetIoSpaceDescriptor	HOB list
GetIoSpaceMap	HOB list
Schedule	HOB list

9.7.3.1 GetMemorySpaceDescriptor()

When the DXE Foundation is notified that the **EFI_CPU_ARCH_PROTOCOL** has been installed, then the DXE Service **GetMemorySpaceDescriptor()** is fully functional. This function is made available when the memory-based services are initialized. However, the *Attributes* field of the **EFI_GCD_MEMORY_SPACE_DESCRIPTOR** is not valid until the **EFI_CPU_ARCH_PROTOCOL** is installed.

9.7.3.2 SetMemorySpaceAttributes()

When the DXE Foundation is notified that the **EFI_CPU_ARCH_PROTOCOL** has been installed, then the DXE Service **SetMemorySpaceAttributes()** can be made available. The DXE Foundation can then use the **SetMemoryAttributes()** service of the

EFI_CPU_ARCH_PROTOCOL to implement the DXE Service **SetMemorySpaceAttributes()**.

9.7.3.3 GetMemorySpaceMap()

When the DXE Foundation is notified that the **EFI_CPU_ARCH_PROTOCOL** has been installed, then the DXE Service **GetMemorySpaceMap()** is fully functional. This function is made available when the memory-based services are initialized. However, the *Attributes* field of the array of **EFI_GCD_MEMORY_SPACE_DESCRIPTOR**s is not valid until the **EFI_CPU_ARCH_PROTOCOL** is installed.

9.8 HOB Translations

9.8.1 HOB Translations Overview

The following topics describe how the DXE Foundation should interpret the contents of the HOB list to initialize the GCD memory space map, GCD I/O space map, and UEFI memory map. After all of the HOBs have been parsed, the Boot Service **GetMemoryMap()** and the DXE Services **GetMemorySpaceMap()** and **GetIoSpaceMap()** should reflect the memory resources, I/O resources, and logical memory allocations described in the HOB list.

See the *Platform Initialization Hand-Off Block Specification* for detailed information on HOBs.

9.8.2 PHIT HOB

The Phase Handoff Information Table (PHIT) HOB describes a region of tested system memory. This region of memory contains the following:

- HOB list
- Some amount of free memory
- Potentially some logical memory allocations

The PHIT HOB is used by the DXE Foundation to determine the size of the HOB list so that the DXE Foundation can relocate the HOB list to a new location in system memory. The base address of the HOB list is passed to the DXE Foundation in the parameter *HobStart*, and the PHIT HOB field *EfiFreeMemoryBottom* specifies the end of the HOB list.

Since the PHIT HOB may contain some of amount of free memory, the DXE Foundation may use this free memory region in its early initialization phase until the full complement of UEFI memory services are available.

See the *Platform Initialization Hand-Off Block Specification* for the definition of this HOB type.

9.8.3 CPU HOB

The CPU HOB contains the field *SizeOfMemorySpaceMap*. This field is used to initialize the GCD memory space map. The *SizeOfMemorySpaceMap* field defines the number of address bits that the processor can use to address memory resources. The DXE Foundation must create the primordial GCD memory space map entry of type **EfiGcdMemoryTypeNonExistent** for the

region from 0 to **(1 << *SizeOfMemorySpaceMap*)**. All future GCD memory space operations must be performed within this memory region.

The CPU HOB also contains the field *SizeOfIoSpaceMap*. This field is used to initialize the GCD I/O space map. The *SizeOfIoSpaceMap* field defines the number of address bits that the processor can use to address I/O resources. The DXE Foundation must create the primordial GCD I/O space map entry of type **EfiGcdIoTypeNonExistent** for the region from 0 to (**1 <<** *SizeOfIoSpaceMap*). All future GCD I/O space operations must be performed within this I/O region.

See the *Platform Initialization Hand-Off Block Specification* for the definition of this HOB type.

9.8.4 Resource Descriptor HOBs

The DXE Foundation must traverse the HOB list looking for Resource Descriptor HOBs. These HOBs describe memory and I/O resources that are visible to the processor. All of the resource ranges described in these HOBs must fall in the memory and I/O ranges initialized in the GCD maps based on the contents of the CPU HOB. The DXE Foundation will use the DXE Services **AddMemorySpace()** and **AddIoSpace()** to register these memory and I/O resources in the GCD maps.

The *Owner* field of the Resource Descriptor HOB is ignored by the DXE Foundation. The *ResourceType* field and *ResourceAttribute* fields are used to determine the GCD memory type or GCD I/O type of the resource. The table below shows this mapping. The resource range is specified by the *PhysicalStart* and *ResourceLength* fields of the Resource Descriptor HOB.

The *ResourceAttribute* field also contains the caching capabilities of memory regions. If a memory region is being added to the GCD memory space map, then the *ResourceAttribute* field will be used to initialize the supported caching capabilities. The *ResourceAttribute* field is also be used to further qualify memory regions. For example, a system memory region cannot be added to the UEFI memory map if it is read protected. However, it is legal to add a firmware device memory region that is write-protected if the firmware device is a ROM.

See the *Platform Initialization Hand-Off Block Specification* for the definition of this HOB type.

Table 18. Resource Descriptor HOB to GCD Type Mapping

Resource Descriptor HOB		**GCD Map**	
Resource Type	Attributes	Memory Type	I/O Type
System Memory	Present	Reserved	
System Memory	Present AND Initialized	Reserved	
System Memory	Present AND Initialized AND Tested	System Memory	
Memory-Mapped I/O		Memory Mapped I/O	
Firmware Device		Memory Mapped I/O	
Memory-Mapped I/O Port		Reserved	
Memory Reserved		Reserved	
I/O			I/O
I/O Reserved			Reserved

9.8.5 Firmware Volume HOBs

The DXE Foundation must traverse the HOB list for Firmware Volume HOBs. There are two types of firmware volume HOBs:

- **EFI_HOB_FIRMWARE_VOLUME**, which describes PI Firmware Volumes.
- **EFI_HOB_FIRMWARE_VOLUME2** which describes PI Firmware Volumes which came from a firmware file within a firmware volume.

When the DXE Foundation discovers a Firmware Volume HOB, the DXE Dispatcher verifies that the firmware volume has not been previously processed. Then a new handle must be created in the handle database, and the **EFI_FIRMWARE_VOLUME2_PROTOCOL** must be installed on that handle. The *BaseAddress* and *Length* fields of the Firmware Volume HOB specific the memory range that the firmware volume consumes. The DXE Service **AllocateMemorySpace()** is used to allocate the memory regions described in the Firmware Volume HOBs to the DXE Foundation. The UEFI Boot Service **InstallProtocolInterface()** is used to create new handles and install protocol interfaces.

See the *Platform Initialization Specification*, Volume 3, for code definitions concerning Hand-Off Blocks, the Firmware Volume Block Protocol and the Firmware Volume Protocol.

9.8.6 Memory Allocation HOBs

Memory Allocation HOBs describe logical memory allocations that occurred prior to the DXE phase. The DXE Foundation must parse the HOB list for this HOB type. When a HOB of this type is discovered, the GCD memory space map must be updated with a call to the DXE Service **AllocateMemorySpace()**. In addition, the UEFI memory map must be updated with logical allocation described by the *MemoryType*, *MemoryBaseAddress*, and *MemoryLength* fields of the Memory Allocation HOB.

Once the DXE Foundation has parsed all of the Memory Allocation HOBs, all of the unallocated system memory regions in the GCD memory space map must be allocated to the DXE Foundation with the DXE Service **AllocateMemorySpace()**. In addition, those same memory regions must be added to the UEFI memory map so those memory regions can be allocated and freed using the Boot Services **AllocatePages()**, **AllocatePool()**, **FreePages()**, and **FreePool()**.

See the *Platform Initialization Hand-Off Block Specification* for the definition of this HOB type.

9.8.7 GUID Extension HOBs

The DXE Foundation does not require any GUID Extension HOBs. Implementations of the DXE Foundation may use GUID Extension HOBs but shall not require them in order to function correctly. GUID Extension HOBs contain private or implementation-specific data that is being passed from the previous execution phase to a specific DXE driver. DXE drivers may choose to parse the HOB list for GUID Extension HOBs.

See the *Platform Initialization Hand-Off Block Specification* for the definition of this HOB type.

10
DXE Dispatcher

10.1 Introduction

After the DXE Foundation is initialized, control is handed to the DXE Dispatcher. The DXE Dispatcher examines every firmware volume that is present in the system. Firmware volumes are either declared by HOBs, or they are declared by DXE drivers. For the DXE Dispatcher to run, at least one firmware volume must be declared by a HOB.

The DXE Dispatcher is responsible for loading and invoking DXE drivers found in firmware volumes. Some DXE drivers may depend on the services produced by other DXE drivers, so the DXE Dispatcher is also required to execute the DXE drivers in the correct order. The DXE drivers may also be produced by a variety of different vendors, so the DXE drivers must describe the services they depend upon. The DXE dispatcher must evaluate these dependencies to determine a valid order to execute the DXE drivers. Some vendors may wish to specify a fixed execution order for some or all of the DXE drivers in a firmware volume, so the DXE dispatcher must support this requirement.

The DXE Dispatcher will ignore file types that it does not recognize.

In addition, the DXE Dispatcher must support the ability to load “emergency patch” drivers. These drivers would be added to the firmware volume to address an issue that was not know at the time the original firmware was built. These DXE drivers would be loaded just before or just after an existing DXE driver.

Finally, the DXE Dispatcher must be flexible enough to support a variety of platform specific security policies for loading and executing DXE drivers from firmware volumes. Some platforms may choose to run DXE drivers with no security checks, and others may choose to check the validity of a firmware volume before it is used, and other may choose to check the validity of every DXE driver in a firmware volume before it is executed.

10.2 Requirements

The DXE Dispatcher must meet the following requirement:

- **Support fixed execution order of DXE drivers.** This fixed execution order is specified in an *a priori* file in the firmware volume.
- **Determine DXE driver execution order based on each driver’s dependencies.** A DXE driver that is stored in a firmware volume may optionally contain a dependency expression section. This section specifies the protocols that the DXE driver requires to execute.
- **Support “emergency patch” DXE drivers.** The dependency expressions are flexible enough to describe the protocols that a DXE drivers may require. In addition, the dependency expression can declare that the DXE driver is to be loaded and executed immediately before or immediately after a different DXE driver.

- **Support platform specific security policies for DXE driver execution.** The DXE Dispatcher is required to use the services of the Security Architecture Protocol every time a firmware volume is discovered and every time a DXE driver is loaded.

When a new firmware volume is discovered, it is first authenticated with the Security Architectural Protocol. The Security Architectural Protocol provides the platform-specific policy for validating all firmware volumes. Then, a search is made for the *a priori* file. The *a priori* file has a fixed file name, and it contains the list of DXE drivers that should be loaded and executed first. There can be at most one *a priori* file per firmware volume, and it is legal to have zero *a priori* files in a firmware volume. Once the DXE drivers from the *a priori* file have been loaded and executed, the dependency expressions of the remaining DXE drivers in the firmware volumes are evaluated to determine the order that they will be loaded and executed. The *a priori* file provides a strongly ordered list of DXE drivers that are not required to use dependency expressions. The dependency expressions provide a weakly ordered execution of the remaining DXE drivers.

The DXE Dispatcher loads the image using **`LoadImage()`** with the *`FilePath`* parameter pointing ot the firmware volume from which the image is located.

Before each DXE driver is executed, it must be authenticated through the Security Architectural Protocol. The Security Architectural Protocol provides the platform-specific policy for validating all DXE drivers.

Control is transferred from the DXE Dispatcher to the BDS Architectural Protocol after the DXE drivers in the *a priori* file and all the DXE drivers whose dependency expressions evaluate to **`TRUE`** have been loaded and executed. The BDS Architectural Protocol is responsible for establishing the console devices and attempting the boot of operating systems. As the console devices are established and access to boot devices is established, additional firmware volumes may be discovered. If the BDS Architectural Protocol is unable to start a console device or gain access to a boot device, it will reinvoke the DXE Dispatcher. This will allow the DXE Dispatcher to load and execute DXE drivers from firmware volumes that have been discovered since the last time the DXE Dispatcher was invoked. Once the DXE Dispatcher has loaded and executed all the DXE drivers it can, control is once again returned to the BDS Architectural Protocol to continue the OS boot process.

10.3 The A Priori File

The *a priori* file is a special file that may be present in a firmware volume. The *a priori* file format described herein must be supported if the DXE Foundation implementation also supports 3rd party firmware volumes. The rule is that there may be at most one *a priori* file per firmware volume present in a platform. The *a priori* file has a known GUID file name, so the DXE Dispatcher can always find the *a priori* file if it is present. Every time the DXE Dispatcher discovers a firmware volume, it first looks for the *a priori* file. The *a priori* file contains the list of DXE drivers from that firmware volume that should be loaded and executed before any other DXE drivers are discovered. The DXE drivers listed in the *a priori* file are executed in the order that they appear. If any of those DXE drivers have an associated dependency expression, then those dependency expressions are ignored. The *a priori* file provides a deterministic execution order of DXE drivers. DXE drivers that are executed solely based on their dependency expression are weakly ordered. This means that the execution order is not completely deterministic between boots or between platforms. There are cases where a deterministic execution order is required. One example would be to list the DXE

drivers required to debug the rest of the DXE phase in the *a priori* file. These DXE drivers that provide debug services may have been loaded much later if only their dependency expressions were considered. By loading them earlier, more of the DXE Foundation and DXE drivers can be debugged. Another example is to use the *a priori* file to eliminate the need for dependency expressions. Some embedded platforms may only require a few DXE drivers with a highly deterministic execution order. The *a priori* file can provide this ordering, and none of the DXE drivers would require dependency expressions. The dependency expressions do have some amount of size overhead, so this method may reduce the size of firmware images. The main purpose of the *a priori* file is to provide a greater degree of flexibility in the firmware design of a platform.

See the next topic for the GUID definition of the *a priori* file, which is the file name that is stored in a firmware volume.

The *a priori* file contains the file names of DXE drivers that are stored in the same firmware volume as the *a priori* file. File names in firmware volumes are GUIDs, so the *a priori* file is simply a list of byte-packed values of type **EFI_GUID**. Type **EFI_GUID** is defined in the UEFI 2.0 specification. The DXE Dispatcher reads the list of **EFI_GUID**s from the *a priori* file. Each **EFI_GUID** is used to load and execute the DXE driver with that GUID file name. If the DXE driver specified by the GUID file name is not found in the firmware volume, then the file is skipped. If the *a priori* file is not en even multiple of **EFI_GUID**s in length, then the DXE driver specified by the last **EFI_GUID** in the *a priori* file is skipped.

After all of the DXE drivers listed in the *a priori* file have been loaded and executed, the DXE Dispatcher searches the firmware volume for any additional DXE drivers and executed them according to their dependency expressions.

EFI_APRIORI_GUID

The following GUID definition is the file name of the *a priori* file that is stored in a firmware volume. This file must be of type **EFI_FV_FILETYPE_FREEFORM** and must contain a single section of type **EFI_SECTION_RAW**. For details on firmware volumes, firmware file types, and firmware file section types, see the *Platform Initialization Specification*, Volume 3 .

GUID

```
#define EFI_APRIORI_GUID  \

{0xfc510ee7,0xffdc,0x11d4,0xbd,0x41,0x0,0x80,0xc7,0x3c,0x88,0x81
}
```

10.4 Firmware Volume Image Files

The PEI Dispatcher will ignore files with the section type of **EFI_SECTION_FIRMWARE_VOLUME_IMAGE**.

For DXE, while processing a firmware volume, if a file of type **EFI_FV_FIRMWARE_VOLUME_IMAGE** is found, the DXE Dispatcher will check whether information about this firmware volume image file was already described in an **EFI_FIRMWARE_VOLUME_HOB2**. If it was, then the file is ignored.

Otherwise, the DXE Dispatcher will search the file for a section with the type **EFI_SECTION_DXE_DEPEX**, and if found, evaluate the expression against the presently installed entries in the protocol database.

If the file has both a dependency expression that evaluates to **TRUE** (or no dependency expression section) and the file is not already described by an **EFI_FIRMWARE_VOLUME_HOB2**, then the DXE Dispatcher will search the file for a section with the type **EFI_SECTION_FIRMWARE_VOLUME_IMAGE**, copy its contents into memory, create a handle and install the **EFI_FIRMWMARE_VOLUME2_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL** on the handle.

10.5 Dependency Expressions

10.6 Dependency Expressions Overview

A DXE driver is stored in a firmware volume as a file with one or more sections. One of the sections must be a PE32+ image. If a DXE driver has a dependency expression, then it is stored in a *dependency section*. A DXE driver may contain additional sections for compression and security wrappers. The DXE Dispatcher can identify the DXE drivers by their file type. In addition, the DXE Dispatcher can look up the dependency expression for a DXE driver by looking for a dependency section in a DXE driver file. The dependency section contains a section header followed by the actual dependency expression that is composed of a packed byte stream of opcodes and operands.

Dependency expressions stored in dependency sections are designed to be small to conserve space. In addition, they are designed to be simple and quick to evaluate to reduce execution overhead. These two goals are met by designing a small, stack based, instruction set to encode the dependency expressions. The DXE Dispatcher must implement an interpreter for this instruction set in order to evaluate dependency expressions. The instruction set is defined in the following topics.

See "Dependency Expression Grammar" on page 193 for an example BNF grammar for a dependency expression compiler. There are many possible methods of specifying the dependency expression for a DXE driver. Dependency Expression Grammar demonstrates one possible design for a tool that can be used to help build DXE driver images.

10.7 Dependency Expression Instruction Set

The following topics describe each of the dependency expression opcodes in detail. Information includes a description of the instruction functionality, binary encoding, and any limitations or unique behaviors of the instruction.

Several of the opcodes require a GUID operand. The GUID operand is a 16-byte value that matches the type **EFI_GUID** that is described in the UEFI 2.0 specification. These GUIDs represent protocols that are produced by DXE drivers and the file names of DXE drivers stored in firmware volumes. A dependency expression is a packed byte stream of opcodes and operands. As a result, some of the GUID operands will not be aligned on natural boundaries. Care must be taken on processor architectures that do allow unaligned accesses.

The dependency expression is stored in a packed byte stream using postfix notation. As a dependency expression is evaluated, the operands are pushed onto a stack. Operands are popped off the stack to perform an operation. After the last operation is performed, the value on the top of the stack represents the evaluation of the entire dependency expression. If a push operation causes a stack overflow, then the entire dependency expression evaluates to **FALSE**. If a pop operation causes a stack underflow, then the entire dependency expression evaluates to **FALSE**. Reasonable implementations of a dependency expression evaluator should not make arbitrary assumptions about the maximum stack size it will support. Instead, it should be designed to grow the dependency expression stack as required. In addition, DXE drivers that contain dependency expressions should make an effort to keep their dependency expressions as small as possible to help reduce the size of the DXE driver.

All opcodes are 8-bit values, and if an invalid opcode is encountered, then the entire dependency expression evaluates to **FALSE**.

If an END opcode is not present in a dependency expression, then the entire dependency expression evaluates to **FALSE**.

If an instruction encoding extends beyond the end of the dependency section, then the entire dependency expression evaluates to **FALSE**.

The final evaluation of the dependency expression results in either a **TRUE** or **FALSE** result.

Table 19 is a summary of the opcodes that are used to build dependency expressions. The following topics describe each of these instructions in detail.

Table 19. Dependency Expression Opcode Summary

Opcode	Description
0x00	BEFORE <File Name GUID>
0x01	AFTER <File Name GUID>
0x02	PUSH <Protocol GUID>
0x03	AND
0x04	OR
0x05	NOT
0x06	TRUE
0x07	FALSE
0x08	END
0x09	SOR

BEFORE

Syntax

```
BEFORE <File Name GUID>
```

Description

This opcode tells the DXE Dispatcher that the DXE driver that is associated with this dependency expression must be dispatched just before the DXE driver with the file name specified by **GUID**. This means that as soon as the dependency expression for the DXE driver specified by **GUID** evaluates to **TRUE**, then this DXE driver must be placed in the dispatch queue just before the DXE driver with the file name specified by **GUID**.

Operation

None.

Table 20 defines the **BEFORE** instruction encoding.

Table 20. BEFORE Instruction Encoding

Byte	Description
0	0x00
1..16	A 16-byte GUID that represents the file name of a different DXE driver. The format is the same as type **EFI_GUID**.

Behaviors and Restrictions

If this opcode is present in a dependency expression, it must be the first and only opcode in the expression. If is appears in any other location in the dependency expression, then the dependency expression is evaluated to **FALSE**.

AFTER

Syntax

```
AFTER <File Name GUID>
```

Description

This opcode tells the DXE Dispatcher that the DXE driver that is associated with this dependency expression must be dispatched just after the DXE driver with the file name specified by **GUID**. This means that as soon as the dependency expression for the DXE driver specified by **GUID** evaluates to **TRUE**, then this DXE driver must be placed in the dispatch queue just after the DXE Driver with the file name specified by **GUID**.

Operation

None.

Table 21 defines the **AFTER** instruction encoding.

Table 21. AFTER Instruction Encoding

Byte	Description
0	0x01
1..16	A 16-byte GUID that represents the file name of a different DXE driver. The format is the same as type **EFI_GUID**.

Behaviors and Restrictions

If this opcode is present in a dependency expression, it must be the first and only opcode in the expression. If is appears in any other location in the dependency expression, then the dependency expression is evaluated to **FALSE**.

PUSH

Syntax

```
PUSH <Protocol GUID>
```

Description

Pushes a Boolean value onto the stack. If the GUID is present in the handle database, then a **TRUE** is pushed onto the stack. If the GUID is not present in the handle database, then a **FALSE** is pushed onto the stack. The test for the GUID in the handle database may be performed with the Boot Service **LocateProtocol()**.

Operation

```
Status = gBS->LocateProtocol (GUID, NULL, &Interface);
if (EFI_ERROR (Status)) {
  PUSH FALSE;
} Else {
  PUSH TRUE;
}
```

Table 22 defines the **PUSH** instruction encoding.

Table 22. PUSH Instruction Encoding

Byte	Description
0	0x02
1..16	A 16-byte GUID that represents a protocol that is produced by a different DXE driver. The format is the same at type **EFI_GUID.**

Behaviors and Restrictions

None.

AND

Syntax

```
AND
```

Description

Pops two Boolean operands off the stack, performs a Boolean AND operation between the two operands, and pushes the result back onto the stack.

Operation

```
Operand1 <= POP Boolean stack element
Operand2 <= POP Boolean stack element
Result <= Operand1 AND Operand2
PUSH Result
```

Table 23 defines the **AND** instruction encoding.

Table 23. AND Instruction Encoding

Byte	Description
0	0x03.

Behaviors and Restrictions

None.

OR

Syntax

```
OR
```

Description

Pops two Boolean operands off the stack, performs a Boolean OR operation between the two operands, and pushes the result back onto the stack.

Operation

```
Operand1 <= POP Boolean stack element
Operand2 <= POP Boolean stack element
Result <= Operand1 OR Operand2
PUSH Result
```

Table 24 defines the **OR** instruction encoding.

Table 24. OR Instruction Encoding

Byte	Description
0	0x04.

Behaviors and Restrictions

None.

NOT

Syntax

```
NOT
```

Description

Pops a Boolean operands off the stack, performs a Boolean NOT operation on the operand, and pushes the result back onto the stack.

Operation

```
Operand <= POP Boolean stack element
Result <= NOT Operand1
PUSH Result
```

Table 25 defines the **NOT** instruction encoding.

Table 25. NOT Instruction Encoding

Byte	Description
0	0x05.

Behaviors and Restrictions

None.

TRUE

Syntax

TRUE

Description

Pushes a Boolean **TRUE** onto the stack.

Operation

PUSH **TRUE**

Table 26 defines the **TRUE** instruction encoding.

Table 26. TRUE Instruction Encoding

Byte	Description
0	0x06.

Behaviors and Restrictions

None.

FALSE

Syntax

FALSE

Description

Pushes a Boolean **FALSE** onto the stack.

Operation

PUSH FALSE

Table 27 defines the **FALSE** instruction encoding.

Table 27. FALSE Instruction Encoding

Byte	Description
0	0x07.

Behaviors and Restrictions

None.

END

Syntax

```
END
```

Description

Pops the final result of the dependency expression evaluation off the stack and exits the dependency expression evaluator.

Operation

```
POP Result
RETURN Result
```

Table 28 defines the **END** instruction encoding.

Table 28. END Instruction Encoding

Byte	Description
0	0x08.

Behaviors and Restrictions

This opcode must be the last one in a dependency expression.

SOR

Syntax

SOR

Description

Indicates that the DXE driver is to remain on the Schedule on Request (SOR) queue until the DXE Service **Schedule()** is called for this DXE. The dependency expression evaluator treats this operation like a No Operation (**NOP**).

Operation

None.

Table 29 defines the **SOR** instruction encoding.

Table 29. SOR Instruction Encoding

Byte	Description
0	0x09.

Behaviors and Restrictions

- If this instruction is present in a dependency expression, it must be the first instruction in the expression. If it appears in any other location in the dependency expression, then the dependency expression is evaluated to **FALSE**.
- This instruction must be followed by a valid dependency expression. If this instruction is the last instruction or it is followed immediately by an **END** instruction, then the dependency expression is evaluated to **FALSE**.

10.8 Dependency Expression with No Dependencies

A DXE driver that does not have any dependencies must have a dependency expression that evaluates to **TRUE** with no dependencies on any protocol GUIDs or file name GUIDs. The DXE Dispatcher will queue all the DXE drivers of this type immediately after the *a priori* file has been processed.

The following code example shows the dependency expression for a DXE driver that does not have any dependencies using the BNF grammar listed in Dependency Expression Grammar. This is followed by the 2-byte dependency expression that is encoded using the instruction set described in "Dependency Expression Instruction Set" on page 94.

```
//
// Source
//
TRUE
END

//
// Opcodes, Operands, and Binary Encoding
//
ADDR   BINARY                      MNEMONIC
====   ======================
=============================================
0x00 : 06                          TRUE
0x01 : 08                          END
```

10.9 Empty Dependency Expressions

If a DXE driver file does not contain a dependency section, then the DXE driver has an empty dependency expression. The DXE Foundation must support DXE driver and UEFI drivers that conform to the UEFI 2.0 specification. These UEFI drivers assume that all the UEFI Boot Services and UEFI Runtime Services are available. If an UEFI driver is added to a firmware volume, then the UEFI driver will have an empty dependency expression, and it should not be loaded and executed by the DXE Dispatcher until all the UEFI Boot Services and UEFI Runtime Services are available. The DXE Foundation cannot guarantee that this condition is true until all of the DXE Architectural Protocols have been installed.

From the DXE Dispatcher's perspective, DXE drivers without dependency expressions cannot be loaded until all of the DXE Architectural Protocols have been installed. This is equivalent to an implied dependency expression of all the GUIDs of the architectural protocols ANDed together. This implied dependency expression is shown below. The use of empty dependency expressions may also save space, because DXE drivers that require all the UEFI Boot Services and UEFI Runtime Services to be present can simply remove the dependency section from the DXE driver file.

The code example below shows the dependency expression that is implied by an empty dependency expression using the BNF grammar listed in "Dependency Expression Grammar" on page 193. It also shows the dependency expression after it has been encoded using the instruction set described in "Dependency Expression Instruction Set" on page 94. This fairly complex dependency expression is encoded into a dependency expression that is 216 bytes long. Typical dependency expressions will contain 2 or 3 terms, so those dependency expressions will typically be less than 60 bytes long.

```
//
// Source
//
EFI_BDS_ARCH_PROTOCOL_GUID                AND
EFI_CPU_ARCH_PROTOCOL_GUID                AND
EFI_METRONOME_ARCH_PROTOCOL_GUID          AND
EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL_GUID  AND
EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL_GUID    AND
EFI_RESET_ARCH_PROTOCOL_GUID              AND
EFI_RUNTIME_ARCH_PROTOCOL_GUID            AND
EFI_SECURITY_ARCH_PROTOCOL_GUID           AND
EFI_TIMER_ARCH_PROTOCOL_GUID              AND
EFI_VARIABLE_ARCH_PROTOCOL_GUID           AND
EFI_VARIABLE_WRITE_ARCH_PROTOCOL_GUID     AND
EFI_WATCHDOG_TIMER_ARCH_PROTOCOL_GUID
END

//
// Opcodes, Operands, and Binary Encoding
//
ADDR   BINARY                      MNEMONIC
====   ======================
=============================================
0x00 : 02                          PUSH
0x01 : F6 3F 5E 66 CC 46 d4 11     EFI_BDS_ARCH_PROTOCOL_GUID
       9A 38 00 90 27 3F C1 4D
0x11 : 02                          PUSH
0x12 : B1 CC BA 26 42 6F D4 11     EFI_CPU_ARCH_PROTOCOL_GUID
       BC E7 00 80 C7 3C 88 81
0x22 : 03                          AND
0x23 : 02                          PUSH
0x24 : B2 CC BA 26 42 6F d4 11     EFI_METRONOME_ARCH_PROTOCOL_GUID
       BC E7 00 80 C7 3C 88 81
0x34 : 02                          PUSH
0x35 : 72 70 A9 1D DC BD 30 4B
EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL_GUID
       99 F1 72 A0 B5 6F FF 2A
0x45 : 03                          AND
0x46 : 03                          AND
0x47 : 02                          PUSH
0x48 : 87 AC CF 27 CC 46 d4 11     EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL_GUID
       9A 38 00 90 27 3F C1 4D


0x58 : 02                          PUSH
0x59 : 88 AC CF 27 CC 46 d4 11     EFI_RESET_ARCH_PROTOCOL_GUID
       9A 38 00 90 27 3F C1 4D
0x69 : 03                          AND
0x6A : 02                          PUSH
0x6B : 53 82 d0 96 83 84 d4 11     EFI_RUNTIME_ARCH_PROTOCOL_GUID
       BC F1 00 80 C7 3C 88 81
```

```
0x7B : 02                                  PUSH
0x7C : E3 23 64 A4 17 46 f1 49             EFI_SECURITY_ARCH_PROTOCOL_GUID
       B9 FF D1 BF A9 11 58 39
       82 CE 5A 89 0C CB 2C 95
0xA0 : 02                                  PUSH
0xA1 : B3 CC BA 26 42 6F D4 11             EFI_TIMER_ARCH_PROTOCOL_GUID
       BC E7 00 80 C7 3C 88 81
0xB1 : 03                                  AND
0xB2 : 02                                  PUSH
0xB3 : E2 68 56 1E 81 84 D4 11             EFI_VARIABLE_ARCH_PROTOCOL_GUID
       BC F1 00 80 C7 3C 88 81
0xC3 : 02                                  PUSH
0xC4 : 18 F8 41 64 62 63 44 4E             EFI_VARIABLE_WRITE_ARCH_PROTOCOL_GUID
       B5 70 7D BA 31 DD 24 53
0xD4 : 03                                  AND
0xD5 : 03                                  AND
0xD6 : 03                                  AND
0xD7 : 02                                  PUSH
0xD8 : F5 3F 5E 66 CC 46 d4 11             EFI_WATCHDOG_TIMER_ARCH_PROTOCOL_GUID
       9A 38 00 90 27 3F C1 4D
0xE8 : 03                                  AND
0xE9 : 08                                  END
```

10.10 Dependency Expression Reverse Polish Notation (RPN)

The actual equations will be presented by the DXE driver in a simple-to-evaluate form, namely postfix.

The following is a BNF encoding of this grammar. See "Dependency Expression Instruction Set" on page 94 for definitions of the dependency expressions.

```
<statement> ::= SOR <expression> END |
                BEFORE <guid> END |
                AFTER <guid> END |
                <expression> END

<expression> ::= PUSH <guid> |
                 TRUE |
                 FALSE |
                 <expression> NOT |
                 <expression> <expression> OR |
                 <expression> <expression> AND
```

10.11 DXE Dispatcher State Machine

The DXE Dispatcher is responsible for tracking the state of a DXE driver from the time that the DXE driver is discovered in a firmware volume until the DXE Foundation is terminated with a call to **`ExitBootServices()`**. During this time, each DXE driver may be in one of several different states. The state machine that the DXE Dispatcher must use to track a DXE driver is shown in Figure 7.

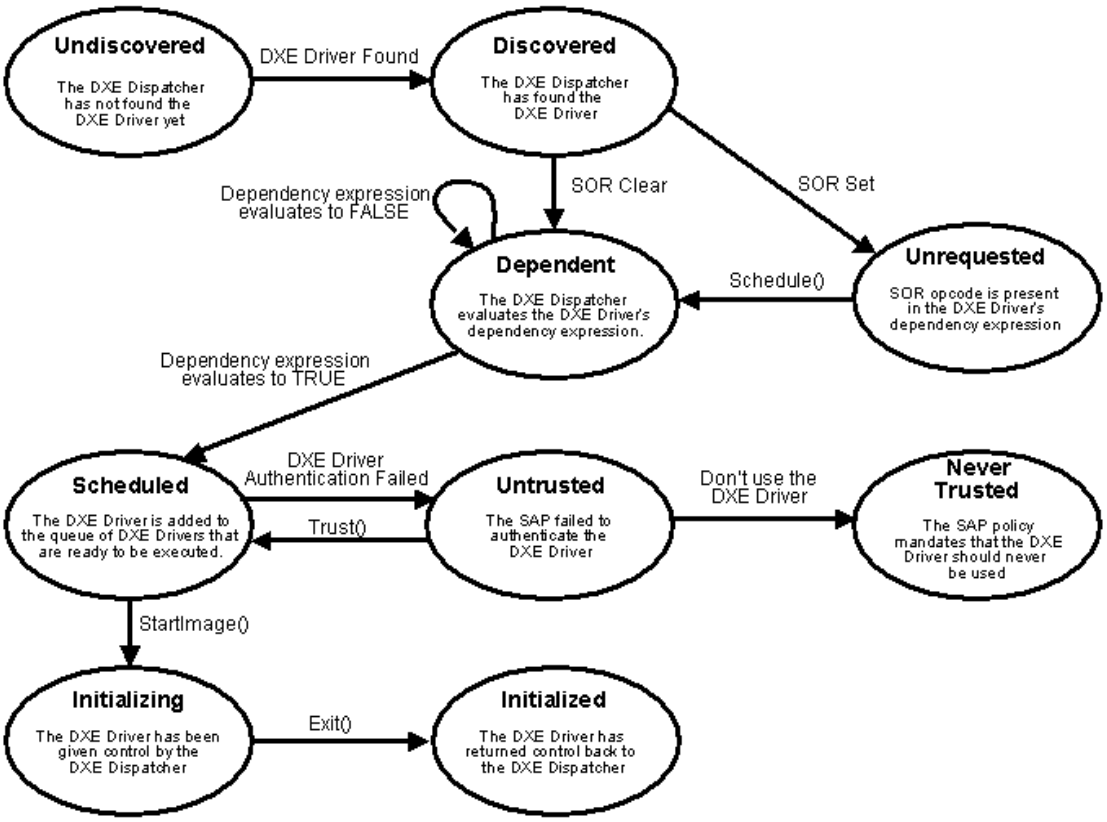

Figure 7. DXE Driver States

A DXE driver starts in the “Undiscovered” state, which means that the DXE driver is in a firmware volume that the DXE Dispatcher does not know about yet. When the DXE Dispatcher discovers a new firmware volume, any DXE drivers from that firmware volume listed in the *a priori* file are immediately loaded and executed. DXE drivers listed in the *a priori* file are immediately promoted to the “Scheduled” state. The firmware volume is then searched for DXE drivers that are not listed in the *a priori* file. Any DXE drivers found are promoted from the “Undiscovered” to the “Discovered” state. The dependency expression for each DXE driver is evaluated. If the SOR opcode is present in a DXE driver’s dependency expression, then the DXE driver is placed in the “Unrequested” state. If the SOR opcode is not present in the DXE driver’s dependency expression, then the DXE driver is placed in the “Dependent” state. Once a DXE driver is in the "Unrequested” state, it may only be promoted to the “Dependent” state with a call to the DXE Service **`Schedule()`**.

Once a DXE Driver is in the “Dependent” state, the DXE Dispatcher will evaluate the DXE driver’s dependency expression. If the DXE driver does not have a dependency expression, then a dependency expression of all the architectural protocols ANDed together is assumed for that DXE driver. If the dependency expression evaluates to **`FALSE`**, then the DXE driver stays in the

"Dependent" state. If the dependency expression never evaluates to **TRUE**, then it will never leave the "Dependent" state. If the dependency expression evaluates to **TRUE**, then the DXE driver will be promoted to the "Scheduled" state.

A DXE driver that is prompted to the "Scheduled" state is added to the end of the queue of other DXE drivers that have been promoted to the "Scheduled" state. When the DXE driver has reached the head of the queue, the DXE Dispatcher must use the services of the Security Authentication Protocol (SAP) to check the authentication status of the DXE Driver. If the Security Authentication Protocol deems that the DXE Driver violates the security policy of the platform, then the DXE Driver is placed in the "Untrusted" state. The Security Authentication Protocol can also tell the DXE Dispatcher that the DXE driver should never be executed and be placed in the "Never Trusted" state. If a DXE driver is placed in the "Untrusted" state, it can only be promoted back to the "Scheduled" state with a call to the DXE Service **Trust()**.

Once a DXE driver has reached the head of the scheduled queue, and the DXE driver has passed the authentication checks of the Security Authentication Protocol, the DXE driver is loaded into memory with the Boot Service **LoadImage()**. Control is then passed from the DXE Dispatcher to the DXE driver with the Boot Service **StartImage()**. When **StartImage()** is called for a DXE driver, that DXE driver is promoted to the "Initializing" state. The DXE driver returns control to the DXE Dispatcher through the Boot Service **Exit()**. When a DXE driver has returned control to the DXE Dispatcher, the DXE driver is in the terminal state called "Initialized."

The DXE Dispatcher is responsible for draining the queue of DXE drivers in the "Scheduled" state until the queue is empty. Once the queue is empty, then DXE Dispatcher must evaluate all the DXE drivers in the "Dependent" state to see if any of them need to be promoted to the "Scheduled" state. These evaluations need to be performed every time one or more DXE drivers have been promoted to the "Initialized" state, because those DXE drivers may have produced protocol interfaces for which the DXE drivers in the "Dependent" state are waiting.

10.12 Example Orderings

The order that DXE drivers are loaded and executed by the DXE Dispatcher is a mix of strong and weak orderings. The strong orderings are specified through *a priori* files, and the weak orderings are specified by dependency expressions in DXE drivers. Figure 8 shows the contents of a sample firmware volume that contains the following:

- DXE Foundation image
- DXE driver images
- An *a priori* file

The order that these images appear in the firmware volume is arbitrary. The DXE Foundation and the DXE Dispatcher must not make any assumptions about the locations of files in firmware volumes. The *a priori* file contains the GUID file names of the DXE drivers that are to be loaded and executed first. The dependency expressions and the protocols that each DXE driver produces is shown next to each DXE driver image in the firmware volume.

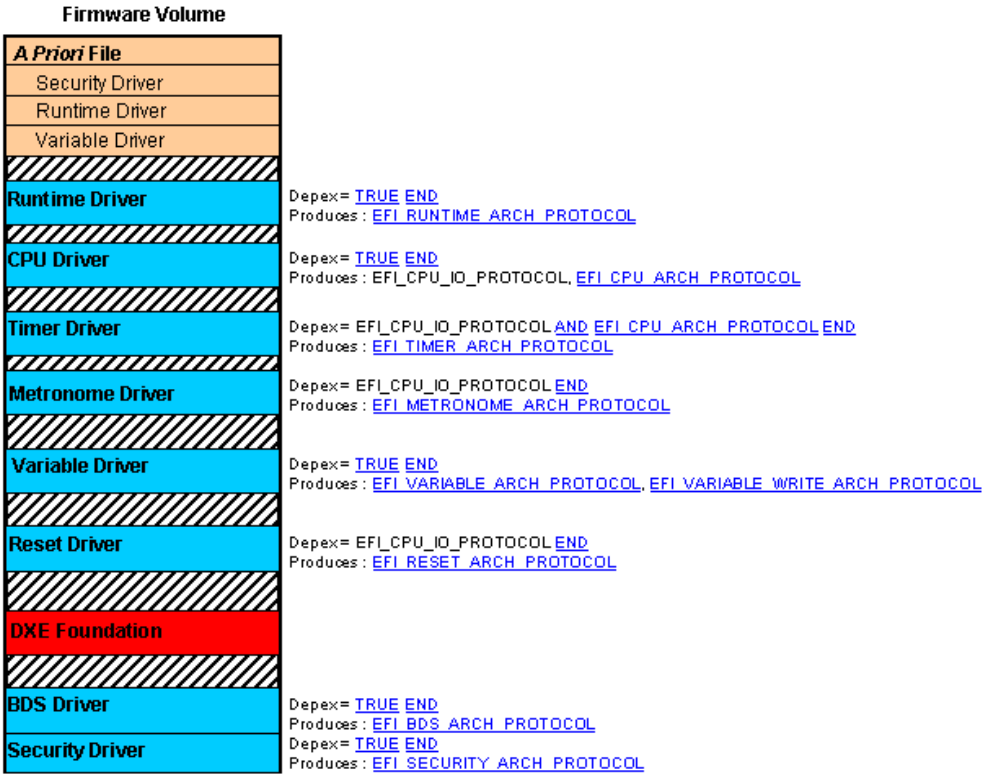

Figure 8. Sample Firmware Volume

Based on the contents of the firmware volume in the figure above, the Security Driver, Runtime Driver, and Variable Driver will always be executed first. This is an example of a strongly ordered dispatch due to the *a priori* file. The DXE Dispatcher will then evaluate the dependency expressions of the remaining DXE drivers to determine the order that they will be executed. Based on the dependency expressions and the protocols that each DXE driver produces, there are 30 valid orderings from which the DXE Dispatcher may choose. The BDS Driver and CPU Driver tie for the next drivers to be scheduled, because their dependency expressions are simply **TRUE**. A dependency expression of TRUE means that the DXE driver does not require any other protocol interfaces to be executed. The DXE Dispatcher may choose either one of these drivers to be scheduled first. The Timer Driver, Metronome Driver, and Reset Driver all depend on the protocols produced by the CPU Driver. Once the CPU Driver has been loaded and executed, the Timer Driver, Metronome Driver, and Reset Driver may be scheduled in any order. The table below shows all 30 possible orderings from the sample firmware volume in the figure above. Each ordering is listed from left to right across the table. A reasonable implementation of a DXE Dispatcher would consistently produce the same ordering for a given system configuration. If the configuration of the system is changed in any way (including a order of files stored in a firmware volume), then a

different dispatch ordering may be generated, but this new ordering should be consistent until the next system configuration change.

Table 30. DXE Dispatcher Orderings

Dispatch Order								
	1	2	3	4	5	6	7	8
1	Security	Runtime	Variable	BDS	CPU	Timer	Metronome	Reset
2	Security	Runtime	Variable	BDS	CPU	Timer	Reset	Metronome
3	Security	Runtime	Variable	BDS	CPU	Metronome	Timer	Reset
4	Security	Runtime	Variable	BDS	CPU	Metronome	Reset	Timer
5	Security	Runtime	Variable	BDS	CPU	Reset	Timer	Metronome
6	Security	Runtime	Variable	BDS	CPU	Reset	Metronome	Timer
7	Security	Runtime	Variable	CPU	BDS	Timer	Metronome	Reset
8	Security	Runtime	Variable	CPU	BDS	Timer	Reset	Metronome
9	Security	Runtime	Variable	CPU	BDS	Metronome	Timer	Reset
10	Security	Runtime	Variable	CPU	BDS	Metronome	Reset	Timer
11	Security	Runtime	Variable	CPU	BDS	Reset	Timer	Metronome
12	Security	Runtime	Variable	CPU	BDS	Reset	Metronome	Timer
13	Security	Runtime	Variable	CPU	Timer	BDS	Metronome	Reset
14	Security	Runtime	Variable	CPU	Timer	BDS	Reset	Metronome
15	Security	Runtime	Variable	CPU	Timer	Metronome	BDS	Reset
16	Security	Runtime	Variable	CPU	Timer	Metronome	Reset	BDS
17	Security	Runtime	Variable	CPU	Timer	Reset	BDS	Metronome
18	Security	Runtime	Variable	CPU	Timer	Reset	Metronome	BDS
19	Security	Runtime	Variable	CPU	Metronome	Timer	BDS	Reset
20	Security	Runtime	Variable	CPU	Metronome	Timer	Reset	BDS
21	Security	Runtime	Variable	CPU	Metronome	BDS	Timer	Reset
22	Security	Runtime	Variable	CPU	Metronome	BDS	Reset	Timer
23	Security	Runtime	Variable	CPU	Metronome	Reset	Timer	BDS
24	Security	Runtime	Variable	CPU	Metronome	Reset	BDS	Timer
25	Security	Runtime	Variable	CPU	Reset	Timer	Metronome	BDS
26	Security	Runtime	Variable	CPU	Reset	Timer	BDS	Metronome
27	Security	Runtime	Variable	CPU	Reset	Metronome	Timer	BDS
28	Security	Runtime	Variable	CPU	Reset	Metronome	BDS	Timer
29	Security	Runtime	Variable	CPU	Reset	BDS	Timer	Metronome
30	Security	Runtime	Variable	CPU	Reset	BDS	Metronome	Timer

10.13 Security Considerations

The DXE Dispatcher is required to use the services of the Security Architectural Protocol every time a firmware volume is discovered and before each DXE driver is executed. Because the Security Architectural Protocol is produced by a DXE driver, there will be at least one firmware volume discovered, and one or more DXE drivers loaded and executed before the Security Architectural Protocol is installed. The DXE Dispatcher should not attempt to use the services of the Security Architectural Protocol until the Security Architectural Protocol is installed. If a platform requires the Security Architectural Protocol to be present very early in the DXE phase, then the *a priori* file may be used to specify the name of the DXE driver that produces the Security Architectural Protocol.

The Security Architectural Protocol provides a service to evaluate the authentication status of a file. This service can also be used to evaluate the authenticate status of a firmware volume. If the authentication status is good, then no action is taken. If there is a problem with the firmware volume's authentication status, then the Security Architectural Protocol may perform a platform specific action. One option is to force the DXE Dispatcher to ignore the firmware volume so no DXE drivers will be loaded and executed from it. Another is to log the fact that the DXE Dispatcher is going to start dispatching DXE driver from a firmware volume with a questionable authentication status.

The Security Architectural Protocol can also be used to evaluate the authentication status of each DXE driver discovered in a firmware volume. If the authentication status is good, then no action is taken. If there is a problem with the DXE driver's authentication status, then the Security Architectural Protocol may take a platform-specific action. One possibility is to force the DXE driver into the "Untrusted" state, so it will not be considered for dispatch until the Boot Service **`Trust()`** is called for that DXE driver. Another possibility is to have the DXE Dispatcher place the DXE driver in the "Never Trusted" state, so it will never be loaded or executed. Another option is to log the fact that a DXE driver with a questionable authentication status is about to be loaded and executed.

11
DXE Drivers

11.1 Introduction

The DXE architecture provides a rich set of extensible services that provides for a wide variety of different system firmware designs. The DXE Foundation provides the generic services required to locate and execute DXE drivers. The DXE drivers are the components that actually initialize the platform and provide the services required to boot an UEFI-compliant operating system or a set of UEFI-compliant system utilities. There are many possible firmware implementations for any given platform. Because the DXE Foundation has fixed functionality, all the added value and flexibility in a firmware design is embodied in the implementation and organization of DXE drivers.

There are two basic classes of DXE drivers:

- Early DXE Drivers
- DXE Drivers that follow the UEFI Driver Model

Additional classifications of DXE drivers are also possible.

All DXE drivers may consume the UEFI Boot Services, UEFI Runtime Services, and DXE Services to perform their functions. DXE drivers must use dependency expressions to guarantee that the services and protocol interfaces they require are available before they are executed. See the following topics for the DXE Architectural Protocols upon which the services depend:

- UEFI Boot Services Dependencies
- UEFI Runtime Services Dependencies
- DXE Services Dependencies

11.2 Classes of DXE Drivers

11.2.1 Early DXE Drivers

The first class of DXE drivers are those that execute very early in the DXE phase. The execution order of these DXE drivers depends on the following:

- The presence and contents of an a priori file
- The evaluation of dependency expressions

These early DXE drivers will typically contain basic services, processor initialization code, chipset initialization code, and platform initialization code. These early drivers will also typically produce the DXE Architectural Protocols that are required for the DXE Foundation to produces its full complement of UEFI Boot Services and UEFI Runtime Services. To support the fastest possible boot time, as much initialization should be deferred to the DXE drivers that follow UEFI Driver Model described in the UEFI 2.0 specification.

The early DXE drivers need to be aware that not all of the UEFI Boot Services, UEFI Runtime Services, and DXE Services may be available when they execute because not all of the DXE Architectural Protocols may be been registered yet.

11.2.2 DXE Drivers that Follow the UEFI Driver Model

The second class of DXE drivers are those that follow the UEFI Driver Model in the UEFI 2.0 specification. These drivers do not touch any hardware resources when they initialize. Instead, they register a Driver Binding Protocol interface in the handle database. The set of Driver Binding Protocols are used by the Boot Device Selection (BDS) phase to connect the drivers to the devices that are required to establish consoles and provide access to boot devices. The DXE drivers that follow the UEFI Driver Model ultimately provide software abstractions for console devices and boot devices, but only when they are explicitly asked to do so.

The DXE drivers that follow the UEFI Driver Model do not need to be concerned with dependency expressions. These drivers simply register the Driver Binding Protocol in the handle database when they are executed, and this operation can be performed without the use of any DXE Architectural Protocols. DXE drivers with empty dependency expressions will not be dispatched by the DXE Dispatcher until all of the DXE Architectural Protocols have been installed.

11.2.3 Additional Classifications

DXE drivers can also be classified as the following:

- Boot service drivers
- Runtime drivers

Boot service drivers provide services that are available until the **`ExitBootServices()`** function is called. When **`ExitBootServices()`** is called, all the memory used by boot service drivers is released for use by an operating system.

Runtime drivers provide services that are available before and after **`ExitBootServices()`** is called, including the time that an operating system is running. All of the services in the UEFI Runtime Services Table are produced by runtime drivers.

The DXE Foundation is considered a boot service component, so the DXE Foundation is also released when **`ExitBootServices()`** is called. As a result, runtime drivers may not use any of the UEFI Boot Services, DXE Services, or services produced by boot service drivers after **`ExitBootServices()`** is called.

12
DXE Architectural Protocols

12.1 Introduction

The DXE Foundation is abstracted from the platform hardware through a set of architectural protocols. These protocols function just like other protocols in every way. The only difference is that these architectural protocols are the protocols that the DXE Foundation itself consumes to produce the UEFI Boot Services, UEFI Runtime Services, and DXE Services. DXE drivers that are loaded from firmware volumes produce the DXE Architectural Protocols. This means that the DXE Foundation must have enough services to load and start DXE drivers before even a single DXE driver is executed.

The DXE Foundation is passed a HOB list that must contain a description of some amount of system memory and at least one firmware volume. The system memory descriptors in the HOB list are used to initialize the UEFI services that require only memory to function correctly. The system is also guaranteed to be running on only one processor in flat physical mode with interrupts disabled. The firmware volume is passed to the DXE Dispatcher, and the DXE Dispatcher must contain a read-only firmware file system driver to search for the a priori file and any DXE drivers in the firmware volumes. When a driver is discovered that needs to be loaded and executed, the DXE Dispatcher will use a PE/COFF loader to load and invoke the DXE driver. The early DXE drivers will produce the DXE Architectural Protocols, so the DXE Foundation can produce the full complement of UEFI Boot Services and UEFI Runtime Services.

Figure 9 shows the HOB list being passed to the DXE Foundation.

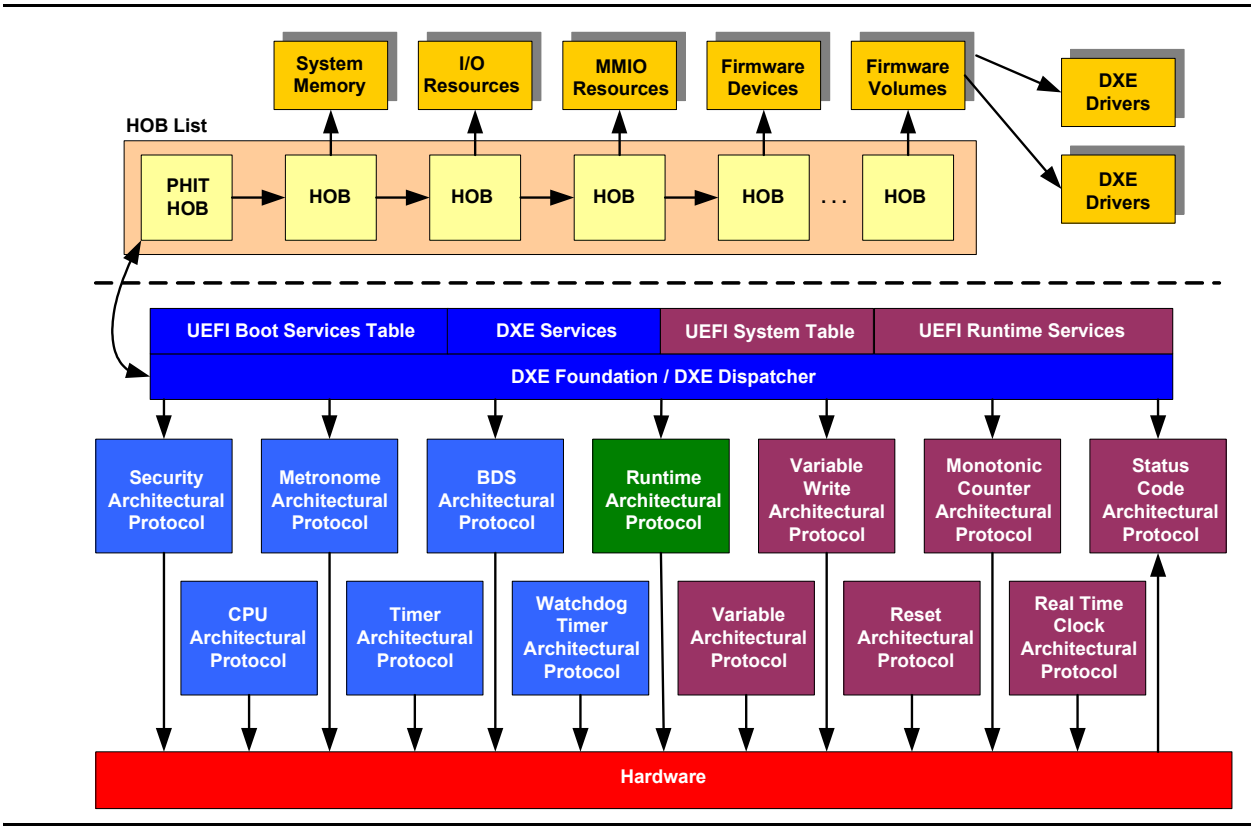

Figure 9. DXE Architectural Protocols

The DXE Foundation consumes the services of the DXE Architectural Protocols and produces the following:

- UEFI System Table
- UEFI Boot Services Table
- UEFI Runtime Services Table
- DXE Services Table

The UEFI Boot Services Table and DXE Services Table are allocated from UEFI boot services memory, which means that the UEFI Boot Services Table and DXE Services Table are freed when the OS runtime phase is entered. The UEFI System Table and UEFI Runtime Services Table are allocated from UEFI runtime services memory, and they persist into the OS runtime phase.

When executing upon an UEFI-compliant system, UEFI drivers, applications, and UEFI-aware operating systems can discern if the platform is built upon the Foundation by searching for the DXE Services Table GUID in the UEFI System configuration table.

The DXE Architectural Protocols shown on the left of the figure are used to produce the UEFI Boot Services and DXE Services. The DXE Foundation and these protocols will be freed when the system transitions to the OS runtime phase. The DXE Architectural Protocols shown on the right are used to produce the UEFI Runtime Services. These services will persist in the OS runtime phase. The Runtime Architectural Protocol in the middle is unique. This protocol provides the services that

are required to transition the runtime services from physical mode to virtual mode under the direction of an OS. Once this transition is complete, the services of the Runtime Architectural Protocol can no longer be used. The following topics describe all of the DXE Architectural Protocols in detail.

12.2 Boot Device Selection (BDS) Architectural Protocol

EFI_BDS_ARCH_PROTOCOL

Summary

Transfers control from the DXE phase to an operating system or system utility. This protocol must be produced by a boot service or runtime DXE driver and may only be consumed by the DXE Foundation.

GUID

```
#define EFI_BDS_ARCH_PROTOCOL_GUID  \
  {0x665E3FF6,0x46CC,0x11d4,\
  0x9A,0x38,0x00,0x90,0x27,0x3F,0xC1,0x4D}
```

Protocol Interface Structure

```
typedef struct {
  EFI_BDS_ENTRY  Entry;
} EFI_BDS_ARCH_PROTOCOL;
```

Parameters

Entry

The entry point to BDS. See the **Entry()** function description. This call does not take any parameters, and the return value can be ignored. If it returns, then the dispatcher must be invoked again, if it never returns, then an operating system or a system utility have been invoked.

Description

The **EFI_BDS_ARCH_PROTOCOL** transfers control from DXE to an operating system or a system utility. If there are not enough drivers initialized when this protocol is used to access the required boot device(s), then this protocol should add drivers to the dispatch queue and return control back to the dispatcher. Once the required boot devices are available, then the boot device can be used to load and invoke an OS or a system utility.

EFI_BDS_ARCH_PROTOCOL.Entry()

Summary

Performs Boot Device Selection (BDS) and transfers control from the DXE Foundation to the selected boot device. The implementation of the boot policy must follow the rules outlined in the Boot Manager chapter of the UEFI 2.0 specification. This boot policy allows for flexibility, so the platform vendor will typically customize the implementation of this service.

Prototype

```
typedef
VOID
(EFIAPI *EFI_BDS_ENTRY) (
  IN CONST EFI_BDS_ARCH_PROTOCOL  *This
  );
```

Parameters

This

The **EFI_BDS_ARCH_PROTOCOL** instance.

Description

This function uses policy data from the platform to determine what operating system or system utility should be loaded and invoked. This function call also optionally uses the user's input to determine the operating system or system utility to be loaded and invoked. When the DXE Foundation has dispatched all the drivers on the dispatch queue, this function is called. This function will attempt to connect the boot devices required to load and invoke the selected operating system or system utility. During this process, additional firmware volumes may be discovered that may contain addition DXE drivers that can be dispatched by the DXE Foundation. If a boot device cannot be fully connected, this function calls the DXE Service **Dispatch()** to allow the DXE drivers from any newly discovered firmware volumes to be dispatched. Then the boot device connection can be attempted again. If the same boot device connection operation fails twice in a row, then that boot device has failed, and should be skipped. This function should never return.

12.3 CPU Architectural Protocol

EFI_CPU_ARCH_PROTOCOL

Summary

Abstracts the processor services that are required to implement some of the DXE services. This protocol must be produced by a boot service or runtime DXE driver and may only be consumed by the DXE Foundation and DXE drivers that produce architectural protocols.

GUID

```
#define EFI_CPU_ARCH_PROTOCOL_GUID  \
 {0x26baccb1,0x6f42,0x11d4,\
  0xbc,0xe7,0x0,0x80,0xc7,0x3c,0x88,0x81}
```

Protocol Interface Structure

```
typedef struct _EFI_CPU_ARCH_PROTOCOL {
  EFI_CPU_FLUSH_DATA_CACHE              FlushDataCache;
  EFI_CPU_ENABLE_INTERRUPT              EnableInterrupt;
  EFI_CPU_DISABLE_INTERRUPT             DisableInterrupt;
  EFI_CPU_GET_INTERRUPT_STATE           GetInterruptState;
  EFI_CPU_INIT                          Init;
  EFI_CPU_REGISTER_INTERRUPT_HANDLER    RegisterInterruptHandler;
  EFI_CPU_GET_TIMER_VALUE               GetTimerValue;
  EFI_CPU_SET_ATTRIBUTES                SetMemoryAttributes;
  UINT32                                NumberOfTimers;
  UINT32                                DmaBufferAlignment;
} EFI_CPU_ARCH_PROTOCOL;
```

Parameters

FlushDataCache

Flushes a range of the processor's data cache. See the **FlushDataCache()** function description. If the processor does not contain a data cache, or the data cache is fully coherent, then this function can just return **EFI_SUCCESS**. If the processor does not support flushing a range of addresses from the data cache, then the entire data cache must be flushed. This function is used by the root bridge I/O abstractions to flush data caches for DMA operations.

EnableInterrupt

Enables interrupt processing by the processor. See the **EnableInterrupt()** function description. This function is used by the Boot Service **RaiseTPL()** and **RestoreTPL()**.

DisableInterrupt

Disables interrupt processing by the processor. See the **DisableInterrupt()** function description. This function is used by the Boot Service **RaiseTPL()** and **RestoreTPL()**.

GetInterruptState

Retrieves the processor's current interrupt state. See the **GetInterruptState()** function description.

Init

Generates an INIT on the processor. See the **Init()** function description. This function may be used by the **EFI_RESET** Protocol depending upon a specified boot path. If a processor cannot programmatically generate an INIT without help from external hardware, then this function returns **EFI_UNSUPPORTED**.

RegisterInterruptHandler

Associates an interrupt service routine with one of the processor's interrupt vectors. See the **RegisterInterruptHandler()** function description. This function is typically used by the **EFI_TIMER_ARCH_PROTOCOL** to hook the timer interrupt in a system. It can also be used by the debugger to hook exception vectors.

GetTimerValue

Returns the value of one of the processor's internal timers. See the **GetTimerValue()** function description.

SetMemoryAttributes

Attempts to set the attributes of a memory region. See the **SetMemoryAttributes()** function description.

NumberOfTimers

The number of timers that are available in a processor. The value in this field is a constant that must not be modified after the CPU Architectural Protocol is installed. All consumers must treat this as a read-only field.

DmaBufferAlignment

The size, in bytes, of the alignment required for DMA buffer allocations. This is typically the size of the largest data cache line in the platform. This value can be determined by looking at the data cache line sizes of all the caches present in the platform, and returning the largest. This is used by the root bridge I/O abstraction protocols to guarantee that no two DMA buffers ever share the same cache line. The value in this field is a constant that must not be modified after the CPU Architectural Protocol is installed. All consumers must treat this as a read-only field.

Description

The **EFI_CPU_ARCH_PROTOCOL** is used to abstract processor-specific functions from the DXE Foundation. This includes flushing caches, enabling and disabling interrupts, hooking interrupt vectors and exception vectors, reading internal processor timers, resetting the processor, and determining the processor frequency.

The GCD memory space map is initialized by the DXE Foundation based on the contents of the HOB list. The HOB list contains the capabilities of the different memory regions, but it does not contain their current attributes. The DXE driver that produces the **EFI_CPU_ARCH_PROTOCOL** is responsible for maintaining the current attributes of the memory regions visible to the processor. This means that the DXE driver that produces the **EFI_CPU_ARCH_PROTOCOL** must seed the

GCD memory space map with the initial state of the attributes for all the memory regions visible to the processor. The DXE Service **SetMemorySpaceAttributes()** allows the attributes of a memory range to be modified. The **SetMemorySpaceAttributes()** DXE Service is implemented using the **SetMemoryAttributes()** service of the **EFI_CPU_ARCH_PROTOCOL**.

To initialize the state of the attributes in the GCD memory space map, the DXE driver that produces the **EFI_CPU_ARCH_PROTOCOL** must call the DXE Service **SetMemorySpaceAttributes()** for all the different memory regions visible to the processor passing in the current attributes. This, in turn, will call back to the **SetMemoryAttributes()** service of the **EFI_CPU_ARCH_PROTOCOL**, and all of these calls must return **EFI_SUCCESS**, since the DXE Foundation is only requesting that the attributes of the memory region be set to their current settings. This will force the current attributes in the GCD memory space map to be set to these current settings. After this initialization is complete, the next call to the DXE Service **GetMemorySpaceMap()** will correctly show the current attributes of all the memory regions. In addition, any future calls to the DXE Service **SetMemorySpaceAttributes()** will in turn call the **EFI_CPU_ARCH_PROTOCOL** so see of those attributes can be modified, and if they can, the GCD memory space map will be updated accordingly.

EFI_CPU_ARCH_PROTOCOL.FlushDataCache()

Summary

Flushes a range of the processor's data cache. If the processor does not contain a data cache, or the data cache is fully coherent, then this function can just return **EFI_SUCCESS**. If the processor does not support flushing a range of addresses from the data cache, then the entire data cache must be flushed. This function is used by the root bridge I/O abstractions to flush caches for DMA operations.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_FLUSH_DATA_CACHE) (
  IN CONST EFI_CPU_ARCH_PROTOCOL  *This,
  IN EFI_PHYSICAL_ADDRESS         Start,
  IN UINT64                       Length,
  IN EFI_CPU_FLUSH_TYPE           FlushType
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

Start

The beginning physical address to flush from the processor's data cache.

Length

The number of bytes to flush from the processor's data cache. This function may flush more bytes than *Length* specifies depending upon the granularity of the flush operation that the processor supports.

FlushType

Specifies the type of flush operation to perform. Type **EFI_CPU_FLUSH_TYPE** is defined in "Related Definitions" below.

Description

This function flushes the range of addresses from *Start* to *Start+Length* from the processor's data cache. If *Start* is not aligned to a cache line boundary, then the bytes before *Start* to the preceding cache line boundary are also flushed. If *Start+Length* is not aligned to a cache line boundary, then the bytes past *Start+Length* to the end of the next cache line boundary are also flushed. If the address range is flushed, then **EFI_SUCCESS** is returned. If the address range cannot be flushed, then **EFI_DEVICE_ERROR** is returned. If the processor does not support the flush type specified by *FlushType*, then **EFI_UNSUPPORTED** is returned. The *FlushType* of *EfiCpuFlushTypeWriteBackInvalidate* must be supported. If the data cache is fully coherent with all DMA operations, then this function can just return **EFI_SUCCESS**. If the processor does not support flushing a range of the data cache, then the entire data cache can be flushed.

Related Definitions

```
typedef enum {
  EfiCpuFlushTypeWriteBackInvalidate,
  EfiCpuFlushTypeWriteBack,
  EfiCpuFlushTypeInvalidate,
  EfiCpuMaxFlushType
} EFI_CPU_FLUSH_TYPE;
```

Status Codes Returned

EFI_SUCCESS	The address range from *Start* to *Start+Length* was flushed from the processor's data cache.
EFI_UNSUPPORTED	The processor does not support the cache flush type specified by *FlushType*.
EFI_DEVICE_ERROR	The address range from *Start* to *Start+Length* could not be flushed from the processor's data cache.

EFI_CPU_ARCH_PROTOCOL.EnableInterrupt()

Summary

Enables interrupt processing by the processor. This function is used to implement the Boot Services **RaiseTPL()** and **RestoreTPL()**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_ENABLE_INTERRUPT) (
  IN CONST EFI_CPU_ARCH_PROTOCOL  *This
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

Description

This function enables interrupt processing by the processor. If interrupts are enabled, then **EFI_SUCCESS** is returned. Otherwise, **EFI_DEVICE_ERROR** is returned.

Status Codes Returned

EFI_SUCCESS	Interrupts are enabled on the processor.
EFI_DEVICE_ERROR	Interrupts could not be enabled on the processor.

EFI_CPU_ARCH_PROTOCOL.DisableInterrupt()

Summary

Disables interrupt processing by the processor. This function is used to implement the Boot Services **RaiseTPL()** and **RestoreTPL()**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_DISABLE_INTERRUPT) (
  IN CONST EFI_CPU_ARCH_PROTOCOL  *This
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

Description

This function disables interrupt processing by the processor. If interrupts are disabled, then **EFI_SUCCESS** is returned. Otherwise, **EFI_DEVICE_ERROR** is returned.

Status Codes Returned

EFI_SUCCESS	Interrupts are disabled on the processor.
EFI_DEVICE_ERROR	Interrupts could not be disabled on the processor.

EFI_CPU_ARCH_PROTOCOL.GetInterruptState()

Summary

Retrieves the processor's current interrupt state.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_GET_INTERRUPT_STATE) (
  IN  CONST EFI_CPU_ARCH_PROTOCOL  *This,
  OUT BOOLEAN                      *State
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

State

A pointer to the processor's current interrupt state. Set to **TRUE** if interrupts are enabled and **FALSE** if interrupts are disabled.

Description

This function retrieves the processor's current interrupt state a returns it in *State*. If interrupts are currently enabled, then **TRUE** is returned. If interrupts are currently disabled, then **FALSE** is returned. If *State* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. Otherwise, **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The processor's current interrupt state was returned in *State*.
EFI_INVALID_PARAMETER	*State* is **NULL**.

EFI_CPU_ARCH_PROTOCOL.Init()

Summary

Generates an INIT on the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_INIT) (
  IN CONST EFI_CPU_ARCH_PROTOCOL   *This,
  IN EFI_CPU_INIT_TYPE             InitType
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

InitType

The type of processor INIT to perform. Type **EFI_CPU_INIT_TYPE** is defined in "Related Definitions" below.

Description

This function generates an INIT on the processor. If this function succeeds, then the processor will be reset, and control will not be returned to the caller. If *InitType* is not supported by this processor, or the processor cannot programmatically generate an INIT without help from external hardware, then **EFI_UNSUPPORTED** is returned. If an error occurs attempting to generate an INIT, then **EFI_DEVICE_ERROR** is returned.

Related Definitions

```
typedef enum {
  EfiCpuInit,
  EfiCpuMaxInitType
} EFI_CPU_INIT_TYPE;
```

Status Codes Returned

EFI_SUCCESS	The processor INIT was performed. This return code should never be seen.
EFI_UNSUPPORTED	The processor INIT operation specified by *InitType* is not supported by this processor.
EFI_DEVICE_ERROR	The processor INIT failed.

EFI_CPU_ARCH_PROTOCOL.RegisterInterruptHandler()

Summary

Registers a function to be called from the processor interrupt handler.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_REGISTER_INTERRUPT_HANDLER) (
  IN CONST EFI_CPU_ARCH_PROTOCOL      *This,
  IN EFI_EXCEPTION_TYPE               InterruptType,
  IN EFI_CPU_INTERRUPT_HANDLER        InterruptHandler
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

InterruptType

Defines which interrupt or exception to hook. Type **EFI_EXCEPTION_TYPE** and the valid values for this parameter are defined in **EFI_DEBUG_SUPPORT_PROTOCOL** of the UEFI 2.0 specification.

InterruptHandler

A pointer to a function of type **EFI_CPU_INTERRUPT_HANDLER** that is called when a processor interrupt occurs. If this parameter is **NULL**, then the handler will be uninstalled. Type **EFI_CPU_INTERRUPT_HANDLER** is defined in "Related Definitions" below.

Description

The **RegisterInterruptHandler()** function registers and enables the handler specified by *InterruptHandler* for a processor interrupt or exception type specified by *InterruptType*. If *InterruptHandler* is **NULL**, then the handler for the processor interrupt or exception type specified by *InterruptType* is uninstalled. The installed handler is called once for each processor interrupt or exception.

If the interrupt handler is successfully installed or uninstalled, then **EFI_SUCCESS** is returned.

If *InterruptHandler* is not **NULL**, and a handler for *InterruptType* has already been installed, then **EFI_ALREADY_STARTED** is returned.

If *InterruptHandler* is **NULL**, and a handler for *InterruptType* has not been installed, then **EFI_INVALID_PARAMETER** is returned.

If *InterruptType* is not supported, then **EFI_UNSUPPORTED** is returned.

The **EFI_CPU_ARCH_PROTOCOL** implementation of this function must handle saving and restoring system context to the system context record around calls to the interrupt handler. It must also perform the necessary steps to return to the context that was interrupted by the interrupt. No chaining of interrupt handlers is allowed.

Related Definitions

```
typedef
VOID
(*EFI_CPU_INTERRUPT_HANDLER) (
  IN EFI_EXCEPTION_TYPE   InterruptType,
  IN EFI_SYSTEM_CONTEXT   SystemContext
  );
```

InterruptType

Defines the type of interrupt or exception that occurred on the processor. This parameter is processor architecture specific. The type **EFI_EXCEPTION_TYPE** and the valid values for this parameter are defined in **EFI_DEBUG_SUPPORT_PROTOCOL** of the UEFI 2.0 specification.

SystemContext

A pointer to the processor context when the interrupt occurred on the processor. Type **EFI_SYSTEM_CONTEXT** is defined in the **EFI_DEBUG_SUPPORT_PROTOCOL** of the UEFI 2.0 specification.

Status Codes Returned

EFI_SUCCESS	The handler for the processor interrupt was successfully installed or uninstalled.
EFI_ALREADY_STARTED	*InterruptHandler* is not **NULL**, and a handler for *InterruptType* was previously installed.
EFI_INVALID_PARAMETER	*InterruptHandler* is **NULL**, and a handler for *InterruptType* was not previously installed.
EFI_UNSUPPORTED	The interrupt specified by *InterruptType* is not supported.

EFI_CPU_ARCH_PROTOCOL.GetTimerValue()

Summary

Returns a timer value from one of the processor's internal timers.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_GET_TIMER_VALUE) (
  IN  CONST EFI_CPU_ARCH_PROTOCOL  *This,
  IN  UINT32                       TimerIndex,
  OUT UINT64                       *TimerValue,
  OUT UINT64                       *TimerPeriod  OPTIONAL
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

TimerIndex

Specifies which processor timer is to be returned in *TimerValue*. This parameter must be between 0 and *NumberOfTimers*-**1**.

TimerValue

Pointer to the returned timer value.

TimerPeriod

A pointer to the amount of time that passes in femtoseconds (10^{-15}) for each increment of *TimerValue*. If *TimerValue* does not increment at a predictable rate, then 0 is returned. The amount of time that has passed between two calls to **GetTimerValue()** can be calculated with the formula (*TimerValue2* - *TimerValue1*) * *TimerPeriod*. This parameter is optional and may be **NULL**.

Description

This function reads the processor timer specified by *TimerIndex* and returns it in *TimerValue*. If *TimerValue* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. If *TimerPeriod* is not **NULL**, then the amount of time that passes in femtoseconds (10^{-15}) for each increment if *TimerValue* is returned in *TimerPeriod*. If the timer does not run at a predictable rate, then a *TimerPeriod* of 0 is returned. If *TimerIndex* does not specify a valid timer in this processor, then **EFI_INVALID_PARAMETER** is returned. The valid range for *TimerIndex* is 0..*NumberOfTimers*-1. If the processor does not contain any readable timers, then this function returns **EFI_UNSUPPORTED**. If an error occurs attempting to read one of the processor's timers, then **EFI_DEVICE_ERROR** is returned.

Status Codes Returned

EFI_SUCCESS	The processor timer value specified by *TimerIndex* was returned in *TimerValue*.
EFI_INVALID_PARAMETER	*TimerValue* is **NULL**.
EFI_INVALID_PARAMETER	*TimerIndex* is not valid.
EFI_UNSUPPORTED	The processor does not have any readable timers.
EFI_DEVICE_ERROR	An error occurred attempting to read one of the processor's timers.

EFI_CPU_ARCH_PROTOCOL.SetMemoryAttributes()

Summary

Attempts to set the attributes for a memory range.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_SET_MEMORY_ATTRIBUTES) (
  IN  CONST EFI_CPU_ARCH_PROTOCOL  *This,
  IN  EFI_PHYSICAL_ADDRESS         BaseAddress,
  IN  UINT64                       Length,
  IN  UINT64                       Attributes
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

BaseAddress

The physical address that is the start address of a memory region. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The size in bytes of the memory region.

Attributes

The bit mask of attributes to set for the memory region. See the UEFI Boot Service **GetMemoryMap()** for the set of legal attribute bits.

Description

This function modifies the attributes for the memory region specified by *BaseAddress* and *Length* from their current attributes to the attributes specified by *Attributes*. If this modification of attributes succeeds, then **EFI_SUCCESS** is returned.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the memory range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If the attributes specified by *Attributes* are not supported for the memory region specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If the attributes for one or more bytes of the memory range specified by *BaseAddress* and *Length* cannot be modified because the current system policy does not allow them to be modified, then **EFI_ACCESS_DENIED** is returned.

If there are not enough system resources available to modify the attributes of the memory range, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The attributes were set for the memory region.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_UNSUPPORTED	The processor does not support one or more bytes of the memory resource range specified by *BaseAddress* and *Length*.
EFI_UNSUPPORTED	The bit mask of attributes is not support for the memory resource range specified by *BaseAddress* and *Length*.
EFI_ACCESS_DENIED	The attributes for the memory resource range specified by *BaseAddress* and *Length* cannot be modified.
EFI_OUT_OF_RESOURCES	There are not enough system resources to modify the attributes of the memory resource range.

12.4 Metronome Architectural Protocol

EFI_METRONOME_ARCH_PROTOCOL

Summary

Used to wait for ticks from a known time source in a platform. This protocol may be used to implement a simple version of the **Stall()** Boot Service. This protocol must be produced by a boot service or runtime DXE driver and may only be consumed by the DXE Foundation and DXE drivers that produce DXE Architectural Protocols.

GUID

```
#define EFI_METRONOME_ARCH_PROTOCOL_GUID  \
  {0x26baccb2,0x6f42,0x11d4,\
  0xbc,0xe7,0x0,0x80,0xc7,0x3c,0x88,0x81}
```

Protocol Interface Structure

```
typedef struct _EFI_METRONOME_ARCH_PROTOCOL {
  EFI_METRONOME_WAIT_FOR_TICK  WaitForTick;
  UINT32                       TickPeriod;
} EFI_METRONOME_ARCH_PROTOCOL;
```

Parameters

WaitForTick

Waits for a specified number of ticks from a known time source in the platform. See the **WaitForTick()** function description. The actual time passed between entry of this function and the first tick is between 0 and *TickPeriod* 100 ns units. To guarantee that at least *TickPeriod* time has elapsed, wait for two ticks.

TickPeriod

The period of platform's known time source in 100 ns units. This value on any platform must not exceed 200 µs. The value in this field is a constant that must not be modified after the Metronome architectural protocol is installed. All consumers must treat this as a read-only field.

Description

This protocol provides access to a known time source in the platform to the DXE Foundation. The DXE Foundation uses this known time source to produce DXE Foundation services that require calibrated delays.

EFI_METRONOME_ARCH_PROTOCOL.WaitForTick()

Summary

Waits for a specified number of ticks from a known time source in a platform.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_METRONOME_WAIT_FOR_TICK) (
  IN CONST EFI_METRONOME_ARCH_PROTOCOL  *This,
  IN UINT32                             TickNumber
  );
```

Parameters

This

The **EFI_METRONOME_ARCH_PROTOCOL** instance.

TickNumber

Number of ticks to wait.

Description

The **WaitForTick()** function waits for the number of ticks specified by *TickNumber* from a known time source in the platform. If *TickNumber* of ticks are detected, then **EFI_SUCCESS** is returned. The actual time passed between entry of this function and the first tick is between 0 and *TickPeriod* 100 ns units. If you want to guarantee that at least *TickPeriod* time has elapsed, wait for two ticks. This function waits for a hardware event to determine when a tick occurs. It is possible for interrupt processing, or exception processing to interrupt the execution of the **WaitForTick()** function. Depending on the hardware source for the ticks, it is possible for a tick to be missed. This function cannot guarantee that ticks will not be missed. If a timeout occurs waiting for the specified number of ticks, then **EFI_TIMEOUT** is returned.

Status Codes Returned

EFI_SUCCESS	The wait for the number of ticks specified by *TickNumber* succeeded.
EFI_TIMEOUT	A timeout occurred waiting for the specified number of ticks.

12.5 Monotonic Counter Architectural Protocol

EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL

Summary

Provides the services required to access the system's monotonic counter. This protocol must be produced by a runtime DXE driver and may only be consumed by the DXE Foundation and DXE drivers that produce DXE Architectural Protocols.

GUID

```
#define EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL_GUID  \
  {0x1da97072,0xbddc,0x4b30,\
  0x99,0xf1,0x72,0xa0,0xb5,0x6f,0xff,0x2a}
```

Description

The DXE driver that produces this protocol must be a runtime driver. This driver is responsible for initializing the **`GetNextHighMonotonicCount()`** field of the UEFI Runtime Services Table and the **`GetNextMonotonicCount()`** field of the UEFI Boot Services Table. See Services - Runtime Services and Services - Boot Services for details on these services. After the field of the UEFI Runtime Services Table and the field of the UEFI Boot Services Table have been initialized, the driver must install the **`EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL_GUID`** on a new handle with a **`NULL`** interface pointer. The installation of this protocol informs the DXE Foundation that the monotonic counter services are now available and that the DXE Foundation must update the 32-bit CRC of the UEFI Runtime Services Table and the 32-bit CRC of the UEFI Boot Services Table.

12.6 Real Time Clock Architectural Protocol

EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL

Summary

Provides the services required to access a system's real time clock hardware. This protocol must be produced by a runtime DXE driver and may only be consumed by the DXE Foundation.

GUID

```
#define EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL_GUID  \
  {0x27CFAC87,0x46CC,0x11d4,\
  0x9A,0x38,0x00,0x90,0x27,0x3F,0xC1,0x4D}
```

Description

The DXE driver that produces this protocol must be a runtime driver. This driver is responsible for initializing the **GetTime()**, **SetTime()**, **GetWakeupTime()**, and **SetWakeupTime()** fields of the UEFI Runtime Services Table. See "Runtime Capabilities" on page 31 for details on these services. After the four fields of the UEFI Runtime Services Table have been initialized, the driver must install the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL_GUID** on a new handle with a **NULL** interface pointer. The installation of this protocol informs the DXE Foundation that the real time clock–related services are now available and that the DXE Foundation must update the 32-bit CRC of the UEFI Runtime Services Table.

12.7 Reset Architectural Protocol

EFI_RESET_ARCH_PROTOCOL

Summary

Provides the service required to reset a platform. This protocol must be produced by a runtime DXE driver and may only be consumed by the DXE Foundation.

GUID

```
#define EFI_RESET_ARCH_PROTOCOL_GUID  \
  {0x27CFAC88,0x46CC,0x11d4,\
  0x9A,0x38,0x00,0x90,0x27,0x3F,0xC1,0x4D}
```

Description

The DXE driver that produces this protocol must be a runtime driver. This driver is responsible for initializing the **`ResetSystem()`** field of the UEFI Runtime Services Table. See "Runtime Capabilities" on page 31 for details on this service. After this field of the UEFI Runtime Services Table has been initialized, the driver must install the **`EFI_RESET_ARCH_PROTOCOL_GUID`** on a new handle with a **`NULL`** interface pointer. The installation of this protocol informs the DXE Foundation that the reset system service is now available and that the DXE Foundation must update the 32-bit CRC of the UEFI Runtime Services Table.

12.8 Runtime Architectural Protocol

The following topics provide a detailed description of the **EFI_RUNTIME_ARCH_PROTOCOL**. The DXE Foundation contains no runtime code, so all runtime code is contained in DXE Architectural Protocols. This is due to the fact that runtime code must be callable in physical or virtual mode. The Runtime Architectural Protocol contains the UEFI runtime services that are callable only in physical mode. The Runtime Architectural Protocol can be thought of as the runtime portion of the DXE Foundation.

The Runtime Architectural Protocol contains support for transition of runtime drivers from physical mode calling to virtual mode calling.

EFI_RUNTIME_ARCH_PROTOCOL

Summary

Allows the runtime functionality of the DXE Foundation to be contained in a separate driver. It also provides hooks for the DXE Foundation to export information that is needed at runtime. As such, this protocol allows services to the DXE Foundation to manage runtime drivers and events. This protocol also implies that the runtime services required to transition to virtual mode, **SetVirtualAddressMap()** and **ConvertPointer()**, have been registered into the UEFI Runtime Table in the UEFI System Table. This protocol must be produced by a runtime DXE driver and may only be consumed by the DXE Foundation.

GUID

```
#define EFI_RUNTIME_ARCH_PROTOCOL_GUID  \
  {0xb7dfb4e1,0x52f,0x449f,\
  0x87,0xbe,0x98,0x18,0xfc,0x91,0xb7,0x33}
```

Protocol Interface Structure

```
typedef struct _EFI_RUNTIME_ARCH_PROTOCOL {
EFI_LIST_ENTRY          ImageHead;
EFI_LIST_ENTRY          EventHead;
UINTN                   MemoryDescriptorSize;
UINT32                  MemoryDesciptorVersion;
UINTN                   MemoryMapSize;
EFI_MEMORY_DESCRIPTOR   *MemoryMapPhysical;
EFI_MEMORY_DESCRIPTOR   *MemoryMapVirtual;
BOOLEAN                 VirtualMode;
BOOLEAN                 AtRuntime;
} EFI_RUNTIME_ARCH_PROTOCOL;
```

Parameters

ImageHead

A list of type **EFI_RUNTIME_IMAGE_ENTRY** where the DXE Foundation inserts items into the list and the Runtime AP consumes the data to implement the **SetVirtualAddressMap()** call.

EventHead

A list of type **EFI_RUNTIME_EVENT_ENTRY** where the DXE Foundation inserts items into the list and the Runtime AP consumes the data to implement the **SetVirtualAddressMap()** call.

MemoryDescriptorSize

Size of a memory descriptor that is returned by **GetMemoryMap()**. This value is updated by the DXE Foundation.

MemoryDescriptorVersion

Version of a memory descriptor that is return by **GetMemoryMap()**. This value is updated by the DXE Foundation.

MemoryMapSize

Size of the memory map in bytes contained in **MemoryMapPhysical** and **MemoryMapVirtual**. This value is updated by the DXE Foundation when memory for *MemoryMapPhysical* gets allocated.

MemoryMapPhysical

Pointer to a runtime buffer that contains a copy of the memory map returned via **GetMemoryMap()**. The memory must be allocated by the DXE Foundation so that it is accounted for in the memory map.

MemoryMapVirtual

Pointer to *MemoryMapPhysical* that is updated to virtual mode after **SetVirtualAddressMap()**. The DXE Foundation updates this value when it updates *MemoryMapPhysical* with the same physical address. The Runtime AP is responsible for converting *MemoryMapVirtual* to a virtual pointer.

VirtualMode

Boolean that is **TRUE** if **SetVirtualAddressMap()** has been called. This field is set by the Runtime AP. When *VirtualMode* is **TRUE** *MemoryMapVirtual* pointer contains the virtual address of the *MemoryMapPhysical*.

AtRuntime

Boolean that is **TRUE** if **ExitBootServices()** has been called. This field is set by the Runtime AP.

Related Definitions

```
//*******************************************************
// EFI_LIST_ENTRY
//*******************************************************
struct _EFI_LIST_ENTRY {
```

```
  struct _EFI_LIST_ENTRY   *ForwardLink;
  struct _EFI_LIST_ENTRY   *BackLink;
} EFI_LIST_ENTRY;
```

ForwardLink

A pointer next node in the doubly linked list.

BackLink

A pointer previous node in the doubly linked list.

```
//*************************************************************
// EFI_RUNTIME_IMAGE_ENTRY
//*************************************************************
typedef struct {
   VOID              *ImageBase;
   UINT64            ImageSize;
   VOID              *RelocationData;
   EFI_HANDLE        Handle;
   EFI_LIST_ENTRY    Link;
} EFI_RUNTIME_IMAGE_ENTRY;
```

ImageBase

Start of image that has been loaded in memory. It is a pointer to either the DOS header or PE header of the image. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** UEFI 2.0 specification.

ImageSize

Size in bytes of the image represented by *ImageBase*.

RelocationData

Information about the fix-ups that were performed on *ImageBase* when it was loaded into memory. This information is needed when the virtual mode fix-ups are reapplied so that data that has been programmatically updated will not be fixed up. If code updates a global variable the code is responsible for fixing up the variable for virtual mode.

Handle

The *ImageHandle* passed into *ImageBase* when it was loaded. See **EFI_IMAGE_ENTRY_POINT** for the definition of *ImageHandle*.

Link

Entry for this node in the **EFI_RUNTIME_ARCHITECTURE_PROTOCOL.ImageHead** list.

```
//*******************************************************
// EFI_RUNTIME_EVENT_ENTRY
//*******************************************************
typedef struct {
  UINT32                    Type;
  EFI_TPL                   NotifyTpl;
  EFI_EVENT_NOTIFY          NotifyFunction;
  VOID                      *NotifyContext;
  EFI_EVENT                 *Event;
  EFI_LIST_ENTRY            Link;
}  EFI_RUNTIME_EVENT_ENTRY;
```

Parameters

Type

The same as *Type* passed into **CreateEvent()**.

NotifyTpl

The same as *NotifyTpl* passed into **CreateEvent()**. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

NotifyFunction

The same as *NotifyFunction* passed into **CreateEvent()**. Type **EFI_EVENT_NOTIFY** is defined in the **CreateEvent()** function description.

NotifyContext

The same as *NotifyContext* passed into **CreateEvent()**.

Event

The **EFI_EVENT** returned by **CreateEvent()**. Event must be in runtime memory. Type **EFI_EVENT** is defined in the **CreateEvent()** function description.

Link

Entry for this node in the **EFI_RUNTIME_ARCHITECTURE_PROTOCOL.EventHead** list.

Description

The DXE driver that produces this protocol must be a runtime driver. This driver is responsible for initializing the **SetVirtualAddressMap()** and **ConvertPointer()** fields of the UEFI Runtime Services Table and the **CalculateCrc32()** field of the UEFI Boot Services Table. See "Runtime Capabilities" on page 31 and "Services - Boot Services" on page 27 for details on these services. After the two fields of the UEFI Runtime Services Table and the one field of the UEFI Boot Services Table have been initialized, the driver must install the **EFI_RUNTIME_ARCH_PROTOCOL_GUID** on a new handle with an **EFI_RUNTIME_ARCH_PROTOCOL** interface pointer. The installation of this protocol informs the DXE Foundation that the virtual memory services and the 32-bit CRC services are now available, and the DXE Foundation must update the 32-bit CRC of the UEFI Runtime Services Table and the 32-bit CRC of the UEFI Boot Services Table.

All runtime DXE Foundation services are provided by the **EFI_RUNTIME_ARCH_PROTOCOL**. This includes the support for registering runtime images that must be fixed up again when a transition is made from physical mode to virtual mode. This protocol also supports all events that are defined to fire at runtime. This protocol also contains a CRC-32 function that will be used by the DXE Foundation as a boot service. The **EFI_RUNTIME_ARCH_PROTOCOL** needs the CRC-32 function when a transition is made from physical mode to virtual mode and the UEFI System Table and UEFI Runtime Table are fixed up with virtual pointers.

12.9 Security Architectural Protocol

EFI_SECURITY_ARCH_PROTOCOL

Summary

Abstracts security-specific functions from the DXE Foundation. This protocol must be produced by a boot service or runtime DXE driver and may only be consumed by the DXE Foundation and any other DXE drivers that need to validate the authentication of files.

GUID

```
#define EFI_SECURITY_ARCH_PROTOCOL_GUID  \
  {0xA46423E3,0x4617,0x49f1,\
  0xB9,0xFF,0xD1,0xBF,0xA9,0x11,0x58,0x39}
```

Protocol Interface Structure

```
typedef struct _EFI_SECURITY_ARCH_PROTOCOL {
  EFI_SECURITY_FILE_AUTHENTICATION_STATE
                                  FileAuthenticationState;
} EFI_SECURITY_ARCH_PROTOCOL;
```

Parameters

FileAuthenticationState

This service is called upon fault with respect to the authentication of a section of a file. See the **FileAuthenticationState()** function description.

Description

The **EFI_SECURITY_ARCH_PROTOCOL** is used to abstract platform-specific policy from the DXE Foundation. This includes locking flash upon failure to authenticate, attestation logging, and other exception operations.

The driver that produces the **EFI_SECURITY_ARCH_PROTOCOL** may also optionally install the **EFI_SECURITY_POLICY_PROTOCOL_GUID** onto a new handle with a **NULL** interface. The existence of this GUID in the protocol database means that the GUIDed Section Extraction Protocol should authenticate the contents of an Authentication Section. The expectation is that the GUIDed Section Extraction protocol will look for the existence of the **EFI_SECURITY_POLICY_PROTOCOL_GUID** in the protocol database. If it exists, then the publication thereof is taken as an injunction to attempt an authentication of any section wrapped in an Authentication Section. See the *Platform Initialization Specification,* Volume 3, for details on the GUIDed Section Extraction Protocol and Authentication Sections.

Additional GUID Definitions

```
#define EFI_SECURITY_POLICY_PROTOCOL_GUID  \

{0x78E4D245,0xCD4D,0x4a05,0xA2,0xBA,0x47,0x43,0xE8,0x6C,0xFC,0xA
B}
```

EFI_SECURITY_ARCH_PROTOCOL.FileAuthenticationState()

Summary

The DXE Foundation uses this service to check the authentication status of a file. This allows the system to execute a platform-specific policy in response the different authentication status values.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SECURITY_FILE_AUTHENTICATION_STATE) (
  IN  CONST EFI_SECURITY_ARCH_PROTOCOL  *This,
  IN  UINT32                            AuthenticationStatus,
  IN  CONST EFI_DEVICE_PATH_PROTOCOL    *File
  );
```

Parameters

This

The **EFI_SECURITY_ARCH_PROTOCOL** instance.

AuthenticationStatus

The authentication type returned from the Section Extraction Protocol. See the *Platform Initialization Specification*, Volume 3, for details on this type.

File

A pointer to the device path of the file that is being dispatched. This will optionally be used for logging. Type **EFI_DEVICE_PATH_PROTOCOL** is defined Chapter 8 of the UEFI 2.0 specification.

Description

The **EFI_SECURITY_ARCH_PROTOCOL** (SAP) is used to abstract platform-specific policy from the DXE Foundation response to an attempt to use a file that returns a given status for the authentication check from the section extraction protocol.

The possible responses in a given SAP implementation may include locking flash upon failure to authenticate, attestation logging for all signed drivers, and other exception operations. The *File* parameter allows for possible logging within the SAP of the driver.

If *File* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If the file specified by *File* with an authentication status specified by *AuthenticationStatus* is safe for the DXE Foundation to use, then **EFI_SUCCESS** is returned.

If the file specified by *File* with an authentication status specified by *AuthenticationStatus* is not safe for the DXE Foundation to use under any circumstances, then **EFI_ACCESS_DENIED** is returned.

If the file specified by *File* with an authentication status specified by *AuthenticationStatus* is not safe for the DXE Foundation to use right now, but it might be possible to use it at a future time, then **EFI_SECURITY_VIOLATION** is returned.

Status Codes Returned

EFI_SUCCESS	The file specified by *File* did authenticate, and the platform policy dictates that the DXE Foundation may use *File*.
EFI_INVALID_PARAMETER	*File* is **NULL**.
EFI_SECURITY_VIOLATION	The file specified by *File* did not authenticate, and the platform policy dictates that *File* should be placed in the untrusted state. A file may be promoted from the untrusted to the trusted state at a future time with a call to the **Trust()** DXE Service.
EFI_ACCESS_DENIED	The file specified by *File* did not authenticate, and the platform policy dictates that *File* should not be used for any purpose.

12.10 Timer Architectural Protocol

EFI_TIMER_ARCH_PROTOCOL

Summary

Used to set up a periodic timer interrupt using a platform specific timer, and a processor-specific interrupt vector. This protocol enables the use of the **SetTimer()** Boot Service. This protocol must be produce by a boot service or runtime DXE driver and may only be consumed by the DXE Foundation or DXE drivers that produce other DXE Architectural Protocols.

GUID

```
#define EFI_TIMER_ARCH_PROTOCOL_GUID  \
  {0x26baccb3,0x6f42,0x11d4,\
  0xbc,0xe7,0x0,0x80,0xc7,0x3c,0x88,0x81}
```

Protocol Interface Structure

```
typedef struct _EFI_TIMER_ARCH_PROTOCOL {
  EFI_TIMER_REGISTER_HANDLER          RegisterHandler;
  EFI_TIMER_SET_TIMER_PERIOD          SetTimerPeriod;
  EFI_TIMER_GET_TIMER_PERIOD          GetTimerPeriod;
  EFI_TIMER_GENERATE_SOFT_INTERRUPT   GenerateSoftInterrupt;
} EFI_TIMER_ARCH_PROTOCOL;
```

Parameters

RegisterHandler

Registers a handler that will be called each time the timer interrupt fires. See the **RegisterHandler()** function description. *TimerPeriod* defines the minimum time between timer interrupts, so *TimerPeriod* will also be the minimum time between calls to the registered handler.

SetTimerPeriod

Sets the period of the timer interrupt in 100 ns units. See the **SetTimerPeriod()** function description. This function is optional and may return **EFI_UNSUPPORTED**. If this function is supported, then the timer period will be rounded up to the nearest supported timer period.

GetTimerPeriod

Retrieves the period of the timer interrupt in 100 ns units. See the **GetTimerPeriod()** function description.

GenerateSoftInterrupt

Generates a soft timer interrupt that simulates the firing of the timer interrupt. This service can be used to invoke the registered handler if the timer interrupt has been masked for a period of time. See the **GenerateSoftInterrupt()** function description.

Description

This protocol provides the services to initialize a periodic timer interrupt and to register a handler that is called each time the timer interrupt fires. It may also provide a service to adjust the rate of the periodic timer interrupt. When a timer interrupt occurs, the handler is passed the amount of time that has passed since the previous timer interrupt.

EFI_TIMER_ARCH_PROTOCOL.RegisterHandler()

Summary

Registers a handler that is called each timer the timer interrupt fires.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_TIMER_REGISTER_HANDLER) (
  IN CONST EFI_TIMER_ARCH_PROTOCOL  *This,
  IN EFI_TIMER_NOTIFY               NotifyFunction
  );
```

Parameters

This

The **EFI_TIMER_ARCH_PROTOCOL** instance.

NotifyFunction

The function to call when a timer interrupt fires. This function executes at **EFI_TPL_HIGH_LEVEL**. The DXE Foundation will register a handler for the timer interrupt, so it can know how much time has passed. This information is used to signal timer based events. **NULL** will unregister the handler. Type **EFI_TIMER_NOTIFY** is defined in "Related Definitions" below.

Description

This function registers the handler *NotifyFunction* so it is called every time the timer interrupt fires. It also passes the amount of time since the last handler call to the *NotifyFunction*. If *NotifyFunction* is **NULL**, then the handler is unregistered. If the handler is registered, then **EFI_SUCCESS** is returned. If the processor does not support registering a timer interrupt handler, then **EFI_UNSUPPORTED** is returned. If an attempt is made to register a handler when a handler is already registered, then **EFI_ALREADY_STARTED** is returned. If an attempt is made to unregister a handler when a handler is not registered, then **EFI_INVALID_PARAMETER** is returned. If an error occurs attempting to register the *NotifyFunction* with the timer interrupt, then **EFI_DEVICE_ERROR** is returned.

Related Definitions

```
typedef
VOID
(EFIAPI *EFI_TIMER_NOTIFY) (
  IN UINT64   Time
  );
```

Paramters

Time

Time since the last timer interrupt in 100 ns units. This will typically be *TimerPeriod*, but if a timer interrupt is missed, and the **EFI_TIMER_ARCH_PROTOCOL** driver can detect missed interrupts, then *Time* will contain the actual amount of time since the last interrupt.

Status Codes Returned

EFI_SUCCESS	The timer handler was registered.
EFI_UNSUPPORTED	The platform does not support timer interrupts.
EFI_ALREADY_STARTED	*NotifyFunction* is not **NULL**, and a handler is already registered.
EFI_INVALID_PARAMETER	*NotifyFunction* is **NULL**, and a handler was not previously registered.
EFI_DEVICE_ERROR	The timer handler could not be registered.

EFI_TIMER_ARCH_PROTOCOL.SetTimerPeriod()

Summary

Sets the rate of the periodic timer interrupt.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_TIMER_SET_TIMER_PERIOD) (
  IN  CONST EFI_TIMER_ARCH_PROTOCOL  *This,
  IN  UINT64                         TimerPeriod
  );
```

Parameters

This

The **EFI_TIMER_ARCH_PROTOCOL** instance.

TimerPeriod

The rate to program the timer interrupt in 100 ns units. If the timer hardware is not programmable, then **EFI_UNSUPPORTED** is returned. If the timer is programmable, then the timer period will be rounded up to the nearest timer period that is supported by the timer hardware. If *TimerPeriod* is set to 0, then the timer interrupts will be disabled.

Description

This function adjusts the period of timer interrupts to the value specified by *TimerPeriod*. If the timer period is updated, then **EFI_SUCCESS** is returned. If the timer hardware is not programmable, then **EFI_UNSUPPORTED** is returned. If an error occurs while attempting to update the timer period, then the timer hardware will be put back in its state prior to this call, and **EFI_DEVICE_ERROR** is returned. If *TimerPeriod* is 0, then the timer interrupt is disabled. This is not the same as disabling the processor's interrupts. Instead, it must either turn off the timer hardware, or it must adjust the interrupt controller so that a processor interrupt is not generated when the timer interrupt fires.

Status Codes Returned

EFI_SUCCESS	The timer period was changed.
EFI_UNSUPPORTED	The platform cannot change the period of the timer interrupt.
EFI_DEVICE_ERROR	The timer period could not be changed due to a device error.

EFI_TIMER_ARCH_PROTOCOL.GetTimerPeriod()

Summary

Retrieves the rate of the periodic timer interrupt.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_TIMER_GET_TIMER_PERIOD) (
  IN  CONST EFI_TIMER_ARCH_PROTOCOL  *This,
  OUT UINT64                         *TimerPeriod
  );
```

Parameters

This

The **EFI_TIMER_ARCH_PROTOCOL** instance.

TimerPeriod

A pointer to the timer period to retrieve in 100 ns units. If 0 is returned, then the timer is currently disabled.

Description

This function retrieves the period of timer interrupts in 100 ns units, returns that value in *TimerPeriod*, and returns **EFI_SUCCESS**. If *TimerPeriod* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. If a *TimerPeriod* of 0 is returned, then the timer is currently disabled.

Status Codes Returned

EFI_SUCCESS	The timer period was returned in *TimerPeriod*.
EFI_INVALID_PARAMETER	*TimerPeriod* is **NULL**.

EFI_TIMER_ARCH_PROTOCOL.GenerateSoftInterrupt()

Summary

Generates a soft timer interrupt.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_TIMER_GENERATE_SOFT_INTERRUPT) (
  IN  CONST EFI_TIMER_ARCH_PROTOCOL     *This
  );
```

Parameters

This

The **EFI_TIMER_ARCH_PROTOCOL** instance.

Description

This function generates a soft timer interrupt. If the platform does not support soft timer interrupts, then **EFI_UNSUPPORTED** is returned. Otherwise, **EFI_SUCCESS** is returned. If a handler has been registered through the **EFI_TIMER_ARCH_PROTOCOL.RegisterHandler()** service, then a soft timer interrupt will be generated. If the timer interrupt is enabled when this service is called, then the registered handler will be invoked. The registered handler should not be able to distinguish a hardware-generated timer interrupt from a software-generated timer interrupt.

Status Codes Returned

EFI_SUCCESS	The soft timer interrupt was generated.
EFI_UNSUPPORTED	The platform does not support the generation of soft timer interrupts.

12.11 Variable Architectural Protocol

EFI_VARIABLE_ARCH_PROTOCOL

Summary

Provides the services required to get and set environment variables. This protocol must be produced by a runtime DXE driver and may be consumed only by the DXE Foundation.

GUID

```
#define EFI_VARIABLE_ARCH_PROTOCOL_GUID  \
  {0x1e5668e2,0x8481,0x11d4,\
  0xbc,0xf1,0x0,0x80,0xc7,0x3c,0x88,0x81}
```

Description

The DXE driver that produces this protocol must be a runtime driver. This driver is responsible for initializing the **GetVariable()**, **GetNextVariableName()**, **SetVariable()** and **QueryVariableInfo()** fields of the UEFI Runtime Services Table. See "Runtime Capabilities" on page 31 for details on these services. After the three fields of the UEFI Runtime Services Table have been initialized, the driver must install the **EFI_VARIABLE_ARCH_PROTOCOL_GUID** on a new handle with a NULL interface pointer. The installation of this protocol informs the DXE Foundation that the read-only and the volatile environment variable related services are now available and that the DXE Foundation must update the 32-bit CRC of the UEFI Runtime Services Table. The full complement of environment variable services are not available until both this protocol and **EFI_VARIABLE_WRITE_ARCH_PROTOCOL** are installed. DXE drivers that require read-only access or read/write access to volatile environment variables must have this architectural protocol in their dependency expressions. DXE drivers that require write access to nonvolatile environment variables must have the **EFI_VARIABLE_WRITE_ARCH_PROTOCOL** in their dependency expressions.

12.12 Variable Write Architectural Protocol

EFI_VARIABLE_WRITE_ARCH_PROTOCOL

Summary

Provides the services required to set nonvolatile environment variables. This protocol must be produced by a runtime DXE driver and may be consumed only by the DXE Foundation.

GUID

```
#define EFI_VARIABLE_WRITE_ARCH_PROTOCOL_GUID  \
  {0x6441f818,0x6362,0x4e44,\
  0xb5,0x70,0x7d,0xba,0x31,0xdd,0x24,0x53}
```

Description

The DXE driver that produces this protocol must be a runtime driver. This driver may update the **SetVariable()** field of the UEFI Runtime Services Table. See "Runtime Capabilities" on page 31 for details on this service. After the UEFI Runtime Services Table has been initialized, the driver must install the **EFI_VARIABLE_WRITE_ARCH_PROTOCOL_GUID** on a new handle with a NULL interface pointer. The installation of this protocol informs the DXE Foundation that the write services for nonvolatile environment variables are now available and that the DXE Foundation must update the 32-bit CRC of the UEFI Runtime Services Table. The full complement of environment variable services are not available until both this protocol and **EFI_VARIABLE_ARCH_PROTOCOL** are installed. DXE drivers that require read-only access or read/write access to volatile environment variables must have the **EFI_VARIABLE_WRITE_ARCH_PROTOCOL** in their dependency expressions. DXE drivers that require write access to nonvolatile environment variables must have this architectural protocol in their dependency expressions.

12.13 EFI Capsule Architectural Protocol

EFI_CAPSULE_ARCH_PROTOCOL

Summary

Provides the services for capsule update.

GUID

```
#define EFI_CAPSULE_ARCH_PROTOCOL_GUID \
  { 0x5053697e, 0x2cbc, 0x4819, \
  0x90, 0xd9, 0x5, 0x80, 0xde, 0xee, 0x57, 0x54 }
```

Description

The DXE Driver that produces this protocol must be a runtime driver. The driver is responsible for initializing the **CapsuleUpdate()** and **QueryCapsuleCapabilities()** fields of the UEFI Runtime Services Table. After the two fields of the UEFI Runtime Services Table have been

initialized, the driver must install the **EFI_CAPSULE_ARCH_PROTOCOL_GUID** on a new handle with a **NULL** interface pointer. The installation of this protocol informs the DXE Foundation that the Capsule related services are now available and that the DXE Foundation must update the 32-bit CRC of the UEFI Runtime Services Table.

12.14 Watchdog Timer Architectural Protocol

The following topics provide a detailed description of the **EFI_WATCHDOG_TIMER_ARCH_PROTOCOL**. This protocol is used to implement the Boot Service **SetWatchdogTimer()**. The watchdog timer may be implemented in software using Boot Services, or it may be implemented with specialized hardware. The protocol provides a service to register a handler when the watchdog timer fires and a service to set the amount of time to wait before the watchdog timer is fired.

EFI_WATCHDOG_TIMER_ARCH_PROTOCOL

Summary

Used to program the watchdog timer and optionally register a handler when the watchdog timer fires. This protocol must be produced by a boot service or runtime DXE driver and may be consumed only by the DXE Foundation or DXE drivers that produce other DXE Architectural Protocols. If a platform wishes to perform a platform-specific action when the watchdog timer expires, then the DXE driver that contains the implementation of the **EFI_BDS_ARCH_PROTOCOL** should use this protocol's **RegisterHandler()** service.

GUID

```
#define EFI_WATCHDOG_TIMER_ARCH_PROTOCOL_GUID  \
  {0x665E3FF5,0x46CC,0x11d4,\
  0x9A,0x38,0x00,0x90,0x27,0x3F,0xC1,0x4D}
```

Protocol Interface Structure

```
typedef struct _EFI_WATCHDOG_TIMER_ARCH_PROTOCOL {
  EFI_WATCHDOG_TIMER_REGISTER_HANDLER  RegisterHandler;
  EFI_WATCHDOG_TIMER_SET_TIMER_PERIOD  SetTimerPeriod;
  EFI_WATCHDOG_TIMER_GET_TIMER_PERIOD  GetTimerPeriod;
} EFI_WATCHDOG_TIMER_ARCH_PROTOCOL;
```

Parameters

RegisterHandler

Registers a handler that is invoked when the watchdog timer fires. See the **RegisterHandler()** function description.

SetTimerPeriod

Sets the amount of time in 100 ns units to wait before the watchdog timer is fired. See the **SetTimerPeriod()** function description. If this function is supported, then the watchdog timer period will be rounded up to the nearest supported watchdog timer period.

GetTimerPeriod

Retrieves the amount of time in 100 ns units that the system will wait before the watchdog timer is fired. See the **GetTimerPeriod()** function description.

Description

This protocol provides the services required to implement the Boot Service **SetWatchdogTimer()**. It provides a service to set the amount of time to wait before firing the watchdog timer, and it also provides a service to register a handler that is invoked when the watchdog timer fires. This protocol can implement the watchdog timer by using the event and timer Boot Services, or it can make use of custom hardware. When the watchdog timer fires, control will be passed to a handler if one has been registered. If no handler has been registered, or the registered handler returns, then the system will be reset by calling the Runtime Service **ResetSystem()**.

EFI_WATCHDOG_TIMER_ARCH_PROTOCOL.RegisterHandler()

Summary

Registers a handler that is to be invoked when the watchdog timer fires.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_WATCHDOG_TIMER_REGISTER_HANDLER) (
  IN CONST EFI_WATCHDOG_TIMER_ARCH_PROTOCOL  *This,
  IN EFI_WATCHDOG_TIMER_NOTIFY               NotifyFunction
  );
```

Parameters

This

The **EFI_WATCHDOG_TIMER_ARCH_PROTOCOL** instance.

NotifyFunction

The function to call when the watchdog timer fires. If this is **NULL**, then the handler will be unregistered. Type **EFI_WATCHDOG_TIMER_NOTIFY** is defined in "Related Definitions" below.

Description

This function registers a handler that is to be invoked when the watchdog timer fires. By default, **EFI_WATCHDOG_TIMER_ARCH_PROTOCOL** will call the Runtime Service **ResetSystem()** when the watchdog timer fires. If a *NotifyFunction* is registered, then *NotifyFunction* will be called before the Runtime Service **ResetSystem()** is called. If *NotifyFunction* is **NULL**, then the watchdog handler is unregistered. If a watchdog handler is registered, then **EFI_SUCCESS** is returned. If an attempt is made to register a handler when a handler is already registered, then **EFI_ALREADY_STARTED** is returned. If an attempt is made to uninstall a handler when a handler is not installed, then return **EFI_INVALID_PARAMETER**.

Related Definitions

```
typedef
VOID
(EFIAPI *EFI_WATCHDOG_TIMER_NOTIFY) (
  IN UINT64   Time
  );
```

Time

The time in 100 ns units that has passed since the watchdog timer was armed. For the notify function to be called, this must be greater than *TimerPeriod*.

Status Codes Returned

EFI_SUCCESS	The watchdog timer handler was registered or unregistered.

EFI_ALREADY_STARTED	*NotifyFunction* is not **NULL**, and a handler is already registered.
EFI_INVALID_PARAMETER	*NotifyFunction* is **NULL**, and a handler was not previously registered.

EFI_WATCHDOG_TIMER_ARCH_PROTOCOL.SetTimerPeriod()

Summary

Sets the amount of time in the future to fire the watchdog timer.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_WATCHDOG_TIMER_SET_TIMER_PERIOD) (
  IN CONST EFI_WATCHDOG_TIMER_ARCH_PROTOCOL  *This,
  IN UINT64                                  TimerPeriod
  );
```

Parameters

This

The **EFI_WATCHDOG_TIMER_ARCH_PROTOCOL** instance.

TimerPeriod

The amount of time in 100 ns units to wait before the watchdog timer is fired. If *TimerPeriod* is zero, then the watchdog timer is disabled.

Description

This function sets the amount of time to wait before firing the watchdog timer to *TimerPeriod* 100 ns units. If *TimerPeriod* is zero, then the watchdog timer is disabled.

Status Codes Returned

EFI_SUCCESS	The watchdog timer has been programmed to fire in *Time* 100 ns units.
EFI_DEVICE_ERROR	A watchdog timer could not be programmed due to a device error.

EFI_WATCHDOG_TIMER_ARCH_PROTOCOL.GetTimerPeriod()

Summary

Retrieves the amount of time in 100 ns units that the system will wait before firing the watchdog timer.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_WATCHDOG_TIMER_GET_TIMER_PERIOD) (
  IN  CONST EFI_WATCHDOG_TIMER_ARCH_PROTOCOL  *This,
  OUT UINT64                                  *TimerPeriod
  );
```

Parameters

This

The **EFI_WATCHDOG_TIMER_ARCH_PROTOCOL** instance.

TimerPeriod

A pointer to the amount of time in 100 ns units that the system will wait before the watchdog timer is fired. If *TimerPeriod* of zero is returned, then the watchdog timer is disabled.

Description

This function retrieves the amount of time the system will wait before firing the watchdog timer. This period is returned in *TimerPeriod*, and **EFI_SUCCESS** is returned. If *TimerPeriod* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

Status Codes Returned

EFI_SUCCESS	The amount of time that the system will wait before firing the watchdog timer was returned in *TimerPeriod*.
EFI_INVALID_PARAMETER	*TimerPeriod* is **NULL**.

13
DXE Boot Services Protocol

13.1 Overview

This chapter defines the services required for the Multiprocessor (MP) Services Protocol of Platform Initialization Specification.

This specification does the following:

- •Describes the basic components of the MP Services Protocol
- •Provides code definitions for the MP Services Protocol and the MP-related type definitions.

13.2 Conventions and Abbreviations

The following terms are used throughout this specification.

AP

Application processor. All other processors in a computer system other than the boot-strap processor are called application processors.

BSP

Boot-strap processor. A processor in an MP platform that is chosen to execute the modules that are necessary for booting the system. It is not necessary that the same processor that was selected earlier as a BSP shall remain a BSP throughout an entire boot session.

DXE

Driver Execute Environment. Environment to support running modular code in the form of EFI drivers; common to all platforms; typically in C language.

EFI

Extensible Firmware Interface – the specification containing interface definitions for firmware. This includes both interfaces used by the operating system for booting as well as interfaces that are used for internal construction of firmware.

MP

Multiprocessor.

13.3 MP Services Protocol Overview

The MP Services Protocol provides a generalized way of performing following tasks:

- •Retrieving information of multi-processor environment and MP-related status of specific processors.
- •Dispatching user-provided function to APs.

- •Maintain MP-related processor status.

The MP Services Protocol must be produced on any system with more than one logical processor.

The Protocol is available only during boot time.

MP Services Protocol is hardware-independent. Most of the logic of this protocol is architecturally neutral. It abstracts the multi-processor environment and status of processors, and provides interfaces to retrieve information, maintain, and dispatch.

MP Services Protocol may be consumed by ACPI module. The ACPI module may use this protocol to retrieve data that are needed for an MP platform and report them to OS.

MP Services Protocol may also be used to program and configure processors, such as MTRR synchronization for memory space attributes setting in DXE Services.

MP Services Protocol may be used by non-CPU DXE drivers to speed up platform boot by taking advantage of the processing capabilities of the APs, for example, using APs to help test system memory in parallel with other device initialization.

Diagnostics applications may also use this protocol for multi-processor.

13.4 MP Services Protocol

This section contains the basic definitions of the MP Services Protocol.

EFI_MP_SERVICES_PROTOCOL

Summary

When installed, the MP Services Protocol produces a collection of services that are needed for MP management.

GUID

```
#define EFI_MP_SERVICES_PROTOCOL_GUID \
    {0x3fdda605,0xa76e,0x4f46,{0xad,0x29,0x12,0xf4,0x53,0x1b,
0x3d,0x08}}
```

Protocol Interface Structure

```
typedef struct _EFI_MP_SERVICES_PROTOCOL {
 EFI_MP_SERVICES_GET_NUMBER_OF_PROCESSORS GetNumberOfProcessors;
 EFI_MP_SERVICES_GET_PROCESSOR_INFO       GetProcessorInfo;
 EFI_MP_SERVICES_STARTUP_ALL_APS          StartupAllAPs;
 EFI_MP_SERVICES_STARTUP_THIS_AP          StartupThisAP;
 EFI_MP_SERVICES_SWITCH_BSP               SwitchBSP;
 EFI_MP_SERVICES_ENABLEDISABLEAP          EnableDisableAP;
 EFI_MP_SERVICES_WHOAMI                   WhoAmI;
} EFI_MP_SERVICES_PROTOCOL;
```

Parameters

GetNumberOfProcessors

Gets the number of logical processors and the number of enabled logical processors in the system.

GetProcessorInfo

Gets detailed information on the requested processor at the instant this call is made.

StartupAllAPs

Starts up all the enabled APs in the system to run the function provided by the caller.

StartupThisAP

Starts up the requested AP to run the function provided by the caller.

SwitchBSP

Switches the requested AP to be the BSP from that point onward. This service changes the BSP for all purposes.

EnableDisableAP

Enables and disables the given AP from that point onward.

WhoAmI

Gets the handle number of the caller processor.

Description

The MP Services Protocol must be produced on any system with more than one logical processor. Before the UEFI event **EFI_EVENT_LEGACY_BOOT_GUID** or **EFI_EVENT_GROUP_EXIT_BOOT_SERVICES** is signaled, the module that produces this protocol is required to place all APs into an idle state whenever the APs are disabled or the APs are not executing code as requested through the **StartupAllAPs()** or **StartupThisAP()** services. The idle state of an AP is implementation dependent before the UEFI event **EFI_EVENT_LEGACY_BOOT_GUID** or **EFI_EVENT_GROUP_EXIT_BOOT_SERVICES** is signaled.

After the UEFI event **EFI_EVENT_LEGACY_BOOT_GUID** or **EFI_EVENT_GROUP_EXIT_BOOT_SERVICES** is signaled, all the APs must be placed in the OS compatible CPU state as defined by the UEFI Specification. Implementations of this protocol may use the UEFI event **EFI_EVENT_LEGACY_BOOT_GUID** or **EFI_EVENT_GROUP_EXIT_BOOT_SERVICES** to force APs into the OS compatible state as defined by the UEFI Specification. Modules that use this protocol must guarantee that all non-blocking mode requests on all APs have been completed before the UEFI event **EFI_EVENT_LEGACY_BOOT_GUID** or **EFI_EVENT_GROUP_EXIT_BOOT_SERVICES** is signaled. Since the order that event notification functions in the same event group are executed is not deterministic, an event of type **EFI_EVENT_LEGACY_BOOT_GUID** or **EFI_EVENT_GROUP_EXIT_BOOT_SERVICES** can not be used to guarantee that APs have completed their non-blocking mode requests.

EFI_MP_SERVICES_PROTOCOL.GetNumberOfProcessors()

Summary

This service retrieves the number of logical processor in the platform and the number of those logical processors that are currently enabled. This service may only be called from the BSP.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_GET_NUMBER_OF_PROCESSORS) (
  IN  EFI_MP_SERVICES_PROTOCOL  *This,
  OUT UINTN                     *NumberOfProcessors,
  OUT UINTN                     *NumberOfEnabledProcessors
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

NumberOfProcessors

Pointer to the total number of logical processors in the system, including the BSP and all enabled and disabled APs.

NumberOfEnabledProcessors

Pointer to the number of logical processors in the platform including the BSP that are currently enabled.

Description

This function is used to retrieve the following information:

- The number of logical processors that are present in the system
- The number of enabled logical processors in the system at the instant this call is made.

Since MP Service Protocol provides services to enable and disable processors dynamically, the number of enabled logical processors may vary during the course of a boot session.

This service may only be called from the BSP.

If this service is called from an AP, then **EFI_DEVICE_ERROR** is returned. If *NumberOfProcessors* or *NumberOfEnabledProcessors* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. Otherwise, the total number of processors is returned in *NumberOfProcessors*, the number of currently enabled processor is returned in *NumberOfEnabledProcessors*, and **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The number of logical processors and enabled logical processors was retrieved.
EFI_DEVICE_ERROR	The calling processor is an AP.

EFI_INVALID_PARAMETER	*NumberOfProcessors* is **NULL**
EFI_INVALID_PARAMETER	*NumberOfEnabledProcessors* is **NULL**

EFI_MP_SERVICES_PROTOCOL.GetProcessorInfo()

Summary

Gets detailed MP-related information on the requested processor at the instant this call is made. This service may only be called from the BSP.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_GET_PROCESSOR_INFO) (
  IN  EFI_MP_SERVICES_PROTOCOL   *This,
  IN  UINTN                       ProcessorNumber,
  OUT EFI_PROCESSOR_INFORMATION  *ProcessorInfoBuffer
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

ProcessorNumber

The handle number of processor. The range is from 0 to the total number of logical processors minus 1. The total number of logical processors can be retrieved by **EFI_MP_SERVICES_PROTOCOL.GetNumberOfProcessors()**.

ProcessorInfoBuffer

A pointer to the buffer where information for the requested processor is deposited. The buffer is allocated by the caller. Type **EFI_PROCESSOR_INFORMATION** is defined in "Related Definitions" below.

Description

This service retrieves detailed MP-related information about any processor on the platform. Note the following:

- The processor information may change during the course of a boot session.
- The data of information presented here is entirely MP related.

Information regarding the number of caches and their sizes, frequency of operation, slot numbers is all considered platform-related information and is not provided by this service.

This service may only be called from the BSP.

Related Definitions

```
//*******************************************************
// EFI_PROCESSOR_INFORMATION
//*******************************************************
typedef struct {
  UINT64                       ProcessorId;
```

```
  UINT32                      StatusFlag;
  EFI_CPU_PHYSICAL_LOCATION   Location;
} EFI_PROCESSOR_INFORMATION;
```

ProcessorId

The unique processor ID determined by system hardware.

For IA32 and X64, the processor ID is the same as the Local APIC ID. Only the lower 8 bits are used, and higher bits are reserved.

For IPF, the lower 16 bits contains id/eid, and higher bits are reserved.

StatusFlag

Flags indicating if the processor is BSP or AP, if the processor is enabled or disabled, and if the processor is healthy. The bit format is defined below.

Location

The physical location of the processor, including the physical package number that identifies the cartridge, the physical core number within package, and logical thread number within core. Type **EFI_PHYSICAL_LOCATION** is defined below.

```
//*******************************************************
// StatusFlag Bits Definition
//*******************************************************
#define PROCESSOR_AS_BSP_BIT          0x00000001
#define PROCESSOR_ENABLED_BIT         0x00000002
#define PROCESSOR_HEALTH_STATUS_BIT   0x00000004
```

PROCESSOR_AS_BSP_BIT

This bit indicates whether the processor is playing the role of BSP. If the bit is 1, then the processor is BSP. Otherwise, it is AP.

PROCESSOR_ENABLED_BIT

This bit indicates whether the processor is enabled. If the bit is 1, then the processor is enabled. Otherwise, it is disabled.

PROCESSOR_HEALTH_STATUS_BIT

This bit indicates whether the processor is healthy. If the bit is 1, then the processor is healthy. Otherwise, some fault has been detected for the processor.

Bits 3..31 are reserved and must be 0. The following table shows all the possible combinations of the *StatusFlag* bits:

Table 31. StatusFlag bits

BSP	ENABLED	HEALTH	Description
0	0	0	Unhealthy Disabled AP.
0	0	1	Healthy Disabled AP.
0	1	0	Unhealthy Enabled AP.

0	1	1	Healthy Enabled AP.
1	0	0	Invalid. The BSP can never be in the disabled state.
1	0	1	Invalid. The BSP can never be in the disabled state.
1	1	0	Unhealthy Enabled BSP.
1	1	1	Healthy Enabled BSP.

```
//*******************************************************
// EFI_CPU_PHYSICAL_LOCATION
//*******************************************************
typedef struct {
  UINT32  Package;
  UINT32  Core;
  UINT32  Thread;
} EFI_CPU_PHYSICAL_LOCATION;
```

Package

Zero-based physical package number that identifies the cartridge of the processor.

Core

Zero-based physical core number within package of the processor.

Thread

Zero-based logical thread number within core of the processor.

Status Codes Returned

EFI_SUCCESS	Processor information was returned.
EFI_DEVICE_ERROR	The calling processor is an AP.
EFI_INVALID_PARAMETER	*ProcessorInfoBuffer* is **NULL**.
EFI_NOT_FOUND	The processor with the handle specified by *ProcessorNumber* does not exist in the platform.

EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()

Summary

This service executes a caller provided function on all enabled APs. APs can run either simultaneously or one at a time in sequence. This service supports both blocking and non-blocking requests. The non-blocking requests use EFI events so the BSP can detect when the APs have finished. See "Non-blocking Execution Support" below for details. This service may only be called from the BSP.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_STARTUP_ALL_APS) (
  IN  EFI_MP_SERVICES_PROTOCOL *This,
  IN  EFI_AP_PROCEDURE         Procedure,
  IN  BOOLEAN                  SingleThread,
  IN  EFI_EVENT                WaitEvent              OPTIONAL,
  IN  UINTN                    TimeoutInMicroSeconds,
  IN  VOID                     *ProcedureArgument     OPTIONAL,
  OUT UINTN                    **FailedCpuList        OPTIONAL
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

Procedure

A pointer to the function to be run on enabled APs of the system. Type **EFI_AP_PROCEDURE** is defined in "Related Definitions" below.

SingleThread

If **TRUE**, then all the enabled APs execute the function specified by *Procedure* one by one, in ascending order of processor handle number.

If **FALSE**, then all the enabled APs execute the function specified by Procedure simultaneously.

WaitEvent

The event created by the caller with **CreateEvent()** service.

If it is **NULL**, then execute in blocking mode. BSP waits until all APs finish or *TimeoutInMicroSeconds* expires.

If it's not **NULL**, then execute in non-blocking mode. BSP requests the function specified by *Procedure* to be started on all the enabled APs, and go on executing immediately. If all return from *Procedure* or *TimeoutInMicroSeconds* expires, this event is signaled. The BSP can use the **CheckEvent()** or **WaitForEvent()** services to check the state of event.

Type **EFI_EVENT** is defined in **CreateEvent()** in the *Unified Extensible Firmware Interface Specification (Version 2.0).*

TimeoutInMicroseconds

Indicates the time limit in microseconds for APs to return from *Procedure*, either for blocking or non-blocking mode. Zero means infinity.

If the timeout expires before all APs return from *Procedure*, then *Procedure* on the failed APs is terminated. All enabled APs are available for next function assigned by **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()** or **EFI_MP_SERVICES_PROTOCOL.StartupThisAP().**

If the timeout expires in blocking mode, BSP returns **EFI_TIMEOUT**.

If the timeout expires in non-blocking mode, *WaitEvent* is signaled with **SignalEvent()**.

ProcedureArgument

The parameter passed into *Procedure* for all APs.

FailedCpuList

If **NULL**, this parameter is ignored.

Otherwise, if all APs finish successfully, then its content is set to **NULL**. If not all APs finish before timeout expires, then its content is set to address of the buffer holding handle numbers of the failed APs. The buffer is allocated by MP Service Protocol, and it's the caller's responsibility to free the buffer with **FreePool()** service.

In blocking mode, it is ready for consumption when the call returns. In non-blocking mode, it is ready when *WaitEvent* is signaled.

The list of failed CPU is terminated by **END_OF_CPU_LIST**. It is defined in "Related Definitions" below.

Description

This function is used to dispatch all the enabled APs to the function specified by *Procedure*.

If any enabled AP is busy, then **EFI_NOT_READY** is returned immediately and *Procedure* is not started on any AP.

If *SingleThread* is **TRUE**, all the enabled APs execute the function specified by *Procedure* one by one, in ascending order of processor handle number. Otherwise, all the enabled APs execute the function specified by *Procedure* simultaneously.

If *WaitEvent* is **NULL**, execution is in blocking mode. The BSP waits until all APs finish or *TimeoutInMicroSecs* expires. Otherwise, execution is in non-blocking mode, and the BSP returns from this service without waiting for APs. If a non-blocking mode is requested after the UEFI Event **EFI_EVENT_GROUP_READY_TO_BOOT** is signaled, then **EFI_UNSUPPORTED** must be returned.

If the timeout specified by *TimeoutInMicroseconds* expires before all APs return from *Procedure*, then *Procedure* on the failed APs is terminated. All enabled APs are always available for further calls to **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()** and **EFI_MP_SERVICES_PROTOCOL.StartupThisAP().** If *FailedCpuList* is not **NULL**, its content points to the list of processor handle numbers in which Procedure was terminated.

This service may only be called from the BSP.

Note: *It is the responsibility of the consumer of the* **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()** *to make sure that the nature of the code that is executed on the BSP and the dispatched APs is well controlled. The MP Services Protocol does not guarantee that the* *Procedure* *function is MP-safe. Hence, the tasks that can be run in parallel are limited to certain independent tasks and well-controlled exclusive code. EFI services and protocols may not be called by APs unless otherwise specified.*

Related Definitions

```
#define END_OF_CPU_LIST    0xffffffff

typedef
VOID
(EFIAPI *EFI_AP_PROCEDURE) (
  IN VOID   *ProcedureArgument
);
```

ProcedureArgument

Pointer to the procedure's argument

Non-Blocking Execution Support

The following usage guidelines must be followed for non-blocking execution support.

In blocking execution mode, BSP waits until all APs finish or *TimeoutInMicroSeconds* expires.

In non-blocking execution mode, BSP is freed to return to the caller and then proceed to the next task without having to wait for APs. The following sequence needs to occur in a non-blocking execution mode:

1. The caller that intends to use this MP Services Protocol in non-blocking mode creates *WaitEvent* by calling the EFI **CreateEvent()** service.

The caller invokes **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()**. If the parameter *WaitEvent* is not **NULL**, then **StartupAllAPs()** executes in non-blocking mode. It requests the function specified by *Procedure* to be started on all the enabled APs, and releases the BSP to continue with other tasks.

2. The caller can use the **CheckEvent()** and **WaitForEvent()** services to check the state of the *WaitEvent* created in step 1.
3. When the APs complete their task or *TimeoutInMicroSecondss* expires, the MP Service signals *WaitEvent* by calling the EFI **SignalEvent()** function. If *FailedCpuList* is not **NULL**, its content is available when *WaitEvent* is signaled. If all APs returned from *Procedure* prior to the timeout, then *FailedCpuList* is set to **NULL**. If not all APs return from *Procedure* before the timeout, then *FailedCpuList* is filled in with the list of the failed APs. The buffer is allocated by MP Service Protocol using **AllocatePool()**. It is the caller's responsibility to free the buffer with **FreePool()** service.
4. This invocation of **SignalEvent()** function informs the caller that invoked **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()** that either all the APs completed

the specified task or a timeout occurred. The contents of *FailedCpuList* can be examined to determine which APs did not complete the specified task prior to the timeout.

Status Codes Returned

EFI_SUCCESS	In blocking mode, all APs have finished before the timeout expired.
EFI_SUCCESS	In non-blocking mode, function has been dispatched to all enabled APs.
EFI_UNSUPPORTED	A non-blocking mode request was made after the UEFI event **EFI_EVENT_GROUP_READY_TO_BOOT** was signaled.
EFI_DEVICE_ERROR	Caller processor is AP.
EFI_NOT_STARTED	No enabled APs exist in the system.
EFI_NOT_READY	All enabled APs are busy.
EFI_TIMEOUT	In blocking mode, the timeout expired before all enabled APs have finished.
EFI_INVALID_PARAMETER	*Procedure* is **NULL**.

EFI_MP_SERVICES_PROTOCOL.StartupThisAP()

Summary

This service lets the caller get one enabled AP to execute a caller-provided function. The caller can request the BSP to either wait for the completion of the AP or just proceed with the next task by using the EFI event mechanism. See the "Non-blocking Execution Support" section in **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()** for more details. This service may only be called from the BSP.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_STARTUP_THIS_AP) (
  IN  EFI_MP_SERVICES_PROTOCOL *This,
  IN  EFI_AP_PROCEDURE         *Procedure,
  IN  UINTN                    ProcessorNumber,
  IN  EFI_EVENT                WaitEvent              OPTIONAL,
  IN  UINTN                    TimeoutInMicroseconds,
  IN  VOID                     *ProcedureArgument     OPTIONAL,
  OUT BOOLEAN                  *Finished              OPTIONAL
);
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

Procedure

A pointer to the function to be run on the designated AP. Type **EFI_AP_PROCEDURE** is defined in **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()**.

ProcessorNumber

The handle number of the AP. The range is from 0 to the total number of logical processors minus 1. The total number of logical processors can be retrieved by **EFI_MP_SERVICES_PROTOCOL.GetNumberOfProcessors()**.

WaitEvent

The event created by the caller with **CreateEvent()** service.

If it is **NULL**, then execute in blocking mode. BSP waits until this AP finishes or *TimeoutInMicroSeconds* expires.

If it's not **NULL**, then execute in non-blocking mode. BSP requests the function specified by *Procedure* to be started on the AP, and go on executing immediately. If this AP finishes or *TimeoutInMicroSeconds* expires, this event is signaled. BSP can use the **CheckEvent()** and **WaitForEvent()** services to check the state of event.

Type **EFI_EVENT** is defined in **CreateEvent()** in the *Unified Extensible Firmware Interface Specification (Version 2.0)*

TimeoutInMicrosecsond

Indicates the time limit in microseconds for this AP to finish the function, either for blocking or non-blocking mode. Zero means infinity.

If the timeout expires before this AP returns from Procedure, then Procedure on the AP is terminated. The AP is available for subsequent calls to **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()** and **EFI_MP_SERVICES_PROTOCOL.StartupThisAP()**.

If the timeout expires in blocking mode, BSP returns **EFI_TIMEOUT**.

If the timeout expires in non-blocking mode, *WaitEvent* is signaled with **SignalEvent()**.

ProcedureArgument

The parameter passed into *Procedure* on the specified AP.

Finished

If **NULL**, this parameter is ignored.

In blocking mode, this parameter is ignored.

In non-blocking mode, if AP returns from *Procedure* before the timeout expires, its content is set to **TRUE.** Otherwise, the value is set to **FALSE**. The caller can determine if the AP returned from *Procedure* by evaluating this value.

Description

This function is used to dispatch one enabled AP to the function specified by *Procedure* passing in the argument specified by *ProcedureArgument*.

If *WaitEvent* is **NULL**, execution is in blocking mode. The BSP waits until the AP finishes or *TimeoutInMicroSecondss* expires. Otherwise, execution is in non-blocking mode. BSP proceeds to the next task without waiting for the AP. If a non-blocking mode is requested after the UEFI Event **EFI_EVENT_GROUP_READY_TO_BOOT** is signaled, then **EFI_UNSUPPORTED** must be returned.

If the timeout specified by *TimeoutInMicroseconds* expires before the AP returns from *Procedure*, then execution of *Procedure* by the AP is terminated. The AP is available for subsequent calls to **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()** and **EFI_MP_SERVICES_PROTOCOL.StartupThisAP()**.

This service may only be called from the BSP.

Status Codes Returned

EFI_SUCCESS	In blocking mode, specified AP finished before the timeout expires.
EFI_SUCCESS	In non-blocking mode, the function has been dispatched to specified AP.

EFI_UNSUPPORTED	A non-blocking mode request was made after the UEFI event **EFI_EVENT_GROUP_READY_TO_BOOT** was signaled.
EFI_DEVICE_ERROR	The calling processor is an AP.
EFI_TIMEOUT	In blocking mode, the timeout expired before the specified AP has finished.
EFI_NOT_READY	The specified AP is busy.
EFI_NOT_FOUND	The processor with the handle specified by *ProcessorNumber* does not exist.
EFI_INVALID_PARAMETER	*ProcessorNumber* specifies the BSP or disabled AP.
EFI_INVALID_PARAMETER	*Procedure* is **NULL**.

EFI_MP_SERVICES_PROTOCOL.SwitchBSP()

Summary

This service switches the requested AP to be the BSP from that point onward. This service changes the BSP for all purposes. This service may only be called from the current BSP.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_SWITCH_BSP) (
  IN EFI_MP_SERVICES_PROTOCOL   *This,
  IN UINTN                      ProcessorNumber,
  IN BOOLEAN                    EnableOldBSP
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

ProcessorNumber

The handle number of AP that is to become the new BSP. The range is from 0 to the total number of logical processors minus 1. The total number of logical processors can be retrieved by **EFI_MP_SERVICES_PROTOCOL.GetNumberOfProcessors()**.

EnableOldBSP

If **TRUE**, then the old BSP will be listed as an enabled AP. Otherwise, it will be disabled.

Description

This service switches the requested AP to be the BSP from that point onward. This service changes the BSP for all purposes. The new BSP can take over the execution of the old BSP and continue seamlessly from where the old one left off. This service may not be supported after the UEFI Event **EFI_EVENT_GROUP_READY_TO_BOOT** is signaled.

If the BSP cannot be switched prior to the return from this service, then **EFI_UNSUPPORTED** must be returned.

This call can only be performed by the current BSP.

Status Codes Returned

EFI_SUCCESS	BSP successfully switched.
EFI_UNSUPPORTED	Switching the BSP cannot be completed prior to this service returning.
EFI_UNSUPPORTED	Switching the BSP is not supported.
EFI_SUCCESS	The calling processor is an AP.

EFI_NOT_FOUND	The processor with the handle specified by *ProcessorNumber* does not exist.
EFI_INVALID_PARAMETER	*ProcessorNumber* specifies the current BSP or a disabled AP.
EFI_NOT_READY	The specified AP is busy.

EFI_MP_SERVICES_PROTOCOL.EnableDisableAP()

Summary

This service lets the caller enable or disable an AP from this point onward. This service may only be called from the BSP.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_ENABLEDISABLEAP) (
  IN EFI_MP_SERVICES_PROTOCOL  *This,
  IN UINTN                     ProcessorNumber,
  IN BOOLEAN                   EnableAP,
  IN UINT32                    *HealthFlag  OPTIONAL
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

ProcessorNumber

The handle number of AP. The range is from 0 to the total number of logical processors minus 1. The total number of logical processors can be retrieved by **EFI_MP_SERVICES_PROTOCOL.GetNumberOfProcessors()**.

EnableAP

Specifies the new state for the processor specified by *ProcessorNumber*. **TRUE** for enabled, **FALSE** for disabled.

HealthFlag

If not **NULL**, a pointer to a value that specifies the new health status of the AP. This flag corresponds to **StatusFlag** defined in **EFI_MP_SERVICES_PROTOCOL.GetProcessorInfo()**. Only the **PROCESSOR_HEALTH_STATUS_BIT** is used. All other bits are ignored.

If it is **NULL**, this parameter is ignored.

Description

This service allows the caller enable or disable an AP from this point onward. The caller can optionally specify the health status of the AP by *Health*. If an AP is being disabled, then the state of the disabled AP is implementation dependent. If an AP is enabled, then the implementation must guarantee that a complete initialization sequence is performed on the AP, so the AP is in a state that is compatible with an MP operating system. This service may not be supported after the UEFI Event **EFI_EVENT_GROUP_READY_TO_BOOT** is signaled.

If the enable or disable AP operation cannot be completed prior to the return from this service, then **EFI_UNSUPPORTED** must be returned.

This service may only be called from the BSP.

Status Codes Returned

EFI_SUCCESS	The specified AP successfully enabled or disabled.
EFI_UNSUPPORTED	Enabling or disabling an AP cannot be completed prior to this service returning.
EFI_UNSUPPORTED	Enabling or disabling an AP is not supported.
EFI_DEVICE_ERROR	The calling processor is an AP.
EFI_NOT_FOUND	Processor with the handle specified by *ProcessorNumber* does not exist.
EFI_INVALID_PARAMETER	*ProcessorNumber* specifies the BSP.

EFI_MP_SERVICES_PROTOCOL.WhoAmI()

Summary

This return the handle number for the calling processor. This service may be called from the BSP and APs.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_WHOAMI) (
  IN  EFI_MP_SERVICES_PROTOCOL *This,
  OUT UINTN                    *ProcessorNumber
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

ProcessorNumber

Pointer to the handle number of AP. The range is from 0 to the total number of logical processors minus 1. The total number of logical processors can be retrieved by **EFI_MP_SERVICES_PROTOCOL.GetNumberOfProcessors()**.

Description

This service returns the processor handle number for the calling processor. The returned value is in the range from 0 to the total number of logical processors minus 1. The total number of logical processors can be retrieved with **EFI_MP_SERVICES_PROTOCOL.GetNumberOfProcessors()**. This service may be called from the BSP and APs. If *ProcessorNumber* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. Otherwise, the current processors handle number is returned in *ProcessorNumber*, and **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The current processor handle number was returned in *ProcessorNumber*.
EFI_INVALID_PARAMETER	*ProcessorNumber* is **NULL**.

14
DXE Runtime Protocols

14.1 Introduction

In addition to the architectural protocols listed earlier, there is also a runtime protocol. Specifically, the ability to report status codes is runtime-callable service that allows for emitting status and progress information. It was formerly part of the 0.9 DXE-CIS runtime table, but in consideration of UEFI 2.0 compatibility, this capability has become a separate runtime protocol.

14.2 Status Code Runtime Protocol

EFI_STATUS_CODE_ PROTOCOL

Summary

Provides the service required to report a status code to the platform firmware. This protocol must be produced by a runtime DXE driver and may be consumed only by the DXE Foundation.

GUID

```
#define EFI_STATUS_CODE_RUNTIME_PROTOCOL_GUID  \

{ 0xd2b2b828, 0x826, 0x48a7, 0xb3, 0xdf, 0x98, 0x3c, 0x0, 0x60,
0x24, 0xf0}
```

Protocol Interface Structure

```
typedef struct _EFI_STATUS_CODE_PROTOCOL {
  EFI_REPORT_STATUS_CODE          ReportStatusCode;
} EFI_STATUS_CODE_PROTOCOL;
```

Parameters

ReportStatusCode

Emit a status code.

Description

The DXE driver that produces this protocol must be a runtime driver. This driver is responsible for providing the **ReportStatusCode()** service with the **EFI_STATUS_CODE_PROTOCOL**.

EFI_STATUS_CODE_PROTOCOL.ReportStatusCode()

Summary

Provides an interface that a software module can call to report a status code.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_REPORT_STATUS_CODE) (
  IN EFI_STATUS_CODE_TYPE           Type,
  IN EFI_STATUS_CODE_VALUE          Value,
  IN UINT32                         Instance,
  IN CONST EFI_GUID                 *CallerId  OPTIONAL,
  IN CONST EFI_STATUS_CODE_DATA     *Data      OPTIONAL
  );
```

Parameters

Type

Indicates the type of status code being reported. Type **EFI_STATUS_CODE_TYPE** is defined in "Related Definitions" below.

Value

Describes the current status of a hardware or software entity. This included information about the class and subclass that is used to classify the entity as well as an operation. For progress codes, the operation is the current activity. For error codes, it is the exception. For debug codes, it is not defined at this time. Type **EFI_STATUS_CODE_VALUE** is defined in "Related Definitions" below.

Instance

The enumeration of a hardware or software entity within the system. A system may contain multiple entities that match a class/subclass pairing. The instance differentiates between them. An instance of 0 indicates that instance information is unavailable, not meaningful, or not relevant. Valid instance numbers start with 1.

CallerId

This optional parameter may be used to identify the caller. This parameter allows the status code driver to apply different rules to different callers. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Data

This optional parameter may be used to pass additional data. Type **EFI_STATUS_CODE_DATA** is defined in "Related Definitions" below. The contents of this data type may have additional GUID-specific data.

Description

Various software modules including drivers can call this function to report a status code. No disposition of the status code is guaranteed. The **ReportStatusCode()** function may choose to log the status code, but this action is not required.

It is possible that this function may get called at **EFI_TPL_LEVEL_HIGH**. Therefore, this function cannot call any protocol interface functions or services (including memory allocation) that are not guaranteed to work at **EFI_TPL_LEVEL_HIGH**. It should be noted that **SignalEvent()** could be called by this function because it works at any TPL including **EFI_TPL_L**

EVEL_HIGH. It is possible for an implementation to use events to log the status codes when the TPL level is reduced.

ReportStatusCode() function can perform other implementation specific work, but that is not specified in the architecture document.

In case of an error, the caller can specify the severity. In most cases, the entity that reports the error may not have a platform wide view and may not be able to accurately assess the impact of the error condition. The DXE driver that produces the Status Code Architectural Protocol, **EFI_STATUS_CODE_ARCH_PROTOCOL**, is responsible for assessing the true severity level based on the reported severity and other information. This DXE driver may perform platform specific actions based on the type and severity of the status code being reported.

If *Data* is present, the Status Code Architectural Protocol driver treats it as read only data. The Status Code Architectural Protocol driver must copy *Data* to a local buffer in an atomic operation before performing any other actions. This is necessary to make this function re-entrant. The size of the local buffer may be limited. As a result, some of the *Data* can be lost. The size of the local buffer should at least be 256 bytes in size. Larger buffers will reduce the probability of losing part of the *Data*. Note than multiple status codes may be reported at elevated TPL levels before the TPL level is reduced. Allocating multiple local buffers may reduce the probability losing status codes at elevated TPL levels. If all of the local buffers are consumed, then this service may not be able to perform the platform specific action required by the status code being reported. As a result, if all the local buffers are consumed, the behavior of this service is undefined.

If the *CallerId* parameter is not **NULL**, then it is required to point to a constant GUID. In other words, the caller may not reuse or release the buffer pointed to by *CallerId*.

Related Definitions

```
//
// Status Code Type Definition
//
typedef UINT32 EFI_STATUS_CODE_TYPE;

//
// A Status Code Type is made up of the code type and severity
// All values masked by EFI_STATUS_CODE_RESERVED_MASK are
// reserved for use by this specification.
//
#define EFI_STATUS_CODE_TYPE_MASK       0x000000FF
#define EFI_STATUS_CODE_SEVERITY_MASK   0xFF000000
#define EFI_STATUS_CODE_RESERVED_MASK   0x00FFFF00

//
// Definition of code types, all other values masked by
```

```
// EFI_STATUS_CODE_TYPE_MASK are reserved for use by
// this specification.
//
#define EFI_PROGRESS_CODE               0x00000001
#define EFI_ERROR_CODE                  0x00000002
#define EFI_DEBUG_CODE                  0x00000003

//
// Definitions of severities, all other values masked by
// EFI_STATUS_CODE_SEVERITY_MASK are reserved for use by
// this specification.
// Uncontained errors are major errors that could not contained
// to the specific component that is reporting the error
// For example, if a memory error was not detected early enough,
// the bad data could be consumed by other drivers.
//
#define EFI_ERROR_MINOR                 0x40000000
#define EFI_ERROR_MAJOR                 0x80000000
#define EFI_ERROR_UNRECOVERED           0x90000000
#define EFI_ERROR_UNCONTAINED           0xa0000000

//
// Status Code Value Definition
//
typedef UINT32 EFI_STATUS_CODE_VALUE;

//
// A Status Code Value is made up of the class, subclass, and
// an operation.
//
#define EFI_STATUS_CODE_CLASS_MASK      0xFF000000
#define EFI_STATUS_CODE_SUBCLASS_MASK   0x00FF0000
#define EFI_STATUS_CODE_OPERATION_MASK  0x0000FFFF

//
// Definition of Status Code extended data header.
// The data will follow HeaderSize bytes from the beginning of
// the structure and is Size bytes long.
//
typedef struct {
  UINT16    HeaderSize;


  UINT16    Size;
  EFI_GUID  Type;
} EFI_STATUS_CODE_DATA;
```

Parameters

HeaderSize

The size of the structure. This is specified to enable future expansion.

Size

The size of the data in bytes. This does not include the size of the header structure.

Type

The GUID defining the type of the data.

Status Codes Returned

EFI_SUCCESS	The function completed successfully
EFI_DEVICE_ERROR	The function should not be completed due to a device error.

15
Dependency Expression Grammar

15.1 Dependency Expression Grammar

This topic contains an example BNF grammar for a DXE driver dependency expression compiler that converts a dependency expression source file into a dependency section of a DXE driver stored in a firmware volume.

15.2 Example Dependency Expression BNF Grammar

```
<depex>     ::= BEFORE <guid>
              | AFTER <guid>
              | SOR <bool>
              | <bool>
<bool>      ::= <bool> AND <term>
              | <bool> OR <term>
              | <term>
<term>      ::= NOT <factor>
              | <factor>
<factor>    ::= <bool>
              | TRUE
              | FALSE
              | GUID
              | END
<guid>      ::= '{' <hex32> ',' <hex16> ',' <hex16> ','
                <hex8> ',' <hex8> ',' <hex8> ',' <hex8> ','
                <hex8> ',' <hex8> ',' <hex8> ',' <hex8> '}'
<hex32>     ::= <hexprefix> <hexvalue>
<hex16>     ::= <hexprefix> <hexvalue>
<hex8>      ::= <hexprefix> <hexvalue>
<hexprefix>::= '0' 'x'
              | '0' 'X'
<hexvalue> ::= <hexdigit> <hexvalue>
              | <hexdigit>
<hexdigit> ::= [0-9]
              | [a-f]
              | [A-F]
```

15.3 Sample Dependency Expressions

The following contains three examples of source statements using the BNF grammar from above along with the opcodes, operands, and binary encoding that a dependency expression compiler would generate from these source statements.

```
//
// Source
//
EFI_CPU_IO_PROTOCOL_GUID AND EFI_CPU_ARCH_PROTOCOL_GUID END

//
// Opcodes, Operands, and Binary Encoding
//
ADDR   BINARY                      MNEMONIC
====   ======================
================================================
0x00 : 02                          PUSH
0x01 : 26 25 73 b0 c8 38 40 4b     EFI_CPU_IO_PROTOCOL_GUID
       88 77 61 c7 b0 6a ac 45
0x11 : 02                          PUSH
0x12 : b1 cc ba 26 42 6f d4 11     EFI_CPU_ARCH_PROTOCOL_GUID
       bc e7 00 80 c7 3c 88 81
0x22 : 03                          AND
0x23 : 08                          END

//
// Source
//
AFTER (EFI_CPU_DRIVER_FILE_NAME_GUID) END

//
// Opcodes, Operands, and Binary Encoding
//
ADDR   BINARY                      MNEMONIC
====   ======================
================================================
0x00 : 01                          AFTER
0x01 : 93 e5 7b 98 43 16 0b 45     EFI_CPU_DRIVER_FILE_NAME_GUID
       be 4f 8f 07 66 6e 36 56
0x11 : 08                          END

//
// Source
//
SOR EFI_CPU_IO_PROTOCOL_GUID END

//
// Opcodes, Operands and Binary Encoding
//
ADDR   BINARY                      MNEMONIC
====   ======================
```

```
=============================================
0x00 : 09                          SOR
0x01 : 02                          PUSH
0x02 : b1 cc ba 26 42 6f d4 11     EFI_CPU_IO_PROTOCOL_GUID
       bc e7 00 80 c7 3c 88 81
0x12 : 03                          END
```

Appendix A
Error Codes

A.1 EFI_REQUEST_UNLOAD_IMAGE

```
#define DXE_ERROR(a)                    (MAX_BIT|MAX_BIT >> 2 | (a))
```

EFI_REQUEST_UNLOAD_IMAGE	DXE_ERROR (1)	If this value is returned by an EFI image, then the image should be unloaded.
EFI_NOT_AVAILABLE_YET	DXE_ERROR (2)	If this value is returned by an API, it means the capability is not yet installed/available/ready to use.

Appendix B
GUID Definitions

B.1 DXE Services Table GUID

```
#define DXE_SERVICES_TABLE_GUID  \
{0x5ad34ba,0x6f02,0x4214,0x95,0x2e,0x4d,0xa0,0x39,0x8e,0x2b,0xb9
}
```

B.2 HOB List GUID

```
#define HOB_LIST_GUID  \
{0x7739f24c,0x93d7,0x11d4,0x9a,0x3a,0x0,0x90,0x27,0x3f,0xc1,0x4d
}
```