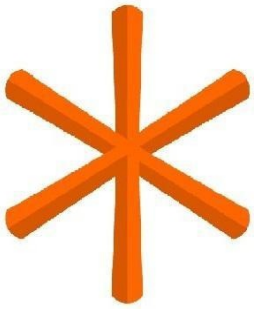

tianocore

EDK II Module Writer's Guide

TABLE OF CONTENTS

EDK II Module Writer's Guide

DRAFT FOR REVIEW

04/30/2025 09:35:19

Acknowledgments

Revision History

Revision Number	Description	Revision Date
0.01	Initial creation.	April 2007.
0.7	Update section 8.10 DXE SMM Driver	March 2010.
	Add note for the usage of	
	PI_SPECIFICATION_VERSION/UEFI_SPECIFICATION_VERSI ON in module's INF	
	Add UEFI_HII_RESOURCE_SECTION usage in INF's [Defines] section	
	Add a limitation of dynamic PCD usage	
0.71	Convert to GitBook	September 2018

Tables

Figures

1 THE BASICS OF EDK II

This document is a guideline for new EDK II module developers, and provides detailed instructions on how to develop and build a new module, and how to release with a package. For information about developing new modules, start with this document.

1.1 Overview

This chapter also clarifies new concepts introduced with EDK II.

Reference the EDK II User Manual, to understand how to obtain EDK II and how to build existing modules.

1.1.1 Module, Package and Platform

1.1.1.1 What is a Module?

A module is the smallest piece of separately compile-able code or pre-built binary. It contains a metadata file (INF) plus source code or binary. The INF file is required by the EDKII build system to describe a module's behavior, such as produced or consumed library classes, ppis, guids, protocols, pcds, and other information.

For example, in $WORKSPACE\MdeModulePkg\Universal\Bus\Pci\UhciDxe, the source files mentioned and the INF file compose a module.

,Refer to the EDK II Extended INF Specification. for the syntax of the INF file.

1.1.1.2 What is a Package?

A package is a group of zero or more modules. A package must contain a package metadata file (DEC), and possibly a platform metadata file (DSC).

Functionally, a package is a logical division of a project. Developers depend on reasonable judgment, such as license or specification compliance, to determine where to place a module. These metadata files and the module's INF files are used by the EDK II build system to automatically generate makefiles and a single module tip or whole flash tip, according to the build options used.

For information regarding the syntax of DSC and DEC files, please refer to EDK II DSC File Specification and EDK II DEC File Specification.

1.1.1.3 What is a Platform?

A platform is a special type of package with additional metadata files. A package must contain one DSC file and zero or one FDF file. The FDF is only required if flash output is required.

Refer to EDK II FDF File Specification for information regarding FDF files..

Refer to the EDK II Platform Port Guide for information about platform porting between EDK and EDK II.

1.1.2 Module Customization

Use the EDK II User Manual to understand the design purpose of the Library class/Library instance and PCD mechanisms. These mechanisms provide ways for developers to customize modules without changing the source code.

1.1.2.1 Library class/Library instance

Developers may choose a proper library instance according to its requirements, such as from performance, image size, or the limitation of module type.

In $WORKSPACE\MdePkg core package, there are many supported library classes and corresponding library instances. Browse the $WORKSPACE\MdePkg \include\library directory for basic information regarding the helper function API provided by these library classes.

1.1.2.2 PCD

Developers may take advantage of the PCD mechanism to extract information from outside a module, and control procedure behavior inside a module. The information may be known at compile time from the platform DSC or the package DEC files, but some files may arrive at flash image generation time, and some may be determined during execution.

Example:, in the following chapter, `PcdDebugPropertyMask` declared in MdePkg.dec file in $WORKSPACE\MdePkg is used to control DebugLib behavior. This PCD is FixedAtBuild type, meaning its value is determined at build time. The EDK II build system converts this pcd to the value configured in DEC or DSC to enable or disable the print ability.

1.1.3 EDK II Development Lifecycle

The lifecycle of EDK II development is divided into the five phases which follow.

1.1.3.1 Phase 1: Create a package

A package is the container of modules. A developer should first consider where the module should be placed. As a general rule, modules newly developed by an IHV/IBV should not be placed into existing EDK II core packages, which include `MdePkg`,

4BThe Basics of EDK II

`MdeModulePkg`, `IntelFrameworkPkg` and `IntelFrameworkModulePkg`. One reason is that these packages are published as a base-supported package to facilitate module/platform development.

Note: These packages are open-source code and compliant with the BSD license. If the developed module is not intended to be open source, it should not be put into those core packages.

To create a new package, developers must create the DEC file to define the package's interfaces, including:

- include directories for modules from other packages
- the value of GUIDs
- the value of Protocol GUIDs
- the value of PPI GUIDs
- the declaration of the PCD entries published by this package.

1.1.3.2 Phase 2: Create module metadata/Implement basic functionality.

After the module to be placed into a package is determined, developers must create an INF file to indicate the module's behavior, including:

- module type
- required library classes
- required ppi/protocol/guid/PCD
- dependency relationship with other modules.

Note: Dependency relationships may exist or not, depending on various module types.

Viewing a module's INF file provides a quick overview to an unfamiliar module.

After finishing the INF file, developers should start writing source code to implement basic functionality.

Note: In $WORKSPACE\MdePkg\Include\Library directory, there are many library classes to provide support functions. There are also entry point libraries for various module types. Developers should browse the header files for details.

1.1.3.3 Phase 3: Create DSC to build

In EDK II, the DSC file describes the build behavior of the package, including:

- modules needed to be built
- chosen library instances for various module type
- the configuration of the PCD entries used by modules.

The single platform DSC and each referenced package's DEC files cooperate to define a package. These files and the module INF files are required to build all modules in the package.

1.1.3.4 Phase 4: Tune modules

To tune modules

- use EDK II libraries for code reuse
- use EDK II PCD mechanism for configuration.

The distinction between an EDK module and an EDK II module is that the EDK II module can be customized either statically or dynamically, as necessary.

- Static customization is preferred to choose the library instance or determine the value of FeatureFlag/Fixed type of PCD at build time.
- Dynamic customization is preferred for using Patchable/Dynamic type PCD to control procedure behavior on the fly.

EDK II module developers should consider what logic in the module could be generalized as early in the development of the module as possible. For example, if some functionality had been implemented in a library class of a core package, the developer should replace it by using the library class.

If a segment of logic can be extracted as common logic and shared by various modules, the developer can create a new library class and instance.

Note: In the EDK II module development, developers are strongly discouraged from using a conditional macro to control procedure behavior. The PCD mechanism provides a unified interface, and developers should use it to configure a module's behavior.

1.1.4 Build Infrastructure

The EDK II build system is based on Python and portable C code to provide crossplatform build-ability. Figure 1, developer illustrates the conceptual workflow of the EDK II build system infrastructure.

4BThe Basics of EDK II

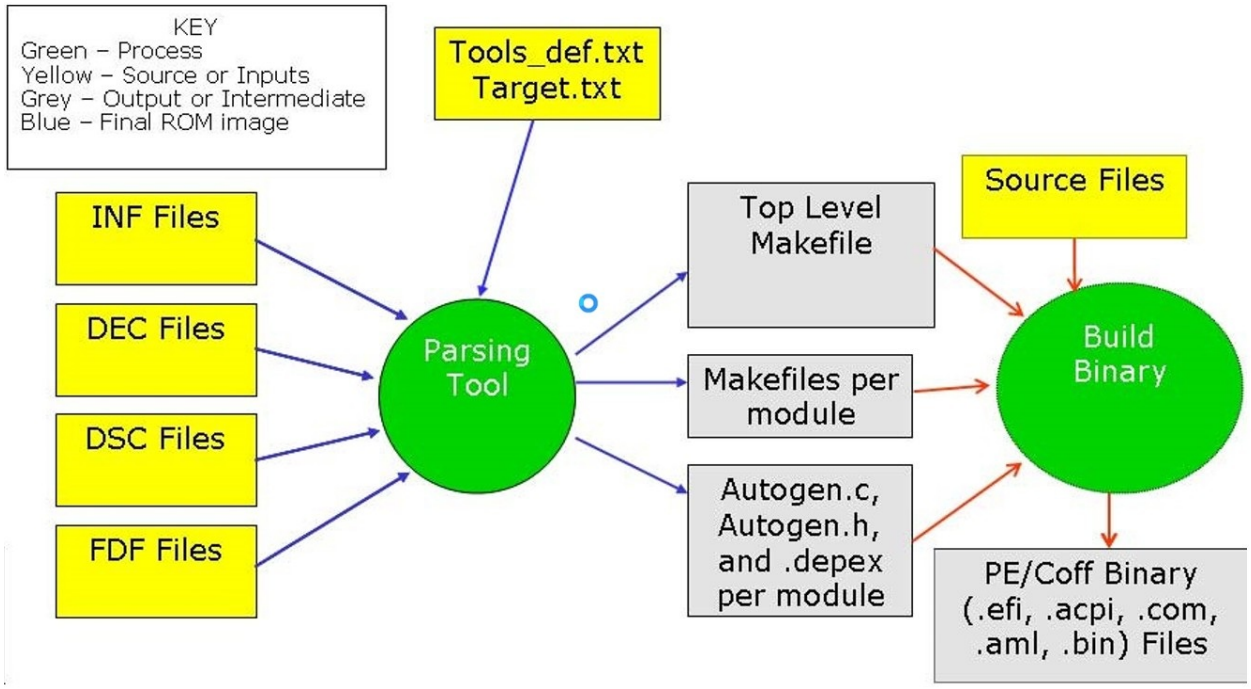

Figure 1 Conceptual workflow

In brief, the EDK II build tool parses the metadata files (DSC, DECs, and INFs) to generate corresponding one top-level makefile and a separate set of makefile and autogen.c/autogen.h files for every module.

In the autogen files, the EDK II build tool generates all definitions of guids, protocols, ppis, and PCDs used by the module, and automatically invokes all of the constructors of used library instances in the module's entry point implementations.

1.2 Related References

The following publications and sources of information may be useful or are referred to by this document:

- Unified Extensible Firmware Interface Specification Version 2.1, Unified EFI, Inc, 2007, http://www.uefi.org.
- Extensible Firmware Interface Specification Version 1.10, Intel, 2001, http://developer.intel.com/technology/efi.
- Intel(R) Platform Innovation Framework for EFI Specifications, Intel, 2006, http://www.intel.com/technology/framework/.

The following publications are available at edk2.tianocore.org:

- EDK II MDE (Module Development Environment) Package Document, Version

1.00, Intel, 2006.

- EDK II DSC File Specification, Version 0.50, Intel, 2007.
- EDK II DEC File Specification, Version 0.50, Intel, 2007.
- EDK II Extended INF Specification, Version 0.50, Intel, 2007.
- EDK II FDF (Flash Description File) File Specification, Version 0.50, Intel, 2007.
- EDK II Build Specification, Version 1.00, Intel, 2008.

1.3 Terms

The following terms are used throughout this document to describe varying aspects of input localization:

EDK

EFI Developer's kit; the open source project of the Intel Platform Innovation Framework for EFI that can be found at http://edk.tianocore.org.

EDK II

A generic term to describe the open source project found at

http://edk2.tianocore.org. In this document, it refers to the new release of EDK II that supports a build infrastructure that makes use of the Extended INF, DEC and Extended DSC.

EDK II Module

A generic term to describe a module that is developed using the new release of EDK II project that supports the library class, library instances, packaging concept and Extended INF, DEC and Extended DSC files.

1.4 Target Audience

This document is intended for the following readers:

- Developers from IBVs and OEMs who will be implementing UEFI/PI drivers or other firmware products based on EDK II.
- Developers from IHVs who will be creating UEFI Driver Model drivers for hardware devices.
- Platform integrators using EDK II components and modules to build platform firmware.

2 AN EDK II PACKAGE

2.1 Introduction

Each EDK II Package is a container that includes a set of modules and their related definitions. Each Package is an EDK II distribution unit. It can be used to manage and release the big project to facilitate a user's distribution and reuse. Entire project sources can be split into different packages to reduce the release granularity. The new project can also be made from released packages from different sources.

2.1.1 EDK II Packages

A Package is a directory that organizes a group of modules with a single package declaration file (DEC).

EDK II provides UEFI and PI compliant packages: MdePkg, MdeModulePkg, etc.

The MdePkg contains the complete definitions in EFI1.1, UEFI2.0, UEFI2.1, PI1.0

Specifications and all library classes and instances defined in EDK II MDE (Module Development Environment) Library Specification. UEFI and PI drivers can be developed based solely on this package.

The MdeModulePkg contains a group of cross-platform drivers that conform to UEFI and PI specifications. They can be referred to when developing new UEFI and PI drivers.

Detailed information of EDK II packages can be found in EDK II User Guide, section 2.2 and in the package specification for each package.

2.1.2 The Package Directory

Each package has a unified directory structure that separate the different source files. The root directories in each package are: Include, Library, Application and Drivers.

- The include directory contains all public header files that are exposed by this package and are used by this package and other packages. Below the Include directory, subdirectories may be created to include Ppi, Protocol, Guid, Industry Standard and library class header files (when these header files become public).
- The library directory contains directories for each library instance module included.
- The application directory contains directories for each UEFI applications module included.
- The driver directory contains directories for each driver group and for each driver.

Each module (library instance, application and driver) has its own directory in which to group its source files. A module may only depend on files under its directory or on public header files. A module is not permitted to depend on source files from another module directory.

2.1.2.1 Sample directories and sub-directories in a package

- Package.dec Package declaration file
- Package.dsc Platform Package build description file
- Include Public header files
    - Protocol\ Public Protocol header files o Ppi\ Public PPI header files o Guid\ Public GUID header files
    - IndustryStandard\ Public Industry Standard header files o Library\ Public Library class header files

- Library\ Libraries instances
  - NameOneLib\ Library instance NameOne source files and INF o NameTwoLib\ Library instance NameTwo source files and INF
- Application\ Uefi Applications o NameOneApp\ Application NameOne source files and INF o NameTwoApp\ Application NameTwo source files and INF NameOneDxe\ Dxe Driver NameOne source files and INF.
- NameTwoPei\ Pei Driver NameTwo source files and INF.

If no related source files exist in a package, the corresponding directory may not be created. For example, if no application is provided in a package, a blank Application directory is not required.

2.1.3 Package Declaration File

Each package has a single package declaration file (DEC) to define the package's public interfaces. The public interfaces are the package's public header files, GUIDs, and PCDs.

The DEC has Defines, Includes, LibraryClasses, Guids, Ppis, Protocols and Pcds sections.

The [Defines] section defines the package name and package GUID.

The [Includes] section must list the root directory of public header file directory.

The [LibraryClasses] section contains every library class header file in the Package\Include\Library directory.

The [Guids] section specifies the Guid value for each Guid in the Package\Include\Guid directory.

The [Ppis] section specifies the Guid value for each PPI in the Package\Include\Ppi directory.

The [Protocols] section specifies the Guid for each Protocol in the Package\Include\Protocol directory.

The PCDs are declared in different PCD sections according to their type (FeatureFlag, FixedAtBuild, PatchableInModule, Dynamic, and DynamicEx). If a PCD supports multiple PCD types, it must be declared in all supported type sections. When a PCD is declared, its data type and default value must also be specified.

The following is a sample DEC file, additional package public information may be added.

2.1.3.1 Example: Package.dec

```
[Defines]
  DEC_SPECIFICATION = 0x00010005
  PACKAGE_NAME      = PackageName
  PACKAGE_GUID      = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
  PACKAGE_VERSION   = 0.1

[Includes]
  Include #Package Include directory

[LibraryClasses]
  ## Library class name is same to library header file name
  OneClassLib|Include/Library/OneClassLib.h

[Guids]

#GuidCName = {xxxxxxxx,xxxx,xxxx,{xx,xx,xx,xx,xx,xx,xx,xx}},
[Ppis]

#PpiGuidCName = {xxxxxxxx,xxxx,xxxx,{xx,xx,xx,xx,xx,xx,xx,xx}},
[Protocols]

#ProtocolGuidCName = {xxxxxxxx,xxxx,xxxx,{xx,xx,xx,xx,xx,xx,xx,xx}},
[PcdsFeatureFlag] #FeatureFlag PCD is BOOLEAN type, the value is TRUR or FALSE.
```

```
#PcdTokenSpaceCGuidName.PcdName|TRUE|BOOLEAN|TokenNumber
#PcdTokenSpaceCGuidName.PcdName|FALSE|BOOLEAN|TokenNumber
[PcdsFixedAtBuild]

#PcdTokenSpaceCGuidName.PcdName|DefaultValue|DataType|TokenNumber
[PcdsPatchableInModule]

#PcdTokenSpaceCGuidName.PcdName|DefaultValue|DataType|TokenNumber
[PcdsDynamic]

#PcdTokenSpaceCGuidName.PcdName|DefaultValue|DataType|TokenNumber
[PcdsDynamicEx]
  #PcdTokenSpaceCGuidName.PcdName|DefaultValue|DataType|TokenNumber
```

Refer to the EDK II DEC File Specification for a detailed description of the DEC file format.

2.1.4 Package DSC File

Each package usually creates another build description file (DSC). All modules can be added into DSC to be compiled and verified. DSC has the following sections:

- Defines
- LibraryClass
- PCD
- Components.

The [Defines] section sets build related information, such as the build output directory, build target, Guid, and build ARCHs.

The [Components] section lists all modules (Drivers, Application, and Library Instances) in the platform.

The [LibraryClasses] section specifies the chosen library instance for every library class, which is consumed by the drivers and applications in the [Components] section.

The [PCDs] section configures PCD type and value for those PCDs used by the modules in the [Components] section. If the PCD value is same as the default value in DEC, and the PCD type has no specific requirement, the PCD may not be configured in the DSC. Its value and type will be the default setting in DEC. If all PCDs are not required in the DSC file, the [PCDs] section may be not created.

Note: Only the DSC file for the active platform is used in a build.

The following is a sample DSC file. More modules may be added.

2.1.4.1 Example: Package.dsc

```
[Defines]
  PLATFORM_NAME           = PacakgeName
  PLATFORM_GUID           = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
  PLATFORM_VERSION        = 0.1
  DSC_SPECIFICATION       = 0x00010005
  OUTPUT_DIRECTORY        = Build/PackageName
  SUPPORTED_ARCHITECTURES = IA32|IPF|X64|EBC
  BUILD_TARGETS           = DEBUG|RELEASE
  SKUID_IDENTIFIER        = DEFAULT

[SkuIds] 0|DEFAULT #The entry: 0|DEFAULT is reserved and required.
```

```
[LibraryClasses]
  ## More library instances need to be added if more library classes are used
  ## by the components in the following [Components] section.
  ## library class name | library instance INF file path from package
  DebugLib | MdePkg/Library/UefiDebugLibStdErr/UefiDebugLibStdErr.inf
  BaseLib | MdePkg/Library/BaseLib/BaseLib.inf
  BaseMemoryLib | MdePkg/Library/BaseMemoryLib/BaseMemoryLib.inf
  ......

##PCDs sections are not specified.
##All PCDs value are from their Default value in DEC.
##[PcdsFeatureFlag]
##[PcdsFixedAtBuild]
[Components]
  # All libraries, drivers and applications are added here to be compiled
  #
  # Module INF file path are specified from package directory.
  PackageNamePkg/Library/NameOneLib/NameOneLib.inf
  PackageNamePkg/NameOneDxe/NameOneDxe.inf
  PackageNamePkg/NameTwoPei/NameTwoPei.inf
```

A detailed description of the DSC file form is given in the DSC Specification.

2.2 Manage Package

2.2.1 Create Package

When current packages do not satisfy a requirement, or the original code base is split into EDK II packages, new packages need to be created. Following are the recommended rules for defining the new package:

- All modules related to the same functionality should go in the same package. For example, different packages should be created for different chipsets.
- Generic modules shared between different platforms should be in another package. For example, the MdePkg and MdeModulePkg are shared.
- Modules should go in packages according to their release requirements. If modules are released only to specific customers, they should go in specific packages.

Note: There is no limitation for source files in a new package. Even if only one file is in a package, this package will be valid.

According to the rules given immediately above, the EDK II project provides several packages for user reference.

When a new package is added, the following steps are used to create it.

- Give the meaningful package name as the package directory name and create a package directory, such PackageNamePkg.
- Create package DEC and DSC files in the package root directory to describe this package.
- Create package sub-directories to contain the different source files.

2.2.2 Using a Package

A package provides public header files, library classes, PCDs and modules, which are required to develop other modules and platforms.

A module is dependent on the package that it resides in and may be dependent on other packages.

A platform is made from the modules contained in its own package, and from other packages.

The EDK II package is the basic development unit. It can be used to configure the development workspace. According to development requirements, the workspace can integrate different packages from the EDK II project and other sources. To develop a module or a platform, their dependent packages need to be integrated into the workspace. For example, to develop UEFI and PI driver, the MdePkg, which contains all UEFI/PI definitions, is required in the workspace.

The following show how to develop modules and platforms based on packages.

- Each package DEC file and Include directory lists package public header files, library classes and PCDs. When a new module is developed, it can include information from the DECs of all packages in current workspaces If it needs information from a package that is not in the current workspace, this package needs to be added.

- Each package DSC file lists all modules provided by this package. The developer can search the DSCs of all packages in the current workspace to obtain the required modules (and move their information into the platform DSC file). Then it specifies those modules in the platform DSC and FDF file. If a new platform still requires the modules from a packages that are not in the current workspace, this package needs to be added.

2.2.3 Updating a Package

The package DEC and DSC files describe the package capability, which should be created according to source files of this package. If source files are changed, removed, or added, the package DEC and DSC file must be updated to match their change.

All changes to source code that affect the DEC and DSC file are introduced one by one.

2.2.3.1 Updating Package Include Directories

When a package Include directory is changed, added, or removed, the [Includes] a section of the DEC must be updated.

2.2.3.1.1 Example: Include section of Package.dec

```
[Includes]
  Include # Package Include path
  LocalInclude # Add new include path
```

2.2.3.2 Updating Guids/Ppis/Protocols

When a Guid value or Guid global CName defined in the package public Guid header file changes, the [Guids] section of the DEC must be updated to the new Guid value or Guid CName. If a public Guid header file is removed, the Guid defined in this file must be removed from the [Guids] section of the DEC. If a new guid header file is added in the package public include directory, the new declared Guid and its value must be added to the [Guids] section of the DEC. Like the Guid header file, any change to Guid values defined in the Ppi and Protocol header files also requires the [Ppis] or the [Protocols] section to be updated.

2.2.3.2.1 Example: Guid section of Package.dec

```
[Guids]
  #gGuidCName = {00000000,0000,0000,{00,00,00,00,00,00,00,00}}
  #updated to
  gNewGuidCName = {00000000,0000,0000,{00,00,00,00,00,00,00,01}}
```

2.2.3.3 Updating Library Classes

When the library class name is changed, the library class header file name needs to update the [LibraryClasses] section of the DEC to map the new library class name to the (new?) header file. The change to the library class name will also require the [LibraryClasses] section (of the DSC to be updated) to map the new library class name and the library instance. When a new library class is introduced, its name and its header file will be specified in the DEC [LibraryClasses] section.

2.2.3.3.1 Example: LibraryClasses section of Package.dsc

```
[LibraryClasses]
  #OneClassLib|Include/Library/OneClassLib.inf updated to
  BaseMemoryLib|MdePkg/Library/BaseMemoryLib/BaseMemoryLib.inf
```

2.2.3.4 Updating PCDs

PCDs are declared in the package DEC and are not related to any header file. However, module source files use them. If a PCD does not exist in any module, its declaration should be removed from the DEC file. The setting for this PCD in DSC file should also be removed.

When a module requires a new PCD, it needs to define this PCD in the DEC file for the package where the module is located. Then the DEC file will specify the PCD type and default value.

2.2.3.4.1 Example: Package.dec

```
[PcdsFixedAtBuild]
  #Add new FixedAtBuild PCD
  #PcdTokenSpaceCGuidName.PcdName|DefaultValue|DataType|TokenNumber
  gEfiMdeModulePkgTokenSpaceGuid.PcdHelloWorldTimes|1|UINT32|0x40000005
```

2.2.3.5 Updating Modules

Changes to modules (Library instance, drivers and applications) cause the dependent header file, library class and PCDs be modified, which requires the DSC file to be updated.

If a module INF file name is changed, the DSC files that refer to this module are updated to new file name. If a module is completely removed, it will not be compiled any more, and is removed from the package DSC. When a new module is added to a package, it should be added to the package DSC to be compiled and verified.

2.2.3.5.1 Example: Package.dsc

```
[Components] #Module INF file path are specified from package directory.
  # PackageNamePkg/NameTwoPei/NameTwoDxe.inf
  # updated to
  MdeModulePkg/Application/HelloWorld/HelloWorld.inf
```

3 MODULE DEVELOPMENT

3.1 What is an EDK II module?

An EDK II module consists of source files or binary files and a module definition file (INF file). An INF file describes a module's basic information and interfaces such as consumed/produced library class/PCD/Protocol/Ppi/Guid. (Please refer to the EDK II Extended INF Specification

A typical EDK II module is a firmware component that is built, put in an FFS file and then put into a FV image. The component may be:

A driver or application which is built to *.efi binary file and put into FFS file as EFI_PE_SECTION:

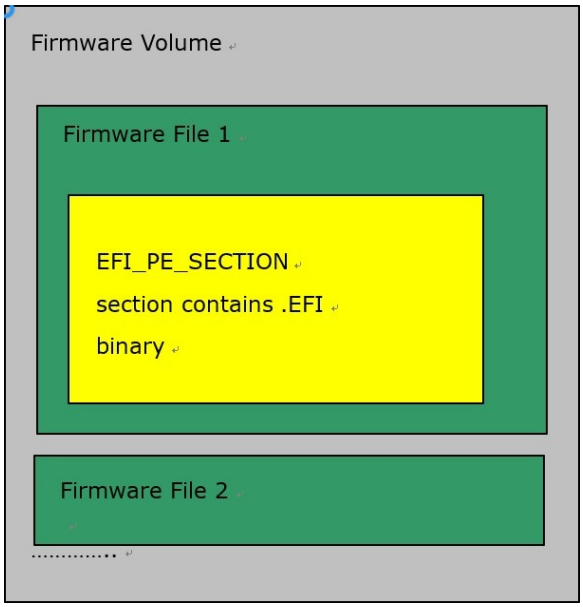

Figure 2 Firmware Volume

- Raw data binary. For example, $(WORKSPACE)\MdeModulePkg\Logo\Logo.inf is a raw binary module which contains logo bitmap image.
- An option ROM driver that is put into a device's option ROM.
- A standalone UEFI driver
- A standalone UEFI application.
- A library instance that is built to a library object file (.lib) and statically linked to another module.

Note: A module can be released in source code or in EFI binary format.

3.1.1 Module Type

EDK II defines many module types. The module type is used to:

- Indicate the lifecycle for different types of modules. For example, PEIM is dispatched in PEI phase and DXE_DRIVER orUEFI_DRIVER is dispatched at DXE phase;
- Indicate the binary image generation for different types of modules. For example, a PEIM/DXE_DRIVER type module can have "depex" section in .efi binary image; a UEFI_DRIVER can have .ui or .ver section in .efi binary image;

- Indicate EntryPoint() or Constructor() API for different types of modules.
- Indicate the suitable library instance for different types of modules. A library instance will point out what module types are supported in INF file.

Table 1 EDK II Module Types

MODULE_TYPE	Description
SEC	Modules of this type are designed to start execution at the reset vector of a CPU. They are responsible for preparing the platform for the PEI Phase. Since there are no standard services defined for SEC, modules of this type follow the same rules as modules of type Base and typically include some amount of CPU specific assembly code to establish temporary memory for a stack. Modules of this type may optionally produce services that are passed to the PEI Phase in HOBs and those services must be compliant with the PI specification.
PEI_CORE	This module type is used by PEI Core implementations that are complaint with the PI specification.
PEIM	This module type is used by PEIMs that are compliant with the PI specification.
DXE_CORE	This module type is used by DXE Core implementations that are compliant with the PI specification.
DXE_DRIVER	This module type is used by DXE Drivers that are complaint with the PI specification. These modules only execute in the
	boot services environment and are destroyed when ExitBootServices() is called.
DXE_RUNTIME_DRIVER	This module type is used by DXE Drivers that are complaint with the PI specification. These modules execute in both boot services and runtime services environments. This means the services that these modules produce are available after ExitBootServices() is called. If SetVirtualAddressMap() is called, then modules of this type are relocated according to virtual address map provided by the operating system.
DXE_SAL_DRIVER	This module type is used by DXE Drivers that can be called in physical mode before SetVirtualAddressMap() is called and either physical mode or virtual mode after
	SetVirtualAddressMap() is called. This module type is only available to IPF CPUs. This means the services that these modules produce are available after ExitBootServices().
DXE_SMM_DRIVER	This module type is used by DXE Drivers that are loaded into
	SMRAM. As a result, this module type is only available for IA32 and x64 CPUs. These modules only execute in physical mode, and are never destroyed. This means the services that these modules produce are available after ExitBootServices().
UEFI_DRIVER	This module type is used by UEFI Drivers that are compliant with the UEFI Specification. These modules provide services in the boot services execution environment. UEFI Drivers that return EFI_SUCCESS are not unloaded from memory. UEFI Drivers that return an error are unloaded from memory.
UEFI_APPLICATION	This module type is used by UEFI Applications that are compliant with the UEFI Specification. UEFI Applications are always unloaded when they exit.

3.2 Creating a Module

Driver and Library modules follows similar steps for creation:

1. Create or select the package in which the module will be located.
2. Create a directory for the module and put the INF file in the directory.
3. Add package dependencies to the INF file.
4. Add PPI,.Protocol, Guid, PCD,or Library Class dependencies to the INF file.
5. Add [depex] section to the INF file if this module depends on a PPI, Protocol, or Guid and the module type supports this section.
6. Create source file(s) and add relative path of source file(s) to the INF file

3.2.1 Location

A module is released and distributed within a package, so creating or selecting the appropriate package for the new module is the first step.

3.2.1.1 Choosing the Package

A Package in EDK II is used to contain similar definitions and modules. The "similar" is recommended to be determined by following rules:

Industry standard

For example, MdePkg package contains the definitions from PIWG, UEFI, SMBIOS, USB, PCI, etc, which are all industry standards.

Similar technology

For example, OptionRomPkg groups the definitions and modules related to Option Rom technology.

Business reason

For example, IntelFrameworkPkg groups the definitions and modules for Intel framework implementation.

Platform

For example, Nt32Pkg groups the definitions and modules required by Nt32 platform. In addition, a platform package also will provide a DSC and FDF file for platform building.

At the beginning of developing a module, the module developers need to consider the purpose and release process for the module to select the appropriate package.

Note: The packages in https://edk2.tianocore.org are basic core packages.

Generally, a new module should not be created in them.

3.2.1.2 Adding a Module Directory

A module directory should be added to the proper package with the following recommendations:

- Put a library module to "`<Package Root Path>`/Library" directory.

- Put PROTOCOL, PPI, GUID, or Library Class definitions in "<Package Root Path>/Include/Protocol" or "<Package Root Path>/Include/Ppi" or "<Package Root Path>/Include/Guid" or "<Package Root Path>/Include/Library" directory respectively.
- Put a driver module in "<Package Root Path>" directory.
- Put an application module in "<Package Root Path>/Application" directory Use recommend directory name for module as follows:

Table 2 Recommended name convention for module directory

Recommended directory name convention	Module Type
XxxPei	PEIM, PEI_CORE
XxxDxe	DXE_DRIVER, UEFI_DRIVER
XxxRuntimeDxe	DXE_RUNTIME_DRIVER
XxxxDxeSal	DXE_SAL_DRIVER
XxxxLib	Library Instance

3.2.2 Sample: Module Meta File - INF

Each module requires a module INF file in the root directory of the module.

A module is INF file (sometimes referred to as the module meta-file) includes the following items:

- The module's basic information, such as name, GUID, module type, etc.
- The path to any packages the module is dependent on.
- The path to binary files or source files included in the module.
- A list of all interfaces required by the module, i.e. Protocol, Ppi, Guid.
- A list of all PCDs and Library classes required by the module.
- Others, such as dependency section depending on the module type.

3.2.2.1 Example: Application Module INF

```
##
[Defines]
  INF_VERSION                = 0x00010005
  BASE_NAME                  = HelloWorld
  FILE_GUID                  = XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
  MODULE_TYPE                = UEFI_APPLICATION
  VERSION_STRING             = 1.0
  UEFI_SPECIFICATION_VERSION = 0x0002001E
  ENTRY_POINT                = UefiMain

#
# The following information is for reference only and not required by the
# build tools.
#
# VALID_ARCHITECTURES = IA32 X64 IPF EBC
#
##
[Sources.common]
  HelloWorld.c

##
[Packages]
  MdePkg/MdePkg.dec
```

```
##
[LibraryClasses]
  UefiBootServicesTableLib
  UefiApplicationEntryPoint
  UefiLib
  DebugLib
```

3.2.2.2 Example: Library Module INF

Following is a sample INF file for PeiSevicesTablePointerLib.inf library instance:

```
[Defines]
  INF_VERSION    = 0x00010005
  BASE_NAME      = PeiServicesTablePointerLib
  FILE_GUID      = XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
  MODULE_TYPE    = PEIM
  VERSION_STRING = 1.0
  LIBRARY_CLASS  = PeiServicesTablePointerLib|PEIM
  PEI_CORE SEC
  CONSTRUCTOR    = PeiServicesTablePointerLibConstructor

#
# VALID_ARCHITECTURES = IA32 X64 IPF EBC (EBC is for build
#only)
#
[Sources.common]
  PeiServicesTablePointer.c

[Packages]
  MdePkg/MdePkg.dec

[LibraryClasses]
  DebugLib
```

Note: if the library supports the cross module types PEIM, UEFI_DIRVER, DXE_DRIVER. Its module type can be PEIM or UEFI_DRIVER or DXE_DRIVER. If it has the library constructor, its module type must be BASE. BASE type library constructor has no input parameter that can link to any driver type.

3.2.3 Adding a Package Dependency

The [Packages] section of the INF file describes all packages dependencies of this module. The EDK II relative path of dependent package DEC file is described in INF's [Packages] section as follows:

```
[Packages]
  MdePkg/MdePkg.dec
  IntelFrameworkPkg/IntelFrameworkPkg.dec
```

Note: The path is from the root of the $workspace, not from the module directory.

A module should use the following rules for determining package dependency:

- The MdePkg package is required for all modules
- If using definitions from Intel framework specification, dependency on the IntelFrameworkPkg is required.
- Beyond the preceding rules, more package dependencies are introduced by referencing or using surface items, such as Protocol, Ppi, Guid, PCD, Library Class, etc. For example, if a module uses definitions or interfaces from the library class "HiiLib" that is defined in MdeModulePkg package, it

would need to be dependent on the MdeModulePkg.

3.2.4 Adding Source Files

All module source code is described in the INF [sources] section, and is based on following rules:

Note: The UNIX-style path separator "/" should be used, not the Windows*-style "\".

Different architecture sources are put in different source sections.

```
[Sources.common] # source in this section is suitable for all arch
  CheckSum.c ...

[Sources.Ia32] # source in this section is suitable for IA32 arch
  Ia32/Wbinvd.c | MSFT ...
  Ia32/WriteMm7.S | GCC ...

[Sources.X64] # source in this section is suitable for X64 arch X64/Thunk16.asm ...

[Sources.IPF] # source in this section is suitable for IPF arch Ipf/AsmCpuMisc.s ...

[Sources.EBC] # source in this section is suitable for EBC arch Synchronization.c ...
```

- Tool Tags are used describe the sources for different tool chains.

```
[Sources.Ia32]
  Ia32/Wbinvd.c | MSFT # source is built when MSFT tool is used ...
  Ia32/WriteMm7.S | GCC # source is built when GCC tool is used ...
  "$(CC)" -o ${dst} $(CC_FLAGS) $(INC) ${src}
```

- All files should be put under the module's main folder. Do not use ". /" All C include files should also be listed in the sources section.

3.2.4.1 Supported Tool Tag

The following tool tag name is supported by edk2.

Table 3 EDK II supported file extensions

Tool Tag	Description
MSFT	Microsoft Family Tool Chain
INTEL	INTEL Tool Chain
GCC	GCC Tool Chain

3.2.5 Add Library Class References

The library class abstracts some macro or structure definitions and function declarations. The library instance, which is the implementation of a given library class, can be different for different platform or for different phases (SEC ,PEI, DXE) in one platform. Therefore, a module will be dependent on a library class for platform or phase specific behavior.

The steps of using a library class in a module are:

1. Add a dependency for the package containing the library class in INF file

2. Add a dependency for the library class in the INF file
3. Include the library class header file in source code.

To include the library class header in C source code use the following syntax:

```
#include <Library/OemHookStatusCodeLib.h>
```

The header including path is relative to the package's public include path, which is defined in the package (containing the library class, not necessarily the module) DEC's [include] section.

Note: Rememberr to use "/" for the path separator.

3.2.6 Adding PCD References

MACRO and global variables are widely used to make modules customizable in different architectures and different platforms. The EDK II introduces the PCD concept to replace these methods. For example, a "FeatureFlag" type PCD is similar to a project MACRO in that some feature or functionality will be enabled if the PCD's value is TRUE, and vice versa.

A PCD entry is defined by the PCD's Token Space Guid C name, followed by a period "." character and the PCD's C name. In one PCD's token space, each PCD's C name is unique.

```
For example, for PCD gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel, gEfiMdePkgTokenSpaceGuid is the token space name
and PcdDebugPrintErrorLevel is the PCD name, and gEfiMdePkgTokenSpaceGuid is mapped to a GUID defined in MdePkg :
gEfiMdePkgTokenSpaceGuid = { 0x914AEBE7, 0x4635, 0x459b, { 0xAA, 0x1C, 0x11, 0xE2, 0x19, 0xB0, 0x3A, 0x10 }}
```

3.2.6.1 PCD Types

EDK II provides the following types of PCDs:

Feature flag type PCD

This PCD type replaces a switch MACRO to enable or disable a feature. This is a Boolean value, and is either TRUE or FALSE.

Fixed at build type PCD

This PCD type replaces a macro that produced a customizable value, such as the PCI Express base address. The value of this PCD type is determined at build time and is stored in the code section of a module's PE image.

Patchable in module type PCD

This PCD type is very similar to the fixed at build PCD type, but the value is stored in the data section, rather than the code section, of the module's PE image.

Dynamic type PCD

This PCD type is different from the other PCD types listed. The value is assigned by one module and is accessed by other modules in execution time. The PEIM PcdPeim and the DXE Driver PcdDXE each maintain a PCD database that contains all dynamic PCD information used by platform in their respective phase.

3.2.6.2 Add the Package Dependency

When using a PCD in a module, package dependencies must be added to INF's [Packages] section. Two packages are required: the package containing the PCD being used, and the MdePkg. The MdePkg is required because the library class "PcdLib" in MdePkg provides PCD accessing functions and macros.

3.2.6.3 Adding PCDs to module's INF file

To reference a PCD entry, the token space guid name and PCD name must be added into the INF's [PCD] section:

```
[PCD.common]
  gEfiMdePkgTokenSpaceGuid.PcdMaximumLinkedListLength gEfiMdePkgTokenSpaceGuid.PcdMaximumAsciiStringLength gEfiMdePkgTokenSpac
eGuid.PcdMaximumUnicodeStringLength
```

We recommended using a general type "PCD" in the module's INF, so that it allows platform integrators to choose any PCD type for different usage cases. For example, in a desktop platform, memory length can be designated as a "Dynamic" PCD, and its value is produced by the memory discovery driver. However, in some special embedded systems, memory length is designed as a "FixedAtBuild" type PCD, and its value is always fixed.

There are limitations for selecting PCD types:

If a PCD value is used as constant value such as array's length, this PCD's type should be "FixedPcd". For example:

```
UINT8 MySampleArray[FixedPcdGet16(PcdArrayLength)] = {0};
```

Note: Avoid using "FixedPcd" in the library instance modules, because the library instance can link to different modules, and the same PCD may have a different value in different modules.

Note: In a single module, avoid using two PCDs with same name but in different token spaces.

Table 4 INF PCD Section Name

PCD Type	INF File Section Name
General type that can be mapped in any PCD type	PCD
Feature Flag	FeaturePcd
Fixed At Build	FixedPcd
Patchable In Module	PatchPcd
Dynamic	PCD

3.2.6.4 Accessing a PCD value from C source code

To obtain or set a PCD's value from source code, the following steps should be taken:

1. Add the dependency for PcdLib to the module INF file.
2. Add the dependency for MdePkg.
3. Add the include for `<Library/PcdLib.h>` in source code.
4. Use the PcdLib interface to access the PCD's value.

Table 5 PCD access functions

Function name	INF PCD Section Name
PcdGetx()/PcdSetx()	Common get/set function for all PCDs type
FeaturePcdGet()/FeaturePcdSet()	Get/set function for "FeaturePcd"
FixedPcdGetx()	Get function for "FixedPcd"
PatchPcdGetx()/PatchPcdSetx	Get/Set function for "PatchPcd"

For example, the PcdGet32 macro is used to obtain the 32-bit value for all types of PCDs:

```
//
// Check driver debug mask value and global mask
//
if ((ErrorLevel &PcdGet32 (PcdDebugPrintErrorLevel)) == 0)
{
  return;
}
```

3.2.7 Referencing a Protocol, PPI, or GUID

A Protocol, PPI, or a Guid is a UEFI architecture interface item and abstract firmware component's interface. This section introduces how to reference one of these interfaces from a module.

3.2.7.1 Adding Protocol, PPI, or GUID to INF file

The name of Protocol, PPI, or GUID must be added into the corresponding section in the module INF file: [Protocol], [Ppi], and [Guid]. For example:

```
[Protocol] gEfiSampleProtocol
```

3.2.7.2 Including the header file in source code

The Protocol, PPI, or GUID header file may define related structures. Use the following procedure to find the header file path:

- The header file for a Ppi is in `<PackagePath>` \Include\Ppi.
- The header file for a Protocol is in `<PackagePath>` \Include\Protocol.
- The header file for a Guid is in `<PackagePath>` \Include\Guid.

Header files must be explicitly included in the module source code. For example:

```
#include <Protocol/DeviceIo.h>
#include <Ppi/Reset.h>
#include <Guid/GlobalVariable.h>
```

3.2.8 Adding a Dependency to a Module

The dependency expression gives the conditions for executing or dispatching a driver's entry point. This helps determine the dispatch order for PEIM and DXE modules.

An expression consists of one or more Protocol, PPI, or GUID with operators such as "AND", "OR", "TRUE", "FALSE", "NOT" etc.

The Platform Initialization Specification gives detailed syntax for PEIM's dependency expression in the "Dependency Expressions" chapter, and details for the DXE module's dependency expression in the "Dependency Expression Grammar" chapter.

The expression should be put in the INF's depex section as follows:

```
[depex]

gEfiSampleGuid AND gEfiSamplePpiGuid
```

3.3 Additional Steps for Library Instances

3.3.1 Define Produced Library Class

A library instance is always related to a single library class and implements all interfaces defined in the library class. Therefore, the library class name must be specified in the [Defines] section of the library instance INF file as follows:

```
[Defines]
  LIBRARY_CLASS = UefiDriverEntryPoint|DXE_DRIVER DXE_RUNTIME_DRIVER
```

In above sample, UefiDriverEntryPoint is the library class name produced by the library instance. In addition, following "`DXE_DRIVER DXE_RUNTIME_DRIVER`" are the type of modules to the library instance supports.

3.3.2 Define a Library Constructor (Optional)

The library instance module can define a library constructor function that is invoked by the entry point of each linked module. In a library constructor function, some initialization work can be done before any library interface is used:

```
[Defines]
  ......
  CONSTRUCTOR = HobLibConstructor
```

3.3.2.1 Types of library constructor functions

There are three types of library constructor functions, according to the different module type of library instance:

- Library instance in BASE module type:

```
EFI_STATUS
EFIAPI
BaseConstructor (
VOID
)
```

- Library instance in PEIM, PEI_CORE module type:

```
EFI_STATUS
EFIAPI
PeiServicesTablePointerLibConstructor (
IN EFI_PEI_FILE_HANDLE FileHandle,
IN CONST EFI_PEI_SERVICES **PeiServices
)
```

- Library instance in DXE_DRIVER, DXE_CORE, DXE_RUNTIME_DRIVER, UEFI_APPLICATION, UEFI_DRIVER module type:

```
EFI_STATUS
EFIAPI
DxeCorePerformanceLibConstructor (
IN EFI_HANDLE ImageHandle,
IN EFI_SYSTEM_TABLE *SystemTable
)
```

3.3.3 Define a Library Destructor (Optional)

The library instance module can define a library destructor function that is invoked by `ExitDriver()` for `DXE_DRIVER`, `UEFI_DRIVER` etc. In a library destructor function, some un-initialization works can be done.

The destructor function should be declared in an INF file explicitly, as follows:

```
[Defines]
  ......
  DESTRUCTOR = HobLibDestructor
```

The prototype of the destructor function is the same as the constructor function mentioned above.

3.4 Additional Steps for Driver

3.4.1 Define a Driver Entry Point

PEIM or DXE drivers expose the entry point function, which is defined in the INF file [Defines] section as follows:

```
[Defines]
  ......
  ENTRY_POINT = PcdDxeInit
```

The prototype of the module entry point function differs according to module type.

See Table 7 for details.

3.5 EDK II Common Library Class

Library Name	Concept	Description
BaseLib	Base Library	Provides string functions, linked list functions, math functions, and CPU architecture specific functions.
SynchronizationLib	Synchronization Library	Provides synchronization functions
PrintLib	Print Library	Provides services to print a formatted string to a buffer. All combinations of Unicode and ASCII strings are supported.
BaseMemoryLib	Base Memory Library	Provides copy memory, fill memory, zero memory, and GUID functions.
MemoryAllocationL	Memory Allocation Library	Provides services to allocate and free memory buffers of various memory types and alignments.
ib		
DebugLib	Debug Library	Provides ser vices to print debug and assert messages to a debug output device.
PostCodeLib	Post Code Library	Provides services to send progress/error codes to a POST card.
StatusCodeLib	Report Status Code Library	Provides services to log status code records
IoLib	I/O Library	Provide services to access I/O Ports and MMIO registers.
PciExpressLib	PCI Express Library	Provides services to access PCI Configuration Space using the MMIO PCI Express window.
PciLib	PCI Library	Provides services to access PCI Configuration Space.
TimerLib	Timer Library	Provides calibrated delay and performance counter services.
PcdLib	PCD Library	Provides library services to get and set Platform Configuration Database entries.

The MdePkg provides many library classes for developing firmware components based on the UEFI and PI specifications. These library classes are often used in module development and detailed in the MdePkg documentation.

Table 6 Commonly use library classes

Table 7 Module Entry Point and Service Table Libraries

Concept	Description
PEIM Entry Point Library	Module entry point library for PEIM.
UEFI Driver Entry Point Library	Module entry point library for UEFI drivers, DXE Drivers, DXE Runtime Drivers, and DXE SMM Drivers.
UEFI Application Entry Point Library	Module entry point library for UEFI Applications.
PEI Services Table Pointer Library	Provides a service to retrieve a pointer to the PEI Services Table.
UEFI Boot Services	Provides a service to retrieve a pointer to the EFI Boot Services Table. Only

Table Library	available to DXE and UEFI module types.
UEFI Runtime Services Table Library	Provides a service to retrieve a pointer to the EFI Runtime Services Table. Only available to DXE and UEFI module types.
DXE Services Table Library	Provides a service to retrieve a pointer to the DXE Services Table. Only available to DXE module types.

3.6 Module using HII

DXE Modules can publish or update the following resources used in the browser during the BDS phase:

Forms

Describes what type of content needs to be displayed to the user.

Strings

The text-based (UCS-2 encoded) representations of the information typically being referenced by the forms.

Font/Image

The contents rendered on a local system.

Please refer to the UEFI Specification, Chapter 27, Human Interface Infrastructure Overview.

3.6.1 Forms

3.6.1.1 Create VFR resource file

The VFR file is used to describe form resources, per the example below. A VFR file is put into a module's directory and referenced in the INF file [Sources] section, just as with other source code.

```
Example:
VFR file
#define FORMSET_GUID { 0x9e0c30bc, 0x3f06, 0x4ba6, 0x82, 0x88, 0x9,
0x17, 0x9b, 0x85, 0x5d, 0xbe
}
#define FRONT_PAGE_CLASS 0x0000
#define FRONT_PAGE_SUBCLASS 0x0002
#define FRONT_PAGE_FORM_ID 0x1000
#define FRONT_PAGE_ITEM_ONE 0x0001
#define FRONT_PAGE_ITEM_TWO 0x0002
#define FRONT_PAGE_ITEM_THREE 0x0003
#define FRONT_PAGE_ITEM_FOUR 0x0004
#define FRONT_PAGE_ITEM_FIVE 0x0005
#define FRONT_PAGE_KEY_CONTINUE 0x1000
#define FRONT_PAGE_KEY_LANGUAGE 0x1234
#define FRONT_PAGE_KEY_BOOT_MANAGER 0x1064
#define FRONT_PAGE_KEY_DEVICE_MANAGER 0x8567
#define FRONT_PAGE_KEY_BOOT_MAINTAIN 0x9876
#define LABEL_SELECT_LANGUAGE 0x1000
#define LABEL_TIMEOUT 0x2000
#define LABEL_END 0xffff
formset guid = FORMSET_GUID, title = STRING_TOKEN (
                                     STR_FRONT_PAGE_TITLE),
                                     help = STRING_TOKEN (STR_NULL_STRING),
                                     classguid = EFI_HII_PLATFORM_SETUP_FORMSET_GUID,
                                     form formid = FRONT_PAGE_FORM_ID, title = STRING_TOKEN (STR_FRONT_PAGE_TITLE
                                     );
banner title = STRING_TOKEN (STR_FRONT_PAGE_COMPUTER_MODEL), line 0, align left;
banner title = STRING_TOKEN (STR_FRONT_PAGE_CPU_MODEL), line 1, align left;
banner title = STRING_TOKEN (STR_FRONT_PAGE_CPU_SPEED), line 1, align right;
banner title = STRING_TOKEN (STR_FRONT_PAGE_BIOS_VERSION), line 2, align left;
banner title = STRING_TOKEN (STR_FRONT_PAGE_MEMORY_SIZE), line 2, align right;
goto FRONT_PAGE_ITEM_ONE, prompt = STRING_TOKEN (STR_CONTINUE_PROMPT), help = STRING_TOKEN (STR_CONTINUE_HELP),
                        flags = INTERACTIVE,
                        key = FRONT_PAGE_KEY_CONTINUE;
label LABEL_SELECT_LANGUAGE;
//
// This is where we will dynamically add a OneOf type op-code to select
// Languages from the currently available choices
```

```
//
label LABEL_END;
goto FRONT_PAGE_ITEM_THREE, prompt = STRING_TOKEN (STR_BOOT_MANAGER), help = STRING_TOKEN (STR_BOOT_MANAGER_HELP),
                            flags = INTERACTIVE, key = FRONT_PAGE_KEY_BOOT_MANAGER;
goto FRONT_PAGE_ITEM_FOUR, prompt = STRING_TOKEN (STR_DEVICE_MANAGER), help = STRING_TOKEN (STR_DEVICE_MANAGER_HELP),
                           flags = INTERACTIVE, key = FRONT_PAGE_KEY_DEVICE_MANAGER;
goto FRONT_PAGE_ITEM_FIVE, prompt = STRING_TOKEN (STR_BOOT_MAINT_MANAGER),
                           help = STRING_TOKEN (STR_BOOT_MAINT_MANAGER_HELP), flags = INTERACTIVE, key = FRONT_PAGE_KEY_BOOT_M
AINTAIN;
endform;
endformset;
```

3.6.1.2 Publish the Form data

When building, VfrCompile will "build" a .vfr file into the IFR binary as a global array variable in the module image. The name of the global array variable is `<VfrFileName>` + Bin.

For example, the content of Inventory.vfr in the

MdeModulePkg\Universal\DriverSampleDxe driver is compiled into the global array variable InventoryBin.

Module developers should use the following code to publish the VFR global array variable into the HII database:

```
//
// Create HII driver handle, paramter DriverHandle will hold the
// returned new handle.
// HiiLibCreateHiiDriverHandle defined in UefiHiiLib library class.
//
Status = HiiLibCreateHiiDriverHandle (&DriverHandle);
//
// Prepare HII package list, parameter InventoryBin is the VFR form data
// HiiLibPreparePackageList defined in UefiHiiLib library class
//
PackageList = HiiLibPreparePackageList (
                2,
                &mInventoryGuid,
                InventoryBin,
                DriverSampleStrings
                );
ASSERT (PackageList != NULL);
//
// Create package into HII database via EFI_HII_PROTOCOL-
> NewPackageList
//
Status = gHiiDatabase->NewPackageList (
                        gHiiDatabase,
                        PackageList,
                        DriverHandle,
                        &HiiHandle
                        );
```

When a driver only produces one `formset` in a VFR file, the IFR binary could be put into a driver's binary as a PE resource section by setting UEFI_HII_RESOURCE_SECTION to TRUE in the driver's INF file:

```
[Defines]
  INF_VERSION    = 0x00010005
  BASE_NAME      = HiiResourcesSample
  FILE_GUID      = D49D2EB0-44D5-4621-9FD6-1A92C9109B99
  MODULE_TYPE    = UEFI_DRIVER
  VERSION_STRING = 1.0
  ENTRY_POINT    = HiiResourcesSampleInit
  UNLOAD_IMAGE   = HiiResourcesSampleUnload
  #
  # This flag specifies whether HII resource section is generated into PE image.
  #
  UEFI_HII_RESOURCE_SECTION = TRUE
```

Module developers should use the following code to publish HII package data into the HII database:

```
//
// Retrieve HII package list from ImageHandle
//
Status = gBS->OpenProtocol (
                ImageHandle,
                &gEfiHiiPackageListProtocolGuid,
                (VOID **) &PackageList,
                ImageHandle,
                NULL,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
if (EFI_ERROR (Status))
{
  return Status;
}
//
// Publish sample Fromset
//
Status = gBS->InstallProtocolInterface (
                &mDriverHandle,
                &gEfiDevicePathProtocolGuid,
                EFI_NATIVE_INTERFACE,
                &mHiiVendorDevicePath
                );
if (EFI_ERROR (Status))
{
  return Status;
}
//
// Publish HII package list to HII Database.
//
Status = gHiiDatabase->NewPackageList (
                        gHiiDatabase,
                        PackageList,
                        mDriverHandle,
                        &mHiiHandle
                        );
if (EFI_ERROR (Status))
{
  return Status;
}
```

3.6.2 Using Unicode Strings

3.6.2.1 Create .uni file

The Unicode strings are put into the .uni file and referenced in the module's INF [Sources] section like others C files. The .uni file is encoding UCS-2 with a 0xFFFE BOM header. For example:

```
/=#

#langdef en-US "English"

#langdef fr-FR "Francais"

#string STR_INV_FORM_SET_TITLE #language en-US "Network

Controller Information"

#language fr-FR "Mi motor Español de arreglo"

#string STR_INV_FORM_SET_HELP #language en-US "The ABC Network Controller
version information, which includes Firmware versions as well as supported
characteristics"
```

```
#language fr-FR "The ABC Network Controller version information, which
includes Firmware versions as well as supported characteristics"

#string STR_INV_FORM1_TITLE #language en-US "ABC Network

Controller Version Data"

#language fr-FR "Mi Primero

Arreglo Página"

#string STR_INV_VERSION_TEXT #language en-US "Firmware

Revision Date: 02/03/2002"

#language fr-FR "Firmware

Revision Date: 02/03/2002"

#string STR_INV_VERSION_HELP #language en-US "The date of the revision of the
Firmware being used."

#language fr-FR "The date of the revision of the Firmware being used."
```

3.6.2.2 Publish the Unicode String file

```
The file content in .uni file will be parsed

and compiled by build tool to a binary string package array for a module. The name of binary array is constructed as

<ModuleName> { + "Strings". For example, the inventorystring.uni defined in MdeModulePkg\Universal\DriverSampleDxe is compiled
 to binary array: extern UINT8 DriverSampleStrings[]; }
```

Module developers should use the following codes to publish the strings array variable into the HII database:

```
//
// Create HII driver handle, paramter DriverHandle will hold the
// returned new handle.
// HiiLibCreateHiiDriverHandle defined in UefiHiiLib library class.
//
Status = HiiLibCreateHiiDriverHandle (&DriverHandle);
//
// Prepare HII package list, parameter DriverSampleStrings is the
// strings binary data.
// HiiLibPreparePackageList defined in UefiHiiLib library class
//
PackageList = HiiLibPreparePackageList (
                2,
                &mFormSetGuid,
                DriverSampleStrings,
                VfrBin
                );
if (PackageList == NULL)
{
  return EFI_OUT_OF_RESOURCES;
}
//
// Create package into HII database via EFI_HII_PROTOCOL-
> NewPackageList
//
Status = HiiDatabase->NewPackageList (
                        HiiDatabase,
                        PackageList,
                        DriverHandle[0],
                        &HiiHandle[0]
                        );
```

As with other types HII resources, if a module publishs the HII data into a PE resource section, UEFI_HII_RESOURCE_SECTION is set to TRUE in the module's INF file and the following code is used:

```
//
// Retrieve HII package list from ImageHandle
//
Status = gBS->OpenProtocol (
                ImageHandle,
                &gEfiHiiPackageListProtocolGuid,
                (VOID **) &PackageList,
                ImageHandle,
                NULL,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
if (EFI_ERROR (Status))
{
  return Status;
}
//
// Publish sample Fromset
//
Status = gBS->InstallProtocolInterface (
                &mDriverHandle,
                &gEfiDevicePathProtocolGuid,
                EFI_NATIVE_INTERFACE,
                &mHiiVendorDevicePath
                );
if (EFI_ERROR (Status))
{
  return Status;
}
//
// Publish HII package list to HII Database.
//
Status = gHiiDatabase->NewPackageList (
                        gHiiDatabase,
                        PackageList,
                        mDriverHandle,
                        &mHiiHandle
                        );
if (EFI_ERROR (Status))
{
  return Status;
}
```

3.7 Building the module

After the module source is finished, the module INF is added into the DSC file to be built to the expected binary image. Library, EFI and OptionRom images are supported by the EDK II build system.

3.7.1 Add the module INF in package DSC

To build a module, the module INF file is specified in DSC [Components] section. Its relative file path from the workspace (beginning from the package directory), up to and including the INF file name, is added per the following example. Some module may be required to be built for the specific ARCH.

The DSC [Defines] section lists all supported architectures for this platform. The [Components.ARCH] section lists the modules for this architecture. The ARCH must be on the list of all ARCHs from the [Defines] section. The separate [Components] section can be created for the modules that support the different architectures.

3.7.1.1 Example: Package.dsc Components

```
[Defines]
  ......
  SUPPORTED_ARCHITECTURES = IA32|IPF|X64|EBC
  ......

[Components] ## All libraries, drivers and applications may be added here to be built.
  ## this library will be built to the IA32, IPF, X64 and EBC arch version.
  PackageNamePkg/Library/NameOneLib/NameOneLib.inf

[Components.IA32] ## This PEI driver will be built to the IA32 arch version.
  PackageNamePkg/NameTwoPei/NameTwoPei.inf

[Components.X64, Components.EBC] ## This DXE driver will be built to the X64 and EBC arch version.
  PackageNamePkg/NameOneDxe/NameOneDxe.inf
```

3.7.2 Select Library Instances

Note: Skip this step if the module is a library instance.

For drivers and applications, a library instance for each library class dependency must be selected and linked to its binary EFI image.

The module INF [LibraryClasses] section lists all required library classes, which are produced by library instances.

Library instances are implemented for the different purposes. Most of them abstract the generic logic as the common interfaces for the crossing platform modules. Some are for performance and size optimization.

For example, in the MdePkg the BaseMemoryLibOptDxe instances produce the "good performance" BaseMemory library class implementation based on the registers to perform a memory operation. Another example, in the MdePkg the PeiIoLibCpuIo library instance implements the Io library class by using the services of CpuIo PPI to reduce code size.

Depending on platform requirements, different library instances can be set in the DSC [LibraryClasses] section. In the initial development, the generic library instances without any optimization are often used to reduce the development risk. After the module basic functionality is finished, it can be further tuned for size and performance.

The EDK II MdePkg provides many common library instances for user selection. The details for each library instance are found in its INF file or in the MdePkg specification.

The following example lists the most basic library instances.

3.7.2.1 Example Package.dsc LibraryClasses

3.7.2.1.1 Example 1: Generic library instances

```
[LibraryClasses]
  ## Basic Library
  BaseLib|MdePkg/Library/BaseLib/BaseLib.inf
  DebugLib|MdePkg/Library/BaseDebugLibNull/BaseDebugLibNull.inf
  SynchronizationLib|MdePkg/Library/BaseSynchronizationLib/BaseSynchronizat ionLib.inf
  CpuLib|MdePkg/Library/BaseCpuLib/BaseCpuLib.inf
  BaseMemoryLib|MdePkg/Library/BaseMemoryLib/BaseMemoryLib.inf
  PrintLib|MdePkg/Library/BasePrintLib/BasePrintLib.inf
  PcdLib|MdePkg/Library/BasePcdLibNull/BasePcdLibNull.inf
  ## Pci Library
  PciCf8Lib|MdePkg/Library/BasePciCf8Lib/BasePciCf8Lib.inf
  PciExpressLib|MdePkg/Library/BasePciExpressLib/BasePciExpressLib.inf
  PciLib|MdePkg/Library/BasePciLibCf8/BasePciLibCf8.inf
  ## Entry Point Library
  PeimEntryPoint|MdePkg/Library/PeimEntryPoint/PeimEntryPoint.inf
  UefiDriverEntryPoint|MdePkg/Library/UefiDriverEntryPoint/UefiDriverEntryP oint.inf
  UefiApplicationEntryPoint|MdePkg/Library/UefiApplicationEntryPoint/UefiAp plicationEntryPoint.inf
  ## PEI service library
  PeiServicesLib|MdePkg/Library/PeiServicesLib/PeiServicesLib.inf
  PeiServicesTablePointerLib|MdePkg/Library/PeiServicesTablePointerLib/PeiS ervicesTablePointerLib.inf
  ## UEFI and DXE service library
  UefiBootServicesTableLib|MdePkg/Library/UefiBootServicesTableLib/UefiBoot
  ServicesTableLib.inf
  DxeServicesTableLib|MdePkg/Library/DxeServicesTableLib/DxeServicesTableLi b.inf
  UefiRuntimeServicesTableLib|MdePkg/Library/UefiRuntimeServicesTableLib/Ue fiRuntimeServicesTableLib.inf
  DxeServicesLib|MdePkg/Library/DxeServicesLib/DxeServicesLib.inf
  UefiRuntimeLib|MdePkg/Library/UefiRuntimeLib/UefiRuntimeLib.inf
  UefiLib|MdePkg/Library/UefiLib/UefiLib.inf
  DevicePathLib|MdePkg/Library/UefiDevicePathLib/UefiDevicePathLib.inf
  ## This library instance should be provide by chipset.
  TimerLib|MdePkg/Library/BaseTimerLibNullTemplate/BaseTimerLibNullTemplate
  .inf
```

The library instances given above are generic ones for use in all drivers. However, according to the module type and CPU architecture, more suitable library instances can be added into [LibraryClass.ARCHs.ModuleType] section to override the common setting, per the following example:

3.7.2.1.2 Example 2: library instances per module type and CPU architecture

```
[Defines]
  ......

[LibraryClasses]
  ......

[LibraryClasses.IA32, LibraryClasses.X64]
  ## these two optimized library istances only for X86 arch.
  ## they will override the above common base memory instance.
  MdePkg/Library/BaseMemoryLibOptDxe/BaseMemoryLibOptDxe.inf
  MdePkg/Library/BaseMemoryLibOptPei/BaseMemoryLibOptPei.inf

[LibraryClasses.common.UEFI_DRIVER]
```

```
  # these two library intances are set for UEFI driver type module.
  #
  # Debug library instance will override the above NULL instance.
  MemoryAllocationLib|MdePkg/Library/UefiMemoryAllocationLib/UefiMemoryAllo cationLib.inf DebugLib|MdePkg/Library/UefiDebugLib
ConOut/UefiDebugLibConOut.inf
```

For the specific requirement, a driver may select its library instances to override all library instances specified in the [LibraryClasses] section. This chosen library instance is set only for this driver. Such usage is also supported in the DSC as follows:

3.7.2.1.3 Example 3: library instances for a specific driver

```
[Defines]
  ......

[LibraryClasses]
  ......

[Components]
  ## For NameOnDxe driver, its linked PCD library instance is DxePcdLib, not
  ## the above BasePcdLibNull instance.
  PackageNamePkg/NameOneDxe/NameOneDxe.inf {
  <LibraryClasses>
  PcdLib|MdePkg/Library/DxePcdLib/DxePcdLib.inf
  }
```

3.7.3 Configure PCDs

Note: Skip this step for library modules

Modules that consume PCDs (including those consumed by linked libraries) need to have those PCDs configured in the DSC. The configured PCDs will be applied both to the module and to its linked library instances. PCDs are declared in package DEC file. When the PCD's value is the same as the default value defined in the DEC, those PCDs need not be specified in DSC again.

In the DSC, the PCD type and value can be configured according to the platform requirements. The PCD type must be set to single type in a DSC file. If not specified in DSC, the PCD type will be same as its declaration PCD type in the package DEC file. If a PCD is declared to support multiple PCD types, the default PCD type is a fixed PCD.

The PCD value may set the different values for the different drivers. If its value is not specified, the value will be from its declaration default value to the chosen PCD type in the package DEC file.

3.7.3.1 PCD types

- PcdsFeatureFlag,
- PcdsFixedAtBuild, PcdsPatchableInModule, PcdsDynamic.

3.7.3.2 Feature Flag PCD

If a PCD is declared as PcdsFeatureFlag, it must be of the FeatureFlag PCD type and BOOLEAN data type. When this type of PCD is used in a module, it must be specified in the [FeaturePcd] section of the module INF.

Note: Only FeaturePcdGet API can access this PCD type.

3.7.3.3 Fixed PCD

If a PCD value is decided during the build time, its type can be set to

PcdsFixedAtBuild. When this PCD type is used in module, it can be specified in the [FixedPcd] or [PCD] section of the module INF. In addition, FixedPcdGet and PcdGet API can be used to access this type PCD in the module source code.

When FixedPcdGet API is used, this type PCD can be used as the array index in a driver.

Note: For a library, no such usage is supported.

3.7.3.4 Patchable PCD

If the PCD value needs to be modified in the binary image, its type will be

PcdsPatchableInModule. When this type PCD is used in module, it can be specified in the [FatchPcd] or [PCD] section of the module INF. In addition,

PatchPcdGet/PatchPcdSet and PcdGet/PcdSet API can be used to access this type PCD in the module source code.

3.7.3.5 Dynamic PCD

If PCD value is obtained from the runtime environment, its type must be Dynamic. If a dynamic PCD is from a PCD database that shares data between drives, its type will be PcdsDynamicDefault. If a dynamic PCD is related to a UEFI variable, its type will be PcdsDynamicHII.

When this type PCD is used in a module, it must be specified in the [PCD] section of the module INF. Only PcdGet/PcdSet API can be used to access this type PCD in module source code.

Dynamic type PCDs must be configured in the DSC file to set the dynamic type and the initial value for the whole platform, which cannot inherit from its declaration DEC file and cannot be overridden by a driver.

The following example gives each type of PCD setting:

3.7.3.5.1 Package.dsc PCDs showing each type of PCD setting

```
[PcdsFeatureFlag]
  #PcdName | Pcdvalue
  gEfiMdeModulePkgTokenSpaceGuid.PcdHelloWorldPrintEnable|TRUE

[PcdsFixedAtBuild]
  #StringPcdName | StringValue| StringType| StringMaxSize
  gEfiMdeModulePkgTokenSpaceGuid.PcdHelloWorldPrintString |L"UEFI Hello
  World!\n"|VOID*|100

[PcdsPatchableInModule]
  #Pring level can be modifed in binary image
  gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel|0x80000046

[PcdsDynamicDefault]
  #Default print times is 1, its can be modifed in runtime.
  gEfiMdeModulePkgTokenSpaceGuid.PcdHelloWorldPrintTimes|0x1
```

```
[PcdsDynamicHii]
  #time out
  #PcdName | Uefi Variable name | Uefi Variable Guid | Offset | Default value
  gEfiIntelFrameworkModulePkgTokenSpaceGuid.PcdPlatformBootTimeOut|L"Timeou t"|gEfiGlobalVariableGuid|0x0|5
```

PCD section also supports ARCH option to set PCDs value only for a specific ARCH image. It can be set in [PcdsType.ARCHs] section. For example:

3.7.3.5.2 Example: Package.dsc PCDs for a specific ARCH image

```
[PcdsFixedAtBuild.IPF]
gEfiMdePkgTokenSpaceGuid.PcdIoBlockBaseAddressForIpf|0x0ffffc000000
```

PCD value can be also set only for a driver to override the PCD section setting. However, Dynamic type PCD must be set as the global value, which cannot be overridden by a driver. Such usage is also supported in DSC like:

3.7.3.5.3 Example: Package.dsc dynamic PCDs

```
[Components]
  # For NameOnDxe driver, its print level PCD value is 0x80000000, not same to
  # the above setting 0x80000046.
  PackageNamePkg/NameOneDxe/NameOneDxe.inf {
  <PcdsPatchableInModule>
  gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel|0x80000000 }
```

3.7.4 Customize Build Options

Build options are the different compiler options to build the image under the different tool chain. They are defined in $(WORKSPACE)/Conf/toolsdef.txt file. This file provides the common compiler options for each tool chain tag. The compiler options are grouped into two main types: compile option and link option. The full option list can refer to tools_def.txt and _EDK II Build Specification.

The following example lists the usual compiler and link option.

3.7.4.1 Example: Tools_def.txt

When the common build options in tools_def.txt do not satisfy the development requirement, they can be extended or replaced.

```
  #
  # Build option syntax
  # TARGET_TOOLCHAIN_ARCH_COMMANDTYPE_ATTRIBUTE = build option
  #
  # TARGET is RELEASE or DEBUG
  # TOOLCHAIN is tool tag name, MYTOOLS is a tag with VS2005 tool chain
  # ARCH is the tool cpu family architecture.
  # COMMANDTYPE is the build option name. CC is compile, DLINK is link.
  # ATTRIBUTE is FLAGS for the build option.
  #
  # Debug related options of VS2005 compiler
  DEBUG_MYTOOLS_IA32_CC_FLAGS    = ... /Zi /Gm
  DEBUG_MYTOOLS_IA32_DLINK_FLAGS = ... /DEBUG
```

The EDK II build system provides four levels of override mechanisms to customize the compiler options. The options override each other in the order given.

3.7.4.2 Modifying Tools_def.txt

Directly modify build options in tool_def.txt, which changes the compiler options and affects all modules and platforms in same workspace.

3.7.4.3 Modifying an INF file

Add the additional compiler option in module INF [BuildOptions], which applies for this module to be built in any build DSC.

3.7.4.3.1 Example: Module.inf

```
[BuildOptions]
  #
  # Tool Chain Family: MSFT, INTEL, GCC for the different compiler tools.
  # The different compiler tools have the different compiler options.
  # * is not specific TARGET, TOOLTAGNAME, ARCHs.
  #
  # '=' append the additional option to the tail.
  # Append /FAsc compile option only for this module
  #
  MSFT:*_*_*_CC_FLAGS = /FAsc
  #
  # '==' replace all options by using new setting
  #
  MSFT:*_*_*_DLINK_FLAGS = = /DEBUG
```

3.7.4.4 Modifying DSC platform options

Add the additional compiler option in build DSC [BuildOptions] section, which will update the compiler options for all modules described in same DSC.

3.7.4.4.1 Example: Package.dsc BuildOptions -compler options for all modules

```
[BuildOptions]
  #
  # Append /Od Compile option in DSC to disable optimiaztion for all modules
  #
  MSFT:*_*_*_CC_FLAGS = /Od
```

3.7.4.5 Modifying a DSC for a single module

Add the additional compiler option in the build DSC [Components] section for a module, which applies for it only in this build DSC.

3.7.4.5.1 Example: Package.dsc BuildOptions-single module compiler options

```
[Components]
  #
  # Append the debug compile option only to NameOneDxe driver
  #
  PackageNamePkg/NameOneDxe/NameOneDxe.inf {
  <BuildOptions>
  MSFT:*_*_*_CC_FLAGS = /Od
  }
```

The higher-level setting will append new options in the tail or replace all options. The four methods work together meet the platform build requirements.

Note: The last two usages are recommended. Both only modify the DSC file.

3.7.5 Build module image

After the settings given above, the EDK II build command can be called to build the module to the binary image. It has many build configurations to support the differing build requirements. The usual used build options are introduced in the following manner:

3.7.5.1 Example: Build option

Build -p Package.dsc -m Module.inf -a ARCH -b TARGET -t TOOLTAG

3.7.5.2 Build Package (-p option)

All modules in the [Components] section of the specified package DSC will be built if the build module option is not added. If specified more than once on the command line, the final selection is used.

3.7.5.2.1 Example: Build -p option

```
# Build all modules in PackageOne DSC
Build -p PackageOnePackageOne.dsc
# Build all modules in PackageTwo DSC
Build -p PackageOnePackageOne.dsc -p PackageTwoPackageTwo.dsc
```

3.7.5.3 Build Module (-m option)

When a single specified module is built, it must be in the [Components] section of the specified DSC. If this option is not added, all modules in the DSC will be built. If specified more than once on the command line, the final selection is used.

3.7.5.3.1 Example: Build -m option

```
# Build single module One in PacakgeOne DSC
Build -p PackageOnePackageOne.dsc -m PackageOneOneOne.inf
# Build single module Two in PackageOne DSC
Build -p PackageOnePackageOne.dsc -m PackageOneOneOne.inf -m
PackageOneTwoTwo.inf
```

3.7.5.4 Build ARCH (-a option)

The supported ARCH option is IA32, X64, IPF and EBC. New arch types may be added in the future. The module with the settings given above will be built to the specified ARCH. If specified more than once on the command line, each ARCH is built sequentially.

3.7.5.4.1 Example: Build -a option

```
# Build all modules in PacakgeOne DSC to IA32 arch
Build -p PackageOnePackageOne.dsc -a IA32
# Build all modules in packageOne DSC to IA32 and X64 arch both
Build -p PackageOnePackageOne.dsc -a IA32 -a X64
```

3.7.5.5 Build Target (-b option)

The supported target is DEBUG and RELEASE, which are for the different compiler option settings. The module will be built under the specified target. If specified more than once on the command line, each Target is built sequentially.

3.7.5.5.1 Example: Build -b option

```
# Build all modules in PacakgeOne to IA32 arch
Build -p PackageOnePackageOne.dsc -b DEBUG
# Build all modules in packageOne to IA32 and X64 arch both
Build -p PackageOnePackageOne.dsc -b DEBUG -b RELEASE
```

3.7.5.6 Build Tool Tag Name (-t option)

Tool tag name are defined in Conf\Tools_def.txt file to represent a compiler tool chain. For example, MYTOOLS is a default tool tag name to Microsoft VS2005 tool chain. The module will be built by the specified tool chain. If specified more than once on the command line, each used tool chain is used sequentially.

3.7.5.6.1 Example: Build -t option

```
# Build all modules in PacakgeOne by MYTOOLS tool chain
Build -p PackageOnePackageOne.dsc -t MYTOOLS
# Build all modules in packageOne by MYTOOLS and ICC tool chain
Build -p PackageOnePackageOne.dsc -t MYTOOLS -t ICC
```

If the options given above are not specified with the build command, their default settings will be from Conf\target.txt file. Details of build command are referenced in the EDK II User Manual 3.2.2 section.

After applying the settings and build given above, the library and EFI image can be generated into the build output directory. The build output directory is introduced in detail in EDK II User Manual, section 3.3 The library is generated into the OUTPUT directory, and the EFI image is generated into the DEBUG directory.

3.7.5.7 Example: Build HelloWorld

```
Build -p MdeModulePkg/MdeModulePkg.dsc -m
MdeModulePkg/Application/HelloWorld/HelloWorld.inf -a IA32 -b DEBUG -t
MYTOOLS
HelloWorld.efi will be generated in DEBUG directory:
$(WORKSAPCE)/Build/MdeModulePkg/DEBUG_MYTOOLS/IA32/MdeModulePkg/Applicati on/HelloWorld/HelloWorld/DEBUG, OUTPUT
In the build DEBUG directory, the following files are created: the EFI image, intermediate files, AutoGen.h, AutopGen.c and th
e Module.map file.
AutoGen.h and AutopGen.c files are generated for each module by the EDK II build tool based on the required module information
. They declare the dependent PCDs, Guid Values and include the module entry point related functions. Those AutoGen functions a
re referred to in the ModuleEntryPoint library instance. For each module, the entry point function first calls AutoGen code, t
hen enters into module functions.
```

Module.map is generated by a compiler tool to list all functions and their relative addresses in this module. They can be used to locate the module function address at run time.

3.7.6 Build EFI Option Rom image

An EFI Option Rom image is a standard EFI image. It can be built by the build module command mentioned in the section given above. The only difference is that its INF includes the related PCI option in [Defines] section. When PCI option is set in the module INF, this module will be built to both EFI and Option Rom images. In the build DEBUG directory, ModuleName.efi and ModuleName.rom will be generated.

The following example contains all PCI options required to create the EFI option rom image.

3.7.6.1 Example: OptionRom INF-all PCI options

```
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME   = OptionRomOne
  FILE_GUID   = XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
  MODULE_TYPE = UEFI_DRIVER
  ENTRY_POINT = UefiMain
  # PCI option for VendorId, DeviceId, ClassCode and Revision
  #
  # PCI option is used to create PCI option rom image.
  PCI_VENDOR_ID  = 0x8086
  PCI_DEVICE_ID  = 0x29c2
  PCI_CLASS_CODE = 0x030000
  PCI_REVISION   = 0x1000

[Sources.common]
  OptionRom.c

[Packages]
  MdePkg/MdePkg.dec

[LibraryClasses]
  UefiBootServicesTableLib
  UefiDriverEntryPoint
  UefiLib
  DebugLib
```

Of the PCI options, VendorId, DeviceId and ClassCode are required.The PCI revision is optional. If the PCI revision is not specified, the default revision is 0x0.

3.7.7 Common build module breaks

The following lists the common build module breaks and their fixes.

error 4000: Value of Guid [gCNameGuid] is not found under [Guids] section in MdePkg\MdePkg.dec

The module INF is missing a package in the [Packages] section. The same error can happen for any Guid, Ppi, Protocol, LibraryClass and PCD used by this module. The package DEC that declares them must be added to the [Packages] section of this module INF.

error LNK2001: unresolved external symbol _gCNameGuid

The module INF is missing a Guid in the [Guids] section. The same error can happen for any Guid, Ppi, Protocol, LibraryClass and PCD used by this module. The Guid, Ppi, or Protocol) CName needs to be added into the [Guids], [Ppis], or [Protocols] sections, respectively.

error LNK2001: unresolved external symbol _LibraryFunctionName

The module INF is missing a library class in the [LibraryClasses] section. This prevents the library instance from being linked to the module. The library class that includes LibraryFunctionName must be added into the [LibraryClasses] section of this module INF.

```
warning C4013: 'FeaturePcdGet' undefined; assuming extern returning int
PcdLib.h is missing in the module source code. When a PCD is used in a module, PcdLib APIs are referenced to access the PCD. T
he PcdLib header file must included in this module source code.
error 0010: File name case mismatch MdeModulePkg\Application\Helloworld\HelloWorld.inf MdeModulePkg\Application\HelloWorld\Hel
loWorld.inf [in file system]
```

Note: Lower case 'w' in' Helloworld', in the first path

The module INF file path specified in the DSC is not same as its file path in the file system. The same error may occur for the source file path specified in the [Source] section of a module INF. According to the error information, the file path in the DSC or INF needs to be corrected to its file path in file system. All files must have their name and case set the same in the metadata files as in the file system.

error 4000: Instance of library class [NameOneLib] is not found consumed by module [MdeModulePkg\Application\HelloWorld\HelloWorld.inf]

The DSC file is missing a library class to library instance mapping for the given library. If the module does not depend on the library class, the unused library class can be removed from the [LibraryClasses] section of module INF to fix this error. If the module requires this library class, the corresponding library instance mapping must be added into the [LibraryClasses] section of the DSC file.

3.8 Debugging a Module

3.8.1 Required steps for debugging a module

The following steps are required before starting to debug a module.

- "Build -b DEBUG" command

  EDK II supports generating DEBUG/RELEASE target. A different target causes different build options. The "BuildTarget" field in target.txt works with the "ToolChain" field to determine the actual path of the compiler tool-chain and build option. Developers can directly open $(WORKSPACE)\Conf\target.txt and change "TARGET = DEBUG" for the debug tip. Developers also can use the command line to override this value, such as "build -b DEBUG" for debug tip.

- Choose the proper DebugLib library instance

  For the DebugLib library class, MdePkg and IntelFrameworkModulePkg core packages provide several library instances, which include `BaseDebugLibNull`, `BaseDebugLibSerialPort`, `UefiDebugLibConOut`, `UefiDebugLibStdErr`, and `PeiDxeDebugLibReportStatusCode`. Developers can choose proper DebugLib library instance in the package DSC file according to the actual requirements.

- Configure the Pcds consumed by DebugLib

  The DebugLib library class header defines two PCDs to be used for debug library configuration.

  The PCDs related to debug ability include `PcdDebugPropertyMask` and `PcdDebugPrintErrorLevel`. The former is used to control enable/disable print/assert abilities, and determines if the ASSERT macro is implemented through CpuDeadLoop or BreakPoint. For the latter, developers can set various values to control if the error information should be printed or filtered.

- Change build option

  Developers can modify or override the module build option. For example, a developer can use the "/Od" option for the Microsoft compiler to disable the optimization of the compiler and avoid disordered instructions. The debug tip can also use the "/FAsc" option for the Microsoft compiler to generate a source and assembly (.cod) file to help debug.

3.8.2 Basic debugging methods

Following are three basic methods for debugging:

- Using DEBUG print statement.

In EDK II project, there is a set of PCD to enable/disable debug capability. Developer can turn on the functionality when starting to debug. Therefore, the DEBUG print statements can be used to get information desired.

- CpuDeadLoop()

Developers can use an API to halt control flow, which is helpful to find the location of an issue quickly.

- Module's Map file

Currently, EDK II generates a corresponding FV map file for every module. Developers can depend on the base address of a loaded module and map file to calculate the memory address of functions.

4 UEFI APPLICATIONS

UEFI Application is an EFI image of the type

EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION. This image is executed and automatically unloaded when the image exits or returns from its entry point.

OS loader is a special type of application that normally does not return or exit. Instead, it calls the EFI Boot Service gBS->ExitBootServices() to transfer control of the platform from the firmware to an operating system.

The EFI Shell is a special EFI application that provides the user with a command-line interface.

4.1 Begin with INF file

The following is an example of the INF file of an application named SampleApplication. For UEFI Application, the `MODULE_TYPE` entry should be `UEFI_APPLICATION.` The difference compared to Pei/Dxe/Uefi driver is that UEFI_APPLICATON has no dependency relationship section.

```
[Defines]
  INF_VERSION                = 0x00010005
  BASE_NAME                  = SampleApplication
  FILE_GUID                  = XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
  MODULE_TYPE                = UEFI_APPLICATION
  ENTRY_POINT                = SampleApplicationEntryPoint
  UEFI_SPECIFICATION_VERSION = 0x0002001E

[Sources]
  Sample.c

[Packages]
  MdePkg/MdePkg.dec

[LibraryClasses]
  UefiApplicationEntryPoint
  DebugLib

[protocol]
  gSampleProtocolGuid

[Guids] gSampleGuid
```

Note: If a module is dependent on the new definitions or features in

---

EFI_BOOT_SERVICES or UEFI_RUNTIME_SERVICES-defined in UEFI specifications from version 2.1 forward- the hex version needs to be given in INF file [Defines] section's UEFI_SPECFIICATION_VERSION field.

4.2 Write UEFI Application Entry Point

Developers must focus on specifying the entry point of UEFI application in the `[Defines]` section of INF file.

Its prototype is list below:

```
EFI_STATUS
EFIAPI
UefiMain (
```

`IN EFI_HANDLE` ImageHandle `,`

`IN EFI_SYSTEM_TABLE` *SystemTable

`);`

As can be seen, there are two parameters for UEFI application entry point,

`ImageHandle` and `SystemTable`. `ImageHandle` refers to the image handle of the UEFI application. `SystemTable` is the pointer to the EFI System Table.

The following is a full UEFI_APPLICATION example located at

$WORKSPACE\MdeModulePkg\Application\HelloWorld. It shows how to print a "UEFI Hello World!" string to console.

Note: This application uses several pcds to demonstrate the usage of PCD. Readers can obtain the default value of these pcds from the

---

$WORKSPACE\MdeModulePkg\MdeModulePkg.dec file.

```
EFI_STATUS
EFIAPI
UefiMain (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  UINT32  Index;
  Index = 0;
  //
  // Three PCD type (FeatureFlag, UINT32 and String) are used as the
  // sample.
  //
  if (FeaturePcdGet (PcdHelloWorldPrintEnable)) {
    for (Index = 0;
         Index < PcdGet32 (PcdHelloWorldPrintTimes); Index ++) {
      //
      // Use UefiLib Print API to print string to UEFI console
      //
      Print ((CHAR16 *)PcdGetPtr (PcdHelloWorldPrintString));
    }
  }
  return EFI_SUCCESS;
}
```

4.3 Get Service Tables

UEFI Application may consume the UEFI Boot Services, UEFI Runtime Services and UEFI System Table.

Refer to UEFI Specification for definitions and detailed descriptions of UEFI Boot Services, UEFI Runtime Services, and UEFI System Table.

EDK II provides UefiBootServicesTableLib, UefiBootServicesTableLib and

UefiRuntimeServicesTableLib to facilitate developer in accessing those services. The following table lists the global symbol provided by those libraries.

Table 8 Global Symbol can be used by UEFI Application

	`Global Variable`	`Library Class`
UEFI System Table	`gST`	UefiBootServicesTableLib
UEFI Boot Services Table	`gBS`	UefiBootServicesTableLib
UEFI Runtime Services Table	`gRT`	UefiRuntimeServicesTableLib

4.4 Communicating with a UEFI driver

4.4.1 Protocol

Uefi Application can use the following protocol service to access the protocol interfaces produced by UEFI drivers.

Services to retrieve the protocol:

- `LocateProtocol()`
- `HandleProtocol()`
- `OpenProtocol()`

Note: Uefi Application cannot use the InstallProcotol service or corresponding Libraries to install the protocol. This is because the UEFI application is unloaded after returning from the entry point. Therefore, it is meaningless to install this protocol.

4.4.2 Variable

Variables are defined as key/value pairs that consist of identifying information plus the attributes (the key) and arbitrary data (the value). Variables are intended for use as a means to store data that is passed between the EFI environment implemented in the platform and EFI OS loaders and other applications that run in the EFI environment.

UEFI application can read and write variable via UEFI Runtime Services

`GetVariable()` and `SetVariable()`. Because UEFI application must run after the Dxe/UEFI driver, Variable Arch protocol must be installed.

5 UEFI DRIVERS

The UEFI Specification defines the UEFI Driver Model. Drivers that follow the UEFI Driver Model are UEFI drivers. The driver initialization routine of a UEFI driver is not allowed to touch any hardware. Instead, it installs an instance of the `EFI_DRIVER_BINDING_PROTOCOL` on the ImageHandle of the UEFI driver.

Later on, the driver may get calls through the `EFI_DRIVER_BINDING_PROTOCOL` to test for support of a given piece of hardware. The test to determine if a driver supports a given controller must be performed in as little time as possible without causing any side effects on any of the controllers it is testing. Most of the controller initialization is done in the start and stop services of the `EFI_DRIVER_BINDING_PROTOCOL`.

5.1 Begin With INF File

The [Defines] section of the INF must set `MODULE_TYPE` to `UEFI_DRIVER`. Example:.

```
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME   = SampleDriverDxe
  FILE_GUID   = XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
  MODULE_TYPE = UEFI_DRIVER
  ENTRY_POINT = SampleDriverEntryPoint
```

Note: A UEFI driver has no [depex] section in the INF file. It always depends on all dxe architectural protocols. To force this, the UEFI driver entry point library instance appends all dxe architectural protocol dependency relationships into the depex section of the module image.

Note: If module dependent on the new definitions/features in

EFI_BOOT_SERVICES/UEFI_RUNTIME_SERVICES-defined in UEFI specifications from version 2.1 forward-the hex version need to be given in INF file [Defines] section's UEFI_SPECFIICATION_VERSION field.

5.2 Write the UEFI Driver entry point

The following table lists the most common protocols used in UEFI driver entry point.

Table 9 Protocols Used to Separate the Loading and Starting/Stopping of Drivers

Protocol	Description
Driver Binding Protocol	Provides functions for starting and stopping the driver, as well as a function for determining if the driver can manage a particular controller. The UEFI Driver Model requires this protocol.
Component Name Protocol	Provides functions for retrieving a human-readable name of a driver and the controllers that a driver is managing. While the UEFI Specification lists this protocol as optional, the Developer's Interface Guide for 64-bit Intel Architecturebased Servers (hereafter referred to as "DIG64 specification" or "DIG64") lists this protocol as required for Itanium-based platforms.
Driver Diagnostics Protocol	Provides functions for executing diagnostic functions on the devices that a driver is managing. While the UEFI
	Specification lists this protocol as optional, DIG64 lists this protocol as required for Itanium-based platforms.

The UefiLib library class is provided to simplify the driver entry point of a UEFI driver.

5.2.1 Example: APIs in UefiLib

The two APIs in UefiLib are shown below:

```
EFI_STATUS
EfiLibInstallDriverBinding (
    IN CONST EFI_HANDLE ImageHandle ,
    IN CONST EFI_SYSTEM_TABLE * SystemTable ,
    IN EFI_DRIVER_BINDING_PROTOCOL * DriverBinding ,
    IN EFI_HANDLE DriverBindingHandle
);

EFI_STATUS
EfiLibInstallAllDriverProtocols2 (
    IN CONST EFI_HANDLE ImageHandle ,
    IN CONST EFI_SYSTEM_TABLE * SystemTable ,
    IN EFI_DRIVER_BINDING_PROTOCOL * DriverBinding ,
    IN EFI_HANDLE DriverBindingHandle ,
    IN CONST EFI_COMPONENT_NAME_PROTOCOL * ComponentName , OPTIONAL
    IN CONST EFI_COMPONENT_NAME2_PROTOCOL * ComponentName2 , OPTIONAL
    IN CONST EFI_DRIVER_CONFIGURATION_PROTOCOL * DriverConfiguration ,OPTIONAL
    IN CONST EFI_DRIVER_CONFIGURATION2_PROTOCOL * DriverConfiguration2 ,OPTIONAL
    IN CONST EFI_DRIVER_DIAGNOSTICS_PROTOCOL * DriverDiagnostics ,OPTIONAL
    IN CONST EFI_DRIVER_DIAGNOSTICS2_PROTOCOL * DriverDiagnostics2 OPTIONAL
);
```

5.2.2 Example: Entry point to the Abc driver

The following shows an example of the entry point to the Abc driver that installs the Driver Binding Protocol `gAbcDriverBindingProtocol`, the Component Name Protocol `gAbcComponentName`, the Component Name2 Protocol `gAbcComponentName2`, the

Diagnostic Protocol `gAbcDriverDiagnostic`s and the Diagnostic2 Protocol

`gAbcDriverDiagnostics2` onto the Abc driver's image handle. This driver simply returns the status from the UefiLib function `EfiLibInstallAllDriverProtocols2()`

```
EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
  AbcDriverBindingSupported,
  AbcDriverBindingStart,
  AbcDriverBindingStop,
  0xa,
  NULL,
  NULL
};
EFI_COMPONENT_NAME_PROTOCOL gAbcComponentName = {
  AbcComponentNameGetDriverName,
  AbcComponentNameGetControllerName,
  "eng"
};
EFI_COMPONENT_NAME2_PROTOCOL gAbcComponentName2 = {
  (EFI_COMPONENT_NAME2_GET_DRIVER_NAME) AbcComponentNameGetDriverName,
  (EFI_COMPONENT_NAME2_GET_CONTROLLER_NAME)
  AbcComponentNameGetControllerName,
  "en"
};
EFI_DRIVER_DIAGNOSTICS_PROTOCOL gAbcDriverDiagnostics = {
  AbcDriverDiagnosticsRunDiagnostics,
  "eng"
};
EFI_DRIVER_DIAGNOSTICS2_PROTOCOL gAbcDriverDiagnostics2 = {
  (EFI_DRIVER_DIAGNOSTICS2_RUN_DIAGNOSTICS) gAbcDriverDiagnosticsRunDiagnostics,
  "en"
};
EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  //
  // Initialize a simple EFI driver that follows the EFI Driver Model
  //
  return EfiLibInstallAllDriverProtocols (
           ImageHandle, // Driver's image handle
           SystemTable, // EFI System Table Pointer
           &gAbcDriverBinding, // Required parameters
           ImageHandle,
           // Handle for driver-related protocols &gAbcComponentName,
           // Component Name Procol. May be NULL.
           &gAbcComponentName2, // Component Name2 Procol. May be NULL.
           NULL, // Configuration Protocol. May be NULL.
           NULL // Configuration Protocol2 May be NULL.
           &gAbcDriverDiagnostics, // Diagnostics Protocol. May be NULL.
           &gAbcDriverDiagnostics2,// Diagnostics Protocol2 May be NULL.
           );
}
```

5.3 Get Service Tables

UEFI drivers may consume the UEFI Boot Services, UEFI Runtime Services, and UEFI System Tables that are defined in the UEFI Specification.

EDK II provides the `UefiBootServicesTableLib` and `UefiRuntimeServicesTableLib` libraries to facilitate developer to access those services. The following table lists the global variables provided by those libraries.

Table 10 Table Global Variables

	`Global variable`	`Library Class`
UEFI System Table	`gST`	UefiBootServicesTableLib
UEFI Boot Services Table	`gBS`	
UEFI Runtime Services Table	`gRT`	UefiRuntimeServicesTableLib

5.4 Communication between UEFI Drivers

This section describes the communication methods used by UEFI drivers.

5.4.1 Protocol

UEFI drivers can use protocol services to access protocol interfaces produced by other modules.

The UEFI Specification defines a group of boot services to handle protocols, including:

Services to install protocols:

- `InstallProtocolInterface()`
- `ReInstallProtocolInterface()`
- `InstallMultipleProtocolInterfaces()`

Services to retrieve protocols:

- `LocateProtocol()`
- `HandleProtocol()`
- `OpenProtocol()`

Section 8.4.1 provides an example of usage.

5.4.2 Variable

UEFI drivers can read and write variables via the UEFI Runtime Services `GetVariable()` and `SetVariable()`.

When using this service, the distinction between a UEFI driver and a Dxe driver is that a Dxe driver must explicitly point out the dependency relationship for

`EFI_VARIABLE_ARCH_PROTOCOL` and `EFI_VARIABLE_WRITE_ARCH_PROTOCOL` in the [depex] section of the Dxe driver's INF file , but a UEFI driver does not have this section in the UEFI driver's INF file

---

Note: For UEFI drivers, the EDK II build system will automatically append the dependency information inherited from the **UefiEntryPointLib** into the image section. This causes UEFI drivers to run after all Dxe architectural protocols are installed.

Section 8.4.2, provides an example of usage.

6 SEC MODULE

The SEC module is the first module executed after power-on. It is responsible for configuring the PEI environment's memory call stack. In addition, this module discovers and passes control to PEI Core and hands information to the PEI Foundation.

6.1 Beginning to Write the INF File

The following is a sample for `[Defines]` section of the SEC module:

```
[Defines]
  INF_VERSION    = 0x00010005
  BASE_NAME      = SampleSec
  FILE_GUID      = XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
  MODULE_TYPE    = SEC
  VERSION_STRING = 1.0
  ENTRY_POINT    = _ModuleEntryPoint
```

For a physical platform, `MODULE_TYPE` must be set to `SEC`. For an Emulation Platform, the SEC module's `MODULE_TYPE` must be set to `SEC` or `USER_DEFINED`.

For IA-32 Intel Architecture `_ModuleEntryPoint` is the default entry point for the SEC module.

For Itanium Processor Family platform, the entry point is configurable, such as SAMPLE_ENTRY. Nevertheless, this entry point should be added in `[BuildOptions]` section as following,

```
[Defines]
  ENTRY_POINT = SAMPLE_ENTRY

[BuildOptions]
  INTEL:*_*_IPF_DLINK_FLAGS = /ENTRY: SAMPLE _ENTRY
  MSFT:*_*_IPF_DLINK_FLAGS  = /ENTRY: SAMPLE _ENTRY
  GCC:*_*_IPF_PP_FLAGS      = --entry _ SAMPLE _ENTRY
```

The implementation of the SEC entry point is commonly in the assembly language.

6.2 Setup Pre-Memory Environment

For IA-32, the main tasks of the SEC module are to:

1. Populate Reset Vector Data structure
2. Save BIST status
3. Enable protected mode
4. Configure temporary RAM (not only limited in processor cache) by using MTRR to configure CAR.

For Itanium Processor Family, the main tasks of the SEC module are to:

1. Save INIT, MCA and RESET vectors.
2. Configure temporary RAM (not only limited in processor cache) by using MTRR to configure CAR.

After enabling temporary RAM, the SEC module must configure Stack and Heap in the temporary memory, so that C code can be run later.

6.3 Prepare for Data PEI Foundation

Upon completion, SEC will call the PEI Foundation entry point and transfer control to it.

The PEI foundation Entry Point is defined as,

```
typedef

VOID

EFIAPI

(*EFI_PEI_CORE_ENTRY_POINT)(

IN CONST EFI_SEC_PEI_HAND_OFF *SecCoreData,

IN CONST EFI_PEI_PPI_DESCRIPTOR *PpiList );
```

6.3.1 EFI_SEC_PEI_HAND_OFF * SecCoreData

SEC conveys the following handoff information to the PEI Foundation, State of the platform.

- Location and size of the Boot Firmware Volume (BFV).
- Location and size of the temporary RAM.
- Location and size of the stack.

The format is defined as the `EFI_SEC_PEI_HAND_OFF` structure.

An example of, the temporary memory layout from Nt32Pkg is shown below:

```
|-----------| <---- TemporaryRamBase + TemporaryRamSize
|   Heap    |
|           |
|-----------| <---- StackBase / PeiTemporaryMemoryBase
|           |
|   Stack   |
|-----------| <---- TemporaryRamBase
```

Figure 3 Temporary Memory Layout

The `EFI_SEC_PEI_HAND_OFF` data structure is populated as follows:

```
SecCoreData->DataSize = sizeof (EFI_SEC_PEI_HAND_OFF);
SecCoreData->BootFirmwareVolumeBase = (VOID *)BootFirmwareVolumeBase;
SecCoreData->BootFirmwareVolumeSize = PcdWinNtFirmwareFdSize;
SecCoreData->TemporaryRamBase = (VOID *)(UINTN)LargestRegion;
SecCoreData->TemporaryRamSize = STACK_SIZE;
SecCoreData->StackBase = SecCoreData->TemporaryRamBase;
SecCoreData->StackSize = PeiStackSize;
SecCoreData->PeiTemporaryRamBase = (VOID *)((UINTN)
                                   SecCoreData->TemporaryRamBase + PeiStackSize);
SecCoreData->PeiTemporaryRamSize = STACK_SIZE - PeiStackSize;
```

6.3.2 EFI_PEI_PPI_DESCRIPTOR *PpiList

Besides the `EFI_SEC_PEI_HAND_OFF` data structure, SEC may transfer one additional `PpiList` to the PEIM Foundation. For example, `PpiList` may include `TEMPORARY_RAM_SUPPORT_PPI` and `SEC_PLATFORM_INFORMAITON_PPI`.

- `TEMPORARY_RAM_SUPPORT_PPI`

This service may be published by the SEC as part of the SEC-to-PEI handoff. If so, it moves the Temporary RAM contents into Permanent RAM.

- `SEC_PLATFORM_INFORMAITON_PPI`

This service abstracts platform-specific information. It is necessary to convey this information to the PEI Foundation so that it can locate the PEIM dispatch order. In addition, it contains the maximum stack capabilities of this platform.

7 PRE-EFI INITIALIZATION MODULES

The Pre-EFI Initialization Modules (PEIMs) provide a standards-based platform initialization. The PEI Phase is responsible for initializing enough of the system to provide a stable base for the follow-on phases.

7.1 Introduction

It is strongly recommended that PEIMs perform only the minimum work to meet the requirements of the subsequence phase.

The PEI Foundation establishes the PEI Services Table that is usable by all PEIMs.

The PEI phase allows C-codes PEIMs to be executed prior to the availability of main memory. This is accomplished via configuring the on-CPU resources, such as the CPU data cache to be used as a memory call stack.

7.2 Beginning to Write a PEIM INF File

Following is a sample for `[Defines]` section of one PEIM:

```
[Defines]
  INF_VERSION    = 0x00010005
  BASE_NAME      = SamplePei
  FILE_GUID      = XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
  MODULE_TYPE    = PEIM
  VERSION_STRING = 1.0
  ENTRY_POINT    = PeimSampleInitialize
```

The `MODULE_TYPE` must be set to `PEIM` for all PEIMs.

Note: If PEIM dependent on the new definitions and features in

---

PEI_SERVICES_TABLE-defined in PI specification from versions1.2 forward-the hex version 0x0001000A needs to be given in INF file [Defines] section's PI_SPECFIICATION_VERSION field.

7.3 Defining a PEIM's entry point

In the sample given above, `ENTRY_POINT` is set to `PeimSampleInitialize`. The entry point value is the name of the entry point function.

Following is the prototype of the PEIMs' entry point.

```
EFI_STATUS
EFIAPI
PeimSampleInitialize (
    IN EFI_PEI_FILE_HANDLE FileHandle,
    IN CONST EFI_PEI_SERVICES PeiServices
);
```

Parameters:

FileHandle

The handle of the file being invoked.

PeiServices

An indirect pointer to the PEI Services Table.

If a PEIM is dispatched successfully, `PeimSampleInitialize()` is invoked. The parameters FileHandle and PeiServices may be used in this function.

7.4 Get Pei Services

EDK II provides all Pei Services APIs in the Pei Services Library class. Developers can use the Pei Services Library directly to invoke PEI services.

EDK II provides PEI Services Table Library to obtain the pointer of the Pei Service Table for PEIMs. Aside from obtaining the PEI Services Table pointer from an input parameter in the PEIM entry point, EDK II also allows using the

`GetPeiServicesTablePointer()` defined in Pei Service Table Pointer Library.

7.5 Communicate between PEIM Modules

There are three methods for PEIMs to communicate with each other: PPIs, HOBs and dynamic PCDs.

7.5.1 PPI

PEIM modules can communicate with each other using a structure called a PEIM-toPEIM Interface (PPI). Each PPI has one GUID. The Pei Service Table provides some Pei services to access the PPI database.

In EDK II, one PEIM module can invoke `PeiServicesInstallPpi()` to publish its PPI services into the PPI database by GUID. Another PEIM module can invoke `PeiServicesLocatePpi()` to locate PPI services from the PPI database by GUID.

7.5.1.1 Installing a PPI

For example, if Module A wants to publish one Template PPI service (including three APIs: `Interface2`, `Interface2` and `Interface3`), it can install the Template PPI by using `PeiServicesInstallPpi.` For example:

```
//
// Template PPI
//
EFI_PEI_TEMPLATE_PPI gEfiTemplatePpi = {
  Interface1,
  Interface2,
  Interface3
};
EFI_PEI_PPI_DESCRIPTOR gPpiListTemplatePpi = {
  (EFI_PEI_PPI_DESCRIPTOR_PPI | EFI_PEI_PPI_DESCRIPTOR_TERMINATE_LIST),
  &gEfiTemplateGuid,
  &gEfiTemplatePpi
};
EFI_STATUS
PeimEntryPoint (
  IN EFI_FFS_FILE_HEADER  *FfsHeader,
  IN EFI_PEI_SERVICES     **PeiServices
  )
{
  EFI_STATUS  Status;
  //
  // Publish Template PPI.
  //
  Status = PeiServicesInstallPpi (&gPpiListTemplatePpi);
  return Status;
}
```

7.5.1.2 Locating a PPI

If Module B needs to invoke `Interface2()` provided by Template PPI, it can locate Template PPI by using the following code:

```
//
// Get Template PPI
//
Status = PeiServicesLocatePpi (
           &gEfiTemplateGuid,
           0,
           NULL,
           (VOID **)&TemplatePpi
           );
ASSERT_EFI_ERROR (Status);
//
// Invoke Interface2()
```

```
//
Status = TemplatePpi->Interface2 (...);
ASSERT_EFI_ERROR (Status);
```

7.5.2 HOB

PEIM modules can build a Hand-Off Block (HOB) to pass some information to the DXE Foundation and DXE modules. In addition, other PEIMs can obtain similar information from a HOB by using the HOB services in the Pei Service Table.

In EDK II, the Hob Library provides the generic interfaces to access HOBs for PEIMs and DXE drivers.

7.5.3 PCD

A PEIM can communicate with other PEIMs through dynamic PCDs. As with HOBs, only PEIMs can obtain dynamic PCDs values, which were previously set by DXE drivers. The usage of getting PCDs is introduced in Appendix A, Dynamic PCD.

7.6 Communicate with DXE Modules

There are three methods for PEIMs to communicate with each other: PPIs, HOBs and PCDs.

7.6.1 HOB

PEIMs can pass some information to the DXE Foundation and DXE modules, such as the information of a memory bank discovered in PEI phase, by using a Hand-Off Block (HOB).

In EDK II, the Hob Library provides a set of interfaces to help to build Hobs, such as `BuildGuidHob()`. For example:

```
EFI_MEMINIT_CONFIG_DATA *ConfigData;
EFI_PEI_HOB_POINTERS Hob;
UINTN BufferSize;
BufferSize = sizeof (EFI_MEMINIT_CONFIG_DATA);
Hob.Raw = BuildGuidHob (
            &gEfiMemoryConfigDataGuid,
            BufferSize
            );
ASSERT (Hob.Raw);
ConfigData = (EFI_MEMINIT_CONFIG_DATA *) Hob.Raw;
CopyMem (
  ConfigData->SpdData,
  SpdData,
  sizeof (MEMINIT_SPD_DATA) * MAX_SOCKETS
  );
```

In EDK II, the Hob Library also provides a set of APIs to locate HOBs for PEIMs and DXE drivers.

7.6.2 Variable

PEIMs can read variables previously assigned by DXE drivers. PEIMs cannot write variables.

PEIMs can use ReadOnlyVariable2 PPI to obtain variables. Follow these steps:

1. Locate ReadOnlyVariable2 PPI.
2. Invoke `GetVariable()` with size is 0, to get variable's actual size.
3. Allocate memory for variable.
4. Invoke `GetVariable(),` again with actual size to get the variable.

The following is one example of how to obtain the variable.

```
Status = PeiServicesLocatePpi (
           &gEfiPeiReadOnlyVariable2PpiGuid,
           0,
           NULL,
           (VOID **) &VariablePpi
           );
ASSERT_EFI_ERROR (Status);
Size = 0;
Status = VariablePpi->GetVariable (
                        VariablePpi,
                        VariableName,
                        (EFI_GUID *) VariableGuid,
                        NULL,
                        &Size,
                        NULL
                        );
if (Status == EFI_BUFFER_TOO_SMALL)
```

```
{
  Status = PeiServicesAllocatePool (Size, &Buffer);
  ASSERT_EFI_ERROR (Status);
  Status = VariablePpi->GetVariable (
                          VariablePpi,
                          (UINT16 *) VariableName,
                          (EFI_GUID *) VariableGuid,
                          NULL,
                          &Size,
                          Buffer
                          );
  ASSERT_EFI_ERROR (Status);
  *VariableSize = Size;
  *VariableData = Buffer;
}
```

7.6.3 PCD

PEIMs can communicate with DXE drivers through dynamic PCDs. As with variables, PEIMs can get dynamic PCDs values that were previously set by DXE drivers. The usage for obtaining PCDs is covered in Appendix A .

7.7 Boot Mode

Sometime, PEIMs need to determine the boot mode (e.g. S3, S5, normal boot, diagnostics, etc.) and take appropriate actions depending on it. For example, the VariablePei module will not install EFI ReadOnlyVariable2Ppi in the recovery boot path.

The Pei Service Table provides one pair of services to Set or Get the mode.

Accordingly, the Pei Service Library APIs are: SetBootMode() and GetBootMode().

The following is one example of how to get boot mode:

```
//
// Check if this is recovery boot path. If no, publish the variable
// access capability to other modules. If yes, the content of variable
// area is not reliable. Therefore, in this case we should not provide
// variable service to other pei modules.
//
Status = PeiServicesGetBootMode (&BootMode);
ASSERT_EFI_ERROR (Status);
if (BootMode == BOOT_IN_RECOVERY_MODE)
{
  return EFI_UNSUPPORTED;
}
Status = PeiServicesInstallPpi (&mPpiListVariable);
Note:
The PI Specification lists all possible boot modes.
```

7.8 Execution in Place PEIMs

Most PEIMs are Execution in Place (XIP) and not compressible as they run prior to permanent memory. There is a tradeoff between the space-complexity of the code and the time complexity of the modules: that is, keeping modules small versus keeping the code paths short.

Minimizing the amount and complexity of code in PEIM should be standard procedure. For example, a big loop needs to be avoided for those codes running on flash.

When a PEIM attempts to load itself into system memory and run twice, it can use `RegisterForShadow()` to do it. `RegisterForShadow()` is in the Pei Service Table.

7.9 Dependency for PEIMs

A PEIM must have a dependency section. The PEIM is dispatched after all conditions in the dependency section are met.

If a PEIM has a dependency section TRUE, it can be dispatched immediately. In an extended INF file, a dependency section is contained in the `[Depex]` section. PPI dependency is defined by the GUID of the PPI.

For example:

```
gEfiPeiReadOnlyVariable2PpiGuid AND gEfiPeiCachePpiGuid AND
gPeiCapsulePpiGuid
```

This module may be dispatched only after Read Only Variable2 Ppi, CachePpi and CapsultPpi are all installed successfully.

In the preceding example, the expression opcode `AND` is used to show the logical relationship between GUIDs. See the EDK II Extended INF Specification for complete details.

Note: A PEIM inherits dependency expressions from all library instances it links with. The dependency expression listed in module INF is a subset of the dependency section in the PE32+ image built from this module.

The PI specification also defines a generic rule to decide the dispatch order for PEIMs: the apriori file. It complements the dependency expression mechanism of the PEI Phase by stipulating a series of modules that must be dispatched in a prescribed order. The `[depex]` sections for these modules are ignored.

8 DXE DRIVERS: NON-UEFI DRIVERS

DXE driver refers to drivers compliant with the PI Specification, which classifies DXE drivers into two classes: UEFI driver model driver, and non-UEFI driver model drivers. The focus of this chapter is the non-UEFI drivers.

Non-UEFI Driver Model drivers are executed early in the DXE phase. These drivers are the prerequisites for the DXE Foundation to produce all required services.

The DXE drivers must be designed so that unavailable services are not required. Given this restriction, all possible work should be deferred to the UEFI drivers.

8.1 Beginning with INF File

Each DXE Driver requires an Extended INF file. For basic introduction of INF file, please refer to Section 3.2.2.

The `[Defines]` section for DXE driver should be modeled after the following.

Note: The MODULE_TYPE entry must be DXE_DRIVER.

```
INF_VERSION = 0x00010005
BASE_NAME = SampleDriverDxe
FILE_GUID = XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
MODULE_TYPE = DXE_DRIVER
ENTRY_POINT = SampleDriverEntryPoint
[Depex] gSampleProtocolGuid
```

Note: If DXE module dependent on the new definitions/features in

---

DXE_SERVICES_TABLE-defined in PI specifications from version 1.2 forward- the hex version 0x0001000A needs to be given in INF file [Defines] section's PI_SPECFIICATION_VERSION field.

Note: If module dependent on the new definitions/features in

EFI_BOOT_SERVICES/UEFI_RUNTIME_SERVICES- defined in UEFI specifications from version 2.1 forward-the hex version need to be given in INF file [Defines] section's UEFI_SPECFIICATION_VERSION field.

8.2 Write DXE Driver Entry Point

The `[Defines]` section of the INF file defines the entry point of the DXE driver.

Unlike the UEFI driver entry point, which is only allowed to install protocol instances onto its own image handle and may not touch any hardware, a DXE driver entry point has no such restriction. It may install any protocol into the system and perform necessary hardware and software initializations.

In the following example (from the WatchDogTimerDxe driver in the MdeModulePkg) the DXE driver entry point installs its Architectural Protocol if the protocol is not yet installed.

```
EFI_STATUS
EFIAPI
WatchdogTimerDriverInitialize (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  EFI_STATUS  Status;
  //
  // Make sure the Watchdog Timer Architectural Protocol has not been
  // installed in the system yet.
  //
  ASSERT_PROTOCOL_ALREADY_INSTALLED (
    NULL,
    &gEfiWatchdogTimerArchProtocolGuid
    );
  //
  // Create the timer event to implement a simple watchdog timer
  //
  Status = gBS->CreateEvent (
                EVT_TIMER | EVT_NOTIFY_SIGNAL,
                TPL_NOTIFY,
                WatchdogTimerDriverExpires,
                NULL,
                &mWatchdogTimerEvent
                );
  ASSERT_EFI_ERROR (Status);
```

The two parameters for the DXE driver entry point are `ImageHandle` and `SystemTable`. `ImageHandle` refers to the image handle of the DXE driver.

`SystemTable` points to the EFI System Table.

8.3 Obtaining Services Tables

DXE drivers may consume the UEFI Boot Services, UEFI Runtime Services, and DXE Services. In addition, a DXE driver can also refer to UEFI System Table.

UEFI Boot Services, UEFI Runtime Services, and UEFI System Table are defined in the UEFI Specification. DXE Services are defined in the PI Specification.

DXE driver can retrieve these tables via global variables provided by the following library classes:

Table 11 Reference to Services Tables for DXE Drivers

	`Global variable`	`Library Class`
UEFI System Table	`gST`	UefiBootServicesTableLib
UEFI Boot Services Table	`gBS`	UefiBootServicesTableLib
UEFI Runtime Services Table	`gRT`	UefiRuntimeServicesTableLib
DXE Services Table	`gDS`	DxeServicesTableLib

8.4 Communication between DXE Drivers

This section introduces communication channels between DXE drivers, including protocol, variable, and PCD.

8.4.1 Protocol

This section will introduce how to produce and consume protocols. The UEFI Specification defines a group of boot services to handle protocols, including:

Services to install protocols

- `InstallProtocolInterface()`
- `ReInstallProtocolInterface()`
- `InstallMultipleProtocolInterfaces()`

Services to retrieve protocols

- `LocateProtocol()`
- `OpenProtocol()`

First, to make use of it, the module writer must declare the protocols for the module in the INF and then write code to use the protocols.

The following example demonstrates how a DXE driver produces a protocol:

```
//
// Handle for new protocol instance. Since it's NULL now, its value will
// be assigned by Boot Service InstallMultipleProtocolInterfaces()
//
EFI_HANDLE mNewHandle = NULL;
//
// The Sample Protocol instance produced by this driver
//
EFI_SAMPLE_PROTOCOL mSampleProtocol = {
  SampleProtocolApi
  //
  // More APIs can be added here
  //
};
//
// This is just a NULL function with no parameters. Necessary parameters
// and code can be added.
//
EFI_STATUS
EFIAPI
SampleProtocolApi (
   VOID
  )
{
  return EFI_SUCCESS;
}
EFI_STATUS
EFIAPI
SampleDriverInitialize (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  EFI_STATUS  Status;
  //
  // More initialization can be added here.
  //
```

```
  //
  // Install the Sample Protocol onto a new handle
  //
  Status = gBS->InstallMultipleProtocolInterfaces (
                  &mNewHandle,
                  &gEfiSampleProtocolGuid,
                  &mSampleProtocol,
                  NULL
                  );
  ASSERT_EFI_ERROR (Status);
  return EFI_SUCCESS;
}
```

The following example demonstrates how a DXE driver retrieves a protocol and invokes the API:

```
EFI_STATUS
SampleFunction (
   VOID
  )
{
  EFI_STATUS           Status;
  EFI_SAMPLE_PROTOCOL  *SampleProtocol;
  //
  // Locates the Sample Protocol from system.
  //
  Status = gBS->LocateProtocol (
                  &gEfiSampleProtocolGuid,
                  NULL,
                  (VOID **) &SampleProtocol
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  Status = SampleProtocol->SampleProtocolApi();
  return Status;
}
```

8.4.2 Variable

Variables are defined as key/value pairs that consist of identifying information plus attributes (the key) and arbitrary data (the value). Variables are intended for use as a means to store data that is passed between the EFI environment implemented in the platform and EFI OS loaders and other applications that run in the EFI environment.

A DXE driver can read and write variables via the UEFI Runtime Services `GetVariable()` and `SetVariable()`.

Note: These services are not available at the beginning of the DXE phase . The PI Specification defines two architectural protocols to indicate the readiness of read/write access to variables: **EFIVARIABLE_ARCH_PROTOCOL** and **EFI_VARIABLE_WRITE_ARCH_PROTOCOL._**

DXE drivers that require read-only access or read/write access to volatile environment variables must have `EFI_VARIABLE_ARCH_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.

The full complement of environment variable services is not available until both `EFI_VARIABLE_ARCH_PROTOCOL` and `EFI_VARIABLE_WRITE_ARCH_PROTOCOL` are installed.

Sample code to read and write variables is as follows:

```
EFI_STATUS
ReadAndWriteVariable (
    IN CHAR16 *Name,
    IN EFI_GUID *VendorGuid,
){
    EFI_STATUS Status;
    UINTN BufferSize;
    VOID *Buffer;
    Buffer = NULL;
    //
    // Pass in a zero-size buffer to find the required buffer size.
    //
    BufferSize = 0;
    Status = gRT->GetVariable (
                    Name,
                    VendorGuid,
                    NULL,
                    &BufferSize,
                    Buffer
                    );
    //
    // If variable exists, the Status should be EFI_BUFFER_TOO_SMALL and
    // BufferSize has been updated.
    //
    if (Status != EFI_BUFFER_TOO_SMALL) {
    return Status;
    }
    //
    // Allocate the buffer according to updated BufferSize.
    //
    Buffer = AllocateZeroPool (BufferSize);
    ASSERT (Buffer != NULL);
    //
    // Read variable into the allocated buffer.
    //
    Status = gRT->GetVariable (
                    Name,
                    VendorGuid,
                    NULL,
                    &BufferSize,
                    Buffer
                    );
    if (EFI_ERROR (Status)) {
    BufferSize = 0;
    }
    //
    // TODO: Process of retrieved variable can be added here.
    //
    //
    // Now write back the processed variable.
    //
    Status = gRT->SetVariable (
                    Name,
                    VendorGuid,
                    EFI_VARIABLE_BOOTSERVICE_ACCESS |
                    EFI_VARIABLE_RUNTIME_ACCESS |
                    EFI_VARIABLE_NON_VOLATILE,
                    BufferSize,
                    Buffer
                    );
    ASSERT_EFI_ERROR (Status);
    return EFI_SUCCESS;
}
```

8.4.3 Dynamic PCD

EDK II provides dynamic PCDs as a high-level mechanism for communication between modules. See Appendix A for details.

8.5 Communication with PEIMs

This section introduces communication channels between DXE driver and PEIM, including HOB, variable, and PCD.

8.5.1 HOB

A HOB is a one-way channel to pass data from PEI to DXE. The HOB list is provided during the PEI phase, and must be treated as a read-only data structure in the DXE phase. It conveys the state of the system at the time the DXE Foundation is started. The DXE drivers must not modify the contents of the HOB list.

HobLib provides a set of APIs to build and parse a HOB list. Since DXE drivers only read the HOB list, module writers of DXE drivers can focus on the APIs to parse HOB list.

Several typical usage types are shown in examples below:

8.5.1.1 Traversing all HOBs in the HOB list

```
EFI_HOB_GENERIC_HEADER *Hob;
UINT16 HobType;
UINT16 HobLength;
for (Hob = GetHobList();
     !END_OF_HOB_LIST (Hob); Hob = GET_NEXT_HOB (Hob))
{
  HobType = GET_HOB_TYPE (Hob);
  HobLength = GET_HOB_LENGTH (Hob);
  //
  // Further operation on the HOB can be added
  //
}
```

8.5.1.2 Retrieving only the first HOB of a specific type in the HOB list

```
(CPU HOB type example)
EFI_HOB_CPU *CpuHob;
CpuHob = GetFirstHob (EFI_HOB_TYPE_CPU);
if (CpuHob != NULL)   //
{
  // Operation on the HOB can be added here.
  //
}
```

8.5.1.3 Traversing specific types of HOBs in the HOB list (CPU HOB type example)

```
EFI_HOB_CPU *Hob;
Hob = GetHobList ();
while ((Hob = GetNextHob (EFI_HOB_TYPE_CPU, Hob)) != NULL)   //
{
  // Operation on the HOB can be added here.
  //
  // At the end of loop, GET_NEXT_HOB must be added here.
  // GetNextHob (
       HobType, HobStart) does not skip the HOB passed by
       // parameter HobStart. It returns HobStart back if HobStart itself
       // meets the requirement. So it is required to use GET_NEXT_HOB() to
       // skip current HOB. Otherwise, it would be in dead loop.
       //
       Hob = GET_NEXT_HOB (Hob
```

```
      );
}
```

8.5.1.4 Retrieving only the first GUIDed HOB with a specific GUID in the HOB list

```
EFI_HOB_GENERIC_HEADER *Hob;
VOID *HobData;
UINTN HobDataSize;
Hob = GetFirstGuidHob (&gAbcGuid);
if (Hob != NULL)
{
  HobData = GET_GUID_HOB_DATA (Hob);
  HobDataSize = GET_GUID_HOB_DATA_SIZE (Hob);
  //
  // Operation on the HOB can be added here.
  //
}
```

8.5.1.5 Traversing GUIDed HOBs with a specific GUID in the HOB list

```
EFI_HOB_GENERIC_HEADER *Hob;
VOID *HobData;
UINTN HobDataSize;
Hob = GetHobList ();
while ((Hob = GetNextGuidHob (&gAbcGuid, Hob)) != NULL)
{
  HobData = GET_GUID_HOB_DATA (Hob);
  HobDataSize = GET_GUID_HOB_DATA_SIZE (Hob);
  //
  // Operation on the HOB can be added here.
  //
  // At the end of loop, GET_NEXT_HOB must be added here.
  // GetNextHob (
       HobType, HobStart) does not skip the HOB passed by
       // parameter HobStart. It returns HobStart back if HobStart itself
       // meets the requirement. So it is required to use GET_NEXT_HOB() to
       // skip current HOB. Otherwise, it would be in dead loop.
       //
       Hob = GET_NEXT_HOB (Hob
       );
}
```

8.5.2 Variable

A non-volatile variable can serve as a channel to pass data from DXE to PEI. Because only a DXE driver can write a variable, and PEIM can only read variables, this channel from DXE to PEI is also a one-way channel.

8.5.3 Dynamic PCD

A non-volatile dynamic PCD is also a high-level mechanism for communication between a DXE driver and a PEIM.

Please refer to in Appendix A.

8.6 Dependency Expressions

A dependency expression specifies the protocols that the DXE driver requires to execute. In EDK II, it is specified in the `[Depex]` section of INF file.

Note: The PI Specification also defines an a priori file as an arbitrary way for a firmware volume to specify driver execution order. Dependency expressions for drivers covered by the apriori file are ignored.

Following is an example of a [Depex] section:

[Depex]

gEfiSimpleTextOutProtocolGuid AND gEfiHiiDatabaseProtocolGuid AND gEfiVariableArchProtocolGuid AND gEfiVariableWriteArchProtocolGuid

The example specifies that this driver can be executed only after all the four protocols listed have been installed.

Note: The four protocols in this example are necessary conditions, not sufficient conditions. More dependency requirement smay be inherited. Details follows.

Module writers must pay special attention to two points on dependency expressions.

- A DXE driver inherits dependency expressions from all library instances it links with. The dependency expression listed in the module INF is a subset of the dependency section in the PE32+ image built from this module. Linked library instances are specified in DSC file.

or

- A "non-UEFI driver model" driver's INF must have a dependency section. If TRUE is in INF's dependency section, because of inheritance, the generated dependency expression maybe not the TRUE.

The EDK II build tool would wipe out the dependency section in PE32+ image when it has exactly all architectural protocols.

8.7 Handler for EVT_SIGNAL_EXIT_BOOT_SERVICES

Some DXE drivers need to place their controllers in a quiescent state or perform other controller-specific actions at the time that an operating system is about to take full control of the platform. In this case, the DXE driver should create a signal type event that is notified when `gBS->ExitBootServices()` is called by the EFI OS Loader.

Note: The notification function for this event is not allowed to use the Memory Allocation Services, or call any functions that use the Memory Allocation Services, and should only call functions that are known not to use Memory Allocation Services, because these services modify the current memory map.

The template code for the notification function and event registration is as follows:

```
VOID
EFIAPI
NotifyExitBootServices (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )
{
  //
  // Put driver-specific actions here.
  // No UEFI Memory Service may be used directly or indirectly.
  //
}
EFI_STATUS
EFIAPI
SampleDriverInitialize (
  IN EFI_HANDLE ImageHandle,
  IN EFI_SYSTEM_TABLE *SystemTable
)
{
  EFI_STATUS Status;
  EFI_EVENT ExitBootServicesEvent;
  //
  // TODO: Other initialization of entry point can be added here.
  //
  //
  // Here is just the sample of registration of
  // EVT_SIGNAL_EXIT_BOOT_SERVICES
  //
  Status = gBS->CreateEventEx (
                EVT_NOTIFY_SIGNAL,
                TPL_CALLBACK,
                NotifyExitBootServices,
                NULL, // Parameter Context can be passed here
                &gEfiEventExitBootServicesGuid,
                &ExitBootServicesEvent
                );
  ASSERT_EFI_STATUS (Status);
}
```

8.8 DXE Runtime Driver

A DXE runtime driver executes in both boot services and runtime services environments. This means the services that these modules produce are available before and after `ExitBootServices()` is called, including the time that an operating system is running. If `SetVirtualAddressMap()` is called, then modules of this type are relocated according to virtual address map provided by the operating system.

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.

A DXE runtime driver defines `MODULE_TYPE` as `DXE_RUNTIME_DRIVER` in the INF file. In addition, because the DXE runtime driver encounters `SetVirtualAddressMap()` during its life cycle, it may need to register an event handler for the event `EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE`.

8.8.1 INF File

Following is the example of `[Defines]` section for a driver named

`SampleDriverRuntimeDxe`. For DXE runtime driver, the `MODULE_TYPE` entry should be `DXE_RUNTIME_DRIVER`

```
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME   = SampleDriverRuntimeDxe
  FILE_GUID   = XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
  MODULE_TYPE = DXE_RUNTIME_DRIVER
  ENTRY_POINT = SampleRuntimeDriverEntryPoint

[Depex] gSampleProtocolGuid
```

Note: If module dependent on the new definitions and features in

---

EFI_BOOT_SERVICES/UEFI_RUNTIME_SERVICES-defined in UEFI specifications from version 2.1 forward-the hex version need to be given in INF file [Defines] section's UEFI_SPECFIICATION_VERSION field.

8.8.2 Handler for EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE

DXE runtime drivers may need to be notified when the operating system calls `SetVirtualAddressMap()`. In this case, the DXE runtime driver must create a signal type event that is notified when `SetVirtualAddressMap()` is called by the operating system. This call allows the DXE runtime driver to convert pointers from physical addresses to virtual addresses.

The notification function for this type of event is not allowed to use any of the UEFI Boot Services, UEFI Console Services, or UEFI Protocol Services either directly or indirectly because those services are no longer available when `SetVirtualAddressMap()` is called.

Instead, this type of notification function typically uses `ConvertPointer()` to convert pointers within data structures that are managed by the DXE runtime driver from physical addresses to virtual addresses.

Template code for notification function and event registration is as follows:

```
//
// This is the global pointer which needs converting
//
VOID *gGlobalPointer;
```

```
VOID
EFIAPI
NotifySetVirtualAddressMap (
  IN EFI_EVENT Event,
  IN VOID *Context
)
{
  gRT->ConvertPointer (
         EFI_OPTIONAL_POINTER,
         (VOID **)&gGlobalPointer
         );
}
EFI_STATUS
EFIAPI
SampleRuntimeDriverInitialize (
  IN EFI_HANDLE ImageHandle,
  IN EFI_SYSTEM_TABLE *SystemTable
)
{
  EFI_STATUS Status;
  EFI_EVENT SetVirtualAddressMapEvent;
  //
  // TODO: Other initialization of entry point can be added here.
  //
  //
  // Here is just the sample of registration of
  // EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE
  //
  Status = gBS->CreateEventEx (
                  EVT_NOTIFY_SIGNAL,
                  TPL_CALLBACK,
                  NotifySetVirtualAddressMap,
                  NULL, // Parameter Context can be passed here
                  &gEfiEventVirtualAddressChangeGuid,
                  &SetVirtualAddressMapEvent
                  );
  ASSERT_EFI_STATUS (Status);
}
```

8.9 DXE SAL Driver

The module type of DXE SAL Driver is only available to the IPF architecture. This module type is used by DXE Drivers that can be called in physical mode before `SetVirtualAddressMap()` is called, and either physical mode or virtual mode after `SetVirtualAddressMap()` is called. This means the services that these modules produce are available after `ExitBootServices()`.

A DXE SAL driver defines `MODULE_TYPE` as `DXE_SAL_DRIVER` in the INF file. In addition, a DXE SAL driver registers SAL Services for the system.

Because a DXE SAL Driver is available after `ExitBootServices()`, it may also need to register an event handler for `EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE`.

8.9.1 INF File

Following is the example of a `[Defines]` section for a driver named

SampleDriverDxeSal. For DXE SAL driver, the `MODULE_TYPE` entry should be as follows:

```
  DXE_SAL_DRIVER

[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME   = SampleDriverDxeSal
  FILE_GUID   = XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
  MODULE_TYPE = DXE_SAL_DRIVER
  ENTRY_POINT = SampleDxeSalDriverEntryPoint

[Depex]
  gSampleProtocolGuid
```

8.9.2 Entry Point

The entry point of DXE SAL Driver must register the SAL services it produces. The template code is as follows.

Note: EDK II does not specify a detailed way for DXE SAL Drivers to produce and register SAL services.

```
EFI_STATUS
EFIAPI
SampleDxeSalDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  //
  // More initialization can be added here.
  //
  //
  // Event creation for EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE can be added
  // here.
  //
  //
  // Register SAL services
  //
  return EFI_SUCCESS;
}
```

8.10 DXE SMM Driver

This module type is used by SMM Drivers that are loaded into SMRAM. As a result, this module type is only available for IA-32 and x64 CPUs. These modules are dispatched by SMM Foundation and are never destroyed. This means the services that these modules produce are available after `ExitBootServices()`.

The lifecycle of SMM drivers can be divided into two phases, which have different constraints. `SMM Initialization:`

This is the phase of SMM Driver initialization that starts with the call to the driver's entry point and ends with the return from the driver's entry point.

SMM Runtime:

This is the phase of SMM Driver initialization that starts after the return from the driver's entry point.

8.10.1 INF File

Following is the example of `[Defines]` section for a driver named

SampleDriverDxeSmm. For a SMM driver, the `MODULE_TYPE` is `DXE_SMM_DRIVER`.

```
[Defines]
  INF_VERSION              = 0x00010005
  BASE_NAME                = SampleDriverDxeSmm
  FILE_GUID                = XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
  MODULE_TYPE              = DXE_SMM_DRIVER
  PI_SPECIFICATION_VERSION = 0x0001000A
  ENTRY_POINT              = SampleDxeSmmDriverEntryPoint

[Depex]
  gSampleProtocolGuid
```

Note: PISPECIFICATIONVERSION should be definitely set to 0x0001000A for PI

1.1 compliant SMM drivers.

8.10.2 Constraints

The SMM driver model has constraints similar to those of DXE Runtime Drivers.

Inside of SMM Runtime, the drivers may not be able to use core protocol services.

There are SMST-based services, which the drivers can access, but the UEFI System Table and other protocols installed during boot services may not necessarily be available.

Inside of SMM Initialization, the full collection of UEFI Boot Services, UEFI Runtime Services and SMST-based services are available.

8.10.2.1 SMM Driver Initialization

An SMM Driver's initialization phase begins when the driver is loaded into SMRAM, and its entry point is called. An SMM Driver's initialization phase ends when the entry point returns.

During SMM Driver initialization, SMM Drivers have access to two sets of protocols:

UEFI protocols and SMM protocols.

UEFI protocols are those installed and discovered using UEFI Boot Services. UEFI protocols can be located and used by SMM drivers only during SMM Initialization.

SMM protocols are those installed and discovered using the System Management Services Table (SMST). SMM protocols can be discovered by SMM drivers during initialization phase and SMM runtime phase.

SMM Drivers must not use the UEFI Boot Services `Exit()` and `ExitBootServices()` during SMM Driver Initialization.

8.10.2.2 SMM Driver Runtime

During SMM Driver runtime, SMM drivers only have access to SMST-based services. In addition, depending on the platform architecture, memory areas outside of SMRAM may not be accessible to SMM Drivers. Likewise, memory areas inside of SMRAM may not be accessible to UEFI drivers.

These SMM Driver Runtime characteristics lead to several restrictions regarding the usage of UEFI services:

- UEFI interfaces and services located during SMM Driver Initialization must not be called or referenced during SMM Driver Runtime. This includes the EFI System Table, the UEFI Boot Services, and the UEFI Runtime Services.
- Events created during SMM Driver Initialization must be closed before exiting the driver entry point.

APPENDIX A DYNAMIC PCD

The dynamic type PCD is used for a configuration/setting whose value is to be determined dynamic. In contrast, the value of static type PCD (FeatureFlag, FixedPcd, PatchablePcd) is fixed in the final generated FD image during build time.

The "dynamic" determination means one of three things:

- The PCD setting value is produced and consumed by drivers during execution.
- The PCD setting value is user configurable from setup.
- The PCD setting value is produced by the platform OEM vendor in a specified area.

A.1.1 Class of Dynamic Type

According to module distribution way, dynamic PCD could be classified as:

Dynamic:

If module is released in source code and will be built with platform DSC, the dynamic PCD used by this module can be accessed as:

```
PcdGetxx(PcdSampleDynamicPcd);
```

In building platform, the build tools translate PcdSampleDynamicPcd to the parameters Token Space Guid: Token Number for this PCD.

DynamicEx:

If a module is released as binary and is not included in the platform build, the dynamic PCD used by this module must be accessed as:

PcdGetxxEx(gEfiMyTokenspaceGuid, PcdSampleDynamicPcd)

Note: Developers need to explicitly pass Token SpaceGuid and TokenNumber as the parameters.

According to PCD value's storage method, dynamic PCD may be classified three ways:

Default Storage:

The PCD value is stored in PCD database maintained by PCD driver in boot-time memory.

This type is used for communication between PEIM/DXE drivers and DXE/DXE drivers. All set/get value are lost after boot-time memory is turn off.

```
[PcdsDynamicDefault] is used as the section name for this type of PCD in the platform DSC file.

[PcdsDynamicExDefault] is used for dynamicEx types of PCDs.
  Variable Storage:
```

The PCD value is stored in a variable area. As the default storage type, this type of PCD could be used for PEI/DXE driver communication. Beside that, this type PCD could also be used to store the value associate with a HII setting via variable interface.

In PEI phase, the PCD value can be obtained but not set because the variable area is read only.

```
[PcdsDynamicHii] is used as section name for this type of PCD in the platform DSC file.

[PcdsDynamicExHii] is for the dynamicEx type of PCD.
  OEM specified storage area:
```

The PCD value is stored in an OEM-specified area whose base address is specified by the FixedAtBuild PCD setting PcdVpdBaseAddress.

The area is read only for PEI and DXE phases.

```
[PcdsDynamicVpd] is used as section name for this type PCD in the platform DSC file.

[PcdsDynamicExVpd] is for a dynamicex type of PCD.
```

A.1.2 When and how to use dynamic PCD

Module developers do not care if the PCD is dynamic or static when writing source code/INF. Dynamic PCD and dynamic type are indicated by the platform integrator in the platform DSC file.