OS Boot with UEFI

System Software on ARM®

Document number: ARM DEN 0008A

OS Boot with UEFI
System Software on ARM

Release information

The following table lists the changes made to this document.

Change history

Date	Issue	Change
14 April 2011	A	First release

Proprietary notice

Confidentiality status

ARM web address

http://www.arm.com

Table of Contents

1 Introduction

This document follows on from the Board Boot Platform Design Document to describe the UEFI boot sequence up to the point the OS kernel is started. It describes the UEFI EDK2 architecture in an ARM environment, differentiating between the common tianocore packages, generic UEFI packages and those specific to ARM platforms. The document also lists those packages that are mandatory/optional for integration into an ARM platform and contains porting tips for some of the more complex ones. The created/modified packages for ARM reference platforms can be used as case studies for further UEFI development.

The document describes the environment and tools required to build and debug UEFI images, including commonly encountered problems (and how to fix them).

1.1 Additional reading

This section lists publications by ARM and by third parties.

See Infocenter, `http://infocenter.arm.com`., for access to ARM documentation.

1.1.1 ARM publications

The following documents contain information relevant to this document:

[1.] Board Boot (ARM DEN 0006)

[2.] Linux Kernel Flattened Device Tree (ARM DEN 0004)

1.1.2 Other publications

The following documents list relevant documents published by third parties:

[3.] EDK II project, `http://sourceforge.net/projects/edk2`

[4.] Platform Initialization Specification – version 1.2, `www.uefi.org/specs`

[5.] ARM Linux Kernel Boot Requirements `http://www.arm.linux.org.uk/developer/booting.php`

[6.] Unified Extensible Firmware Interface Specification – version 2.3, `www.uefi.org/specs`

2 UEFI Overview

The Unified EFI Forum, a non-profit collaborative trade defines the *Unified Extensible Firmware Interface* (UEFI) and *Platform Initialization* (PI) specifications. ARM became a UEFI member in 2008, to support *Original Equipment Manufacturers* (OEMs) that are developing ARM processor-based solutions.

The aim of UEFI is to standardize the hardware platform bring-up process, from power on to when the Operating System boots (Figure 1). The firmware also handles any other boot modes such as booting from a low-power state.

The UEFI boot sequence consists of four phases:

- The Security (SEC) phase initializes the CPU and its coprocessors. This phase ensures the chain of trust from the first instruction of the Reset Vector located into the ROM to the transition from Secure world to Normal world. If a Secure OS is present then this phase initializes it.

  The Board Boot Platform Design Document [1.] provides a more detailed picture about the design of this initial phase.
- The Pre-EFI Initialization (PEI) phase initializes the platform drivers required by the DXE core and migrates the stack and heap from temporary memory (SRAM) to permanent memory (DRAM).
- The Driver Execution Environment (DXE) phase identifies and dispatches the available hardware and services drivers.
- The Boot Device Select (BDS) phase, the last UEFI phase before booting an Operating System starts the chain of drivers required to load and start its kernel. This stage also gives the ability to launch a shell for user interactions.

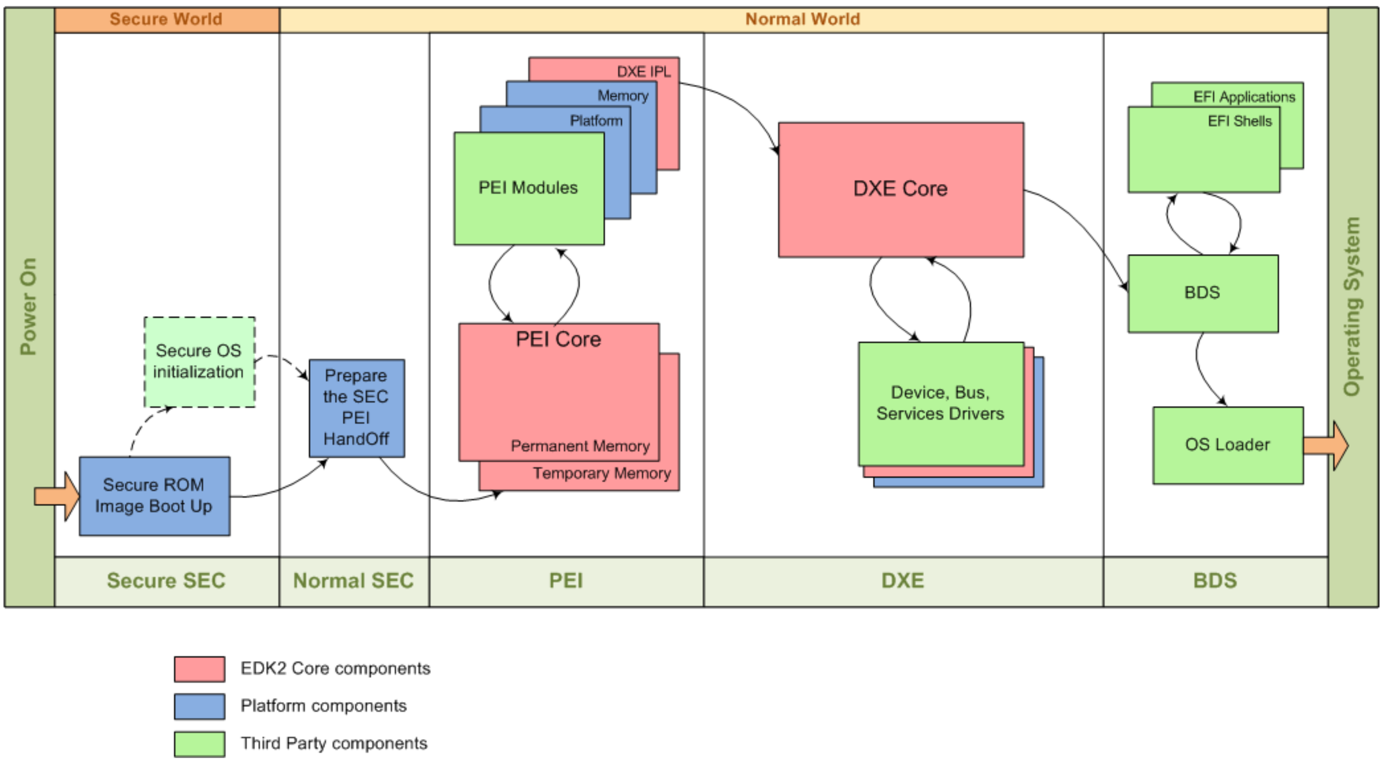

Figure 1 UEFI boot sequence

2.1 UEFI Boot Sequence

Each phase of the UEFI boot sequence enables its own set of hardware devices and platform features. Devices required by the boot firmware progressively enable the memory map.

Figure 2 is an example of a system that has its UEFI firmware located in a XIP ROM. This platform also includes a PCI bus that is initialized during the DXE phase.

Some platforms have firmware in Secure ROM responsible for loading the UEFI firmware from NAND memory to RAM memory. Prior to loading the NAND image, this initial firmware checks the integrity of available updated firmwares. This strategy is generally used by platforms providing secure features. The Board Boot Platform Design Document covers this design more in details.

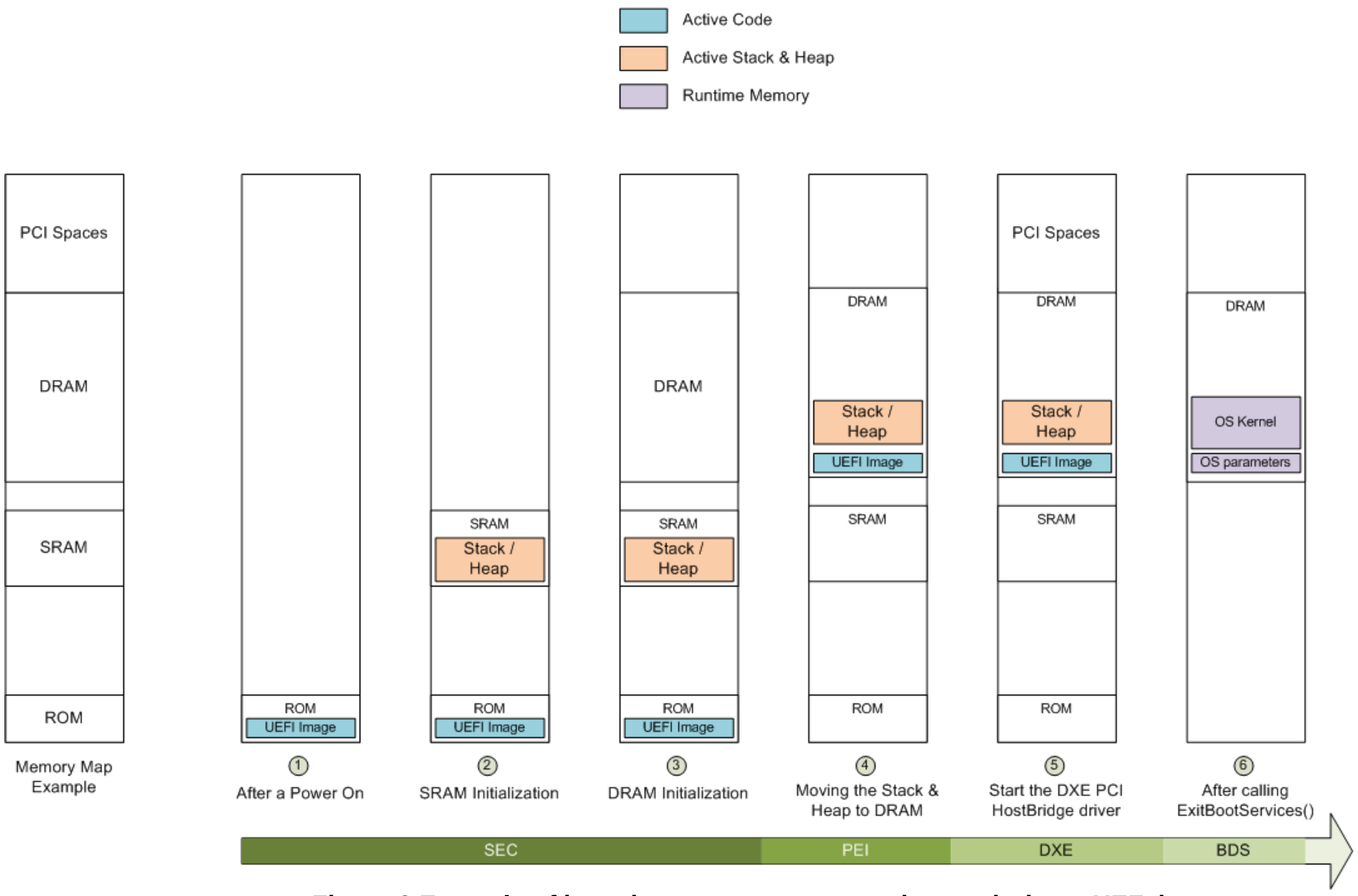

Figure 2 Example of how the memory map can change during a UEFI boot

1) From a cold boot, only the ROM is accessible to the processor. The first executed instruction is the first entry of the Reset Vector located at the address `0x0000 0000`.

   A UEFI firmware can contain a Reset Vector at the beginning of its own code to be directly started from a cold boot. See Figure 4.

2) The early stage of the SEC phase is written in assembler. After the UEFI SEC phase initializes the initial memory and sets up the stack, the firmware can switch from assembler to a more portable language, such as the C programming language.

3) Some platforms require the initialization of the DRAM memory controller and other critical hardware devices in the Secure world.

4) Once all the secure initialization is complete, the firmware jumps to the Normal world and continues the boot sequence in the Non-Secure world. The Secure world would probably not be switched back by the time the Rich OS is started up.

   If the SEC phase sets its stack in temporary memory such as SRAM, the PEI phase would normally move the stack from this temporary location to DRAM memory. The firmware code would normally be copied to and then executed from the permanent memory for performance reasons.

5) After the platform drivers are set up, the DXE core discovers the available drivers from the different UEFI firmware volumes. These firmware volumes are either part of the ROM image or located on alternative devices (e.g.: a file system partition).

   These additional drivers provide further hardware support for booting an OS from various locations or adding services to enable standard or non-standard protocols.

6) Before booting the OS, the UEFI `ExitBootServices()` function stops device drivers that require to be shut down prior to any OS kernel initialization (e.g.: interrupt controller) and discards all the non-runtime memory formerly allocated by UEFI.

   The Secure OS (or trusted OS) is not part of the UEFI software stack. This component is generally provided by a third-party company. The UEFI firmware does not have any dependencies on the Secure OS used by the platform. The trusted OS provides secure services to the Non-Secure/Normal world software stack. It is not a mandatory service. Its initialization is generally done while the platform is still in the Secure world to ensure the chain of trust.

2.2 UEFI Implementation

ARM uses the Open Source UEFI EDK2 [3.] implementation. This implementation provides UEFI core components under the BSD license. These components are portable to various architectures. In addition to the UEFI core components, this environment provides a development environment to generate the final firmware.

Table 1 lists some EDK2 core components and utilities from BaseTools directory that are used to generate the UEFI firmware. EDK2 defines module and firmware properties into its own configuration file format, see Table 2.

Table 1: Overview of the EDK2 components

EDK2 path or file format	Description
\MdeModulePkg\Core\Pei	PEI core sources
\MdeModulePkg\Core\Dxe	DXE core sources
\MdeModulePkg\Application\HelloWorld	HelloWorld EFI Application sources
\BaseTools	Contain the EDK2 specific tools
\BaseTools\Source\C\GenFw	Convert host executable formats (e.g.: ELF binary format) into UEFI compatible executable format derived from the PE/COFF format)
\BaseTools\Source\C\GenFv	Generate the final firmware image.
\Nt32Pkg	UEFI emulator running under Microsoft Windows
\UnixPkg	UEFI emulator running under UNIX
\ArmPkg	Provides ARM architectural protocols common to most of ARM Platforms
\ArmPlatformPkg	Supports UEFI for ARM development boards and their drivers
\ArmPlatformPkg\ArmVExpressPkg	Supports UEFI on ARM Versatile Express
\Conf\tools_def.txt	Defines the supported toolchains. This file is generated from \BaseTools\Conf\tools_def.template

Table 2: Overview of the EDK2 configuration files

EDK2 Configuration File	Description
DEC file	Defines the include directories and Platform Configuration Database (PCD) for a specific package
DSC file	Specifies the libraries, PCD values and components that are required to build a package
FDF file	Describes the content and layout of a firmware device
INF file	Defines a UEFI component (library, driver, application)

2.3 EDK2 Build Process Flow

The EDK2 Build System starts by generating makefiles and additional source files from the EDK2 Platform configuration files. The different components listed in the DSC file are compiled and linked to generate EFI binary files. When all the modules built, EDK2 firmware tools generate a firmware file ready to be written into the platform memory.

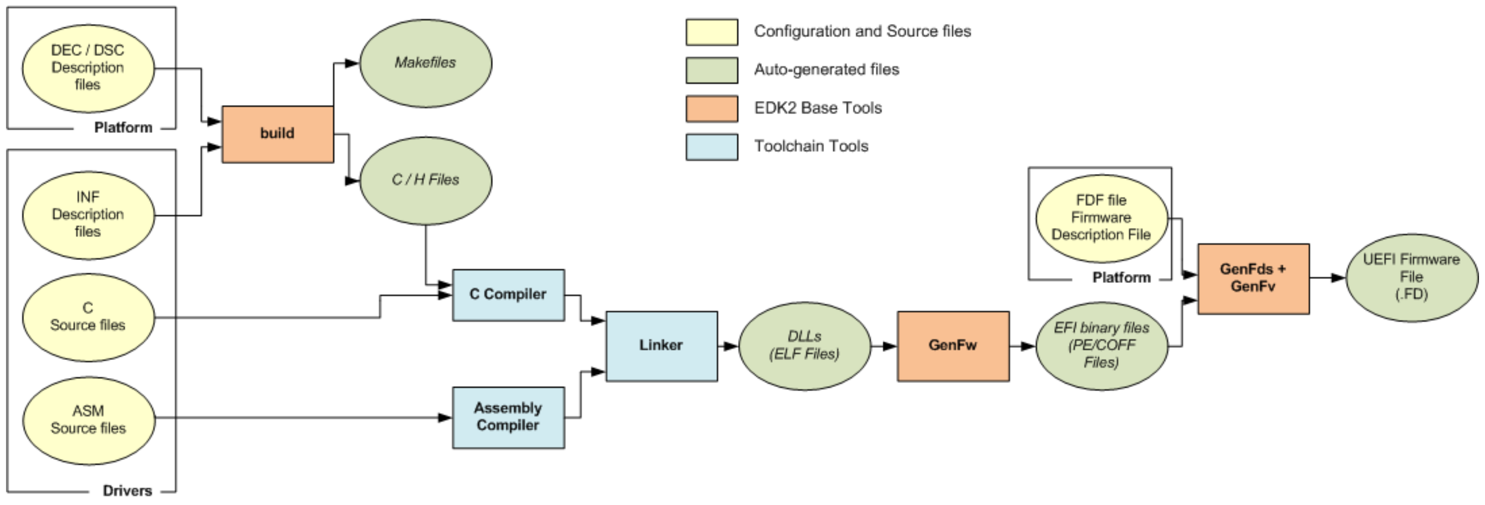

Figure 3 EDK2 Build Flow

2.4 UEFI Protocol

One of the key concepts of UEFI is the protocol. The UEFI Protocol is a software interface. It is implemented by a C-structure and identified by a Globally Unique Identifier (GUID). The most important protocols are defined by the UEFI Specification [6.]

Device and Service drivers implement protocol interfaces and register their implementation to the core modules (PEI or DXE cores). The benefit of using protocols is UEFI components are not tied to a specific implementation. A UEFI component may express its dependencies toward external protocols by specifying their protocol GUIDs in its component configuration file (the [Depex] section of the INF file).

2.5 Architectural Protocols

A UEFI firmware includes various device and services drivers. But the presence of some of those drivers is mandatory for the foundation of the DXE core. These architectural drivers are required to produce some of the key UEFI services such as the UEFI Boot Services, the UEFI Runtime Services, and the DXE Services.

The Platform Initialization Specification defines the Architectural Protocols that are the interfaces for these drivers.

These protocols may be implemented by Pre-EFI Initialization Modules (PEIM) (during the PEI phase) or by DXE drivers. Most of the drivers producing an architectural protocol must only be consumed by either the DXE core or drivers that produce architecture protocols.

Table 3: List of the Architectural Protocols defined by Platform Initialization Specification[4]

GUID	Protocol Description
gEfiSecurityArchProtocolGuid	Abstracts the Security-Specific functions
gEfiCpuArchProtocolGuid	Abstracts some platforms services such as interrupt management and memory attribute settings.
gEfiMetronomeArchProtocolGuid	Provides time source with a predefined period.
gEfiTimerArchProtocolGuid	Sets up a periodic timer interrupt. This protocol is required to enable the Event Timer.
gEfiBdsArchProtocolGuid	Responsible for the BDS phase execution. Its only function is called by the DXE core when it finishes to dispatch its drivers.
gEfiWatchdogTimerArchProtocolGuid	Supports the platform watchdog
gEfiRuntimeArchProtocolGuid	Supports the Runtime functionality.
gEfiVariableArchProtocolGuid	Gets and Sets Environment Variables
gEfiVariableWriteArchProtocolGuid	Require to set non-volatile environment variables.
gEfiCapsuleArchProtocolGuid	Provides the services for capsule update.
gEfiMonotonicCounterArchProtocolGuid	Accesses the platform’s monotonic counter
gEfiResetArchProtocolGuid	Provides the service required to reset a platform
gEfiRealTimeClockArchProtocolGuid	Accesses the platform’s real time clock hardware

2.6 Common ARM Platform Components

To reduce code duplication and ease the UEFI port to new ARM platforms, ARM has introduced Common ARM Platform Components. There are a set of components that are used at the different phases of the UEFI boot sequence.

In addition to speed up the UEFI support on new platforms, these components also enable features inherent to these modules. Multi-Core and TrustZone® support are part of these features.

All these components and features have been tested on real hardware and ARM Models.

The common ARM Platform Components are:

- `ArmPlatformPkg/Sec`: implements the SEC phase
- `ArmPlatformPkg/PrePeiCore`: ensures the transition between the Secure World and the PEI Core
- `ArmPlatformPkg/Pei/PlatformPeim.inf`: declares the Hand-Off Blocks describing the ARM Platform to PEI Core
- `ArmPlatformPkg/Pei/MemoryInitPeim.inf`: initializes the Virtual Memory and declare the System Memory to PEI Core
- ArmPkg/Library/BdsLib: provides helper functions to implement a custom BDS and boot an Operating System.

All these components are described in detail in this document. Section 8. - *Porting UEFI to a new ARM Platform* - explains the porting process using these common components.#

3 ARM Development Environment

This section covers the ARM development environment in EDK2 such as the supported and provided tools and libraries as well as the way to build and debug a UEFI firmware on ARM platform.

3.1 ARM Toolchains

EDK2 currently supports the following ARM toolchains:

- ARM Realview toolchain under Microsoft Windows (toolchain name: RVCT)
- ARM Realview toolchain under Cygwin (toolchain name: RVCTCYGWIN)
- ARM Realview toolchain under GNU/Linux (toolchain name: RVCTLINUX)
- GNU toolchain under GNU/Linux (toolchain name: ARMGCC)

3.2 Build an UEFI ARM Firmware

For example, to build the EDK2 sources to produce the UEFI firmware for the ARM Versatile Express platform with Core Tile Cortex-A9x4 (the sources for this platform are in the `ArmPlatformPkg/ArmVExpressPkg` folder):

... On Linux:

```
cd [EDK2_ROOT]
export EDK_TOOLS_PATH=`pwd`/BaseTools
. edksetup.sh `pwd`/BaseTools/
make -C $EDK_TOOLS_PATH        # Build EDK2 BaseTools
build -a ARM -p ArmPlatformPkg/ArmVExpressPkg/ArmVExpress-CTA9x4.dsc -
t RVCTLINUX
```

... On Windows:

```
cd [EDK2_ROOT]
edksetup.bat
build -a ARM -p ArmPlatformPkg/ArmVExpressPkg/ArmVExpress-CTA9x4.dsc -
t RVCT
```

Note: Prebuilt Windows EDK2 Basetools are available from `[EDK2_ROOT]\BaseTools\Bin\Win32`

To clean the EDK2 source tree, call:

```
build -a ARM -p ArmPlatformPkg/ArmVExpressPkg/ArmVExpress-CTA9x4.dsc -
t RVCT clean
```

In some case, it is convenient to have conditional builds. This could be the case of a development platform where hardware modules could be added to the motherboard (e.g.: ARM RealView Versatile Express board with a Core Tile). EDK2 configuration files (DSC, FDF, INF files) support macro conditions. The condition is declared as follows:

```
!if $(EDK2_PCI_LOGIC_TILE) == 1
    # PCI specific configuration
!endif
```

You can use the `–D` option to enable the macro, for example:

```
build -a ARM -p ArmPlatformPkg/ArmVExpressPkg/ArmVExpress-CTA9x4.dsc -
t RVCTCYGWIN -D EDK2_PCI_LOGIC_TILE=1
```

There are different ways to redefine the toolchain build flags.

- Either you edit the `[EDK2_ROOT]\BaseTools\Conf\tools_def.template`:

```
*_RVCT_ARM_CCPATH_FLAG         = ENV(RVCT_TOOLS_PATH)armcc
*_RVCT_ARM_ASMPATH_FLAG        = ENV(RVCT_TOOLS_PATH)armasm
*_RVCT_ARM_PLATFORM_FLAGS  =

*_RVCT_ARM_ARCHCC_FLAGS    = --thumb --cpu 7-A
*_RVCT_ARM_ARCHASM_FLAGS   = --cpu 7-A

*_RVCT_ARM_ASM_FLAGS       = "$(ASMPATH_FLAG)" $(ARCHASM_FLAGS) --apcs
/interwork
DEBUG_RVCT_ARM_CC_FLAGS  = "$(CCPATH_FLAG)" $(ARCHCC_FLAGS)
$(PLATFORM_FLAGS) --c90 -c -g -O2 --no_autoinline --asm --gnu --apcs
/interwork --signed_chars --no_unaligned_access --split_sections --
preinclude AutoGen.h --diag_warning 167
RELEASE_RVCT_ARM_CC_FLAGS  = "$(CCPATH_FLAG)" $(ARCHCC_FLAGS)
$(PLATFORM_FLAGS) --c90 -c --no_autoinline --asm --gnu --apcs
/interwork --signed_chars --no_unaligned_access --split_sections --
preinclude AutoGen.h --diag_warning 167
```

- Or you add/edit the section `[BuildOptions]` in the DSC file for platform specific flags or in the INF file for module specific flags:

```
 [BuildOptions]
  RVCT:*_*_ARM_ARCHCC_FLAGS  == --cpu Cortex-A9 --thumb
  RVCT:*_*_ARM_ARCHASM_FLAGS == --cpu Cortex-A9
  RVCT:RELEASE_*_*_CC_FLAGS  = -DMDEPKG_NDEBUG
```

You can also redefine the way the toolchain is invoked by EDK2 by editing `BaseTools\Conf\ build_rule.template`.

Note: From the above example of build flags from `tools_def.template`, we can notify C files are compiled in Thumb mode (in fact in Thumb-2 as it is compiled for ARMv7 architecture) and ASM files can mix ARM and Thumb codes (--apcs /interwork).

EDK2 core components do not do any unaligned accesses. ARM recommends that for ARMv7 firmware development the boot firmware disables support for unaligned accesses, at least during the debugging phase. This ensures that the code is portable.

3.3 ARM Firmware File

Later in the build flow (see Figure 3), a *Firmware Device* (FD) file is produced by the `GenFv` BaseTool. This tool adds an ARM Reset Vector at the head of the FD file. The first entry of this vector is a branch to the SEC entry point computed by GenFv. The second entry contains the address of the PEI Core that also computed by the tool. The reason of this PEI address in the Reset Vector is to avoid adding functions in the SEC core to scan the FD file for searching the PEI entry point.

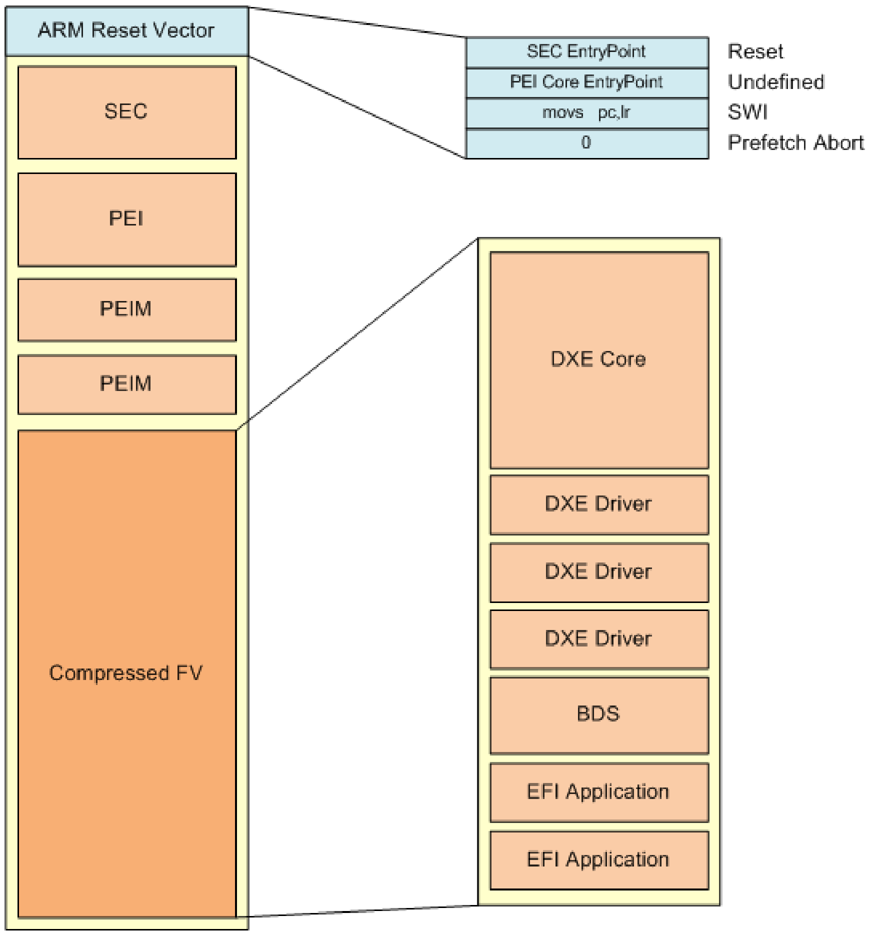

Figure 4 Example of a Firmware Device file

A *Firmware Volume* (FV) as part of a FD file may contain additional compressed FV files. Compressed FV file are used for size or performance reasons. Figure 4 shows the case where the DXE and BDS components (DXE core, DXE drivers, BDS and EFI Applications) are compressed into a single FV file. In this situation the root FV file needs to embed a library to decompress the FV at runtime. A standard or custom decompression library may be used.

3.4 ARM UEFI Package

EDK2 is provided with various packages. Each package represents a core package (EDK2 common components) or an architecture package (architecture specific components – used by all hardware platforms using this architecture) or a platform package (platform specific components). One of these packages is named ‘ArmPkg’. This architecture package contains various libraries and drivers to make easier porting UEFI on ARM platform. Some ARM PrimeCell drivers are available under ArmPkg\Drivers (e.g.: PL34xDmc, PL35xSmc, PL310L2Cache, PL390Gic) as well as some helper libraries such as ArmV7Lib (for setting up MMU and ARMv7 CPU) and memory libraries.

3.5 ARM Debugging Environment

To debug the early stage of UEFI, a hardware debugger is required (e.g.: ARM RealView ICE debugger). The hardware debugger is generally coupled to a debugging User Interface on the host machine (e.g.: ARM RealView Debugger).

The location in memory of the UEFI binary is not known in advance. Except the SEC and PEI cores that run from *eXecute In Place* (XIP) memory, the other UEFI binaries are dynamically loaded in memory. Unfortunately, the load address of the debugged module is required to load the symbols at the correct location.

As a solution to this limitation, the EDK2 PE/COFF loader calls the `PeCoffExtraActionLib` library after loading new binaries. The load address of binary is passed to this library which may make extra actions.

The libraries producing this interface (defined by `PeCoffExtraActionLib`.h) are:

- `MdePkg/Library/BasePeCoffExtraActionLibNull/BasePeCoffExtraActionLibNull.inf`: This library does not make any extra actions
- `ArmPkg/Library/DebugPeCoffExtraActionLib/DebugPeCoffExtraActionLib.inf`: It returns a command line to copy and paste into the debugger in the serial terminal. This command line loads the symbols of the newly loaded EFI binary.
- `ArmPkg/Library/RvdPeCoffExtraActionLib/RvdPeCoffExtraActionLib.inf`: Do the same as DebugPeCoffExtraActionLib except the command line is returned into the RealView Debugger output window (much slower than printing in the serial terminal).

Unfortunately it is generally too late to halt the execution with the debugger when the targeted binary is loaded, the fault has already occurred. It is likely that the load address is consistent between multiple run of a same FD file. It is just needed to launch the same image again to debug the faulty binary.

For the same reason as it is not known in advance where a binary will be loaded into the system memory, you cannot use software breakpoints to stop the application because the EFI binary is not loaded in memory. To debug an EFI binary you must use hardware instruction breakpoints.

Example of debugging a faulty DXE driver with a Windows host machine:

1) Enable one of the `PeCoffExtraActionLib` library which returns the command line for the debugger in the DSC file
2) Build the FD file
3) Reproduce the fault on the target platform
4) Catch the debug command line of the suspected faulty binary. Example:

```
load /a /ni /np
c:\dev\edk2\Build\ArmEB\DEBUG_RVCT\ARM\EmbeddedPkg\MetronomeDxe\Metron
omeDxe\DEBUG\MetronomeDxe.dll &0x07CBF240
```

5) Connect the debugger.
6) Copy the above line into your debugger
7) Create a Hardware Breakpoint into the suspected function or the entry point of the DXE driver
8) Restart the platform

   Once the targeted driver is loaded and the breakpoint is caught, the debugger halts.
9) Step through the code to find the error.

3.6 ARM ELF binary to EFI binary

The ARM toolchains (ARM RealView and GNU Toolchains) produce ELF binary files but UEFI only supports PE/COFF binaries. During the build process, binaries are going through GenFw EDK2 Base Tool that converts host binary formats into UEFI PE/COFF compatible format (See Figure 3).

GenFw tool constructs a PE/COFF file with its specific headers. Then extracts the code and data sections from the original ELF file, apply the ELF relocations defined by the initial file to these sections (see Figure 5).

The tool also creates new sections such as the debug and relocation sections compatible with the EFI file format.

It is also important to note that ARM RealView Debugger uses the original ELF binary to step through the code as the tool does not support PE/COFF format either.

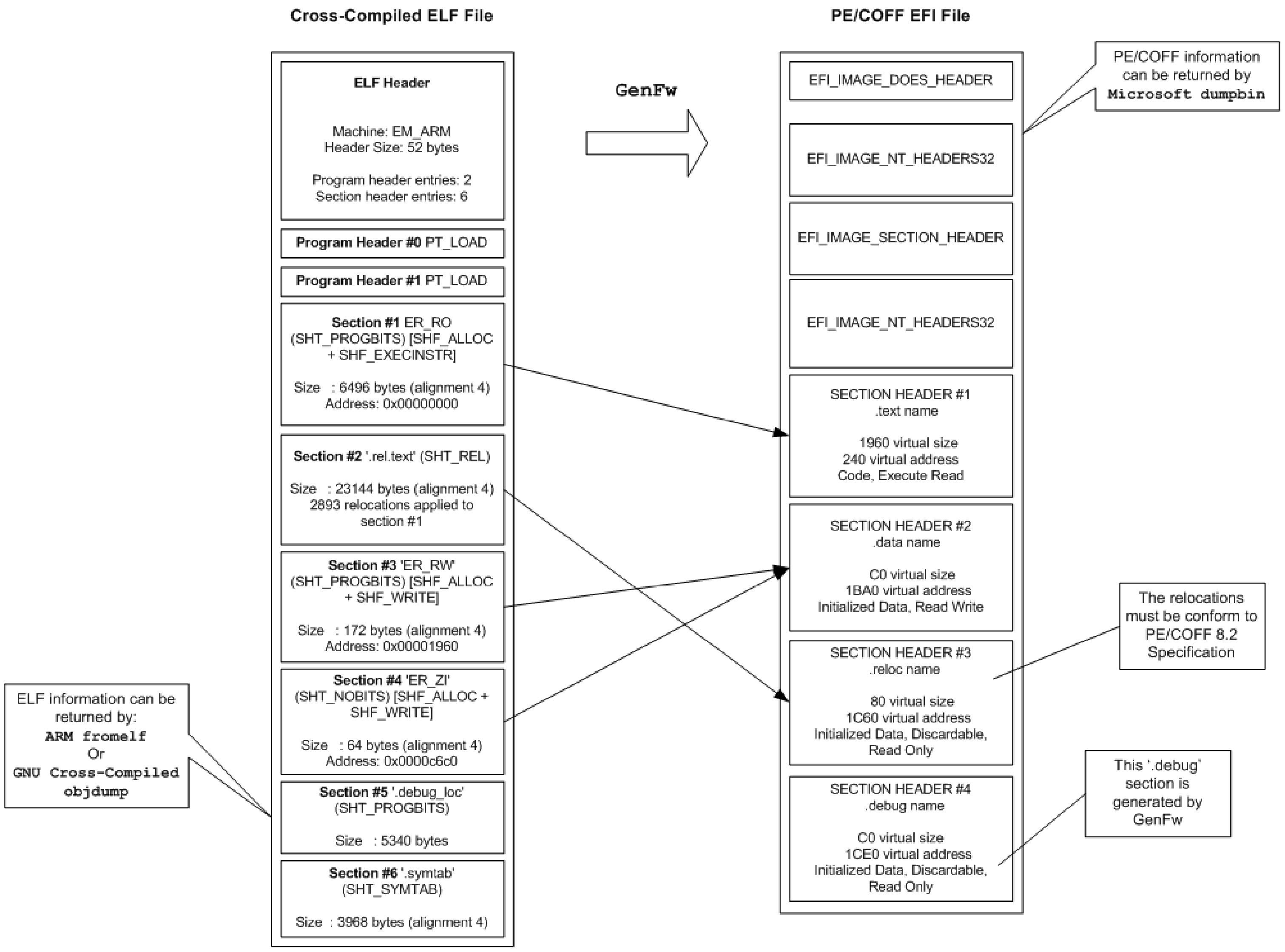
Cross-Compiled ELF File
PE/COFF EFI File
GenFw
ELF Header
Machine: EM_ARM
Header Size: 52 bytes
Program header entries: 2
Section header entries: 6
Program Header #0 PT_LOAD
Program Header #1 PT_LOAD
Section #1 ER_RO (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
Size : 6496 bytes (alignment 4)
Address: 0x00000000
Section #2 '.rel.text' (SHT_REL)
Size : 23144 bytes (alignment 4)
2893 relocations applied to section #1
Section #3 'ER_RW' (SHT_PROGBITS) [SHF_ALLOC + SHF_WRITE]
Size : 172 bytes (alignment 4)
Address: 0x00001960
Section #4 'ER_ZI' (SHT_NOBITS) [SHF_ALLOC + SHF_WRITE]
Size : 64 bytes (alignment 4)
Address: 0x0000c6c0
Section #5 '.debug_loc' (SHT_PROGBITS)
Size : 5340 bytes
Section #6 '.symtab' (SHT_SYMTAB)
Size : 3968 bytes (alignment 4)
EFI_IMAGE_DOES_HEADER
EFI_IMAGE_NT_HEADERS32
EFI_IMAGE_SECTION_HEADER
EFI_IMAGE_NT_HEADERS32
SECTION HEADER #1
.text name
1960 virtual size
240 virtual address
Code, Execute Read
SECTION HEADER #2
.data name
C0 virtual size
1BA0 virtual address
Initialized Data, Read Write
SECTION HEADER #3
.reloc name
80 virtual size
1C60 virtual address
Initialized Data, Discardable, Read Only
SECTION HEADER #4
.debug name
C0 virtual size
1CE0 virtual address
Initialized Data, Discardable, Read Only
PE/COFF information can be returned by Microsoft dumpbin
The relocations must be conform to PE/COFF 8.2 Specification
This '.debug' section is generated by GenFw
ELF information can be returned by: ARM fromelf Or GNU Cross-Compiled objdump

Figure 5 ELF format to PE/COFF conversion

The load address returned by PeCoffExtraActionLib (see ARM Debugging Environment) is the load address for the ELF file. The load addresses differ from both formats as their file headers have different size.

4 Board Boot (SEC Phase)

This section limits its scope to the generic UEFI SEC responsibilities and the initialization of an ARM processor.

Note: The Board Boot Platform Design Document provides a more detailed picture about the design of this initial phase on ARM Reference Hardware.

The SEC phase is the first stage of the UEFI Boot Sequence, see Figure 1. The UEFI SEC core is generally called from the Reset Vector after a power on. This module initializes the processor, its coprocessors, and the platform security.

If the platform does not have an ARM processor with the Security/TrustZone Extension or this extension is not used by the OEM then the SEC phase still initializes the processor and coprocessors.

ARM processors with the TrustZone extension always start in the Secure world. If the security extension is not used then the platform can stay in Secure world.

The security extensions provide an additional bus bit (NS) to control access to secure on-chip hardware and a full set of secure page tables. Any access to secure addresses by the CPU in non-secure state will cause an abort exception.

If the ARM platform supports TrustZone controllers then the TrustZone Protection Controller and the TrustZone Address Space Controller must be configured during the SEC phase, to enable Non-Secure accesses.

ARM MPCore is another feature which needs to be supported in the SEC phase. Secondary cores need to be initialized and parked in a state which fit the Operating System requirements. Figure 8 shows the example of the initialization of an MPCore system and the steps expressly done by the primary and secondary cores.

UEFI ARM Package 'ArmPkg' provides helper libraries to initialize the processor, set up the Security extension, and initialize the architectural devices, such as the *Generic Interrupt Controller* (GIC) and memory controllers. Helper libraries are available for ARM9, ARM11 and ARMv7 processors.

The PEI core entry point requires two arguments from the SEC phase:

- The `EFI_SEC_PEI_HAND_OFF` structure (see Figure 6**.**) that defines the location of the firmware volume, the temporary memory and the stack.
- A list of PEIM-to-PEIM Interfaces (PPIs). This list must at least contain the Temporary RAM Support PPI (`gEfiTemporaryRamSupportPpiGuid`). This interface is required by the PEI phase to move the temporary memory to its permanent location in system memory.

  Other PPIs can complete the list. Sometimes the status code PPI (`gEfiPeiStatusCodePpiGuid`) is part of the list as it may be used at the early stage of the boot sequence to return information about the status of the initialization.

Table 4 EFI_SEC_PEI_HAND_OFF structure

Field	Description
`BootFirmwareVolumeBase / BootFirmwareVolumeSize`	Firmware that contains the PEI modules
`TemporaryRamBase / TemporaryRamSize`	Temporary RAM region (generally SRAM). This region is used before the permanent memory (DRAM) is initialized
`PeiTemporaryRamBase / PeiTemporaryRamSize`	Region part of the Temporary RAM available to the PEI Foundation
`StackBase / StackSize`	Stack region

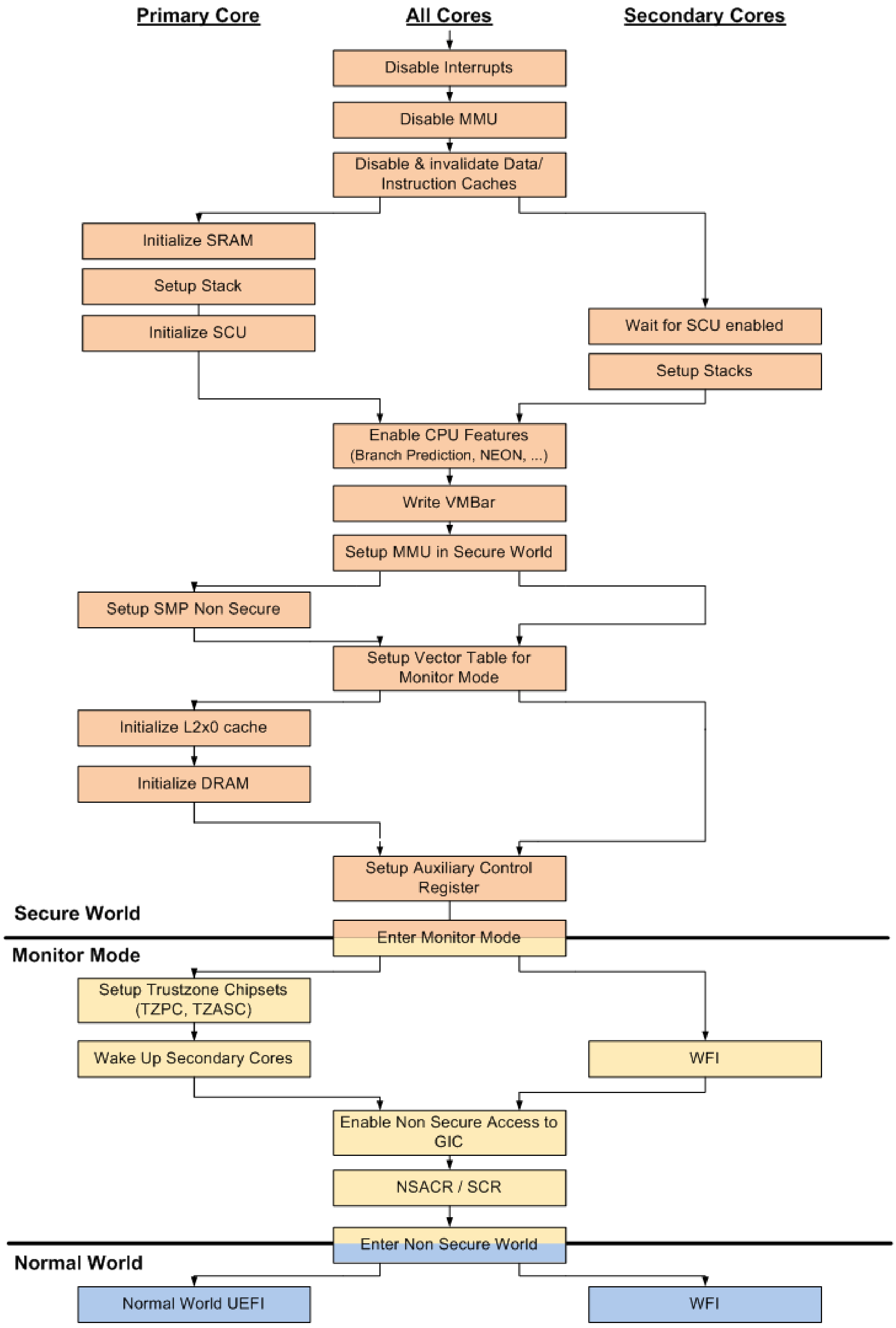

Figure 6: MPCore initialization with the Secure Extension during the SEC phase

4.1 ARM Platform Sec and PrePeiCore

These are the two first common ARM Platform Components of the UEFI Firmware executed by the application processor.

- **ArmPlatformPkg/Sec:** The SEC module covers all the sequence from the Reset Vector to the transition to Normal world. It will ensure all the CPU cores are properly initialized.

  In case Trustzone is not supported by the OEM, the SEC does not setup the Monitor Mode and does not make the transition to Normal world.

- **ArmPlatformPkg/PrePeiCore:** The PrePeicore runs in Normal world (or in Secure world if Trustzone is not supported). The module is responsible to park the secondary cores in Wait For Interrupt (WFI) and make them ready to be woken up by the Operating System. The primary core prepares the PeiCore arguments before calling the module.

  A Vector table is also installed to catch any unexpected event before the CPU DXE driver installs its table.

5 Platform Foundation for EFI (PEI Phase)

The two main purposes of this phase are to move the temporary memory to a permanent location and declare the architectural protocols and *Hand-Off Blocks* (HOBs) required by the DXE phase. The PEI core module leads the execution of this phase by dispatching the PEI modules contained in the registered locations (e.g.: Firmware Volumes, File Systems).

The transition from temporary to permanent memory is initialized when the platform specific PEI Module declares its system memory through the `PeiServicesInstallPeiMemory()` function of the PEI Services library.

The PEI phase must provide the DXE phase with some hardware specific HOBs, to enable the DXE core to control the hardware platform. These HOBs are:

- The `EFI_HOB_TYPE_CPU` which defines the size of the memory and IO spaces
- At least one `EFI_HOB_RESOURCE_DESCRIPTOR` of type `EFI_RESOURCE_SYSTEM_MEMORY`. If the system memory is divided into multiple chunks then a `EFI_RESOURCE_SYSTEM_MEMORY` could be defined for every region. One of these HOBs must contain the free system memory region defined by `FreeMemoryBaseAddress` and `FreeMemoryLength` in the Phase Handoff Information Table (PHIT).
- The `EFI_HOB_TYPE_FV` which defines the location of the Firmware Volume in the memory map. The PEIM that declares this HOB can define the memory reserved for the firmware into a `HOB_RESOURCE_DESCRIPTOR` of type `EFI_RESOURCE_FIRMWARE_DEVICE`.

The `PeiHobLib` provides some helper functions to ease the declaration of these HOBs.

This phase can also support the decompression of compressed firmware volumes which contain code for the DXE and BDS phases..

UEFI can support various boot modes. For passing the boot mode to PEI and DXE phases, a PEIM needs to call the PEI Services function `SetBootMode()` with the specific boot mode.

The PPI `gEfiPeiMasterBootModePpiGuid` may also be installed by the same PEIM so that the PEIM can signal to the other PEIMs that the boot mode has been determined. The other PEIMs that have a dependency on the boot mode use the `NotifyPpi()` from PEI Services to detect when the boot mode has been determined.

```
Status    = (**PeiServices).SetBootMode (PeiServices, (UINT8)
BootMode);
ASSERT_EFI_ERROR (Status);
// With gEfiPeiMasterBootModePpiGuid defined into mPpiListBootMode
Status = (**PeiServices).InstallPpi (PeiServices, &mPpiListBootMode);
ASSERT_EFI_ERROR (Status);
```

Note: If a boot mode is not specified by any PEIMs then the mode `BOOT_WITH_FULL_CONFIGURATION` is applied.

To prevent memory fragmentation by different types of memory region when the UEFI drivers are loaded, a Memory Type Information HOB (`gEfiMemoryTypeInformationGuid`) may define regions of specific memory types. This HOB defines how many pages of each memory type should be allocated by this firmware. Otherwise, memory pages are converted in-flight to the appropriate type during the boot sequence.

5.1 Skip Temporary Memory

For some platforms, the DRAM system memory is initialized and is available at the first stage of the boot sequence either by design or for security reasons. Setting up the stack in a temporary memory does not make sense for these configurations.

As seen earlier, one of the primary goals of the PEI core is to move the UEFI stack from the temporary memory to the system memory. The PEI Core is executed twice in this situation, a first time using the temporary memory and a second time with its data in permanent memory.

The EDK2 PEI core is not a mandatory component of the UEFI Specification. In situation where the stack already exists in the permanent memory, the EDK2 PEI Core can be omitted (see Figure 7).

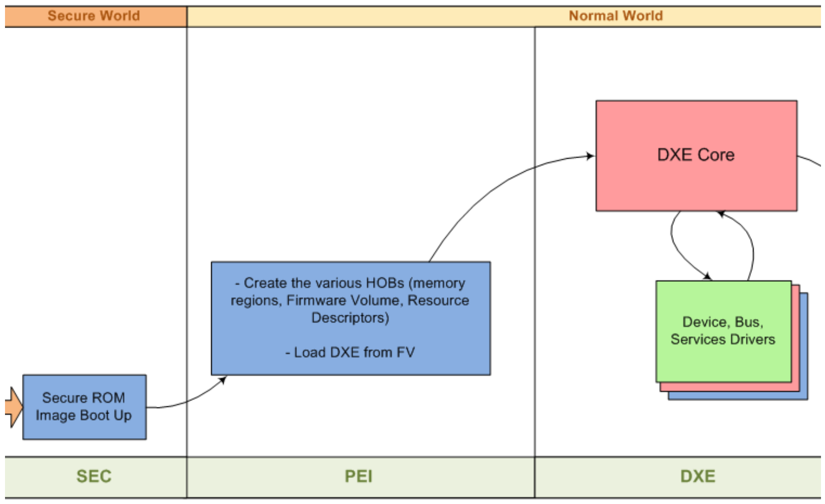
Secure World
Normal World
DXE Core
- Create the various HOBs (memory regions, Firmware Volume, Resource Descriptors)
- Load DXE from FV
Device, Bus, Services Drivers
Secure ROM Image Boot Up
SEC
PEI
DXE

Figure 7 Boot sequence when PEI Core is skipped

The DXE core expects some architectural protocols and HOBs to be implemented in the PEI phase. All of these requirements must be implemented in the phase between the Secure to Normal World transition and the call to DXE core.

Previously handled by the PEI Core, the memory regions allocated in the permanent memory which are still relevant during the DXE phase and later on must be tracked in Memory Allocation HOBs with their appropriate memory types. The MMU Translation Table could be one of these tracked allocations.

5.2 ARM Platform Pre-EFI Initialization Modules

- **ArmPlatformPkg/Pei/PlatformInitPeim.inf:** This component creates HOBs that declare:
  - the CPU Memory and IO sizes
  - the location of the Firmware Volume
  - the Boot Mode

  to the PEI Core.
- **ArmPlatformPkg/Pei/MemoryInitPeim.inf:** This module starts by initializing the system memory (DRAM), and then declares a region in this memory to be used by the PEI Foundation. The PEI Core migrates its stack and other memory allocations to this memory region for performance reasons.

  The MMU and its Translation Page Table are also configured at this stage. The DRAM resources are declared to PEI Core through Resource Descriptor HOBs.

6 Writing Driver (DXE Phase)

The DXE Phase initializes devices and services prior to the Rich OS boot. In addition to the required architectural protocol implementation, the other DXE drivers enable the OS to reside at various physical or non-physical (e.g.: boot through the network) locations of the platform.

6.1 Driver Creation

The first steps needed for creating a new DXE driver are as follows:

1) Select which package the new DXE driver should belong to (e.g.: `MdeModulePkg`, `ArmVePkg`, `ArmPkg`)
2) Create a folder for the driver and populate it with its configuration and source files
3) Generate a GUID to identify this driver
4) Fill the required sections of the INF file. [Defines], [Sources], [Packages], [LibraryClasses], [Protocols], [FixedPcd], [Depex] are some of the sections that a driver might require.
5) If this driver introduces new PCDs then add these PCDs to the DEC file of the package.

Note: These steps do not attach the driver to a hardware platform package. For integrating a driver to a platform firmware, the driver needs to be added to the DSC and FDF files of the platform package.

At boot time, before the DXE Core `CoreDispatcher()` function loads DXE drivers, the core module extracts the driver dependencies and puts the binaries in the correct order. Driver dependencies are written in their own section of the EFI binary. Dependencies are expressed by the [Depex] section of the INF file. Omitting this section implies a dependency on all architectural protocols.

After the DXE core loads a DXE driver into memory, it invokes its entry point (defined by ENTRYPOINT in the [Defines] section of the driver INF file).

The first parameter of the entry point function is an image handle which identifies the loaded driver during its execution time. Protocol instances produced by the driver could be attached to this handle.

UEFI defines different types of driver:

- The Device Driver consumes a bus I/O protocol and initializes its hardware. This driver must not create any child handles.
- The Bus Driver initializes, enumerates and manages a bus controller. The driver creates child handles for any controllers attached to its bus.
- The Services Driver does not manage any hardware. This driver provides services to other drivers, such as off-loading the complexity of some device drivers (e.g.: PCI Bus driver).

6.2 UEFI Driver Model

The UEFI specification defines the UEFI Driver Model that describes a generic model for UEFI driver. This definition includes a set of services and protocols that applies to drivers respecting this model.

From the UEFI Driver Model requirements, the entry point function is not allowed to touch any hardware. Instead, the UEFI Specification allows this function to install protocol instances to its image handle. To comply, the driver must install a Driver Binding Protocol instance. The instances of the Driver Configuration Protocol, the Driver Diagnostics

Protocol, and the Component Name Protocol may also be attached to the image handle in the entry point function.

Drivers that are UEFI Driver Model compliant, only initialize their hardware when required (e.g.: by the OS Loader). The aim is to reduce boot time by avoiding the initialization of unused drivers.

The Driver Binding Protocol exports three functions: `Supported()`, `Start()` and `Stop()`. The `Supported()` function checks if the controller handle passed as a parameter supports the requirements defined by the driver. If the requirements are satisfied then the function `Start()` of the Driver Binding Protocol instance is called and it installs its own additional protocol instances and initializes its hardware.

Note: The `Supported()` functions of all the Driver Binding Protocol instances are invoked every time a new EFI handle is connected.

A Device Path Protocol instance may be installed in the `Start()` function to represent the physical device in the system. This protocol is not required for all device handles. It must only be attached to hardware device drivers.

The Driver Binding Protocol does not apply to a Services Driver.

6.3 Example of the UEFI Driver Binding Protocol for a PCI controller

Below is an example of the UEFI Driver Binding Protocol implementation and initialization of a PCI controller.

```
EFI_DRIVER_BINDING_PROTOCOL gMyPCIDriverBinding = {
  MyPCIDriverBindingSupported, MyPCIDriverBindingStart,
MyPCIDriverBindingStop,
  0x30,NULL,NULL
};

EFI_STATUS EFIAPI MyPCIDriverBindingSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL *This,
  IN EFI_HANDLE                  Controller,
  IN EFI_DEVICE_PATH_PROTOCOL    *RemainingDevicePath
) {
    EFI_STATUS              Status;
    EFI_PCI_IO_PROTOCOL     *PciIo;
    UINT32                   PciID;

    // Test if the controller produces the PCI IO Protocol
   // Use the OpenProtocol function from Boot Services Table (gBS)
    Status = gBS->OpenProtocol (
        Controller,&gEfiPciIoProtocolGuid,
        (VOID **) &PciIo,
        This->DriverBindingHandle,Controller,
        EFI_OPEN_PROTOCOL_BY_DRIVER
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }

    // Read the Device and Vendor ID from the PCI configuration space
    Status = PciIo->Pci.Read (
        PciIo,EfiPciIoWidthUint32,PCI_VENDOR_ID_OFFSET,1,&PciID);
    if (EFI_ERROR (Status)) {
        Status = EFI_UNSUPPORTED;
        goto ON_EXIT;
    }

    // Test whether the controller belongs to MyPCI type
    if (PciID != ((MYPCI_DEVICE_ID << 16) | MYPCI_VENDOR_ID)) {
        Status = EFI_UNSUPPORTED;
    }

ON_EXIT:
    gBS->CloseProtocol (
         Controller,&gEfiPciIoProtocolGuid,This-
>DriverBindingHandle,Controller);
    return Status;
}

EFI_STATUS EFIAPI MyPCIDriverBindingStart (
  IN EFI_DRIVER_BINDING_PROTOCOL *This,
  IN EFI_HANDLE                  Controller,
  IN EFI_DEVICE_PATH_PROTOCOL    *RemainingDevicePath
) {
    EFI_STATUS               Status;
    EFI_PCI_IO_PROTOCOL      *PciIo;
    UINT64                   Supports;
    UINT64                   OriginalPciAttributes;
    BOOLEAN                  PciAttributesSaved;
    UINT32                   PciID;
    MYPCI_INSTANCE           **pMyPCIInstance = NULL;

    // Open the PciIo Protocol instance of the controller
    Status = gBS->OpenProtocol (
        Controller,&gEfiPciIoProtocolGuid,
        (VOID **) &PciIo,
        This->DriverBindingHandle,Controller,
        EFI_OPEN_PROTOCOL_BY_DRIVER
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }

    // Save the original PCI attributes
    //   (they must be restored in MyPCIDriverBindingStop())
    PciAttributesSaved = FALSE;
    Status = PciIo->Attributes (
        PciIo,EfiPciIoAttributeOperationGet,0,&OriginalPciAttributes);
    if (EFI_ERROR (Status)) {
        goto CLOSE_PCIIO;
    }
    PciAttributesSaved = TRUE;

    // Enable the PCI device on the bus
    Status = PciIo->Attributes (
        PciIo,EfiPciIoAttributeOperationSupported,0,&Supports);
    if (!EFI_ERROR (Status)) {
        Supports &= EFI_PCI_DEVICE_ENABLE;
        Status = PciIo->Attributes (
            PciIo,EfiPciIoAttributeOperationEnable,Supports,NULL);
    }
    if (EFI_ERROR (Status)) {
        goto CLOSE_PCIIO;
    }

    // Get the Pci device class code.
    Status = PciIo->Pci.Read (
        PciIo,EfiPciIoWidthUint32,PCI_VENDOR_ID_OFFSET,1,&PciID);
    if (EFI_ERROR (Status)) {
        Status = EFI_UNSUPPORTED;
        goto CLOSE_PCIIO;
    }

    // Test whether the controller belongs to MyPCI type
```

```
        if (PciID != ((MYPCI_DEVICE_ID << 16) | MYPCI_VENDOR_ID)) {
            Status = EFI_UNSUPPORTED;
            goto CLOSE_PCIIO;
        }

        // Create the Instance /Initialize HW /Install Protocols
        Status = MyPCIInitialization(PciIo, &pMyPCIInstance);
        if (EFI_ERROR (Status)) {
            goto CLOSE_PCIIO;
        }
        return EFI_SUCCESS;

    CLOSE_PCIIO:
        if (PciAttributesSaved) {
            // Restore original PCI attributes
            PciIo->Attributes (

    PciIo,EfiPciIoAttributeOperationSet,OriginalPciAttributes,NULL);
        }

        gBS->CloseProtocol (
             Controller,&gEfiPciIoProtocolGuid,This-
    >DriverBindingHandle,Controller);
        return Status;
    }

    EFI_STATUS EFIAPI InitializeMyPCI (
      IN EFI_HANDLE        ImageHandle,
      IN EFI_SYSTEM_TABLE  *SystemTable
    ) {
        return EfiLibInstallDriverBindingComponentName2 (
               ImageHandle,SystemTable,
               &gMyPCIDriverBinding,
               ImageHandle,
               &gMyPCIComponentName,&gMyPCIComponentName2);
    }
```

The entry point of this driver is the function `MyPCIInitialization()`. The only action done by this function is the installation of the instances for the Driver Binding Protocol and Component Name Protocols.

DXE core calls `MyPCIDriverBindingSupported()` after every new driver creation. This function returns EFI_SUCCESS if a new driver supports the PCI IO Protocol and if the PCI Device ID and PCI Vendor ID values match its own values. If this function succeeds then the DXE core calls the `MyPCIDriverBindingStart()` function.

`MyPCIDriverBindingStart()` enables the PCI space from the PCI Attribute register and creates its own instance and initializes the rest of its hardware in the function `MyPCIInitialization()`.

Figure 8 presents the interactions between some UEFI drivers and the handle creation when the file system on a SATA hard drive is mounted.

[1] When the DXE Core dispatches the `PciHostBridgeDxe` driver –the driver for the PCI Root Bridge of the platform– the driver creates two handles:

- The first handle manages resources for all the Root Bridges of the PCI controller
- A second handle for its PCI Root Bridge

[2] The PciBusDxe is later dispatched by DXE Core. It installs a Driver Binding Protocol on its image handle. The `Supported()` function of its Driver Binding Protocol is called for every EFI handles created. The function checks if the `PCI_ROOT_BRIDGE` and `DEVICE_PATH` protocols are attached to the newly created handle.

[3] Then PciBusDxe discovers the PCI Root Bridge handles installed by `PciHostBridgeDxe` that has just satisfied the condition. And it starts a PCI enumeration on this Root Bridge (**[4]**). A child handle with the `EFI_PCI_IO_PROTOCOL` and `DEVICE_PATH` protocols is created for each PCI controller detected on the PCI Bus.

[5] The SataDxe driver for a specific SATA controller is started. Its Supported() function waits for an EFI handles that supports `EFI_PCI_IO_PROTOCOL` (case of EFI handles attached to a PCI controller). If supported, the function opens the protocol and read the PCI configuration space of this controller to compare the VendorID and DeviceID with the ones that it supports. The `SataDxe` initializes its hardware controller and install an instance of `ATA_PASS_THRU` protocol to the EFI handle **[6]**.

[7] The `Supported()` function of the `AtaBusDxe` Driver Binding Protocol can now support the EFI handle that the `ATA_PASS_THRU` protocol has been installed to. This driver creates an EFI Handle that exposes the `DISK_IO`, `BLOCK_IO` and `DEVICE_PATH` protocols **[8]**.

[9], [10], [11] The `DiskIoDxe`, `PartitionDxe` and `FatDxe` drivers that all implement the Driver Binding Protocol get their Supported() function return TRUE during the drivers initialization process. An EFI Handle with a Simple File System protocol instance is created and can be used by any UEFI services to access files on the SATA driver.

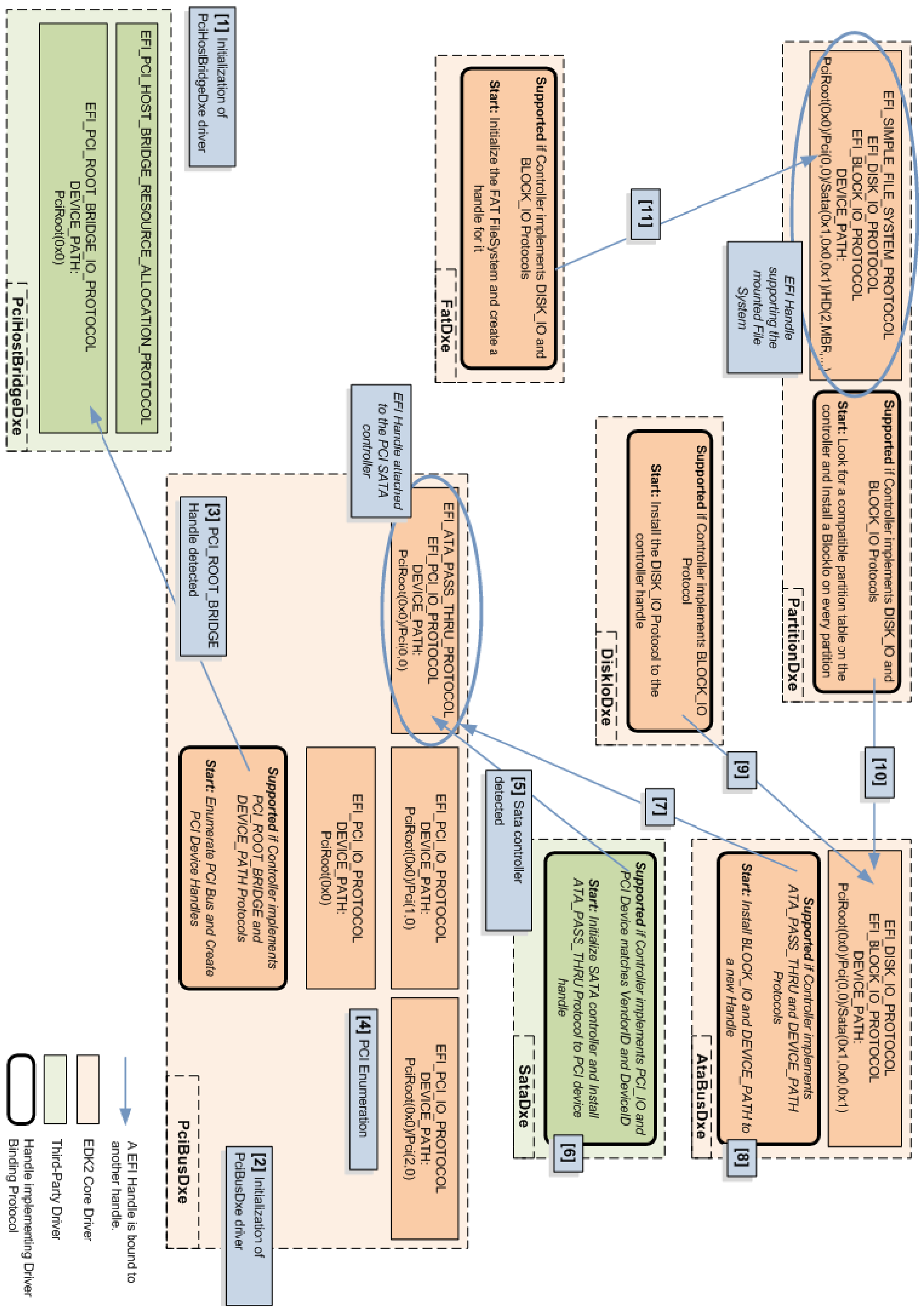
[1] Initialization of PciHostBridgeDxe driver
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL DEVICE_PATH: PciRoot(0x0)
PciHostBridgeDxe
Supported if Controller implements DISK_IO and BLOCK_IO Protocols
Start: Initialize the FAT FileSystem and create a handle for it
FatDxe
[11]
EFI Handle supporting the mounted File System
EFI_SIMPLE_FILE_SYSTEM_PROTOCOL EFI_DISK_IO_PROTOCOL EFI_BLOCK_IO_PROTOCOL DEVICE_PATH: PciRoot(0x0)/Pci(0,0)/Sata(0x1,0x0,0x1)/HD(2,MBR,...)
Supported if Controller implements DISK_IO and BLOCK_IO Protocols
Start: Look for a compatible partition table on the controller and install a BlockIo on every partition
PartitionDxe
Supported if Controller implements BLOCK_IO Protocol
Start: Install the DISK_IO Protocol to the controller handle
DiskIoDxe
EFI Handle attached to the PCI SATA controller
EFI_ATA_PASS_THRU_PROTOCOL EFI_PCI_IO_PROTOCOL DEVICE_PATH: PciRoot(0x0)/Pci(0,0)
[3] PCI_ROOT_BRIDGE Handle detected
EFI_PCI_IO_PROTOCOL DEVICE_PATH: PciRoot(0x0)/Pci(1,0)
EFI_PCI_IO_PROTOCOL DEVICE_PATH: PciRoot(0x0)
Supported if Controller implements PCI_ROOT_BRIDGE and DEVICE_PATH Protocols
Start: Enumerate PCI Bus and Create PCI Device Handles
[4] PCI Enumeration
EFI_PCI_IO_PROTOCOL DEVICE_PATH: PciRoot(0x0)/Pci(2,0)
PciBusDxe
[2] Initialization of PciBusDxe driver
[5] Sata controller detected
Supported if Controller implements PCI_IO and PCI Device matches VendorID and DeviceID
Start: Initialize SATA controller and Install ATA_PASS_THRU Protocol to PCI device handle
SataDxe
[6]
[7]
[9]
[10]
EFI_DISK_IO_PROTOCOL EFI_BLOCK_IO_PROTOCOL DEVICE_PATH: PciRoot(0x0)/Pci(0,0)/Sata(0x1,0x0,0x1)
Supported if Controller implements ATA_PASS_THRU and DEVICE_PATH Protocols
Start: Install BLOCK_IO and DEVICE_PATH to a new Handle
AtaBusDxe
[8]
A EFI Handle is bound to another handle.
EDK2 Core Driver
Third-Party Driver
Handle implementing Driver Binding Protocol

Figure 8 Example of the EFI handles created when a File System on a SATA Hard drive is mounted

6.4 Memory management API

The memory map of the platform is managed by the Global Coherency Domain (GCD) API. These functions are part of the DXE Services Table. The GCD contains the regions, the types and the owner of all the memory and IO regions.

Memory attributes are changed by using the GCD `SetMemorySpaceAttributes()`function. Any operation is mirrored to the hardware through the CPU Architectural protocol.

A DXE driver that enables a new region of the platform memory map needs to declare this memory or IO space to the GCD through the `AllocateMemorySpace()/AddMemorySpace()` and `AllocateIoSpace()/AddIoSpace()` functions respectively. A PCI Root Bridge is typically a driver which needs to declare its space to the DXE core by using these functions.

Find below the example of setting a memory region as uncached memory:

```
Status = gDS->GetMemorySpaceDescriptor (BaseAddress, &GcdDescriptor);
ASSERT_EFI_ERROR(Status);

Status = gDS->SetMemorySpaceAttributes (
   BaseAddress,Length,GcdDescriptor.Attributes | EFI_MEMORY_UC);
ASSERT_EFI_ERROR(Status);
```

6.5 Event API

DXE drivers can react to various types of events. The Boot Services Table contains functions to manage these events. Events can be created to signal:

- **Exit Boot Services:** this event is called when `ExitBootServices()` function is invoked generally just before the Operating System is starting up. Drivers generally create this event to disable the hardware they manage. For example, the Linux kernel expects interrupts to be disabled before it boots.

  The following code snippet declares the `ExitBootServicesEvent()` function as the hook to be called when the `EfiExitBootServicesEvent` is signalled.

  ```
  Status = gBS->CreateEvent (EVT_SIGNAL_EXIT_BOOT_SERVICES,
  TPL_NOTIFY,
     ExitBootServicesEvent, NULL, &EfiExitBootServicesEvent);
  ASSERT_EFI_ERROR (Status);
  ```

- **Timer events**: They are events that could be invoked once or periodically. After creating a timer event, the SetTimer() function must be called to specify when the event is signalled. Timer events are only supported if the Timer Architectural Protocol is installed.

  ```
  WaitList[0] = gST->ConIn->WaitForKey;
  gBS->CreateEvent (EVT_TIMER, 0, NULL, NULL, &WaitList[1]);
  gBS->SetTimer (WaitList[1], TimerPeriodic,
  EFI_SET_TIMER_TO_SECOND);
  /* (...) */
  Status = gBS->WaitForEvent (2, WaitList, &WaitIndex);
  ```

  This example is an extract from EBL shell. It creates a timer event which occurs every second. Later in the code, the `WaitForEvent()` function waits for either the timer event or a pressed key event to be signalled.

  Another way to get signalled timer event is to attach a function handle to the event. In the following example, `EhcMonitorAsyncRequests()` function is called every time the event `Ehc->PollTimer` is signalled.

  ```
  Status = gBS->CreateEvent (
        EVT_TIMER | EVT_NOTIFY_SIGNAL, TPL_CALLBACK,
  ```

```
                EhcMonitorAsyncRequests, Ehc, &Ehc->PollTimer
          );
```

- **Event Notification**: After the event handle is created, it can be attached to any other notification mechanism. One use case of this type of event is Protocol Notification. Drivers can request to be notified when another driver installs a specific protocol.

```
Status = gBS->CreateEvent (
      EVT_NOTIFY_SIGNAL,
TPL_CALLBACK,GdbSymbolEventHandler,NULL,&gEvent);
ASSERT_EFI_ERROR (Status);
Status = gBS->RegisterProtocolNotify (

&gEfiLoadedImageProtocolGuid,gEvent,&gGdbSymbolEventHandlerRegis
tration);
```

The above example is extracted from the GDB driver. This driver is notified every time a new driver has been created (a Loaded Image Protocol instance is created for any new driver).

6.6 Parallel Tasks

UEFI can create tasks which could be invoked at any time. The concept of *Task Priority Level* (TPL) is used to avoid tasks of the same priority being overlapped. Priority levels start from the lowest priority level `TPL_APPLICATION` to the highest priority `TPL_HIGH_LEVEL`. Only tasks with a higher priority can interrupt an active task.

This common use case is the timer based events that their invocations are difficult to predict in a non real-time system.

To prevent a section of code being interrupted by a timer event related task, you must raise the task priority level for this code to the highest level (`TPL_HIGH_LEVEL`). This disables interrupts and therefore prevents timer events from interrupting the task.

```
OriginalTPL = gBS->RaiseTPL (TPL_HIGH_LEVEL);
/* Critical Section */
gBS->RestoreTPL (OriginalTPL);
```

6.7 Exception and Interrupt Supports

The CPU architectural protocol supports exception handling. A handle may be attached to any of the seven ARM exception entries (Reset, Undefined instruction, Software interrupt, Prefetch Abort, IRQ, FIQ). With the exception of the producer of the Hardware Interrupt Protocol (`gHardwareInterruptProtocolGuid`), a device driver should not attach its own handler to an exception entry because it could discard the active handler (there is no function to save and restore an exception handler).

To register a handler to a specific interrupt source on ARM system, a driver must register its handler with the `RegisterInterruptSource()` function of the Hardware Interrupt Protocol.

The example below is from the TimerDxe driver (producing the Timer Architectural Protocol) of the ARM RealView Emulation Board:

```
Status = gBS->LocateProtocol (
    &gHardwareInterruptProtocolGuid, NULL,(VOID **)&gInterrupt);
ASSERT_EFI_ERROR (Status);
Status = gInterrupt->RegisterInterruptSource (
    gInterrupt, EB_TIMER01_INTERRUPT_NUM, TimerInterruptHandler);
```

6.8 DXE Driver debugging

In the early stages of DXE driver development it is preferable to only build the driver sources and not the whole firmware. The module INF file can be passed to the build tool to only build the component:

```
build -a ARM -p ArmVePkg/ArmVePkg.dsc -t RVCTCYGWIN -m
ArmVePkg/Library/EblCmdLib/EblCmdLib.inf
```

If the argument `-p [dsc_file]` is not passed then the build tool uses the active platform defined by the `Conf\target.txt` file.

The `DEBUG()` macro is useful for printing statements in a debug build. Multiple levels of debug support are available. Each of them can be individually turn on/off in the DSC file of the active platform. It is the PCD value of `gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel` that specifies which debug levels are active.

Code that is specific to debug builds must be enclosed by the macros `DEBUG_CODE_BEGIN()` and `DEBUG_CODE_END()`.

Level of debug support (Assert, Debug Statement, Debug Code, etc) is regulated by `gEfiMdePkgTokenSpaceGuid.PcdDebugPropertyMask`.

See ARM Debugging Environment on page 14 for information about how to use a debugger.

6.9 DXE Driver Testing

Unit tests can be encapsulated by the UEFI Diagnostic Protocol. Debug messages must not directly print in the terminal. The protocol instance must allocate and use the buffer pointer passed as argument to its `RunDiagnostics()` function to return any output.

Diagnostic Protocol interfaces can be invoked through the EFI Shell command `drvdiag`.

7 Shells and OS Loaders (BDS Phase)

After the previous phases have initialized the platform and drivers, the BDS - the last phase of the boot sequence - brings the Rich OS or any UEFI shell to live.

7.1 UEFI Shells

The EFI Shell and *EFI Boot Loader* (EBL) are two shells that allow the user to interact with UEFI. Both must be loaded from the BDS phase. The motivation of EBL is to deliver a simpler shell than the EFI Shell. The average size of EBL shell is around 65KB compared to the 670KB of the EFI Shell.

Shells are useful for debugging the UEFI drivers and firmware. The following paragraphs briefly cover each shell and some of their commands.

The EBL defines the environment variable `default-cmdline` that is used when the shell starts up for executing some initial commands. If this variable does not exist then it executes the commands specify by the PCD:

`gEmbeddedTokenSpaceGuid.PcdEmbeddedAutomaticBootCommand`.

All EBL commands are not enabled by default. PCDs need to be edited to enable some of them. For example the EBL does not support batch script by default.

`gEmbeddedTokenSpaceGuid.PcdEmbeddedScriptCmd`

needs to be set to true to enable the batch script feature.

The following EBL example lists the boot devices, then lists all the files of the first Firmware Volume and then starts the EFI Application located on FV1 with a GUID starting with C57AD6B7.

```
ArmVexpress >help device
device; Show information about boot devices
ArmVexpress >device
Firmware Volume Devices:
  fv0: 0x6F800000 - 0x6FEFFFFF : 0x00700000
  fv1: 0x67DBE000 - 0x67EC1CBF : 0x00103CC0
Block IO Devices:
  blk0: Removable No Media
ArmVexpress >cd fv0:
ArmVexpress fv0:\>dir
   27,896     SEC C536BBFE-C813-4E48-9F90-01FE1ECF9D54
  307,883      FV 9E21FD93-9C72-4C15-8C4B-E77F1DB2D792
            335,779 bytes in files 3,071,625 bytes free
ArmVexpress fv0:\>start fv1:\C57AD6B7
```

Additional commands can easily be added to this shell. The EBL requires the `EblCmdLib` to build. This library contains the additional commands. This file is generally provided by the platform specific package. If no additional command is required, the platform description file (DSC) points to an empty implementation of this library, that is:

`\EmbeddedPkg\Library\EblCmdLibNull\EblCmdLibNull.inf`.

For creating a new `EblCmdLib` library to extend the EBL:

1) Create a new `EblCmdLib` library (create a new folder and link this folder to `EblCmdLib` into the platform DSC file)
2) Create one or multiple functions that handle the new EBL commands. These function must follow this definition:

   ```
   EFI_STATUS (EFIAPI *EBL_COMMMAND)(IN UINTN  Argc,IN CHAR8  **Argv)
   ```
3) Create an array of type `EBL_COMMAND_TABLE` and add the new exported commands to this array.

4) Create the function `EblInitializeExternalCmd()` (check the function prototype from `\EmbeddedPkg\Include\Library\EblCmdLib.h`) and call the function `EblAddCommands()` with the `EBL_COMMAND_TABLE` type array.

The EFI Shell provides many more commands. In consequence, the size of the shell is much bigger than the EBL. Environment variables and batch scripts (.nsh) are supported. Firmware volumes cannot be listed in the same way as the EBL.

Table 5: A few of the more common EBL commands

EFI Shell Command	Description
help [cmd]	List all available EFI Shell commands. Or return information about the command passed as an argument.
dh	Display information about all EFI handles.
drvdiag	Invokes the Driver Diagnostics Protocol
map	Displays or defines mappings. Devices need to be mapped to be accessible.
set	Displays or modifies EFI Shell environment variables

7.2 OS Loader

An OS Loader should be able to launch a specific Operating System from any device providing file support (for example through a file system or through the network).

A UEFI OS Loader should take advantage of the Device path Protocol. This protocol attaches a device name defined by the UEFI Specification to a specific hardware device handle. Thus, a file location can be defined by a Define Path such as the one presented in Figure 10. The loader scans the Device Path and ensures every Device Path node is attached to a running driver.

The function `ConnectController()` of the Boot Services Table helps to start the drivers required by a device path.

On a UEFI firmware composed by only drivers following the UEFI Driver Model, the only drivers that touch the hardware should be the drivers implementing the architectural protocols and the drivers attached to Device Path Nodes defining the location of the Operating System (or its kernel).

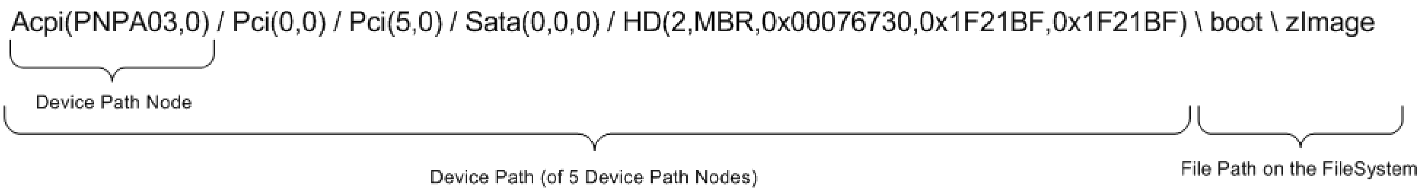
Acpi(PNPA03,0) / Pci(0,0) / Pci(5,0) / Sata(0,0,0) / HD(2,MBR,0x00076730,0x1F21BF,0x1F21BF) \ boot \ zImage
Device Path Node
Device Path (of 5 Device Path Nodes)
File Path on the FileSystem

Figure 9: Example of Device Path defining a path to a partition on a SATA bus

Table 6: Description of the Device Path Nodes of Figure 10

Device Path Node	Name	Protocol Instances attached to the device handle
Acpi(PNPA03,0)	Pci Root Bridge	PciRootBridgeIo
Pci(0,0)	Pci to Pci Bridge	PciIo
Pci(5,0)	Pci Controller	PciIo
Sata(0,0,0)	SATA controller	DiskIo BlkIo
HD(2,MBR,0x00076730,0x1F21BF,0x1F21BF))	2nd partition of the Hard Drive	Fs DiskIo BlkIo

Once all files and information are loaded from UEFI services, the OS loader calls `ExitBootServices()` to ensure all the drivers and services are correctly disabled to avoid any interference during the OS kernel booting.

`ExitBootServices()` takes `MapKey` as an argument. MapKey is returned by `GetMemoryMap()` and represents the memory map at the time this function is called. If the `MapKey` passed to `ExitBootServices()` does not match the current value then the function returns the `EFI_INVALID_PARAMETER` error code. Even if `GetMemoryMap()` is called just before `ExitBootServices()`, `MapKey` could not match the current one as some device drivers may have changed the memory map over the time. This may be the case of drivers responding to timer based events (e.g.: Ethernet or USB drivers).
Therefore, multiple calls to `ExitBootServices()` might occur before it succeeds.

Each Operating System has its own requirements about how to handle arguments and expect the state of the hardware. ARM Linux kernel has its requirements defined in the *ARM Linux Kernel Boot Requirements* .

ARM Linux kernel can take either ATAG or a Flattened Device Tree (FDT) binary address as argument. The UEFI Linux OS Loader might have to support both formats. These arguments are expected to be located at the first 16KB of the System Memory. The OS Loader is the last component of the UEFI boot sequence and it can rearrange the memory, after calling `ExitBootServices()` to move information to the correct location.

The Linux kernel also requires the MMU to be turned off and the data and instruction caches to be disabled.

7.3 Custom BDS

All OEMs do not have the same expectations from the BDS phase. Some of them wish start a Rich OS as fast as they can. Other prefers to provide a shell after the DXE phase. The Human Interface might differ between platforms following their requirements. A Graphic User Interface with KMI (Keyboard and Mouse Interface) over USB support is expected for some devices while a serial terminal could be the only interface of UEFI firmware on development boards. Creating a custom BDS is the answer to all these variations.

The new BDS module must implement the BDS architectural protocol (`EFI_BDS_ARCH_PROTOCOL`). This protocol interface contains a single function called by the DXE core when all the DXE drivers have been dispatched.

7.4 ARM BdsLib

The ARM `BdsLib` is part of the common ARM Platform components. This helper library exposes functions to help the BDS developer to only focus on the Human Interface Machine and reduce the inclusion of platform specific code into the BDS driver.

```
/**
  Connect all DXE drivers

  @retval EFI_SUCCESS           All drivers have been connected
  @retval EFI_NOT_FOUND         No handles match the search.
  @retval EFI_OUT_OF_RESOURCES  There is not resource pool memory to
store the matching results.

**/
EFI_STATUS
BdsConnectAllDrivers (
  VOID
  );

/**
  Start a Linux kernel from a Device Path

  @param  LinuxKernel           Device Path to the Linux Kernel
  @param  CommandLine           Linux kernel agruments
  @param  Fdt                   Device Path to the Flat Device Tree

  @retval EFI_SUCCESS           All drivers have been connected
  @retval EFI_NOT_FOUND         The Linux kernel Device Path has not
been found
  @retval EFI_OUT_OF_RESOURCES  There is not enough resource memory to
store the matching results.

**/
EFI_STATUS
BdsBootLinux (
  IN  CONST CHAR16* LinuxKernel,
  IN  CONST CHAR8*  CommandLine,
  IN  CONST CHAR16* Fdt
  );

/**
  Start an EFI Application from any Firmware Volume

  @param  EfiApp                EFI Application Name

  @retval EFI_SUCCESS           All drivers have been connected
  @retval EFI_NOT_FOUND         The Linux kernel Device Path has not
been found
  @retval EFI_OUT_OF_RESOURCES  There is not enough resource memory to
store the matching results.

**/
EFI_STATUS
BdsLoadApplication (
  IN  CHAR16* EfiApp
  );

/**
  Start an EFI Application from a Device Path

  @param  EfiAppPath            Device Path to the EFI Application

  @retval EFI_SUCCESS           All drivers have been connected
  @retval EFI_NOT_FOUND         The Linux kernel Device Path has not
been found
  @retval EFI_OUT_OF_RESOURCES  There is not enough resource memory to
store the matching results.

**/
EFI_STATUS
```

```
    BdsLoadApplicationFromPath (
      IN  CHAR16* EfiAppPath
      );
```

`BdsBootLinux()` takes care about the ATAG passed to the Linux Kernel. The function fills ATAG memory structures with the System Memory Resource Hobs declared to DXE Core.

Default values are defined to set the maximum addresses where the ATAG list and the Linux kernel image must be located. The platform designer can overwrite these values if required.

8 Porting UEFI to a new ARM Platform

The aim of this last section is to explain how to port UEFI on a ARM platform. This section makes references to the information previously covered by this document. Some answers to these questions are covered by the Board Boot Design Document .

Every platform has its own memory map and boot and security strategies. The UEFI Common ARM Platform Components have been designed to be flexible enough to be reused on any platform.

8.1 Platform and UEFI Memory Maps

8.1.1 Initial Memory

On a platform that implements secure boot it is likely you have got two firmwares; a first firmware to manage the secure boot phase and a second firmware for the all the phase between the transition to normal world and the Operating System booting.

ARMv7-A series processors always start in the Secure world. Before going from Secure to Normal world, it is advised to initialize the Secure Monitor world. This is the mode in which the CPU transit before going back to Secure World coming from Normal world.

These three modes (Secure, Secure Monitor, and Normal worlds) require stacks. For an MPCore processor, each core also needs its own stack in every mode.

Figure 11 presents the memory map in the initial memory before the DRAM has been initialized.

Table 7 Memory map in the initial memory

PCD	Description
gArmTokenSpaceGuid.PcdSecureFdBaseAddress gArmTokenSpaceGuid.PcdSecureFdSize	Defines the region that contains the Secure UEFI Firmware. This firmware generally implements the UEFI SEC phase.
gArmTokenSpaceGuid.PcdNormalFdBaseAddress gArmTokenSpaceGuid.PcdNormalFdSize	Defines the region that contains the UEFI Firmware for the Normal world. This firmware manages the boot sequence from the PEI to BDS phases
gArmPlatformTokenSpaceGuid.PcdCPUCoresSecStackBase gArmPlatformTokenSpaceGuid.PcdCPUCoreSecStackSize	The PcdCPUCoresSecStackBase PCD represents the base of the stacks for the Secure World. PcdCPUCoreSecStackSize is the size of a stack for one core.
gArmPlatformTokenSpaceGuid.PcdCPUCoresSecMonStackBase gArmPlatformTokenSpaceGuid.PcdCPUCoreSecMonStackSize	The PcdCPUCoresSecMonStackBase PCD represents the base of the stacks for the Secure Monitor World. PcdCPUCoreSecMonStackSize is the size of a stack for one core.
gArmPlatformTokenSpaceGuid.PcdCPUCoresNonSecStackBase gArmPlatformTokenSpaceGuid.PcdCPUCoresNonSecStackSize	The PcdCPUCoresNonSecStackBase PCD represents the base of the stacks for the Normal (Non Secure) World. PcdCPUCoreNonSecStackSize is the size of a stack for one core.

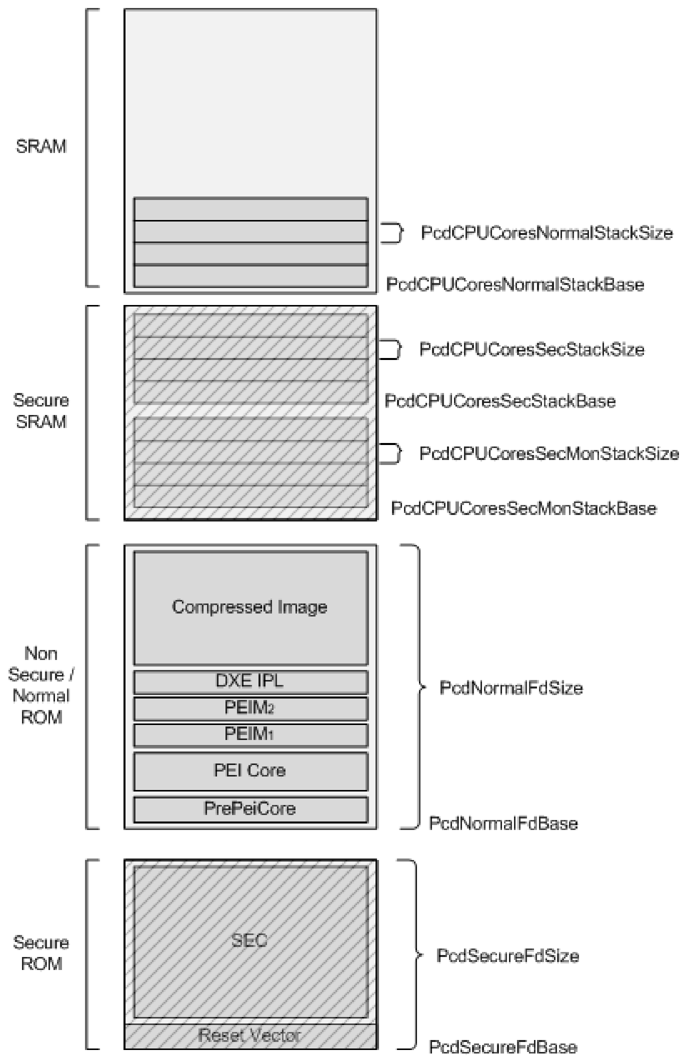

Figure 10: UEFI Firmwares and Initial Stacks on a MPCore (x4 cores) Platform

8.1.2 Permanent Memory

The PEIM that initializes the DRAM reserved a region of the system memory for the PEI Foundation (See Section 5. Platform Foundation for EFI (PEI Phase)). The DXE Foundation also uses this region for its allocations.

Some UEFI drivers might require allocation at a specific location. These locations are generally at the base of the system memory.

The Operation System loader may require loading the kernel and its parameters in a specific region.

Table 8 PCDs defining the System Memory partitionning

PCD	Description
gArmTokenSpaceGuid.PcdSystemMemoryBase gArmTokenSpaceGuid.PcdSystemMemorySize	Define the biggest built-in DRAM memory region. DRAM extensions are declared in a different way (See 8.3 ArmPlatformLib interface)
gArmPlatformTokenSpaceGuid. PcdSystemMemoryUefiRegionSize	Defines the size of the memory region reserved for the PEI and DXE Foundations. This region is reserved at the top of the built-in system memory.
gArmPlatformTokenSpaceGuid. PcdSystemMemoryFixRegionSize	Defines the size of the memory region reserved for the fixed address allocations. This region is reserved at the bottom of the built-in system memory.

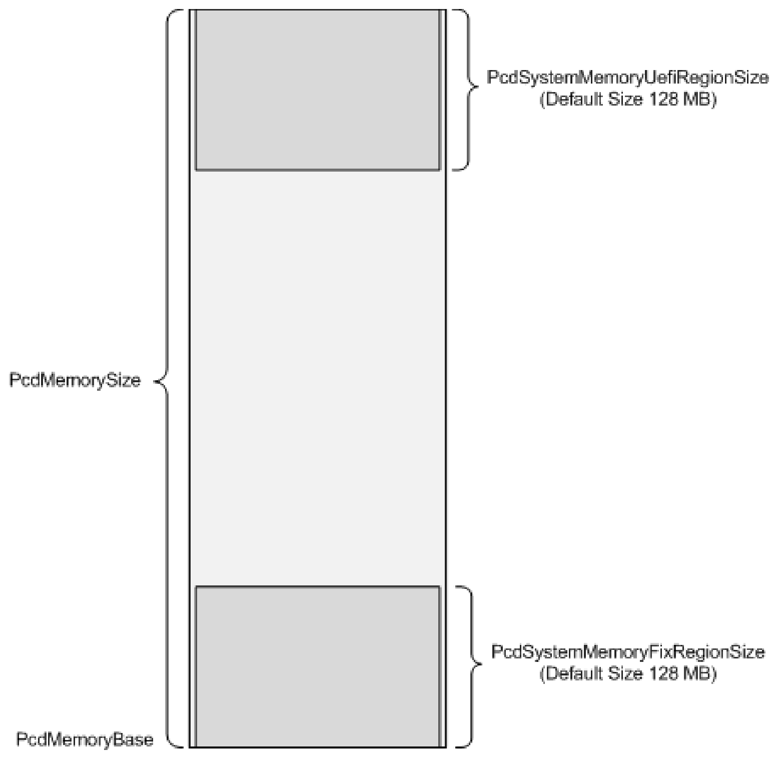

Figure 11: System Memory partitionning

8.2 UEFI Component Selection and Firmware Creation

Once the UEFI memory map defined, the UEFI designers have to select the base and the architectural and platform components they want to be supported by their Secure and Normal firmwares.

8.2.1 Secure Firmware

The Secure firmware is the first firmware executed on the ARM processor.

A Reset Vector is added at the base of the firmware by the EDK2 BaseTool GenFw.

The tool patches the first entry (fetched by the CPU at reset) of the Reset Vector to make a jump to the SEC module of this firmware.

The firmware contains at least the component `ArmPlatformPkg/Sec` (or any similar SEC module).

Note: Some peripherals need to be initialized in the Secure world.

8.2.2 Normal Firmware

This firmware supports the PEI, DXE and BDS phases. It must at least contain the DXE core and an implementation of all the architectural protocols (See Table 3). Some architectural protocols might not be relevant to the platform. In this case, universal implementations can be used instead.

The PEI phase can be realized either by using the EDK2 PEI Core and its PEIMs (`ArmPlatformPkg/Pei/PlatformInitPeim.inf`, `ArmPlatformPkg/Pei/MemoryInitPeim.inf`, etc) or by using the `ArmPlatformPkg/Pei/Pei.inf` component that installs the HOBs required by the DXE Core.

Some Bus supports and their generic drivers might also be added. EDK2 provides drivers that support the PCI and USB buses. Generic drivers for the USB keyboard, mouse and mass storage also exist.

The different supports from which an Operating System can be booted on the targeted board define some of the drivers to include in the firmware.

Note: External UEFI Firmware Volumes can be used to contain additional PEI and DXE drivers.

8.2.3 UEFI Configuration Files

Once the UEFI memory map and the components are chosen, the UEFI Platform designer can start to fill the UEFI configuration files used to generate the UEFI firmwares.

The easiest solution to generate this file is to copy those from an existing platform similar to the targeted platform. Do not forget to generate new GUIDs for new files and unique objects.

There are two configuration files required to generate a new platform: the DSC and FDF files (See Figure 2).

A FDF file contains the description of one or more UEFI firmwares.

8.3 ArmPlatformLib interface Implementation

ArmPlatformLib is the library that allows platform abstraction in the Common ARM Platform components. It is required to implement this library for the new platform in order to take advantage of the common components.

The interface of the library is defined by the header:
`ArmPlatformPkg/Include/Library/ArmPLatformLib.h`

This is the list of the functions to implement:

Function:

```
VOID
ArmPlatformIsMemoryInitialized (
  VOID
  );
```

Description: Called at the early stage of the Boot phase to know if the memory has already been initialized.

	Running the code from the reset vector does not mean we start from cold boot. In some case, we can go through this code with the memory already initialized. Because this function is called at the early stage, the implementation must not use the stack. Its implementation must probably be done in assembly to ensure this requirement.
Parameters:	None
Return Value:	This function expect to set the ‘zero’ CPSR flag

Function:	`VOID`
	`ArmPlatformInitializeBootMemory` `(`
	`  VOID`
	`  );`

Description:	Initialize the memory where the initial stacks will reside This memory might contain the initial stacks (Secure and Secure Monitor stacks). In some platform, this region is already initialized and the implementation of this function can do nothing. This function can also initialize the Secure RAM. This function is called before the stack has been set up. Its implementation must ensure the stack pointer is not used (probably required to use assembly language).
Parameters:	None
Return Value:	None

Function:	`EFI_BOOT_MODE`
	`ArmPlatformGetBootMode` `(`
	`  VOID`
	`  );`

Description:	Return the current Boot Mode This function returns the boot reason on the platform.
Parameters:	None
Return Value:	Return the current Boot Mode of the platform.

Function:	`VOID`
	`ArmPlatformInitialize (`
	`  VOID`
	`  );`

Description:	Initialize controllers that must setup at the early stage Some peripherals must be initialized in Secure World. For example, some L2x0 requires to be initialized in Secure World
Parameters:	None
Return Value:	None

Function:	`VOID`
	`ArmPlatformInitializeSystemMemory` `(`
	`  VOID`
	`  );`

Description:	Initialize the system (or sometimes called permanent) memory This memory is generally implemented by the DRAM.
Parameters:	None
Return Value:	None

Function:	`VOID`
	`ArmPlatformBootRemapping` `(`
	`  VOID`
	`  );`

Description:	Remap the memory at 0x0 Some platform requires or gives the ability to remap the memory at the address 0x0. This function can do nothing if this feature is not relevant to your platform.
Parameters:	None
Return Value:	None

Function:	`UINTN`
	`ArmPlatformTrustzoneSupported` `(`
	`  VOID`
	`  );`

Description:	A non-zero value must be returned if you want to support a Secure world on your platform. `ArmPlatformTrustzoneInit()` will later set up the secure regions. This function can return 0 even if TrustZone is supported by your processor. In this case, the platform will continue to run in Secure world.
Parameters:	None
Return Value:	Return if TrustZone is supported by your platform

Function:	`VOID`
	`ArmPlatformTrustzoneInit` `(`
	`  VOID`
	`  );`

Description:	Initialize the Secure peripherals and memory regions If TrustZone is supported by your platform then this function makes the required initialization of the secure peripherals and memory regions.
Parameters:	None
Return Value:	None

Function:	`VOID` `ArmPlatformGetVirtualMemoryMap (` `OUT ARM_MEMORY_REGION_DESCRIPTOR** VirtualMemoryMap` `);`
Description:	Return the Virtual Memory Map of your platform This Virtual Memory Map is used by the `MemoryInitPei` Module to initialize the MMU on your platform.
Parameters:	*VirtualMemoryMap*: Array of `ARM_MEMORY_REGION_DESCRIPTOR` describing a Physical-to-Virtual Memory mapping. This array must be ended by a zero-filled entry.
Return Value:	None

Function:	`EFI_STATUS` `ArmPlatformGetAdditionalSystemMemory (` `OUT ARM_SYSTEM_MEMORY_REGION_DESCRIPTOR** EfiMemoryMap` `);`
Description:	Return the EFI Memory Map of your platform This EFI Memory Map of the System Memory is used by the `MemoryInitPei` module to create the Resource Descriptor HOBs used by DXE core.
Parameters:	*EfiMemoryMap:* Array of `ARM_SYSTEM_MEMORY_REGION_DESCRIPTOR` describing an EFI Memory region. This array must be ended by a zero-filled entry
Return Value:	Return `EFI_SUCCESS` if implement or `EFI_UNSUPPORTED` if not.

8.4 Case Study: UEFI on a ARM Platform with a proprietary firmware

Some OEMs have their own proprietary firmware that initializes the platform interconnects and memory controllers. These proprietary firmwares might also be responsible for initializing the Secure world and making the transition to Normal world.

In this case, the UEFI firmware is limited to the Normal world phases and the SEC phase can be skipped as it is realized by the proprietary firmware.

The PEI Core can also be skipped as the DRAM is already initialized.

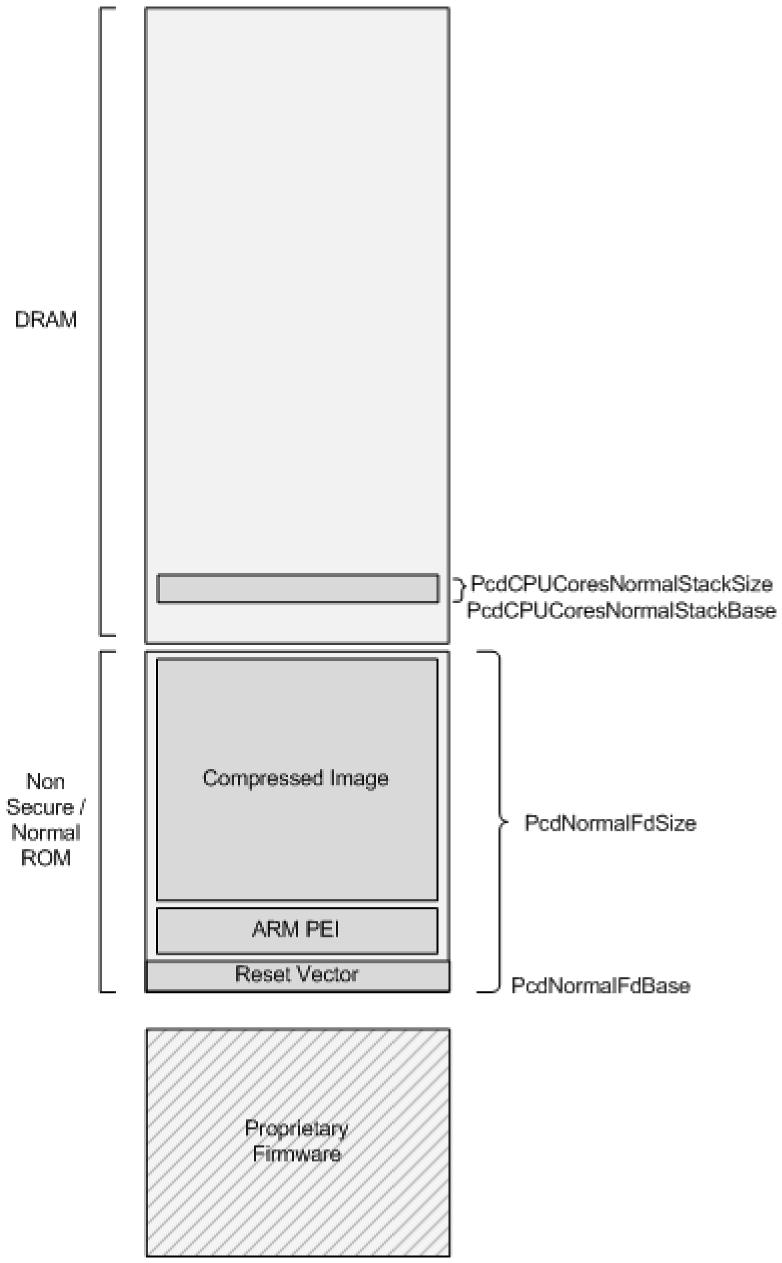

Figure 12: UEFI on a ARM Platform with a proprietary firmware

9 UEFI Testing and Measurement

9.1 UEFI Testing

The UEFI Forum provides UEFI members with a conformance suite, the *Self Certification Test* (SCT). The SCT verifies conformance to the UEFI and PI specifications. The SCT is provided as two different packages for each specification.

The SCT for UEFI specification covers the UEFI Services (Boot and Runtime Services) and the protocols defined by the specification, such as the PCI Bus Support, the USB Bus Support, and the Network Protocols.

The version for PI specification tests the PEI core services (PEI Services Table, the PPIs), the DXE core services (Global Coherency Domain, Architectural protocols, DXE Services Table), the HOBs, and the Firmware Volume Protocols.

BDS Shells can also be used for testing drivers. File commands (create, copy, delete files) can be used to test file system oriented drivers. Memory leaks can be investigated with the `memmap` command in the EFI Shell. The `dh` command can dump the Device Paths and EFI Handles.

9.2 UEFI Measurement

9.2.1 Time Measurement

The Performance Protocol (`gPerformanceProtocolGuid`) exposes functions for measuring time performance. Time acquisitions are done in the different parts of the UEFI firmware by this interface. The protocol can also directly be invoked by any module to measure a specific section of code. These time durations are stored in the performance library and can be dumped by any external application such as a shell command.

The time spent in the initialization of all the DXE drivers and UEFI phases can be listed by the command ‘perf’ under the EBL shell.

The Linux loader implemented by ARM BdsLib also prints out the time spent in the different UEFI phases before booting the kernel. The time measure for the BDS phase does not count the time spent in the various shells and boot manager. It only measures the time elapses before the kernel image is loaded to the time just before calling the kernel.

9.2.2 Size Measurement

We saw in Section 2.9 part of the firmware can be compressed to reduce the firmware foot print. To get an idea of the size of the binaries contained in the firmware, the EDK2 BaseTools command `volinfo` can be used on a Firmware Volume (`.fv` file extension) or a Firmware Device (`.fd` file extension).

This command returns information about a firmware file. The list of files and their details (offset, size, type, dependencies, etc.) are printed out in the standard output.

The filenames printed by the tool are the GUID of the related files. A file-guid cross reference file (`.xref` extension) may be passed as an argument to print a human readable name. The EDK2 Build system creates a file ‘Guid.xref’ after the firmware file generation.

```
> volinfo -x Guid.xref ARMVEXPRESS_EFI.fd

(...)
============================================================
File Name:        B336F62D-4135-4A55-AE4E-4971BBF0885D  RealTimeClock
File Offset:      0x00034420
File Length:      0x00002744
File Attributes:  0x00
File State:       0xF8
        EFI_FILE_DATA_VALID
File Type:        0x07  EFI_FV_FILETYPE_DRIVER
------------------------------------------------------------
  Type:  EFI_SECTION_DXE_DEPEX
  Size:  0x00000006
        TRUE
        END DEPEX
------------------------------------------------------------
  Type:  EFI_SECTION_PE32
  Size:  0x00002704
------------------------------------------------------------
  Type:  EFI_SECTION_USER_INTERFACE
  Size:  0x00000020
============================================================
File Name:        4C6E0267-C77D-410D-8100-1495911A989D  MetronomeDxe
File Offset:      0x00036B68
File Length:      0x00001F22
File Attributes:  0x00
File State:       0xF8
        EFI_FILE_DATA_VALID
File Type:        0x07  EFI_FV_FILETYPE_DRIVER
------------------------------------------------------------
  Type:  EFI_SECTION_DXE_DEPEX
  Size:  0x00000006
        TRUE
        END DEPEX
------------------------------------------------------------
  Type:  EFI_SECTION_PE32
  Size:  0x00001EE4
------------------------------------------------------------
  Type:  EFI_SECTION_USER_INTERFACE
  Size:  0x0000001E
(...)
```

Appendix A. Linux Kernel Booting Requirements

This appendix contains a brief overview of the requirements to boot a Linux kernel. The Linux kernel documentation is more appropriate to get the full details of the requirements.

Hardware Requirements

The kernel requires the entire RAM to be initialized by the bootloader. The initialization of one serial port is also mandated.

The bootloader must ensure all the interrupts (IRQs and FIQs) are disabled. It is also advised to clear any pending asynchronous external abort.

The MMU must be disabled as well as data caches. The instruction cache may still be enabled.

The kernel expects to run in SVC mode.

Linux Arguments

Parameters can be either passed through an ATAG list or through a Flat Device Tree (FDT) blob.

The ATAG list is a list of entry in a continuous memory region. These entries represent various entities (RAM description, Command line, Video Frame Buffer, etc). Each entity has a specific ATAG type ID.

The ATAG list must contain at least three entries. The first entry must be a `ATAG_CORE` type and the last entry a `ATAG_NONE` type. The list must also have at least one `ATAG_MEM` entry.

The Flat Device Tree blob is a data structure describing the platform hardware configuration. It includes information about CPUs, memory buses and banks, interrupt configuration and peripherals. The device tree data format follows the convention defined within IEEE standard 1275 Open Firmware. (See Linux Kernel Flattened Device Tree [2.])

The Linux kernel takes three arguments:

- The first argument must be 0.
- The second argument is the machine type number. This ID represents the platform on which the Linux kernel is running from.
- The third argument is either the base address of the ATAG list or the FDT blob.

When a FDT blob is passed to the third argument, the Machine Type number must be 0xFFFFFFFF.

Location of the Linux kernel and the ATAG list

Even if there is no strict requirement, it is recommended to place the ATAG in the first 16KB of and the compressed or not Linux Kernel under the first 128MB of RAM.

Appendix B. Glossary

The following table describes some of the terms used in this document.

Table 9 Glossary terms

Term	Description
EDK2	EFI Developer Kit – version 2
FD	Firmware Device
FDT	Flat Device Tree
FV	Firmware Volume
GUID	Globally Unique Identifier
HOB	Hand-Off Block
PEIM	Pre-EFI Initialization Module
Rich OS	Any high-level OS running in non-secure mode, for example Linux
Secure OS	The software containing the secure monitor, which mediates transitions between secure and non-secure processor states.
PCD	Platform Configuration Database
UEFI	Unified Extensible Firmware Interface
WFI	Wait For Interrupt
XIP	eXecute-In-Place