CHAPTER 7

Building Firmware for Quark Processors

"Three things cannot be long hidden: the sun, the moon, and the truth."

—The Buddha

The Intel Quark SoC X1000 is Intel's lowest-power SoC, designed to provide performance and reduce development costs for securely managed Internet of Things endpoint devices. It is initially offered as a single-core, single-threaded microprocessor, making it an ideal solution for low-cost, small form factor, fan less and headless designs.

This chapter will discuss the EDK II infrastructure, Quark, and building a minimal Quark tree with the EDK II. The purpose of this review is to touch upon the salient aspects of the EDK II source construction technology and how it relates to the UEFI PI and UEFI standards, respectively.

This Intel implementation of EDK II at TianoCore demonstrates the possibilities available using the scalable architecture of both the code base and the associated underlying industry standards (see www.uefi.org). The UEFI firmware size for this Intel Galileo EDK II implementation (http://uefidk.intel.com/projects/quark) is 64KB, and given its diminutive size relative to the full Quark EDK II build, it is referred to as "TinyQuark" throughout the rest of this document. TinyQuark boots Yocto Linux (www.yoctoproject.org) on the Intel Galileo board using the onboard flash. You can build this solution from the source code available to download using the following URL. Specifically, the TinyQuark code is at http://uefidk.intel.com/content/get-started-intel-galileo-development-board.

This chapter presents the internal design of TinyQuark, which can be generalized by developers to make their own small-footprint UEFI firmware.

Overview of UEFI and PI

Before getting into TinyQuark, however, the next sections will describe some of the design intent of the EDK II software infrastructure and the association to the UEFI and PI specifications. These specifications describe interoperability between binary and/or source components. Books like *Beyond BIOS* by Vincent Zimmer, Michael Rothman, and Suresh Marisetty (Intel Press, 2011) describe the specifications, but there hasn't been a single place to describe the implementation. The next section is intended to help with that gap.

History of Implementations and Specifications

Starting with the Extensible Firmware Interface (EFI) 0.92 specification in 1998, there has always been a reference EFI implementation. The sample implementations are intended to help clarify some of the design intent of the specification. As shown in the diagram in Figure 7-1, every corresponding specification has had an associated implementation. Historically, these implementations were of the core components that are portable across a broad set of hardware platforms, but the implementations did not include a full platform source tree.

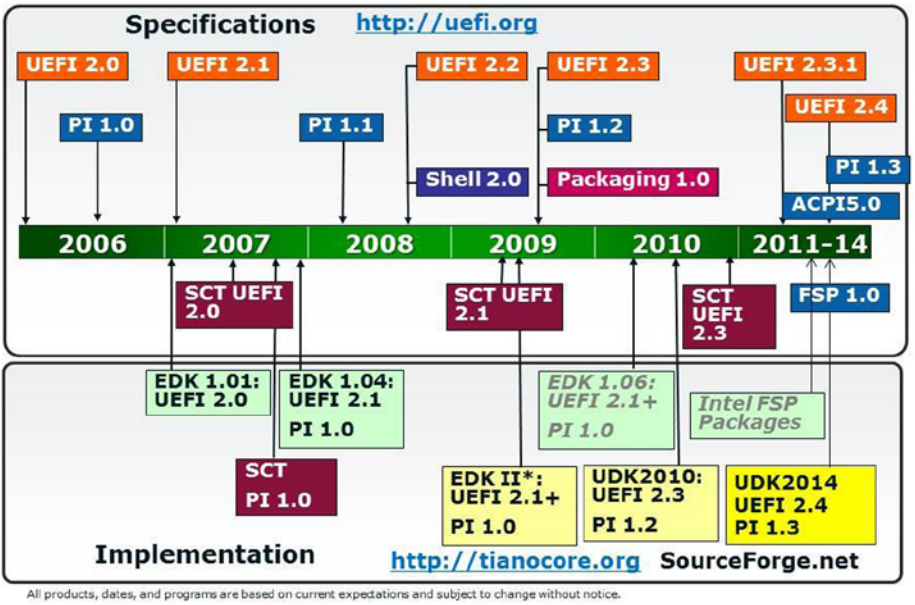

Figure 7-1. *Specification and implementation time line*

The time line shows the original sample implementation, pre-2006. The original EFI Developer Kit (EDK) had challenges in construction since it was a monolithic tree, and the addition of third-party sources or data was ad hoc, as was library support since EDK didn't codify the set of libraries that were usable by the different phases of execution. The introduction of the packaging concept in EDK II, along with PCDs and the base libraries, provided a way to compose source modules from different entities, have reusable sources across many different architectures, and host development environments. Specifically, EDK only supported the Microsoft tool chain, whereas EDK II supports building under Apple OS X, Microsoft Windows, and various Linux distributions. The clean ANSI-C source files and the Python-based build tools of EDK II help contribute to this portability.

Introduction to EDK II Building Blocks

The EFI Development Kit II (EDK II) is an implementation of the UEFI and PI standards. The EDK II is hosted at www.tianocore.org and features many technologies, including an OS-portable build system, ANSI-C code, and GCC/NASM/MASM–based assembly language sources.

In addition, the source technology is decomposed via "packages." The package concept is covered by the UEFI PI packaging specification and is a means by which to segregate binary and sources. The package boundaries are typically driven by business considerations. The packing concept has many interrelated elements for construction, including the DEC, DSC, FDF, and INF files, along with the Platform Configuration Database. The relationship of these elements, details, and some examples of the same are shown next.

Regarding packages, the most prominent packages are listed next, with brief notes about functionality.

PKG: Packaging

Packaging describes the units of decomposition for various technologies. The packaging boundaries may appear somewhat arbitrary at first but are usually motivated by both technology and business criteria. The former includes aggregating a given type of component in one place, such as the generic bus drivers and core elements in the MDE Module Package. The latter includes things like licensing, wherein the package may contain closed-source binaries and sources for a proprietary technology.

MdePkg

The Module Development Environment (MDE) Package (MdePkg) includes files and libraries for Industry Standard Specifications (i.e., UEFI, PI, PCI, USB, SMBIOS, ACPI, SMBIOS, etc.). You can think of the MdePkg, along with the build tools, as the minimum components to build a PEI Module (PEIM), a DXE driver, or a UEFI driver.

The EDK II code is managed on www.sourceforge.net, but there is also a mirror on GitHub. As such, the source code for this package can be found at https://github.com/tianocore/edk2/tree/master/MdePkg.

The important components are the include and library directories. Within the include directory there are industry standard definitions, protocol and PPIs corresponding to the UEFI and UEFI PI specifications, and architecture-specific files. These files need a corresponding white cover industry standard, a public document, or a published UEFI specification in order to reside in the MdePkg.

The library directory, on the other hand, contains a series of library classes. The directories prefixed by "Base" should be use able in the PEI, DXE, UEFI runtime, and UEFI boot services phases. These are the most generic, portable libraries that do not depend upon underlying interfaces. The other libraries are alternately prefixed by the phase of execution, such as "PEI", "SMM", "DXE", "SEC", or "UEFI." These latter terms designate the phase of execution wherein these libraries apply.

Figure 7-2 describes the various phases of UEFI PI execution.

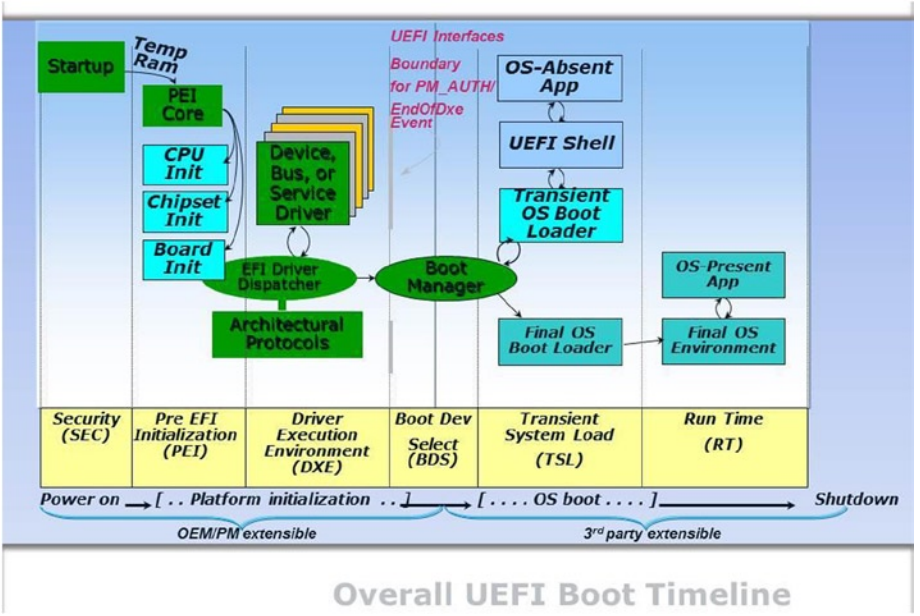

Figure 7-2. *UEFI PI boot flow*

MdeModulePkg

Building upon the MdePkg are implementations of modules, namely the Module Development Environment Modules (MdeModulePkg). These components can be found at https://github.com/tianocore/edk2/tree/master/MdeModulePkg. The PEIMs, DXE drivers, UEFI drivers, and UEFI applications-only definitions from the Industry Standard Specifications are defined in the MdePkg. These components should be portable across a broad class of platforms and CPU bindings, including 32-bit and 64-bit ARM, Intel Itanium, IA32, and X64.

IntelFrameworkPkg

The IntelFrameworkPkg (https://github.com/tianocore/edk2/tree/master/IntelFrameworkPkg) includes files and libraries for those parts of the Intel Platform Innovation Framework for EFI specifications (a.k.a. "Framework") that were not adopted "as is" by the UEFI or PI specifications. These packages provide a bridge between code written against the Framework Specifications (http://www.intel.com/content/www/us/en/architecture-and-technology/unified-extensible-firmware-interface/efi-specifications-general-technology.html) and the subsequent UEFI PI specifications. Some of the interfaces changed between Framework and PI, such as the SMM-CIS; whereas other interfaces only exist in the Framework corpus, such as the Compatibility Support Module (CSM).

IntelFrameworkModulePkg

The IntelFrameworkModulePkg (https://github.com/tianocore/edk2/tree/master/IntelFrameworkModulePkg) contains modules (PEIMs + DXE drivers+ UEFI drivers) that make reference to one or more definitions in the IntelFrameworkPkg. A diagram of these packages is shown in Figure 7-3.

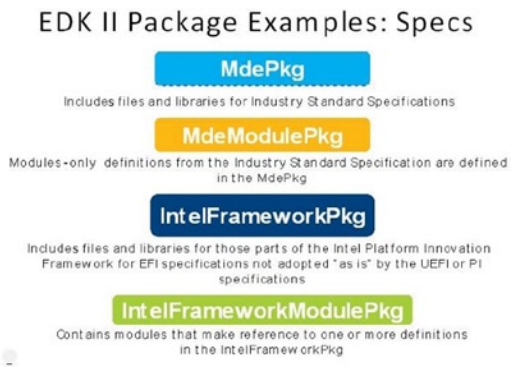

Figure 7-3. *Important packages for EDK II*

Packages

Packages in the EDK II are groups of modules. A package may support one or more drivers, libraries, or combinations thereof. Example packages, in addition to the ones listed earlier, include drivers and applications related to specific hardware, or drivers and applications related to software components, such as the UEFI specification. The MdeModulePkg is an example of the latter, and the former will be discussed in the context of TinyQuark.

Packages can also leverage definitions and elements of other packages. A hardware package should reference the core UEFI packages, such as the MdePkg, for the definitions of standard UEFI protocols and structure.

Packages have related files, such as XML manifest, DSC, and INF files, in addition to the C and/or assembly-language source files. Figure 7-4 shows the packages' relationship with supporting files.

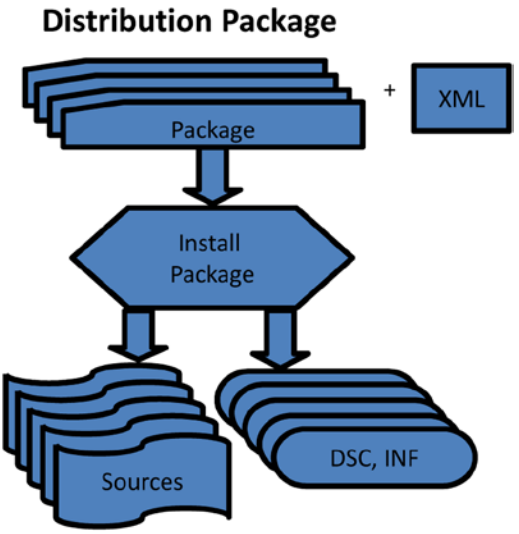

Figure 7-4. *Packages and supporting files*

So the package provides the partitioning of the sources and binaries, but it does not provide for fine-grain control of build options in the actual code artifacts. For that, the PCD comes into play.

PCD: Platform Configuration Database

So what is the platform configuration database goal? First, PCD entries are used for module parameterization; examples include define statements and variables. Among other things, the benefit of PCDs includes reducing the need to edit the source code. Also, there is no need to search for a magic #define statement, like base address registers, for example. These can all be PCD values.

PCDs allow for reusing values across many modules. These fixed-at-build PCDs are very much akin to `#defines`, but herein they are tied into the build system.

Beyond PCDs, the PCD concept can also be used dynamically, namely to store platform information, like the vital product data serial number. You can use dynamic PCDs for setup options and so forth.

PCDs are related to other build files, such as INF, DEC, and DSC, as shown in Figure 7-5.

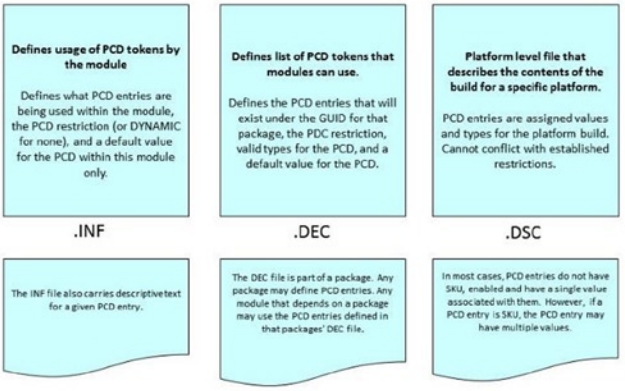

Figure 7-5. *PCD relationship to INF, DEC, and DSC*

There are various types of PCDs, including FeatureFlag, FixedAtBuild, PatchableInModule, and Dynamic.

- *FeatureFlag*: Replaces a switch MACRO to enable/disable a feature (TRUE or FALSE).
- *FixedAtBuild*: Replaces a macro that produced a customizable value. The value of this PCD type is determined at build time and is stored in the code section of a module's PE image.
- *PatchableInModule*: The value is stored in the data section, rather than the code section, of the module's PE image.
- *Dynamic/DyanmicEx/DynamicHii/DynamicVpd*: The value is assigned by one module and is accessed by other modules in execution time.

The PCDs are related to the build process, as follows in Figure 7-6. Specifically, the PCDs are ascertained from the DEC, INF, DSC, and FDF file and included in the autogen. The autogen source files are in turn compiled with the other sources for the resultant driver or PEI module.

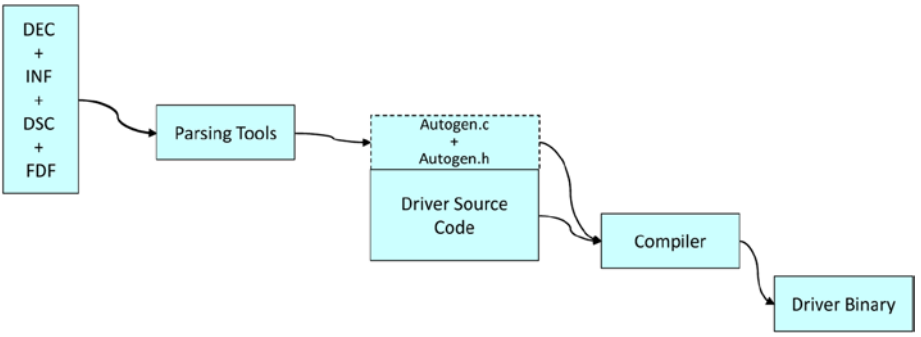

Figure 7-6. *PCDs and build flow*

In addition to extracting the PCDs from metadata files like INF/DEC/DSC, the PCDs can also be used directly in source files. In this case, the relationship of the source and the build is shown in Figure 7-7.

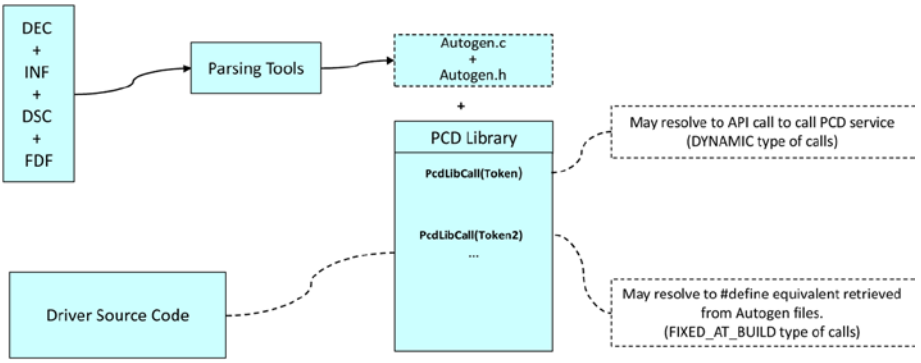

Figure 7-7. *PCDs via build and source construction*

Syntax

Given the background on the PCDs, the following is an example of the declaration of PCDs for a given module.

```
[PcdsFeatureFlag.common] [PcdsFixedAtBuild.IA32] [PcdsFixedAtBuild.X64]
[PcdsFixedAtBuild.IPF] [PcdsFixedAtBuild.EBC]   [PcdsDynamic.IA32]
[PcdsDynamicEx.X64]
```

```
Example of a PCD during DXE
Defined in ICH X Package DEC
  [PcdsDynamic.common]
  gEfiIchTokenSpaceGuid.PcdIchSataPataConfigs|0|UINT8|0x40000016
The Module INF lists which PCDs get accessed
  [Pcd]
  gEfiIchTokenSpaceGuid.PcdIchSataPataConfigs
The Value to use in New Project Package DSC
  [PcdsDynamicDefault.common.DEFAULT]
  gEfiIchTokenSpaceGuid.PcdIchSataPataConfigs|0
```

Here is an example used in the CODE:

```
DXE - Referenced in the DXE code in NewProjectPkg\ SetupDxe\Platform.c

IchSataPataConfigs.Uint8 = PcdGet8(PcdIchSataPataConfigs);
  . . .
  PcdSet8(PcdIchSataPataConfigs, IchSataPataConfigs.Uint8);
```

Finally, the PCDs can show up in the resultant flash image in many ways, including as a Firmware File System file in the flash image. Figure 7-8 shows one possible layout of the PCDs, along with other binary content, such as the UEFI variable data and the vital product data.

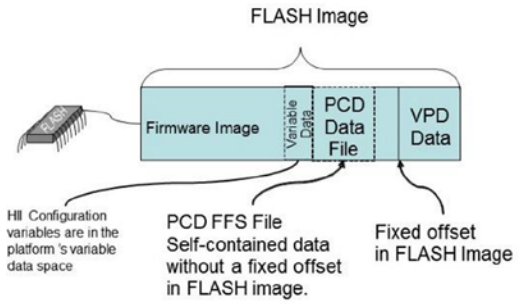

Figure 7-8. *PCDs in a flash image*

Beyond PCDs for parameterizing the build and source files, there is the Platform Declaration File that describes the collection of modules in a build.

DEC: Platform Declaration File

The DEC is the Platform Declaration File (the "D" in DEC is for *declaration*). There is just one DEC file per package. A DEC file is required for EDK II modules using extended INF and extended DSC format files. If you make a new package you must have a DEC file for it.

Syntax

The DEC has a defines section that states what the package is. It gives it a GUID and a name. Every other section described here is optional.

The DEC file may have an includes section stating, "The include directories for this package are as follows:". For example, you might be able to say, "This is my IA64 include and this is my X64 include," and so forth.

In addition, the DEC file also has an optional library class section. It exposes the library classes that are defined in the package.

If you declare any GUIDs in the system, the DEC file has a GUID section. Certain structures have GUIDs defined for them; if that structure is defined in this package, it would be listed here.

The DEC file has a protocol GUID listed for every protocol header file that is in your package. You list the GUID of that protocol in the protocol section. The same is true for PPIs; they are also identified by GUID.

If any module contained in your package defines a new PCD, this is where you look it up.It is possible to reference a PCD from another package, but do not list it here. This location is for new PCDs.

Coincidentally, as soon as you make a new PCD, you must make a new token space GUID, because all the PCDs are defined by a token space GUID, followed by the PCD name. A new token space means you must have a GUID for the token space. So, any new PCDs are also going to have a GUID.

Finally, user extensions are rarely used, but are optionally present. The following is an example DEC file.

```
Example dec
##  @file  ShellPkg.dec
##
[Defines]
  DEC_SPECIFICATION     = 0x00010005
  PACKAGE_NAME          = ShellPkg
  PACKAGE_GUID          = 9FB7587C-93F7-40a7-9...
  PACKAGE_VERSION       = 0.40
[Includes.common]
  Include
[LibraryClasses.common]
  ##  @libraryclass  Provides most Shell APIs. Only available for Shell applications
  ShellLib|Include/Library/ShellLib.h
  ##  @libraryclass  Provides shell internal support Only available for
shell internal commands
  ShellCommandLib|Include/Library/ShellCommandLib.h
  ##  @libraryclass  provides EFI_FILE_HANDLE services used by Shell and ShellLib
  FileHandleLib|Include/Library/FileHandleLib.h
  ## @libraryclass   Allows for a shell application to have a C style entry point
  ShellCEntryLib|Include/Library/ShellCEntryLib.h
  ## @libraryclass   Provides sorting functions
  SortLib|Include/Library/SortLib.h
```

```
  ## @libraryclass   Provides advanced parsing functions
  HandleParsingLib|Include/Library/HandleParsingLib.h
[Guids.common]
  gEfiShellEnvironment2ExtGuid  = {0xd2c18636, 0x40e5, 0x4eb5, {0xa3, 0x1b,
0x36, 0x69, 0x5f, 0xd4, 0x2c, 0x87}}
```

This completes the description of the DEC file. Beyond the DEC file, there also needs to be a DSC.

DSC: Platform Description File

A DSC file must define all libraries, components, and/or modules that will be used by one package. DSC files are a list of the following:

- EDK Component or EDK II Module INF files
- EDK libraries (for EDK Components)
- EDK II Library Class Instance Mappings (for EDK II Modules)
- EDK II PCD Entry Settings

The following is an example of a DSC file for the UEFI Shell Package:

```
#/** @file
# Shell Package
#**/
[Defines]
  PLATFORM_NAME           = Shell
  PLATFORM_GUID           = E1DC9BF8-7013-4c99-9437-...
  PLATFORM_VERSION        = 0.4
  DSC_SPECIFICATION       = 0x00010006
  OUTPUT_DIRECTORY        = Build/Shell
  SUPPORTED_ARCHITECTURES = IA32|IPF|X64|EBC
  BUILD_TARGETS           = DEBUG|RELEASE
  SKUID_IDENTIFIER        = DEFAULT
[LibraryClasses.common]
  UefiApplicationEntryPoint|MdePkg/Library/UefiApplicationEntryPoint/
  UefiApplicationEntryPoint.inf
  UefiBootServicesTableLib|MdePkg/Library/UefiBootServicesTableLib/
  UefiBootServicesTableLib.inf
  DevicePathLib|MdePkg/Library/UefiDevicePathLib/UefiDevicePathLib.inf
  DebugLib|MdePkg/Library/BaseDebugLibNull/BaseDebugLibNull.inf
  PcdLib|MdePkg/Library/BasePcdLibNull/BasePcdLibNull.inf
```

FDF: Flash Description File

The FDF file describes information about the flash part. It has rules for combining binaries built from a DSC file. You can create firmware images and optional ROM images for nearly anything you need.

It is possible to have PCD information used in the definition, as well as in some of the PCDs. The patchable ones will be stored at specific places inside the FV file.

Syntax

The FDF file has a header and a FD section, as well as a number of FV sections. It might have a capsule, a VTF, rules, and an optional ROM section if you are trying to build a PCI option on some user extensions. The following is a Backus-Naur Form (BNF) style notation of the FDF file.

```
FDFfile ::= [<Header>]
                [<Defines>]
        <FD>
        <FV>
       [<Capsule>]
                [<VTF>]
                [<Rules>]
                [<OptionRom>]
       [<UserExtensions>]
```

The FD section definitions for flash devices must be in the FDF file. The FV section definitions for firmware volumes must be in the FDF file.

Build: The EDK II Build Command

The EDK II build system is based on Python. This is one way to achieve the cross-OS build environment portability. The build tools directory in the EDK II tree root hosts the source code for the tool. Schematically, the EDK II build process proceeds as shown in Figure 7-9.

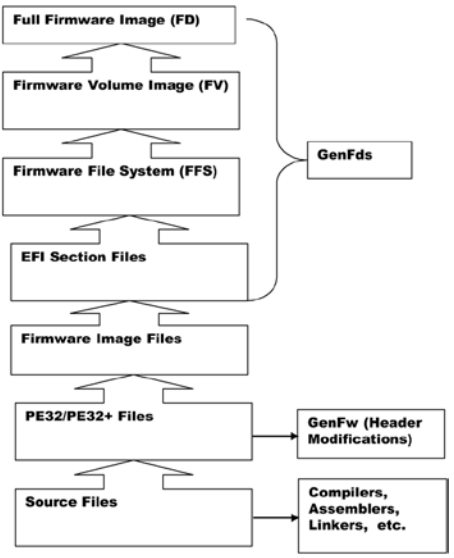

Figure 7-9. *Build flow for binary creation*

Usage of the command is as follows:

```
EDK2 build command
Usage: build.exe [options] [all|fds|genc|genmake|clean|cleanall|cleanlib|
modules|libraries|run]
Options:
  --version         show program's version number and exit
  -h, --help        show this help message and exit
  -a TARGETARCH, --arch=TARGETARCH
                        ARCHS is one of list: IA32, X64, IPF, ARM or EBC,
                        which overrides target.txt's TARGET_ARCH definition
                        To specify more archs, please repeat this option.
  -p PLATFORMFILE, --platform=PLATFORMFILE
                        Build the platform specified by the DSC file name
                        argument, overriding target.txt's ACTIVE_PLATFORM
                        definition.
  -m MODULEFILE, --module=MODULEFILE
                        Build the module specified by the INF file name
argument.
```

To bring all of the metadata and build files together, Figure 7-10 shows the relationship of the source with the FDF, INF, DEC, and DSC files.

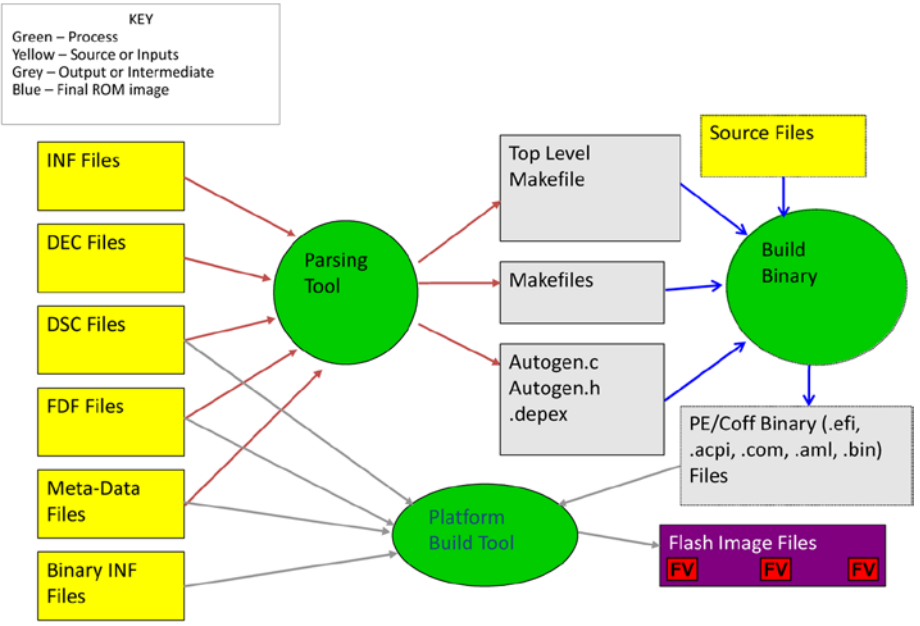

Figure 7-10. *Relationship of all files to the complete build*

INF: INF File

The INF file is updated to define all sources (`.c`, `.h`, `.uni`), libraries, packages, GUIDs, and PCDs used by the module. See the EDK II INF File Specification for more information and examples.

An INF is like a local make-maker file or metadata to inform the build system about which files to use and how to integrate them. The following is an example of an INF file for a serial driver.

```
INF Example SerialDxe
C file
EFI_STATUS
EFIAPI
InitializeSerial (
  IN EFI_HANDLE          ImageHandle,
  IN EFI_SYSTEM_TABLE    *SystemTable
  )
{
  SerialPortInitialize ();
  return
   gBS->InstallMultipleProtocolInterfaces (
        &mSerialIoHandle,
        &gEfiDevicePathProtocolGuid,
```

```
        &mSerialIoDevicePath,
        &gEfiSerialIoProtocolGuid,
        &mSerialIo,
        NULL  );
}

INF file
[Defines]
  INF_VERSION  = 0x00010005
  BASE_NAME    = SerialDxe
  FILE_GUID    = 7507 . . .
MODULE_TYPE  = UEFI_DRIVER
  VERSION_STRING  = 1.0
  ENTRY_POINT  = InitializeSerial
[Sources.common]
  Serial.c
[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
[LibraryClasses]
  PcdLib
  UefiBootServicesTableLib
    . . .
[Protocols]
  gEfiSerialIoProtocolGuid
  gEfiDevicePathProtocolGuid

INFfile ::=[<Header>]
<Defines>
          [<BuildOptions>]
          [<Sources>]
          [<Binaries>]
          [<Guids>]
          [<Protocols>]
          [<Ppis>]
          [<Packages>]
          [<LibraryClasses>]
          [<Pcds>]
         [<UserExtensions>]
    [<Depex>]
```

More Information

All of the preceding build specifications can be found at http://tianocore.sourceforge.net/wiki/EDK_II_Specifications.

Introduction to the EDK II Subset

EDK II is open source implementation for UEFI firmware, which can boot multiple UEFI-aware operating systems. Section 2.6 of the UEFI Specification [UEFI] defines the minimum set of capabilities that UEFI-aware firmware, such as EDK II, must support. We use EDK II BIOS for the Galileo board, which uses the Quark processor.

The Quark build for Galileo is the first fully open-source EDK II–based platform. It leverages the UDK2010 packages, including MdePkg and MdeModulePkg, and adds TinyBootPkg, Ia32FamilyCpuBasePkg, QuarkPlatformPkg, and QuarkSocPkg.

The standard build is 1MB and can be found at `https://downloadcenter.intel.com/Detail_Desc.aspx?DwnldID=23197`.

This full build includes features described in the UEFI 2.4 and PI1.3 specifications on `www.uefi.org`, and include the capsule update, SMM, S3, PCI, recovery, FAT file system support, and UEFI variables.

Now that the overview of the EDK II build system and associated elements have been discussed, details on the internals of TinyQuark will be used to demonstrate this infrastructure in practice. As such, this section provided an overview of Quark and EDK II. Before we begin on the software, a description of the Quark platform itself is in order.

Introduction to Quark

The Intel Quark System-on-Chip (SoC) X1000 is the first product in a new road map of innovative, small core products targeted at rapidly growing areas, ranging from the industrial IoT to wearables. It brings low-power and Intel compute capabilities for thermally constrained, fanless, and headless applications. With its security and manageability features, this SoC is ideally suited for the Internet of Things and for the next wave of cost-effective, intelligent connected devices. An overview of the Quark hardware platform is shown in Figure 7-11.

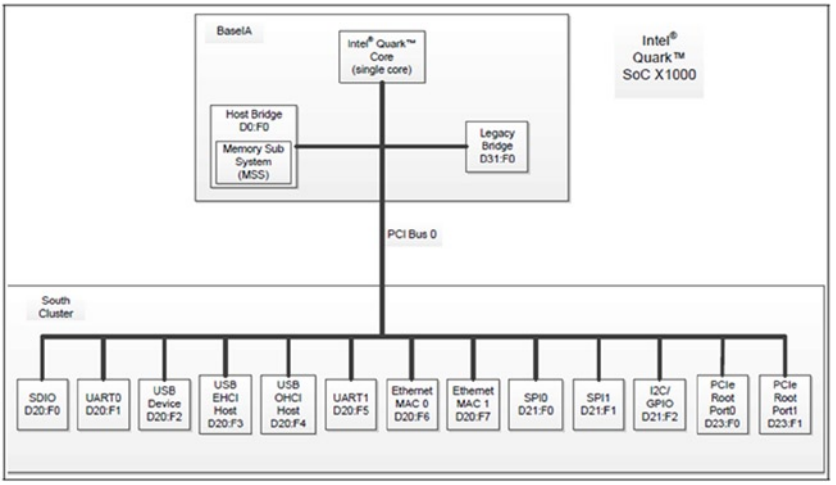

Figure 7-11. *Quark hardware platform*

ROM Flash Image Size Optimization

A set of UEFI/EDK II–related size reduction technologies to make TinyQuark are listed in the following sections. "Write good C code" is *not* mentioned because it is a generic advocacy to have robust, testable code. Instead, this section defines how to apply some logic to provide the minimum feature to the EDK II Quark firmware and still maintain basic UEFI conformance.

In the next several sections, we will discuss the size reduction techniques, one by one. The various techniques used for the EDK II Quark image size reduction are schematically shown in Figure 7-12.

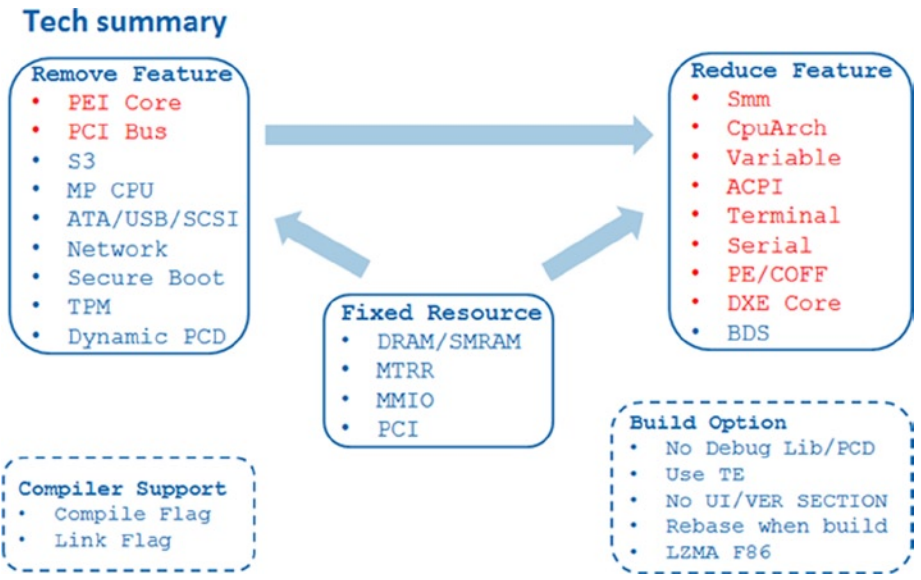

Figure 7-12. *Technology summary for Quark image size reduction*

Fixed Resource

To begin, the support of fixed system resources is important because it will lead to the direct feature set of the platform.

DRAM/SMRAM

If a platform can have fixed DRAM resource, then we can remove the complicated Memory-Type Range Register (MTRR) calculation algorithm in the CPU driver. The MTRR settings can be a table-driven configuration for this design with a known physical memory map. Refer to `QuarkPlatformPkg\Library\QuarkSecLib\SecPlatform.c` and `http://uefidk.intel.com/projects/quark` for more information.

If a platform can have fixed SMRAM, then we can remove the SmmAccess driver that supports the PI SMM interfaces, and just use library to get that value. Refer to `QuarkPlatformPkg\Library\SmmPlatformHookLib`.

If a platform can have fixed-memory mapped I/O (MMIO) and PCI resources, then we can remove PCI driver. The PCI resource setting can be done by a table-driven configuration. Refer to `QuarkPlatformPkg\Pci\PlatformFixedPciResource` found in https://uefidk.com/sites/default/files/Intel_Galileo_TinyQuark_64K.zip.

Remove Features

Not all UEFI features are needed in TinyQuark. Some features can be removed directly, like S3 (ACPI), ATA bus (ATA), USB bus (USB), SCSI bus, network (UEFI), HII (UEFI), UEFI secure boot (UEFI), TPM (TCG), or dynamic PCD(UEFI PI Specification).

Removing some features needs fixed resource support. Today, there are complex resource managers in a full EDK II firmware, like the PCI Bus Driver (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Bus/Pci), which discovers a set of PCI devices and balances the resources. This is an algorithmically complex process that entails significant code logic. For deeply embedded platforms like Quark, wherein the designer elides the ability to add arbitrary devices, a simple driver that declares a fixed set of resources can be used. An example from TinyQuark includes TinyQuark_EDK II\QuarkPlatformPkg\Pci\ PlatformFixedPciResource.

The biggest component removed in TinyQuark is the PEI core. TinyQuark has SEC linked to the DecompressLib, and the code in SEC jumps directly into DxeIpl. DxeIpl links into the memory reference code (MRC), finishes memory initialization, and then jumps into the DxeCore. The DXE core provides the basic UEFI capabilities, such as the boot services.

The detail flow of SEC ➤ DXE is shown in Figure 7-13.

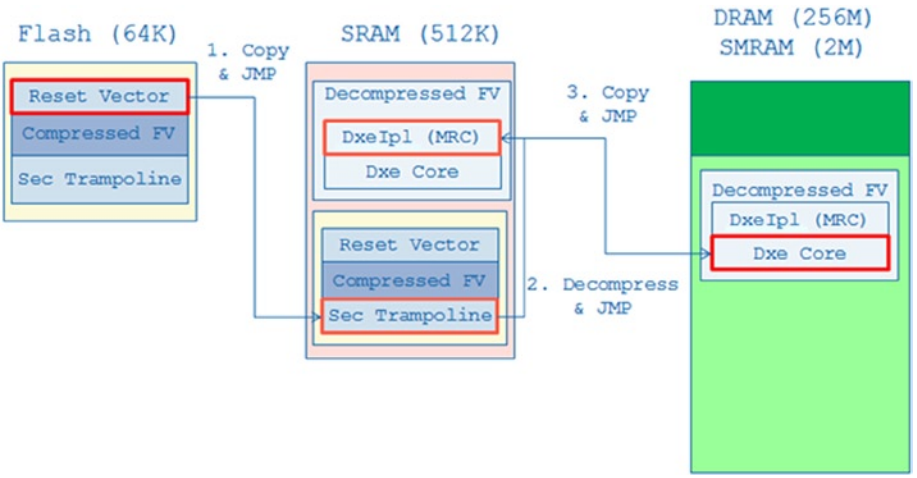

Figure 7-13. *SEC ➤ DXE solution*

Please refer to QuarkPlatformPkg.dsc and QuarkPlatformPkg.fdf for more information on how many of the features are removed:

- For ResetVector module, refer to QuarkPlatformPkg/Cpu/Sec/ResetVector.
- For SecTrampoline module, refer to QuarkPlatformPkg/Cpu/SecTrampoline.
- For DxeIpl module, refer to QuarkPlatformPkg/Cpu/SecCore.

Reduce Features

Even if a component is still needed in TinyQuark, we can make a simplified version. For example:

- *CpuArch*: No SetMemoryAttribute( ) support, because MTRR is fixed and programmed in DxeIpl. Refer to IA32FamilyCpuBasePkg/SimpleCpuArchDxe
- *Variable*: Just expose empty variable driver. Or later we can have some fixed data integrated in this driver. So there is no need to allocate specific Variable FV region. Refer to TinyBootPkg/Universal/Variable/NullVariableRuntimeDxe/NullVariableRuntimeDxe.inf
- *ACPI*: No generic ACPI_TABLE or ACPI_SDT driver. We created AcpiLib to support SetAcpi() only in the AcpiPlatform driver. Refer to TinyBootPkg/Library/AcpiTableLib
- *Terminal*: No driver model. Only supportsPcAnsi. Refer to TinyBootPkg/Universal/Console/SimpleCombinedTerminalDxe
- *Serial*: No driver model. Link SerialPortLib directly. Refer to TinyBootPkg/Universal/Console/SimpleCombinedTerminalDxe
- *PE/COFF lib*: Support PE32 only, no PE32+ or TE. Refer to TinyBootPkg/Library/BasePeCoffLibPe32
- *DXE core*: No GUIDED_SECTION, no Decompression, no FVB, no EBC, no HII, no DebugInfo table. Refer to TinyBootPkg/Core/SimpleDxeCore
- *SMM*: SMM is redesigned. We removed SmmCore (see next for more information).

The current EDK II has the SmmIpl, SmmCore, and SmmCpu drivers. SmmIpl will load SmmCore into SMRAM. SmmCore will load all SMM drivers into SMRAM, including the SmmCpu driver, the SmmPch driver, the SmmPlatform driver, and so forth.

After the SmmCpu driver is loaded, SMBASE rebase happens. Then, the next SMI will trigger into the SmmCpu driver. Next, the SmmCpu driver passes control to the SmmCore, and the SmmCore calls each Smm root handler driver, as registered by the SmmPch driver, to dispatch the respective SMI handler. This full SMM topology can be found in Figure 7-14, in which the upper portion of the figures is SMRAM, or memory protected from ring 0 code by hardware, and the lower portion of the boxes designates normal DRAM.

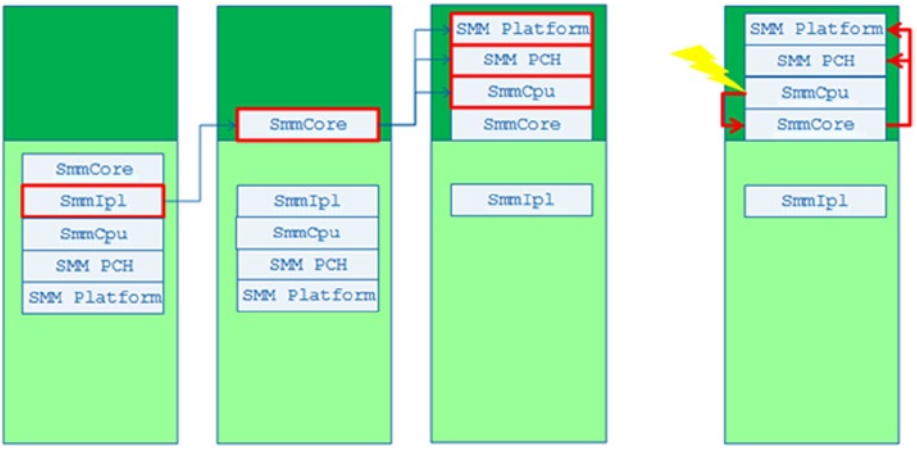

Figure 7-14. *Full SMM core solution*

In this simplified version, we remove the SmmCore. We let the SmmIpl find the SmmCpu driver and load it into SMRAM to do the SMBASE rebase operation. Another activity entails the conversion of all SmmPlatform drivers into a library, and links this library into the SmmCpu drive.

The next SMI will trigger a machine mode switch so that control is passed into the SmmCpu driver, and the SmmCpu driver will call a SmmPlatform library to dispatch the SMI handler. Figure 7-15 provides a diagram of the simplified SMM solution.

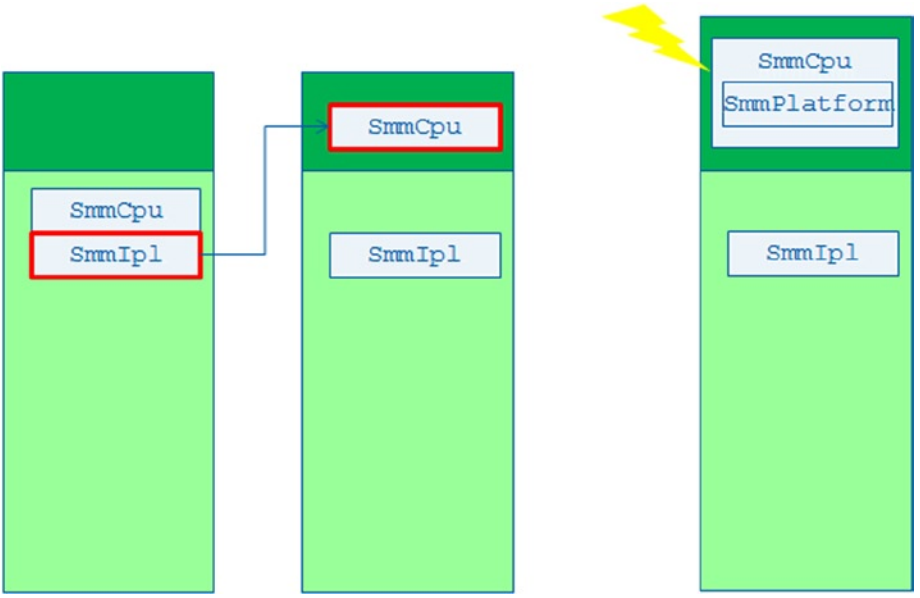

Figure 7-15. *Simplified SMM solution*

Please refer to QuarkPlatformPkg.dsc and QuarkPlatformPkg.fdf for more information on a simplified version driver.

For the SmmIpl/SmmCpu module, refer to IA32FamilyCpuBasePkg/SimplePiSmmCpuDxeSmm.

For SmmPlatform lib, refer to QuarkPlatformPkg/Library/SmmPlatformHookLib.

Compiler Options

In order to support a minimal image size, there are a couple of guiding rules: *do not use the /Zi compiler flag* and *do not use the /DEBUG link flag*. These flags can be added in the debug phase, but do not use them in final release, so that debug information in PE image is removed.

Build Options

Use the NULL Debug lib: DebugLib|MdePkg/Library/BaseDebugLibNull/BaseDebugLibNull.inf. Refer to QuarkPlatformPkg.dsc.

Do not use dynamic PCDs: PcdLib|MdePkg/Library/BasePcdLibNull/BasePcdLibNull.inf. Refer to QuarkPlatformPkg.dsc.

For the Execute-in-Place (XIP) images, use the TE image. Refer to QuarkPlatformPkg.fdf. The following provides an example of the FRF file that provides the TE images.

```
[Rule.Common.SEC]
  FILE SEC = $(NAMED_GUID) RELOCS_STRIPPED {
        TE  TE    Align = 8       $(INF_OUTPUT)/$(MODULE_NAME).efi
        RAW BIN   Align = 16    |.com
}
```

The TE Image format is not applicable for the DXE or UEFI drivers, but it does provide a space savings for images that cannot be compressed by instead have to execute directly from the memory mapping SPI NOR flash, such as SEC and PEI.

Do not use UI/VER section. Refer to QuarkPlatformPkg.fdf.

```
[Rule.Common.DXE_DRIVER]
  FILE DRIVER = $(NAMED_GUID) {
        DXE_DEPEX DXE_DEPEX Optional       $(INF_OUTPUT)/$(MODULE_NAME).depex
        PE32    PE32                      $(INF_OUTPUT)/$(MODULE_NAME).efi
#   UI          STRING="$(MODULE_NAME)" Optional
#   VERSION   STRING="$(INF_VERSION)" Optional BUILD_NUM=$(BUILD_NUMBER)
}
```

Although it doesn't necessarily contribute to the image size reduction, the image must be rebased in order to support Execute-in-Place (XIP). Sometimes the DXE volume can be rebased to the location in the main memory where it will be decompressed. This build-time rebasing omits the time-overhead of the PE/COFF loader to "fix-up" the image during each machine restart. In general, this technique won't work for open platforms because we do not know at firmware build time the size of physical memory or "where" the main DXE firmware volume will be decompressed and copied. Only for deeply embedded platforms with integrated DRAM can such a build-time technique be employed. Additional details from the FDF on how to configure this rebasing follow.

```
[FV.EDK II_BOOT_SLIM]
BlockSize       = 0x1000
FvBaseAddress   = 0x80010000
FvForceRebase   = TRUE
...
```

In order to have compression support, the TinyQuark employs LZMA F86 compression. Refer to QuarkPlatformPkg.fdf.

```
FILE FV_IMAGE = 9E21FD93-9C72-4c15-8C4B-E77F1DB2D791 {
        SECTION GUIDED D42AE6BD-1352-4bfb-909A-CA72A6EAE889 PROCESSING_
REQUIRED = TRUE { # LzmaF86
        SECTION FV_IMAGE = EDK II_BOOT_SLIM
        }
        }
```

Although earlier we mentioned "No GUIDED_SECTION support in the DxeCore", this is a GUIDed section. The reason is that this Guided section will be only supported by DxeIpl to decompress the whole DXE FV. As such, there is no need to have individual drivers in the the DXE FV supported by compression or GUIDed section, thus allowing the DxeCore to remove these support to reduce the size.

Results of the TinyQuark Optimization

After the application of the techniques covered earlier, the TinyQuark ROM demonstrates significant code size savings in the ROM image. Table 7-1 lists some of the metrics.

Table 7-1. *TinyQuark ROM Module Size*

NonCompressed	Size (byt	Category
SecTrampoline	8012	Platforma
ResetVector	1208	Platforma

Summary	
Generic	0
Silicon	0
Platforma	9220

Compressed	Size (byt	Category
DxeIpl (MRC - 17600)	22084	Platforma
SimpleDxeCore	46652	Generic
SimpleCpuArch	5124	Generic
Metronome	1252	Generic
RuntimeDxe	3844	Generic
PlatformVariableRuntime	1764	Generic
ResetSystemRuntime	1572	Generic
SimplePcRtc	3748	Generic
PlatformInitDxe	5732	Platforma
PlatformFixedPciResource	3140	Platforma
QNCInitDxe	5316	Silicon
AcpiPlatform	4100	Platforma
AcpiTable	8850	Platforma
SimpleSmaIpl	6164	Generic
SimplePiSmaCpu	6724	Platforma
IohInitDxe	1268	Silicon
CombindedTerminalDxe	5332	Generic
Legacy8259	2020	Generic
TinyBds	2116	Platforma

Summary	
Generic	77472
Silicon	24184
Platforma	35146

	Final (by	Percentage	Meaning
Generic	**30989**	**48.46%**	UEFI generic API
Silicon	**9674**	**15.13%**	Silicon Initialization
Platforma	**23278**	**36.41%**	Platforma Initialization
total	63941		

In addition to the fine-grain metrics in Table 7-1, Figure 7-16 provides a print message from the firmware itself, essentially showing that the entire firmware volume in the flash part occupies less that 64KB, or the smallest region of today's symmetrically blocked SPI NOR flash parts.

```
FV Space Information
EDKII_BOOT_STAGE1_IMAGE1 [99%Full] 65536 total, 65216 used, 320 free
```

Figure 7-16. *Message from the firmware on its size*

This section described how to reduce ROM flash size, especially as larger flash images lead to larger parts, with an impact to the bill of materials (BOM) and ultimate cost of the platform. Beyond cost savings, the other reason to have a smaller image in flash entails performance; copying a smaller binary from flash to DRAM consumes less of the boot time. To reach the smaller binary, the PEI core was omitted and purpose-built PCI Bus drivers and SMM infrastructure were employed. TinyQuark shows an extreme end of the spectrum, but even a generic platform could benefit from a subset of the techniques described.

RAM Footprint Optimization

In the preceding section, we reduced the amount of real estate consumed in the SPI NOR flash, thus allowing more space for the operating system and other data. But the SPI NOR isn't the only factor to impact the BOM. The other factor that impacts a board cost is volatile memory, or RAM usage. RAM is often noted as DRAM, too, for most systems.

As such, we did an analysis on RAM footprint on the 64K TinyQuark, and we realized that the image in Quark firmware is copied four times during boot, which can be avoided in practice.

1. DxeIpl, will prepare Decompressed FV to DxeCore. This is the first copy.
2. Once DxeCore finds a FV, it will copy FV into RAM. The reason is that DxeCore does not know if it is on flash or DRAM. This flash vs. DRAM independence only makes sense if other optimizations are in place, such as setting the MTRRs to ensure that the firmware volume in flash is cached. If not, there is a significant performance penalty in doing direct flash access for each code fetch.
3. Then in the Driver Dispatch phase, the DxeCore constructs a list for all FFS and SECTIONs there. The section stream for FFS is the third copy for each PE/COFF image.
4. Finally, when DxeCore starts loading the UEFI image, it allocates another memory and uses PeCoff library to load and relocate PE image. This is the fourth copy.

The detailed memory layout before optimization is shown in Figure 7-17.

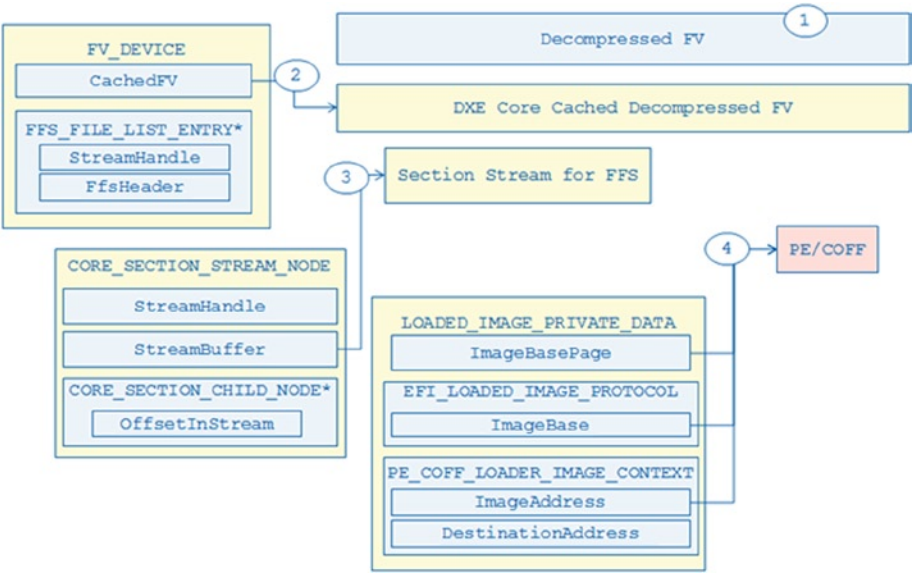

Figure 7-17. *Current DXE core*

Optimization

In order to optimize memory usage, the design technique entails avoiding additional buffer allocations. Instead, the optimization entails reuse of the old buffer as much as possible.

To begin, the Decompressed Fv buffer is the base. This buffer is then reused as the CachedFvcan be a pointer to the original Decompressed Fv buffer. Correspondingly, the section stream for FFS can still point to the original buffer.

Then for the PE/COFF image, we relocate all PE/COFF images at build time. A normal DXE driver or UEFI driver is rebased for its execution address in DRAM so that it can be run directly, without the need for fix-ups. The Dxe Runtime driver is special because it needs to reside in runtime memory (i.e., memory of type EFI_MEMORY_ RUNTIME [UEFI]). So the DxeCore just needs to allocate Runtime Paged memory and do the relocation for the runtime driver.

The detailed memory layout after optimization of memory usage can be found in Figure 7-18.

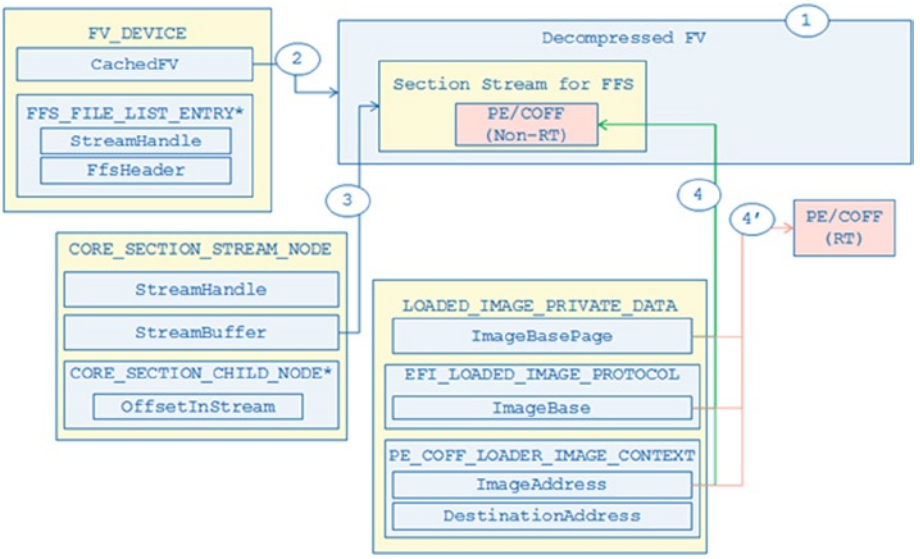

Figure 7-18. *Enhanced simplified DXE core*

Result of Memory Usage Optimization

Before optimization, the memory size used is 760K (624K allocated during boot + 136K decompressed FV). After optimization, the memory size used is 340K (204K allocated during boot + 136K for the decompressed FV).

In addition to the overall metric, a finer-grain accounting of memory usage can be found in Table 7-2.

Table 7-2. *TinyQuark Component RAM Size*

Component (MemType)	Size (K)	Page
1) Decompressed FV	136	34
ACPI Reclaim (9)	48	12
ACPI NVS (0xA)	8	2
Runtime Code (5)	16	4
Runtime Data (6)	12	3
Boot Code (3)	0	0
Boot Data (4)	120	30
-- Stack	32	8
2) Total allocated during boot	204	51
Total memory Used	340	85

This section describes the how to reduce the RAM footprint size. Historically, memory usage in the pre-OS has been considered "free" since the operating system will reclaim most of the resources, but for resource-constrained embedded systems, the DRAM size can be strictly limited. As such, optimizing the boot-time drivers to reuse buffers instead of allocating additional pages allows for an optimum memory footprint during the early phase of execution.

Conclusion

This Intel implementation of the EDK II is a demonstration showing the possibilities available using the scalable architecture of EDK II source code technology and the flexibility of the UEFI Specification. The 64K "TinyQuark" demonstrates the scalability of the EDK II architecture and how to create a UEFI-conformant firmware solution that has a very small flash size and can minimize DRAM usage. This allows for a slim boot load environment for a subsequent UEFI OS, or a slim execution environment for bare-metal execution that can suffice just using UEFI services.