Open Virtual Machine Firmware (OVMF) Status Report

Version 1.0

July 2014 (with updates in August 2014 - January 2015)

Laszlo Ersek lersek@redhat.com
Laura Novich lnovich@redhat.com

100 East Davie Street
Raleigh NC 27601USA
Phone: +1 919 754 4950
Phone: 888 733 4281
Fax: +1 919 800-3804

Keywords: ACPI, boot options, CSM, edk2, firmware, flash, fw_cfg, KVM, memory map, non-volatile variables, OVMF, PCD, QEMU, reset vector, S3, Secure Boot, Smbios, SMM, TianoCore, UEFI, VBE shim, Virtio

Table of Contents

1 Executive Summary

The Unified Extensible Firmware Interface (UEFI) is a specification that defines a software interface between an operating system and platform firmware. UEFI is designed to replace the Basic Input/Output System (BIOS) firmware interface.

Hardware platform vendors have been increasingly adopting the UEFI Specification to govern their boot firmware developments. OVMF (Open Virtual Machine Firmware), a sub-project of Intel's EFI Development Kit II (edk2), enables UEFI support for Ia32 and X64 Virtual Machines. This paper reports on the status of the OVMF project, highlights key features and explains current limitations, gives end-user hints, and examines some areas in-depth.

2 Motivation

OVMF extends the benefits of virtualization to UEFI. Reasons to use OVMF

include:

- Legacy-free guests. A UEFI-based environment eliminates dependencies on legacy address spaces and devices. This is especially beneficial when used with physically assigned devices where the legacy operating mode is troublesome to support. Such as assigned graphics cards operating in legacy-free, non-VGA mode in the virtual machine, for example.
- Future proof guests. The x86 market is steadily moving towards a legacy-free platform and guest operating systems may eventually require a UEFI environment. OVMF provides next generation firmware support for such applications.
- GUID partition tables (GPTs). Master Boot Record (MBR) partition tables represent partition offsets and sizes with 32-bit integers, in units of 512 byte sectors. This limits the addressable portion of the disk to 2 TB. GPT represents logical block addresses with 64 bits.
- Liberating boot loader binaries from residing in a contested and poorly defined space between the partition table and the partitions.
- Support for booting off disks (pass-through physical SCSI devices, for example) with a 4kB physical and logical sector size, which do not have 512-byte block emulation.
- Development and testing of Secure Boot-related features in guest operating systems. Although OVMF's Secure Boot implementation is currently not secure against malicious UEFI drivers, UEFI applications, and guest kernels, trusted guest code that only uses standard UEFI interfaces will find a valid Secure Boot environment under OVMF, which incorporates working key enrollment and signature validation. This

enables development and testing of portable Secure Boot-related guest code.

- Presence of non-volatile UEFI variables. This furthers the development and testing of OS installers, UEFI boot loaders, and unique, dependent guest OS features. For example, an efivars-backed pstore (persistent storage) file system works under Linux.
- Overall, a near production-level UEFI environment for virtual machines when Secure Boot is not required.

3 Scope

UEFI and especially Secure Boot have been topics fraught with controversy and political activism. This paper sidesteps these aspects and strives to focus on use cases, hands-on information for end users, and technical details.

Unless stated otherwise, the expression "X supports Y" means "X is technically compatible with interfaces provided or required by Y". It does not imply support as an activity performed by natural persons or companies.

This paper is based on the status of OVMF at a state no earlier than edk2 SVN revision 16158. The paper concentrates on upstream projects and communities, but occasionally it pans out about OVMF as it is planned to be shipped (as Technical Preview) in Red Hat Enterprise Linux 7.1. In such cases, the margin is noted with the following notation: ♦

Although other VM managers and accelerators are known to support (or plan to support) OVMF to various degrees -- for example, VirtualBox, Xen, BHyVe --, this paper will emphasize OVMF on QEMU/KVM, because the QEMU virtualizer and the KVM hypervisor constitute the foundation of Red Hat's virtualization stack.

The recommended upstream QEMU version is 2.1 or or later. The recommended host Linux kernel (KVM) version is 3.10 or later. The recommended QEMU machine type is "qemu-system-x86_64 -M pc-i440fx-2.1" or later.
The term "TianoCore" is used interchangeably with "edk2" in this paper.

4 Installation and Configuration Instructions

4.1 QEMU Invocation Example

The following commands give a quick overview of installing a UEFI operating system on OVMF, relying only on upstream edk2 and QEMU.

To download the OVMF repository and build OVMF, run the following commands:

```
$ git clone https://github.com/tianocore/edk2.git
$ cd edk2
$ nice OvmfPkg/build.sh -a X64 -n $(getconf _NPROCESSORS_ONLN)
```

*Note that this is an ad-hoc build and it will not include the Secure Boot feature.

The resulting build output file, "OVMF.fd", includes not only the executable firmware code, but the non-volatile variable store as well. For this reason, it is advised to make a VM-specific copy of the build output (the variable store should be private to the virtual machine) using the following command:

```
$ cp Build/OvmfX64/DEBUG_GCC4?/FV/OVMF.fd fedora.flash
```

The variable store and the firmware executable are also available in the build output as separate files entitled: "OVMF_VARS.fd" and "OVMF_CODE.fd". This enables central management and updates of the firmware executable, while each virtual machine can retain its own variable store.

To download a Fedora LiveCD, run the following command:

```
$ wget https://dl.fedoraproject.org/pub/fedora/linux/releases/20/Live/x86_64/Fedora-Live-Xfce-x86_64-20-1.iso
```

To create a 20GB qcow2 virtual disk named fedora.img, run the following command:

```
$ qemu-img create -f qcow2 fedora.img 20G
```

Create a QEMU wrapper script named “fedora.sh”, with the following contents:

```
# Basic virtual machine properties: a recent i440fx machine type, KVM
# acceleration, 2048 MB RAM, two VCPUs.
OPTS="-M pc-i440fx-2.1 -enable-kvm -m 2048 -smp 2"


# The OVMF binary, including the non-volatile variable store, appears as a
# "normal" qemu drive on the host side, and it is exposed to the guest as a
# persistent flash device.
OPTS="$OPTS -drive if=pflash,format=raw,file=fedora.flash"
```

```
# The hard disk is exposed to the guest as a virtio-block device. OVMF has a
# driver stack that supports such a disk. We specify this disk as first boot
# option. OVMF recognizes the boot order specification.
OPTS="$OPTS -drive id=disk0,if=none,format=qcow2,file=fedora.img"
OPTS="$OPTS -device virtio-blk-pci,drive=disk0,bootindex=0"


# The Fedora installer disk appears as an IDE CD-ROM in the guest. This is
# the 2nd boot option.
OPTS="$OPTS -drive id=cd0,if=none,format=raw,readonly"
OPTS="$OPTS,file=Fedora-Live-Xfce-x86_64-20-1.iso"
OPTS="$OPTS -device ide-cd,bus=ide.1,drive=cd0,bootindex=1"


# The following setting enables S3 (suspend to RAM). OVMF supports S3
# suspend/resume.
OPTS="$OPTS -global PIIX4_PM.disable_s3=0"


# OVMF emits a number of info / debug messages to the QEMU debug console, at
# ioport 0x402. We configure qemu so that the debug console is indeed
# available at that ioport. We redirect the host side of the debug console to
# a file.
OPTS="$OPTS -global isa-debugcon.iobase=0x402 -debugcon file:fedora.ovmf.log"


# QEMU accepts various commands and queries from the user on the monitor
# interface. Connect the monitor with the qemu process's standard input and
# output.
OPTS="$OPTS -monitor stdio"


# A USB tablet device in the guest allows for accurate pointer tracking
# between the host and the guest.
OPTS="$OPTS -device piix3-usb-uhci -device usb-tablet"


# Provide the guest with a virtual network card (virtio-net).
#
# Normally, qemu provides the guest with a UEFI-conformant network driver
# from the iPXE project, in the form of a PCI expansion ROM. For this test,
# we disable the expansion ROM and allow OVMF's built-in virtio-net driver to
# take effect.
#
```

```
# On the host side, we use the SLIRP ("user") network backend, which has
# relatively low performance, but it doesn't require extra privileges from
# the user executing qemu.
OPTS="$OPTS -netdev id=net0,type=user"
OPTS="$OPTS -device virtio-net-pci,netdev=net0,romfile="


# A Spice QXL GPU is recommended as the primary VGA-compatible display
# device. It is a full-featured virtual video card, with great operating
# system driver support. OVMF supports it too.
OPTS="$OPTS -device qxl-vga"
qemu-system-x86_64 $OPTS
```

To start the Fedora guest, run the following command:

```
$ sh fedora.sh
```

This command can be used for both installing the guest and afterwards for running the guest.

To verify the basic OVMF network activity:

- Assuming that the non-privileged user running QEMU belongs to group G (where G is a numeric identifier), ensure as root on the host that the group range in file "/proc/sys/net/ipv4/ping_group_range" includes G.
- Boot the guest using the non-privileged user credentials.
- When the TianoCore splash screen appears, press the ESC key.
- Navigate to the Boot Manager | EFI Internal Shell

In the UEFI Shell, issue the following commands, where A.B.C.D is a valid public IPv4 address.

```
Shell> ifconfig -s eth0 dhcp
Shell> ping A.B.C.D
```

Type "quit" at the (QEMU) monitor prompt.

4.2 Installing OVMF Guests with virt-manager and virt-install

4.2.1 Prerequisites

The following steps must be done to insure a proper installation.

1. Assuming OVMF has been installed on the host with the following files:

   - `/usr/share/OVMF/OVMF_CODE.fd`
   - `/usr/share/OVMF/OVMF_VARS.fd`

   locate the "nvram" stanza in "/etc/libvirt/qemu.conf", and edit it as follows:

   ```
   nvram = [ "/usr/share/OVMF/OVMF_CODE.fd:/usr/share/OVMF/OVMF_VARS.fd" ]
   ```

2. Restart libvirtd with your Linux distribution's service management tool; for example:

   ```
   systemctl restart libvirtd
   ```

4.2.2 Installation Using Virtual Machine Manager (virt-manager)

To install using virt-manager, proceed with the guest installation using the wizard as follows:

1. Select File > New Virtual Machine and proceed with the wizard until Step 5 of 5.
2. In Step 5, check the “Customize configuration before install” check box and then click Finish.
3. In the Virtual Machine Details window click Overview and select Firmware, and choose UEFI.
4. Click Apply and begin the installation.

4.2.3 Installation Using virt-install

To install using virt-install run the following command:

```
LDR="loader=/usr/share/OVMF/OVMF_CODE.fd,loader_ro=yes,loader_type=pflash"
```

```
virt-install \
--name fedora20 \
--memory 2048 \
--vcpus 2 \
--os-variant fedora20 \
--boot hd,cdrom,$LDR \
```

```
--disk size=20 \
--disk path=Fedora-Live-Xfce-x86_64-20-1.iso,device=cdrom,bus=scsi
```

4.3 Installation Using a Distribution-independent OVMF Package

A popular, distribution-independent, bleeding-edge OVMF package is available to download here: https://www.kraxel.org/repos/, courtesy of Gerd Hoffmann.

The "edk2.git-ovmf-x64" package provides the following files, among others:

- `/usr/share/edk2.git/ovmf-x64/OVMF_CODE-pure-efi.fd`
- `/usr/share/edk2.git/ovmf-x64/OVMF_VARS-pure-efi.fd`

To install using the edk2.git-ovmf-x64 package:

1. Using the link above, download the specified packages and adapt the pathnames accordingly using the virt-manager and virt-install installation methods as described in Installation Using Virtual Machine Manager (virt-manager) and Installation Using virt-install.
2. It should also be noted that the "edk2.git-ovmf-x64" package seeks to simplify enabling Secure Boot in a virtual machine (strictly for development and testing purposes) as follows:
   a) Boot the virtual machine off the CD-ROM image called `"/usr/share/edk2.git/ovmf-x64/UefiShell.iso";` before or after installing the main guest operating system.
   b) When the UEFI shell appears, issue the following commands:
      - `EnrollDefaultKeys.efi`
      - `reset -s`
   c) The EnrollDefaultKeys.efi utility enrolls the following keys:
      - A static example X.509 certificate (CN=TestCommonName) as Platform Key and first Key Exchange Key.

        The private key matching this certificate has been destroyed (but you shouldn't trust this statement).
      - `"Microsoft Corporation KEK CA 2011"` as second Key Exchange Key `(SHA1: 31:59:0b:fd:89:c9:d7:4e:d0:87:df:ac:66:33:4b:39:31:25:4b:30)`
      - `"Microsoft Windows Production PCA 2011"` as first DB entry `(SHA1: 58:0a:6f:4c:c4:e4:b6:69:b9:eb:dc:1b:2b:3e:08:7b:80:d0:67:8d)`
      - `"Microsoft Corporation UEFI CA 2011"` as second DB entry `(SHA1: 46:de:f6:3b:5c:e6:1c:f8:ba:0d:e2:e6:63:9c:10:19:d0:ed:14:f3)`

These keys suffice to boot released versions of popular Linux distributions (through the shim.efi utility), and Windows 8 and Windows Server 2012 R2, in Secure Boot mode.

5 Supported Guest Operating Systems

Upstream OVMF does not favor some guest operating systems over others for political or ideological reasons. However, some operating systems are harder to obtain and/or technically more difficult to support. The general expectation is that recent UEFI OSes should just work. Please consult the "OvmfPkg/README" file for additional information on this matter.

The following guest OSes were tested with OVMF:

- Red Hat Enterprise Linux 6
- Red Hat Enterprise Linux 7
- Fedora 18
- Fedora 19
- Fedora 20
- Windows Server 2008 R2 SP1
- Windows Server 2012
- Windows 8

Notes about Windows Server 2008 R2 (paraphrasing the "OvmfPkg/README" file):

- QEMU should be started only with one of the following options:
  - "-device qxl-vga"
  - "-device VGA"
- Only one video mode, 1024x768x32, is supported at OS runtime.
- Please refer to the Video Driver section, (OVMF's built-in video driver) for more details on this limitation.
- The qxl-vga video card is recommended ("-device qxl-vga"). After booting the installed guest OS, select the video card in Device Manager, and upgrade the video driver to the QXL XDDM one.

  The QXL XDDM driver can be downloaded from

  <http://www.spice-space.org/download.html>, under Guest | Windows binaries.

  This driver enables additional graphics resolutions at OS runtime, and provides S3 (suspend/resume) capability.

Notes about Windows Server 2012 and Windows 8:

- QEMU should be started with the "-device qxl-vga,revision=4" option (or a later revision, if available).
- The guest OS's built-in video driver inherits the video mode / frame buffer from OVMF. There's no way to change the resolution at OS runtime. For this reason, a platform driver has been developed for OVMF, which allows users to change the preferred video mode in the firmware. Please refer the Platform Driver section for more details.
- It is recommended to upgrade the guest OS's video driver to QXL WDDM using the Device Manager.

  Binaries for the QXL WDDM driver can be found at: http://people.redhat.com/~vrozenfe/qxlwddm (pick a version greater than or equal to 0.6), while the source code resides at: https://github.com/vrozenfe/qxl-dod. This driver enables additional graphics resolutions at OS runtime, and provides S3 (suspend/resume) capability.

6 Compatibility Support Module (CSM)

Collaboration between SeaBIOS and OVMF developers has enabled SeaBIOS to be

built as a Compatibility Support Module, and OVMF to embed and use it. The benefits of a SeaBIOS CSM include:

- The ability to boot legacy (non-UEFI) operating systems, such as legacy Linux systems, Windows 7, OpenBSD 5.2, FreeBSD 8/9, NetBSD, DragonflyBSD, Solaris 10/11.
- Legacy (non-UEFI-compliant) PCI expansion ROMs, such as a VGA BIOS, mapped by QEMU in emulated devices' ROM BARs, are loaded and executed by OVMF. For example, this grants the Windows Server 2008 R2 SP1 guest's native, legacy video driver access to all modes of all QEMU video cards.

Building the CSM target of the SeaBIOS source tree is out of scope for this report. Additionally, upstream OVMF does not enable the CSM by default.

Interested users and developers should look for OVMF's "-D CSM_ENABLE" build-time option, and check out the https://www.kraxel.org/repos/ continuous integration repository, which provides CSM-enabled OVMF builds.

♦ The "OVMF_CODE.fd" firmware image made available for the Red Hat Enterprise Linux 7.1 host does not include a Compatibility Support Module, for the following reasons:

- Virtual machines running officially supported, legacy guest operating systems should just use the standalone SeaBIOS firmware. Firmware selection is flexible in virtualization, refer to Installing OVMF Guests with virt-manager and virt-install for more information.
- The 16-bit thunking interface between OVMF and SeaBIOS is very complex and presents a large debugging and support burden, based on past experience.
- Secure Boot is incompatible with CSM.
- Inter-project dependencies should be minimized whenever possible.
- Using the default QXL video card, the Windows 2008 R2 SP1 guest can be installed with its built-in, legacy video driver. This driver will select the only available video mode: 1024x768x32. After installation, the video driver can be upgraded to the full-featured QXL XDDM driver.

7 Phases of the Boot Process

The PI and UEFI specifications, and Intel's UEFI and EDK II Learning and Development materials, provide ample information on PI and UEFI concepts. The following is an absolutely minimal rough glossary that is included only to help readers who are new to PI and UEFI understand references in later, OVMF-specific sections. This paper defers heavily to the official specifications and the training materials, and frequently quotes them below. A central concept to mention early is the GUID -- globally unique identifier. A GUID is a 128-bit number, written as XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX, where each X stands for a hexadecimal nibble. GUIDs are used to name everything in PI and in UEFI. Programmers introduce new GUIDs with the "uuidgen" utility, and standards bodies standardize well-known services by positing their GUIDs.

The boot process is roughly divided in the following phases:

1. Reset vector code.
2. SEC: Security phase. This phase is the root of firmware integrity.
3. PEI: Pre-EFI Initialization. This phase performs "minimal processor, chipset and platform configuration for the purpose of discovering memory". Modules in PEI collectively save their findings about the platform in a list of HOBs (hand-off blocks).

   When developing PEI code, the Platform Initialization (PI) specification should be consulted.
4. DXE: Driver eXecution Environment, pronounced as "Dixie". This "is the phase where the bulk of the booting occurs: devices are enumerated and initialized, UEFI services are supported, and protocols and drivers are implemented. Also, the tables that create the UEFI interface are produced".

   On the PEI/DXE boundary, the HOBs produced by PEI are consumed. For example, this is how the memory space map is configured initially.
5. BDS: Boot Device Selection. It is "responsible for determining how and where you want to boot the operating system".

   When developing DXE and BDS code, it is mainly the UEFI specification that should be consulted. When speaking about DXE, BDS is frequently considered to be a part of it.

The following concepts are tied to specific boot process phases:

1. PEIM: a PEI Module (pronounced "PIM"). A binary module running in the PEI phase, consuming some PPIs and producing other PPIs, and producing HOBs.
2. PPI: PEIM-to-PEIM interface. A structure of function pointers and related data members that establishes a PEI service, or an instance of a PEI service. PPIs are identified by GUID. An example is EFI_PEI_S3_RESUME2_PPI (6D582DBC-DB85-4514-8FCC-5ADF6227B147).
3. DXE driver: a binary module running in the DXE and BDS phases, consuming some protocols and producing other protocols.
4. Protocol: A structure of function pointers and related data members that establishes a

DXE service, or an instance of a DXE service. Protocols are identified by GUID. An example is EFI_BLOCK_IO_PROTOCOL (964E5B21-6459-11D2-8E39-00A0C969723B).

5. Architectural protocols: a set of standard protocols that are within the foundation of the working of a UEFI system. Each architectural protocol has at most one instance. Architectural protocols are implemented by a subset of DXE drivers. DXE drivers explicitly list the set of protocols (including architectural protocols) that they need to work. UEFI drivers can only be loaded once all architectural protocols have become available during the DXE phase. An example is EFI_VARIABLE_WRITE_ARCH_PROTOCOL (6441F818-6362-4E44-B570-7DBA31DD2453).

8 Project Structure

The term "OVMF" usually denotes the project (community and development effort) that provides and maintains the subject matter UEFI firmware for virtual machines. However, the term is also frequently applied to the firmware binary proper that a virtual machine executes.

OVMF emerges as a compilation of several modules from the edk2 source repository. "edk2" stands for EFI Development Kit II; it is a "modern, feature-rich, cross-platform firmware development environment for the UEFI and PI specifications".

The composition of OVMF is dictated by the following build control files:

- `OvmfPkg/OvmfPkgIa32.dsc`
- `OvmfPkg/OvmfPkgIa32.fdf`
- `OvmfPkg/OvmfPkgIa32X64.dsc`
- `OvmfPkg/OvmfPkgIa32X64.fdf`
- `OvmfPkg/OvmfPkgX64.dsc`
- `OvmfPkg/OvmfPkgX64.fdf`

The format of these files is described in the edk2 DSC and FDF specifications.

In general, the DSC file determines:

- Library instance resolutions for library class requirements presented by the modules to be compiled

- The set of modules to compile

The FDF file roughly determines:

- Which binary modules (compilation output files, precompiled binaries, graphics image files, verbatim binary sections) to include in the firmware image
- How to lay out the firmware image

The Ia32 version of these files builds a firmware where both PEI and DXE phases are 32-bit. The Ia32X64 version builds a firmware where the PEI phase consists of 32-bit modules, and the DXE phase is 64-bit. The X64 flavor builds a purely 64-bit firmware.

As the word size of the DXE phase must match the word size of the runtime OS, a 32-bit DXE can't cooperate with a 64-bit OS, and a 64-bit DXE can't work with a 32-bit OS.

OVMF pulls together modules from across the edk2 tree. For example:

- Common drivers and libraries that are platform independent are usually located under MdeModulePkg and MdePkg
- Common but hardware-specific drivers and libraries that match QEMU's pc-i440fx-* machine type are pulled in from IntelFrameworkModulePkg, PcAtChipsetPkg and UefiCpuPkg
- The platform independent UEFI Shell is built from ShellPkg
- OvmfPkg includes drivers and libraries that are useful for virtual machines and may or may not be specific to QEMU's pc-i440fx-* machine type.

9 Platform Configuration Database (PCD)

Like the Phases of the Boot Process section, this section introduces a concept in very raw form. It is based on the PCD related edk2 specifications, and does not discuss implementation details. The purpose is only to offer you a usable (albeit possibly inaccurate) definition, so that PCDs can be referred to in other sections of this document. It should be noted that when "PCD" is used, it may also refer to a "PCD entry"; which is an entry stored in the Platform Configuration Database.

The Platform Configuration Database is a firmware-wide, name-value store of scalars and buffers, where each entry may be any of the following:

- build-time constant
- run-time dynamic
- theoretically, a middle option: patchable in the firmware file itself, using a dedicated tool. (OVMF does not utilize externally patchable entries.)

A PCD entry is declared in the DEC file of the edk2 top-level Package directory whose modules (drivers and libraries) are the primary consumers of the PCD entry. (See OvmfPkg/OvmfPkg.dec). In short, a PCD in a DEC file exposes a simple customization point. The request for the PCD entry is communicated to the build system by naming the PCD entry in the INF file of the requesting module (application, driver or library). The module may read and write the PCD entry depending on the PCD entry's category.

The database and the PCD entries have the following characteristics:

- Firmware-wide: technically, all modules may access all entries they are interested in, assuming they advertise their interest in their INF files. With careful design, PCDs enable inter-driver propagation of (simple) system configuration. PCDs are available in both PEI and DXE. (UEFI drivers meant to be portable (ie. from third party vendors) are not supposed to use PCDs, since PCDs qualify internal to the specific edk2 firmware in question.)
- Name-value store of scalars and buffers: each PCD has a symbolic name, and a fixed scalar type (UINT16, UINT32 etc.), or VOID* for buffers. Each PCD entry belongs to a namespace, which is a GUID defined in the DEC file.
- A DEC file can permit several categories for a PCD:
    - build-time constant ("FixedAtBuild"),
    - patchable in the firmware image ("PatchableInModule", unused in OVMF),
    - runtime modifiable ("Dynamic").

The platform description file (DSC) of a top-level Package directory may choose the exact category for a given PCD entry that its modules wish to use, and assign a default (or constant) initial value to it. In addition, the edk2 build system too can initialize PCD entries to values that it calculates while laying out the flash device image. Such PCD assignments are described in the FDF control file.

10 Firmware Image Structure

This paper assumes the common X64 choice for both PEI and DXE, and the default DEBUG build target. The OvmfPkg/OvmfPkgX64.fdf file defines the following layout for the flash device image "OVMF.fd":

Description	Compression Type	Size
Non-volatile data storage	open-coded binary data	128 KB
Variable store		56 KB
Event log		4 KB
Working block		4 KB
Spare area		64 KB
FVMAIN_COMPACT	uncompressed	1712 KB
FV Firmware File System file	LZMA compressed	
PEIFV	uncompressed	896 KB
individual PEI modules	uncompressed	
DXEFV	uncompressed	8192 KB
individual DXE modules	uncompressed	
SECFV	uncompressed	208 KB
SEC driver		
reset vector code		

The top-level image consists of three regions (three firmware volumes):

- Non-volatile data store (128 KB),
- Main firmware volume (FVMAIN_COMPACT, 1712 KB),
- Firmware volume containing the reset vector code and the SEC phase code (208 KB).

In total, the OVMF.fd file has size 128 KB + 1712 KB + 208 KB == 2 MB.

10.1 Non-Volatile Data Store

The firmware volume with non-volatile data store (128 KB) has the following internal structure, in blocks of 4 KB:

```
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+   L: event log
LIVE  | varstore                   |L|W| W: working block
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
SPARE |                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The first half of this firmware volume is "live", while the second half is "spare". The spare half is important when the variable driver reclaims unused storage and reorganizes the variable store.

The live half dedicates 14 blocks (56 KB) to the variable store itself. On top of those, one block is set aside for an event log, and one block is used as the working block of the fault tolerant write protocol. Fault tolerant writes are used to recover from an occasional (virtual) power loss during variable updates.

The blocks in this firmware volume are accessed, in stacking order from least abstract to most abstract, by:

- `EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL (provided by OvmfPkg/QemuFlashFvbServicesRuntimeDxe),`
- `EFI_FAULT_TOLERANT_WRITE_PROTOCOL (provided by MdeModulePkg/Universal/FaultTolerantWriteDxe),`
- `Architectural protocols instrumental to the runtime UEFI variable services:`
  - `EFI_VARIABLE_ARCH_PROTOCOL,`
  - `EFI_VARIABLE_WRITE_ARCH_PROTOCOL.`

In a non-secure boot build, the DXE driver that provides these architectural protocols is: MdeModulePkg/Universal/Variable/RuntimeDxe. In a secure boot build, where authenticated variables are available, the DXE driver that offers these protocols is: SecurityPkg/VariableAuthenticated/RuntimeDxe.

10.2 Main Firmware Volume

The main firmware volume (FVMAIN_COMPACT, 1712 KB) embeds further firmware volumes. The outermost layer is a Firmware File System (FFS), carrying a single file. This file holds an LZMA-compressed section, which embeds two firmware volumes: PEIFV (896 KB) with PEIMs, and DXEFV (8192 KB) with DXE and UEFI drivers.

This scheme enables 896 KB PEI drivers and 8192 KB DXE and UEFI drivers to be created. They are then compressed with LZMA in one file. The compressed 1712 KB file saves room on the flash device.

10.3 SECFV Firmware Volume

The SECFV firmware volume (208 KB) is not compressed. It carries the "volume top file" with the reset vector code, to end at 4 GB in guest-physical address space, and the SEC phase driver (OvmfPkg/Sec).

The last 16 bytes of the volume top file (mapped directly under 4 GB) contains a NOP slide and a jump instruction. This is where QEMU starts executing the firmware, using the address 0xFFFF_FFF0. The reset vector and the SEC driver run directly from the flash.

The SEC driver locates FVMAIN_COMPACT in the flash, and decompresses the main firmware image to the RAM. The rest of OVMF (PEI, DXE, BDS phases) runs from RAM.

10.4 How QEMU Maps the Firmware Image

As already mentioned, the OVMF.fd file is mapped by QEMU's "hw/block/pflash_cfi01.c" device just under 4 GB located in the guest-physical address space, as shown in the following command line option:

```
-drive if=pflash,format=raw,file=fedora.flash
```

Refer to the QEMU Invocation Example for more information. Note that pflash is a "ROMD device", which can switch out of "ROMD mode" and back into it.

Namely, in the default ROMD mode, the guest-physical address range backed by the flash device reads and executes as ROM (it does not trap from KVM to QEMU).The first write access in this mode traps to QEMU, and flips the device out of ROMD mode. In non-ROMD mode, the flash chip is programmed by storing CFI (Common Flash Interface) command values at the flash-covered addresses; both reads and writes trap to QEMU, and the flash contents are modified and synchronized to the host-side file. A special CFI command flips the flash device back to ROMD mode.

QEMU implements the above based on the KVM_CAP_READONLY_MEM / KVM_MEM_READONLY KVM features, and OVMF puts it to use in its EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL implementation, under "OvmfPkg/QemuFlashFvbServicesRuntimeDxe".

IMPORTANT

Never pass OVMF.fd to QEMU with the -bios option enabled. This option maps the firmware image as ROM into the guest's address space, and forces OVMF to emulate non-volatile variables with a fallback driver that is bound to have insufficient and confusing semantics.

The 128 KB firmware volume with the variable store, discussed under Non-Volatile Data Store, is also built as a separate host-side file, named "OVMF_VARS.fd". The "rest" is

built into a third file, "OVMF_CODE.fd", which is only 1920 KB in size. The

variable store is mapped into its usual location, at 4 GB - 2 MB = 0xFFE0_0000,

through the following QEMU options:

```
-drive if=pflash,format=raw,readonly,file=OVMF_CODE.fd \
-drive if=pflash,format=raw,file=fedora.varstore.fd
```

This way qemu configures two flash chips consecutively, with start addresses

growing downwards, which is transparent to OVMF.

♦ Red Hat Enterprise Linux 7.1 ships a Secure Boot-enabled, X64, DEBUG firmware only. Furthermore, only the split files ("OVMF_VARS.fd" and "OVMF_CODE.fd") are available.

11 S3 (Suspend to RAM and Resume)

As noted in QEMU Invocation Example, the `-global PIIX4_PM.disable_s3=0` command line option tells QEMU and OVMF if the user would like to enable S3 support. (This is corresponds to the /domain/pm/suspend-to-mem/@enabled libvirt domain XML attribute.)

Implementing / orchestrating S3 was a considerable community effort in OVMF. A detailed description exceeds the scope of this report; as such only a few points will be pointed out.

- S3-related PPIs and protocols are well documented in the PI specification.
- Edk2 contains most modules that are needed to implement S3 on a given platform. One abstraction that is central to the porting / extending of the S3-related modules to a new platform is the LockBox library interface, which a specific platform can fill in by implementing its own LockBox library instance.

  The LockBox library provides a privileged name-value store (to be addressed by GUIDs). The privilege separation stretches between the firmware and the operating system. That is, the S3-related machinery of the firmware saves some items in the LockBox securely, under well-known GUIDs, before booting the operating system. During resume (which is a form of warm reset), the firmware is activated again, and retrieves items from the LockBox. Before jumping to the OS's resume vector, the LockBox is secured again. This paper will return to this later and will separately discuss SMRAM and SMM in more detail.
- During resume, the DXE and later phases are never reached; only the reset vector, and the SEC and PEI phases of the firmware run. The platform is supposed to detect a resume in progress during PEI, and to store that fact in the BootMode field of the Phase Handoff Information Table (PHIT) HOB. OVMF keys this off the CMOS, see OvmfPkg/PlatformPei.

  At the end of PEI, the DXE IPL PEIM (Initial Program Load PEI Module, see MdeModulePkg/Core/DxeIplPeim) examines the Boot Mode, and if it says "S3 resume in progress", then the IPL branches to the PEIM that exports EFI_PEI_S3_RESUME2_PPI (provided by UefiCpuPkg/Universal/Acpi/S3Resume2Pei) rather than loading the DXE core.

  S3Resume2Pei executes the technical steps of the resumption, relying on the contents of the LockBox.

- When DXE runs during the first boot (or after a normal platform reset), the hardware drivers in the DXE phase are encouraged to "stash" their hardware configuration steps (accessing PCI configuration space, I/O ports, memory mapped addresses, and so on) in a centrally maintained, so called "S3 boot script". Hardware accesses are represented with opcodes of a special binary script language.

  This boot script is to be replayed during resume, by S3Resume2Pei. The general goal is to quickly bring back hardware devices which were powered off during suspend, to their original after-first-boot state.

  At the moment, OVMF saves only one opcode in the S3 resume boot script: an INFORMATION opcode, with contents 0xDEADBEEF (in network byte order). The consensus between Linux developers seems to be that boot firmware is only responsible for restoring basic chipset state, which OVMF does during PEI independently of S3 vs. normal reset. (One example is the power management registers of the i440fx chipset.) Device and peripheral state is the responsibility of the runtime operating system.

  Although an experimental OVMF S3 boot script was at one point captured for the virtual Cirrus VGA card, such a boot script cannot follow video, for example, in some mode changes effected by the OS. Hence, the operating system can never avoid restoring device state, and most Linux display drivers (such as stdvga and QXL) already cover S3 resume fully.
  The XDDM and WDDM driver models used under Windows seem to recognize this notion of runtime OS responsibility as well. (See the list of operating systems supported by OVMF in a separate section.)

The S3 suspend/resume data flow in OVMF is included in brief as follows:

```
(a) BdsLibBootViaBootOption()
      EFI_ACPI_S3_SAVE_PROTOCOL [AcpiS3SaveDxe]
            - saves ACPI S3 Context to LockBox ----------------------+
              (including FACS address -- FACS ACPI table             |
              contains OS waking vector)                             |
                                                                     |
            - prepares boot script:                                  |
              EFI_S3_SAVE_STATE_PROTOCOL.Write() [S3SaveStateDxe]    |
              S3BootScriptLib [PiDxeS3BootScriptLib]                 |
            - opcodes & arguments are saved in NVS -------+          |
                                                          |          |
            - issues a notification by installing         |          |
              EFI_DXE_SMM_READY_TO_LOCK_PROTOCOL          |          |
                                                          |          |
(b) EFI_S3_SAVE_STATE_PROTOCOL [S3SaveStateDxe]           |          |
      S3BootScriptLib [PiDxeS3BootScriptLib]              |          |
            - closes script with special opcode     <-----+          |
            - script is available in non-volatile memory             |
              via PcdS3BootScriptTablePrivateDataPtr----+            |
                                                        |            |
      BootScriptExecutorDxe                             |            |
      S3BootScriptLib [PiDxeS3BootScriptLib]            |            |
      - Knows about boot script location by   <---------+            |
        synchronizing with the other library                         |
        instance via                                                 |
        PcdS3BootScriptTablePrivateDataPtr.                          |
      - Copies relocated image of itself to                          |
        reserved memory. -----------------------------------+        |
      - Saved image contains pointer to boot script.  ------|-----+  |
                                                            |     |  |
      Runtime:                                              |     |  |
                                                            |     |  |
(c) OS is booted, writes OS waking vector to FACS,          |     |  |
      suspends machine                                      |     |  |
                                                            |     |  |
S3 Resume (PEI):                                            |     |  |
                                                            |     |  |
(d) PlatformPei sets S3 Boot Mode based on CMOS             |     |  |
                                                            |     |  |
(e) DXE core is skipped and EFI_PEI_S3_RESUME2 is           |     |  |
      called as last step of PEI                            |     |  |
                                                            |     |  |
(f) S3Resume2Pei retrieves from LockBox:                    |     |  |
      - ACPI S3 Context (path to FACS)    <-----------------|-----|--+
                                          |                 |     |
                                          +-----------------|-----|--+
      - Boot Script Executor Image   <----------------------+     |  |
                                                                  |  |
(g) BootScriptExecutorDxe                                         |  |
      S3BootScriptLib [PiDxeS3BootScriptLib]                      |  |
      - executes boot script         <----------------------------+  |
                                                                     |
(h) OS waking vector available from ACPI S3 Context / FACS  <--------+
      is called
```

12 A Comprehensive Memory Map of OVMF

The following section gives a detailed analysis of the memory ranges below 4 GB

that OVMF statically uses. In the rightmost column, the PCD entry is identified by which the source refers to the address or size in question.

The flash-covered range has been discussed previously in Firmware Image Structure, so it is only included here to complete the picture. Due to the fact that this range is always backed by a memory mapped device (and never RAM), it is unaffected by S3 (suspend to RAM and resume).

```
+--------------------------+ 4194304 KB
|                          |
|           SECFV          | size: 208 KB
|                          |
+--------------------------+ 4194096 KB
|                          |
|     FVMAIN_COMPACT       | size: 1712 KB
|                          |
+--------------------------+ 4192384 KB
|                          |
|     variable store       | size: 64 KB PcdFlashNvStorageFtwSpareSize
|     spare area           |
|                          |
+--------------------------+ 4192320 KB  PcdOvmfFlashNvStorageFtwSpareBase
|                          |
|     FTW working block    | size: 4 KB  PcdFlashNvStorageFtwWorkingSize
|                          |
+--------------------------+ 4192316 KB  PcdOvmfFlashNvStorageFtwWorkingBase
|                          |
|     Event log of         | size: 4 KB  PcdOvmfFlashNvStorageEventLogSize
|     non-volatile storage |
|                          |
+--------------------------+ 4192312 KB  PcdOvmfFlashNvStorageEventLogBase
|                          |
|     variable store       | size: 56 KB  PcdFlashNvStorageVariableSize
|                          |
+--------------------------+ 4192256 KB  PcdOvmfFlashNvStorageVariableBase
```

The flash-mapped image of OVMF.fd covers the entire structure above (2048 KB).

When using the split files, the address 4192384 KB (PcdOvmfFlashNvStorageFtwSpareBase + PcdFlashNvStorageFtwSpareSize) is the boundary between the mapped images of OVMF_VARS.fd (56 KB + 4 KB + 4 KB + 64 KB = 128 KB) and OVMF_CODE.fd (1712 KB + 208 KB = 1920 KB).

S3 (suspend to RAM and resume) complicates matters regarding RAM that is statically used by OVMF. Many ranges have been introduced only to support S3, hence for all ranges below, the following questions will be audited:

1. How and when a given range is initialized after the VM's first boot?
2. How is it protected from memory allocations during DXE?
3. How it is protected from the OS?
4. How it is accessed on the S3 resume path?
5. How it is accessed on the warm reset path?

It should be noted that in these questions, the term "protected" is defined as protection against inadvertent reallocations and overwrites by co-operating DXE and OS modules. It does not imply security against malicious code.

```
+--------------------------+ 17408 KB
|DXEFV from FVMAIN_COMPACT | size: 8192 KB PcdOvmfDxeMemFvSize
|  decompressed firmware   |
| volume with DXE modules  |
|                          |
+--------------------------+ 9216 KB      PcdOvmfDxeMemFvBase
|                          |
|PEIFV from FVMAIN_COMPACT | size: 896 KB  PcdOvmfPeiMemFvSize
|  decompressed firmware   |
| volume with PEI modules  |
+--------------------------+ 8320 KB      PcdOvmfPeiMemFvBase
|                          |
| permanent PEI memory for | size: 32 KB   PcdS3AcpiReservedMemorySize
|   the S3 resume path     |
+--------------------------+ 8288 KB      PcdS3AcpiReservedMemoryBase
|                          |
|  temporary SEC/PEI heap  | size: 32 KB   PcdOvmfSecPeiTempRamSize
|      and stack           |
+--------------------------+ 8256 KB      PcdOvmfSecPeiTempRamBase
|                          |
|          unused          | size: 32 KB
|                          |
+--------------------------+ 8224 KB
|                          |
|      SEC's table of      | size: 4 KB    PcdGuidedExtractHandlerTableSize
| GUIDed section handlers  |
+--------------------------+ 8220 KB      PcdGuidedExtractHandlerTableAddress
|                          |
|       LockBox storage    | size: 4 KB    PcdOvmfLockBoxStorageSize
|                          |
+--------------------------+ 8216 KB      PcdOvmfLockBoxStorageBase
|                          |
| early page tables on X64 | size: 24 KB   PcdOvmfSecPageTablesSize
|                          |
+--------------------------+ 8192 KB      PcdOvmfSecPageTablesBase
```

12.1 Early Page Tables Considerations

Early page tables on X64 architecture raise the following questions:

1. When and how it is initialized after first boot of the VM?
   The range is filled in during the SEC phase [OvmfPkg/ResetVector/Ia32/PageTables64.asm]. The CR3 register is verified against the base address in SecCoreStartupWithStack() [OvmfPkg/Sec/SecMain.c].
2. How it is protected from memory allocations during DXE phase?
   If S3 was enabled on the QEMU command line (see "-global PIIX4_PM.disable_s3=0" earlier), then InitializeRamRegions() [OvmfPkg/PlatformPei/MemDetect.c] protects the range with an AcpiNVS memory allocation HOB, in PEI.
   If S3 was disabled, then this range is not protected. DXE's own page tables are first built while still in PEI (see HandOffToDxeCore() [MdeModulePkg/Core/DxeIplPeim/X64/DxeLoadFunc.c]). The page tables are located in permanent PEI memory. After CR3 is switched over to them (which occurs before jumping to the DXE core entry point), we don't have to preserve the initial tables.
3. How it is protected from the OS?
   If S3 is enabled, then as previously mentioned in question 2, it is reserved from the OS as well. If S3 is disabled, then the range needs no protection.
4. How it is accessed on the S3 resume path?
   It is rewritten same as in question 1, which is fine because Question 3 reserved it.
5. How it is accessed on the warm reset path?
   It is rewritten in the same way as in Question 1.

12.2 LockBox Storage Considerations

LockBox storage raises the following questions:

1. When and how it is initialized after first boot of the VM?
InitializeRamRegions() [OvmfPkg/PlatformPei/MemDetect.c] zeroes out the area during PEI. This is correct, but not strictly necessary, since the area is zero-filled on first boot. The LockBox signature of the area is filled in by the PEI module or DXE driver that has been linked against OVMF's LockBoxLib and runs first. The signature is written in LockBoxLibInitialize()[OvmfPkg/Library/LockBoxLib/LockBoxLib.c]. Any module calling SaveLockBox() [OvmfPkg/Library/LockBoxLib/LockBoxLib.c] will co-populate this area.

2. How it is protected from memory allocations during DXE?
If S3 is enabled, then InitializeRamRegions() [OvmfPkg/PlatformPei/MemDetect.c] protects the range as AcpiNVS. Otherwise, the range is covered with a BootServicesData memory allocation HOB.

3. How it is protected from the OS?
If S3 is enabled, then question two protects it sufficiently. Otherwise the range requires no runtime protection, and the BootServicesData allocation type from question two ensures that the range will be released to the OS.

4. How it is accessed on the S3 resume path?
The S3 Resume PEIM restores data from the LockBox, which has been correctly protected in question three.

5. How it is accessed on the warm reset path?
InitializeRamRegions() [OvmfPkg/PlatformPei/MemDetect.c] zeroes out the range during PEI, effectively emptying the LockBox. Modules will re-populate the LockBox as described in question one.

12.3 SEC's Table of GUIDed Section Handlers

The SEC's table of GUIDed section handlers, raises the following questions:

1. When and how it is initialized after first boot of the VM?
   The following two library instances are linked into SecMain:
    - IntelFrameworkModulePkg/Library/LzmaCustomDecompressLib,

- MdePkg/Library/BaseExtractGuidedSectionLib.

The first library registers its LZMA decompressor plugin (called a "section handler") by calling the second library:
LzmaDecompressLibConstructor() [GuidedSectionExtraction.c]
ExtractGuidedSectionRegisterHandlers() [BaseExtractGuidedSectionLib.c]

The second library maintains its table of registered "section handlers", to be indexed by GUID, in this fixed memory area, independently of S3 enablement. (The decompression of FVMAIN_COMPACT's FFS file section that contains the PEIFV and DXEFV firmware volumes occurs with the LZMA decompressor registered above. See PEIFV Considerations and DXEVF Considerations below.)

2. How it is protected from memory allocations during DXE?
   There is no need to protect this area from DXE: because nothing else in OVMF links against BaseExtractGuidedSectionLib, the area loses its significance as soon as OVMF progresses from SEC to PEI, therefore DXE is allowed to overwrite the region.
3. How it is protected from the OS?
   When S3 is enabled, we cover the range with an AcpiNVS memory allocation HOB in InitializeRamRegions(). When S3 is disabled, the range is not protected.
4. How it is accessed on the S3 resume path?
   The table of registered section handlers is again managed by BaseExtractGuidedSectionLib linked into SecMain exclusively. Section handler registrations update the table in-place (based on GUID matches).
5. How it is accessed on the warm reset path?
   If S3 is enabled, then the OS won't damage the table (due to question three), thus see question four. If S3 is disabled, then the OS has most probably overwritten the range with its own data, hence question one -- complete reinitialization -- will come into effect, based on the table signature check in BaseExtractGuidedSectionLib.

12.4 Permanent PEI Memory Considerations

Permanent PEI memory for the S3 resume path, raises the following questions:

1. When and how it is initialized after first boot of the VM?
   No particular initialization or use.
2. How it is protected from memory allocations during DXE?
   There is no need to protect this area during DXE.
3. How it is protected from the OS?
   When S3 is enabled, InitializeRamRegions()[OvmfPkg/PlatformPei/MemDetect.c] makes sure the OS doesn’t use this range, by covering the range with an AcpiNVS memory allocation HOB. When S3 is disabled, the range needs no protection.
4. How it is accessed on the S3 resume path?
   PublishPeiMemory() installs the range as permanent RAM for PEI. The range will serve as stack and will satisfy allocation requests during the rest of PEI. OS data won't overlap due to question three.
5. How it is accessed on the warm reset path?
   Same as question one.

12.5 Temporary SEC/PEI Heap and Stack Considerations

The temporary SEC/PEI heap and stack range, raises the following questions:

1. When and how it is initialized after the VM’s first boot?
   The range is configured in [OvmfPkg/Sec/X64/SecEntry.S] and SecCoreStartupWithStack() [OvmfPkg/Sec/SecMain.c]. The stack half is read and written by the CPU transparently. The heap half is used for memory allocations during PEI. Data is migrated out (to permanent PEI stack & memory) in (or soon after) PublishPeiMemory() [OvmfPkg/PlatformPei/MemDetect.c].
2. How it is protected from memory allocations during DXE?
   It is not necessary to protect this range during DXE because its use ends still in PEI.
3. How it is protected from the OS?
   If S3 is enabled, then InitializeRamRegions() [OvmfPkg/PlatformPei/MemDetect.c] reserves it as AcpiNVS. If S3 is disabled, then the range doesn't require protection.
4. How it is accessed on the S3 resume path?
   Same as described in question one, except the target area of the migration triggered by PublishPeiMemory() [OvmfPkg/PlatformPei/MemDetect.c] is different – see

Permanent PEI Memory Considerations for the S3 resume path.

5. How is it accessed on the warm reset path?
   Same as in question one, however in this case the stack and heap halves both may contain garbage, but even if they do, it doesn't matter.

12.6 PEIFV Considerations

PEIFV -- decompressed firmware volume with PEI modules, raises the following questions:

1. When and how it is initialized after first boot of the VM?
   DecompressMemFvs() [OvmfPkg/Sec/SecMain.c] populates the area, by decompressing the flash-mapped FVMAIN_COMPACT volume's contents. (Refer to Firmware Image Structure.)

2. How it is protected from memory allocations during DXE?
   When S3 is disabled, PeiFvInitialization() [OvmfPkg/PlatformPei/Fv.c] covers the range with a BootServicesData memory allocation HOB. When S3 is enabled, the same is coverage is ensured, just with the stronger AcpiNVS memory allocation type.

3. How it is protected from the OS?
   When S3 is disabled, it is not necessary to keep the range from the OS. Otherwise the AcpiNVS type allocation from question two provides coverage.

4. How it is accessed on the S3 resume path?
   Rather than decompressing it again from FVMAIN_COMPACT, GetS3ResumePeiFv() [OvmfPkg/Sec/SecMain.c] reuses the protected area for parsing / execution from question three.

5. How it is accessed on the warm reset path?
   Same as question one.

12.7 DXEVF Considerations

DXEFV -- decompressed firmware volume with DXE modules, raises the following questions:

1. When and how it is initialized after first boot of the VM? Same as question one in PEIFV Considerations.

2. How it is protected from memory allocations during DXE? PeiFvInitialization()

[OvmfPkg/PlatformPei/Fv.c] covers the range with a BootServicesData memory allocation HOB.

3. How it is protected from the OS?
   The OS is allowed to release and reuse this range.
4. How it is accessed on the S3 resume path?
   It is not accessed. DXE never runs during S3 resume.
5. How it is accessed on the warm reset path?
   Same as in question one.

13 Known Secure Boot Limitations

Under chapter Motivation it is mentioned that OVMF's Secure Boot implementation is not suitable for production use yet, and that it's only good for development and testing of standards-conformant, non-malicious guest code (UEFI and operating system alike).

Now that we've examined the persistent flash device, the workings of S3, and the memory map, we can discuss two currently known shortcomings of OVMF's Secure Boot that in fact make it insecure. (Clearly problems other than these two might exist; the set of issues considered here is not meant to be exhaustive.)

One trait of Secure Boot is tamper-evidence. Secure Boot may not prevent malicious modification of software components (for example, operating system drivers), but by being the root of integrity on a platform, it can catch (or indirectly contribute to catching) unauthorized changes, by way of signature and certificate checks at the earliest phases of boot.

If an attacker can tamper with key material stored in authenticated and/or boot-time only persistent variables (for example, PK, KEK, db, dbt, dbx), then the intended security of this scheme is compromised. The UEFI 2.4A specification says in section 28.3.4:

“Platform Keys:

> *The public key must be stored in non-volatile storage which is tamper and delete resistant.*

Key Exchange Keys:

The public key must be stored in non-volatile storage which is tamper resistant."[1]

In section 28.6.1 it says:

> *"The signature database variables db, dbt, and dbx must be stored in tamper-resistant non-volatile storage."*[2]

The known Secure Boot Limitations are:

1. The combination of QEMU, KVM, and OVMF does not provide this kind of resistance. The variable store in the emulated flash chip is directly accessible to, and reprogrammable by, UEFI drivers, applications, and operating systems.

One way to address these issues is SMM and SMRAM (System Management Mode and System Management RAM).

During boot and resume, the firmware can enter and leave SMM and access SMRAM. Before the DXE phase completes, and control is transferred to the BDS phase (when third party UEFI drivers and applications can be loaded, and an operating system can be loaded), SMRAM is locked in hardware, and subsequent modules cannot access it directly. (See EFI_DXE_SMM_READY_TO_LOCK_PROTOCOL.)

Once SMRAM has been locked, UEFI drivers and the operating system can enter SMM by raising a System Management Interrupt (SMI), at which point trusted code (part of the platform firmware) takes control. SMRAM is also unlocked by platform reset, at which point the boot firmware takes control again.

14 Variable Store and LockBox in SMRAM

Edk2 provides almost all components to implement the variable store and the LockBox in SMRAM. In this section we summarize ideas for utilizing those facilities. The SMRAM and SMM infrastructure in edk2 is built up as follows:

1. The platform hardware provides SMM / SMI / SMRAM.
   QEMU/KVM doesn't support these features currently and should implement them in the longer term.

1 - The UEFI 2.4A specification section 28.3.4

2 - The UEFI 2.4A specification section 28..6.1

2. The platform vendor (in this case, OVMF developers) implements device drivers for the platform's System Management Mode:

    - EFI_SMM_CONTROL2_PROTOCOL: for raising a synchronous (and/or) periodic SMI(s); that is, for entering SMM.
    - EFI_SMM_ACCESS2_PROTOCOL: for describing and accessing SMRAM.

    These protocols are documented in the PI Specification, Volume 4.

3. The platform DSC file is to include the following platform-independent modules:

    - MdeModulePkg/Core/PiSmmCore/PiSmmIpl.inf: SMM Initial Program Load
    - MdeModulePkg/Core/PiSmmCore/PiSmmCore.inf: SMM Core

2. In the section S3 (Suspend to RAM and Resume), it was noted that the LockBox storage must be similarly secure and tamper-resistant.

    On the S3 resume path, the PEIM providing EFI_PEI_S3_RESUME2_PPI (UefiCpuPkg/Universal/Acpi/S3Resume2Pei) restores and interprets data from the LockBox that has been saved there during boot. This PEIM, being part of the firmware, has full access to the platform. If an operating system can tamper with the contents of the LockBox, then at the next resume the platform's integrity might be subverted.

    OVMF stores the LockBox in normal guest RAM (refer to A Comprehensive Memory Map of OVMF for more information). Operating systems and third party UEFI drivers and UEFI applications that respect the UEFI memory map will not inadvertently overwrite the LockBox storage, but there's nothing to prevent something such as a malicious kernel from modifying the LockBox.

4. At this point, modules of type DXE_SMM_DRIVER can be loaded.
    Such drivers are privileged. They run in SMM, have access to SMRAM, and are separated and switched from other drivers through SMIs. Secure communication between unprivileged (non-SMM) and privileged (SMM) drivers happens through EFI_SMM_COMMUNICATION_PROTOCOL (implemented by the SMM Core, (refer back to point number 3)). DXE_SMM_DRIVER modules must sanitize their input (coming from unprivileged drivers) carefully.

5. The authenticated runtime variable services driver (for Secure Boot builds) is located

under "SecurityPkg/VariableAuthenticated/RuntimeDxe". OVMF currently builds the driver (a DXE_RUNTIME_DRIVER module) with the "VariableRuntimeDxe.inf" control file (refer to "OvmfPkg/OvmfPkgX64.dsc"), which does not use SMM. The directory includes two more INF files:

- VariableSmm.inf -- module type: DXE_SMM_DRIVER. A privileged driver that runs in SMM and has access to SMRAM.
- VariableSmmRuntimeDxe.inf -- module type: DXE_RUNTIME_DRIVER. A non-privileged driver that implements the variable runtime services (replacing the current "VariableRuntimeDxe.inf" file) by communicating with the above privileged SMM half via EFI_SMM_COMMUNICATION_PROTOCOL.

6. An SMRAM-based LockBox implementation needs to be discussed in two parts, because the LockBox is accessed in both PEI and DXE.
    - During DXE, drivers save data in the LockBox. A save operation is layered as follows:
        - The unprivileged driver wishing to store data in the LockBox links against the "MdeModulePkg/Library/SmmLockBoxLib/SmmLockBoxDxeLib.inf" library instance. The library allows the unprivileged driver to format requests for the privileged SMM LockBox driver (see below), and to parse responses.
        - The privileged SMM LockBox driver is built from "MdeModulePkg/Universal/LockBox/SmmLockBox/SmmLockBox.inf". This driver has module type DXE_SMM_DRIVER and can access SMRAM. The driver delegates command parsing and response formatting to "MdeModulePkg/Library/SmmLockBoxLib/SmmLockBoxSmmLib.inf".
        - The above two halves (unprivileged and privileged) mirror what we've seen in case of the variable service drivers, under point number 5.
    - In PEI, the S3 Resume PEIM (UefiCpuPkg/Universal/Acpi/S3Resume2Pei) retrieves data from the LockBox.

    - Presumably, S3Resume2Pei should be considered an "unprivileged PEIM", and the SMRAM access should be layered as seen in DXE. Unfortunately, edk2 does not implement all of the layers in PEI as the code either doesn't exist, or it is not open source.

Role	**DXE: protocol/module**	**PEI: PPI/module**
Unprivileged driver	Any	S3Resume2Pei.inf
Command formatting and response parsing	LIBRARY_CLASS = LockBoxLib SmmLockBoxDxeLib.inf	LIBRARY_CLASS = LockBoxLib SmmLockBoxPeiLib.inf
Privilege separation	EFI_SMM_COMMUNICATION_PROTOCOL PiSmmCore.inf	EFI_PEI_SMM_COMMUNICATION_PPI missing!
platform SMM and SMRAM access	EFI_SMM_CONTROL2_PROTOCOL EFI_SMM_ACCESS2_PROTOCOL to be done in OVMF	PEI_SMM_CONTROL_PPI PEI_SMM_ACCESS_PPI to be done in OVMF
command parsing and response formatting	LIBRARY_CLASS = LockBoxLib SmmLockBoxSmmLib.inf	LIBRARY_CLASS = LockBoxLib missing!
Privileged LockBox driver	SmmLockBox.inf	missing!

Alternatively, in the future OVMF might be able to provide a LockBoxLib instance (an SmmLockBoxPeiLib substitute) for S3Resume2Pei that accesses SMRAM directly, eliminating the need for deeper layers in the stack (that is, EFI_PEI_SMM_COMMUNICATION_PPI and deeper).

In fact, a "thin" EFI_PEI_SMM_COMMUNICATION_PPI implementation whose sole Communicate() member invariably returns EFI_NOT_STARTED would cause the current SmmLockBoxPeiLib library instance to directly perform full-depth SMRAM access and LockBox search, obviating the "missing" cells. (With reference to A Tour Beyond BIOS: Implementing S3 Resume with EDK2, by Jiewen Yao and Vincent Zimmer, October 2014.)

15 Select Features

This section covers the top-level "OvmfPkg" package directory, and discusses the more interesting drivers and libraries that have not been mentioned thus far.

15.1 X64-specific Reset Vector for OVMF

The "OvmfPkg/ResetVector" directory customizes the reset vector (found in "UefiCpuPkg/ResetVector/Vtf0") for "OvmfPkgX64.fdf", that is, when the SEC/PEI phases run in 64-bit (ie. long) mode. The reset vector's control flow can be simplified as follows:

```
resetVector                               [Ia16/ResetVectorVtf0.asm]

EarlyBspInitReal16                        [Ia16/Init16.asm]
Main16                                    [Main.asm]
  EarlyInit16                             [Ia16/Init16.asm]


  ; Transition the processor from
  ; 16-bit real mode to 32-bit flat mode
  TransitionFromReal16To32BitFlat         [Ia16/Real16ToFlat32.asm]


  ; Search for the
  ; Boot Firmware Volume (BFV)
  Flat32SearchForBfvBase                  [Ia32/SearchForBfvBase.asm]


  ; Search for the SEC entry point
  Flat32SearchForSecEntryPoint            [Ia32/SearchForSecEntry.asm]


  %ifdef ARCH_IA32
    ; Jump to the 32-bit SEC entry point
  %else
    ; Transition the processor
    ; from 32-bit flat mode
    ; to 64-bit flat mode
    Transition32FlatTo64Flat              [Ia32/Flat32ToFlat64.asm]


      SetCr3ForPageTables64               [Ia32/PageTables64.asm]
        ; set CR3 to page tables
        ; built into the ROM image


      ; enable PAE
      ; set LME
      ; enable paging
    ; Jump to the 64-bit SEC entry point

  %endif
```

On physical platforms, the initial page tables referenced by SetCr3ForPageTables64 are built statically into the flash device image, and are present in ROM at runtime. This is fine on physical platforms because the pre-built page table entries have the Accessed and Dirty bits set from the start.

Accordingly, for OVMF running in long mode on qemu/KVM, the initial page tables were mapped as a KVM_MEM_READONLY slot, as part of QEMU's pflash device (refer to How QEMU Maps the Firmware Image for more information).

In spite of the accessed and dirty bits being pre-set in the read-only, in-flash PTEs, in a virtual machine attempts are made to update said PTE bits, differently from physical hardware. The component attempting to update the read-only PTEs can be one of the following:

- The processor itself, if it supports nested paging, and the user enables that processor feature
- KVM code implementing shadow paging.

The first case presents no user-visible symptoms, but the second case (KVM, shadow paging) used to cause a triple fault, prior to Linux commit ba6a354 ("KVM: mmu: allow page tables to be in read-only slots").

For compatibility with earlier KVM versions, the OvmfPkg/ResetVector directory adapts the generic reset vector code as follows:

```
Transition32FlatTo64Flat                   [UefiCpuPkg/.../Ia32/Flat32ToFlat64.asm]

      SetCr3ForPageTables64            [OvmfPkg/ResetVector/Ia32/PageTables64.asm]

            ; dynamically build the initial page tables in RAM, at address
            ; PcdOvmfSecPageTablesBase (refer to the memory map above),
            ; identity-mapping the first 4 GB of address space

            ; set CR3 to PcdOvmfSecPageTablesBase

      ; enable PAE
      ; set LME
      ; enable paging
```

In this method, the PTEs that earlier KVM versions try to update (during shadow paging) are located in a read-write memory slot, and the write attempts succeed.

15.2 Client Library for QEMU's Firmware Configuration Interface

QEMU provides a write-only, 16-bit wide control port, and a read-write, 8-bit wide data port for exchanging configuration elements with the firmware. The firmware writes a selector (a key) to the control port (0x510), and then reads the corresponding configuration data (produced by QEMU) from the data port (0x511).

If the selected entry is writable, the firmware may overwrite it. If QEMU has associated a callback with the entry, then when the entry is completely rewritten, QEMU runs the callback. (OVMF does not rewrite any entries at the moment.)

A number of selector values (keys) are predefined. In particular, key 0x19 selects (returns) a directory of { name, selector, size } triplets, roughly speaking.

The firmware can request configuration elements by well-known name as well, by looking up the selector value first in the directory, by name, and then writing the selector to the control port. The number of bytes to read subsequently from the data port is known from the directory entry's "size" field.

By convention, directory entries (well-known symbolic names of configuration elements) are formatted as POSIX pathnames. For example, the array selected by the "etc/system-states" name indicates (among other things) whether the user enabled S3 support in QEMU.

The above interface is called "fw_cfg". The binary data associated with a symbolic name is called an "fw_cfg file".

OVMF's fw_cfg client library is found in "OvmfPkg/Library/QemuFwCfgLib". OVMF discovers many aspects of the virtual system with it; we refer to a few examples below.

15.3 Guest ACPI Tables

Operating systems discover most ot their hardware by parsing ACPI tables, and by interpreting the ACPI objects and methods. On the physical hardware, the platform vendor's firmware installs ACPI tables in memory that match both the hardware present in the system and the user's firmware configuration ("BIOS setup").

Under QEMU/KVM, the owner of the (virtual) hardware configuration is QEMU. Hardware can easily be reconfigured on the command line. Furthermore, features such as CPU hotplug, PCI hotplug, and memory hotplug are continuously developed for QEMU, and operating systems need direct ACPI support to exploit these features.

For this reason, QEMU builds its own ACPI tables dynamically, in a self-descriptive manner, and exports them to the firmware through a complex, multi-file fw_cfg interface. It is rooted in the "etc/table-loader" fw_cfg file. Additional details about this interface are out of the scope of this report and are therefore not discussed.

OVMF's AcpiPlatformDxe driver fetches the ACPI tables, and installs them for the guest OS with the EFI_ACPI_TABLE_PROTOCOL, which is in turn provided by the generic "MdeModulePkg/Universal/Acpi/AcpiTableDxe" driver.

For earlier QEMU versions and machine types (which we generally don't recommend for OVMF (refer to Scope for more information)), the "OvmfPkg/AcpiTables" directory contains a few static ACPI table templates. When the "etc/table-loader" fw_cfg file is unavailable, AcpiPlatformDxe installs these default tables using a little bit of dynamic patching.

When OVMF runs in a Xen domU, AcpiTableDxe also installs ACPI tables that originate from the hypervisor's environment.

15.4 Guest SMBIOS Tables

Quoting the SMBIOS Reference Specification:

> *[...] the System Management BIOS Reference Specification addresses how motherboard and system vendors present management information about their products in a standard format [...]*

In practice, SMBIOS tables are just another set of tables that the platform vendor's firmware installs in RAM for the operating system, and, importantly, for management applications running on the OS. Without rehashing the "Guest ACPI Tables" section in full, the following table maps the OVMF roles seen there from ACPI to SMBIOS:

Role	ACPI	SMBIOS
fw_cfg file	etc/table-loader	etc/smbios/smbios-tables
OVMF driver under "OvmfPkg"	AcpiPlatformDxe	SmbiosPlatformDxe
Underlying protocol implemented by generic driver under "MdeModulePkg/Universal"	EFI_ACPI_TABLE_PROTOCOL Acpi/AcpiTableDxe	EFI_SMBIOS_PROTOCOL SmbiosDxe
default tables available for earlier QEMU machine types, with hot-patching	yes	♦ yes, Type0 and Type1 tables
tables fetched in Xen domUs	Yes	Yes

15.5 OVMF’s Platform-specific Boot Policy

OVMF's BDS (Boot Device Selection) phase is implemented by IntelFrameworkModulePkg/Universal/BdsDxe. In general, this large driver does the following:

- Provides the EFI BDS architectural protocol (which DXE transfers control to after dispatching all DXE drivers)
- Connects drivers to devices
- Enumerates boot devices
- Auto-generates boot options
- Provides "BIOS setup" screens, such as:
  - Boot Manager, for booting options
  - Boot Maintenance Manager, for adding, deleting, and reordering boot options, changing console properties etc.
  - Device Manager, where devices can register configuration forms, including:
    - Secure Boot configuration forms
    - OVMF's Platform Driver form (refer to Platform Driver for more information)

Firmware that includes the "IntelFrameworkModulePkg/Universal/BdsDxe" driver can customize its behavior by providing an instance of the PlatformBdsLib library class. The driver links against this platform library, and the platform library can call Intel's BDS utility functions from "IntelFrameworkModulePkg/Library/GenericBdsLib".

OVMF's PlatformBdsLib instance can be found in "OvmfPkg/Library/PlatformBdsLib". The main function where the BdsDxe driver enters the library is PlatformBdsPolicyBehavior().

Two important details should be noted:

- OVMF is capable of loading kernel images directly from fw_cfg, matching QEMU's -kernel, -initrd, and -append command line options. This feature is useful for rapid, repeated Linux kernel testing, and is implemented in the following call tree:

  ```
  PlatformBdsPolicyBehavior() [OvmfPkg/Library/PlatformBdsLib/BdsPlatform.c]
      TryRunningQemuKernel() [OvmfPkg/Library/PlatformBdsLib/QemuKernel.c]
          LoadLinux*() [OvmfPkg/Library/LoadLinuxLib/Linux.c]
  ```

  OvmfPkg/Library/LoadLinuxLib ports the efilinux bootloader project into OvmfPkg.

- OVMF seeks to comply with the boot order specification passed down by QEMU over fw_cfg (Explained further in OVMF Boot Order Specification).

15.6 OVMF Boot Order Specification

As explained in the section above, OVMF seeks to comply with the boot order specification passed down by QEMU over fw_cfg.

15.6.1 OVMF Boot Modes

During the PEI phase, OVMF determines and stores the Boot Mode in the PHIT HOB (refer to S3 (Suspend to RAM and Resume)) for more information. The boot mode is supposed to influence the rest of the system, for example it distinguishes S3 resume (BOOT_ON_S3_RESUME) from a "normal" boot.

In general, "normal" boots can be further differentiated from each other; for example for speed reasons. When the firmware can tell during PEI that the chassis has not been opened since last power-up, then it might want to save time by not connecting all devices and not enumerating all boot options from scratch; it could just rely on the stored results of the last enumeration. The matching BootMode value, to be set during PEI, would be BOOT_ASSUMING_NO_CONFIGURATION_CHANGES.

OVMF only sets one of the following two boot modes, based on CMOS contents:

- BOOT_ON_S3_RESUME,
- BOOT_WITH_FULL_CONFIGURATION.

For BOOT_ON_S3_RESUME, please refer to "S3 (Suspend to RAM and Resume)". The other boot mode supported by OVMF, BOOT_WITH_FULL_CONFIGURATION, is an appropriate "catch-all" for a virtual machine, where hardware can easily change from boot to boot.

15.6.2 Auto-generation of Boot Options

When not resuming from S3 sleep[3], OVMF always connects all devices, and enumerates all bootable devices as new boot options (non-volatile variables named Boot####).

The auto-enumerated boot options are stored in the BootOrder non-volatile variable after any preexisting options. (Boot options may exist before auto-enumeration for example because the user added them manually with the Boot Maintenance Manager or the efibootmgr utility. They could also originate from an earlier auto-enumeration.)

```
PlatformBdsPolicyBehavior()   [OvmfPkg/.../BdsPlatform.c]
TryRunningQemuKernel()        [OvmfPkg/.../QemuKernel.c]
BdsLibConnectAll()            [IntelFrameworkModulePkg/.../BdsConnect.c]
BdsLibEnumerateAllBootOption() [IntelFrameworkModulePkg/.../BdsBoot.c]
BdsLibBuildOptionFromHandle() [IntelFrameworkModulePkg/.../BdsBoot.c]
BdsLibRegisterNewOption() [IntelFrameworkModulePkg/.../BdsMisc.c]
//
// Append the new option number to the original option order
//
```

15.6.3 Relative UEFI Device Paths in Boot Options

The handling of relative ("short-form") UEFI device paths is best demonstrated through an

3 During S3 resume, DXE is not reached, and as such BDS isn't as well.

example, and by quoting the UEFI 2.4A specification. A short-form hard drive UEFI device path could be (displaying each device path node on a separate line for readability), as shown:

```
HD(1,GPT,14DD1CC5-D576-4BBF-8858-BAF877C8DF61,0x800,0x64000)/
\EFI\fedora\shim.efi
```

This device path lacks prefix nodes (such as. hardware or messaging type nodes) that would lead to the hard drive. During load option processing, the above short-form or relative device path could be matched against the following absolute device path:

```
PciRoot(0x0)/
Pci(0x4,0x0)/
HD(1,GPT,14DD1CC5-D576-4BBF-8858-BAF877C8DF61,0x800,0x64000)/

\EFI\fedora\shim.efi
```

The motivation for this type of device path matching / completion is to allow the user to move around the hard drive (for example, to plug a controller in a different PCI slot, or to expose the block device on a different iSCSI path) while enabling the firmware to find the hard drive. According to the UEFI specification:

> *“[...] Section 3.1.2 defines special rules for processing the Hard Drive Media Device Path. These special rules enable a disk's location to change and still have the system boot from the disk. [...]”*[4]

3.1.2 Load Option Processing

> *“[...] The boot manager must [...] support booting from a short-form device path that starts with the first element being a hard drive media device path [...]. The boot manager must use the GUID or signature and partition number in the hard drive device path to match it to a device in the system. If the drive supports the GPT partitioning scheme the GUID in the hard drive media device path is compared with the UniquePartitionGuid field of the GUID Partition Entry [...]. If the drive supports the PC-AT MBR scheme the signature in the hard drive media device path is compared with the UniqueMBRSignature in the Legacy Master Boot Record [...]. If a signature match is made, then the partition number must also be matched. The hard drive device path can be appended to the matching hardware device path and normal boot behavior can then be used. If more than one device matches the hard drive device path, the boot*

4 UEFI Specification Section 9.3.6 Media Device path and Section 9.3.6.1 Hard Drive

> *manager will pick one arbitrarily. Thus the operating system must ensure the uniqueness of the signatures on hard drives to guarantee deterministic boot behavior."*[5]

Edk2 implements and exposes the device path completion logic in the already referenced "IntelFrameworkModulePkg/Library/GenericBdsLib" library, in the BdsExpandPartitionPartialDevicePathToFull() function.

15.6.4 Filtering and Reordering the Boot Options Based on fw_cfg

Once an "all-inclusive", partly preexisting, partly freshly auto-generated boot option list is created (as mentioned in Auto-generation of Boot Options), OVMF loads QEMU's requested boot order from fw_cfg, and filters and reorders the Auto-generation of Boot Options with the boot order requested by QEMU:

```
PlatformBdsPolicyBehavior()                       [OvmfPkg/.../BdsPlatform.c]
      TryRunningQemuKernel()                       [OvmfPkg/.../QemuKernel.c]
      BdsLibConnectAll()          [IntelFrameworkModulePkg/.../BdsConnect.c]
      BdsLibEnumerateAllBootOption() [IntelFrameworkModulePkg/.../BdsBoot.c]
      SetBootOrderFromQemu()                    [OvmfPkg/.../QemuBootOrder.c]
```

According to the (preferred) "-device ...,bootindex=N" and the (legacy) '-boot order=drives' command line options, QEMU requests a boot order from the firmware through the "bootorder" fw_cfg file. (For a bootindex example, refer to QEMU Invocation Example) The "bootorder" fw_cfg file consists of OpenFirmware (OFW) device paths (note: not UEFI device paths!), one path per line. An example of such a list is as follows:

```
/pci@i0cf8/scsi@4/disk@0,0
/pci@i0cf8/ide@1,1/drive@1/disk@0
/pci@i0cf8/ethernet@3/ethernet-phy@0
```

5 UEFI Specification Section `3.1.2 Load Option Processing`

Using this list, OVMF filters and reorders the auto-generated boot option list using the list as shown in Auto-generation of Boot Options, with the following nested loops algorithm:

```
new_uefi_order := <empty>
for each qemu_ofw_path in QEMU's OpenFirmware device path list:
  qemu_uefi_path_prefix := translate(qemu_ofw_path)


  for each boot_option in current_uefi_order:
    full_boot_option := complete(boot_option)


    if match(qemu_uefi_path_prefix, full_boot_option):
      append(new_uefi_order, boot_option)
      break


  for each unmatched boot_option in current_uefi_order:
    if survives(boot_option):
    append(new_uefi_order, boot_option)

  current_uefi_order := new_uefi_order
```

OVMF iterates over QEMU's OFW device paths in order, translates each to a UEFI device path prefix, tries to match the translated prefix against the UEFI boot options (which are completed from relative form to absolute form for the purpose of prefix matching), and if there's a match, the matching boot option is appended to the new boot order (which starts out empty).

The translate() function is further elaborated in Translating QEMU's OpenFirmware Device Paths to UEFI Device Path Prefixes, and the complete() function has been explained in Relative UEFI Device Paths in Boot Options.

In addition, UEFI boot options that remain unmatched after filtering and reordering are post-processed, and some of them "survive". Due to the fact that OpenFirmware device paths have less expressive power than their UEFI counterparts, some UEFI boot options are simply inexpressible (and as such are never matched) using the nested loops algorithm.

An important example is the memory-mapped UEFI shell, whose UEFI device path is inexpressible by QEMU's OFW device paths, as shown:

```
MemoryMapped(0xB,0x900000,0x10FFFFF)/
FvFile(7C04A583-9E3E-4F1C-AD65-E05268D0B4D1)
```

Note:

The address range visible in the MemoryMapped() node corresponds to DXEFV as specified in "A Comprehensive Memory Map of OVMF".

In addition, the FvFile() node's GUID originates from the FILE_GUID entry of "ShellPkg/Application/Shell/Shell.inf".)

The UEFI shell can be booted by pressing ESC in OVMF on the TianoCore splash screen, and navigating to Boot Manager | EFI Internal Shell.

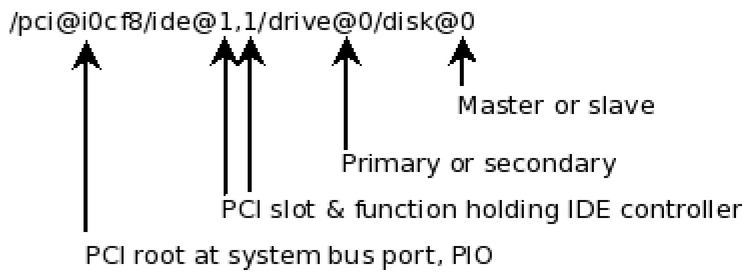
If the "survival policy" was not implemented, the UEFI shell's boot option would always be filtered out. The current "survival policy" preserves all boot options that start with neither PciRoot() nor HD().

15.6.5 Translating QEMU's OpenFirmware Device Paths to UEFI Device Path Prefixes

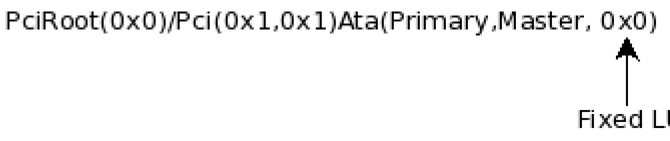
This section covers the (strictly heuristic) mappings currently performed by OVMF.
The "prefix only" nature of the translation output is rooted minimally in the fact that QEMU's OpenFirmware device paths cannot carry path names within file systems. There's no way to specify path names such as \EFI\fedora\shim.efi, in an OFW device path, therefore a UEFI device path translated from an OFW device path can at best be a prefix (not a full match) of a UEFI device path that ends with "\EFI\fedora\shim.efi".

__IDE disk, IDE CD-ROM:__

OpenFirmware device path:

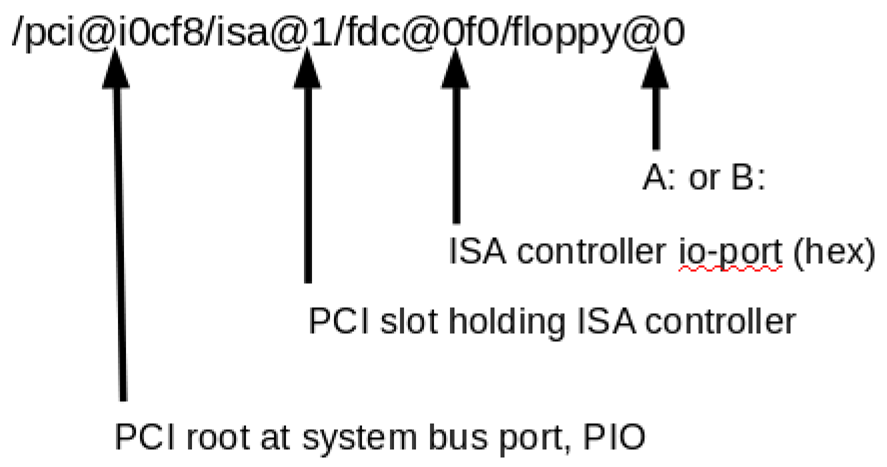

/pci@i0cf8/ide@1,1/drive@0/disk@0
Master or slave
Primary or secondary
PCI slot & function holding IDE controller
PCI root at system bus port, PIO

UEFI device path prefix:

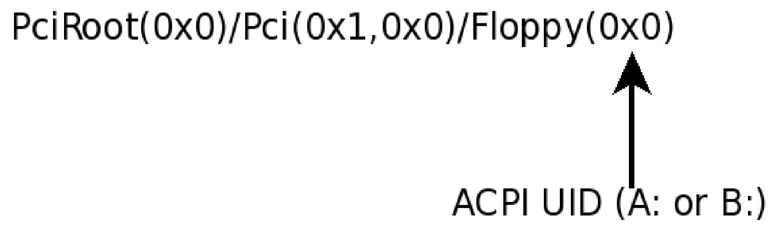


Floppy disk:

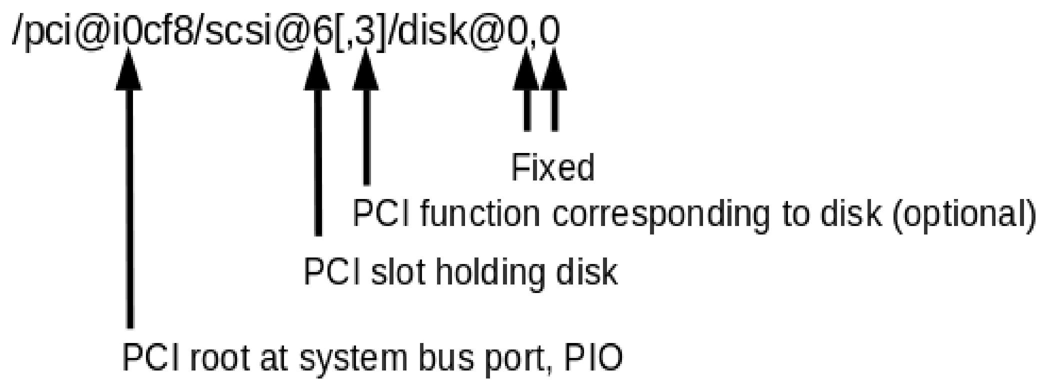
OpenFirmware device path:


/pci@i0cf8/isa@1/fdc@0f0/floppy@0
A: or B:
ISA controller io-port (hex)
PCI slot holding ISA controller
PCI root at system bus port, PIO

UEFI device path prefix:



Virtio-block disk:


/pci@i0cf8/scsi@6[,3]/disk@0,0
Fixed
PCI function corresponding to disk (optional)
PCI slot holding disk
PCI root at system bus port, PIO

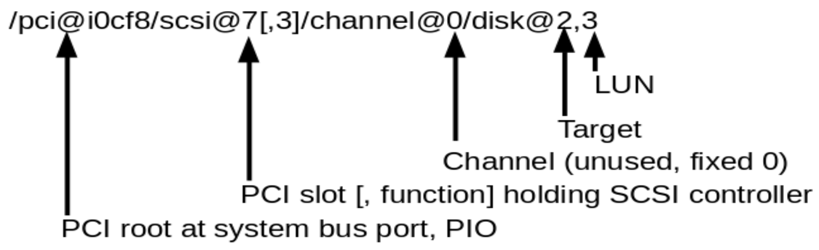
OpenFirmware device path:

UEFI device path prefixes (dependent on the presence of a nonzero PCI function in the OFW device path):

```
	PciRoot(0x0)/Pci(0x6,0x0)/HD(
	PciRoot(0x0)/Pci(0x6,0x3)/HD(
```

<u>Virtio-scsi disk and virtio-scsi passthrough:</u>

OpenFirmware device path:

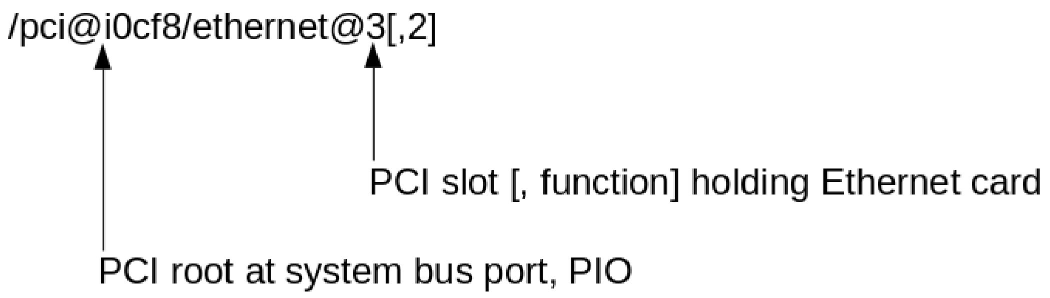

/pci@i0cf8/scsi@7[,3]/channel@0/disk@2,3
LUN
Target
Channel (unused, fixed 0)
PCI slot [, function] holding SCSI controller
PCI root at system bus port, PIO

UEFI device path prefixes (dependent on the presence of a nonzero PCI function in the OFW device path):

```
	PciRoot(0x0)/Pci(0x7,0x0)/Scsi(0x2,0x3)
	PciRoot(0x0)/Pci(0x7,0x3)/Scsi(0x2,0x3)
```

<u>Emulated and passed-through (physical) network cards:</u>

OpenFirmware device path:

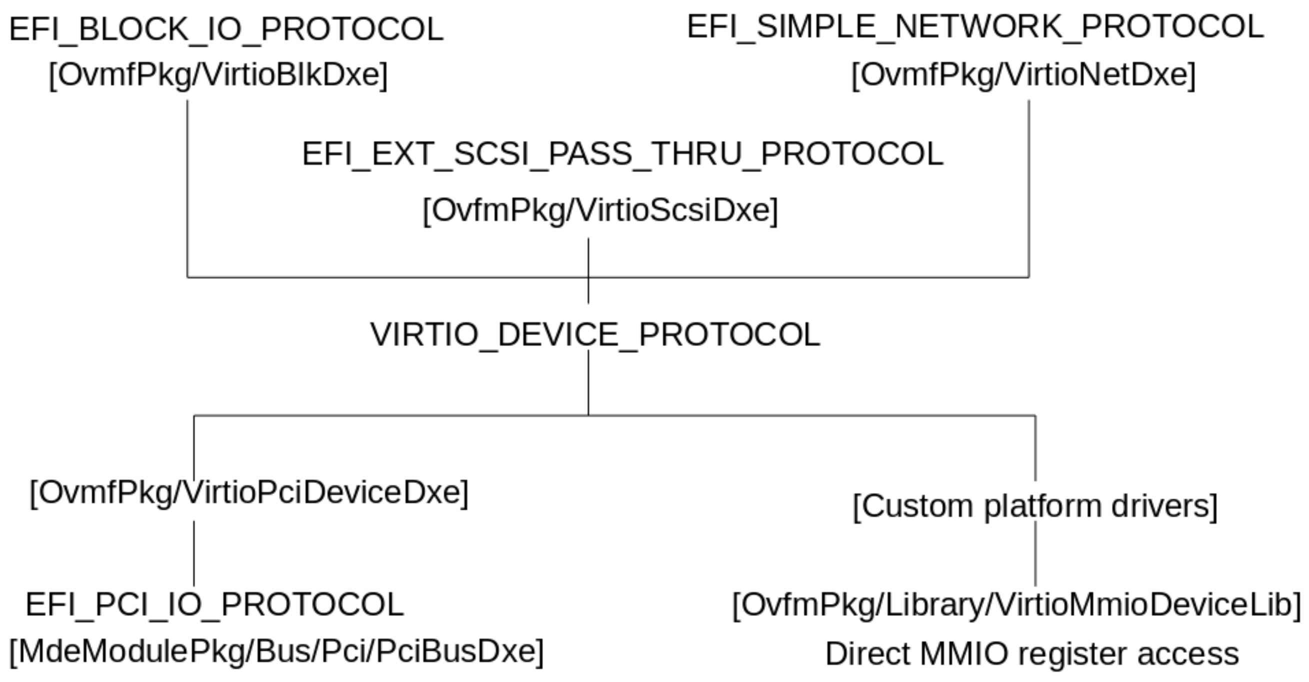

/pci@i0cf8/ethernet@3[,2]
PCI slot [, function] holding Ethernet card
PCI root at system bus port, PIO

UEFI device path prefixes (dependent on the presence of a nonzero PCI function in the OFW device path):

```
	PciRoot(0x0)/Pci(0x3,0x0)
	PciRoot(0x0)/Pci(0x3,0x2)
```

15.7 Virtio Drivers

UEFI abstracts various types of hardware resources into protocols, and allows firmware developers to implement those protocols in device drivers. The Virtio Specification defines various types of virtual hardware for virtual machines. Connecting the two specifications, OVMF provides UEFI drivers for QEMU's virtio-block, virtio-scsi, and virtio-net devices. The following diagram presents the protocol and driver stack related to Virtio devices in edk2 and OVMF. Each node in the graph identifies a protocol and/or the edk2 driver that produces it. Nodes on the top are more abstract.



EFI_BLOCK_IO_PROTOCOL
[OvmfPkg/VirtioBlkDxe]
EFI_SIMPLE_NETWORK_PROTOCOL
[OvmfPkg/VirtioNetDxe]
EFI_EXT_SCSI_PASS_THRU_PROTOCOL
[OvfmPkg/VirtioScsiDxe]
VIRTIO_DEVICE_PROTOCOL
[OvmfPkg/VirtioPciDeviceDxe]
[Custom platform drivers]
EFI_PCI_IO_PROTOCOL
[MdeModulePkg/Bus/Pci/PciBusDxe]
[OvfmPkg/Library/VirtioMmioDeviceLib]
Direct MMIO register access

The top three drivers produce standard UEFI abstractions: the Block IO Protocol, the Extended SCSI Pass Thru Protocol, and the Simple Network Protocol, for virtio-block, virtio-scsi, and virtio-net devices, respectively.

Comparing these device-specific virtio drivers to each other, it can be determined that:

- They all conform to the UEFI Driver Model. This means that their entry point functions don't immediately start to search for devices and drive them, as they only register instances of the EFI_DRIVER_BINDING_PROTOCOL. The UEFI Driver Model then enumerates devices and chains matching drivers automatically.
- They are as minimal as possible, while remaining correct (refer to source code comments for details). For example, VirtioBlkDxe and VirtioScsiDxe both support only one request in flight.
  In theory, VirtioBlkDxe could implement EFI_BLOCK_IO2_PROTOCOL, which allows queueing. Similarly, VirtioScsiDxe does not support the non-blocking mode of

EFI_EXT_SCSI_PASS_THRU_PROTOCOL.PassThru(). (Which is permitted by the UEFI specification.) Both VirtioBlkDxe and VirtioScsiDxe delegate synchronous request handling to "OvmfPkg/Library/VirtioLib". This limitation helps keep the implementation simple, and testing thus far seems to imply satisfactory performance, for a virtual boot firmware.
VirtioNetDxe cannot avoid queueing, because EFI_SIMPLE_NETWORK_PROTOCOL requires it on the interface level. Consequently, VirtioNetDxe is significantly more complex than VirtioBlkDxe and VirtioScsiDxe. Technical notes are provided in "OvmfPkg/VirtioNetDxe/TechNotes.txt".

- None of these drivers access hardware directly. Instead, the Virtio Device Protocol (OvmfPkg/Include/Protocol/VirtioDevice.h) collects / extracts virtio operations defined in the Virtio Specification, and these backend-independent virtio device drivers go through the abstract VIRTIO_DEVICE_PROTOCOL.

  IMPORTANT!

  The VIRTIO_DEVICE_PROTOCOL is not a standard UEFI protocol. It is internal to edk2 and not described in the UEFI specification. It should only be used by drivers and applications that live inside the edk2 source tree.

Currently two providers exist for VIRTIO_DEVICE_PROTOCOL:

- The first one is the "more traditional" virtio-pci backend, implemented by OvmfPkg/VirtioPciDeviceDxe. This driver also complies with the UEFI Driver Model. It consumes an instance of the EFI_PCI_IO_PROTOCOL, and, if the PCI device/function under probing appears to be a virtio device, it produces a Virtio Device Protocol instance for it. The driver translates abstract virtio operations to PCI accesses.
- The second provider, the virtio-mmio backend, is a library, not a driver, living in OvmfPkg/Library/VirtioMmioDeviceLib. This library translates abstract virtio operations to MMIO accesses.

15.7.1 VirtioMmioDeviceLib: the virtio-mmio backend

The virtio-mmio backend is only a library -- rather than a standalone, UEFI Driver Model-compliant driver -- because the type of resource it consumes, an MMIO register block base address, is not enumerable.

In other words, while the PCI root bridge driver and the PCI bus driver produce instances of EFI_PCI_IO_PROTOCOL automatically, thereby enabling the UEFI Driver Model to probe devices and stack up drivers automatically, no such enumeration exists for MMIO register blocks.

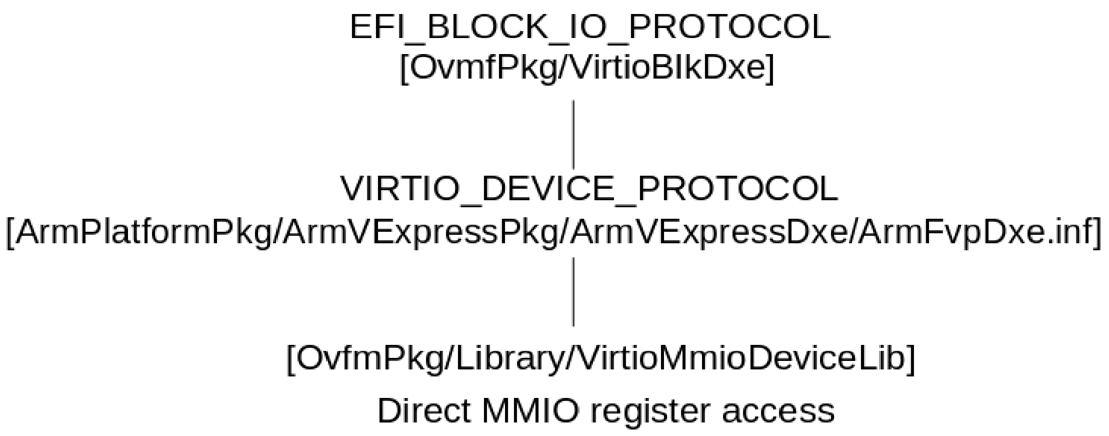
For this reason, VirtioMmioDeviceLib needs to be linked into thin, custom platform drivers that dispose over this kind of information. As soon as a driver knows about the MMIO register block base addresses, it can pass each to the library, and then the VIRTIO_DEVICE_PROTOCOL will be instantiated (assuming a valid virtio-mmio register block of course). From that point on the UEFI Driver Model again takes care of the chaining.

Typically, such a custom driver does not conform to the UEFI Driver Model (because that would presuppose auto-enumeration for MMIO register blocks). Hence it has the following responsibilities:

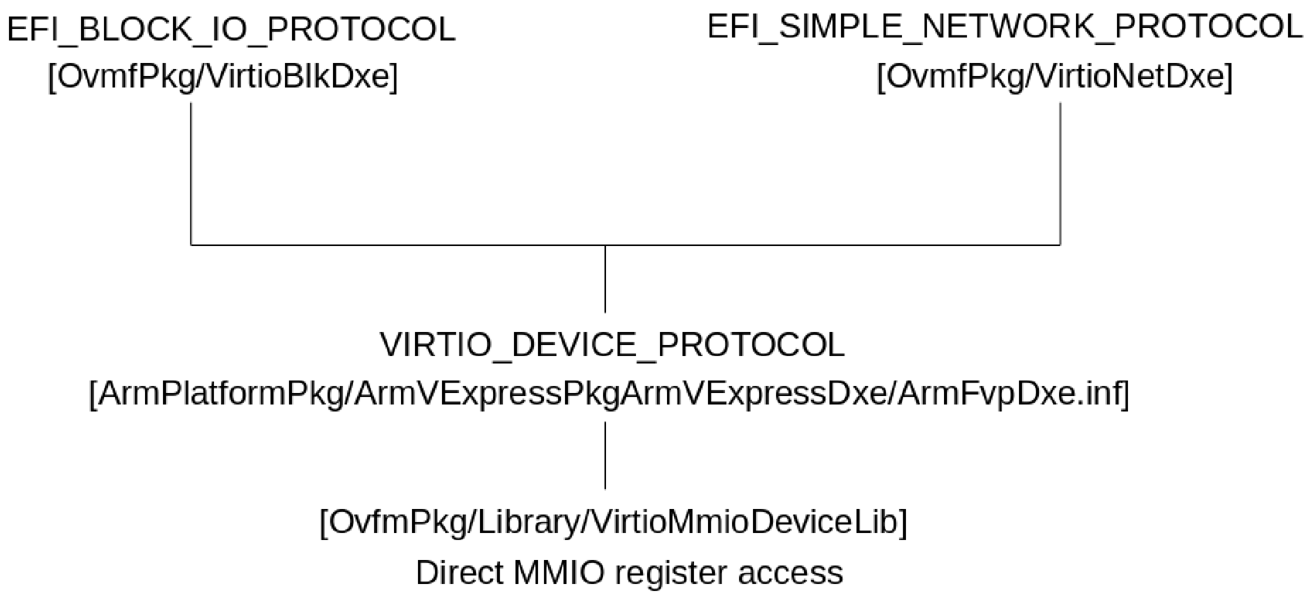
- To behave as a "wrapper" UEFI driver around the library
- To know virtio-mmio base addresses
- In its entry point function, it shall create a new UEFI handle with an instance of the EFI_DEVICE_PATH_PROTOCOL for each virtio-mmio device it knows the base address for,
- It shall call VirtioMmioInstallDevice() on those handles, with the corresponding base addresses.

OVMF itself does not employ VirtioMmioDeviceLib. However, the library is used (or has been tested as Proof-of-Concept) in the following 64-bit and 32-bit ARM emulator setups.

- In "RTSM_VE_FOUNDATIONV8_EFI.fd" and "FVP_AARCH64_EFI.fd", on ARM Holdings' ARM(R) v8-A Foundation Model and ARM(R) AEMv8-A Base Platform FVP emulators, respectively:


EFI_BLOCK_IO_PROTOCOL
[OvmfPkg/VirtioBlkDxe]
VIRTIO_DEVICE_PROTOCOL
[ArmPlatformPkg/ArmVExpressPkg/ArmVExpressDxe/ArmFvpDxe.inf]
[OvfmPkg/Library/VirtioMmioDeviceLib]
Direct MMIO register access

- In "RTSM_VE_CORTEX-A15_EFI.fd" and "RTSM_VE_CORTEX-A15_MPCORE_EFI.fd", on "qemu-system-arm -M vexpress-a15":


EFI_BLOCK_IO_PROTOCOL
[OvmfPkg/VirtioBlkDxe]
EFI_SIMPLE_NETWORK_PROTOCOL
[OvmfPkg/VirtioNetDxe]
VIRTIO_DEVICE_PROTOCOL
[ArmPlatformPkg/ArmVExpressPkgArmVExpressDxe/ArmFvpDxe.inf]
[OvfmPkg/Library/VirtioMmioDeviceLib]
Direct MMIO register access

In the above ARM / VirtioMmioDeviceLib configurations, VirtioBlkDxe was tested with booting Linux distributions, while VirtioNetDxe was tested with pinging public IPv4 addresses from the UEFI shell.

15.8 Platform Driver

Sometimes, elements of persistent firmware configuration are best exposed to the user in a friendly way. OVMF's platform driver (OvmfPkg/PlatformDxe) presents such settings on the "OVMF Platform Configuration" dialog:

Press ESC on the TianoCore splash screen, Navigate to Device Manager | OVMF Platform Configuration. At the moment, OVMF's platform driver handles only one setting: the preferred graphics resolution.

This is useful for two purposes:

- Some UEFI shell commands, like DRIVERS and DEVICES, benefit from a wide display. Using the MODE shell command, the user can switch to a larger text resolution (limited by the graphics resolution), and see the command output in a more easily consumable way.

  ♦ The list of text modes available to the MODE command is also limited by ConSplitterDxe (found under MdeModulePkg/Universal/Console). ConSplitterDxe builds an intersection of text modes that are simultaneously supported by all consoles that ConSplitterDxe multiplexes console output to.

  In practice, the strongest text mode restriction comes from TerminalDxe, which provides console I/O on serial ports. TerminalDxe has a very limited built-in list of text modes, heavily pruning the intersection built by ConSplitterDxe, and made available to the MODE command.

  On the Red Hat Enterprise Linux 7.1 host, TerminalDxe's list of modes has been extended with text resolutions that match the Spice QXL GPU's common graphics resolutions. This way a "full screen" text mode should always be available in the MODE command.

- The other advantage of controlling the graphics resolution lies with UEFI operating systems that don't as of now have a native driver for QEMU's virtual video cards such as the Spice QXL GPU. Such operating systems may choose to inherit the properties of OVMF's EFI_GRAPHICS_OUTPUT_PROTOCOL (refer to Virtio Drivers for more information).
  Although the display can be used at runtime in such cases, by direct framebuffer access, its properties, for example, the resolution, cannot be modified. The platform

driver allows the user to select the preferred GOP resolution, reboot, and let the guest OS inherit that preferred resolution.

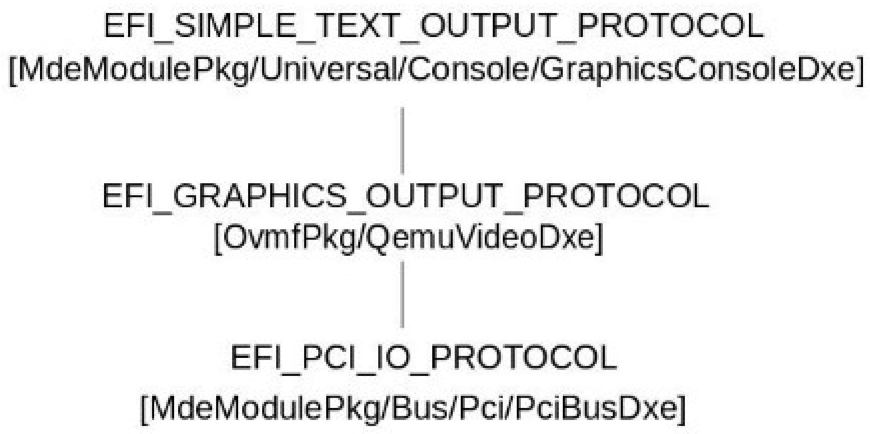
The platform driver has three access points: the "normal" driver entry point, a set of HII callbacks, and a GOP installation callback. These are explained further in Driver Entry Point: the PlatformInit() Function, HII Callbacks and the User Interface, and GOP Installation Callback.

15.8.1 Driver Entry Point: the PlatformInit() Function

First, this function loads any available settings, and makes them take effect. For the preferred graphics resolution in particular, this means setting the following PCDs:

- gEfiMdeModulePkgTokenSpaceGuid.PcdVideoHorizontalResolution
- gEfiMdeModulePkgTokenSpaceGuid.PcdVideoVerticalResolution

These PCDs influence the GraphicsConsoleDxe driver (located under MdeModulePkg/Universal/Console), which switches to the preferred graphics mode, and produces EFI_SIMPLE_TEXT_OUTPUT_PROTOCOLs on GOPs:


EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL
[MdeModulePkg/Universal/Console/GraphicsConsoleDxe]
EFI_GRAPHICS_OUTPUT_PROTOCOL
[OvmfPkg/QemuVideoDxe]
EFI_PCI_IO_PROTOCOL
[MdeModulePkg/Bus/Pci/PciBusDxe]

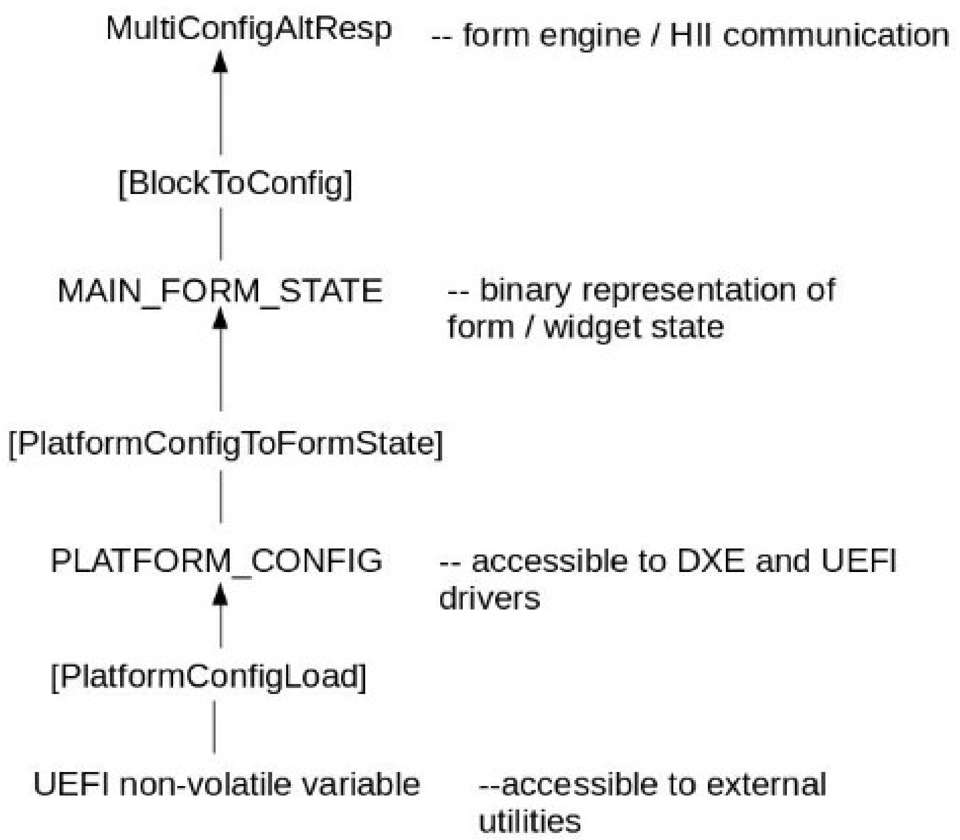
After which, the driver entry point registers the user interface, including HII callbacks. Finally, the driver entry point registers a GOP installation callback.

15.8.2 HII Callbacks and the User Interface

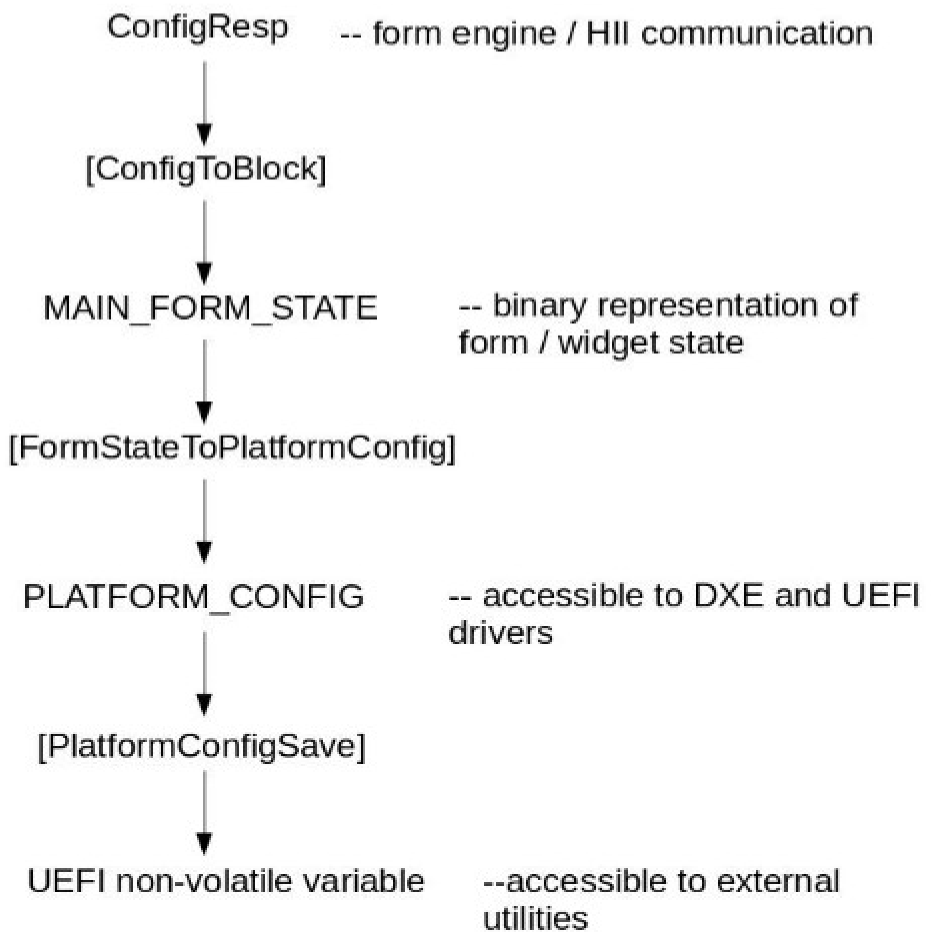
The Human Interface Infrastructure (HII) "is a set of protocols that allow a UEFI driver to provide the ability to register user interface and configuration content with the platform firmware".

OVMF's platform driver:

- Provides a static, basic, visual form (PlatformForms.vfr), written in the Visual Forms Representation language
- Includes a UCS-16 encoded message catalog (Platform.uni).
- Includes source code that dynamically populates parts of the form, with the help of MdeModulePkg/Library/UefiHiiLib -- this library simplifies the handling of IFR (Internal Forms Representation) opcodes,
- Processes form actions that the user takes (Callback() function),
- Loads and saves platform configuration in a private, non-volatile variable (ExtractConfig() and RouteConfig() functions).


MultiConfigAltResp
-- form engine / HII communication
[BlockToConfig]
MAIN_FORM_STATE
-- binary representation of form / widget state
[PlatformConfigToFormState]
PLATFORM_CONFIG
-- accessible to DXE and UEFI drivers
[PlatformConfigLoad]
UEFI non-volatile variable
--accessible to external utilities

The ExtractConfig() HII callback implements the following stack of conversions, for loading configuration and presenting it to the user:The layers are very similar for the reverse direction, ie. when taking input from the user, and saving the configuration (RouteConfig() HII callback):


ConfigResp
-- form engine / HII communication
[ConfigToBlock]
MAIN_FORM_STATE
-- binary representation of form / widget state
[FormStateToPlatformConfig]
PLATFORM_CONFIG
-- accessible to DXE and UEFI drivers
[PlatformConfigSave]
UEFI non-volatile variable
--accessible to external utilities

15.8.3 GOP Installation Callback

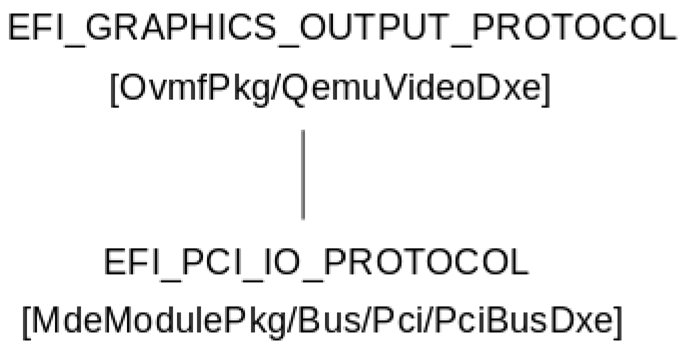
When the platform driver starts, a GOP may not be available yet. Thus the driver entry point registers a callback (the GopInstalled() function) for GOP installations. When the first GOP is produced (usually by QemuVideoDxe, or potentially by.a third party video driver), PlatformDxe retrieves the list of graphics modes the GOP supports, and dynamically populates the drop-down list of available resolutions on the form. The GOP installation callback is then removed.

15.9 Video Driver

OvmfPkg/QemuVideoDxe is OVMF's built-in video driver. It provides two types of services: graphics output protocol (primary service), and Int10h (VBE) shim (secondary service).

15.9.1 Primary Video Service: Graphics Output Protocol

QemuVideoDxe conforms to the UEFI Driver Model; it produces an instance of EFI_GRAPHICS_OUTPUT_PROTOCOL (GOP) for each PCI display that it supports and is connected to:


EFI_GRAPHICS_OUTPUT_PROTOCOL
[OvmfPkg/QemuVideoDxe]
EFI_PCI_IO_PROTOCOL
[MdeModulePkg/Bus/Pci/PciBusDxe]

It supports the following QEMU video cards:

- Cirrus 5430 ("-device cirrus-vga"),
- Standard VGA ("-device VGA"),
- QXL VGA ("-device qxl-vga", "-device qxl").

For Cirrus the following resolutions and color depths are available: 640x480x32, 800x600x32, 1024x768x24. On stdvga and QXL a long list of resolutions is available. The list is filtered against the frame buffer size during initialization.

The size of the QXL VGA compatibility framebuffer can be changed with the

```
-device qxl-vga,vgamem_mb=$NUM_MB
```

QEMU option. If $NUM_MB exceeds 32, then the following is necessary instead:

```
-device qxl-vga,vgamem_mb=$NUM_MB,ram_size_mb=$((NUM_MB*2))
```

Because the compatibility framebuffer can't cover more than half of PCI BAR #0. The latter defaults to 64MB in size, and is controlled by the "ram_size_mb" property.

15.9.2 Secondary Video Service: Int10h (VBE) Shim

When QemuVideoDxe binds the first Standard VGA or QXL VGA device, and there is no real VGA BIOS present in the C to F segments (which could originate from a legacy PCI option ROM -- refer to Compatibility Support Module (CSM), then QemuVideoDxe installs a minimal, "fake" VGA BIOS -- an Int10h (VBE) "shim".

The shim is implemented in 16-bit assembly in "OvmfPkg/QemuVideoDxe/VbeShim.asm". The "VbeShim.sh" shell script assembles it and formats it as a C array ("VbeShim.h") with the help of the "nasm" utility. The driver's InstallVbeShim() function copies the shim in place (the C segment), and fills in the VBE Info and VBE Mode Info structures. The real-mode 10h interrupt vector is pointed to the shim's handler.

The shim is (correctly) irrelevant and invisible for all UEFI operating systems we know about -- except Windows Server 2008 R2 and other Windows operating systems in that family.

Namely, the Windows 2008 R2 SP1 (and Windows 7) UEFI guest's default video driver dereferences the real mode Int10h vector, loads the pointed-to handler code, and executes what it thinks to be VGA BIOS services in an internal real-mode emulator. Consequently, video mode switching used not to work in Windows 2008 R2 SP1 when it ran on the "pure UEFI" build of OVMF, making the guest uninstallable. Hence the (otherwise optional, non-default) Compatibility Support Module (CSM) ended up a requirement for running such guests.

The hard dependency on the sophisticated SeaBIOS CSM and the complex supporting edk2 infrastructure, for enabling this family of guests, was considered sub-optimal by some members of the upstream community,

♦ and was certainly considered a serious maintenance disadvantage for Red Hat Enterprise Linux 7.1 hosts.

Thus, the shim has been collaboratively developed for the Windows 7 / Windows Server 2008 R2 family. The shim provides a real stdvga / QXL implementation for the few services that are in fact necessary for the Windows 2008 R2 SP1 (and Windows 7) UEFI guest, plus some "fakes" that the guest invokes but whose effect is not important. The only supported mode is 1024x768x32, which is enough to install the guest and then upgrade its video driver to the full-featured QXL XDDM one.

The C segment is not present in the UEFI memory map prepared by OVMF. Memory space that would cover it is not added (either in PEI, in the form of memory resource descriptor HOBs, or in DXE, via gDS->AddMemorySpace()). This way the handler body is invisible to all other UEFI guests, and the rest of edk2.

The Int10h real-mode IVT entry is covered with a Boot Services Code page, making that too inaccessible to the rest of edk2. Due to the allocation type, UEFI guest OSes different from the Windows Server 2008 family can reclaim the page at zero. (The Windows 2008 family accesses that page regardless of the allocation type.)

16 Afterword

After the overwhelming majority of this document was written in July 2014, OVMF development has obviously continued. As such, there is a need to recognize two significant code contributions from the community and as of January 2015, OVMF now runs on the "q35" machine type of QEMU, and it features a driver for Xen paravirtual block devices (and another for the underlying Xen bus).

Furthermore, a dedicated virtualization platform has been contributed to the ArmPlatformPkg package that plays a role parallel to the role of the OvmfPkg package. It targets the "virt" machine type of qemu-system-arm and qemu-system-aarch64. Parts of the OvmfPkg package are being refactored and modularized can be reused in the following platform description file: "ArmPlatformPkg/ArmVirtualizationPkg/ArmVirtualizationQemu.dsc".