UEFI Booting 64-bit Redhat Enterprise Linux 6

Finnbarr P. Murphy

(fpm@fpmurphy.com)

I recently decided to UEFI install the second beta of 64-bit Red Hat Enterprise Linux, version 6 (RHEL6 Beta 2) to see what was the current state of UEFI booting as far as Red Hat was concerned.

The platform I choose to use was an Intel DX48BT2 motherboard which has unofficial support for UEFI booting. By unofficial support, I mean that you can configure the firmware to support UEFI but, according to all reports that I have read, Intel will not help you if you encountered any problems. How a company like Intel can get away with such a policy is beyond my comprehension – especially since this lack of official support was not clearly stated in the warranty documentation or on the product packaging.

While the DX48BT2 supports UEFI booting, it does not have a built-in UEFI boot manager. To install RHEL6, I booted into a separate disk which contained a EFI shell and other assorted EFI utilities I have accumulated over the years. By placing the DVD containing the RHEL6 image in the DVD drive, and by using the EFI *map -r* command to update the list of devices and filesystems that the EFI shell recognized, I added the DVD drive to the EFI shell device list. I then loaded the *BOOTX64.EFI* image from the DVD drive and the install of RHEL6 proceeded smoothly and completed successfully.

Note that, when booting a DVD containing the RHEL6 bootable ISO image, very early in the boot process you are presented with a menu containing 2 options. I could not screen capture this menu because my equipment for doing video capture decided not work for some unknown reason that I still need to figure out. However the menu looks similar to this:

```
1.
2.

Select CD-ROM Boot Type:
```

I believe that this is the result of how the RHEL6 ISO is created. See Peter Jones page on Bootable CDs for BIOS and UEFI. I have not examined the build script but I strongly suspect that Red Hat is using the mechanism described by Jones to implement a dual booting DVD. Going back to the menu, select option 2, not option 1, to continue the UEFI install of 64-bit RHEL6. Red Hat really needs to improve this menu such that a non-technical user can understand what the menu is for and the purpose of each of the two options.

My first real problem occurred when system was rebooted after the install. It failed to boot into RHEL6. Nothing. Nada. I had no other operating systems on this platform. I had installed RHEL6 on the first disk. So what had gone wrong? I rebooted into the EFI shell on the second disk and looked around the first disk and quickly discovered why I could not boot RHEL6.

To explain the problem and understand the simple solution, you need to have some background knowledge of the UEFI booting process. With UEFI, bootstrapping an operating system on a platform requires a boot manager that is built in to the firmware. UEFI platforms do not rely upon bootstrap programs stored in boot records as used by Microsoft Windows and some other

operating systems. The firmware knows how to read GPT a partition table and understands a variant of the well-known FAT filesystem format. A designated partition, formatted with the variant of the FAT filesystem format and identified with a specific well-known GUID (Globally Unique Identifier), is known as the EFI System Partition (ESP).

The ESP contains the boot loader programs for one or more operating systems installed (in other partitions) on the platform, and may include device driver files for other devices, and system utility programs that are intended to be run before an operating system is booted. Boot loader programs, which are EFI executable programs that are loaded and run by the boot manager. By the way, the GUID for the ESP in the GUID Partition Table (GPT) is *C12A7328-F81F-11D2-BA4B-00A0C93EC93B*. Whether a disk contains an ESP or not is unrelated to the partition table scheme (GPT or MBR) that it uses. Note that Microsoft recommends that when partitioning a disk, the EFI System Partition be the first partition on the disk. However this is not mandated by the UEFI specification.

How a UEFI boot manager is actually implemented is totally up to the firmware implementer, but all boot managers are intended to be configurable using well-known EFI global variables, which are specified by the UEFI specification for holding firmware configuration data. To quote from the UEFI specification:

> The UEFI boot manager is a firmware policy engine that can be configured by modifying architecturally defined global NVRAM variables.

The boot manager attempts to load UEFI drivers, utilities and OS loaders in an order defined by these global variables. The boot manager must use the specified boot order but may add extra boot options or remove invalid boot options from the boot order list. Here is a list of the relevant EFI *boot* global variables.

Variable Name	Attribute	Description
Boot####	NV, BS, RT	A boot load option
BootOrder	NV, BS, RT	The ordered boot option load list
BootNext	NV, BS, RT	The boot option for the next boot only
BootCurrent	BS, RT	The boot option that was selected for the current boot
BootOptionSupport	BS,RT,RO	The types of boot options supported by the boot manager

This table defines the set of environmental (global) variables used by the boot manager together with an attribute that indicates when the variable may be accessed. Variables with an attribute of *NV* are nonvolatile. This means that their values are persistent across resets and power cycles. The value of any environment variable that does not have this attribute will be lost when power is removed from the system and the state of firmware reserved memory is not otherwise preserved. The variables with an attribute of *BS* are only available before *ExitBootServices* is called. This means that these environment variables can only be retrieved or modified in the preboot environment. They are not visible to an operating system. Environment variables with an attribute of *RT* are available before and after *ExitBootServices* is called. Environment variables of this type can be retrieved and modified in the preboot environment, and from an operating system.

Each *Boot####* variable consists of the prefix *Boot* followed by a unique four digit hexadecimal number, e.g. *Boot0001*, *Boot0002*, *Boot00D*, contains an *EFI_LOAD_OPTION*. The *BootOrder* variable contains an array of *UINT16* that make an ordered list of the *Boot####* variables. The first element in the array is the value for the first logical boot option, the second element is the value for the second logical boot option and so on. The *BootOrder* order list is used by the boot manager to determine the default boot order. The *BootNext* variable is a single UINT16 that defines the *Boot####* option that is to be tried first on the next boot. After the *BootNext* boot

option is tried the normal *BootOrder* list is used. To prevent loops, the boot manager deletes this variable before transferring control to the preselected boot option. The *BootCurrent* variable is a single UINT16 that defines the *Boot####* option that was selected on the current boot. The *BootOptionSupport* variable is a UINT32 that defines the types of boot options supported by the boot manager. Note that there is an analogous set of variables global variables whose prefix is *Driver* instead of *Boot* defined in the UEFI specification but these appear to be rarely used and will not be discussed any further in this post.

Boot managers are required to inspect the *BootOrder* variable for a list of boot options. A boot option contains the device and path name of an EFI executable program to load and run, and a set of parameters to pass to that program. If there is no *BootOrder* variable, a boot manager is required to enumerate all removable and fixed disks. For removable disks, the disk is treated as containing a single volume. For fixed disks, each disk is searched for an ESP and if found that disk is used. In both cases, the boot manager looks for an EFI executable program called *\EFI\BOOT\BOOTtype.EFI* (where type is IA32, X64, or some other such string denoting the machine architecture) to load and run. This is a default operating system boot loader.

EFI executable programs are standalone programs that use only machine firmware services and that do not require an operating system in order to run. They can be either operating system boot loaders, a shell or utilities. By convention, all of the boot loaders for operating systems are stored in the ESD in a vendor-specific subdirectory of the *\EFI* directory. It is important to note that vendor specific directories are not mandated by, or even mentioned, in the UEFI specification. Rather they originated as part of the Itanium DIG64 specification with control of the what came to be know as the EFI System Partition Subdirectory Registry subsequently transferred to UEFI.

The default state of globally-defined variables is firmware vendor specific. However the UEFI specification mandates a default boot scheme where, for whatever reason, the *BootOrder* variable does not exist or points to nonexistent boot options. If this occurs, the boot manager will enumerate all removable media devices followed by all fixed media devices. The order within each group of devices is undefined. The boot manager will attempt to boot from each device. If the device supports the *EFI_SIMPLE_FILE_SYSTEM_PROTOCOL* then the removable media boot behaviour is executed. Otherwise, the firmware will attempt to boot the device via the *EFI_LOAD_FILE_PROTOCOL*. A UEFI protocol is essentially a block of function pointers and data structures, similar to a class in object orientated programming. Protocols are indentified by GUIDs. See the UEFI specification for an detailed explanation of these two particular protocols if you are interested in learning more about them. It is expected that this default boot will either load an operating system, an EFI shell or a maintenance utility. If the loaded item is an operating system setup program it is then responsible for setting the requisite environment variables for subsequent boots.

UEFI can boot from a device using either the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL or the EFI_LOAD_FILE_PROTOCOL. A device that supports the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL must materialize a file system protocol for that device to be bootable. If a device does not wish to support a complete file system it may instead produce an EFI_LOAD_FILE_PROTOCOL which allows it to materialize an image directly. The boot manager will attempt to boot using the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL first. If that fails, then the EFI_LOAD_FILE_PROTOCOL will be used.

When booting via the *EFI_SIMPLE_FILE_SYSTEM_PROTOCOL*, the *FilePath* starts with a device path that points to the device that “speaks” the *EFI_SIMPLE_FILE_SYSTEM_PROTOCOL*. The next part of *FilePath* points to the file name, including sub directories that contain the bootable image. If the file name is a null device path, the file name must be discovered on the media using the rules defined in the UEFI specification for removable media devices with ambiguous file names.

When booting via the *EFI_LOAD_FILE_PROTOCOL* protocol, the *FilePath* is a device path that

points to a device that uses the *EFI_LOAD_FILE_PROTOCOL*. The image is loaded directly from the device using *EFI_LOAD_FILE_PROTOCOL*. The remainder of *FilePath* contains information that is specific to the device. The firmware passes this device-specific data to the loaded image, but does not use it to load the image. If the remainder of *FilePath* is a null device path it is the loaded image's responsibility to implement a policy to find the correct boot device. Note that the *EFI_LOAD_FILE_PROTOCOL* is used for devices that do not directly support file systems. Network devices, where the image is materialized without the need for a file system, typically used this method.

On a removable media devices such as a DVD reader it is not possible for *FilePath* to contain a file name, including sub directories. In this case the UEFI firmware will attempt to boot from the removable media by adding a default file name in the form \EFI\BOOT\BOOT{machine-type-short-name}.EFI to the *FilePath* where *machine-type-short-name defines a PE32+ image format architecture. Each file only contains one UEFI image type, and a platform may support booting from one or more images types. Here is a list of the default file names defined in the current UEFI specification (version 2.3) :*

Architecture	File Name Convention	PE Execution Machine Type
IA32	BOOTIA32.EFI	0x14c
X64	BOOTX64.EFI	0x8664
IA64 (Itanium)	BOOTIA64.EFI	0x200
ARM	BOOTARM.EFI	0x01c2

The PE Executable machine type is contained in the machine field of the COFF file header as defined in the Microsoft Portable Executable and Common Object File Format Specification, Revision 6.0.

It is expected that on a non-removable media device such as a hard disk, a complete *FilePath* which includes sub directories and a file name can be used for the boot target and the platform will boot using this *FilePath*. However,in the case where all the *Boot####* variables that are referenced in the *BootOrder* variable point to devices that are not present, the boot devices have timed out, the specific boot file did not exist, or there was no valid boot variable, default boot processing behaviour may optionally occur. This default boot processing will consist of the boot manager searching non-removable media that supports the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL or EFI_BLOCK_IO_PROTOCOL. In general the boot manager will search all candidate media but policy may optionally limit the search to a subset of all possible devices connected to a given system; choices for such policy limits are implementation specific.

If the device supports the *EFI_SIMPLE_FILE_SYSTEM_PROTOCOL* layered on an ESB, then the system firmware will attempt to boot from the media by executing a default file name in the form *\EFI\BOOT\BOOT{machine type short-name}.EFI* where machine type short-name defines a PE32+ image. A device may support multiple architectures by simply having a *\EFI\BOOT\BOOT{machine type short-name}.EFI* image file for each supported machine type.

The reason why RHEL6 was not booting on the DX48BT2 quickly became clear. The firmware on the DX48BT2 only supports the *BootCurrent* global variable. The install script correctly created the ESP but only placed the GNU/Linux bootloader (*grub.efi*) in /EFI/redhat:

```
# ls -al /boot/efi/EFI/redhat
drwx------. 2 root root   4096 Aug 18 15:31 .
drwx------. 4 root root   4096 Aug 17 22:03 ..
-rwx------. 1 root root   1222 Aug 18 15:31 grub.conf
-rwx------. 1 root root 226825 Nov 20  2009 grub.efi
```


The solution was to create a /EFI/BOOT subdirectory and copy *grub.efi* and *grub.conf* into that subdirectory as *BOOTX64.EFI* and *BOOTX64.CONF* respectively.

```
# ls -al /boot/efi/EFI/BOOT
drwx------. 2 root root   4096 Aug 17 22:04 .
drwx------. 4 root root   4096 Aug 17 22:03 ..
-rwx------. 1 root root    840 Aug 17 22:03 BOOTX64.CONF
-rwx------. 1 root root 226825 Aug 17 22:03 BOOTX64.EFI
```

This change ensured that the UEFI firmware was able to find a boot loader using the default boot mechanism and successfully load RHEL6.

Red Hat should modify their install script to check to see if */EFI/BOOT/BOOTX64.EFI* exists and, if not, create the subdirectory and default boot loader. It is not enough to assume that the global boot variables specified by UEFI exist and are modifiable by a user; the installation should ensure that a default UEFI boot mechanism always exists. This will provide a better user experience and less calls to customer support.

This hack, and yes I call it a hack, should not be necessary in a few years but certainly is at present as implementations of UEFI firmware vary in their interpretation of the requirements of the UEFI specification. This is not necessarily the fault of the UEFI firmware developers. The UEFI specification is vague (probably deliberately so) in a number of areas relating to UEFI booting and should be tightened up.

