UEFI

From OSDev Wiki

(U)EFI or (Unified) Extensible Firmware Interface is a specification that defines a software interface between the operating system(s) and the platform's firmware. In the mid 90s Intel was creating a new processor architecture that was 64-bit, but wasn't backwards-compatible with the old x86. This architecture was the Itanium 64. Because the IA-64 only supports 64-bit instructions, the PC BIOS couldn't be used, therefore Intel developed the EFI specification. Later on this specification was managed (and still is) by the UEFI board, an association of several companies such as AMD, Microsoft, Intel, Apple and so on.

Contents

Support

The (U)EFI specification is currently defined for the Itanium platform, the x86-platform (64-bit included) and the ARM platform. Since Apple's movement with Mac OS X from the PowerPC architecture to the x86-architecture, they have been using EFI, although with minor differences here and there. Most modern operating systems, boot loaders and boot managers support (U)EFI as well nowadays. Most 64-bit versions of Microsoft Windows, FreeBSD, Apple Mac OS X and Linux all have support for UEFI to some extent. As for boot loaders and managers you generally have (E)GRUB and ELILO. As for emulators and virtual machines, you can generally use VirtualBox and Qemu.

Developing for (U)EFI

One can develop applications, boot loaders or drivers for (U)EFI if he/she has the required hardware or software to develop for it. As said before you can use VirtualBox and Qemu as both of them seem to have a (U)EFI implementation of some sort. Another option is to use Intel's TianoCore, which is supposedly their own (U)EFI firmware interface implementation to run on top of the PC BIOS and/or as a Coreboot payload. The simplest way is by getting the right hardware. Generally there are several motherboards available that have an (U)EFI implementation, sometimes with a PC BIOS implementation next to it. Below is a list of noticeable BIOS companies:

- AMI (Aptio).
- Phoenix (SecureCore, TrustedCore, AwardCore).
- Insyde (InsydeH20).

Apple macs are also well known to implement EFI 1.0 (without the U, which indicates 2.0 and later)

Tools

The (U)EFI Development Kit, the (U)EFI Toolkit and the (U)EFI specifications might be interesting to use. As for writing actual (U)EFI software you can use several compilers such as GCC and Microsoft Visual Studio for C and FASM for Assembly. The (U)EFI Toolkit might be interesting when using a C compiler, as it contains several C/C++ headers.

Note that the C language is incompatible with EBC (EFI Byte Code) and thus a special compiler supporting the EFI C dialect is required. GCC is not one such compiler.

Booting

The (U)EFI system does not look for a boot sector, instead it looks for a file located on a FAT formatted disk. Other filesystems may be supported.

The system looks for a file in an approved directory. Normally, there are a bunch of "UEFI variables" stored in the computer's NVRAM that specify the path of the file that must be loaded as an UEFI application. Actually, the NVRAM stores multiple bootable entries in it, each one with it's own label, just as a menu-driven boot loader would do.

But that's not everything, because there is some incompatible behaviour. Some (U)EFI implementations just boot from the file /EFI/BOOT/BOOTX64.EFI. In the case of Apple, apple has modified the firmware to support HFS+. In apple computers a file is "Blessed" with the ability to boot from a certain file.

Tools to configure the beast

<biased mode!>UEFI is an anti-standard, meaning that it's promoters invented it so that it looked like they were creating a more flexible technology, while in reality they just managed to create something even more fragile than the legacy BIOSes ever managed to be.</biased mode!>

In practise the right tool to configure the beast depends on your specific system. What there is is the following:

- **efibootmgr**, Linux command line utility that configs UEFI boot entries using efivarfs kernel

module. Type *efibootmgr* to see current entries.
- **bless** utility, which chooses the "blessed" file in Macintosh systems.
- **UEFI Shell**, which is built in some firmwares but not in others. The shell command used to configure boot entries is "bcfg". Type *bcfg boot dump -v* in shell to see current entries.
- Others

And what about a nice boot menu provided by the firmware?

At first I thought that this menu existed. However, in rEFInd Boot Manager's website there is the explanation you were waiting for:

```
In theory, EFI implementations should provide boot managers. Unfortunately, in practice
these boot managers are often so poor as to be useless. The worst I've personally
encountered is on Gigabyte's Hybrid EFI, which provides you with no boot options whatsoever,
beyond choosing the boot device (hard disk vs. optical disc, for instance). I've heard of
others that are just as bad. For this reason, a good EFI boot manager — either standalone or
as part of a boot loader — is a practical necessity for multi-booting on an EFI computer.
That's where rEFIt and rEFInd come into play.
http://www.rodsbooks.com/refind/
```

Using GNU toolchain for compiling and debugging EFI applications

GNU development toolchain consist of binutils package (assembler, linker, various utilities for manipulating executable images), GCC compiler, GNU make and GDB debugger. Advantages are obvious - build system is very flexible, the tools have very rich functionality, they are free and open source, your EFI application code can be easily integrated to any project. Most of open source software prefers this way.

Building tools

The first step is to compile your tools with required options. Firstly you need to compile binutils package. Obtain the latest from http://ftp.gnu.org/gnu/binutils/. You will need to enable "i386-efi-pe" target to have support for EFI PE+ executable format. If you use the same toolchain for compiling something else in your project (e.g. OS kernel) you can also specify required target (e.g. x86_64-myOS-freebsd) Note that it is BFD target, not the target platform. If you want to develop for 64-bits platform add "--enable-64-bit-bfd" options to "configure" script.

```
../src/configure --prefix=$PREFIX --target=$TARGET --disable-nls --build=
    --enable-targets=i386-efi-pe,x86_64-phoenix-freebsd
make all
make install
```

Here and below $PREFIX variable points to your prefix where you want to install your build tools (e.g. "export PREFIX=/home/John/projects/myOS/build-tools") and $TARGET is you target platform (e.g. "x86_64-myOS-elf").

If your build machine has not the same architecture as target platform you will need a cross compiler. There are instructions about compiling GCC for cross platform development. Just use binutils compiled above with these instructions.

You will need a debugger for debugging your applications. Obtain it from http://ftp.gnu.org/gnu/gdb/ and compile:

```
./configure --prefix=$PREFIX --target=$TARGET --enable-64-bit-bfd
```

Now you are ready to compile your first EFI application. Download gnu-efi package from https://sourceforge.net/projects/gnu-efi/ and read its README files. Follow the instructions there, check Makefiles are pointing to your build tools and have correct architecture specified. Run "make" command and you will get several sample applications in "apps" directory. We will describe how to run them a bit later but for now you need to examine the build log and notice all commands which where executed and all required options for them. This will be a basis for your Makefile if you will wish to integrate EFI application to your project. Several pieces of advice:

- If you build a single binary in your project (e.g. OS loader) you will not need to make static libraries like it is done in the gnu-efi package. Just compile all required C and Assembler file and link them together in the final executable file.
- If you project has only one target platform you can throw away unnecessary source files. Just select for gnu-efi build log all files which where compiled and throw away all the others.

Here (http://ast-phoenix.git.sourceforge.net/git/gitweb.cgi?p=ast-phoenix/ast-phoenix;a=tree;f=kernel/boot;h=646835063c1362732f209c1312ce0d2ba7b558a5;hb=HEAD) is an example of the package integration. Pay attention to the Makefile. We will touch some its aspects later.

Now we need to run the resulted application(s) and somehow debug it. Qemu virtual machine is a good choice because of its rich built-in debugging functionality.

Download and compile it. Specify your target platform. You can use "--enable-kvm" option to significantly increase emulation speed if you have Linux kernel and kvm package installed.

```
./configure --prefix=$PREFIX --target-list=x86_64-softmmu --enable-kvm
make
make install
```

Qemu does not have EFI firmware so you need to download it separately. You can use OVMF firmware based on TianoCore from http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=OVMF. Download 32 or 64 bits version depending on your target platform. Create some directory $YOUR_PREFIX/share/qemu/myOS and place there "vgabios-cirrus.bin" and "OVMF.fd" binaries from the OVMF package.

Making a disk image manually

The last thing left is a disk image with EFI system partition and our application there. EFI requires GUID partitions table and FAT32 filesystem for EFI system partition. We will need gdisk utility from http://sourceforge.net/projects/gptfdisk/ for GUID partitions table creation.

Select some reasonable size of your disk. Below I am assuming that $BYTES_PER_SECTOR is number of bytes per sector on your disk (typically 512) and $NUM_SECTORS is total disk size expressed in sectors. Firstly create disk image initial file filled with zeroes:

```
export filename=$PREFIX/share/qemu/myOS/myOS.disk
dd if=/dev/zero of=$filename bs=$BYTES_PER_SECTOR count=$NUM_SECTORS
```

After that create partition table by gdisk:

```
gdisk $filename
```

It has interface similar to fdisk utility. Use "o" command to create new partition table, "n" for new partition with default parameters to use the whole disk (partition type "ef00" for EFI system partition), "w" to write all changes and exit. Now you have disk image with GUID partition table on it but the partition is still unformatted. We will use Linux loopback device to access the file as block device. We need to know exact position of the partition on the disk:

```
gdisk -l $filename
```

This command will output list of partitions and their first and last sectors indexes. If you have several partitions then just use numbers for your one. Let's assume $START is the first sector index * bytes per sector and $SIZE size is the partition size in bytes. Associate your image file with the loopback device:

```
losetup --offset $start --sizelimit $size /dev/loop0 $filename
```

So now we can create filesystem there. We need FAT for EFI system partition. You can use FAT12 if your partition size is small to prevent from big space overhead from larger FAT filesystems:

```
mkdosfs -F 12 /dev/loop0
```

Now you can mount your partition to some mount point:

```
mkdir /tmp/myOsDisk
mount /dev/loop0 /tmp/myOsDisk
```

Just copy all files you need there (e.g. your EFI application executable image). You can create "stratup.nsh" script which will be executed automatically if no other options are configured in EFI built-in boot-manager. Script could contain just your file launching command with required parameters, e.g. "fs0:\\efi\\boot\\myOsLoader some parameters here". According to specification you can create "/EFI/BOOT/BOOTx64.EFI" file which will be loaded automatically. Finally unmount the partition and release the loopback device:

```
umount /tmp/myOsDisk
losetup -d /dev/loop0
```

Your disk image is ready and you can finally launch emulation. Obviously creating disk image could be easily automated in order to not execute these actions manually each time. Automation example can be found there (http://ast-phoenix.sourceforge.net/doc/doku.php?id=athena:project:phoenix:dev_env#disk_image) .

Making a disk image (alternative version without root)

You can also get a few tools that do not rely on having certain features in a Linux kernel. Instead, we make use of dd, mtools and parted rather than gdisk and loopback devices. These can be automated into a relatively brief makefile:

```
FAT_SECTORS:=65536
DISK_SECTORS:=69632
DISK_START:=2048
DISK_END:=67584

TEMP_IMG := temp.img
PARTED := /usr/sbin/parted
PARTED_PARAMS := -s -a minimal

partition.img: $(FILES)
        dd if=/dev/zero of=$(TEMP_IMG) bs=512 count=$(FAT_SECTORS)
        mformat -i $(TEMP_IMG) -h 32 -t 32 -n 64 -c 1 ::
        mcopy -i $(TEMP_IMG) $(FILES) ::
        cp $(TEMP_IMG) $@
        rm $(TEMP_IMG)

hd.img: partition.img
        dd if=/dev/zero of=$(TEMP_IMG) bs=512 count=$(DISK_SECTORS)
        $(PARTED) $(TEMP_IMG) $(PARTED_PARAMS) mklabel gpt
        $(PARTED) $(TEMP_IMG) $(PARTED_PARAMS) mkpart EFI FAT16 $(DISK_S1
        $(PARTED) $(TEMP_IMG) $(PARTED_PARAMS) toggle 1 boot
        dd if=$(BUILDROOT)/partition.img of=$(TEMP_IMG) bs=512 obs=512 cc
        cp $(TEMP_IMG) $@
        rm $(TEMP_IMG)
```

Some firmwares only work with FAT32, which means that our disk image should have more clusters than FAT16 allows. Therefore we pick 2^{16} sectors as it doesn't fit into a 16-bit number, and make clusters 1 sector in size, which makes the end result close to the shortest possible disk size. Since the bootblocks and partitioning structures need some space of their own, we reserve some space at either end for the total image, here set to 1MB on each end. All the interesting tools here won't create files, so we have to fill them in in advance. A simple dd command can do that.

mformat works perfectly fine on images, but needs manually passed geometry information for performing a format (as it ends up in the FAT structures) Heads, cylinders, and sectors per track are calculated so that they make the entire disk in total ($32 * 32 * 64 = 2^{16}$), and we tell it to set the sectors per cluster to one.

Parted works perfectly fine on files, although it will often complain you're doing it wrong if you try. Commands normally performed on the built-in prompt can be passed on the command line. The sequence is to make a GPT disk, create a FAT partition on it (with the sizes specified), then set the bootable flag on the first partition. Note that the command actually toggles the flag, so repeating that command on the same image file is generally a bad idea.

For best use, parted needs a few extra parameters. Pass -s (script) so that it doesn't ask for confirmation on anything - we know what we're doing, and we also set -a (alignment) minimal so that it doesn't magically try to guess the cylinder size, leaving no complaints about our partition boundaries or attempts to shift them for a "better" fit, although the explicit passing of sectors as unit should prevent the latter behaviour from occurring just as well.

Finally, we use a typical dd command to drop the partition contents into the centre of the partitioned disk.

Note that both steps have an intermediate .img file. This covers the case where something goes wrong, and you are left with an file named as the output with the current timestamp, which would cause make to think it's up to date when you retry (when it's actually corrupt), giving you some weird bugs to hunt for later.

Running the emulator

There are some advices for emulation running:

- Use "-serial" option to have serial console available for the virtual machine. You will have console logs in your terminal and a possibility to use simple ports writing to output debug tracing to serial console.
- Use "-s" option to enable built-in GDB stab which will wait for connection on TCP port 1234.

Launch Qemu providing path to the directory where your firmware binaries are located:

```
$PREFIX/bin/qemu-system-x86_64 -L $PREFIX/share/qemu/myOS -bios OVMF.fd -
    -vga cirrus -monitor stdio -serial tcp::666,server -s -hdb $PREFIX/sh
```

Qemu will start and wait for incoming connection to serial console. In the example above it waits on TCP port 666. You can use, for example, socat utility to connect:

```
socat -,raw,echo=0 tcp4:localhost:666
```

Once connected the emulation will start. You can use EFI shell command to navigate through filesystems, output system information or launch your application. It has help for all commands so refer to it for details.

Sample application

The next important question is the application debugging. The first moment is that the EFI application should be stopped at some point and wait for debugger. The simplest way to do this is to insert some endless loop in your application. The loop can be enclosed in the block which is executed, for example, when your application receives "--debug" option in its arguments. Let's assume you have inserted such code:

```
EFI_STATUS
efi_main (EFI_HANDLE image, EFI_SYSTEM_TABLE *systab)
{
    EFI_LOADED_IMAGE *loaded_image = NULL;
    EFI_STATUS status;

    InitializeLib(image, systab);
    status = uefi_call_wrapper(systab->BootServices->HandleProtocol,
                               3,
                              image,
                              &LoadedImageProtocol,
                              (void **)&loaded_image);
    if (EFI_ERROR(status)) {
        Print(L"handleprotocol: %r\n", status);
    }

    Print(L"Image base: 0x%lx\n", loaded_image->ImageBase);

    int wait = 1;
    while (wait) {
        __asm__ __volatile__("pause");
    }

    return EFI_SUCCESS;
}
```

When this code will be executed "pause" instruction will be executed in the loop.

The next thing required for GDB is executable image with symbols. If you carefully examined build log and Makefiles you should note that when EFI executable is created from ELF shared object file only limited set of sections are copied to the resulted image:

```
.text .sdata .data .dynamic .dynsym .rel .rela .reloc
```

For having debug symbols we need additionally these sections (in case you have compiled files with "-ggdb" option):

```
.debug_info .debug_abbrev .debug_loc .debug_aranges .debug_line .debug_ma
```

But if you create EFI binary which additionally contains these sections the EFI firmware will be unable to launch it. Fortunately, we do not need the file with debug symbols on the target machine since we will use remote debugging anyway. So what you need is to create two EFI binaries - one with only required sections to upload it to target system and another one with debug symbols to use it with GDB. Actually you just need to run objcopy utility twice with different set of sections to copy and different output files. See Makefile example there (http://ast-phoenix.git.sourceforge.net/git/gitweb.cgi?p=ast-phoenix/ast-phoenix;a=tree;f=kernel/boot;h=646835063c1362732f209c1312ce0d2ba7b558a5;hb=HEAD) .

Now you can launch GDB. You need to specify some file to use as the target binary - you can specify EFI binary with debug symbols but it will have no sense for debugging because the image will be relocated to different address. Note that in the example code above the actual image base address is output. It is required to properly load file with symbols. Let's say after you have launched your application it provided this output:

```
Image base: 0x2EE30000
```

So now you need to start GDB, connect to local TCP port 1234 where Qemu is waiting for GDB connection and load image with symbols to relocated address. We need to specify relocated addresses for .text and .data sections. Their addresses in non-relocated binary should be added to image base which is provided in the output above:

```
# gdb myOS.efi
GNU gdb (GDB) 7.3
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.h
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/John/myOS/source/kernel/boot/build/DEBUG/myOS.
(gdb) info files
Symbols from "/home/John/myOS/source/kernel/boot/build/DEBUG/myOS.efi".
Local exec file:
        `/home/John/myOS/source/kernel/boot/build/DEBUG/myOS.efi', file t
        Entry point: 0x3000
        0x0000000000003000 - 0x000000000000b9ce is .text
        0x000000000000b9ce - 0x000000000000b9d8 is .reloc
        0x000000000000c000 - 0x000000000000e148 is .data
        0x000000000000f000 - 0x000000000000f0f0 is .dynamic
        0x0000000000010000 - 0x0000000000011098 is .rela
        0x0000000000012000 - 0x0000000000013788 is .dynsym
(gdb) file
No executable file now.
No symbol file now.
(gdb) add-symbol-file debug.myOS.efi 0x2EE33000 -s .data 0x2EE3c000
add symbol table from file "debug.myOS.efi" at
        .text_addr = 0x2ee33000
```

```
        .data_addr = 0x2ee3c000
(y or n) y
Reading symbols from /home/John/myOS/source/kernel/boot/build/DEBUG/debug
(gdb) set architecture i386:x86-64:intel
The target architecture is assumed to be i386:x86-64:intel
(gdb) target remote :1234
Remote debugging using :1234
WaitDebugger () at loader/main.c:80
80          while (wait) {
(gdb) set variable wait = 0
```

We need to unload executable binary by "file" command after sections layout is displayed because otherwise its symbols will override debug symbols loaded by "add-symbol-file" command (at least for data section). You do not need to load it each time because sections addresses will change only after next recompilation. Alternatively "objdump" utility can be used to dump sections. As you can see after setup is done you can normally debug your application using whole power of the GDB. Set your "wait" variable to zero and you will exit from endless loop. Set breakpoints/watchpoints, continue execution, enjoy debugging!

Binary Format

(U)EFI generally uses the PE-executable format, with its very own subtypes. Every (U)EFI application is basically a DLL without symbol tables et al, and another subtypes:

- (U)EFI application (10).
- (U)EFI boot service driver (11).
- (U)EFI run-time driver (12).

Calling Conventions

The EFI specifications specify the calling conventions for 32-bit 80x86 and Itanium. The (later) UEFI specifications define the calling conventions for 32-bit 80x86, Itanium and 64-bit 80x86

For 32-bit 80x86, the calling convention used is the standard C calling convention. For Itanium, the calling convention is defined in the "Intel(R) Itanium(R) System Abstraction Layer Specification".

For 64-bit 80x86, Microsoft's x64 calling convention is used. This calling convention requires the stack be aligned on a 16-byte boundary (from the callee's perspective), and that a 32-byte "shadow space" be reserved on the stack (immediately above the return RIP). The shadow space is not described by the UEFI specifications. This can lead to frustrated assembly language programmers (who are writing code based on the UEFI specifications alone and don't know about the shadow space, and therefore waste **hours** trying to figure out why their stack gets trashed by some EFI functions).

Example in C

Below is an example of an EFI application written in C that displays: "Hello World".

```
#include <efi.h>
```

```
#include <efilib.h>

EFI_STATUS efi_main(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable
{
  SIMPLE_TEXT_OUTPUT_INTERFACE *conout;
  InitializeLib(ImageHandle, SystemTable);
  conout = SystemTable->ConOut;

  uefi_call_wrapper(conout->OutputString, 2, conout, (CHAR16 *)L"Hello Wo

  return EFI_SUCCESS;
}
```

Example in FASM

Below is an example of an EFI application written in x86 assembly (to be assembled with FASM) that displays: "Hello World".

```
format pe64 dll efi
entry main

section '.text' code executable readable

include 'efi.inc'

main:
    sub rsp, 32                         ; Reserve shadow space
    mov [Handle], rcx                   ; ImageHandle
    mov [SystemTable], rdx              ; Pointer to SystemTable.
    lea rdx, [_hello]
    mov rcx, [SystemTable]
    mov rcx, [rcx + EFI_SYSTEM_TABLE.ConOut]
    call [rcx + SIMPLE_TEXT_OUTPUT_INTERFACE.OutputString]
    add rsp, 32                         ; Free shadow space
    mov eax, EFI_SUCCESS
    retn

section '.data' data readable writeable

Handle                                  dq ?
SystemTable                             dq ?
_hello                                  db 'Hello World',13,10,0

section '.reloc' fixups data discardable
```

Instead of efi.inc, you can use our own uefi.inc which is a lot more friendlier, and aimed at boot loader purposes.

```
format pe64 dll efi
entry main

section '.text' code executable readable

include 'uefi.inc'

main:
    ; initialize UEFI library
    InitializeLib
    jc @f

    ; call uefi function to print to screen
    uefi_call_wrapper ConOut, OutputString, ConOut, _hello

@@: mov eax, EFI_SUCCESS
    retn

section '.data' data readable writeable

_hello                                  db 'Hello World',13,10,0

section '.reloc' fixups data discardable
```

Common Problems

My bootloader hangs/resets after about 5 minutes

When control is handed to your EFI application by the firmware boot manager, it sets a watchdog timer for 5 minutes, after which the firmware is reinvoked as it assumes your application has hung. The firmware in this case will normally try to reset the system (although the OVMF firmware in VirtualBox simply causes the screen to go black and hang). To counteract this, you are required to refresh the watchdog timer before it times out. Alternatively, you can disable it completely with code like

```
BS->SetWatchdogTimer(0, 0, 0, NULL);
```

Obviously this is not a problem for most bootloaders, but can cause an issue if you have an interactive loader which waits for user input. Also note that you are required to disable the watchdog timer if you exit back to the firmware.

My bootloader hangs if I use user defined EFI_MEMORY_TYPE values

For the memory management functions in EFI, an OS is meant to be able to use "memory type" values above 0x80000000 for its own purposes. In the OVFM EFI firmware release "r11337" (for Qemu, etc) there is a bug where the firmware assumes the memory type is within the range of values defined for EFI's own use, and uses the memory type as an array index. The end result is an "array index out of bounds" bug; where the higher memory type values (e.g. perfectly legal values above 0x80000000)

cause the 64-bit version of the firmware to crash (page fault), and cause incorrect "attribute" values to be reported by the 32-bit version of the firmware. This same bug is also present in whatever version of the EFI firmware VirtualBox uses (which looks like an older version of OVFM); and I suspect (but don't know) that the bug may be present in a wide variety of firmware that was derived from the tianocore project (not just OVFM). Brendan 15:30, 29 July 2011 (UTC)

See also

UEFI Bare Bones

Wikipedia

- EFI.

External Links

- Intel EFI specifications et al. (http://www.intel.com/technology/efi)
- UEFI specifications et al. (http://www.uefi.org/home)
- Several articles about UEFI (http://x86asm.net/articles)
- PE specification covering the (U)EFI binary format (http://www.microsoft.com/whdc/system/platform/firmware/pecoff.mspx)
- Blog about UEFI, with bits about UEFI development (http://uefi.blogspot.com/)
- UEFI Development Kit 2014 (http://tianocore.github.io/)
- UEFI firmware images (http://tianocore.github.io/ovmf/) for use with QEMU
- Presentation guiding through simple UEFI application setup (http://internshipatdell.wikispaces.com/file/view/How+to+build+an+UEFI+application.pptx)
- Presentation giving an overview of windows uefi booting (http://www.uefi.org/sites/default/files/resources/UEFI-Plugfest-WindowsBootEnvironment.pdf)

Categories: X86 | X86-64 | IA-64 | ARM | Firmware
