EFI System Partition

The **EFI System Partition** (also called ESP or EFISYS) is a FAT32 formatted physical partition (in the main partition table of the disk, not under LVM or software RAID etc.) from where the **UEFI** firmware launches the UEFI bootloader and application.

It is an OS independent partition that acts as the storage place for the EFI bootloaders and applications to be launched by the EFI firmware. It is mandatory for UEFI boot.

Warning: If **dual-booting** with an existing installation of Windows on a UEFI/GPT system, avoid reformatting the UEFI partition, as this includes the Windows *.efi* file required to boot it. In other words, use the existing partition as is and simply **#Mount the partition**.

Contents

Create the partition

The following two sections show how to create an EFI System Partition (ESP).

Note: It is recommended to use **GPT** for UEFI boot, because some UEFI firmwares do not allow UEFI/MBR boot.

To avoid potential problems with some EFIs, ESP size should be at least 512 MiB. 550 MiB is recommended to avoid confusion with FAT16 **[1] (http://www.rodsbooks.com/efi-bootloaders/principles.html)**, although larger sizes are fine.

According to a Microsoft note[2] (https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/configure-uefigpt-based-hard-drive-partitions#diskpartitionrules), the minimum size for the EFI System Partition (ESP) would be 100 MiB, though this is not stated in the UEFI Specification. Note that for **Advanced Format** 4K Native drives (4-KiB-per-sector) drives, the size is at least 256 MiB, because it is the minimum partition size of FAT32 drives (calculated as sector size (4KiB) x 65527 = 256 MiB), due to a limitation of the FAT32 file format.

GPT partitioned disks

Choose one of the following methods to create an ESP for a GPT partitioned disk:

- **fdisk/gdisk**: Create a partition with partition type EFI System (`EFI System` in *fdisk* or `EF00` in *gdisk*). Proceed to **#Format the partition** section below.
- **GNU Parted**: Create a FAT32 partition and in Parted set/activate the `boot` flag (**not** `legacy_boot` flag) on that partition. Proceed to **#Mount the partition** section below.

MBR partitioned disks

Create a partition with partition type *EFI System* using fdisk. Proceed to **#Format the partition**.

Format the partition

After creating the ESP, you **must** **format** it as **FAT32**:

```
# mkfs.fat -F32 /dev/sdxY
```

If you used GNU Parted above, it should already be formatted.

If you get the message `WARNING: Not enough clusters for a 32 bit FAT!`, reduce cluster size with `mkfs.fat -s2 -F32 ...` or `-s1`; otherwise the partition may be unreadable by UEFI. See **mkfs.fat(8) (https://jlk.fjfi.cvut.cz/arch/manpages/man/mkfs.fat.8)** for supported cluster sizes.

Mount the partition

The kernels and initramfs files need to be accessible by the **boot loader** or UEFI itself to successfully boot the system. Thus if you want to keep the setup simple, your boot loader choice limits the available mount points for EFI System Partition.

The simplest scenarios for mounting EFI System Partition are:

- **mount** ESP to `/boot/efi` and use a **boot loader** which has a driver for your root file system (eg. **GRUB**, **rEFInd**).
- **mount** ESP to `/boot`. This is the preferred method when directly booting a **EFISTUB** kernel from UEFI.

Alternative mount points

If you do not use one of the simple methods from **#Mount the partition**, you will need to copy your boot files to ESP (referred to hereafter as `esp`).

```
# mkdir -p esp/EFI/arch
# cp -a /boot/vmlinuz-linux esp/EFI/arch/
# cp -a /boot/initramfs-linux.img esp/EFI/arch/
# cp -a /boot/initramfs-linux-fallback.img esp/EFI/arch/
```

Note: When using an Intel CPU, you may need to copy the **Microcode** to the boot-entry location.

Furthermore, you will need to keep the files on the ESP up-to-date with later kernel updates. Failure to do so could result in an unbootable system. The following sections discuss several mechanisms for automating it.

Using bind mount

Instead of mounting the ESP itself to `/boot`, you can mount a directory of the ESP to `/boot` using a bind **mount** (see **mount(8) (https://jlk.fjfi.cvut.cz/arch/manpages/man/mount.8)**). This allows **pacman** to update the kernel directly while keeping the ESP organized to your liking.

Note:

- This requires a kernel and bootloader compatible with FAT32. This is not an issue for a regular Arch install, but could be problematic for other distributions (namely those that require symlinks in `/boot/`). See the forum post **here (https://bbs.archlinux.org/viewtopic.php?pid=1331867#p1331867)**.
- You *must* use the `root=` **kernel parameter** in order to boot using this method.

Just like in **#Alternative mount points**, copy all boot files to a directory on your ESP, but mount the ESP **outside** `/boot`. Then bind mount the directory:

```
# mount --bind esp/EFI/arch /boot
```

After verifying success, edit your **Fstab** to make the changes persistent:

```
/etc/fstab
```

```
esp/EFI/arch /boot none defaults,bind 0 0
```

Using systemd

Systemd features event triggered tasks. In this particular case, the ability to detect a change in path is used to sync the EFISTUB kernel and initramfs files when they are updated in `/boot/`. The file watched for changes is `initramfs-linux-fallback.img` since this is the

last file built by mkinitcpio, to make sure all files have been built before starting the copy. The *systemd* path and service files to be created are:

```
/etc/systemd/system/efistub-update.path
```

```
[Unit]
Description=Copy EFISTUB Kernel to EFI System Partition

[Path]
PathChanged=/boot/initramfs-linux-fallback.img

[Install]
WantedBy=multi-user.target
WantedBy=system-update.target
```

```
/etc/systemd/system/efistub-update.service
```

```
[Unit]
Description=Copy EFISTUB Kernel to EFI System Partition

[Service]
Type=oneshot
ExecStart=/usr/bin/cp -af /boot/vmlinuz-linux esp/EFI/arch/
ExecStart=/usr/bin/cp -af /boot/initramfs-linux.img esp/EFI/arch/
ExecStart=/usr/bin/cp -af /boot/initramfs-linux-fallback.img esp/EFI/arch/
```

Then **enable** and **start** `efistub-update.path`.

Tip: For **Secure Boot** with your own keys, you can set up the service to also sign the image using **sbsigntools (https://www.archlinux.org/packages/?name=sbsigntools)**:

```
ExecStart=/usr/bin/sbsign --key /path/to/db.key --cert /path/to/db.crt --output esp/EFI/arch/vmlinuz-linux /boot/vmlinuz-linux
```

Using incron

`incron (https://www.archlinux.org/packages/?name=incron)` can be used to run a script syncing the EFISTUB Kernel after kernel updates.

`/usr/local/bin/efistub-update`

```
#!/bin/sh
cp -af /boot/vmlinuz-linux esp/EFI/arch/
cp -af /boot/initramfs-linux.img esp/EFI/arch/
cp -af /boot/initramfs-linux-fallback.img esp/EFI/arch/
```

Note: The first parameter `/boot/initramfs-linux-fallback.img` is the file to watch. The second parameter `IN_CLOSE_WRITE` is the action to watch for. The third parameter `/usr/local/bin/efistub-update` is the script to execute.

`/etc/incron.d/efistub-update.conf`

```
/boot/initramfs-linux-fallback.img IN_CLOSE_WRITE /usr/local/bin/efistub-update
```

In order to use this method, **enable** the `incrond.service`.

Using mkinitcpio hook

Mkinitcpio can generate a hook that does not need a system level daemon to function. It spawns a background process which waits for the generation of `vmlinuz`, `initramfs-linux.img`, and `initramfs-linux-fallback.img` before copying the files.

Add `efistub-update` to the list of hooks in `/etc/mkinitcpio.conf`.

/etc/initcpio/install/efistub-update

```
#!/usr/bin/env bash
build() {
        /usr/local/bin/efistub-copy &
}

help() {
        cat <<HELPEOF
This hook waits for mkinitcpio to finish and copies the finished ramdisk and kernel to the ESP
HELPEOF
}
```

/usr/local/bin/efistub-copy

```
#!/usr/bin/env bash

wait $PPID

cp -af /boot/vmlinuz-linux esp/EFI/arch/
cp -af /boot/initramfs-linux.img esp/EFI/arch/
cp -af /boot/initramfs-linux-fallback.img esp/EFI/arch/

echo "Synced kernel with ESP"
```

Using mkinitcpio hook (2)

Another **alternative** to the above solutions, that is potentially cleaner because there are less copies and does not need a system level daemon to function. The logic is reversed, the initramfs is directly stored in the EFI partition, not copied in `/boot/`. Then the kernel and any other additional files are copied to the ESP partition, thanks to a mkinitcpio hook.

Edit the file `/etc/mkinitcpio.d/linux.preset` :

/etc/mkinitcpio.d/linux.preset

```
# mkinitcpio preset file for the 'linux' package

# Directory to copy the kernel, the initramfs...
ESP_DIR="esp/EFI/arch"

ALL_config="/etc/mkinitcpio.conf"
ALL_kver="/boot/vmlinuz-linux"

PRESETS=('default' 'fallback')

#default_config="/etc/mkinitcpio.conf"
default_image="${ESP_DIR}/initramfs-linux.img"
default_options="-A esp-update-linux"

#fallback_config="/etc/mkinitcpio.conf"
fallback_image="${ESP_DIR}/initramfs-linux-fallback.img"
fallback_options="-S autodetect"
```

Then create the file `/etc/initcpio/install/esp-update-linux` which need to be executable :

/etc/initcpio/install/esp-update-linux

```
# Directory to copy the kernel, the initramfs...
ESP_DIR="esp/EFI/arch"

build() {
        cp -af /boot/vmlinuz-linux "${ESP_DIR}/"
        # If ucode is used uncomment this line
        #cp -af /boot/intel-ucode.img "${ESP_DIR}/"
}

help() {
        cat <<HELPEOF
This hook copies the kernel to the ESP partition
HELPEOF
}
```

To test that, just run:

```
# rm /boot/initramfs-linux-fallback.img
# rm /boot/initramfs-linux.img
# mkinitcpio -p linux
```

Using pacman hook

A last option relies on the **pacman hooks** that are run at the end of the transaction.

The first file is a hook that monitors the relevant files, and it is run if they were modified in the former transaction.

/etc/pacman.d/hooks/999-kernel-efi-copy.hook

```
[Trigger]
Type = File
Operation = Install
Operation = Upgrade
Target = boot/vmlinuz*
Target = usr/lib/initcpio/*
Target = boot/intel-ucode.img

[Action]
Description = Copying linux and initramfs to EFI directory...
When = PostTransaction
Exec = /usr/local/bin/kernel-efi-copy.sh
```

The second file is the script itself. Create the file and make it **executable**:

/usr/local/bin/kernel-efi-copy.sh

```
#!/usr/bin/env bash
#
# Copy kernel and initramfs images to EFI directory
#
```

```
ESP_DIR="esp/EFI/arch"

for file in /boot/vmlinuz*
do
        cp -af "$file" "$ESP_DIR/$(basename "$file").efi"
        [[ $? -ne 0 ]] && exit 1
done

for file in /boot/initramfs*
do
        cp -af "$file" "$ESP_DIR/"
        [[ $? -ne 0 ]] && exit 1
done

[[ -e /boot/intel-ucode.img ]] && cp -af /boot/intel-ucode.img "$ESP_DIR/"

exit 0
```

Known issues

ESP on RAID

It is possible to make the ESP part of a RAID1 array, but doing so brings the risk of data corruption, and further considerations need to be taken when creating the ESP. See **[3] (https://bbs.archlinux.org/viewtopic.php?pid=1398710#p1398710)** and **[4] (https://bbs.archlinux.org/viewtopic.php?pid=1390741#p1390741)** for details.

See also

- **The EFI System Partition and the Default Boot Behavior (http://blog.uncooperative.org/blog/2014/02/06/the-efi-system-partition/)**