Fallback mechanisms

A fallback mechanism is supported to allow to overcome failures to do a direct filesystem lookup on the root filesystem or when the firmware simply cannot be installed for practical reasons on the root filesystem. The kernel configuration options related to supporting the firmware fallback mechanism are:

- CONFIG_FW_LOADER_USER_HELPER: enables building the firmware fallback mechanism. Most distributions enable this option today. If enabled but CONFIG_FW_LOADER_USER_HELPER_FALLBACK is disabled, only the custom fallback mechanism is available and for the request_firmware_nowait() call.
- CONFIG_FW_LOADER_USER_HELPER_FALLBACK: force enables each request to enable the kobject uevent fallback mechanism on all firmware API calls except request_firmware_direct(). Most distributions disable this option today. The call request_firmware_nowait() allows for one alternative fallback mechanism: if this kconfig option is enabled and your second argument to request_firmware_nowait(), uevent, is set to false you are informing the kernel that you have a custom fallback mechanism and it will manually load the firmware. Read below for more details.

Note that this means when having this configuration:

CONFIG_FW_LOADER_USER_HELPER=y CONFIG_FW_LOADER_USER_HELPER_FALLBACK=n

the kobject uevent fallback mechanism will never take effect even for request_firmware_nowait() when uevent is set to true.

Justifying the firmware fallback mechanism

Direct filesystem lookups may fail for a variety of reasons. Known reasons for this are worth itemizing and documenting as it justifies the need for the fallback mechanism:

- Race against access with the root filesystem upon bootup.
- Races upon resume from suspend. This is resolved by the firmware cache, but the firmware cache is only supported if you use uevents, and its not supported for request_firmware_into_buf().
- Firmware is not accessible through typical means:
    - It cannot be installed into the root filesystem
    - The firmware provides very unique device specific data tailored for the unit gathered with local information. An example is calibration data for WiFi chipsets for mobile devices. This calibration data is not common to all units, but tailored per unit. Such information may be installed on a separate flash partition other than where the root filesystem is provided.

Types of fallback mechanisms

There are really two fallback mechanisms available using one shared sysfs interface as a loading facility:

- Kobject uevent fallback mechanism
- Custom fallback mechanism

First lets document the shared sysfs loading facility.

Firmware sysfs loading facility

In order to help device drivers upload firmware using a fallback mechanism the firmware infrastructure creates a sysfs interface to enable userspace to load and indicate when firmware is ready. The sysfs directory is created via fw_create_instance(). This call creates a new struct device named after the firmware requested, and establishes it in the device hierarchy by associating the device used to make the request as the device's parent. The sysfs directory's file attributes are defined and controlled through the new device's class (firmware_class) and group (fw_dev_attr_groups). This is actually where the original firmware_class module name came from, given that originally the only firmware loading mechanism available was the mechanism we now use as a fallback mechanism, which registers a struct class firmware_class. Because the attributes exposed are part of the module name, the module name firmware_class cannot be renamed in the future, to ensure backward compatibility with old userspace.

To load firmware using the sysfs interface we expose a loading indicator, and a file upload firmware into:

- /sys/$DEVPATH/loading
- /sys/$DEVPATH/data

To upload firmware you will echo 1 onto the loading file to indicate you are loading firmware. You then write the firmware into the data file, and you notify the kernel the firmware is ready by echo'ing 0 onto the loading file.

The firmware device used to help load firmware using sysfs is only created if direct firmware loading fails and if the fallback

mechanism is enabled for your firmware request, this is set up with :c:func:`firmware_fallback_sysfs`. It is important to re-iterate that no device is created if a direct filesystem lookup succeeded.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\firmware\(linux-master)(Documentation)(driver-api)(firmware)fallback-mechanisms.rst, line 94); *backlink***
>
> Unknown interpreted text role "c:func".

Using:

```
echo 1 > /sys/$DEVPATH/loading
```

Will clean any previous partial load at once and make the firmware API return an error. When loading firmware the firmware_class grows a buffer for the firmware in PAGE_SIZE increments to hold the image as it comes in.

firmware_data_read() and firmware_loading_show() are just provided for the test_firmware driver for testing, they are not called in normal use or expected to be used regularly by userspace.

firmware_fallback_sysfs

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\firmware\(linux-master)(Documentation)(driver-api)(firmware)fallback-mechanisms.rst, line 114)**
>
> Unknown directive type "kernel-doc".
>
> ```
> .. kernel-doc:: drivers/base/firmware_loader/fallback.c
>    :functions: firmware_fallback_sysfs
> ```

Firmware kobject uevent fallback mechanism

Since a device is created for the sysfs interface to help load firmware as a fallback mechanism userspace can be informed of the addition of the device by relying on kobject uevents. The addition of the device into the device hierarchy means the fallback mechanism for firmware loading has been initiated. For details of implementation refer to fw_load_sysfs_fallback(), in particular on the use of dev_set_uevent_suppress() and kobject_uevent().

The kernel's kobject uevent mechanism is implemented in lib/kobject_uevent.c, it issues uevents to userspace. As a supplement to kobject uevents Linux distributions could also enable CONFIG_UEVENT_HELPER_PATH, which makes use of core kernel's usermode helper (UMH) functionality to call out to a userspace helper for kobject uevents. In practice though no standard distribution has ever used the CONFIG_UEVENT_HELPER_PATH. If CONFIG_UEVENT_HELPER_PATH is enabled this binary would be called each time kobject_uevent_env() gets called in the kernel for each kobject uevent triggered.

Different implementations have been supported in userspace to take advantage of this fallback mechanism. When firmware loading was only possible using the sysfs mechanism the userspace component "hotplug" provided the functionality of monitoring for kobject events. Historically this was superseded be systemd's udev, however firmware loading support was removed from udev as of systemd commit be2ea723b1d0 ("udev: remove userspace firmware loading support") as of v217 on August, 2014. This means most Linux distributions today are not using or taking advantage of the firmware fallback mechanism provided by kobject uevents. This is specially exacerbated due to the fact that most distributions today disable CONFIG_FW_LOADER_USER_HELPER_FALLBACK.

Refer to do_firmware_uevent() for details of the kobject event variables setup. The variables currently passed to userspace with a "kobject add" event are:

- FIRMWARE=firmware name
- TIMEOUT=timeout value
- ASYNC=whether or not the API request was asynchronous

By default DEVPATH is set by the internal kernel kobject infrastructure. Below is an example simple kobject uevent script:

```
# Both $DEVPATH and $FIRMWARE are already provided in the environment.
MY_FW_DIR=/lib/firmware/
echo 1 > /sys/$DEVPATH/loading
cat $MY_FW_DIR/$FIRMWARE > /sys/$DEVPATH/data
echo 0 > /sys/$DEVPATH/loading
```

Firmware custom fallback mechanism

Users of the request_firmware_nowait() call have yet another option available at their disposal: rely on the sysfs fallback mechanism but request that no kobject uevents be issued to userspace. The original logic behind this was that utilities other than udev might be required to lookup firmware in non-traditional paths -- paths outside of the listing documented in the section 'Direct filesystem

lookup'. This option is not available to any of the other API calls as uevents are always forced for them.

Since uevents are only meaningful if the fallback mechanism is enabled in your kernel it would seem odd to enable uevents with kernels that do not have the fallback mechanism enabled in their kernels. Unfortunately we also rely on the uevent flag which can be disabled by request_firmware_nowait() to also setup the firmware cache for firmware requests. As documented above, the firmware cache is only set up if uevent is enabled for an API call. Although this can disable the firmware cache for request_firmware_nowait() calls, users of this API should not use it for the purposes of disabling the cache as that was not the original purpose of the flag. Not setting the uevent flag means you want to opt-in for the firmware fallback mechanism but you want to suppress kobject uevents, as you have a custom solution which will monitor for your device addition into the device hierarchy somehow and load firmware for you through a custom path.

Firmware fallback timeout

The firmware fallback mechanism has a timeout. If firmware is not loaded onto the sysfs interface by the timeout value an error is sent to the driver. By default the timeout is set to 60 seconds if uevents are desirable, otherwise MAX_JIFFY_OFFSET is used (max timeout possible). The logic behind using MAX_JIFFY_OFFSET for non-uevents is that a custom solution will have as much time as it needs to load firmware.

You can customize the firmware timeout by echo'ing your desired timeout into the following file:

- /sys/class/firmware/timeout

If you echo 0 into it means MAX_JIFFY_OFFSET will be used. The data type for the timeout is an int.

EFI embedded firmware fallback mechanism

On some devices the system's EFI code / ROM may contain an embedded copy of firmware for some of the system's integrated peripheral devices and the peripheral's Linux device-driver needs to access this firmware.

Device drivers which need such firmware can use the firmware_request_platform() function for this, note that this is a separate fallback mechanism from the other fallback mechanisms and this does not use the sysfs interface.

A device driver which needs this can describe the firmware it needs using an efi_embedded_fw_desc struct:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\firmware\(linux-master)(Documentation)(driver-api)(firmware)fallback-mechanisms.rst, line 222)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/efi_embedded_fw.h
   :functions: efi_embedded_fw_desc
```

The EFI embedded-fw code works by scanning all EFI_BOOT_SERVICES_CODE memory segments for an eight byte sequence matching prefix; if the prefix is found it then does a sha256 over length bytes and if that matches makes a copy of length bytes and adds that to its list with found firmwares.

To avoid doing this somewhat expensive scan on all systems, dmi matching is used. Drivers are expected to export a dmi_system_id array, with each entries' driver_data pointing to an efi_embedded_fw_desc.

To register this array with the efi-embedded-fw code, a driver needs to:

1. Always be builtin to the kernel or store the dmi_system_id array in a separate object file which always gets builtin.
2. Add an extern declaration for the dmi_system_id array to include/linux/efi_embedded_fw.h.
3. Add the dmi_system_id array to the embedded_fw_table in drivers/firmware/efi/embedded-firmware.c wrapped in a #ifdef testing that the driver is being builtin.
4. Add "select EFI_EMBEDDED_FIRMWARE if EFI_STUB" to its Kconfig entry.

The firmware_request_platform() function will always first try to load firmware with the specified name directly from the disk, so the EFI embedded-fw can always be overridden by placing a file under /lib/firmware.

Note that:

1. The code scanning for EFI embedded-firmware runs near the end of start_kernel(), just before calling rest_init(). For normal drivers and subsystems using subsys_initcall() to register themselves this does not matter. This means that code running earlier cannot use EFI embedded-firmware.
2. At the moment the EFI embedded-fw code assumes that firmwares always start at an offset which is a multiple of 8 bytes, if this is not true for your case send in a patch to fix this.
3. At the moment the EFI embedded-fw code only works on x86 because other archs free EFI_BOOT_SERVICES_CODE before the EFI embedded-fw code gets a chance to scan it.
4. The current brute-force scanning of EFI_BOOT_SERVICES_CODE is an ad-hoc brute-force solution. There has been discussion to use the UEFI Platform Initialization (PI) spec's Firmware Volume protocol. This has been rejected because

the FV Protocol relies on *internal* interfaces of the PI spec, and: 1. The PI spec does not define peripheral firmware at all 2. The internal interfaces of the PI spec do not guarantee any backward compatibility. Any implementation details in FV may be subject to change, and may vary system to system. Supporting the FV Protocol would be difficult as it is purposely ambiguous.

Example how to check for and extract embedded firmware

To check for, for example Silead touchscreen controller embedded firmware, do the following:

1. Boot the system with efi=debug on the kernel commandline
2. cp /sys/kernel/debug/efi/boot_services_code? to your home dir
3. Open the boot_services_code? files in a hex-editor, search for the magic prefix for Silead firmware: F0 00 00 00 02 00 00 00, this gives you the beginning address of the firmware inside the boot_services_code? file.
4. The firmware has a specific pattern, it starts with a 8 byte page-address, typically F0 00 00 00 02 00 00 00 for the first page followed by 32-bit word-address + 32-bit value pairs. With the word-address incrementing 4 bytes (1 word) for each pair until a page is complete. A complete page is followed by a new page-address, followed by more word + value pairs. This leads to a very distinct pattern. Scroll down until this pattern stops, this gives you the end of the firmware inside the boot_services_code? file.
5. "dd if=boot_services_code? of=firmware bs=1 skip=<begin-addr> count=<len>" will extract the firmware for you. Inspect the firmware file in a hexeditor to make sure you got the dd parameters correct.
6. Copy it to /lib/firmware under the expected name to test it.
7. If the extracted firmware works, you can use the found info to fill an efi_embedded_fw_desc struct to describe it, run "sha256sum firmware" to get the sha256sum to put in the sha256 field.