How to access information in sysfs

Contents

Stable: 24.01.2020 - 09:32 / Revision: 24.01.2020 - 09:31

Stable: 24.01.2020 - 08:32 / Revision: 24.01.2020 - 08:31

A quality version of this page, approved on *24 January 2020*, was based off this revision.

Contents

1 Article purpose

This article provides some information about the sysfs pseudo filesystem usage from the user space.

2 Sysfs (/sys) pseudo filesystem

Sysfs provides a mean to export kernel data structures, their attributes, and the linkages between them to the user space.

Please refer to sysfs part of pseudo filesystem page.

3 Sysfs usage

Linux kernel provides a documentation[1] about the rules for sysfs usage.

Some examples are also described below with two different approaches for using sysfs entries from the user space:

- Linux application in C language
- bash script.

3.1 Example from Linux application

The below example is a typical sequence for using sysfs entry (here a PWM component):

- open a file descriptor of the sysfs entry file

```
10  len=snprintf(buf, sizeof(buf), "/sys/class/pwm/pwmchip0/pwm%d/duty_cycle", pwm_channel
);
11  fd = open(buf, O_RDWR);
```

- if fd is correctly opened, write/read value in the file: pay attention to the "text" format

```
12  if (fd < 0)
13  {
14      perror("pwm/duty_cycle");
15      return fd;
16  }
```

- read: store data to buffer

```
18  read(fd, buf, sizeof(buf));
```

- write: write data from buffer

```
20  len = snprintf(buf, sizeof(buf), "%d", 900000);
21  write(fd, buf, len);
```

- close file descriptor

```
30  close(fd);
```

3.2 Example for shell command / bash script

Operations on sysfs entries can be done by using command lines (i.e. *echo* for writing, *cat* for reading).

In this way, it is possible to use a bash script to execute a configuration sequence, similarly to what a user would do by typing multiple shell commands.

An example is provided in How_to_use_PWM_with_sysfs_interface.

4 References

- Documentation/admin-guide/sysfs-rules.rst

System File System (See https://en.wikipedia.org/wiki/Sysfs for more details)

Linux® is a registered trademark of Linus Torvalds.

Pulse Width Modulation

Stable: 18.02.2021 - 10:45 / Revision: 18.02.2021 - 10:40

A quality version of this page, approved on *18 February 2021*, was based off this revision.

This article gives information about the Linux®PWM framework.
It explains how to activate the PWM interface and, based on examples, how to use it.

Contents

1 Framework purpose

PWM (Pulse Width Modulation) framework offers a unified interface for the users to:

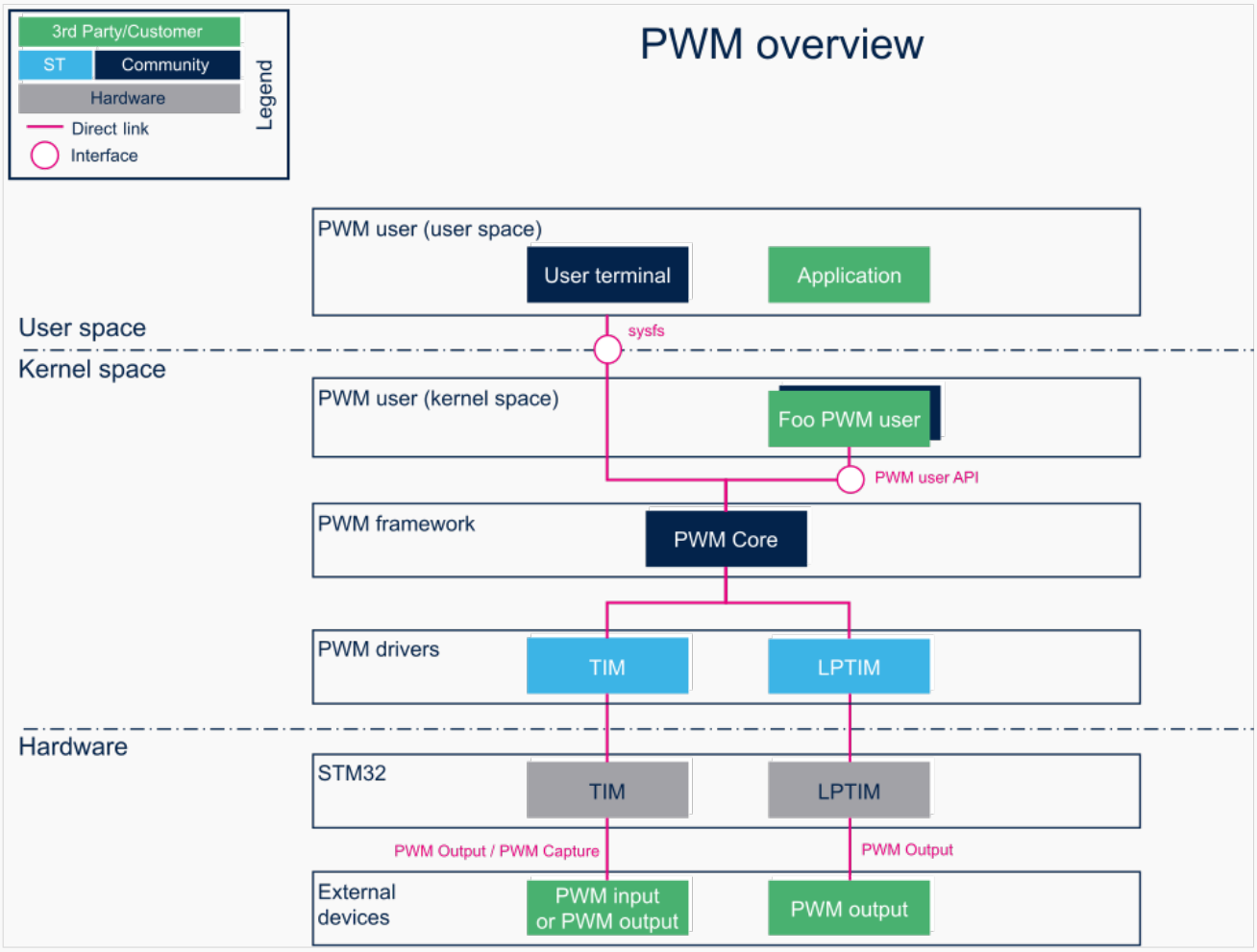
- control PWM output(s) such as period, duty cycle and polarity.
- capture a PWM signal and report its period and duty cycle (e.g. input).

The interface can be used from:

- user space (sysfs)
- kernel space (API)

PWMs can be used in various use cases, as mentioned in How to use the framework to control LEDs, beepers, vibrators or fans...

2 System overview



PWM Implementation architecture

2.1 Component description

- **PWM user** (User space)

The user can use PWM sysfs interface, from a user terminal or a custom application, to control PWM device(s) from user space.

- **PWM user** (Kernel space)

User drivers can use PWM API to control PWM external device(s) from kernel space (such as back-light, vibrator, LED or fan drivers).

- **PWM framework** (Kernel space)

The PWM core provides sysfs interface and PWM API. They can be used to implement PWM user and PWM controller drivers.

- **PWM drivers** (Kernel space)

Provider drivers such as STM32 TIM Linux driver and STM32 LPTIM Linux driver that expose PWM controller(s) to the core.

- **PWM hardware**

PWM controller(s) such as *TIM internal peripheral*[1] and *LPTIM internal peripheral*[2] used to drive external PWM controlled devices.

2.2 API description

Documentation on PWM interface can be found under kernel *Documentation/driver-api/pwm.rst*[3]

2.2.1 Kernel PWM API

The main useful user API are the following:

- **devm_pwm_get()** or pwm_get() / pwm_put(): this API is used to look up, request, then free a PWM device.
- **pwm_init_state()**, **pwm_get_state()**, **pwm_apply_state()**: this API is used to initialize, retrieve and apply the current PWM device state.
- **pwm_config()**: this API updates the PWM device configuration (period and duty cycle).
- ...

2.2.2 Sysfs interface

In addition to *Documentation/driver-api/pwm.rst*[3], details on ABI are available in *Documentation/ABI/testing/sysfs-class-pwm*[4]
.

3 Configuration

3.1 Kernel configuration

Activate PWM framework in the kernel configuration through the Linux menuconfig tool, Menuconfig or how to configure kernel (CONFIG_PWM=y):

```
Device Drivers  --->
   [*] Pulse-Width Modulation (PWM) Support  --->
```

Activate PWM drivers for STM32 PWM drivers: STM32 TIM Linux driver and/or STM32 LPTIM Linux driver

3.2 Device tree configuration

- PWM generic DT bindings:

PWM DT bindings documentation[5] describes device tree properties related to standard **PWM user nodes** and **PWM controller nodes**.

- Detailed DT configuration for STM32 internal peripherals:

TIM device tree configuration and/or LPTIM device tree configuration

4 How to use the framework

PWM can be used either from the user or the kernel space.

4.1 How to use PWM with sysfs interface

The available PWM controllers are listed in sysfs:

```
Board $> ls /sys/class/pwm
pwmchip0
```

The number of channels per controller can be read in npwm (read-only)

```
Board $> cd /sys/class/pwm/pwmchip0
Board $> cat npwm
4
```

Each channel is exported (requested for sysfs activation) by writing the corresponding number in 'export'.

Information

TIMx_CH1 is exported as **"pwm0"**, TIMx_CH2 as "pwm1", and so on:

- PWM channels are numbered from **0 to 'npwm' - 1**
- TIM[1] channels are numbered from **1 to 'npwm'**.

As an example, proceed as follows to export the first channel (TIMx_CH1, e.g. channel 0):

```
Board $> echo 0 > export
Board $> ls
device  export  npwm  power  pwm0  subsystem  uevent  unexport
```

The period and duty cycle must be configured before enabling any channel.

As an example, proceed as follows to set a period of 100 ms with a duty cycle of 60% on channel 0:

```
Board $> echo 100000000 > pwm0/period
Board $> echo 60000000 > pwm0/duty_cycle
Board $> echo 1 > pwm0/enable
```

The polarity can be inverted or set to normal by using the polarity entry:

```
Board $> echo "inversed" > pwm0/polarity
Board $> cat pwm0/polarity
inversed

Board $> echo "normal" > pwm0/polarity
Board $> cat pwm0/polarity
normal
```

4.2 How to use PWM capture with sysfs interface

PWM capture is available on some PWM controllers such as *TIM internal peripheral*[1] (see TIM configured in PWM input capture mode).

Information

PWM output and capture mode are mutually exclusive on a TIM instance

```
# First export a channel (e.g. 0), then capture PWM input on it:
Board $> cd /sys/class/pwm/pwmchip0
Board $> echo 0 > export

Board $> cd pwm0
Board $> ls
capture  duty_cycle  enable  period  polarity  power  uevent

Board $> cat capture
10000 1002              # capture result is in nano-seconds, e.g.: 100KHz, 10% duty cycle
```

4.3 Example of PWM usage with kernel PWM API

Several in-kernel drivers use kernel PWM API. Below a few examples:

- pwm-beeper: *drivers/input/misc/pwm-beeper.c*[6] driver, Documentation/devicetree/bindings/input/pwm-beeper.txt DT binding documentation.
- pwm-vibrator: *drivers/input/misc/pwm-vibra.c*[7] driver, Documentation/devicetree/bindings/input/pwm-vibrator.txt DT binding documentation.

5 How to trace and debug the framework

5.1 How to monitor with debugfs

PWM usage can be monitored from debugfs 'pwm' entry. For example:

```
Board $> cd /sys/kernel/debug/
Board $> cat pwm
platform/44000000.timer:pwm, 4 PWM
devices                                                    <-- One timer
instance exposes 4 PWM channels.
 pwm-0   (sysfs              ): requested enabled period: 1000000 ns duty: 500000 ns
polarity: normal   <-- Channel 0 has been exported, enabled and configured via sysfs
 pwm-1   ((null)             ): period: 0 ns duty: 0 ns polarity: normal
 pwm-2   ((null)             ): period: 0 ns duty: 0 ns polarity:
normal                                <-- Other channels aren't used currently
 pwm-3   ((null)             ): period: 0 ns duty: 0 ns polarity: normal
```

5.2 Troubleshooting PWM capture

Here are some clues on how to debug possible errors in PWM capture mode.

See How to use PWM capture with sysfs interface as a pre-requisite.

```
Board $> cat capture
cat: capture: Connection timed out
```

This may be due to:

- the input signal isn't recognized as a PWM input (or there's no input signal to capture).
- a wrong alternate function number is used for the input pin configuration in the device-tree.

See "TIM configured in PWM input capture mode" for further details.

```
Board $> cat capture
cat: capture: Device or resource busy
```

This may be due to:

- a PWM channel on the same TIM instance is already running (in capture or output mode)

```
Board $> cat capture
cat: capture: No such device
```

This may be due to:

- the DMA isn't configured properly in the device-tree.

See "TIM configured in PWM input capture mode" for further details.

```
Board $> cat capture
cat: capture: Function not implemented
```

This may be due to:

- a wrong TIM instance is being used (e.g. "/sys/class/pwm/pwmchip/pwmchipN"), and it doesn't support capture (like LPTIM)
- the DMA support isn't enabled (CONFIG_DMA_ENGINE)

6 References

- 1.01.11.2 TIM internal peripheral
- LPTIM internal peripheral
- 3.03.1 Documentation/driver-api/pwm.rst , Pulse Width Modulation interface
- Documentation/ABI/testing/sysfs-class-pwm , Pulse Width Modulation ABI
- Documentation/devicetree/bindings/pwm/pwm.txt , PWM DT bindings documentation
- drivers/input/misc/pwm-beeper.c , Example to use kernel PWM API
- drivers/input/misc/pwm-vibra.c , Example to use kernel PWM API

Pulse Width Modulation

System File System (See https://en.wikipedia.org/wiki/Sysfs for more details)

Application programming interface

Light-emitting diode

low-power timer (STM32 specific)

Application binary interface. (In computer software, an application binary interface (ABI) describes the low-level interface between a computer program and the operating system or another program.)

Device Tree

Debug File System (See https://en.wikipedia.org/wiki/Debugfs for more details)

Direct Memory Access

Stable: 31.01.2020 - 13:237 Revision: 31.01.2020 - 13:16

A quality version of this page, approved on *31 January 2020*, was based off this revision.

Contents

1 Introduction

When running, Linux® operating system creates and populates some filesystems which are not present on the rootfs fileystem of the Linux distribution image:

- **pseudo filesystems**: **sysfs** (/sys), **procfs** (/proc), **debugfs** (/sys/kernel/debug), **configfs** (/sys/kernel/config), **tracefs** (/sys /kernel/tracing)
- **temporary filesystems**: **tmppfs** (/dev/shm, /run, /sys/fs/cgroup, /tmp/, /var/volatile, /run/user/*<id>*), **devtmpfs** (/dev)

Pseudo filesystems contain many informations, configurations and logs about the current running kernel.
Informations from those pseudo filesystems are very helpful and any debugging session should start by browsing them.

These both filesystem groups are part of the File Hierarchy Standard (FHS) for the Linux operating system.

As they are placed in volatile memory, they are only available at run time, and they disappear at shutdown.

2 procfs (/proc) - Kernel and process information

Procfs[1] is enabled and ready to be used in all STM32MPU Embedded Software distribution, via the Linux® kernel configuration **CONFIG_PROC_FS**, set to yes by default.

```
Symbol: PROC_FS
Location:
  File systems --->
    Pseudo filesystems -->
      [*] /proc file system support
```

Please refer to Menuconfig or how to configure kernel article to get instructions for modifying the configuration and recompiling the Linux kernel image in the Distribution Package context.

Procfs is sometimes referred to as a process information pseudo-file system. It does not contain 'real' files but runtime system information (e.g. system memory, devices mounted, hardware configuration, etc).

For this reason it can be seen as a control and information center for the kernel. In fact, quite a lot of system utilities are simply calls to files in this directory.

For example, 'lsmod' is the same as 'cat /proc/modules' while 'lspci' is a synonym for 'cat /proc/pci'. By altering files located in this directory you can even read/change kernel parameters (sysctl) while the system is running.

Procfs is explain in details in The Linux Documentation Project[2], or Wikipedia[3].

2.1 Some procfs useful entries

Name	Description
/proc/<*PID*>/	directory containing information about the kernel process <*PID*>
/prod /cmdline	gives the boot options passed to the kernel
/proc /cpuinfo	provides information about the processor: its type, make, model, and performance
/proc /devices	lists all of the devices (divided into the "block" and "character" categories) available on the currently running system, giving also the major part of the /dev name too
/proc /device-tree	contains devices tree information for all nodes organized as defined in the device-tree source file(s).
/proc /interrupts	shows which interrupts are in use, and how many of each there have been.
/proc	

Name	Description
/iomem	gives information about memory mapping
/proc /kallsyms	contains the dynamic kernel symbol table
/proc /kmsg	holds messages output by the kernel.
/proc /meminfo	contains a summary of how the kernel is managing its memory (both physical and swap).
/proc /misc	Miscellaneous pieces of information, lists also the misc features devices
/proc /modules	one of the most important files in /proc; contains a list of the kernel modules currently loaded. It gives some indication of dependencies.

3 sysfs (/sys) - System filesystem

Sysfs[4] is a RAM-based filesystem initially based on ramfs. It provides a means to export kernel data structures, their attributes, and the linkages between them, to user space.

Sysfs is enabled and ready to be used in all STM32MPU Embedded Software distribution, via the Linux® kernel configuration **CONFIG_SYSFS**, set to yes by default.

```
Symbol: SYSFS
Location:
  File systems --->
    Pseudo filesystems -->
      [*] sysfs file system support
```

Please refer to Menuconfig or how to configure kernel article to get instructions for modifying the configuration and recompiling the Linux kernel image in the Distribution Package context.

Useful information also given in How to access information in sysfs article.

3.1 Top Level sysfs directory layout

Sysfs directory arrangement exposes the relationship of kernel data structures.

/sys has a sub-hierarchy file system:

sub-Directory	Description
/sys/block/	Contains one symbolic link for each block device that has been discovered on the system. The symbolic links point to corresponding directories under /sys/devices.
/sys/bus/	Contains one subdirectory for each of the bus types in the kernel. Each bus's directory contains two subdirectories: ./devices/ ./drivers/
/sys	Contains a single layer of further subdirectories for each of the device classes that have been

sub-Directory	Description
/class/	registered on the system (e.g., terminals, network devices, block devices, graphics devices, sound devices, and so on). Inside each of these subdirectories are symbolic links for each of the devices in this class. These symbolic links refer to entries in the /sys/devices directory
/sys/class/net/	Each of the entries in this directory is a symbolic link representing one of the real or virtual networking devices that are visible in the network namespace of the process that is accessing the directory. Each of these symbolic links refers to entries in the /sys/devices directory
/sys/dev/	This directory contains two subdirectories block/ and char/, corresponding, respectively, to the block and character devices on the system. Inside each of these subdirectories are symbolic links with names of the form major-ID:minor-ID, where the ID values correspond to the major and minor ID of a specific device. Each symbolic link points to the sysfs directory for a device. The symbolic links inside /sys/dev thus provide an easy way to look up the sysfs interface using the device IDs returned by a call to stat tool (or similar)
/sys/devices/	Contains a filesystem representation of the kernel device tree, which is a hierarchy of device structures within the kernel
/sys/firmware/	Contains interfaces for viewing and manipulating firmware-specific objects and attributes
/sys/fs	

sub-Directory	Description
/	Contains subdirectories for some filesystems. A filesystem will have a subdirectory here only if it chose to explicitly create the subdirectory
/sys/kernel/	Contains various files and subdirectories that provide information about the running kernel
/sys/module/	Contains one subdirectory for each module that is loaded into the kernel. The name of each directory is the name of the module

3.2 Zoom to debugfs (/sys/kernel/debug)

Please refer to debugfs article.

3.3 Zoom to configfs (/sys/kernel/config)

Please refer to configfs article.

3.4 Zoom to tracefs (/sys/kernel/tracing)

Tracefs is used with the Linux kernel tracing framework.

Example of usage is given in Ftrace article.

- Command to mount tracefs:

```
Board $> mount -t tracefs nodev /sys/kernel/tracing
```

To find out which tracers are available, simply read available_tracers file in the tracing directory:

```
Board $> cat /sys/kernel/tracing/available_tracers
 function_graph function nop
```

More tracers can be added by kernel build configurations. Please refer to Ftrace#More_tracers paragraph.

4 Information about temporary filesystem

4.1 tmpfs

Tmpfs[5] is a file system which keeps all files in virtual memory.

It is enabled and ready to be used in all STM32MPU Embedded Software distribution, via the Linux® kernel configuration **CONFIG_TMPFS**, set to yes by default.

```
Symbol: TMPFS
Location:
  File systems --->
    Pseudo filesystems -->
      [*] Tmpfs virtual memory file system support (former shm fs)
```

Please refer to Menuconfig or how to configure kernel article to get instructions for modifying the configuration and recompiling the Linux kernel image in the Distribution Package context.

Everything in tmpfs is temporary in the sense that no files will be created on your hard drive. If you unmount a tmpfs instance, everything stored therein is lost.

On the board target, you can check for the directory path mount with the tmpfs:

```
Board $> mount | grep tmpfs
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
tmpfs on /run type tmpfs (rw,nosuid,nodev,mode=755)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,mode=755)
tmpfs on /tmp type tmpfs (rw,nosuid,nodev)
tmpfs on /var/volatile type tmpfs (rw,relatime)
tmpfs on /run/user/0 type tmpfs (rw,nosuid,nodev,relatime,size=43812k,mode=700)
```

For all details you can refer to the Linux documentation about tmpfs[5].

4.2 devtmpfs

Devtmpfs is enabled and ready to be used in all STM32MPU Embedded Software distribution, via the Linux® kernel configuration **CONFIG_DEVTMPFS** and **CONFIG_DEVTMPFS_MOUNT**, set to yes by default.

```
Symbol: DEVTMPFS
Location:
  Device Drivers --->
    Generic Driver Options -->
      [*] Maintain a devtmpfs filesystem to mount at /dev

Symbol: DEVTMPFS_MOUNT
Location:
  Device Drivers --->
    Generic Driver Options -->
      [*] Maintain a devtmpfs filesystem to mount at /dev
        [*] Automount devtmpfs at /dev, after the kernel mounted the rootfs
```

- **/dev** - special or device files

Devtmpfs is mounted on /dev which is the location of special or device files. Many of these are generated at boot time or even on the fly.

It is a very interesting directory that highlights one important aspect of the Linux filesystem: **everything is a file or a directory**.

Look through this directory and you can see device file system entries which represent the various partitions on the first master drive of the system:
For example:

- mmcblk0p<*id*> (microSD Card),
- mmcblk1p<*id*> (eMMC),
- sda<*id*>,
- sdb<*id*> (NAND or USB Key),
- ttySTM<*id*> (tty Serial link),
- etc...

These entries can be both read from and written to.
Take /dev/ttyUSB0, for instance. This file represents the USB Serial port. Sending data to and reading from /dev/ttyUSB0 will allow you to communicate with host PC through the minicom application (or equivalent).

/dev is very helpful, more info could be found in the Linux Documentation Project[6].

5 References

- Documentation/filesystems/proc.txt
- http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html
- https://en.wikipedia.org/wiki/Procfs
- Documentation/filesystems/sysfs.txt
- 5.05.1 Documentation/filesystems/tmpfs.txt
- http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/dev.html

System File System (See https://en.wikipedia.org/wiki/Sysfs for more details)

Process File System (See https://en.wikipedia.org/wiki/Procfs for more details)

Debug File System (See https://en.wikipedia.org/wiki/Debugfs for more details)

Configuration File System (See https://en.wikipedia.org/wiki/Configfs for more details)

Random Access Memory (Early computer memories generally hadserial access. Memories where any given address can be accessed when desired were then called "random access" to distinguish them from the memories where contents can only be accessed in a fixed order. The term is used today for volatile random-acces ssemiconductor memories.)

Read Only

former spelling for e•MMC ('e' in italic)