GPIO Sysfs Interface for Userspace

> **Warning**
>
> THIS ABI IS DEPRECATED, THE ABI DOCUMENTATION HAS BEEN MOVED TO Documentation/ABI/obsolete/sysfs-gpio AND NEW USERSPACE CONSUMERS ARE SUPPOSED TO USE THE CHARACTER DEVICE ABI. THIS OLD SYSFS ABI WILL NOT BE DEVELOPED (NO NEW FEATURES), IT WILL JUST BE MAINTAINED.

Refer to the examples in tools/gpio/* for an introduction to the new character device ABI. Also see the userspace header in include/uapi/linux/gpio.h

The deprecated sysfs ABI

Platforms which use the "gpiolib" implementors framework may choose to configure a sysfs user interface to GPIOs. This is different from the debugfs interface, since it provides control over GPIO direction and value instead of just showing a gpio state summary. Plus, it could be present on production systems without debugging support.

Given appropriate hardware documentation for the system, userspace could know for example that GPIO #23 controls the write protect line used to protect boot loader segments in flash memory. System upgrade procedures may need to temporarily remove that protection, first importing a GPIO, then changing its output state, then updating the code before re-enabling the write protection. In normal use, GPIO #23 would never be touched, and the kernel would have no need to know about it.

Again depending on appropriate hardware documentation, on some systems userspace GPIO can be used to determine system configuration data that standard kernels won't know about. And for some tasks, simple userspace GPIO drivers could be all that the system really needs.

DO NOT ABUSE SYSFS TO CONTROL HARDWARE THAT HAS PROPER KERNEL DRIVERS. PLEASE READ THE DOCUMENT AT Documentation/driver-api/gpio/drivers-on-gpio.rst TO AVOID REINVENTING KERNEL WHEELS IN USERSPACE. I MEAN IT. REALLY.

Paths in Sysfs

There are three kinds of entries in /sys/class/gpio:

- Control interfaces used to get userspace control over GPIOs;
- GPIOs themselves; and
- GPIO controllers ("gpio_chip" instances).

That's in addition to standard files including the "device" symlink.

The control interfaces are write-only:

/sys/class/gpio/

"export" ...

Userspace may ask the kernel to export control of a GPIO to userspace by writing its number to this file.

Example: "echo 19 > export" will create a "gpio19" node for GPIO #19, if that's not requested by kernel code.

"unexport" ...

Reverses the effect of exporting to userspace.

Example: "echo 19 > unexport" will remove a "gpio19" node exported using the "export" file.

GPIO signals have paths like /sys/class/gpio/gpio42/ (for GPIO #42) and have the following read/write attributes:

/sys/class/gpio/gpioN/

"direction" ...

reads as either "in" or "out". This value may normally be written. Writing as "out" defaults to initializing the value as low. To ensure glitch free operation, values "low" and "high" may be written to configure the GPIO as an output with that initial value.

Note that this attribute *will not exist* if the kernel doesn't support changing the direction of a GPIO, or it was exported by kernel code that didn't explicitly allow userspace to reconfigure this GPIO's

direction.

"value" ...

reads as either 0 (low) or 1 (high). If the GPIO is configured as an output, this value may be written; any nonzero value is treated as high.

If the pin can be configured as interrupt-generating interrupt and if it has been configured to generate interrupts (see the description of "edge"), you can poll(2) on that file and poll(2) will return whenever the interrupt was triggered. If you use poll(2), set the events POLLPRI and POLLERR. If you use select(2), set the file descriptor in exceptfds. After poll(2) returns, either lseek(2) to the beginning of the sysfs file and read the new value or close the file and re-open it to read the value.

"edge" ...

reads as either "none", "rising", "falling", or "both". Write these strings to select the signal edge(s) that will make poll(2) on the "value" file return.

This file exists only if the pin can be configured as an interrupt generating input pin.

"active_low" ...

reads as either 0 (false) or 1 (true). Write any nonzero value to invert the value attribute both for reading and writing. Existing and subsequent poll(2) support configuration via the edge attribute for "rising" and "falling" edges will follow this setting.

GPIO controllers have paths like /sys/class/gpio/gpiochip42/ (for the controller implementing GPIOs starting at #42) and have the following read-only attributes:

/sys/class/gpio/gpiochipN/

"base" ...
same as N, the first GPIO managed by this chip
"label" ...
provided for diagnostics (not always unique)
"ngpio" ...
how many GPIOs this manages (N to N + ngpio - 1)

Board documentation should in most cases cover what GPIOs are used for what purposes. However, those numbers are not always stable; GPIOs on a daughtercard might be different depending on the base board being used, or other cards in the stack. In such cases, you may need to use the gpiochip nodes (possibly in conjunction with schematics) to determine the correct GPIO number to use for a given signal.

Exporting from Kernel code

Kernel code can explicitly manage exports of GPIOs which have already been requested using gpio_request():

```
/* export the GPIO to userspace */
int gpiod_export(struct gpio_desc *desc, bool direction_may_change);

/* reverse gpio_export() */
void gpiod_unexport(struct gpio_desc *desc);

/* create a sysfs link to an exported GPIO node */
int gpiod_export_link(struct device *dev, const char *name,
              struct gpio_desc *desc);
```

After a kernel driver requests a GPIO, it may only be made available in the sysfs interface by gpiod_export(). The driver can control whether the signal direction may change. This helps drivers prevent userspace code from accidentally clobbering important system state.

This explicit exporting can help with debugging (by making some kinds of experiments easier), or can provide an always-there interface that's suitable for documenting as part of a board support package.

After the GPIO has been exported, gpiod_export_link() allows creating symlinks from elsewhere in sysfs to the GPIO sysfs node. Drivers can use this to provide the interface under their own device in sysfs with a descriptive name.