Introduction

GPIO Interfaces

The documents in this directory give detailed instructions on how to access GPIOs in drivers, and how to write a driver for a device that provides GPIOs itself.

Due to the history of GPIO interfaces in the kernel, there are two different ways to obtain and use GPIOs:

- The descriptor-based interface is the preferred way to manipulate GPIOs, and is described by all the files in this directory excepted gpio-legacy.txt.
- The legacy integer-based interface which is considered deprecated (but still usable for compatibility reasons) is documented in gpio-legacy.txt.

The remainder of this document applies to the new descriptor-based interface. gpio-legacy.txt contains the same information applied to the legacy integer-based interface.

What is a GPIO?

A "General Purpose Input/Output" (GPIO) is a flexible software-controlled digital signal. They are provided from many kinds of chips, and are familiar to Linux developers working with embedded and custom hardware. Each GPIO represents a bit connected to a particular pin, or "ball" on Ball Grid Array (BGA) packages. Board schematics show which external hardware connects to which GPIOs. Drivers can be written generically, so that board setup code passes such pin configuration data to drivers.

System-on-Chip (SOC) processors heavily rely on GPIOs. In some cases, every non-dedicated pin can be configured as a GPIO; and most chips have at least several dozen of them. Programmable logic devices (like FPGAs) can easily provide GPIOs; multifunction chips like power managers, and audio codecs often have a few such pins to help with pin scarcity on SOCs; and there are also "GPIO Expander" chips that connect using the I2C or SPI serial buses. Most PC southbridges have a few dozen GPIO-capable pins (with only the BIOS firmware knowing how they're used).

The exact capabilities of GPIOs vary between systems. Common options:

- Output values are writable (high=1, low=0). Some chips also have options about how that value is driven, so that for example only one value might be driven, supporting "wire-OR" and similar schemes for the other value (notably, "open drain" signaling).
- Input values are likewise readable (1, 0). Some chips support readback of pins configured as "output", which is very useful in such "wire-OR" cases (to support bidirectional signaling). GPIO controllers may have input de-glitch/debounce logic, sometimes with software controls.
- Inputs can often be used as IRQ signals, often edge triggered but sometimes level triggered. Such IRQs may be configurable as system wakeup events, to wake the system from a low power state.
- Usually a GPIO will be configurable as either input or output, as needed by different product boards; single direction ones exist too.
- Most GPIOs can be accessed while holding spinlocks, but those accessed through a serial bus normally can't. Some systems support both types.

On a given board each GPIO is used for one specific purpose like monitoring MMC/SD card insertion/removal, detecting card write-protect status, driving a LED, configuring a transceiver, bit-banging a serial bus, poking a hardware watchdog, sensing a switch, and so on.

Common GPIO Properties

These properties are met through all the other documents of the GPIO interface and it is useful to understand them, especially if you need to define GPIO mappings.

Active-High and Active-Low

It is natural to assume that a GPIO is "active" when its output signal is 1 ("high"), and inactive when it is 0 ("low"). However in practice the signal of a GPIO may be inverted before is reaches its destination, or a device could decide to have different conventions about what "active" means. Such decisions should be transparent to device drivers, therefore it is possible to define a GPIO as being either active-high ("1" means "active", the default) or active-low ("0" means "active") so that drivers only need to worry about the logical signal and not about what happens at the line level.

Open Drain and Open Source

Sometimes shared signals need to use "open drain" (where only the low signal level is actually driven), or "open source" (where only the high signal level is driven) signaling. That term applies to CMOS transistors; "open collector" is used for TTL. A pullup or

pulldown resistor causes the high or low signal level. This is sometimes called a "wire-AND"; or more practically, from the negative logic (low=true) perspective this is a "wire-OR".

One common example of an open drain signal is a shared active-low IRQ line. Also, bidirectional data bus signals sometimes use open drain signals.

Some GPIO controllers directly support open drain and open source outputs; many don't. When you need open drain signaling but your hardware doesn't directly support it, there's a common idiom you can use to emulate it with any GPIO pin that can be used as either an input or an output:

> **LOW**: `gpiod_direction_output(gpio, 0)` ... this drives the signal and overrides the pullup.
>
> **HIGH**: `gpiod_direction_input(gpio)` ... this turns off the output, so the pullup (or some other device) controls the signal.

The same logic can be applied to emulate open source signaling, by driving the high signal and configuring the GPIO as input for low. This open drain/open source emulation can be handled transparently by the GPIO framework.

If you are "driving" the signal high but gpiod_get_value(gpio) reports a low value (after the appropriate rise time passes), you know some other component is driving the shared signal low. That's not necessarily an error. As one common example, that's how I2C clocks are stretched: a slave that needs a slower clock delays the rising edge of SCK, and the I2C master adjusts its signaling rate accordingly.