opiframe
Knowledge equals power

Opiframe Oy

GPIO and Block Driver basics

GPIO

- A "General Purpose Input/Output" (GPIO) is a flexible software-controlled digital signal.
- They are provided from many kinds of chip, 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.

GPIO

- 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.

GPIO

- 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 busses.

GPIO

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

- Output values are writable (high=1, low=0)

- Input values are likewise readable (1, 0)

- Inputs can often be used as IRQ signals, often edge triggered but sometimes level triggered.

GPIO

- 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 writeprotect status, driving a LED, configuring a transceiver, bitbanging a serial bus, poking a hardware watchdog, sensing a switch, and so on.

GPIO

- GPIOs are often used for the kind of board-specific glue logic that may even change between board revisions, and can't ever be used on a board that's wired differently.

- There are cases where portability is not the main issue.

- Only least-common-denominator functionality can be very portable.

GPIO

- The GPIO calls are available, either as "real code" or as optimized-away stubs, when drivers use the include file:
- #include <linux/gpio.h>
- GPIOs are identified by unsigned integers in the range 0..MAX_INT.
- That reserves "negative" numbers for other purposes like marking signals as "not available on this board", or indicating faults.

GPIO

- Code that doesn't touch the underlying hardware treats these integers as opaque cookies.

- Platforms define how they use those integers, and usually #define symbols for the GPIO lines so that board-specific setup code directly corresponds to the relevant schematics.

- In contrast, drivers should only use GPIO numbers passed to them from that setup code, using platform_data to hold board-specific pin configuration data (along with other board specific data they need).

GPIO

- To test if such number from such a structure could reference a GPIO, you may use this predicate:
- int gpio_is_valid(int number);
- The first thing a system should do with a GPIO is allocate it, using the gpio_request() call.

GPIO

- One of the next things to do with a GPIO, often in board setup code when setting up a platform_device using the GPIO, is mark its direction:

- int gpio_direction_input(unsigned gpio);
- int gpio_direction_output(unsigned gpio, int value);

- The return value is zero for success, else a negative errno.

- It should be checked, since the get/set calls don't have error returns and since misconfiguration is possible.

GPIO

- You should normally issue these calls from a task context. However, for spinlock-safe GPIOs it's OK to use them before tasking is enabled, as part of early board setup.

- For output GPIOs, the value provided becomes the initial output value. This helps avoid signal glitching during system startup.

- Setting the direction can fail if the GPIO number is invalid, or when that particular GPIO can't be used in that mode.

GPIO

- It's generally a bad idea to rely on boot firmware to have set the direction correctly, since it probably wasn't validated to do more than boot Linux.

- Similarly, that board setup code probably needs to multiplex that pin as a GPIO, and configure pullups/pulldowns appropriately.

GPIO

- Most GPIO controllers can be accessed with memory read/write instructions.

- Those don't need to sleep, and can safely be done from inside hard (nonthreaded) IRQ handlers and similar contexts.

- Use the following calls to access such GPIOs,
- int gpio_get_value(unsigned gpio);
- void gpio_set_value(unsigned gpio, int value);

GPIO

- The get/set calls have no error returns because "invalid GPIO" should have been reported earlier from gpio_direction_*().

- However, note that not all platforms can read the value of output pins; those that can't should always return zero.

- Also, using these calls for GPIOs that can't safely be accessed without sleeping (see below) is an error.

GPIO

- Some GPIO controllers must be accessed using message based busses like I2C or SPI.

- Commands to read or write those GPIO values require waiting to get to the head of a queue to transmit a command and get its response.

- This requires sleeping, which can't be done from inside IRQ handlers.

GPIO

- Platforms that support this type of GPIO distinguish them from other GPIOs by returning nonzero from this call

- int gpio_cansleep(unsigned gpio);

- To access such GPIOs, a different set of accessors is defined:

- int gpio_get_value_cansleep(unsigned gpio);
- void gpio_set_value_cansleep(unsigned gpio, int value);

GPIO

- Other than the fact that these accessors might sleep, and will work on GPIOs that can't be accessed from hardIRQ handlers, these calls act the same as the spinlock-safe calls.
- To help catch system configuration errors, two calls are defined.
- int gpio_request(unsigned gpio, const char *label);
- void gpio_free(unsigned gpio);

GPIO

- Passing invalid GPIO numbers to gpio_request() will fail, as will requesting GPIOs that have already been claimed with that call.
- The return value of gpio_request() must be checked.
- You should normally issue these calls from a task context.
- However, for spinlock-safe GPIOs it's OK to request GPIO before tasking is enabled, as part of early board setup.

GPIO

- These calls serve two basic purposes.

- One is marking the signals which are actually in use as GPIOs, for better diagnostics; systems may have several hundred potential GPIOs, but often only a dozen are used on any given board.

- Another is to catch conflicts, identifying errors when
- (a) two or more drivers wrongly think they have exclusive use of that signal
- (b) something wrongly believes it's safe to remove drivers needed to manage a signal that's in active use.

GPIO

- That is, requesting a GPIO can serve as a kind of lock.
- Note that requesting a GPIO does NOT cause it to be configured in any way; it just marks that GPIO as in use.
- Separate code must handle any pin setup (e.g. controlling which pin the GPIO uses, pullup/pulldown).
- Also note that it's your responsibility to have stopped using a GPIO before you free it.

GPIO

- Considering in most cases GPIOs are actually configured right after they are claimed, three additional calls are defined:
- int gpio_request_one(unsigned gpio, unsigned long flags, const char *label);
- int gpio_request_array(struct gpio *array, size_t num);
- void gpio_free_array(struct gpio *array, size_t num);

GPIO

- where 'flags' is currently defined to specify the following properties:

- GPIOF_DIR_IN - to configure direction as input
- GPIOF_DIR_OUT - to configure direction as output

- GPIOF_INIT_LOW - as output, set initial level to LOW
- GPIOF_INIT_HIGH - as output, set initial level to HIGH

GPIO as IRQ

- GPIO numbers are unsigned integers; so are IRQ numbers.
- These make up two logically distinct namespaces (GPIO 0 need not use IRQ 0).
- You can map between them using calls like:
- int gpio_to_irq(unsigned gpio);
- int irq_to_gpio(unsigned irq);

GPIO as IRQ

- Those return either the corresponding number in the other namespace, or else a negative errno code if the mapping can't be done.

- It is an unchecked error to use a GPIO number that wasn't set up as an input using gpio_direction_input()

- Or to use an IRQ number that didn't originally come from gpio_to_irq().

GPIO as IRQ

- These two mapping calls are expected to cost on the order of a single addition or subtraction.
- They're not allowed to sleep.
- Non-error values returned from gpio_to_irq() can be passed to request_irq() or free_irq()
- They will often be stored into IRQ resources for platform devices, by the board-specific initialization code.

GPIO in userspace

- As an exercise go through following blog post:
- http://falsinsoft.blogspot.fi/2012/11/access-gpio-from-linux-user-space.html