New GPIO interface for linux user space

Linux Piter 2018

Bartosz Golaszewski

About us

- Embedded Linux Engineering Firm
- ~30 senior engineers, coming from the semiconductor world
- HW and SW products: from concept to manufacturing
- Upstream Linux kernel development and maintenance
- Founding developers of kernelCI.org project

About me

- 9 years experience
- Kernel and user-space developer
- Maintainer of libgpiod and co-maintainer of GPIO kernel framework

Mandatory cute/funny picture

Agenda

1. What are GPIOs?
2. GPIO sub-system in the kernel
3. Interacting with GPIOs from user-space
    a. deprecated sysfs interface
    b. new character device
4. libgpiod
    a. What is it and what it improves
    b. Examples
    c. Bindings
    d. Future

GPIO - overview

- Stands for: General-Purpose Input/Output
- Generic pin
- Can be configured at run time
  - Input (readable)/output (writable)
  - Enabled/disabled
  - Source of interrupts
- Provided by SoCs, expanders or multifunction devices (I2C, SPI, ...)

GPIO - overview

Applications

- Pumps
- Buzzers
- Buttons
- Power Switches
- Sensors
- Stepper motors

GPIO - software stack

User space

sysfs (/sys/class/gpio)

GPIO character device

Kernel space

gpiolib kernel framework

GPIO chip driver

Hardware

GPIO in the kernel

- Two co-existing frameworks

Legacy, numberspace based
Deprecated - don't use

```
int rv = gpio_request(123,"foobar");
gpio_direction_output(123, 1);
```

:(

Currently supported, descriptor based
Associating resources with consumers

```
struct gpio_desc *gpio;
gpio = gpiod_get("foobar", GPIOD_OUT_HIGH);
gpiod_set_value_cansleep(gpio, 1);
```

:)

GPIO in the kernel - provider consumer model

GPIO chip

Line 0
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7

write-protect
enable/disable
reset assert

EEPROM driver
Regulator
Reset controller

GPIO in userspace

- Writing drivers for devices using GPIOs is encouraged wherever possible, but...
- Needed when no kernel device drivers provided/possible
  - Power switches
  - Relays
  - GPS
  - Bluetooth
- Certain users prefer to toggle GPIOs from user space
  - Intelligent home systems
  - Robotics

/sys/class/gpio - legacy user API

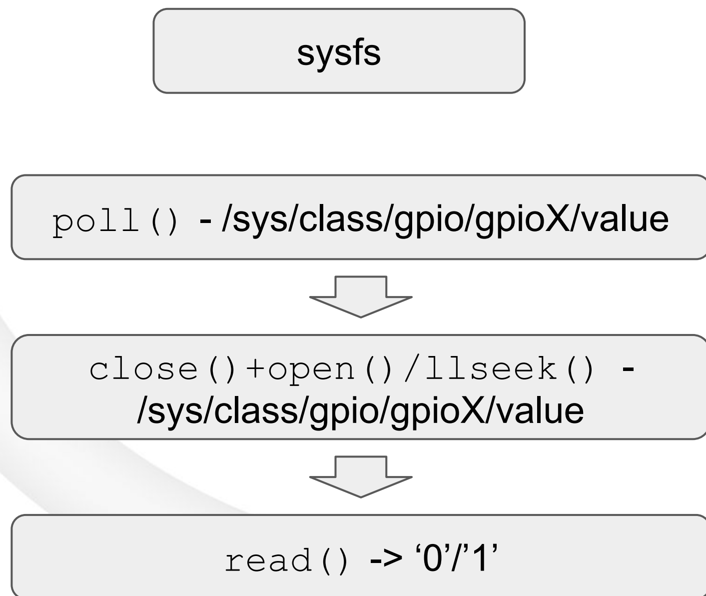
- d8f388d8 (“gpio: sysfs interface”)
- State not tied to process
- Concurrent access to sysfs attributes
- If process crashes, the GPIOs remain exported
- Cumbersome API
- Single sequence of GPIO numbers representing a two-level hierarchy - necessary to calculate the number of the GPIO, numbers not stable

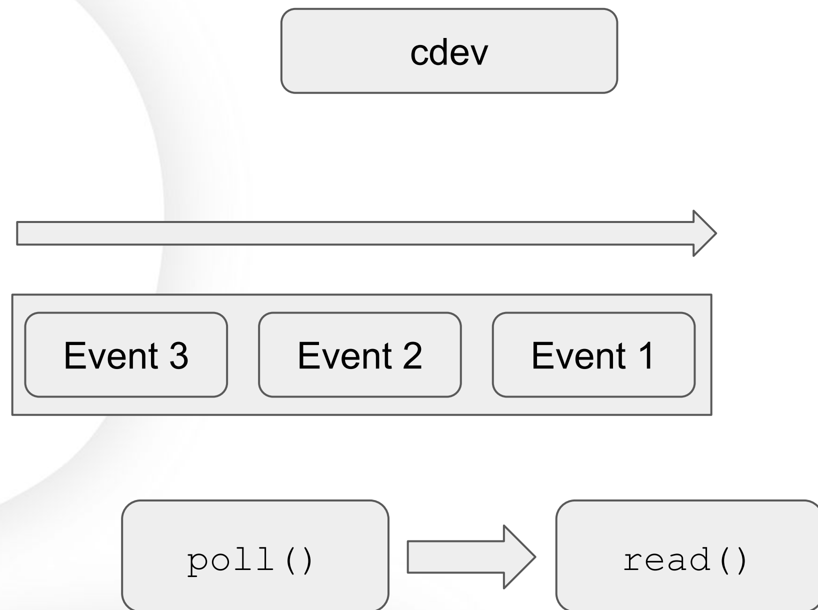
Demo: sysfs attributes with gpio-mockup

GPIO character device - new user API

- Merged in linux v4.8
- One device file per gpiochip
  - /dev/gpiochip0, /dev/gpiochip1, /dev/gpiochipX…
- Similar to other kernel interfaces: `open() + ioctl() + poll() + read() + close()`
- Possible to request multiple lines at once (for reading/setting values)
- Possible to find GPIO lines and chips by name
- Open-source and open-drain flags, user/consumer strings, uevents
- Reliable polling

GPIO event polling





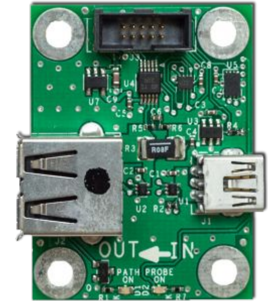
Events never get lost!

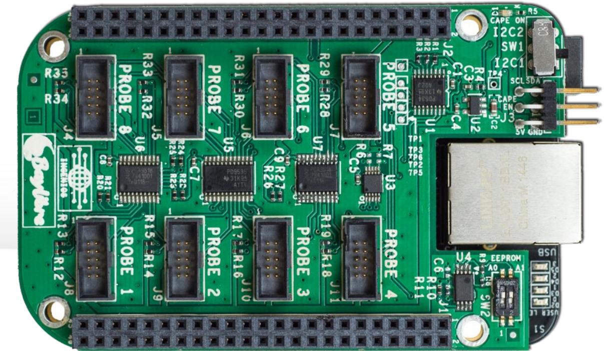
Character device – user API (linux/gpio.h)

- Logically split into:
  - Chip info
  - Line info
  - Line request for values
  - Reading values
  - Setting values
  - Line request for events
  - Polling for events
  - Reading events

libgpiod – C library & tools for GPIO chardev

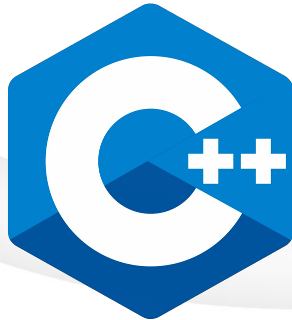
- History
  - Needed a solution for toggling power switches on BayLibre ACME
    - ~~IIO attributes~~
    - ~~Regulators controlled from user space~~
    - GPIO character device
  - Version 0.1 released on January 18th 2017
  - v1.0 released on February 7th 2018
  - Current stable version is 1.1.1
  - 1.2 coming soon
  - 0.3 series still supported





libgpiod – C library & tools for GPIO chardev

- Features
  - C API, fully documented in doxygen
  - Command-line tools: gpiodetect, gpioinfo, gpioset, gpioget, gpiofind & gpiomon
  - Custom test suite (working together with gpio-mockup kernel module and irq_sim)
- Language bindings



libgpiod – C library & tools for GPIO chardev

- C API split into logical parts:
  - Simple API
  - Chip operations
  - Line operations
    - Info
    - requests
    - events
  - Iterators

libgpiod – C API examples

```
struct gpiod_chip *chip;
struct gpiod_line *line;
int rv, value;

chip = gpiod_chip_open("/dev/gpiochip0");
if (!chip)
    return -1;

line = gpiod_chip_get_line(chip, 3);
if (!line) {
    gpiod_chip_close(chip);
    return -1
}

rv = gpiod_line_request_input(line, “foobar”);
if (rv) {
    gpiod_chip_close(chip);
    return -1;
}

value = gpiod_line_get_value(line);

gpiod_chip_close(chip)
```

libgpiod – C API examples

```
/* Error checking omitted for brevity */

struct timespec ts = { 0, 1000000 };
struct gpiod_line_event event;
struct gpiod_chip *chip;
struct gpiod_line *line;
int rv, value;

chip = gpiod_chip_open("/dev/gpiochip0");
line  = gpiod_chip_get_line(chip, 3);
gpiod_line_request_rising_edge_events(line, "foobar");

do {
      rv = gpiod_line_event_wait(line, &ts);
} while (rv <= 0);

rv = gpiod_line_event_read(line, &event);
if (!rv)
      printf("event: %s timestamp: [%8ld.%09ld]\n",
              event.event_type, event.ts.tv_sec, event.ts.tv_nsec);

gpiod_chip_close(chip)
```

Demo: libgpiod utils with gpio-mockup

libgpiod – C++ bindings

- C API wrapped in C++11 classes
- Reference counting to libgpiod resources
- Fully documented in Doxygen
- Exception-safe
- Tools reimplemented in C++ as an example
- Many examples included

libgpiod – C++ bindings examples

```
try {
        ::gpiod::chip chip("gpiochip0");
        auto lines = chip.get_lines({ 0, 4, 5, 6 });

        lines.request({ "foobar", ::gpiod::line_request::DIRECTION_OUTPUT, 0 }, { 0, 1, 0, 1 });

        lines.set_values({ 1, 0, 1, 0 });
} catch (const ::std::system_error& ex) {
        ::std::cerr << ex.what() << ::std::endl;
}
```

libgpiod – C++ bindings examples

```
::gpiod::chip chip("gpiochip0");
auto lines = chip.get_lines({ 0, 1, 2 });

lines.request({ "foobar", ::gpiod::line_request::EVENT_BOTH_EDGES, 0});

for (;;) {
    auto events = lines.event_wait(::std::chrono::nanoseconds(1000000000));
    if (events) {
        for (auto& it: events)
            print_event(it.event_read());
    }
}
```

libgpiod – Python 3 bindings

- C API wrapped in a set of Python 3 classes
- Fully documented in pydoc
- Native Python3 module written in C
- Tools reimplemented in Python as an example
- Many examples included
- Adopted by Adafruit Blinka

libgpiod – Python 3 bindings examples

```
with gpiod.Chip('gpiochip0') as chip:
    lines = chip.get_lines([ 0, 2, 3, 4 ])
    lines.request(consumer='foobar', type=gpiod.LINE_REQ_DIR_OUT, default_vals=[ 0, 1, 0, 1 ])
    vals = lines.set_values([ 1, 0, 1, 0 ])


with gpiod.Chip(sys.argv[1]) as chip:
    lines = chip.get_lines([ 0, 1, 2, 3 ])
    lines.request(consumer='foobar', type=gpiod.LINE_REQ_EV_BOTH_EDGES)

    try:
        while True:
            ev_lines = lines.event_wait(sec=1)
            if ev_lines:
                for line in ev_lines:
                    event = line.event_read()
                    print(event)
    except KeyboardInterrupt:
        sys.exit(130)
```

libgpiod – dbus bindings (coming soon)

- Work-in-progress
- git@github.com:brgl/libgpiod.git topic/gpio-dbus
- Daemon written in C and based on GDBus and Gudev
- Chip and line objects
- Properties: name, label, offset etc.
- Methods: request, set_value, get_value etc.
- Signals: line events
- DBus over network will be used with BayLibre ACME (complements IIO)

libgpiod – future

- Feature complete soon (after dbus bindings)
- Proper tests for Python and C++ bindings
- Support new user space features of future kernel versions
- Run processes on events in gpiomon

libgpiod – C library & tools for GPIO chardev

- Where to get it:
  - Hosted at kernel.org
  - Source: https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git/
  - Releases: https://www.kernel.org/pub/software/libs/libgpiod/
- Packaging
  - Available in meta-openembedded & buildroot
  - Packaged in Fedora, Arch, Debian linux and more
- Contributions & bug reports:
  - Send e-mails to linux-gpio@vger.kernel.org
  - Use [libgpiod] prefix

Q & A

Kernel uAPI code examples

Character device – chip info

```
struct gpiochip_info {
    char name[32];
    char label[32];
    __u32 lines;
};

void get_chip_info(void)
{
    struct gpiochip_info info;
    int fd, rv;

    fd = open("/dev/gpiochip0", O_RDWR);
    rv = ioctl(fd, GPIO_GET_CHIPINFO_IOCTL, info);
}
```

Character device – line info

```
struct gpioline_info {
	__u32 line_offset;
	__u32 flags;
	char name[32];
	char consumer[32];
};

#define GPIOLINE_FLAG_KERNEL		(1UL << 0)
#define GPIOLINE_FLAG_IS_OUT		(1UL << 1)
#define GPIOLINE_FLAG_ACTIVE_LOW	(1UL << 2)
#define GPIOLINE_FLAG_OPEN_DRAIN	(1UL << 3)
#define GPIOLINE_FLAG_OPEN_SOURCE	(1UL << 4)

void get_line_info(void)
{
    struct gpioline_info info;

    memset(&info, 0, sizeof(info));
    info.line_offset = 3;

    rv = ioctl(fd, GPIO_GET_LINEINFO_IOCTL, &info);
}
```

Character device – requesting lines

```
#define GPIOHANDLES_MAX                   64
#define GPIOHANDLE_REQUEST_INPUT          (1UL << 0)
#define GPIOHANDLE_REQUEST_OUTPUT         (1UL << 1)
#define GPIOHANDLE_REQUEST_ACTIVE_LOW     (1UL << 2)
#define GPIOHANDLE_REQUEST_OPEN_DRAIN     (1UL << 3)
#define GPIOHANDLE_REQUEST_OPEN_SOURCE    (1UL << 4)

struct gpiohandle_request {
      __u32 lineoffsets[GPIOHANDLES_MAX];
      __u32 flags;
      __u8 default_values[GPIOHANDLES_MAX];
      char consumer_label[32];
      __u32 lines;
      int fd;
};
```

```
void request_output(void)
{
      struct gpiohandle_request req;
      int rv;

      req.flags |= GPIOHANDLE_REQUEST_OUTPUT;
      req.lines = 2;
      req.lineoffsets[0] = 3;
      req.lineoffsets[1] = 5;
      req.default_values[0] = 1;
      req.default_values[1] = 0;
      strcpy(req.consumer_label, "foobar");

      rv = ioctl(fd, GPIO_GET_LINEHANDLE_IOCTL, &req);
}
```

Character device – reading/setting values

```
#define GPIOHANDLE_GET_LINE_VALUES_IOCTL _IOWR(0xB4, 0x08, struct gpiohandle_data)
#define GPIOHANDLE_SET_LINE_VALUES_IOCTL _IOWR(0xB4, 0x09, struct gpiohandle_data)

struct gpiohandle_data {
      __u8 values[GPIOHANDLES_MAX];
};
void get_values(void)
{
    struct gpiohandle_data data;
    int rv;

    memset(&data, 0, sizeof(data));

    rv = ioctl(req.fd, GPIOHANDLE_GET_LINE_VALUES_IOCTL,&data);
}
void set_values(void)
{
    struct gpiohandle_data data;
    int rv;

    data.values[0] = 0;
    data.values[1] = 1;

    rv = ioctl(req.fd, GPIOHANDLE_SET_LINE_VALUES_IOCTL,&data);
}
```

Character device – event requests

```
#define GPIOEVENT_REQUEST_RISING_EDGE           (1UL << 0)
#define GPIOEVENT_REQUEST_FALLING_EDGE          (1UL << 1)
#define GPIOEVENT_REQUEST_BOTH_EDGES            ((1UL << 0) | (1UL << 1))

struct gpioevent_request {
      __u32 lineoffset;
      __u32 handleflags;
      __u32 eventflags;
      char consumer_label[32];
      int fd;
};

void request_event(void)
{
    struct gpioevent_request req;
    int rv;

    req.lineoffset = 4;
    req.handleflags = GPIOHANDLE_REQUEST_INPUT;
    req.eventflags = GPIOEVENT_REQUEST_BOTH_EDGES;
    strcpy(req.consumer_label, "foobar");

    rv = ioctl(fd, GPIO_GET_LINEEVENT_IOCTL, &req);
}
```

Character device – polling & reading events

```
#define GPIOEVENT_EVENT_RISING_EDGE        0x01
#define GPIOEVENT_EVENT_FALLING_EDGE       0x02

struct gpioevent_data {
        __u64 timestamp;
        __u32 id;
};

void recv_event(void)
{
    struct gpioevent_data event;
    struct pollfd pfd;
    ssize_t rd;
    int rv;

    pfd.fd = req.fd;
    pfd.events = POLLIN | POLLPRI;

    rv = poll(&pfd, 1, 1000);
    if (rv > 0)
        rd = read(req.fd, &event, sizeof(event));
}
```