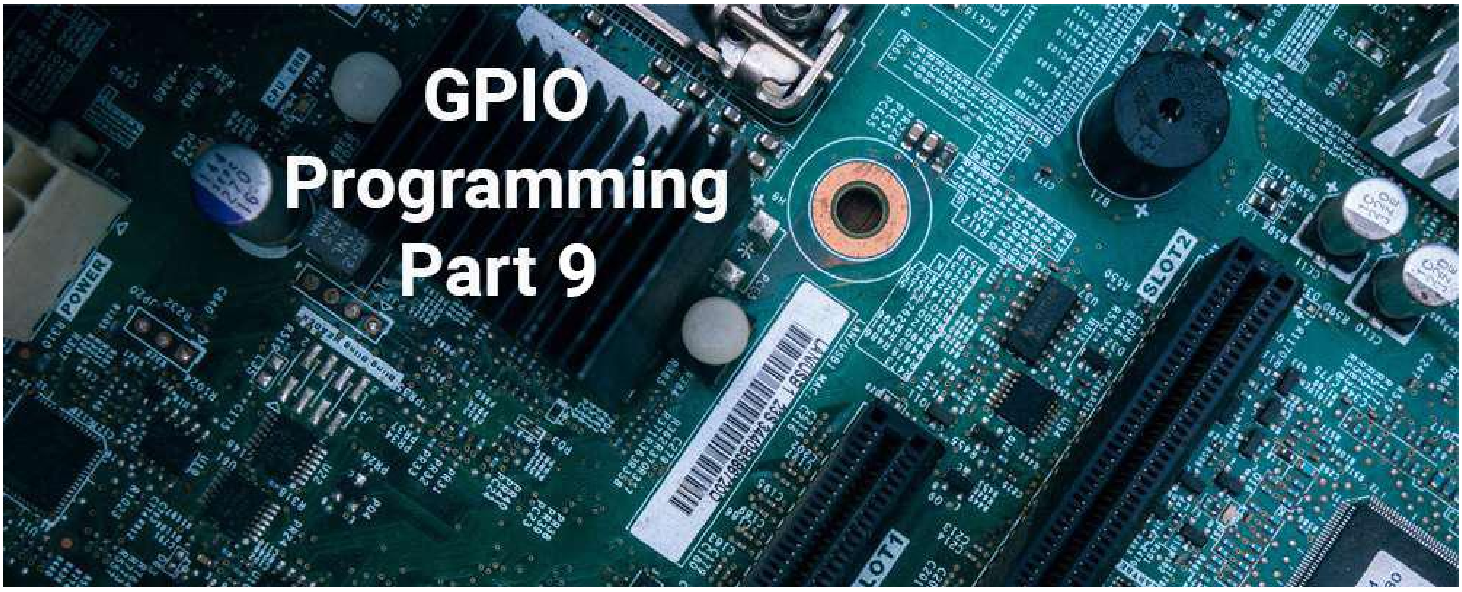

GPIO Programming: Exploring the libgpiod Library

By Jeff Tranter (/author/jeff-tranter) | Wednesday, January 20, 2021

After a hiatus of more than a year, I wanted to pick up this blog series and cover another GPIO library available on Linux. Since the last time I wrote on this topic, this library has become more widely supported and available on platforms like the Raspberry Pi, so the time was right to explore it further.

I mentioned in an earlier blog post (https://www.ics.com/blog/gpio-programming-using-sysfs-interface) that the GPIO sysfs interface was being deprecated in favor of a new GPIO character device API. In this blog post we'll look at *libgpiod*, a C library and tools for interacting with the Linux GPIO hardware.

And yes, it is a library and not a daemon process, as I initially assumed based on the common Linux convention of having long-running background processes that handle requests for services named ending in "d", like *systemd* and *cupsd*. In this context *gpiod* stands for GPIO device.

Installation

If you are running a recent version of the Raspberry Pi OS (formerly known as Raspbian), you can install the gpiod command line tools and library by installng the package *gpiod*. The header files required for development are contained in the package *libgpdiod-dev*. Similar packages are available on Ubuntu and other Debian-derived Linux distributions.

Command Line Tools

Gpiod provides a set of command line tools that are very useful for interactively exploring GPIO functions, and can be used in shell scripts to avoid the need to write C or C++ code if you only need to perform basic GPIO functions. The following commands are provided:

gpiodetect - List all GPIO chips present on the system, their names, labels and number of GPIO lines.
gpioinfo - List all lines of specified GPIO chips, their names, consumers, direction, active state and additional flags.
gpioget - Read values of specified GPIO lines.
gpioset - Set values of specified GPIO lines, and potentially keep the lines exported and wait until timeout, user input or signal.
gpiofind - Find the GPIO chip name and line offset given the line name.
gpiomon - Wait for events on GPIO lines, specifying which events to watch, how many events to process before exiting or if the events should be reported to the console.

Here is some sample output taken from a Raspberry Pi system. The *gpiodetect* program will detect the GPIO chips that are present. The library uses the term "chip" to identify groups of GPIO hardware functions which may or may not correspond to hardware-level chips. In the case of the Raspberry Pi the GPIO hardware is all contained in the Broadcom SOM (system on a module).

```
% <strong>gpiodetect</strong>
gpiochip0 [pinctrl-bcm2835] (54 lines)
gpiochip1 [raspberrypi-exp-gpio] (8 lines)
```

The output of *gpioinfo* reports all of the available GPIO lines, by default for all chips:

```
% <strong>gpioinfo</strong>
gpiochip0 - 54 lines:
    line   0:     "ID_SDA"       unused   input  active-high
    line   1:     "ID_SCL"       unused   input  active-high
    line   2:       "SDA1"       unused   input  active-high
    line   3:       "SCL1"       unused   input  active-high
    line   4:  "GPIO_GCLK"       unused   input  active-high
    line   5:      "GPIO5"       unused   input  active-high
    line   6:      "GPIO6"       unused   input  active-high
    line   7:  "SPI_CE1_N"       unused   input  active-high
    line   8:  "SPI_CE0_N"       unused   input  active-high
    line   9:   "SPI_MISO"       unused   input  active-high
    line  10:   "SPI_MOSI"       unused   input  active-high
    line  11:   "SPI_SCLK"       unused   input  active-high
    line  12:     "GPIO12"       unused   input  active-high
    line  13:     "GPIO13"       unused   input  active-high
    line  14:       "TXD1"       unused   input  active-high
    line  15:       "RXD1"       unused   input  active-high
    line  16:     "GPIO16"       unused   input  active-high
    line  17:     "GPIO17"       unused   input  active-high
    line  18:     "GPIO18"       unused   input  active-high
    line  19:     "GPIO19"       unused   input  active-high
    line  20:     "GPIO20"       unused   input  active-high
    line  21:     "GPIO21"       unused   input  active-high
    line  22:     "GPIO22"       unused   input  active-high
    line  23:     "GPIO23"       unused   input  active-high
    line  24:     "GPIO24"       unused   input  active-high
    line  25:     "GPIO25"       unused   input  active-high
    line  26:     "GPIO26"       unused   input  active-high
    line  27:     "GPIO27"       unused   input  active-high
    line  28: "RGMII_MDIO"       unused   input  active-high
    line  29:  "RGMIO_MDC"       unused   input  active-high
    line  30:       "CTS0"       unused   input  active-high
    line  31:       "RTS0"       unused   input  active-high
    line  32:       "TXD0"       unused   input  active-high
    line  33:       "RXD0"       unused   input  active-high
    line  34:    "SD1_CLK"       unused   input  active-high
    line  35:    "SD1_CMD"       unused   input  active-high
    line  36:  "SD1_DATA0"       unused   input  active-high
    line  37:  "SD1_DATA1"       unused   input  active-high
    line  38:  "SD1_DATA2"       unused   input  active-high
    line  39:  "SD1_DATA3"       unused   input  active-high
    line  40:  "PWM0_MISO"       unused   input  active-high
    line  41:  "PWM1_MOSI"       unused   input  active-high
    line  42: "STATUS_LED_G_CLK" "led0" output active-high [used]
    line  43: "SPIFLASH_CE_N" unused input active-high
    line  44:       "SDA0"       unused   input  active-high
    line  45:       "SCL0"       unused   input  active-high
    line  46: "RGMII_RXCLK" unused input active-high
    line  47: "RGMII_RXCTL" unused input active-high
    line  48: "RGMII_RXD0"       unused   input  active-high
    line  49: "RGMII_RXD1"       unused   input  active-high
    line  50: "RGMII_RXD2"       unused   input  active-high
    line  51: "RGMII_RXD3"       unused   input  active-high
    line  52: "RGMII_TXCLK" unused input active-high
    line  53: "RGMII_TXCTL" unused input active-high
gpiochip1 - 8 lines:
    line   0:      "BT_ON"       unused  output  active-high
    line   1:      "WL_ON"       unused  output  active-high
    line   2: "PWR_LED_OFF" "led1" output active-low [used]
    line   3: "GLOBAL_RESET" unused output active-high
    line   4: "VDD_SD_IO_SEL" "vdd-sd-io" output active-high [used]
    line   5:   "CAM_GPIO"       unused  output  active-high
    line   6:  "SD_PWR_ON" "sd_vcc_reg"  output  active-high [used]
    line   7:    "SD_OC_N"       unused   input  active-high
```

As you can imagine, the gpioget and gpioset commands allow reading and writing GPIO input and output lines.
A simple example is the following which sets line 24 of the first chip to a high output level for one second and then releases it:

```
% <strong>gpioset --mode=time -s 1 0 24=1</strong>
```

The following will read input channel 6 of the first GPIO chip and output it:

```
% <strong>gpioget 0 6</strong>
1
```

As I mentioned, the command line tools are often adequate for simple low-speed applications, and can be put in shell scripts or called as external programs. For better control over GPIO functions or applications which require more critical timing, you can call the APIs in the libgpiod library from a high level language such as C or C++. Note that there are also Python and other language bindings available as well.

Libgpiod Library API

The C API allows calling the gpiod library from C or languages that support C APIs like C++. The API is well documented, and too extensive to fully cover here. The basic use cases usually follows these steps:

1. Open the desired GPIO chip by calling one of the gpiod_chip_open functions such as **gpiod_chip_open_by_name()**. This returns a gpiod_chip struct which is used by subsequent API calls.
2. Open the desired GPIO line(s) by calling **gpiod_chip_get_line()** or **gpiod_chip_get_lines()**, obtaining a gpiod_line struct.
3. Request use of the line as an input or output by calling **gpiod_line_request_input()** or **gpiod_line_request_output()**.
4. Read the value of an input by calling **gpiod_line_get_value()** or set the level of an output by calling **gpiod_line_set_value()**.
5. When done, release the lines by calling **gpiod_line_release()** and chips by **calling gpiod_chip_close()**.

Other APIs are provided for more advanced functions like setting pin modes for pullup or pulldown resistors or defining a callback function to be called when an event occurs, like the level of an input pin changing.

Example Program

Let's look at a simple example, which works on a Raspberry Pi using the GPIO board that was previously discussed in this blog series. It toggles the three LEDs in a binary pattern until the pushbutton connected to a GPIO input pin is pressed. The code here was slightly simplified for readability by removing error checking. You can download the full example and a suitable CMake project file from reference 2 listed at the end of the blog post. I encourage you to try it out on a Raspberry Pi.

```
#include <gpiod.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
  const char *chipname = "gpiochip0";
  struct gpiod_chip *chip;
  struct gpiod_line *lineRed;    // Red LED
  struct gpiod_line *lineGreen;  // Green LED
  struct gpiod_line *lineYellow; // Yellow LED
  struct gpiod_line *lineButton; // Pushbutton
  int i, val;

  // Open GPIO chip
  chip = gpiod_chip_open_by_name(chipname);

  // Open GPIO lines
  lineRed = gpiod_chip_get_line(chip, 24);
  lineGreen = gpiod_chip_get_line(chip, 25);
  lineYellow = gpiod_chip_get_line(chip, 5);
  lineButton = gpiod_chip_get_line(chip, 6);

  // Open LED lines for output
  gpiod_line_request_output(lineRed, "example1", 0);
  gpiod_line_request_output(lineGreen, "example1", 0);
  gpiod_line_request_output(lineYellow, "example1", 0);

  // Open switch line for input
  gpiod_line_request_input(lineButton, "example1");

  // Blink LEDs in a binary pattern
  i = 0;
  while (true) {
    gpiod_line_set_value(lineRed, (i & 1) != 0);
    gpiod_line_set_value(lineGreen, (i & 2) != 0);
    gpiod_line_set_value(lineYellow, (i & 4) != 0);

    // Read button status and exit if pressed
    val = gpiod_line_get_value(lineButton);
    if (val == 0) {
      break;
    }

    usleep(100000);
    i++;
  }

  // Release lines and chip
  gpiod_line_release(lineRed);
  gpiod_line_release(lineGreen);
  gpiod_line_release(lineYellow);
  gpiod_line_release(lineButton);
  gpiod_chip_close(chip);
  return 0;
}
```

Summary

If you need to perform GPIO programming on a Raspberry Pi or other Linux-based embedded platform, the recommended approach is to use *gpiod*, either from a high level language like C or C++ or by using the provided command line tools. Replacing the older and now deprecated sysfs-based interface, it is more flexible, efficient, and easier to use from a high-level language.

If you missed earlier installments in our GPIO series, start here (https://www.ics.com/blog/introduction-gpio-programming).

References

1. https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git (https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git)
2. https://github.com/tranter/blogs/tree/master/gpio/part9 (https://github.com/tranter/blogs/tree/master/gpio/part9)

About the author

Jeff Tranter (/author/jeff-tranter)

Jeff Tranter is a Qt Consulting Manager at ICS. He oversees the architectural and high-level design of software systems for clients. Jeff’s leadership organizes global teams of developers on desktop and embedded applications. He has been published in *Electronic Design* magazine.