CHAPTER 13

C Program GPIO

Whether your application needs fast access or specialized access from GPIO, a C/C++ program is the most convenient way to go. Python programs likewise can have direct access with the help of a module.

This chapter looks at how to access the GPIO ports directly from within a program, starting with the unfinished business of using the `uevent` file, to detect input GPIO changes with the help of interrupts behind the scenes.

Edge Events

The previous chapter introduced the `uevents` pseudo file that the GPIO driver provides. There it was explained that you needed to use one of the system calls `poll(2)` or `select(2)` to take advantage of this notification. Here I'll illustrate the use of `poll(2)`, since it is the preferred system call of the two.

The idea behind `poll(2)` is that you supply a structured array of open file descriptors and indicate the events that you are interested in. The structure that poll(2) uses is defined as:

```
struct pollfd {
    int   fd;         /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};
```


W. Gay, *Advanced Raspberry Pi*, https://doi.org/10.1007/978-1-4842-3948-3_13

The open file descriptor is placed into the `fd` member, while the events of interest are saved to member `events`. The structure member `revents` is populated by the system call, which is available upon return.

In directory `~/RPi/evinput` you will find the C program source file `evinput.c`. The portion that performs the `poll(2)` call is illustrated in Listing 13-1.

Listing 13-1. The gpio_poll() function, invoking the poll(2) system call

```
0126: static int
0127: gpio_poll(int fd) {
0128:   struct pollfd polls[1];
0129:   char buf[32];
0130:   int rc, n;
0131:
0132:   polls[0].fd = fd;           /* /sys/class/gpio17/value */
0133:   polls[0].events = POLLPRI; /* Events */
0134:
0135:   do  {
0136:       rc = poll(polls,1,-1);  /* Block */
0137:       if ( is_signaled )
0138:           return -1;          /* Exit if ^C received */
0139:   } while ( rc < 0 && errno == EINTR );
0140:
0141:   assert(rc > 0);
0142:
0143:   lseek(fd,0,SEEK_SET);
0144:   n = read(fd,buf,sizeof buf); /* Read value */
0145:   assert(n>0);
0146:   buf[n] = 0;
0147:
0148:   rc = sscanf(buf,"%d",&n);
0149:   assert(rc==1);
```

```
0150:   return n;                    /* Return value */
0151: }
```

In this program, we are interested in only one GPIO, so the array is declared with one element:

```
0128:   struct pollfd polls[1];
```

Prior to calling poll(2), the structure polls[0] is initialized:

```
0132:   polls[0].fd = fd;          /* /sys/class/gpio17/value */
0133:   polls[0].events = POLLPRI; /* Events */
```

If there was a second entry, then polls[1] would be initialized also. After this, the system call can be invoked:

```
0136:       rc = poll(polls,1,-1);  /* Block */
```

The first argument supplies the address of the first structure entry (equivalent to `&polls[0]`). The second argument indicates how many entries apply to this call (there is only one). The last argument is a timeout parameter in milliseconds, with the negative value meaning to block forever.

If the system call returns a positive value (`rc`), this indicates how many structure entries returned an event (in member `revents`). When this happens, the caller must scan the array (`polls`) for any returned events. In theory, the program should test:

```
if ( polls[0].revents & POLLPRI )
```

to see if there was activity for this file descriptor. In this program we don't test it because only one file descriptor is provided (it's the only file descriptor that can return activity). But if you were testing for two or more GPIOs, this test would be required.

When the returned value from poll(2) is zero, it simply means that the timeout has occurred. In this program, no timeout is used, so this cannot happen.

If the returned value is -1, then the system call has returned because of an error. There is one special error code, EINTR, which will be explained shortly.

For normal read data, the event macro name to use is `POLLIN`. For the `uevent` pseudo file, the event macro name is `POLLPRI`, indicating urgent data to be read. The data to be read is indeed urgent because the state of the GPIO port could change by the time you read the `value` pseudo file. So if you're hoping to catch rising events, don't be surprised if you sometimes read back a zero. When that has happened, the rising event has come and gone by the time that the GPIO state was read.

EINTR Error

Unix veterans are quick to come to grips with the EINTR error code. We see reference to it in this loop:

```
0135:   do  {
0136:       rc = poll(polls,1,-1);  /* Block */
0137:       if ( is_signaled )
0138:           return -1;          /* Exit if ^C received */
0139:   } while ( rc < 0 && errno == EINTR );
```

The problem with `poll(2)` is that when no timeout is possible, there is no way to respond to a signal (like the terminal Control-C). The signal handler is limited in what it can do, since it is an asynchronous call, which could be interrupting the middle of a malloc(3) call, for example. For this reason, the `evinput.c` program specifies a safe interrupt handler for Control-C. It simply sets the variable `is_signaled` to `1`.

```
0018: static int is_signaled = 0;   /* Exit program when
signaled */
...
0156: static void
0157: sigint_handler(int signo) {
```

```
0158:   is_signaled = 1;            /* Signal to exit program */
0159: }
```

In order for the program to notice that the variable has changed to non-zero, the kernel returns with `rc=-1` to indicate an error, and sets the `errno=EINTR`. The code EINTR simply means that the system call was *interrupted* and should be retried. In the code presented, line 137 tests to see if that variable was set to non-zero. If it was, the function immediately returns. Otherwise, the `while` loop in line 139 keeps the system call retried in line 136.

Reading uevent

Once it has been determined that there is urgent data to be read, there is a bit of a two-step that needs to happen next. This is *not* a `poll(2)` requirement but is the driver requirement for the pseudo file `uevent`:

```
0143:   lseek(fd,0,SEEK_SET);
0144:   n = read(fd,buf,sizeof buf); /* Read value */
0145:   assert(n>0);
0146:   buf[n] = 0;
0147:
0148:   rc = sscanf(buf,"%d",&n);
0149:   assert(rc==1);
0150:   return n;                    /* Return value */
```

Line 143 effectively performs a *rewind* on the file descriptor before it reads it in line 144. This informs the driver to make its event data available for the upcoming read. Line 146 simply puts a null byte at the end of the read data, so that `sscanf(3)` can use it. Since we are expecting a `0` or `1` in text form, this is converted into the integer value `n` in line 148 and then returned.

Demonstration

To build a demonstration program, perform the following (do a "`make clobber`" if you need to force a rebuild):

```
$ make
gcc -c -Wall -O0 -g evinput.c -o evinput.o
gcc evinput.o -o evinput
sudo chown root ./evinput
sudo chmod u+s ./evinput
```

This program will not require you to sudo, because it sets the `evinput` executable to `setuid root`. On secure systems, you may want to review that.

Do display usage info, use the `-h` option:

```
$ ./evinput -h
Usage: ./evinput -g gpio [-f] [-r] [-b]
where:
       -f    detect rising edges
       -r    detect falling edges
       -b    detect both edges

Defaults are: -g17 -b
```

Specify the GPIO you want to input from with the `-g` option (17 is the default). By default, the program assumes the `-b` option, to report rising and falling edges. Let's try this now:

```
$ ./evinput -g 17 -b
Monitoring for GPIO input changes:

GPIO 17 changed: 0
GPIO 17 changed: 1
GPIO 17 changed: 0
GPIO 17 changed: 1
```

```
GPIO 17 changed: 0
^C
```

The example session shows a few changes from zero to one and back. This will not always be so clean because of contact bounce and the speed that these changes occur. Try now with rising changes:

```
$ ./evinput -g 17 -r
Monitoring for GPIO input changes:

GPIO 17 changed: 0
GPIO 17 changed: 1
GPIO 17 changed: 1
GPIO 17 changed: 1
GPIO 17 changed: 0
GPIO 17 changed: 1
^C
```

The expected value read is a `1` after a rising edge. However, notice that one zero snuck in there, which is a reminder that contact bounce and timing plays a role. The first value displayed by this program is always the initial state of the GPIO.

Multiple GPIO

The `evinput.c` program has been kept simple for illustration purposes. But the usefulness of the edge detection may have you applying it to multple GPIO ports at once. The beauty of the `poll(2)` approach is that your application will not waste CPU cycles waiting for an event to occur. Instead the GPIO *interrupt* will inform the kernel of the change when it occurs, and thus inform the `uevent` driver. This in turn will inform `poll(2)` when performed on the pseudo file.

To expand the demo code to multiple GPIOs, you will first need to open multiple `uevent` pseudo files after putting the GPIO into the correct configuration. Then you will need to expand the array `polls[]` to include the number of GPIOs of interest (line 128). Then initialize each entry as shown in lines 132 and 133.

The second argument to the `poll(2)` call in line 136 needs to match the number of initialized array elements. If you are monitoring five GPIOs, then argument two needs to be the value `5`.

After the `do while` loop ending at line 139, you will need to scan the array polls[] to determine which GPIO file descriptors reported an event with something like:

```
for ( x=0; x<rc; ++x ) {
    if ( polls[x].revents & EPOLLPRI ) {
        // read polls[x].fd for GPIO value
    }
}
```

In this manner, your application can very efficiently monitor several GPIO inputs for changes. Your code must however be able to cope with contact bounce. Some ICs like the PCF8574 I2C GPIO expander, sport an *INT* pin that can be monitored using this approach.

Direct Register Access

It is sometimes necessary for a user mode program to have direct access to the GPIO registers for performance or other reasons. This requires root access to control user access and because if done incorrectly, can crash your system. A crash is highly undesirable because it can cause loss of files.

The introduction of new Raspberry Pi models has added the challenge of dealing with different hardware platforms. With the original Raspberry Pi Model B and subsequent Model A, there was one fixed hardware offset

to the peripheral registers. However, that has changed, and we now need to calculate the correct register address depending upon the hardware model involved.

Peripheral Base Address

In order to access the GPIO peripheral registers, we need to accomplish two things:

1. Determine base address of our register set
2. Need to map physical memory into our virtual address space

Given that Raspberry Pis now differ on where the registers are physically located, we need to determine the peripheral base address. Listing 13-2 shows how the pseudo file is opened and read, to determine the actual base address.

Listing 13-2. Determining the peripheral base address

```
0315: uint32_t
0316: peripheral_base() {
0317:   static uint32_t pbase = 0;
0318:   int fd, rc;
0319:   unsigned char buf[8];
0320:
0321:   fd = open("/proc/device-tree/soc/ranges",O_RDONLY);
0322:   if ( fd >= 0 ) {
0323:       rc = read(fd,buf,sizeof buf);
0324:       assert(rc==sizeof buf);
0325:       close(fd);
0326:       pbase = buf[4] << 24 | buf[5] << 16 | buf[6] << 8 |
            buf[7] << 0;
```

```
0327:    } else  {
0328:        // Punt: Assume RPi2
0329:        pbase = BCM2708_PERI_BASE;
0330:    }
0331:
0332:    return pbase;
0333: }
```

The basic steps are:

1. Open the pseudo file (line 321).
2. Read the first 8 bytes into character array buf (line 323).
3. Once read, the file descriptor can be closed (line 325).
4. Piece together the address in line 326.
5. If step 1 fails, assume the value of macro `BCM2708_PERI_BASE` (which is `0x3F00000`).

Mapping Memory

The next step in direct access to GPIO registers involves mapping physical memory into the C/C++ program's virtual memory. Listing 13-3 illustrates how physical memory is mapped.

Listing 13-3. Mapping physical memory

```
0274: void *
0275: mailbox_map(off_t offset,size_t bytes) {
0276:    int fd;
0277:
0278:    fd = open("/dev/mem",O_RDWR|O_SYNC);
0279:    if ( fd < 0 )
0280:        return 0;       // Failed (see errno)
```

```
0281:
0282:   void *map = (char *) mmap(
0283:       NULL,                   // Any address
0284:       bytes,                  // # of bytes
0285:       PROT_READ|PROT_WRITE,
0286:       MAP_SHARED,             // Shared
0287:       fd,                     // /dev/mem
0288:       offset
0289:   );
0290:
0291:   if ( (long)map == -1L ) {
0292:       int er = errno;     // Save errno
0293:       close(fd);
0294:       errno = er;         // Restore errno
0295:       return 0;
0296:   }
0297:
0298:   close(fd);
0299:   return map;
0300: }
```

The basic steps performed are as follows:

1. First memory is accessed by opening `/dev/mem`, for read and write (line 278). This step requires root access to protect the integrity of the system.
2. Once that file is open, the `mmap(2)` system call is used to map it into the caller's virtual memory (lines 282 to 289).
    a. The first argument of the call is NULL, to specify that any virtual memory address is acceptable. This address can be specified, but the call will fail if the kernel finds it unacceptable.

b. Argument two is the number of bytes to map for this region. In our demo program, this is set to the kernel's page size. It needs to be a multiple of the page size.

c. Argument three indicates that we want to read and write the mapped memory. If you only want to query registers, macro `PROT_WRITE` can be dropped.

d. Argument four is `MAP_SHARED` allowing our calling program to share with any other processes on the system that might be accessing the same region.

e. The fifth argument is the file descriptor that we have open.

f. The last argument is the starting offset of physical memory that we wish to have access to.

3. If the `mmap(2)` call fails for any reason, the return value will be a long negative one. The value `errno` will reflect the reason why (lines 291 to 296).

4. Otherwise, the file can be closed (line 298) since the memory access has already been granted. The virtual memory address is returned in 299.

Register Access

Once the required memory has been *mapped*, it is possible to directly access the peripheral registers. To calculate the correct virtual memory address of a given register, macros are used like this one:

```
0040: #define GPIO_BASE_OFFSET  0x200000    // 0x7E20_0000
```

Additional macros reference a specific register relative to the base offset. For example, this macro provides an offset to the register that permits setting of GPIO bits.

```
0052: #define GPIO_GPSET0   0x7E20001C
```

These register accesses are rather messy. In the example `gp.c` program, the following `gpio_read()` function uses a set_gpio32() helper function to determine:

1. The register address (saved to `gpiolev`, line 232).
2. The bit `shift` needed (saved to variable shift, line 227).
3. From the required register to be accessed (`GPIO_GPLEV0`, line 232).

This procedure provides the calculated word address in `gpiolev`, and a `shift` value to use to reference a specific bit. Listing 13-4 illustrates the code for this procedure.

Listing 13-4. C function, gpio_read() to read a GPIO input bit

```
0225: int
0226: gpio_read(int gpio) {
0227:     int shift;
0228:
0229:     if ( gpio < 0 || gpio > 31 )
0230:         return EINVAL;
0231:
0232:     uint32_v *gpiolev = set_gpio32(gpio,&shift,GPIO_
          GPLEV0);
0233:
0234:     return !!(*gpiolev & (1<<shift));
0235: }
```

Line 234 then accesses the register containing the GPIO bit of interest and returns it to the caller.

Write access is similar, except that the register is written with values (Listing 13-5).

Listing 13-5. Writing GPIO registers by writing to the register address

```
0241: int
0242: gpio_write(int gpio,int bit) {
0243:   int shift;
0244:
0245:   if ( gpio < 0 || gpio > 31 )
0246:       return EINVAL;
0247:
0248:   if ( bit ) {
0249:       uint32_v *gpiop = set_gpio32(gpio,&shift,GPIO_
            GPSET0);
0250:           *gpiop = 1u << shift;
0251:   } else  {
0252:       uint32_v *gpiop = set_gpio32(gpio,&shift,GPIO_
            GPCLR0);
0253:       *gpiop = 1u << shift;
0254:   }
0255:   return 0;
0256: }
```

The one difference between read and write. however, is that the Pi has different registers to set GPIO bits (line 249) and another to clear them (line 252).

Demonstration Program

Build the source code in ~/RPi/gpio (perform "make clobber" if you wish to force a complete rebuild):

```
$ make
gcc -c -Wall -O0 -g gp.c -o gp.o
gcc gp.o -o gp
sudo chown root ./gp
sudo chmod u+s ./gp
```

Once again, this program uses setuid root so that you are not forced to do a sudo to use it. The program has usage information, with the application of the -h option:

```
$ ./gp -h | expand -t 8
Usage: ./gp -g gpio { input_opts | output_opts | -a | drive_
opts} [-v]
where:
        -g gpio GPIO number to operate on
        -A n    Set alternate function n
        -a      Query alt function
        -q      Query drive, slew and hysteresis
        -v      Verbose messages

Input options:
        -i n    Selects input mode, reading for n seconds
        -I      Input mode, but performing one read only
        -u      Selects pull-up resistor
        -d      Selects pull-down resistor
        -n      Selects no pull-up/down resistor

Output options:
        -o n    Write 0 or 1 to gpio output
        -b n    Blink for n seconds
```

```
Drive Options:
        -D n    Set drive level to 0-7
        -S      Enable slew rate limiting
        -H      Enable hysteresis
```

All invocations require the specification of the `-g` option to provide the GPIO number to operate upon. Option `-v` can be added to provide additional output.

GPIO Input

The following example session configures the GPIO port 17 as an input and selects pull-up high reading for 60 seconds:

```
$ ./gp -g17 -i60
GPIO = 1
GPIO = 0
GPIO = 1
GPIO = 0
GPIO = 1
GPIO = 0
GPIO = 1
GPIO = 0
```

Your session output might show some contact bounce, so don't expect all transitions to be consistently ones alternating with zeros.

For a one time read of input, use `-I` instead:

```
$ ./gp -g17 -I
GPIO = 1
```

GPIO Output

To configure a GPIO as an output and write a value to it, use the following command:

```
$ ./gp -g17 -o1 -v
gpio_peri_base = 3F000000
Wrote 1 to gpio 17
$ ./gp -g17 -o0 -v
gpio_peri_base = 3F000000
Wrote 0 to gpio 17
```

In this session, the verbose option was added for visual confirmation.

It is useful to have a blinking output when testing. Do this using the -b option. The argument specifies the number of seconds to blink for:

```
$ ./gp -g17 -b4 -v
gpio_peri_base = 3F000000
GPIO 17 -> 1
GPIO 17 -> 0
GPIO 17 -> 1
GPIO 17 -> 0
```

Drive, Hysteresis, and Slew

The drive, slew rate limiting, and hysteresis options can be both set and queried by the -D and -q options. -D sets the values and -q queries:

```
$ ./gp -g17 -D7, -S1 -H0 -q -v
gpio_peri_base = 3F000000
  Set Drive=7, slew=true, hysteresis=false
  Got Drive=7, slew=true, hysteresis=false
```

The Set Drive line is suppressed without the verbose option. The -q option is performed after the set operation and reports on the configuration after the change. It can be used to query only:

```
$ ./gp -g17 -q
  Got Drive=7, slew=true, hysteresis=false
```

Alternate Mode

The alternate mode of the GPIO can also be queried and set:

```
$ ./gp -g17 -a
GPIO 17 is in Output mode.
```

Setting the alternate mode is done with the -A option:

```
$ ./gp -g17 -A5
$ ./gp -g17 -a
GPIO 17 is in ALT5 mode.
```

Transistor Driver

Before we leave the topic of GPIO, let's review a simple transistor driver that can be used in situations where a complete IC solution might be overkill. The GPIO pins of the Raspberry Pi are limited in their ability to drive current. Even configured for full drive, they are limited to 16 mA.

Rather than enlist a buffer IC, you might find that you only need one signal buffered. A cheap utility transistor like the 2N2222A may be all you need. Figure 13-1 illustrates the circuit.

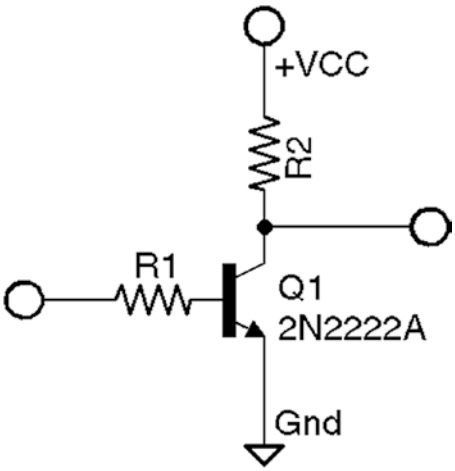

Figure 13-1. *A simple bipolar transistor driver*

The input signal arrives from the GPIO output on the left side of the circuit and flows through resistor R1, through the base emitter junction to ground. R1 limits this current to a safe value. Resistor R2 connects between the collector of Q1 and the power supply, which can be somewhat higher than +3.3 V. This is safe because the collector base junction is reversed biased. Be careful to not exceed the collector base voltage, however.

The maximum power that Q1 can handle is 0.5 W at 25°C. When the transistor is conducting (saturated), the voltage across Q1 (V_{CE}) is between 0.3 and 1 V. The remainder of the voltage is developed across the load. If we assume the worst case of 1 V for V_{CE} we can compute the maximum current for Q1:

$$\begin{aligned} I_C &= \frac{P_{Q1}}{V_{CE}} \\ &= \frac{1}{0.3} \\ &= 3.3A \end{aligned}$$

This calculated current exceeds the datasheet limit for I_C=600 mA, so we now switch to using 600 mA instead. Let's assume that we only need 100 mA, rather than the absolute limit.

Next we want to know the lowest applicable H_{FE} for the part being used at the collector current chosen. Based upon a STMicroelectronic datasheet, it is estimated that the lowest H_{FE} is about 50 near 100 mA. This value is important because it affects how much base current drive is needed.

$$\begin{aligned} I_B &= \frac{I_C}{H_{FE}} \\ &= \frac{100mA}{50} \\ &= 2mA \end{aligned}$$

Knowing now, the minimum base current to drive the transistor, we can compute the base resistor R1:

$$\begin{aligned} R_1 &= \frac{GPIO_{HIGH} - V_{BE}}{I_B} \\ &= \frac{3 - 0.7}{0.002} \\ &= 1150ohms \end{aligned}$$

The nearest 10% resistor value is 1.2 kohms.

Inductive Loads

It's not unusual to drive a relay coil when larger currents or voltages are involved. The problem with inductive loads, however, is that when the magnetic field collapses, a reverse voltage is induced into the driving circuit. This occurs when the coil current is removed. Special care must be taken to suppress this. Figure 13-2 illustrates a transistor driving a relay coil. The relay opens and closes the load contacts K1.

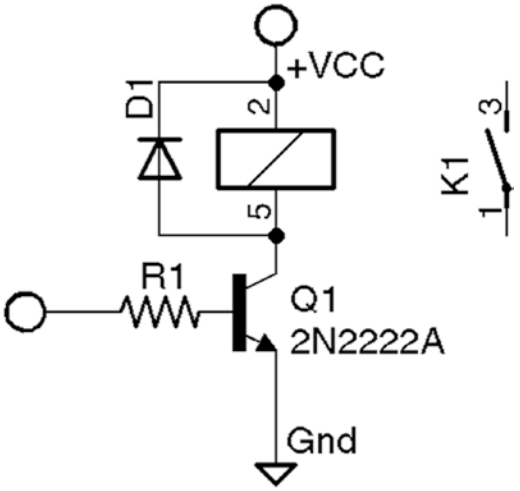

Figure 13-2. *An inductive load driven by Q1*

The relay coil needs a reversed biased diode (D1) across it to bleed any reverse kick that occurs when the current is removed from the coil (pins 5 and 2 in the figure). This will have the effect of slowing the release of the contacts. But this is preferred over the inductive spike causing a system crash.

Summary

The source code provided in `gp.c` is written entirely in the C language and kept to the bare essentials. It not only demonstrates the direct register access steps involved but provides you with code that you can reuse in your own C/C++ programs.

The chapter finished with a brief look at using a driver transistor when a driver is needed. Often an IC is sought when a cheaper one-transistor solution may be enough.