Realtime embedded coding under Linux

Bernd Porr

February 23, 2022

Contents

Chapter 1

Writing C++ device driver classes

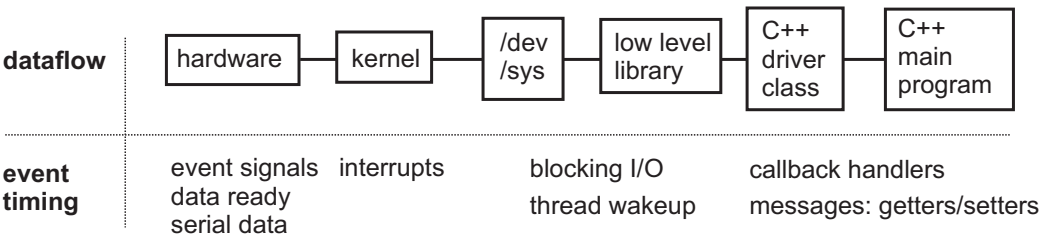

Figure 1.1: Dataflow and timing in low level realtime coding

1.1 Introduction

Fig. 1.1 shows the basic dataflow and how event timing is established. While it's obvious that data needs to flow from/to the hardware it's even more important to guarantee its timing in realtime applications. On the hardware-side the timing is guaranteed by event signals, data-ready signals and also by the timing of a serial interface. The Linux kernel translates this timing info into blocking I/O on pseudo filesystems such as /dev or /sys which means that a read operation blocks till data has arrived or an event has happened. Some low level libraries such as pigpio translate them back into callbacks but generally that needs to be done by you within a C++ class.

Ultimately, data transmission between the client and the C++ device driver is achieved with *setters* and *callbacks*:

- C++ device driver class → client **Callbacks**
- Client → C++ device driver class **Setters**

This chapter focuses on writing your own C++ class hiding away the complexity (and messy) low level C APIs and/or raw device access to /dev and /sys.

1.2 General recommendations how to write your C++ classes for devices

As said above the main purpose of object oriented coding here is to hide away the complexity of low level driver access and offer the client a simple and safe way of connecting to the sensor. In particular:

1. Getters, setters and callbacks hand over *physical units* (temperature, acceleration, ...) and not raw integer values which have no meaning.
2. The sensor is configured by specifying physical units (time, voltage, temperature) and not sensor registers. Default config parameters should be specified that the class can be used straight away with little knowledge of the details.
3. The class handles the realtime processing by offering callback interfaces (i.e. based on classes with virtual and abstract methods) to transmit data from the sensor and methods to transmit data to the sensor.
4. The class is re-usable outwith of your specific project and has its own cmake project (for example in a subdirectory), i.e. is a library. It has doc-strings for all public functions and constants, has documentation generated by doxygen.
5. It has simple demo programs demonstrating how to use the class by a client.

Keep S.O.L.I.D. https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design in mind when writing your C++ device classes:

1. *Single responsibility*: If you have a temperature sensor and an accelerometer then write two classes, one for the temperature sensor and one of the accelerometer. In terms of

2. *Open-Closed principle* For example an ADC class has a callback which returns voltage to the client. However, you'll be connecting for example a temperature sensor to it so you'd like to be able to extend the class by for example overloading the callback methods so that you add the conversion from volt to degrees but not hacking the existing ADC class.

3. *Interface Segregation Principle*: Keep functionality separate and rather divide it up in different classes. Imagine you have a universal IO class with SPI and I2C but your client really just needs SPI. Then the client is forced to deactivate I2C or in the worst case the class causes collateral damage without the client knowing why.

1.3 Low level userspace device access

The following sections provide pointers of how to write the C++ driver classes for different hardware protocols.

1.3.1 SPI

Table 1.1: SPI modes

SPI Mode	CPOL	CPHA	Idle state
0	0	0	L
1	0	1	L
2	1	1	H
3	1	0	H

SPI is a protocol which usually transmits and receives at the same time. Even that data might not be used it needs to be matched up. So for example sending 8 byes and recieving 8 bytes at the same time.

Transfer to/from SPI is best managed by the low level access to /dev. Open the SPI device with the standard open() function:

```
int fd = open( "/dev/spidev0.0", O_RDWR);
```

Then set the SPI mode (see table. 1.1):

```
int ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
```

which is explained, for example, here: https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html.

Since SPI transmits and receives at the same time we need to use ioctl to do the communication. Populate this struct:

```
struct spi_ioc_transfer tr = {
  .tx_buf = (unsigned long)tx1,
  .rx_buf = (unsigned long)rx1,
  .len = ARRAY_SIZE(tx1),
  .delay_usecs = delay,
  .speed_hz = speed,
  .bits_per_word = 8,
};
```

which points to two character buffers "tx" and "rx" with the same length.

Reading and simultaneous writing is happening then via the ioctrl function:

```
int ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
```

Sometimes the SPI protocol of a chip is so odd that even the raw I/O via /dev won't work and you need to write your own bit banging interface, for example done here for the ADC on the alphabot: https://github.com/berndporr/alphabot/blob/main/alphabot.cpp#L58. This is obviously far from ideal as it might require "usleep" commands so that acquisition needs to be run in a separate thread (the alphabot uses a timer callback in a separate thread).

Overall the SPI protocol is often device dependent and calls for experimentation to get it to work. Often the SPI clock is also the ADC conversion clock which requires a longer lasting clock signal by transmitting dummy bytes in addition to the payload.

As a general recommendation do not use SAR converters which use the SPI data clock also as acquisition clock as they are often not compatible with the standard SPI transfers via /dev. Use sensors or ADCs which have their own clock signal.

1.3.2 I2C

The I2C bus has two signal lines (SDA & SDL) which must be pulled up by resistors. Every I2C device has an address on the bus. You can scan a bus with "i2cdetect" (part of the i2c-tools package):

```
root@raspberrypi:/home/pi# i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- 1e --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- 58 -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- 6b -- -- -- --
70: -- -- -- -- -- -- -- --
root@raspberrypi:/home/pi#
```

In this case there are 3 I2C devices on the I2C bus at addresses 1E, 58 and 6B and need to be specified when accessing the I2C device.

Raw /dev/i2c access

I2C either transmits or receives but never at the same time so here we can use the standard C read/write commands. However, we need to use ioctrl to tell the kernel the I2C address:

```
char buf[2];
int file = open("/dev/i2c-2",O_RDWR);
int addr = 0x58;
ioctl(file, I2C_SLAVE, addr);
write(file,buf,1)
read(file,buf,2)
```

where "addr" is the I2C address. Then use standard read() or write() commands. Usually the 1st write() operation tells the chip which register to read or write to. Then write/read its register.

I2C access via pigpio

Access via pigpio (http://abyz.me.uk/rpi/pigpio/cif.html) is preferred in contrast to direct access of the raw /dev/i2c because many different devices can be connected to the I2C bus and pigpio manages this. Simply install the development package:

```
sudo apt-get install libpigpio-dev
```

which triggers then the install of the other relevant packages. For example writing a byte to a register in an I2C sensor can be done with a few commands:

```
int fd = i2cOpen(i2c_bus, address, 0);
i2cWriteByteData(fd, subAddress, data);
i2cClose(fd);
```

where i2c_bus is the I2C bus number (usually 1 on the RPI) and the address is the I2C address of the device on that bus. The subAddress here is the register address in the device.

1.3.3 Access GPIO pins

/sys filesystem

The GPIO of the raspberry PI can easily be controlled via the /sys filesystem. This is slow but good for debugging as you can directly write a "0" or "1" string to it and print the result. The pseudo files are here:

```
/sys/class/gpio
```

which contains files which directly relate to individual pins. To be able to access a pin we need to tell Linux to make it visible:

```
/sys/class/gpio/export
```

For example, writing a 5 (in text form) to this file would create the subdirectory `/sys/class/gpio/gpio5` for GPIO pin 5.

Then reading from

```
/sys/class/gpio/gpio5/value
```

would give you the status of GPIO pin 5 and writing to it would change it. A thin wrapper around the GPIO sys filesystem is here: https://github.com/berndporr/gpio-sysfs.

GPIO interrupt handling via /sys The most important application for the /sys filesystem is to do interrupt processing in userspace. A thread can be put to sleep until an interrupt has happend on one of the GPIO pins. This is done by monitoring the "value" of a GPIO pin in the /sys filesystem with the "poll" command:

```
struct pollfd fdset[1];
int nfds = 1;
int gpio_fd = open("/sys/class/gpio/gpio5/value", O_RDONLY | O_NONBLOCK );
memset((void*)fdset, 0, sizeof(fdset));
fdset[0].fd = gpio_fd;
fdset[0].events = POLLPRI;
int rc = poll(fdset, nfds, timeout);
if (fdset[0].revents & POLLPRI) {
   // dummy
   read(fdset[0].fd, buf, MAX_BUF);
}
```

makes the thread go to sleep until an interrupt has occurred on GPIO pin 5. Then the thread wakes up and execution continues.

pigpio

The above section has given you a deep understanding what's happening under the hood on the sysfs-level but it's highly recommended to use the pigpio library (http://abyz.me.uk/rpi/pigpio/cif.html) to read/write to GPIO pins or do interrupt programming.

For example to set GPIO pin 24 as an input just call:

```
gpioSetMode(24,PI_INPUT);
```

To read from GPIO pin 24 just call:

```
int a = gpioRead(24)
```

interrupt handling via pigpio pigpio manages GPIO interrupt handling by wrapping all the above functionality into a single command where the client registers a callback function which is called whenever a change has occurred on a GPIO pin. Specifically a method of the form:

```
class mySensorClass {
  ...
  static void gpioISR(int gpio, int level, uint32\_t tick, void* userdata)
  ...
}
```

is registered with pigpio:

```
gpioSetISRFuncEx(24,RISING_EDGE,ISR_TIMEOUT,gpioISR,(void*)this);
```

where "this" is the pointer to your class instance which is then used to call a class method, here: "dataReady()".

```
class LSM9DS1 {
  void dataReady();
  static void gpioISR(int gpio, int level, uint32_t tick, void* userdata)
    {
        ((LSM9DS1*)userdata)->dataReady();
    }
};
```

where here within the static function the void pointer is cast back into the instance pointer. See https://github.com/berndporr/LSM9DS1_RaspberryPi_CPP_Library for the complete code.

1.3.4 Access to hardware via special devices in /sys

Some sensors are directly available via the sys filesystem in human readable format.

For example

```
cat /sys/class/thermal/thermal_zone0/temp
```

gives you the temperature of the CPU.

1.3.5 I2S: Audio

The standard framework for audio is alsa: https://github.com/alsa-project.

ALSA works packet based where a read command returns a chunk of audio or a chunk is written to.

First, the parameters are requested and the driver can modify or reject them:

```
/* Signed 16-bit little-endian format */
  snd_pcm_hw_params_set_format(handle, params,
                               SND_PCM_FORMAT_S16_LE);
```

```
  /* One channel (mono) */
  snd_pcm_hw_params_set_channels(handle, params, 1);

  /* 44100 bits/second sampling rate (CD quality) */
  val = 44100;
  snd_pcm_hw_params_set_rate_near(handle, params,
                                  &val, &dir);
```

Then playing sound is done in an endless loop were a read() or write() command is issued. Both are blocking so that it needs to run in a thread:

```
while(running) {
    rc = snd_pcm_writei(handle, buffer, frames);
    if (rc == -EPIPE) {
      /* EPIPE means underrun */
      fprintf(stderr, "underrun occurred\n");
      snd_pcm_prepare(handle);
    } else if (rc < 0) {
      fprintf(stderr,
              "error from writei: %s\n",
              snd_strerror(rc));
    }  else if (rc != (int)frames) {
      fprintf(stderr,
              "short write, write %d frames\n", rc);
    }
}
```

For a full coding example "aplay" is a very good start or "arecord". Both can be found here: https://github.com/alsa-project.

1.3.6 Accessing physical memory locations (danger!)

In case you really need to access registers you can access also memory directly. This should only be used as a last resort. For example, setting the clock for the AD converter requires turning a GPIO pin into a clock output pin. This is not yet supported by the drivers so we need to program registers on the RPI.

- Linux uses virtual addressed so that a pointer won't point to a physical address. It points to three page tables with an offset.

- Special device /dev/mem which allows access of physical memory.
- The command "mmap" provides a pointer to a physical address by opening /dev/mem.
- Example:

```
int *addr;
if ((fd = open("/dev/mem", O_RDWR|O_SYNC)) < 0 ) {
    printf("Error opening file. \n");
    close(fd);
    return (-1);
}
addr = (int *)mmap(0, num*STRUCT_PAGE_SIZE, PROT_READ, MAP_PRIVATE,
            fd, 0x0000620000000000);
printf("addr: %p \n",addr);
printf("addr: %d \n",*addr);
```

- Use this with care! It's dangerous if not used properly.

1.4 Callbacks in C++ device classes

As said in the introduction the hardware device class has callback interfaces which are more complicated as setters as they call the client.

There are different ways of tackling the issue of callbacks but the simplest one is defining a method as *abstract* and asking the client to implement it in a derived class. That abstract function can either be in a separate class or part of the device class itself. So, we have two options:

1. The callback is part of the device driver class:

```
class MyDriver {
          void start(DevSettings settings = DevSettings() );
          void stop();
          virtual void callback(float sample) = 0;
};
```

2. The callback is port of an interface class:

```
class CallbackInterface {
          virtual void callback(float sample) = 0;
};
```

and then registering it in the main device driver class:

```
class MyDriver {
          void registerCallback(CallbackInterface* cb);
};
```

These two options are now explained in greater detail.

1.4.1 Creating a callback interface

Here, we create a separate interface class containing a callback as an abstract method:

```
class LSM9DS1callback {
public:
        virtual void hasSample(LSM9DS1Sample sample) = 0;
};
```

The client then implements the abstract method "hasSample()", instantiates the interface class and then saves its pointer in the device class, here called "lsm9ds1Callback".

```
void LSM9DS1::dataReady() {
        LSM9DS1Sample sample;
        // fills the sample struct with data
        // ...
        lsm9ds1Callback->hasSample(sample);
}
```

The pointer to the interface instance is transmitted via a setter which receives the pointer of the interface as an argument, for example:

```
        void registerCallback(LSM9DS1callback* cb);
```

This allows to register a callback optionally. The client might not need one or not always. See https://github.com/berndporr/rpi_AD7705_daq for a complete example.

1.4.2 Adding directly an abstract method to the device driver class

Instead of creating a separate class containing the callback you can also add the callback straight to the device driver class.

```
class ADS1115rpi {
        ...
        virtual void hasSample(float sample) = 0;
        ...
};
```

This forces the client to implement the callback to be able to use the class. This creates a very safe environment as all dependencies are set at compile time and the abstract nature of the base class makes clear what needs to be implemented. See https://github.com/berndporr/rpi_ads1115 for a complete example.

1.5 Kernel driver programming

This is beyond the scope of this handout put the best approach is to find a kernel driver which does approximately what you want and then modify it for your purposes. Kernel drivers then create an special device in /dev or /sys and then your userspace C++ can then communicate with it. For example the COMEDI framework for data acquisition has many drivers for data acquisition cards and it shows up as /dev/comedi0,1,2 etc.

1.6 Conclusion

From the sections above it's clear that Linux userspace low level device access is complex, even without taking into account the complexity of contemporary chips which have often a multitude of registers and pages of documentation. Your task is to hide away all this (scary) complexity in a C++ class with a callback and manage the settings with a struct.