Realtime embedded coding in C++ under Linux

Bernd Porr & Nick Bailey

March 15, 2023

Contents

Chapter 1

Introduction

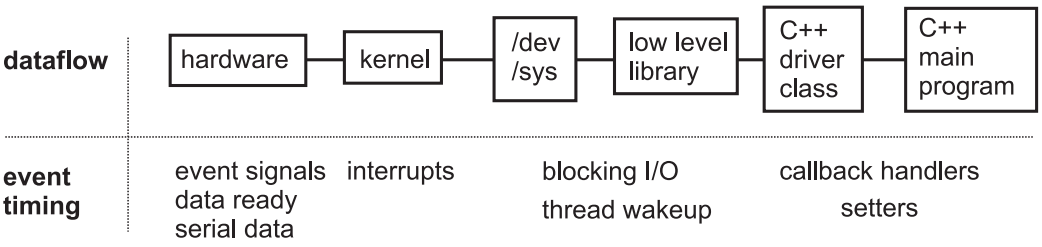

Figure 1.1: Dataflow and timing in low level realtime coding

1.1 Event based coding in the Linux userspace

Realtime embedded coding is all about *events*. These can be a binary signal such as somebody opening a door or an ADC signalling that a sample is ready to be picked up. Fig. 1.1 shows the basic dataflow and how event timing is established: devices by themselves have event signals such as "data ready" or "crash sensor has been triggered". The Linux kernel receives these as interrupt callbacks. However, userspace has no direct interrupt mechanism; instead it has blocking I/O where a read or write operation blocks until a kernel-side interrupt has happened. A blocking I/O call returning may then be translated callbacks between classes by waking up threads. Data is transmitted back to the hardware via methods called "setters", which change an object's attributes and potentially do other processing.

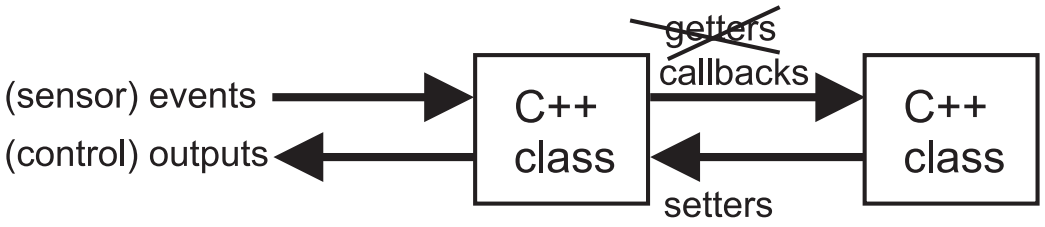

Figure 1.2: A realtime system with two C++ classes. Communication between classes is achieved with callbacks (not getters) for incoming events and setters to send out control events. The control output itself receives its timing from the events so that the loop is traversed as quickly as possible.

Fig. 1.2 shows the overall communication between C++ classes in a realtime system. This communication is done via **callbacks** (*not* getters) and setters, where an event from a sensor traverses according to its realtime requirements through the C++ classes via callbacks and then back to the control output via setters. For example, a collision sensor at a robot triggers a GPIO pin, which then triggers a callback to issue an avoidance action which in turn then sets the motors in reverse.

1.2 S.O.L.I.D.

When developing the C++ classes keep S.O.L.I.D. in mind:

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. Don't write one class `Hardware` which takes an argument `Hardware::temp` or `Hardware::accel` and then does one of two completely different things. It's a debugging nightmare and wasteful if you want to reuse your class in hardware with *only* a temperature sensor *or* an accelerometer.

Open-Closed principle: Your class is open to extension but closed to modification. 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 overloading the callback methods so that you add the conversion from volt to degrees. This is a Good Thing™ so long as you create a derived class. It's a bad idea to hack

the existing ADC class adding a `to_kelvin` method: why would somebody using the ADC to read a value from an accelerometer need that?

Liskov substitution principle: Strictly, substituting a derived class for its base class does not result in the program becoming incorrect. That is to say, any derived class from your device driver class can be used in place of the base class if the base class is all that's required, because the extra functionality in the derived class shouldn't break the basic required functionality of the base class. For example, if you have a super-duper DAC with lots of extra features, it shouldn't stop you using it when you only need a very simple one. This also means that sensible default values should be set so that the client won't need to understand the nerdy features of that super-duper DAC.

Interface Segregation principle: Keep functionality separate and aim to 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.

Dependency inversion: That is about abstracting the essential features of a class of interfaces. For example, ideally you want a base class covering a range of similar ADC converters from the same manufacturer and not a base class being a driver for a particular chip. Individual ADC chip driver classes then inherit from the abstract ADC driver. The authors of the ADC's driver will no doubt consider their chip's particular capabilities to be the core ideas, but this needs inverting. Why would one ADC driver need to provide all the necessaray code for a different one of the same family? Instead, the driver is dependent on the abstract idea of an ADC, not the other way around. The concrete depends on the abstract.

1.3 Additional coding requirements

Besides S.O.L.I.D it's also essential that:

1. you use **C++11 (or later) memory management**: copy constructors, standard template library (STL) containers and no raw pointers managed with new/delete.

2. the project has a **build system** such as cmake. It's strongly recommended to use **cmake** (autoconf only for older existing projects). This is especially true for Qt projects, where cmake support is a stated priority of the Qt developers.

3. classes (in particular driver classes) are **re-usable** outwith of the specific project and have their own cmake-projects in their subdirectories. Except for the main cmake project all sub-projects are libraries. This aids testing and reuse.

4. all public interfaces have **doc-strings** for all public methods/constants and an automatically generated reference, for example with the help of **doxygen**.

5. classes which perform internal processing such as filters, databases, detectors, ..., have **unit tests** and are run via the cmake testing framework.

6. the documentation provides comprehensive information about the project itself, how to install and run the project.

In the following sections we are describing how to write event driven code in C++. Important here is the interplay between blocking I/O and threads: chapter 2 focuses on how to wrap the low level blocking Linux I/O in a class and how to establish the communication between classes while the chapter 3 describes how threads can effectively be used to trigger callbacks with blocking I/O. Chapter 4 then presents the event based communication in Qt, in particular user interaction and animations. Chapter 5 tackles the same issues as for Qt but for website server communication. The final chapter 6 then describes how events are transmitted back from a C++ class to the user with the help of setters.

Chapter 2

Writing C++ device driver classes

This chapter focuses on writing your own C++ device driver class hiding away the complexity (and messy) low level C APIs and/or raw device access. How are events translated into I/O operations? On the hardware-side we have event signals such as data-ready signals or by the timing of a serial or audio 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 until data has arrived or an event has happened. Some low level libraries such as **pigpio** translate them back into C callbacks. The task of a C++ programmer is to hide this complexity and these quite different approaches in C++ classes which communicate via callbacks and setters with the client classes. The rest of the program will then appear simple and be easy to maintain.

2.1 General recommendations on 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. Setters and callbacks hand over *physical units* (temperature, acceleration, ...) or relative units but not raw integer values with 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 default parameters.

3. The class comes with simple demo programs demonstrating how a client program might use it.

2.2 Callbacks in C++ device classes

As said in the introduction your hardware device class has callback interfaces to hand back the data to 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 interface 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 part 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.

2.2.1 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 LSM9DS1::registerCallback(LSM9DS1callback* cb);
```

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

2.2.2 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.

2.2.3 Callback arguments

Above the callbacks just delivered one floating point value. However, often more than one sample or more complex data are transmitted:

- Complex data: do not put loads of arguments into the callback but define a *struct*. For example an ADC might deliver all 4 channels at once:

```
class ADmulti {

        struct ADCSample {
            float ch1;
            float ch2;
            float ch3;
            float ch4;
        };

        ...
        virtual void hasSample(ADCSample sample) = 0;
        ...
};
```

  Depending on your application, you might consider the values not useful individually and therefore prefer a `std::tuple`.

- Arrays: Use arrays which contain the length of the arrays: either std::array, std::vector, etc or const arrays and then references to these so that the callback knows the length. For example the LIDAR callback uses a reference to a const length array:

```
/**
 * Callback interface which needs to be implemented by the user.
 **/
struct DataInterface {
```

```
        virtual void newScanAvail(
                float rpm,
                A1LidarData (&)[A1Lidar::nDistance]) = 0;
};
```

Here `A1Lidar::nDistance` is a constant-expression giving the fixed array length, and `(&)[A1Lidar::nDistance]` a reference to a constant-length array which containing `A1LidarData` stucts. If you're going to use types that hard to key-in often, `typedef` them.

In terms of *memory management*:

1. Low sampling rate complex data structures: allocate as a local variable. It can be a simple type or a struct. See `dataReady()` in: https://github.com/berndporr/LSM9DS1_RaspberryPi_CPP_Library/blob/master/LSM9DS1.cpp.

2. High sampling rate buffers: allocate memory on the heap in the constructor or in the private section of the class as a const length array and pass on a *reference*. See `getData()` in https://github.com/berndporr/rplidar_rpi.

2.3 Low level userspace device access

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

Note that the best I/O is blocking and by itself provides the timing for example when waiting for a GPIO change, doing audio or video I/O.

2.3.1 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 happened 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);
}
```

This code 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 output just call:

```
gpioSetMode(24,PI_OUTPUT);
```

To set the GPIO pin 24 to high just call:

```
int a = gpioWrite(24,1)
```

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. The callback occurs whenever a GPIO pin changes. 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. The callback registered will then be `this->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.

2.3.2 Video camera capture (openCV)

Reading from a video camera is usually done with the help of openCV which is a wrapper around the raw video 4 Linux devices. These are blocking to establish the capture events so that a loop needs to run in a thread:

```
while(running) {
    cv::Mat cap;
    videoCapture.read(cap);
    sceneCallback->nextScene(cap);
}
```

The `read(cap)` command is blocking till a new frame has arrived which is then transmitted with the callback `nextScene` to the client. The full code of the example camera class is here: https://github.com/berndporr/opencv-camera-callback.

2.3.3 I2S: Audio

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

ALSA is packet based with a read command returning a chunk ("buffer") of audio and write emitting one. There are calls to set the sample format, sample rate, buffer size and so forth.

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) {
   if ((err = snd_pcm_readi (handle, buffer, buffer_frames)) != buffer_frames) {
     if (errCallback) errCallback->hasError();
   }
   if (sampleCallback) sampleCallback->hasData(buffer);
 }
```

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

2.3.4 SPI

Table 2.1: 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. Its calls are blocking for the duration of receive and transmit but cannot be used for event based processing. If an SPI device is used events should be transmitted via an additional GPIO line. Even though data might not be used it needs to be matched up, because the same clock is used to send and collect the data signal (it is *isochronous*). So if you send 8 bytes the hardware receives 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 2.1):

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

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

Because SPI is isochronous, `read()` and `write()` can't be used to transmit and receive data. Instead, the simultaneous read and write is performed using an `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[1].

Reading and simultaneous writing then happens via this `ioctrl()` function call:

```
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 indeed 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. On some ADCs the SPI clock is also the conversion clock and a longer lasting clock signal sequence is required, making it necessary to transmit 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.

2.3.5 I2C

I2C is a protocol which either receives or transmits. Its calls are blocking for the duration of receive or transmit but only in rare cases can be used for event based processing. It is recommended to use an additonal GPIO pin for events for example for a data-ready event.

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):

[1] This code fragment's use of designated initialisers officially requires C++2a, although most C++ compilers support it when compiling with older standards. In C it's been fine for a while!

```
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. Subsequent operations read or write that 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.

2.3.6 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.

2.3.7 Accessing physical memory locations (danger!)

Don't. In case you really need to access registers you can also access 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 addresses so that a pointer won't point to a physical memory location. It points to three page tables with an offset.
- Special device `/dev/mem` allows access to 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.

2.4 Kernel driver programming

You can also create your own `/dev/mydevice` in the `/dev` filesystem by writing a kernel driver and a matching userspace library. For example the USB mouse has a driver in kernel space and translates the raw data from the mouse into coordinates. However, this is beyond the scope of this handout. If you want to embark on this adventure then the best approach is to find a kernel driver which does approximately what you want and modify it for your purposes.

2.5 Conclusion

The communication between C++ classes is achieved via callbacks and setters. The event from the sensor traverses the C++ classes via callbacks and then back to the control output via setters.

From the sections above it's clear that Linux user-space 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 and offer the client an easy-to-understand interface.

Chapter 3

Threads

3.1 Introduction

In realtime systems threads have two distinct functions:

1. Endless loops with blocking I/O or GPIO wakeups to establish **precise timing** for callbacks.
2. **Asynchronous execution** of time-consuming tasks with a callback after the task has completed.

3.2 Processes and Threads

Processes are different programs which seem to be running at the same time. A small embedded system may only have a single CPU core, so this this is achieved by the operating system switching approximately every 10ms from one process to the next so it feels as if they are running concurrently. A thread is a lightweight process. A process may have multiple threads which share the same address-space and are all started from within the parent process. As with processes, the threads seem to be running at the same time. When a thread is started it runs simultaneously with the main process which created it.

3.3 Thread and worker

A thread is just a *wrapper* for the actual method which is running independently. The method being run in the thread is often called a *worker*.

3.3.1 Creating threads

In C++ a worker is a method within a class:

```
uthread = new std::thread(&MyClassWithAThread::run, this);
```

where `MyClassWithAThread` is a class containing the function "run":

```
class MyClassWithAThread {
        void run() {
                // ... hard work is done here
                doCallback(result); // hand the result over
        }
}
```

Note that the & signifies a functor, a method prototype. Instead of using functors you can also use a lambda function to call a method in the instance:

```
uthread = std::thread([this](){run();});
```

3.3.2 Lifetime of a thread

Threads terminate simply once the worker has finished its job. To tell the client that a thread has finished you can use a *callback* to trigger an event.

Sometimes it's important to wait for the termination of the thread, for example when your whole program is terminating or when you stop an endless loop in a thread. To wait for the termination of the thread use the "join()" method:

```
        void stop() {
                uthread->join();
                delete uthread;
        }
```

It's also important is also to release the memory of a thread after it has finished to avoid memory leaks, hence the `delete` command.

3.3.3 Running/stopping workers with endless loops

Threads with endless loops are often used in conjunction with blocking I/O which provide the timing:

```
void run() {
       running = true;
       while (running) {
              read(buffer); // blocking
              doCallback(buffer); // hand data to client
       }
}
```

Note the flag `running` which is controlled by the main program and is set to zero to terminate the thread:

```
 void stop() {
        running = false; // <----- HERE!!
        uthread->join();
        delete uthread;
 }
```

Note that `join()` is a blocking operation and needs to be used with care not to lock up the main program. You probably only need it when your program is terminating. See https://github.com/berndporr/rpi_AD7705_daq for an example.

If your program creates and joins several threads while executing, with care you might be able to design your program to allow an end-of-life thread to carry on tidying up in the background after you stop it by setting `running = false;`. In this case only need execute `join()` to be sure that the thread has finished. Joining a terminated thread is OK (the call just returns immediately) but you absolutely must not join a thread more than once, nor delete a thread until you've joined it.

3.3.4 Timing within threads

Threads are perfect to create timing without using sleep commands with the help of *blocking I/O*.

select/poll commands waiting for GPIO interrupts

In section 2.3.1 we introduced the so called "poll" command which is not polling an IRQ pin but *putting a thread to sleep* till an external event has happened. Then of course a callback function should be called reacting to the external event. This is the preferred method for low latency responses.

As said previously, use **pigpio** on the Raspberry PI which wraps the select/poll commands into a thread and calls a *callback* function whenever an GPIO pin has been triggered.

Timing with blocking I/O

Blocking I/O (read, write, etc) *is by far the best approach* to time the data coming in because the thread goes to sleep when it's waiting for I/O but wakes up very quickly after new data has arrived.

In this example the blocking `read` command creates the timing of the callback:

```
void run() {
       running = 1;
       while (running) {
              read(buffer); // blocking
              doCallback(buffer); // hand data to client
       }
}
```

If the I/O is non-blocking or if you are making blocking I/O non-blocking with:

```
int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

you can then use `select()` to wait for data:

```
FD_ZERO(&rdset);
FD_SET(fd,&rdset);
struct timeval timeout;
timeout.tv_sec = 0;
timeout.tv_usec = 500000;
int ret = select(fd+1,&rdset,NULL,NULL,&timeout);
if (ret < 0) return ret;
```

```
// non-blocking I/O read
ret = read(fd,buffer,bytes_per_sample * n_chan);
```

which has the advantage that it has a timeout so that you can create both error and data callbacks.

3.4 Timing with Linux/pigpio timers

As a last resort one can use a timer. Similar to threads one can create timers which are then called at certain intervals. As with threads timers should be *hidden* within a C++ class as *private* members which then trigger *public callbacks* via C++ callback mechanisms as described above. These timers emit a Linux signal at a specified interval and then this signal is caught by a global (static) function. Generally it's *not recommended* to use timers for anything which needs to be reliably sampled, for example ADC converters or sensors with sampling rates higher than a few Hz. On the raspberry PI use the pigpio library and its timer callbacks — if needed at all.

3.5 Conclusion

Threads play a central role in real-time coding as, together with blocking I/O, they establish the callback interfaces. Every event handler runs in a separate thread.

Callbacks are also used to signal the termination of threads, which shows again the close relationship between threads and callbacks.

Chapter 4

Realtime/event processing within the Graphical User Interface Qt

4.1 Introduction

Qt is a cross-platform windows development environment for Linux, Windows and Mac. Such graphical user interfaces have realtime requirements because, for example, pressing a button should trigger an instant response by the application so that the user thinks this has been instantaneous. QT also works with callbacks as introduced above but they are wrapped in a QT-specific signal/slot concept.

Elements in Qt are *Widgets* which can contain anything form plots, buttons or text fields. They are classes. You can define your own widgets or use ready-made ones. Realtime communication within QT happens between widgets where then one widget calls another widget, for example a button the calls the main window if a button press has happened.

The arrangements of widgets on the screen is managed by layouts. There are different ways of declaring layout in Qt. One is using a markup language which then has matching classes; another is creating to use only C++ classes. We show how to organise the layout using the second method. This avoids having the learn an additional language and is consistent with the general trend to use code to declare the layout in this and other frameworks (**SwiftUI**, for example).

This is an example how widgets are organised into nested vertical and horizontal layouts (see Fig. 4.1 for the result).

```
// create 3 widgets
button = new QPushButton;
```

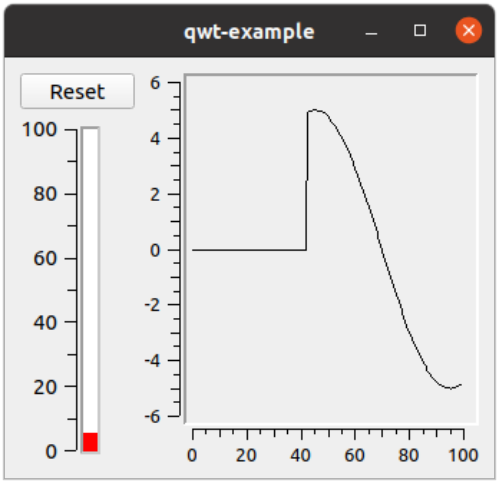

Figure 4.1: QT example layout

```
thermo = new QwtThermo;
plot = new QwtPlot;

// vertical layout
vLayout = new QVBoxLayout;
vLayout->addWidget(button);
vLayout->addWidget(thermo);

// horizontal layout
hLayout = new QHBoxLayout;
hLayout->addLayout(vLayout);
hLayout->addWidget(plot);

// main layout
setLayout(hLayout);
```

4.2 Callbacks in Qt

4.2.1 Events from widgets

In contrast to our low level callback mechanism using interfaces, Qt rather directly calls methods in classes. The problem is that function pointers cannot be directly used as a class has instance pointers to its local data. So a method of a class needs to be combined with the instance pointer. The Qt method "connect" does exactly that:

```
connect(button, &QPushButton::clicked,
        this, &Window::reset);
```

The QPushButton instance `button` has a method called `clicked()` which is called whenever the user clicks on the button. This is then forwarded to the method `reset()` in the application Widget.

4.2.2 Plotting realtime data arriving via a callback

The general idea is to store the real-time samples from a callback in a buffer and trigger a screen refresh at a lower rate. For example, we may choose to replot the samples in the buffer every 40 ms because that fast enough for the user, whereas plotting the whole buffer every sample would be too CPU-intensive.

A callback `addSample()` is called in real-time whenever a sample has arrived:

```
void Window::addSample( float v ) {
    // add the new input to the plot
    std::move( yData, yData + plotDataSize - 1, yData+1 );
    yData[0] = v;
}
```

which stores the sample `v` in the shift buffer `yData`.

Then the screen refresh (which is slow) is done at a lower and unreliable rate:

```
void Window::timerEvent( QTimerEvent * )
{
    curve->setSamples(xData, yData, plotDataSize);
    plot->replot();
    thermo->setValue( yData[0] );
    update();
}
```

`update()` in the timer event-handler generates a paint event and Qt then invokes the repaint handler "as soon as possible" (which is to say, not in real-time) to repaint the canvas of the widget:

```
void ScopeWindow::paintEvent(QPaintEvent *) {
        QPainter paint( this );

        paint.drawLine( ... )
}
```

Note that neither the timer nor the `update()` function is called in a reliable way but whenever Qt chooses to do it. So Qt timers cannot be used to sample data but should only be used for screen refresh and other non-time-critical reasons.

4.3 Conclusion

Events in Qt are generated by user interaction, for example a button press or moving the mouse. As before Qt provides a callback mechanism via the `connect()` method. Callbacks from Qt timers may be used for animations but must not be used for real-time events as Qt timers won't guarantee a reliable timing.

Chapter 5

Realtime web server/client communication

5.1 Introduction

There is a wide diversity of Web server / client applications ranging from shopping baskets on vendor sites to social media.

Generally it's easy to create dynamic content (see PHP or nodejs) and this is well documented. However, feeding realtime data from C++ to a web page or realtime button presses back to C++ is a bit more difficult.

It's important to recognise where *events* are generated: it is always the client (web browser or mobile app) which triggers an event, be it sending data over to the server or requesting data. It's always initiated by the client.

5.2 REST

The interface between a web client (browser or phone app) is usually implemented as a Representational State Transfer Architectural (REST) API by communicating via an URL on a web server. The requirements for this API are very general and won't define the actual data format:

Uniform interface. Any device connecting to the URL should get the same reply. No matter if a web page or mobile phone requests the temperature of a sensor the returned format must always be the same.

Client-server decoupling. The only information the client needs to know is the URL of the server to request data or send data.

Statelessness. Each request needs to include all the information necessary and must not depend on previous requests. For example a request to a buffer must not alter the buffer but just read from it so that another user reading the buffer shortly after receives the same data.

See https://www.ibm.com/cloud/learn/rest-apis for the complete list of REST design principles.

5.3 Data format: JSON

The most popular dataformat is JSON (`application/json`) which is basically a map of (nestable) key/value pairs:

```
{
    temperature: [20, 21, 20, 19, 17],
    steps: 100,
    comment: "all good!"
}
```

Since JSON is human-readable text a web server can simply generate that text send it over via http or https. There is no difference except that the MIME format is 'application/json' instead of html.

5.4 Server

On the Linux system a web server needs to be set up. There are a variety of different options available but we are focusing here on the ones which can be used for C++ communication (i.e. CGI).

5.4.1 Web servers (http/https)

- NGINX: Easy to configure but very flexible web server. Pronounced "Engine-X".
- Apache: Hard to configure but safe option

- lighttpd: Smaller web server with a small memory footprint. Pronounced "lighty".

Note that it's possible to run different web servers at the same time where they then act as proxies for a central web server visible to the outside world. In particular nginx makes it very easy to achieve this.

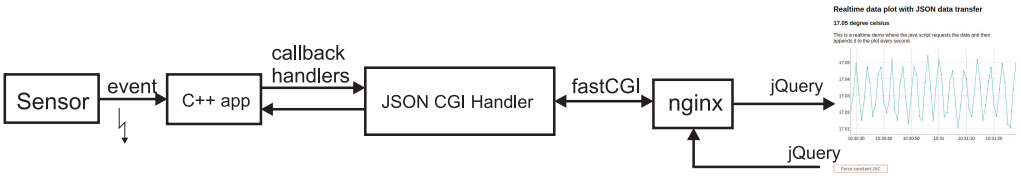

Figure 5.1: FastCGI dataflow.

5.4.2 FastCGI

FastCGI (see Fig 5.1) is written in C++ and generates the entire content of the http/https request. In particular here we generate JSON packets which can then be processed by client JavaScripts and vice versa. For realtime applications JSON transmission is perfect because the client-side JavaScript can request and receive JSON packages

A fast CGI program is a UNIX commandline program which communicates with the web server (nginx, Apache, ...) via a UNIX socket which in turn is a pseudo file located in a temporary directory for example `/tmp/sensorsocket`.

The web server then maps certain http/https requests to this socket. An example configuration for nginx looks like this:

```
location /sensor/ {
   include        fastcgi_params;
   fastcgi_pass   unix:/tmp/sensorsocket;
 }
```

If the user does a request via the URL `www.mywebpage.com/sensor/` then nginx contacts the fastcgi program via this socket. The fastcgi program then needs to return the content. Internally this will be a C++ callback inside of the fastcgi program.

The C++ fastcgi API https://github.com/berndporr/fastcgi_json_cpp_api is wrapper around the quite cryptic fastcgi C library and we discuss its callback handlers now.

5.4.3 Server → client

The fastCGI callback expects a JSON string (`application/json`) with the data transmitted form the server to the client. Use the `jsoncpp` library (standard debian/Ubuntu package) to create JSON.

```
class JSONcallback : public JSONCGIHandler::GETCallback {
public:
/**
* Gets the data and sends it to the webserver.
* The callback creates two JSON entries. One with the
* timestamp and an array of sensor readings.
**/
virtual std::string getJSONString() {
  Json::Value root;
  root["epoch"] = (long)time(NULL);
  Json::Value values;
  for(int i = 0; i < datasink->values.size(); i++) {
    values[i] = datasink->values[i];
  }
  root["values"]  = values;
  Json::StreamWriterBuilder builder;
  const std::string json_file = Json::writeString(builder, root);
  return json_file;
};
```

5.4.4 Client → server: POST

Like in any GUI the client can press a button and create an event. Here, the server then receives the JSON data as a callback called "postArg":

```
virtual void postString(std::string postArg) {
  const auto rawJsonLength = static_cast<int>(postArg.length());
  JSONCPP_STRING err;
```

```
  Json::Value root;
  Json::CharReaderBuilder builder;
  const std::unique_ptr<Json::CharReader> reader(builder.newCharReader());
  reader->parse(postArg.c_str(), postArg.c_str() + rawJsonLength, &root, &err)
  // do something with root
}
```

where `root` is a `std::map`.

5.5 Client code: javascript for websites

Generally on the client side (= web page), HTML with embedded *JavaScript* is used to generate realtime output/input without reloading the page. JavaScript is *event driven* and has callbacks so it's perfect for realtime applications. Use `jQuery` to request and post JSON from/to the server.

For example here we request data from the server as a JSON packet every second:

```
// callback when the JSON data has arrived
function getterCallback(result) {
  var temperatureArray = result.temperatures;
  // plot the array here
}

// timer callback (same idea as in Qt to define a refresh rate)
function getTemperature() {
  // get the JSON data
  $.getJSON("/data/:80",getterCallback);
}

// document ready callback
function documentReady() {
  // request new data from the server every second
  window.intervalId = setInterval(getTemperature , 1000);
}

// called when the web page has been loaded
$(document).ready( documentReady );
```

Mobile phone programming in JAVA, Kotlin or Swift is also purely callback driven as the JS code above and differs only in its syntax.

5.6 Conclusion

Events in web based communication are always triggered by the web browser or the mobile app in exactly the same way as Qt does it, for example by a button press. The same applies for animations where a client-side timer requests data from the server. Thus, these client side events either cause transmission of data from the web browser to the web server or request data from the web server. Nowadays the protocol is always http or https and a RESTful interface with JSON being the most popular data format.

Chapter 6

Setters

In Fig. 1.2 we have seen that data flows from the sensors to the C++ classes via *callbacks* then returns from the inner C++ classes to motor or display outputs is via *setters*. Setters are also used for setting configuration parameters.

A setter is a simple method in a class, for example to set the speed of a motor:

```
class Motor {
  /**
  * Set the Left Wheel Speed
  * @param speed between -1 and +1
  **/
  void setLeftWheelSpeed(float speed);
};
```

Again as with callbacks it's important to *abstract* away from the hardware, for example normalising the speed of the motor between -1 and $+1$ and *hiding* away the complexity of the PWM or GPIO ports in the class.

If a setter has more than one argument, in particular for configuration, it's highly recommended to use a *struct* to set the values. For example setting the parameters of the ADS1115:

```
/**
 * ADS1115 initial settings when starting the device.
 **/
struct ADS1115settings {
```

```
/**
 * I2C bus used (99% always set to one)
 **/
int i2c_bus = 1;

/**
 * I2C address of the ads1115
 **/
uint8_t address = DEFAULT_ADS1115_ADDRESS;
};

/**
 * Starts the data acquisition in the background and the
 * callback is called with new samples.
 * \param settings A struct with the settings.
 **/
void start(ADS1115settings settings = ADS1115settings() );
```

If a setter sets large buffers then it's highly recommended to allocate the memory in the constructor of the class and then call the setter by reference while running. Use array types which convey their length, for example std::array or a standard const array which implicitly carries their length.

Constant sampling rate output (audio, . . .) There are many applications where the output device has a fixed sampling rate, for example digital to analogue converters. In this case the C++ driver class will again have a blocking write-loop periodically reading a buffer populated by the setter, which is ideally always ahead of time. You need to decide what happens if no fresh data has arrived, for example interpolating the output or putting it on hold. Of course you can also implement a callback by the audio write-loop to *request* samples but ultimately the conflict between audio arriving and being dispatched needs to be resolved.

6.1 Conclusion

Setters are simply methods which transmit an event back to the physical device. Setters should, as callbacks, always hide the low level complexity of the hardware device and receive normalised or physical units.

Appendix A

License