# Realtime embedded coding in C++ under Linux

Bernd Porr & Nick Bailey

July 22, 2022

# Linux下C++的实时嵌入式编码

伯恩德*波尔和尼克*贝利

July 22, 2022

# Contents





# Contents

# Chapter 1

# Introduction

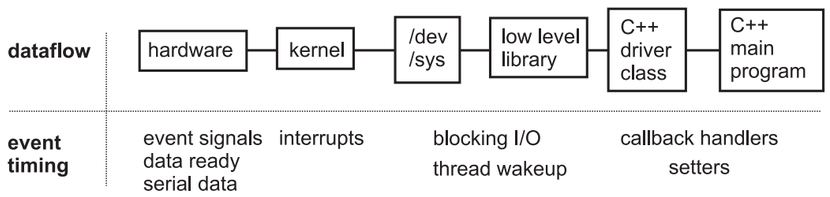


Figure 1.1: Dataflow and timing in low level realtime coding

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.

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



# Chapter 1

# Introduction

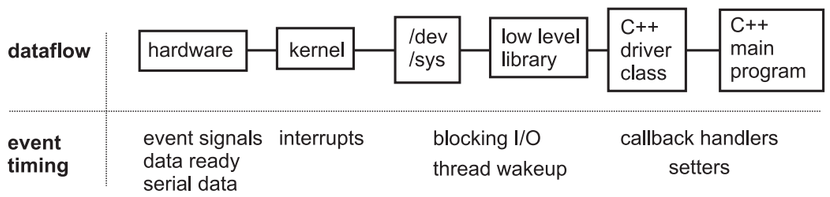


图1.1：低层实时编码中的数据流和时序

实时嵌入式编码是所有关于事件。这些信号可以是二进制信号，如有人开门或ADC信号，表明样品已准备好被提取。无花果。1.1显示了基本的数据流以及如何建立事件定时：设备本身具有事件信号，如"数据就绪"或"崩溃传感器已触发"。Linux内核接收这些作为中断回调。然而，用户空间没有直接的中断机制；相反，它有阻塞IO，其中读或写操作阻塞，直到内核端中断发生。然后，通过唤醒线程，返回的阻塞IO调用可能会在类之间转换回调。数据通过称为"setter"的方法传输回硬件，这些方法更改对象的属性并可能进行其他处理。

无花果。1.2显示了实时系统中C++类之间的整体通信。
这种通信是通过回调（而不是getter）和setter完成的，其中来自传感器的事件通过回调通过C++类根据其实时要求遍历，然后通过

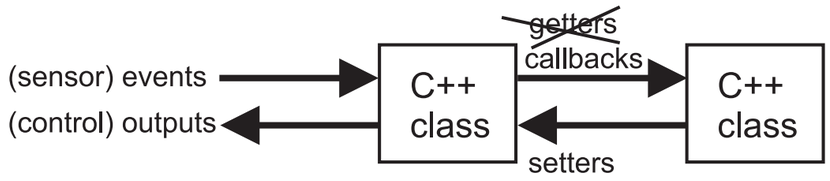


Figure 1.2: A realtime system with two C++ classes. Communication between classes is achieved with callbacks (not getters) for incoming events and setters to send out control events. The control output itself receives its timing from the events so that the loop is traversed as quickly as possible.

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.

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



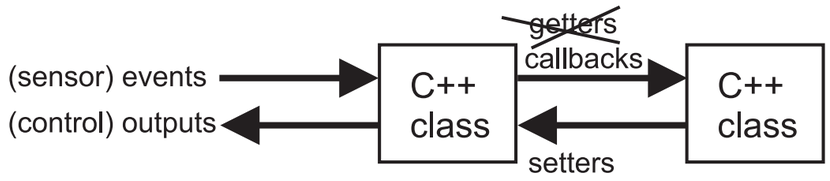


图1.2：一个包含两个C++类的实时系统。类之间的通信是通过传入事件的回调（而不是getter）和发送控制事件的setter来实现的。控制输出本身接收来自事件的定时，以便尽可能快地遍历循环。

setters。例如，机器人上的碰撞传感器触发GPIO引脚，然后触发回调以发出避让动作，然后将电机反向设置。

在开发C++类时，请记住S.O.L.I.D.:

单一责任：如果你有一个温度传感器和一个加速度计，那么写两个类：一个用于温度传感器，一个用于加速度计。不要编写一个类硬件，它需要一个参数硬件::temp或硬件::accel，然后做两个完全不同的事情之一。如果您想在硬件中仅使用温度传感器或加速度计重用您的类，这是一个调试噩梦和浪费。

开闭原则：你的类对扩展是开放的，但对修改是封闭的。例如，ADC类有一个回调函数，它将电压返回给客户端。但是，例如，您将连接一个温度传感器到它，因此您希望能够扩展重载回调方法的类，以便添加从volt到degrees的转换。这是一件好事-只要你创建一个派生类。破解现有的ADC类为kelvin方法添加a是一个坏主意：为什么有人使用ADC从加速度计读取值需要这个？

Liskov替换原则：严格地说，将派生类替换为其基类不会导致程序变得不正确。也就是说，如果基类是所有必需的，则可以使用设备驱动程序类中的任何派生类来代替基类，因为派生类中的额外功能不应破坏

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.

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

1. 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.

2. 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.

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

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



基类。例如，如果您有一个具有许多额外功能的super-duperDAC，那么当您只需要一个非常简单的DAC时，它不应该阻止您使用它。这也意味着应该设置合理的默认值，这样客户端就不需要理解那个超级骗子DAC的书呆子特性。

接口隔离原则：保持功能分离，并将其划分为不同的类。想象一下，你有一个通用的IO类与SPI和I2C，但你的客户端真的只需要SPI。然后，客户端被迫停用I2C，或者在最坏的情况下，该类会在客户端不知道原因的情况下造成附带损害。

依赖倒置：即抽象出一类接口的基本功能。例如，理想情况下，您希望基类复盖来自同一制造商的一系列类似ADC转换器，而不是基类作为特定芯片的驱动程序。然后，各个ADC芯片驱动程序类从抽象ADC驱动程序继承
。
的作者
ADC的驱动程序无疑会将其芯片的特定功能视为核心思想，但这需要反相。为什么一个ADC驱动器需要为同一系列中的另一个提供所有必需的代码？相反，驱动程序依赖于ADC的抽象概念，而不是相反。具体取决于抽象。

除了S.O.L.I.D，这也是必不可少的:

1.该项目有一个构建系统，如cmake。强烈建议使用cmake（autoconf仅适用于较旧的现有项目）。对于Qt项目尤其如此，其中cmake支持是Qt开发人员的优先事项。

2.类（特别是驱动程序类）可以在特定项目之外重复使用，并且在其子目录中有自己的cmake-projects。除了主cmake项目之外，所有子项目都是库。这有助于测试和重复使用。

3.所有公共接口都有所有公共方法常量的文档字符串和自动生成的引用，例如在doxygen的帮助下。

4.执行内部处理的类，例如过滤器，数据库，检测器，。..，有单元测试，并通过cmake测试框架运行。

5. 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.



5.该文档提供了有关项目本身、如何安装和运行项目的全面信息。

在以下部分中，我们将描述如何用C++编写事件驱动代码。
这里重要的是阻塞IO和线程之间的相互作用：第2章着重于如何将低级阻塞LinuxIO包装在一个类中，以及如何建立类之间的通信，而第3章描述了如何有效地使用线程来触发阻塞IO的回调。第4章接着介绍了Qt中基于事件的通信，特别是用户交互和动画。第5章处理了与Qt相同的问题，但对于网站服务器通信。最后的第6章描述了如何在setter的帮助下将事件从C++类传回给用户。

# 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.



# Chapter 2

# 编写C++设备驱动程序类

本章的重点是编写您自己的c++设备驱动程序类，隐藏复杂性（和混乱）低级CApi或原始设备访问。如何将事件转换为IO操作？在硬件方面，我们有事件信号，如数据就绪信号或串行或音频接口的定时。Linux内核将此定时信息转换为在伪文件系统（如dev或sys）上阻塞IO，这意味着读取操作会阻塞，直到数据到达或事件发生为止。一些低级库（如pigpio）将它们翻译回C回调。C++程序员的任务是在C++类中隐藏这种复杂性和这些完全不同的方法，这些方法通过回调和setter与客户端类进行通信。程序的其余部分将显得简单，易于维护。

## 2.1 关于如何为设备编写C++类的一般建议

如上所述，面向对象编码的主要目的是隐藏低级驱动程序访问的复杂性，并为客户端提供连接到传感器的简单而安全的方式。特别是：

1.设置器和回调器移交物理单位（温度，加速度，。..）或相对单位，但不是没有任何意义的原始整数值。

2.传感器是通过指定物理单位（时间、电压、温度）而不是传感器寄存器来配置的。默认配置参数应指定该类可以直接与默认参数一起使用。

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

## 2.2 Low level userspace device access

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

### 2.2.1 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. 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:



3.该类带有简单的演示程序，演示客户端程序如何使用它。

## 2.2 低级用户空间设备访问

以下各节提供了如何为不同的硬件协议编写C++驱动程序类的指针。

### 2.2.1 SPI

表2.1：SPI模式

| SPI模式 | CPOL | CPHA | 空闲状态 |
|---|---|---|---|
| 0 | 0 | 0 | L |
| 1 | 0 | 1 | L |
| 2 | 1 | 1 | H |
| 3 | 1 | 0 | H |

SPI是一种通常在同一时间发送和接收的协议.即使数据可能不被使用，它也需要进行匹配，因为发送和收集数据信号使用相同的时钟（它是同步的）。因此，如果您发送8个字节，硬件会同时接收8个字节。

从SPI传输最好由对dev的低级访问来管理。使用标准open()函数打开SPI设备:

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

然后设置SPI模式（见表2.1）：

intret=ioctl（fd，SPI_IOC_WR_MODE，＆mode）；

正如所解释的，例如，在这里：https：www.analog.comen模拟对话文章介绍到spi接口.html。

因为SPI是同步的，所以read()和write()不能用于发送和接收数据。相反，同时读取和写入是使用ioctl()执行通信。填充此结构:

```
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.

### 2.2.2 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):

[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!



```
struct spi_ioc_transfer tr {
.tx_buf=（无符号长）tx1，。rx_buf
=（无符号长）rx1，。len=ARRAY_
SIZE（tx1），。delay_usecs=延迟
，。speed_hz=速度，。bits_per_w
ord=8，；
```

其指向长度相同的两个字符缓冲区"tx"和"rx"1。
读取和同时写入然后通过此ioctrl()函数调用发生:

intret=ioctl（fd，SPI_IOC_MESSAGE（1），&tr）；

有时芯片的SPI协议非常奇怪，即使是通过dev的原始IO也不起作用，您需要编写自己的位敲打接口，例如在这里为alphabot上的ADC完成：https：github.comberndporralphabotblob主alphabot.cpp#L58.这显然远非理想，因为它可能需要usleep()命令，以便获取需要在单独的线程中运行（alphabot确实在单独的线程中使用计时器回调）。总体而言，SPI协议通常依赖于设备，需要实验才能使其工作。在某些Adc上，SPI时钟也是转换时钟，需要更长时间的时钟信号序列，因此除了有效负载外，还需要传输虚拟字节。

作为一般建议，不要使用使用SPI数据时钟也作为采集时钟的SAR转换器，因为它们通常与通过dev的标准SPI传输不兼容。使用具有自己时钟信号的传感器或Adc。

### 2.2.2 I2C

I2C总线有两条信号线(SDA和SDL)，必须通过电阻上拉。每个I2C器件在总线上都有一个地址。您可以使用i2cdetect（i2c-tools软件包的一部分）扫描总线:

1此代码段使用指定的初始化器正式要求C++2a，尽管大多数C++编译器在使用较旧的标准编译时都支持它。在C已经好了一段时间！

```
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
```



```
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#
```

在这种情况下，地址1e、58和6b的I2C总线上有3个I2C设备，需要在访问I2C设备时指定。

### 原始开发i2c访问

I2C发送或接收，但从不在同一时间，所以在这里我们可以使用标准的c读写命令。但是，我们需要使用ioctrl来告诉内核I2C地址:

charbuf[2];intfile=open("devi2c−2" O_RD
WR);intaddr=0x58;ioctl(file I2C_SLAVE add
r);write(file buf 1)read(file buf 2)

其中addr是I2C地址。然后使用标准的read()或write()命令。通常第1次写操作告诉芯片读或写哪个寄存器。后续操作读取或写入该寄存器。

### 通过pigpio访问I2C

通过pigpio访问（http：abyz.me.ukrpipigpiocif.与直接访问原始开发i2c相比，html）是首选，因为许多不同的设备可以连接到I2C总线，而pigpio管理这一点。只需安装开发包:

```
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.

### 2.2.3 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 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:



然后触发其他相关软件包的安装。例如，将一个字节写入I2C传感器中的寄存器可以通过一些命令完成:

intfd=i2cOpen（i2c_bus，地址，0）;i2cWriteByt
eData（fd，子地址，数据）;i2cClose（fd）;

其中i2c总线是I2C总线号（RPI上通常为1），地址是该总线上器件的I2C地址。这里的子地址是设备中的寄存器地址。

### 2.2.3 访问GPIO引脚

**`/sys` filesystem**

RaspberryPI的GPIO可以通过sys文件系统轻松控制。这很慢，但很适合调试，因为您可以直接向其写入"0"或"1"字符串并打印结果。伪文件在这里:

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

其中包含直接与单个引脚相关的文件。为了能够访问pin，我们需要告诉Linux使其可见:

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

例如，将5（以文本形式）写入此文件将为GPIO引脚5创建子目录sys类gpiogpio5。然后从

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

会给你GPIO引脚5的状态，写入它会改变它。一个关于GPIOsys文件系统的薄包装器在这里：https：github.comberndporrgpio-sysfs.

GPIO通过sys中断处理。Sys文件系统最重要的应用是在用户空间中进行中断处理。线程可以进入休眠状态，直到其中一个GPIO引脚发生中断。这是通过使用"poll"命令监视sys文件系统中GPIO引脚的"值"来完成的:

```
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 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. 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)
  ...
}
```



structpollfdfdset[1];intnfds=1;intgpio_fd=open("sys类gpiogpio5值" O_RDONLYO_NONBLOCK);memset((void*)fdset 0 sizeof(fdset));fdset[0]。fd=gpio_fd;fdset[0]。events=POLLPRI;intrc=poll(fdset nfds timeout);if(fdset[0].revents&POLLPRI){dummyread(fdset[0].fd，buf，MAX_BUF);

}

该代码使线程进入睡眠状态，直到GPIO引脚5发生中断。然后线程唤醒并继续执行。

**pigpio**

上面的部分已经让您深入了解了sysfs级别的引擎盖下发生了什么，但强烈建议使用pigpio库（http：abyz.me.ukrpipigpiocif.html)读写GPIO引脚或做中断编程.

例如设置GPIO引脚24作为输入只是调用:

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

要从GPIO引脚24读取，只需调用:

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

**通过pigpio中断处理。pigpio通过将上述所有功能包装到一个客户端注册回调函数的命令中来管理GPIO中断处理。每当GPIO引脚发生变化时，就会发生回调。具体为所述形式的方法:**

类mySensorClass{…静态voidgpioISR（intgpio，intlevel，uint32_ttick，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.

### 2.2.4 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.

### 2.2.5 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);
```



已在pigpio注册:

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

其中"this"是指向类实例的指针。然后注册的回调将是this->datarready()。

```
class LSM9DS1 {
  voiddataready();staticvoidgpioISR(intgpio intlevel uint32_ttick void*userdata){


        ((LSM9DS1*)userdata)->dataReady();
    }
};
```

在静态函数中，void指针被转换回实例指针。请参阅https：github.comber ndporrLSM9DS1_RaspberryPi_CPP_Library的完整代码。

### 2.2.4 通过sys中的特殊设备访问硬件

一些传感器可以通过sys文件系统以人类可读的格式直接获得。例如

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

给你CPU的温度。

### 2.2.5 I2S: Audio

音频的标准框架是alsa：https：github.comalsa-项目.
ALSA是基于数据包的读命令返回一个音频块（"缓冲区"）和写发射一个。有调用设置样本格式，采样率，缓冲区大小等。

首先，请求参数，驱动程序可以修改或拒绝它们:

```
*有符号的16位小端格式*
  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.

### 2.2.6 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.



```
*单声道(mono)*snd_pcm_hw_params_set_channels(handle par
ams 1);

*44100比特秒采样率(CD质量)*val=44100;snd_pcm_hw_p
arams_set_rate_near(handle params &val &dir);
```

然后播放声音是在一个无休止的循环中完成的，是一个读（）或写（）命令发出。两者都是阻塞的，因此它需要在线程中运行:

```
while (running) {
  if((err=snd_pcm_readi(handle buffer buffer_frames))!=buffer_frames){if(errCallback)errCallback->h
  asError();
  }
  if (sampleCallback) sampleCallback->hasData(buffer);
}
```

对于一个完整的编码示例，"aplay"和"arecord"是一个良好的开端。两者都可以在这里找到：https：github.comalsa-项目.

### 2.2.6 摄像机捕获(openCV)

从摄像机读取通常是在openCV的帮助下完成的，openCV是rawvideo4Linux设备的包装器。这些是阻塞建立捕获事件，以便循环需要在线程中运行:

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

读取（cap）命令被阻塞，直到一个新的帧到达，然后用回调nextScene传输到客户端。示例相机类的完整代码在这里：https：github.comberndporropencv-camera-callback。

### 2.2.7 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.

## 2.3 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.



### 2.2.7 访问物理内存位置（危险！）

不要。如果你真的需要访问寄存器，你也可以直接访问内存。这只能作为最后的手段。例如，为AD转换器设置时钟需要将GPIO引脚转换为时钟输出引脚。驱动程序还不支持这一点，因此我们需要在RPI上编程寄存器。

Linux使用虚拟地址，以便指针不会指向物理内存位置。它指向三个带有偏移量的页表。

特殊设备devmem允许访问物理内存。

命令mmap通过打开devmem提供指向物理地址的指针。

- Example:

int*addr;if((fd=open("dev" O_RDWRO_SYNC))<0){printf("
错误打开文件。关闭(fd);返回(-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);
```

小心使用！如果使用不当，这是危险的。

## 2.3 内核驱动程序编程

您还可以通过编写内核驱动程序和匹配的用户空间库在dev文件系统中创建自己的devmydevice。例如，USB鼠标在内核空间中有一个驱动程序，并将鼠标的原始数据转换为坐标。但是，这超出了本讲义的范围。如果你想开始这次冒险，那么最好的方法是找到一个内核驱动程序，它大约做你想要的，并修改它为你的目的。

## 2.4 Callbacks in C++ device classes (getting data)

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.

### 2.4.1 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;
};
```



## 2.4 C++设备类中的回调（获取数据）

正如介绍中所说，您的硬件设备类具有回调接口以将数据交还给客户端。

有不同的方法来解决回调问题，但最简单的方法是将方法定义为抽象，并要求客户端在派生类中实现它。该抽象函数既可以位于单独的接口类中，也可以位于设备类本身的一部分中。所以，我们有两个选择:

1.回调是设备驱动程序类的一部分:

```
class MyDriver {
          voidstart(DevSettingssettings=DevSettings());voidstop();vi
          rtualvoidcallback(floatsample)=0;

};
```

2.回调是接口类的一部分:

```
class CallbackInterface {
          virtualvoidcallback(floatsample)=0;
};
```

然后在主设备驱动程序类中注册它:

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

现在更详细地解释这两个选项。

### 2.4.1 创建回调接口

在这里，我们创建一个单独的接口类，其中包含一个回调作为抽象方法:

```
class LSM9DS1callback {
public:
        virtualvoidhasSample(LSM9DS1Samplesample)=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 may or may not need one. See https://github.com/berndporr/rpi_AD7705_daq for a complete example.

### 2.4.2 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.



然后，客户端实现抽象方法hasSample()，实例化接口类，然后将其指针保存在设备类中，这里称为lsm9ds1Callback。

```
void LSM9DS1::dataReady() {
        Lsm9ds1samplesample；用数据填充samp
        lestruct。.lsm9ds1Callback->hasSample
        （示例）；

}
```

指向接口实例的指针通过setter传输，setter接收接口的指针作为参数，例如:

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

这允许可选地注册回调。客户端可能需要也可能不需要。请参阅https：github.comberndporrrpi_AD7705_daq为一个完整的例子。

### 2.4.2 直接向设备驱动程序类添加抽象方法

您也可以将回调直接添加到设备驱动程序类，而不是创建包含回调的单独类。

```
类ADS1115rpi{。..virtualvoidhasSample(floatsample)=0;
。..


};
```

这迫使客户端实现回调以能够使用该类。这创建了一个非常安全的环境，因为所有依赖项都是在编译时设置的，基类的抽象性质明确了需要实现的内容。请参阅https：github.comberndporrrpi_ads1115为一个完整的例子。

### 2.4.3 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;
};
```



### 2.4.3 回调参数

上面的回调只是传递了一个浮点值。然而，通常不止一个样本或更复杂的数据被传输:

复杂数据：不要将大量参数放入回调，而是定义一个结构体。例如，ADC可能同时提供所有4个通道:

```
class ADmulti {

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

        ...virtualvoidhasSample(ADCSamplesample)=0;。..
};
```

根据您的应用程序，您可能认为这些值不是单独有用的，因此更喜欢std::tuple。

Arrays：使用包含数组长度的数组：std::array，std::vector等或const数组，然后引用这些数组，以便回调知道长度。例如，LIDAR回调使用对const长度数组的引用:

```
/**
*需要用户实现的回调接口。**结构数据面{

        virtualvoidnewScanAvail(floatrpm 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.

## 2.5 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.



这里A1Lidar::nDistance是一个常量表达式，给出固定的数组长度，(&)[a1lidar::nDistance]是对包含a1lidardatastucts的常量长度数组的引用。如果您要使用难以经常输入的类型，请键入它们。

在内存管理方面：

1.低采样率复杂数据结构：分配为局部变量。它可以是简单类型或结构。请参阅datarready（）在：https：github。comberndporrLSM9DS1_RaspberryPi_CPP_Libraryblob主LSM9DS1.cpp。

2.高采样率缓冲区：在构造函数或类的私有部分中的堆上分配内存作为const长度数组并传递引用。请参阅https中的getData（）：github.com berndporrrplidar_rpi.

## 2.5 Conclusion

C++类之间的通信是通过回调和setter来实现的。来自传感器的事件通过回调遍历C++类，然后通过setter返回到控制输出。

从上面的部分可以清楚地看出，Linux用户空间的低级设备访问是复杂的，即使没有考虑到当代芯片的复杂性，这些芯片通常具有大量的寄存器和文档页面。你的任务是在C++类中隐藏所有这些（可怕的）复杂性，并为客户端提供一个易于理解的界面。

# 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.



# Chapter 3

# Threads

## 3.1 Introduction

在实时系统中，线程有两个不同的功能:

1.具有阻塞IO或GPIO唤醒的无限循环，为回调建立精确的时间。

2.在任务完成后用回调异步执行耗时的任务。

## 3.2 进程和线程

进程是不同的程序，似乎在同一时间运行。一个小型的嵌入式系统可能只有一个CPU核心，所以这是通过操作系统大约每10ms从一个进程切换到下一个进程来实现的，所以感觉好像它们同时运行。线程是一个轻量级进程。一个进程可能有多个线程，它们共享相同的地址空间，并且都是从父进程中启动的。与进程一样，线程似乎在同一时间运行。当一个线程被启动时，它与创建它的主进程同时运行。

## 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 and needs to be *static* which means it won't be able to access the instance variables of a class. The trick is to pass a pointer to the instance of the class (`this`) as the argument of the worker, in the following example called `exec`:

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

where `MyClassWithAThread` is a class containing the static function "exec":

```
class MyClassWithAThread {
        void run() {
                // ... hard work is done here
                doCallback(result); // hand the result over
        }
        static void exec(MyClassWithAThread* cppThread) {
                cppThread->run();
        }
}
```

which in turn then calls a non-static class method `run()`. `run` will then have access to all of the instance's attributes and methods.

### 3.3.2 Lifetime of a thread

Threads terminate simply once the static 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;
}
```



## 3.3 线程和worker

线程只是独立运行的实际方法的包装器。
在线程中运行的方法通常称为worker。

### 3.3.1 创建线程

在C++中，worker是类中的一个方法，需要是静态的，这意味着它不能访问类的实例变量。诀窍是传递一个指向类实例的指针（this）作为worker的参数，在下面的exec中:

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

其中MyClassWithAThread是一个包含静态函数"exec"的类:

```
类MyClassWithAThread{voidrun(
){
                        ...努力工作在这里完成doCallback(result);交出结果
         staticvoidexec(MyClassWithAThread*cppThread){cppThr
        ead->run();

        }
}
```

然后调用一个非静态类方法run()。然后，run将有权访问实例的所有属性和方法。

### 3.3.2 线程的生命周期

一旦静态工作线程完成其工作，线程就会终止。要告诉客户端线程已完成，您可以使用回调来触发事件。

有时等待线程的终止是很重要的，例如当你的整个程序正在终止或当你停止一个线程中的死循环时。要等待线程的终止，请使用"join()"方法:

```
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*.



同样重要的是在完成后释放线程的内存以避免内存泄漏，因此使用delete命令。

### 3.3.3 无休止的循环运行停止工人

具有无限循环的线程通常与提供定时的阻塞IO一起使用:

```
void run() {
        running=true;while(r
        unning){
                read(buffer);blockingdoCallback(buffer);handsdatato
                client
        }
}
```

请注意由主程序控制并设置为零以终止线程的标志运行:

```
        void stop() {
                running=false;<----这里！!uthread->
                join();删除uthread;

        }
```

请注意，join()是一个阻塞操作，需要小心使用，不要锁定主程序。您可能只需要在程序终止时使用它。请参阅https：github.comberndporrrpi_AD7705_daq为例。

如果您的程序在执行时创建并加入多个线程，那么您可以小心地设计您的程序，以便在通过设置running=false;停止后，允许一个生命终结线程在后台进行整理。在这种情况下只需要执行join()以确保线程已经完成。加入一个终止的线程是可以的（调用只是立即返回），但是你绝对不能加入一个线程超过一次，也不能删除一个线程，直到你加入它。

### 3.3.4 线程内的定时

线程非常适合在阻塞IO的帮助下创建定时而不使用睡眠命令。

### select/poll commands waiting for GPIO interrupts

In section 2.2.3 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
        }
}
```

## 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.



### 选择等待GPIO中断的轮询命令

在第2.2.3节中，我们介绍了所谓的"poll"命令，它不是轮询IRQ引脚，而是将线程置于睡眠状态，直到外部事件发生。当然，应该调用一个回调函数来响应外部事件。

这是低延迟响应的首选方法。
如前所述，在RaspberryPI上使用pigpio，它将select轮询命令包装到一个线程中，并在触发GPIO引脚时调用回调函数。

### 与阻塞iO定时

阻塞IO（读取、写入等）是迄今为止计算数据传入时间的最佳方法，因为线程在等待IO时进入睡眠状态，但在新数据到达后很快醒来。

在此示例中，阻塞读取命令创建回调的时间:

```
void run() {
        运行=1；同时（运行
        ）{
                read(buffer);blockingdoCallback(buffer);handsdatato
                client
        }
}
```

## 3.4 使用Linuxpigpio定时器计时

作为最后的手段，可以使用计时器。与线程类似，可以创建定时器，然后以一定的时间间隔调用。与线程一样，计时器应该作为私有成员隐藏在C++类中，然后通过如上所述的c++回调机制触发公共回调。这些定时器以指定的间隔发出一个Linux信号，然后这个信号被一个全局（静态）函数捕获。通常不建议对任何需要可靠采样的事物使用定时器，例如adc转换器或采样速率高于几Hz的传感器。在raspberryPI上使用pigpio库及其定时器回调-如果需要的话。

## 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.



## 3.5 Conclusion

线程在实时编码中起着核心作用，因为它们与阻塞IO一起建立回调接口。每个事件处理程序都在单独的线程中运行。

回调也用于发出线程终止的信号，这再次显示了线程和回调之间的密切关系。

# 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;
```



# Chapter 4

# 图形用户界面Qt内的实时事件处理

## 4.1 Introduction

Qt是Linux，Windows和Mac的跨平台windows开发环境。这样的图形用户界面具有实时性要求，因为例如，按下按钮应该触发应用程序的即时响应，以便用户认为这是瞬时的。QT也可以使用上面介绍的回调，但它们被包装在QT特定的信号槽概念中。Qt中的元素是小部件，可以包含任何形式的绘图，按钮或文本字段。他们是班级。您可以定义自己的小部件或使用现成的小部件。QT中的实时通信发生在小部件之间，然后一个小部件调用另一个小部件，例如一个按钮，如果按下按钮，则调用主窗口。

窗口小部件在屏幕上的排列由布局管理。Qt中有不同的声明布局的方法。一种是使用标记语言，然后具有匹配的类;另一种是创建只使用C++类。我们展示了如何使用第二种方法组织布局。这避免了学习额外的语言，并且与使用代码在这个框架和其他框架（例如SwiftUI）中声明布局的一般趋势一致。这是一个小部件如何组织成嵌套的垂直和horizontal布局的例子（见图）。4.1为结果）。

创建3小部件按钮=新QPushButt
on;

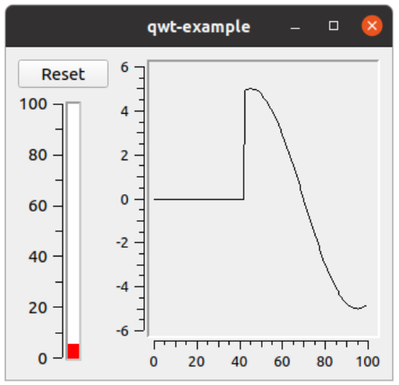


Figure 4.1: QT example layout

```
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);
```



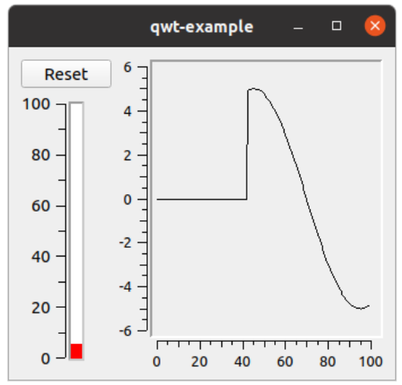


图4.1：QT示例布局

```
thermo=新QwtThermo;plot=
新QwtPlot;

垂直布局vLayout=newQVBoxLayo
ut;vLayout->addWidget(button);
vLayout->addWidget(thermo);


水平布局hLayout=newQHBoxLa
yout;hLayout->addLayout(vLay
out);hLayout->addWidget(plot);


主布局setLayout(hLayo
ut);
```

## 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();
}
```



## 4.2 Qt中的回调

### 4.2.1 来自小部件的事件

与我们使用接口的低级回调机制相比，Qt更直接调用类中的方法。问题是函数指针不能直接用作类具有指向其本地数据的实例指针。所以一个类的方法需要和实例指针结合起来。Qt方法"connect"正是这样做的:

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

QPushButton实例按钮有一个名为clicked()的方法，每当用户单击按钮时都会调用该方法。然后将其转发到应用程序小部件中的方法reset()。

### 4.2.2 绘制通过回调到达的实时数据

一般的想法是将回调的实时样本存储在缓冲区中，并以较低的速率触发屏幕刷新。例如，我们可以选择每40毫秒重放一次缓冲区中的样本，因为这对用户来说足够快，而绘制每个样本的整个缓冲区将过于占用CPU。

当样本到达时，会实时调用回调addSample():

voidWindow::addSample(floatv){将新输入添加到绘图std::move(yD
ata yData+plotDataSize1 yData+1);yData[0]=v;

}

其将样本v存储在移位缓冲器yData中。

然后屏幕刷新（这是缓慢的）以较低和不可靠的速率完成:

```
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.



update()在定时器事件处理程序中生成一个paint事件，Qt然后"尽快"（也就是说，不是实时）调用重绘处理程序来重绘小部件的画布:

voidScopeWindow::paintEvent(QPaintEvent*){QPainterp
aint(this);

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

请注意，定时器和update()函数都不是以可靠的方式调用的，而是在Qt选择这样做的时候。所以Qt定时器不能用于采样数据，而只能用于屏幕刷新和其他非时间关键的原因。

## 4.3 Conclusion

Qt中的事件是由用户交互生成的，例如按下按钮或移动鼠标。和以前一样，Qt通过connect()方法提供了一个回调机制。Qt定时器的回调可以用于动画，但不能用于实时事件，因为Qt定时器不能保证可靠的定时。

# 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.



# Chapter 5

# 实时web服务器客户端通信

## 5.1 Introduction

有各种各样的Web服务器客户端应用程序，从供应商网站上的购物篮到社交媒体。

通常很容易创建动态内容（请参阅PHP或nodejs），这是有据可查的。但是，将实时数据从C++馈送到网页或实时按钮按回C++有点困难。

识别事件生成的位置非常重要：触发事件的始终是客户端（web浏览器或移动应用程序），无论是将数据发送到服务器还是请求数据。它总是由客户发起的。

## 5.2 REST

Web客户端（浏览器或电话应用程序）之间的接口通常通过web服务器上的URL进行通信来实现为表示状态传输体系结构（REST）API。此API的要求非常一般，不会定义实际的数据格式:

统一界面。任何连接到URL的设备都应该得到相同的回复。无论网页或手机请求传感器的温度，返回的格式必须始终相同。

**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 formats

### 5.3.1 Server → client: 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.

### 5.3.2 Client → server: POST

When a website or mobile app wants to send data (`application/x-www-form-urlencoded`) back to the server it needs to encode it in the form of a single text-line where the key/value pairs are combined with &-signs:

```
temperature=20&steps=100&comment=all+good%33
```

The receiver then has the task to entangle this stream into a suitable data-format, for example a map.



客户端-服务器解耦。客户需要知道的唯一信息是
服务器的URL来请求数据或发送数据。

无国籍状态。每个请求都需要包括所有必要的信息，并且不能依赖于以前的请求。例如，对缓冲区的请求不能改变缓冲区，而只是从它读取，以便在不久之后读取缓冲区的另一个用户收到相同的数据。

请参阅https：www.ibm.com云学习rest-api的完整列表
REST设计原则。

## 5.3 数据格式

### 5.3.1 Server → client: JSON

最流行的dataformat是JSON（应用程序json），它基本上是（可嵌套）键值对的映射:

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

由于JSON是人类可读的文本，web服务器可以简单地生成该文本通过http或https发送。除了MIME格式是'applicationjson'而不是html之外，没有区别。

### 5.3.2 客户端→服务器：POST

当网站或移动应用程序想要将数据（应用程序x-www-form-urlencoded）发送回服务器时，它需要以单个文本行的形式对其进行编码，其中键值对与&-符号组合:

```
temperature=20&steps=100&comment=all+good%33
```

然后，接收器的任务是将此流纠缠成合适的数据格式，例如地图。

## 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.

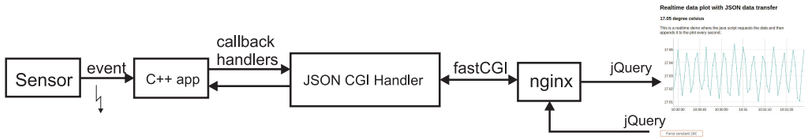


Figure 5.1: FastCGI dataflow.

### 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 server side which can then be processed by client JavaScripts. For realtime applications JSON transmission is perfect because the client-side JavaScript can request JSON packages and directly turn them into variables.

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`.



## 5.4 Server

在Linux系统上需要设置web服务器。有多种不同的选项可用，但我们在这里关注可用于C++通信（即CGI）的选项。

### 5.4.1 Web服务器（httphttps）

NGINX：易于配置，但非常灵活的web服务器。发音为"EngineX"。

Apache：难以配置但安全的选项

Lighttpd：较小的web服务器，内存占用较小。发音为"轻"。

请注意，可以在同一时间运行不同的web服务器，然后它们作为外部世界可见的中央web服务器的代理。
特定的nginx使得实现这一点非常容易。

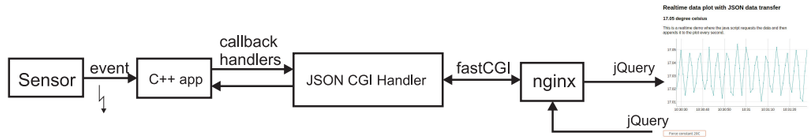


Figure 5.1: FastCGI dataflow.

### 5.4.2 FastCGI

FastCGI（见图5.1）是用C++编写的，并生成httphttps请求的全部内容。特别是在这里，我们生成JSON数据包服务器端，然后可以由客户端JavaScripts处理。对于实时应用程序来说，JSON传输是完美的，因为客户端JavaScript可以请求JSON包并直接将它们转换为变量。

快速CGI程序是一个UNIX命令行程序，它与web服务器（nginx，Apache，...）通过UNIX套接字，该套接字反过来是位于临时目录中的伪文件，例如tmpsensorsocket。

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/json_fastcgi_web_api is wrapper around the quite cryptic fastcgi C library and we discuss its callback handlers now.

### 5.4.3 Server → client: JSON

The fastCGI callback expects a JSON string (`application/json`) with the data transmitted form the server to the client. There is helper class JSONGenerator which generates the JSON data from various C++ types:

```
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 one with the temperature from the sensor.
**/
virtual std::string getJSONString() {
      JSONCGIHandler::JSONGenerator jsonGenerator;
      jsonGenerator.add("epoch",(long)time(NULL));
      jsonGenerator.add("temperatures",temperatureArray);
      return jsonGenerator.getJSON();
    }
};
```



然后，web服务器将某些httphttps请求映射到此套接字。Nginx的示例配置如下所示:

```
位置传感器{包括
                    astcgi_params;
   fastcgi_pass   unix:/tmp/sensorsocket;
 }
```

如果用户通过URL进行请求www.mywebpage.com然后nginx通过这个socket与fastcgi程序联系。然后fastcgi程序需要返回内容。在内部，这将是fastcgi程序内部的c++回调。

C++fastcgiAPIhttps:github.comberndporrjson_fastcgi_web_api是相当神秘的fastcgic库的包装器，我们现在讨论它的回调处理程序。

### 5.4.3 Server → client: JSON

FastCGI回调需要一个JSON字符串（应用程序json），其中包含从服务器传输到客户端的数据。有一个帮助类JSONGenerator，它从各种C++类型生成JSON数据:

```
classJSONcallback:publicJSONCGIHandler::GETCallback{public:***
获取数据并将其发送到web服务器。*回调创建两个JSON条目。一
个带有*时间戳，一个带有来自传感器的温度。**虚拟std::stringget
JSONString(){
      JSONCGIHandler::JSONGenerator jsonGenerator;
      jsonGenerator.add("epoch",(long)time(NULL));
      jsonGenerator.add("temperatures",temperatureArray);
      return jsonGenerator.getJSON();
    }
};
```

### 5.4.4 Client → server: POST

Like in any GUI the client can press a button and create an event. On the client side this is packaged with jquery into an `application/x-www-form-urlencoded` stream and then sent over to the server. In the server, `libcurl` is the standard tool for decoding the `application/x-www-form-urlencoded` stream back into a C++ data structure, for example a map.

Here, the server then receives the `application/x-www-form-urlencoded` data as a callback called "postArg":

```
virtual void postString(std::string postArg) {
    auto m = JSONCGIHandler::postDecoder(postArg);
    float temp = atof(m["volt"].c_str());
    std::cerr << m["hello"] << "\n";
    sensorfastcgi->forceValue(temp);
}
```

which is then decoded into key/value pairs in the form of a C++ `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);
```



### 5.4.4 客户端→服务器：POST

就像在任何GUI中一样，客户端可以按下一个按钮并创建一个事件。在客户端，它与jquery打包成一个应用程序x-www-form-urlencoded流，然后发送到服务器。在服务器中，libcurl是将应用程序x-www-form-urlencoded流解码回C++数据结构（例如地图）的标准工具。

在这里，服务器然后接收应用程序x-www-form-urlencoded数据作为称为"post Arg"的回调:

```
virtualvoidpostString(std::stringpostArg){
    autom=JSONCGIHandler::postDecoder(postArg);float
    temp=atof(m["volt"].c_str());std::cerr<<m["hello"]<<"\
    n";sensorfastcgi->forceValue(temp);

}
```

然后以c++std::map的形式将其解码为键值对。

## 5.5 客户端代码：网站的javascript

通常在客户端（=网页），使用嵌入JavaScript的HTML生成实时输出输入，而无需重新加载页面。JavaScript是事件驱动的，并且有回调，所以它非常适合实时应用程序。使用jQuery从服务器请求和发布JSON。

例如，我们每秒钟从服务器请求数据作为JSON数据包:

```
当JSON数据到达时回调函数getterCallback(result){

  vartemperatureArray=结果。温度;在这里绘制阵列

}
```

定时器回调（与Qt中定义刷新率的想法相同）函数getTemperature（）{
获取JSON数据$。getJSON（"数据：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 requested from the server.



```
}

文档准备回调函数文档准备(){

  每隔一个窗口向服务器请求新数据。intervalId=setInterval（getTemp
  erature，1000）；
}

当网页已加载$(document)时调用。准备就绪(docume
ntReady);
```

JAVA，Kotlin或Swift中的手机编程也像上面的JS代码一样纯粹是回调驱动的，并且仅在语法上有所不同。

## 5.6 Conclusion

基于web的通信中的事件总是由web浏览器或移动应用程序以与Qt完全相同的方式触发，例如按下按钮。这同样适用于客户端计时器从服务器请求数据的动画。因此，这些客户端侧事件或者导致从web浏览器向web服务器传输数据或者从web服务器请求数据。现在的协议总是http或https和RESTful接口

JSON是从服务器请求的最流行的数据格式。

# 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 {
```



# Chapter 6

# Setters

在图中。1.2我们已经看到，数据通过回调从传感器流到C++类，然后通过setter从内部c++类返回到电机或显示输出。设置器还用于设置配置参数。

Setter是类中的一个简单方法，例如设置电机的速度:

classMotor{***设置左轮转速*@param速度在−1和+1之间**voidsetLeftWheelSpeed(floatspeed);；

与回调一样，重要的是要从硬件中抽象出来，例如将电机的速度在 1和+1之间标准化，并隐藏类中PWM或GPIO端口的复杂性。

如果一个setter有多个参数，特别是对于配置，强烈建议使用结构来设置值。例如设置ADS1115的参数:

/**
*ADS1115启动设备时的初始设置。**结构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.



```
/**
*使用I2C总线(99%总是设置为1)**inti2c_bus=
1;

/**
*Ads1115的I2C地址**uint8_t地址=DEFAULT_AD
S1115_ADDRESS;;

/**
*在后台启动数据采集，并使用新样本调用*回调。*\param设置带有
设置的结构。**voidstart(ADS1115settingssettings=ADS1115settin
gs());
```

如果setter设置了大的缓冲区，那么强烈建议在类的构造函数中分配内存，然后在运行时通过引用调用setter。使用表示其长度的数组类型，例如std::array或隐式表示其长度的标准const数组。

**恒定采样率输出（音频，。..在许多应用中，输出设备具有固定的采样率，例如数字到模拟转换器。在这种情况下，c++驱动程序类将再次具有阻塞写入循环，定期读取由setter填充的缓冲区，理想情况下总是提前。您需要决定如果没有新数据到达会发生什么，例如插值输出或将其搁置。当然，您也可以通过音频写入循环实现回调以请求样本，但最终需要解决音频到达和发送之间的冲突。**

## 6.1 Conclusion

Setter是将事件传回物理设备的简单方法。作为回调，setter应该始终隐藏硬件设备的低级复杂度，并接收标准化或物理单元。

# Appendix A

# License





# Appendix A

# License



37