driverCallback.h*

① //…… 单行注释

② /*……*/ 多行注释

```
#ifndef DRIVERCALLBACK_H
#define DRIVERCALLBACK_H

class DriverCallback {
    public:
        //virtual functions to be override
        /*!
        * This virtual method is used to fast Fourier transform the raw data and
        * output the frequency domain data
        */
        //这种虚拟方法被用来对原始数据进行快速傅里叶变换，  输出频域数据
        virtual void fftData(int *, int) = 0;

        /*!
        * This virtual method is used to iir filter the raw data in realtime
        */
        //这种虚拟方法被用来实时过滤原始数据
        virtual int* lpData(int *) = 0;
};

#endif
```

被覆盖的虚拟函数

一个输入 一个输出

原始数据

实时 过滤原始数据

fftClass.cpp*

```
#include "FftClass.h"
#include <cmath>
#include <cstdint>
```

导入三个包!

```
/*
 * used for fft process
 */
//用于 FFT 过程
FftClass::FftClass(int buffer_size){
    num_samples=buffer_size;
    max_fre = 0;
    //* malloc 一个 fft 缓冲区 *
    /* malloc a fft buffer */
    in = (double*)fftw_malloc(sizeof(double)*num_samples);
    //奈奎斯特频率输出理论
    /* nyquist frequency output theory  */
    n_out = ((num_samples/2)+1);
    //要求缓冲
    /* claim a buffer */
```

FFT过程

输入

① 有一个fft缓冲区

缓冲区尺寸

fftClass

输出理论

要一个缓冲区域

claim a buffer

```
out = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * n_out);
nFreqSamples = num_samples / 2;

x = new double[nFreqSamples];
y = new double[nFreqSamples];
for(int i = 0; i < nFreqSamples;i++) {
    x[i] = 0.0;
    y[i] = 0.0;
  }
}
```

fftclass::update(){}

```
double FftClass::update(){

  fftw_plan plan_forward;
  //实在在复杂在复杂在出来
  /* real in complex out */
  plan_forward = fftw_plan_dft_r2c_1d(num_samples,in,out,FFTW_ESTIMATE);
```

实际复杂的输入然后输出

```
  /* do it */
  fftw_execute(plan_forward);
```

一个执行

```
  //销毁计划
  //destroy plan
  fftw_destroy_plan(plan_forward);
```

一个破坏

```
  yMax = 0.0;
  //打印输出
  //print the output
```

打印输出

```
  for(int i=0;i<n_out;i++)
  {
      /* only care about its magnitude */
      //只关心它的大小
      mag = std::sqrt(out[i][0]*out[i][0] + out[i][1]*out[i][1]);
      array[i] = mag;
      if ((mag > yMax) && (i > 9)){
          yMax=mag;
```

只关心大小、

```
          max_fre = i*((double)SAMPLE_RATE / FFT_BUFFER_SIZE);
      }
  }
  array[0] = 0;
  std::cout << max_fre << std::endl;
  return max_fre;
}
```

然后输出一个 max_fre

1) global scope (全局作用域符), 用法 (::name)
2) class scope (类作用域符), 用法 (::name)
3) namespace scope (命名空间作用域符), 用法: (namespace::name)

```
FftClass::~FftClass(){
        fftw_free(in);
        fftw_free(out);
        delete []x;
        delete []y;
}

/*!
  * fill data into fft buffer
  */
```

将数据填入fft区域

```
void FftClass::fill_buffer(int* buffer_tmp){
        for(int i=0;i<num_samples;i++){
                double buffer_num=(double)*buffer_tmp;
                buffer_num /= INT32_MAX;
                buffer_tmp++;
                in[i]=buffer_num;
        }
}
```

一步步填充进入

fftClass.h*

```
#ifndef FFT_CLASS_H
#define FFT_CLASS_H
#include <iostream>
#include <fftw3.h>
// #include <qmainwindow.h>
// #include <qobjectdefs.h>
// #include <qtimer.h>
#include <stdint.h>
// #include <qwt/qwt_plot.h>
// #include <qwt/qwt_plot_curve.h>
// #include <QMainWindow>
#include "i2s_mems_mic.h"
#define FFT_BUFFER_SIZE 1024

class FftClass{

public:
        /*! constructor:
          *
          * Initialize all the data that needs to be used in the Fast Fourier transform
          *
          *
          *   @param int The data length of the data to be processed by the Fast Fourier
```

初始化所有需要使用的数据

数据的长度

* Transform
*/
FftClass(int buffer_size = FFT_BUFFER_SIZE);
//初始化所有需要在快速傅里叶变换中使用的数据

// @param int 要被快速傅里叶变换处理的数据的长度 变换的数据长度
/*! destructor:
*
* All the memory applied for on the heap are released here, otherwise it will
* cause memory overflow, thereby reducing the stability of the program；
*

*/
内存在这里释放，以防内存溢出
~FftClass();
//所有在堆上申请的内存都在这里释放，否则会导致内存溢出，从而降低程序的稳定性；*。
public:
/*! fill the buffer
*
* Put the audio data collected in real time into the array to be fast Fourier
* transformed
*
* @param int * Pointer to the first address of the audio data
*/
//将实时收集的音频数据放入阵列中，进行快速傅立叶变换。转化
// @param int * 指向音频数据的第一个地址的指针

void fill_buffer(int *);

执行变换，更新数据
/*! execute the Fase fourier Transform and update the Data needed to update the
*UI
最大振幅值
*
* @return double The largest value of amplitude in the current spectrum
*/
double update();

在频域中存储样本数据
double array[513]; /*!< Stores per-sample data in the frequency domain */
double max_fre;
double *max_fre_p = &max_fre;
//执行 Fase fourier 变换并更新所需的数据。UI
// @return double 当前频谱中振幅的最大值
private:
时域中的音频数据
double *in; //audio data in time domain//时域中的音频数据
fftw_complex *out; // audio data in frequency domain// 频域中的音频数据
频域中的音频

```
    double *x; //<not used>
    double *y; //<not used>
```

在频域中的量级

```
    double mag; //magnitude of audio data in frequency domain//音频数据在频域中的量
级
    double yMax; // the maxium amplitude samples//最大振幅的样本

    int num_samples;
    int n_out; //the number of samples in the output of fast fourier Transform//快速傅立叶
变换的输出中的样本数
    int nFreqSamples; //the number of samples in frequency domain //频域中的样本数
};

#endif
```

global.cpp

```
#include "Global.h"

bool global_program_exit = false;
```

global.h

```
#ifndef GLOBAL_H
#define GLOBAL_H

/*!
 * A global semaphore used to determine whether the current
 * current process has ended
 */
extern bool global_program_exit;
//一个全局信号，用于确定当前的
//*  进程是否已经结束
#endif //! Globale.h
```

一个全局信号，用于确定当前是否结束了进程

i2s_mems_mic.cpp*

```
/* Use the newer ALSA API */
#include "i2s_mems_mic.h"
#include <cmath>
#include "Global.h"
#define ALSA_PCM_NEW_HW_PARAMS_API
```

使用较新的 ALSA API 接口

```
void I2Smic::open_pcm(){
    //open PCM device
```

打开PCM设备

打开PCM设备的方法

名字、模型

```
	rc = snd_pcm_open(&handle, pcm_name,
	                  stream, open_mode);
	if (rc < 0) {
		fprintf(stderr,
			"unable to open pcm device: %s\n",
			snd_strerror(rc));

		exit(1);
	}
}

void I2Smic::set_params(void) {
	snd_pcm_hw_params_t *params;
```

obj

分配一个硬件参数

```
	/* allocate a hardware params obj  */
	//*  分配一个硬件参数 obj */
	snd_pcm_hw_params_alloca(&params);
	int err;
```

params

```
	//snd_pcm_hw_params_alloca(&params);
	err = snd_pcm_hw_params_any(handle, params);
	if (err < 0) {
		fprintf(stderr,
			"Broken configuration for this PCM: no configurations avaliable: %s",
			 //"此 PCM 的配置被破坏：没有可用的配置：%s"、
			 snd_strerror(rc));

		exit(1);
	}
```

选一个解释文件

交错模式

```
	/* Interleaved mode */
	//*  交错模式  */
	err = snd_pcm_hw_params_set_access(handle, params,
	                                   SND_PCM_ACCESS_RW_INTERLEAVED);
	if (err < 0) {
		fprintf(stderr,
			"Access type not available: %s",
			snd_strerror(rc));

		exit(1);
	}
```

访问类型不可用

```
	/* format */
```

格式

```
        err = snd_pcm_hw_params_set_format(handle, params,
                                                hwparams.format);
        if (err < 0) {
                fprintf(stderr,
                                "Sample format non available: %s",
                                snd_strerror(err));

                exit(1);
        }
//"访问类型不可用：%s"
//"样本格式不可用：%s"、

        /* one channel (mono)*/
```

单声道

```
        err = snd_pcm_hw_params_set_channels(handle, params, hwparams.channels);
        if (err < 0) {
                fprintf(stderr, "Channels count non avaliable");
```

通道数不可用

```
                exit(1);
        }
//"通道数不可用"
//* 设置采样率 *
        /* set sampling rate */
```

设置采样率

```
        err = snd_pcm_hw_params_set_rate_near(handle, params, &hwparams.rate, 0);
        assert(err >= 0);

        /* set period size */
```

设定周期大小

```
        frames = frames_number;
        err = snd_pcm_hw_params_set_period_size_near(handle, params, &frames, 0);
        assert(err >= 0);
  //将参数写入驱动程序
        /* write parameters to the driver   */
```

将参数写入驱动程序

```
        err = snd_pcm_hw_params(handle, params);
        if (err < 0) {
                fprintf(stderr, "unable to installl hw params: ");
                exit(1);
        }
//使用一个足够大的缓冲区来容纳周期
        /* Use a buffer large enough to hold period */
```

加载一个足够大的缓冲区来容纳周期

```
        snd_pcm_hw_params_get_period_size(params, &frames, 0);
  //获取周期时间
        /* get period time */
        snd_pcm_hw_params_get_period_time(params, &val, 0);
}
```

获取周期时间

```
void I2Smic::run(){
	while (!global_program_exit) {
		rc = snd_pcm_readi(handle, &(buffer[currentBufIdx][0]), frames);

		if (rc == -EPIPE) {
			/* EPIPE means overrun */
```
超额完成任务

```
			fprintf(stderr, "overrun occurred\n");
			snd_pcm_prepare(handle);
		} else if (rc < 0) {
			fprintf(stderr,
				"error from read: %s\n",
```
错误的读入。

```
				snd_strerror(rc));
		} else if (rc != (int)frames) {
			fprintf(stderr, "short read, read %d frames\n", rc);
		}
```
短暂的读取。

在这里回调，低调数据和fft.

```
		/* callback here, lowpass data and fft process */
		callback->lpData(&(buffer[currentBufIdx][0]));
		callback->fftData(&(buffer[currentBufIdx][0]), frames);

		/*
		rc = write(1, buffer, size); // write to stdout
		if (rc != size)
			fprintf(stderr,
				"short write: wrote %d bytes\n", rc);
				*/

		/*
		 * switching buffer
		 */
```
开关缓冲器

```
		readoutMtx.lock();
		currentBufIdx = !currentBufIdx;
		readoutMtx.unlock();

	}
}
//这里的回调，低通数据和 FFT 处理
int I2Smic::get_rc(){
	return this->rc;
}
```
注册回调

```
/* register callback */
void I2Smic::registercallback(DriverCallback* cb) {
	this->callback = cb;
```

```
}
```

停止采集数据

```
/* stop data acquisition */
void I2Smic::close_pcm() {
	global_program_exit=true;
	snd_pcm_drain(handle);
	snd_pcm_close(handle);
	///free(buffer);
}
```

i2s_mems_mic.h

```
#ifndef I2S_H
#define I2S_H

#define ALSA_PCM_NEW_HW_PARAMS_API
#define SAMPLE_RATE 8000
#include <alsa/asoundlib.h>
#include <alsa/pcm.h>
#include <cstdint>
#include <errno.h>
#include <thread>
#include <mutex>
#include "DriverCallback.h"
//buffer size
```

缓冲区大小

```
#define frames_number 1024

static struct snd_params{
	snd_pcm_format_t format = SND_PCM_FORMAT_S32_LE;
	unsigned int channels = 1;
	unsigned int rate = SAMPLE_RATE;
} hwparams;

class I2Smic {

public:
	/*! open PCM device
```

打开PCM设备
打开数据和写入通道

```
	 *
	 * open the data read and write channel of mic;
	 */
	void open_pcm();

	/*! set parameters
```

建立通道参数

```
	 *
	 * Set data channel parameters to mic
```

```
 */
void set_params(void);
/*! close PCM device
```

关闭PCM设备

```
 *
 * important: When the program ends or the driver class
 * is destructed, this method must be called to close the
 * data channel of mic, otherwise it will cause unpredictable
 * errors
 */
void close_pcm();
```

关闭mic的通道，否则会引起不可预知的错误

```
/*!
 * obtain sound sample
```

获得声音样本

```
 *
 * Continuously obtain audio data, and call the callback
 * function to perform real-time iir filtering and Fast
 * Fourier transformation on the data
 */
void run();
```

不断的获取音乐数据，回调函数和实时进行傅利叶变换

```
/*! register callback
```

注册回调函数

```
 *
 * External classes can register callback functions to the
 * driver class through this method
 */
void registercallback(DriverCallback* cb);

/*! destructor
```

关闭麦克风的方法

```
 *
 * The method to close the mic will be called in this method

 */
~I2Smic() {
        this->close_pcm();
}

int get_rc();
private:

snd_pcm_t *handle;
const int open_mode = 0;
const snd_pcm_stream_t stream = SND_PCM_STREAM_CAPTURE;
char const* pcm_name = "plughw:1";//sound device name
snd_pcm_uframes_t frames;
```

音乐设备的名字

```
    unsigned int val;

    snd_pcm_hw_params_t *params;
    snd_pcm_info_t *info;

    DriverCallback* callback;
    int rc;
    std::mutex readoutMtx;
    int buffer[2][frames_number];
    unsigned currentBufIdx = 0;
};

#endif
```

lp.cpp

```
#include "lp.h"

//lowpass constructor
```

低通构构造器

```
Lp::Lp(int sample_rate) {
    lp.setup(sample_rate, CUTOFF);
}

/*!
*filter data
```

过滤数据

```
*/
double Lp::filter(int v) {
    return lp.filter(v);
}
```

lp.h

```
#ifndef LP_H
#define LP_H

#include <Iir.h>
#include <iir/Butterworth.h>

//cutoff frequency
```

断电频率

```
#define CUTOFF 1000

class Lp {
```

```
    public:
        /*!
        * IIR filter constructor    滤波器构造函数
        *
        * this is a lowpass realtime IIR filter    低通的实时IIR滤波器
        *
        * @param int the sample rate
        */
        Lp(int);

        /*!    对当前传入的音频数据执行
        * execute real-time low-pass filtering on the current
        * incoming audio data    实时低通滤波
        *
        * @param int the sample
        */
        double filter(int v);

    private:
        Iir::Butterworth::LowPass<> lp;
};

#endif
```