

# 逻辑分析仪项目 基于 DshanMCU-F103

## User Manual

Rev. 2.0

2024/07/06

手册属性

类别	嵌入式开发
文档名	逻辑分析仪项目_基于 DshanMCU-F103
当前版本	2.0
适用型号	DshanMCU-F103
编辑	百问科技文档编辑团队
审核	韦东山

更新记录

更新日期	更新内容	更新版本	审核
2024/04/16	首次发布	V1.0	
2024/07/06	课程完结	V2.0	

# 第1章 课程概述与逻辑分析仪体验

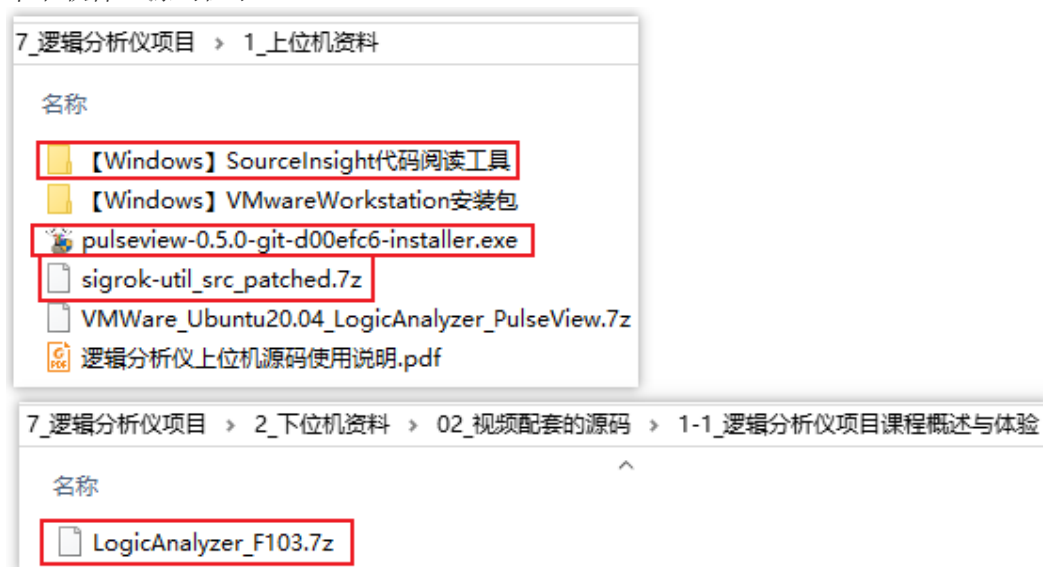
本课程分为 2 部分：

- ① 在下位机里构造虚假数据，体验逻辑分析仪原理
  - 课程概述与逻辑分析仪体验
  - 逻辑分析仪原理与上位机代码分析
- ② 深入分析单片机性能，读取硬件数据，实现真正的逻辑分析仪
  - 如何实现高速采样：使用汇编指令
  - 汇编指令执行时间测量
  - 实现最高频率的采样
  - 使用 `udelay` 实现低频率的采样
  - 使用中断改造逻辑分析仪

本节进行如下体验：

- 安装上位机软件
- 编译、运行下位机软件
- 体验逻辑分析仪

本节软件、源码位于：



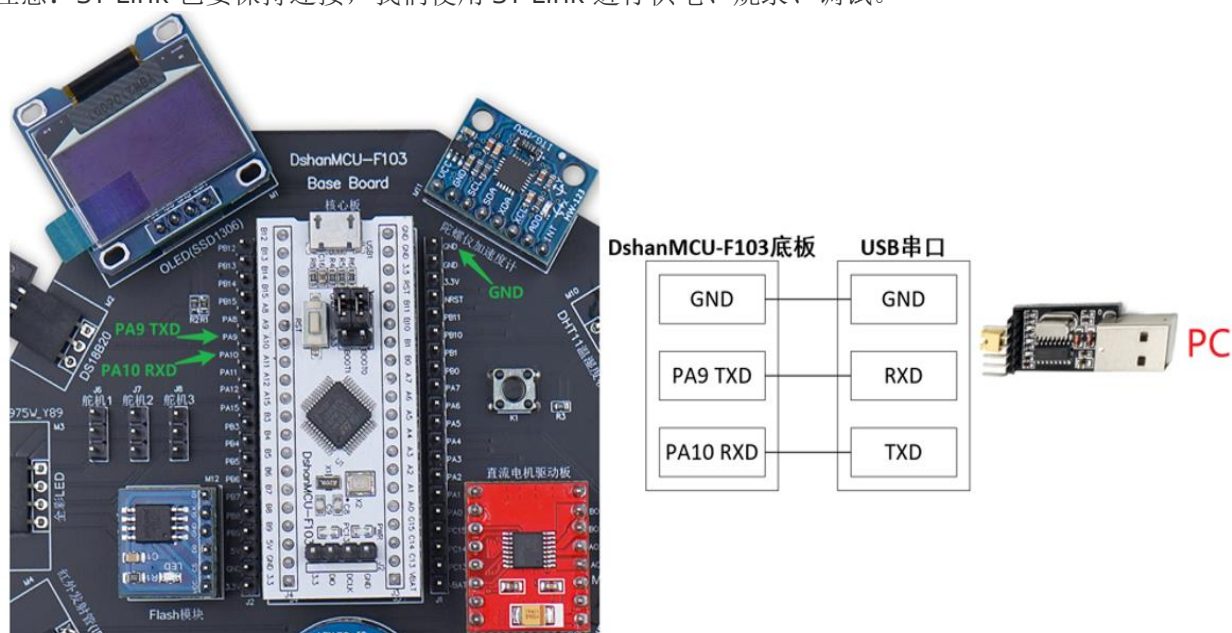
## 1.1 安装软件

安装"pulseview-0.5.0-git-d00efc6-installer.exe"。

## 1.2 连接开发板串口

请按照下图连线：底板的 TXD、RXD 和 USB 串口 RXD、TXD 交叉连接，GND 要互相连接。

注意：ST-Link 也要保持连接，我们使用 ST-Link 进行供电、烧录、调试。

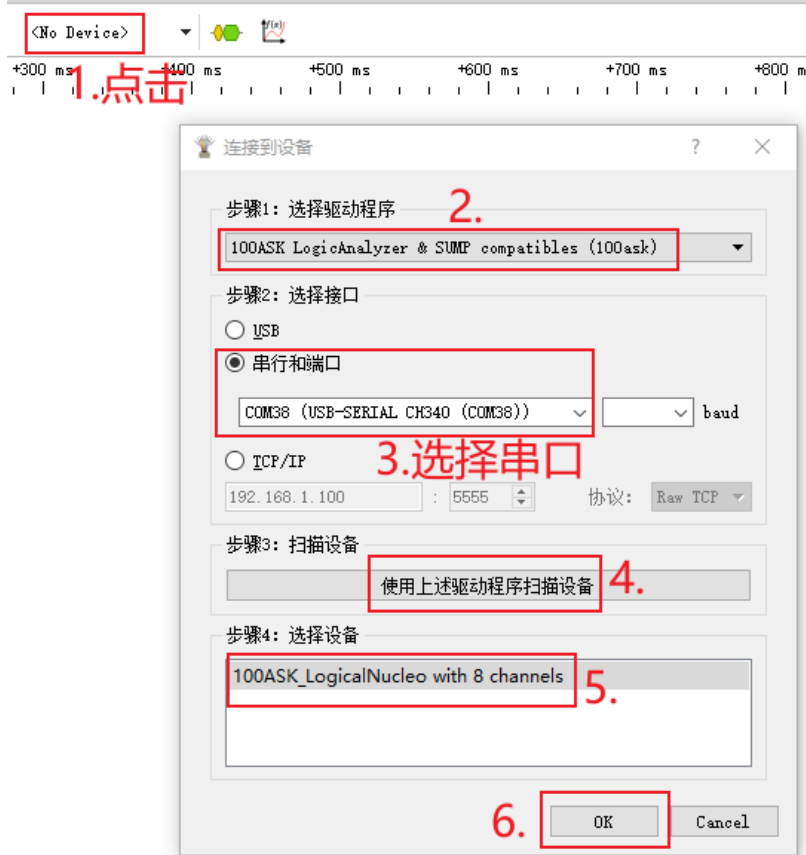


## 1.3 烧录程序

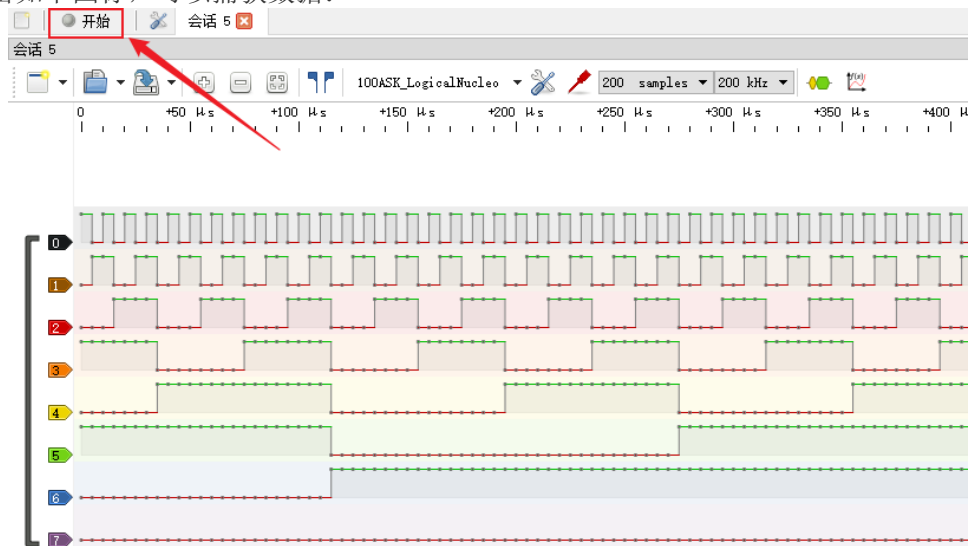
烧录"LogicAnalyzer\_F103"

## 1.4 使用逻辑分析仪

启动 PulseView，如下操作可以识别出逻辑分析仪：



点击如下图标，可以捕获数据：



## 1.5 SourceInsight 使用技巧

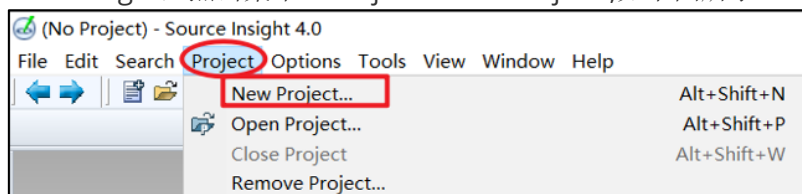
本文档来自我们的 Linux 资料，请灵活变通。

### 1.5.1 建立工程示例

本节新建一个 linux kernel 的 source Insight 工程，你也可以为其他 APP 建立工程，方法是一样的。

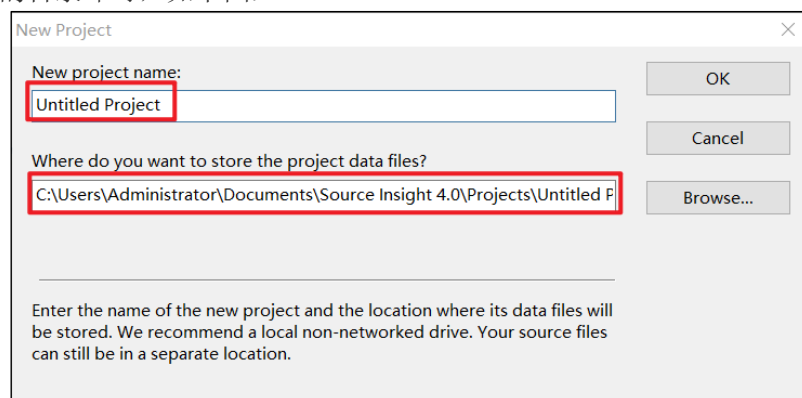
#### 1. 新建工程

运行 source Insight，点击菜单 “Project->New Project”，如下图所示：



#### 2. 设置工程名及工程数据目录

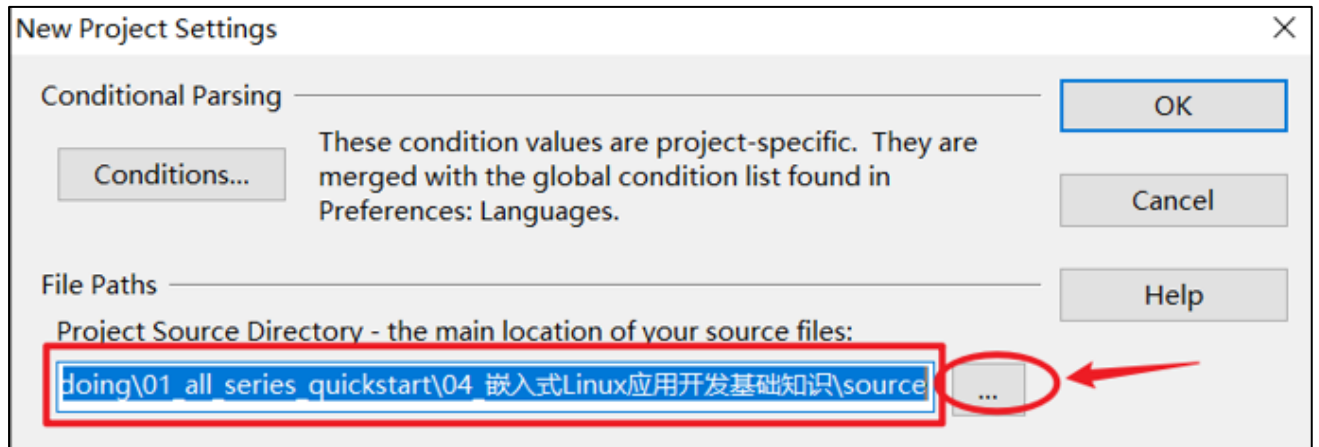
在弹出的 New Project 对话框中设置 “New project name”(项目的名称)，然后设置 Where do you want to store the project data file? (项目文件保存位置)，点击 Browse 按钮选择源码的目录即可，如下图：



### 3. 指定源码目录

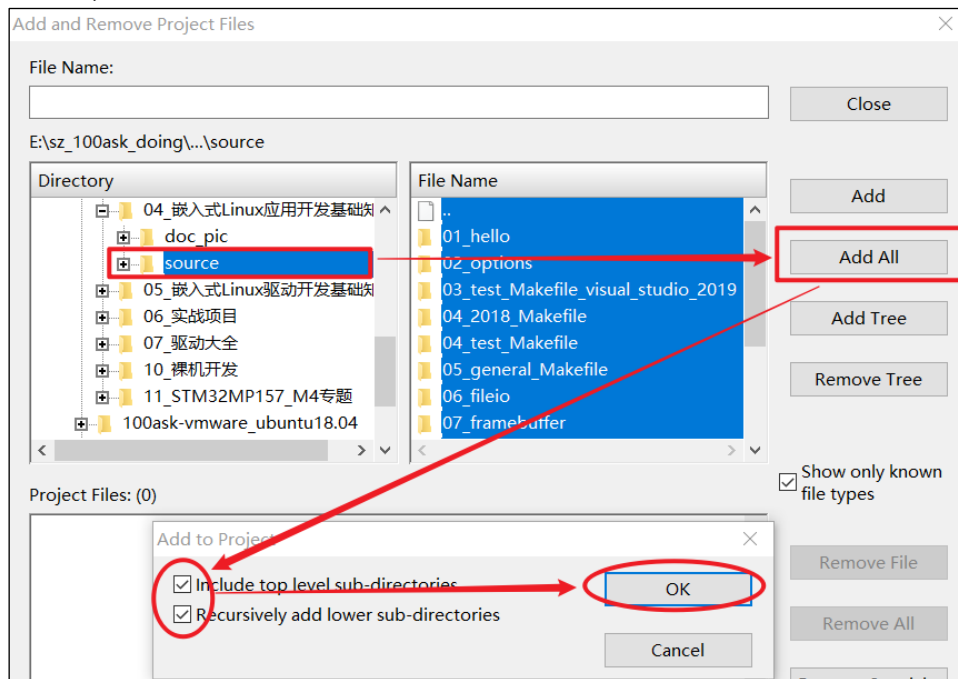
设置源码目录: Project Source Directory – the main location of your source files”()

点击红框左边 “...”选择源码目录, 点击 OK, 如下图所示:

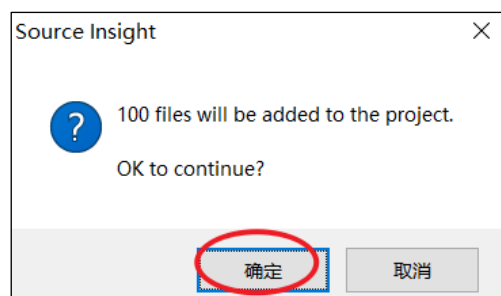


### 3. 添加源码

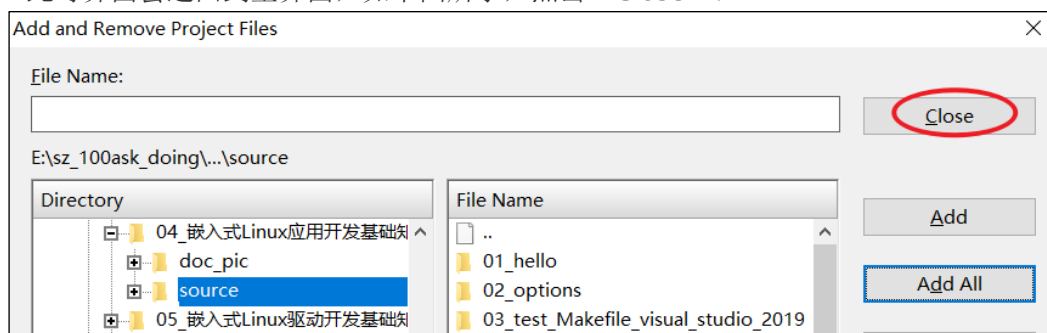
在新弹出的对话框中, 点击 “Add”或 “Add All”。“Add” 是手动选择需要添加的文件, 而 “Add All”是添加所有文件。我们使用 “Add All”,在弹出的提示框中选中 “Recursively add lower sub-directories”(递归添加下级的子目录)并点击 OK。同样的 Remove File,Remove All 是移除单个文件或者移除所有文件, 如下图所示:



添加文件完成后会弹出下面窗口, 点击 “确定” 即可:



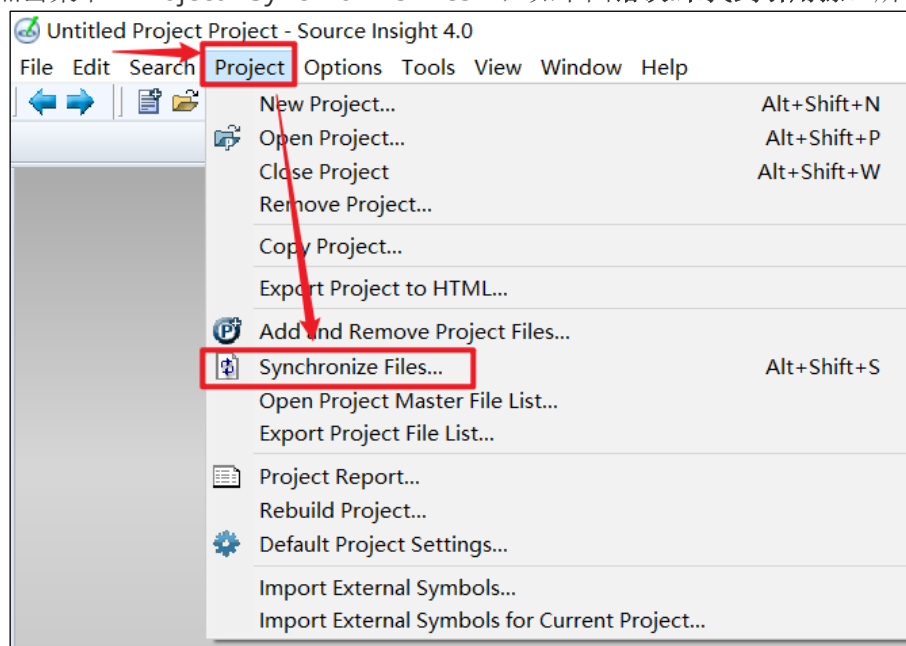
此时界面会返回到主界面，如下图所示，点击“Close”：



#### 4. 同步文件

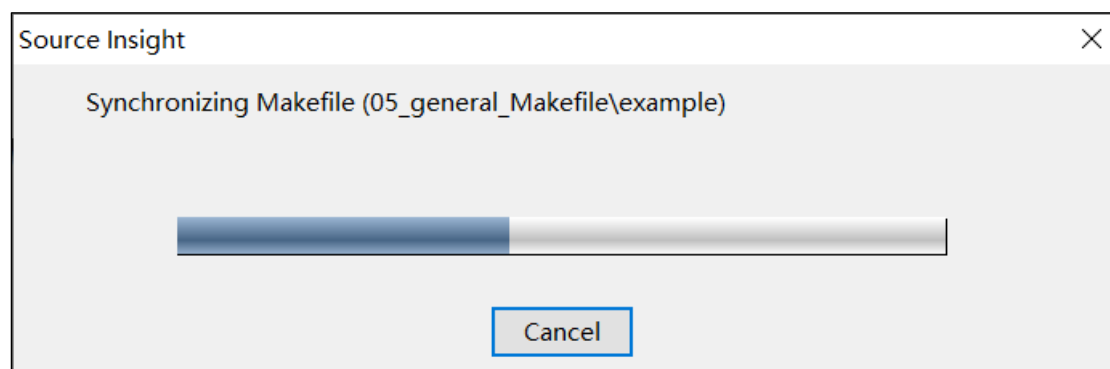
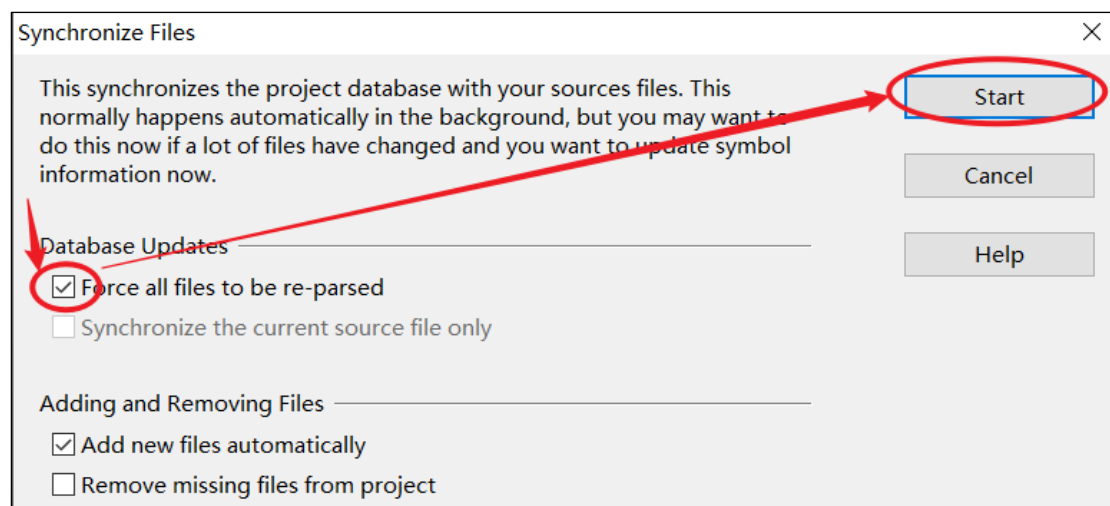
同步文件的意思是让 Source Insight 去解析源码，生成数据库，这样有助于以后阅读源码。比如点击某个函数时就可以飞快地跳到它定义的地方。

先点击菜单“Project->Synchronize Files”，如下图错误!未找到引用源。所示：



在弹出的对话框中选中“Force all files to be re-parsed”(强制解析所有文件)，并点击“Start”按钮开始同步，如下图所示：

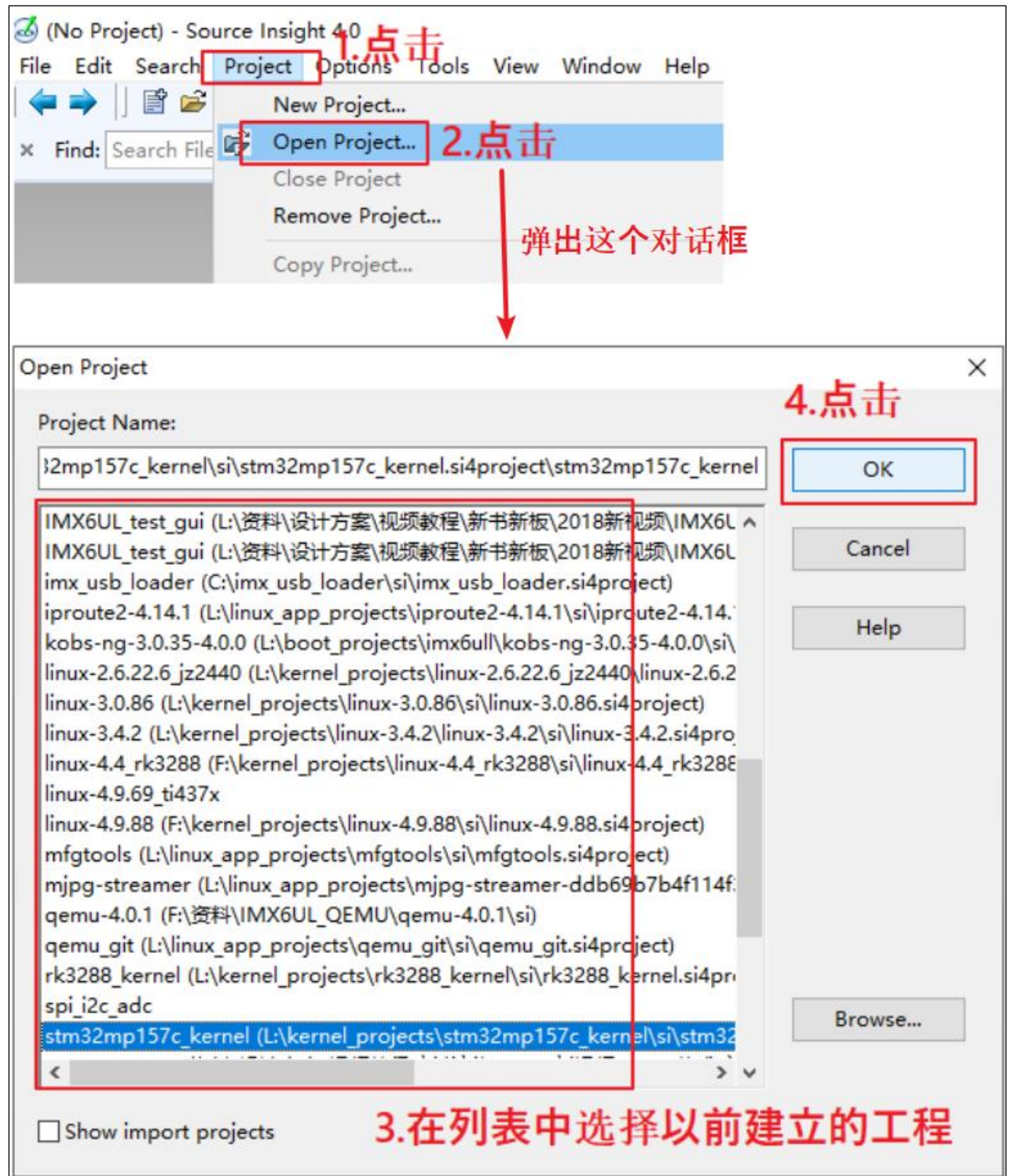




### 1. 5. 2 操作示例

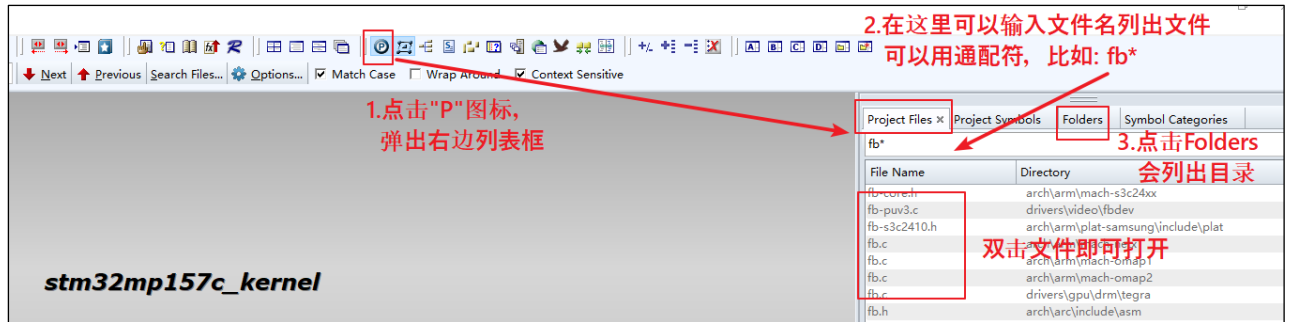
#### a) 打开工程

前面建议工程后，就会自动打开了工程。如果下次你想打开工程，启动 Souce Insight 后，点击菜单“Project -> Open Porject”就可以在一个列表中选择以前建立的工程，如下图所示：



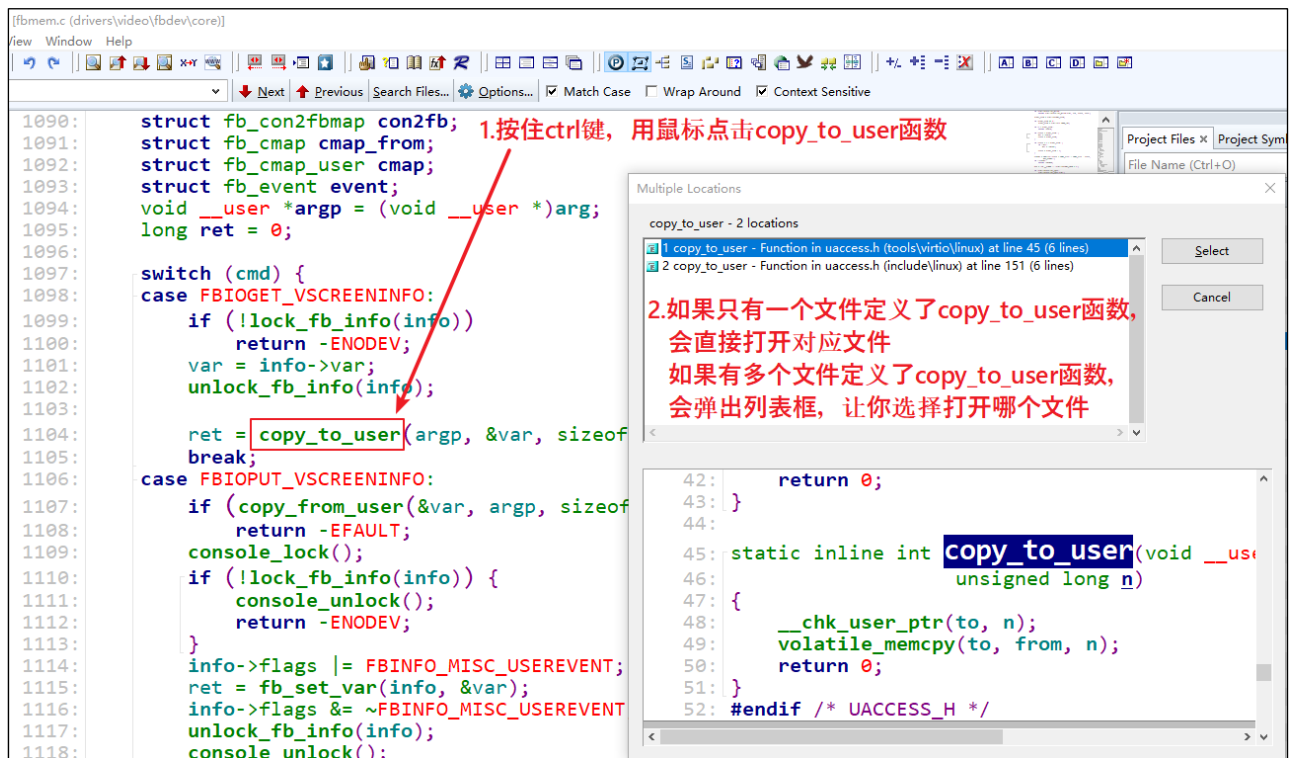
### 1. 在工程中打开文件

点击"P"图标打开文件列表，双击文件打开文件，也可以输入文件名查找文件，如下图所示：



## 2. 在文件中查看函数或变量的定义

打开文件后，按住 ctrl 键的同时，用鼠标点击函数、变量，就会跳到定义它的位置，如下图所示：



3. 查找函数或变量的引用

双击函数，右键点击弹出对话框选择“Lookup Reference”；或者双击函数后，使用快捷键“ctrl+/”来查找引用，如下图所示：



4. 其他快捷键

快捷键	说明
Alt + ,	后退
Alt + .	前进
F8	高亮选中的字符
Ctrl+F	查找
F3 或 Shift+F3	往前查找
F4 或 Shift+F4	往后查找

## 第2章 逻辑分析仪 SUMP 协议分析

根据使用流程分析上位机程序、下位机程序的交互过程，就可以弄清楚逻辑分析仪的协议。逻辑分析仪的协议有很多种类型，我们使用的上位机程序，借用了“openbench-logic-sniffer”逻辑分析仪的代码，它使用 SUMP 协议。SUMP 协议网址：<https://www.sump.org/projects/analyzer/protocol/>。

上位机、下位机源码：

1\_上位机资料\sigrok-util\_src\_patched.7z

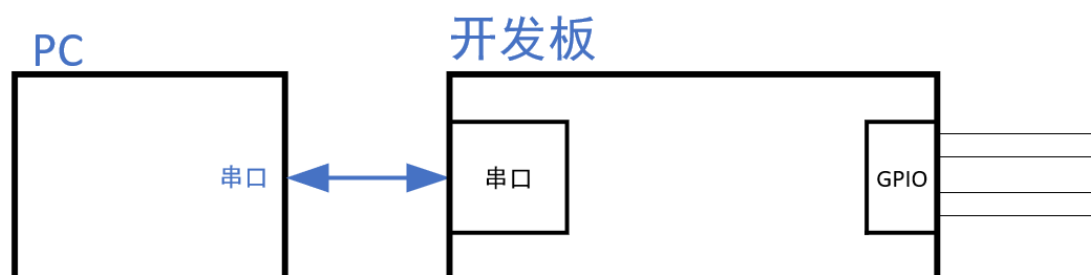
2\_下位机资料\02\_视频配套的源码\1-1\_逻辑分析仪项目课程概述与体验\LogicAnalyzer\_F103.7z

### 2.1 硬件结构

下图是逻辑分析仪的最简单的硬件结构：

- ① 上位机、下位机之间使用串口通信
- ② 下位机使用 GPIO 采集数据

**注意：**商用的逻辑分析仪一般使用 FPGA 采集数据，使用 USB 跟上位机通信。



### 2.2 SUMP 协议

#### 2.2.1 上位机发出的命令及参数

上位机和下位机之间的交互过程为：

- ① 上位机发送命令、数据
- ② 下位机“可能”回复

完整的命令和参数说明如下：

命令	命令值	作用
CMD_RESET	0x00	复位下位机
CMD_ID	0x02	让下位机上报 ID
CMD_METADATA	0x04	让下位机上报参数
CMD_SET_BASIC_TRIGGER_MASK0	0xC0	使能某个通道的触发功能 示例数值：0x01 0x02 0x00 0x00 表示 channel0, 9 使能了触发功能
CMD_SET_BASIC_TRIGGER_VALUE0	0xC1	设置通道的触发值 示例数值：0x01 0x00 0x00 0x00 表示 channel 0 的触发值为高电平 channel 9 的触发值为低电平
CMD_SET_BASIC_TRIGGER_CONFIG0	0xC2	最后一个字节的 bit3 为 1 表示启动

		触发功能 示例数值: 0x00 0x00 0x00 0x08 最后一个字节的 bit3 表示启动触发功能
CMD_SET_DIVIDER	0x80	根据用户设置的采样频率计算出分频系数 注意: 当采样频率大于 100MHz 时, 会"Enable demux mode", 让逻辑分析工作于 200MHz, 分频系数=200MHz/采样频率 - 1 当采样频率小于 100MHz 时, 分频系数=100MHz/采样频率 - 1 示例数值: 0xf3 0x01 0x00 0x00 分配系数为: $0x01f3=499=100\text{MHz}/200\text{KHz}-1$
CMD_CAPTURE_SIZE	0x81	使用 1 个命令发送 READCOUNT DELAYCOUNT 两个参数 示例数值: 0x0c 0x00 0x0c 0x00 前 2 字节表示要采样的次数为 $0x0c * 4 = 48$ 后 2 字节表示要延迟的次数为 $0x0c * 4 = 48$
CMD_SET_FLAGS	0x82	设置 flag, 比如使用启动 demux 模式 根据用户选择的通道, 使能 group (见后面注释)
CMD_CAPTURE_DELAYCOUNT	0x83	示满足触发条件开始采样后, 延迟多少次采样, 才保存数据 示例数值: 0x0c 0x00 0x00 0x00 表示延迟次数为 $0x0c * 4 = 48$
CMD_CAPTURE_READCOUNT	0x84	表示要采样的次数 示例数值: 0x0c 0x00 0x00 0x00 采样次数为 $0x0c * 4 = 48$

对于命令 CMD\_SET\_FLAGS, 它的数值为 32 位的数据, 含义如下 (bit2~bit5 对于 group1~4, 比如 bit2 为 0 表示 group 1 使能):

```
#define CAPTURE_FLAG_RLEMODE1 (1 << 15)
#define CAPTURE_FLAG_RLEMODE0 (1 << 14)
#define CAPTURE_FLAG_RESERVED1 (1 << 13)
#define CAPTURE_FLAG_RESERVED0 (1 << 12)
#define CAPTURE_FLAG_INTERNAL_TEST_MODE (1 << 11)
#define CAPTURE_FLAG_EXTERNAL_TEST_MODE (1 << 10)
#define CAPTURE_FLAG_SWAP_CHANNELS (1 << 9)
#define CAPTURE_FLAG_RLE (1 << 8)
#define CAPTURE_FLAG_INVERT_EXT_CLOCK (1 << 7)
#define CAPTURE_FLAG_CLOCK_EXTERNAL (1 << 6)
#define CAPTURE_FLAG_DISABLE_CHANGROUP_4 (1 << 5)
#define CAPTURE_FLAG_DISABLE_CHANGROUP_3 (1 << 4)
```

```
#define CAPTURE_FLAG_DISABLE_CHANGROUP_2 (1 << 3)
#define CAPTURE_FLAG_DISABLE_CHANGROUP_1 (1 << 2)
#define CAPTURE_FLAG_NOISE_FILTER          (1 << 1)
    #define CAPTURE_FLAG_DEMUX              (1 << 0)
```

2.2.2 下位机发送的回复

1. CMD\_ID 的回复

上位机发送 CMD\_ID 命令（“0x02”）给下位机后，下位机要回复 4 个字节“1ALS”。

2. CMD\_METADATA 命令的回复

上位机发送 CMD\_ID 命令（“0x04”）给下位机后，下位机要回复很多参数给上位机。参数格式为“1 字节的数据类别，多个字节的数据”，说明如下：

上报的数据类别	上报的数据	说明
0x01	"100ASK_LogicalNucleo"	名字
0x20	大字节序的 4 字节	最大采样通道数
0x21	大字节序的 4 字节	保存采样数据的 buffer 大小
0x22	大字节序的 4 字节	动态内存大小(未使用)
0x23	大字节序的 4 字节	最大采样频率
0x24	大字节序的 4 字节	协议版本
0x40	1 字节	最大采样通道数
0x41	1 字节	协议版本
0x00		结束标记

3. 上报的采样数据

对于我们借用的“openbench-logic-sniffer”逻辑分析仪协议，它上报的数据是：先上报最后一个采样的数据，最后上报第 1 个采样点的数据。

每一个采样的数据里，含有使能的 group 的数据，比如使能了 group1、group2、group4，没有使能 group3，那么上报的一个采样数据为 3 字节：第 1 字节对应 group1 的 channel 0~7，第 2 字节对应 group2 的 channel 8~15，第 3 字节对应 group4 的 channel 24~31。

每一个字节数据里，bit0 对应这个 group 里最低的通道的值。

2.3 上位机和下位机时序图

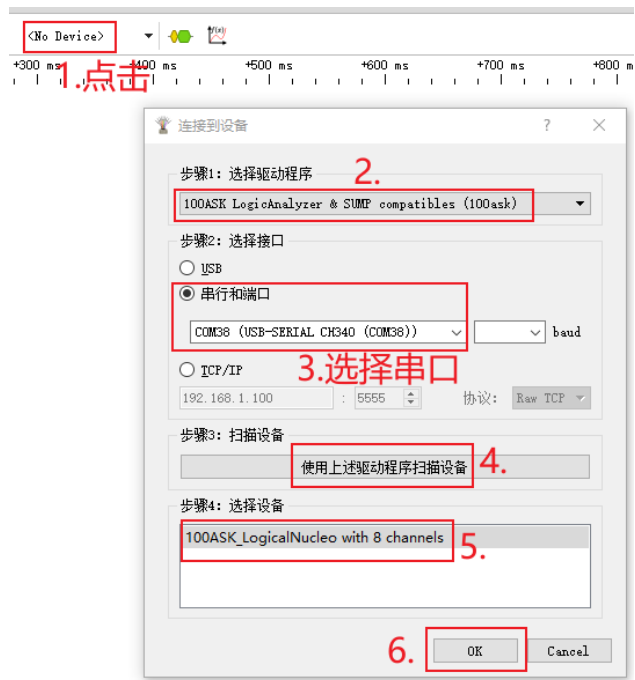
2.3.1 扫描操作的时序图

要理解时序图，建议对比着上位机源码、下位机源码进行分析：

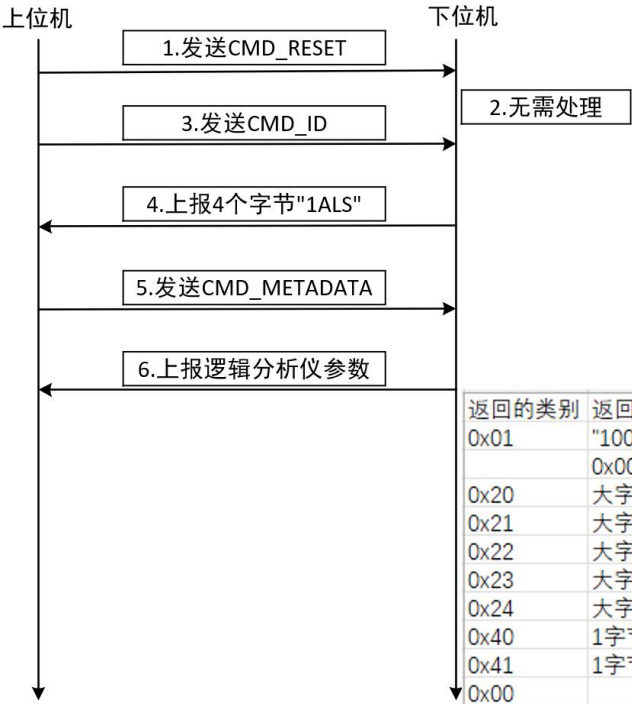
```
// 上位机
sigrok-util_src_patched\libsigrok\src\hardware\openbench-logic-sniffer\api.c, scan函数
// 下位机
LogicAnalyzer_F103\Core\Src\logicanalyzer.c, LogicalAnalyzerTask函数
```

启动 PulseView，如下操作可以识别出逻辑分析仪：





其中涉及的操作、时序图，如下：



返回的类别	返回的数据	说明
0x01	"100ASK_LogicalNucleo"	名字
	0x00	以0结尾
0x20	大字节序的4字节	最大采样通道数
0x21	大字节序的4字节	保存采样数据的buffer大小
0x22	大字节序的4字节	动态内存大小(未使用)
0x23	大字节序的4字节	最大采样频率
0x24	大字节序的4字节	协议版本
0x40	1字节	最大采样通道数
0x41	1字节	协议版本
0x00		结束标记

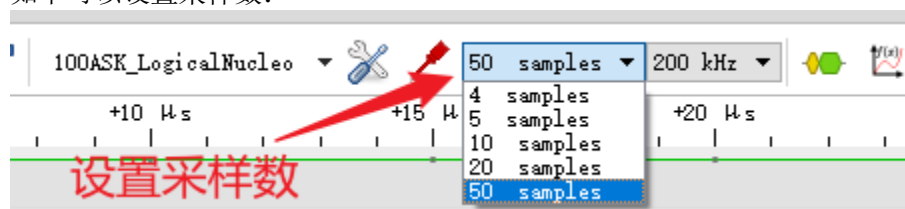


## 2.3.2 逻辑分析仪的设置操作

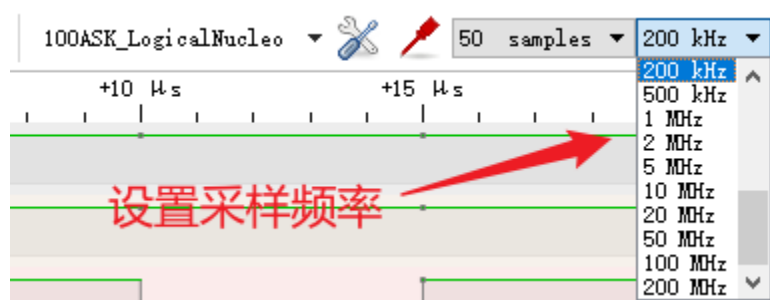
SUMP 协议的逻辑分析仪，最大支持 32 个采样通道，分为 4 组：group 1、group 2、group 3、group 4，每组含有 8 个通道：channel 0~channel 7 属于 group 1，以此类推。

### 1. 设置采样数

如下可以设置采样数：

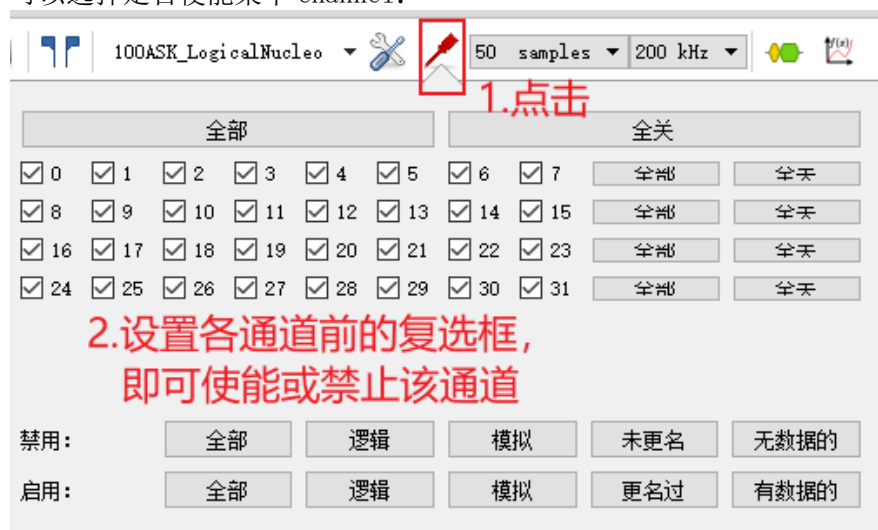


### 2. 设置采样频率



### 3. 使能或禁止通道

可以选择是否使能某个 channel：



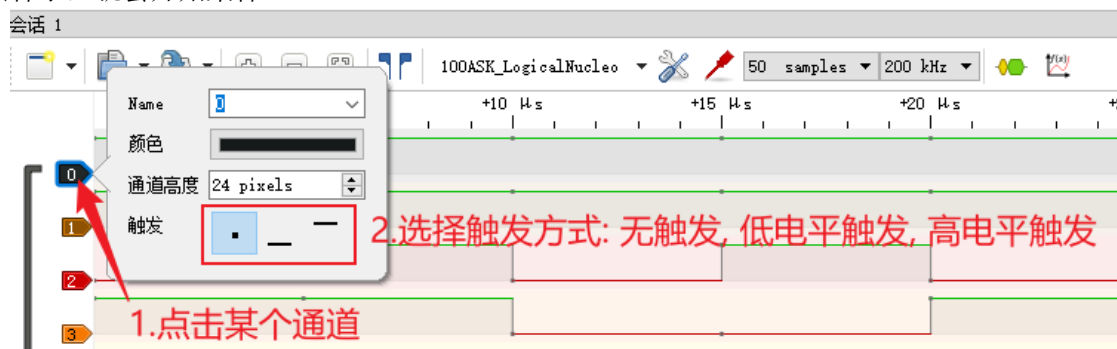
当 32 个通道都使能时，一次采样得到 32 位数据，bit0 对应通道 0，bit31 对应通道 31。

当 group n 里 8 个通道都禁止的话，那么一次采样就可以少传输 1 字节。比如 group 3 里的 channel 16~channel 23 都被禁止后，一次采样就可以得到 3 字节数据，bit16 原来对应 channel 16，现在对应 channel 24，以此类推。

### 4. 设置通道的触发条件

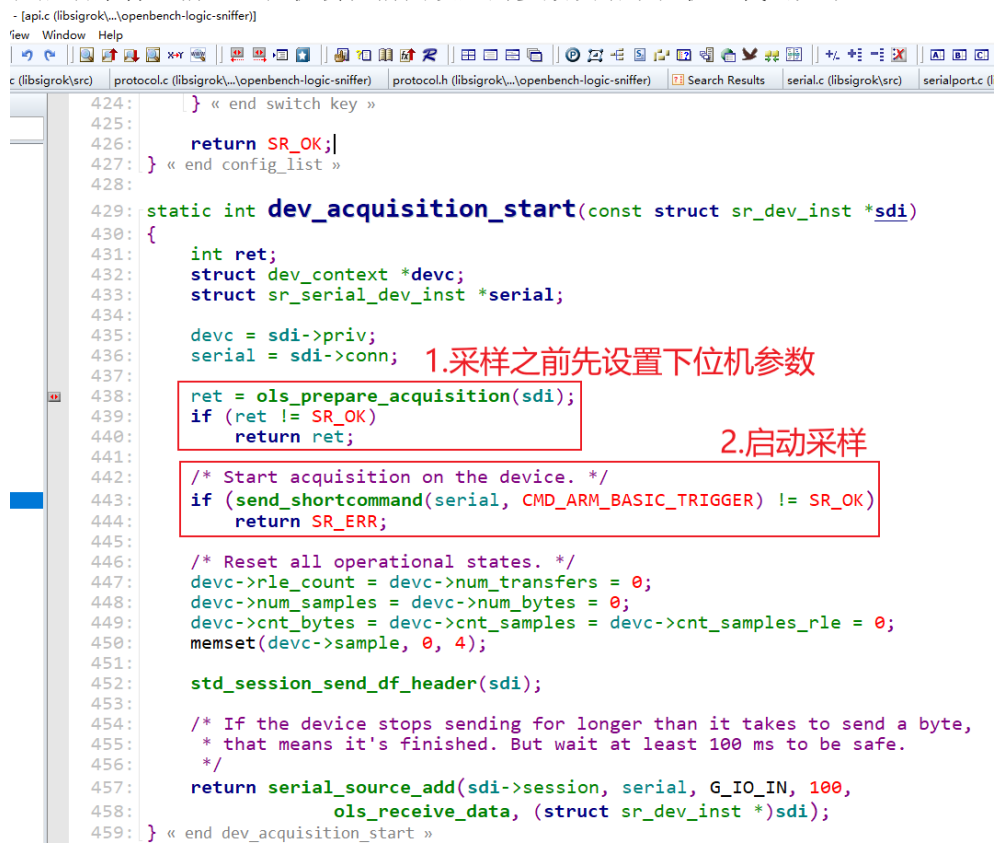
可以设置采样的触发条件（对于使能了触发的多个通道，只要有某个通道的值符合触

发条件了，就会开始采样）：



### 2.3.3 设置操作的时序图

在启动采样之前，上位机会把前面设置的参数发给下位机，代码如下：



```

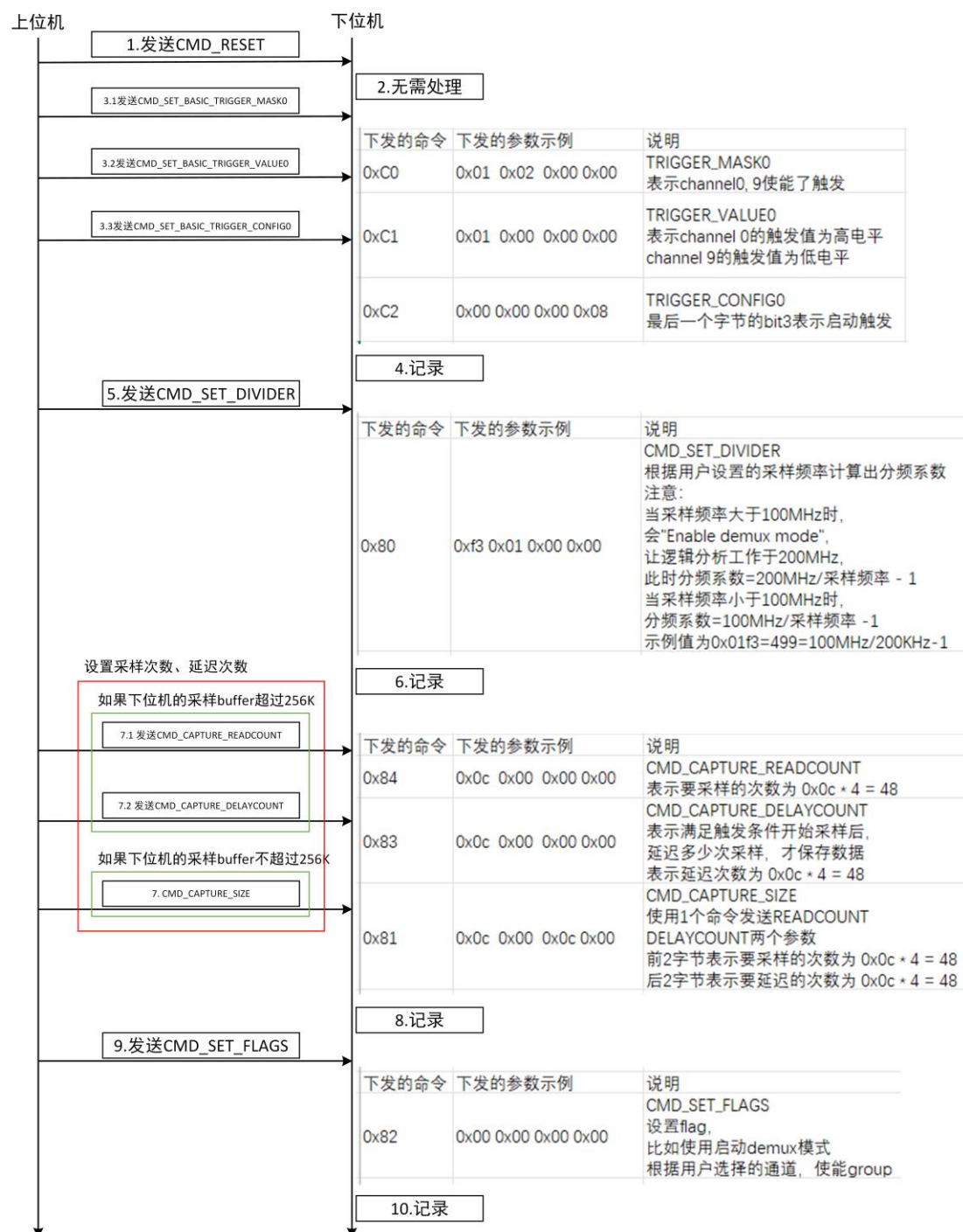
424:     } « end switch key »
425:
426:     return SR_OK;
427: } « end config_list »
428:
429: static int dev_acquisition_start(const struct sr_dev_inst *sdi)
430: {
431:     int ret;
432:     struct dev_context *devc;
433:     struct sr_serial_dev_inst *serial;
434:
435:     devc = sdi->priv;
436:     serial = sdi->conn;
437:
438:     ret = ols_prepare_acquisition(sdi);
439:     if (ret != SR_OK)
440:         return ret;
441:
442:     /* Start acquisition on the device. */
443:     if (send_shortcommand(serial, CMD_ARM_BASIC_TRIGGER) != SR_OK)
444:         return SR_ERR;
445:
446:     /* Reset all operational states. */
447:     devc->rle_count = devc->num_transfers = 0;
448:     devc->num_samples = devc->num_bytes = 0;
449:     devc->cnt_bytes = devc->cnt_samples = devc->cnt_samples_rle = 0;
450:     memset(devc->sample, 0, 4);
451:
452:     std_session_send_df_header(sdi);
453:
454:     /* If the device stops sending for longer than it takes to send a byte,
455:      * that means it's finished. But wait at least 100 ms to be safe.
456:      */
457:     return serial_source_add(sdi->session, serial, G_IO_IN, 100,
458:         ols_receive_data, (struct sr_dev_inst *)sdi);
459: } « end dev_acquisition_start »

```

ols\_prepare\_acquisition 函数在如下文件中：

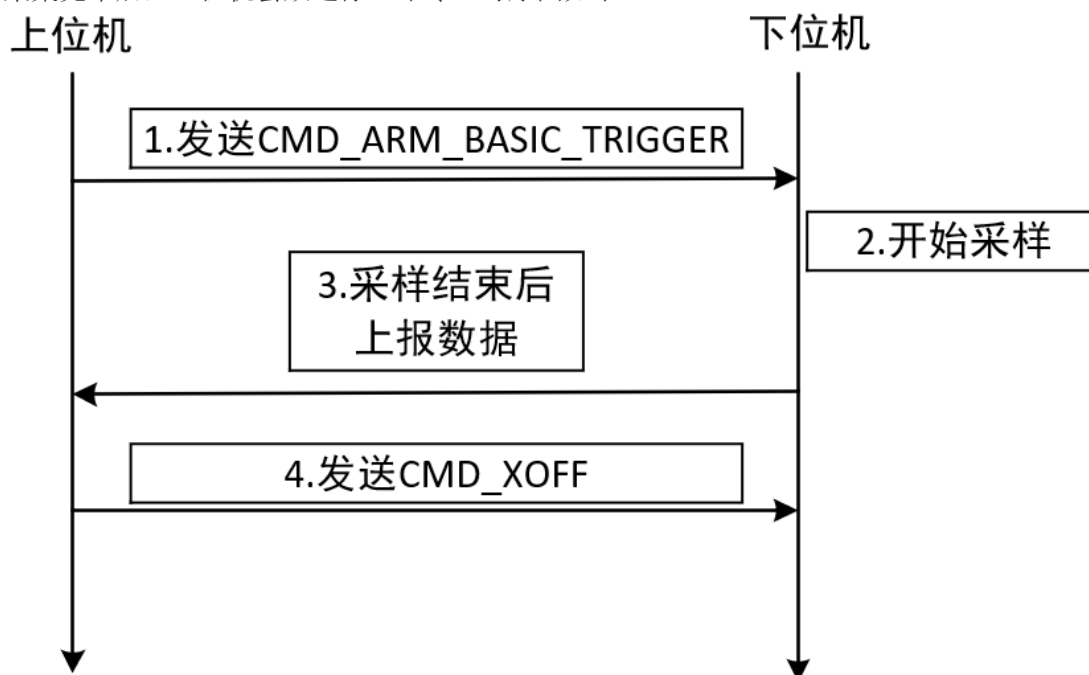
```
// 上位机 ols_prepare_acquisition函数
sigrok-util_src_patched\libsigrok\src\hardware\openbench-logic-sniffer\protocol.c,
```

分析它的代码，得到如下时序图：



### 2.3.4 采样操作的时序图

当用户点击如下按钮后，上位机会先设置逻辑分析仪，最后发送启动命令开始采集数据；采集完毕后，上位机会放送停止命令。时序图如下：



## 第3章 实现逻辑分析仪

### 3.1 软件设计方案

商用的逻辑分析仪一般使用 FPGA 采集数据，并且有比较大的内存（比如 512MB）。FPGA 可以使用非常高的速率采集数据，然后存放在内存里。

对于单片机，内存小，速度慢，我们需要压榨它的性能。

#### 3.1.1 如何实现最高的采样率

逻辑分析仪采样数据的示例代码如下：

```
// 1. 关闭中断
// 2. 循环
while (1)
{
    // 2.1 读GPIO
    // 2.2 写buffer
    // 2.3 延时
}
// 3. 开中断
```

为了达到最高的采样率，循环里面的“读 GPIO”、“写 buffer”操作使用汇编指令；循环里的“延时”代码要去除。在这种情况下，我们再去测量一次循环的耗时，就可以算出最高采样率。

当使用更低的采样率时，在上述代码里插入“耗时”操作。

这都需要我们事先知道“读 GPIO”、“写 buffer”、“延时指令”的耗时。

#### 3.1.2 汇编指令

要能深刻理解汇编指令，需要先学习《ARM 架构与编程\_基于 STM32F103》的课程。

为了简化程序，我们只使用 PB8~PB15 这 8 个引脚。要去读取这几个引脚的数值，需要读取 GPIOB\_IDR 寄存器。这个寄存器的说明在“2\_官方资料\2.0\_STM32F103xx 参考手册(英文原版)【重要】.pdf”里，寄存器的 bit8~15 分别对应 PB8~PB15 引脚，对应上位机的 channel0~7。寄存器如下：

##### 9.2.3 Port input data register (GPIOx\_IDR) (x=A..G)

Address offset: 0x08h

GPIOB\_IDR寄存器地址为: 0x40010C08

Reset value: 0x0000 XXXX

0x4001 0C00 - 0x4001 0FFF GPIO Port B

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **IDRy**: Port input data (y= 0 .. 15)

These bits are read only and can be accessed in Word mode only. They contain the input value of the corresponding I/O port.

读 GPIO 的汇编指令代码为：

```
LDR R1, =0x40010C08
```

```
LDR R0, [R1]
```

写 buffer 的汇编指令代码为：

```
LDR R1, =buffer地址
```

```
LDR R0, [R1]
```

在汇编里，我们可以在循环之间插入几条 NOP 指令来实现延时，比如：

```
NOP
```

```
NOP
```

### 3.1.3 精确测量时间

先列出结果，精确测量的时间列表如下：

操作	汇编指令	耗时
读取 GPIO	//R1 为 0x40010C08 LDR R0, [R1]	44ns
读内存	//R1 为 0x20000000 LDR R0, [R1]	15ns
写内存	//R1 为 0x20000000 STRB R0, [R1]	16ns
NOP 指令	NOP	15ns
逻辑右移	LSR R0, #8	24ns
累加	ADD R0, #1	23ns
Tick 中断处理		10us

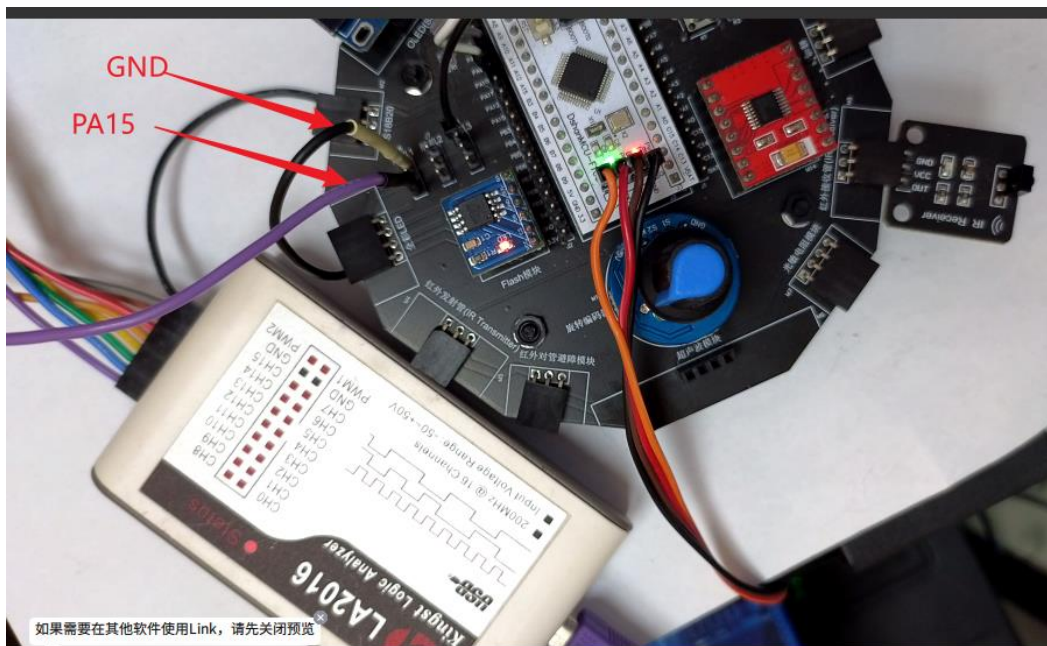
我们需要测量读一次 GPIO 的精确时间，写一次 buffer 的精确时间，一条 NOP 指令执行的时间。

示例代码如下：

```
// 1. 关闭中断
{
    // 2.1 设置某个引脚为高电平
    // 2.2 循环100万次读GPIO，或写Buffer，或执行NOP指令
    // 2.3 设置某个引脚为低电平
}
// 3. 开中断
```

我们借助外部工具监测 GPIO 引脚为高电平的时间，就可以算出每次操作的耗时。可以使用逻辑分析仪来测量 GPIO 为高电平的时间。

逻辑分析仪测量 PA15，如下图接线：





## 1. 测量读 GPIO 操作的时间

本节源码放在“7\_逻辑分析仪项目\2\_下位机资料\02\_视频配套的源码\3-2\_精确测量读 GPIO 的时间”目录下。在“7\_逻辑分析仪项目\2\_下位机资料\02\_视频配套的源码\1-1\_逻辑分析仪项目课程概述与体验”的基础上修改得来。

测试函数的代码为：

```
364: void MeasureTime(void)
365: {
366:     /* 0. 让引脚输出低电平 */
367:     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_15, GPIO_PIN_RESET);
368:     HAL_Delay(100);
369:
370:     /* 1. 关中断 */
371:     __disable_irq();
372:
373:     /* 2.1 让引脚输出高电平 */
374:     //HAL_GPIO_WritePin(GPIOA, GPIO_PIN_15, GPIO_PIN_SET);
375:
376:     /* 2.2 执行汇编指令 */
377:     //for (int i = 0; i < 10000; i++)
378:         asm_measure();
379:
380:     /* 2.3 让引脚输出低电平 */
381:     //HAL_GPIO_WritePin(GPIOA, GPIO_PIN_15, GPIO_PIN_RESET);
382:
383:     /* 3. 开中断 */
384:     __enable_irq();
385: } « end MeasureTime »
```

在汇编里读 GPIO 的代码为：

```

2:                                THUMB
3:                                AREA |.text|, CODE, READONLY
4:
5: ; asm_measure handler
6: asm_measure PROC
7:                                EXPORT asm_measure
8:                                ; 设置PA15输出高电平
9:                                LDR R1, =0X40010810
10:                               LDR R0, =(1<<15)
11:                               STR R0, [R1]
12:
13:                               LDR R1, =0x40010C08
14:                               LDR R0, [R1] ; 读GPIOB_IDR
15:                               LDR R0, [R1] ; 读GPIOB_IDR
16:                               LDR R0, [R1] ; 读GPIOB_IDR
17:                               LDR R0, [R1] ; 读GPIOB_IDR
18:                               LDR R0, [R1] ; 读GPIOB_IDR
19:                               LDR R0, [R1] ; 读GPIOB_IDR
20:                               LDR R0, [R1] ; 读GPIOB_IDR
21:                               LDR R0, [R1] ; 读GPIOB_IDR
112:                              LDR R0, [R1] ; 读GPIOB_IDR
113:                              LDR R0, [R1] ; 读GPIOB_IDR
114:
115:                               ; 设置PA15输出低电平
116:                               LDR R1, =0X40010810
117:                               LDR R0, =(1<<31)
118:                               STR R0, [R1]
119:
120:                               BX LR
121:                                ENDP

```

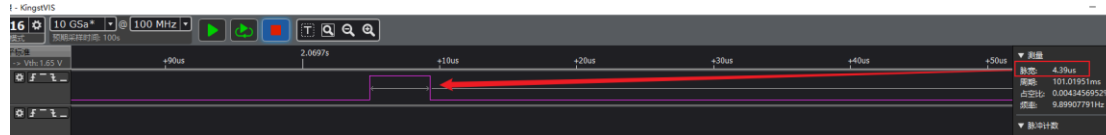
1.使用汇编让PA15输出高电平  
供测量

2. 读GPIO  
手写100条指令

3.使用汇编让PA15输出低电平  
供测量

4. 返回

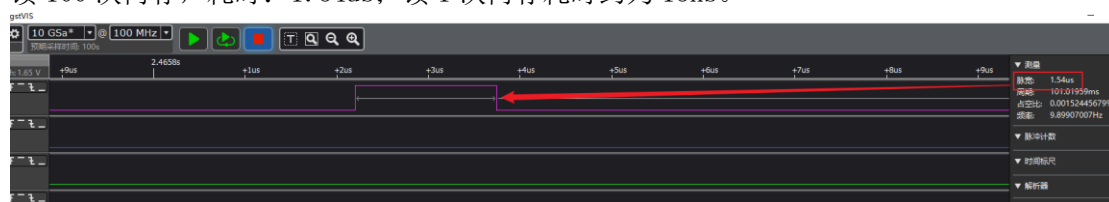
读 100 次 GPIO，耗时：4.39us；读 1 次 GPIO 耗时约为 44ns。



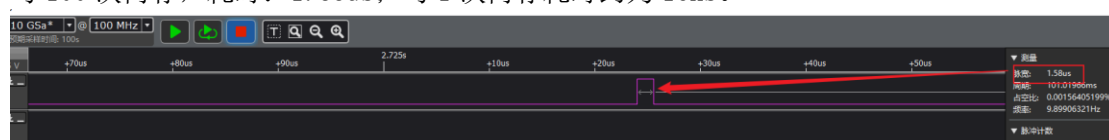
## 2. 测量读写 buffer 的时间

本节源码放在“7\_逻辑分析仪项目\2\_下位机资料\02\_视频配套的源码\3-3\_精确测量其他操作的时间”目录下。

读 100 次内存，耗时：1.54us；读 1 次内存耗时约为 15ns。

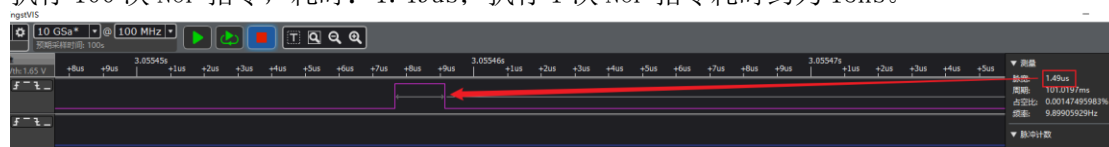


写 100 次内存，耗时：1.58us；写 1 次内存耗时约为 16ns。



## 3. 测量 NOP 指令的时间

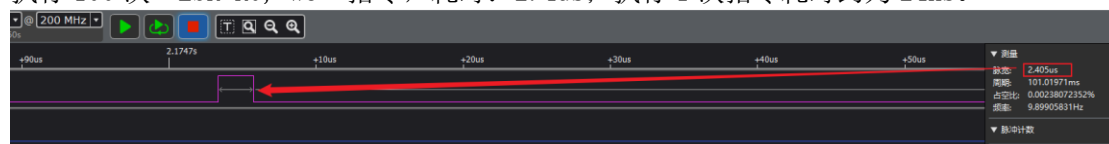
执行 100 次 NOP 指令，耗时：1.49us；执行 1 次 NOP 指令耗时约为 15ns。



## 4. 逻辑右移

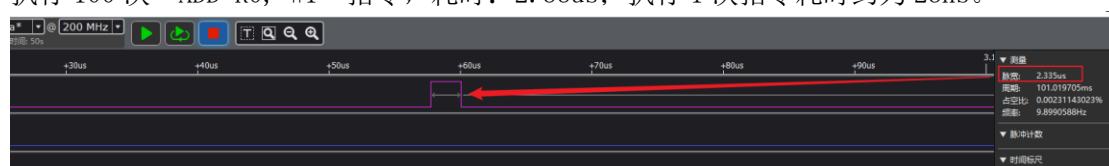
LSR Rd, Rn	; Rd=Rd>>Rn	逻辑右移
LSR Rd, Rn, Rm	; Rd=Rn>>Rm	

执行 100 次“LSR R0, #8”指令，耗时：2.4us；执行 1 次指令耗时约为 24ns。



## 5. 加法操作

执行 100 次“ADD R0, #1”指令，耗时：2.33us；执行 1 次指令耗时约为 23ns。



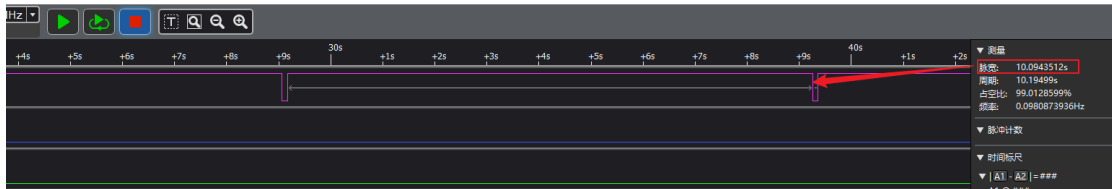
## 6. 测量处理 Tick 中断函数的时间

测量关闭中断情况下一段代码的执行时间（约为 10s）：

```

369:
370: void MeasureTime(void)
371: {
372:     /* 0. 让引脚输出低电平 */
373:     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_15, GPIO_PIN_RESET);
374:     HAL_Delay(100);
375:
376:     /* 1. 关中断 */
377:     __disable_irq();                关中断情况下10.09s
378:
379:     /* 2.1 让引脚输出高电平 */
380:     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_15, GPIO_PIN_SET);
381:
382:     /* 2.2 执行汇编指令 */
383:     for (int i = 0; i < 2277904; i++) /* 10s */
384:         asm_measure();
385:
386:     /* 2.3 让引脚输出低电平 */
387:     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_15, GPIO_PIN_RESET);
388:
389:     /* 3. 开中断 */
390:     __enable_irq();
391: } « end MeasureTime »

```

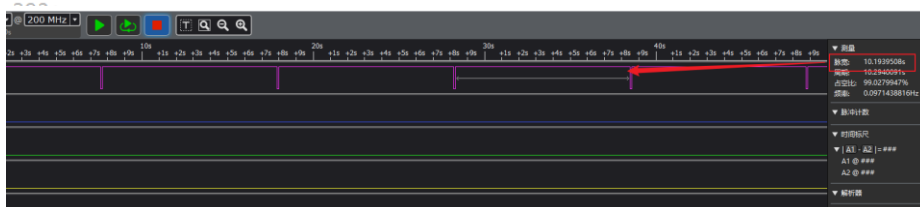


测量开中断情况下，同一段代码的执行时间：

```

370: void MeasureTime(void)
371: {
372:     /* 0. 让引脚输出低电平 */
373:     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_15, GPIO_PIN_RESET);
374:     HAL_Delay(100);
375:
376:     /* 1. 关中断 */
377:     // __disable_irq();                开中断情况下耗时10.1939s
378:
379:     /* 2.1 让引脚输出高电平 */
380:     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_15, GPIO_PIN_SET);
381:
382:     /* 2.2 执行汇编指令 */
383:     for (int i = 0; i < 2277904; i++) /* 10s */
384:         asm_measure();
385:
386:     /* 2.3 让引脚输出低电平 */
387:     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_15, GPIO_PIN_RESET);
388:
389:     /* 3. 开中断 */
390:     // __enable_irq();
391: } « end MeasureTime »

```



相差:  $10.1939508 - 10.0943512 = 0.0995996\text{s}$ ，对应 10000 次 tick 中断，每次 tick 中断耗时:  $0.0995996/10000=0.00995996\text{ms}=9.95996\mu\text{s}$ ，约为 10us。

### 3.2 实现功能

#### 3.2.1 方案修订

精确测量的时间列表如下：

操作	汇编指令	耗时
读取 GPIO	//R1 为 0x40010C08 LDR R0, [R1]	44ns
读内存	//R1 为 0x20000000 LDR R0, [R1]	15ns
写内存	//R1 为 0x20000000 STRB R0, [R1]	16ns
NOP 指令	NOP	15ns
逻辑右移	LSR R0, #8	24ns
累加	ADD R0, #1	23ns
Tick 中断处理		10us

对于如下代码：

```
// 1. 关闭中断
// 2. 循环
while (1)
{
    // 2.1 读GPIO、逻辑右移
    // 2.2 写buffer、累加地址
    // 2.3 延时
}
// 3. 开中断
```

去掉延时，循环一次耗时 44+24+16+23=107ns，理论上最高的采样频率=1/107ns=9MHz。而 STM32F103C8 的内存为 20K，即使全部用来保存采样的数据，也只能保存 20\*1024/9000000=0.002 秒，没有任何实用价值。

即使降低采样频率，比如降到 100KHz（I2C 频率一般为 100KHz，再低的话就没有实用价值了），20K 内存全部用来保存采样数据，能保存 20\*1024/100000=0.2048 秒，也没有什么使用价值。

瓶颈在于：用来保存采样数据的内存太小了。看看商用的逻辑分析仪，它的内存是巨大的：

参数信息

型号	LA2016	品牌	KINGST
产地	中国大陆	适用场景	电子开发测量
输入方式	数字	存储容量	2GB
频宽	40MHz	记录模式	buffer
存储深度 (bits/C H)	100MB		

在有限的内存里，我们需要提高内存的使用效率：不变的数据就不要保存了。新方案如下：

- ① 定义两个数组：uint8\_t data\_buf[5000]、uint8\_t cnt\_buf[5000]
- ① 以比较高的、频率周期性地读取 GPIO 的值
- ② 只有 GPIO 值发生变化了，才存入 data\_buf[i++]；GPIO 值无变化时，cnt\_buf[i-1]累

加

③ 以后，根据 data\_buf、cnt\_buf 恢复各个采样点的数据，上报给上位机

假设 data\_buf 大小为 5000，能记录 5000 个变化的数据，这足够我们日常使用了。

其他考虑：使用新方案后，能记录很长时间的数据，在程序运行期间，要判断是否收到“上位机发来的 CMD\_XOFF 停止命令”，所以：串口接收中断要打开。

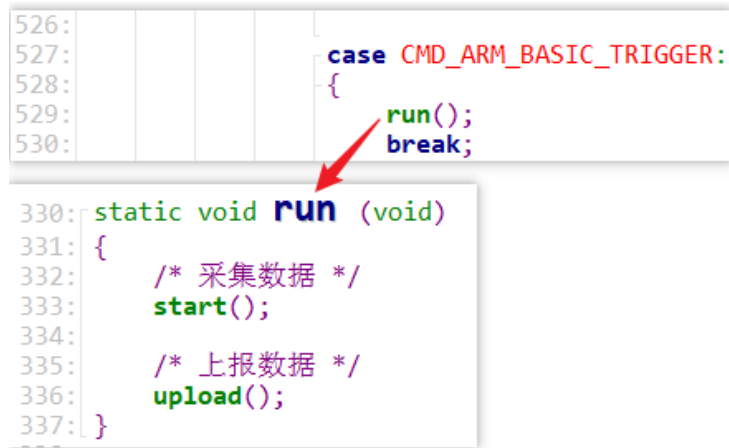
### 3.2.2 编写程序

本节源码放在“7\_逻辑分析仪项目\2\_下位机资料\02\_视频配套的源码\3-5\_实现功能”目录下。在“7\_逻辑分析仪项目\2\_下位机资料\02\_视频配套的源码\3-3\_精确测量其他操作的时间”的基础上修改得来。

核心代码为“Core\Src\logicanalyzer.c”，主要有 2 大功能：

- ① 采集数据：读取 GPIO 数据、保存起来
- ② 上报数据：把数据发给上位机

当下位机收到“CMD\_ARM\_BASIC\_TRIGGER”命令后，启动采集、上报：



```
526:
527:
528:
529:
530:
case CMD_ARM_BASIC_TRIGGER:
{
    run();
    break;
}

330: static void run (void)
331: {
332:     /* 采集数据 */
333:     start();
334:
335:     /* 上报数据 */
336:     upload();
337: }
```

The image shows a code editor with two snippets. The top snippet is a switch case for CMD\_ARM\_BASIC\_TRIGGER, which calls run() and then breaks. The bottom snippet is the definition of the run function, which calls start() and upload(). A red arrow points from the run() call in the top snippet to the run function definition in the bottom snippet.

#### 1. 采集数据

start 函数采集数据，功能为：

- ① 禁止中断：这是为了在采集数据时以最快的频率采集，不让中断干扰。  
除了串口中断之外，其他中断都禁止。下位机只有 tick 中断、串口中断，所以只需要禁止 tick 中断。  
保留串口中断的原因在于：上位机可能发来命令停止采样。
- ② 等待触发条件：用户可能设置触发采样的条件
- ③ 触发条件满足后，延时一会：没有必要
- ④ 循环：以最高频率采样  
退出的条件有三：收到上位机发来的停止命令、采集完毕、数据 buffer 已经满
- ⑤ 恢复中断

关键功能是采集、记录数据：

```

223:    /* 记录第1个数据 */
224:    data = (*data_reg) >> 8;
225:    g_rxdata_buf[0] = data;
226:    g_rxcnt_buf[0] = 1;
227:    g_cur_sample_cnt = 1;
228:    pre_data = data;
229:
230:    /* 4. 以最高的频率采集数据 */
231:    while (1)
232:    {
233:        *pa15_reg = (1<<15); /* PA15输出高电平 */
234:
235:        /* 4.1 读取数据 */
236:        data = (*data_reg) >> 8;
237:
238:        /* 4.2 保存数据 */
239:        g_cur_pos += (data != pre_data)? 1 : 0; /* 数据不变的话,写位置不变 */
240:        g_rxdata_buf[g_cur_pos] = data; /* 保存数据 */
241:        g_rxcnt_buf[g_cur_pos]++; /* 增加"相同的数据"个数 */
242:        g_cur_sample_cnt++; /* 累加采样个数 */
243:        pre_data = data;
244:
245:        /* 4.3 串口收到停止命令 */
246:        if (get_stop_cmd)
247:            break;
248:
249:        /* 4.4 采集完毕? */
250:        if (g_cur_sample_cnt >= converted_sample_count)
251:            break;
252:
253:        /* 4.5 buffer满? */
254:        if (g_cur_pos >= BUFFER_SIZE)
255:            break;
256:
257:        /* 4.6 加入这些延时凑出1MHz,加入多少个nop需要使用示波器或逻辑分析仪观察、调整 */
258:        __asm volatile( "nop" );
259:        __asm volatile( "nop" );
260:        __asm volatile( "nop" );
261:        __asm volatile( "nop" );
262:        __asm volatile( "nop" );
263:        __asm volatile( "nop" );
264:        __asm volatile( "nop" );
265:        __asm volatile( "nop" );
266:        __asm volatile( "nop" );
267:        __asm volatile( "nop" );
268:        __asm volatile( "nop" );
269:        __asm volatile( "nop" );
270:        __asm volatile( "nop" );
271:        __asm volatile( "nop" );
272:        __asm volatile( "nop" );
273:        __asm volatile( "nop" );
274:        __asm volatile( "nop" );
275:        __asm volatile( "nop" );
276:        __asm volatile( "nop" );
277:        __asm volatile( "nop" );
278:        __asm volatile( "nop" );
279:        __asm volatile( "nop" );
280:        __asm volatile( "nop" );
281:        __asm volatile( "nop" );
282:        __asm volatile( "nop" );
283:
284:        *pa15_reg = (1UL<<31); /* PA15输出低电平 */
285:    } « end while 1 »

```

## 2. 上报数据

采集数据时是以最大频率采集的,比如以 1MHz 采集。如果上位机要求的采样频率是 200KHz:  $1\text{MHz}/200\text{KHz}=5$ , 采集到的数据量是上报数据量的 5 倍。我们只需要每隔 5 个数据上报一个即可。

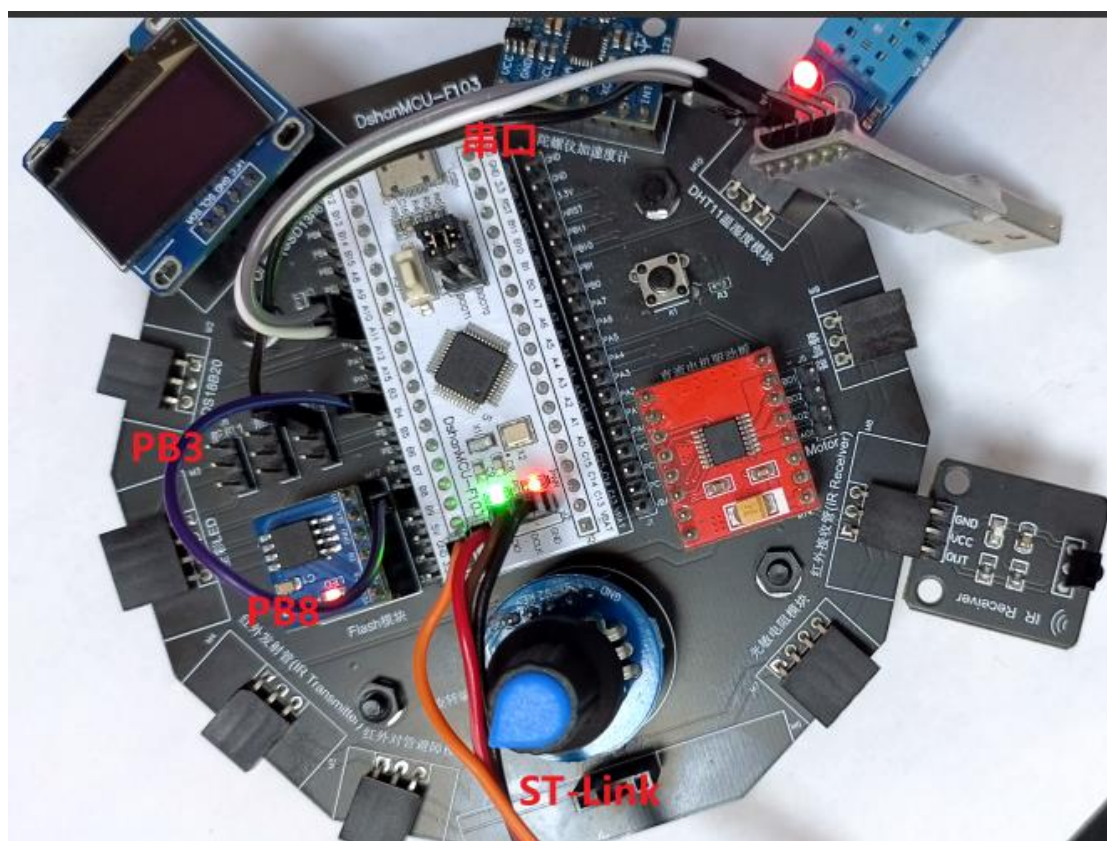
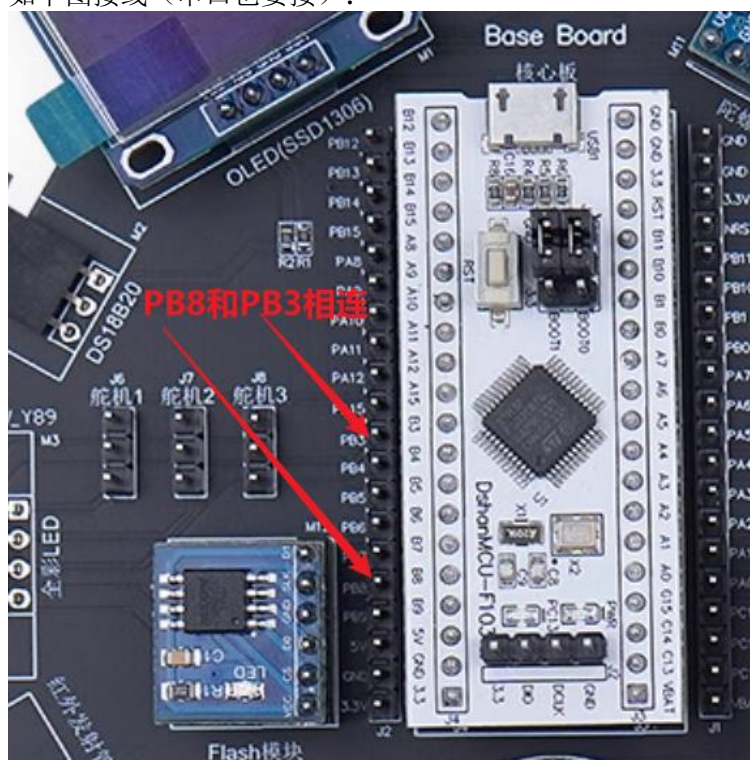
```
291: /*****
292:  * 函数名称:  upload
293:  * 功能描述:  上报数据
294:  * 输入参数:  无
295:  * 输出参数:  无
296:  * 返回 值:  无
297:  * 修改日期:  版本号  修改人    修改内容
298:  * -----
299:  * 2024/07/04      V1.0    韦东山    创建
300: *****/
301: static void upload (void)
302: {
303:     int32_t i = g_cur_pos;
304:     uint32_t j;
305:     uint32_t rate = MAX_FREQUENCY / g_samplingRate;
306:     int cnt = 0;
307:
308:     for (; i >= 0; i--)
309:     {
310:         for (j = 0; j < g_rxcnt_buf[i]; j++)
311:         {
312:             cnt++;
313:             /* 我们以最大频率采样，假设最大频率是1MHz
314:              * 上位机想以200KHz的频率采样
315:              * 那么在得到的数据里，每5个里只需要上报1个
316:              */
317:             if (cnt == rate)
318:             {
319:                 uart_send(&g_rxdata_buf[i], 1, 10);
320:                 cnt = 0;
321:             }
322:         }
323:     }
324: } « end upload »
```



### 3.2.3 上机演示

为了采集数据，设置下位机的 PB3 输出周期为 1ms、占空比为 50% 的 PWM 波。让 PB8 连接到 PB3，PB8 是 channel 0，就可以再上位机观察 channel 0 的波形。

如下图接线（串口也要接）：



烧录“7\_逻辑分析仪项目\2\_下位机资料\02\_视频配套的源码\3-5\_实现功能”工程。

### 3.3 改进功能

本节源码放在“7\_逻辑分析仪项目\2\_下位机资料\02\_视频配套的源码\3-6\_改进功能”目录下。在“7\_逻辑分析仪项目\2\_下位机资料\02\_视频配套的源码\3-5\_实现功能”的基础上修改得来。

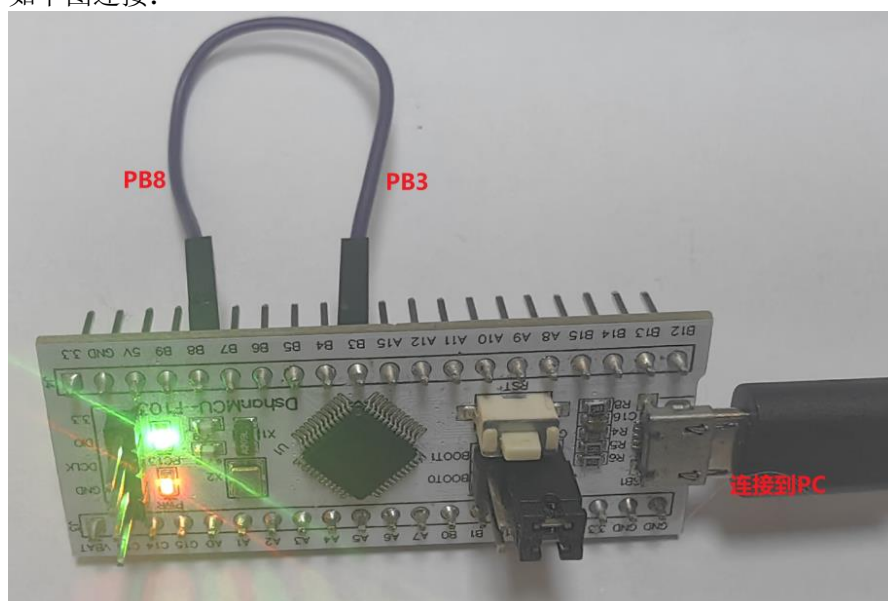
### 3.3.1 使用说明

先烧录程序。

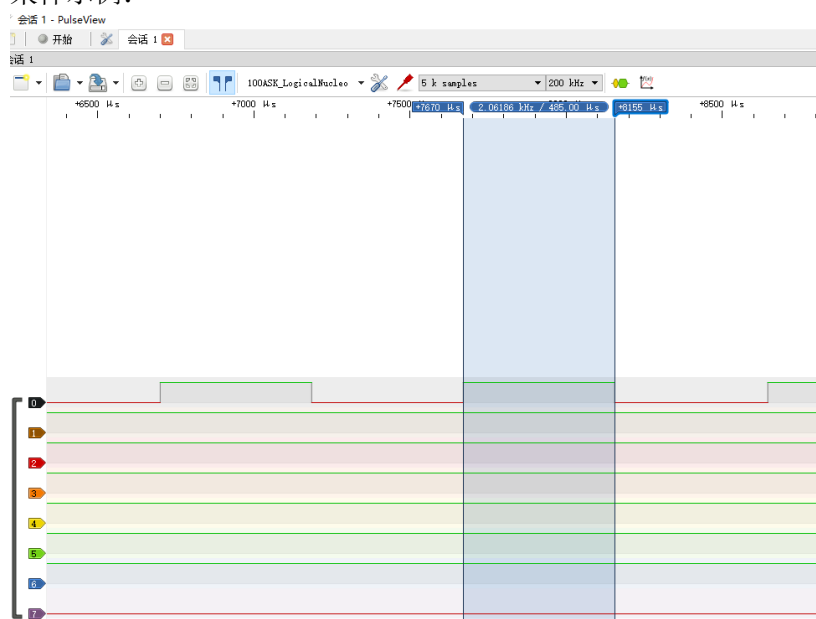
PB8~PB15 是通道 0~7, 可以用来连接要测试的引脚。注意: 测量其他电子设备时要共地。

PB3 输出周期 1ms、50%占空比的 PWM 波，可以用来验证功能是否正常。

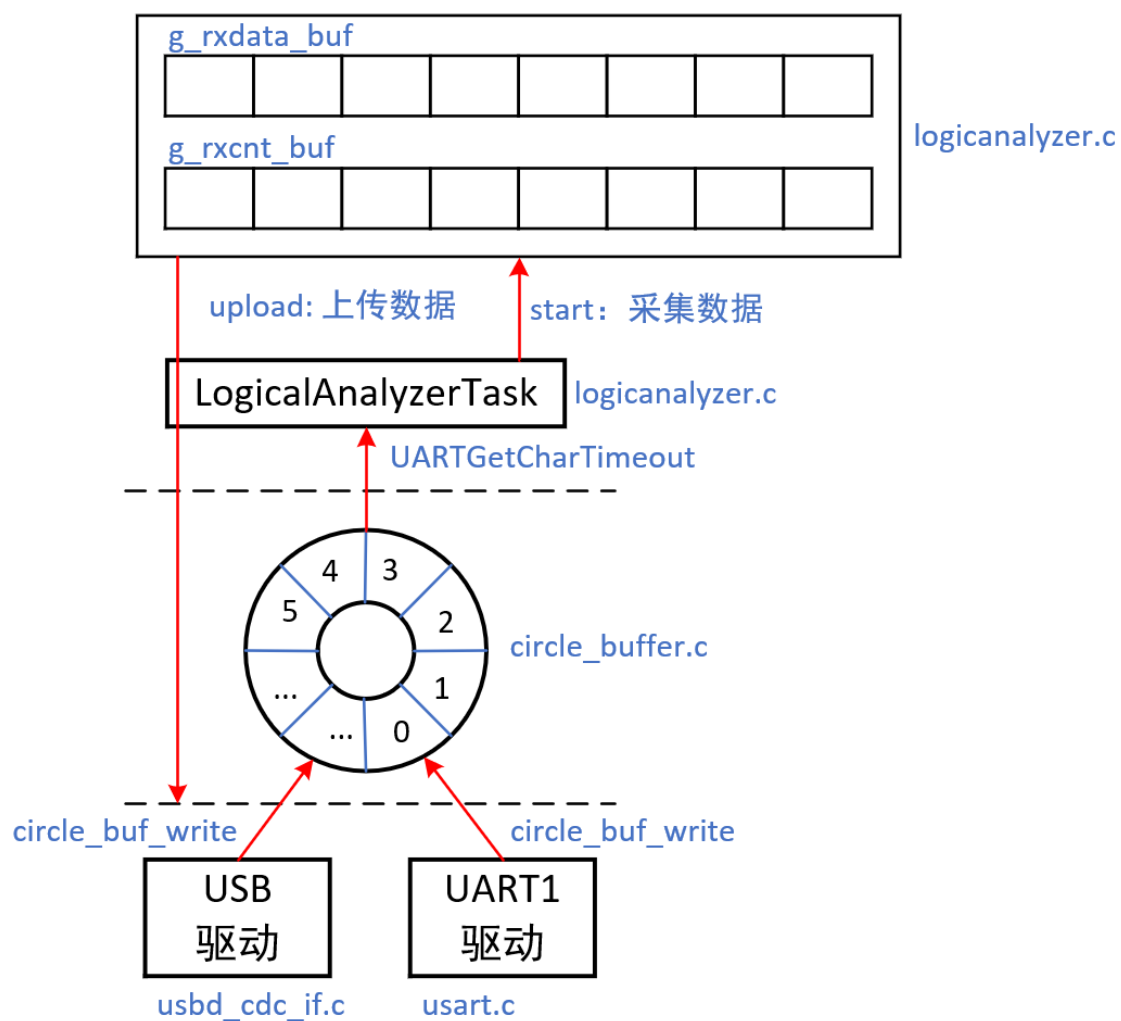
如下图连接:



采样示例:



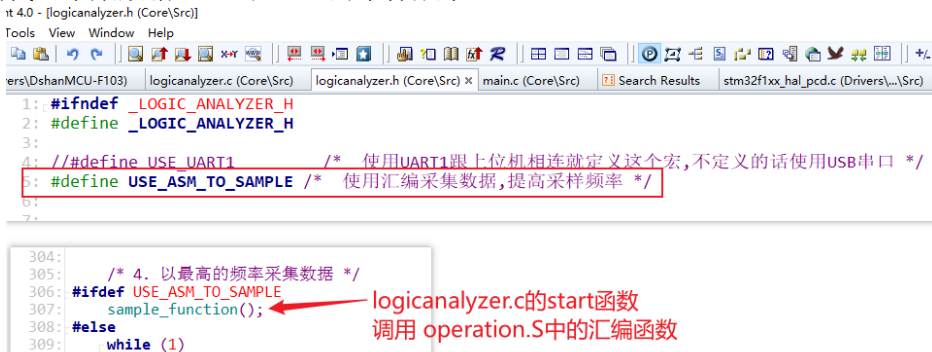
## 3.3.2 最终程序的结构



### 3.3.3 提高采样率

使用汇编采集数据，把最大采样频率提高一倍，达到 2MHz。

在工程的“Core\Src\logicanalyzer.h”中定义“USE\_ASM\_TO\_SAMPLE”，就可以使用汇编代码采集数据，达到 2MHz 的采样频率。



汇编代码在“Core\Src\operation.S”中，如下：

```
BUFFER_SIZE equ 2700
```

```
THUMB
AREA |.text|, CODE, READONLY
```

```
; sample_function handler
```

```
sample_function PROC
    EXPORT sample_function
    IMPORT g_rxdata_buf
    IMPORT g_rxcnt_buf
    IMPORT g_cur_pos
    IMPORT g_cur_sample_cnt
    IMPORT get_stop_cmd
    IMPORT g_convreted_sample_count
```

```

PUSH    {R4, R5, R6, R7, R8, R9, R10, R11, R12, LR}
LDR R0, =g_rxdata_buf ; 得到这些变量的地址,并不是得到它们的值
LDR R1, =g_rxcnt_buf ; 得到g_rxcnt_buf变量的地址,并不是得到它的值
LDR R2, =g_cur_pos ; 得到g_cur_pos变量的地址,并不是得到它的值
LDR R2, [R2] ; 得到g_cur_pos变量的值
LDR R3, =g_cur_sample_cnt
LDR R3, [R3]
LDR R4, =get_stop_cmd
LDR R5, =g_convreted_sample_count
LDR R5, [R5]

```

```

LDR R8, [R0] ; pre_data
LDR R10, =BUFFER_SIZE

```

```
LDR R6, =0x40010C08
```

```

    ; 设置PA15的值备用
    LDR R11, =0X40010810
    LDR R12, =(1<<15)
    LDR LR, =(1<<31)
Loop
    ; 设置PA15输出高电平
    STR R12, [R11]

    LDRH R7, [R6] ; 读GPIOB_IDR
    LSR R7, #8 ; data = (*data_reg) >> 8;
    CMP R7, R8
    ADDNE R2, #1 ; g_cur_pos += (data != pre_data)? 1 : 0;
    STRB R7, [R0, R2] ; g_rxdata_buf[g_cur_pos] = data;
    MOV R8, R7 ; pre_data = data
    LDR R7, [R1, R2, LSL #2] ; R7 = g_rxcnt_buf[g_cur_pos]
    ADD R7, #1
    STR R7, [R1, R2, LSL #2] ; g_rxcnt_buf[g_cur_pos]++;
    ADD R3, #1 ; g_cur_sample_cnt++;

    CMP R3, R5 ; if (g_cur_sample_cnt >= g_convreted_sample_count) break;
    BGE LoopDone

    LDR R7, [R4] ; R7 = get_stop_cmd
    CMP R7, #0 ; if (get_stop_cmd) break;
    BNE LoopDone

    CMP R2, R10 ; if (g_cur_pos >= BUFFER_SIZE) break;
    BGE LoopDone

    NOP
    NOP ; 延时, 凑出2MHz

    ; 设置PA15输出高电平
    STR LR, [R11]

    B Loop

LoopDone
    LDR R0, =g_cur_pos ; 得到g_cur_pos变量的地址, 并不是得到它的值
    STR R2, [R0] ; 保存g_cur_pos变量的值
    LDR R0, =g_cur_sample_cnt
    STR R3, [R0] ; 保存g_cur_sample_cnt变量的值

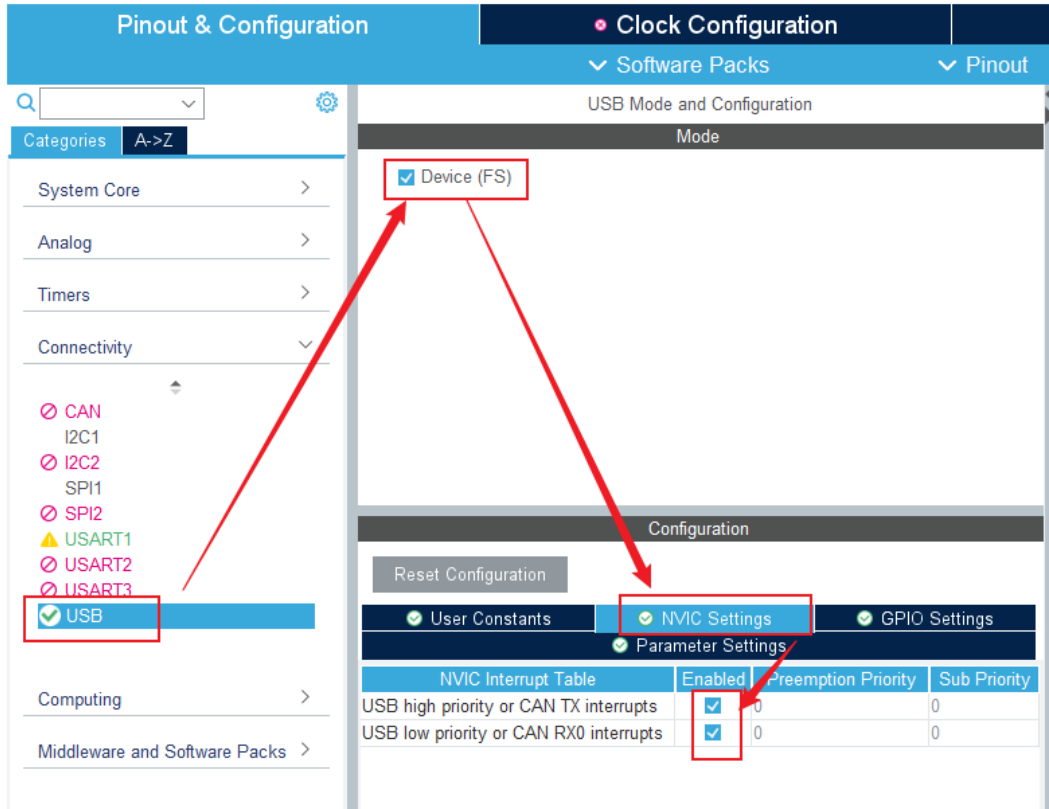
```

POP	{R4, R5, R6, R7, R8, R9, R10, R11, R12, PC}
ENDP	

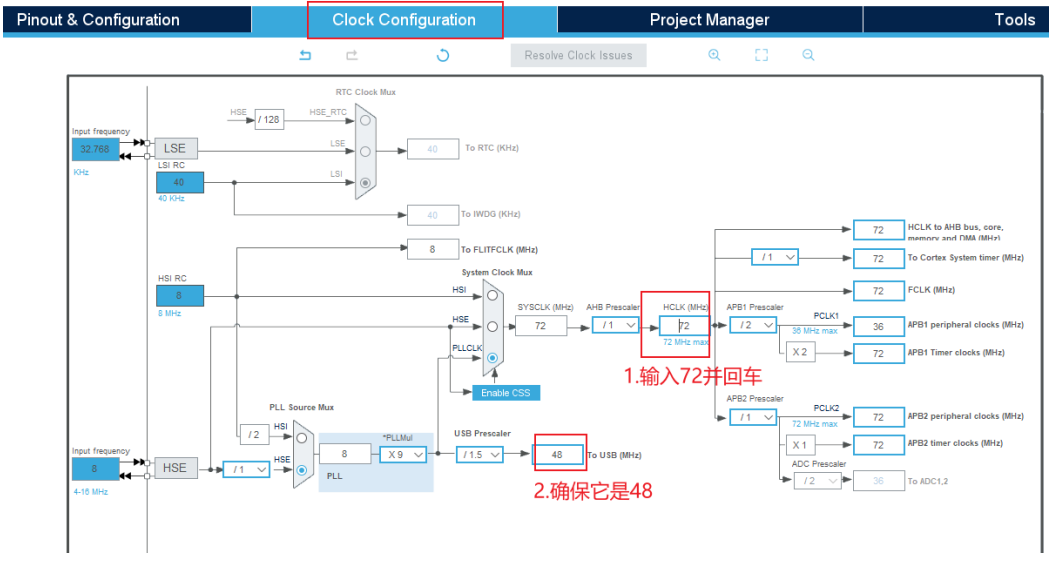
3.3.4 增加改进 USB 串口功能

1. 在 STM32CubeMX 增加 USB 串口功能

第 1 步：使能 USB 功能，如下图操作：

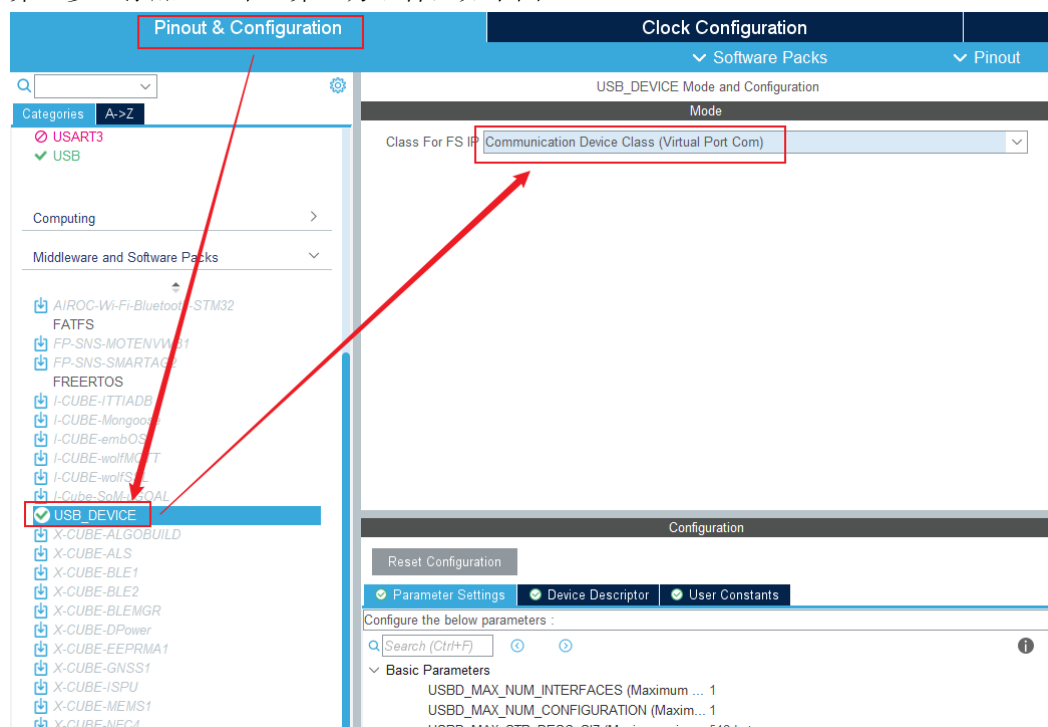


第 2 步：设置时钟，确保 CPU 频率为最大的 72MHz，USB 频率为 48MHz，如下图：

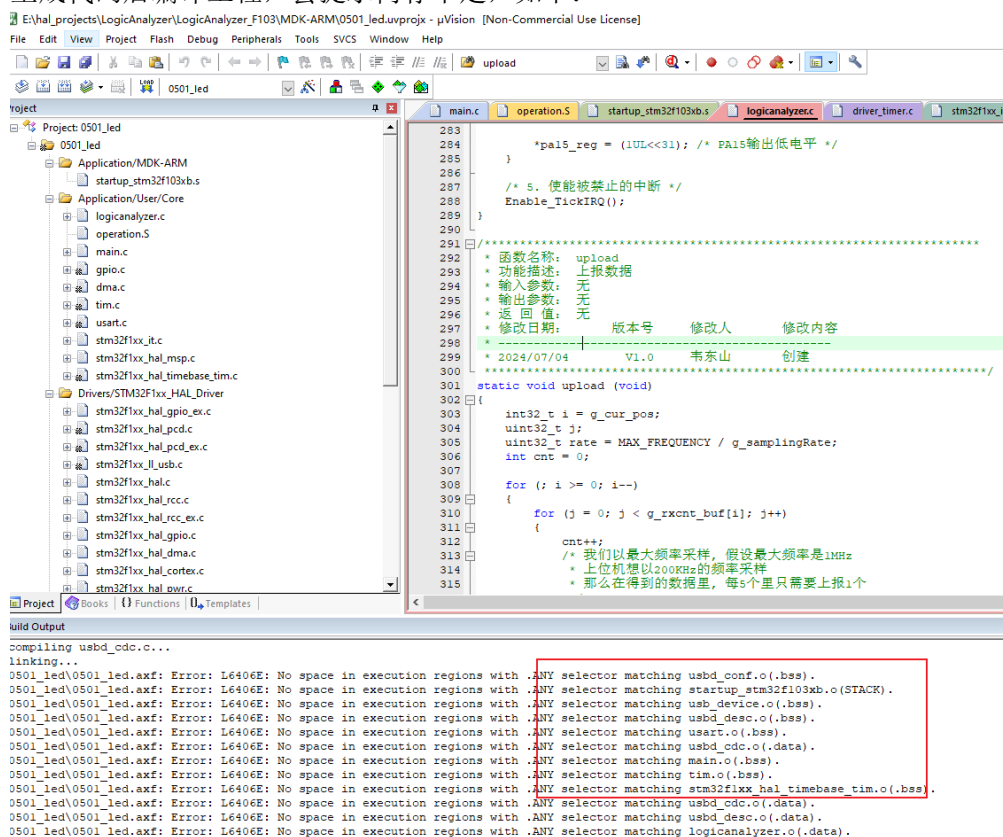




第 3 步：添加 USB 串口第 3 方组件，如下图：



生成代码后编译工程，会提示内存不足，如下：





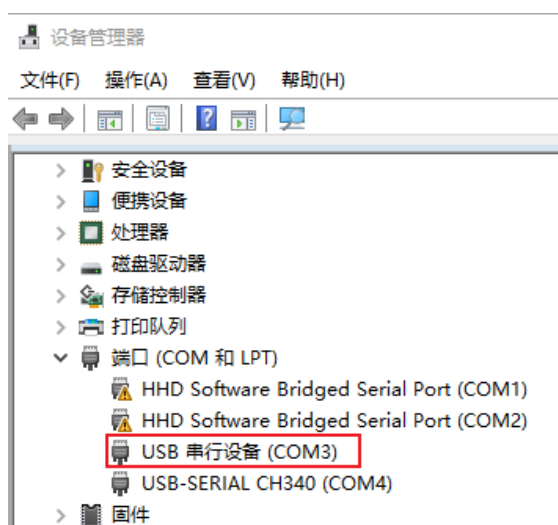
修改如下代码，把 BUFFER\_SIZE 减小为 2700 或更小：

```

: 4.0 - [logicanalyzer.c (Core\Src)]
ools View Window Help
rs\DshanMCU-F103) logicanalyzer.c (Core\Src) logicanalyzer.h (Core\Src) main.c (Core\Src) operation.S (Core\Src) x
36: }
37:
38:
39: /*****
40:
41: #define BUFFER_SIZE 2700 /* 注意这个数值要跟operation.S中的BUFFER_SIZE保持一致 */
42:

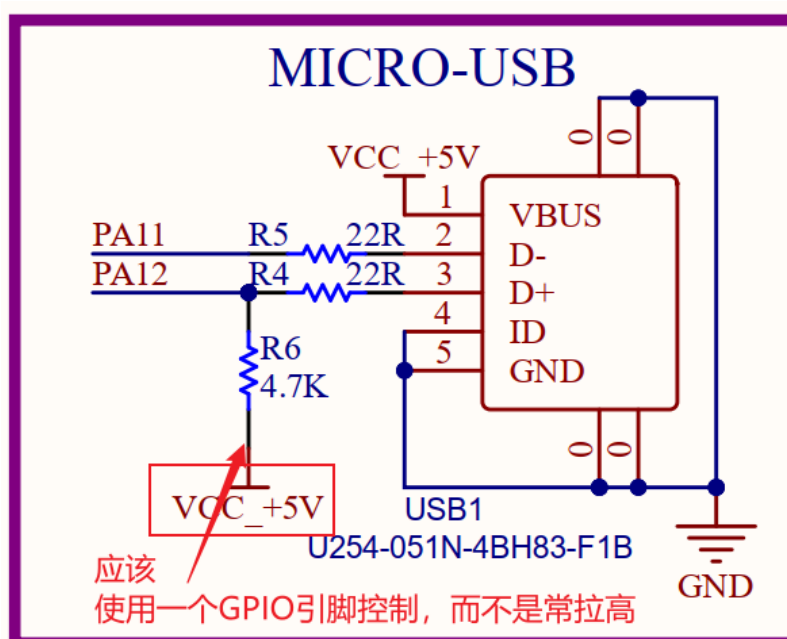
```

烧录程序后，使用 USB 线连接开发板和 PC，可以在 PC 的设备管理器看到新的串口设备：



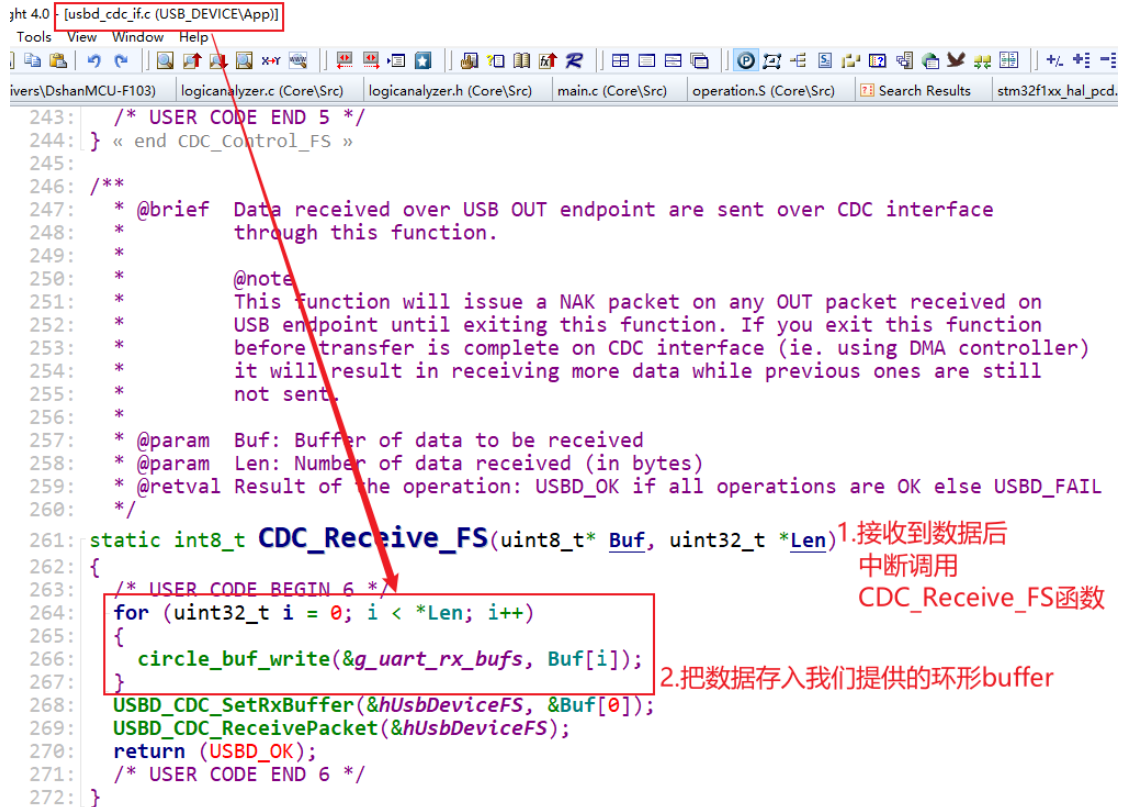
注意：由于硬件设计问题，每次烧写程序后都要重新接插 USB 线。要调试 USB 串口功能的话，每次启动调试之后也要重新接插 USB 线。

硬件问题在于 USB 口的使能引脚常拉高，导致无法让 PC 重新识别 USB 设备（只能重插）：



## 2. USB 串口收发函数改造

当下位机通过 USB 口接收到数据时，它的如下函数被调用：



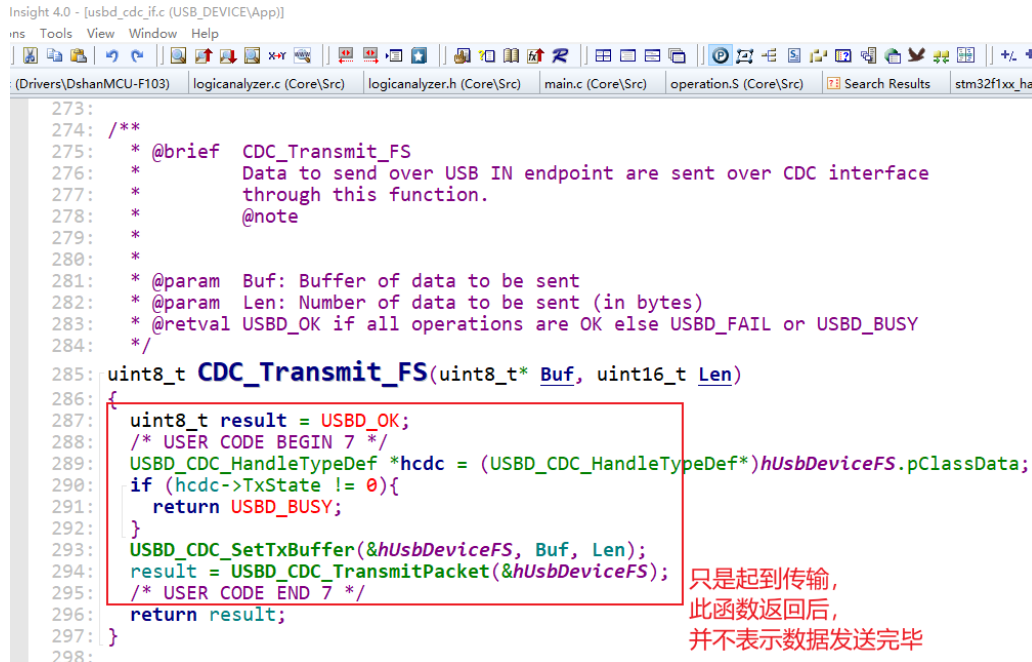
```

243:  /* USER CODE END 5 */
244:  } « end CDC_Control_FS »
245:
246:  /**
247:   * @brief Data received over USB OUT endpoint are sent over CDC interface
248:   *        through this function.
249:   *
250:   * @note
251:   *       This function will issue a NAK packet on any OUT packet received on
252:   *       USB endpoint until exiting this function. If you exit this function
253:   *       before transfer is complete on CDC interface (ie. using DMA controller)
254:   *       it will result in receiving more data while previous ones are still
255:   *       not sent.
256:   *
257:   * @param Buf: Buffer of data to be received
258:   * @param Len: Number of data received (in bytes)
259:   * @retval Result of the operation: USB_OK if all operations are OK else USB_FAIL
260:   */
261: static int8_t CDC_Receive_FS(uint8_t* Buf, uint32_t *Len)
262: {
263:     /* USER CODE BEGIN 6 */
264:     for (uint32_t i = 0; i < *Len; i++)
265:     {
266:         circle_buf_write(&g_uart_rx_bufs, Buf[i]);
267:     }
268:     USBDCDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);
269:     USBDCDC_ReceivePacket(&hUsbDeviceFS);
270:     return (USB_OK);
271:     /* USER CODE END 6 */
272: }
  
```

1.接收到数据后  
中断调用  
CDC\_Receive\_FS函数

2.把数据存入我们提供的环形buffer

当下位机想通过 USB 口发送数据时，使用如下函数：



```

273:
274:  /**
275:   * @brief CDC_Transmit_FS
276:   *        Data to send over USB IN endpoint are sent over CDC interface
277:   *        through this function.
278:   *
279:   * @note
280:   *
281:   * @param Buf: Buffer of data to be sent
282:   * @param Len: Number of data to be sent (in bytes)
283:   * @retval USB_OK if all operations are OK else USB_FAIL or USB_BUSY
284:   */
285: uint8_t CDC_Transmit_FS(uint8_t* Buf, uint16_t Len)
286: {
287:     uint8_t result = USB_OK;
288:     /* USER CODE BEGIN 7 */
289:     USBDCDC_HandleTypeDef *hcdc = (USBDCDC_HandleTypeDef*)hUsbDeviceFS.pClassData;
290:     if (hcdc->TxState != 0){
291:         return USB_BUSY;
292:     }
293:     USBDCDC_SetTxBuffer(&hUsbDeviceFS, Buf, Len);
294:     result = USBDCDC_TransmitPacket(&hUsbDeviceFS);
295:     /* USER CODE END 7 */
296:     return result;
297: }
298:
  
```

只是起到传输，  
此函数返回后，  
并不表示数据发送完毕

发送函数需要改造，如下：

```

ht 4.0 - [usbdc_cdc_if.c (USB_DEVICE\App)]
Tools View Window Help
vers\DshanMCU-F103) logicanalyzer.c (Core\Src) logicanalyzer.h (Core\Src) main.c (Core\Src) operation.S (Core\Src) Search Results stm32f1xx_hal_p
298:
299: /* USER CODE BEGIN PRIVATE_FUNCTIONS_IMPLEMENTATION */
300: uint8_t usb_send(uint8_t *datas, int len, int timeout)
301: {
302:     USBDC_CDC_HandleTypeDef *hcdc = (USBDC_CDC_HandleTypeDef*)hUsbDeviceFS.pClassData;
303:
304:     while(1)
305:     {
306:         if (hcdc->TxState == 0) 1. 等待上次数据发送完毕
307:         {
308:             break;
309:         }
310:         if (timeout-->0)
311:         {
312:             mdelay(1);
313:         }
314:         else
315:         {
316:             return HAL_BUSY;
317:         }
318:     }
319:
320:     return CDC_Transmit_FS(datas, len); 2.再启动新的发送
321: } « end usb_send »
322:

```

代码如下:

```

/**
 * @brief Data received over USB OUT endpoint are sent over CDC interface
 *        through this function.
 *
 *
 * @note
 *
 * This function will issue a NAK packet on any OUT packet received on
 * USB endpoint until exiting this function. If you exit this function
 * before transfer is complete on CDC interface (ie. using DMA controller)
 * it will result in receiving more data while previous ones are still
 * not sent.
 *
 *
 * @param Buf: Buffer of data to be received
 * @param Len: Number of data received (in bytes)
 * @retval Result of the operation: USB_OK if all operations are OK else USB_FAIL
 */
static int8_t CDC_Receive_FS(uint8_t* Buf, uint32_t *Len)
{
    /* USER CODE BEGIN 6 */
    for (uint32_t i = 0; i < *Len; i++)
    {
        circle_buf_write(&g_uart_rx_bufs, Buf[i]);
    }
    USBDC_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);
    USBDC_CDC_ReceivePacket(&hUsbDeviceFS);
    return (USB_OK);
}

```

```
/* USER CODE END 6 */
}

/* USER CODE BEGIN PRIVATE_FUNCTIONS_IMPLEMENTATION */
uint8_t usb_send(uint8_t *datas, int len, int timeout)
{
    USBDCDC_HandleTypeDef *hcdc = (USBDCDC_HandleTypeDef*)hUsbDeviceFS.pClassData;

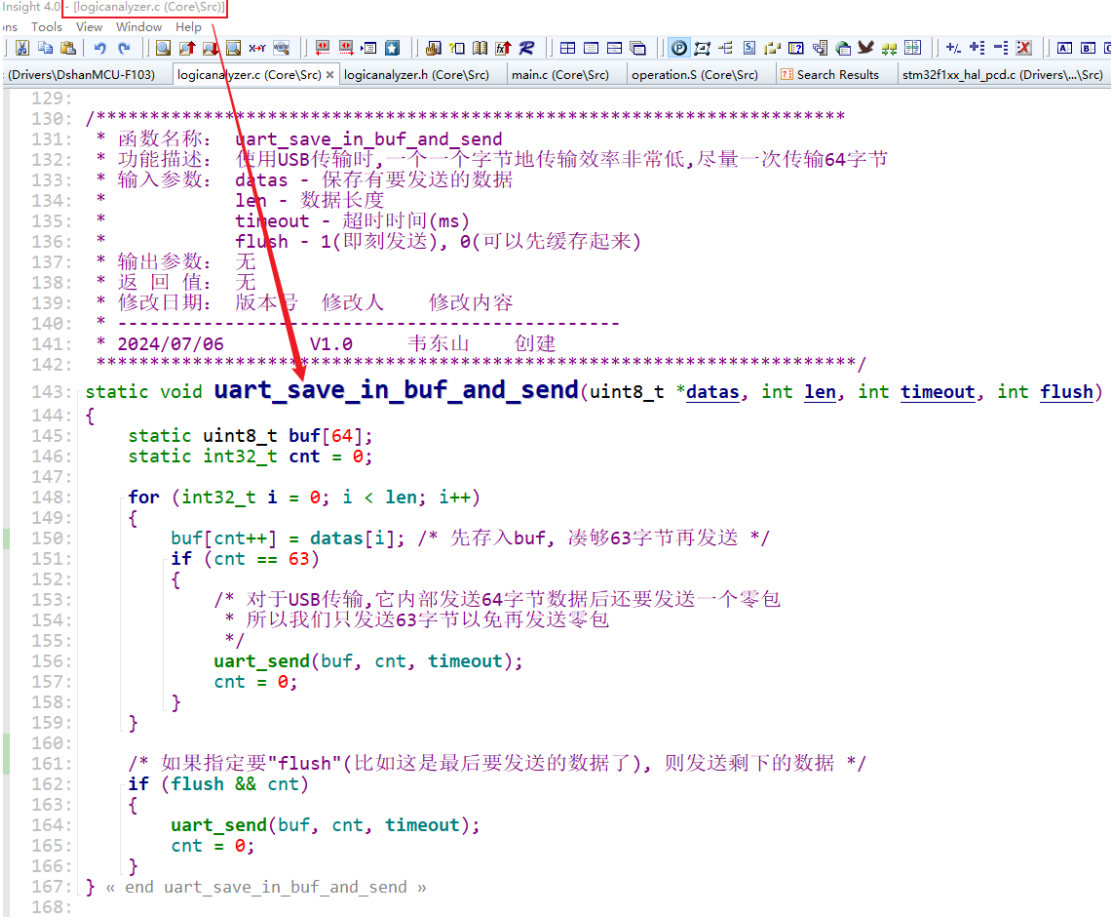
    while(1)
    {
        if (hcdc->TxState == 0)
        {
            break;
        }
        if (timeout-->0)
        {
            mdelay(1);
        }
        else
        {
            return HAL_BUSY;
        }
    }

    return CDC_Transmit_FS(datas, len);
}
```

### 3. 提高 USB 串口发送效率

虽然 USB 速度远高于 UART，但是如果使用 USB 传输数据时是一个字节一个字节地传输，那么效率极低。我们需要根据 USB 的特性，一次尽可能传输更多数据。STM32F103 的 USB 传输，一次能传输最多 64 字节的数据。上位机怎么知道当前数据传输完毕了呢？下位机可以传输少于 64 字节的数据，上位机就知道当前传输完毕了（没传完你干嘛不传输 64 字节呢？）；如果下位机刚好要传输 64 字节的数据，那么 USB 驱动还要额外传输一个“零包”给上位机，为了避免传输零包，我们尽量每次传输 63 字节。

改进后打 USB 串口发送函数如下：



```
129:
130: /******
131: * 函数名称:  uart_save_in_buf_and_send
132: * 功能描述:  使用USB传输时,一个一个字节地传输效率非常低,尽量一次传输64字节
133: * 输入参数:  datas - 保存有要发送的数据
134: *           len  - 数据长度
135: *           timeout - 超时时间(ms)
136: *           flush - 1(即刻发送), 0(可以先缓存起来)
137: * 输出参数:  无
138: * 返回值:    无
139: * 修改日期:  版本号  修改人    修改内容
140: * -----
141: * 2024/07/06      V1.0    韦东山    创建
142: *****/
143: static void uart_save_in_buf_and_send(uint8_t *datas, int len, int timeout, int flush)
144: {
145:     static uint8_t buf[64];
146:     static int32_t cnt = 0;
147:
148:     for (int32_t i = 0; i < len; i++)
149:     {
150:         buf[cnt++] = datas[i]; /* 先存入buf, 凑够63字节再发送 */
151:         if (cnt == 63)
152:         {
153:             /* 对于USB传输,它内部发送64字节数据后还要发送一个零包
154:              * 所以我们只发送63字节以免再发送零包
155:              */
156:             uart_send(buf, cnt, timeout);
157:             cnt = 0;
158:         }
159:     }
160:
161:     /* 如果指定要"flush"(比如这是最后要发送的数据了), 则发送剩下的数据 */
162:     if (flush && cnt)
163:     {
164:         uart_send(buf, cnt, timeout);
165:         cnt = 0;
166:     }
167: } « end uart_save_in_buf_and_send »
168:
```

代码如下：

```
/******
* 函数名称:  uart_save_in_buf_and_send
* 功能描述:  使用USB传输时,一个一个字节地传输效率非常低,尽量一次传输64字节
* 输入参数:  datas - 保存有要发送的数据
*           len  - 数据长度
*           timeout - 超时时间(ms)
*           flush - 1(即刻发送), 0(可以先缓存起来)
* 输出参数:  无
* 返回值:    无
* 修改日期:      版本号    修改人    修改内容
* -----
* 2024/07/06      V1.0    韦东山    创建
*****
```

```

/*****
static void uart_save_in_buf_and_send(uint8_t *datas, int len, int timeout, int flush)
{
    static uint8_t buf[64];
    static int32_t cnt = 0;

    for (int32_t i = 0; i < len; i++)
    {
        buf[cnt++] = datas[i]; /* 先存入buf, 凑够63字节再发送 */
        if (cnt == 63)
        {
            /* 对于USB传输, 它内部发送64字节数据后还要发送一个零包
             * 所以我们只发送63字节以免再发送零包
             */
            uart_send(buf, cnt, timeout);
            cnt = 0;
        }
    }

    /* 如果指定要"flush"(比如这是最后要发送的数据了), 则发送剩下的数据 */
    if (flush && cnt)
    {
        uart_send(buf, cnt, timeout);
        cnt = 0;
    }
}

```

### 3.3.5 使用 RLE 提升重复数据的传输效率

#### 1. RLE 功能

假设要传输 9 个相同的数据, 比如 9 个 0x12, 那么常规方法就要发送 9 个 0x12。如果规定发送的数据里, 某一个数据表示“相同的数据个数”, 后面跟着这个数据, 不就只需要发送 2 个字节的数据了吗? 比如“0x09 0x12”。我们怎么分辨一个数据是长度, 还是数据本身? 可以使用最高位来分辨: 比如 0x89 表示要传输 9 个数据 (SUMP 协议里表示要传输 10 个数据), 0x12 表示数据本身。缺点是: 数据里最高位必须清为 0。

RLE: Run Length Encoding, 在数据里嵌入长度。在传输重复的数据时可以提高效率。SUMP 协议里规定:

- ① 传输长度: 最高位为 1, 去掉最高位的数值为 n, 表示有 (n+1) 个数据
- ② 传输数据: 数据的最高位必须为 0

例子 1: 对于 8 通道的数据, channel 7 就无法使用了。要传输 10 个数据 0x12 时, 只需要传输 2 字节: 0x89 0x12。

0x89 的最高位为 1, 表示有 (9+1) 个相同的数据, 数据为 0x12。

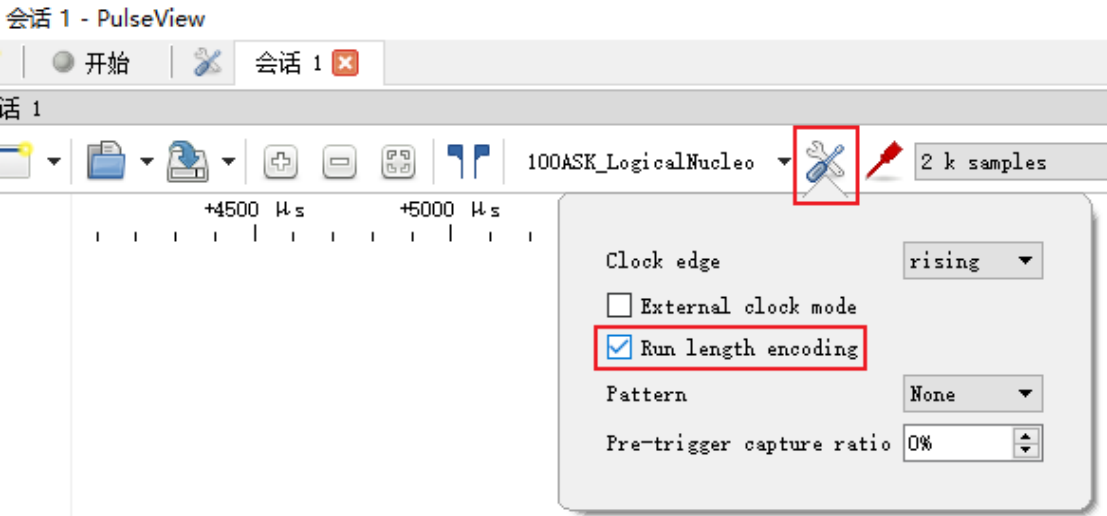
例子 2: 对于 32 通道的数据, channel 31 就无法使用了。要传输 10 个数据 0x12345678 时, 只需要传输 8 字节: 0x09 0x00 0x00 0x80 0x78 0x56 0x34 0x12

“0x09 0x00 0x00 0x80”的最高位为 1, 表示有 (9+1) 个相同的数据, 数据为“0x78 0x56

0x34 0x12”

2. 上位机是能 RLE

如下设置：



3. 代码解读

代码如下：

```

/*****
* 函数名称:  upload
* 功能描述:  上报数据
* 输入参数:  无
* 输出参数:  无
* 返回值:  无
* 修改日期:      版本号      修改人      修改内容
* -----
* 2024/07/04      V1.0      韦东山      创建
*****/

static void upload (void)
{
    int32_t i = g_cur_pos;
    uint32_t j;
    uint32_t rate = MAX_FREQUENCY / g_samplingRate;
    int cnt = 0;
    uint8_t pre_data;
    uint8_t data;
    uint8_t rle_cnt = 0;

    for (; i >= 0; i--)
    {
        for (j = 0; j < g_rxcnt_buf[i]; j++)
        {
            cnt++;
            /* 我们以最大频率采样，假设最大频率是1MHz

```

```

    * 上位机想以200KHz的频率采样
    * 那么在得到的数据里，每5个里只需要上报1个
    */
    if (cnt == rate)
    {
        if (g_flags & CAPTURE_FLAG_RLE)
        {
            /* RLE : Run Length Encoding, 在数据里嵌入长度，在传输重复的数据时可以提
高效率
            * 先传输长度：最高位为1表示长度，去掉最高位的数值为n，表示有(n+1)个数据
            * 再传输数据本身（数据的最高位必须为0）
            * 例子1：对于8通道的数据，channel 7就无法使用了
            * 要传输10个数据 0x12时，只需要传输2字节：0x89 0x12
            * 0x89的最高位为1，表示有(9+1)个相同的数据，数据为0x12
            *
            * 例子2：对于32通道的数据，channel 31就无法使用了
            * 要传输10个数据 0x12345678时，只需要传输8字节：0x09 0x00 0x00 0x80
0x78 0x56 0x34 0x12
            * "0x09 0x00 0x00 0x80"的最高位为1，表示有(9+1)个相同的数据，数据为
"0x78 0x56 0x34 0x12"
            */

            data = g_rxddata_buf[i] & ~0x80; /* 使用RLE时数据的最高位要清零 */;

            if (rle_cnt == 0)
            {
                pre_data = data;
                rle_cnt = 1;
            }
            else if (pre_data == data)
            {
                rle_cnt++; /* 数据相同则累加个数 */
            }
            else if (pre_data != data)
            {
                /* 数据不同则上传前面的数据 */

                if (rle_cnt == 1) /* 如果前面的数据只有一个，则无需RLE编码 */
                    uart_save_in_buf_and_send(&pre_data, 1, 100, 0);
                else
                {
                    /* 如果前面的数据大于1个，则使用RLE编码 */
                    rle_cnt = 0x80 | (rle_cnt - 1);
                    uart_save_in_buf_and_send(&rle_cnt, 1, 100, 0);

```



```
        uart_save_in_buf_and_send(&pre_data, 1, 100, 0);
    }
    pre_data = data;
    rle_cnt = 1;
}

if (rle_cnt == 128)
{
    /* 对于只有8个通道的逻辑分析仪，只使用1个字节表示长度,最大长度为128
    * 当相同数据个数累加到128个时,
    * 就先上传
    */
    rle_cnt = 0x80 | (rle_cnt - 1);
    uart_save_in_buf_and_send(&rle_cnt, 1, 100, 0);
    uart_save_in_buf_and_send(&pre_data, 1, 100, 0);
    rle_cnt = 0;
}
}
else
{
    /* 上位机没有起到RLE功能则直接上传 */
    uart_save_in_buf_and_send(&g_rxdata_buf[i], 1, 100, 0);
}

cnt = 0;
}
}

/* 发送最后的数据 */
if ((g_flags | CAPTURE_FLAG_RLE) && rle_cnt)
{
    if (rle_cnt == 1)
        uart_save_in_buf_and_send(&pre_data, 1, 100, 0);
    else
    {
        rle_cnt = 0x80 | (rle_cnt - 1);
        uart_save_in_buf_and_send(&rle_cnt, 1, 100, 0);
        uart_save_in_buf_and_send(&pre_data, 1, 100, 0);
    }
}

/* 为了提高USB上传效率,我们“凑够一定量的数据后才发送”,
* 现在都到最后一步了,剩下的数据全部flush、上传
```

```
    */  
    uart_save_in_buf_and_send(NULL, 0, 100, 1);  
}
```