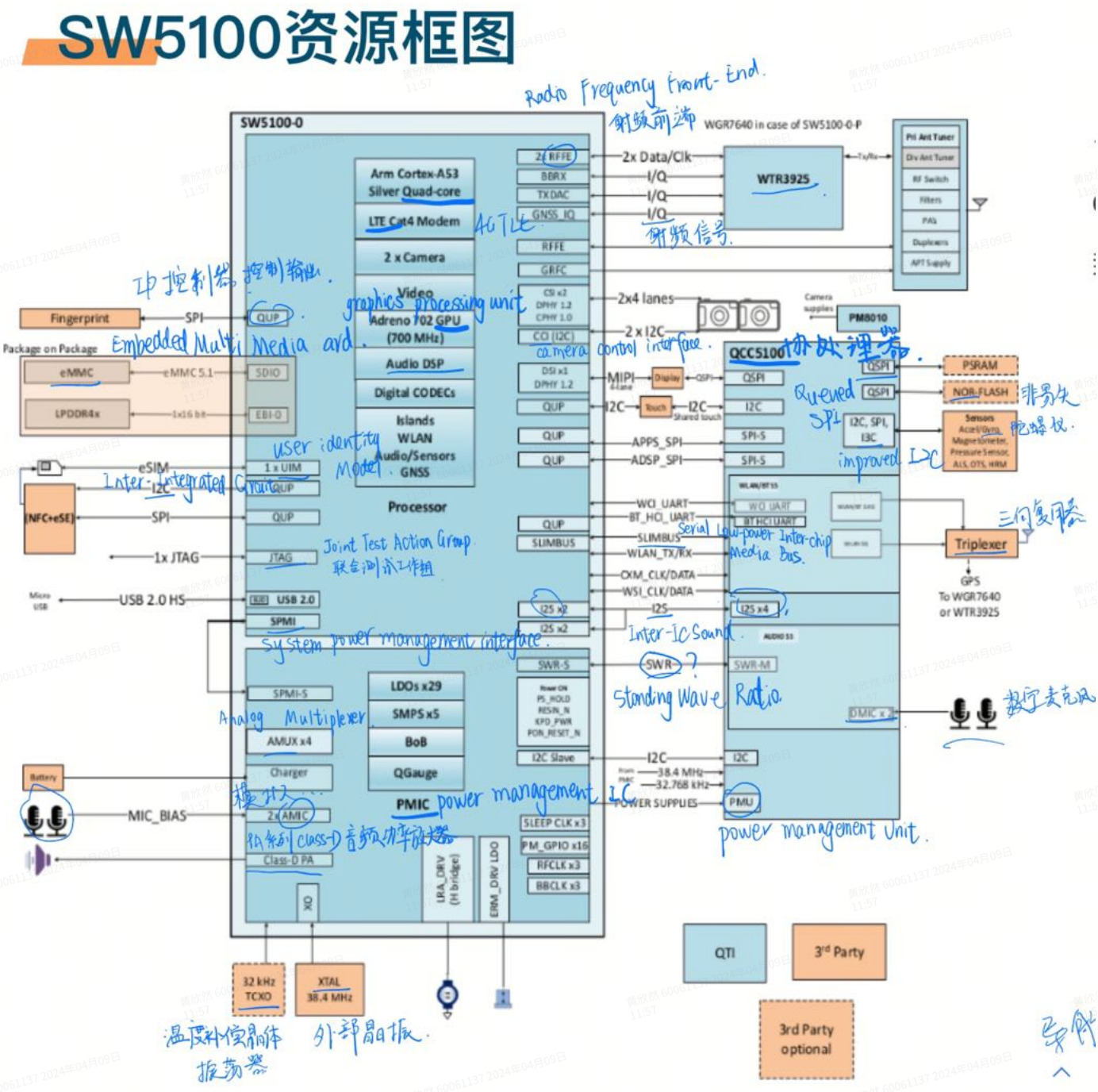
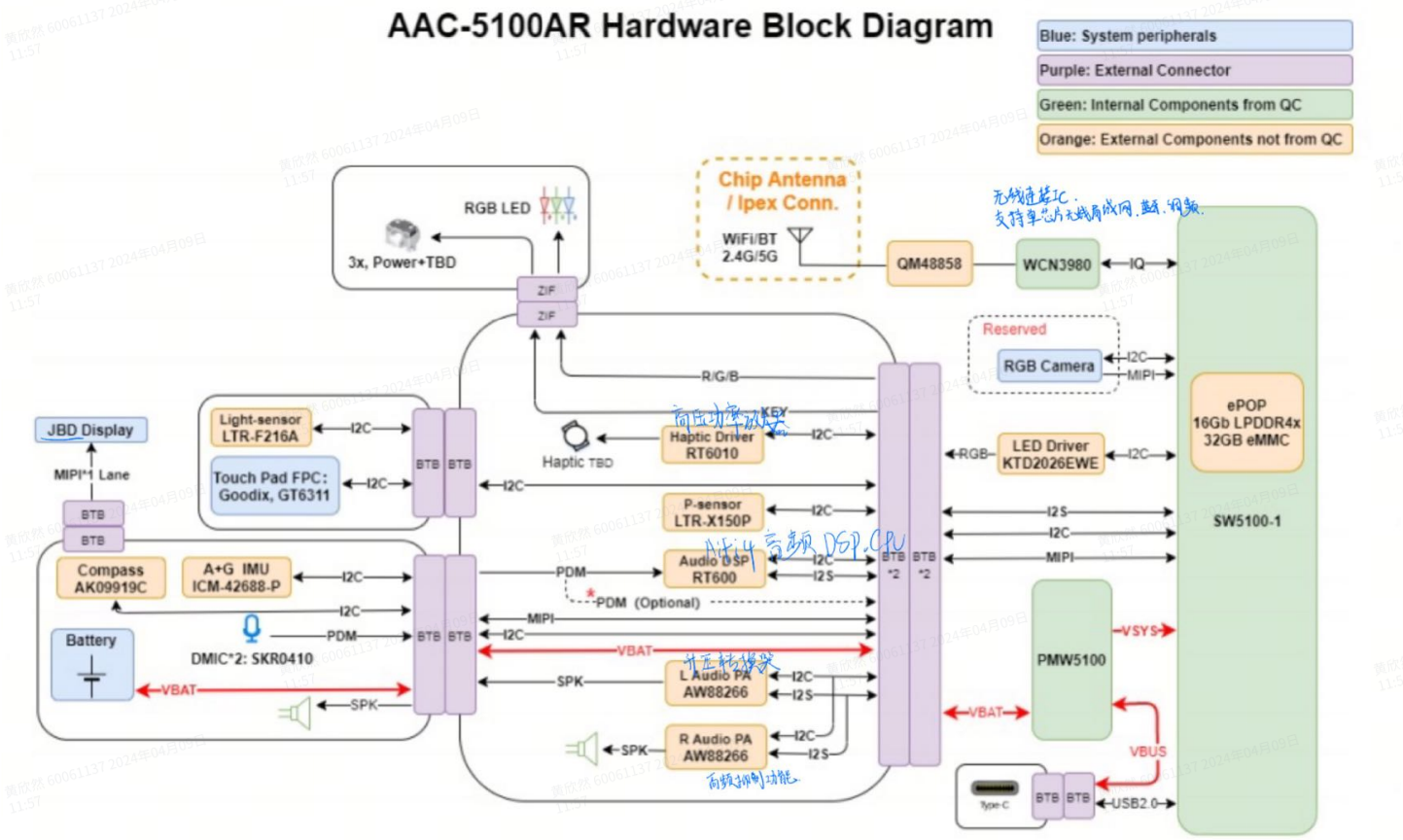
SW5100学习（更新中）

硬件资源及原理图

sw5100资源框图





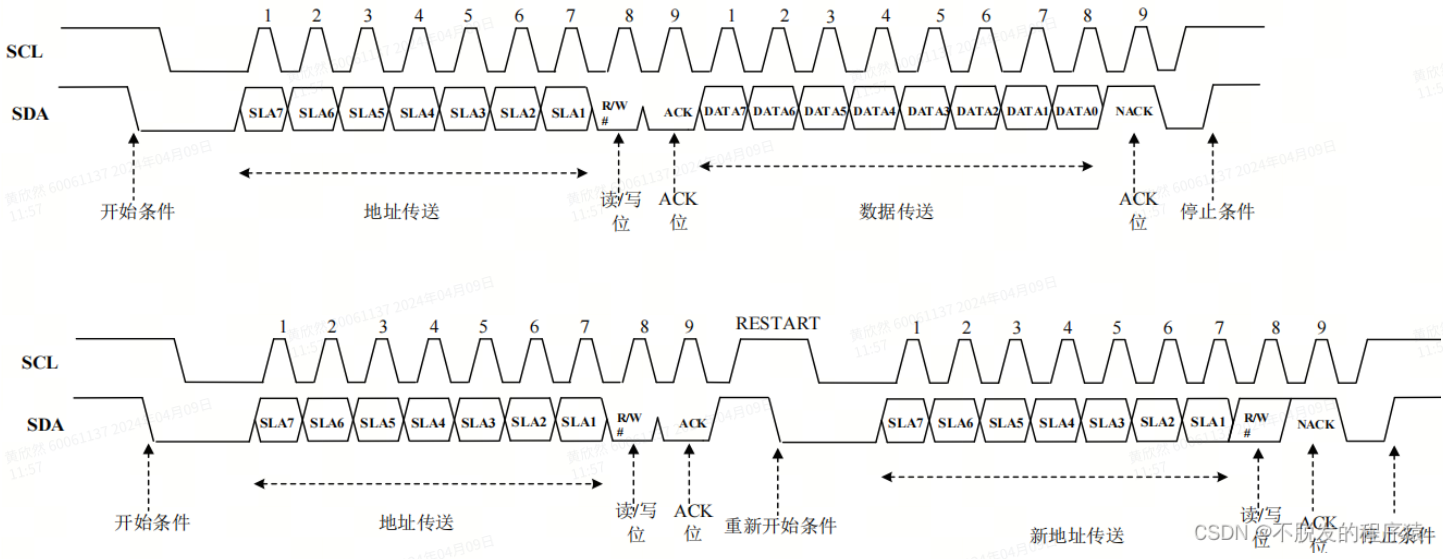
接口与总线

I2C

I2C（集成电路总线）,用于连接微处理器及外围设备。

是双线双向的同步串行总线，它利用一根时钟线（SCL）和一根数据线（SDA）在连接总线的两个器件之间进行信息的传递。每个连接到总线上的器件都有唯一的地址，任何器件既可以作为主机也可以作为从机，但同一时刻只允许有一个主机。

标准的I2C时序如下图所示：



当总线上的主机都不驱动总线，总线进入空闲状态，SCL和SDA都为高电平;

CSDN @不脱发的程序猿

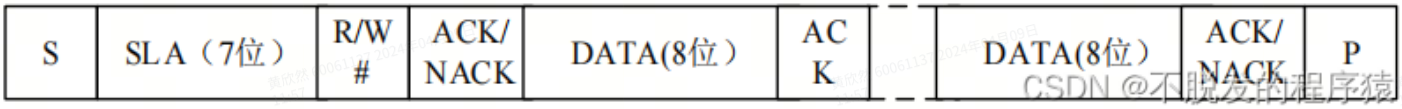
当SCL 为高时，SDA 上出现由高到低的信号，表明总线上产生了起始信号；

当SDA 上出现由低到高的信号，表明总线上产生了停止信号。

地址传送：

I2C通讯支持7 位寻址和10 位寻址两种模式。

7位：



R/W#：表示发送和接收的方向。当 R/W 为1 时，将数据从从机发送到主机；当 R/W 为0 时，将数据从主机发送到从机；

I2S

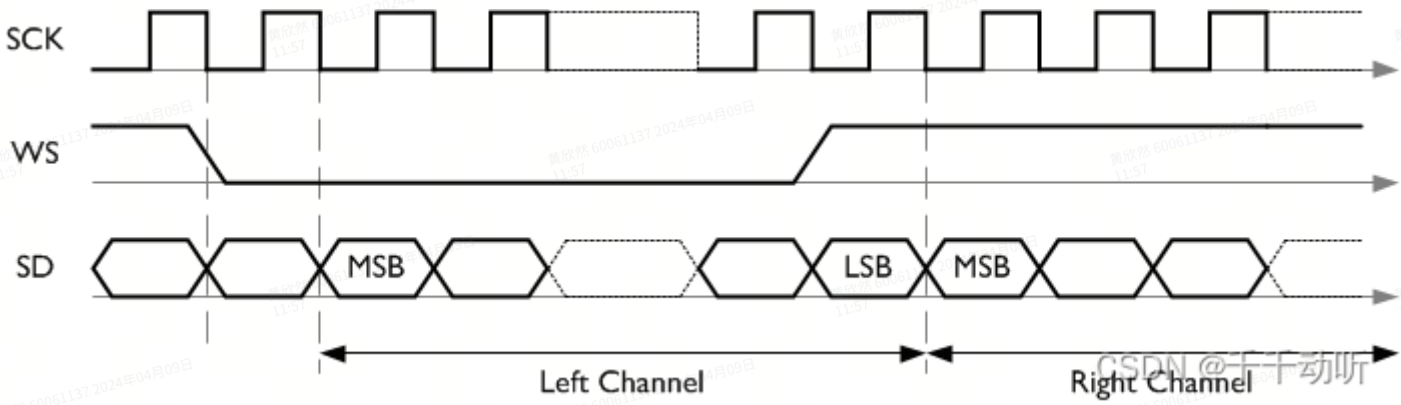
I2S（Inter-IC Sound）是一种广泛应用于数字音频传输的串行接口标准。

时钟线（Continuous Serial clock ，SCK）

左/右声道线（Word Select，WS）

数据线（Serial Data，SD）

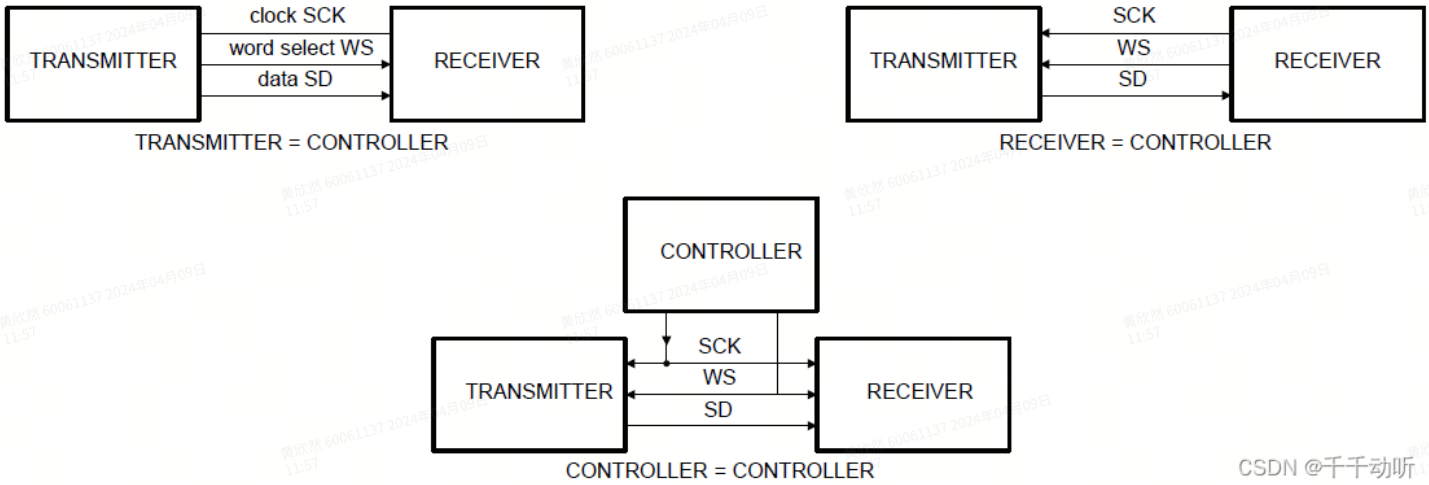
左声道数据的LSB在LRCLK下降沿的前一个SCK/BCLK上升沿有效，右声道数据的LSB在LRCLK上升沿的前一个SCK/BCLK上升沿有效。



工作模式：

主模式（Master Mode）：主设备负责提供时钟信号和帧同步信号，控制音频数据的传输。

从模式（Slave Mode）：在从模式下，从设备接收来自主设备的时钟和帧同步信号，并接收或发送音频数据。



CSDN @千千动听

I3C

I3C总线协议，包括SDR（Single Data Rate）模式和HDR（High Data Rate）模式

SPI

SPI(Serial Peripheral Interface),即串行外部设备接口,是一种全双工、高速、同步的串行通信总线。

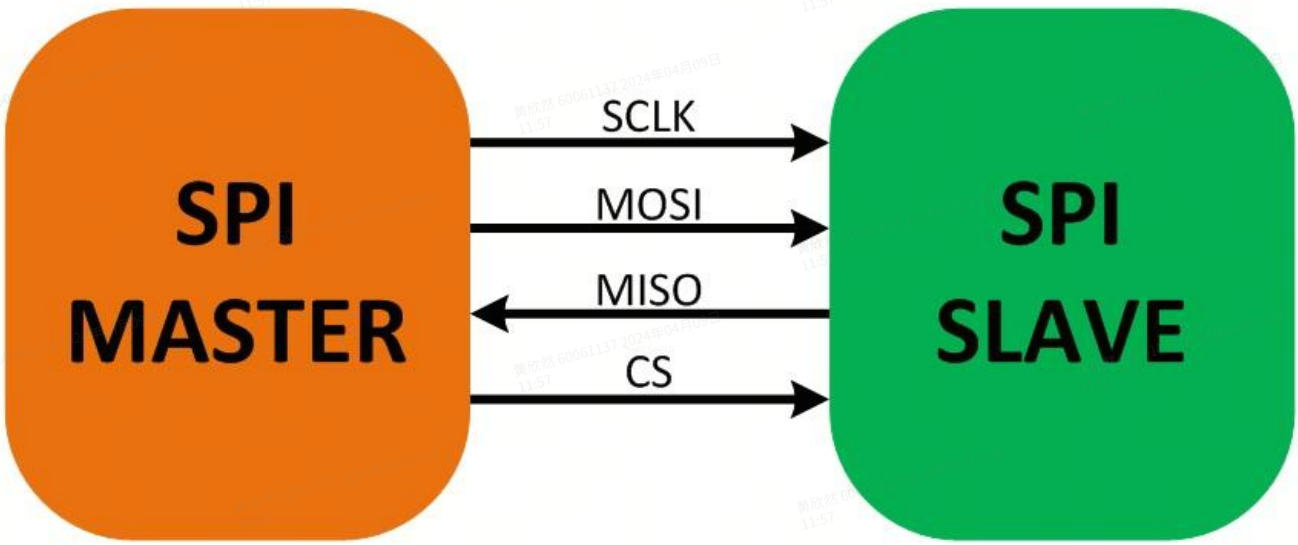
4信号线：

SCK：串行时钟信号(Serial Clock),由主设备产生发送给从机

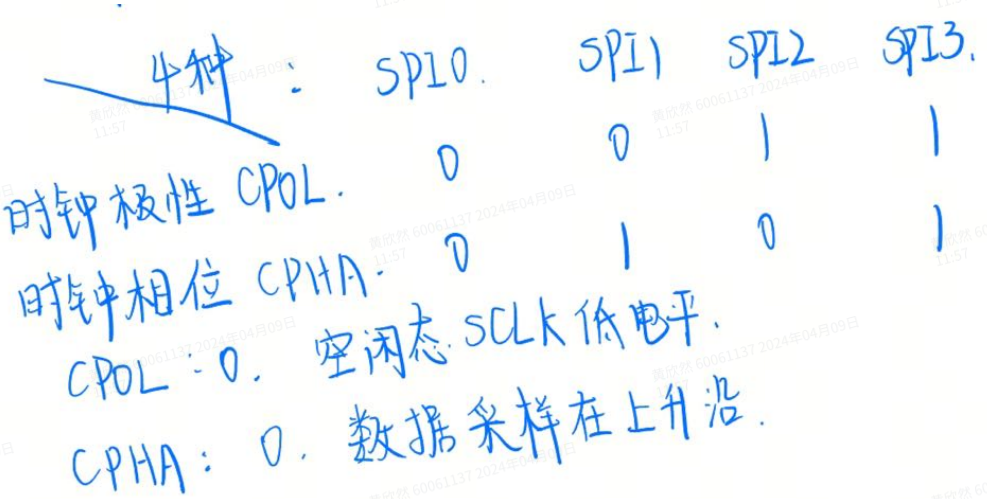
MOSI：主发从收信号(Master Output Slave Input),主设备输出/从设备输入引脚,该引脚在主模式下发送数据,在从模式下接收数据

MISO：主收从发信号(Master Input Slave Output),主设备输入/从设备输出引脚,该引脚在从模式下发送数据,在主模式下接收数据

CS/SS：片选信号(Slave Select),由主设备控制。它的功能是用来作为“片选引脚”,也就是选择指定的从设备,让主设备可以单独地与特定从设备通讯,避免数据线上的冲突



4种SPI模式：

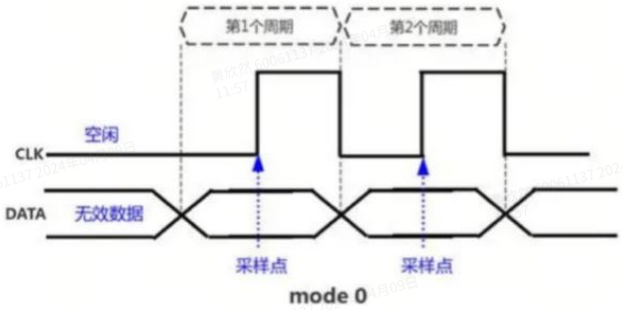


1.模式0(CPOL=0，CPHA=0)

模式0特性：

CPOL = 0：空闲时是低电平，第1个跳变沿是上升沿，第2个跳变沿是下降沿

CPHA = 0：数据在第1个跳变沿（上升沿）采样



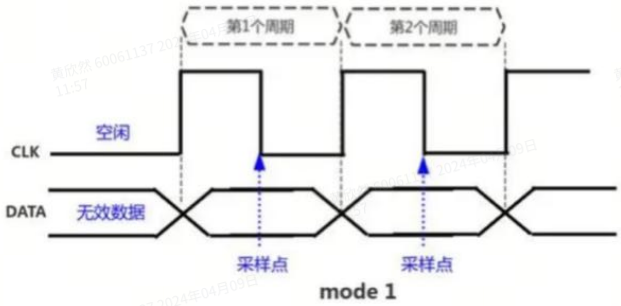
mode 0

2.模式1(CPOL=0，CPHA=1)

模式1特性：

CPOL = 0：空闲时是低电平，第1个跳变沿是上升沿，第2个跳变沿是下降沿

CPHA = 1：数据在第2个跳变沿（下降沿）采样



mode 1

时序：

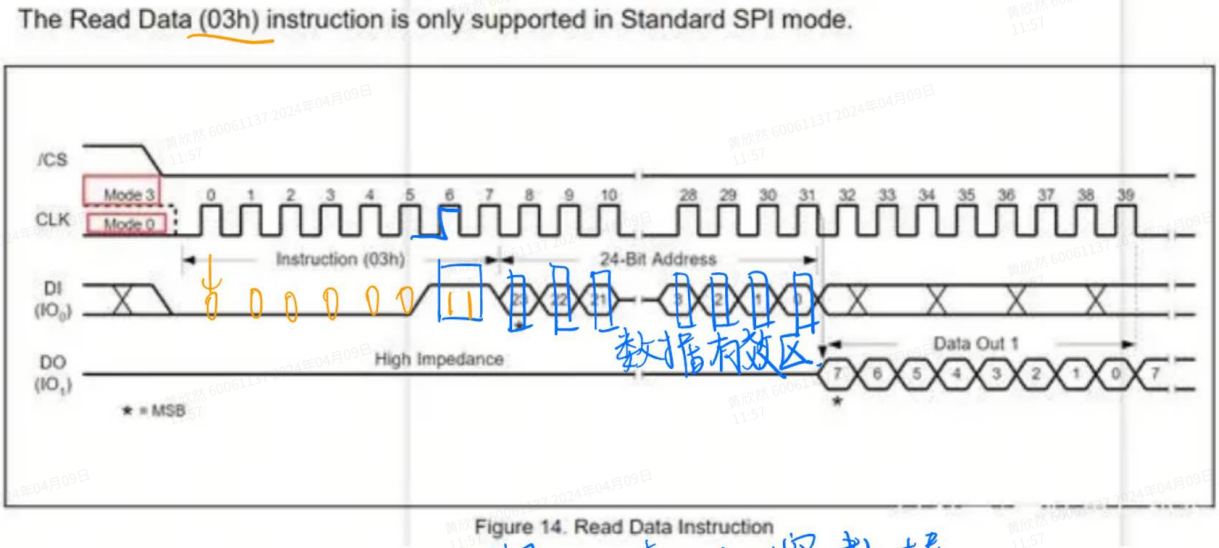


Figure 14. Read Data Instruction

DI：从机的MISO引脚，接收主控数据

DO：从机的MOSI引脚，现变为主机，发送指定地址数据。

MIPI

移动产业处理器接口（Mobile Industry Processor Interface 简称MIPI）

UART

UART(Universal Asynchronous Receiver/Transmitter)通用异步收发传输器，是一种通用串行数据总线，用于异步通信。该总线双向通信，可以实现全双工传输和接收。

UART作为异步串口通信协议的一种，工作原理是将传输数据的每个字符一位接一位地传输。

其中各位的意义如下:

起始位：先发出一个逻辑"0"的信号，表示传输字符的开始。

资料位：紧接着起始位之后。资料位的个数可以是4、5、6、7、8等，构成一个字符。通常采用ASCII码。从最低位开始传送，靠时钟定位。

奇偶检验位：资料位加上这一位后，使得"1"的位数应为偶数(偶校验)或奇数(奇校验)，以此来校验资料传送的正确性。

停止位：它是一个字符数据的结束标志。可以是1位、1.5位、2位的高电平。 由于数据是在传输线上定时的，并且每一个设备有其自己的时钟，很可能在通信中两台设备出现了小小的不同步。因此停止位不仅仅是表示传输的结束，并且提供计算机校正时钟同步的机会。适用于停止位的位数越多，不同时钟同步的容忍程度越大，但是数据传输率同时也越慢。

空闲位：处于逻辑"1"状态，表示当前线路上没有资料传送。

波特率：是衡量资料传送速率的指标。表示每秒钟传送的符号数(symbol)。一个符号代表的信息量(比特数)与符号的阶数有关。例如资料传送速率为120字符/秒，传输使用256阶符号，每个符号代表8bit，则波特率就是120baud，比特率是120*8=960bit/s。

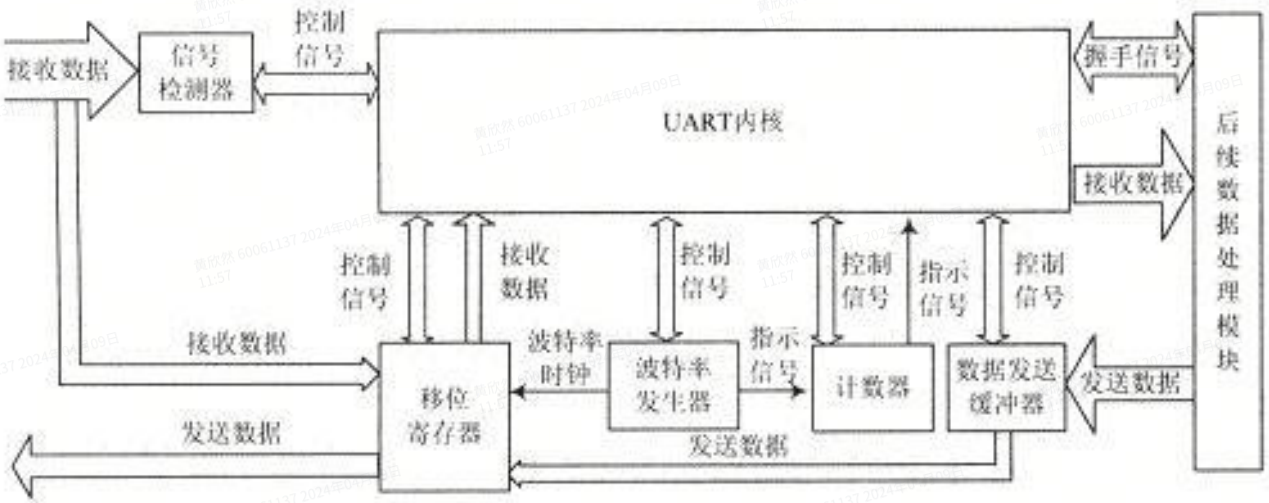


图 2 UART 基本结构

基本结构：

⑴ 输出缓冲寄存器，它接收CPU从数据总线上送来的并行数据，并加以保存。

⑵ 输出移位寄存器，它接收从输出缓冲器送来的并行数据，以发送时钟的速率把数据逐位移出，即将并行数据转换为串行数据输出。

⑶ 输入移位寄存器，它以接收时钟的速率把出现在串行数据输入线上的数据逐位移入，当数据装满后，并行送往输入缓冲寄存器，即将串行数据转换成并行数据。

⑷ 输入缓冲寄存器，它从输入移位寄存器中接收并行数据，然后由CPU取走。

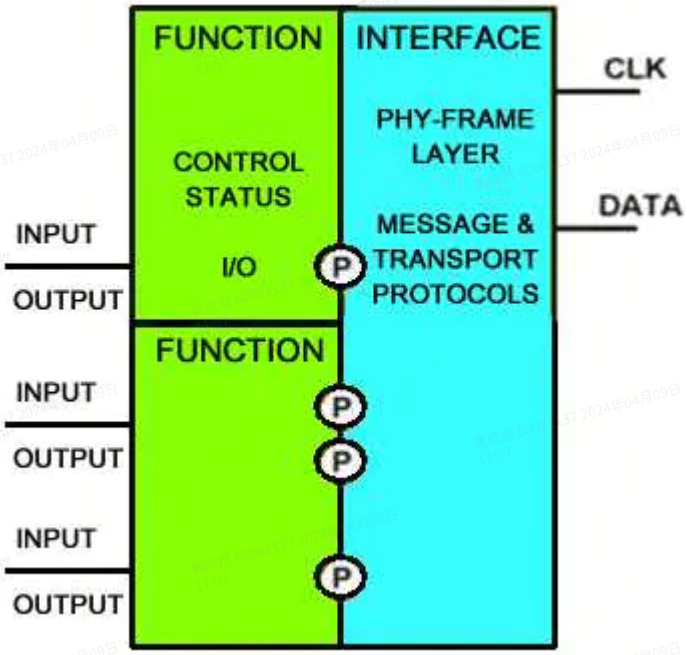
⑸控制寄存器，它接收CPU送来的控制字，由控制字的内容，决定通信时的传输方式以及数据格式等。例如采用异步方式还是同步方式，数据字符的位数，有无奇偶校验，是奇校验还是偶校验，停止位的位数等参数。

⑹状态寄存器。状态寄存器中存放着接口的各种状态信息，例如输出缓冲区是否空，输入字符是否准备好等。在通信过程中，当符合某种状态时，接口中的状态检测逻辑将状态寄存器的相应位置"1"，以便让CPU查询。

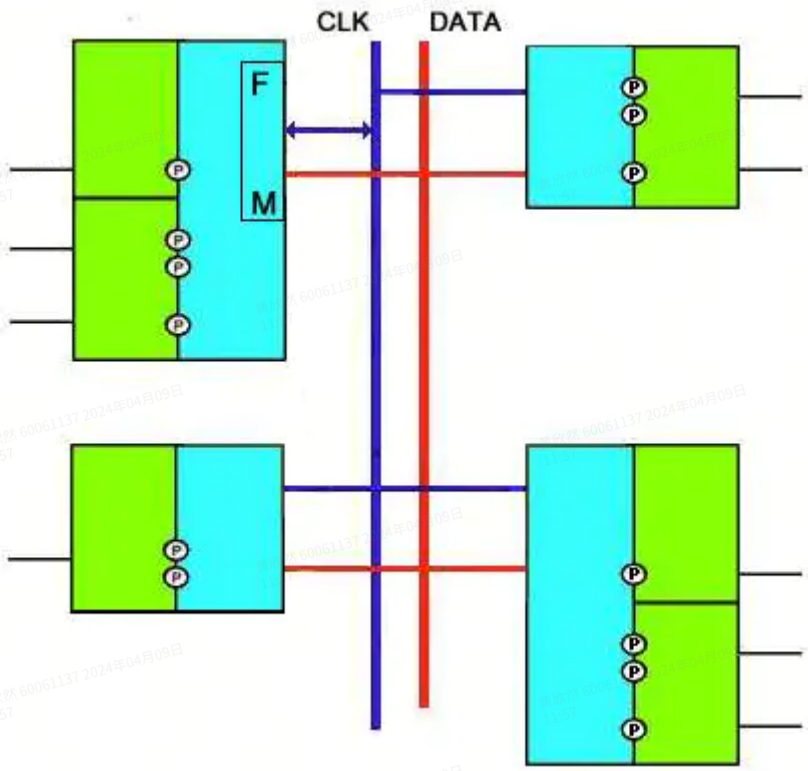
Slimbus

Serial Low-power Inter-chip Media Bus，是MIPI联盟指定的一种音频接口，用于连接基带/应用处理器和音频芯片，总线协议保证既能发控制信息，又能发数据信息，这样就可以替换传统的数据和控制两种接口如I2S和I2C。

slimbus组件：



slimbus系统：



DATA和CLK

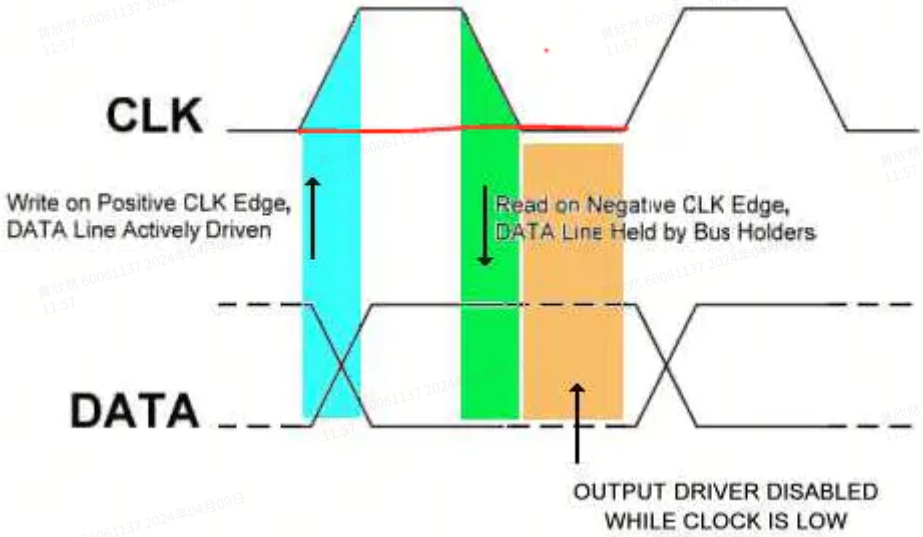
- 包含帧设备的组件的CLK是双向，其他都只输入。
- 所有组件的DATA都是双向，收发用NRZI编码。
- CLK正跳变写DATA，负跳变读DATA。

帧结构

由Cells, Slots, 帧, 子帧和超帧组成。

- Cell

是两个CLK正跳变的间隔，包含一次bit读和写，如图中红线间隔。



- Slot

由4个Cell组成，表示从高位到低位传4个bit。

- 帧

1. 由192个Slot组成，分别传Slot0-Slot191。
2. Slot0包含4个帧同步bit，Slot96包含4个组帧bit。

- 子帧

3. 是帧的划分。
4. 由多个控制slot和0到多个数据slot组成。
5. 长度可配置成6、8、24、32个slot。

控制slot中有4个被保留，用于传4bit帧同步、4bit组帧、8bit引导字节。

- 超帧

由8个帧组成，是一个完整的数据单元。

WCN3980

无线连接IC，支持单芯片无线局域网，蓝牙，调频。

内部通过Slimbus连接；

IQ信号：

（1）IQ信号即同相正交信号，I为in-phase，Q为quadrature，与I的相位相差了90度。

（2）因为信号在传输的时候需要的是单一信道、单一频率，因此在最早的通讯技术中往往选择通过滤波器滤掉另外一个频率的信号，结果是不理想的，使用滤波器很难滤掉另外一个，而且因为另外一个频带的存在，浪费了很多频带的资源，在频带资源尤为紧张的现在，解决该问题非常重要，因此发展出了IQ技术。

（3）I 是指同相分量，Q 是指正交分量。I 路和 Q 路是完全正交的。

QM48858

编译命令

```
SDK编译说明:
source build/envsetup.sh
lunch monaco_go-userdebug
./build.sh dist -j16 #根据CPU选择核数,此命令是编译全部镜像.
单独编译内核:
make dtboimage -j16 #编译kernel dts
make bootimage -j16 #编译kernel镜像
#编译模块 > mmm /path/to/module/dir  #不自动编译模块依赖
#自动编译模块 > mmma /path/to/module/dir #自动编译模块依赖

FASTBOOT烧写:
1. 进入系统后运行
adb reboot bootloader
adb wait-for-device
adb reboot bootloader
fastboot devices
fastboot flash dtbo dtbo.img
fastboot flash boot boot.img
fastboot flash vbmeta vbmeta.img
fastboot flash vbmeta_system vbmeta_system.img
fastboot flash userdata userdata.img
fastboot flash persist persist.img
fastboot flash super super.img
fastboot reboot
即可完成镜像更新。

当主板无法启动时,可选择QFIL包进行烧写,选择emmc

进入烧录模式QFIL:
adb reboot edl
```

```
内核打印信息:
dmesg | grep 关键字

push test 到 system:
adb root
adb remount
adb push test /system/bin/

获取当前目录代码修改状态
git status .
git diff make/envsetup.sh
git checkout make/envsetup.sh

git log
git show ...

找某个文件:
find ./ -name ...

显示设备界面:
scrcpy

从家目录向设备传送文件
adb push /home... /sdcard
从设备向家目录传送文件
adb  pull /data/mic.wav /home/huangxinran

git status .
rm src -rf
git checkout .

git add .
git commit -m "注释"

sudo chown liyang:liyang proprietary -R

beyond compare


```

Android驱动编写

```
目录: /disk1/work/test1/sw5100_for_ar/android
```

注册一个平台驱动，加入probe log打印，在dev目录创建文件结点，执行open,write等操作

主要参考文档：

https://www.jianshu.com/p/9a8e833124c7

进入/disk1/work/test1/sw5100_for_ar/android/kernel/msm-5.4/drivers/misc，创建nxp_rt.c

1.注册平台驱动

```
static struct platform_driver nxp_rt_driver{};
```

2.模块安装和卸载函数

```
 static int __init nxp_rt_init(void) {
     return platform_driver_register(&nxp_rt_driver);
 }
 static void __exit nxp_rt_exit(void) {
     platform_driver_unregister(&nxp_rt_driver);
 }
```

```
make bootimage -j16 #编译kernel镜像
```

进入vendor目录

3.device在设备树注册（目录monaco.dtsi &soc）

```
nxp_rt{
  compatible="qcom,nxp_rt";
};
```

```
make dtboimage -j16 #编译kernel dts
```

记得复制镜像文件到本地：

```
scp liyang@10.0.21.134:/disk1/work/test1/sw5100_for_ar/android/out/target/product/monaco_go/dtbo.img /home/huangxinran
scp liyang@10.0.21.134:/disk1/work/test1/sw5100_for_ar/android/out/target/product/monaco_go/boot.img /home/huangxinran
scp liyang@10.0.21.134:/disk1/work/test1/sw5100_for_ar/android/out/target/product/monaco_go/vbmeta.img /home/huangxinran
scp liyang@10.0.21.134:/disk1/work/test1/sw5100_for_ar/android/out/target/product/monaco_go/vbmeta_system.img /home/huangxinran
scp liyang@10.0.21.134:/disk1/work/test1/sw5100_for_ar/android/out/target/product/monaco_go/userdata.img /home/huangxinran
scp liyang@10.0.21.134:/disk1/work/test1/sw5100_for_ar/android/out/target/product/monaco_go/persist.img /home/huangxinran
scp liyang@10.0.21.134:/disk1/work/test1/sw5100_for_ar/android/out/target/product/monaco_go/super.img /home/huangxinran
```

```
scp liyang@10.0.21.134:/disk1/work/test1/sw5100_for_ar/android/out/target/product/monaco_go/system/bin/nxp_rt /home/huangxinran
进入adb:
adb reboot bootloader
./flash_all.sh

4.字符驱动创建步骤:
// 注册指定的设备号
static dev_t mydev;
register_chrdev_region(mydev,CHRDEV_MAION,CHRDEV_COUNT,CHRDEV_NAME);
// 绑定字符设备操作函数集
cdev_init(&leddev.chrdevcdev, &fops);
// 添加字符设备
cdev_add(&leddev.chrdevcdev,leddev.dev,CHRDEV_COUNT);
// 创建类，类名为testledclass
class_create(THIS_MODULE, "testledclass");
// 创建设备
device_create(leddev.led_dev_class, NULL, MKDEV(leddev.major,0), NULL, "testled");
在/dev下查看注册的设备

进入/disk1/work/test1/sw5100_for_ar/android/external,创建文件夹driver_test
5.驱动测试程序编写
1) Android.bp
//模块
cc_binary {
     name: "nxp_rt",
     host_supported: true,
     srcs:["nxp_rt.c"],
     cflags:["-Werror"],
     target: {
         darwin: {
             enabled: false,
         },
     },
 }


2)测试文件nxp_rt.c
在/disk1/work/test1/sw5100_for_ar/android/下执行:
source build/envsetup.sh
lunch 29
mmm external/clang/
记得要把文件复制到本地:
scp liyang@10.0.21.134:/disk1/work/test1/sw5100_for_ar/android/out/target/product/m
```

```
onaco_go/system/bin/nxp_rt /home/huangxinran
然后进入adb:
adb root
adb remount
adb push nxp_rt /system/bin/
adb shell
nxp_rt
查看是否有open和write信息输出


```

UEFI

```
参考:
https://www.cnblogs.com/loongson-artc-lyc/p/14823387.html
https://blog.csdn.net/qq_39180804/article/details/106089511
https://blog.csdn.net/weixin_43453149/article/details/129587539?spm=1001.2101.3001.6650.1&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-129587539-blog-127040223.235%5Ev38%5Epc_relevant_yljh&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-129587539-blog-127040223.235%5Ev38%5Epc_relevant_yljh&utm_relevant_index=2

UEFI的前身是Intel在1998年开始开发的Intel Boot Initiative，后来被重命名为可扩展固件接口（Extensible Firmware Interface，缩写EFI）

ABL:
入口:
/disk1/work/test1/sw5100_for_ar/android/bootable/bootloader/edk2/QcomModulePkg/Application/LinuxLoader/LinuxLoader.inf
/disk1/work/test1/sw5100_for_ar/android/bootable/bootloader/edk2/QcomModulePkg/Application/LinuxLoader/LinuxLoader.c


XBL:

SEC(安全验证):
汇编代码入口位于:
/disk1/work/test1/sw5100-law-1-0_ap_standard_oem/boot_images/QcomPkg/XBLCore/ModuleEntryPoint.masm
C代码入口位于:
/disk1/work/test1/sw5100-law-1-0_ap_standard_oem/boot_images/QcomPkg/XBLCore/Sec.c
```

PEI（EFI前期初始化）

入口函数 _ModuleEntryPoint位于：

/disk1/work/test1/sw5100-law-1-0_ap_standard_oem/boot_images/MdePkg/Library/PeimEntryPoint/PeimEntryPoint.c

入口函数PEICore位于：

/disk1/work/test1/sw5100-law-1-0_ap_standard_oem/boot_images/MdeModulePkg/Core/Pei/PeiMainPeiMain.c

PEI划分：

PEI内核（PEI Foundation）：负责PEI基础服务和流程

PEIM（PEI Module）派遣器：找出系统中的所有PEIM，并根据PEIM之间的依赖关系按顺序执行PEIM

PEI阶段对系统的初始化主要由PEIM完成

PEI阶段执行流程：

1.SEC模块找到PEI Image的入口函数 _ModuleEntryPoint

2._ModuleEntryPoint函数最终调用PEI模块的入口函数PEICore

3.进入PEICore后，首先根据从SEC阶段出入的信息设置PEI Core Services，然后调用PEIDispatcher执行系统总的PEIM，在内存初始化完成后，系统切换栈并重新进入PEICore,重新进入PEICore后使用的不再是 临时RAM 而是真正的内存。

4.在所有PEIM执行完成后，调用PEIServices的LocatePPI服务得到DXE IPL PPI，并调用DXE IPL PPI的Entry服务（即DEXLoadCore），找出DEX Image的入口函数，执行DXE Image函数并将HOB列表传递给DXE。

DXE（驱动执行环境）

DXE的加载位置在：

/disk1/work/test1/sw5100-law-1-0_ap_standard_oem/boot_images/EmbeddedPkg/Library/PrePiLib/PrePiLib.c

DXE的入口代码位于：

/disk1/work/test1/sw5100-law-1-0_ap_standard_oem/boot_images/MdeModulePkg/Core/Dxe/DxeMain/DxeMain.c

在DXE阶段已经有足够的内存可以使用，因此可以完成大量的驱动加载和初始化工作。

遍历固件中所有的Driver，当Driver所依赖的资源都满足要求时，调度Driver到执行队列执行，直到所有的Driver都被加载和执行完毕，系统完成初始化。

BDS（启动设备选择）

在BDS阶段，主要是初始化控制台设备，加载执行必要的设备驱动，根据用户的选择，执行相应的启动项。

位于：

/disk1/work/test1/sw5100-law-1-0_ap_standard_oem/boot_images/QcomPkg/Drivers/QcomBds/QcomBds.c的BdsEntry

/disk1/work/test1/sw5100-law-1-0_ap_standard_oem/boot_images/QcomPkg/Drivers/QcomBds/QcomBds.c的BdsEntry

1.注册按键事件，按下按键后会回调到HotkeyEvent() 函数，最终调用到HotkeyCallback()函数中，解析其中的
  key scancode
2.平台BDS初始化，PlatformBdsInit ();
3.初始化所有 DriverOptionList 上的 驱动协议，BdsLibLoadDrivers (&DriverOptionList);
4.根据选择的启动方式，启动对应的的系统，BdsBootDeviceSelect ();

XBL Loader

RT(Run Time)
在RT阶段，OS Loader已经完全取得了系统的控制权，因此要清理和回收一些之前被UEFI占用的资源，runtime
services随着操作系统的运行提供相应的运行时的服务，这个期间一旦出现错误和异常，将进入AL进行修复。

XBL屏驱动：
jbd driver:
/disk1/work/test1/sw5100-law-1-0_ap_standard_oem/boot_images/QcomPkg/SocPkg/AthertonPkg/Library/MDPPlatformLib/MDPPlatformLib.c

boot_images/QcomPkg:
//加入环境变量
```
export SECTOOLS=/disk1/work/test1/sw5100-law-1-0_ap_standard_oem/common/sectoolsv2/ext/Linux/sectools
python buildex.py  --variant LAA -r RELEASE -t AthertonPkg,QcomToolsPkg
```
//复制
```
scp liyang@10.0.21.134:/disk1/work/test1/sw5100-law-1-0_ap_standard_oem/boot_images/QcomPkg/SocPkg/AthertonPkg/Bin/LAA/RELEASE/xbl.elf /home/huangxinran
scp liyang@10.0.21.134:/disk1/work/test1/sw5100-law-1-0_ap_standard_oem/boot_images/QcomPkg/SocPkg/AthertonPkg/Bin/LAA/RELEASE/xbl_config.elf /home/huangxinran
```

adb:
```
fastboot flash xbl xbl.elf
fastboot flash xbl_config xbl_config.elf
fastboot reboot
```

串口:

sudo minicom

gpio操作：
TLMMProtocol->ConfigGpio((UINT32)EFI_GPIO_CFG(69, 0, GPIO_OUTPUT, GPIO_PULL_UP, GPIO_2MA), TLMM_GPIO_ENABLE)
TLMMProtocol->GpioOut((UINT32)EFI_GPIO_CFG(69, 0, GPIO_OUTPUT, GPIO_NO_PULL, GPIO_2MA), GPIO_LOW_VALUE)

I2C：
i2c_status i2c_write_config(I2C_DriverCtx *pI2CDriverCtx, uint32 slave_address, I2C_Data_uint32 *pInitData, UINT32 size)

i2c_status i2c_write_data(i2c_instance eI2CCoreInstance, uint32 slave_address, uint32 address, uint32 value)

屏驱动：
进入/disk1/work/test1/sw5100-law-1-0_ap_standard_oem/boot_images
以9211为例，流程如下：
1.在/QcomPkg/SocPkg/AthertonPkg/Library/MDPPlatformLib中添加文件xbl_lt9211.h

2.在/QcomPkg/SocPkg/AthertonPkg/Library/MDPPlatformLib中，添加文件xbl_lt9211.c
在文件xbl_lt9211.c：

3.在目录/QcomPkg/Include/Library下：
MDPPlatformLib.h：
在MDPPlatformPanelType枚举类型中加入MDPPLATFORM_PANEL_DLP160_RGB360P_VIDEO;

4.在目录/QcomPkg/Library/MDPLib下：
DisplayUtils.c：
在static void CheckPanelOverride()的最后加入pDisplayContext->bOverride = ParsePanelOverrideCommand("dsi_dlp160_360p_video", &pDisplayContext->sDisplayParams);//解析覆盖字符串中加入我们需要的屏，注释掉不需要的

5.在目录/QcomPkg/Settings/Panel下添加.xml文件
Panel_dlp160_360p_vid.xml：
添加屏幕配置，一般修改 <HorizontalActive>640</HorizontalActive>, <VerticalActive>360</VerticalActive>, <DSIRefreshRate>0x5A0000</DSIRefreshRate>;

6.在目录/QcomPkg/SocPkg/AthertonPkg/Library/MDPPlatformLib下：
MDPPlatformLib.c：
1）添加*#include "xbl_lt9211.h"*
2）在 static PlatformDSIDetectParams uefiPanelList[]：添加dlp160屏参数
3）在const PanelDTInfoType fastBootPanelList[] =：添加 PANEL_CREATE_ENTRY("dsi_dlp160_360p_video",,,,)

```
4） 在MDP_Status MDPPlatformConfigure中，添加lt9211_Init();注释掉不用的JBD4020ManualInitialization(TRUE);

7.在目录/QcomPkg/SocPkg/AthertonPkg/Library/MDPPlatformLib
MDPPlatformLib.inf:
1)在 [Sources.common]中添加xbl_lt9211.c
2)在[Protocols] 中添加gQcomPmicVregProtocolGuid

8.在目录./SocPkg/AthertonPkg/Library/MDPPlatformLib/
MDPPlatformLibPanelConfig.h:
在MDPPlatformPanelFunctionTable sMDPPlatformPanelFunction[MDPPLATFORM_PANEL_MAX]中添加:
{ /* Atherton Panel */
    MDPPLATFORM_PANEL_DLP160_RGB360P_VIDEO,                    // ePanelSelected
    "Panel_dlp160_360p_vid.xml",                                // pPanelXmlConfig
    Panel_Default_PowerUp,
// pPanel_PowerUp
    Panel_Default_PowerDown,                                                // pPanel_PowerDown
    Panel_Default_Reset,
// pPanel_Reset
    NULL,                                          // pPanel_Peripheral_Power
    NULL,                                    // pPanel_Brightness_Enable
    NULL                                           // pPanel_Brightness_Level
  },
```

kernel

dts

```
以rt685为例:
1.看硬件框图及原理图，查看它属于什么（rt685属于挂在i2c sec0）

2.找到monaco.dts->monaco.dtsi->monaco-pinctrl.dtsi->/i2c找到sec0_i2c的表示->用grep命令查找有
  qupv3_se0_i2c的文件，找到符合添加条件的文件（monaco-audio-overlay.dtsi）
```

```
3.在monaco-audio-overlay.dtsi中的&qupv3_se0_i2c中，添加设备驱动;
nxp_rt685_codec: nxp_rt685@5e {
        status = "ok";
        compatible = "qcom,nxp_rt685";
        reg = <0x5e>;
        //qcom,sub-board-vdd-gpio = <&tlmm 42 0>;
        qcom,sub-board-dsp-rst-gpio = <&tlmm 9 0>;
        qcom,sub-board-dsp-int-gpio = <&tlmm 19 0>;
        qcom,sub-board-core-en-gpio = <&tlmm 55 0>;
        qcom,avdd-name = "avdd";
        vcc_i2c-supply = <&L21A>;
        avdd-supply = <&L28A>;


        qcom,cdc-micbias1-mv = <1800>;
        qcom,cdc-micbias2-mv = <1800>;
        qcom,cdc-static-supplies = "cdc-mic-bias";
        qcom,misc-name = "nxp_rt685";
    };
结点命名: [label:] node-name[@unit-address] {
    [properties definitions]
    [child nodes]
}
配置gpio：根据原理图查看要配置的gpio有哪些。


```

driver

i2c子系统

参考：https://blog.csdn.net/qq_53144843/article/details/127089880

https://blog.csdn.net/m0_64560763/article/details/127008302

读写

i2c框架

1. **I2C 核心**： I2C 核心提供了 I2C 总线驱动和设备驱动的注册、 注销方法
2. **I2C 总线驱动**：对 I2C 硬件体系结构中适配器端的实现， 适配器可由CPU 控制， 甚至可以直接集成在 CPU 内部。I2C 总线驱动就是 SOC 的 I2C 控制器驱动，也叫做 I2C 适配器驱动

3. **I2C 设备驱动**：对 I2C 硬件体系结构中设备端的实现， 设备一般挂接在受 CPU 控制的 I2C 适配器上， 通过 I2C 适配器与 CPU 交换数据

i2c总线驱动

```
I2C 总线驱动的重点是 I2C 适配器驱动，主要涉及到两个结构体： i2c_adapter 和 i2c_algorithm。
i2c_adapter 结构体来表示 I2C 适配器。
i2c_algorithm 是 I2C 适配器与 IIC 设备进行通信的方法。

 i2c_adapter 结构体定义在 include/linux/i2c.h 文件中：
 struct i2c_adapter {
    struct module *owner;
    unsigned int class; /* classes to allow probing for */
    const struct i2c_algorithm *algo; /* 总线访问算法 */
    void *algo_data;

    /* data fields that are valid for all devices */
    struct rt_mutex bus_lock;

    int timeout; /* in jiffies */
    int retries;
    struct device dev; /* the adapter device */

    int nr;
    char name[48];
    struct completion dev_released;

    struct mutex userspace_clients_lock;
    struct list_head userspace_clients;

    struct i2c_bus_recovery_info *bus_recovery_info;
    const struct i2c_adapter_quirks *quirks;
};

i2c_algorithm 结构体定义在 include/linux/i2c.h 文件中：
struct i2c_algorithm
{
    ......
    int (*master_xfer)(struct i2c_adapter *adap,struct i2c_msg *msgs,int num);
    int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr,
    unsigned short flags, char read_write,
    u8 command, int size, union i2c_smbus_data *data);

    /* To determine what the adapter supports */
    u32 (*functionality) (struct i2c_adapter *);
```

```
    ......
};

```

i2c设备驱动

```
在 I2C 设备驱动中主要有两个重要的结构体： i2c_client 和 i2c_driver。
i2c_client 是描述设备信息的，i2c_driver 描述驱动内容。

i2c_client 结构体定义在include/linux/i2c.h 文件中：
struct i2c_client {
    unsigned short flags; /* 标志 */
    unsigned short addr; /* 芯片地址， 7 位，存在低 7 位*/
    ......
    char name[I2C_NAME_SIZE]; /* 名字 */
    struct i2c_adapter *adapter; /* 对应的 I2C 适配器 */
    struct device dev; /* 设备结构体 */
    int irq; /* 中断 */
    struct list_head detected;
    ......
};

 i2c_driver 结构体定义在 include/linux/i2c.h 文件中：
 struct i2c_driver {
    unsigned int class;

    /* Notifies the driver that a new bus has appeared. You should
    * avoid using this, it will be removed in a near future.
    */
    int (*attach_adapter)(struct i2c_adapter *) __deprecated;

    /* Standard driver model interfaces */
    int (*probe)(struct i2c_client *, const struct i2c_device_id *);
    int (*remove)(struct i2c_client *);

    /* driver model interfaces that don't relate to enumeration */
    void (*shutdown)(struct i2c_client *);

    /* Alert callback, for example for the SMBus alert protocol.
    * The format and meaning of the data value depends on the
    * protocol.For the SMBus alert protocol, there is a single bit
    * of data passed as the alert response's low bit ("eventflag"). */
    void (*alert)(struct i2c_client *, unsigned int data);
    /* a ioctl like command that can be used to perform specific
    * functions with the device.
```

```
     */
     int (*command)(struct i2c_client *client, unsigned int cmd,void *arg);

     struct device_driver driver;
     const struct i2c_device_id *id_table;

     /* Device detection callback for automatic device creation */
     int (*detect)(struct i2c_client *, struct i2c_board_info *);
     const unsigned short *address_list;
     struct list_head clients;
};

使用i2c_register_driver 函数向 Linux 内核中注册 i2c 设备:
int i2c_register_driver(struct module *owner,struct i2c_driver *driver)


```

i2c核心

```
在 I2C 核心层完成的是 I2C 设备和I2C 驱动的匹配过程。
I2C 核心部分的文件是drivers/i2c/i2c-core.c。

I2C 核心层提供了一些与硬件无关的 API 函数。

i2c_adapter 注册/注销函数:
int i2c_add_adapter(struct i2c_adapter *adapter)
int i2c_add_numbered_adapter(struct i2c_adapter *adap)
void i2c_del_adapter(struct i2c_adapter * adap)

i2c_driver 注册/注销函数:
int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
int i2c_add_driver (struct i2c_driver *driver)
void i2c_del_driver(struct i2c_driver *driver)


```

gpio子系统

当我们在驱动代码中要使用内核中提供的GPIO子系统，需要在驱动代码中包含<linux/gpio.h>头文件。

关于API接口函数的实现在内核源码drivers/gpio/gpiolib.c文件中。

常用的API接口：

1. 函数gpio_is_valid()用来判断获取到的gpio号是否有效

```
 static inline bool gpio_is_valid(int number)
{
    return number >= 0 && number < ARCH_NR_GPIOS;
}
```

2. gpio_request()和gpio_free()函数用来向系统中申请GPIO和释放已经申请的GPIO，在函数gpio_request()中传入的形参中，gpio为IO号，label为向系统中申请GPIO使用的标签，类似于GPIO的名称。

```
/* Always use the library code for GPIO management calls,
 * or when sleeping may be involved.
 */
extern int gpio_request(unsigned gpio, const char *label);
extern void gpio_free(unsigned gpio);
```

3. 结构体struct gpio用来描述一个需要配置的GPIO。

```
/**
 * struct gpio - a structure describing a GPIO with configuration
 * @gpio:    the GPIO number
 * @flags:    GPIO configuration as specified by GPIOF_*
 * @label:    a literal description string of this GPIO
 */
struct gpio {
    unsigned    gpio;
    unsigned long    flags;
    const char    *label;
};
```

4. 函数gpio_request_one()用来申请单个GPIO，但是在申请的时候可以设置flag标志，例如，该函数在申请GPIO资源的同时，直接将GPIO的方向设置为输入或者输出，函数gpio_request_array()和gpio_free_array()用来向系统中申请或者释放多个GPIO资源。

```
extern int gpio_request_one(unsigned gpio, unsigned long flags, const char
*label);
extern int gpio_request_array(const struct gpio *array, size_t num);
extern void gpio_free_array(const struct gpio *array, size_t num);
```

5. 函数带有devm_前缀，也就是说，这是带设备资源管理版本的函数，因此在使用函数时，需要指定设备的struct device指针。

```
/* CONFIG_GPIOLIB: bindings for managed devices that want to request gpios */

struct device;

int devm_gpio_request(struct device *dev, unsigned gpio, const char *label);
int devm_gpio_request_one(struct device *dev, unsigned gpio,
                unsigned long flags, const char *label);
void devm_gpio_free(struct device *dev, unsigned int gpio);
```

6. 函数gpio_direction_input()用来设置GPIO的方向为输入，函数gpio_direction_output()用来设置GPIO的方向为输出，并且通过value值可以设置输出的电平

```
static inline int gpio_direction_input(unsigned gpio)
{
    return gpiod_direction_input(gpio_to_desc(gpio));
}
static inline int gpio_direction_output(unsigned gpio, int value)
{
    return gpiod_direction_output_raw(gpio_to_desc(gpio), value);
}
```

7. 将GPIO的方向设置为输入时，可以使用函数gpio_get_value()来获取当前的IO口电平值，当GPIO的方向设置为输出时，使用函数gpio_set_value()可以设置IO口的电平值。

```
static inline int gpio_get_value(unsigned int gpio)
{
    return __gpio_get_value(gpio);
}

static inline void gpio_set_value(unsigned int gpio, int value)
{
    __gpio_set_value(gpio, value);
}
```

pinctrl子系统

Sysfs

https://www.cnblogs.com/lifexy/p/9799778.html

input子系统

参考：https://blog.csdn.net/ldl617/article/details/117450573

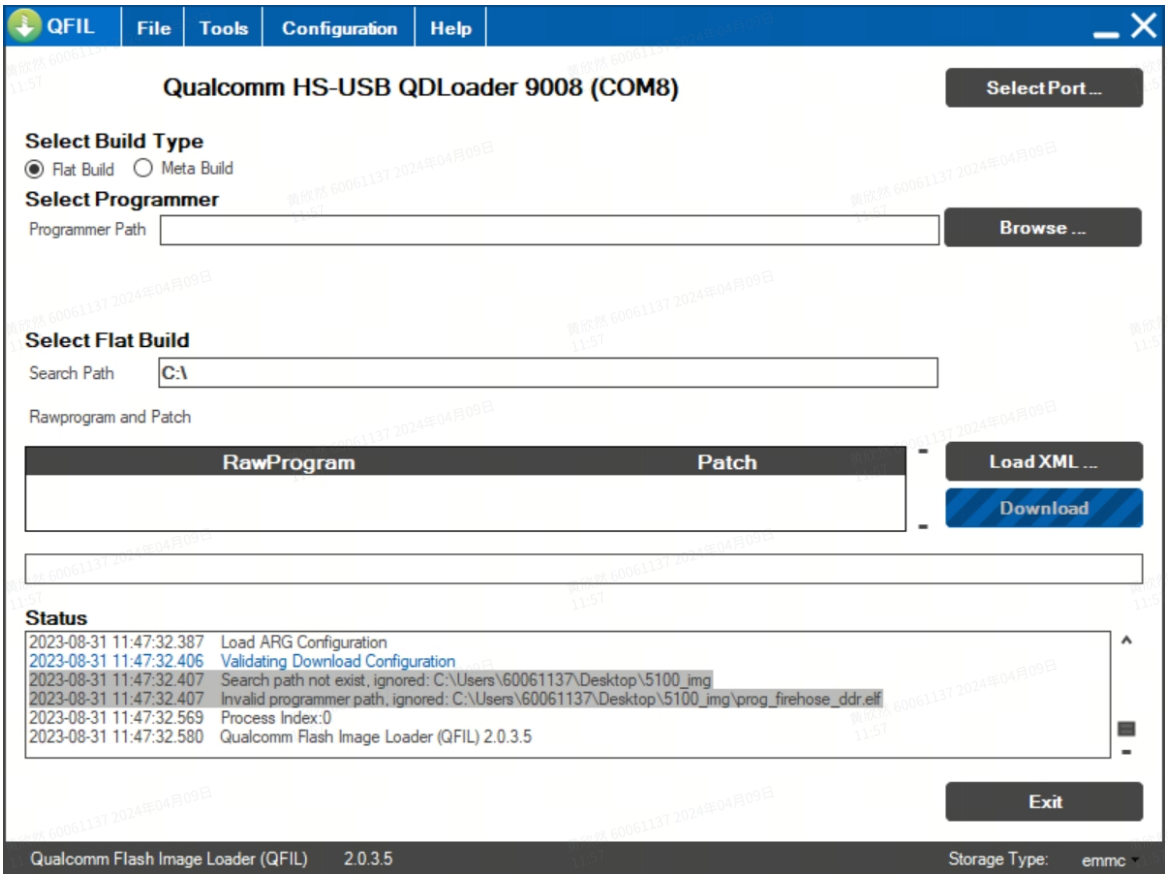
input要实现自动重复，首先要支持事件EV_REP

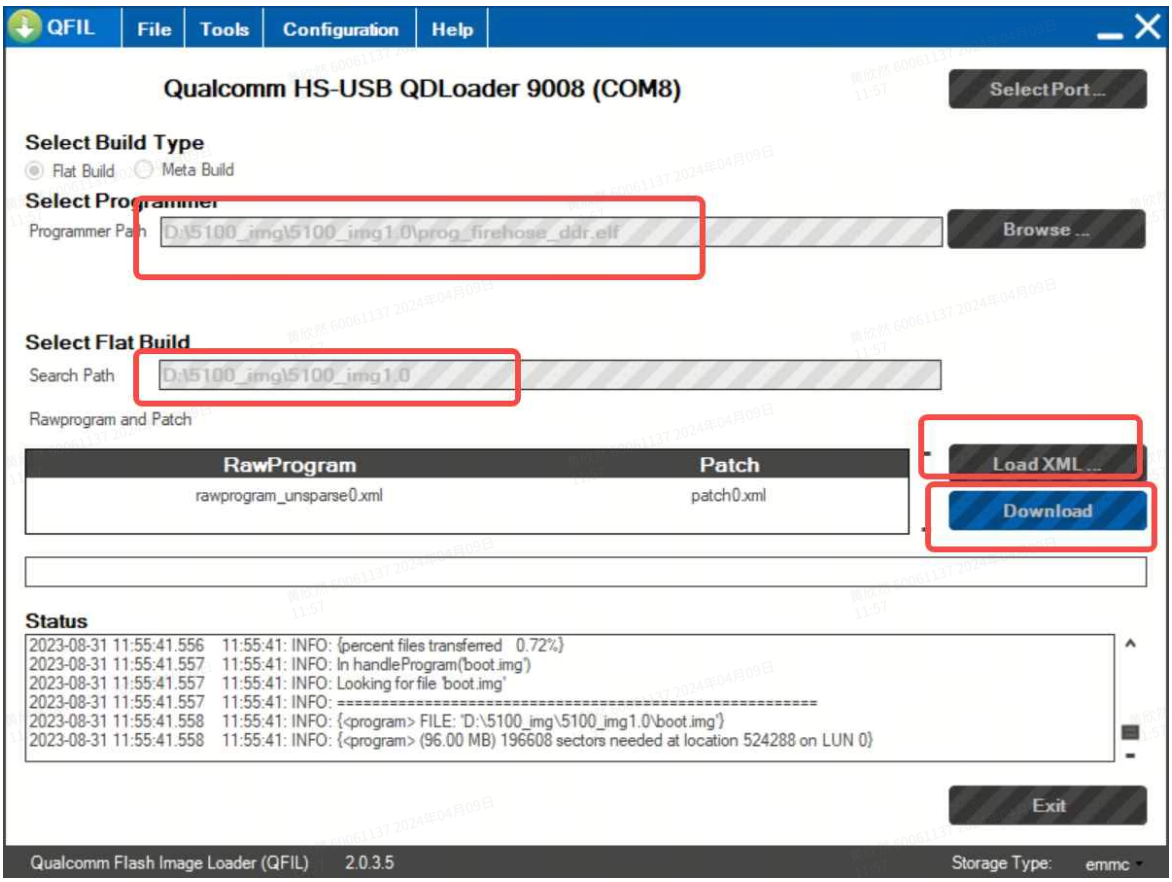
自动上报的流程如下：

input_event() -> input_handle_event() -> input_pass_values()

Qfil

```
短接edl或输入adb reboot edl
SelectPort 9008
选择flat Build
右下角选择emmc
browse选择ddr
```


QFIL
File
Tools
Configuration
Help
Qualcomm HS-USB QDLoader 9008 (COM8)
Select Port ...
Select Build Type
Flat Build
Meta Build
Select Programmer
Programmer Path
Browse ...
Select Flat Build
Search Path
C:\
Rawprogram and Patch
RawProgram
Patch
Load XML ...
Download
Status
2023-08-31 11:47:32.387 Load ARG Configuration
2023-08-31 11:47:32.406 Validating Download Configuration
2023-08-31 11:47:32.407 Search path not exist, ignored: C:\Users\60061137\Desktop\5100_img
2023-08-31 11:47:32.407 Invalid programmer path, ignored: C:\Users\60061137\Desktop\5100_img\prog_firehose_ddr.elf
2023-08-31 11:47:32.569 Process Index:0
2023-08-31 11:47:32.580 Qualcomm Flash Image Loader (QFIL) 2.0.3.5
Exit
Qualcomm Flash Image Loader (QFIL) 2.0.3.5
Storage Type:
emmc


QFIL
File
Tools
Configuration
Help
Qualcomm HS-USB QDLoader 9008 (COM8)
Select Port ...
Select Build Type
Flat Build
Meta Build
Programmer Path
D:\5100_img\5100_img1.0\prog_firehose_ddr.elf
Browse ...
Select Flat Build
Search Path
D:\5100_img\5100_img1.0
Rawprogram and Patch
RawProgram
Patch
rawprogram_unsparse0.xml
patch0.xml
Load XML ...
Download
Status
2023-08-31 11:55:41.556 11:55:41: INFO: {percent files transferred 0.72%}
2023-08-31 11:55:41.557 11:55:41: INFO: In handleProgram('boot.img')
2023-08-31 11:55:41.557 11:55:41: INFO: Looking for file 'boot.img'
2023-08-31 11:55:41.557 11:55:41: INFO: ==========
2023-08-31 11:55:41.558 11:55:41: INFO: {<program> FILE: 'D:\5100_img\5100_img1.0\boot.img'}
2023-08-31 11:55:41.558 11:55:41: INFO: {<program> (96.00 MB) 196608 sectors needed at location 524288 on LUN 0}
Exit
Qualcomm Flash Image Loader (QFIL) 2.0.3.5
Storage Type:
emmc

middleware

```
把ArGlassMiddlewareClient放在目录/system/priv-app/ArMiddleware，名字为ArGlassMiddleware.apk


手动开启app服务
am startservice -n com.aac.arglass.middlewareclient/.service.ArGlassMainService

su

关闭selinux防火墙
setenforce 0

chmod 777 /dev/rt685

app权限修改：
ArGlassMiddlewareClient的ArGlassMainService和DeviceBootService

jar包在：
/disk1/work/test1/sw5100_for_ar/android/frameworks/base/services
里面有arglass_server.jar和arglass_util.jar


```

以调试tpl1401的寄存器为例：

```
ioctl:提供ioctl的封装类
1.在ioctl中添加一个模块：nactivetplvol
其中有一个.cpp文件和一个.java文件
nativetplvol.cpp:使用JNI（Java Native Interface），创建NativeLib.open等方法
NativeLib.java:创建NativeLib类，声明open等方法
NativeTTplVol.java:实现了NativeLib的单例模式

2.在app中添加模块，在main.java.com中加入tplvol.ioctl包，实现测试程序MainActivity2和测试页面activity_main.xml

ArGlassMiddleware:
arglassapi
arglassserver
arglassutil

ArGlassMiddlewareClient:
```

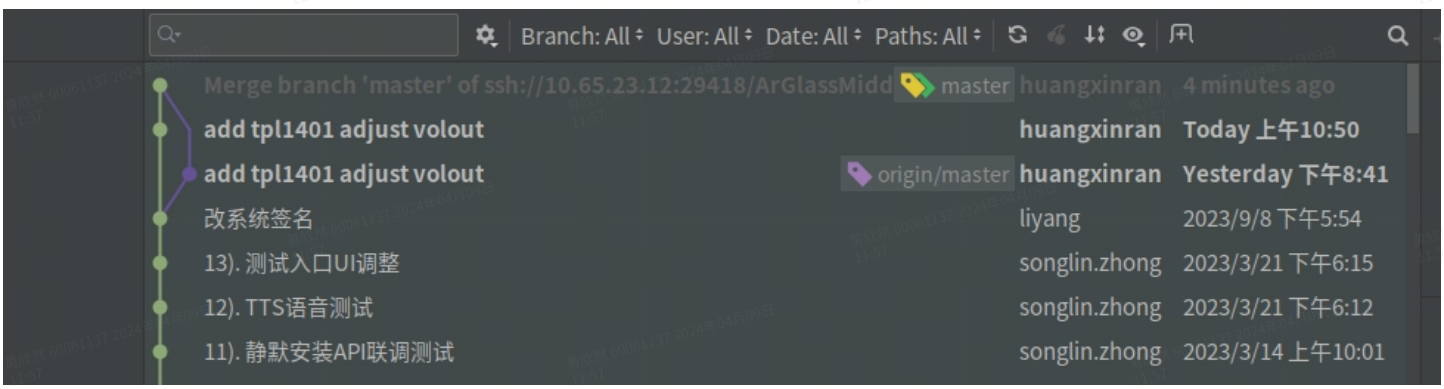
```
1.在service的ArGlassMainService.java的on create方法中
添加registerTplOutVolCallback注册一个回调函数或监听器
```

```
android调试：
Tasks/other目录下：
make jar -->jar包，放到libs
assembleRelease-->aar库，放到aar
                -->apk


git提交：
git log
git config remote.origin.push refs/heads/*:refs/for/*
git push origin master
等待gerrit管理员同意commit
git pull

git回滚：
git reset --soft log号
从缓冲区全部提交：
git commit -m [message]
```

git修复冲突（当你的git和服务器不在一个分支时）：




```
提示冲突发生的地方：
git pull
```

```
修改冲突发生的地方为你最后想要的结果:
git diff [conflict file]
修改后提交:
git add  [conflict file]
git commit  [conflict file]

```

cpu

```
察看cpu主频:
cd ./sys/devices/system/cpu/cpu0/cpufreq

```

服务器

```
登录编译服务器
ssh username@10.161.137.100
ssh huangxinran@10.161.137.100
```

```
下载rk3308代码
repo init -u ssh://username@10.161.137.101:29418/rk3308b/manifest.git
repo sync

repo start master --all

```

一些命令

```
对比文件:
```

```
vimdiff file1 file2
```

link_phone_connect

Git

```
git status
git log
git pull orgin master

```

ipd调试

rk主机写：

```
i2cset -f -y 2 0x72 0x02 0x01 b
i2cset -f -y 2 0x72 0x08 0x157C
i2cset -f -y 2 0x72 0x01 0x01 b
```

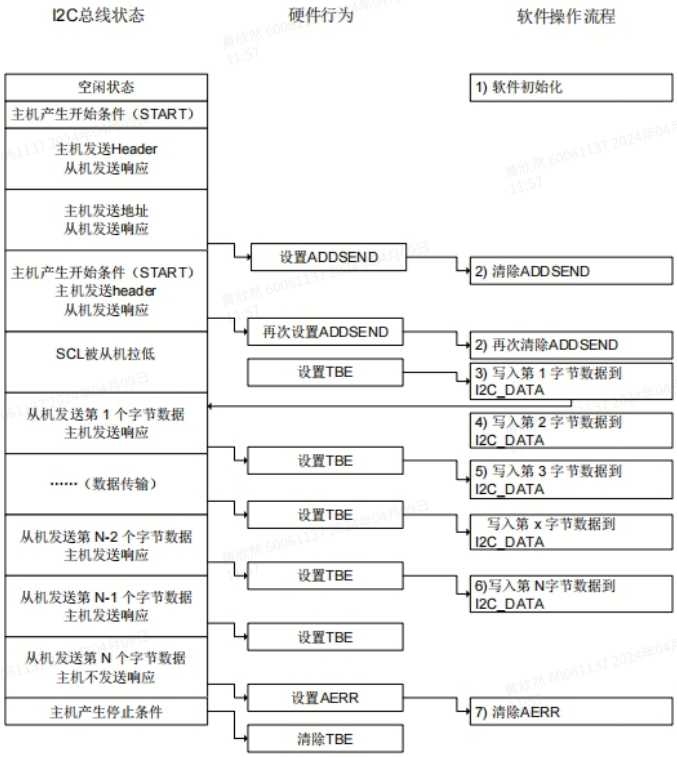
rk主机读：

```
i2cget -f -y 2 0x72 0x01 b（暂时只能读单次）

i2cget -f -y 2 0x72 0x01 c
i2cget -f -y 2 0x72 0x02 c
```

从机：

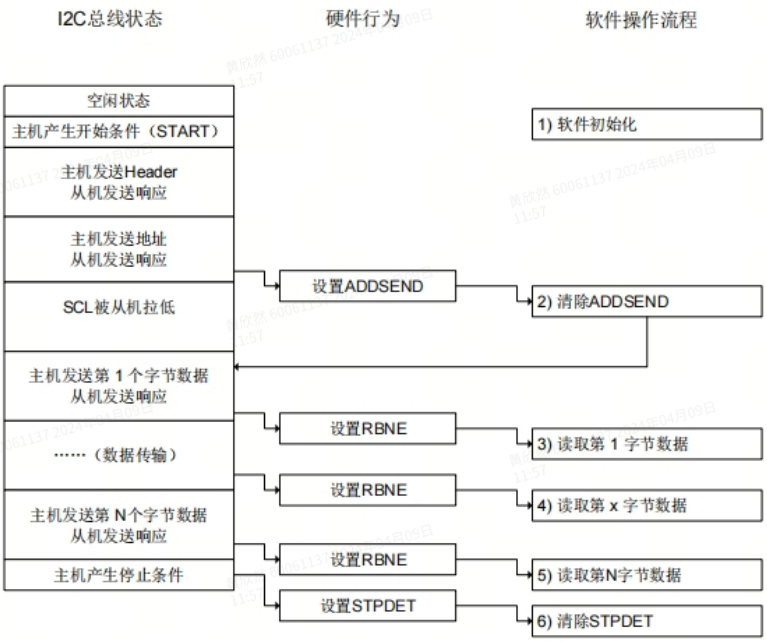
图 19-9. 从机发送模式（10 位地址模式）



目前log:

```
777
   go to interruppt I2C_INT_FLAG_RBNE
                                    777
                                       go to interruppt I2C_INT_FLAG_TBE
                                                                    go to interruppt I2C_INT_FLAG_TBE
                                                                                                    go to i
terruppt I2C_INT_FLAG_AERR
                            16 times sample, 4 bits shift: 0x0
```

图 19-10. 从机接收模式（10 位地址模式）



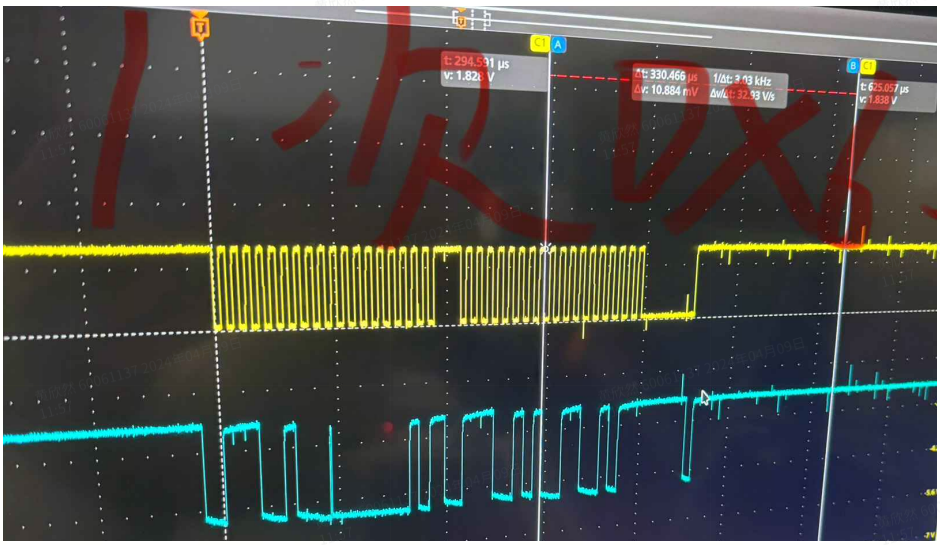
Log:

```
                              16 times sample, 4 bits shift: 0x0
The value of receive num is: 0
                              777
                                 go to interruppt I2C_INT_FLAG_RBNE
                                                                    go to interruppt I2C_INT_FLAG_RBNE
                                                                                                      go to interru
ppt I2C_INT_FLAG_RBNE
                        go to interruppt I2C_INT_FLAG_STPDET
                                                             16 times sample, 4 bits shift: 0x0
The value of receive num is: 2
                              16 times sample, 4 bits shift: 0x0
The value of receive num is: 2
                              16 times sample, 4 bits shift: 0x0
```

只能读单次问题描述：

第一次读正常，如何第二次读发送完地址位会把i2c电压拉低

第一次波形：



第二次波形：

解决：

由波形知，第二次读时i2c被拉低，尝试修改I2C_CTL0的ss位，阻止从机拉低i2c

19.4.1. 控制寄存器 0（I2C_CTL0）

地址偏移：0x00

复位值：0x0000 0000

该寄存器可以按半字（16位）或字（32位）访问。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRESET	保留	SALT	PECTRANS	POAP	ACKEN	STOP	START	SS	GCEN	PECEN	ARPEN	SMBSEL	保留	SMBEN	I2CEN
rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw	rw

7	SS	在从机模式下数据未就绪是否将 SCL 拉低 软件置 1 和清0。 0：拉低 SCL 1：不拉低 SCL

```
在主函数加上：
i2c_stretch_scl_low_config(I2C1, I2C_SCLSTRETCH_DISABLE);
之后可以多次读了
```

ipd git

```
git pull origin master
git add ipd_app_bootloader
git log
git commit -m "ipd app bootloader"
git push origin master//直接push到服务器

git reset --hard HEAD硬复位，还原修改，当commit了错误的东西时这个指令可以还原代码到最初
git reset --soft HEAD^软复位，当commit了错误的东西时这个指令可以撤回add和commit

git push origin HEAD:refs/for/master//待审查
```

i2c固件升级功能，程序由A区跳到B区之后，进入不了SysTick_Handler中断

```
解决：地址偏移
SCB->VTOR = FLASH_BASE | 0x2800;
GD给出的解决方法：开关总中断
__disable_irq();
__enable_irq();
```

CRC校验

```
// CRC-8 算法
uint8_t calculateCRC8(const uint8_t *data, size_t length) {
    const uint8_t polynomial = 0x1D; // CRC-8算法的多项式
    uint8_t crc = 0;
                int i,j;

    for (i = 0; i < length; ++i) {
        crc ^= data[i];

        for (j = 0; j < 8; ++j) {
            crc = (crc & 0x80) ? (crc << 1) ^ polynomial : (crc << 1);
        }
    }

    return crc;
}
```

休眠：gpio中断唤醒电流1.1ma（本来500ua，加上唤醒1.1ma）

```
/* disable the I2C1 interrupt */
    //关闭i2c中断
    i2c_interrupt_disable(I2C1, I2C_INT_ERR);
    i2c_interrupt_disable(I2C1, I2C_INT_EV);
    i2c_interrupt_disable(I2C1, I2C_INT_BUF);
    //gpio设置成模拟输出（除了jlink）
    gpio_mode_set(GPIOA, GPIO_MODE_ANALOG, GPIO_PUPD_NONE, GPIO_PIN_ALL&
(!GPIO_PIN_0)&(!GPIO_PIN_4)&(!GPIO_PIN_13)&(!GPIO_PIN_14));
```

```
    gpio_mode_set(GPIOB, GPIO_MODE_ANALOG, GPIO_PUPD_NONE, GPIO_PIN_ALL&(!GPIO_PIN_11));
    gpio_mode_set(GPIOC, GPIO_MODE_ANALOG, GPIO_PUPD_NONE, GPIO_PIN_ALL);
    gpio_mode_set(GPIOD, GPIO_MODE_ANALOG, GPIO_PUPD_NONE, GPIO_PIN_ALL);
    gpio_mode_set(GPIOF, GPIO_MODE_ANALOG, GPIO_PUPD_NONE, GPIO_PIN_ALL);

    rcu_periph_clock_enable(RCU_PMU);
    //关闭adc
    adc_disable();
//    usart_disable(USART0);
//    usart_deinit(USART0);
    //hall sleep拉低
    gpio_mode_set(GPIOA, GPIO_MODE_OUTPUT, GPIO_PUPD_NONE, GPIO_PIN_0);
    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_0);
    aw8646_set_gpio(GPIOA, GPIO_PIN_0, AW_GPIO_LOW);
//    aw8646 sleep拉低
//    gpio_mode_set(GPIOA, GPIO_MODE_OUTPUT, GPIO_PUPD_NONE, GPIO_PIN_4);
//    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_4);
//    aw8646_set_gpio(GPIOA, GPIO_PIN_4, AW_GPIO_LOW);

    printf("into deepsleepmode\r\n");
    pmu_to_deepsleepmode(PMU_LDO_LOWPOWER, PMU_LOWDRIVER_DISABLE, WFI_CMD);
    //唤醒后主频不对，需重新配置时钟
    system_clock_84m_hxtal();
//    重新打开gpio（似乎不需要？）
//    rcu_periph_clock_enable(RCU_GPIOA);
//    rcu_periph_clock_enable(RCU_GPIOB);
//    rcu_periph_clock_enable(RCU_GPIOC);
//    rcu_periph_clock_enable(RCU_GPIOD);
//    rcu_periph_clock_enable(RCU_GPIOF);

//    gpio_mode_set(GPIOA, GPIO_MODE_AF, GPIO_PUPD_PULLUP, GPIO_PIN_ALL);
//    gpio_mode_set(GPIOB, GPIO_MODE_AF, GPIO_PUPD_PULLUP, GPIO_PIN_ALL);
//    gpio_mode_set(GPIOC, GPIO_MODE_AF, GPIO_PUPD_PULLUP, GPIO_PIN_ALL);
//    gpio_mode_set(GPIOD, GPIO_MODE_AF, GPIO_PUPD_PULLUP, GPIO_PIN_ALL);
//    gpio_mode_set(GPIOF, GPIO_MODE_AF, GPIO_PUPD_PULLUP, GPIO_PIN_ALL);
    //hall sleep拉高
    gpio_mode_set(GPIOA, GPIO_MODE_OUTPUT, GPIO_PUPD_NONE, GPIO_PIN_0);
    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_0);
    aw8646_set_gpio(GPIOA, GPIO_PIN_0, AW_GPIO_HIGH);
//    aw8646 sleep拉高
//    gpio_mode_set(GPIOA, GPIO_MODE_OUTPUT, GPIO_PUPD_NONE, GPIO_PIN_4);
//    gpio_output_options_set(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_4);
```

```
50 //    aw8646_set_gpio(GPIOA, GPIO_PIN_4, AW_GPIO_HIGH);
51
52 //    adc_enable();
53     //重新初始化adc
54     adc_init();
55 //    usart_enable(USART0);
56 //    usart0_config();
57     systick_config();
58     i2c_init();
59
60     printf("out of deepsleepmode\r\n");
```

hall线性拟合曲线

```
1 https://blog.csdn.net/einstein10147/article/details/79205109
```

模式切换

1. 定义切换条件：

确定触发模式切换的条件。这些条件可能包括：

- 用户手动操作： Pogo PIN切换到车载内置音箱模式
- 环境条件：例如，车内DHU连接判断为车载DHU连接，通过车的wifi ssid?还是蓝牙mac地址？还是用户在app自己切换?
- 连接状态： 根据与SMART音箱或其他设备的连接状态触发模式切换。在失去与车载SMART音箱的连接时，切换到家庭模式。

2. 设计事件监听：

设置监听线程还是直接在主循环run中直接检测模式切换?

3. 触发模式切换：

一旦检测到触发条件，执行相应的模式切换操作。这可能涉及修改状态变量、调用相关函数、切换蓝牙配置等。

4. 考虑优先级：

如果存在多个触发条件同时满足，车内DHU连接和 Pogo PIN触发哪个优先级更高？默认家庭模式。

5. 异常处理：

比如在切换过程中出现错误，实现适当的错误处理机制，例如回滚到之前的模式或者提供用户通知。

6. 用户反馈：

是否让用户清楚地了解当前的工作模式切换，可能通过音频提示？

```cpp
#include <iostream>
#include <thread>
#include <chrono>

class BluetoothSpeaker {
public:
    BluetoothSpeaker() : mode("car") {}

    void manualModeSwitch(const std::string& newMode) {
        mode = newMode;
        updateMode();
    }

    bool environmentConditionCheck() {
        // 检查车辆状态、连接状态等环境条件
        // 返回true表示满足条件，可以切换模式
        // 返回false表示不满足条件
        return true; // Placeholder, 实际应根据具体条件实现
    }

    bool timeConditionCheck() {
        // 检查时间条件
        // 返回true表示满足条件，可以切换模式
        // 返回false表示不满足条件
        return true; // Placeholder, 实际应根据具体条件实现
    }

    void eventListener() {
        while (true) {
            if (environmentConditionCheck() || timeConditionCheck()) {
                // 触发模式切换
                mode = (mode == "car") ? "home" : "car";
                updateMode();
            }
            // 等待一段时间后重新检查条件
            std::this_thread::sleep_for(std::chrono::seconds(10));
```

```
        }
    }

    void updateMode() {
        // 执行模式切换的相关操作
        std::cout << "Switched to " << mode << " mode." << std::endl;
    }

private:
    std::string mode;
};

int main() {
    // 创建BluetoothSpeaker对象
    BluetoothSpeaker bluetoothSpeaker;

    // 启动事件监听
    std::thread listenerThread(&BluetoothSpeaker::eventListener,
&bluetoothSpeaker);

    // 模拟用户手动切换
    std::this_thread::sleep_for(std::chrono::seconds(30));
    bluetoothSpeaker.manualModeSwitch("home");

    // 等待事件监听线程结束（实际应该使用更复杂的线程管理）
    listenerThread.join();

    return 0;
}

```

```
#include <iostream>
#include <chrono>
#include <thread>

// 定义系统的工作模式
enum SystemMode {
    MODE_HOME,
    MODE_CAR_DHU,
    MODE_CAR_INTERNAL,
    MODE_COUNT // 用于计数模式数量
};

```

```
// 定义事件监听类
class EventListener {
public:
    // 检测用户手动操作或Pogo PIN插入的模拟函数
    bool isUserActionTriggered() {
        // 在实际应用中，通过实际的用户操作或Pogo PIN读取来判断状态
        // 这里仅作为示例，返回一个模拟值
        return true;  // 用户手动操作或Pogo PIN插入
    }

    // 检测与DHU连接的模拟函数
    bool isDHUConnected() {
        // 在实际应用中，可能通过蓝牙状态、DHU传感器等方式实现
        // 这里仅作为示例，返回一个模拟值
        return true;
    }

    // 检测失去与SMART音箱连接的模拟函数
    bool hasLostSmartSpeakerConnection() {
        // 在实际应用中，可能通过蓝牙状态、连接状态等方式实现
        // 这里仅作为示例，返回一个模拟值
        return false;
    }

    // 检测当前时间的模拟函数
    bool isCurrentTimeInSpecifiedRange() {
        // 在实际应用中，可能通过实时时钟等方式实现
        // 这里仅作为示例，返回一个模拟值
        return true;
    }
};


class ModeSwitcher {
private:
    SystemMode currentMode;
    EventListener eventListener;

public:
    ModeSwitcher() : currentMode(MODE_HOME) {}

    // 切换模式，可能抛出异常
    void switchMode(SystemMode newMode) {
        // 在这里添加执行模式切换的相关操作
        // 例如，修改硬件配置、启动/停止相应功能、切换蓝牙配置等

        // 模拟在切换过程中可能出现的异常
```

```
60          bool simulateError = false; // 将此值设置为 true 以模拟异常
61
62          if (simulateError) {
63              throw std::runtime_error("Error during mode switch");
64          }
65
66          // 更新当前模式
67          currentMode = newMode;
68
69          // 在这里添加适当的用户反馈，例如音频提示
70          std::cout << "Switched to mode: " << currentMode << std::endl;
71      }
72
73      // 检测模式切换条件
74      void checkModeSwitchConditions() {
75          // 1. 用户手动操作或Pogo PIN插入
76          if (eventListener.isUserActionTriggered()) {
77              try {
78                  switchMode(MODE_CAR_INTERNAL);
79              } catch (const std::exception& e) {
80                  // 处理异常，例如提供用户通知或回滚到之前的模式
81                  std::cerr << "Exception during mode switch: " << e.what() <<
    std::endl;
82                  rollbackToPreviousMode();
83              }
84              return;
85          }
86
87          // ...（其他条件检测，类似地添加异常处理）
88
89      }
90
91      // 回滚到之前的模式
92      void rollbackToPreviousMode() {
93          // 在这里执行回滚的操作，例如重新设置硬件配置等
94          // 这里只是一个示例，实际操作根据应用需求进行
95          std::cout << "Rolling back to previous mode: " << currentMode <<
    std::endl;
96      }
97
98      // 在主循环中调用此方法，替代原先的 run 方法
99      void processInMainLoop() {
100         // 监听事件，检测模式切换条件
101         checkModeSwitchConditions();
102
103         // 在这里执行其他系统任务
104         // ...
```

```
105
106         // 模拟一些延迟，以模拟实际应用中的轮询或事件驱动
107         std::this_thread::sleep_for(std::chrono::milliseconds(100));
108     }
109 };
110
111 int main() {
112     // 创建模式切换器对象
113     ModeSwitcher modeSwitcher;
114
115     // 在现有的主循环中调用处理方法
116     while (true) {
117         // 在这里执行现有的主循环任务
118
119         // 调用处理方法，处理模式切换和其他任务
120         modeSwitcher.processInMainLoop();
121     }
122
123     return 0;
124 }
125
```

```
1  #include <iostream>
2  #include <chrono>
3  #include <thread>
4
5  // 定义系统的工作模式
6  enum SystemMode {
7      MODE_HOME,
8      MODE_CAR_DHU,
9      MODE_CAR_INTERNAL,
10     MODE_COUNT // 用于计数模式数量
11 };
12
13 class ModeSwitcher {
14 private:
15     SystemMode currentMode;
16
17     // 模拟事件监听的状态
18     bool isUserActionTriggered;
19     bool isDHUConnected;
20     bool hasLostSmartSpeakerConnection;
21
22 public:
```

```
23      ModeSwitcher() : currentMode(MODE_HOME), isUserActionTriggered(false),
    isDHUConnected(false), hasLostSmartSpeakerConnection(false) {}
24
25      // 检测模式切换条件
26      void checkModeSwitchConditions() {
27          // 1. 用户手动操作或Pogo PIN插入
28          if (isUserActionTriggered) {
29              try {
30                  switchMode(MODE_CAR_INTERNAL);
31              } catch (const std::exception& e) {
32                  // 处理异常，例如提供用户通知或回滚到之前的模式
33                  std::cerr << "Exception during mode switch: " << e.what() <<
    std::endl;
34                  rollbackToPreviousMode();
35              }
36              return;
37          }
38
39          // 2. 与DHU连接
40          if (isDHUConnected) {
41              try {
42                  switchMode(MODE_CAR_DHU);
43              } catch (const std::exception& e) {
44                  std::cerr << "Exception during mode switch: " << e.what() <<
    std::endl;
45                  rollbackToPreviousMode();
46              }
47              return;
48          }
49
50          // 3. 失去与SMART音箱连接
51          if (hasLostSmartSpeakerConnection) {
52              try {
53                  switchMode(MODE_HOME);
54              } catch (const std::exception& e) {
55                  std::cerr << "Exception during mode switch: " << e.what() <<
    std::endl;
56                  rollbackToPreviousMode();
57              }
58              return;
59          }
60
61          // ...（其他条件检测，类似地添加异常处理）
62      }
63
64      // 切换模式，可能抛出异常
65      void switchMode(SystemMode newMode) {
```

```
66          // 在这里添加执行模式切换的相关操作
67          // 例如，修改硬件配置、启动/停止相应功能、切换蓝牙配置等
68
69          // 模拟在切换过程中可能出现的异常
70          bool simulateError = false; // 将此值设置为 true 以模拟异常
71
72          if (simulateError) {
73              throw std::runtime_error("Error during mode switch");
74          }
75
76          // 更新当前模式
77          currentMode = newMode;
78
79          // 在这里添加适当的用户反馈，例如音频提示
80          std::cout << "Switched to mode: " << currentMode << std::endl;
81      }
82
83      // 回滚到之前的模式
84      void rollbackToPreviousMode() {
85          // 在这里执行回滚的操作，例如重新设置硬件配置等
86          // 这里只是一个示例，实际操作根据应用需求进行
87          std::cout << "Rolling back to previous mode: " << currentMode <<
    std::endl;
88      }
89
90      // 模拟事件监听的状态更新
91      void updateEventListenerState(bool userAction, bool dhuConnected, bool
    lostConnection) {
92          isUserActionTriggered = userAction;
93          isDHUConnected = dhuConnected;
94          hasLostSmartSpeakerConnection = lostConnection;
95      }
96
97      // 在主循环中调用此方法，替代原先的 run 方法
98      void processInMainLoop() {
99          // 模拟事件监听的状态更新
100         updateEventListenerState(/* 获取实际状态的函数调用 */);
101
102         // 监听事件，检测模式切换条件
103         checkModeSwitchConditions();
104
105         // 在这里执行其他系统任务
106         // ...
107
108         // 模拟一些延迟，以模拟实际应用中的轮询或事件驱动
109         std::this_thread::sleep_for(std::chrono::milliseconds(100));
110     }
```

```
};

int main() {
    // 创建模式切换器对象
    ModeSwitcher modeSwitcher;

    // 在现有的主循环中调用处理方法
    while (true) {
        // 在这里执行现有的主循环任务

        // 调用处理方法，处理模式切换和其他任务
        modeSwitcher.processInMainLoop();
    }

    return 0;
}

```

BT测试

1.经典蓝牙连接：

测试条件：单独打开a2dp或打开所有

结论：对于一台没有测试过经典蓝牙的手机，无论单独打开a2dp或打开所有模式，第一次连接经典蓝牙的时候大概率会失败。会出现nvram中key检索失败的错误。

[SMARTSPEAKER_APP][cypress_bt_write_eir]:a2dp_sink_write_eir failed, ret=0x1fa6

[SMARTSPEAKER_APP][cypress_management_callback]:Write EIR Failed

[SMARTSPEAKER_APP][SendTextTraceToSpy]:00:02:54.702 wiced_bt_avrc_ct_init: [3]

[SMARTSPEAKER_APP][wiced_internal_btm_management_cback]:
[wiced_internal_btm_management_cback] event:0x14

[SMARTSPEAKER_APP][cypress_management_callback]:cypress management callback: 14

[SMARTSPEAKER_APP][cypress_management_callback]:Linkkey request, BDA:

[SMARTSPEAKER_APP][print_bd_address]:84:D3:28:5F:6D:ED

[SMARTSPEAKER_APP][cypress_management_callback]:Key retrieval failure

ios

2.先连BLE，后连经典蓝牙

结论：先连BLE，再连经典蓝牙(A2DP/HFP)，正常。

1.1.先连经典蓝牙(A2DP/HFP)，后连BLE，ble连不上

Android

2.先连BLE，后连经典蓝牙

结论：先连BLE，后连经典蓝牙，经典蓝牙配对不上

Ble:

ios有时候连不上，无设备，安卓连过之后就能连上了，安卓也有小概率连不上

wifi：

用脚本断开WiFi后没显示断连

wifi_state_connect状态变化有问题，连上wifi，然后断开后，还显示在连接中