基于 LPC55S69 平台的

多媒体控制系统

修订记录：

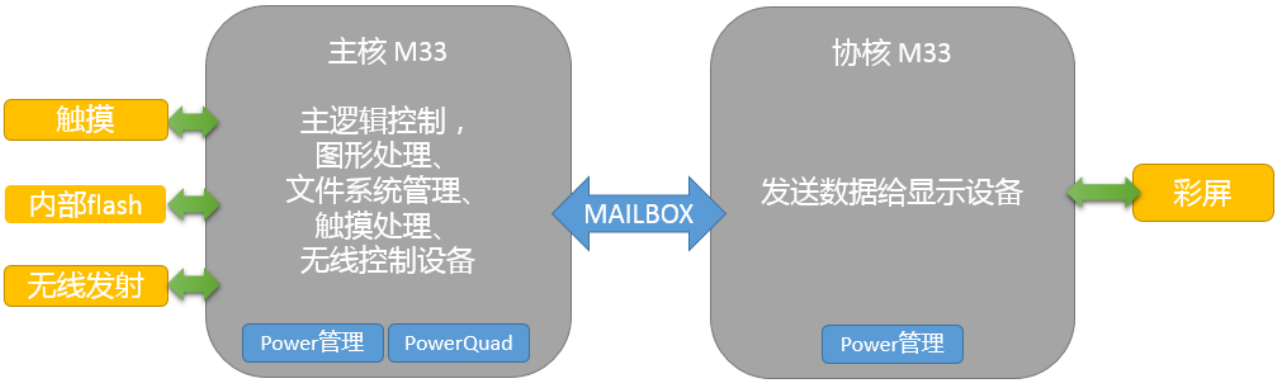
日期	说明	修改人
2019.12.29	创建文档	JACE
2019.12.31	完善	JACE
2020.01.01	完善	JACE

1 项目概述

多媒体控制系统基于 LPC55s69 主控，使用 3.2 寸触摸彩屏做为人机交互，旨在让用户通过简单的触摸即可实现对设备的控制，如控制室内的灯光、音乐、空调等设备。

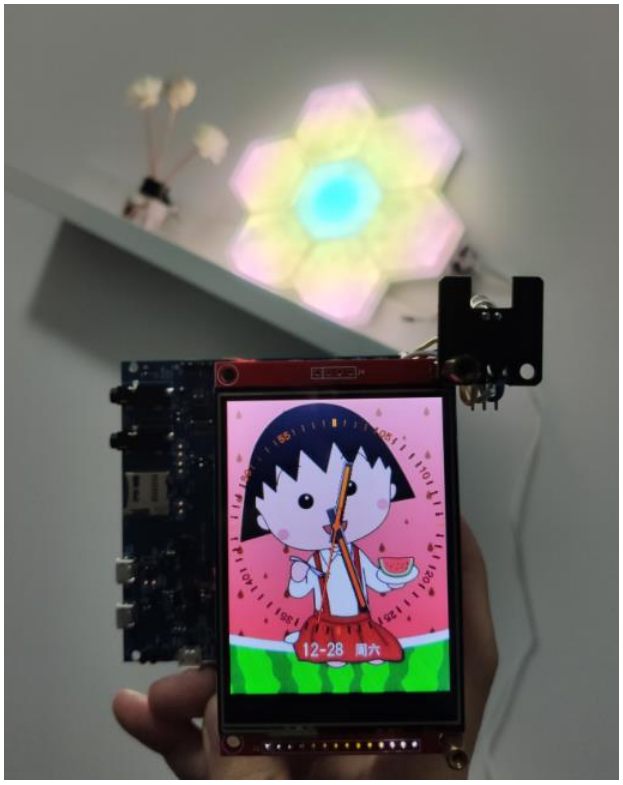
该系统充分发挥了 LPC55s69 的性能，在 FreeRTOS 系统中，150M 主频双核 M33 相互配合完成系统任务：主核 Core0 用于处理主逻辑，包括显示图形处理、触摸数据处理、功能逻辑控制等，在图形处理中同时引入 POWERQUAD 加速计算速度；协核 Core1 用于刷新彩屏，其通过 MCU 的高速 SPI（50M）+DMA 方式驱动 3.2 寸彩屏，240*320 的彩屏刷新频率可高达 60Hz 以上。该主控优秀的性能，使得本系统操作非常流畅！

本系统在发挥 LPC55s69 高性能的同时，也使用了它的 POWER 管理功能，以达到性能功耗的平衡。主核在没有事情处理时就会进入睡眠低功耗模式，此时通过中断（FreeRTOS 的系统 TICK）唤醒。协核在没有事情处理时也进入低功耗模式，其通过主核的通知中断来唤醒。



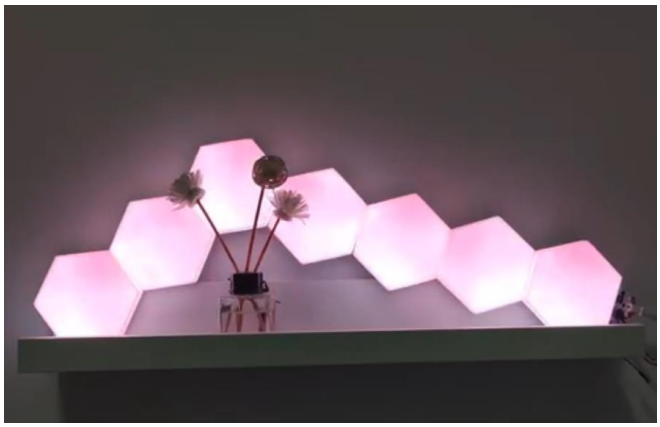
2 作品实物图

主角：基于 LPC55s69 多媒体控制系统



配角：可自由组合的灯光积木



3 演示视频

多媒体控制系统：
https://v.youku.com/v_show/id_XNDQ4ODg3ODYzMg==.html?spm=a2h3j.8428770.3416059.1

灯光效果：
https://v.youku.com/v_show/id_XNDQ4OTAxNDk0MA==.html?spm=a2h3j.8428770.3416059.1

4 项目文档

资料已经上传网盘（代码+产品图片+文档）：
链接：https://pan.baidu.com/s/1i78XibQosa4uM1nKDZ_Mtw
提取码：zxvd

代码已经开源到 git：
https://gitee.com/jacelin/multimedia_control_system

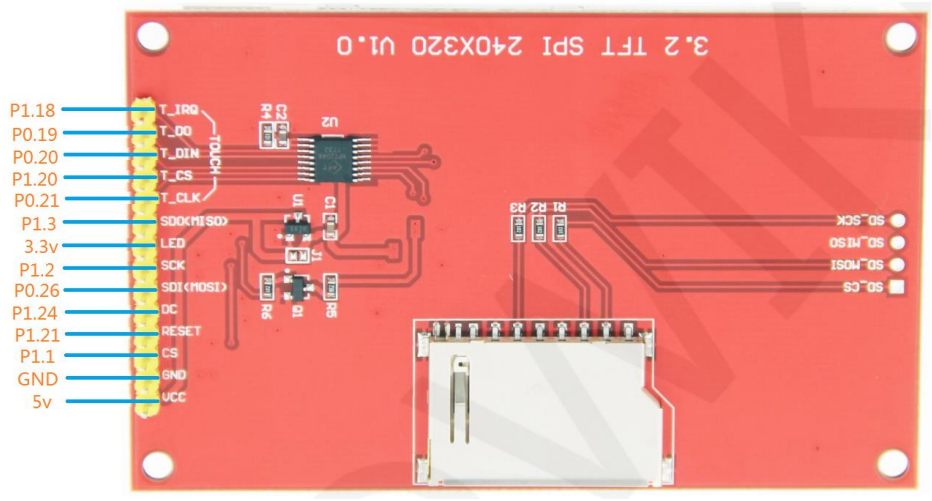
5 硬件

本系统硬件包括：

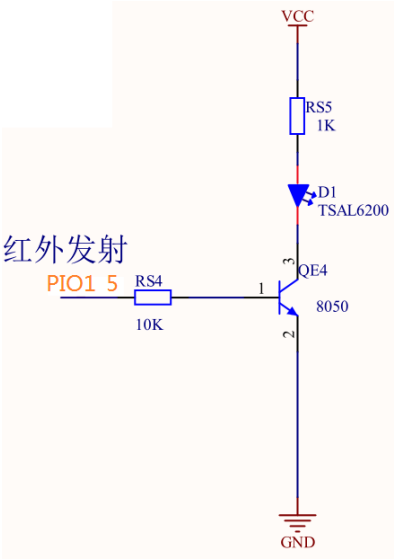
硬件	数量	用途	规格
LPCXpresso55S69 开发板	1	主控	
3.2 寸，240*320 电阻触摸屏	1	交互	显示驱动 ILI9341 TP　xpt2046
红外发射管	1	红外遥控	38KHZ，插件红外 LED 灯

5.1 硬件连线

LCD 和 TP



红外



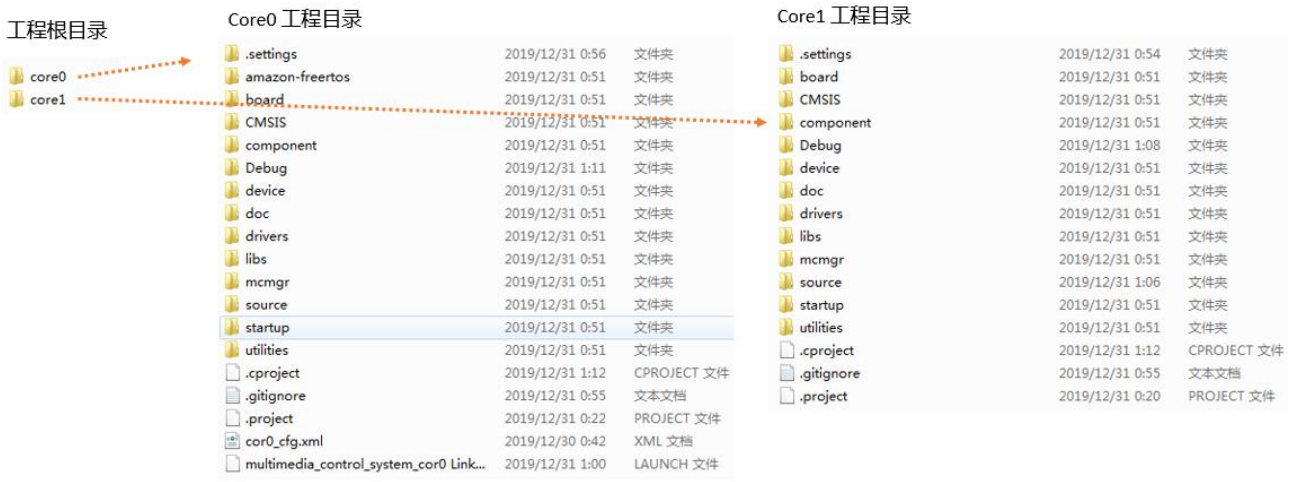
6 系统设计说明

这里将描述系统的实现过程，以便给看到此工程的开发人员一些参考。本系统基于 NXP 的开发 IDE **MCUXpresso IDE v11.0.1_2563** 开发，使用的 LPC55S69 SDK 版本为 **SDK_2.6.3_LPCXpresso55S69_MCUX.zip**。

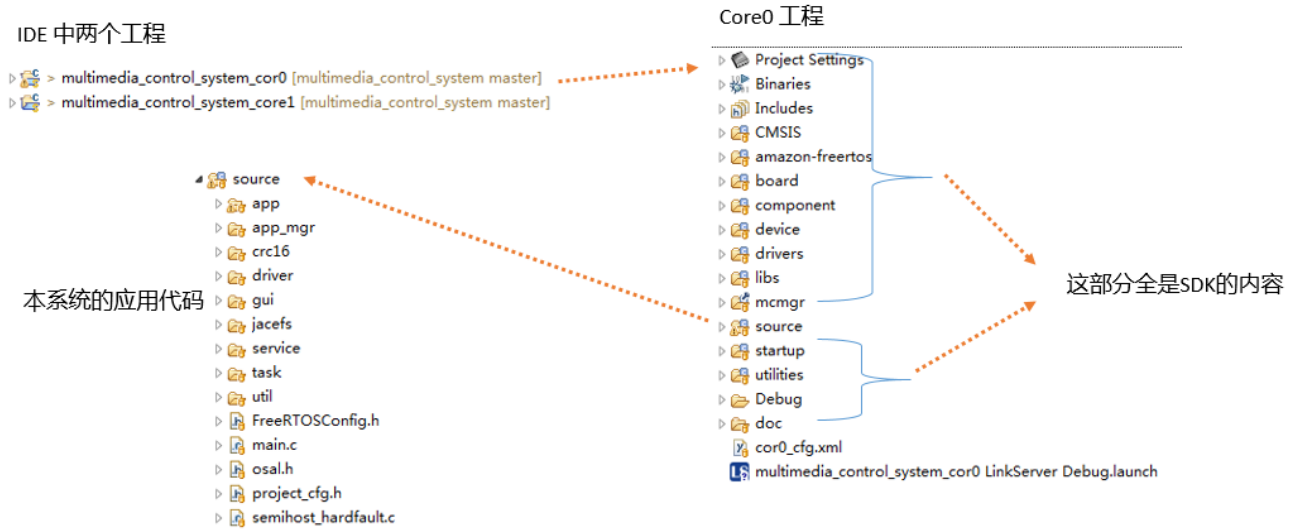
IDE 和 SDK 可到官网下载：
https://www.nxp.com.cn/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc5500-cortex-m33/high-efficiency-arm-cortex-m33-based-microcontroller-family:LPC55S6x?&tab=Documentation_Tab

6.1 工程目录结构

windows 资源管理器下的工程目录结构：



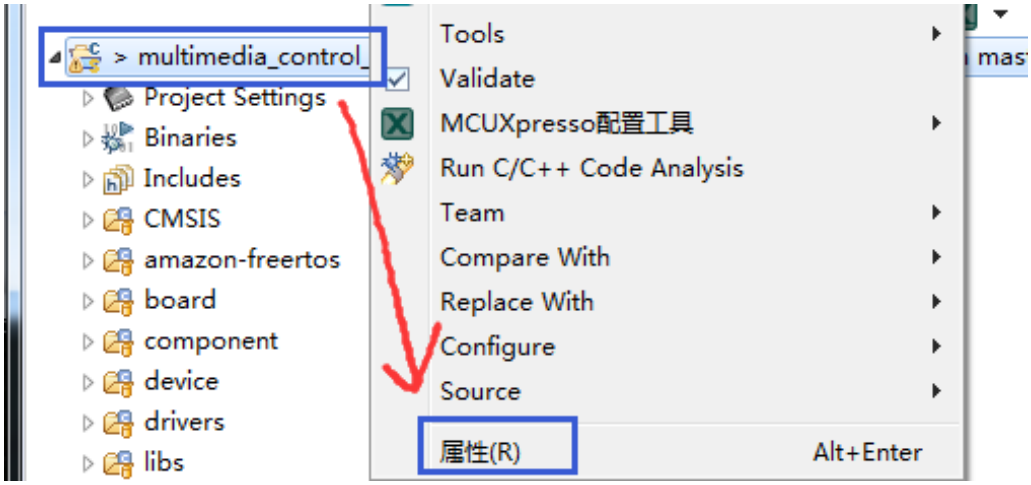
MCUXpresso 中的目录：



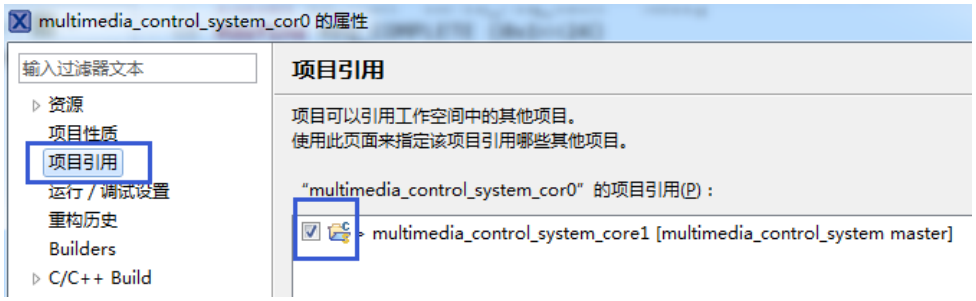
6.2 双核工程关联编译

系统的 Core0 和 Core1 代码是相互独立的，要建立两个工程来开发彼此的功能。MCUXpresso 提供了工程关联编译、链接的功能，需要作以下操作。

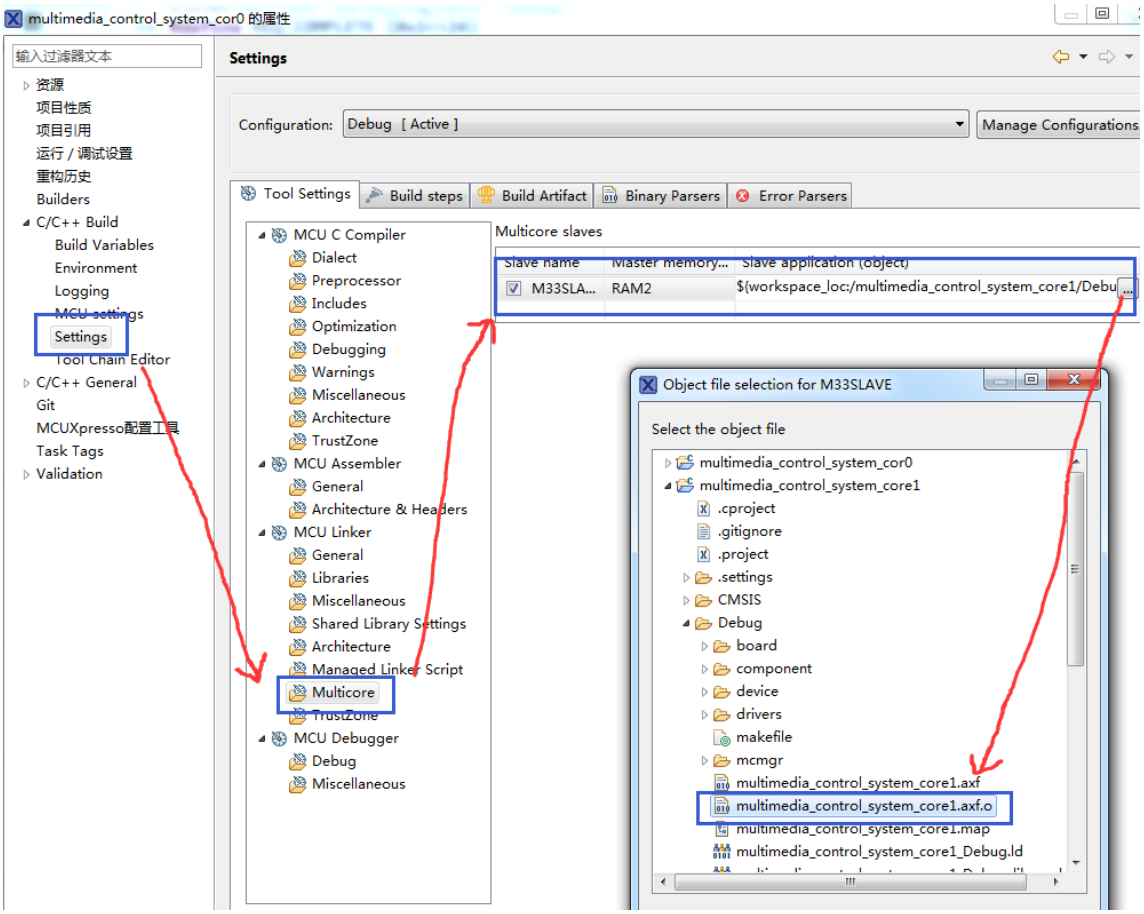
1、右键 Core0 工程进入 ’属性’，



2、在属性中的‘项目引用‘，勾选 Core1 的工程：



3、之后按下图设置：C/C++ Build -> Setting -> Mcu Linker ，在弹出的对话框中选中 Core1 工程的.o 文件（core1 工程需要单独编译后才有.o 文件）



6.3 系统空间分配

MCU 自带 640KB 的 FLASH 和 320KB 的 RAM。**MCU 上电后先启动的 Core0，Core0 将 Core1 的代码从 FLASH 中复制到 RAM 中，将 Core1 从 RAM 启动。**

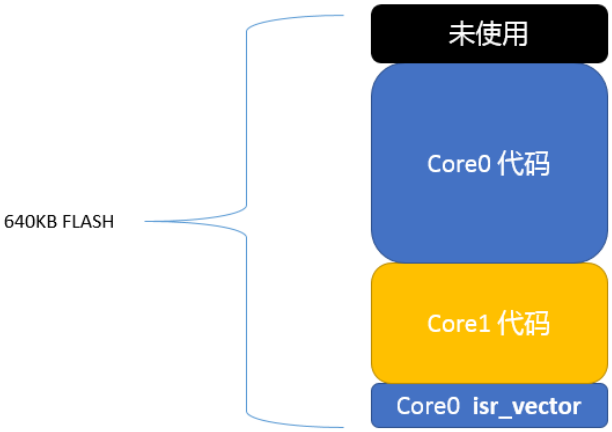
6.3.1 RAM 空间分配

将 RAM 分为 3 个区域，`Ram0` 198KB 给 CORE0 使用，`Ram1` 68KB 给 CORE1 使用，`rpmsg_sh_mem` 6KB 预留给双核共享内存。

```
MEMORY
{
  /* Define each memory region */
  PROGRAM_FLASH (rx) : ORIGIN = 0x0, LENGTH = 0x98000 /* 608K bytes (alias Flash) */
  Ram0 (rwx) : ORIGIN = 0x20000000, LENGTH = 0x31800 /* 198K bytes (alias RAM) */
  Ram1 (rwx) : ORIGIN = 0x20033000, LENGTH = 0x11000 /* 68K bytes (alias RAM2) */
  rpmsg_sh_mem (rwx) : ORIGIN = 0x20031800, LENGTH = 0x1800 /* 6K bytes (alias RAM3) */
}
```

6.3.2 FLASH 空间分配

FLASH 一共 640KB，现在只使用了前 608KB。Flash 最前面存放`.isr_vector`，随后是编译到 `Ram1` 中的 Core1 代码，再后面是 Core0 的代码。

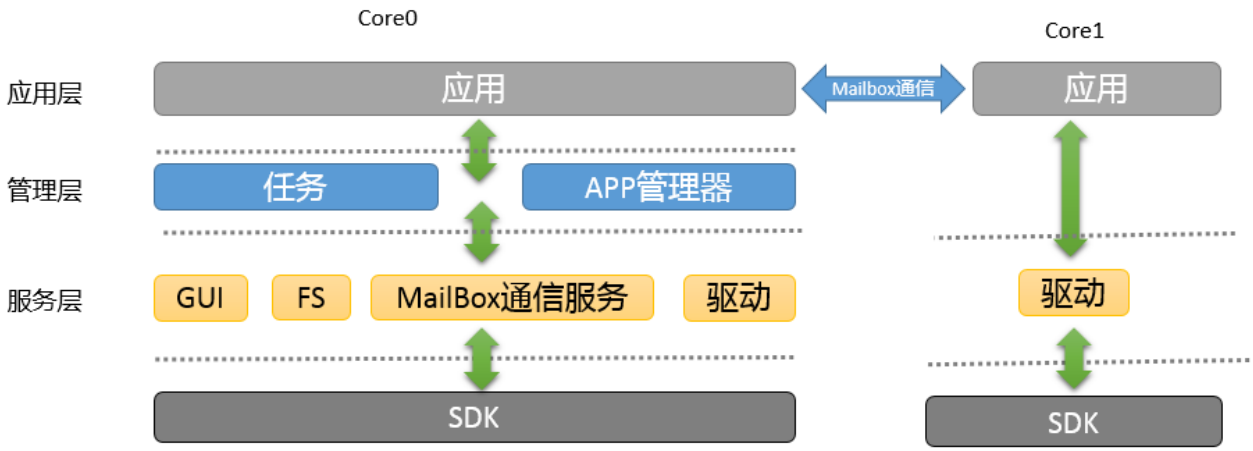


6.4 软件逻辑结构

本机为全触摸机器，所有功能操作都只能通过触摸实现。机器支持左右滑动切换界面，单点打开应用，右滑能出应用，而应用中的操作可以是各式各样的。

系统中的 APP，有控制类的（控制灯、空调、音乐等），有小游戏之类的。

软件整体框架如图：



Core0 基于 FreeRTOS 系统开发，Core1 为裸机程序。Core0 中引包括了本人发明的 jace FS（文件系统）、jace GUI （图形库）两大内容，并且使用了独家创作的操作系统任务管理方式，各大软件相互高效配合，使得本系统操作非常流畅！

Core0 通过触摸输入、系统事件触发调用 GUI 实现界面图形的处理，处理完成发送信号给 Core1，Core1 进入刷屏。

6.5 代码实现

系统代码比较多，这里列出几处理关键的地方。

6.5.1 双核通信

双核通过一个叫做和 MailBox 的东西相互通信，这个模块只有几个寄存器：

Table 1035.Register overview: Mailbox (base address 0x5008 B000) bit description.

Name	Access	Offset	Description	Reset value	Section
IRQ0	R/W	0x000	Interrupt request register for the Cortex-M33 (CPU1).	0x0	Section 54.6.1
IRQ0SET	WO	0x004	Set bits in IRQ0.	-	Section 54.6.2
IRQ0CLR	WO	0x008	Clear bits in IRQ0.	-	Section 54.6.3
IRQ1	R/W	0x010	Interrupt request register for the Cortex-M33 (CPU0).	0x0	Section 54.6.4
IRQ1SET	WO	0x014	Set bits in IRQ1.	-	Section 54.6.5
IRQ1CLR	WO	0x018	Clear bits in IRQ1.	-	Section 54.6.6
MUTEX	R/W	0x0F8	Mutual exclusion register[1].	0x1	Section 54.6.7

双核之间的通信（叫通知可能更贴切）每次只能传输 4 字节，如 Core0 通过把 uint32_t 类型的数据给 `IRQ1SET` 寄存器，Core1 就会产生中断，在中断里面通过读取 `IRQ1` 寄存器就可以获取到 Core0 传过来的 4 字节数据。

所以如果要更好的使用双核，MailBox 要配合共享内存空间使用，才能实现更多功能。

1、初始化

Core0 启动 Core1：

```
MCMGR_Init();
MCMGR_StartCore(kMCMGR_Core1, CORE1_BOOT_ADDRESS, (uint32_t)gui_get_front_fb(), kMCMGR_Start_Synchronous);
```

本系统中，Core0 启动 Core1 时，把屏幕 FB 地址通过启动参数传输给 Core1，Core1 使用该 BUF 刷新屏幕。

2、注册通知回调

Core0 为了能接收 Core1 的通知，需要注册中断回调函数：

```
static void RPMsgRemoteReadyEventHandler(uint16_t eventData, void *context)
{
    if(eventData==RPMSG_FLUSH_SCREEN_DONE)
    {
        if(core1_req_task)
        {
            OS_TASK_NOTIFY_FROM_ISR(core1_req_task,REQ_COMPLETE,OS_NOTIFY_SET_BITS);
        }
    }
}
MCMGR_RegisterEvent(kMCMGR_RemoteApplicationEvent, RPMsgRemoteReadyEventHandler,0);
```

在 Core0 中断回调函数中，把 Core1 通知发送给主任务，由主任务处理该事件。

3、发送通知

完成以上两步，接下来 Core0 就可以给 Core1 发数据了，以下展示 Core0 通知 Core1 刷屏的操作代码：

```
//请求核1刷屏
void notif_cor1_flush_screen()
{
    uint32_t notif;
```

```
    core1_req_task=OS_GET_CURRENT_TASK();

    OS_ENTER_CRITICAL_SECTION();
    MCMGR_TriggerEventForce(kMCMGR_RemoteApplicationEvent, RPMSG_FLUSH_SCREEN_REQ);
    OS_LEAVE_CRITICAL_SECTION();

    //等待核1处理完成,超时100ms
    OS_TASK_NOTIFY_WAIT(0x0, OS_TASK_NOTIFY_ALL_BITS, &notif, OS_MS_2_TICKS(100));
    if ((notif&REQ_COMPLETE) == 0)
    {
        OS_LOG("core1 handle err!\r\n");
    }
    else
    {
        OS_LOG("core1 done!\r\n");
    }
}
```

以上代码中，Core0 发送通知 RPMSG_FLUSH_SCREEN_REQ 给 Core1 后，就进入等待 Core1 的处理回应，超时 100ms。此时 Core1 那边接收到 Core0 的 RPMSG_FLUSH_SCREEN_REQ 请求后，会处理刷屏并返回一个 RPMSG_FLUSH_SCREEN_DONE 事件。到此一个刷屏通信完成。

Core1 的处理方式这里就不描述了，和 Core0 类似，都是比较简单的处理，更多内容可以到工程中看代码。

6.5.2 PowerQuad 使用

这个功能非常牛逼，能加速数学计算，常用的三角函数、开方、商等操作都能用该模块加速。本系统主要把 powerquad 功能用于 GUI 中的图形旋转、透明等操作，以减少时间，加速界面切换速度。

1、初始化

```
PQ_Init(POWERQUAD);
```

2、使用

在 gui_math.c 中，把函数封装给 GUI 使用。

```
float gui_sqrt(float __x)
{
    PQ_SqrtF32(&__x, &__x);
    return __x;
}
float gui_cos(float __x)
{
    PQ_CosF32(&__x, &__x);
    return __x;
}
float gui_sin(float __x)
```

```
{
    PQ_SinF32(&__x, &__x);
    return __x;
}
float gui_div(float __x,float __y)
{
    PQ_DivF32(&__x, &__y, &__x);
    return __x;
}
```

6.5.3 功耗控制

功耗管理使用的是 MCU 的 Power Management 功能，这个模块把系统电源分为几大块：

14.2.2 Power domains

The device is partitioned into five power domains:

- PD_CORE: Power Domain *Core*: most of all digital core logic (CPU0, CPU1, and multilayer matrix).
- PD_SYSTEM: Power Domain *System:* Some critical system components like clocks controller, reset controller and Syscon.
- PD_AO: Power Domain *Always On*: Power management controller and RTC.This domain always has power as long as sufficient voltage is available on VBAT ([1.8 V – 3.6 V]).
- PD_MEM_0: First Power Domain *Memories*: Two 4 KB SRAM instances.
- PD_MEM_1: Second Power Domain *Memories*: All other SRAM instances.

Table 276 shows the detailed list of all modules per power domain.

同时，系统支持的功耗模式：

Table 277. Power modes

	PD_CORE	PD_SYSTEM	PD_AO	PD_MEM_0	PD_MEM_1
ACTIVE	ON	ON	ON	ON	ON
SLEEP	ON	ON	ON	ON	ON
DEEP SLEEP	ON	ON	ON	ON/OFF	ON/OFF
POWER DOWN	OFF	ON	ON	ON/OFF	ON/OFF
DEEP POWER DOWN	OFF	OFF	ON	ON/OFF	ON/OFF

系统上电后是 ACTIVE 模式，其他模式都是要软件去配置才能进入的。在 ACTIVE 模式中，MCU 全速 150MHZ 双核在运行，此时本系统的功耗非常大（后面有数据）,很有必要使用 Power 管理功能。

遗憾的是 SDK 没有提供 FreeRTOS 的功耗管理用例（或者是我没有找到），所以本系统的功耗管理方法是本人根据开发 FreeRTOS 经验去配置的，但管理确实是有效果的。下面我将把我的方法描述下来，另外不得不说功耗管理是个很麻烦的东西，这里是仔细查看了芯片手册的 Chapter 14 和 Chapter 15 这两章设置的。

值得注意的是，SDK 提供的 FreeRTOS 系统是用的 System Tick，而这个时钟不能配置为睡眠唤醒时钟的，所以要想要 FreeRTOS 中使用 DeepSleep 或者更低功耗的模式，必须要把 FreeRTOS 的时钟更换为能作为唤醒源的定时器，如 CTIMER、UTICK、RTC 等。这里由于时间关系暂时不以实现。

1、配置 FreeRTOS 低功耗模式

在 FreeRTOSConfig.h 中，增加以下配置：

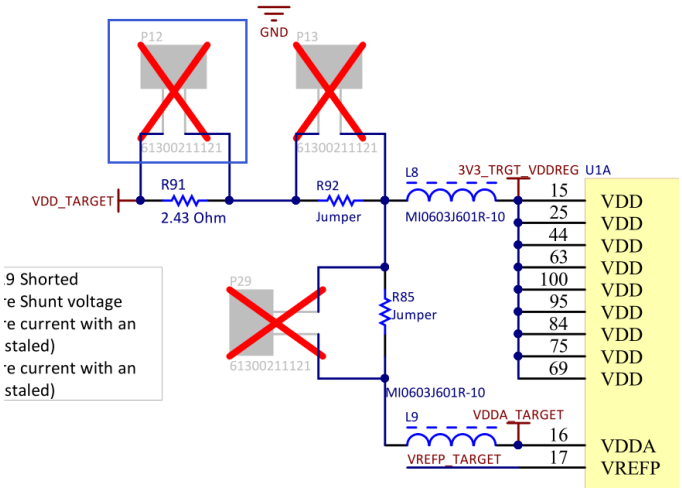
```
extern void prvSystemSleep( uint32_t xExpectedIdleTime );
#define portSUPPRESS_TICKS_AND_SLEEP( x )       prvSystemSleep( x )
#define configUSE_TICKLESS_IDLE                 2
```

然后实现 **prvSystemSleep** 函数。

```
#if configUSE_TICKLESS_IDLE>0
void prvSystemSleep( uint32_t xExpectedIdleTime )
{
    POWER_EnterSleep();
}
#endif
```

2、电流测量

将电流表接在开发板 P12 座子上：



电流数据（使用 4 位半电流表测量，可能有误差）：

模式	功耗	功耗相比 ACTIVE 下降
ACTIVE	平均 11.58ma,峰值 14.00 ma	
SLEEP	平均 7.44 ma，峰值 13.3ma	平均 4.14ma,峰值 0.7 ma
DeepSleep	平均 0.3 ma	平均 11.28ma

6.5.4 系统代码启动流程

//有空再补充

6.5.5 APP 开发

本系统的开发以 APP 为单位（类似于智能机），比如本系统中的“灯光设置”应用，需要这样去定义：

```
static const app_inst_info_t _app_info=
{
    .app_id=SYS_APP_ID_LIGHT,
    .file_id=APP_INST_PKG_FILE_ID,
```

```
    .type=APP_TYPE_TOOL,
    .reserved={0,0},

    .elf_inrom_addr=INVALID_ELF_INROM_ADDR,
    .elf_inrom_size=0,

    .elf_inram_addr=INVALID_ELF_INRAM_ADDR,
    .main=_main,
};
static os_app_node_t app_node =
{
    .list = LIST_INIT(app_node.list),
    .info=&_app_info,
    .priority = APP_PRIORITY_LOWEST,
};
void app_light_init(void)
{
    os_app_add(&app_node);
}
```

所有的 APP 都要添加到系统 APP 列表中，它们的启动、退出都由系统的 APP 管理模块管理着，非常的统一、方便、智能。