密级状态：绝密(　　)　　秘密(　　)　　内部资料(　)　　公开(√)

Armv8 trusted Firmware 培训

(技术研发部，底层平台中心)

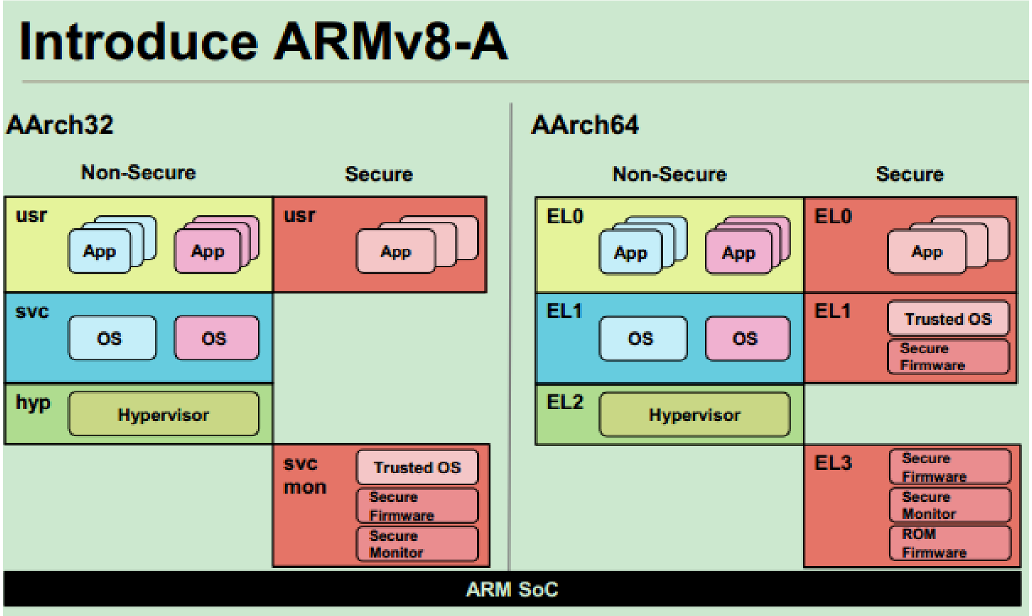
文件状态：	文件标识：	Trusted Firmware（arm v8）
[ ] 草稿	当前版本：	1.2
[√] 正式发布	审　　核：	
[ ] 正在修改	作　　者：	谢修鑫
	完成日期：	2017-5-22

版 本 历 史

版本号	作者	修改日期	修改说明
1.0	谢修鑫	2015.8.11	初稿
1.1	谢修鑫	2016.8.7	基于 atf1.2 版本、linux4.4 版本更改
1.2	谢修鑫	2017.5.22	Plat_pm.c 函数调用实现

一 架构简介

1 ARM V8 应用架构



2 为什么需要 trusted Firmware?

1、 To use AArch64, EL3 must be AArch64.

2、AArch64 demands a different approach in the Secure Monitor.

1） EL1 (operating system) processor state must saved and restored by the Secure Monitor software.

3、Separation of the Trusted OS at Secure-EL1 from the Secure Monitor at EL3 requires a redesign of the interaction between the Trusted OS and Monitor.

4、Everyone writing secure privileged code has some substantial work to do – it's not just a port of ARM assembler code to A64 instructions。

5、A single kernel image has to work on all platforms – including the ones that have not been created yet

- Particularly for Enterprise systems
- This demands that interaction with the hardware platform is standardized around specified peripheral and Firmware interfaces.

3 ARM Trusted Firmware 提供了那些标准？

1、ARM has been creating some of these standards to make this possible:

1） SMC Calling Convention – to enable standard and vendor specific firmware services to coexist

2）PSCI – a firmware interface for CPU power control

2、How many implementations of the standards do we need?

1） Defines a standard calling convention Secure Monitor

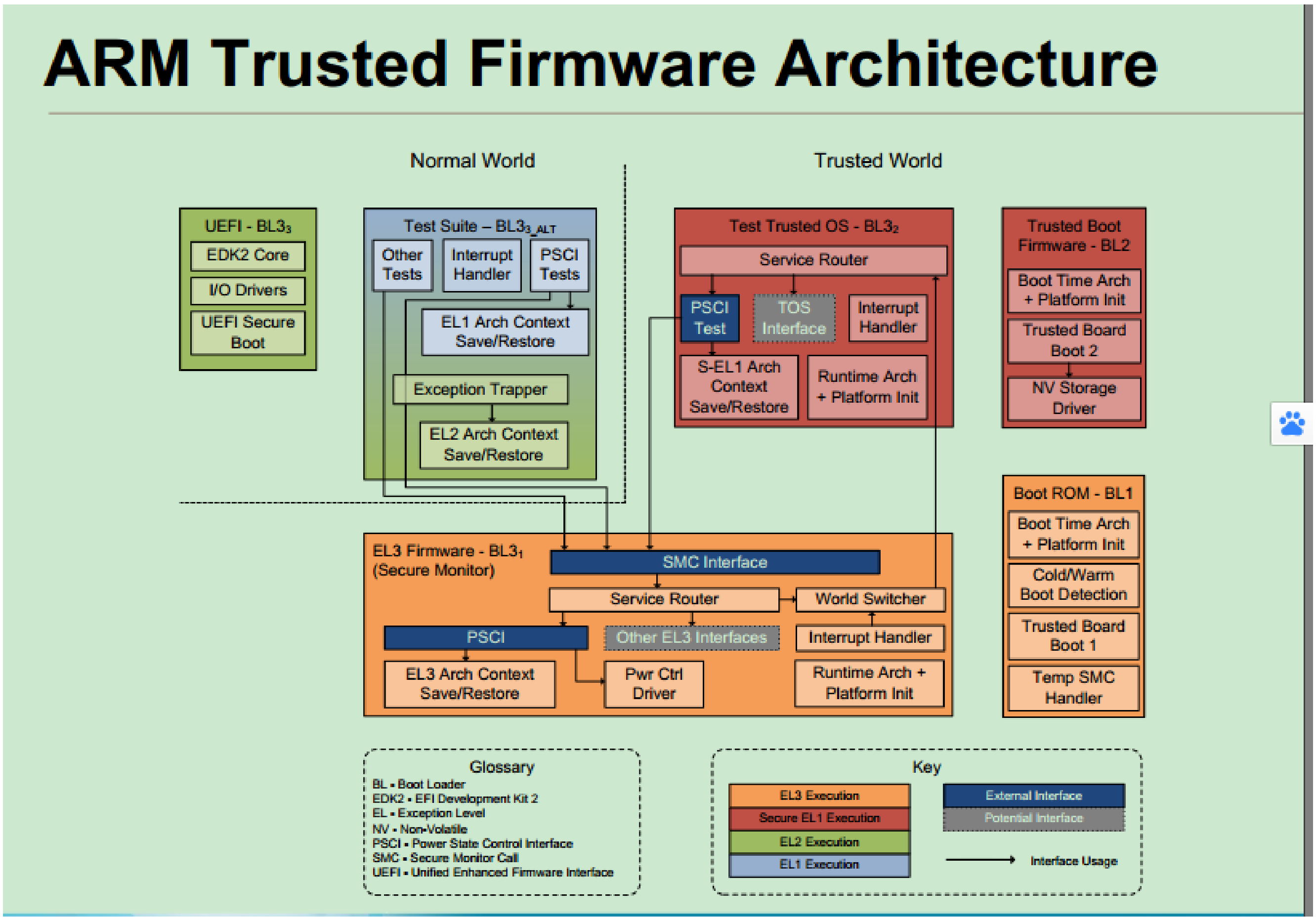
2） Defines a partitioning of function ID space to allow multiple vendors to coexist in secure firmware

- OEMs, SiPs and Trusted OS vendors

3） Providing number of services e.g.

- Standard firmware services (e.g. power management)
- Trusted OS
- Errata management

4 ARM Trusted Firmware Architecture



4 SMC Calling

参考 linux4.4

可以通过 SMC 指令实现从 EL1（nosecure、secure os）到 EL3 的切换。

@ psci-call.S (kernel\arch\arm64\kernel)

ENTRY(__invoke_psci_fn_smc)

smc#0

ret

ENDPROC(__invoke_psci_fn_smc)

6 CPU ID

每个 CPU 都有一个自己的 MPIDR_EL1 (Multiprocessor Affinity Register)寄存器，用于记录自己在多 cluster 系统中的 ID.

Aff2, bits [23:16]:Affinity level 2. Second highest level affinity field.

Aff1, bits [15:8]:Affinity level 1. Third highest level affinity field. 即为 Cluster ID 编号

Aff0, bits [7:0]:Affinity level 0. Lowest level affinity field. 每个 Cluster 中的 IC 编号

例如，RK 平台大核为 Cluster ID 为 1，小核 ID 为 0. 大核的 CPU3 的 MPIDR 低 16 位为 0x103。

二 Trusted firmware 实现

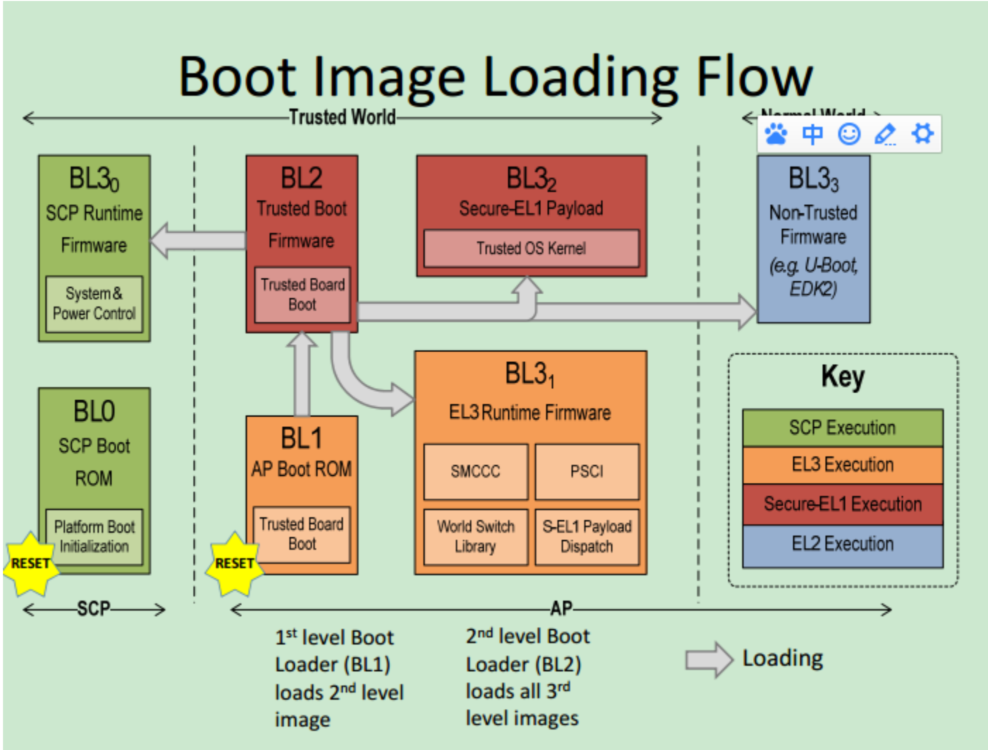
2.1 Firmware 分类

ARM trusted Firmware 将系统启动分为几个步骤，对应不同的 image。

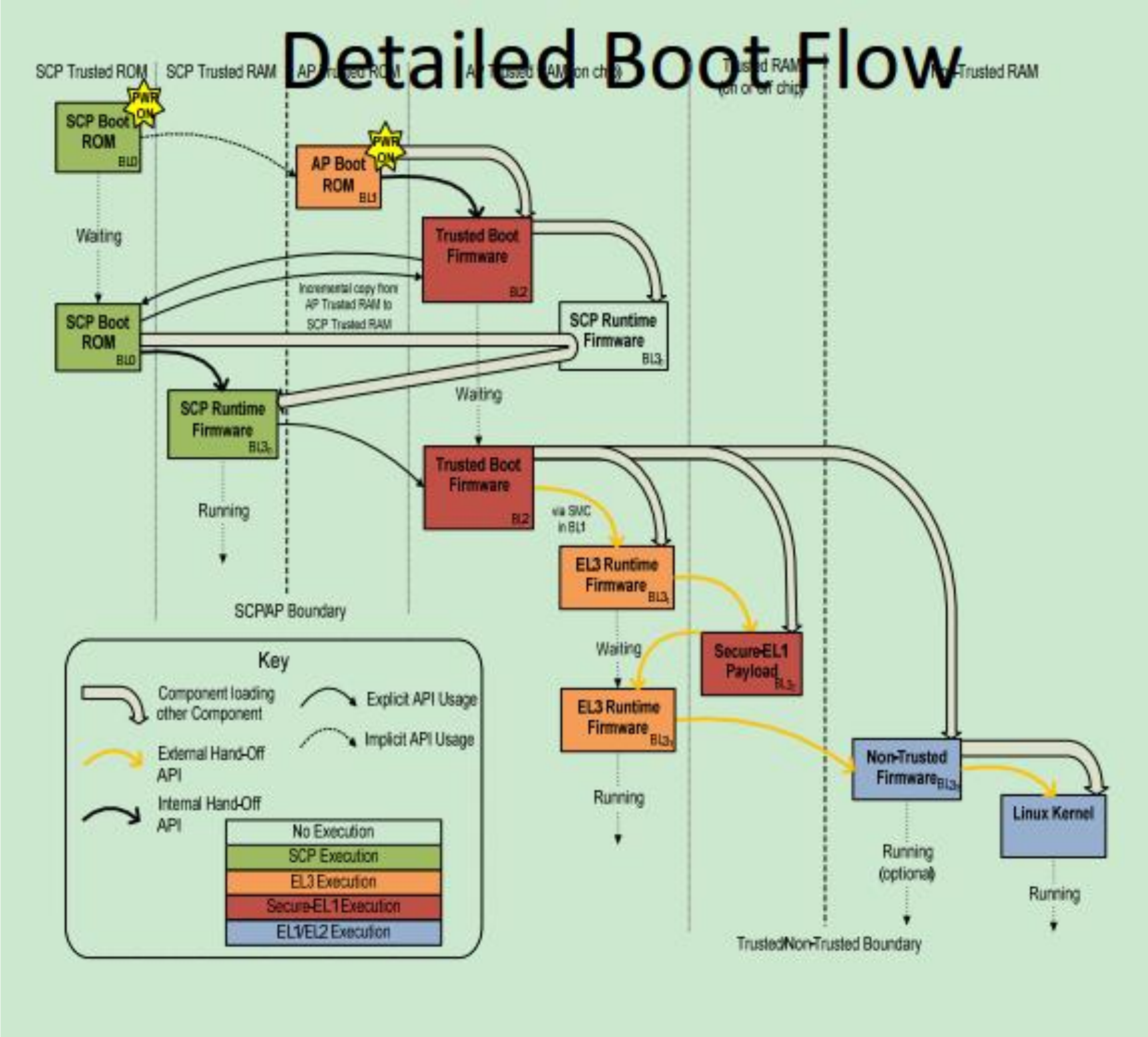
* Boot Loader stage 1 (BL1) _AP Trusted ROM_
  对应 RK 的 Maskrom
* Boot Loader stage 2 (BL2) _Trusted Boot Firmware_
  对应 RK 的 Miniloader
* Boot Loader stage 3-1 (BL3-1) _EL3 Runtime Firmware_
  涉及 PSCI、Secure Monitor 功能支持的固件
* Boot Loader stage 3-2 (BL3-2) _Secure-EL1 Payload_ (optional)
  安全固件
* Boot Loader stage 3-3 (BL3-3) _Non-trusted Firmware_
  与 Uboot 相同。

目前 RK 使用的是 BL31 功能，该功能提高功耗管理相关的接口及 Secure Monitor 功能。

1、Firmware Loading



2、Firmware Booting



2.2 怎么触发 BL31-Kernel 层的交互

No Secure OS（Linux）和 Secure OS,如果需要同 BL31 进行交互，可以通过两种方法。

方法 1：通过显示的调用 SMC 指令，主动申请陷入 BL31.

方法 2：将中断配置为需要在 EL3 中处理，这个功能主要针对安全的中断，系统运行在 Linux Kernel 时，系统会先进入 BL31，然后在 BL31 中切换到 Secure OS 中进行处理。

2.3 BL31 层架构

基于 1.3 版本

主要功能为 No Secure OS（Linux）和 Secure OS 通过 SMC 命令或异常陷入到 EL3（BL31）中实现 Power Management、Secure Monitor 功能。

1 Function Handle

OS 陷入到 BL31 时，会传入一个参数，这个参数表明 BL31 怎么解析这个 SMC 并找到相应的处理 handler。格式定义如下：

1）Function ID 定义

Table 2-1 Bit usage within the SMC Function Identifier

<table>
<tr><th>Bit Numbers</th><th>Bit Mask</th><th>Description</th></tr>
<tr><td>31</td><td>0x80000000</td><td>If set to 0 then this is Standard call (pre-emptible)
If set to 1 then this is a Fast Call (atomic)</td></tr>
<tr><td>30</td><td>0x40000000</td><td>If set to 0 then this is the SMC32 calling convention.
If set to 1 then this is the SMC64 calling convention.</td></tr>
<tr><td>29:24</td><td>0x3F000000</td><td>
<table>
<tr><th>Owning Entity Number</th><th>Bit Mask</th><th>Description</th></tr>
<tr><td>0</td><td>0x00000000</td><td>ARM Architecture Calls</td></tr>
<tr><td>1</td><td>0x01000000</td><td>CPU Service Calls</td></tr>
<tr><td>2</td><td>0x02000000</td><td>SIP Service Calls</td></tr>
<tr><td>3</td><td>0x03000000</td><td>OEM Service Calls</td></tr>
<tr><td>4</td><td>0x04000000</td><td>Standard Service Calls</td></tr>
<tr><td>5-47</td><td>0x05000000 – 0x2F000000</td><td>Reserved for future use</td></tr>
<tr><td>48-49</td><td>0x30000000 – 0x31000000</td><td>Trusted Application Calls</td></tr>
<tr><td>50-63</td><td>0x32000000 – 0x3F000000</td><td>Trusted OS Calls</td></tr>
</table>
These ranges are further defined in section 6.</td></tr>
<tr><td>23:16</td><td>0x00FF0000</td><td>Must be zero (MBZ), for all Fast Calls, when bit[31] == 1.
All other values reserved for future use
Note: Some ARMv7 legacy Trusted OS Fast Call implementations have all bits set to 1.</td></tr>
<tr><td>15:0</td><td>0x0000FFFF</td><td>Function number within the range call type defined by bits[29:24].</td></tr>
</table>

定义如下：

#define PSCI_CPU_SUSPEND_AARCH32 0x84000001

#define PSCI_CPU_SUSPEND_AARCH64 0xc4000001

SIP：Silicon Partner. In this document, the silicon manufacturer.

Table 6-7: Standard Service Call range

SMC Function Identifier	Reserved use and sub-range ownership	Notes
	Owner: Standard Service Calls	
0x84000000-0x8400001F	PSCI SMC32 bit Calls	A range of SMC calls. See [5] for details of functions and arguments.
0x84000020-0x8400FEFF	SMC32: Standard Service Calls	Service calls defined by ARM standards. The arguments are defined by the relevant ARM standard.
0x8400FF00	SMC32: Standard Service Call Count	This call returns a 32-bit count of the available Service Calls. A return value of zero means no services are available.
0x8400FF01	SMC32: Standard Service Call UID	Each Implementation of Standard Service Calls must provide a unique Identifier (UID).
0x8400FF02	Reserved	
0x8400FF03	SMC32: Standard Service Call Revision details	This SMC returns the revision information for the Standard service calls.
0x8400FF04-0x8400FFFF	Reserved for future expansion	
0xC4000000-0xC400001F	PSCI SMC64 bit Calls	A range of SMC calls. See [5] for details of functions and arguments.
0xC4000004-0xC400FEFF	SMC64: Standard Service Calls	Service calls defined by ARM standards. The arguments are defined by the relevant ARM standard.
0xC4FFFF00-0xC4FFFFFF	Reserved for future expansion	

2) Function Handle 添加

通过下面操作加入 Handler，以 Power PSCI 的接口为例。

```
DECLARE_RT_SVC(
        std_svc,
        OEN_STD_START,
        OEN_STD_END,
        SMC_TYPE_FAST,
        NULL,
        std_svc_smc_handler
    );
#define DECLARE_RT_SVC(_name, _start, _end, _type, _setup, _smch) \
    static const rt_svc_desc_t __svc_desc_ ## _name \
        __section("rt_svc_descs") __used = { \
            .start_oen = _start, \
            .end_oen = _end, \
            .call_type = _type, \
            .name = #_name, \
            .init = _setup, \
```

```
.handle = _smch }
```

其中
#define OEN_STD_START 4 /* Standard Calls */
#define OEN_STD_END 4
表示 Function ID 对应的 OEN 为 Standard Calls。

2 Power 相关 PSCI 处理流程

代码参考：std_svc_setup.c (services\std_svc)
通过下面代码注册一类 SMC 的处理 handler：

```
DECLARE_RT_SVC(
        std_svc,
        OEN_STD_START,
        OEN_STD_END,
        SMC_TYPE_FAST,
        NULL,
        std_svc_smc_handler
);
```

这个宏指定了 function id 类型为:**SMC_TYPE_FAST**

standard service call 的范围：OEN_STD_START~ OEN_STD_END，相关宏定义如下：
#define OEN_STD_START 4
#define OEN_STD_END 4

满足上面条件的 SMC 调度都会运行 std_svc_smc_handler。std_svc_smc_handler 包含了很多相关 psci 功能实现，下面对 cpu_operations 相关操作进行讲解。

```
uintptr_t std_svc_smc_handler(uint32_t smc_fid,
                u_register_t x1,
                u_register_t x2,
                u_register_t x3,
                u_register_t x4,
                void *cookie,
                void *handle,
                u_register_t flags)
{
        /*
         * Dispatch PSCI calls to PSCI SMC handler and return its return
         * value
         */
        if (is_psci_fid(smc_fid)) {
```

```
        SMC_RET1(handle,
            psci_smc_handler(smc_fid, x1, x2, x3, x4,
                cookie, handle, flags));
    }
}
```

Smc_fid 即 function id，代码后继会根据不同的 ID 值，进行不同的处理。

如下面不同功能 ID 值：

```
#define PSCI_CPU_SUSPEND_AARCH32    0x84000001
#define PSCI_CPU_SUSPEND_AARCH64    0xc4000001
#define PSCI_CPU_OFF                0x84000002
#define PSCI_CPU_ON_AARCH32         0x84000003
#define PSCI_CPU_ON_AARCH64         0xc4000003
```

3 power management 相关操作

@Plat_pm.c (plat\rockchip\common)

```
const plat_psci_ops_t plat_rockchip_psci_pm_ops = {
    .cpu_standby = rockchip_cpu_standby,
    .pwr_domain_on = rockchip_pwr_domain_on,
    .pwr_domain_off = rockchip_pwr_domain_off,
    .pwr_domain_suspend = rockchip_pwr_domain_suspend,
    .pwr_domain_on_finish = rockchip_pwr_domain_on_finish,
    .pwr_domain_suspend_finish = rockchip_pwr_domain_suspend_finish,
    .system_reset = rockchip_system_reset,
    .system_off = rockchip_system_poweroff,
    .validate_power_state = rockchip_validate_power_state,
    .get_sys_suspend_power_state = rockchip_get_sys_suspend_power_state
};
```

上面回调主要涉及到下面几个功能：

- CPU on/off
- CPU suspend
- System suspend
- System reset
- System off

CPU off/on 注意事项：

CPU OFF 操作时，如果一个 cluster 有 4 个 CPU，如果最后一个 CPU 关闭，就需要将 cluster（我们称作为 SCU）关闭。

CPU ON 操作时，如果一个 cluster 有 4 个 CPU，开启第一个 CPU 时，要先将 cluster（我们称作为 SCU）开启。

上面两点，架构上都有支持，我们需要匹配。

4 代码讲解

以 Suspend 为例：

1）、PSCI 架构层实现

```
u_register_t psci_smc_handler(uint32_t smc_fid,
                          u_register_t x1,
                          u_register_t x2,
                          u_register_t x3,
                          u_register_t x4,
                          void *cookie,
                          void *handle,
                          u_register_t flags)
{
        case PSCI_SYSTEM_SUSPEND_AARCH32:
            return psci_system_suspend(x1, x2);
}
```

2）、rockchip 平台架构实现

最终会调用到各个平台的 suspend 处理。

```
@Plat_pm.c (plat\rockchip\common)
void rockchip_pwr_domain_suspend(const psci_power_state_t *target_state)
{
    uint32_t lvl;
    plat_local_state_t lvl_state;
    if (RK_CORE_PWR_STATE(target_state) != PLAT_MAX_OFF_STATE)
        return;
    if (rockchip_ops) {
        if (RK_SYSTEM_PWR_STATE(target_state) ==
PLAT_MAX_OFF_STATE &&
            rockchip_ops->sys_pwr_dm_suspend) {
            rockchip_ops->sys_pwr_dm_suspend();
        } else if (rockchip_ops->cores_pwr_dm_suspend) {
            rockchip_ops->cores_pwr_dm_suspend();
        }
    }

    /* Prevent interrupts from spuriously waking up this CPU */
    plat_rockchip_gic_cpuif_disable();

    /* Perform the common cluster specific operations */
        if (RK_CLUSTER_PWR_STATE(target_state) ==
    PLAT_MAX_OFF_STATE)
            plat_cci_disable();
```

```
        if (!rockchip_ops || !rockchip_ops->hlvl_pwr_dm_suspend)
            return;

        for (lvl = MPIDR_AFFLVL1; lvl <= PLAT_MAX_PWR_LVL; lvl++) {
            lvl_state = target_state->pwr_domain_state[lvl];
            rockchip_ops->hlvl_pwr_dm_suspend(lvl, lvl_state);
        }
    }
```

System suspend 及 CPU suspend 操作都会进入这个处理函数，区分标准就是两个 cluster 都需要 suspend 时，系统进入 suspend。

为了使用我们平台的不同芯片，不同芯片的处理通过 rockchip_ops callback 进行处理。

3） rockchip 不同平台芯片实现

以 rk3399 为例：

@Pmu.c (plat\rockchip\rk3399\drivers\pmu):

（1）、对于旧的版本

通过回掉数组的形式实现，如：

```
static struct rockchip_pm_ops_cb pm_ops = {
    .cores_pwr_dm_on = cores_pwr_domain_on,
    .cores_pwr_dm_off = cores_pwr_domain_off,
    .cores_pwr_dm_on_finish = cores_pwr_domain_on_finish,
    .cores_pwr_dm_suspend = cores_pwr_domain_suspend,
    .cores_pwr_dm_resume = cores_pwr_domain_resume,
    .hlvl_pwr_dm_suspend = hlvl_pwr_domain_suspend,
    .hlvl_pwr_dm_resume = hlvl_pwr_domain_resume,
    .hlvl_pwr_dm_off = hlvl_pwr_domain_off,
    .hlvl_pwr_dm_on_finish = hlvl_pwr_domain_on_finish,
    .sys_pwr_dm_suspend = sys_pwr_domain_suspend,
    .sys_pwr_dm_resume = sys_pwr_domain_resume,
    .sys_gbl_soft_reset = soc_soft_reset,
    .system_off = soc_system_off,
};

static int sys_pwr_domain_suspend(void)
{
    // soc  相关操作
}
```

（2）、对于新的版本（2017 年 3 月以后）

是通过 weak 函数的形式实现，函数如下：

```
int rockchip_soc_sys_pwr_dm_suspend(void)
```

2.4 Linux 相关接口

参考代码 linux4.4

Psci.c (arch\arm64\kernel)

1、PSCI 提供相关操作函数

Linux 将各个功能的 SMC 操作，封装了对应的相关函数，操作时直接调用相关函数即可。

```
@Psci.c (kernel\drivers\firmware)
struct psci_operations {
	int (*cpu_suspend)(u32 state, unsigned long entry_point);
	int (*cpu_off)(u32 state);
	int (*cpu_on)(unsigned long cpuid, unsigned long entry_point);
	int (*migrate)(unsigned long cpuid);
	int (*affinity_info)(unsigned long target_affinity,
			unsigned long lowest_affinity_level);
	int (*migrate_info_type)(void);
};
static void __init psci_0_2_set_functions(void)
{
	pr_info("Using standard PSCI v0.2 function IDs\n");
	psci_function_id[PSCI_FN_CPU_SUSPEND] =
				PSCI_FN_NATIVE(0_2, CPU_SUSPEND);
	psci_ops.cpu_suspend = psci_cpu_suspend;

	psci_function_id[PSCI_FN_CPU_OFF] = PSCI_0_2_FN_CPU_OFF;
	psci_ops.cpu_off = psci_cpu_off;

	psci_function_id[PSCI_FN_CPU_ON]	=	PSCI_FN_NATIVE(0_2,
CPU_ON);
	psci_ops.cpu_on = psci_cpu_on;

	psci_function_id[PSCI_FN_MIGRATE]	=	PSCI_FN_NATIVE(0_2,
MIGRATE);
	psci_ops.migrate = psci_migrate;

	psci_ops.affinity_info = psci_affinity_info;

	psci_ops.migrate_info_type = psci_migrate_info_type;

	arm_pm_restart = psci_sys_reset;
```

```
        pm_power_off = psci_sys_poweroff;
    }
```

2、CPU_operation 相关操作处理

```
@ Psci.c (kernel\arch\arm64\kernel)
    CPU 需要进行操作时 const struct cpu_operations cpu_psci_ops = {
    .name           = "psci",
#ifdef CONFIG_CPU_IDLE
    .cpu_init_idle      = cpu_psci_cpu_init_idle,
    .cpu_suspend        = cpu_psci_cpu_suspend,
#endif
    .cpu_init       = cpu_psci_cpu_init,
    .cpu_prepare= cpu_psci_cpu_prepare,
    .cpu_boot       = cpu_psci_cpu_boot,
#ifdef CONFIG_HOTPLUG_CPU
    .cpu_disable = cpu_psci_cpu_disable,
    .cpu_die= cpu_psci_cpu_die,
    .cpu_kill       = cpu_psci_cpu_kill,
#endif
};
```

会调用 psci_operations 对应函数同 BL31 进行交互。

3、代码讲解-suspend 操作

```
@Psci.c (kernel\drivers\firmware)
static const struct platform_suspend_ops psci_suspend_ops = {
    .valid              = suspend_valid_only_mem,
    .enter              = psci_system_suspend_enter,
    .prepare            = psci_system_suspend_prepare,
    .finish             = psci_system_suspend_finish,
};

static int psci_system_suspend(unsigned long unused)
{
    return invoke_psci_fn(PSCI_FN_NATIVE(1_0, SYSTEM_SUSPEND),
                        virt_to_phys(cpu_resume), 0, 0);
}

static int psci_system_suspend_enter(suspend_state_t state)
{
    return cpu_suspend(0, psci_system_suspend);
}
```

Enter 函数已经注册为 psci_system_suspend，该函数在 Linux 不进行相关的 suspend 操作，只是通过 SMC 指令传递一个 Function ID 告诉 BL31 我需要进行 suspend 操作，SMC 执行时陷入到 BL31.

三 代码下载及编译

代码获取

1、Arm trusted Firmware 的开源代码从下面代码下载：

git clone https://github.com/ARM-software/arm-trusted-firmware.git

目前支持的平台有：RK3368、RK3399、RK3328.

2、公司会对外发布各个平台的开发代码。

编译

1、Rockchip 平台通用的编译方法为：

CROSS_COMPILE=../gcc-linaro-aarch64-none-elf-4.9-2014.07_linux/bin/aarch64-none-elf- \

make DEBUG=0 LOG_LEVEL=40 ERROR_DEPRECATED=0 PLAT=rk3399 bl31

不同的平台通过 PLAT 定义，如果 PLAT=RK3328，即编译 RK3328 的平台。

2、 RK3399

1）、由于其中需要编译 m0 的代码，可以通过下面方式指定编译：

M0_CROSS_COMPILE=../../../../../../gcc-arm-none-eabi-4_8-2014q3/bin/arm-none-eabi-

2）、通过安装包

sudo apt-get install gcc-arm-none-eabi