关于应用了 Bootloader 后解决时钟冲突的补充文档

by John Li (nxa08200)

本文对在应用了Bootloader的情况下，如何解决A53 Linux与Bootloader，和M4 AutoSar系统有可能的时钟冲突，导致ATF启动失败的情况，做补充说明。

历史	说明	作者
V1	● 创建本文	John.Li

目录

1 需要的文档

文档	名称	获得方法
App Note	S32G_Bootloader_V*.pdf	https://community.nxp.com/t5/NXP-Designs-Knowledge-Base/S32G-Bootloader-Customzition/ta-p/1519838
App Note	AN13750: Enabling Multicore Application on S32G2 using S32G2 Platform Software Integration – Application note	从www.nxp.com下载
芯片手册	S32G2RM.pdf，S32G3RM.pdf	从www.nxp.com下载

2 背景说明

在使用 Bootloader 来启动 M7 AutoSar 系统与 A53 Linux 系统时，需要解决两个异构系统之间的时钟冲突问题，文档《S32G_Bootloader_V*.pdf》，描述了一种比较简单和合理的方法，即：

- 使用 M7 中运行 Bootloader 来配置 M7/A53 共同需要的时钟及源，比如说 Core Clock 及其源 PLL/DFS 等，所以在设计 Bootloader 时钟时，需要提前获得 Linux 的 Clock tree 做参考。
- 对于 A53 所需要的外设时钟，如果只是 A53 需要的，则 Bootloader 可以不配置，由 A53 的 ATF 自己去初始化，比如说 DDR 时钟。但是有一个例外就是 UART0 的时钟，可以预告配好，方便 ATF 的 Debug，此外，UART 也可能是 M7/A53 都需要的外设时钟。

在使用了以上方法后，可以做到 Bootloader 启动 G2 的 M7+A53，A53 Linux 端 ATF 代码可以不做修改就可以启动，从而简化了 Linux 的修改工作量，这个是基于以下原因：

- 对同样的寄存器赋同样的值，有可能不会导致硬件的挂死，比如如下代码：

  Atf/drivers/nxp/s32/clk/enable_clk.c:

```
static void setup_fxosc(struct s32gen1_clk_priv *priv)
{
        void *fxosc = priv->fxosc;

        /* According to "Initialization information" chapter from
         * S32G274A Reference Manual, "Once FXOSC is turned ON, DO NOT change
```

```
        * any signal (on the fly) which is going to analog module input.
        * The inputs can be changed when the analog module is OFF...When
        * disabling the IP through Software do not change any other values in
        * the registers for at least 4 crystal clock cycles."
        *
        * Just make sure that FXOSC wasn't already started by BootROM.
        */
       uint32_t ctrl;

       if (mmio_read_32(FXOSC_CTRL(fxosc)) & FXOSC_CTRL_OSCON)
               return;

       ctrl = FXOSC_CTRL_COMP_EN;
       ctrl &= ~FXOSC_CTRL_OSC_BYP;
       ctrl |= FXOSC_CTRL_EOCV(0x1);
       ctrl |= FXOSC_CTRL_GM_SEL(0x7);
       mmio_write_32(FXOSC_CTRL(fxosc), ctrl);

       /* Switch ON the crystal oscillator. */
       mmio_write_32(FXOSC_CTRL(fxosc),
                     FXOSC_CTRL_OSCON | mmio_read_32(FXOSC_CTRL(fxosc)));

       /* Wait until the clock is stable. */
       while (!(mmio_read_32(FXOSC_STAT(fxosc)) & FXOSC_STAT_OSC_STAT))
               ;
}
```

在实际的测试过程中，发现如果 Bootloader 已经使能了 FXOSC 后，就算 ATF 再次初始化它，也没有出现问题，当然，最恰当的方法是把此函数的调用从 ATF 中去掉。

- ATF 中对时钟的配置操作，如果代码发现需要配置的值，与之前配置的值是一样的，则会直接返回，不会操作硬件，如下示例：

Atf/drivers/nxp/s32/clk/enable_clk.c:

```
static int enable_pll(struct s32gen1_clk_obj *module,
                      struct s32gen1_clk_priv *priv, int enable)
{…
```

```
    if (clk_src == mux->source_id &&
        is_pll_enabled(pll_addr) &&
        get_module_rate(module, priv) == pll->vco_freq) {//此外如果读出来的时钟与要设置的时钟一致，则直接返回
            return 0;
    }

    return program_pll(pll, pll_addr, priv, mux->source_id);
}
```

所以说就算是 Bootloader 已经配置过 A53 的相关时钟，如果配置的值与 ATF 中需要配置的值是一样的，那么当 A53 ATF 代码去再次配置时，实际上也不会操作实际硬件，在建立好软件的 Clock Tree 数据结构后，就直接退出了。

不过在客户实际开发过程中，会遇到以下两种情况，导致客户可能需要修改 ATF 中的时钟相关代码才能做到与 Bootloader 集成：

- 客户的 Bootloader 与 Linux 开发团队往往是两个团队，所以在开发过程中可能做不到 Bootloader 里配置的 A53 相关时钟与 Linux 默认代码时钟是一样的，这个时钟 Linux 的团队往往会自己修改 ATF 的代码来暂时规避这个问题，这个规避可能是临时措施，也可能在无法协调 Bootloader 修改时会是一个永久的措施，这样的话，就需要修改 ATF 中的时钟配置代码。
- G2 的 M7=400Mhz, A53=1G，而 G3 的 M7=400Mhz, A53=1.3G，他们共享一个 Core PLL，而这一路 Core PLL 在满足 M7=400Mhz 的情况下，可以满足 A53=1G，但无法满足 A53=1.3G，也就意味做 G3 是肯定需要修改 ATF 代码来做一定的降频的，细节如下：

Core PLL 的 Clock Tree 框图如下：

24.1.2.1 Core-related clock overview

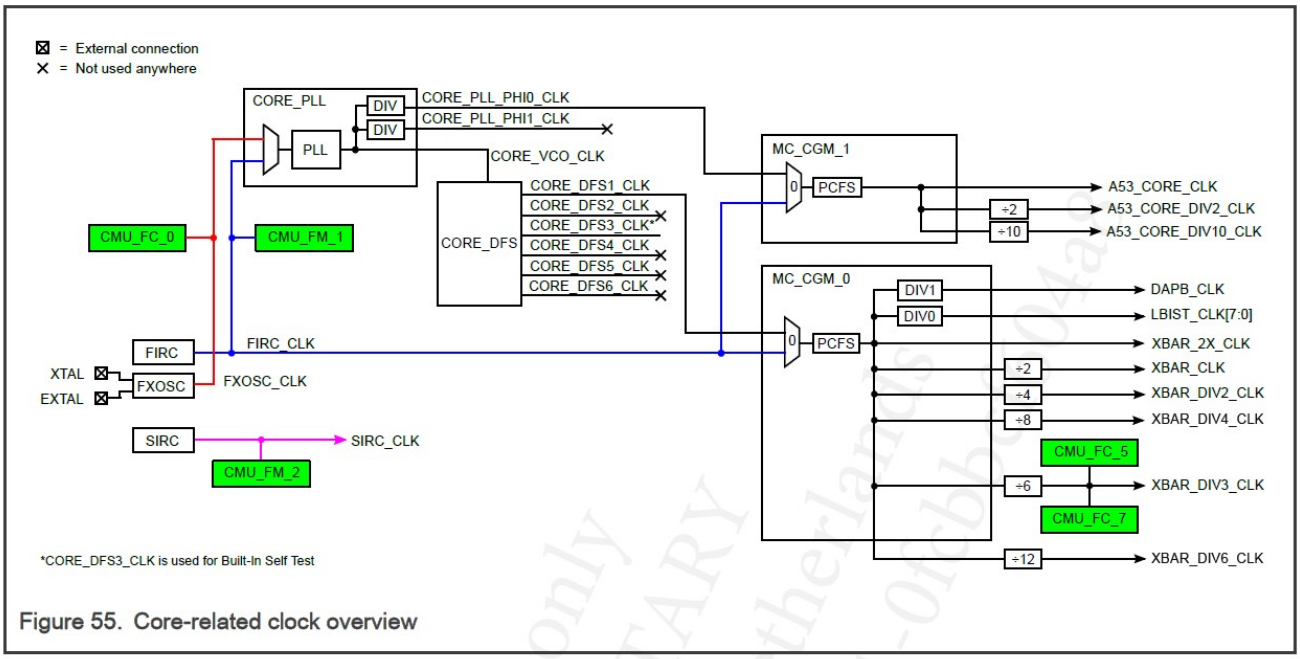

Figure 55. Core-related clock overview

所以 A53 Clock 来自于 Core_pll_ph0_clk=2G<-Core Pll/(PHI0_DIV+1)=1G，M7 来自于 XRAR_CLK=400Mhz<-XBAR_2X_CLK=800Mhz/2<-PCFS<-CORE_DFS1_CLK=800Mhz<-CORE_VCO_CLK=2Ghz/1.25=800Mhz。其中 PLL 与 PHI 公式如下：

- Integer-only mode:
  - When PLLDV[RDIV] is 0:

$$f_{pll\_VCO} = f_{pll\_ref} \times PLLDV[MFI]$$

Equation 1. PLL VCO frequency in integer-only mode when PLLDV[RDIV] is 0

  - When PLLDV[RDIV] is not 0:

$$f_{pll\_VCO} = \frac{f_{pll\_ref}}{PLLDV[RDIV]} \times PLLDV[MFI]$$

Equation 2. PLL VCO frequency in integer-only mode when PLLDV[RDIV] is not 0

- Fractional mode:
  - When PLLDV[RDIV] is 0:

$$f_{pll\_VCO} = f_{pll\_ref} \times \left(PLLDV[MFI] + \frac{PLLFD[MFN]}{18432}\right)$$

Equation 3. PLL VCO frequency in Fractional mode when PLLDV[RDIV] is 0

  - When PLLDV[RDIV] is not 0:

$$f_{pll\_VCO} = \frac{f_{pll\_ref}}{PLLDV[RDIV]} \times \left(PLLDV[MFI] + \frac{PLLFD[MFN]}{18432}\right)$$

Equation 4. PLL VCO frequency in Fractional mode when PLLDV[RDIV] is not 0

ee the equation below and the corresponding register configuration that determine the relationship between reference and LL_PHI*n* output frequencies.

$$f_{pll\_phi} = \frac{f_{pll\_VCO}}{PLLODV\_n[DIV+1]}$$

Equation 5. PLL PHI output frequency

DVFS 公式如下：

$$f_{dfsclkout} n = \frac{f_{dfsclkin} n}{2 \times \left(DVPORTn[MFI] + \frac{DVPORTn[MFN]}{36}\right)}$$

Figure 141. Equation for input and output clocks of each port divider

所以在满足 M7=400Mhz 的前提下，又希望保持整数倍 Divider 的情况下，则可以选择：DVFS.MFN=18，MFI=1，则 800MhzX2X(1+18/36)=2400，则 A53 可以最高配置为 2400/2=1200Mhz。当然，在考虑小数 divider 的情况下可以配置得更加接近并小于 1.3Ghz。

而 G3 Linux ATF 中的配置为：

Atf/include/dt-bindings/clock/s32gen1-clock-freq.h:

```
#elif defined(PLAT_s32g3)
#define S32GEN1_A53_MAX_FREQ                    (1300 * MHZ)
#define S32GEN1_A53_MIN_FREQ                    (48 * MHZ)
#define S32GEN1_ARM_PLL_VCO_MAX_FREQ            (2600 * MHZ)
#define S32GEN1_ARM_PLL_PHI0_MAX_FREQ           (1300 * MHZ)
#define S32GEN1_A53_FREQ                        (1300 * MHZ)
#define S32GEN1_ARM_PLL_VCO_FREQ                (2600 * MHZ)
#define S32GEN1_ARM_PLL_PHI0_FREQ               (1300 * MHZ)
#define S32GEN1_XBAR_2X_FREQ                    (793236859UL)
```

所以 Linux 默认是配置为 A53=1.3G，而 M7=793Mhz/2，这个与实际 M7 的 Mcal 要求是不同的。

根据以上分析，所以在 A53 的 ATF 中，需要做的 clock 相关修改可能最小到最大包括如下：

3 A53 ATF 初始化的时钟

参考代码：

Atf/drivers/nxp/s32/clk/early_clocks.c

```
int s32_plat_clock_init(void)
{..
s32_enable_a53_clock();
enable_lin_clock();
setup_periph_pll();
enable_sdhc_clock(); or enable_qspi_clock();
s32_enable_ddr_clock();
```

所以实际影响 ATF 启动的 clock 就是如上几个，而一般建议：

- Bootloader 中需要配置 M7 的时钟与源，由于 M7 与 A53 共源，所以建议 Bootloader 中将 M7 与 A53 时钟一起配置，而因为将就 M7=400Mhz 的要求，所以 A53 需要相应降频，比如说如前所说配置为 1.2G。
- 串口时钟也建议由 Bootloader 配置，因为 Mcal 也可能使用串口 1，还有就是为了提前打开调试串口。
- QSPI NOR 时钟肯定是由 Bootloader 配置，SDHC 可以由 Bootloader 来配置，也可以由 A53 配置。
- DDR 时钟一般由 A53 去配置。
- 以上代码会将时钟及其 clock tree 上的所有源全部配置。

所以经上分析可以得出实际上真正需要 A53 配置的 early clock 只有 DDR(或加上 SDHC)。其它的时钟配置可以去掉，但是不能简单注掉，之后说明。

4 A53 ATF 时钟的 Debug 支持方法

参考代码：

Atf\plat\nxp\s32\s32g\s32g_bl2_el3.c

```
void bl2_el3_early_platform_setup(…)
{
  s32_early_plat_init
  |-> s32_plat_config_pinctrl
  |    -> linflex_config_pinctrl
  |-> s32_plat_clock_init
  console_s32_register
```

所以 ATF 代码本身的设计思路是没有考虑 Bootloader 初始化时钟的情况，他是先配置了时钟，再配置管脚，最后注册串口，在注册串口代码之后，串口就可以工作了。这样在配置时钟是 ATF 挂死的话，是没有打印的，导致调试困难。

考虑 Bootloader 已经初始化了串口时钟的情况下，可以简单的将 console_s32_register 调用提到 s32_early_plat_init 之前，这样时钟初始化过程中就可以使用串口打印 Debug 了。(先初始化串口，再配置管脚也没有问题)。

5 G3 A53 ATF 最小修改方法

根据之前的分析，在 G3 上，最简单的 ATF 修改方法应该是：

Atf/include/dt-bindings/clock/s32gen1-clock-freq.h:

```
#elif defined(PLAT_s32g3)
#define S32GEN1_A53_MAX_FREQ                    (1300 * MHZ) //(1300 * MHZ)
#define S32GEN1_A53_MIN_FREQ                    (48 * MHZ)
#define S32GEN1_ARM_PLL_VCO_MAX_FREQ            (2400 * MHZ)  //(2600 * MHZ)
#define S32GEN1_ARM_PLL_PHI0_MAX_FREQ           (1200 * MHZ)  //(1300 * MHZ)
#define S32GEN1_A53_FREQ                        (1200 * MHZ)  //(1300 * MHZ)
#define S32GEN1_ARM_PLL_VCO_FREQ                (2600 * MHZ)
#define S32GEN1_ARM_PLL_PHI0_FREQ               (1200 * MHZ)  // (1300 * MHZ)
#define S32GEN1_XBAR_2X_FREQ                    (800 * MHZ)//(793236859UL)
```

这样如果 Bootloader 已经配置了 M7=400Mhz, A53=1200Mhz，则 Linux 的配置与 Bootloader 配置相同，就不会再实际操作时钟寄存器配置硬件了。

当然，最好建议以下 FXOSC 的调用也注掉，它会实际操作寄存器：

Atf/drivers/nxp/s32/clk/enable_clk.c:

```
static int enable_osc(struct s32gen1_clk_obj *module,
                      struct s32gen1_clk_priv *priv, int enable)
{…
    case S32GEN1_FXOSC:
            if (enable)
            {
                    //setup_fxosc(priv);
            }
            else
                    disable_fxosc(priv);
            return 0;
…
}
```

6 A53 ATF 最大修改方法(可选)

如之前说明，理论上使用第 5 章的方法，结合 Bootloader 正确配置了和 Linux 一样的时钟，就可以正常 Boot ATF 了，但是有一些客户的 Linux 团队也希望注掉更多的代码来确保 ATF 的 Boot 不会因为设置时钟与 Bootloader 不一致时挂掉，但是不能简单的注掉时钟初始化调用，说明如下：

以 s32_enable_a53_clock 为例：

```
int s32_enable_a53_clock(void)
```

```
{
    int ret;
    unsigned long rate;

    ret = s32gen1_get_early_clks_freqs(&early_freqs);
    if (ret)
            return ret;

    ret = s32gen1_set_parent(&arm_pll_mux, &fxosc);
    if (ret)
            return ret;

    ret = s32gen1_set_parent(&mc_cgm1_mux0, &arm_pll_phi0);
    if (ret)
            return ret;

    rate = s32gen1_set_rate(&fxosc, S32GEN1_FXOSC_FREQ);
    if (rate != S32GEN1_FXOSC_FREQ)
            return -EINVAL;

    rate = s32gen1_set_rate(&arm_pll_vco,
                            early_freqs.arm_pll_vco_freq);
    if (rate != early_freqs.arm_pll_vco_freq)
            return -EINVAL;

    rate = s32gen1_set_rate(&arm_pll_phi0,
                            early_freqs.arm_pll_phi0_freq);
    if (rate != early_freqs.arm_pll_phi0_freq)
            return -EINVAL;

    rate = s32gen1_set_rate(&a53_clk,
                            early_freqs.a53_freq);
    if (rate != early_freqs.a53_freq)
            return -EINVAL;
```

```
    ret = s32gen1_enable(&a53_clk, 1);
    if (ret)
            return ret;

    return enable_xbar_clock();
}
```

所以实际上此函数会建立整个 A53 时钟的 clock tree，这个函数调用中如 s32gen1_set_parent，s32gen1_set_rate 之类的调用，实际上只会建立 clock tree 的软件链表，最终依赖 s32gen1_enable 调用将整个 clock tree，也就是 A53 需要的所有 clock 及其时钟源配置好。

这样如果将此函数的调用注掉，则此 clock tree 的软件链表就不会建立起来，这样在其它 clock 也依赖相同源的时候，就会直接找不到时钟源而退出。

所以正确的做法是把实际配置寄存器的代码注掉，所以：

```
int s32_enable_a53_clock(void)
{…
/*
    ret = s32gen1_enable(&a53_clk, 1);
    if (ret)
            return ret;
*/
…
}
static int enable_lin_clock(void)
{…
    return 0;//s32gen1_enable(&lin_baud, 1);
}

static int enable_sdhc_clock(void)
{..
    return 0;// s32gen1_enable(&sdhc, 1);// 如果 Bootloader 已经初始化了 SDHC 的情况下
}
s32_enable_ddr_clock();//保持不变
```

7 ATF Clock Tree 初始化代码分析

客户在解决 ATF 与 Bootloader&MCAL clock 冲突时，除开调试手段的缺乏，最主要的原因还是对 ATF 的 clock 初始化机制不了解，所以本章简单说明一下，以作参考。还是以 s32_enable_a53_clock 为例，它首先建立好 clock tree 链表：

Fxosc-> arm_pll_mux-> arm_pll_phi0-> mc_cgm1_mux0。

然后再设置好 clock tree 上各个节点的时钟值。

Fxosc= S32GEN1_FXOSC_FREQ= (40 * MHZ)

arm_pll_vco= S32GEN1_ARM_PLL_VCO_FREQ= (2400 * MHZ) // (2600 * MHZ)

arm_pll_phi0= S32GEN1_ARM_PLL_PHI0_FREQ= (1200 * MHZ) // (1300 * MHZ)

a53_clk= S32GEN1_A53_FREQ=(1200 * MHZ) // (1300 * MHZ)

最终是依赖 s32gen1_enable(&a53_clk, 1);实际配置与打开所有这些时钟

```
struct s32gen1_clk *clk;
struct s32gen1_clk_priv *priv;
priv = s32gen1_get_clk_priv(c);
 clk = get_clock(c->id);
|->enable_module(&clk->desc, priv, enable);
static struct clk a53_clk = CLK_INIT(S32GEN1_CLK_A53_CORE=S32GEN1_PERIPH_CLK(3);
#define CLK_INIT(ID)        \
{                           \
    .id = (ID),             \
    .drv = &fake_clk_dev, \
}
static struct s32gen1_clk_priv s32_priv = {
    .accelpll = (void *)ACCEL_PLL_BASE_ADDR,
    .armdfs = (void *)ARM_DFS_BASE_ADDR,
    .armpll = (void *)ARM_PLL_BASE_ADDR,
    .cgm0 = (void *)MC_CGM0_BASE_ADDR,
    .cgm1 = (void *)MC_CGM1_BASE_ADDR,
    .cgm2 = (void *)MC_CGM2_BASE_ADDR,
    .cgm5 = (void *)MC_CGM5_BASE_ADDR,
    .ddrpll = (void *)DRAM_PLL_BASE_ADDR,
    .fxosc = (void *)S32_FXOSC_BASE_ADDR,
```

```
    .mc_me = (void *)MC_ME_BASE_ADDR,
    .periphdfs = (void *)PERIPH_DFS_BASE_ADDR,
    .periphpll = (void *)PERIPH_PLL_BASE_ADDR,
    .rdc = (void *)RDC_BASE_ADDR,
    .rgm = (void *)MC_RGM_BASE_ADDR,
};

static struct clk_driver fake_clk_dev = {
    .data = &s32_priv,
};
[CC_ARR_CLK(S32GEN1_CLK_A53_CORE)] = &a53_core_clk,
/* A53_CORE */
static struct s32gen1_clk a53_core_clk =
        S32GEN1_FREQ_MODULE_CLK(cgm1_mux0_clk, S32GEN1_A53_MIN_FREQ,
                                S32GEN1_A53_MAX_FREQ);
```

|-> // enable_module 函数利用嵌套函数的方式，将本 clock 及所有的父源 clock 的 enable_clbs 函数从叶到根调用一次，来初始化整个 clock tree 时钟。

```
order = get_en_order(module, enable);
    if (order == PARENT_FIRST) {
        first_en = enable_module;
        first_mod = parent;
        second_en = enable_clbs[index];
        second_mod = module;
    } else {
        first_en = enable_clbs[index];
        first_mod = module;
        second_en = enable_module;
        second_mod = parent;
    }

    ret = first_en(first_mod, priv, enable);
    if (ret)
        return ret;

    ret = second_en(second_mod, priv, enable);
```

```
    if (ret)
            return ret;

static const enable_clk_t enable_clbs[] = {
    [s32gen1_clk_t] = no_enable,
    [s32gen1_part_block_t] = enable_part_block,
    [s32gen1_shared_mux_t] = enable_mux,
    [s32gen1_mux_t] = enable_mux,
    [s32gen1_cgm_div_t] = enable_cgm_div,
    [s32gen1_dfs_div_t] = enable_dfs_div,
    [s32gen1_dfs_t] = enable_dfs,
    [s32gen1_pll_t] = enable_pll,
    [s32gen1_osc_t] = enable_osc,
    [s32gen1_fixed_clk_t] = no_enable,
    [s32gen1_fixed_div_t] = no_enable,
    [s32gen1_pll_out_div_t] = enable_pll_div,
};
```

考虑最终根源 arm_pll 时钟的类型：

```
static struct s32gen1_pll armpll = {
    .desc = {
            .type = s32gen1_pll_t,
    },
    .ndividers = 2,
    .source = &arm_pll_mux_clk.desc,
    .instance = S32GEN1_ARM_PLL,
};
```

以最终源时钟 arm pll 的初始化调用为例：

```
static int enable_pll(struct s32gen1_clk_obj *module,
                      struct s32gen1_clk_priv *priv, int enable)
{…
if (clk_src == mux->source_id &&
        is_pll_enabled(pll_addr) &&
        get_module_rate(module, priv) == pll->vco_freq) {//此处如上所说，如果已经配置了同样的时钟，则直接退出。
```

```
                return 0;
        }

        return program_pll(pll, pll_addr, priv, mux->source_id);//此处根据 clock 要求计算寄存器值，实际设置寄存器。
}
```

其它 clock tree 节点的时钟初始化调用可以参考分析。