实验 6 鸿蒙 LiteOS-a 内核移植——系统时钟移植

22920212204396　　黄子安

一、实验目的

1、为 demochip 单板移植时钟

2、对操作系统的时钟及时钟中断等做进一步了解

二、实验环境

1、物理机：Windows

2、虚拟机：Ubuntu 18.04.6

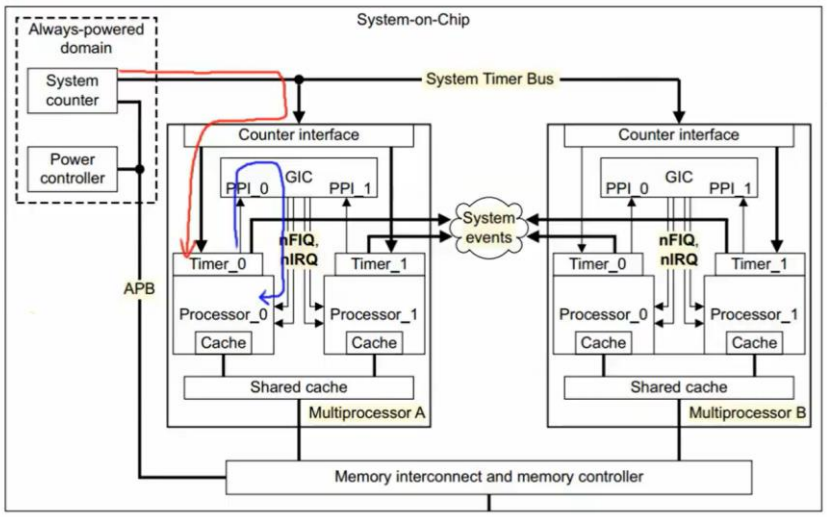
3、开发板：IMX6ULL mini

三、实验内容

通用定时器：操作系统必须要有系统时 钟，但是各种不同的芯片使用不同的定时器会为编程带来难度， 在 ARM v7 架构中提出了通用计时器

通用定时器的硬件框图如下所示，包含两个部分：

- System counter：给所有处理器提供统一的时间，该部分与外界的时钟源相连，时钟源在硬件上无法统一，当时钟源每次产生一个脉冲 System counter 就自增 1，之后传给后面的 Timer
- Timer（定时器）：可以设置周期性的事件，给处理器提供中断信号，每个处理器都有各自的不同 Timer；每个 Timer 可以**各自设置比较值**，当 System Counter 的值达到这个比较值就产生一次时钟中断给 GIC（中断控制器），这也就决定了每个定时器产生中断的时间可以不一样



System Counter 特性

规格	描述
位宽(Width)	至少56位，跟硬件实现。 读取时，可以得到64位的数值。
频率(Frequency)	1M～50MHz，增加值可以调整： 比如时钟为8MHz时，每来一个时钟计数值增加1， 设置为4MHz时，每来一个时钟计数值增加2， 降低频率时可以降低功耗，同时增加步进值以维持时钟精度
溢出(Roll-over)	不少于40年
精度(Accuracy)	推荐：误差在24小时内不超过10秒
复位值(Start-up)	从0开始

System Counter 两种访问方式

SystemCounter 是给所有 Processor 使用的，它有两种访问方式：

- CP15 协处理器命令：某个 Processor 去访问它时可以使用 CP15 协处理器命令。
- MemoryMapped 寄存器：既然它是给所有 Processor 使用的，那么应该提供更高级的访问方法(System Level)，而且有些 Processor 并没有实现 CP15，所有也应该提供 MemoryMapped（内存映射）的方法，**CPU 通过修改内存就可以等价修改 System Counter**

Timer 特性

每个 Processor 都有一个 Timer，它有 3 个寄存器，只能使用协处理器命令方位(CP15)

1、64 位的比较寄存器(CVAL)：当 SystemCounter 的值等于它时，产生事件(中断)

- SystemCounter 总是增长的，所以 Timer 的 64 位比较寄存器也只能设置为大于 SystemCounter 的值
- 被称为 upcounter

2、32 位的 TimerValue 寄存器(TVAL)

- 它是 downconter
- 比如设置为 1000，表示再经过 1000 个时钟之后，就会产生事件(中断)
- 实质：设置 64 位的比较寄存器，让它等于 SystemCounter+1000

3、32 位的控制寄存器(CTL)

- 使能/禁止 Timer
- 使能输出：是否能产生事件(中断) * 状态：是否能产生了事件(中断)

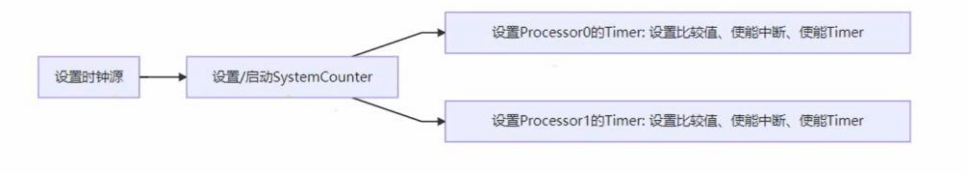
Table B8-1 Timer registers summary for the Generic Timer

	PL1 physical timer[a]	PL2 physical timer[b]	Virtual timer
CompareValue register	CNTP_CVAL	CNTHP_CVAL	CNTV_CVAL
TimerValue register	CNTP_TVAL	CNTHP_TVAL	CNTV_TVAL
Control register	CNTP_CTL	CNTHP_CTL	CNTV_CTL

Generic Timer 源码分析

使用通用定时器的三个步骤：1.设置外部时钟源 2.使能 System Counter 3.设置 Timer 的比较值

前两个步骤在 u-boot 中完成了，根据视频只分析第三部分的鸿蒙源码



该部分源码位于 drivers\timer\arm_generic_timer.c 文件中，重要的函数是有以下三个

1、HalClockInit 初始化函数：HalClockFreqRead 函数读出频率（时钟源的频率），之后注册时钟中断，也就是在内核中注册到向量表中并设置中断处理函数的入口

```
LITE_OS_SEC_TEXT_INIT VOID HalClockInit(VOID)
{
    UINT32 ret;
    SET_SYS_CLOCK(HalClockFreqRead());
    ret = LOS_HwiCreate(OS_TICK_INT_NUM, OS_HWI_PRIO_LOWEST, 0, HalTickEntry, 0);
    if (ret != LOS_OK) {
        PRINT_ERR("%s, %d create tick irq failed, ret:0x%x\n", __FUNCTION__, __LINE__, ret);
    }
}
```

时钟中断的中断号是 29，该中断号定义在了 STM157 的手册中，根据视频说法这里 29 和 30 都和时钟有关，经过测试是 29 号中断

PPI (Active Low level sensitive)				
-	16 to 24	-	Reserved	-
-	25	PPI6	Virtual maintenance interrupt.	-
-	26	PPI5	Hypervisor timer event.	-
-	27	PPI4	Virtual timer event.	-
-	28	PPI0	Legacy nFIQ signal. Not used.	-
-	29	PPI1	Secure physical timer event.	-
-	30	PPI2	Non-secure physical timer event.	-
-	31	PPI3	Legacy nIRQ signal. Not used.	-

2、HalClockStart：初始化之后由 HalClockStart 去使能定时器，该函数先使能 29 号中断（代码阅读时发现在中断控制器初始化的时候会禁用所有中断，需要调用 Unmask 函数来使能中断），Tval 将 Timer 的 CVAL（比较寄存器）设置为 OS_CYCLE_PER_TICK，也就是在 10ms 后会触发第一次时钟中断；Ctl 寄存器是控制寄存器，为 0 时禁用 Timer，为 1 时启用 Timer

```
#define LOSCFG_BASE_CORE_TICK_PER_SECOND 100
#define OS_CYCLE_PER_TICK (g_sysClock / LOSCFG_BASE_CORE_TICK_PER_SECOND)
```

```
LITE_OS_SEC_TEXT_INIT VOID HalClockStart(VOID)
{
    (void)ArchIrqUnmask(OS_TICK_INT_NUM);

    /* triggle the first tick */
    TimerCtlWrite(0);
    TimerTvalWrite(OS_CYCLE_PER_TICK);
    TimerCtlWrite(1);
}
```

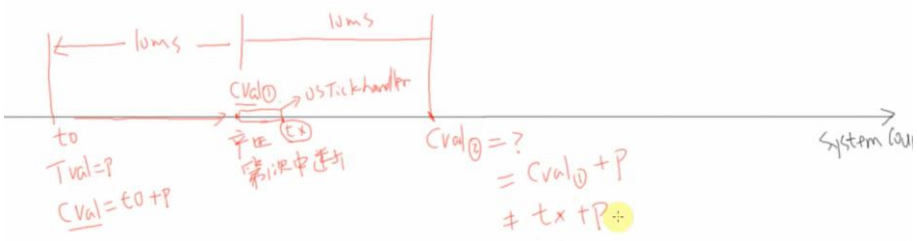
3、HalTickEntry：中断处理函数入口，这里会调用对应的处理函数 OsTickHandler；之后重新设置 Cavl 寄存器为当前值加上 1000，也就是在下一个 10ms 让 Timer 产生一次中断

```
LITE_OS_SEC_TEXT VOID HalTickEntry(VOID)
{
    TimerCtlWrite(0);

    OsTickHandler();

    /*
     * use last cval to generate the next tick's timing is
     * absolute and accurate. DO NOT use tval to drive the
     * generic time in which case tick will be slower.
     */
    TimerCvalWrite(TimerCvalRead() + OS_CYCLE_PER_TICK);
    TimerCtlWrite(1);
}
```

这里有一点要注意的是 osTickhandle 的时间是算在下一个 10ms 里了，10ms 和 10ms 之间是没有空的



四、实验心得

本次实验从硬件层了解了计算机中的系统时钟，了解了**时钟源脉冲信号变成时钟中断**的过程，可以算作是中断部分代码阅读和任务调度部分的硬件补充，Timer 和多处理器这一点是之前完全没有接触到的，同时也看到了计算机对于硬件的强大可编程性，可以在 OS 软件层去修改时钟中断的周期，体现了数字电路的灵活性。