Smart Hardware Design

Conception de matériel intelligent

Diseño de hardware inteligente

スマートハードウェア設計

智能硬件设计

第四章 智能硬件的计时外设

Design de hardware inteligente

تصميم الأجهزة الذكية

Slimme hardwareontwerpen

Σχεδίαση έξυπνου υλικού

大连理工大学-朱明

Progettazione di hardware intelligente

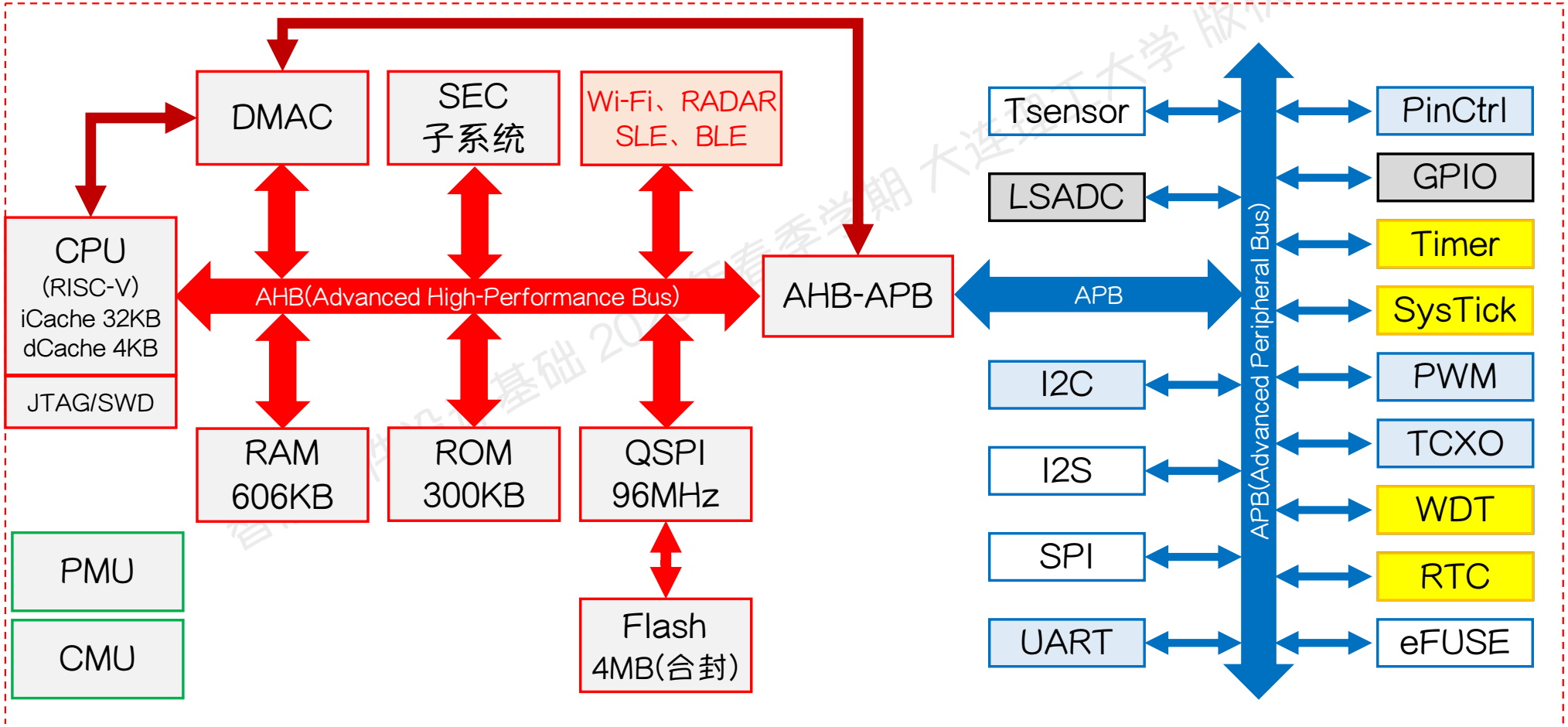
스마트 하드웨어 설계

Smart-Hardware-Design

Проектирование умного оборудования

4.0 思考回顾

- [4.0.0] 智能硬件的SoC内部结构(华为海思WS63)



4.1 智能硬件系统中的时间

- [4.1.1] 时间的作用
  - 时间是描述事物变化和事件发生顺序和间隔的基本概念
    - 时间的基本单位：秒s，信息系统常用的还包括毫秒ms和微秒us
  - 信息系统中的时间的作用
    - 任务调度与运行：实现任务的调度与协作；实现定时任务处理等
    - 数据采集与处理：对采集的数据进行时间标记，便于后续处理
    - 通信与同步：与服务器之间的时钟同步；控制通信协议
    - 实时响应与控制：根据系统设定的时间要求做出响应和控制
    - 系统节能与优化：根据时间管理控制系统工作在不同能耗等级下
    - 系统日志：位系统日志附加时间信息，便于追踪和故障排查
    - 数据分析与智能决策：根据时间分析用户的行为习惯，优化系统功能

4.1 智能硬件系统中的时间

- [4.1.2] 时间的表示方法
  - 信息系统中时间的表示方法
    - 时间戳形式 - 大尺度计时
      - Unix时间：自1970年1月1日0时0分0秒起的秒数
        - 如2025年01月29日0时0分0秒为：1738080000
      - ISO8601标准(更易于人类理解)
        - 如2025-01-29T01:23:34+08:00、2025-03-01T02:34:45Z
    - 时间间隔 - 小尺度计时
      - 两个时间点时间间隔
    - 逻辑时间
      - 逻辑时间表示事件的相对顺序

智能硬件系统内部一般使用时间间隔进行计时和任务调度，使用时间戳进行用户交互

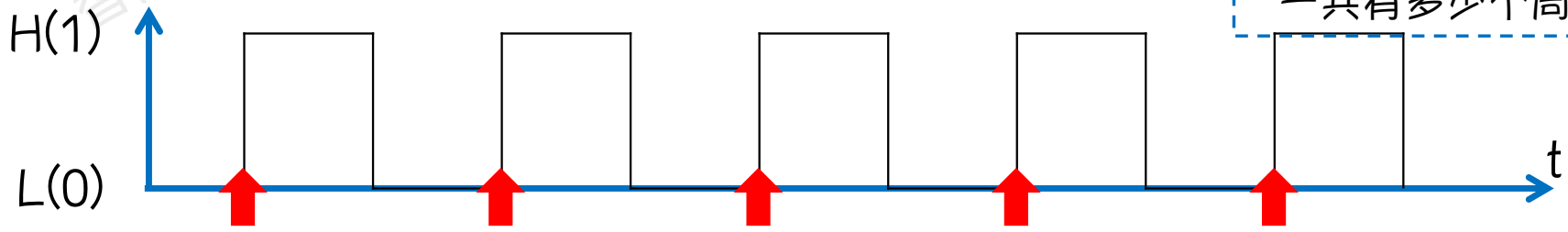
4.1 智能硬件系统中的时间

- [4.1.3] 计时的实现基础
  - 人类计时的历史
    - 古代：日晷、漏壶、沙漏
    - 近代：机械钟、摆钟、石英钟
    - 现代：原子钟

计时实际上是对一种可被测量的发生的时间间隔比较稳定的事件进行计数

  - 信息设备(智能硬件设备)的计时实现基础
    - 有能够产生稳定的固定频率信号的器件
    - 有能够对该器件产生的信号进行捕获/采集的电路或功能
    - 有能够对该捕获信号的周期进行统计的电路或功能

第一个红色尖头开始
一共有多少个周期



4.1 智能硬件系统中的时间

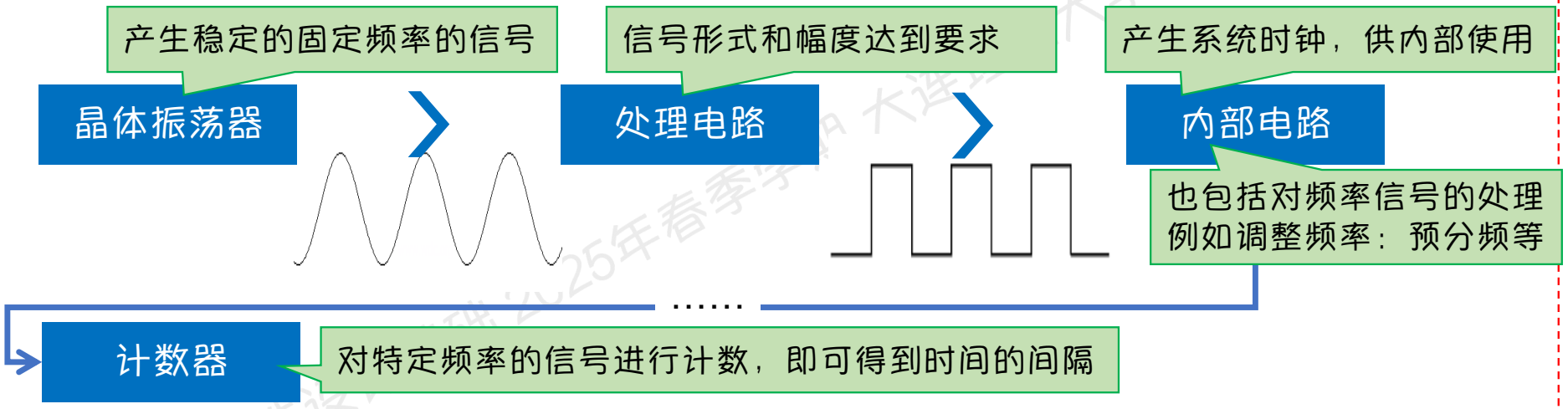
- [4.1.3] 计时的实现基础
  - 信息设备(智能硬件设备)的计时实现基础
    - 有能够产生稳定的特定频率信号的器件 -> 晶体振荡器或振荡电路
    - 有能够对该器件产生的信号进行捕获/采集的电路或功能
    - 有能够对该捕获信号的周期进行统计的电路或功能

    > 电路简单
  - 常用的稳定的频率信号源
    - 民用信息设备一般使用晶体振荡器作为高精度频率信号源
    - A quartz crystal oscillator is a highly stable electronic oscillator that uses the mechanical resonance of a vibrating quartz crystal to create an electrical signal with a precise frequency. Its purpose is to provide a reliable and accurate time or frequency reference for electronic systems.

石英晶体的机械共振在外加电压下会产生稳定的振荡信号

4.1 智能硬件系统中的时间

- [4.1.3] 计时的实现基础
  - 信息设备(智能硬件设备)计时系统的主要结构



  - 理论计算演示
    - 计数器输入频率为32768Hz，计数器进行计数16384次的时间是多少
    - 计数器输入频率为240MHz，计数器长度16位，达到最大计数时间是多少

4.2 通用定时器

[4.2.1] 定时器系统的基本结构

- WS63定时器的简化结构

- 系统时钟CLK(Clock)
  - 由晶体振荡器(其他振荡器/信号源)输出的信号经处理后产生时钟信号
  - 时钟信号可以被配置调整频率，如进行分频或PLL倍频等
  - MCU/SoC内部可能会有多个系统时钟，提供不同的频率或备份
  - 系统时钟为CPU、RAM和各类外设提供运行必须的时钟信号
  - 同一外设，工作的时钟频率越高，功耗就越高

4.2 通用定时器

[4.2.1] 定时器系统的基本结构

- WS63定时器的简化结构

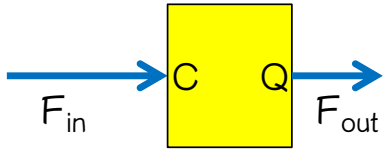
- (二进制)预分频器PSC(Prescaler)：产生定时器时钟
  - 降低特定模块(Timer)的时钟信号的频率，以实现灵活的控制要求
  - 只有单一时钟的系统为满足CPU的运行需求，系统时钟频率一般比较高
    - 用户端上通用定时器的定时时间一般在宏观级别计时(毫秒和秒级别)
    - 高频率的系统时钟很难在8位或16位计数器上直接实现上述定时时间
  - 思考：解决高频系统时钟与宏观级别计时需求之间的矛盾的解决方法

4.2 通用定时器

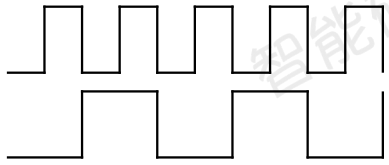
[4.2.1] 定时器系统的基本结构

- WS63定时器的简化结构
  - 计数器CNT(Counter)与自动重装载器ARR(Auto Reload Register)

设计一种定时器结构



设计一种元器件：触发器
只响应下降沿，检测到
下降沿，输出电平翻转



$$F_{out} = \frac{1}{2} F_{in}$$

N个触发器串联：只能实现二进制数值的计数
例如实现2、4、8、16、32、64、128、256等

$$F_{overflow} = \frac{1}{2^N} F_{clk}$$

假设N=8，要实现的计数值目标是97，需要如何处理：97=0x61=0b01100001


```
if((BIT0==1) && (BIT1==0) && (BIT2==0) && (BIT3==0) && (BIT4==0)
   && (BIT5==1) && (BIT6==1) && (BIT7==0)) { /* 97 NOW */}
```

可由硬件实现

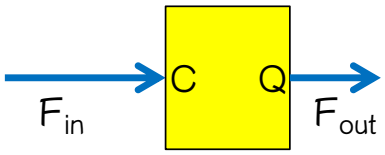
二进制计数器与逻辑判断机制配合，可以实现任意值的计数功能

4.2 通用定时器

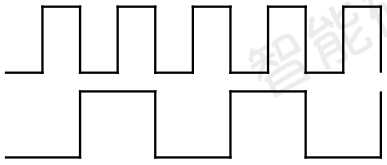
●[4.2.1] 定时器系统的基本结构

- WS63定时器的简化结构
  - 计数器CNT(Counter)与自动重装载器ARR(Auto Reload Register)

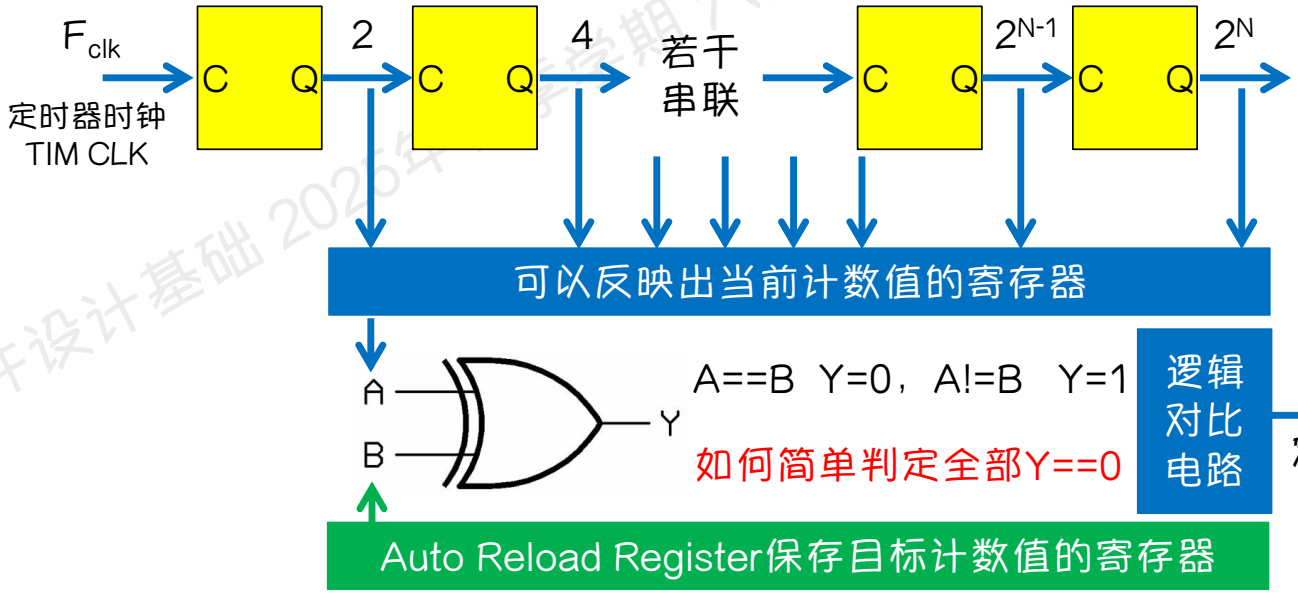
设计一种定时器结构



设计一种元器件：触发器
只响应下降沿，检测到
下降沿，输出电平翻转



$$F_{out} = \frac{1}{2}F_{in}$$

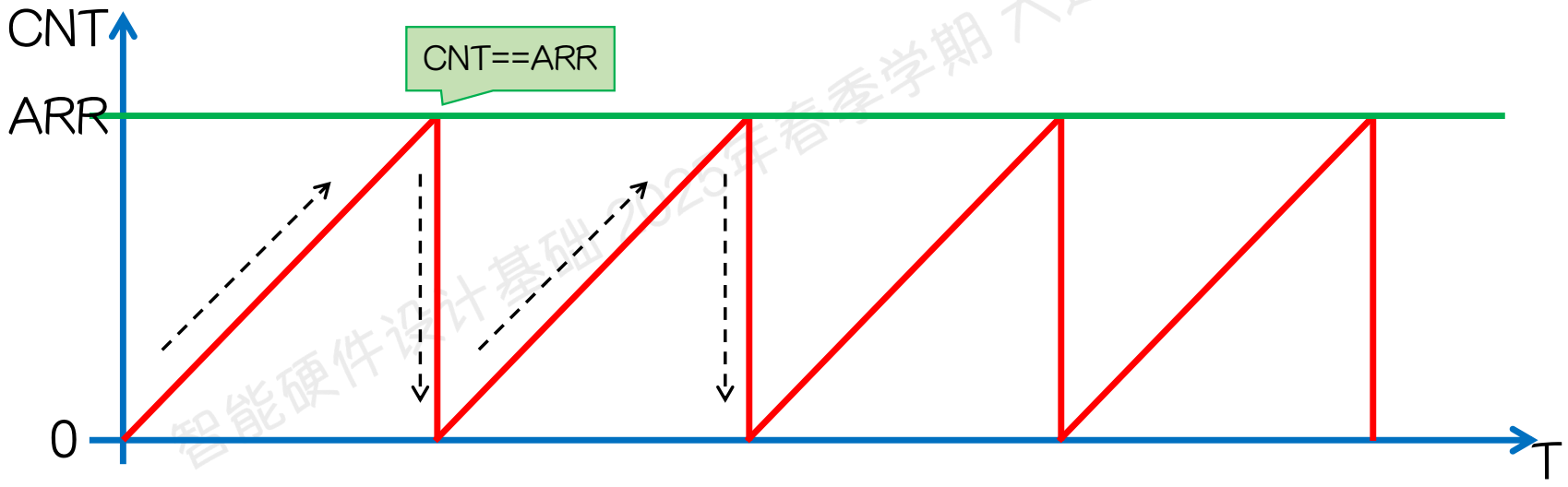


本图是增计数器示例，实际定时器实现可能有差别，工作模式也更丰富

4.2 通用定时器

[4.2.1] 定时器系统的基本结构

- WS63定时器的简化结构
  - 计数器CNT(Counter)与设定值(ARR)之间的比较关系

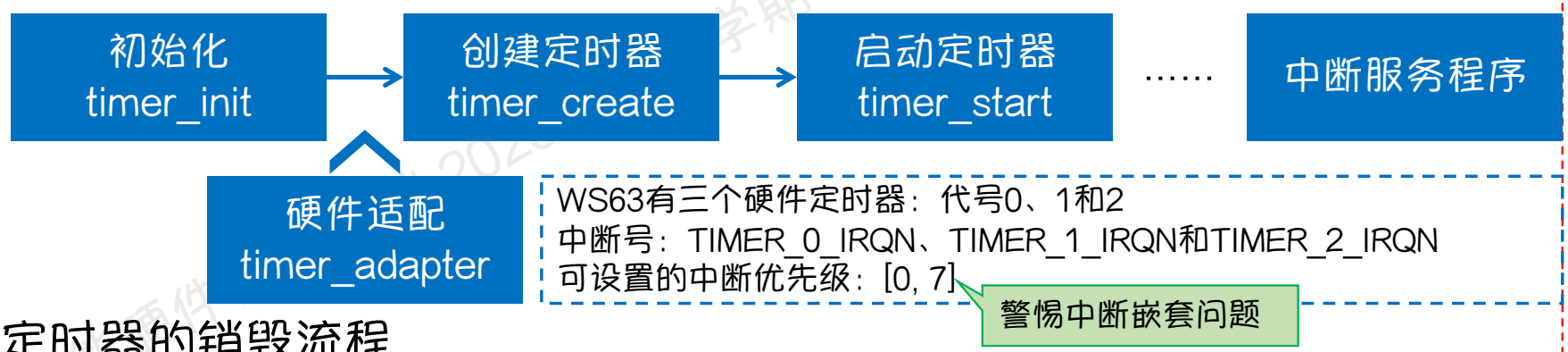


大部分MCU/SoC都会用类似图片解释定时器的工作原理

4.2 通用定时器

[4.2.2] 定时器系统的基本使用

- WS63定时器的控制流程
  - 定时器在智能硬件系统中的目的：特定时间间隔后的提醒
  - 定时器的启用流程



  - 定时器的销毁流程

4.2 通用定时器

●[4.2.2] 定时器系统的基本使用

- WS63 Timer的主要API

①：errcode_t uapi_timer_init(void)	初始化定时器功能
②：errcode_t uapi_timer_deinit(void)	去初始化定时器功能
③：errcode_t uapi_timer_adapter(timer_index_t index, uint32_t int_id, uint16_t int_priority)	适配定时器
④：errcode_t uapi_timer_create(timer_index_t index, timer_handle_t *timer)	创建定时器
⑤：errcode_t uapi_timer_delete(timer_handle_t timer)	删除定时器
⑥：errcode_t uapi_timer_start(timer_handle_t timer, uint32_t time_us, timer_callback_t callback, uintptr_t data)	启动定时器
⑦：errcode_t uapi_timer_stop(timer_handle_t timer)	停止定时器
⑧：errcode_t uapi_timer_get_current_time_us(timer_index_t index, uint32_t *current_time_us)	获取当前的定时器时间数值

4.2 通用定时器

[4.2.2] 定时器系统的基本使用

- WS63定时器的控制流程：适配
  - 硬件复杂性：三套定时器，八级优先级
  - errcode_t uapi_timer_adapter(timer_index_t index, uint32_t int_id, uint16_t int_priority)
    - timer_index_t index：硬件定时器代号，共三个，代号0、1、2
    - uint32_t int_id：必须与硬件定时器对应的中断源，0对应TIMER_0_IRQN，1对应TIMER_1_IRQN，2对应TIMER_2_IRQN
    - uint16_t int_priority：中断(中断源下所有中断)优先级，范围是[0, 7]

```
#define   TIMER_INDEX        1              //硬件定时器的代号：1
#define   TIMER_PRIO         1              //中断优先级(后续create一致)
uapi_timer_init();
uapi_timer_adapter(TIMER_INDEX, TIMER_1_IRQN, TIMER_PRIO);
```

4.2 通用定时器

[4.2.2] WS63定时器系统的基本使用

- WS63定时器的控制流程：创建
  - errcode_t uapi_timer_create(timer_index_t index, timer_handle_t *timer)
    - timer_index_t index：硬件定时器代号，共三个，代号0、1、2
    - timer_handle_t *timer：返回值，通过其可实现对定时器的访问和控制

```
#define TIMER_TIMERS_NUM        4          //将创建四个定时器
timer_handle_t timer_index[TIMER_TIMERS_NUM] = { 0 };
for (uint32_t i = 0; i < TIMER_TIMERS_NUM; i++) {
    uapi_timer_create(TIMER_INDEX, &timer_index[i]);
    /*  略 */
}
```

```
"CONFIG_TIMER_MAX_NUM=3",
"CONFIG_TIMER_MAX_TIMERS_NUM_0=0",
"CONFIG_TIMER_MAX_TIMERS_NUM_1=6",
"CONFIG_TIMER_MAX_TIMERS_NUM_2=4",
```

WS63最多有三个硬件定时器:0、1和2
硬件定时器0，用户可以创建0个定时器(系统占用)
硬件定时器1，用户可以创建6个定时器
硬件定时器2，用户可以创建4个定时器

4.2 通用定时器

[4.2.2] WS63定时器系统的基本使用

- WS63定时器的控制流程：启动
  - errcode_t uapi_timer_start(timer_handle_t timer, uint32_t time_us, timer_callback_t callback, uintptr_t data)
    - timer_handle_t timer：已经创建好的定时器
    - uint32_t time_us：指定时器的定时时间，单位是微秒
    - timer_callback_t callback：定时器到达计时时间的回调函数
    - uintptr_t data：传递给回调函数的参数

data有什么必要性

```
for (uint32_t i = 0; i < TIMER_TIMERS_NUM; i++) {
    uapi_timer_create(TIMER_INDEX, &timer_index[i]);
    g_timers_info[i].start_time = uapi_tcxo_get_ms();
    uapi_timer_start(timer_index[i], g_timers_info[i], timer_timeout_callback, i);
    osal_msleep(TIMER_DELAY_INT);
}
```

g_timers_info[TIMER_TIMERS_NUM] = {1000, 2000, 3000, 4000};

4.2 通用定时器

[4.2.2] WS63定时器系统的基本使用

- WS63定时器的控制流程：回调
  - typedef void(*timer_callback_t)(uintptr_t data)
    - uintptr_t data：timer_start传递来给回调函数的参数

传递标识位，可区分硬件定时器下的(软件)定时器

```
for (uint32_t i = 0; i < TIMER_TIMERS_NUM; i++) {
    uapi_timer_create(TIMER_INDEX, &timer_index[i]);
    g_timers_info[i].start_time = uapi_tcxo_get_ms();
    uapi_timer_start(timer_index[i], g_timers_info[i], timer_timeout_callback, i);
    osal_msleep(TIMER_DELAY_INT);
}
```

```
static void timer_timeout_callback(uintptr_t data) {
    uint32_t timer_index = (uint32_t)data;
    //Do something here, e.g. start timer again -_-||
}
```

当前SDK的BUG，只能单次工作
TIMER_V150_MODE_ONE_SHOT

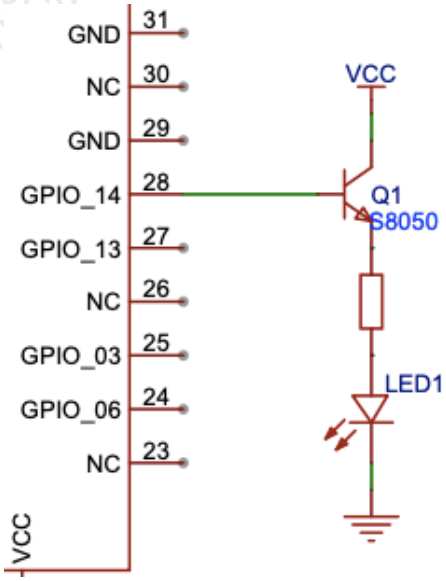
WS63 SoC的API是将定时器时间转换为了人类时间单位(us等)，也有MCU按照时钟数量计量的

4.2 通用定时器

[4.2.3] 定时器系统的使用范例

- 示例一：LED闪烁控制
  - LED1：循环点亮0.5秒熄灭0.5秒
  - 常规循环模式的关键源码

```
#define   LED_GPIO    GPIO_14
static int *blinky_task(const char *arg) {
    uapi_pin_set_mode(LED_GPIO, PIN_MODE_0);
    uapi_gpio_set_dir(LED_GPIO, GPIO_DIRECTION_ OUTPUT);
    while (1) {
        uapi_gpio_toggle(LED_GPIO);
        osal_msleep(500);
    }
    return 0;
}
```



第三章内容
GPIO模式、输出模式
输出电平翻转+延时

4.2 通用定时器

[4.2.3] 定时器系统的使用范例

- 示例一：LED闪烁控制
  - LED1：循环点亮0.5秒熄灭0.5秒
  - 定时器模式(单次触发模式)：定时器中断回调函数

```
#define   LED_GPIO     GPIO_14
timer_handle_t              timer_index;
static void timer_timeout_callback(uintptr_t data) {
    uint32_t timer_index = (uint32_t)data;
    uapi_gpio_toggle(LED_GPIO);                                          //GPIO输出翻转
    uapi_timer_start(timer_index, 500000, timer_timeout_callback, 0);    //下一次定时
}
```

目前API默认为单次模式
1 单次模式TIMER_V150_MODE_ONE_SHOT：定时器只工作一次，但不会自动删除，用户需要重新调用start设置定时器的定时时间和回调函数等
2 周期模式MER_V150_MODE_PERIODIC：该模式在API中尚未实现，定时器以周期的形式工作，到达定时时间后保持原有设置的定时时间，并触发中断

当前API版本指2025年2月1日的版本

4.2 通用定时器

- [4.2.3] 定时器系统的使用范例
  - 示例一：LED闪烁控制
    - LED1：循环点亮0.5秒熄灭0.5秒
    - 定时器模式(单次触发模式)：定时器初始化

```
static void *timer_task(const char *arg) {
    //……                                                          //其他初始化
    uapi_timer_init();                                            //初始化
    uapi_timer_adapter(1, TIMER_1_IRQN, 1);                       //适配
    uapi_timer_create(TIMER_INDEX, timer_index);                  //创建
    uapi_timer_start(timer_index, 500000, timer_timeout_callback, 0);   //启动一个定时器
    while(1) {
      osal_msleep(1000);                                          //模拟其他工作
    }
    return NULL;
}
```

4.2 通用定时器

- [4.2.4] 定时器系统的使用练习
  - 示例二：基于定时器的数字钟
    - 要求：使用osal_printk(“%02d:%02d:%02d”,hour, min, sec)输出时间信息
      - 每秒钟使用上述格式输出一次时间信息
    - 设计思路提示
      - 设计原则：高内聚、低耦合
      - 1秒时基来源：使用且仅可以使用一个定时器进行1000000us定时
      - 三个全局变量hour、min和sec，用于存储时钟的时、分和秒
    - 注意事项提示
      - 前面讲过的使用中断的注意事项，严禁在中断中使用osal_printk
      - 基础课后作业，请勿扩展年月日等功能
    - 提交要求：下次上课时提交纸质版程序源码(A4单面小四白底1.0倍行距)

4.3 看门狗定时器

[4.3.1] 看门狗定时器的作用

- 看门狗(Watchdog)
  - 信息系统(智能硬件)系统中的看门狗：一种简单的系统监控机制，基于软件和硬件的协同作业原理，确保系统能够在可预测的程序路径上执行，一旦系统出现流程异常，能够在规定的时间内强迫系统复位，重新运行
  - 看门狗定时器(Watchdog Timer, WDT)是一种硬件计时器，主要用于检测和恢复系统故障，其核心作用是防止系统长时间无响应或陷入死循环，从而提高系统的稳定性和可靠性
    - 硬件：首次设置看门狗定时器的定时时间，开始进行减计数
      - 正常进行减计数的过程是按照WDT时钟进行的
        - 减计数溢出之后，触发系统复位(WS63还可以选择先触发中断)
    - 软件：可重新设置看门狗定时器的定时时间(喂狗)，重新开始进行减计数

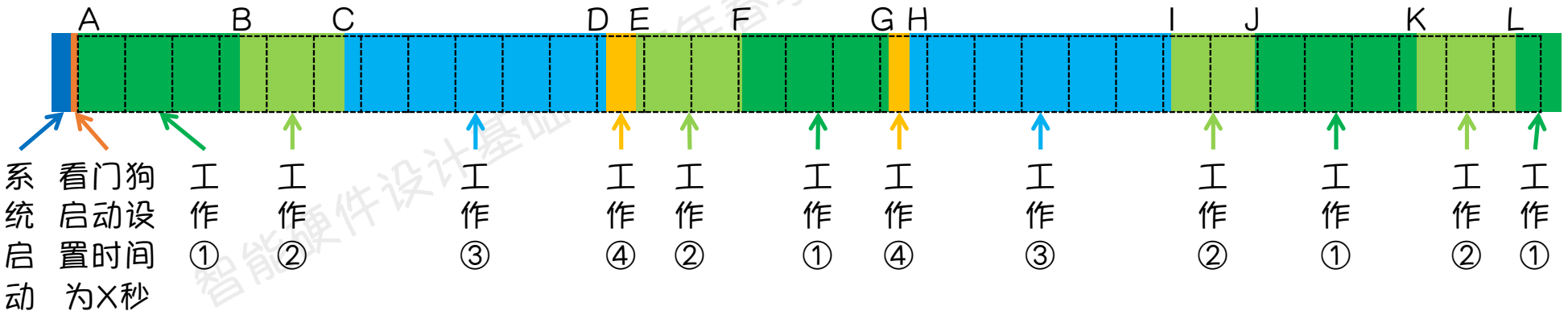
看门狗定时器是目前最简单且有效的软硬件协同的系统监控机制，使用极其广泛，硬件标配

4.3 看门狗定时器

- [4.3.2] 看门狗定时器的应用原则
  - 监控类软硬件的增设/使用原则
    - 硬件简洁：不增加电路，或增加的电路简单、体积小、功耗占比低等
      - 例如：早期的外置1.6秒WDT芯片MAX706AT，只需要4个元件即可构成
    - 机制简单：不增加复杂的软件功能，不占用大量系统资源
      - 例如：WS63 SoC执行uapi_watchdog_kick()，实际是写寄存器即可
      - 例如：MAX706AT，只需要输入一个上升沿或下降沿即可
  - MCU/SoC的看门狗应用原则
    - 固定定时时间的外置看门狗，如MAX706AT为1.6秒，设计硬件电路即可
    - 可设定定时时间的内置看门狗，评估可以接受的系统出错时间
      - 智能硬件系统性能较低，出错容忍度较高，通常在数秒甚至更高
      - 喂狗时间一般在看门狗定时时间的[1/2, 3/4]范围内 经验的时间原则

4.3 看门狗定时器

- [4.3.2] 看门狗定时器的应用原则
  - MCU/SoC的看门狗应用原则
    - 可设定定时时间的内置看门狗，评估可以接受的系统出错时间
      - 智能硬件系统性能较低，出错容忍度较高，通常在数秒甚至更高
      - 评估原则：比执行时间最长的不可拆分函数的单次最长执行时间略长

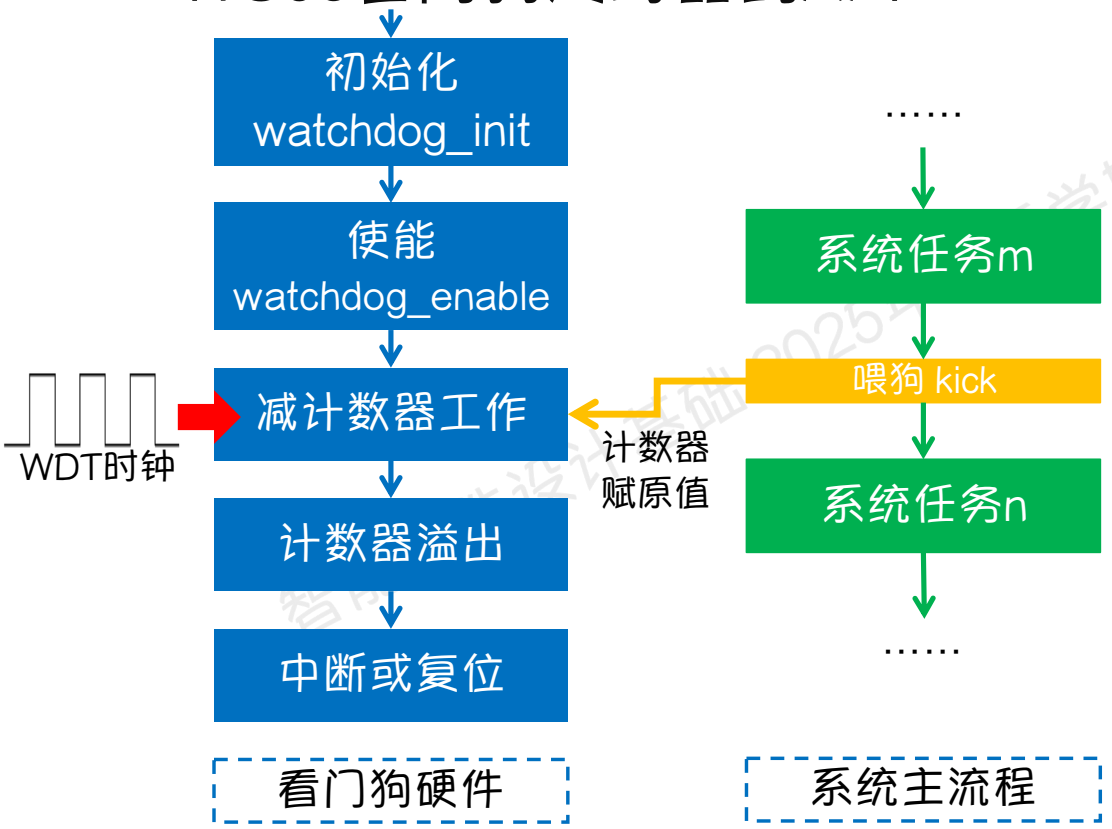


讨论：应该如何合理的设置看门狗，来保证系统的正常运行不受影响(影响最小)
Q1：看门狗定时器的定时时间最小应该设置为多久(整数，最小单位为秒)
Q2：在最小可设置时间的前提下，在上述A至L个点中，在哪些点设置较为合理

4.3 看门狗定时器

[4.3.3] WS63的看门狗定时器

WS63看门狗定时器的API



①：errcode_t uapi_watchdog_init(uint32_t timeout)

②：errcode_t uapi_watchdog_deinit(void)

③：errcode_t uapi_watchdog_enable(wdt_mode_t mode)

④：errcode_t uapi_watchdog_disable(void)

⑤：errcode_t uapi_watchdog_kick(void)

⑥：errcode_t uapi_watchdog_set_time(uint32_t timeout)

⑦：errcode_t uapi_watchdog_get_left_time(uint32_t *timeout)

⑧：errcode_t uapi_register_watchdog_callback(watchdog_callback_t callback)

4.3 看门狗定时器

[4.3.3] WS63的看门狗定时器

- WS63看门狗定时器的API
  - errcode_t uapi_watchdog_init(uint32_t timeout)
  - errcode_t uapi_watchdog_deinit(void)
  - errcode_t uapi_watchdog_disable(void)
  - errcode_t uapi_watchdog_kick(void)
  - errcode_t uapi_watchdog_enable(wdt_mode_t mode)
    - wdt_mode_t mode：工作模式，支持直接复位和优先中断两种模式
      - 直接复位模式：WDT_MODE_RESET，溢出则直接引起系统复位
      - 优先中断模式：WDT_MODE_INTERRUPT，溢出后先触发中断，在中断回调函数中，如果不执行喂狗操作才会引起系统复位(WS63的特有模式)
      - 特别注意：除了WS63的优先中断模式之外，严禁在任何中断中喂狗

初始化(秒)
去初始化
禁用
喂狗

位置原则

4.3 看门狗定时器

- [4.3.3] WS63的看门狗定时器
  - WS63看门狗定时器的API
    - errcode_t uapi_watchdog_set_time(uint32_t timeout)
      - 修改看门狗定时器的定时时间(看门狗初始化之后执行，重新enable后生效)
    - errcode_t uapi_watchdog_get_left_time(uint32_t *timeout)
      - 获取看门狗计数器的剩余值(单位为秒)
    - errcode_t uapi_register_watchdog_callback(watchdog_callback_t callback)
      - 注册看门狗定时器的中断回调函数
      - 必须在该回调函数中进行一次喂狗操作，否则系统将复位
      - 禁止在其他中断回调函数中进行喂狗操作
        - 系统主程序异常时，定时器等基于硬件的中断可能仍可以正常执行的

4.4 嘀嗒定时器

[4.4.1] 嘀嗒定时器的作用

- Systick：一种为系统提供连续的时间间隔计数的硬件定时器
  - 用于实现延时函数、计算时间间隔、执行任务调度等
  - 与常规定时器的区别
    - Systick结构简单，一般只进行初始化和读操作
    - Systick随MCU/SoC不同，可能有增计数和减计数两种类型
    - Systick的计数器长度更长，一般为24位、32位、48位或更长
    - Systick通常具备中断能力，但一般被系统占用，用于任务切换
    - Systick随系统启动之后一直运行，一般不会停止或复位
    - 用户在使用Systick时，应对Systick的只进行读取访问

带有OS的MCU/SoC的Systick一般会被系统默认占用，用户应避免进行控制和写入操作

4.4 嘀嗒定时器

[4.4.2] WS63的嘀嗒定时器

- WS63嘀嗒定时器的API
  - void uapi_systick_init(void)
  - void uapi_systick_deinit(void)
  - errcode_t uapi_systick_count_clear(void)
  - uint64_t uapi_systick_get_s(void)
  - uint64_t uapi_systick_get_ms(void)
  - uint64_t uapi_systick_get_us(void)
  - errcode_t uapi_systick_delay_s(uint32_t s_delay)
  - errcode_t uapi_systick_delay_ms(uint32_t m_delay)
  - errcode_t uapi_systick_delay_us(uint32_t u_delay)

WS63嘀嗒定时器的特征
工作频率：32KHz(32768Hz)
定时器长度：48位

获取Systick计时

使用Systick延时

利用Systick可获知Systick初始化后的系统累计运行时间(48位计数器很长)

4.5 RTC时钟

- [4.5.1] RTC时钟的特点
  - 时间间隔定时器
    - 通用定时器Timer：用户设定定时时间，到达时间触发中断
    - 看门狗定时器WDT：用户设定时间，软硬件协同监控系统状态
    - 嘀嗒定时器Systick：系统设定，用户只进行读取操作(建议)
  - 时间戳计时器
    - RTC(Real Time Clock)实时时钟，用于提供持续的、准确的时间跟踪功能，以人类易于阅读的年月日时分秒的形式进行记录、计时和输出
      - RTC在绝大多数MCU/SoC中，都需要外部备用电池功能(纽扣电池)
      - RTC的耗电量极低，工作电流一般在uA甚至更低的级别
      - RTC自带年月日时分秒星期闰年等计算功能，无需用户计算(ISO8601)

RTC实现保持和计时功能的基础是外部供电，都是外部供电会导致全部信息丢失

4.5 RTC时钟

[4.5.2] RTC时钟的应用

- RTC常用模式
  - Unix时间：自1970年1月1日0时0分0秒起的秒数
    - 与RTC硬件交互使用的是秒数，一次读写，后续转换
    - 如1738080000表示2025年01月29日0时0分0秒

    > STM32、GD32 WS63是该模式（集成RTC常见）

  - 年月日时分秒格式
    - 与RTC硬件交互使用的是独立数据
    - 年、月、日、时、分、秒等信息保存在独立的寄存器中，需要多次读取

    > 外部独立芯片

地址	功能	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	读/写
0x00	SEC	O	BCD 码，秒十位，0-5			BCD 码，秒个位，0-9				R/W
0x01	MIN	O	BCD 码，分十位，0-5			BCD 码，分个位，0-9				R/W
0x02	HOUR	O	O	BCD 码，时十位，0-2		BCD 码，时个位，0-9				R/W
0x03	WEEK	O	6	5	4	3	2	1	0	R/W
0x04	DAY	O	O	BCD 码，日十位，0-3		BCD 码，日个位，0-9				R/W

Unix时间与年月日时分秒之间的转换可使用mktime和localtime完成

4.6 本章作业

- [4.6.0] 作业与思考

1. 信息设备实现计时的基础有哪三点
2. WS63 SoC计时设备的基本结构是什么样的
3. 预分频器PSC在定时器系统中的主要作用是什么
4. 自动重装载器ARR的主要功能是什么
5. 在实际编程中，WS63是区分同一硬件定时器下，多个定时器所触发中断有哪两种方法
6. 简述看门狗定时器在智能硬件系统中的作用
7. 总结看门狗定时器的定时时间的设定原则
8. 简述看门狗定时器喂狗操作的原则

4.6 本章作业

- [4.6.0] 作业与思考

9. WS63看门狗定时器直接复位模式下的设置流程有哪几个步骤
10. 从用户应用程序层面，Systick定时器的主要作用是什么
11. 常用的时间间隔定时器有哪些
12. 常用的时间戳计时器有哪些
13. RTC的外部电池的作用是什么
14. 完成基于定时器的数字钟的代码(1秒的定时来源基于WS63)
15. 思考继续：SoC如何通过外设与外界进行信息交互(数据传输)

GPIO	UART	I2C	LSADC	PWM
Wi-Fi、RADAR、SLE、BLE		Timer	WDT	TCXO