智能车嵌入式系统设计

谈谈我在智能车中使用的嵌入式的开发经验吧。

系统分层设计与模块化

单文件单函数处理一些比如简单计算，简单的控制，当代码体量上升到百行以上，就要开始切分模块分割函数了。通过将部分功能模块化抽离出函数，原本千行的代码就可以被切割为n个小体量的函数和文件。随着功能继续完善，将不同的功能拿出来封装起来称为一个独立的模块，这样既方便对项目进行迭代又方便对项目进行维护，项目的结构也会更加清晰。

随着系统功能进一步复杂，有各种的输入和输出的设备，为了控制系统的复杂度，会进一步进行分层，不停的地向上封装，代码高度模块化。还记得软件开发的准则吗？**高内聚，低耦合**。我们的开发最后都是奔着这六个字去的，符合这个准的的代码将会有非常高的可移植性和可维护性。我们还要知道分层和模块化也不是完美的，它会带来**效率地降低**，比如增加额外的函数调用和时间的损耗，但是带来的确实可读性和可维护性的提高。不过千万不要**过度**地**无意义**地封装代码，这样没有任何好处。

智能车系统的模块分层大致可以分成三层：

- 控制算法和图像处理（速度与方向的控制、图像处理或电感信号处理）
- 嵌入式平台（信号采集、外设驱动、任务调度）
- 硬件与机械（电机、舵机、编码器、摄像头、电源等）

其实大部分的嵌入式项目也是这样的结构。

这篇文章我们将浅谈一下前面两层。

任务调度设计

模块是对功能代码的封装，分层是在整个系统层面进行封装，都是为解决复杂的项目而做出的设计。我们再回到执行这些任务的最底层的工具：单片机，来看看这些任务（比如ADC采样、PID控制、I2C通讯、输出PWM等）都是怎么在单片机中运行的，我们要怎么去设计这些任务的执行方式，要怎么去调度它们？

大循环调度

这是最简单的调度方式，就像下面的代码一样：

```
void main(void)
{
    System_Init();

    while(1){
        Task0_Run();
        Task1_Run();
        Task2_Run();
    }
}

void Task0_Run()
{
    ...
}
...
```

```
...
```

这种调度方式简单直接，比较适合简单的系统，但是存在的问题也是巨大的：

- 每个任务的调度周期和时间是不固定的，无法保证确定的周期性执行任务、任务的实时性无法得到保证。
- 随着任务的增加，系统将会变得越来越慢。
- 如果遇到执行时间较长的任务就会拖慢整个系统

定时任务调度

为了克服使用大循环进行调度的缺点，提出了定时任务调度，需要单片机提供一个定时器来实现一个简单的任务调度器。利用定时器将CPU切割为一个等周期的时间片调度单元，然后利用标志位控制在每个时间片只调用一个任务。或者利用时间片将定时器时间划分，执行不同的任务。

方式一

```c
uint32_t flag_5ms, flag_10ms;

void main(void)
{
    System_Init();
    En_Interrupt();

    while(1) {
    }
}

// 1ms定时器中断服务函数
void TIM1_IRQHandler(void)
{
    flag_5ms++;
    flag_10ms++;
    if(flag_5ms >= 5) {
        flag_5ms = 0;
        Task0_Run();
        ...
    }
    if(flag_10ms >= 10) {
        flag_10ms = 0;
        Task1_Run();
        ...
    }
}

void Task0_Run()
{
    ...
}

void Task1_Run()
{
    ...
}
...
...
```

方式二

```
#define TASK_MAX_LENGTH 10
typedef struct {
    uint8_t Flag[TASK_MAX_LENGTH];
    uint32_t Timer;
    uint32_t Number;
} USERTASK;
USERTASK UserTask0 = {0,0,0,0,0,0,0,0,0,0,TASK_MAX_LENGTH};  // 任务初始化

// 任务调度函数
void TaskScheduler(USERTASK* v)
{
    v->Flag[v->Timer++] = 1;
    if(v->Timer >= v->Number) {
        v->Timer = 0;
    }
}

// 主函数
int main(void)
{
    System_Init();
    En_Interrupt();

    while(1) {
        Task0_Run();
        Task1_Run();
        Task2_Run();
    }
}

// 1ms定时器中断服务函数
void TIM1_IRQHandler(void)
{
    TaskScheduler(&UserTask0);
}

//单个任务示例函数
void Task0_Run(void)
{
    if(UserTask0.Flag[0]) {
        ...

        UserTask0.Flag[0] = 0;
    }
}
...
...
```

分析

与大循环调度方式对比，这种方式能够实现周期性的任务调度，同时随着任务的增加，依然能够保证调度的周期性，这种调度能够应对大多数的控制系统。

使用时需要注意下面三点：

- 单个任务的最长时间长度务必保证不超过单个时间片，否则会导致周期性延迟（方式一和方式二都是）

- 使用方式二对于严格实时的控制周期任务，定时调度器不能够保证，但是方式一可以保证。对于一些实时性要求高的，并且周期快的任务，使用方式一将任务放到中断中去执行是可以接受的。但是一般来说不要把任务放到中断中去执行，常见的方式是中断中放标志位，然后任务在中断外执行，例如方式二的实现方式。
- 对于长周期任务（比如通讯等待等），定时任务调度器要么把任务切割为小任务，要么安排几个连续的空闲周期来执行

针对第三点的情况，我们可以将整个任务拆分或者将这个任务放到任务列表中的最后面，给这个任务多分配几个周期。但是如果有多尔衮长任务仍然会拖慢整个系统。于是就出现了基于优先级的任务调度方式，高优先级的任务可以中断低优先级的任务，在保证长周期任务调度的同时，短周期任务的调度依然能够保证，这就是**RTOS**。

RTOS

RTOS是**实时操作系统**(Real Time OS)的简称，常用的RTOS有FreeRTOS、ucOS、RT-thread等。智能车竞赛也可以使用RTOS，官方推荐使用的RTOS是RT-thread，并且设立的相应的奖项，对RTOS感兴趣的同学可以去了解一些，这里对RTOS内核的代码不做详细介绍。

对于项目是否使用RTOS：如果任务多并且要求实时性，需要使用RTOS。上RTOS虽然带来了实时性和任务调度的便利性，但同时带来**软件逻辑的复杂性**，这是一把双刃剑。对**简单的系统**不建议使用RTOS，大多数简单系统，定时任务调度器配合中断就可以做得很好。

智能车任务调度

智能车**传统项目**的调度平台总体上只有两个任务：**控制**和**信号处理**。这里介绍不使用RTOS的实现方法，后面有时间会写基于RTOS的任务设计。

代码示例如下：

```
void main(void)
{
   sys_Init();  // 外设初始化
   En_Interrupts();

   while(1){
       if(mt9v03x_finish_flag_dvp == 1) {  // 图像处理
           mt9v03x_finish_flag_dvp = 0;
           image_process();
       }
   }
}

// 10ms定时器中断
void TIM1_IRQHandler(void)
{
    SpeedControlTask();  // 控制
}
```

我将图像处理放到了主循环中，这里的意思是只要完成了图像的传输就开始进行图像处理，mt9v03x_finish_flag_dvp是在摄像头场中断中置1的，也就是采集完一帧后这个标志位就会被置1，并且由软件置0。为什么要将图像处理放到主循环？处理一帧图像使用的时间不同，并且没有强实时性的要求，所以就放到了主循环中。

嵌入式驱动层设计

驱动层大部分采用了逐飞科技的bsp库，逐飞的库包含了硬件抽象层（比如tim、uart的配置）到外设的驱动（比如ips114、icm20602），底层调用的还是芯片官方提供的标准库，使用逐飞的库就不需要我们去再写一些基本的驱动，节省了很多时间。我们要使用但是逐飞科技没有写出的功能需要使用官方提供的标准库进行配置。工程设计上的总体思路就是，**.h文件负责接口，.c文件负责功能实现。**

我们还要在逐飞提供的库之上再进行一层封装。以电机驱动为例，逐飞科技提供了pwm的初始化函数，但是我们还要根据硬件设计封装为速度设置等功能，供更上层的代码调用。

比如下面的代码：

```
/**
 * @brief Motorinit 初始化电机和舵机
 *
 * @param None
 */
void Motorinit(void)
{
    // PWM_DUTY_MAX = 10000
    pwm_init(TIM2_PWM_CH1_A15, 17000, 0);  // 17kHz  左轮
    pwm_init(TIM2_PWM_CH2_B3, 17000, 0);  // 17kHz  右轮
    pwm_init(TIM3_PWM_CH3_B0, 50, 0);  // 50Hz  舵机

    gpio_init(MDIR1, GPO, 1, GPIO_PIN_CONFIG);
    gpio_init(MDIR2, GPO, 1, GPIO_PIN_CONFIG);
}

/**
 * @brief SetDuty 设置占空比
 *
 * @param num
 *              0  左轮
 *              1  右轮
 *              2  舵机
 *
 *              Steering gear limiting: 6.061% - 6.566% - 7.576%
 *              Steering duty Range: 606 - 656 - 757
 *
 * @param dir
 *              0   正转
 *              1   反转
 */
void SetDuty(uint8_t num, uint16_t duty, uint8_t dir)
{
    switch(num) {
        case 0: {
            if(dir == 0) {
                gpio_set_level(MDIR1, 1);
                pwm_set_duty(TIM2_PWM_CH1_A15, duty);  // 左轮
            }
            else if(dir == 1) {
                gpio_set_level(MDIR1, 0);
                pwm_set_duty(TIM2_PWM_CH1_A15, duty);  // 左轮
            }
            break;
        }
```

```
        case 1: {
            if(dir == 1) {
                gpio_set_level(MDIR2, 1);
                pwm_set_duty(TIM2_PWM_CH2_B3, duty); // 右轮
            }
            else if(dir == 0) {
                gpio_set_level(MDIR2, 0);
                pwm_set_duty(TIM2_PWM_CH2_B3, duty); // 右轮
            }
            break;
        }
        case 2: {  // 舵机
            /* 限幅 */
            if(duty >= DUTY_MAX) {
                duty = DUTY_MAX;
            }
            else if(duty <= DUTY_MIN) {
                duty = DUTY_MIN;
            }
            pwm_set_duty(TIM3_PWM_CH3_B0, duty);
            break;
        }
        default: break;
    }
}
```