# 嵌入式系统实验报告



|  |  |
| --- | --- |
| 实验名称： | TIM PWM 与蜂鸣器音乐播放器 |
| 姓 名： | 江姝潼 |
| 学 号： | 2019211653 |
| 学 院(系)： | 计算机学院 |
| 专 业： | 网络工程 |
| 指导教师： | 刘健培、戴志涛 |

2021年 12 月 15 日

# 实验目的

* 学习使用 STM32 定时器 TIM 的基本操作方式。
* 学习使用 STM32 定时器 TIM 输出 PWM 波的方式。
* 学习操作系统环境下编写设备驱动的方式。
* 学习基于 RTOS 的多任务应用程序设计与编程方式。
* 学习在一个综合应用中同时驱动多个外设的方法。

# 实验环境

* FS-STM32F407开发平台
* ST-Link 仿真器
* RealView MDK5.23集成开发软件
* PC机Window7/8/10 (32/64bit)
* 串口调试工具

# 实验要求

* 使用 PWM 波驱动蜂鸣器发声，实验一个简单的蜂鸣器音乐播放器。
* 播放器可以通过按键进行控制，支持：显示播放列表、暂停/播放、上一曲、下一曲、音量调节等功能。

K3——短按，暂停/播放键。

K4——短按作为下一曲的按键，长按为音量加。 

K5——短按作为上一曲的按键，长按为音量减。 

* 在参考代码的基础上，基于操作系统的I/O子系统框架，编写通用的PWM驱动，实现音乐的播放。

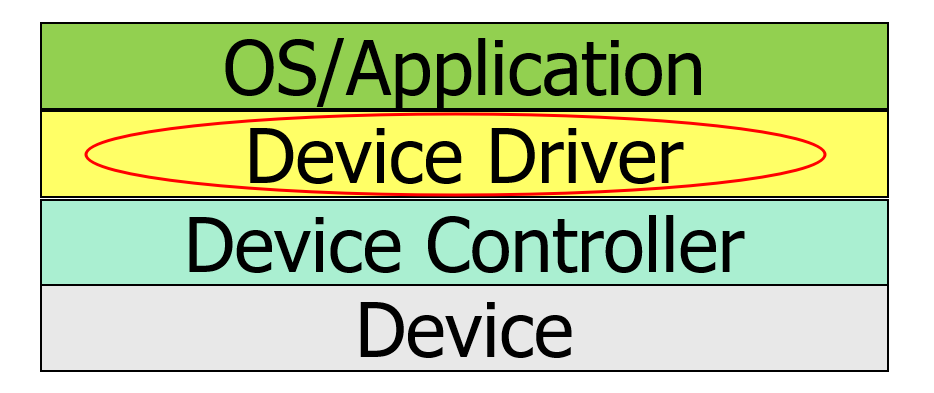
# 实验原理

1. **2种类型的I/O驱动**

设备驱动是指驱使硬件设备行动的特定软件，是与硬件设备直接打交道的底层软件。主要功能是负责应用软件和硬件设备之间的沟通，将应用程序的标准调用映射为设备特定的命令序列(操作寄存器)。

向下，直接管理底层硬件设备，按照设备的具体工作方式，读写设备控制器（device controller）的寄存器，完成设备的轮询、中断处理或 DMA 通信（Device-specific）。向上，为上层程序提供操作设备的编程接口，隐藏硬件设备控制的具体细节（OS-specific）。核心操作有：发送数据给设备、从设备接收数据、控制设备。

根据操作系统是否定义了驱动架构，可把驱动分为 2 类：不基于 OS 的设备驱动与基于 OS 的设备驱动。



不基于 OS 的设备驱动，使用非标准接口。接口可根据硬件设备的特点自行定义或和应用程序协商定义应用程序直接调用该接口来使用设备。应用与驱动紧耦合。

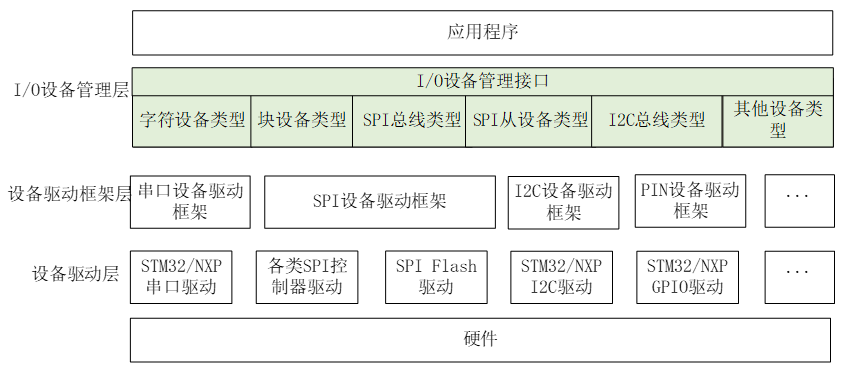
基于 OS 的设备驱动，则使用标准接口。OS 定义了驱动的架构和独立于具体设备的抽象接口，驱动必须实现这些接口才能融入 OS。应用程序通过 OS 的API 间接调用驱动接口。应用、OS 与驱动相对独立。例如：RT-Thread、Linux等操作系统下的设备驱动。本实验可归于这一类。

**2. 操作系统 I/O 子系统框架下的设备驱动**

本实验的主要目的之一即是学习如何在操作系统环境下编写标准接口的设备驱动，虽然不同操作系统实现的 I/O 设备模型框架细节不同，但主要的层次以及设计方式是很类似的。

以 RT-Thread 为例，它提供了一套简单的 I/O 设备模型框架，如下图所示，它位于硬件和应用程序之间，共分成三层，从上到下分别是 I/O 设备管理层、

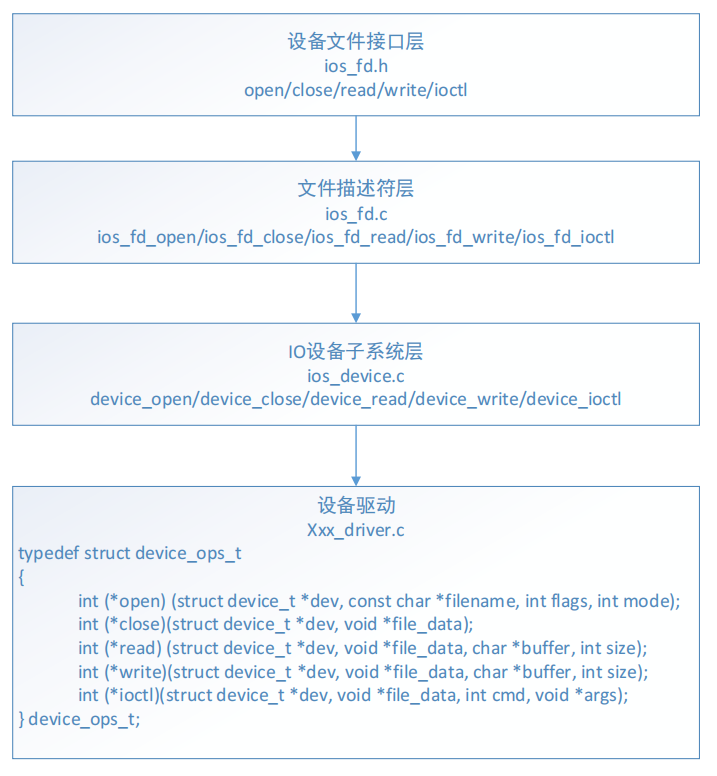
设备驱动框架层、设备驱动层。



应用程序通过 I/O 设备管理接口获得正确的设备驱动，然后通过这个设备驱动与底层 I/O 硬件设备进行数据（或控制）交互。I/O 设备管理层实现了对设备驱动程序的封装。应用程序通过 I/O 设备层提供的标准接口访问底层设备，设备驱动程序的升级、更替不会对上层应用产生影响。这种方式使得设备的硬件操作相关的代码能够独立于应用程序而存在，双方只需关注各自的功能实现，从而降低了代码的耦合性、复杂性，提高了系统的可靠性。

设备驱动框架层是对同类硬件设备驱动的抽象，将不同厂家的同类硬件设备驱动中相同的部分抽取出来，将不同部分留出接口，由驱动程序实现。设备驱动层是一组驱使硬件设备工作的程序，实现访问硬件设备的功能。它负责创建和注册 I/O 设备，对于操作逻辑简单的设备。

在我们的教学操作系统内核里，实现了一个简单的字符类 I/O 设备的驱动模型。本实验在此模型框架下编写驱动。该模型分为 4 层。



1) 最顶层是设备文件接口层，提供类 unix 系统的 open/close/read/write/ioctl文件操作函数（posix 标准）。但不支持挂载文件系统等操作。

2) 往下是文件描述符层。Posix 文件接口使用文件描述符的概念表示一个打开的文件。顶层打开一个文件时，是这样的使用方式：int fd =open(filename\_string, flag, mode)，使用文件字符串名字调用 open，返回一个 int 类型的文件描述符 fd，后续的 read/write/ioctl/close 操作都是基于这个 fd 操作。这个 fd 就是本层的文件描述符表 fd\_table[] 的索引号，表中每一项都是 fd\_t 类型的文件描述符结构体，存储了文件的设备、名称、私有数据等信息。

3) 再往下是实际完成工作的 IO 设备子系统层。该层最重要的功能是绑定设备与驱动。IO 设备子系统层管理者一张设备链表，device\_list，其中每一项的类型是device\_t，对应一个特定的设备驱动。一个设备驱动对应一类设备。一个设备驱动驱动管理所有同类型的的多个设备，这些设备可能只有参数不同，例如端口号、IO地址、中断号等。一般在系统初始化时注册设备驱动，在应用程序使用时，打开具体的设备，调用设备驱动进行操作。

4) 设备驱动根据设备模型定义，创建出具备硬件访问能力的设备实例，将该设备通过 device\_register() 接口注册到 I/O 设备子系统中。应用程序通过 open-> ios\_fd\_open ->device\_open->device\_find() 接口查找到设备（使用文件名），然后使用 I/O 设备子系统接口来访问硬件。

路径对应关系：

在高层级，每个设备驱动程序均与某些路径前缀相关联。当一个 C 库函数需要打开文件时，IO 设备子系统将搜索与该文件所在文件路径相关联的设备驱动程序，并将调用传递给该驱动程序。针对该文件的读取、写入等其他操作的调用也将传递给这个驱动程序。

操作系统的文件可以看做是一个字符串到数据以及操作这些数据的程序的一个映射。这个字符串就是文件路径，而数据即是设备，操作数据的程序即是驱动。一般操作系统都已树状方式组织路径，但在如何将路径映射到设备与驱动，则有细微差异，例如 unix、linux 系列操作系统全部由根目录/开始寻址，而Windows 则是以盘符如 c:开始。

我们使用的文件路径模型非常简单：使用线性表实现路径到驱动的映射。方法是已注册的设备驱动程序均有一个路径前缀与之关联。在打开文件时使用具有最长匹配路径前缀匹配方式寻址驱动。

**3. 音乐数据的PWM编码与解码**

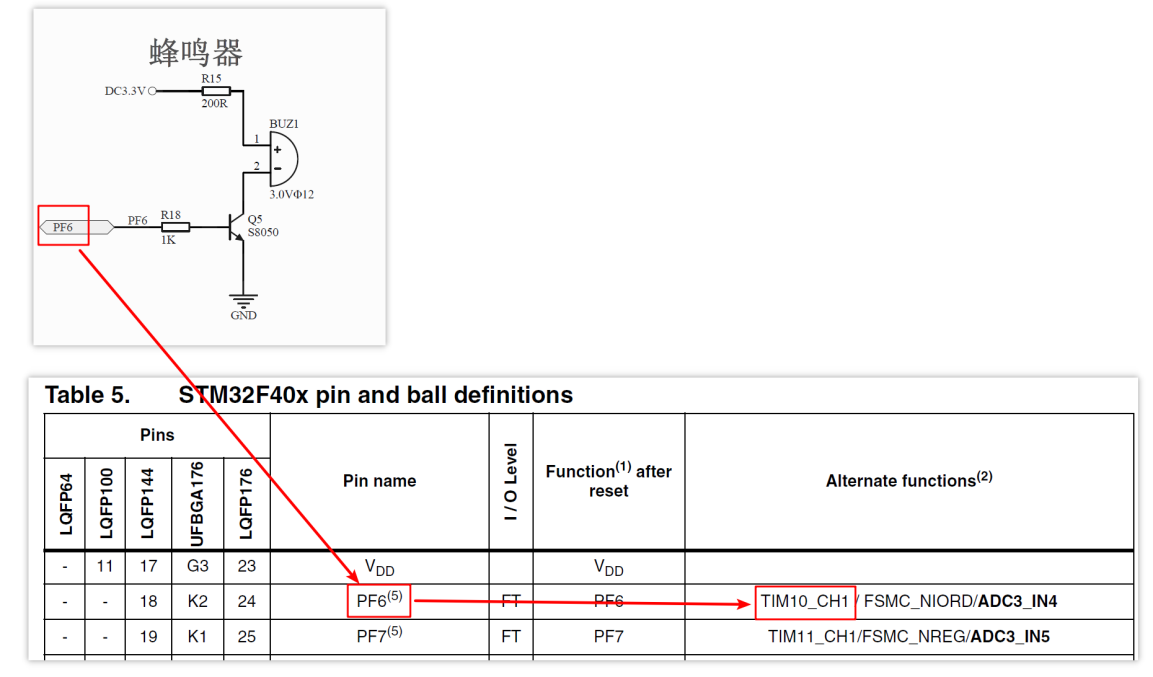
声音的三要素是：响度、音色、音调。利用这三个要素就可以演奏不同的音乐。一般说来，蜂鸣器演奏音乐只能是单音频率，因此蜂鸣器奏乐只需弄清楚两个概念即可，也就是“音调”和“节拍”。音调表示一个音符唱多高的频率，节拍表示一个音符唱多长的时间。十二平均律就规定了每一个音符的标准频率。

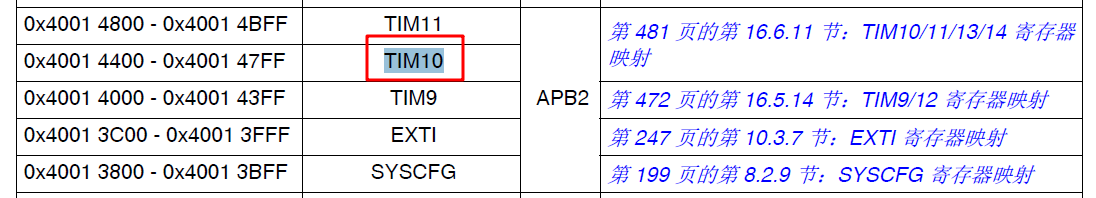
十二平均律，是一种音乐定律方法，将一个纯八度平均分成十二等份，每等分称为半音，是最主要的调音法。十二平均律中各音的频率： C: 262 Hz、#C: 277 Hz、D: 294 Hz、#D: 311 Hz、E: 330 Hz、F: 349 Hz、#F: 370 Hz、G: 392 Hz、#G:

415 Hz、A: 440 Hz、#A: 466 Hz、B: 494 Hz。

**4.使用定时器的PWM输出驱动蜂鸣器**

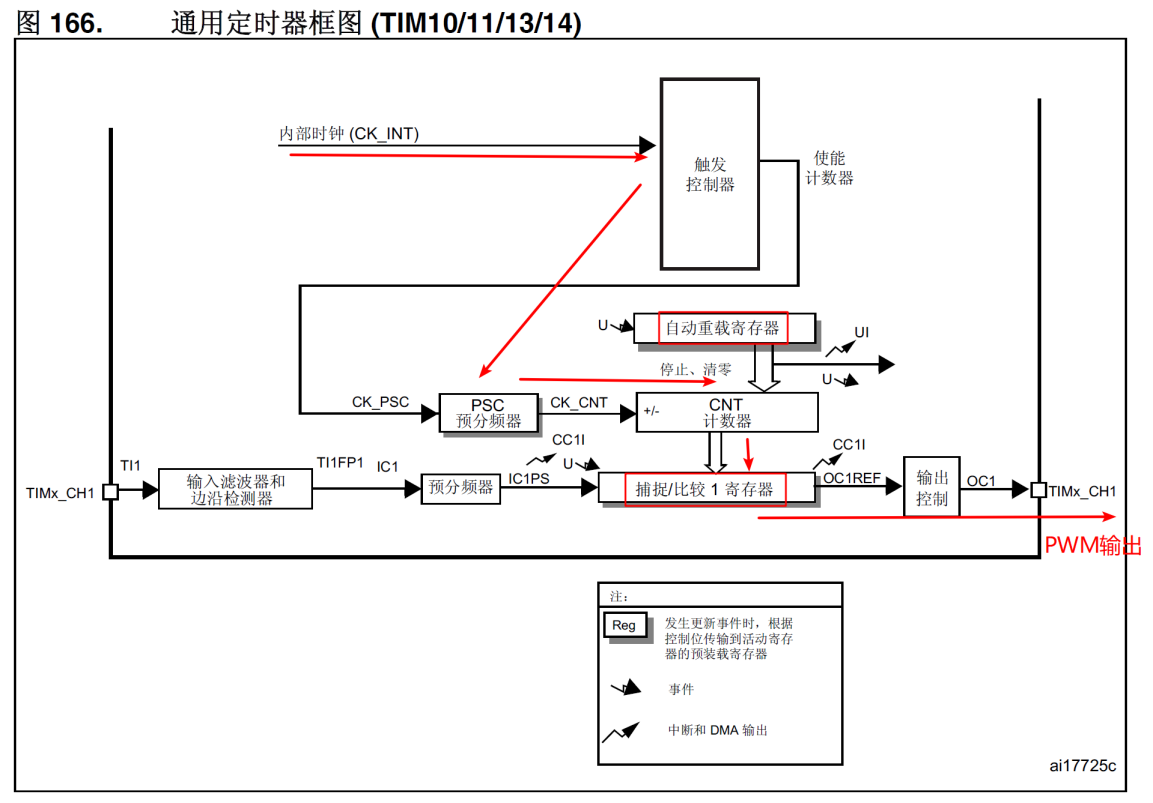
只有无源蜂鸣器才能够用来播放音乐。因为无源蜂鸣器振动的频率是可调的，而有源蜂鸣器的振动频率是固定的。只有频率（也就是音高）可调，才能够播放简单的音乐。由于无源蜂鸣器需要有震荡信号才能发出声音，所以需要使用PWM 设备来控制蜂鸣器的播放。实验中使用的TIM10的通道1。



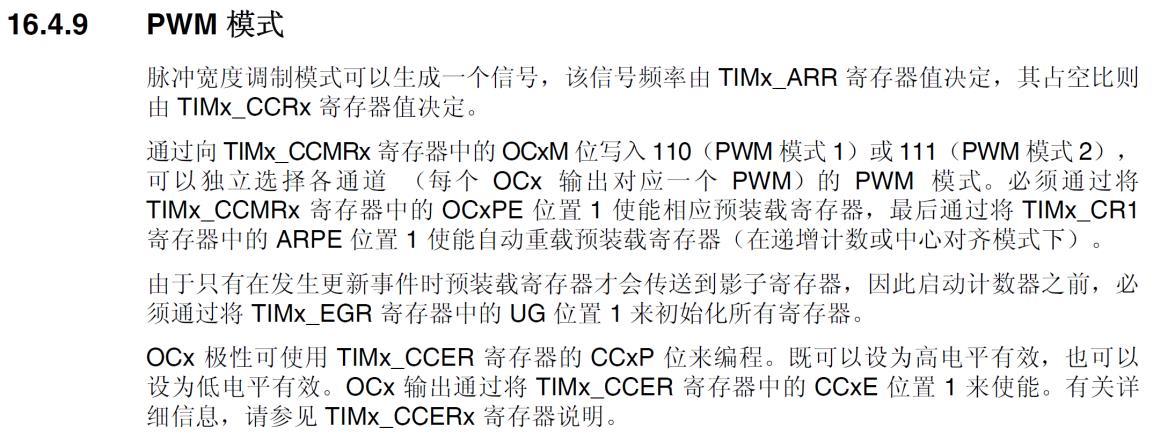


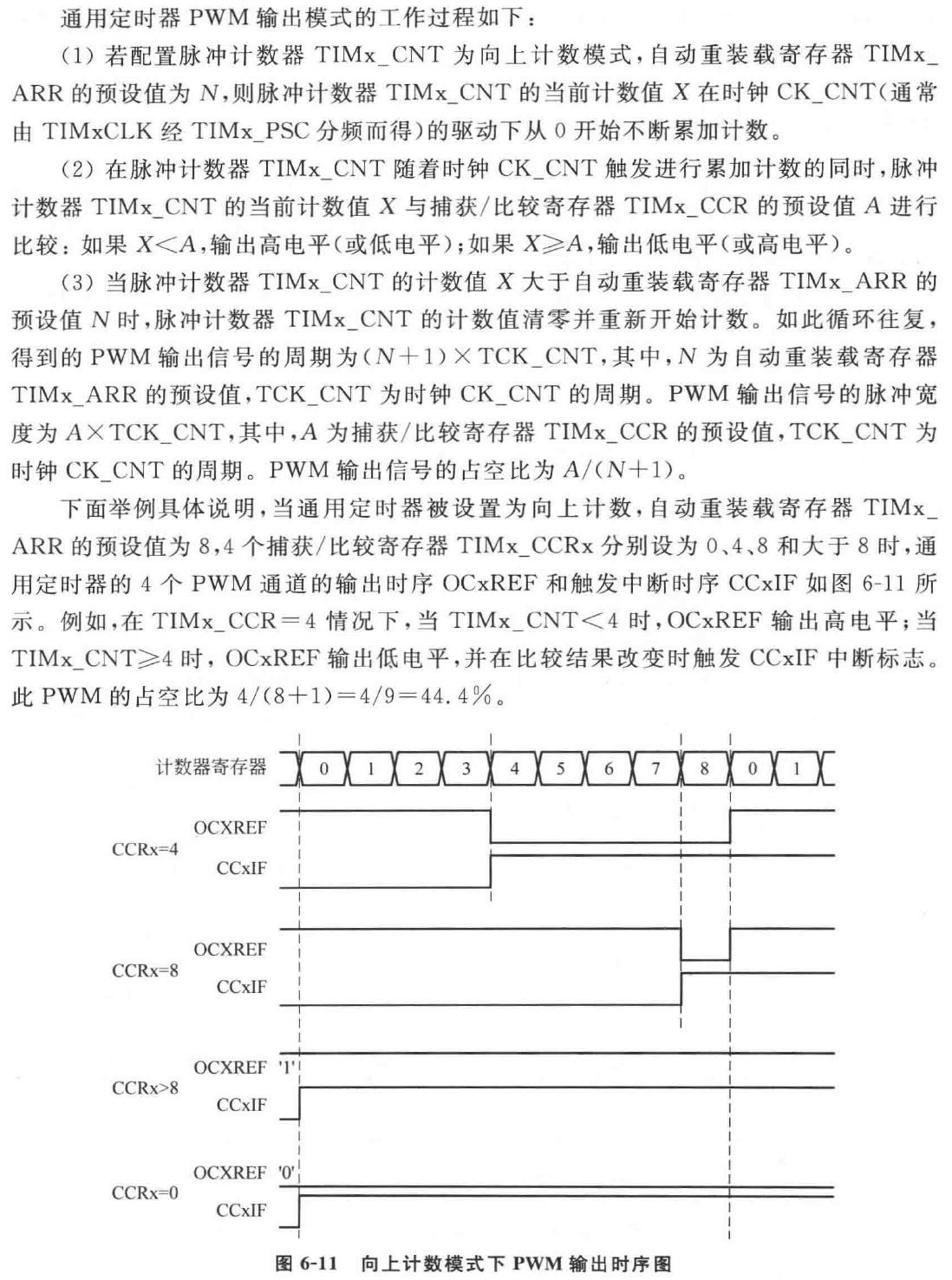
**5. TIM的PWM输出模式**

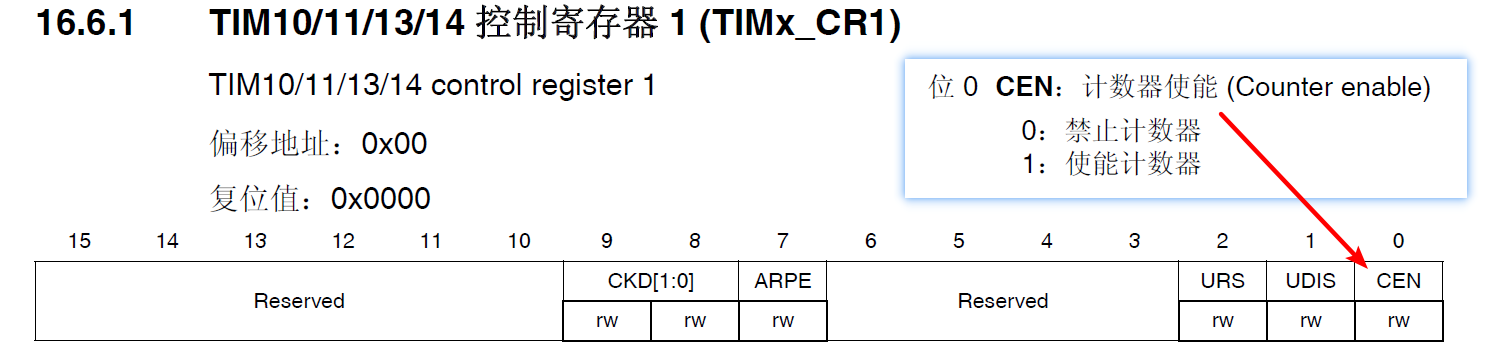
PWM 输出就是对外输出脉宽（即占空比）可调的方波信号。通用定时器 TIM10的基本功能之一就是产生PWM输出。



PWM 方波信号频率由自动重装寄存器 ARR 的值决定，占空比由比较寄存器 CCR 的值决定。







计数值的计算公式为：





# 实验步骤

本次实验需要完成 pwm\_driver.c 中 对pwm 设备驱动的4个接口函数的编写，

需要实现的 4 个接口函数是：

int stm32\_pwm\_close(struct device\_t \*dev, void\* file\_data)

int stm32\_pwm\_open(struct device\_t \*dev, const char \*filename, int flags, int mode)

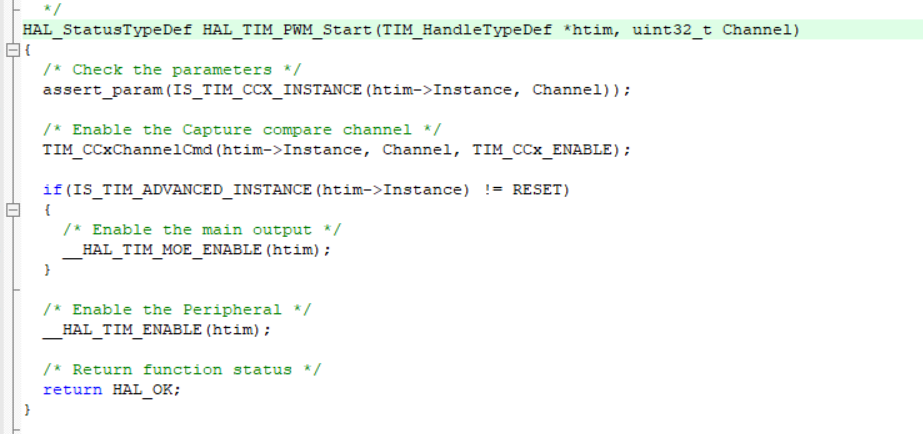
int stm32\_pwm\_write(struct device\_t \*dev, void\* file\_data, char \*buffer, int size)

int stm32\_pwm\_ioctl(struct device\_t \*dev, void\* file\_data, int cmd, void \*args)

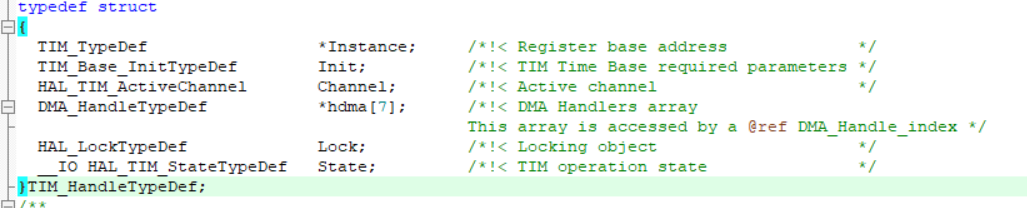
1.编写stm32\_pwm\_open函数：

在此函数中，需要完成从filename提取定时器号和通道号，编码到pwmcode中，并return回上层。

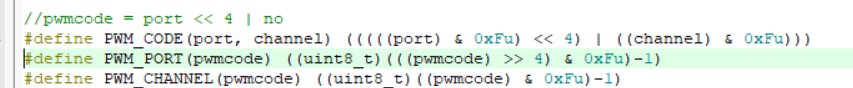
在这一层中，需要调用下一层HAL\_TIM\_PWM\_Start函数来实现启PWM设备驱动。查阅它的函数定义和参数，可以知道，它需要传入一个TIM\_HandleTypeDef型的变量，以及它的通道号。



查看TIM\_HandleTypeDef型的变量定义，从定义中可以看到该结构体比较重要的变量是channel通道号和instance寄存器基址这两个变量。寄存器基址已经提前给定在timmap，把对应的值传入进去，保存在一个变量中就可以调用该函数了。



那么问题的关键就在于，如何获得的通道号和定时器号呢？我们观察它的编码过程，可以看到，pwmcode相当于把通道号前移四位放在前四位，并把channel号即通道号放到了后四位。据此就可以给出对应的解码操作，这里已经有了宏定义，直接调用PWM\_PORT和PWM\_CHANNEL就可以获得它的端口号和通道号了。

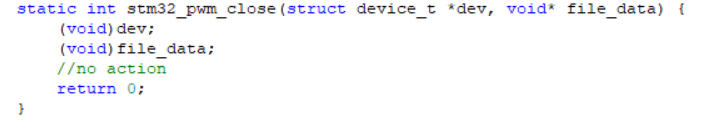


接下来，我们还需要去初始化ARR和CCR1寄存器的初始值，这里参考了smoke文件，选用了如下的值。最后在函数末尾返回pwmcode，就完成了对pwm的初始化。



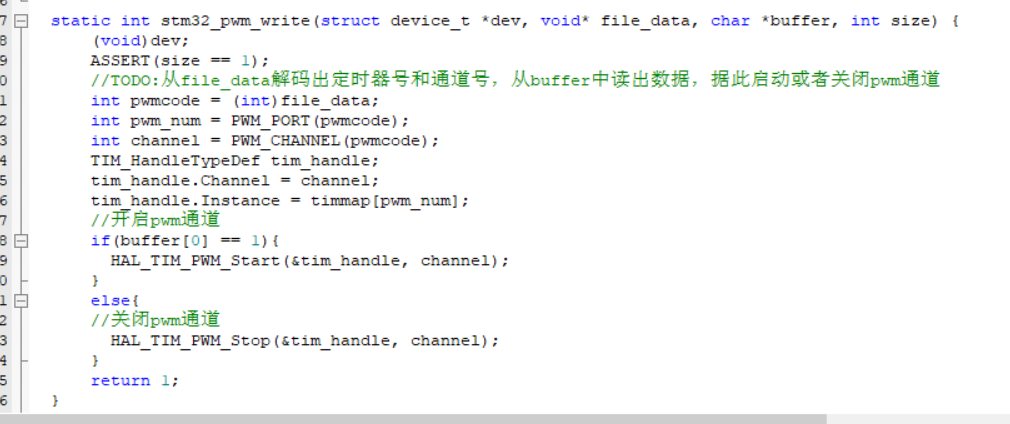
1. 编写stm32\_pwm\_close函数

stm32\_pwm\_close函数不需要我们添加多余的操作，直接返回零即可。



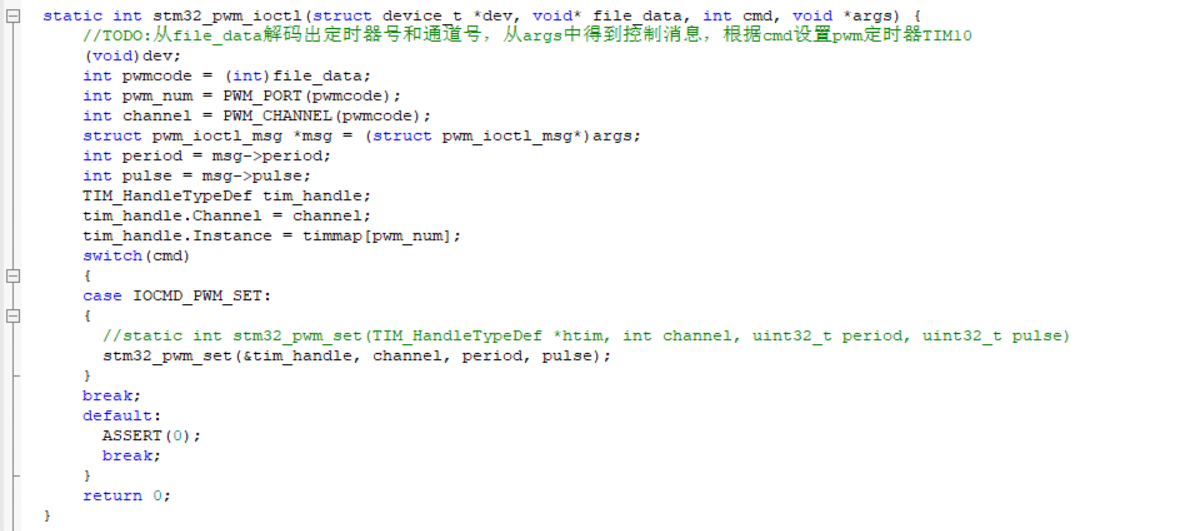
1. 编写stm32\_pwm\_write函数

在stm32\_pwm\_write函数中，和上述的open函数的编写较为类似，都是通过file data获取pwmcode，并将pwmcode解码得到通道号和端口号，再创建一个handle作为后续关闭/开启pwm通道的参数，并根据buffer的值选择关闭还是启动PWM通道。如果buffer[0]的值为1则选择启动PWM通道，否则则关闭，这里同样是调用了下一层的函数HAL\_TIM\_PWM\_Start和HAL\_TIM\_PWM\_Stop来驱动硬件执行操作。



1. 编写stm32\_pwm\_ioctl函数

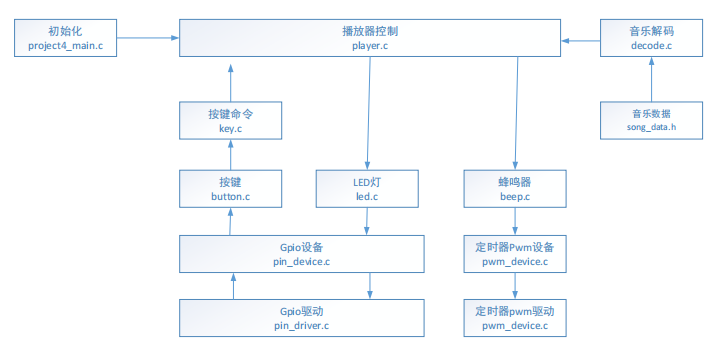
该函数同样是通过file data获取pwmcode，并将pwmcode解码得到通道号和端口号，再创建一个handle作为后续传入的参数。不同的是，需要根据cmd设置pwm定时器TIM10，当cmd为IOCMD\_PWM\_SET时调用stm32\_pwm\_set函数。



# 实验方案与实现

## 软件结构

播放器由 player.c 控制，初始化时启动一个任务 player\_entry 控制播放器的暂停与播放。控制命令来自 key.c ，初始化是启动了一个 20ms 周 期 的 定 时 器button\_manage.timer，定期扫描按键的值。音乐数据来自 song\_data.h，调用 decode.c 解码后，调用 beep.c 改变TIM10 的 PWM 输出，控制蜂鸣器的声音。



## 源代码

/\*

 \* driver\_pwm.c

 \*

 \*      Author: ljp

 \*/

#include <string.h>

#include "stm32f4xx\_hal.h"

#include "pwm.h"

static TIM\_TypeDef \* timmap[] = {

    TIM1, TIM2, TIM3, TIM4, TIM5, TIM6, TIM7, TIM8, TIM9, TIM10, TIM11, TIM12, TIM13, TIM14, 0

};

static struct pwm\_device \_pwm\_dev;

// /10.1 -> 10.1

static const char \* get\_file\_name(const char \*pathname) {

    int i;

    const char \*filename = pathname;

    //skip path name

    for (i = 0; pathname[i] != 0; i++) {

        if(pathname[i] == '/') {

            filename = pathname + i + 1;

        }

    }

    return filename;

}

//name "10.1" -> pwm\_num = 10, channel = 1

static int stm32\_pwm\_get(const char \*name)

{

    int pwmcode = 0;

    int pwm\_num, channel;

    int name\_len;

    name = get\_file\_name(name);

    name\_len = strlen(name);

    if ((name\_len < 3) || (name\_len > 4)) {

        return -1;

    }

    if(name[1] == '.') {

        pwm\_num = (int)(name[0] - '0');

        channel = (int)name[2] - '0';

        pwmcode = PWM\_CODE(pwm\_num, channel);

        return pwmcode;

    } else if(name[2] == '.') {

        pwm\_num = (int)((name[0] - '0') \* 10 + (name[1] - '0'));

        channel = (int)name[3] - '0';

        pwmcode = PWM\_CODE(pwm\_num, channel);

        return pwmcode;

    } else {

        return -1;

    }

}

/\* APBx timer clocks frequency doubler state related to APB1CLKDivider value \*/

static void pclkx\_doubler\_get(uint32\_t \*pclk1\_doubler, uint32\_t \*pclk2\_doubler)

{

    uint32\_t flatency = 0;

    RCC\_ClkInitTypeDef RCC\_ClkInitStruct = {0};

    ASSERT(pclk1\_doubler != NULL);

    ASSERT(pclk1\_doubler != NULL);

    HAL\_RCC\_GetClockConfig(&RCC\_ClkInitStruct, &flatency);

    \*pclk1\_doubler = 1;

    \*pclk2\_doubler = 1;

#if defined(SOC\_SERIES\_STM32MP1)

    if (RCC\_ClkInitStruct.APB1\_Div != RCC\_APB1\_DIV1)

    {

        \*pclk1\_doubler = 2;

    }

    if (RCC\_ClkInitStruct.APB2\_Div != RCC\_APB2\_DIV1)

    {

       \*pclk2\_doubler = 2;

    }

#else

    if (RCC\_ClkInitStruct.APB1CLKDivider != RCC\_HCLK\_DIV1)

    {

         \*pclk1\_doubler = 2;

    }

#if !defined(SOC\_SERIES\_STM32F0) && !defined(SOC\_SERIES\_STM32G0)

    if (RCC\_ClkInitStruct.APB2CLKDivider != RCC\_HCLK\_DIV1)

    {

         \*pclk2\_doubler = 2;

    }

#endif

#endif

}

//dev name as "/pwm", filename as "10.1" -> 0xA1

static int stm32\_pwm\_open(struct device\_t \*dev, const char \*filename, int flags, int mode) {

    (void)dev;

    (void)flags;

    (void)mode;

    //get pwmcode

    //enable TIM

        int pwmcode;

        pwmcode = stm32\_pwm\_get(filename);

        int pwm\_num = PWM\_PORT(pwmcode);

        int channel = PWM\_CHANNEL(pwmcode);

        TIM\_HandleTypeDef tim\_handle;

        tim\_handle.Channel = channel;

        tim\_handle.Instance = timmap[pwm\_num];

        HAL\_TIM\_PWM\_Start(&tim\_handle, channel);

    timmap[pwm\_num]->ARR = 1000000 / 262;

    timmap[pwm\_num]->CCR1 = 1000000 / 262 \* 0.8;

    //TODO:从filename提取定时器号和通道号，编码到pwmcode中，return会传回上层的fd->file\_data中

    if(pwmcode == -1) {

        return -1;

    }

    return pwmcode;

}

static int stm32\_pwm\_close(struct device\_t \*dev, void\* file\_data) {

    (void)dev;

    (void)file\_data;

    //no action

    return 0;

}

static int stm32\_pwm\_write(struct device\_t \*dev, void\* file\_data, char \*buffer, int size) {

    (void)dev;

    ASSERT(size == 1);

    //TODO:从file\_data解码出定时器号和通道号，从buffer中读出数据，据此启动或者关闭pwm通道

        int pwmcode = (int)file\_data;

        int pwm\_num = PWM\_PORT(pwmcode);

        int channel = PWM\_CHANNEL(pwmcode);

        TIM\_HandleTypeDef tim\_handle;

        tim\_handle.Channel = channel;

        tim\_handle.Instance = timmap[pwm\_num];

        //关闭pwm通道

        if(buffer[0] == 1){

            HAL\_TIM\_PWM\_Start(&tim\_handle, channel);

        }

        else{

        //启动pwm通道

            HAL\_TIM\_PWM\_Stop(&tim\_handle, channel);

        }

    return 1;

}

#define MAX\_PERIOD 65535

#define MIN\_PERIOD 3

#define MIN\_PULSE 2

static int stm32\_pwm\_set(TIM\_HandleTypeDef \*htim, int channel, uint32\_t period, uint32\_t pulse)

{

    uint64\_t tim\_clock, psc;

    uint32\_t pclk1\_doubler, pclk2\_doubler;

    /\* Converts the channel number to the channel number of Hal library \*/

    channel = 0x04 \* (channel - 1);

    pclkx\_doubler\_get(&pclk1\_doubler, &pclk2\_doubler);

    if (1)

    {

        tim\_clock = (uint32\_t)(HAL\_RCC\_GetPCLK2Freq() \* pclk2\_doubler);

    }

    else

    {

        tim\_clock = (uint32\_t)(HAL\_RCC\_GetPCLK1Freq() \* pclk1\_doubler);

    }

    /\* Convert nanosecond to frequency and duty cycle. 1s = 1 \* 1000 \* 1000 \* 1000 ns \*/

    tim\_clock /= 1000000UL;

    period = (unsigned long long)period \* tim\_clock / 1000ULL ;

    psc = period / MAX\_PERIOD + 1;

    period = period / psc;

    \_\_HAL\_TIM\_SET\_PRESCALER(htim, psc - 1);

    if (period < MIN\_PERIOD)

    {

        period = MIN\_PERIOD;

    }

    \_\_HAL\_TIM\_SET\_AUTORELOAD(htim, period - 1);

    pulse = (unsigned long long)pulse \* tim\_clock / psc / 1000ULL;

    if (pulse < MIN\_PULSE)

    {

        pulse = MIN\_PULSE;

    }

    else if (pulse > period)

    {

        pulse = period;

    }

    \_\_HAL\_TIM\_SET\_COMPARE(htim, channel, pulse - 1);

    \_\_HAL\_TIM\_SET\_COUNTER(htim, 0);

    /\* Update frequency value \*/

    HAL\_TIM\_GenerateEvent(htim, TIM\_EVENTSOURCE\_UPDATE);

    return 0;

}

static int stm32\_pwm\_ioctl(struct device\_t \*dev, void\* file\_data, int cmd, void \*args) {

    //TODO:从file\_data解码出定时器号和通道号，从args中得到控制消息，根据cmd设置pwm定时器TIM10

    (void)dev;

        int pwmcode = (int)file\_data;

        int pwm\_num = PWM\_PORT(pwmcode);

        int channel = PWM\_CHANNEL(pwmcode);

        struct pwm\_ioctl\_msg \*msg = (struct pwm\_ioctl\_msg\*)args;

        int period = msg->period;

        int pulse = msg->pulse;

        TIM\_HandleTypeDef tim\_handle;

        tim\_handle.Channel = channel;

        tim\_handle.Instance = timmap[pwm\_num];

        switch(cmd)

        {

        case IOCMD\_PWM\_SET:

        {

            //static int stm32\_pwm\_set(TIM\_HandleTypeDef \*htim, int channel, uint32\_t period, uint32\_t pulse)

            stm32\_pwm\_set(&tim\_handle, channel, period, pulse);

        }

        break;

        default:

            ASSERT(0);

            break;

        }

    return 0;

}

void driver\_pwm\_init(void) {

    \_pwm\_dev.parent.dev\_ops.open = stm32\_pwm\_open;

    \_pwm\_dev.parent.dev\_ops.write = stm32\_pwm\_write;

    \_pwm\_dev.parent.dev\_ops.close = stm32\_pwm\_close;

    \_pwm\_dev.parent.dev\_ops.ioctl = stm32\_pwm\_ioctl;

    device\_register(&\_pwm\_dev.parent, PWM\_DEV\_PATH);

}

# 实验结果与分析

1.请描述整个应用程序的结构。包括哪些模块，彼此之间的关系如何。

整个应用程序的结构与上述软件结构中的图类似。首先有一个控制播放器控制模块player.c，它主要控制整个播放器的操作，接收按键命令返回的数据，并把这些命令传递给LED灯和蜂鸣器来进行播放。

此外还有四个小的模块，分别包括按键、LED、蜂鸣器和音乐解码模块。在音乐解码模块，接收来自data.h的音乐数据，并在decode.c文件中解码，把这相关的数据传送给播放器。在蜂鸣器模块，它接受来自播放器控制传来的数据，传送给PWM设备，调动PWM驱动进行播放。Led灯和按键模块均属于GPIO设备，其中按键模块为输入模块，它返回K3到K5的按键的命令的值并返回给播放器，再由播放器传送给对应的led灯，Led灯输出对应的状态。

2.整个应用程序创建了几个任务，都完成什么功能？

观察project4\_main函数中可以看到，整个应用程序主要创建了player播放器和key两大任务，分别用于管理播放器和扫描获取按键状态两大任务。其中，在播放器任务中，需要获取来自音乐解码模块的歌曲音乐信息，并将其传递给LED灯和蜂鸣器来实现播放和音乐的展示。

3.应用层的 open/read/write 文件操作是如何映射到底层的 GPIO 和 PWM 接

口寄存器操作的？

应用层的open/read/write等文件操作映射到底层的寄存器操作关系如下。首先在最顶层是设备文件接口层，该层提供类unix系统的相关文件操作函数包括open/read/write等，然后将其传输到文件描述符ios\_fd.c文件中，函数映射在此层中均基于一个文件描述符fd来进行操作，存储了一些文件的设备名称等私有数据信息。接下来将文件描述符层的函数映射到io设备子系统层，这一个层主要的功能是绑定设备和驱动，在这一个系统层里面管理了一张设备列表。在设备列表中，每一项对应的是一个特定的设备驱动，一个设备驱动对应一类设备。在这里的话，我们是将其绑定的GPIO和PWM驱动，以此来调用设备驱动进行操作。



# 实验总结

最后一个实验感觉是一个较为综合的实验，它包括了前几次实验的内容，如GPIO输入输出以及PWM时钟周期处理的内容。就代码编写整体而言，难度也是较前面几次实验大一些。与前几次实验有相关的参照不同，这次实验更多的是需要我们独立完成代码的编写，这就非常的考验我们对整体代码结构的理解和认识，以及对硬件结构的了解。

所以在编写代码时，最深刻的感受就是递归式查询，根据原理知道要调用某个函数时，这个函数有哪些参数，参数又由哪些变量构成的，那些变量分别对应哪些数据，需要我们一一进行查询处理。例如，在编写stm32\_pwm\_open函数的时候，查询HAL\_TIM\_PWM\_Start函数定义知道需要传入哪些参数，但是它对应TIM\_HandleTypeDef变量，就需要再查这个结构体中含有哪些数据，要把一些关键性的数据传进去才能实现这个函数的功能。而这个不断递归式查询的过程，也让我更深刻地体会了硬件和应用程序间一层一层函数调用的过程。每层的功能都相对独立，这种方式可能执行起来会繁琐一点（要一层层调用），但是各部分的功能都是更加明确和清晰，加深了我对操作系统对设备驱动的理解和认知。