蜂鸣器音乐播放器

目录

[功能描述 3](#_Toc170596253)

[软件功能 3](#_Toc170596254)

[安全功能 3](#_Toc170596255)

[设计方案 4](#_Toc170596256)

[环境 4](#_Toc170596257)

[硬件 4](#_Toc170596258)

[软件设计 4](#_Toc170596259)

[状态机 4](#_Toc170596260)

[main.c/main() 5](#_Toc170596261)

[main.c/state\_wait() 7](#_Toc170596262)

[main.c/state\_play() 8](#_Toc170596263)

[scores.c/play\_score() 8](#_Toc170596264)

[scores.c/play\_sound() 9](#_Toc170596265)

[乐谱表示 10](#_Toc170596266)

[安全设计描述 11](#_Toc170596267)

[数据备份与校验（data.c） 11](#_Toc170596268)

[数据结构设计 11](#_Toc170596269)

[备份设计 13](#_Toc170596270)

[热启动 15](#_Toc170596271)

[看门狗 16](#_Toc170596272)

[引入 16](#_Toc170596273)

[初始化&启动 17](#_Toc170596274)

[喂狗 17](#_Toc170596275)

[输入滤波 18](#_Toc170596276)

[输出滤波 19](#_Toc170596277)

[控制冗余（检查执行顺序） 20](#_Toc170596278)

[执行冗余 21](#_Toc170596279)

[独立代码随机顺序执行 21](#_Toc170596280)

[等价实现随机选择执行 21](#_Toc170596281)

[随机Delay 22](#_Toc170596282)

[睡眠降低功耗、躲避干扰 22](#_Toc170596283)

[代码、数据优化 23](#_Toc170596284)

[空间优化实验 24](#_Toc170596285)

[思路1：提取公共代码 24](#_Toc170596286)

[思路2：循环压缩 28](#_Toc170596287)

[设计总结（心得体会） 31](#_Toc170596288)

# 功能描述

软件功能概述：

主功能为根据存储的乐谱**播放音乐**，其它功能有：**按键选择**曲目、**按键打断**播放、**数码管显示**当前状态。

## 软件功能

1. 启动后进入WAIT状态，数码管显示“0”，并等待用户按键。
2. 用户按下3、6、9可分别选择播放曲目1、曲目2、曲目3，播放时数码管会显示对应曲目的序号。
3. 用户按下其它按键则仍为WAIT状态。
4. 曲目播放过程中可以按任意键打断播放，打断后将回到WAIT态。
5. 曲目播放完毕会自动回到WAIT态。
6. 此外，每次状态切换前会短暂地回显“200”，表示当前状态已正常结束。

## 安全功能

* 数据备份
* 数据校验
* 热启动
* 看门狗
* 输入滤波
* 输出滤波
* 控制冗余（检查执行顺序）
* 执行冗余
  + 独立代码随机顺序执行
  + 等价实现随机选择执行
  + 随机Delay
* 睡眠降低功耗、躲避干扰
* 代码、数据优化

# 设计方案

## 环境

机箱：华清远见嵌入式ARM实验箱Ⅲ。

芯片：ARM Cortex-M4微处理器。

环境：Keil uVision5 on Windows 10。

项目：基于**3\_DC\_Motor**项目文件。

## 硬件

音频实现：蜂鸣器。

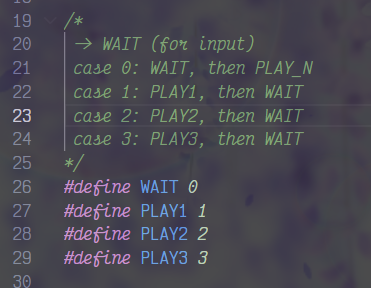
交互实现：数码管、键盘。

# 软件设计

## 状态机

共四个状态：

1. 等待（用户键盘输入选择曲目）。
2. 播放曲目1。
3. 播放曲目2。
4. 播放曲目3。



状态转移关系：



状态机实现（伪码）：

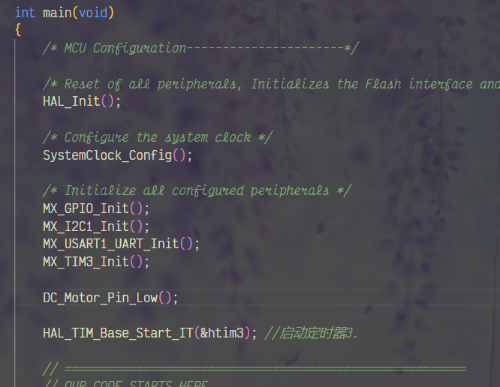


即，在while循环中每次switch(state)，并在每个case结束前修改状态，这样下一次循环即转移到下一状态。

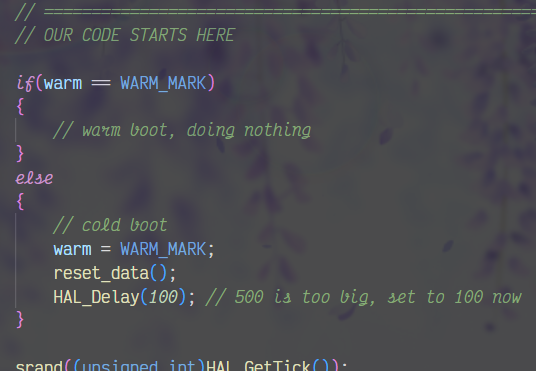
## main.c/main()

main函数流程为：

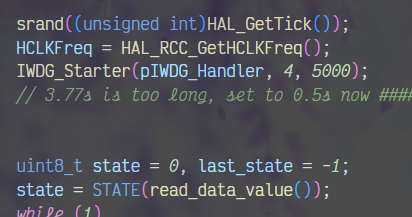
1. 初始化外设等



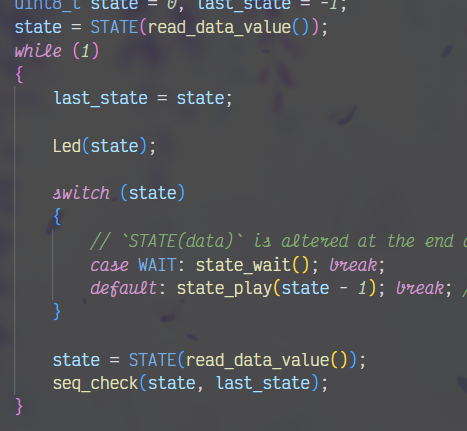
1. 冷热启动检查



1. 初始化全局变量等



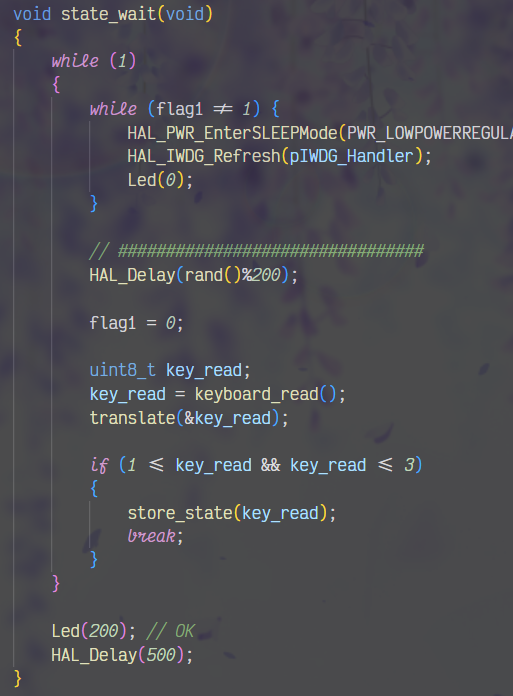
1. 进入状态机



## main.c/state\_wait()

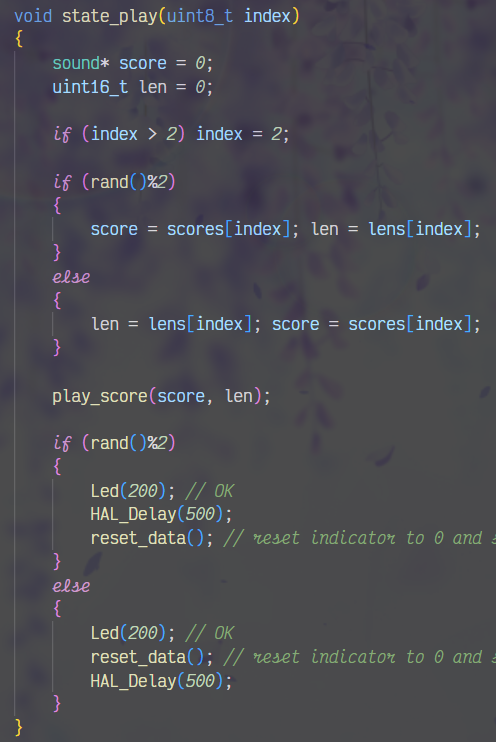
等待状态：

* 只要用户没有按键，就循环进行睡眠、喂狗、刷新显示。
* 发生按键时，检查按键是否对应曲目1~3，是，则修改状态，退出state\_wait()。
* 否则回到上述无按键时的循环。



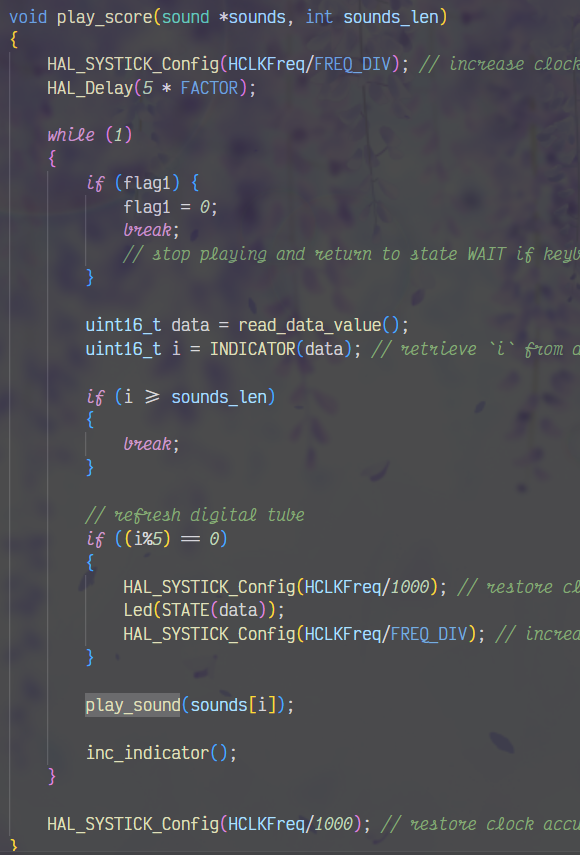
## main.c/state\_play()

* 根据传入的曲目序号，取出对应的乐谱等数据，然后转到play\_score()。
* 执行完毕后，修改状态，退出。



## scores.c/play\_score()

* 主循环：根据indicator取出乐谱中的第indicator个音符，调用play\_sound()播放该音符，播放后递增indicator。
* 循环中：
  + 定期刷新数码管显示。
  + 每次都检测是否发生了按键中断，如发生则退出循环，退出播放。
* 此外，由于play\_sound()中需要较高精度的时钟，play\_score也负责来回调节时钟精度。



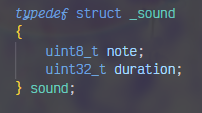
## scores.c/play\_sound()

* 根据音符，查表得到该音高对应的震动周期。
* 如果是休止符，HAL\_Delay()后返回即可。
* 否则，按1/2周期不断翻转蜂鸣器，实现发声。
* 发声持续0.95个duration，而后延时（静默）0.05个duration，返回。

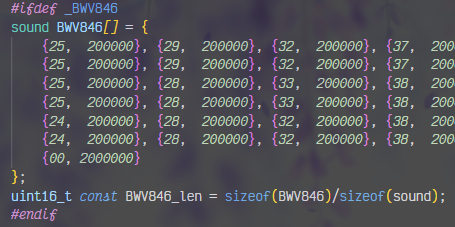


## 乐谱表示

note为音符序号，duration为持续时间。



例：



更详细的设计请参考“安全设计描述”的“代码、数据优化” 章节。

# 安全设计描述

## 数据备份与校验（data.c）

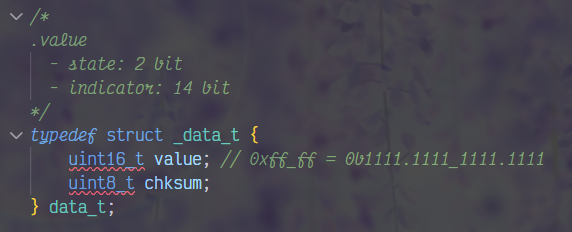
### 数据结构设计

#### 定义

对以下数据做了加强保护：

1. 状态机状态。
2. 当前曲目的播放进度。

数据结构定义如下：



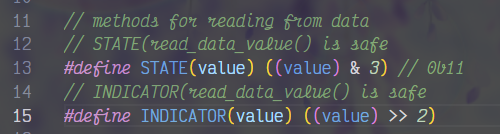
其中data.value的低两位存储状态（state），高14位储存曲目播放进度（indicator）。

这样设计的原因是：

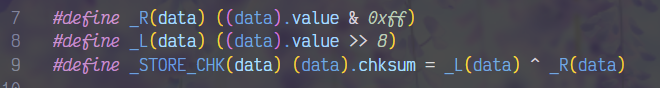
1. state总共四个值，2 bit正好。
2. 根据估算，曲目播放进度的存储大概需要8 bit以上，16 bit以下的内存。

#### 操作

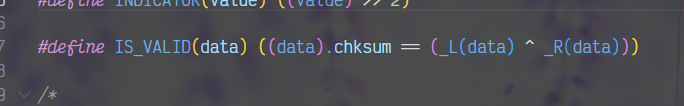
读取数据：



**计算校验和**：

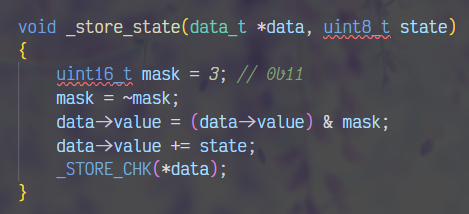


**检验校验和**：

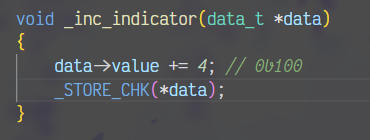


以下为修改操作，修改数据时，**同时更新校验和**。

修改state：



递增indicator：



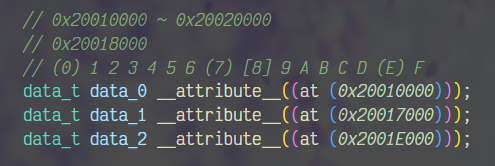
重置数据：

见下方“备份”章节。

### 备份设计

#### 定义

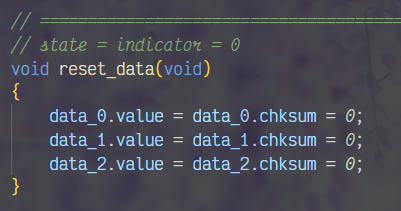
共三份备份，注意**隔开存放**：



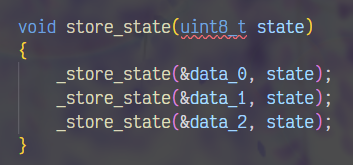
#### 写备份

提供了一次性操作三份数据的统一接口，如下。

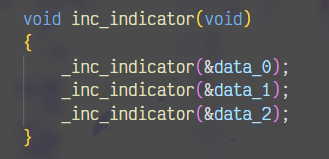
数据归零：



修改state：



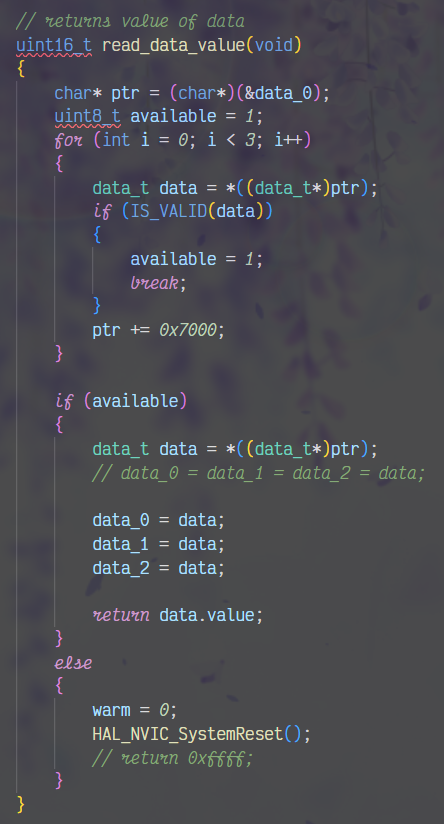
递增indicator：



#### 读备份

1. 循环检查三份备份中有无可用数据（校验和正确即认为可用）。
2. 若可用，用该数据依次更新三份备份，而后返回该数据。
3. 不可用，则冷启动。

注意在更新三份备份的时候，**避免使用data0=data1=data2=data**这样的操作，因为对这种写法的处理在标准中是未定义的，**不同编译器处理可能不一样**，不能保证是依次更新三份备份，**可能导致三份备份全部损坏**。



## 热启动

首先需要在项目设置中将原本的内存分出一块给热启动数据。设置上，勾上**NoInit**选项即可，这样，这些数据在设备重启时不会被初始化，仍为重启之前的值。

这里选择了0x200**10000**~0x200**20000**的内存段。

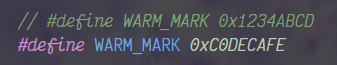
热启动标记变量：

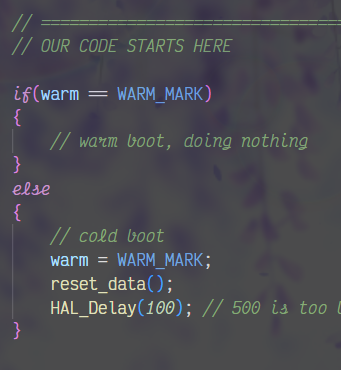


以及数据备份中的各数据也是定义在这一段中。

main()中，外设初始化指令结束后即**检查标记是否为给定值**。

* 不是，则设置warm值，重置数据，并延时等待外设初始化。
* 是，则什么都不做（因为数据在**读取时**会被**校验**，这里不需要再校验了）。

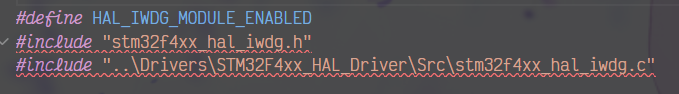




## 看门狗

### 引入

看门狗头文件、源文件分别是stm32f4xx\_hal\_iwdg.h和stm32f4xx\_hal\_iwdg.c。由于原项目中没有添加这个源文件，这里选择手动include，注意需要#define HAL\_IWDG\_MODULE\_ENABLED启动IWDG。

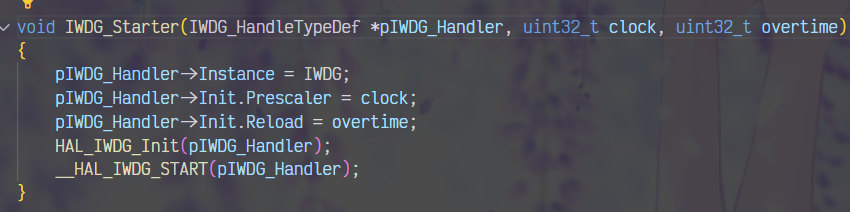


看门狗全局变量：

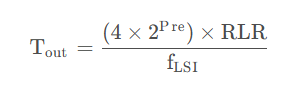


### 初始化&启动

自己封装了一个函数：



看门狗超时时间计算公式（Pre和RLR分别是参数中的Prescaler和Reload）：



在main()中，启动看门狗：

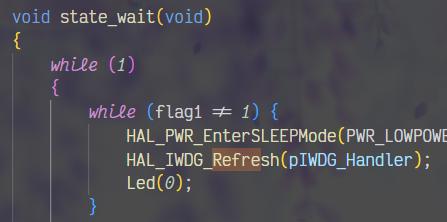


之前设置的超时间隔太长，经老师建议后，现在设置为0.5秒左右。

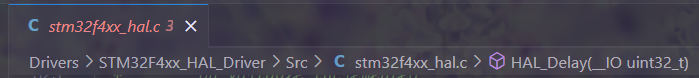
### 喂狗

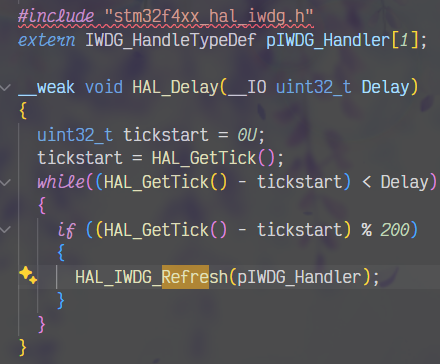
为防止程序在“跑飞”的情况下击中喂狗指令，应尽量在空间上减少喂狗指令的出现，故本程序中仅在三个必要点进行了喂狗。

1. 等待输入时，每次被中断唤醒即喂狗。

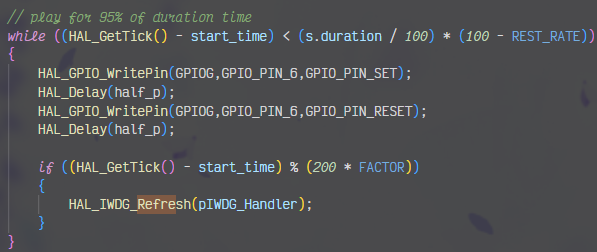


1. HAL\_Delay()中，每0.2s喂一次狗。





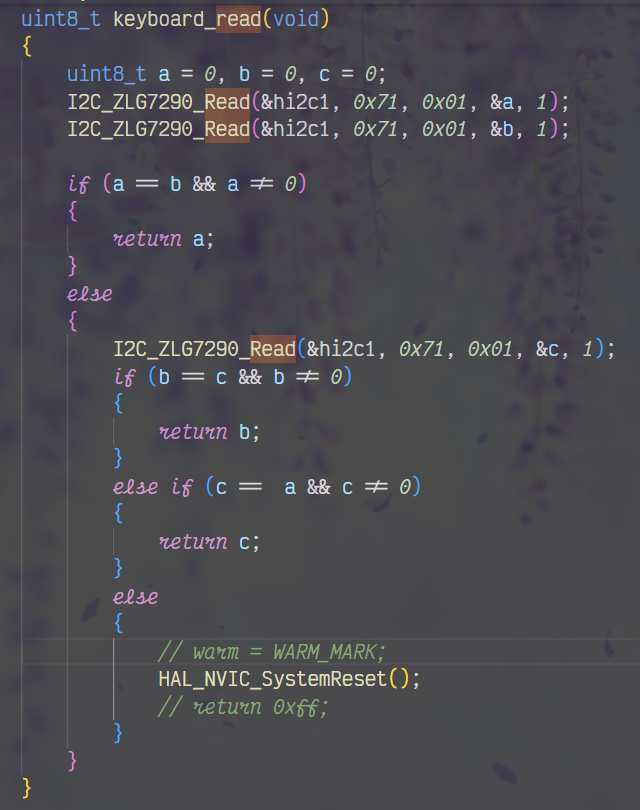
1. play\_sound()的发声循环中，每0.2s喂一次狗。



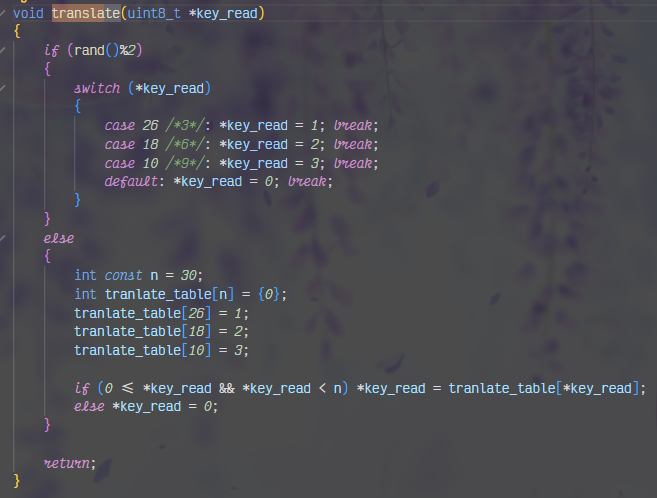
## 输入滤波

main.c/keyboard\_read()：

1. 两次读取，若相等且不为空则返回。
2. 否则读第三次并在非空数据中按少数服从多数返回。
3. 若非空数据中没有两两相等的，视为硬件故障，热启动重启。



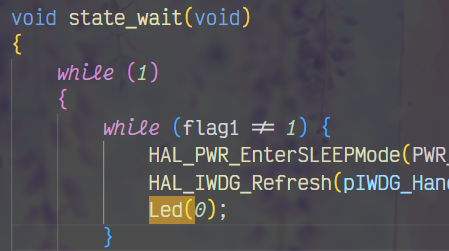
main.c/translate()检查、转换输入内容：



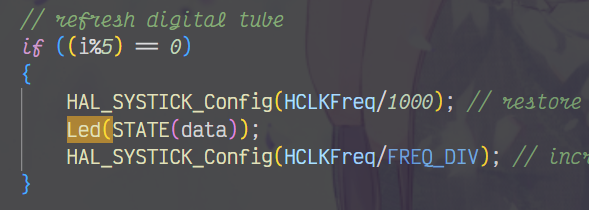
## 输出滤波

输出滤波体现在数码管的定期刷新上。

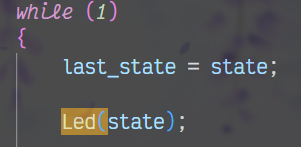
* 等待输入时每次中断唤醒即刷新显示。



* play\_score()中，每播放五个音符即刷新一次显示。

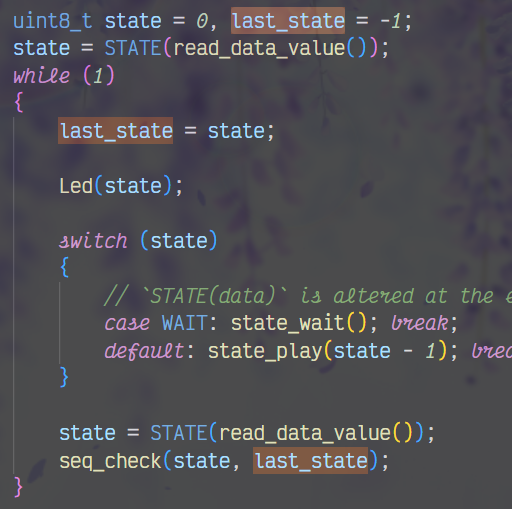


* 状态机循环中，每次切换状态即刷新一次。

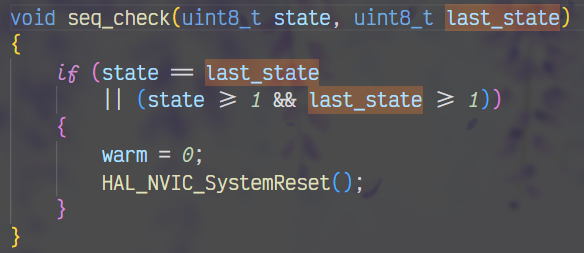


## 控制冗余（检查执行顺序）

由于本项目的状态机较简单，执行序列只有WAIT -> PALYN -> WAIT -> …的循环，只需**存下上一状态**，检查是否是**WAIT与PLAYN交错**即可。



main.c/**seq\_check()**：

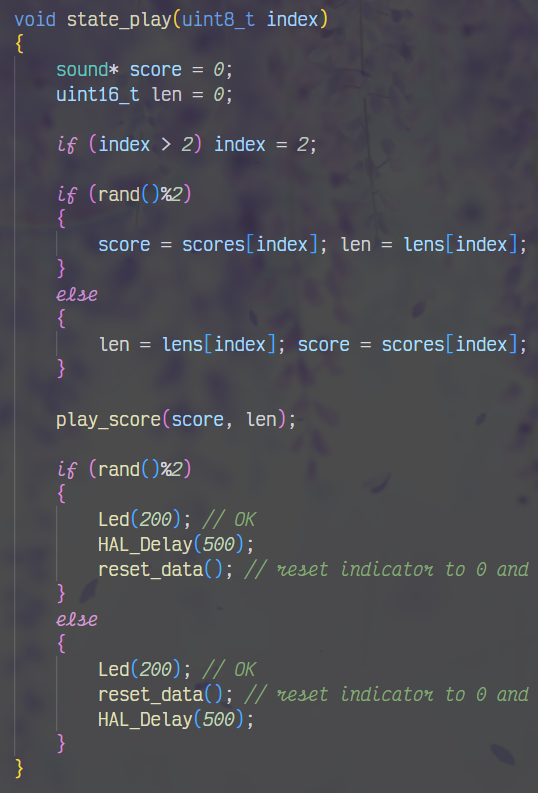


检查不合法直接冷启动重启。

## 执行冗余

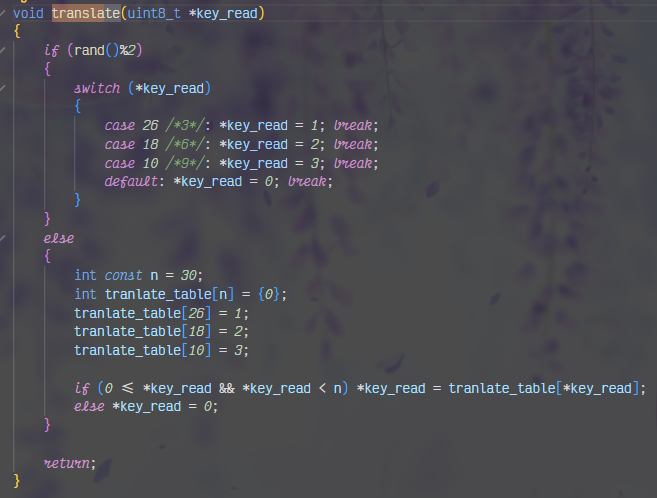
### 独立代码随机顺序执行

main.c/state\_play()中的两处分枝：



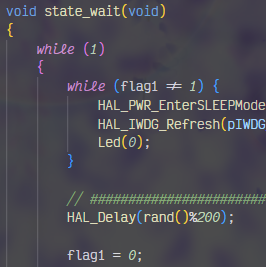
### 等价实现随机选择执行

main.c/translate()的翻译分别通过查表和switch语句实现：



### 随机Delay

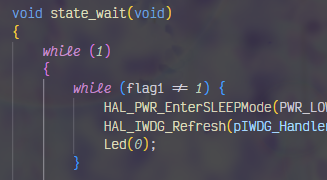
等待状态下得到输入后，随机延时0~0.2s再相应。



## 睡眠降低功耗、躲避干扰

等待状态下无键盘中断则循环睡眠：

* 中断发生时醒来，刷新显示和看门狗。
* 检查中断是否是键盘中断。
* 否：继续睡眠。
* 是：处理输入。



## 代码、数据优化

初始代码的一些问题：

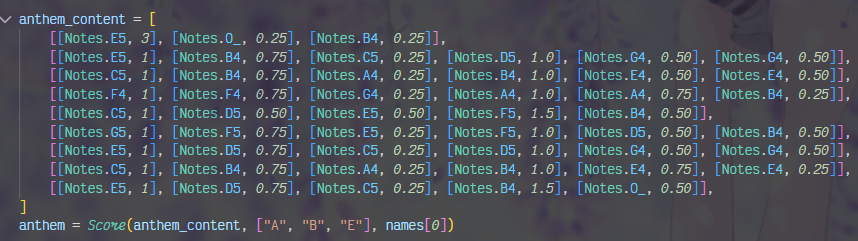
1. 时长相关的各种数据（比如sound的duration成员）都使用**double**类型存储，且未使用**uintN\_t**来指定各种整型的精确大小。
2. 由于乐谱**数据**需要进行调号等**预处理**，在状态机中实现较为麻烦且不美观。

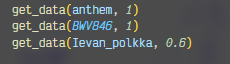
解决如下：

首先，所有数据都使用整形，且使用**uintN\_t**指定好精确的所需大小。

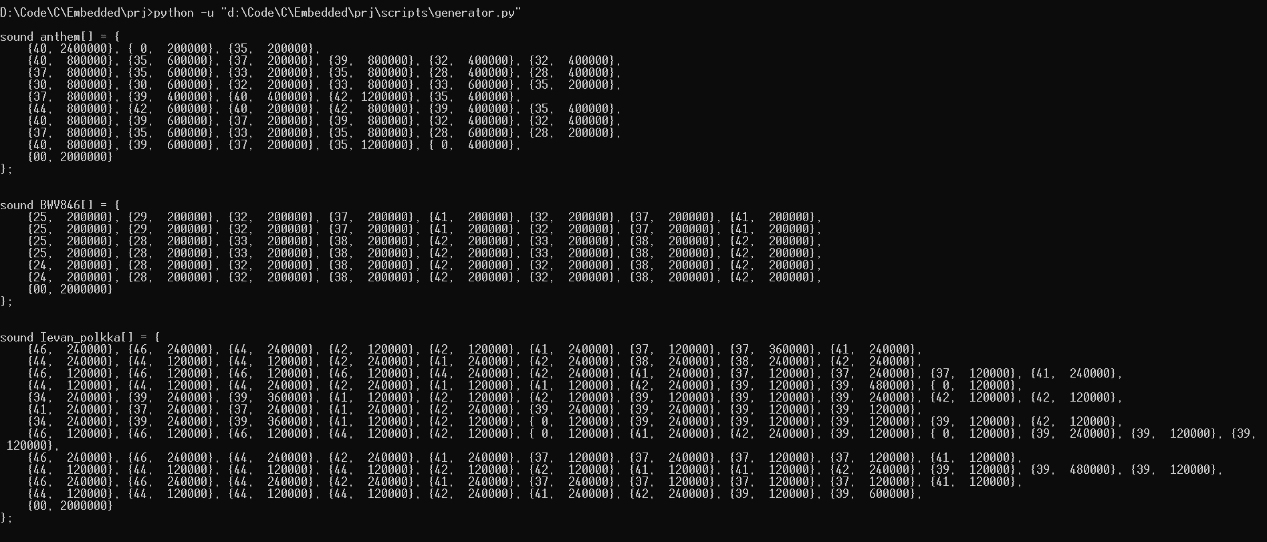
此外，使用Python进行**数据预处理**。

**scripts/generator.py**预处理乐谱：





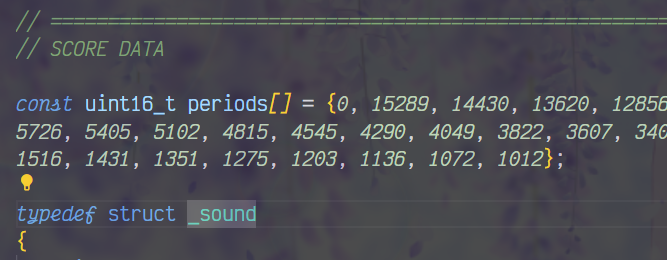
执行后得到处理好的C数组：



**scripts/periods.py**根据十二平均律，计算各音高的振动周期：



C数组：



# 设计总结（心得体会）

掌握了基于**状态机**的软件/功能设计方法。

掌握了该开发板的蜂鸣器、数码管等**硬件**的使用方法。

实操了数据类型优化、数据预处理优化、循环压缩等代码**优化**技术。

实操了数据备份、校验的实现与**封装**。

掌握了**热启动**的原理与实现方法，深刻体会到了热启动的**强大作用**。

掌握了**看门狗**的配置、使用方法，以及使用中的各种**细节/原则**，如超时时间设置、喂狗语句限制等等。

掌握了输入、输出**滤波**的基本方法。

掌握了控制、执行**冗余**的实现方法。

掌握了该开发板下**睡眠**功能的原理和一些使用技巧，如可以使用while循环来在中断后继续睡眠等等。