目录

[一. 实训目标 4](#_Toc472359697)

[二. 达成目标情况 4](#_Toc472359698)

[三. 具体分析过程 4](#_Toc472359699)

[1、 将输入输出具体的逻辑分离。 4](#_Toc472359700)

[1) 将整体黑箱化，分离输入输出 4](#_Toc472359701)

[2) 显示刷新函数的编写 5](#_Toc472359702)

[3) 轮询与定时结合的定时查询法编写键盘扫描 6](#_Toc472359703)

[4) 显示闪烁标志位的编写 8](#_Toc472359704)

[2、 将各个功能状态化 9](#_Toc472359705)

[1) 关于状态机的描述 9](#_Toc472359706)

[2) 有关于状态机的个人改进 10](#_Toc472359707)

[3) 有限状态机的具体代码实现 12](#_Toc472359708)

[4) 编写任务四中的显示时间模块和秒表模块 14](#_Toc472359709)

[5) 这种方法的优缺点 18](#_Toc472359710)

[3、 为需要输入缓存的功能编写缓存器 18](#_Toc472359711)

[1) 简述缓存器 18](#_Toc472359712)

[2) 缓存器的基本工作原理 19](#_Toc472359713)

[3) 自己总结的有关于缓存器的基本构成 21](#_Toc472359714)

[4、 将功能模块组合起来 22](#_Toc472359715)

[1) 使用switch语句来判断状态 22](#_Toc472359716)

[5、 让程序更易读 25](#_Toc472359717)

[1) 分离c/h文件 25](#_Toc472359718)

[2) 文件结构化 26](#_Toc472359719)

[6、 一些小的推荐扩展 26](#_Toc472359720)

[1) 去除延时 26](#_Toc472359721)

[2) 空闲模式 27](#_Toc472359722)

[7、 注意事项的思考 27](#_Toc472359723)

[1) 每个过程所需的时间考虑 27](#_Toc472359724)

[2) 时序的控制 27](#_Toc472359725)

[8、 整体系统的优缺点 28](#_Toc472359726)

[1) 优点 28](#_Toc472359727)

[2) 缺点 28](#_Toc472359728)

[四. 实训心得 29](#_Toc472359729)

# 实训目标

实现任务四的基本功能

将各个功能模块化

总结出属于自己的编程模式和程序框架

# 达成目标情况

将功能基本实现

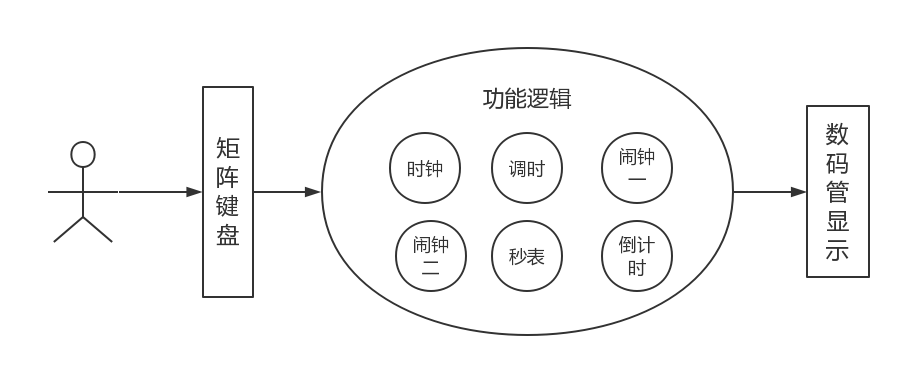
从JAVA的监听机制和数电状态机中得到启示，构建了一个程序框架，将功能实现规范化了。

# 具体分析过程

## 将输入输出具体的逻辑分离。

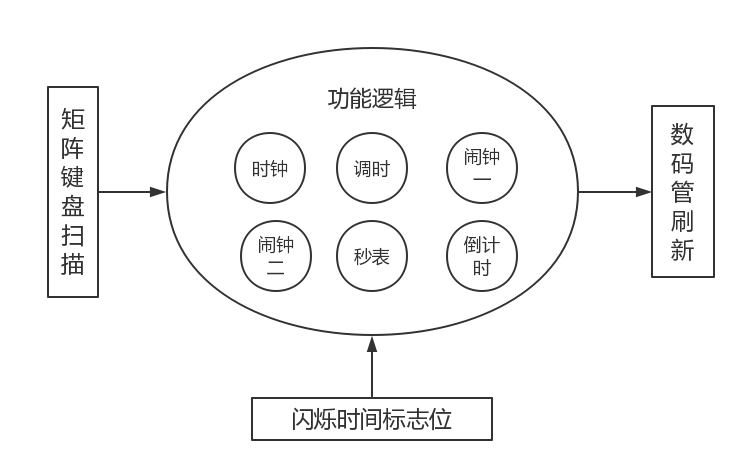
### 将整体黑箱[[1]](#footnote-1)化，分离输入输出

对于每一个所实现的功能，我们都可以把他黑箱化，以任务四为例子，可以将其输入输出分离，得到如下的的结构图：



因此，我们可以先把输入与输出的功能实现好，一般以我个人来讲，在输入方面多采用扫描输入，而当系统需要实时反馈的时候，可以采用中断输入。显示方面，因为是面向一个黑匣子得到的输入，所以编写一个时时刻刻关注显示缓存区[[2]](#footnote-2)的显示刷新函数或许更合适一些。另外，根据经验，对于一些显示要用到的标志位，如闪烁标志位等，先写出来也是会方便许多的。

这样我们就得到了如下的结构图：



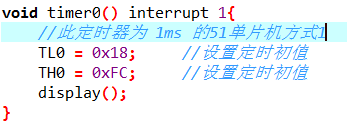
也就是说，我们在这一步，要完成好显示刷新函数，矩阵键盘扫描函数和闪烁时间标志位函数。

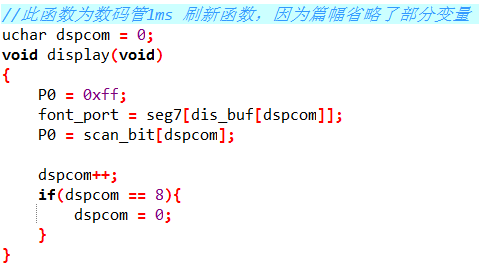
### 显示刷新函数的编写

一般来说数码管的显示刷新分为定时器法和大循环法。由于一般功能需求而言，都要求输入不能影响到数码管显示，而相较于放入主函数的大循环法，定时器法更容易做到这一点，所以推荐优先使用定时器法。当然定时器法同样的有许多弊端，不过只要稍加注意，也是能顺利完成功能实现的，在之后的注意事项里头，我会谈谈一些自己平时遇到的一些定时器法出现的状况和解决方法，可以参考一下。

对于定时器法，我使用的是1ms的定时器，每次定时中断后会刷新一个位，这样的话，以8位数码管来举例，一秒可以刷新125帧，而红光人眼能感受到的最高帧数不过50帧，也就是说运行过程中如果关闭数码管相应的定时器不超过1ms的话，是感受不到那个位的闪烁，为之后在一些需要严格控制时间的外部时序芯片做了一定的预留时间。

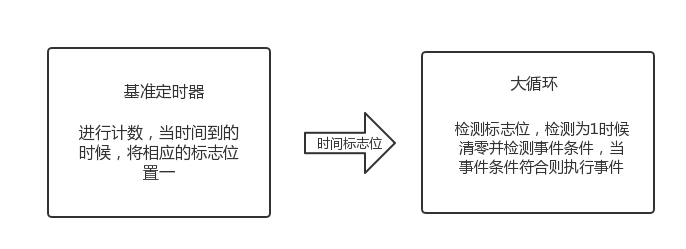
具体的实现代码较为简单，（此处没给出定时器的初始化）如下：





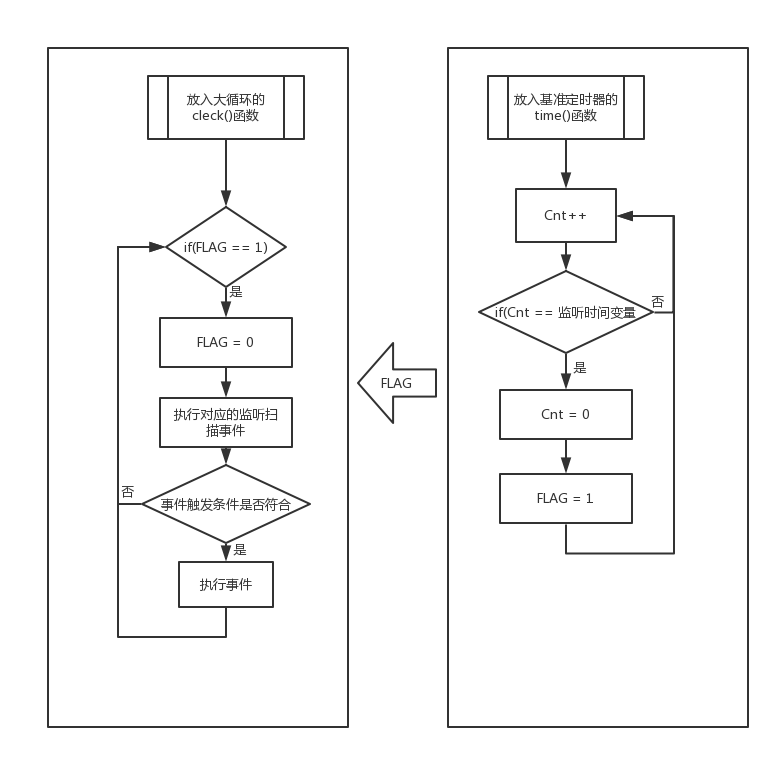
### 轮询与定时结合的定时查询法编写键盘扫描

一般来讲，我们在检测事件是否触发时，例如检测键盘是否出发，有两种方法，一种是不断的检测是否有键盘按下，另外一种是通过硬件提供的外部中断来检测是否有键盘按下。第一种受限于大循环的逻辑，一但功能过于复杂，就会出现说无法及时的相应键盘，并且在debug的过程中也略难排查，而第二种则受限于硬件，扩展需要成本。所以我将第一种方法与定时器结合，让其，具体结构如下：

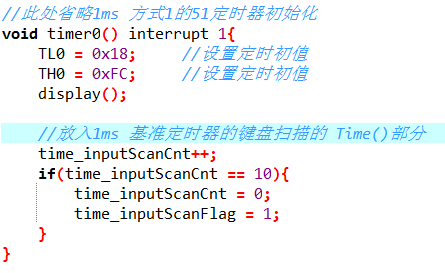


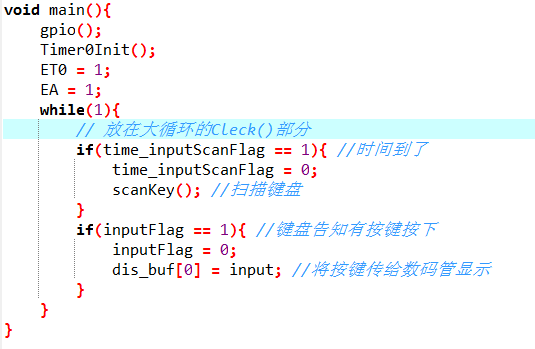
是从查询法的“直接检测事件条件，符合条件则执行”，变成了“在基准定时器（这个）中定时计数，到了一定时间后再检测事件条件，符合条件执行”，这样子增加了一个定时检测的功能，可以更好的符合某些需要定时检测的功能，如每半秒检测一次温度，每10ms检测一次键盘等。

为了能快速的写出定时查询法，我将其分成了两个部分，一个是Cleck部分，一个是Time部分。具体流程图如下：



以任务四的键盘为例子，具体实现代码如下：

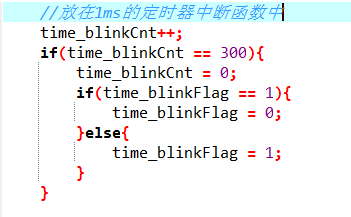




对于Cnt变量，如果使用uchar类型的话，不要超过255，不然会死循环。为了避免这种错误，推荐全部采用uint类型，在完成功能实现后再进行优化。

### 显示闪烁标志位的编写

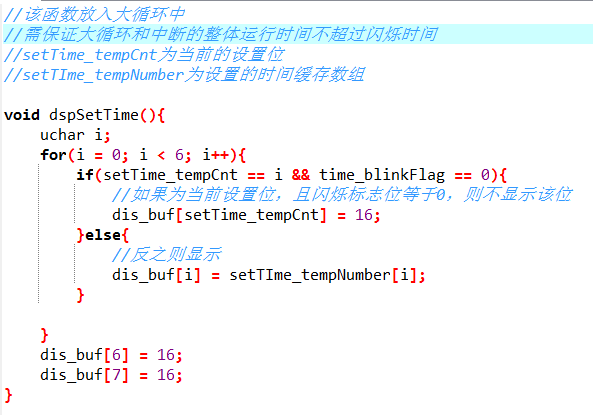
闪烁标志位我比较喜欢先提出来写，这样的话在显示方面就会省下很多事情，如下面的300ms闪烁位。



这样的话当其他c文件使用的时候只要extern一下就可以了。（更详细的有关C/H文件的写法在一些扩展这章中提到，可供参考）



之后刷新的过程中对显示缓存器进行闪烁标记就直接对这个标志位进行判断就可以了，以任务四中调时的闪烁标志位为例子，如下：

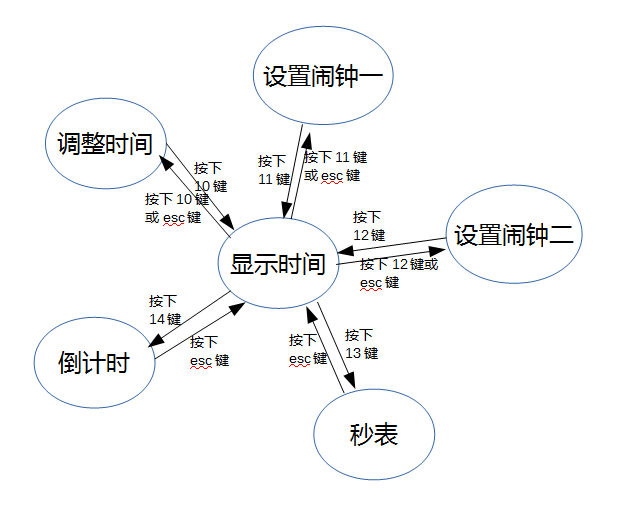


其中的设置位指的是当前设置的第几个位，而缓存数组则是暂时存放这些所设置的数字的数组，（因为用户可以选择不保存，所以不能直接写入时间中），关于更详细的解释，在之后的缓存器设计中会提到，以供参考。

## 将各个功能状态化

### 关于状态机的描述

状态机，就是将功能分解为各个状态，然后根据不同的模块执行不同的事件，比如任务四的状态转移图可以画成如下：



### 有关于状态机的个人改进

状态机有四个性质，现态，条件，动作，次态。

①现态：是指当前所处的状态。

②条件：又称为“事件”，当一个条件被满足，将会触发一个动作，或者执行一次状态的迁移。

③动作：条件满足后执行的动作。动作执行完毕后，可以迁移到新的状态，也可以仍旧保持原状态。动作不是必需的，当条件满足后，也可以不执行任何动作，直接迁移到新状态。

④次态：条件满足后要迁往的新状态。“次态”是相对于“现态”而言的，“次态”一旦被激活，就转变成新的“现态”了。

一般来讲，分析阶段，写出每个状态跳转的四个性质就可以了，不过实际编程之后，发现，如果将动作拆分成状态进入的初始化和状态退出的销毁函数，并写出状态之间的数据传递，会更有利于编程。因此在画出状态跳转图后，可以按照下图思路将状态细化：

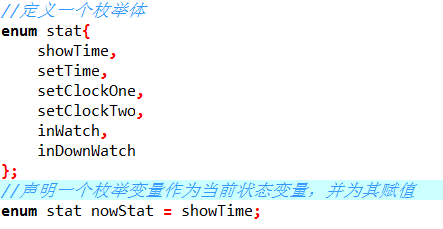


因为数据一般比较多，每一个状态都分析并画成图会很凌乱，可以将其列成表，以任务四为例子：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 现态 | 自身任务 | 跳转条件 | 次态 | 数据流与动作以及备注 |
| 显示时间 | 1显示时间  2检查闹钟1  3 检测闹钟2 | 按键10 | 调整时间 | 传入时间 |
| 按键11 | 调整闹钟一 |  |
| 按键12 | 调整闹钟二 |  |
| 按键13 | 秒表 |  |
| 按键14 | 倒计时 |  |
| 调整时间 | 通过键盘读入要调整的时间 | 按键10 | 显示时间 | 跳转后保存时间 |
| 按键ESC | 显示时间 | 不保存时间 |
| 调整闹钟一 | 通过键盘读入要调整的闹钟一 | 按键11 | 显示时间 | 保存闹钟 |
| 按键ESC | 显示时间 | 不保存闹钟 |
| 调整闹钟二 | 通过键盘读入要调整的闹钟二 | 按键12 | 显示时间 | 保存闹钟 |
| 按键ESC | 显示时间 | 不保存闹钟 |
| 秒表 | 按下12键开启秒表计时，反之暂停 | 按键ESC | 显示时间 |  |
| 倒计时 | 先通过键盘读入数字，然后按下后启动倒计时，时间到响铃 | 按键ESC | 显示时间 |  |

### 有限状态机的具体代码实现

对于当前状态的表示，我们除了传统的使用uchar变量来表示，还可以使用枚举来表示，这样的好处就在于，不用写过多的注释也可以让程序变得更易读，以任务四为例子，状态的表示如下：



接着，我将每个状态根据状态表分解成6种函数，分别是

*initStat()函数，用于当前状态初始化，在每次进入该状态时候执行一次。*

*jumpStat()函数，用于判断输入并根据输入跳转次态，当检测到输入后执行一次。*

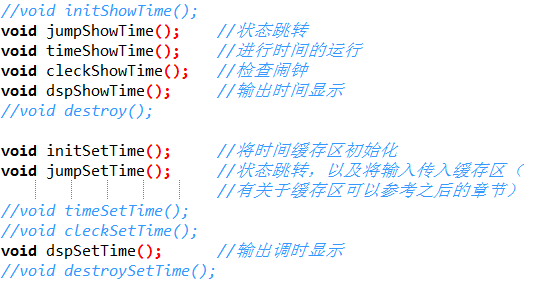
*timeStat()函数，用于放入基准定时器的定时任务（参考之前的定时查询法）*

*cleckStat()函数，用于大循环中检查标志位或者检测警报的监控任务（参考之前的定时查询法）*

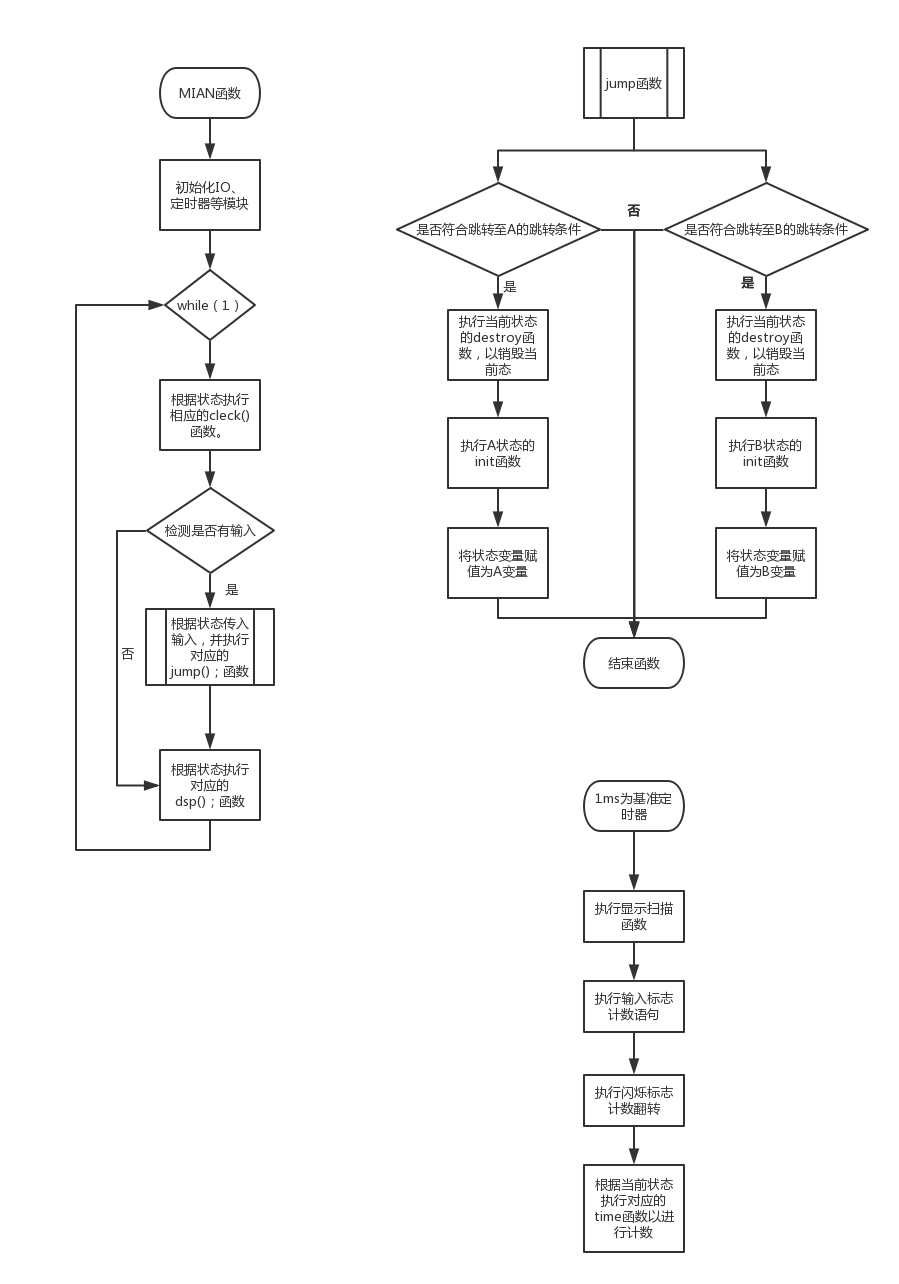
*dspStat()函数，用于修改显示缓冲区，以输出显示。*

*destroyStat()函数，用于退出当前状态的时候消除相关变量以防干扰到其他状态。*

并不是所有状态都需要实现所有的函数，应该根据实际情况实现对应的函数功能。以显示时间为例子，实现了其中四个函数，就足以满足其功能的实现，而调时功能只需要实现三个函数。

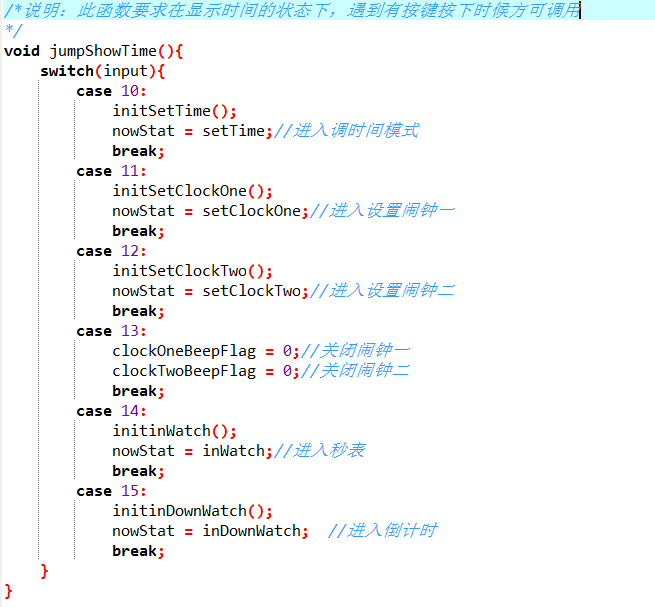


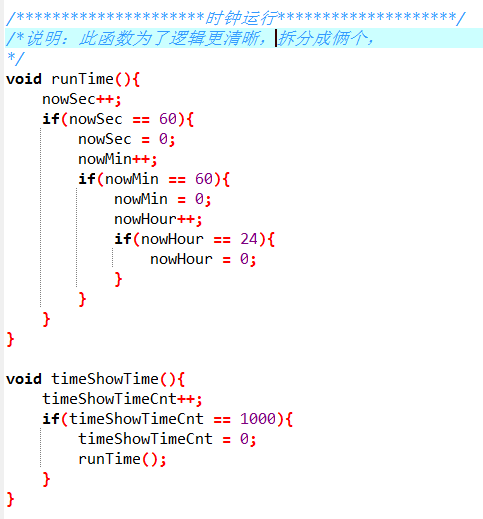
至此我们的主函数如下：

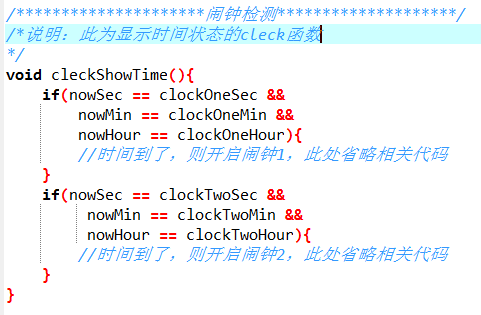


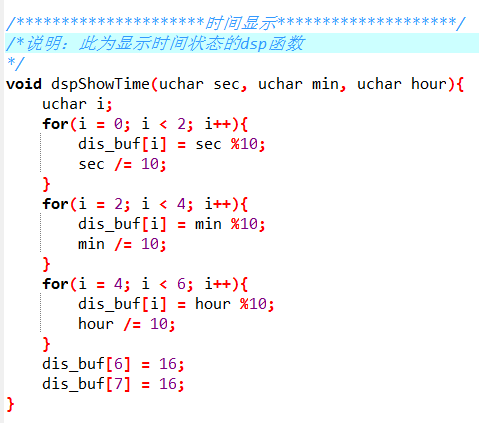
### 编写任务四中的显示时间模块和秒表模块

以显示时间为例子，实现四个函数代码如下：



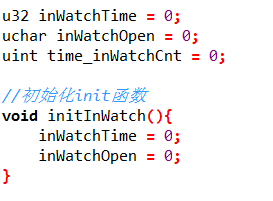


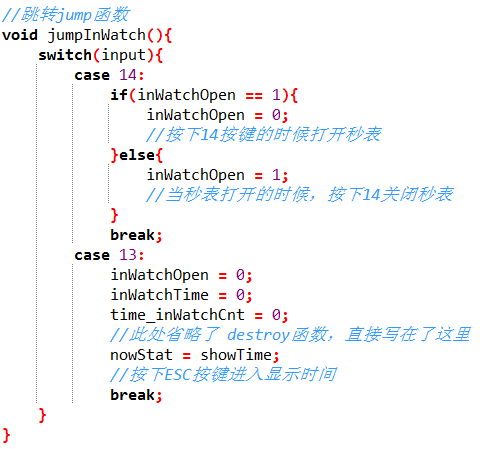


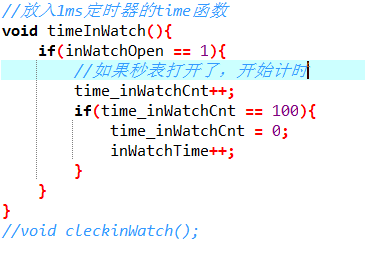


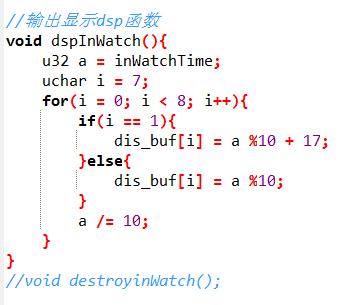
写出这四个函数后，只要在定时器里跑time函数，然后在大循环里头不断的跑cleck和dsp函数就可以实现时钟显示功能，再加上当有输入的时候执行jump函数就可以跳转到其他功能了。

同理可得到秒表的代码：

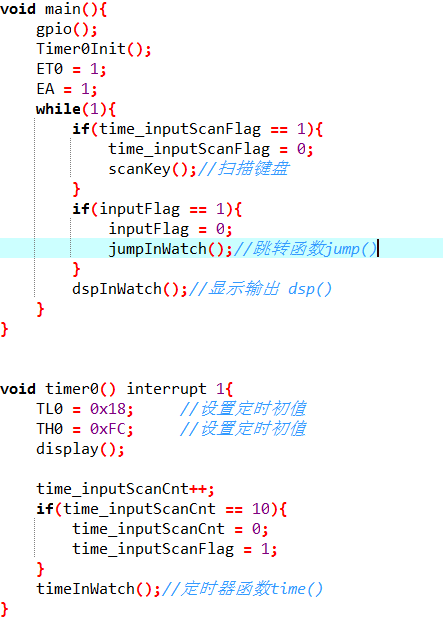








如果要进行测试，可以在main函数和定时器中如下调用就能看到功能的大致实现了：



### 这种方法的优缺点

需要控制好中断以及主函数之间的运行的运行时间，以及在外接芯片的时候，需要控制好实训问题，在之后注意事项一章有说明，可供参考

## 为需要输入缓存的功能编写缓存器

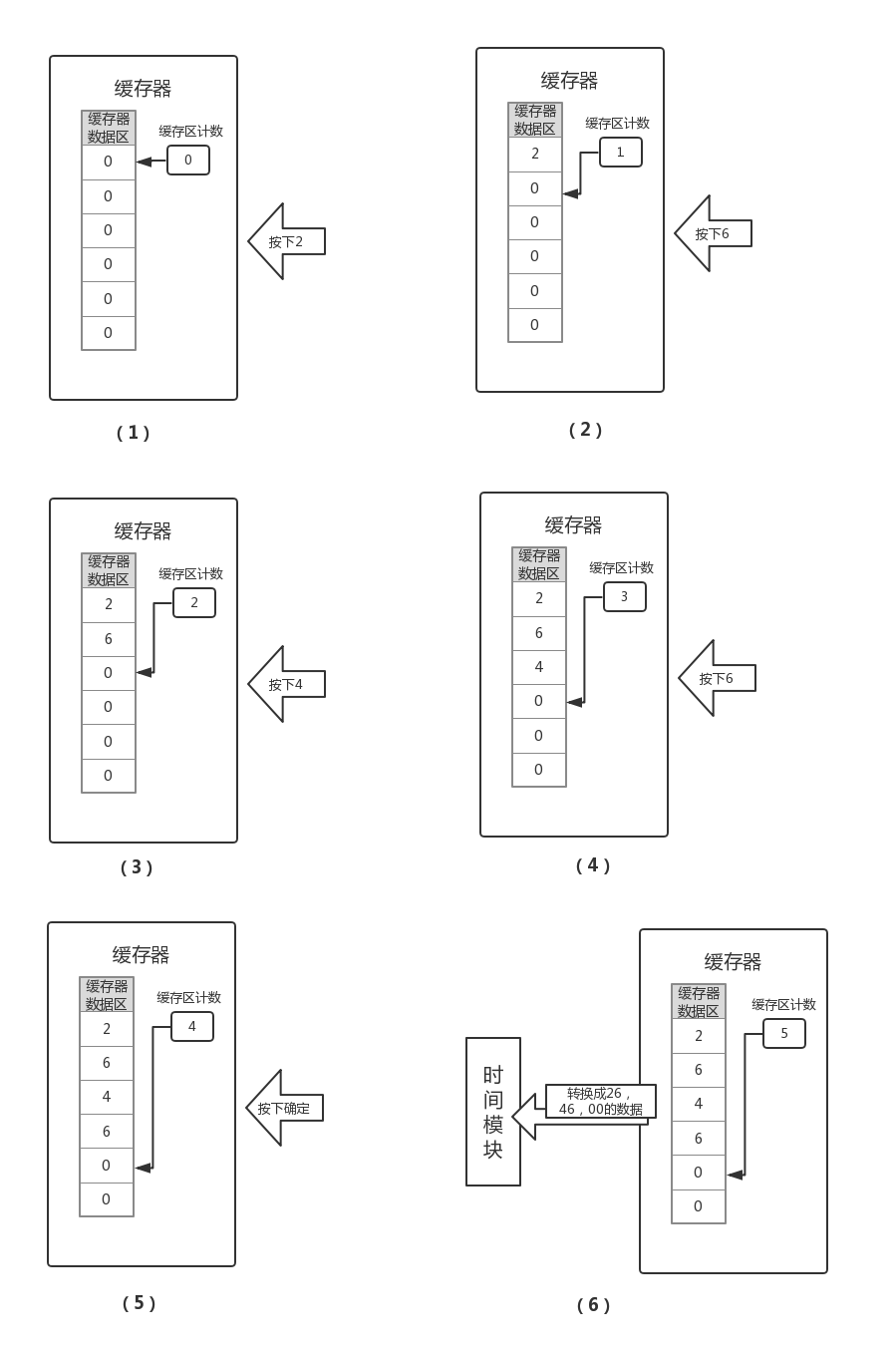
### 简述缓存器

缓存器的作用，就是将输入先存起来，等需要调用的时候，再全部拿出来，比如C语言的scanf函数执行的时候，他会先从键盘读取键值保存到缓存区，直到敲入回车再传入程序处理。而在单片机中，也是经常预先将输入的数据暂时的存起来，等到需要的时候再正式的传入功能模块，因此，在编程过程中，单独的把输入暂存独立出来作为一个缓存器，能提升逻辑性和可读性。

以任务四为例子，其中调整时间，闹钟一，闹钟二可以很明显的看出是可以使用同一个缓存器，等用户输入完数据按下确定键，再将缓存器中的数据赋值给对应状态下的变量就可以了，同样的道理，我们也可以为倒计时再设计一个缓存器。

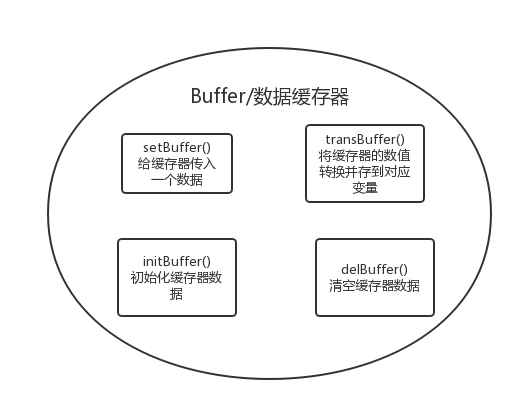
### 缓存器的基本工作原理

缓存器的数据结构则一般由缓冲区域（用数组表示）和缓存计数（缓存）俩个部分组成，以任务四中的时间输入缓存器为例子，其具体工作过程如下：

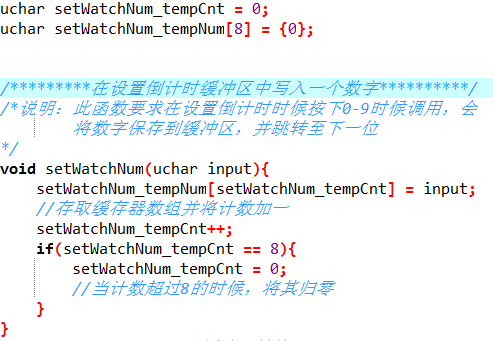


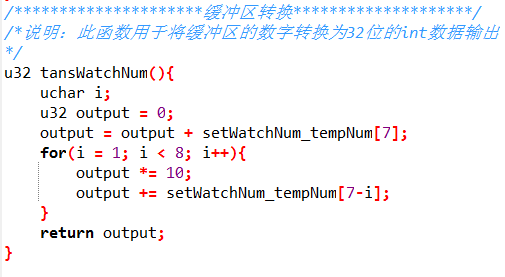
### 自己总结的有关于缓存器的基本构成

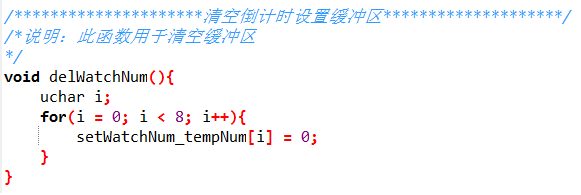
根据个人的一些经验总结，缓存器基本由以下四个部分组成：



与之前的状态模型一样，数据缓存器中的函数不需要全部实现，应当根据具体的情况实现所需要的函数。以倒计时输入缓存器为例子，其所实现的代码如下：







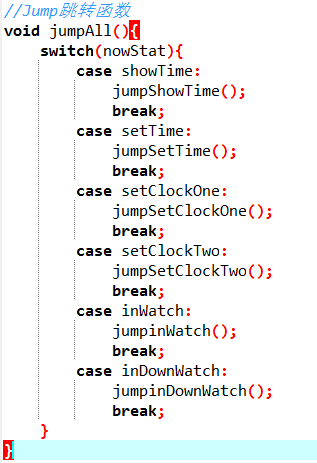
另外，在之前的dsp函数中，也可以通过访问缓存数据和缓存计数来实现当前输入的显示。

## 将功能模块组合起来

### 使用switch语句来判断状态

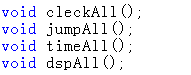
之前提到，各个状态可分解为六个函数，包括init初始化，jump跳转，cleck监控任务，time定时任务，dsp输出任务，destroy销毁。

在实际编程中，根据状态来运行对应状态的函数可以使用switch语句，实现对应的jump选择代码如下：



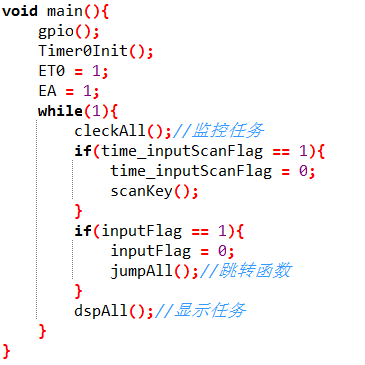
其他状态函数类似。

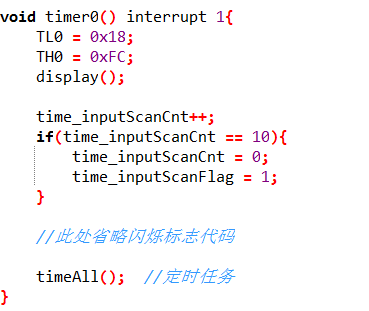
因为init初始化函数和destroy销毁函数是在jump函数中进行调用，所以要写的选择函数只有四个：



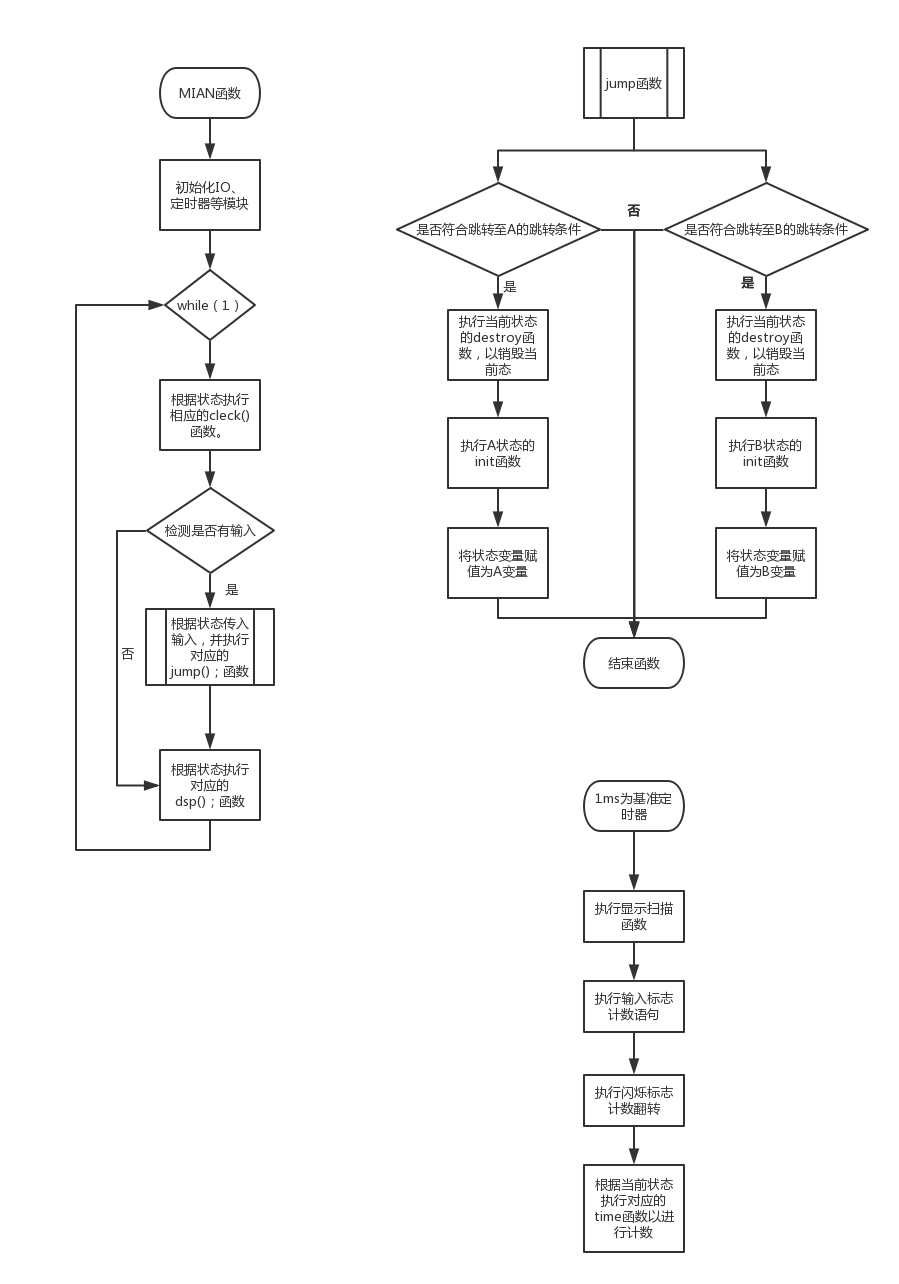
其中，cleckAll在大循环中调用，jumpAll在检测到输入时候调用，timeAll放入1ms的定时器中，而dspAll同样也是在大循环中调用。

这样就可以得到如下代码：





而init初始化函数和destroy销毁函数则在jump函数中，其流程图如下：

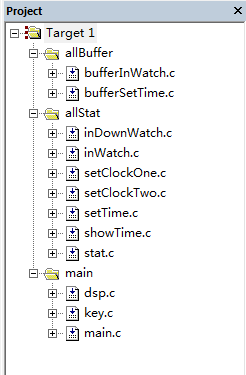


至此，我们的功能就整合完成了。

## 让程序更易读

### 分离c/h文件

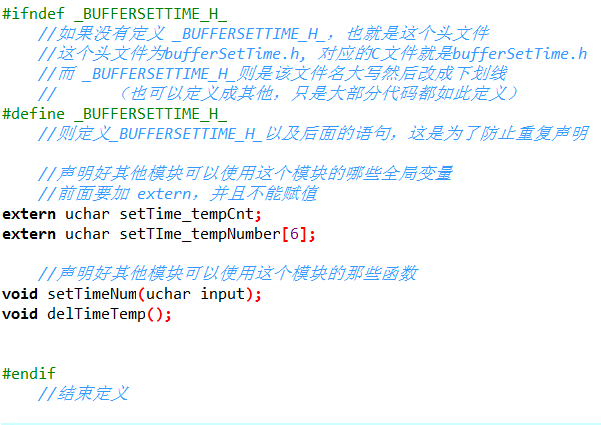
分离C/H文件后，我们可以把各个模块独立成一个个C文件，C文件之间相互调用则只要include对方的头文件就可以了，通过C/H结构，可以让项目更加清晰，如图：



在编写的时候，应当遵循一下的规范：

①、C文件中的要求不多，只要不进行函数声明直接编写函数即可，另外，如果该文件中的变量要被其他C文件调用，注意要把变量改为全局变量。

②、H文件中需要写入#ifnfer #define #endif语句，并声明对应的C文件中可以被其他程序调用的函数和变量，以时间缓存器的头文件为例子，其代码如下。



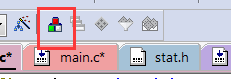
这样如果其他C文件要使用时间缓存器，就只要加入以下语句：

#include <bufferSetTime.h>

就可以使用头文件声明的变量和函数了。

### 文件结构化

在分类完文件后，我们点击一下按钮可以把C文件分类一下。



## 一些小的推荐扩展

以下扩展均没有经过代码验证，纯粹只是想法而已，仅供参考

### 去除延时

任务四中唯一有延时的只有键盘的抖动检测，可以通过状态机的形式进行去除，每10ms检测一次键盘并进行状态判断。可分为四个状态，状态表如下。

未按下 → 按下未消抖 → 按下已经消抖 → 持续按下中

### 空闲模式

去除延时后可以发现主函数大部分时间都是在检测标志位，但是我们的标志位每1ms才可能置位，而1ms对于51来说，可以执行1000个指令，而STC15则可以执行一至两万个指令，很明显我们的主函数远远达不到这个量，因此我们可以在主函数检测完一次标志位后进入空闲模式，然后等待定时器将标志位置位并唤醒，在一些电池供电的系统中可以节省不少功耗。

## 注意事项的思考

### 每个过程所需的时间考虑

因为程序分成了主函数和时间函数俩个部分，因此控制好这两个部分的时间就需要自行发挥了。一般来说主函数最大运行时间超过基准时间的话，会让数据无法得到处理，所以，当需要实时处理的时候，主函数和中断函数的最大运行时间的总和不能超过基准时间。日常应用并不需要实时处理的话，只要中断函数不要过长堵塞主函数的话，主函数的时间是不限制的，不过数据处理会拖慢，也要进行适当的控制。

### 时序的控制

当系统需要读取外部芯片，并且中断时间干预到通信时序的时候，我们就需要对定时器进行人工控制了。

一般来说一个uint类型的Cnt变量进行计数标志在编译后，所需的汇编指令约10条，加上一些其他条件的判断，我们可以以30条指令作为一个计数标志在中断函数所需的资源。其在51中需要的时间为30us，而stc15保守估计快12倍，则为3us左右，加上对应的display函数，可以估算STC15中一个time中断最多占100us。

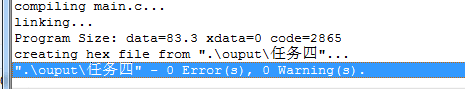
因此，如果时序图中部分时间并非要求精确到百、十位级别的us时序，则无需进行控制，否则则需要在驱动的相应位置加上ET0 = 0和ET0 = 1来关闭中断以防干扰。

另外，因为刷新函数预留不闪烁时间约1ms，所以关闭时间不能过长，否则会有闪烁感。

## 整体系统的优缺点

### 优点

* + - 1. 整体可读性很高，并且逻辑较为清晰，可以适用大部分简单输入输出的系统中
      2. 代码编写很有规律，或许可以通过JAVA等对字符串处理能力高的高级语言进行软件直接输出，让开发更为快速
      3. 在保证规范的情况下，未做优化的文件占用内存为81B，文件大小为2865B，与传统的结构，基本无多少资源消耗。



### 缺点

* + - 1. 对于多状态嵌套或者多状态共存的系统无法适用，需要做大规模改动
      2. 代码有一定的沉余，在flash内存苛刻的情况下无法适用
      3. 对于有实时时间要求的系统或者处理量过大的系统无法适用
      4. 对于外接有严格时序且时序时间长的芯片会出现中断干扰的情况。

# 实训心得

在这次实训中，自己有调理的总结了这一年有关于51单片机编程的知识与新的，对单片机有了更深刻的认识，并且从中也发现了许多的问题，通过对多任务的开发，明白了许多单纯看书无法理解的知识，今后只有多自己动手操作，才能更好的学习单片机。

——2017年1月15日

1. 所谓“黑箱”，就是指那些既不能打开，又不能从外部直接观察其内部状态的系统，比如人们的大脑只能通过信息的输入输出来确定其结构和参数。可参考百度百科。 [↑](#footnote-ref-1)
2. 显示缓存区，用于存放要显示出来的数据，即uchar dis\_buf[]={0,16,16,16,16,16,16,16}; [↑](#footnote-ref-2)