# 四轴 DIY 小结

徐江　　cnmusic@163.net

## 一、 概述

四轴可以说是机械结构最简单的飞行器了，而且自己做起来也不是很难。维护起来成本也比一般航模要低不少，所以我就花了差不多 6 个月的时间摸索着做了 2 个来玩。

这篇文章就是一个大概的记录，希望对后来人能有些帮助。由于不少都是自己摸索的，难免有不少错误，所以“ **仅供参考** ”！☺

在这里要感谢网友 feng_matrix 对我的帮助，通过和他的交流让我少走了不少弯路。

## 二、 马达、电调、桨的选则

在我第一次选择马达的时候，我选择的是有刷马达。原因很简单，不需要复杂的电调，直接用 MOS 管就可以驱动了，而且响应速度又快，价格又便宜。可惜没有买到合适的有刷马达，只好用减速组配高转速马达。这样一来成本反而高了，而且实际的测试结果是马达里面火化直冒也无法将四轴自身拉离地面。原因就是马达转速和减速组搭配不合理，转速过快但拉力不够。

经历过失败后，决定不在冒险，于是选择了大众配置：新西达 2212，1000KV 外转子无刷马达，新西达 30A 电调（好赢兼容的程序），在解决了如何安装的问题后，终于可以将四轴自身拉离地面了。

对于桨，由于条件所限，只能在淘宝上买到 GWS 三叶正反桨。我测试的结果是 10 英寸桨最结实，因为它最大，最重，带来的结果就是低转速。优点就是抗撞击。一般的 9 英寸桨稍微碰到一点东西就断了，而 10 英寸桨一点事没有。以前担心 10 英寸桨可能引起响应时间过长造成四轴无法稳定，后来发现真正影响响应时间的是电调，桨的关系倒不是很大。当然这不是说 10 英寸桨就是金刚不坏之躯，只是比 9 英寸桨要结实一些罢了。

### 用商品化电调还是 I2C 电调？

我一开始的四轴采用的就是商品电调，原因很简单，自己焊 I2C 电调多麻烦啊，还是用买的现成的省事。但随着后面深入做下去，发现这 2 种电调的差异还是很大的。

对于我开始的商品电调，由于里面自带的 PID 控制器。严重影响了转速的快速反应。这就造成了对于四轴稳定性之一的“ 自动悬停 ”基本无法做到了。由于自动悬停的首要要求就是在四轴倾斜时能够在最短时间内自动回到水平位置，这就要求马达转速在四轴有倾斜时需要加快，而到快回到平衡位置时需要降下来。商品电调里的 PID 恰好阻止了这个的发生。转速上去可以，想要马上降下来？没门！我试验的结果就是一旦转速上去了，想要降下来，等 0.2 秒吧。即使这段时间里通知电调降低转速也没效果。带来的后果就是四轴一旦倾斜后就会不可控地晃来晃去，好像抬轿子。

这并不是说不能用商品电调飞四轴。完全可以飞，而且也可以飞的很稳。从旁观者眼里看这 2 者区别并不是很大，而对操作者来说，差别是相当大的。商品电调的四轴基本好像是当年赖特兄弟发明的飞机，所有操作都要操作者来处理，包括方向飞行后的机身偏转修正。而用 I2C 电调的四轴就好像 F22，相当智能，操作起来很容易。

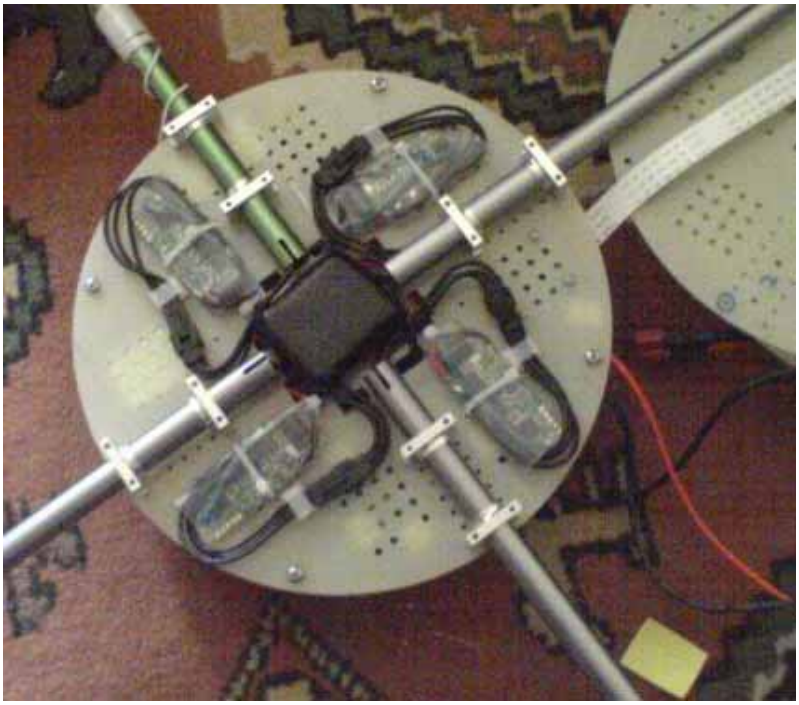
对 PI 参数来说，商品电调和 I2C 电调主要差别在 I 参数上。商品电调的 I 参数不能大，否则就会变成一个“ 飞行轿子 ”，而 I2C 电调就没有这个问题，它的高响应速度可以适应大转速变化。

# 三、 机架设计

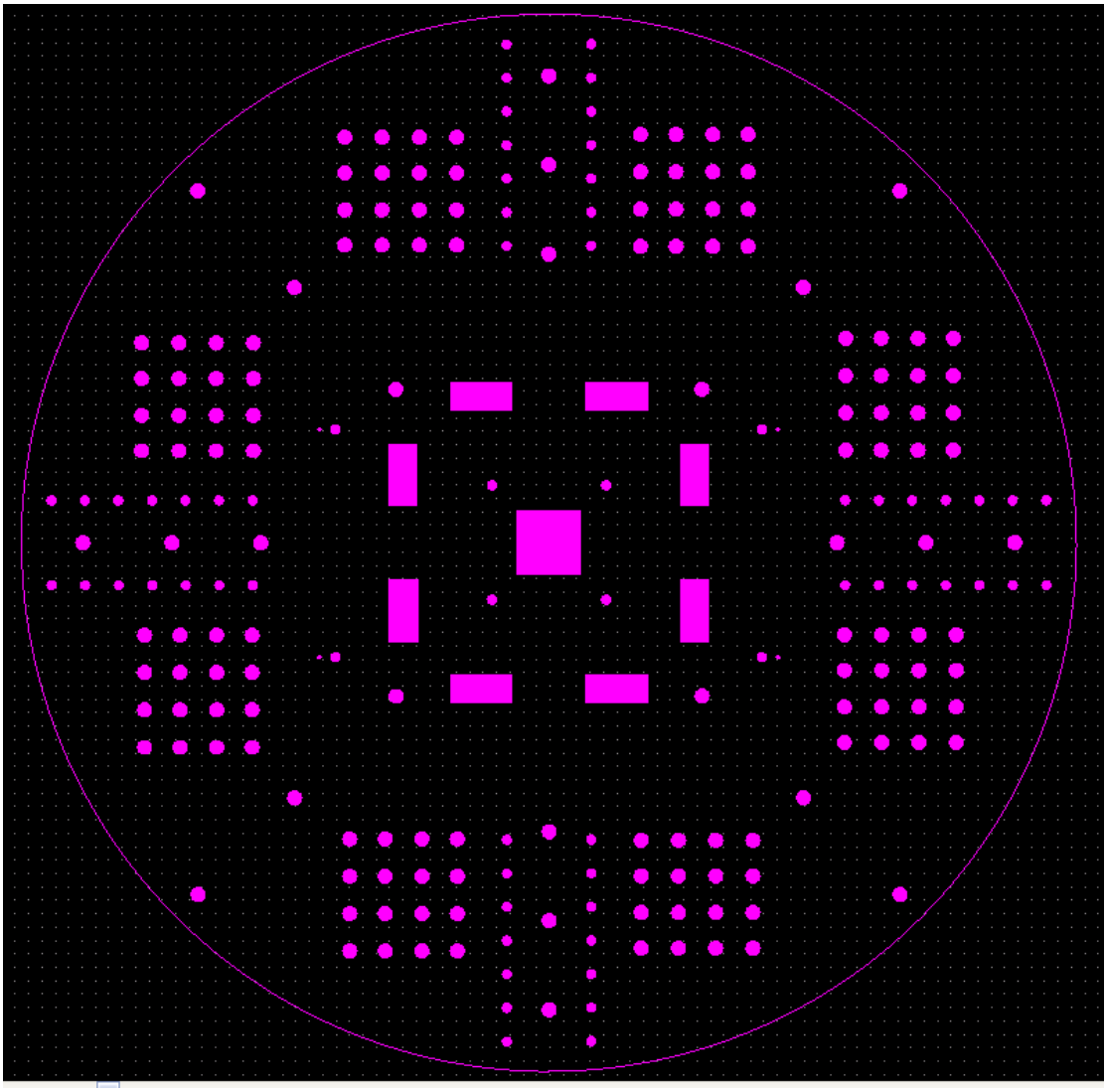
对于像我这种没有机械加工条件，本身又没有任何机械加工经验的人来说，最小加工程度是我对机架的基本要求之一。我的第一架四轴就是按照这个目的设计的。

利用 450 电直的尾管做四轴的轴，用 450 尾管固定座来固定，包括马达的固定。然后利用波纤板将四个轴整合到一起就完成了。

最后就是这个样子



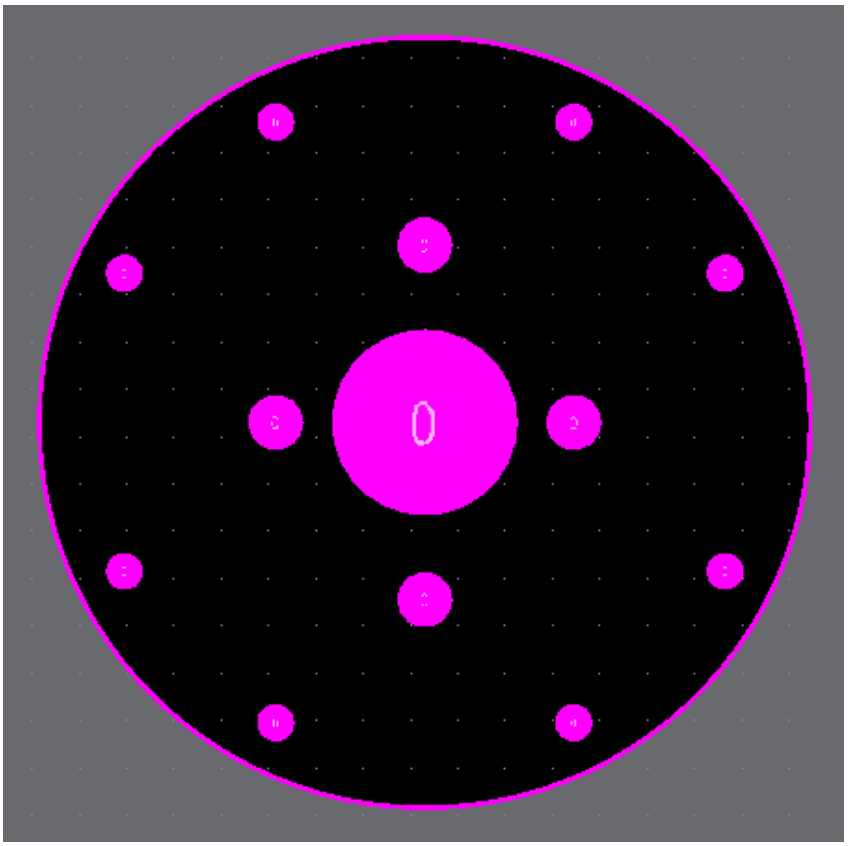
中间的大板就是玻纤板，说更简单一点就是我送去加工的一块电路板。



这就是原始的设计稿，只有 KeepOutLayer 上的打孔和开槽。拿回来后所需要我做的就是拧螺丝。

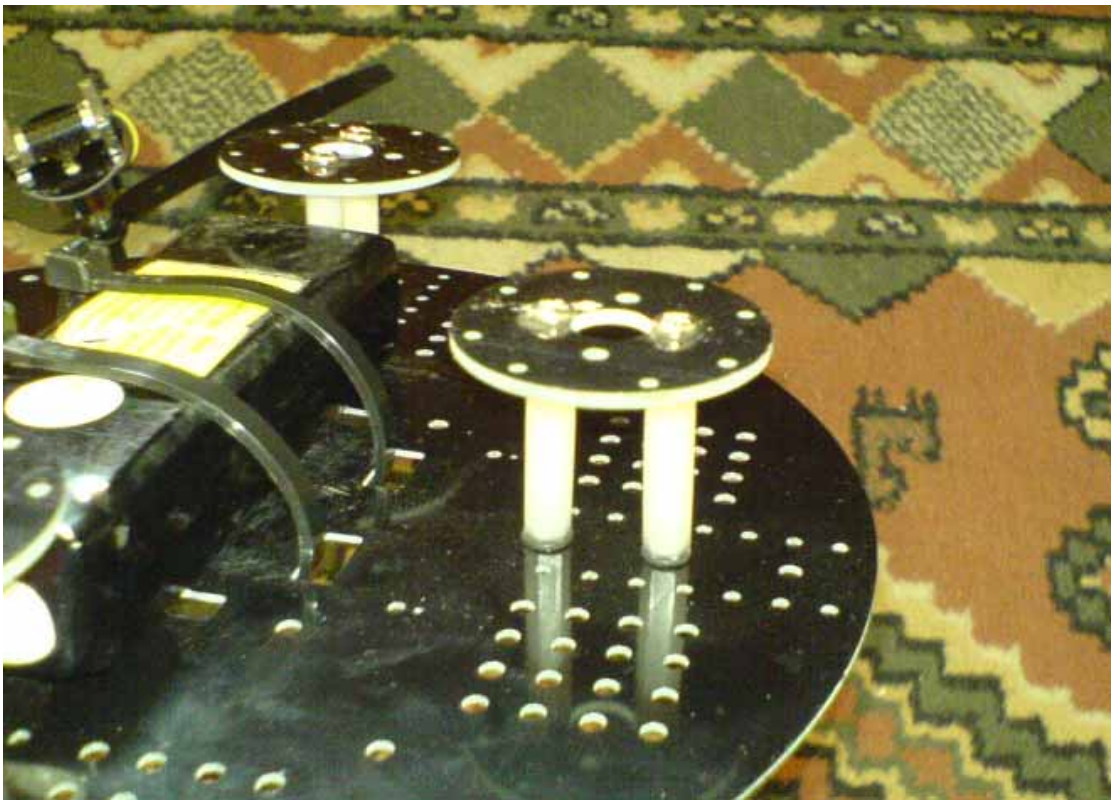
对于马达部分的固定，我也采用了电路板+尾管固定座的方法。

同样用电路板做了一个马达座，这样比单纯用铝的马达固定座强度大了不止一点。



这是原始设计图，照例还是电路板。

对于脚架的固定，我也用了这种板。通常的塑料 450 脚架虽然看起来漂亮、重量轻，但强度不够。但水平速度很大的情况下滑翔着地时，基本都会断掉。



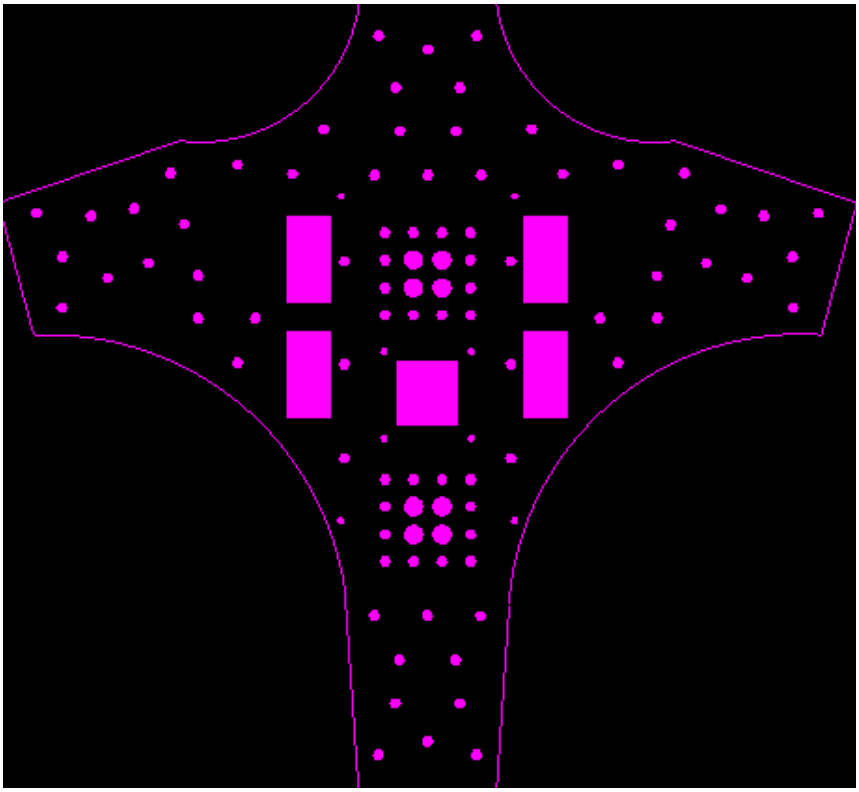
所以我用塑料柱加马达座来当脚架，到目前为止还一个都没坏掉呢。

整合到一起就是这个样子

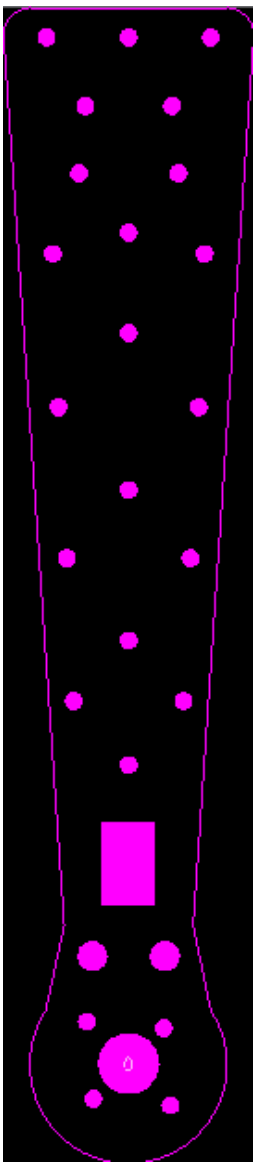
不过这种机架的问题不少，最最严重的问题就是那个 450 碳纤尾杆非常容易断掉。用铝杆的话长时间冲击又会弯掉。为了修理这个也花了我好多钱呢。

另外一个问题就是重量，整个四轴加上电池已经 1.2 公斤了，带来的问题就是续航时间太短，很快就飞不动了。

于是下一个机架又开始设计了，这会依旧用电路板来做。整个机架分为 2 部分。

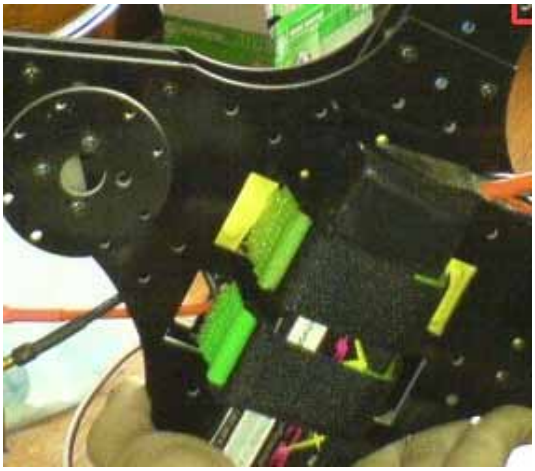


和



2 片上面的大“ 衬衣 ”组成中间的机架，下面的 2 片小的组成一个轴，一共需要 8 片小的才能组成完整的一架。就是这个样子：

最后所有东西都装好后，算上电池大约 900 克，至少比以前的轻了。而且这些电路板加起来的总成本也比以前用 450 尾管固定座要便宜不少，大约￥150 左右就都搞定了。



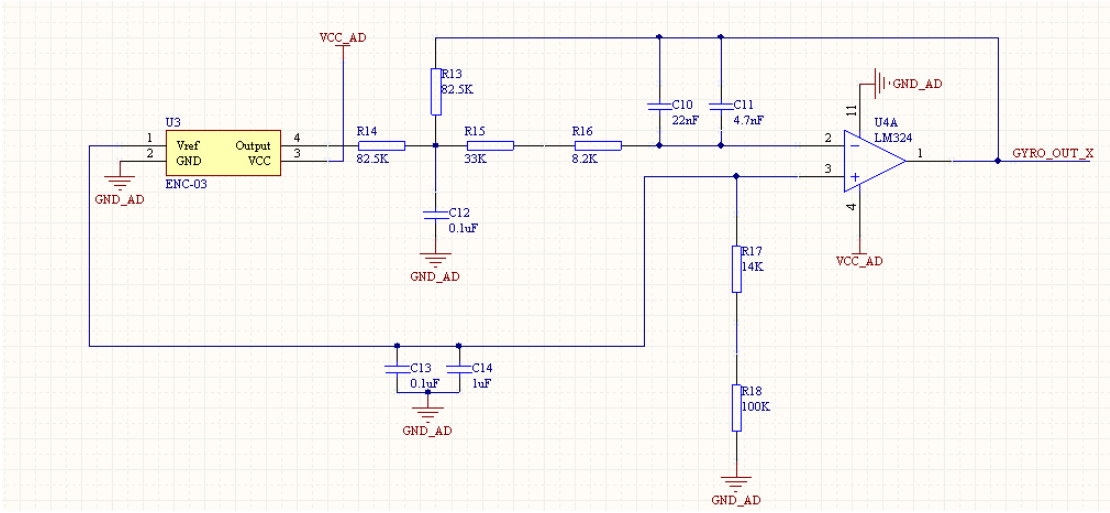
电池固定采用了更结实的方法，电池里面还用魔术贴粘到机架上了。起落架依旧使用经过考验的马达固定座，而且是 3 个塑料柱组成一个三角形，抗水平冲击强度更大。这也是现在的固定座的唯一用途了。

# 四、 硬件电路设计

在硬件上，我采用了一个单独的 Mega8 处理器对遥控器的输入舵量进行解析，M8 的 IO 口数量可以做到 8 路输入和 8 路输出。姿态控制我用了一个 ARM7 核心的 AT91SAM7S256，最

高 55Mhz 的主频，有 64KB 的内存、256KB 的 Flash 可以供我“ 挥霍 ”，这样我就可以将注意力集中到算法上而不比担心资源不够用了。这 2 个处理器利用 I2C 总线通讯，在比较关键的诸如舵量数据等都加了 CRC16 校验。

对于陀螺仪传感器，我采用了下面的滤波电路。这个电路没有放大，只有滤波。

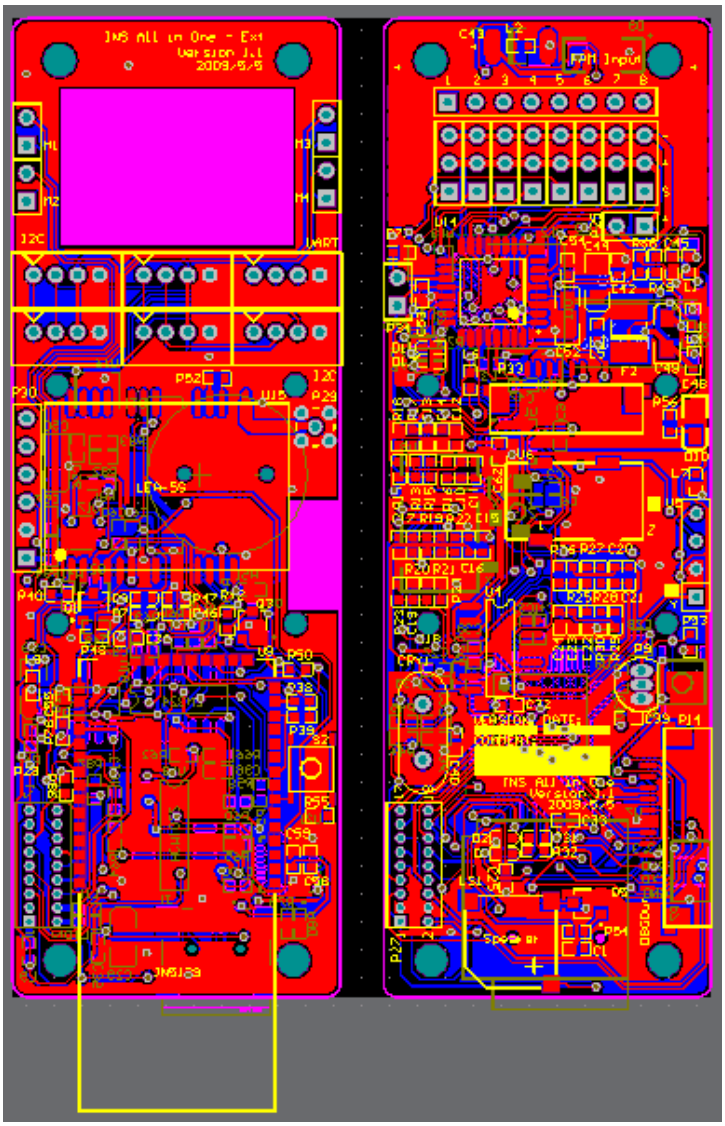



而加速度计我就直接并上一个高频滤波电容直接接到处理器的 AD 上了。

为了便于分析，我使用了一个 MicroSD 卡做飞行数据记录。利用 AT91SAM7S256 的 SPI 口和它进行通讯，同时移植了 EFSL 文件系统，这样就可以按照文件形式一个文件一个文件地记录下去了。不过用这个文件系统最麻烦的地方就是断电的时候，由于 EFSL 的缓存的关系，如果没有在断电前将数据写回 SD 卡，有可能造成数据丢失。这里我的方法是在没有起飞前间歇性地将文件关闭，SD 卡卸载，然后再重新装载。在起飞后就不必这么麻烦了，就直接一直写下去就可以了。这种策略兼顾了方便性同时最大限度避免文件损坏。

由于有了 SD 卡，对于 M8 上的程序升级我也转成了 I2C 方式。通过一个小巧的 I2C BootLoader 直接把 SD 卡上的 BIN 文件传输上去，非常方便了。

在第一版四轴上，我采用了分板的设计。PPM 解码/输出一个，陀螺仪一个，加速度计一个，姿态处理器一个，SD 卡和电源一个。分了好多个板，当时为了测试各种安装位置，看到地要怎么装才是最佳组合。后来被这些板之间的连线搞到头大，经常因为线接触不良引起莫名其妙的故障。最后又回归单板设计，处理器，SD 卡，PPM 编码/解码，3 轴加速度计/3 轴陀螺仪，温度传感器，包括一个电源分线器作为一个板。GPS、气压传感器、电子罗盘、ZigBee 无线模块等作为另外一块板。2 个板之间用 1.27 间距的插针连接，如果调试完成后可以直接焊到一起。这样就避免了多余电缆的接触不良问题，实践检验效果还是很好的。



## 五、 软件设计

相比硬件，软件才是四轴当中的最大难点。

对于 PPM 编码/解码的部分相对简单，我采用了 M8 的 16 位定时器，在 ICP 引脚收到 PPM 波形的时候开始计时，一直记录到波形结束，记录下当时的计时器数值，然后切换到下一个通道，没有去管那些前 1.5ms 的引导波形。输出也是如此，只是为了加快输出速度，我一下子输出 8 个通道，利用硬件定时器计数来判断哪个通道的输出可以结束了。这样整体输出速度最大可以达到 200Hz，这个频率一般舵机是受不了的。不过新西达的商品电调确可以接受这个频率，工作正常不会有任何问题。

这部分的麻烦之处是要兼顾输入舵量的解析速度和 I2C 的响应。由于我没有将所有通道都合并到 ICP 引脚上，所以要想采集完完整 8 个通道需要至少 1.5ms * 8 = 12ms，最多可能有 (1.5+2)ms * 8 = 28ms 的时间。而且有些接收机输出的速度达到 63Hz，算下来基本上 M8 都

会忙于解析 PPM 信号而无暇顾及 I2C 了。这里我的方法是采集完后只有等到数据被取走后才会再次解析一遍，这样就可以给 M8 足够的时间来处理别的事情了。

在姿态控制算法里，我只用到了标准的 PI 和加权移动平均这 2 种算法，没有采用卡尔曼滤波或别的“ 高深 ”算法。具体流程如下：

首先利用处理器的 AD 功能，将陀螺仪、加速度计的数值读取近来。由于加速度计对振动非常敏感，所以我这里对加速度计做了一个移动期为 10 的加权移动平均，用来过滤掉部分振动。由于我的四轴对这些数据的采集是工作在 300Hz 的频率下，所以移动期为 10 对我的四轴来说只相当于延迟 33ms，再加上适当的加权，基本可以消除由于加权移动平均带来的响应滞后问题。

对于陀螺仪，由于这个比较重要，所以我没有做多余的处理，就把 AD 的数据直接拿来用了。以前也测试过移动期为 2 的情况下，也没有问题。

下面对每个轴的陀螺仪做 PI，就可以得到马达修正量。将这个量加上油门后输出给马达，就完成姿态调整任务了。

不过这里就有一个“ 隐患 ”。由于陀螺仪有漂移，也就是以恒定速度右转 90 度，然后再左转 90 度后，陀螺仪积分不会为 0。如下图所示：


可以看到在四轴最后回到平衡位置时，陀螺仪积分并不为 0。这个偏差在所有陀螺仪里都存在，在 ENC-03 上更加厉害一些，所以需要对陀螺仪的积分数据进行适当处理后才能使用，否则积分会越积越大。这就是下面要讨论的。

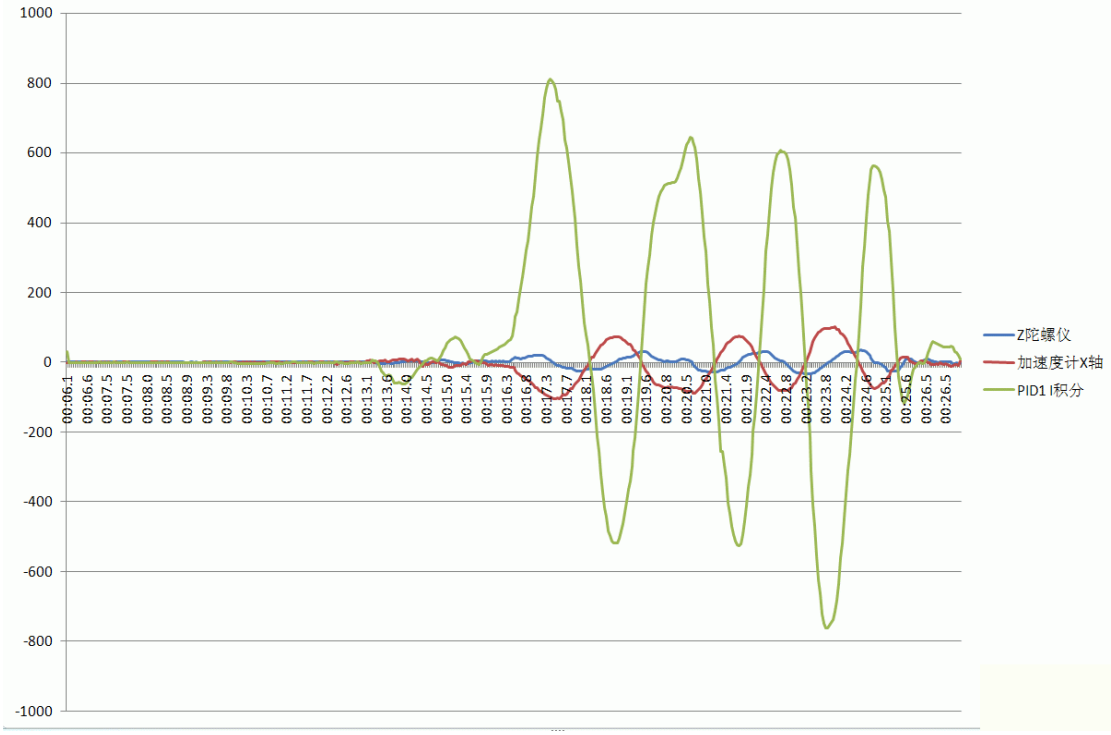
# 如何将加速度计的数据融合进陀螺仪积分里

根据上面的说明，已经可以大概了解为什么要对陀螺仪积分进行处理了。但这里的陀螺仪积分还有另一个重要的物理意义，那就是这个积分记录了四轴偏转的绝对角度（虽然不是直接的角度，但是只要乘一个系数就可以变角度了），而不再是角速度。为了能做到四轴的稳定悬停，这个角度的重要性不可小看。

通常的做法，是利用加速度计一个轴的偏转和陀螺仪积分建立对照关系，通过试验测出一个系数。然后程序不断通过加速度计的偏转计算得出陀螺仪应该的积分大小，根据这个数值对陀螺仪积分进行修正。当然这个修正也有讲究，决不能直接替换掉陀螺仪的积分。通常都是将通过加速度计得出的积分“猜测值”缩小，限幅后加入陀螺仪的积分里。过大或过小都可能造成问题，须要反复试验才能得出合理值。

在我的四轴里，我测出的陀螺仪和加速度计的对照关系是：当加速度计偏差=94 时，陀螺仪积分 = -696。这是 100Hz 积分频率，加速度计数据放大 6 倍，3.3V 参考电压下得出的。这个系数决定了加速度计积分猜测值的“斜率”大小。然后我把这个计算值缩小 22 倍，然后限制幅度为正负 5，最后才将数据加入陀螺仪积分里。

在加入了积分修正代码后，陀螺仪积分就可以正确收敛了。




更高级的做法是利用卡尔曼滤波将加速度计和陀螺仪数据进行融合，得出绝对角度，以次来进行姿态纠正。不过卡尔曼滤波比较复杂，数据延迟的问题我一直没能很好地解决，所以我的四轴里没有使用这种方法。

## 如何将遥控器的控制信号融合进来。

对于刚完成部分代码的四轴来说，想要自主飞行是不可能的，最简单的方法就是利用遥控器进行操作。

对于方向控制，基本都是通过直接增加对应马达的转速来实现。比如在我的四轴里，我将遥控器的输入信号影射到-1 到 1 之间，用一个浮点表示。比如我假设对于方向舵输入为正 1，也就是 100%前进舵量的情况下，对应陀螺仪偏转 12。我将这个数值直接加入陀螺仪 AD 后的数据当中，这样一来，就可以直接利用 P 参数实现方向控制，不需要单独再写代码了。

比较不同的是在遥控器操作时对于陀螺仪积分的处理。有的方法是在遥控器输入时完全不去理会积分，只工作在 P 模式下。这种得好处是可以避免方向操作时可能引起的震荡。但有可能引起方向过渡倾斜。这是我目前采用的方法。

另外一种方法是一直使用积分数据，不管有没有遥控器操作。这样可以保证四轴自身不会倾斜太厉害，由于四轴必须倾斜才能实现前后左右飞行，这种方法带来的影响就是飞行速度不会太快，而且如果参数不合适，非常容易引起震荡，因为积分的作用是总想要四轴会到水平位置，不达目的誓不罢休。

## 如何调整 PI 参数

在有了这些基本的代码后，四轴基本已经可以飞了。下面就是要找出合适的 P 和 I 参数。

对于 PID 控制器，P 参数的意思相当于对一个瞬间变化的硬性反应程度。在我测试时，首先只设置一个 P 参数，I 参数=0，然后拿住四轴对角位置的两个轴，旋转四轴，当我感觉我使用的力基本上 80%都被马达加速产生的升力抵消时，这时的 P 参数就差不多可以了。那种感觉就好像四轴下面有一个气垫把四轴托住了。

对于 I 参数的调整，我是这么做的。首先只设置 I 参数，P=0，然后倾斜四轴一边。由于 I 积分的作用，四轴会加快马达转速来抵消倾斜，这种调整是一直持续的，不像 P 参数那样只有旋转的时候才有反作用力。

逐渐调整 I 参数，感觉到倾斜比较小的角度的时候四轴已经可以提供一个比较大的力往平衡位置拉的时候，这个 I 就差不多了。过小的 I 可能引起四轴自动会到水平位置时间过长，过大的 I 可能引起四轴在一个很小的倾斜时就大马达转速修正引起震荡。

下面就需要通过试飞来不断微调这 2 个参数。我一般先固定一个参数，然后调整另一个参数。比如我固定 I 不变，调整 P。由于 P 过大或者过小都可能引起振荡，所以我就以 50 为单位进行调整。先减小 50，看振荡幅度有没有变化，如果不减还勉强能飞，减了晃几下就一头栽地了，说明减 50 过小了，那么就加 50 ，再飞。就这样一直找到一个比较合适的 P 参数。

# 六、 更进一步

通过以上的代码，我就得到了一个飞的不算太好，但是可以飞的四轴。在我通过方向操作让四轴倾斜 40 度时，由于 I 的左右，可以在我松开遥控器操作杆的瞬间给我自动回到水平位置。

但是现在的问题就是四轴自己总是在正负 10 度之间晃来晃去，这是由于加速度计被震动干扰引起陀螺仪积分被干扰得结果。


可以看出积分非常乱，但是就像一般震动那样，总是差不多对称的。所以我就针对积分部分又做了一个移动期为 50 的加权移动平均，然后四轴就稳定了。

不过这样一来，又出现了一个新的问题。那就是在距离平衡位置正负 15 度的范围内，四轴的自动平衡修正不那么明显了。下面我想采取的步骤就是再对加速度计移动期为 10 后的数据继续做加权移动平均，以 100Hz 的速度来处理 300Hz 的数据。从 Excel 的计算结果看来这样就可以在一个比较长一点的时间范围内给出我一个相对准确的四轴姿态了。

## 附录 1：PPM 转发代码

http://bbs.5imx.com/bbs/viewthread.php?tid=219838&page=26#pid3319190

花费了将近 3 天时间，挂掉一片 AVR Mega8，终于完成了一个 PPM 转发的代码。
想要完成舵量输出，非常简单。光是输出想做到没有舵机抖动也很容易。但要做到舵量采集后再输出没有抖动就复杂了。
一共 2ms 的舵量信息，M8 内置的 8M 频率，代码上稍微多执行几行、少执行几行，采集回来的舵量就变了。

时间都花在这上面了，本来第一天就完成了代码，在我的辉盛 5 克上没有发现抖动的问题。挺开心，心想蛮简单的。可换到辉盛 9 克上，就开始抖。而且很频繁，基本没不抖的时候。

反复修改软件，调整计时顺序，舵量输出顺序，甚至想用外部 12M 晶振提高速度。我的那片 M8 就是为这个挂掉的。熔丝位设置了外部晶振，但是没起来，ISP 也进不去了。按照网上说的用外部有源晶振的方法倒是 ISP 能认了，但认的芯片型号不对。当时也没在意，直接修改熔丝位到内部晶振。结果可能由于数据 M8 认的有误，RESET 脚给关掉了，而且内部频率也不是 8M，倒像是 1M 的。这下彻底没着.... 我的第一片 M8 就这样交待了。

反复试验了 N 种方法，最后决定必须使用硬件的定时器来确定舵量。然后修改代码，将定时器 0 作为舵量输出用，利用定时器 2 进行舵量采集。
流程是，在 main 里先完成一次舵量采集，完全根据定时器的计时信息决定舵量，没有使用变量累加计数。然后定时器 0 启动，输出一次完整舵量。
输出完成后，定时器 0 暂停计时，再交由 main 继续下一次采集舵量。就这样一直循环下去。

舵量的输出也采用了新方法，定时器在输出完一部分后，直接修改 TIME0 的计时器到下一个状态的触发时间点上。都取消了软件变量累加的计时方法。

效果还是不错的，舵机已经基本不抖了。为了做到更好，稍微牺牲了一点灵敏度，代码里将舵量和上次的采集结果进行比较，发现差异超过 1 时再更新。这样最后的效果已经和直接接到接收机上没什么差别了。

下面是代码，希望对想做同样东西的人能有些帮助：

```
//-----------------------------------------
//
// PPM  解码/转发代码
// Version 1.0
// cnmusic@163.net
//  使用 AVR Mega8,使用内部 8M 晶振
//
//-----------------------------------------
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdlib.h>
```

```
#define FEED_DOG                    asm("wdr");
#define NOP                             asm("nop");
#define INT_ON                      sei();
#define INT_OFF                      cli();

typedef unsigned char           BOOL;
typedef unsigned char           BYTE;
typedef unsigned char           CHAR;

#define TRUE                        1
#define FALSE                       0

#define CHANNELCOUNT            1

#define  TICK_COUNT_LEAD                70                  // 前 0.5ms 对应的定时器 0
的 64 分频的计数数值，虽然不是实际的 0.5ms，但这是我这里舵机能
                                                                // 承受的最小数值了


typedef struct tagChannelData
{
    BYTE            nChannelOutValue;       // 舵量数据
    BYTE            nChannelOutStep;        // 舵量的输出状态
}CHANNELDATA;

volatile CHANNELDATA            g_ChannelData[CHANNELCOUNT] = {{0}};
volatile BOOL                   g_bOutAllComplete = TRUE;
volatile BYTE                   g_nOutIndex = 0;

// TIME0 定时器中断
// 这个中断用来切换各个舵机端口的状态，包含 3 个部分：
// 前 0.5ms 的引导，中间最大 2ms 的舵量，以及后面的低电平部分
// 为了精确计时，使用 TIME0 的计数器，而没用软件进行变量累加计数
ISR(TIMER0_OVF_vect)
{
    if (g_ChannelData[g_nOutIndex].nChannelOutStep == 0)
    {
        PORTD |= (1<<PD7);
        TCNT0 = 0xFF - TICK_COUNT_LEAD;
        g_ChannelData[g_nOutIndex].nChannelOutStep = 1;
    }
    else if (g_ChannelData[g_nOutIndex].nChannelOutStep == 1)
    {
        TCNT0 = 0xFF - g_ChannelData[g_nOutIndex].nChannelOutValue;
```

```
            g_ChannelData[g_nOutIndex].nChannelOutStep = 2;
        }
        else
        {
            PORTD &= ~(1<<PD7);
            g_nOutIndex++;

            if (g_nOutIndex >= CHANNELCOUNT)
            {
                g_bOutAllComplete = TRUE;
                TCCR0 = 0;
            }
            else
            {
                TCNT0 = 0xFF - 1;
            }
        }
}


int main()
{
        BYTE            nChannelValueFinal = 0;

        DDRD = (1<<DDD2) | (1<<DDD3) | (1<<DDD7);           // PD2，PD3 是指示灯，PD7 是舵机输出端口

        PORTD &= ~((1<<PD6) | (1<<PD7));
        PORTD |= (1<<PD3);                  // 指示灯亮

        INT_ON;                                     // 开总中断

        // 定时器 0 设置
        TIMSK |= (1<<TOIE0);            // 允许中断


        while (1)
        {
            // 下面的代码循环扫描所有舵机的输入端口，
            // 由于是测试板，所以这里只有一个 PD6 端口用作输入

            asm("nop");

            // TIME2 用来计数，看输入舵量是多大。
```

```
            TCCR2 = (1<<CS22);                  // 64 分频

            // 一直等待直到高电平来临
            while (!(PIND & (1<<PIND6)))
            {
                //asm("nop");                                   // 如果使用更高级别的
优化，要保留这个语句，否则整个 while 会被优化掉
            }

            // 重新设置计数器，开始计时
            TCNT2 = 0;

            // 等待前引导的 0.5ms 结束
            while (PIND & (1<<PIND6))
            {
                if (TCNT2 == TICK_COUNT_LEAD)           // 计数数值达到 0.5ms，计时
器清零
                {
                    TCNT2 = 0;
                    break;
                }
            }

            // 一直等到高电平结束
            while (PIND & (1<<PIND6))
            {
                //asm("nop");
            }

            // 高电平结束，根据计时信息将舵量换成数字信息。
            TCCR2 = 0;
            nChannelValueFinal = TCNT2 - 2;             // 示波器显示，这么测量后的
舵量信息和实际的有 16uS 的差异，所以这里减去一点

              if ((abs((int)g_ChannelData[0].nChannelOutValue - (int)nChannelValueFinal)) >
1)
            {
                g_ChannelData[0].nChannelOutValue = nChannelValueFinal;          // 保
存最后的舵量信息
            }

            g_ChannelData[0].nChannelOutStep = 0;           // 设置舵量输出标志
```

```
            // 重起定时器 0
            g_nOutIndex = 0;                                    // 从 0 通道开
始输出舵量
            TCNT0 = 0xFF - 1;                                   // 延迟 1，基本
是立即触发中断
            TCCR0 = (1<<CS00)|(1<<CS01);                        // 64 分频，定时器开
始工作

            // 继续等待下一次定时器 0 完成所有通道的舵量输出
            g_bOutAllComplete = FALSE;

            while (!g_bOutAllComplete);
        }
}
```

这种方法没有去管那个总脉冲时长，完全根据输入脉冲的长度决定。
缺点就是如果某个 IO 口没接接收器，那么主循环就会一直等待下去，而舵量就不会输出了。
所以在进行扫描前要先确定哪个端口有没有悬空。

这种方式不会造成漏采集，因为整个脉冲里，真正有用的部分非常短，都算上才 2.5ms，剩下的 17.5ms 都是低电平。
所以我们就利用这个时间差完成采集过程。

示波器显示，我们输出的舵机控制信号紧接着接收机发出的信号，频率和正脉宽也和接收机给出的一致。舵机也基本认可，基本上没有抖动现象发生。

测试过天地飞的发射/接收机和 ESKY 的发射/接收机，OK。

舵机：
辉盛 5g，9g
ESKY0500，0508
ELE ES03

效果还是不错的。

## 附录 2：M8 的 I2C BootLoader

http://bbs.5imx.com/bbs/viewthread.php?tid=219838&page=34#pid3638185

Atmel 的 AVR M8 的 I2C/TWI BootLoader 终于稳定了，真的还有不少问题呢。下面总结一下，希望对后来人有帮助。

我的目的是，将整个 BootLoader 都放到 M8 自带的 BootLoader 段内，然后加电时从 BootLoader 启动，再由程序通过 I2C 总线上传新固件或者引导 BootLoader 启动程序。

看起来并不复杂，将整个 BootLoader 的代码都编译进 BootLoader 段也很容易。通过在调用的函数前加上 BOOTLOADER_SECTION，这个是在<avr/boot.h>里定义的一个宏。告诉编译器这个函数要放到 bootloader 段里。最后的实际编译效果可以查看那个.map 文件。

这里有一点比较麻烦，就是必须将主函数放到最前面，否则加电后可能先执行某个子程序去了。通过将所有调用的函数前加上 static 就可以做到这点。

以上步骤都相对容易，设置 bootloader 的段地址也可以在网上找到相关资料。但下面就郁闷了。

经过这样的过程，Bootloader 可以被引导，但有的时候不正常。我在测试中加了一个灯的闪烁代码，结果发现有时候灯闪烁的速度很慢。当时不知道原因，以为编译器的问题、内置晶振的不稳定问题、加电电压不稳损坏了代码、芯片损坏等等原因。足足查了 4 天啊，真是痛苦。3 个芯片，同样的代码，有一个怎么都好使，另外 2 个时好时坏。

最后，在网上找到老外的一个 BootLoader，也是 I2C 的，终于发现问题所在。

原来设置成从 BootLoader 区引导后，必须在进入 C 的 main 前设置好堆栈。以前这部编译器帮我完成了，所以我从没注意。但这回就需要自己来写了。于是一段比较简单的汇编代码后，终于我的 BootLoader 可以正常执行了。

vectors.S 文件内容：

```
#include <avr/io.h>

#if __AVR_MEGA__
 #define XJMP jmp
 #define XCALL call
#else
 #define XJMP rjmp
 #define XCALL rcall
#endif

        .section .boot_vectors,"ax",@progbits
        XJMP    start

        .word   0               // pad (vector 7)
        .word   0               // pad (vector 8)
```

```
        //  中断向量表，我们没有调用中断，所以都是空。
        //  保留这个向量表的意思是，一旦我们需要调用中断了
        //  可以加在下面。
//vector  __vector_9      // OverFlow0
        .word   0
//vector  __vector_10
        .word   0
//vector  __vector_11     // OverFlow0
        .word   0
//vector  __vector_12
        .word   0
//vector  __vector_13
        .word   0
//vector  __vector_14
        .word   0
//vector  __vector_15
        .word   0
//vector  __vector_16     // OverFlow0
        .word   0
//vector  __vector_17     // TWI
        .word   0
//vector  __vector_18
        .word   0
//vector  __vector_19     // TWI
        .word   0
//vector  __vector_20
        .word   0
//vector  __vector_21
        .word   0
//vector  __vector_22
        .word   0
//vector  __vector_23
        .word   0
//vector  __vector_24     // TWI
        .word   0
//vector  __vector_25
        .word   0
//vector  __vector_26
        .word   0
//vector  __vector_27
        .word   0
//vector  __vector_28
        .word   0
```

```
        //vector  __vector_29
                .word    0
        //vector  __vector_30
                .word    0
        //vector  __vector_31
                .word    0
        //vector  __vector_32
                .word    0
        //vector  __vector_33     // TWI
                .word    0

        // We don't need any vectors higher than 33, so don't waste the flash


#define __zero_reg__      r1

                // 全局的__stack 变量
        .global __stack
        .set     __stack, RAMEND


start:
                // 设置堆栈，这个非常重要！
                // 必须在进入 C 程序前设置好
        clr     __zero_reg__
        out      _SFR_IO_ADDR(SREG), __zero_reg__
        ldi     r28,lo8(__stack)
        ldi     r29,hi8(__stack)
        out      _SFR_IO_ADDR(SPH), r29
        out      _SFR_IO_ADDR(SPL), r28

        ldi     r24,lo8(0)
        ldi     r25,hi8(0)

        XJMP      main
```

修改了老外的引导代码，简化了一些。这个是针对 M8 芯片的。

由于换文件后如果有同名的 section，编译器会在后面自动加一个.1。所以这里用了 2 个 section。
.bootloader 0xc25，这个是给 C 的程序用的。偏移了 0x25 个字节，因为那个汇编编译后生成了这么多的代码。
.boot_vectors 0xc00，这个是给这个汇编用的。
需要在 Project Options 的 Memory Settings 里设置好。

## 附录 3：完整的 TWIBootLoader.c 的代码

完整的 C 实现部分代码如下，参考 Atmel 的 TWI 例子修改的，执行的功能包括 BootLoader 签名验证，上传代码的 CRC16 校验验证，跳转到用户区等功能：

```
//
// TWI BootLoader.c
//

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/boot.h>
#include <util/delay.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

//---------------------------------------------
// 参数配置，包括地址和亮灯的 IO 口代码，程序会反复调用 BL_LIGHT1 和 BL_LIGHT2
//---------------------------------------------
#ifdef GPSBOARD
    #define TWI_ADDR            30

    #define BL_PORT_INIT()      DDRB = (1<< DDB0) | (1<<PB1) | (1<<PB2);
    #define BL_LIGHT1()             PORTB = (1<<PB0)|(1<<PB1);
    #define BL_LIGHT2()             PORTB = (1<<PB2)|(1<<PB1);
#elif DRIVERBOARD
    #define TWI_ADDR            20

    #define BL_PORT_INIT()      DDRC = (1<<DDC0) | (1<<DDC1);
    #define BL_LIGHT1()             PORTC = (1<<PC0);
    #define BL_LIGHT2()             PORTC = (1<<PC1);
#else
    #define TWI_ADDR            31

    #define BL_PORT_INIT()      DDRB = (1<<DDB3) | (1<<DDB1) | (1<< DDB0) |
(1<<DDB2);
    #define BL_LIGHT1()             PORTB = (1<<PB1) | (1<<PB0) | (1<<PB2) | (1<<PB3);
    #define BL_LIGHT2()             PORTB = 0;
#endif
```

```
#define BL_ADDR                         0x1800          // BootLoader 的地址
//-------------------------------------
// TWI_Slave.h  文件内容
//-------------------------------------

/************************************************************************
   TWI Status/Control register definitions
************************************************************************/

#define TWI_BUFFER_SIZE          (SPM_PAGESIZE+2)          // Reserves memory for the drivers transceiver buffer.
                                                           // Set this to the largest message size that will be sent including address byte.

/************************************************************************
   Global definitions
************************************************************************/

union TWI_statusReg                                 // Status byte holding flags.
{
      unsigned char all;
      struct
      {
            unsigned char lastTransOK:1;
            unsigned char RxDataInBuf:1;
            unsigned char genAddressCall:1;                                 // TRUE = General call, FALSE = TWI Address;
            unsigned char unusedBits:5;
      };
};

//extern union TWI_statusReg TWI_statusReg;

/************************************************************************
   Function definitions
************************************************************************/
static void TWI_Slave_Initialise( unsigned char );
static unsigned char TWI_Transceiver_Busy( void );
//static unsigned char TWI_Get_State_Info( void );
static void TWI_Start_Transceiver_With_Data( unsigned char * , unsigned int );
static void TWI_Start_Transceiver( void );
//static unsigned char TWI_Get_Data_From_Transceiver( unsigned char *, unsigned char );
```

```
static unsigned char* TWI_Get_Data_Pointer_From_Transceiver(void);
static void TWI_Get_Data_Pointer_From_Transceiver_Release(void);

//static void TWI_Prepare_Transceiver_CleanDataSize(void);
//static void TWI_Prepare_Transceiver_With_Data( unsigned char *msg, unsigned int msgSize);
//static void TWI_Start_TransceiverData(void);
static void TWI_ISR(void);
static void TWIProcess(void);

//------------------------------------------------------------------------------
typedef unsigned int      UINT;
typedef unsigned char     BYTE;

typedef struct tagInfo
{
      BYTE      szSign[2];
      BYTE      nSize;
      BYTE      PageSize;
      BYTE      nBLVersion;
      BYTE      CurrentAddr;
}INFO;


UINT      g_BL_FlashAddr = 0;
INFO      g_BL_CurrentInfo;


//用户程序起始地
#define PROG_START              0x0000

#define BOOTLOADER_ADDR_START_USER_APP      1           // 启动用户程序
#define BOOTLOADER_ADDR_GET_INFO            2           // 获得芯片信息
#define BOOTLOADER_ADDR_UPLOAD              3           // 上传代码
#define BOOTLOADER_ADDR_RESET               4           // 复位指针


/*****************************************************************************
   Bit and byte definitions
*****************************************************************************/
#define TWI_READ_BIT   0     // Bit position for R/W bit in "address byte".
#define TWI_ADR_BITS   1     // Bit position for LSB of the slave address bits in the init byte.
#define TWI_GEN_BIT    0     // Bit position for LSB of the general call bit in the init byte.

#define TRUE               1
```

```
#define FALSE               0

/*************************************************************************
  TWI State codes
*************************************************************************/
// General TWI Master staus codes
#define TWI_START                   0x08  // START has been transmitted
#define TWI_REP_START               0x10  // Repeated START has been transmitted
#define TWI_ARB_LOST                0x38  // Arbitration lost

// TWI Master Transmitter staus codes
#define TWI_MTX_ADR_ACK             0x18  // SLA+W has been tramsmitted and ACK received
#define TWI_MTX_ADR_NACK            0x20  // SLA+W has been tramsmitted and NACK received
#define TWI_MTX_DATA_ACK            0x28  // Data byte has been tramsmitted and ACK received
#define TWI_MTX_DATA_NACK           0x30  // Data byte has been tramsmitted and NACK received

// TWI Master Receiver staus codes
#define TWI_MRX_ADR_ACK             0x40  // SLA+R has been tramsmitted and ACK received
#define TWI_MRX_ADR_NACK            0x48  // SLA+R has been tramsmitted and NACK received
#define TWI_MRX_DATA_ACK            0x50  // Data byte has been received and ACK tramsmitted
#define TWI_MRX_DATA_NACK           0x58  // Data byte has been received and NACK tramsmitted

// TWI Slave Transmitter staus codes
#define TWI_STX_ADR_ACK             0xA8  // Own SLA+R has been received; ACK has been returned
#define TWI_STX_ADR_ACK_M_ARB_LOST 0xB0  // Arbitration lost in SLA+R/W as Master; own SLA+R has been received; ACK has been returned
#define TWI_STX_DATA_ACK            0xB8  // Data byte in TWDR has been transmitted; ACK has been received
#define TWI_STX_DATA_NACK           0xC0  // Data byte in TWDR has been transmitted; NOT ACK has been received
#define TWI_STX_DATA_ACK_LAST_BYTE 0xC8  // Last data byte in TWDR has been transmitted (TWEA = ??; ACK has been received

// TWI Slave Receiver staus codes
#define TWI_SRX_ADR_ACK             0x60  // Own SLA+W has been received ACK has been
```

returned
#define TWI_SRX_ADR_ACK_M_ARB_LOST 0x68 // Arbitration lost in SLA+R/W as Master; own SLA+W has been received; ACK has been returned
#define TWI_SRX_GEN_ACK 0x70 // General call address has been received; ACK has been returned
#define TWI_SRX_GEN_ACK_M_ARB_LOST 0x78 // Arbitration lost in SLA+R/W as Master; General call address has been received; ACK has been returned
#define TWI_SRX_ADR_DATA_ACK 0x80 // Previously addressed with own SLA+W; data has been received; ACK has been returned
#define TWI_SRX_ADR_DATA_NACK 0x88 // Previously addressed with own SLA+W; data has been received; NOT ACK has been returned
#define TWI_SRX_GEN_DATA_ACK 0x90 // Previously addressed with general call; data has been received; ACK has been returned
#define TWI_SRX_GEN_DATA_NACK 0x98 // Previously addressed with general call; data has been received; NOT ACK has been returned
#define TWI_SRX_STOP_RESTART 0xA0 // A STOP condition or repeated START condition has been received while still addressed as Slave

// TWI Miscellaneous status codes
#define TWI_NO_STATE 0xF8 // No relevant state information available; TWINT = ??
#define TWI_BUS_ERROR 0x00 // Bus error due to an illegal START or STOP condition

//---------------------------------
// TWI_Slave.c 文件内容
//---------------------------------
static unsigned char TWI_buf[TWI_BUFFER_SIZE]; // Transceiver buffer. Set the size in the header file
static unsigned int TWI_msgSize = 0; // Number of bytes to be transmitted.
static unsigned char TWI_state = TWI_NO_STATE; // State byte. Default set to TWI_NO_STATE.

union TWI_statusReg TWI_statusReg = {0}; // TWI_statusReg is defined in TWI_Slave.h

BOOTLOADER_SECTION int main(void)
{
    int nCount=0;

    cli();

```
    BL_PORT_INIT();

    memset(TWI_buf,0,TWI_BUFFER_SIZE);
    g_BL_FlashAddr = 0;
    g_BL_CurrentInfo.szSign[0] = 'B';
    g_BL_CurrentInfo.szSign[1] = 'L';
    g_BL_CurrentInfo.nSize = sizeof(g_BL_CurrentInfo);
    g_BL_CurrentInfo.nBLVersion = 1;
    g_BL_CurrentInfo.CurrentAddr = g_BL_FlashAddr;
    g_BL_CurrentInfo.PageSize = SPM_PAGESIZE;

    TWI_Slave_Initialise((TWI_ADDR<<TWI_ADR_BITS) | (0<<TWI_GEN_BIT));

    TWI_Start_Transceiver();

    while (1)
    {
        //while (TWI_Transceiver_Busy())
        if (TWCR & (1<<TWIE))
        {
            if (TWCR & (1<<TWINT))
            {
                TWI_ISR();
            }
        }

        TWIProcess();

        nCount++;

        if ((nCount >= 10000) && (nCount < 20000))
        {
            BL_LIGHT1();
        }
        else if (nCount >= 20000)
        {
            BL_LIGHT2();
            nCount = 0;
        }
    }

    return 0;
}
```

```
/************************************************************************
Call this function to set up the TWI slave to its initial standby state.
Remember to enable interrupts from the main application after initializing the TWI.
Pass both the slave address and the requrements for triggering on a general call in the
same byte. Use e.g. this notation when calling this function:
TWI_Slave_Initialise( (TWI_slaveAddress<<TWI_ADR_BITS) | (TRUE<<TWI_GEN_BIT) );
The TWI module is configured to NACK on any requests. Use a TWI_Start_Transceiver function to
start the TWI.
************************************************************************/
static BOOTLOADER_SECTION void TWI_Slave_Initialise( unsigned char TWI_ownAddress )
{
  TWAR = TWI_ownAddress;                                  // Set own TWI slave address.
Accept TWI General Calls.
  TWDR = 0xFF;                                            // Default content = SDA released.
  TWCR = (1<<TWEN)|                                       // Enable TWI-interface and
release TWI pins.
       (0<<TWIE)|(0<<TWINT)|                              // Disable TWI Interupt.
       (0<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|                   // Do not ACK on any requests,
yet.
       (0<<TWWC);                                         //
}

/************************************************************************
Call this function to test if the TWI_ISR is busy transmitting.
************************************************************************/
static BOOTLOADER_SECTION unsigned char TWI_Transceiver_Busy( void )
{
  return ( TWCR & (1<<TWIE) );                    // IF TWI interrupt is enabled then the
Transceiver is busy
}

/************************************************************************
Call this function to fetch the state information of the previous operation. The function will hold
execution (loop)
until the TWI_ISR has completed with the previous operation. If there was an error, then the
function
will return the TWI State code.
************************************************************************/
/*
static BOOTLOADER_SECTION unsigned char TWI_Get_State_Info( void )
{
  while ( TWI_Transceiver_Busy() );                // Wait until TWI has completed the
```

```
transmission.
   return ( TWI_state );                                  // Return error state.
}
*/
/*************************************************************************
Call this function to send a prepared message, or start the Transceiver for reception. Include
a pointer to the data to be sent if a SLA+W is received. The data will be copied to the TWI buffer.
Also include how many bytes that should be sent. Note that unlike the similar Master function, the
Address byte is not included in the message buffers.
The function will hold execution (loop) until the TWI_ISR has completed with the previous operation,
then initialize the next operation and return.
*************************************************************************/
static BOOTLOADER_SECTION void TWI_Start_Transceiver_With_Data( unsigned char *msg, unsigned int msgSize )
{
   unsigned int temp;

   while ( TWI_Transceiver_Busy() );                      // Wait until TWI is ready for next transmission.

   TWI_msgSize = msgSize;                                 // Number of data to transmit.
   for ( temp = 0; temp < msgSize; temp++ )               // Copy data that may be transmitted if the TWI Master requests data.
      TWI_buf[ temp ] = msg[ temp ];
   TWI_statusReg.all = 0;
   TWI_state               = TWI_NO_STATE ;
   TWCR = (1<<TWEN)|                                      // TWI Interface enabled.
          (1<<TWIE)|(1<<TWINT)|                           // Enable TWI Interupt and clear the flag.
          (1<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|                // Prepare to ACK next time the Slave is addressed.
          (0<<TWWC);                                      //
}

/*
static BOOTLOADER_SECTION void TWI_Prepare_Transceiver_CleanDataSize()
{
      while ( TWI_Transceiver_Busy() );                   // Wait until TWI is ready for next transmission.

      TWI_msgSize = 0;
}
```

```
*/

/*
static BOOTLOADER_SECTION void TWI_Prepare_Transceiver_With_Data( unsigned char *msg, unsigned int msgSize )
{
	unsigned int temp;

	while ( TWI_Transceiver_Busy() );			// Wait until TWI is ready for next transmission.

	for ( temp = TWI_msgSize; temp < TWI_msgSize+msgSize; temp++ )		// Copy data that may be transmitted if the TWI Master requests data.
	{
		TWI_buf[temp] = msg[temp-TWI_msgSize];
	}

	TWI_msgSize += msgSize;
}*/

/*
static BOOTLOADER_SECTION void TWI_Start_TransceiverData()
{
	while ( TWI_Transceiver_Busy() );			// Wait until TWI is ready for next transmission.
	TWI_statusReg.all = 0;
	TWI_state		= TWI_NO_STATE ;
	TWCR = (1<<TWEN)|					// TWI Interface enabled.
		(1<<TWIE)|(1<<TWINT)|				// Enable TWI Interupt and clear the flag.
		(1<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|		// Prepare to ACK next time the Slave is addressed.
		(0<<TWWC);					//
}*/

/*************************************************************************
Call this function to start the Transceiver without specifing new transmission data. Usefull for restarting
a transmission, or just starting the transceiver for reception. The driver will reuse the data previously put
in the transceiver buffers. The function will hold execution (loop) until the TWI_ISR has completed with the
previous operation, then initialize the next operation and return.
*************************************************************************/
```

```
static BOOTLOADER_SECTION void TWI_Start_Transceiver( void )
{
  while ( TWI_Transceiver_Busy() );                    // Wait until TWI is ready for next transmission.
  TWI_statusReg.all = 0;
  TWI_state          = TWI_NO_STATE ;
  TWCR = (1<<TWEN)|                                    // TWI Interface enabled.
         (1<<TWIE)|(1<<TWINT)|                         // Enable TWI Interupt and clear the flag.
         (1<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|              // Prepare to ACK next time the Slave is addressed.
         (0<<TWWC);                                    //
}
/*****************************************************************************
Call this function to read out the received data from the TWI transceiver buffer. I.e. first call
TWI_Start_Transceiver to get the TWI Transceiver to fetch data. Then Run this function to collect the
data when they have arrived. Include a pointer to where to place the data and the number of bytes
to fetch in the function call. The function will hold execution (loop) until the TWI_ISR has completed
with the previous operation, before reading out the data and returning.
If there was an error in the previous transmission the function will return the TWI State code.
*****************************************************************************/
/*
static BOOTLOADER_SECTION unsigned char TWI_Get_Data_From_Transceiver( unsigned char *msg, unsigned char msgSize )
{
  unsigned int i;

  while ( TWI_Transceiver_Busy() );                    // Wait until TWI is ready for next transmission.

  if( TWI_statusReg.lastTransOK )                      // Last transmission competed successfully.
  {
    for ( i=0; i<msgSize; i++ )                        // Copy data from Transceiver buffer.
    {
      msg[ i ] = TWI_buf[ i ];
    }
    TWI_statusReg.RxDataInBuf = FALSE;                 // Slave Receive data has been read from buffer.
  }
  return( TWI_statusReg.lastTransOK );
}
```

```
*/

static BOOTLOADER_SECTION unsigned char* TWI_Get_Data_Pointer_From_Transceiver()
{
    //while ( TWI_Transceiver_Busy() );                    // Wait until TWI is ready for next transmission.

    if( TWI_statusReg.lastTransOK )                        // Last transmission competed successfully.
    {
        return TWI_buf;
    }
    else
    {
        return 0;
    }
}

static BOOTLOADER_SECTION void TWI_Get_Data_Pointer_From_Transceiver_Release()
{
    TWI_statusReg.RxDataInBuf = FALSE;                     // Slave Receive data has been read from buffer.
}

// ********** Interrupt Handlers ********** //
/*****************************************************************************
This function is the Interrupt Service Routine (ISR), and called when the TWI interrupt is triggered;
that is whenever a TWI event has occurred. This function should not be called directly from the main
application.
*****************************************************************************/
//BOOTLOADER_SECTION ISR(TWI_vect)
static BOOTLOADER_SECTION void TWI_ISR(void)
{
  static unsigned int TWI_bufPtr;
  switch (TWSR)
  {
    case TWI_STX_ADR_ACK:                  // Own SLA+R has been received; ACK has been returned
//      case TWI_STX_ADR_ACK_M_ARB_LOST: // Arbitration lost in SLA+R/W as Master; own SLA+R has been received; ACK has been returned
        TWI_bufPtr    = 0;                                 // Set buffer pointer to first data location
```

```
      case TWI_STX_DATA_ACK:                // Data byte in TWDR has been transmitted; ACK has
been received
         TWDR = TWI_buf[TWI_bufPtr++];
         TWCR = (1<<TWEN)|                                  // TWI Interface enabled
                (1<<TWIE)|(1<<TWINT)|                       // Enable TWI Interupt and
clear the flag to send byte
                (1<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|            //
                (0<<TWWC);                                  //
         break;
      case TWI_STX_DATA_NACK:               // Data byte in TWDR has been transmitted; NACK
has been received.
                                            // I.e. this could be the end of the transmission.
         if (TWI_bufPtr == TWI_msgSize) // Have we transceived all expected data?
         {
           TWI_statusReg.lastTransOK = TRUE;                // Set status bits to completed
successfully.
         }else                              // Master has sent a NACK before all data where
sent.
         {
           TWI_state = TWSR;                                // Store TWI State as
errormessage.
         }
                                                            // Put TWI Transceiver in
passive mode.
         TWCR = (1<<TWEN)|                                  // Enable TWI-interface and
release TWI pins
                (0<<TWIE)|(0<<TWINT)|                       // Disable Interupt
                (0<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|            // Do not acknowledge on
any new requests.
                (0<<TWWC);                                  //
         break;
      case TWI_SRX_GEN_ACK:                 // General call address has been received; ACK has
been returned
//       case TWI_SRX_GEN_ACK_M_ARB_LOST: // Arbitration lost in SLA+R/W as Master; General
call address has been received; ACK has been returned
         TWI_statusReg.genAddressCall = TRUE;
      case TWI_SRX_ADR_ACK:                 // Own SLA+W has been received ACK has been
returned
//       case TWI_SRX_ADR_ACK_M_ARB_LOST: // Arbitration lost in SLA+R/W as Master; own
SLA+W has been received; ACK has been returned
                                                            // Dont need to clear
TWI_S_statusRegister.generalAddressCall due to that it is the default state.
         TWI_statusReg.RxDataInBuf = TRUE;
         TWI_bufPtr     = 0;                                // Set buffer pointer to first
```

```
data location
                                                        // Reset the TWI Interupt to
wait for a new event.
        TWCR = (1<<TWEN)|                               // TWI Interface enabled
                (1<<TWIE)|(1<<TWINT)|                   // Enable TWI Interupt and
clear the flag to send byte
                (1<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|        // Expect ACK on this
transmission
                (0<<TWWC);                              //
        break;
    case TWI_SRX_ADR_DATA_ACK:      // Previously addressed with own SLA+W; data has
been received; ACK has been returned
    case TWI_SRX_GEN_DATA_ACK:      // Previously addressed with general call; data has
been received; ACK has been returned
        TWI_buf[TWI_bufPtr++]       = TWDR;
        TWI_statusReg.lastTransOK   = TRUE;             // Set flag transmission
successfull.
                                                        // Reset the TWI Interupt to
wait for a new event.
        TWCR = (1<<TWEN)|                               // TWI Interface enabled
                (1<<TWIE)|(1<<TWINT)|                   // Enable TWI Interupt and
clear the flag to send byte
                (1<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|        // Send ACK after next
reception
                (0<<TWWC);                              //
        break;
    case TWI_SRX_STOP_RESTART:      // A STOP condition or repeated START condition has
been received while still addressed as Slave
                                                        // Put TWI Transceiver in
passive mode.
        TWCR = (1<<TWEN)|                               // Enable TWI-interface and
release TWI pins
                (0<<TWIE)|(0<<TWINT)|                   // Disable Interupt
                (0<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|        // Do not acknowledge on
any new requests.
                (0<<TWWC);                              //
        break;
    case TWI_SRX_ADR_DATA_NACK:     // Previously addressed with own SLA+W; data has
been received; NOT ACK has been returned
    case TWI_SRX_GEN_DATA_NACK:     // Previously addressed with general call; data has
been received; NOT ACK has been returned
    case TWI_STX_DATA_ACK_LAST_BYTE: // Last data byte in TWDR has been transmitted
(TWEA = ??; ACK has been received
//      case TWI_NO_STATE           // No relevant state information available; TWINT
```

```
= ??
      case TWI_BUS_ERROR:               // Bus error due to an illegal START or STOP condition
      default:
        TWI_state  =  TWSR;                                          // Store TWI State as
errormessage, operation also clears the Success bit.
        TWCR = (1<<TWEN)|                                            // Enable TWI-interface and
release TWI pins
                (0<<TWIE)|(0<<TWINT)|                               // Disable Interupt
                (0<<TWEA)|(0<<TWSTA)|(1<<TWSTO)|                    // Do not acknowledge on
any new requests.
                (0<<TWWC);                                           //
   }
}

//---------------------------------------------------------------
// 使用静态函数确保 BootLoader 的 main 位于 bootloader 端的最开始，
// 这样可以通过直接从 Bootloader 段启动的融丝位直接跑到 BootLoader 里。
//---------------------------------------------------------------

//更新一个 Flash 页
static BOOTLOADER_SECTION void BootLoader_WriteOnePage(BYTE *pBuffer)
{
      UINT              PagePointer;

      if (g_BL_FlashAddr < BL_ADDR)
      {
            boot_page_erase(g_BL_FlashAddr);                        //擦除一个 Flash 页
            boot_spm_busy_wait();

            for (PagePointer = 0; PagePointer < SPM_PAGESIZE; PagePointer += 2)
            {
                  boot_page_fill(g_BL_FlashAddr    +    PagePointer,    pBuffer[PagePointer]    |
((pBuffer[PagePointer + 1] << 8))); // 一次写入一个 WORD，2 个 BYTE。
            }

            boot_page_write(g_BL_FlashAddr);                        //将缓冲页数据写入一个
Flash 页
            boot_spm_busy_wait();                                   //等待页编程完成

            g_BL_FlashAddr += SPM_PAGESIZE;
            g_BL_CurrentInfo.CurrentAddr = g_BL_FlashAddr/SPM_PAGESIZE;
      }
}
```

```
// CRC16 校验
static BOOTLOADER_SECTION void CRC16(BYTE *buf,int nSize,BYTE *pHighByte,BYTE *pLowByte)
{
	unsigned int j;
	unsigned char i;
	unsigned int t;
	unsigned int crc;

	crc = 0;

	for(j = nSize; j > 0; j--)
	{
		//标准 CRC 校验
		crc = (crc ^ (((unsigned int) *buf) << 8));
		for(i = 8; i > 0; i--)
		{
		   t = crc << 1;
		   if(crc & 0x8000)
		      t = t ^ 0x1021;
		   crc = t;
		}
		buf++;
	}

	*pHighByte = crc / 256;
	*pLowByte = crc % 256;
}

//跳转到用户程序
static BOOTLOADER_SECTION void quit()
{
	boot_rww_enable();                                    //允许用户程序区读写
	TWCR = (0<<TWEN)|                                     // Disable TWI-interface and
release TWI pins
	       (0<<TWIE)|(1<<TWINT)|                          // Disable Interupt
	       (0<<TWEA)|(0<<TWSTA)|(1<<TWSTO)|               // Do not acknowledge on any
new requests.
	       (0<<TWWC);                                     //

	(*((void(*)(void))PROG_START))();                     //跳转，这样比'jmp 0'节省空间
}
```

```
static BOOTLOADER_SECTION void TWIProcess()
{
	BYTE		*pMessageBuf = 0;
	BYTE	CRCHigh = 0;
	BYTE	CRCLow = 0;
	BYTE	nInternalAddr = 0;


	// TWI 接收器是否忙？
	if (!TWI_Transceiver_Busy())
	{
		// 上次操作是否成功？
		if (TWI_statusReg.lastTransOK)
		{
			// 最后一次操作是否收到了请求？
			if (TWI_statusReg.RxDataInBuf)
			{
				pMessageBuf = TWI_Get_Data_Pointer_From_Transceiver();

				nInternalAddr = pMessageBuf[0];

				pMessageBuf = &pMessageBuf[1];
				//TWI_Get_Data_From_Transceiver(messageBuf, 2);

				// 是否为一个 0 地址的广播消息？
				if (TWI_statusReg.genAddressCall)
				{
					// 0 地址的广播消息，我们不去理会。
				}
				else
				{
					// 亮灯提示进度
					/*
					if (PINB & (1<<PB2))
					{
						PORTB &= ~(1<<PB2);
					}
					else
					{
						PORTB |= (1<<PB2);
					}
					*/

					// 地址正确，下面解析收到的命令
```

```
                    //-------------------------------------------------
                    // 写命令部分
                    if (nInternalAddr == BOOTLOADER_ADDR_START_USER_APP)
                    {
                        // 跳转到用户代码区。
                        quit();

                        while(1);
                    }
                    else if (nInternalAddr == BOOTLOADER_ADDR_RESET)
                    {
                        g_BL_FlashAddr = 0;
                        g_BL_CurrentInfo.CurrentAddr = 0;
                    }
                    else if (nInternalAddr == BOOTLOADER_ADDR_UPLOAD)
                    {
                        // 上传用户代码
                        // 前 SPM_PAGESIZE 个字节为数据，后 2 个字节为 CRC16 校验。
                        CRC16(pMessageBuf,SPM_PAGESIZE,&CRCHigh,&CRCLow);

                        if ((CRCHigh == pMessageBuf[SPM_PAGESIZE]) && (CRCLow ==
pMessageBuf[SPM_PAGESIZE+1]))
                        {
                            // CRC 正确，将舵量信息保存下来。
                            // 将这个扇区的内容写入。
                            BootLoader_WriteOnePage(pMessageBuf);
                        }

                        TWI_Get_Data_Pointer_From_Transceiver_Release();

                    }
                    //-------------------------------------------------
                    // 读命令部分
                    else if (nInternalAddr == BOOTLOADER_ADDR_GET_INFO)
                    {

    TWI_Start_Transceiver_With_Data((BYTE*)&g_BL_CurrentInfo,sizeof(g_BL_CurrentInfo));
                        TWI_Get_Data_Pointer_From_Transceiver_Release();
                    }
                }
            }

            // Check if the TWI Transceiver has already been started.
            // If not then restart it to prepare it for new receptions.
```

```
                if (!TWI_Transceiver_Busy())
                {
                    TWI_Start_Transceiver();
                }
            }
            else
            {
                // 上次操作不成功，重新启动接收器
                //TWI_Act_On_Failure_In_Last_Transmission( TWI_Get_State_Info() );
                TWI_Start_Transceiver();
            }
        }
}
```

```
/* 测试用
int main(void)
{

    while(1);

    main1();

    DDRB = (1<<DDB2) | (1<<DDB1) | (1<< DDB0);

    while (1)
    {
        PORTB = (1<<PB2);

        _delay_ms(200);

        PORTB = 0;

        _delay_ms(200);
    }

}
*/
```

## 附录 4：GPS NMEA 参数解析的 M8 代码。

直接在串口接受中断里调用就可以了，一次处理一个字节。

```
/*
    AVR GPS  数据解码模块。

    将 GPS 的 NMEMA 信息解码，然后通过 I2C 总线传递到主机
*/

#include <stdio.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <string.h>


#define INT_ON                  sei();
#define INT_OFF                 cli();

typedef struct tagGPSPosition
{
    //float         fLon;                       // 经度  Longitude
    USHORT      nLonHigh;
    USHORT      nLonLow;
    BYTE    WE;
    BYTE    Reserved1;                  // 由于 AVR 是 8 位单片机，所以存储结构上可以
按照 1 字节对齐。
                                        // 但 ARM 和 PC 是 32 位单片机，所以必须补
齐一个空余的字节。
    //float         fLat;                   // 纬度  Latitude
    USHORT      nLatHigh;
    USHORT      nLatLow;
    BYTE    NS;
    BYTE    Reserved2;
}GPSPOSITION;

#define NOP                     asm("nop");
#define TX_BUFFER_SIZE 20
#define UDR_EMPTY (1<<UDRE)

BOOL                    g_bState = FALSE;           // GPS 定位状态
GPSPOSITION             g_Position = {0};
```

```
BYTE tx_buffer[TX_BUFFER_SIZE]={0};
BYTE tx_wr_index=0;
BYTE tx_rd_index=0;
BYTE tx_counter=0;

void GPS_Decode(BYTE nData)
{
    /*
    解析这组信息：
    $GPRMC,121252.000,A,3958.3032,N,11629.6046,E,15.15,359.95,070306,,,A*54
    $GPRMC,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,<10>,<11>,<12>*hh<CR><LF>

    <1> UTC 时间，hhmmss（时分秒）格式
    <2> 定位状态，A=有效定位，V=无效定位
    <3> 纬度 ddmm.mmmm（度分）格式（前面的 0 也将被传输）
    <4> 纬度半球 N（北半球）或 S（南半球）
    <5> 经度 dddmm.mmmm（度分）格式（前面的 0 也将被传输）
    <6> 经度半球 E（东经）或 W（西经）
    <7> 地面速率（000.0~999.9 节，前面的 0 也将被传输）
    <8> 地面航向（000.0~359.9 度，以真北为参考基准，前面的 0 也将被传输）
    <9> UTC 日期，ddmmyy（日月年）格式
    <10> 磁偏角（000.0~180.0 度，前面的 0 也将被传输）
    <11> 磁偏角方向，E（东）或 W（西）
    <12> 模式指示（仅 NMEA0183 3.00 版本输出，A=自主定位，D=差分，E=估算，N=数据
无效）

    */

    static    int         nDataIndex = 0;
    static BOOL    bDataFieldOK = FALSE;
    static    int     nItemIndex = 0;
    static    char szBuffer[15] = {0};
    static    char *pBufferIndex = 0;
    BYTEi = 0;
    USHORT   nData1 = 0;
    USHORT   nData2 = 0;

    if (nData == '$')
    {
        // 新的一组数据开始。
        nDataIndex = 0;
        nItemIndex = 0;
        pBufferIndex = &szBuffer[0];
        bDataFieldOK = FALSE;
```

```
    }
    else
    {
        nDataIndex ++;
    }

    if ((nDataIndex == 5) && (nData == 'C'))        // 确认第 6 个字符是 C，也就是 GPS 模块返回的$GPRMC 数据条目
    {
        if (!g_bState)
        {
            PORTB |= (1<<PB0);
        }
        // 可以扫描后续信息。
        bDataFieldOK = TRUE;
    }
    else if ((nDataIndex == 5) && (nData != 'C'))  // 字段长度足够，但不是$GPRMC 数据条目
    {
        // 不是我们需要的字段，忽略不去处理。
    }

    if ((bDataFieldOK) && (nDataIndex > 6))
    {
        // 后续数据已经过来了，需要进行拆分。
        if ((nData == ',') || (nData == '*'))
        {
            PORTB &= ~(1<<PB0);

            *pBufferIndex = 0;          // 写入终止符。

            // 分析上一组数据
            switch(nItemIndex)
            {
                case 0:
                    // UTC 时间，hhmmss
                    //TRACE("UTC Time: %s",szBuffer);
                    break;
                case 1:
                    // 定位状态，A=有效定位，V=无效定位
                    //TRACE("定位状态: %s",szBuffer);

                    if (strcmp(szBuffer,"A") == 0)
                    {
                        PORTB |= (1<<PB2);
```

```
                g_bState = TRUE;
            }
            else
            {
                PORTB &= ~(1<<PB2);
                g_bState = FALSE;
            }
            break;
        case 2:
            // 纬度 ddmm.mmmm（度分）格式（前面的 0 也将被传输）
            // 由于 AVR 和 Intel32 位计算机的浮点数存储格式不大一样，所以不能直接转成浮点传递。
            //g_Position.fLat = atof(szBuffer);
            for (i=0;i<sizeof(szBuffer);i++)
            {
                if (szBuffer[i] == '.')
                {
                    szBuffer[i] = 0;
                    nData1 = atoi(szBuffer);
                    nData2 = atoi(&szBuffer[i+1]);

                    g_Position.nLatHigh = nData1;
                    g_Position.nLatLow = nData2;
                    break;
                }
            }

            break;
        case 3:
            // 纬度半球 N（北半球）或 S（南半球）
            g_Position.NS = szBuffer[0];
            break;
        case 4:
            // 经度 dddmm.mmmm（度分）格式（前面的 0 也将被传输）
            // g_Position.fLon = atof(szBuffer);
            // 由于 AVR 和 Intel32 位计算机的浮点数存储格式不大一样，所以不能直接转成浮点传递。
            for (i=0;i<sizeof(szBuffer);i++)
            {
                if (szBuffer[i] == '.')
                {
                    szBuffer[i] = 0;
                    nData1 = atoi(szBuffer);
                    nData2 = atoi(&szBuffer[i+1]);
```

```
                        g_Position.nLonHigh = nData1;
                        g_Position.nLonLow = nData2;
                        break;
                    }
                }
                break;
            case 5:
                // 经度半球 E（东经）或 W（西经）
                g_Position.WE = szBuffer[0];
                break;
            case 6:
                // 地面速率（000.0~999.9 节，前面的 0 也将被传输）
                break;
            case 7:
                // 地面航向（000.0~359.9 度，以真北为参考基准，前面的 0 也将被传输）
                break;
            case 8:
                // UTC 日期，ddmmyy（日月年）格式
                break;
            case 9:
                // 磁偏角（000.0~180.0 度，前面的 0 也将被传输）
                break;
            case 10:
                // 磁偏角方向，E（东）或 W（西）
                break;
            case 11:
                // 模式指示（仅 NMEA0183 3.00 版本输出，A=自主定位，D=差分，E=估算，N=数据无效）

                // 这是$GPRMC 的最后一项数据。
                // 我们需要的数据已经获得了，下面的数据就要等待下一次 1s 后再来了。
                // 现在可以允许采集温度了。
                g_bAllowGetTemperature = FALSE;
                break;
            default:
                break;

        }

        // 逗号，需要切换到下一组数据。
        nItemIndex++;
```

```
                // 重新设置缓冲区起始位置。
                pBufferIndex = &szBuffer[0];
            }
            else
            {
                // 数据写入缓冲区。
                *pBufferIndex = nData;
                pBufferIndex++;
            }
        }

}
```

## 附录 5：GPS 的 NMEA 里的 ddmm.mmmm 坐标到 GoogleEarth 的坐标转换。

其实是非常简单的。
比如：GPS 返回 ddmm.mmmm 格式的一个坐标 3958.2399，这里 39 表示 dd，也就是度，58 表示分。这 2 个和 GoogleEarth 里的一样，不用转换。唯一需要调整的就是后面的这个 2399。将这个分变成 GoogleEarch 里的秒就可以了，也就是*60。

输入：3958.2399
直接得出度和分，也就是 39 度 58 分，下面的 2399 要写成 0.2399，然后*60=0.2399*60=14.394。所以最后换成 GoogleEarth 的坐标就是 39 度 58 分 14.394 秒，也就是 39 58'14.394''，再配合上东西经度和南北纬度，就可以直接在 GoogleEarth 里看到了。