# LPC54606平台的周期信号基波幅值提取程序使用文档

# 一、引言

## 1、说在前面

虽然目前做电磁的不多，但是后期真正做比赛时，电磁还是必不可少的，因此能有一个稳定完善的电磁方案还是非常重要的，由于这个程序并不完善，是否稳定还需实际应用测试。

在编写这个底层的时候，我踩到的坑太多太多，需要解决的问题和之前做智能车遇到的完全不同，就像是白手起家，我不想学弟学妹在未来移植和完善这个程序的时候，依然要解决相同的问题，因此我写下这篇说明，记录我程序的流程和遇到的困难，希望能帮助你们理解和运用本程序。

因为理论和实际往往会相差很大，这里我需要你们务必趁早使用这个程序，调试过程中我遇到了无数次的数组越界，堆栈溢出的情况，因为能力限制我至今未确认真正原因，不过很大可能性是DMA传输和FFT迭代过程出了问题。这个问题在今后调车会是致命的，他会直接导致程序卡死，并且无法恢复，只能重新复位。

所以这就需要你们在训练阶段进行大量的测试，需要每个人的优化，如果遇到任何有关这段程序的问题，一定要趁早解决，并且告诉我。

## 2、改用此程序的缘由

需要采集到的电磁信号是一个20Khz 的正弦信号，且信号的峰峰值与电感和电磁线的距离呈倍数关系

传统的办法是利用预先设计好的滤波电路将20Khz 的正弦信号转换为直流信号，再通过单片机AD转换直接得到信号的强度，但是如果精度要求比较高，会增加电路的复杂程度，并且由于电磁前瞻板的空间和体积限制，电路板不能做的很大，如果没有扎实的焊接基础和电路基础，很难一次性焊接成功，且后续的电路检修也非常困难。

因此就萌生了这样一个想法，直接通过简单的电路将信号一次放大后输入单片机，通过程序来分析正弦信号的峰峰值，这也就大大减轻了硬件上的压力，也降低了电路板出现问题的可能性。

## 3、硬件提取幅值和软件FFT提取幅值在程序上的优缺点

### (1)硬件优点

程序实现简单，只需使用AD转换测出电压即可，RAM消耗极低，不占用芯片资源，没有芯片平台的限制

### (2)硬件缺点

滤波电路复杂，不容易焊接，检修困难，对电路的设计者要求较高，元器件精度要求较高

### (3)软件优点

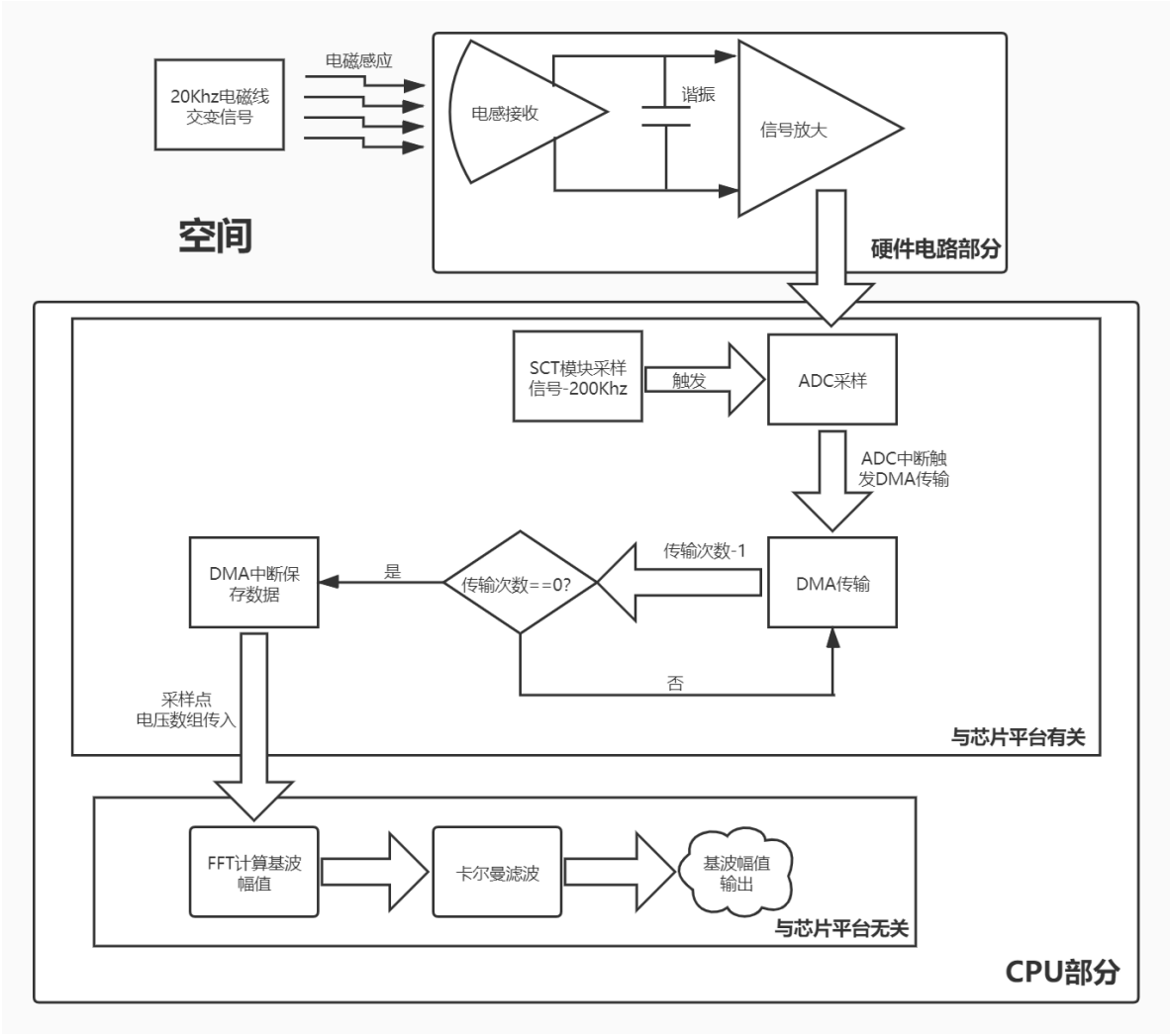
所需电路非常简单，且对电路的稳定性和元器件的精度要求没有硬件这么高，由于使用的是傅里叶算法，他只会提取到信号内的20Khz基波幅值，自动滤除其他的干扰信号，在底层编写正确的情况下，软件提取一次基波幅值只需115us左右，并不比硬件的时间长很多。最终的结果添加了卡尔曼滤波，保证了在信号不变，12bit的ADC最高精度下，最终结果波动不超过0.2%，且信号跟随性比较好，不同通道切换没有很大的干扰。

### (4)软件缺点

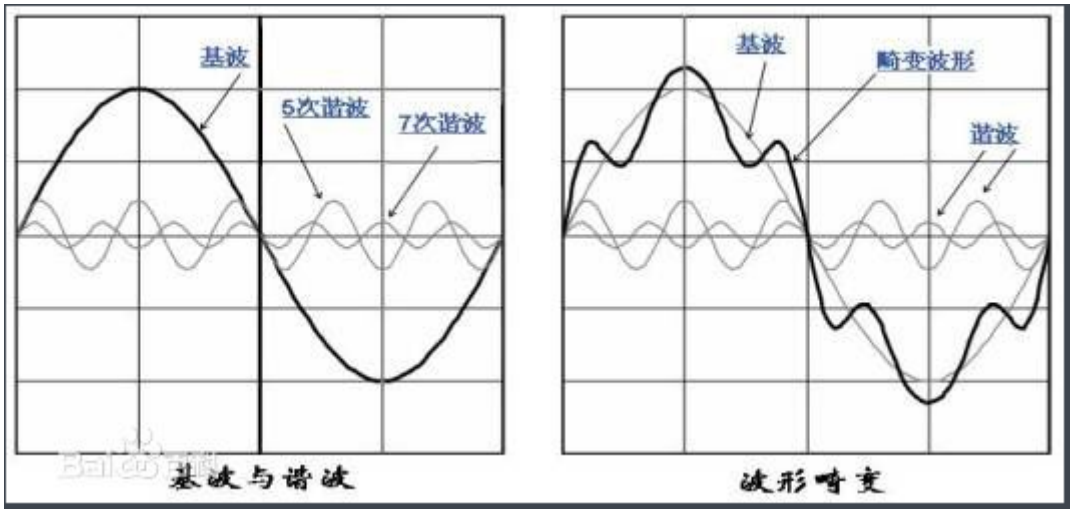
所需芯片RAM较高，且移植比较困难，由于使用的是DMA直接传输，较容易造成不明原因的数组越界，程序是否稳定需要比较多的测试，这也是为什么我写下这篇教程，是为了能让使用到它的同学能比较快的查找到错误原因，也能辅助使用者完善本程序。

# 二、原理

## 1、架构




## 2、FFT数学原理




假设被检测信号如以下公式所示

$$u=\sum_{n=1}^{\infty} A_n \cos\left(n\omega t+\varphi_n\right)$$

其中， An 、 nω、φn 为 **n 次谐波**的**幅值**、**角频率**、**初始相位**， ω =2π f ， f 为**基波频率**。

在离散系统中，若采样频率为 f s ，则采样周期 T s =1/ f s ，每一个基波周期的采样点数为 N = f s / f ,按照离散傅里叶变换的基本原理，在 t 时刻u(t)可以表示为以下公式，其中u(t)为某个采样时刻的电压值

$$u(t)=a_0+\sum_{n=1}^{\infty}a_n\cos(n\omega t)+\sum_{n=1}^{\infty}b_n\sin(n\omega t)$$

其中：

$$a_0=\frac{1}{N}\sum_{t=t_0}^{t_0+(N-1)T_s}u(t)$$

$$a_n=\frac{2}{N}\sum_{t=t_0}^{t_0+(N-1)T_s}u(t)\cos(n\omega t)$$

$$b_n=\frac{2}{N}\sum_{t=t_0}^{t_0+(N-1)T_s}u(t)\sin(n\omega t)$$

根据上述公式所示，以 n 次谐波电压为例，可得其电压的傅里叶级数表达式如以下公式所示

$$u_n(t)=a_n\cos(n\omega t)+b_n\sin(n\omega t)=$$

$$\sqrt{a_n^2+b_n^2}\left[\frac{a_n}{\sqrt{a_n^2+b_n^2}}\cos(\omega t)+\frac{b_n}{\sqrt{a_n^2+b_n^2}}\sin(\omega t)\right]=$$

$$\sqrt{a_n^2+b_n^2}[\cos\varphi_n\cos(n\omega t)-\sin\varphi_n\sin(n\omega t)]=$$

$$\sqrt{a_n^2+b_n^2}\cos(n\omega t+\varphi_n) \qquad (10)$$

其中 $A_n=\sqrt{a_n^2+b_n^2}$ 即为n次谐波电压的幅值

因此，cos( nωt )和sin(nωt )便可以转化为一个信号周期中的采样点。以基波信号为例，正弦sin( ωt)和余弦cos( ωt )分为 N 个采样点，由于sin( ωt )和cos( ωt )是以2π/ ω 为周期的函数，因此在 k = N +1 时，可令 k =1重新开始计数，从而使得sin( ωt )和cos( ωt )一直反复利用 N 点采样值循环计算。

**通俗来说**，如果需要计算一个20Khz正弦信号的幅值，只需采集到他的一个完整周期，例如10个采样点，那么这个信号的幅值即为

> 每个采样点的电压u(t)乘以sin(2πf*(t * Ts))，再乘以2/N-------得到an
>
> 每个采样点的电压u(t)乘以cos(2πf*(t * Ts))，再乘以2/N------得到bn

> (**f为基波频率**，这里为20Khz，**Ts为采样周期**，这里为1/200Khz，即为5us，**N为采样点数目**，这里为10)

最后根据公式 $A_n=\sqrt{a_n^2+b_n^2}$ 得到本信号的基波幅值An

由于公式中的2πf*(t * Ts)除了T是变量外，其他都为常量，且t是以0-10规律变化，因此可以提前计算好sin和cos的值，最后只需用每个采样点的电压乘以预先计算好的的sin和cos即可，以加快运算速度。

# 3、程序原理

## (1)向用户开放的接口函数

**------- Fourier_Init**

```
void Fourier_Init(DMACH_enum dmach,ADCCH_enum ch);
/**
 *基波幅值提取初始化
 *@param DMACH_enum dmach 设置不同ADC端口数据传输的DMA通道号,范围DMA_CH0~DMA_CH29
 *       ADCCH_enum ch       设置AD转换端口
 *@return 无
 *@example Fourier_Init(DMA_CH0,ADC_CH6_B0);  设置B0口的基波幅值提取方式，数据传输用DMA_CH0通道
 */
```

**------- Fourier_Once**

```
uint16 Fourier_Once(ADCCH_enum ch,ADCRES_enum resolution);
/**
 *计算一个通道的一次基波幅值
*@param   ADCCH_enum ch          设置AD转换端口
          ADCRES_enum resolution 设置AD转换位数，范围6位，8位，10位，12位
*@return  卡尔曼滤波以后的基波幅值
*@example FFT(Data_Buff) 计算Data_Buff数组内的基波幅值
*/
```

## (2)不向用户开放的内部函数（如果仅仅是使用无需关心内部如何实现）

**------- Fourier_dma_reload**

```
__STATIC_INLINE void Fourier_dma_reload(DMACH_enum dmach, void *SADDR, void *DADDR, uint16 count);
/**
 *DMA参数重载（内部函数，无需调用）
 *使用__STATIC_INLINE为了将这段函数内嵌到使用该函数的地方，这样可以减少函数调用的时间
 *@param DMACH_enum dmach 设置传输的DMA通道号,范围DMA_CH0~DMA_CH29
 *       void *SADDR             DMA传输源地址
         void *DADDR             DMA传输目的地址
         uint16 count            传输次数
 *@return 无
 *@example Fourier_dma_reload(DMA_CH0, (void*)&ADC0->SEQ_GDAT[0], (void*)&Buff[0],20); 将ADC结果寄存器内的数据传输至Buff数组内，循环20次
*/
```

**------- Fourier_adc_reload**

```
__STATIC_INLINE void Fourier_adc_reload(ADCCH_enum ch,ADCRES_enum resolution);
/**
 *ADC参数重载（内部函数，无需调用）
 *使用__STATIC_INLINE为了将这段函数内嵌到使用该函数的地方，这样可以减少函数调用的时间
 *@param ADCCH_enum ch       设置AD转换端口
 *       ADCRES_enum resolution 设置AD转换位数，范围6位，8位，10位，12位
 *@return 无
 *@example Fourier_adc_reload(ADC_CH6_B0,ADC_12BIT);设置B0口的AD转换参数，并设置为12位ADC
*/
```

**------- IRQ_Ctrl_ADC_DMA**

```
__STATIC_INLINE void IRQ_Ctrl_ADC_DMA(uint8 ONorOFF);
/**
 *中断控制（内部函数，无需调用）
 *使用__STATIC_INLINE为了将这段函数内嵌到使用该函数的地方，这样可以减少函数调用的时间
 *@param uint8 ONorOFF  中断开关，1为开，0为关，控制DMA和ADC中断
 *@return 无
 *@example IRQ_Ctrl_ADC_DMA(1); 打开ADC和DMA中断
*/
```

------- **Fourier_DMAReadBuff**

```
void Fourier_DMAReadBuff(ADCCH_enum ch,ADCRES_enum resolution,uint16 *Buff_Out);
/**
 *获取N此电压数据至指定数组内
*@param ADCCH_enum ch        设置AD转换端口
        ADCRES_enum resolution 设置AD转换位数，范围6位，8位，10位，12位
        uint16 *Buff_Out  传入N个采样点电压数据数组的指针
 *@return 无
 *@example Fourier_DMAReadBuff(ADC_CH6_B0,ADC_12BIT,Data_Buff); 将B0口的N个采样点的
12位电压数据传入Data_Buff数组内
*/
```

------- **FFT**

```
double FFT(uint16 *adc_data);
/**
 *FFT基波幅值提取函数
*@param uint16 *adc_data     传入保存了N个采样点的电压数据
 *@return 基波幅值，如果传入的电压数据为6位的 ，返回基波幅值范围为0-31
 *                如果传入的电压数据为8位的 ，返回基波幅值范围为0-127
 *                如果传入的电压数据为10位的，返回基波幅值范围为0-512
 *                如果传入的电压数据为12位的，返回基波幅值范围为0-2048
 *@example FFT(Data_Buff) 计算Data_Buff数组内的基波幅值
*/
```

------- **kalmanFilter**

```
float kalmanFilter(struct kalman_Data_t* KALMAN,float input);
/**
 *卡尔曼滤波器
 *@param struct kalman_Data_t* KALMAN 卡尔曼结构体参数
 *       float input 需要滤波的参数的测量值（即传感器的采集值）
 *@return 滤波后的参数（最优值）
*/
```

# 三、使用指南

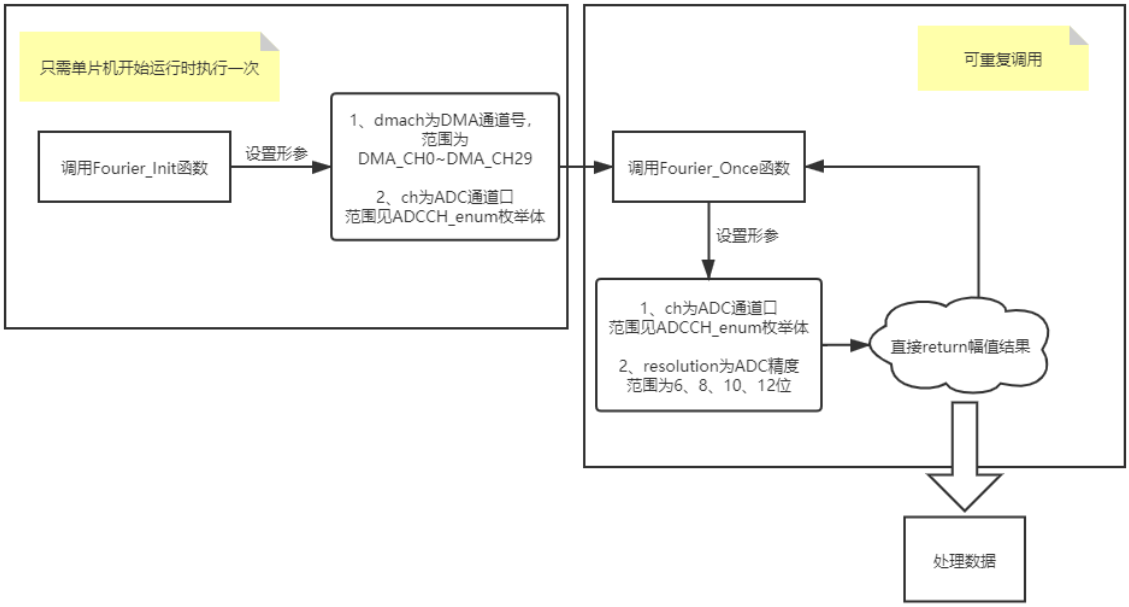
## 1、使用步骤（非常重要！！！！）

（1）将Fourier_DMA.c和Fourier_DMA.h文件添加进工程文件夹内，并且设置好头文件路径

（2）将Fourier_interrupt_Func()函数添加进isr.c文件内的中断服务函数DMA0_DriverIRQHandler

```
void DMA0_DriverIRQHandler(void)
{
        Fourier_interrupt_Func();
}
```

（2）看图使用




## 2、注意事项（非常重要！！！！）

（1）如果需要采样多个通道，最好将每个ADC通道分配不同的DMA传输通道，但是设置相同的通道也没有特别大的问题。

（2）程序使用和原来逐飞的LPC库里的ADC底层使用完全相同，除了这个在初始化的时候需要多配置一个DMA通道，因此你们自己的程序无需作任何改动，只需将原来的adc_init改成Fourier_Init，并添加一个DMA通道的形参，将原来的adc_convert替换成Fourier_Once，就可以直接使用。

（3）这里要注意一点，当ADC精度配置为6位时，原来的adc_convert返回的值范围是0-63，但是这个程序返回的值范围是原来的1/2，即为0-31，那么8位，10位，12位同理。--------至于原理，其实很好理解，由于输入的正弦波OFFSET是1.65V，峰峰值不超过3.3V，那么信号的幅值不会超过1.65V，1.65V电压如果单片机用6位精度采集的话就是31，8位精度的话是127，以此类推。

（4）Fourier_Once每调用一次读取数据的时间约为115us

（5）我写好的程序只适用于20Khz的信号处理，如果想要用于测量其他频率的信号，需要修改H文件内的几个宏定义，并且要修改C文件内的三角函数查表数组。

^^^^^^^^^^^^^^^^^^^^示例^^^^^^^^^^^^^^^^^^^^^^^^

```
//cos_sin_search[n][0]=(2/采样点数n)*cos(2*Pi*信号频率f*n*采样周期T)
//cos_sin_search[n][1]=(2/采样点数n)*sin(2*Pi*信号频率f*n*采样周期T)
float cos_sin_search[11][2]=        //正余弦查表，采样点10个，原信号频率20khz
{                                  //k=
  {  0.2000,  0.0000},             //0
  {  0.1618,  0.1175},             //1
  {  0.0618,  0.1902},             //2
  { -0.0618,  0.1902},             //3
  { -0.1618,  0.1175},             //4
```

```
  { -0.2000,  0.0000},          //5
  { -0.1618, -0.1175},          //6
  { -0.0618, -0.1902},          //7
  {  0.0618, -0.1902},          //8
  {  0.1618, -0.1175},          //9
  {  0.2000,  0.0000}           //10
};
```

# 四、问题汇总和部分解决办法

这里仅列举我之前遇到的问题和我的一些解决办法，不是很完全，还有其他超多的不明原因，希望大家能携手解决，如果能让程序变得越来越完善、稳定，可移植性越来越高，这个程序才能真正意义上被称为祖传程序。

## 1、读取数据时卡住

这个应该是困扰住我最长时间的问题了，至今我也没办法完全避免，究其已知的原因，目前暂时只可总结为以下几点：

（1）**FFT迭代运算导致数据溢出**

**原因**：由于这里的基波幅值运算是通过迭代的办法来运算的，我使用的采样是每个信号周期采样10个点，当每个点的精度是12位时，会对他进行累加运算，最终居然还要对他开平方，再开根号，这就导致了在对累加后的数据开平方时，保存值的变量有可能会溢出。

**解决办法**：

1）在FFT进行累加运算前尽量的缩小每次迭代的值的大小，具体办法是例如可以在做三角函数表的时候，对每个数据除以10，在最终开完平方并且开根号了以后，再乘以10，这样也许会减小迭代过程中的数据溢出可能性，但也会损失精度。

2）在H文件内，降低ADC_Samp_num，即每个周期的采样点数，可以直接降低迭代次数防止溢出，但是修改这里的话需要重新计算cos_sin_search三角函数查表，比较麻烦，且会降低精度，不建议使用，但他是一定能够解决数据溢出问题。

（2）**DMA传输目的寄存器在其他重要数据寄存器的附近**（主要原因）

**原因**：之前我是使用了结构体的方式来保存数据缓冲区，标志位，结果等等数据，由于结构体内各变量在单片机内部所占的是一片连续的存储空间。我的猜想是，DMA搬运数据也不是百分百正常，会出现偶尔的将数据传输至目标数组附近的一些数据寄存器内，这个时候如果旁边恰好是某个很重要的标志位，那么就会导致这个标志位被错误的置位或清零，会造成整个程序逻辑的错乱。例如程序中的START_Flag是一个非常重要的标志位，如果他没在该被清零的时候清零，会导致程序一直卡住，也是导致读取数据卡住的一个主要原因。

**解决办法**：

1) 将Fourier_Once函数内部最后返回的值改为Result_WithoutFilt，并且屏蔽滤波程序，看是否会再出现问题

2) 慎用结构体，需要在详细了解了结构体在内存中的占用方式后再去合理使用他，结构体固然好，但是也不能滥用，能用指针传递参数的时候就最好不要使用全局变量和结构体。

3) 在结构体内各个数据间添加空的变量空间来隔离数据。

```
struct Fourier_Data_t
{
```

```
    uint8 reserve0[100];              //保留位，防止附近寄存器溢出导致的数据异常

    DMACH_enum DMA_CH[14];  //保存每个ADC通道的DMA传输通道

    uint8 reserve1[100];              //保留位，防止附近寄存器溢出导致的数据异常

    ADCCH_enum ADCCH_Save;  //保存此时正在转换的通道

    uint8 reserve2[100];              //保留位，防止附近寄存器溢出导致的数据异常

    uint8 Busy_Flag;                  //全局忙标志

    uint8 reserve3[100];              //保留位，防止附近寄存器溢出导致的数据异常
}Fourier_Data;
```

例如上面的的结构体定义，在各个数据之间添加了reserve保留区的方式来隔离重要变量。

当然这是万不得已的笨办法，它会浪费RAM空间，不到迫不得已最好不要使用这种方法。

4) 有时会有出现局部变量的溢出也会导致卡住，这个问题的原因我到现在还没有搞明白，暂且把他归为玄学哈哈哈哈，这也是为何我需要你们帮助我做大量的测试。

## 2、数据有延迟或抖动厉害

**原因**：滤波算法或卡尔曼滤波的参数问题

本来这个最后的滤波我是选用滑动滤波的，但是鉴于滑动滤波所占RAM比较大，而这个程序又对数据溢出异常敏感，因此最终我选择了卡尔曼滤波，这是一个非常优秀的滤波算法。

**解决办法**：

改动Fourier_Init函数内初始化的参数kalman_Data[ch].R，

这个参数调大了会造成数据跟随性变差，简单来说数据会有延迟，但是抖动会变弱，但是参数调小了数据跟随性会变好，但是抖动会变得厉害。

我调出的值是0.2，也并不是仔细调整出来的，你们可以根据实际需要改动参数，