**GRBL源代码分析**

https://blog.csdn.net/xufeixueren/article/details/79663068/

这段日子喜事连连，暂时把写博客的事情放下了，有时候想想好久没有写博客了，要不要写点啥呢。转念一想，好像也没有啥值得写的心得体会，加上最近忙着结婚的事情，也就把写博客的事搁置了。周五本来是要上班的，但是公司大厦供电系统维护，所以调休一天。借着安静的周五，加上最近一个多月研究GRBL源代码的心得，写下这篇博客，供后来者参考学习。网上关于GRBL源代码分析的资料几乎找不到，这篇博客里的内容大多是自己对源代码的理解，也有一部分是QQ群里的同行的心得。博客里没有写到，或者写的不对的地方，欢迎大家留言共同探讨，共同进步。

GRBL的核心是带有梯形加减速过程的DDA直线插补算法的实现，整个GRBL源代码中包含了以下内容：

（1）串口中断接收上位机的指令，包括自定义的系统命令和G代码指令；

（2）串口指令解析，自定义的系统命令直接执行，G代码指令调用相关操作，这里只关注直线段、圆弧指令的解析；

（3）圆弧拆分成直线段进行插补的方法；

（4）多条直线段之间转角速度优化的前瞻速度控制的方法；

（5）单条线段梯形加减速过程换算成定时器定时不同时间长短来输出脉冲的方法；

（6）限位条件的判断及轴自动归位的方法；

（7）其它spindle、coolant接口等，这些我也不知道干嘛用的，推测是给用户二次开发预留的接口。

下面详细介绍这些模块的实现细节，串口接收、指令解析这些很容易理解，就简单说说；spindle、coolant这些还没有琢磨透，好像用到的情况也不多，不做描述；重点还是关注圆弧拆分线段、转角速度优化、线段梯形加减速插补和自动归位的实现方法。

一、串口接收

串口配置成中断接收和中断发送模式，并创建了串口接收环形队列和串口发送环形队列，中断接收的数据存放在串口接收环形队列里，串口需要发送的数据放到串口发送环形队列里。当串口产生中断时，如果接收中断标志位置位，说明接收到数据，把数据读出放到接收队列，如果发送中断标志置位，说明发送寄存器空，把发送队列里的数据写入发送寄存器里。这就是串口处理的流程，具体的实现难度不大，就不详细解释源码了 。

二、串口接收到的数据解析

串口接收到的数据放在串口接收环形队列里，主程序里每次从接收队列里读出一个字节，以'\r'或'\n'为标志截取一行完整的指令，如果指令以'$'开始，说明是自定义的系统指令，其它的是G代码指令。如果是系统自定义指令，就调用uint8\_t system\_execute\_line(char \*line)函数，里面包含了一些读取软件版本信息、读取默认配置参数信息和把外部设置的参数信息写入eeprom里，轴归位操作等功能。如果是G代码指令，就调用uint8\_t gc\_execute\_line(char \*line)函数，里面包含了很多G代码的指令解析过程，由于重点放在研究直线插补算法上，没有对G代码解析源码深入分析，这里只关注里面的执行圆弧和直线插补的代码，即只需要关心void mc\_line(float \*target, float feed\_rate, uint8\_t invert\_feed\_rate)和void mc\_arc(float \*position, float \*target, float \*offset, float radius, float feed\_rate,uint8\_t invert\_feed\_rate, uint8\_t axis\_0, uint8\_t axis\_1, uint8\_t axis\_linear)两个函数即可。

三、圆弧拆分成多线段

GRBL中把圆弧拆分成多条逼近的直线段，然后对直线段进行插补，这种方法也就是俗称的把复杂曲线拆分成多条逼近的直线的插补方法。圆弧拆分成直线段的方法，在void mc\_arc(float \*position, float \*target, float \*offset, float radius, float feed\_rate,uint8\_t invert\_feed\_rate, uint8\_t axis\_0, uint8\_t axis\_1, uint8\_t axis\_linear)函数里实现，下面对该函数进行详细介绍：

position，圆弧起始点位置坐标，为了后面解释方便，这里设为(x0,y0,z0)

target，圆弧终点坐标，这里设为(x1,y1,z1)

offset，圆心相对于起始点的偏移向量，这里设为(rx,ry,yz)，那么圆心坐标为(x0+rx,y0+ry,z0+rz)

radius，圆弧半径长度

feed\_rate，轴的进给速率

invert\_feed\_rate，进给速率含义标志位，这里默认为零，表示进给速率的单位是min/mm，即分钟/毫米

axis\_0，圆弧所在平面的第一个轴，可以是x/y/z中任意一个

axis\_1，圆弧所在平面的第二个轴，可以是x/y/z中任意一个

axis\_linear，除了圆弧平面之外的第三个轴，即与圆弧平面垂直的轴

void mc\_arc(float \*position, float \*target, float \*offset, float radius, float feed\_rate,

uint8\_t invert\_feed\_rate, uint8\_t axis\_0, uint8\_t axis\_1, uint8\_t axis\_linear)

{

//圆弧所在平面的圆心坐标

float center\_axis0 = position[axis\_0] + offset[axis\_0];

float center\_axis1 = position[axis\_1] + offset[axis\_1];

//圆心指向圆弧起始点的向量坐标

float r\_axis0 = -offset[axis\_0];

float r\_axis1 = -offset[axis\_1];

//圆心指向圆弧终点的向量坐标

float rt\_axis0 = target[axis\_0] - center\_axis0;

float rt\_axis1 = target[axis\_1] - center\_axis1;

//计算圆心到圆弧起始点向量b(r\_axis0 ,r\_axis1)和圆心到圆弧终点向量c(rt\_axis0 ,rt\_axis1)的夹角的正切值，注意夹角a的方向是起始点向量b逆时针转向终点向量c的角，这个在后面判断角度值符号时会用到。根据向量夹角余弦公式，推出向量夹角正切公式，套入向量b和向量c的坐标，即可得出tana。下面公式中的atan2是反正切函数，a=atan2(y,x)，即角度a=y/x的正切值

float angular\_travel = atan2(r\_axis0\*rt\_axis1-r\_axis1\*rt\_axis0, r\_axis0\*rt\_axis0+r\_axis1\*rt\_axis1);

if (gc\_state.modal.motion == MOTION\_MODE\_CW\_ARC) {

//如果圆弧顺时针移动，角度应该是负值，如果计算出的角度为正值，需要在计算出的角度基础上减去2\*pi（pi为圆周率）

if (angular\_travel >= 0) { angular\_travel -= 2\*M\_PI; }

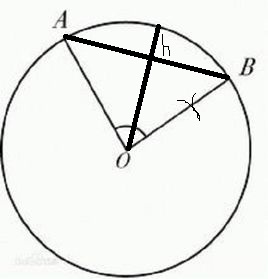
} else {

//如果圆弧逆时针移动，角度应该是正值，如果计算出的角度为负值，需要在计算出的角度基础上加上2\*pi（pi为圆周率）

if (angular\_travel <= 0) { angular\_travel += 2\*M\_PI; }

}

//计算起点到终点的圆弧可以划分多少条小线段，计算方法：总共的弧长/每条小线段的长度，angular\_travel是圆弧的弧度，radius是圆弧的半径，那么它们的乘积angular\_travel\*radius就是圆弧的弧长，再乘以0.5就是弧长的一半。settings.arc\_tolerance是圆弧上两点之间连接的小线段到这段圆弧的最大距离，即圆弧上的小线段到弧顶的最大距离，这里设为h，



，有图可知，假设线段|AB|长度的一半为k，那么有勾股定理可知，r\*r=k\*k+(r-h)\*(r-h)。知道了r和h，那么k\*k=h\*(2\*r-h)。这样总共的小线段个数也就出来了，这就是下面这个公式的含义。

uint16\_t segments = floor(fabs(0.5\*angular\_travel\*radius)/

sqrt(settings.arc\_tolerance\*(2\*radius - settings.arc\_tolerance)) );

}

......

//这里是计算圆心与每条小线段所夹的角T，即上图中角AOB的余弦值和正弦值，由于角度很小，这里采用了三角函数的泰勒级数展开公式计算cosT和sinT。cosT的二阶泰勒级数为1-T\*T/2，sinT的三阶泰勒级数为T-T\*T\*T/6，为了计算方便，cos\_T 被放大了两倍，后面又乘上了0.5复原了。有cosT和sinT的泰勒级数公式，cosT被放大了两倍，可以推出sinT=T\*(4+cosT)/6，即下面计算sinT的公式的来源。

float cos\_T = 2.0 - theta\_per\_segment\*theta\_per\_segment;

float sin\_T = theta\_per\_segment\*0.16666667\*(cos\_T + 4.0);

cos\_T \*= 0.5;

//循环累加每一条小线段，圆的极坐标公式为x=r\*cosa,y=r\*sina，假设当前线段的起始坐标为(rcosa,rsina),下一条线段比当前线段移动的角度已知为T，那么下一条线段的起始坐标为(rcos(a+T),rsin(a+T)),运算得到rcos(a+T)=r\*cosa\*cosT-r\*sina\*sinT，rsin(a+T)=r\*sina\*cosT+r\*cosa\*sinT。由于我们知道当前线段的坐标为(r\_axis0,r\_axis1)，又知道sinT和cosT的值，下一条线段的起始坐标根据公式可立即求出。

for (i = 1; i<segments; i++) {

if (count < N\_ARC\_CORRECTION) {

//这就是用上述公式求下一条线段的起始坐标的具体计算。当累计计算线段数超过N\_ARC\_CORRECTION个数时，要调用N\_ARC\_CORRECTION个线段总共转动的角度来计算最终的坐标移动值，这样可以消除因每次使用sinT、cosT的近似泰勒值运算的累积误差

r\_axisi = r\_axis0\*sin\_T + r\_axis1\*cos\_T;

r\_axis0 = r\_axis0\*cos\_T - r\_axis1\*sin\_T;

r\_axis1 = r\_axisi;

count++;

} else {

//调用N\_ARC\_CORRECTION个线段总共转动的角度来计算最终的坐标移动值

cos\_Ti = cos(i\*theta\_per\_segment);

sin\_Ti = sin(i\*theta\_per\_segment);

r\_axis0 = -offset[axis\_0]\*cos\_Ti + offset[axis\_1]\*sin\_Ti;

r\_axis1 = -offset[axis\_0]\*sin\_Ti - offset[axis\_1]\*cos\_Ti;

count = 0;

}

// 计算出下一条线段的起始坐标，也就是当前线段的终点坐标，前面已知当前线段的起始坐标，这样就可以把线段的坐标传递给直线插补函数mc\_line进行线段插补了

position[axis\_0] = center\_axis0 + r\_axis0;

position[axis\_1] = center\_axis1 + r\_axis1;

position[axis\_linear] += linear\_per\_segment;

//线段插补

mc\_line(position, feed\_rate, invert\_feed\_rate);

......

}

}

//把圆弧终点作为最后一条线段的终点坐标进行直线插补，确保圆弧上所有的点包含进了直线里

mc\_line(target, feed\_rate, invert\_feed\_rate);

}

四、多线段速度规划前瞻算法

连续执行多条线段插补的时候，为了加快轴的移动速度，执行完一条直线指令后不能停下来，然后重新启动执行下一条直线指令。而是需要保持一定的速度去执行下一条直线插补，但是由于相邻两条直线之间有一定的夹角，导致转弯的时候，轴的速度不能过快，还要考虑两条直线执行的最大速度限制和直线头尾速度衔接等问题，这些问题的处理方法就是前瞻算法。

GRBL中使用了环形队列的方式存储每一条直线段的信息，这个队列的名称是block\_buffer[BLOCK\_BUFFER\_SIZE]，这个结构体数组里存放的是线段的初速度、最大初速度限制、最大转角速度限制、正常运行速度、加速度、线段长度信息。当G代码解析出一条线段指令或者圆弧拆分出线段后，调用void mc\_line(float \*target, float feed\_rate, uint8\_t invert\_feed\_rate)函数，把当前线段的信息存入block\_buffer队列中，然后把当前线段和队列里前一条线段结合在一起，用前瞻算法修正队列里前一条线段的最大运行速度，以便保证在前一条线段执行结束时的速度与当前线段的初速度一致。另外，根据两条线段的夹角确定最大转角速度，用于修正前一条线段的结束速度和当前线段的初速度不能超过最大转角速度。下面对mc\_line函数进行详细分析：

target，线段移动到的最终位置，单位是mm，也就是说当前线段移动的长度是target值减去之前的所有线段的移动长度；

feed\_rate，线段的最大运行速度，梯形加减速值是提前设定好保存在eeprom里的；

invert\_feed\_rate，线段运行速度含义标志位，feed\_rate有多种含义，这里我们只了解feed\_rate的单位是min/mm即可；

void mc\_line(float \*target, float feed\_rate, uint8\_t invert\_feed\_rate)

{

//如果限位使能，就检查target值是否超出了轴能到达的最远位置，如果超出了就告警限位错误，并复位系统

if (bit\_istrue(settings.flags,BITFLAG\_SOFT\_LIMIT\_ENABLE)) { limits\_soft\_check(target); }

......

do {

//这个函数的功能很多，有很多地方都会调用它，这里调用的目的是判断有没有系统异常发生，比如系统告警或复位，如果有异常，这个函数里处理异常的代码就会执行，没有就退出函数，继续运行

protocol\_execute\_runtime(); // Check for any run-time commands

if (sys.abort) { return; } // Bail, if system abort.

//检查block\_buffer是否满，如果满就执行尝试打开线段插补执行使能开关。如果系统配置里开启了auto-cycle功能，就可以自动开始执行线段插补，block\_buffer里的线段会被系统执行插补操作而空出一些空间，这样队列就不满了，也就退出下面的do-while循环继续执行代码

if ( plan\_check\_full\_buffer() ) { protocol\_auto\_cycle\_start(); } // Auto-cycle start when buffer is full.

else { break; }

} while (1);

//把当前线段的的信息添加到block\_buffer队列里，这个函数里包含了前瞻算法的处理过程，在下面会详细介绍

plan\_buffer\_line(target, feed\_rate, invert\_feed\_rate);

......

}

void plan\_buffer\_line(float \*target, float feed\_rate, uint8\_t invert\_feed\_rate)

{

......

for (idx=0; idx<N\_AXIS; idx++) {

//target是轴移动的距离，单位是毫米，系统设定了steps\_per\_mm值，也就是每毫米代表的轴移动步数，直接换算得到轴移动步数target\_steps

target\_steps[idx] = lround(target[idx]\*settings.steps\_per\_mm[idx]);

//target表示轴从原点移动到终点的总距离，所以当前线段的移动步数需要用target减去之前所有线段移动的总步数

block->steps[idx] = labs(target\_steps[idx]-pl.position[idx]);

//获得三个轴里移动距离最远的轴移动的距离，后面DDA直线插补时会用到这个值。关于DDA插补算法的方法后面会介绍

block->step\_event\_count = max(block->step\_event\_count, block->steps[idx]);

//根据步数换算出真实移动的距离，保存在unit\_vec中，后面计算两条线段夹角时会用到

delta\_mm = (target\_steps[idx] - pl.position[idx])/settings.steps\_per\_mm[idx];

unit\_vec[idx] = delta\_mm;

// 这个值小于零，说明这个轴需要向与原来方向相反的方向移动

if (delta\_mm < 0 ) { block->direction\_bits |= get\_direction\_pin\_mask(idx); }

//三个轴是正交的，知道了每个轴移动的距离，那么线段在空间里移动的真实距离是s\*s=x\*x+y\*y+z\*z，这个值在后面也会用到

block->millimeters += delta\_mm\*delta\_mm;

}

//开平方求出线段空间里移动的距离

block->millimeters = sqrt(block->millimeters);

......

for (idx=0; idx<N\_AXIS; idx++) {

if (unit\_vec[idx] != 0) {

//这里是为计算两条线段的夹角做准备工作，为了便于理解，这里假设当前线段的坐标是向量c=(x2,y2,z2)，上一条线段的坐标向量d=(x1,y1,z1) ,它们的空间向量长度分别是s2和s1，那么有坐标正交可得，s2\*s2=x2\*x2+y2\*y2+z2\*z2，s1\*s1=x1\*x1+y1\*y1+z1\*z1，inverse\_millimeters表示1/s，uint\_vec[0/1/2]分别表示x/y/z，经过运算后uint\_vec[0/1/2]表示的是x/s,y/s或者z/s，那么inverse\_unit\_vec\_value 表示s/x，s/y或者s/z的绝对值

unit\_vec[idx] \*= inverse\_millimeters;

inverse\_unit\_vec\_value = fabs(1.0/unit\_vec[idx]);

//配置里的max\_rate是表示线段向量每个分坐标的最大速度限制，feed\_rate是线段向量合成后的速度，所以feed\_rate要与每个轴分向量最大速度换算成合成后的最大速度比较，然后取最小值作为最终的线段运行最大速度，最大加速度也是用同样的方法进行比较

feed\_rate = min(feed\_rate,settings.max\_rate[idx]\*inverse\_unit\_vec\_value);

block->acceleration = min(block->acceleration,settings.acceleration[idx]\*inverse\_unit\_vec\_value);

//计算两条线段的夹角余弦值，夹角余弦公式cosa=(x1\*x2+y1\*y2+z1\*z2)/(s1\*s2)，因为两条线段是首尾相连，那么用两条线段的向量坐标计算出来的夹角其实是它的补角，夹角和它补角的余弦值刚好取负值即可，所以下面计算夹角余弦的方法里多了一个负号

junction\_cos\_theta -= pl.previous\_unit\_vec[idx] \* unit\_vec[idx];

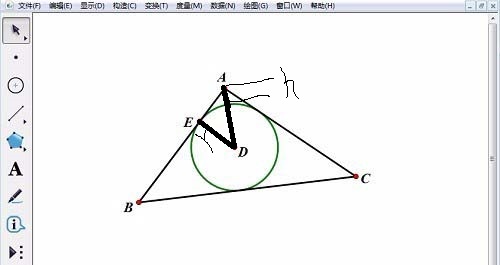
}

}

//这个用半角公式sin(a/2)=sqrt((1-cosa)/2)直接运算

float sin\_theta\_d2 = sqrt(0.5\*(1.0-junction\_cos\_theta));

// 计算转角最大速度v，有圆弧加速度公式可知，v\*v=a\*r,其中a是圆弧向心加速度，这里近似值为block->acceleration，r是圆弧的半径。settings.junction\_deviation是两条线段内切圆弧到两条线段交点的距离，这里设为h，如下图所示，角EAD即为上面的a/2，内切圆的半径为r，那么AD的长度即为r+h，那么sin\_theta\_d2=r/(r+h)，已知h的值，那么r=h\*sin\_theta\_d2/(1-sin\_theta\_d2)，那么套入v\*v=a\*r即可求得v\*v，即block->max\_junction\_speed\_sqr的值。



，

block->max\_junction\_speed\_sqr = max( MINIMUM\_JUNCTION\_SPEED\*MINIMUM\_JUNCTION\_SPEED,

(block->acceleration \* settings.junction\_deviation \* sin\_theta\_d2)/(1.0-sin\_theta\_d2) );

......

//对没有优化过的线段进行优化

planner\_recalculate();

}

static void planner\_recalculate()

{

......

//如果所有的线段都已经优化过了，直接退出函数

if (block\_index == block\_buffer\_planned) { return; }

......

//当前线段的起始速度取最大起始限制速度与末速度为零反推的最大起始速度的最小值

current->entry\_speed\_sqr = min( current->max\_entry\_speed\_sqr, 2\*current->acceleration\*current->millimeters);

......

//这段代码的含义是从当前线段往前推，直到所有的线段都优化过退出循环，即每条线段的初速度不能超过线段设置的最大初速度的限制

while (block\_index != block\_buffer\_planned) { ......}

......

//这段代码的含义是从第一个没有优化过的线段往前，直到到达当前线段时退出循环，即每条线段的末速度不能超过下一条线段的初速度，这样多条线段才能保持连续的速度运行

while (block\_index != block\_buffer\_head) {}

}

到这里多线段速度前瞻规划已经完成了，下面分析GRBL中怎么把带有加减速的线段转化成用定时器输出脉冲的过程。

五、线段转化成不同频率的输出脉冲

第四节里block\_buffer队列里存放的就是每条线段的详细信息，根据线段的初速度、末速度、加速度和线段距离信息，计算出这条线段运行时轴需要移动的总步数。假定把这条线段总的运行时间截取成多个微小的时间段DT，即可求出每个DT时间段内的平均速度，同时可以求出DT时间段内轴移动的步数n，这样就可以求出DT内每步需要的时间dt=DT/n，把dt设定为定时器定时间隔，直到中断计数次数到达n结束。由于线段总步数选取的是线段向量坐标(x,y,z)里的最大值，所以定时器中断里不能每次都输出脉冲，而是需要用DDA插补算法运算出每次中断哪个轴需要输出脉冲。

这里介绍一下GRBL中用到的DDA算法的实现过程，假设线段向量坐标a(x,y,z)，选取x,y,z绝对值最大的作为累加溢出值c=|max(x,y,z)|，假定累加初值b=c/2，那么三个轴输出脉冲的DDA算法如下：

m=l=k=b;

for(i=0;i<c;i++)

{

m+=x;

l+=y;

k+=z;

if(m>=c)

{

x轴输出一个脉冲；

m-=c;

}

if(l>=c)

{

y轴输出一个脉冲；

l-=c;

}

if(k>=c)

{

z轴输出一个脉冲；

k-=c;

}

}

下面开始进行源码分析，第四节里对每个线段进行预处理之后，进入主循环里protocol\_auto\_cycle\_start()函数和protocol\_execute\_runtime()函数，我们从protocol\_auto\_cycle\_start()函数开始分析。

//当GRBL默认配置里使能了auto\_start功能，就把系统执行标志sys.execute设置为EXEC\_CYCLE\_START，也就是系统可以自动执行线段输出脉冲，如果没有使能auto\_start，在系统运行过程中可以用串口命令手动开启线段输出脉冲

void protocol\_auto\_cycle\_start() { if (sys.auto\_start) { bit\_true\_atomic(sys.execute, EXEC\_CYCLE\_START); } }

void protocol\_execute\_runtime()

{

//省略了一些系统告警标志位的处理

......

//当系统没有使能线段输出脉冲功能时，只调用st\_prep\_buffer

if (rt\_exec & EXEC\_FEED\_HOLD)

{

.....

st\_prep\_buffer();

......

}

if (rt\_exec & EXEC\_CYCLE\_START)

{

......

//修改系统状态为STATE\_CYCLE，这样启动定时器后下次循环就不会在进入这里重复启动定时器了

sys.state = STATE\_CYCLE;

//把线段换算成定时器输出脉冲频率和脉冲个数

st\_prep\_buffer();

//启动定时器开始输出脉冲

st\_wake\_up();

......

}

......

//当定时器已经启动后，以后的循环就是不断的把线段换算成定时器输出脉冲

if (sys.state & (STATE\_CYCLE | STATE\_HOLD | STATE\_HOMING)) { st\_prep\_buffer(); }

}

void st\_prep\_buffer()

{

//线段拆分成多个DT时间片，每个时间片轴运行的总步数和每步需要的时间存放在segment\_buffer队里中，while循环判断这个队列是否满，如果满了就退出循环，没满继续把线段拆分的时间片存入队列

while (segment\_buffer\_tail != segment\_next\_head) {

//判断当前线段拆分时间片是否完成，如果没有完成，pl\_block不为空，if里的语句不会被执行。如果pl\_block为空，说明当前线段时间片拆分完成，执行if里的语句，开始把下一条线段的信息读出来进行时间片拆分

if (pl\_block == NULL) {

//从block\_buffer队列中获取一条新的线段

pl\_block = plan\_get\_current\_block();

......

//开辟新的队列st\_block\_buffer存放新线段拆分时间片计算过程数据

st\_prep\_block = &st\_block\_buffer[prep.st\_block\_index];

//记录线段每个轴的运行方向

st\_prep\_block->direction\_bits = pl\_block->direction\_bits;

//AMASS功能用于平滑脉冲频率太慢的线段，如果AMASS功能使能，线段步数放大MAX\_AMASS\_LEVEL倍，但是定时器定时间隔将会缩短，相当于定时器中断加快了，更多次的中断累积才输出一个脉冲，这样输出脉冲变得更平滑了

#ifndef ADAPTIVE\_MULTI\_AXIS\_STEP\_SMOOTHING

st\_prep\_block->steps[X\_AXIS] = pl\_block->steps[X\_AXIS];

st\_prep\_block->steps[Y\_AXIS] = pl\_block->steps[Y\_AXIS];

st\_prep\_block->steps[Z\_AXIS] = pl\_block->steps[Z\_AXIS];

st\_prep\_block->step\_event\_count = pl\_block->step\_event\_count;

#else

st\_prep\_block->steps[X\_AXIS] = pl\_block->steps[X\_AXIS] << MAX\_AMASS\_LEVEL;

st\_prep\_block->steps[Y\_AXIS] = pl\_block->steps[Y\_AXIS] << MAX\_AMASS\_LEVEL;

st\_prep\_block->steps[Z\_AXIS] = pl\_block->steps[Z\_AXIS] << MAX\_AMASS\_LEVEL;

st\_prep\_block->step\_event\_count = pl\_block->step\_event\_count << MAX\_AMASS\_LEVEL;

#endif

//inv\_2\_accel 是加速度a的过程值，inv\_2\_accel=1/2a

float inv\_2\_accel = 0.5/pl\_block->acceleration;

if (sys.state == STATE\_HOLD) {

//STATE\_HOLD就是线段没有开始输出脉冲，这样线段的末速度为零，整个线段都是减速过程，后面分析了线段输出脉冲的过程，再回头看这里就很简单了

......

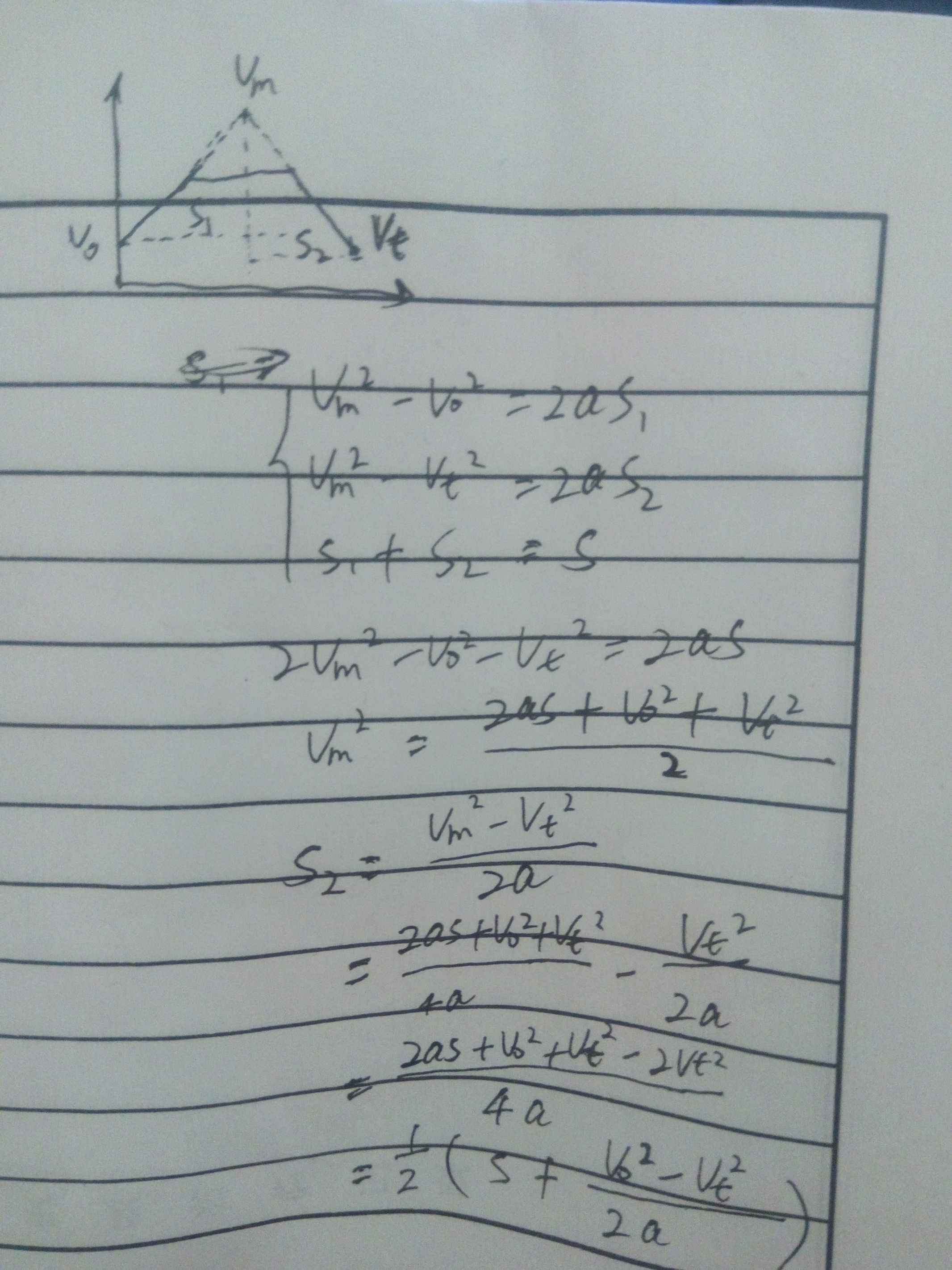
}

else

{

......

//intersect\_distance是线段加速和减速过程速度最大值交点离线段末尾的距离s2，如下图所示：



如图中所示，假设这条线段的长度s刚好只有加速和减速过程，最大速度为vm，已知线段的初速度为v0，线段的加速度为a，线段的末速度为vt，根据公式可求出当前假设条件下交点vm到线段末尾的距离s2，即intersect\_distance

float intersect\_distance =

0.5\*(pl\_block->millimeters+inv\_2\_accel\*(pl\_block->entry\_speed\_sqr-exit\_speed\_sqr));

if (intersect\_distance > 0.0) {

if (intersect\_distance < pl\_block->millimeters){

//计算线段最大限制速度减速到末速度需要的减速距离

prep.decelerate\_after = inv\_2\_accel\*(pl\_block->nominal\_speed\_sqr-exit\_speed\_sqr);

if (prep.decelerate\_after < intersect\_distance) {

......

//如果初速度等于最大限制速度，那么线段只有匀速和减速过程

if (pl\_block->entry\_speed\_sqr == pl\_block->nominal\_speed\_sqr) {

}

else

{

//线段有加速、匀速和减速过程

......

}

else{

//减速距离大于交点距离，说明线段设置的最大限制速度大于交点处的最大速度，那么线段只有三角形状的加速和减速过程，没有匀速过程

......

}

}

else{

//交点到末尾的距离大于线段总长度，说明只有线段只有减速过程

......

}

}

else{

//交点到末尾的距离小于零，说明加减速没有交点，即线段只有加速过程

......

}

}

}

......

//设定了时间片为DT\_SEGMENT，上面我们已经知道了线段的加减速过程，通过下面的do-while循环，计算出每个时间片线段的平均速度和移动的步数，由于它们的值是根据线段的加减速计算得来的，所以里面隐含了脉冲发送频率的信息。注意：每个segment\_buffer队列里的数据的时间粒度一般都是DT\_SEGMENT，也就是说假设一个时间片内，加速移动的距离是s1，但是这时候加速完成了，而加速过程所用的时间dt小于DT\_SEGMENT，那么时间片剩余的时间DT\_SEGMENT-dt，必须用来计算线段在剩余时间里匀速或减速移动的距离s2，那么在DT\_SEGMENT时间片内移动的总距离是s1+s2。另外，要注意由于脉冲频率太低，导致在时间片DT\_SEGMENT内移动的距离可能为零，为了防止这种情况发生，倍数放大DT\_SEGMENT的值，直到至少移动的距离大于一步为止。

do {

......

}while (mm\_remaining > prep.mm\_complete);

}

......

//计算出时间片内移动的步数

prep\_segment->n\_step = last\_n\_steps\_remaining-n\_steps\_remaining;

......

//计算出每步需要的时间

float inv\_rate = dt/(last\_n\_steps\_remaining - steps\_remaining);

......

//每步需要的时间设置为定时器的定时时间间隔

uint32\_t cycles = ceil( (TICKS\_PER\_MICROSECOND\*1000000\*60)\*inv\_rate );

......

//如果使能AMASS，定时器周期缩短，移动步数放大

cycles >>= prep\_segment->amass\_level;

prep\_segment->n\_step <<= prep\_segment->amass\_level;

......

}

调用void st\_wake\_up() 使能定时器1后，就会进入定时器1中断处理函数，在函数里使用DDA插补算法，输出脉冲

ISR(TIMER1\_COMPA\_vect)

{

......

//使能定时器0，用于恢复脉冲输出管脚的电平。例如定时器1里面拉高了管脚，表示输出了一个脉冲，这时候需要定时器0把管脚复位为低，便于定时器1输出下一个脉冲

TCNT0 = st.step\_pulse\_time;

TCCR0B = (1<<CS01);

......

//st.exec\_segment 为空，表示从segment\_buffer队列里取出的段已经输出脉冲完成，调用if里的语句获取segment\_buffer队列里的下一个执行时间片，主要是获取该时间片里每个轴移动的步数、插补步数，然后重新设置定时器1的定时周期

if (st.exec\_segment == NULL) {......}

......

//对x步数进行DDA累加溢出操作

#ifdef ADAPTIVE\_MULTI\_AXIS\_STEP\_SMOOTHING

st.counter\_x += st.steps[X\_AXIS];

#else

st.counter\_x += st.exec\_block->steps[X\_AXIS];

#endif

//如果溢出，管脚输出一个高电平，表示输出一个脉冲，没有溢出，管脚不输出电平

if (st.counter\_x > st.exec\_block->step\_event\_count) {

st.step\_outbits |= (1<<X\_STEP\_BIT);

st.counter\_x -= st.exec\_block->step\_event\_count;

if (st.exec\_block->direction\_bits & (1<<X\_DIRECTION\_BIT)) { sys.position[X\_AXIS]--; }

else { sys.position[X\_AXIS]++; }

}

......

}

到这里第五节终于讲完了，最难的部分也已经过去，下面说说归为操作的过程。

六、轴归位操作

轴归位操作就是轴以一定的速度进行最大距离的线段插补操作，插补过程中遇到限位信号的时候强行终止插补操作，反复多次进行这种插补后，轴就停在了限位点附近了。

void mc\_homing\_cycle()

{

......

//归位操作

limits\_go\_home(HOMING\_CYCLE\_0);

......

}

void limits\_go\_home(uint8\_t cycle\_mask)

{

......

//循环n\_cycle次进行直线插补，直线距离设置为轴允许的最大移动距离，插补过程中遇到限位信号强行复位系统来终止插补操作，并记录每次轴移动到限位信号的实际距离，通过改变每次插补的速度，计算出最合理的实际距离

do {

......

} while (n\_cycle-- > 0);

......

//最后用最合理的插补距离和插补速度把轴移动到归位的位置

plan\_buffer\_line(target, settings.homing\_seek\_rate, false);

......

}

经过几天的奋战，终于写完了这篇博客，长舒一口气。

————————————————

版权声明：本文为CSDN博主「xufeixueren」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/xufeixueren/article/details/79663068/