**GRBL五：定时器控制策略解析**

本文讲述的是根据grbl原始程序两个定时器控制脉冲输出的方式，由于在stm32上有PWM输出功能，所以可以用一个定时器的方式输出PWM来控制脉冲(因为pwm的占空比不影响步进速度，只需要固定占空就好，值修改脉冲周期就OK了)，可以节约一个定时器，关于这种设计方式我只是有这个想法，本文未作实践。

GRBL的脉冲输出靠两个定时器协同控制输出的

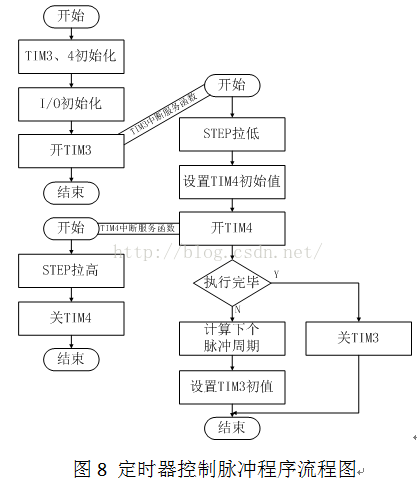
具体控制策略：第一个定时器控制脉冲周期(因为步进电机脉冲周期决定速度)

第二个定时器控制一个周期中低电平的时间(脉冲宽度不重要，只要CPU能检测的到就好)，相当于延时

流程如下图所示

TCNT0 🡪 TIM4控制脉宽

TCNT1 🡪 TIM3控制周期



所有代码都在stepper中定义

第一步：初始化引脚和定时器

在初始化定时器中没有给两个定时器设置分频和初值，也没有打开定时器开关

用的STM32通用16位定时器3和4

// Initialize and start the stepper motor subsystem

void st\_init()

{

// Configure directions of interface pins

// STEPPING\_DDR |= STEPPING\_MASK;

// STEPPING\_PORT = (STEPPING\_PORT & ~STEPPING\_MASK) | settings.invert\_mask;

// STEPPERS\_DISABLE\_DDR |= 1<<STEPPERS\_DISABLE\_BIT;

//ZJK ADD FOR STEP\_IO INIT

GPIO\_InitTypeDef GPIO\_InitStructure;

//X\_STEP:PA8

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOA,ENABLE);

GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_8;

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;

GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_PP;

GPIO\_Init(GPIOA, &GPIO\_InitStructure);

X\_STEP\_PORT = (X\_STEP\_PORT & ~X\_STEP\_MASK) | (settings.invert\_mask&X\_STEP\_MASK);

//Y\_STEP:PD15

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOD,ENABLE);

GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_15;

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;

GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_PP;

GPIO\_Init(GPIOD, &GPIO\_InitStructure);

Y\_STEP\_PORT = (Y\_STEP\_PORT & ~Y\_STEP\_MASK) | (settings.invert\_mask&Y\_STEP\_MASK);

//X\_DIR :PA6

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOA,ENABLE);

GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_6;

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;

GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_PP;

GPIO\_Init(GPIOA, &GPIO\_InitStructure);

X\_DIRECTION\_PORT = (X\_DIRECTION\_PORT & ~X\_DIRECTION\_MASK) | (settings.invert\_mask&X\_DIRECTION\_MASK);

//Y\_DIR :PA2

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOA,ENABLE);

GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_2;

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;

GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_PP;

GPIO\_Init(GPIOA, &GPIO\_InitStructure);

Y\_DIRECTION\_PORT = (Y\_DIRECTION\_PORT & ~Y\_DIRECTION\_MASK) | (settings.invert\_mask&Y\_DIRECTION\_MASK);

//X\_RES :PA3

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOA,ENABLE);

GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_3;

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;

GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_PP;

GPIO\_Init(GPIOA, &GPIO\_InitStructure);

//Y\_RES :PA4

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOA,ENABLE);

GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_4;

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;

GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_PP;

GPIO\_Init(GPIOA, &GPIO\_InitStructure);

//END ZJK STEPP\_IO INIT

// waveform generation = 0100 = CTC

// TCCR1B &= ~(1<<WGM13);

// TCCR1B |= (1<<WGM12);

// TCCR1A &= ~(1<<WGM11);

// TCCR1A &= ~(1<<WGM10);

//

// // output mode = 00 (disconnected)

// TCCR1A &= ~(3<<COM1A0);

// TCCR1A &= ~(3<<COM1B0);

//timer3 init 通用定时器

RCC->APB1ENR|=1<<1;//使能TIMER3时钟

//设置允许更新中断，必须同时设置了才能更新中断

TIM3->DIER|=1<<0;//允许中断更新

TIM3->DIER|=1<<6;//触发中断使能

//设置NVIC

//NVIC\_PreemptionPriority:抢占优先级

//NVIC\_SubPriority :响应优先级

//NVIC\_Channel :中断编号

//NVIC\_Group :中断分组 0~4

//注意优先级不能超过设定的组的范围!否则会有意想不到的错误

//组划分:

//组0:0位抢占优先级,4位响应优先级

//组1:1位抢占优先级,3位响应优先级

//组2:2位抢占优先级,2位响应优先级

//组3:3位抢占优先级,1位响应优先级

//组4:4位抢占优先级,0位响应优先级

//抢占：低优先级的中断执行的同时高优先级可以将其抢回，高结束后再执行低

//响应：同时中断时谁先响应（高优先级，数小的）

//NVIC\_SubPriority和NVIC\_PreemptionPriority的原则是,数值越小,越优先

//中断分组设置，设置NVIC相关寄存器，使能中断

//sys.c中，直接调用

MY\_NVIC\_Init(1,2,TIM3\_IRQChannel,2); //抢占1，响应2，组2

// Configure Timer 2

// TCCR2A = 0; // Normal operation

// TCCR2B = 0; // Disable timer until needed.

// TIMSK2 |= (1<<TOIE2); // Enable Timer2 Overflow interrupt

// #ifdef STEP\_PULSE\_DELAY

// TIMSK2 |= (1<<OCIE2A); // Enable Timer2 Compare Match A interrupt

// #endif

//timer4 init 通用定时器

RCC->APB1ENR|=1<<2;//使能TIMER4时钟

//设置允许更新中断，必须同时设置了才能更新中断

TIM4->DIER|=1<<0;//允许中断更新

TIM4->DIER|=1<<6;//触发中断使能

//设置NVIC

//NVIC\_PreemptionPriority:抢占优先级

//NVIC\_SubPriority :响应优先级

//NVIC\_Channel :中断编号

//NVIC\_Group :中断分组 0~4

//注意优先级不能超过设定的组的范围!否则会有意想不到的错误

//组划分:

//组0:0位抢占优先级,4位响应优先级

//组1:1位抢占优先级,3位响应优先级

//组2:2位抢占优先级,2位响应优先级

//组3:3位抢占优先级,1位响应优先级

//组4:4位抢占优先级,0位响应优先级

//抢占：低优先级的中断执行的同时高优先级可以将其抢回，高结束后再执行低

//响应：同时中断时谁先响应（高优先级，数小的）

//NVIC\_SubPriority和NVIC\_PreemptionPriority的原则是,数值越小,越优先

//中断分组设置，设置NVIC相关寄存器，使能中断

//sys.c中，直接调用

MY\_NVIC\_Init(0,1,TIM4\_IRQChannel,2); //抢占0，响应1，组2

// Start in the idle state, but first wake up to check for keep steppers enabled option.

st\_wake\_up();

st\_go\_idle();

}

第二步：设置定时器4的初值在st\_wake\_up中计算的，在TIM3的中断服务函数中赋值的

st\_wake\_up()主要是开定时器3，计算定时器4的初值

st\_go\_idle（）主要是关定时器

// Stepper state initialization. Cycle should only start if the st.cycle\_start flag is

// enabled. Startup init and limits call this function but shouldn't start the cycle.

void st\_wake\_up()

{

// Enable steppers by resetting the stepper disable port

if (bit\_istrue(settings.flags,BITFLAG\_INVERT\_ST\_ENABLE)) {

//STEPPERS\_DISABLE\_PORT |= (1<<STEPPERS\_DISABLE\_BIT);

STEPPERS\_DISABLE\_EN;

} else {

//STEPPERS\_DISABLE\_PORT &= ~(1<<STEPPERS\_DISABLE\_BIT);

STEPPERS\_DISABLE\_DIS;

}

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

// Initialize stepper output bits

out\_bits = (0) ^ (settings.invert\_mask);

// Initialize step pulse timing from settings. Here to ensure updating after re-writing.

#ifdef STEP\_PULSE\_DELAY

// Set total step pulse time after direction pin set. Ad hoc computation from oscilloscope.

step\_pulse\_time = -(((settings.pulse\_microseconds+STEP\_PULSE\_DELAY-2)\*TICKS\_PER\_MICROSECOND) >> 3);

// Set delay between direction pin write and step command.

OCR2A = -(((settings.pulse\_microseconds)\*TICKS\_PER\_MICROSECOND) >> 3);

#else // Normal operation

// Set step pulse time. Ad hoc computation from oscilloscope. Uses two's complement.

//设置一个周期中低电平时间，也就是TIM4的计时时间，因为TIM4中拉高的输出脚，所以它计时的时间其实为拉低时候的延时时间

//>>3是因为定时器八分频

//取反因为AVR用的普通定时器模式，计数器从填入的值开始计时直到溢出产生溢出中断，所以填入的值=256-实际的定时值

//step\_pulse\_time = -(((settings.pulse\_microseconds-2)\*TICKS\_PER\_MICROSECOND) >> 3);

//张纪宽修改bug，因为STM32的定时器模式跟AVR不同，STM32是从0开始计数，计数到设定值后进入中断,所以不需要取反

step\_pulse\_time = ((settings.pulse\_microseconds-2)\*TICKS\_PER\_MICROSECOND) >> 3;

#endif

// Enable stepper driver interrupt

//TIMSK1 |= (1<<OCIE1A);

//bit0=1开启定时器，bit6,5=00边沿,bit4=0向上计数 1向下，bit9,8=00无分频 01=2\* 10=4\*

TIM3->CR1|=0X01;

}

}

// Stepper shutdown

void st\_go\_idle()

{

// Disable stepper driver interrupt

//TIMSK1 &= ~(1<<OCIE1A);

//bit0=1关闭定时器

TIM3->CR1&=~0X01;

// Disable steppers only upon system alarm activated or by user setting to not be kept enabled.

if ((settings.stepper\_idle\_lock\_time != 0xff) || bit\_istrue(sys.execute,EXEC\_ALARM)) {

// Force stepper dwell to lock axes for a defined amount of time to ensure the axes come to a complete

// stop and not drift from residual inertial forces at the end of the last movement.

delay\_ms(settings.stepper\_idle\_lock\_time);

if (bit\_istrue(settings.flags,BITFLAG\_INVERT\_ST\_ENABLE)) {

//STEPPERS\_DISABLE\_PORT &= ~(1<<STEPPERS\_DISABLE\_BIT);

STEPPERS\_DISABLE\_DIS;

} else {

//STEPPERS\_DISABLE\_PORT |= (1<<STEPPERS\_DISABLE\_BIT);

STEPPERS\_DISABLE\_EN;

}

}

}

TIM4定时时间计算方法：TIM4控制一个周期中低电平时间的，相当于就是延时时间，因为在TIM4中断函数里拉高了

step引脚，进入此中断服务函数证明定时结束，定时结束将引脚拉高，作用就是延时低电平时间

step\_pulse\_time = ((settings.pulse\_microseconds-2)\*TICKS\_PER\_MICROSECOND) >> 3;

settings.pulse\_microseconds = DEFAULT\_STEP\_PULSE\_MICROSECONDS;//setting default中赋值的

#define DEFAULT\_STEP\_PULSE\_MICROSECONDS 10 //default.h中 这个值代表延时的时间，单位us

因为步进电机靠频率大小控制速度，所以脉宽没要求，只要cpu能检测出来就好了

左移三位代表TIM4设置的8分频

第三步：设置定时器3的赋值

因为定时器三设置的是周期时间，就是电机的速度控制，由于有加减速，这个时间肯定是不定的，通过计算随时改变的，那就是一个函数中通过参数设置，执行一个周期后计算该参数赋给寄存器

//设置TIM3定时时间，也就是一个周期的时间

//参数steps\_per\_minute其实为频率，次数/min f=次数/60 每秒钟多少下

//所以其他都是定制，主要就是steps\_per\_minute的值决定了周期,这里的参数其实就是速度，steps/min,因为脉冲周期最终决定的就是运动速度，在这里把两者联系了起来

static void set\_step\_events\_per\_minute(uint32\_t steps\_per\_minute)

{

if (steps\_per\_minute < MINIMUM\_STEPS\_PER\_MINUTE) { steps\_per\_minute = MINIMUM\_STEPS\_PER\_MINUTE; }

//st.cycles\_per\_step\_event = config\_step\_timer((TICKS\_PER\_MICROSECOND\*1000000\*60)/steps\_per\_minute); //(TICKS\_PER\_MICROSECOND\*1000000\*60)超出32位，所以改变了下顺序

//计算定时器填入的值，不是定时时间

//填入的值=系统每秒钟跳动的次数(次数/s) 除以 电机运动速度（steps/s）

//注意steps\_per\_minute/60为steps/s,正好与分子上72MHZ的单位：跳动次数/s相对应，注意这里跳动次数与steps的区别，跳动次数指的是单片机定时计数器每秒钟的跳动次数，steps指的是每秒钟电机需要几个脉冲，相除之后的值就代表每个脉冲单片机跳动次数，正好就是需要填入的值，哇塞太巧妙了

st.cycles\_per\_step\_event = config\_step\_timer(TICKS\_PER\_MICROSECOND\*1000000/steps\_per\_minute\*60);

}

// Configures the prescaler and ceiling of timer 1 to produce the given rate as accurately as possible.

// Returns the actual number of cycles per interrupt

//设置周期时间 TIM3的定时时间就是周期时间

static uint32\_t config\_step\_timer(uint32\_t cycles)

{

uint16\_t ceiling;

uint16\_t prescaler;

uint32\_t actual\_cycles;

if (cycles <= 0xffffL) {

ceiling = cycles;

prescaler = 72-1; // prescaler: 0

actual\_cycles = ceiling;

} else if (cycles <= 0x7ffffL) {

ceiling = cycles >> 3;

prescaler = 72\*8-1; // prescaler: 8

actual\_cycles = ceiling \* 8L;

} else if (cycles <= 0x3fffffL) {

ceiling = cycles >> 6;

prescaler = 72\*64-1; // prescaler: 64

actual\_cycles = ceiling \* 64L;

} else if (cycles <= 0xffffffL) {

ceiling = (cycles >> 8);

prescaler = 72\*256-1; // prescaler: 256

actual\_cycles = ceiling \* 256L;

} else if (cycles <= 0x3ffffffL) {

ceiling = (cycles >> 10);

//prescaler = 72\*1024-1; // prescaler: 1024 这里超了十六位，设置错误

prescaler = 65535; //不精确了

actual\_cycles = ceiling \* 1024L;

} else {

// Okay, that was slower than we actually go. Just set the slowest speed

ceiling = 0xffff;

prescaler = 65535;

actual\_cycles = 0xffff \* 1024;

}

// Set prescaler 设置时钟分频

//TCCR1B = (TCCR1B & ~(0x07<<CS10)) | (prescaler<<CS10);

TIM3->PSC=prescaler; //设置分频

// Set ceiling

TIM3->ARR=ceiling; //设置计数值

return(actual\_cycles);

}

定时器值=TICKS\_PER\_MICROSECOND\*1000000/steps\_per\_minute\*60

本来是TICKS\_PER\_MICROSECOND\*1000000\*60 /steps\_per\_minute 由于这样会超出32bit范围，所以修改成上面了

TICKS\_PER\_MICROSECOND\*1000000\*60代表每分钟系统时钟跳动次数

steps\_per\_minute代表电机每分钟需要的脉冲个数，也就是速度

填入的值=系统每秒钟跳动的次数(次数/s) 除以 电机运动速度（steps/s）

//注意steps\_per\_minute/60为steps/s,正好与分子上72MHZ的单位：跳动次数/s相对应，注意这里跳动次数与steps的区别，跳动次数指的是单片机定时计数器每秒钟的跳动次数，steps指的是每秒钟电机需要几个脉冲，相除之后的值就代表每个脉冲单片机跳动次数，正好就是需要填入的值，哇塞太巧妙了

static uint32\_t config\_step\_timer(uint32\_t cycles)这个函数就是具体的填入值方法，分频和余量的方式

因为很有可能设置时间超出了16位定时器的时间，所以对应分频设置可以设置最大的时间

第四步：两个定时器的中断服务函数，在中断服务函数中进行的参数的赋值，定时器的赋值，定时器的开关等

这是定时器控制最重要的程序

// "The Stepper Driver Interrupt" - This timer interrupt is the workhorse of Grbl. It is executed at the rate set with

// config\_step\_timer. It pops blocks from the block\_buffer and executes them by pulsing the stepper pins appropriately.

// It is supported by The Stepper Port Reset Interrupt which it uses to reset the stepper port after each pulse.

// The bresenham line tracer algorithm controls all three stepper outputs simultaneously with these two interrupts.

//ISR(TIMER1\_COMPA\_vect)

//定时器3中断服务函数

/\*\*\*\*\*\*\*\*\*\*中断服务函数完成的工作：

1.TIM3定时时间就是一个周期时间

2.TIM3把信号拉低开始定时接着打开TIM4定时拉低时间，TIM4时间到拉高信号TIM3定时到拉低信号

3.数据操作：计算下一个周期的时间，XYZ的位置，运动的方向，需要的加速度速度等参数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void TIM3\_IRQHandler(void) //中断服务函数名一定正确

{

if(TIM3->SR&0X0001)//再次判断是否更新事件

{

if (busy) { return; } // The busy-flag is used to avoid reentering this interrupt

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*从这里开始是方向和输出(拉低)控制，开TIM4拉低延时时间定时器，一个周期的开始，TIM3从0开始计数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Set the direction pins a couple of nanoseconds before we step the steppers

//STEPPING\_PORT = (STEPPING\_PORT & ~DIRECTION\_MASK) | (out\_bits & DIRECTION\_MASK);

//change by zjk

X\_DIRECTION\_PORT = (X\_DIRECTION\_PORT & ~X\_DIRECTION\_MASK) | (out\_bits & X\_DIRECTION\_MASK);

Y\_DIRECTION\_PORT = (Y\_DIRECTION\_PORT & ~Y\_DIRECTION\_MASK) | (out\_bits & Y\_DIRECTION\_MASK);

//end zjk

// Then pulse the stepping pins

#ifdef STEP\_PULSE\_DELAY

step\_bits = (STEPPING\_PORT & ~STEP\_MASK) | out\_bits; // Store out\_bits to prevent overwriting.

#else // Normal operation

//STEPPING\_PORT = (STEPPING\_PORT & ~STEP\_MASK) | out\_bits;

//change by zjk

X\_STEP\_PORT = (X\_STEP\_PORT & ~X\_STEP\_MASK) | (out\_bits& X\_STEP\_MASK);

Y\_STEP\_PORT = (Y\_STEP\_PORT & ~Y\_STEP\_MASK) | (out\_bits& Y\_STEP\_MASK);

//end zjk

#endif

// Enable step pulse reset timer so that The Stepper Port Reset Interrupt can reset the signal after

// exactly settings.pulse\_microseconds microseconds, independent of the main Timer1 prescaler.

// TCNT2 = step\_pulse\_time; // Reload timer counter

// TCCR2B = (1<<CS21); // Begin timer2. Full speed, 1/8 prescaler

//设置TIM4为八分频，TIM4开始计时

TIM4->PSC=72\*8-1; //设置分频 八分频

TIM4->ARR=step\_pulse\_time; //设置计数值

busy = true;

// Re-enable interrupts to allow ISR\_TIMER2\_OVERFLOW to trigger on-time and allow serial communications

// regardless of time in this handler. The following code prepares the stepper driver for the next

// step interrupt compare and will always finish before returning to the main program.

//sei(); //开中断

//bit0=1开启定时器，bit6,5=00边沿,bit4=0向上计数 1向下，bit9,8=00无分频 01=2\* 10=4\*

TIM4->CR1|=0X01;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

从这里往下就是数据的操作

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// If there is no current block, attempt to pop one from the buffer

if (current\_block == NULL) {

// Anything in the buffer? If so, initialize next motion.

//从主线程中获取信息，在中断服务函数中计算

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

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

数据操作1：计算下个周期大小

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if (current\_block != NULL) {

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

// During feed hold, do not update rate and trap counter. Keep decelerating.

//st.trapezoid\_adjusted\_rate决定周期大小，(st.trapezoid\_adjusted\_rate/60)代表速度

st.trapezoid\_adjusted\_rate = current\_block->initial\_rate;//初始速度，也就是启动时候的速度

set\_step\_events\_per\_minute(st.trapezoid\_adjusted\_rate); // Initialize cycles\_per\_step\_event

st.trapezoid\_tick\_cycle\_counter = CYCLES\_PER\_ACCELERATION\_TICK/2; // Start halfway for midpoint rule.

}

st.min\_safe\_rate = current\_block->rate\_delta + (current\_block->rate\_delta >> 1); // 1.5 x rate\_delta

st.counter\_x = -(current\_block->step\_event\_count >> 1);

st.counter\_y = st.counter\_x;

st.counter\_z = st.counter\_x;

st.event\_count = current\_block->step\_event\_count;

st.step\_events\_completed = 0;

} else {

st\_go\_idle();

bit\_true(sys.execute,EXEC\_CYCLE\_STOP); // Flag main program for cycle end

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

数据操作2：计算位置信息

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if (current\_block != NULL) {

// Execute step displacement profile by bresenham line algorithm

out\_bits = current\_block->direction\_bits;

st.counter\_x += current\_block->steps\_x;

if (st.counter\_x > 0) {

out\_bits |= (1<<X\_STEP\_BIT);

st.counter\_x -= st.event\_count;

if (out\_bits & (1<<X\_DIRECTION\_BIT)) { sys.position[X\_AXIS]--; }

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

}

st.counter\_y += current\_block->steps\_y;

if (st.counter\_y > 0) {

out\_bits |= (1<<Y\_STEP\_BIT);

st.counter\_y -= st.event\_count;

if (out\_bits & (1<<Y\_DIRECTION\_BIT)) { sys.position[Y\_AXIS]--; }

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

}

// st.counter\_z += current\_block->steps\_z;

// if (st.counter\_z > 0) {

// out\_bits |= (1<<Z\_STEP\_BIT);

// st.counter\_z -= st.event\_count;

// if (out\_bits & (1<<Z\_DIRECTION\_BIT)) { sys.position[Z\_AXIS]--; }

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

// }

st.step\_events\_completed++; // Iterate step events

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

数据操作3：由加速度计算下一个周期大小，步进电机周期决定速度

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// While in block steps, check for de/ac-celeration events and execute them accordingly.

if (st.step\_events\_completed < current\_block->step\_event\_count) {

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

// Check for and execute feed hold by enforcing a steady deceleration from the moment of

// execution. The rate of deceleration is limited by rate\_delta and will never decelerate

// faster or slower than in normal operation. If the distance required for the feed hold

// deceleration spans more than one block, the initial rate of the following blocks are not

// updated and deceleration is continued according to their corresponding rate\_delta.

// NOTE: The trapezoid tick cycle counter is not updated intentionally. This ensures that

// the deceleration is smooth regardless of where the feed hold is initiated and if the

// deceleration distance spans multiple blocks.

if ( iterate\_trapezoid\_cycle\_counter() ) {

// If deceleration complete, set system flags and shutdown steppers.

if (st.trapezoid\_adjusted\_rate <= current\_block->rate\_delta) {

// Just go idle. Do not NULL current block. The bresenham algorithm variables must

// remain intact to ensure the stepper path is exactly the same. Feed hold is still

// active and is released after the buffer has been reinitialized.

st\_go\_idle();

bit\_true(sys.execute,EXEC\_CYCLE\_STOP); // Flag main program that feed hold is complete.

} else {

st.trapezoid\_adjusted\_rate -= current\_block->rate\_delta;

set\_step\_events\_per\_minute(st.trapezoid\_adjusted\_rate);

}

}

} else {

// The trapezoid generator always checks step event location to ensure de/ac-celerations are

// executed and terminated at exactly the right time. This helps prevent over/under-shooting

// the target position and speed.

// NOTE: By increasing the ACCELERATION\_TICKS\_PER\_SECOND in config.h, the resolution of the

// discrete velocity changes increase and accuracy can increase as well to a point. Numerical

// round-off errors can effect this, if set too high. This is important to note if a user has

// very high acceleration and/or feedrate requirements for their machine.

if (st.step\_events\_completed < current\_block->accelerate\_until) {

// Iterate cycle counter and check if speeds need to be increased.

if ( iterate\_trapezoid\_cycle\_counter() ) {

st.trapezoid\_adjusted\_rate += current\_block->rate\_delta;

if (st.trapezoid\_adjusted\_rate >= current\_block->nominal\_rate) {

// Reached nominal rate a little early. Cruise at nominal rate until decelerate\_after.

st.trapezoid\_adjusted\_rate = current\_block->nominal\_rate;

}

set\_step\_events\_per\_minute(st.trapezoid\_adjusted\_rate);

}

} else if (st.step\_events\_completed >= current\_block->decelerate\_after) {

// Reset trapezoid tick cycle counter to make sure that the deceleration is performed the

// same every time. Reset to CYCLES\_PER\_ACCELERATION\_TICK/2 to follow the midpoint rule for

// an accurate approximation of the deceleration curve. For triangle profiles, down count

// from current cycle counter to ensure exact deceleration curve.

if (st.step\_events\_completed == current\_block-> decelerate\_after) {

if (st.trapezoid\_adjusted\_rate == current\_block->nominal\_rate) {

st.trapezoid\_tick\_cycle\_counter = CYCLES\_PER\_ACCELERATION\_TICK/2; // Trapezoid profile

} else {

st.trapezoid\_tick\_cycle\_counter = CYCLES\_PER\_ACCELERATION\_TICK-st.trapezoid\_tick\_cycle\_counter; // Triangle profile

}

} else {

// Iterate cycle counter and check if speeds need to be reduced.

if ( iterate\_trapezoid\_cycle\_counter() ) {

// NOTE: We will only do a full speed reduction if the result is more than the minimum safe

// rate, initialized in trapezoid reset as 1.5 x rate\_delta. Otherwise, reduce the speed by

// half increments until finished. The half increments are guaranteed not to exceed the

// CNC acceleration limits, because they will never be greater than rate\_delta. This catches

// small errors that might leave steps hanging after the last trapezoid tick or a very slow

// step rate at the end of a full stop deceleration in certain situations. The half rate

// reductions should only be called once or twice per block and create a nice smooth

// end deceleration.

if (st.trapezoid\_adjusted\_rate > st.min\_safe\_rate) {

st.trapezoid\_adjusted\_rate -= current\_block->rate\_delta;

} else {

st.trapezoid\_adjusted\_rate >>= 1; // Bit shift divide by 2

}

if (st.trapezoid\_adjusted\_rate < current\_block->final\_rate) {

// Reached final rate a little early. Cruise to end of block at final rate.

st.trapezoid\_adjusted\_rate = current\_block->final\_rate;

}

set\_step\_events\_per\_minute(st.trapezoid\_adjusted\_rate);

}

}

} else {

// No accelerations. Make sure we cruise exactly at the nominal rate.

if (st.trapezoid\_adjusted\_rate != current\_block->nominal\_rate) {

st.trapezoid\_adjusted\_rate = current\_block->nominal\_rate;

set\_step\_events\_per\_minute(st.trapezoid\_adjusted\_rate);

}

}

}

} else {

// If current block is finished, reset pointer

current\_block = NULL;

plan\_discard\_current\_block();

}

}

out\_bits ^= settings.invert\_mask; // Apply step and direction invert mask

busy = false;

}

TIM3->SR&=~(1<<0); //状态寄存器清除

}

// This interrupt is set up by ISR\_TIMER1\_COMPAREA when it sets the motor port bits. It resets

// the motor port after a short period (settings.pulse\_microseconds) completing one step cycle.

// NOTE: Interrupt collisions between the serial and stepper interrupts can cause delays by

// a few microseconds, if they execute right before one another. Not a big deal, but can

// cause issues at high step rates if another high frequency asynchronous interrupt is

// added to Grbl.

//ISR(TIMER2\_OVF\_vect)

//定时器4中断服务函数

void TIM4\_IRQHandler(void) //中断服务函数名一定正确

{

if(TIM4->SR&0X0001)//再次判断是否更新事件

{

// Reset stepping pins (leave the direction pins)

//STEPPING\_PORT = (STEPPING\_PORT & ~STEP\_MASK) | (settings.invert\_mask & STEP\_MASK);

X\_STEP\_PORT = (X\_STEP\_PORT & ~X\_STEP\_MASK) | (settings.invert\_mask & X\_STEP\_MASK);

Y\_STEP\_PORT = (Y\_STEP\_PORT & ~Y\_STEP\_MASK) | (settings.invert\_mask & Y\_STEP\_MASK);

//TCCR2B = 0; // Disable Timer2 to prevent re-entering this interrupt when it's not needed.

//bit0=1关闭定时器

TIM4->CR1&=~0X01;

}

TIM4->SR&=~(1<<0); //状态寄存器清除

}

在定时器3的中断服务函数中开的定时器4，定时器4中关的自己，一条G代码执行完关闭的定时器3

由此可见，定时器4是通过抢占的方式中断的，所以要设置定时器4的中断优先级要高于TIM3

定时器3的定时间是从主程序里获得赋值到st.trapezoid\_adjusted\_rate的，这个值就代表时刻变化着的速度

这里速度是如何实时计算的，有待于好好研究，因为速度是变化的，是梯形状的，是启动停止时候缓慢改变的

————————————————

版权声明：本文为CSDN博主「zhangjikuan」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/zhangjikuan/article/details/46697657