4 步进电机控制板的固件代码设计

对于接入USB总线的步进电机控制板上而言，USB接口芯片只是起到对总线上的数据依照传输协议进行接收和发送的作用。具体到对设备本身的控制和对总线上主机命令的响应，全部是运行于步进电机控制板上AT89C52中的固件代码的功能。本章将对控制板中的固件代码的整体结构进行介绍，并对其中最重要的三个模块进行详细分析。

4.1 固件代码编写思路及注意事项

对于USB设备类控制程序，目前没有任何厂商提供自动生成固件（firmware ）的工具,因此所有程序都要由自己手工编制。

固件代码的编写采用C语言在Keil uVisions2集成开发环境下编写。

AT89C52控制程序通常由三部分组成：

(1) 初始化AT89C52端口和所有的外围电路（包括PDIUSBD12）。

(2) 主循环部分其任务是可以中断的。

(3) 中断服务程序，其任务是对时间敏感的，必须马上执行。

PDIUSBD12 的固件设计成完全的中断驱动。当CPU 处理前台任务时，USB 的传输可在后台进行。这就确保了最佳的传输速率和更好的软件结构同时简化了编程和调试。后台ISR 中断服务程序和前台主程序循环之间的数据交换，通过事件标志和数据缓冲区来实现。例如PDIUSBD12 的批量输出端点，可使用循环的数据缓冲区。当PDIUSBD12 从USB 收到一个数据包，那么就对CPU 产生一个中断请求，CPU 立即响应中断。在ISR中，固件将数据包从PDIUSBD12 内部缓冲区移到循环数据缓冲区，并在随后清零PDIUSBD12 的内部缓冲区，以使能接收新的数据包。CPU 可以继续它当前的前台任务，直到完成。然后返回到主循环，检查循环缓冲区内是否有新的数据，并开始其它的前台任务。如图4-1所示。

由于这种结构，主循环不关心数据是来自USB 串口还是其它，它只检查循环缓冲区内需要处理的新数据。这个概念很重要，这样主循环程序专注于数据的处理而ISR 能够以最大可能的速度进行数据的传输。相似的，控制端点在数据包处理时采用了同样的概念。ISR 接收和保存数据缓冲区中的控制传输并设置相应的标志寄存器。主循环向协议处理程序发出请求。由于所有的标准器件级别和厂商请求都是在协议处理程序

中进行处理。ISR 得以保持它的效率，而且一旦增加新的请求只需要在协议层进行修改。

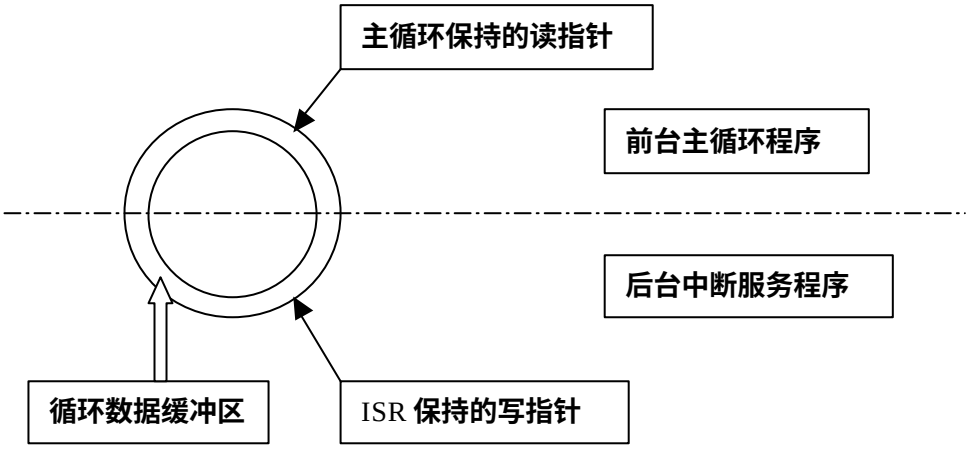


图4-1 固件处理流程

根据USB 协议，任何传输都是由主机开始的。这样AT89C52做它的前台工作，等待中断。主机首先要发令牌包给USB设备（这里是PDIUSBD12 ）。PDIUSBD12 接收到完整无误的令牌包后，就给AT89C52发中断，AT89C52进入中断服务程序。

在中断服务程序中，AT89C52首先读PDIUSBD12的中断寄存器，以判断USB令牌包的类型，然后对USB总线的标准命令和厂商类命令进行响应。标准命令的响应有固定的规则。此处的厂商类命令，便是我们的步进电机控制系统特有的、自己设计的步进电机控制命令。当AT89C52收到厂商类命令时，会执行相应的步进电机驱动控制操作。因此AT89C52的固件程序主要就是中断服务程序的编写。

在固件程序中要完成对各种令牌包的响应，其中比较难处理的是SETUP包，主要是端点0的编程。AT89C52与PDIUSBD12的通信，主要是靠AT89C52给PDIUSBD12发命令和数据来实现的。

步进电机控制板的固件有着如图4-2的层次式结构：

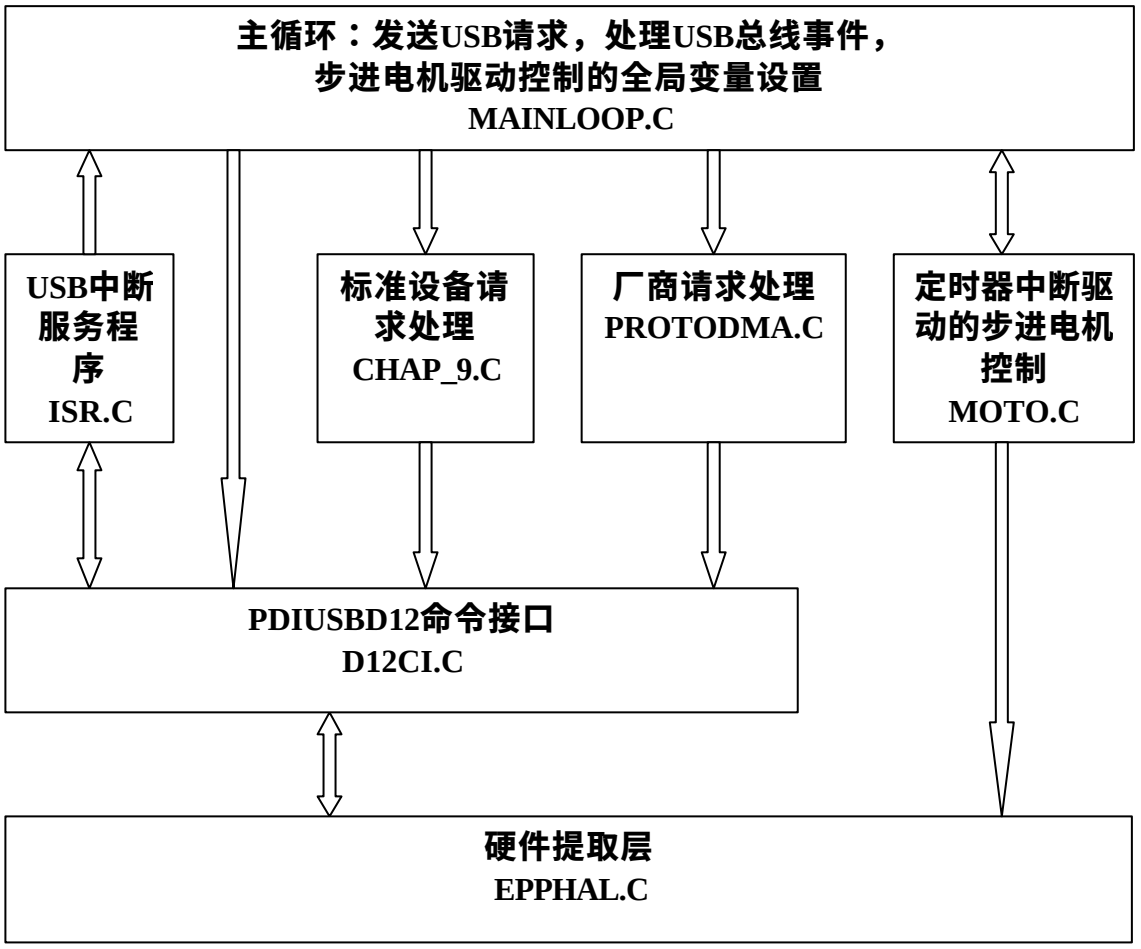


图4-2 固件结构和数据流向

各模块程序的分工如下：

(1) 硬件提取层：对AT89C52的I/O口、数据总线等硬件接口进行操作。

(2) PDIUSBD12命令接口：对PDIUSBD12器件进行操作的模块子程序集。

(3) USB中断服务程序：当PDIUSBD12向AT89C52发出中断请求时，读取PDIUSBD12的中断传输来的数据，并设定事件标志“EPPFLAGS”和SETUP包数据缓冲区“CONROL_XFER”传输给主循环程序。

(4) 标准设备请求处理程序：对USB的标准设备请求进行处理。

(5) 厂商请求处理程序：对用户添加的厂商请求进行处理。

(6) 定时器中断驱动的步进电机控制程序：利用定时器的精确定时功能，对步进电机进行驱动控制。与USB中断服务程序和主循环等模块交互的变量有enstep（步进电

机驱动允许），width（脉冲宽度），direction（方向设定），STEPNUMS（驱动步数）;inprocess（是否处于步进驱动中），channel（驱动通道设定）。

(7) 主循环程序：发送USB请求，处理USB总线事件，步进电机驱动控制的全局变量设置。

在编写固件程序时需要注意以下几点：

(1) AT89C52的中断应设置为电平触发中断后，一定要读上次传输状态寄存器命令40-45H ,以清除中断寄存器中的中断标志。这样PDIUSBD12的中断输出才能变回高电平，这一点非常重要。

(2) 在接收到Setup包后，一定要调用ACK Setup命令，重新使能端口0。

(3) 在向IN端点写完数据后，一定调用Validate Buffer（命令FAH），指明缓冲区的数据有效，可以发送到主机。

(4) 当读完数据后，一定调用Clear Buffer（命令F2H），以保证可以接收新的包。

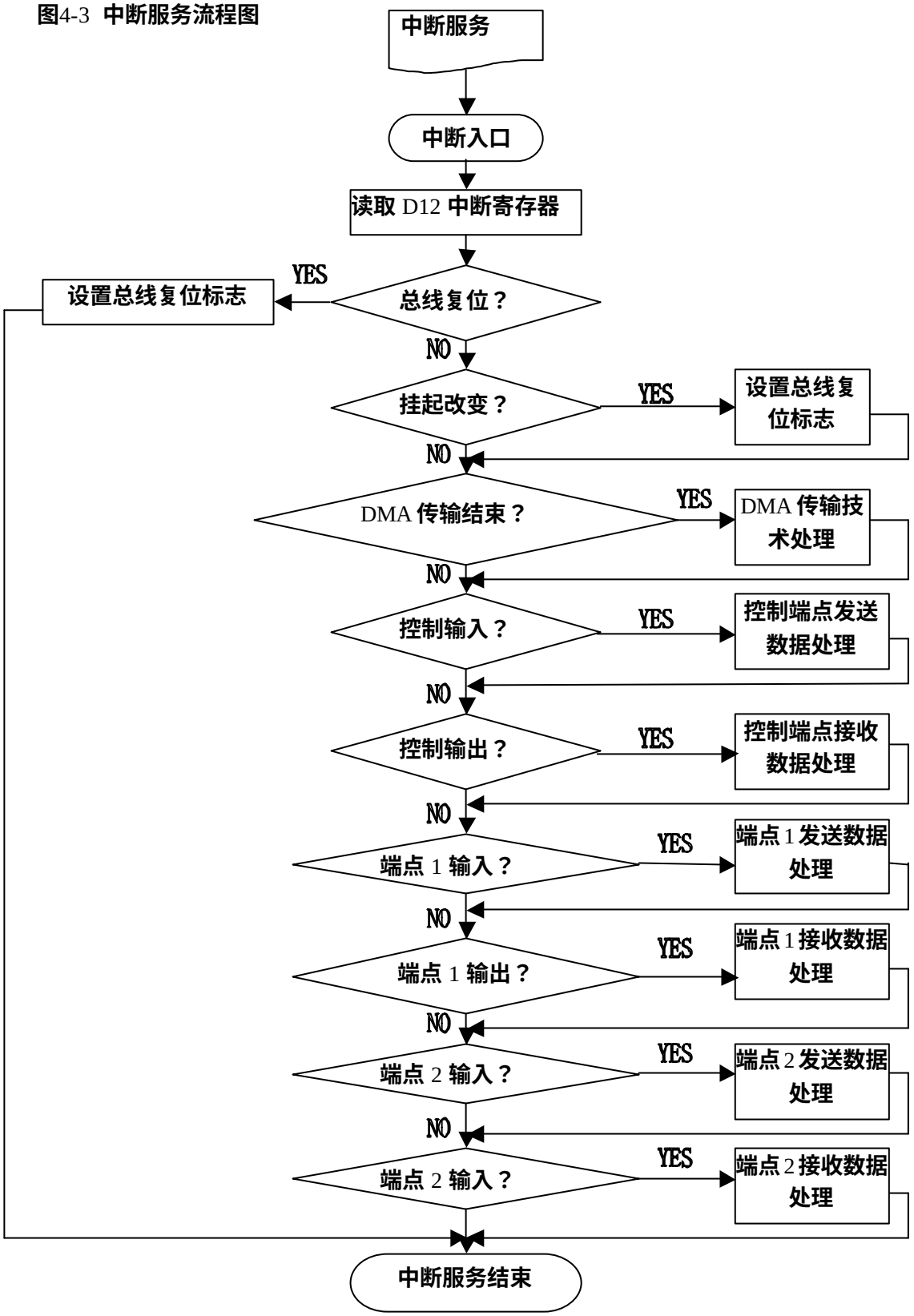
(5) 可以通过调用Read Chip ID（命令FDH），检查PDIUSBD12 是否工作。该命令要读两个字节数据。

4. 2 USB 中断服务程序流程及函数编写

这部分代码处理由PDIUSBD12 产生的中断。它将数据从PDIUSBD12 的内部FIFO 取回到CPU 存储器，并建立正确的事件标志，以通知主循环程序进行处理。

USB中断服务程序的处理流程如图4-3所示。

图4-3 中断服务流程图


中断服务
中断入口
读取 D12 中断寄存器
总线复位？
YES
设置总线复位标志
NO
挂起改变？
YES
设置总线复位标志
NO
DMA 传输结束？
YES
DMA 传输技术处理
NO
控制输入？
YES
控制端点发送数据处理
NO
控制输出？
YES
控制端点接收数据处理
NO
端点 1 输入？
YES
端点 1 发送数据处理
NO
端点 1 输出？
YES
端点 1 接收数据处理
NO
端点 2 输入？
YES
端点 2 发送数据处理
NO
端点 2 输入？
YES
端点 2 接收数据处理
中断服务结束

对应的中断处理程序主例程如下：

```
/*中断服务程序*/
void  fn_usb_isr(void)        interrupt    0
{
     unsigned int data i_st;

      EX0=0;
     bEPPflags.bits.in_isr = 1;
     i_st=  D12_ReadInterruptRegister();                    //读取中断寄存器

     if(i_st != 0)
       {
          if(i_st & D12_BUSRESET)
               bEPPflags.bits.bus_reset = 1;
          if(i_st & D12_INT_EOT)
               dma_eot();                                   //DMA 传输结束
          if(i_st & D12_INT_SUSPENDCHANGE)
               bEPPflags.bits.bus_suspend = 1;        //挂起改变
          if(i_st & D12_INT_ENDP0IN)
               ep0_txdone();                            //端点 0   IN 中断处理
          if(i_st & D12_INT_ENDP0OUT)
               ep0_rxdone();                            //端点 0   OUT 中断处理
          if(i_st & D12_INT_ENDP1IN)
               ep1_txdone();                            //端点 1   IN 中断处理
          if(i_st & D12_INT_ENDP1OUT)
               ep1_rxdone();                            //端点 1   OUT 中断处理
          if(i_st & D12_INT_ENDP2IN)
               main_txdone();                           //端点 2   IN 中断处理
          if(i_st & D12_INT_ENDP2OUT)
               main_rxdone();                          //端点 2   OUT 中断处理
     }
```

```
    bEPPflags.bits.in_isr = 0;
    EX0=1;
}
```

4.3 主循环程序流程及函数编写

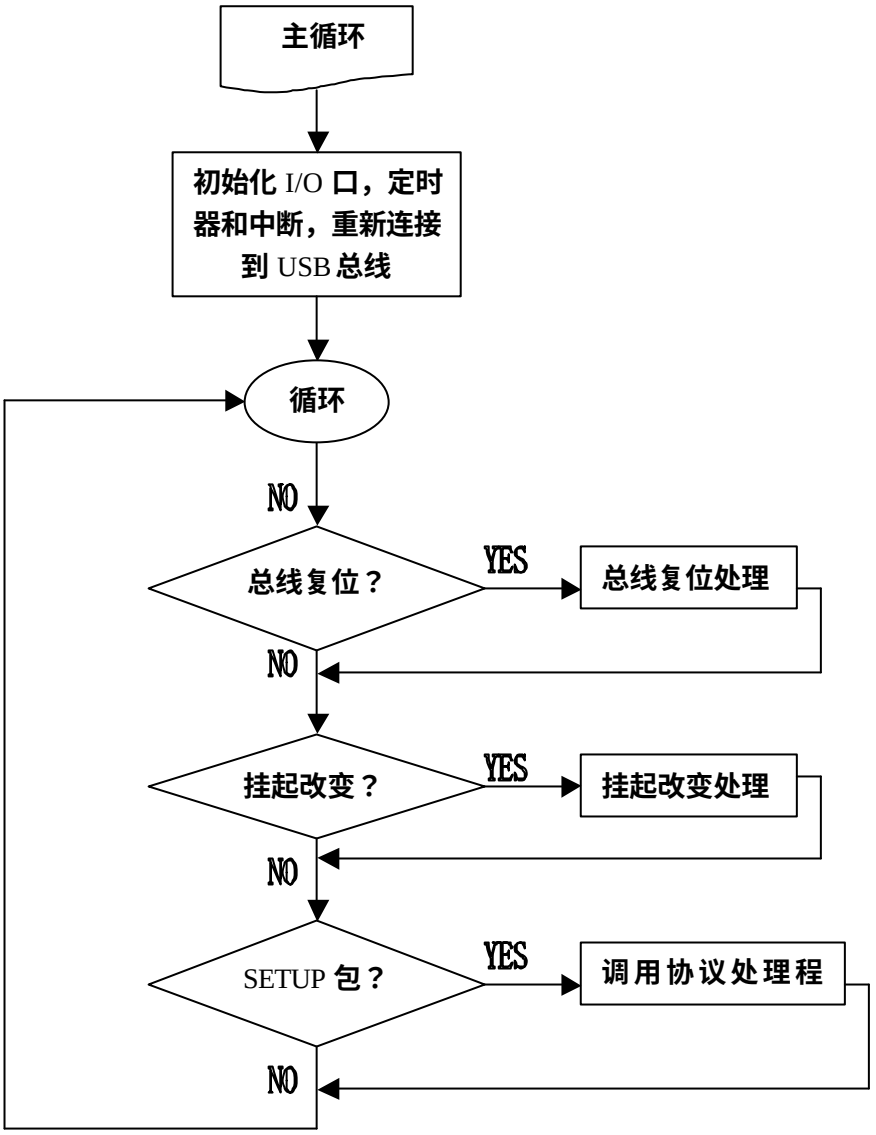


图 4-4 主循环流程图

主循环处理流程如图 4-4，具体程序及注释如下：

```
void main(void)
{
    Vflag = 0;                  //高低电平标志，初值：低
    cycount = 0;                //时钟输出循环计数，初值：0
    enstep = 0;                 //步进电机驱动使能，初值：不允许
    direction = 0;              //方向设定初值，初值：正向
    width = 90;                 //脉冲宽度微调，初值：90
    channel=1;                  //驱动通道，初值：X 轴
    STEPNUMS=0;                 //驱动步数，初值：0

    init_port();                //初始化端口
    InitTimer0();               //初始化定时器 0
    init_special_interrupts();  //设置中断

    D12CS=0x1;                  //解除片选 PDIUSBD12
    D12CS=0x0;                  //片选 PDIUSBD12

    bEPPflags.value=0;          //清除事件标志寄存器 bEPPflags
    reconnect_USB();            //重新连接 PDIUSBD12 到 USB 总线
    EX0=1;                      //开 INT0(USB 通信)中断
    /* 主程序循环 */
    while(1)
     {
        if (bEPPflags.bits.bus_reset)          //总线复位处理
         {
            EX0=0;
            bEPPflags.bits.bus_reset = 0;      //清除总线复位处理
            EX0=1;
            SUSPEND=1;                         //释放 PDIUSBD12 挂起控制
         }

        if (bEPPflags.bits.bus_suspend)        //挂起改变处理
```

```
        {
            EX0=0;
            bEPPflags.bits.bus_suspend=  0;          //清除挂起改变处理
            EX0=1;
           D12CS=1;                                  //直接唤醒 PDIUSBD12
        } // if suspend change

       if  (bEPPflags.bits.setup_packet)            //Setup 包处理
        {
            EX0=0;
            bEPPflags.bits.setup_packet  =  0;      //清除 Setup 包标志
            EX0=1;
            control_handler();                       //调用协议处理子程序
            D12CS=0;
            SUSPEND=1;                               //释放挂起状态
        }
    } // Main Loop End
}
```

其中负责对 SETUP 包处理的 control_handler()函数如下：

```
/*协议控制子程序*/
void control_handler()
{
    unsigned char data type,  req;                  //读取请求类型，保留 bit5 和 bit6

    type          =          ControlData.DeviceRequest.bmRequestType          &
USB_REQUEST_TYPE_MASK;
                                                    //读取请求代码，保留低 4 位
    req = ControlData.DeviceRequest.bRequest & USB_REQUEST_MASK;

    if (type == USB_STANDARD_REQUEST)               //标准请求处理
        (*StandardDeviceRequest[req])();
    else  if  (type  ==  USB_VENDOR_REQUEST)       //厂商请求处理
```

```
        (*VendorDeviceRequest[req])();
    else
        stall_ep0();                                //无效请求，返回 STALL
}
```

其中厂商请求是对本步进电机控制板、进行控制命令请求处理的函数入口表。其代码如下：

```
/*用户厂商请求入口地址表*/
code   void   (*VendorDeviceRequest[ ])(void)=
{
  RUN,
  STOP,
  X_SELECT,
  Y_SELECT,
  OPPOSITE,
  NEGATIVE,
  RESET,
  SIN_WAITE,
  STEPNUMS,
  PLUSWIDTH,
  reserved,
  reserved,
  reserved,
  reserved,
  reserved,
  reserved
};
```

厂商请求处理函数入口的表项，具体含义如下表 4-1。

表 4-1 控制命令表

厂商控制命令编码 bRequest	控制命令含意	厂商请求的处理函数入口表中的对应处理函数
0	启动	RUN（）
1	停止	STOP（）
2	选择 X 轴驱动	X_SELECT（）
3	选择 Y 轴驱动	Y_SELECT（）
4	正方向运动	OPPOSITE（）
5	负方向运动	NEGATIVE（）
6	系统重启	RESET（）
7	等待设备状态恢复	SIN_WAITE（）
8	步进电机步数设定	STEPNUMS（）
9	驱动脉冲宽度微调	PLUSWIDTH（）

表 4-1 中的处理函数，主要是对定时器中断处理中使用的、步进电机控制的全局量进行赋值。

这些函数列出如下：

```
void RUN(void)
{
enstep = 1;                    //步进电机驱动使能
single_transmit(0, 0);         //控制端点接收应答
}
void STOP(void)
{
enstep = 0;                    //  步进电机停止
single_transmit(0, 0);
}
void X_SELECT(void)
{
channel=1;                     //  X 轴驱动
single_transmit(0, 0);
}
void Y_SELECT(void)
```

```
{
channel=2;                    // Y 轴驱动
single_transmit(0, 0);
}
void OPPOSITE(void)
{
direction=1;                  //  步进电机正向驱动
single_transmit(0, 0);
}
void NEGATIVE(void)
{
direction=0;                  //  步进电机反向驱动
single_transmit(0, 0);
}
void RESET(void)
{
single_transmit(0, 0);     //先对主机应答，
main();            //  然后才能复位设备。以避免主机误解设备为损坏状态！
}
void SIN_WAITE(void)
{
if(inprocess==1)           //  步进电机状态索取。
EpBuf[0]=0;                //  如果电机正在运动中，返回 0，否则返回 1。
else
EpBuf[0]=1;
single_transmit(EpBuf, 1); //  向控制读端点写入返回的数据
}
void STEPNUMS(void)
{
STEPNUMS=ControlData.DeviceRequest.wValue; //  步进数赋值
single_transmit(0, 0);
}
```

```
void PLUSWIDTH(void)
{
width=ControlData.DeviceRequest.wValue;      //  脉冲宽度赋值
single_transmit(0, 0);
}
```

4. 4　定时器中断处理流程及函数编写

定时器中断处理主要负责对步进电机的驱动控制。其流程如图 4-4。

本系统的步进电机驱动脉冲输出，采用时钟中断进行计时控制，并且对该中断赋予了高优先级。这样就保证了驱动脉冲的脉宽精确度，进而对步进电机的运动精度提供了保证。

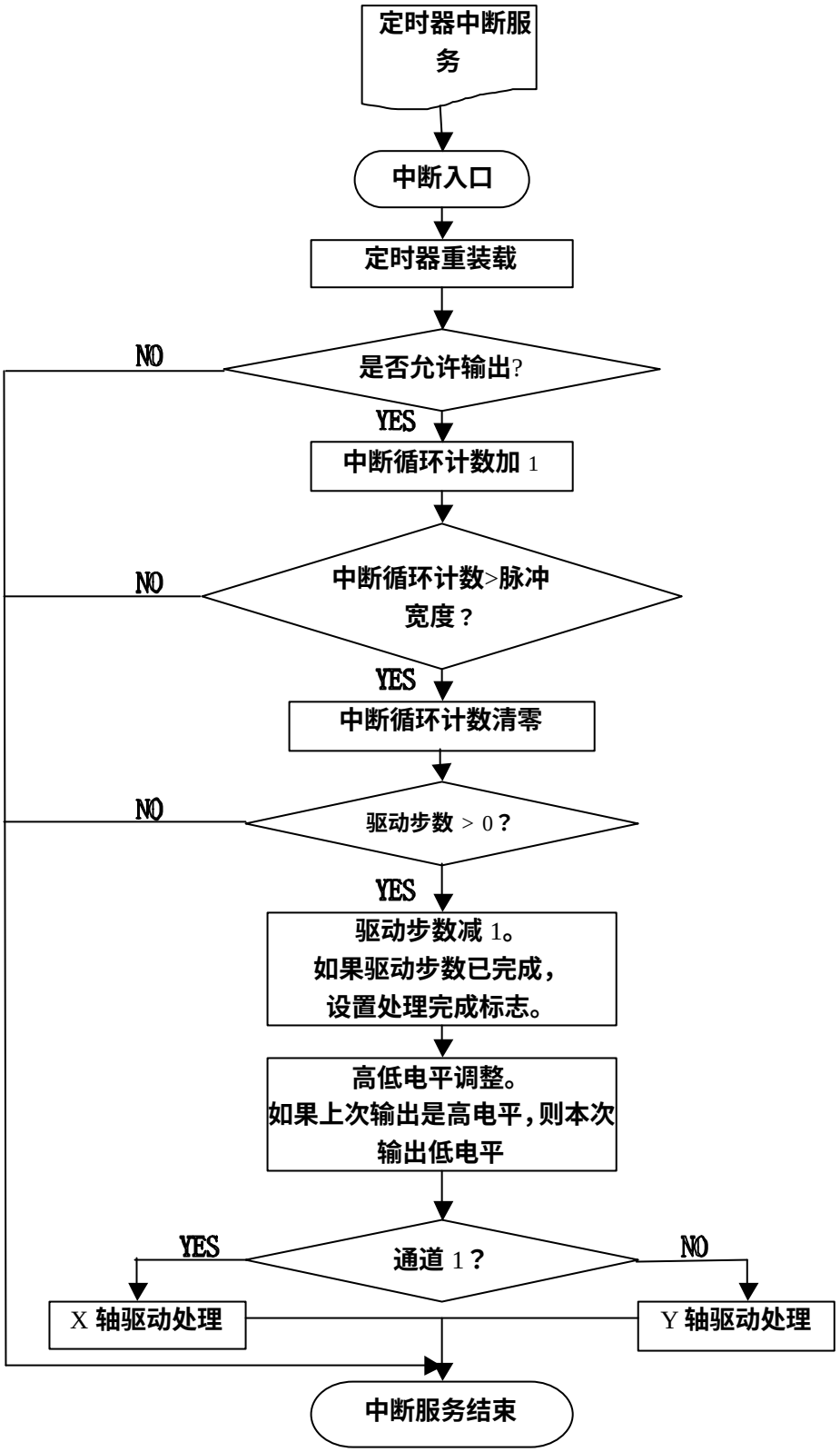


图 4-4 定时器中断处理流程

定时器中断处理函数如下：

```
void timer0_interrupt (void)    interrupt 1    //10MS 定时器 0 中断
{
    TR0 = 0;
    TH0 = HIGH(TIME10MS); //定时器重装载
    TL0 = LOW(TIME10MS);
    TR0 = 1;
   if (enstep)    //是否允许输出
   {
    cycount++;              //中断循环计数
    if (cycount > width)   //10ms 脉冲计数，如果超过步进电机脉宽设定值就输出一步
    {
        if (STEPNUMS > 0)        //是否还有步进计数
        {
         STEPNUMS--;     //步进计数递减
        if(STEPNUMS ==0) inprocess=0;//如果驱动步数已完成，设置处理完成标志
         Vflag ++;
         if (Vflag > 1)   Vflag = 0;      //如果上次输出是高电平，则本次输出低电平
            switch (channel)     //驱动通道选择
          {
            case 1:
            {
            MOTODRIVER_X(i);          //X 轴驱动脉冲输出
            MOTODIRECT_X(direction);   //X 轴驱动方向设置
            break;
            }
            case 2:
            {
            MOTODRIVER_Y(i);              //Y 轴驱动脉冲输出
            MOTODIRECT_Y(direction);   //Y 轴驱动方向输出
            break;
            }
```

```
                default:
                {
                  break;
                }
              }
            }
            cycount = 0; //输出一步后，中断循环计数清零
        }
    }
}
```

4.5 本章小结

本章首先对本系统的固件代码编写思路及注意事项进行了介绍，随后对本系统固件中的最重要的三个函数模块（USB 中断服务函数、主循环函数、定时器中断处理函数）进行了细致深入的分析。