第4章

STM32 单片机中断编程与机器人触觉导航

通过前面几章的学习，你已经掌握如何用单片机的 I/O 端口来控制机器人的各种运动。这些连接机器人伺服电机的单片机端口是作为输出使用，而且使用非常简单。

本章你将学习如何使用这些端口来获取外界信息，即将 STM32 单片机端口作为输入端口使用。例如，获取按键的信息进行人机交互，给机器人小车增加触觉传感器判断是否碰到了障碍物。实际上，对于任何一个嵌入式系统，如自动控制系统，都需要通过传感器来获取外界信息，由计算机或单片机根据反馈的信息进行计算和决策，生成控制命令，然后通过输出端口去控制系统相应的执行机构，完成相关任务。因此，学习如何使用 STM32 单片机的输入端口同学习使用输出端口同等重要。本章除了学习按键检测方法外，还可以在机器人前端安装并测试一个称为“胡须”的触觉开关，通过编程来监视触觉开关的状态，以及决定当它遇到障碍物时如何动作，最终的结果就是通过触觉实现机器人自动导航。

4.1 STM32 单片机按键输入检测

为了检测按键是否被按下，可以将按键与 STM32 单片机的 I/O 端口相连，其电路图如图 4.1 所示。当有按键被按下时，相应的端口为低电平，当没有按键被按下时，相应的端口为高电平。

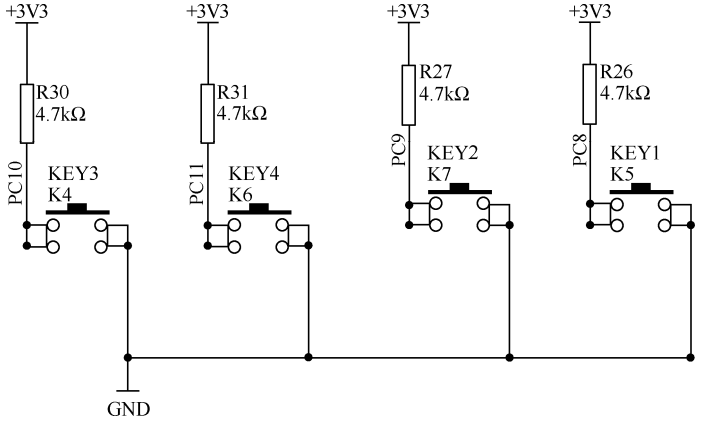

图 4.1　按键电路图

任务一　按键检测

下面这段代码的功能是当某个按键被按下时，与之对应的发光二极管亮灭状态交替变化一次。PC8 端口的 KEY1 键与 PB8 端口的发光二极管对应，PC9 端口的 KEY2 键与 PB9 端口的发光二极管对应，PC10 端口的 KEY3 键与 PC12 端口的发光二极管对应，PC11 端口的 KEY4 键与 PC13 端口的发光二极管对应。

例程：KeyNoEINT.c

```
#include "stm32f10x_heads.h"
#include "HelloRobot.h"

int main(void)
{
    BSP_Init();                                    //开发板初始化函数
    USART_Configuration();
    printf("Program Running!\n");
    while (1)
    {
        if(GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_8)==0)
        {
            if(GPIO_ReadOutputDataBit(GPIOB, GPIO_Pin_8)==0)
             GPIO_SetBits(GPIOB, GPIO_Pin_8);
            else
             GPIO_ResetBits(GPIOB, GPIO_Pin_8);
        }

        if(GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_9)==0)
        {
            if(GPIO_ReadOutputDataBit(GPIOB, GPIO_Pin_9)==0)
             GPIO_SetBits(GPIOB, GPIO_Pin_9);
            else
             GPIO_ResetBits(GPIOB, GPIO_Pin_9);
        }

        if(GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_10)==0)
        {
            if(GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_12)==0)
             GPIO_SetBits(GPIOC, GPIO_Pin_12);
            else
             GPIO_ResetBits(GPIOC, GPIO_Pin_12);
        }

        if(GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_11)==0)
        {
          if(GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_13)==0)
```

```
                GPIO_SetBits(GPIOC, GPIO_Pin_13);
              else
                GPIO_ResetBits(GPIOC, GPIO_Pin_13);
            }
            delay_nms(120);
          }
      }
```

在 GPIO 配置函数 GPIO_Configuration 中，我们已将 PC8、PC9、PC10、PC11 设置为按键输入引脚，下面的代码是将 PC11 端口设置为浮空输入模式：

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOC, &GPIO_InitStructure);
```

许多自动化机械都依赖于各种触觉开关，如当机器人碰到障碍物时，触觉开关就会察觉，通过编程让机器人躲开障碍物；旅客登机桥在靠近飞机时为了保护昂贵的飞机，在登机桥接口安装触须，当登机桥离飞机很近时触须就会碰到飞机，立即通知控制器提醒离飞机已经很近了，需要降低靠近速度；工厂利用触觉开关来计量生产线上的工件数量；在工业加工过程中，触觉开关也被用来排列物体。在所有这些实例中，触觉开关提供的输入通过计算机或者单片机处理后生成其他形式的程序化的输出。

4.2　STM32 单片机输入端口的应用

通过第 2 章的学习，已经知道 STM32 系列单片机有 5 个 16 位的并行 I/O 口：PA、PB、PC、PD 和 PE。这 5 个端口，既可以作为输入，又可以作为输出，既可以按 16 位处理，又可以按位方式使用。实际上，在单片机复位期间和刚复位时，复用功能未开启，I/O 端口被配置成浮空输入模式，所有端口都有外部中断能力。为了使用外部中断线，端口必须配置成输入模式。

作为输入，如果 I/O 引脚上的电压为高电平（5V 或 3.3V），则与其相对应的 I/O 口寄存器中的相应位存储 1；如果电压为低电平（0V），则存储 0。

布置恰当的电路，可以使胡须达到以下效果：当胡须没有被碰到时，I/O 引脚上的电压为高电平（5V 或 3.3V）；当胡须被碰到时，I/O 引脚上的电压为低电平（0V）。单片机读入上述数据，进行分析和处理，控制机器人的运动。安装好胡须的机器人小车如图 4.2 所示。

任务二　安装并测试机器人的触觉——胡须

让机器人通过触觉胡须进行导航，首先必须安装并测试胡须。如图 4.3 所示是所需的硬件元件清单，包括：

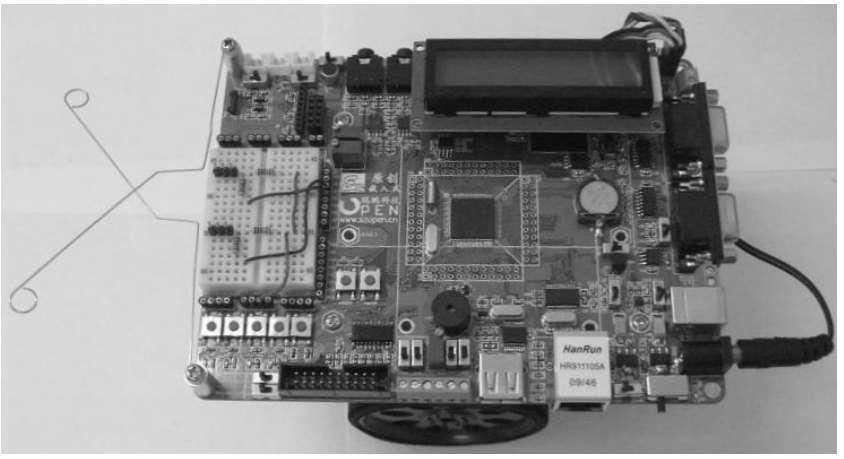

图 4.2　安装好胡须的机器人小车

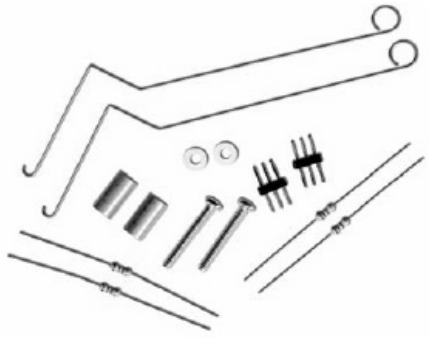

图 4.3　胡须硬件

（1）金属丝 2 根。
（2）平头 M3×22 盘头螺钉 2 个。
（3）13mm 圆形立柱 2 个。
（4）M3 尼龙垫圈 2 个。
（5）3-pin 公—公接头 2 个。
（6）2 个 220Ω 电阻（色环：红—红—黑—黑）。
（7）2 个 10kΩ 电阻（色环：棕—黑—黑—红）。

安装胡须（见图 4.4）

（1）螺钉依次穿过 M3 尼龙垫圈和 13mm 圆形立柱。
（2）螺钉穿过主板上的圆孔之后，拧进主板下面的支架中，但不要拧紧。
（3）把须状金属丝的其中一个钩在尼龙垫圈之上，另一个钩在尼龙垫圈之下，调整它们的位置，使它们横向交叉但又不接触；拧紧螺钉到支架上。
（4）参考接线图 4.5，搭建胡须电路。注意：右边胡须状态信息输入是通过 PE 口的第 1 引脚完成的，而左边胡须状态信息输入是通过 PE 口的第 0 引脚完成的。
（5）确定两条胡须比较靠近，但又不接触面包板上的 3-pin 头，推荐保持 3mm 的距离。
如图 4.6 所示是实际的参考接线图。安装好触觉胡须的教学开发板如图 4.7 所示。

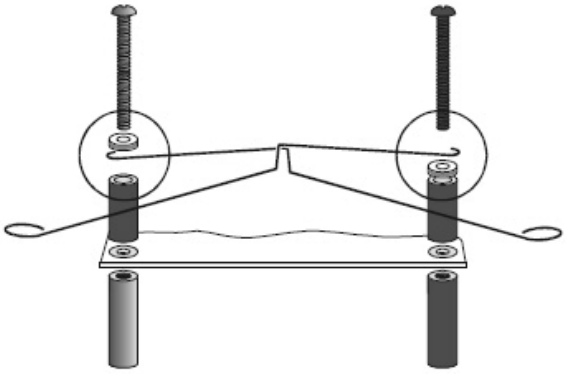

图 4.4　安装机器人胡须

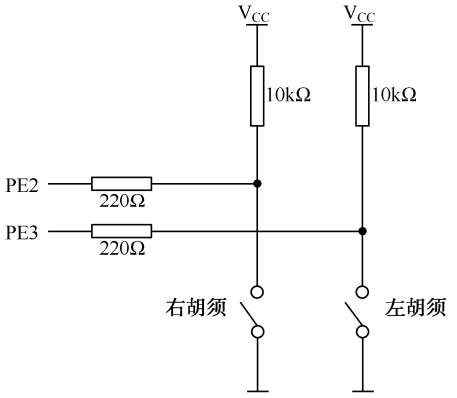

图 4.5　胡须电路示意图

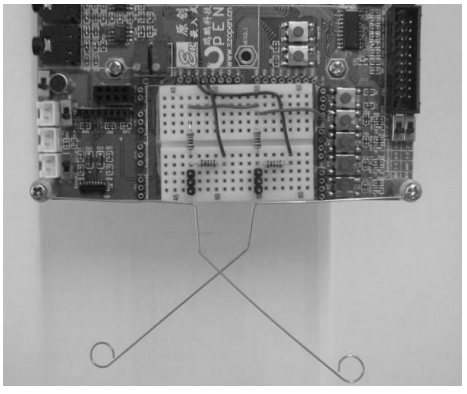

图 4.6　胡须接线图

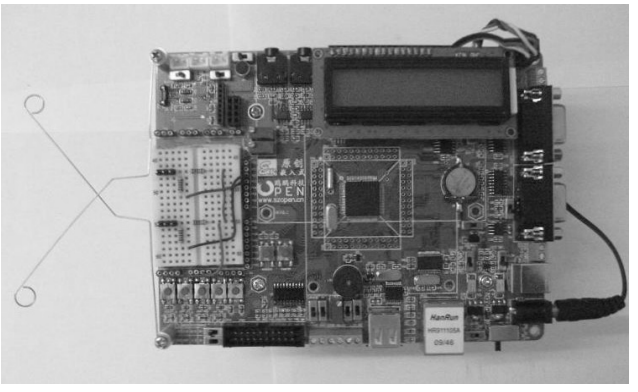

图 4.7　安装好触觉胡须的教学开发板

测试胡须

观察图 4.5 所示的胡须电路示意图，显然每条胡须都是一个机械式的、接地常开的开关（类似按键）。胡须接地（GND）是因为教学板外围的镀金孔都连接到 GND。金属支架和螺钉提供电气连接给胡须。

通过编程让单片机探测什么时候胡须被触动。由图 4.5 可知，连接到每个胡须电路的 I/O 引脚监视着 10kΩ上拉电阻上的电压变化。当胡须没有被触动时，连接胡须的 I/O 引脚的电压是高电平；当胡须被触动时，I/O 短接到地，所以 I/O 引脚的电压是低电平。

上拉电阻

上拉电阻就是与电源相连，起到拉高电平作用的电阻。此电阻还起到限流的作用，如图 4.5 中的 10kΩ电阻即为上拉电阻。与之对应的还有“下拉电阻”，它与“地（GND）”相连，可把电平拉至低位。

例程：TestWhiskers.c

```
#include "stm32f10x_heads.h"
#include "HelloRobot.h"
```

```
int PE2state(void)                    //获取 PE2 的状态
{
    return GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_2);
}
int PE3state(void)                    //获取 PE3 的状态
{
    return GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_3);
}

int main(void)
{
    BSP_Init();
    USART_Configuration();
    printf("Program Running!\r\n");

    while(1)
    {
        printf("右边胡须的状态:%d ", PE2state());
        printf("左边胡须的状态:%d\r\n",PE3state());
        delay_nms(150);
    }
}
```

上面的例程用于测试胡须的功能是否正常。首先，定义两个无参数有返回值的子函数 int PE2state(void)和 int PE3state(void)来获取左右两个胡须的状态。STM32 单片机的 5 个端口 PA、PB、PC、PD 和 PE 是可以按位来操作的，从低到高依次为第 0 口、第 1 口、……、第 15 口。

在弄清楚整个程序的执行原理后，按照下面的步骤执行程序，对触觉胡须进行测试。

- 连接好串口电缆，接通教学板和伺服电机的电源；
- 输入、保存并运行程序 TestWhiskers.c；
- 检查图 4.5，弄清楚哪条胡须是左胡须，哪条是右胡须；
- 此时调试终端显示：“右边胡须的状态:1 左边胡须的状态:1”，如图 4.8 所示；

图 4.8　左右胡须均未碰到

- 把右胡须按到 3-pin 转接头上，显示为：“右边胡须的状态:0 左边胡须的状态:1”，如图 4.9 所示；
- 把左胡须按到 3-pin 转接头上，显示为：“右边胡须的状态:1 左边胡须的状态:0”，如图 4.10 所示；

图 4.9　右胡须碰到

图 4.10　左胡须碰到

- 同时把两个胡须按到各自的 3-pin 转接头上，显示为：“右边胡须的状态:0 左边胡须的状态:0”，如图 4.11 所示；

图 4.11　左右胡须均碰到

- 如果两个胡须都通过测试，就可以继续下面的内容了；否则检查程序或电路中存在的错误。

任务三　基于胡须的机器人触觉导航

在任务二中，你已经通过编程检测胡须是否被触碰到。在本任务中将利用这些信息对机器人进行运动导航。在机器人行走过程中，如果有胡须被触碰到，那就意味着碰到了障碍物。导航程序需要接收这些输入信息，判断它的意义，调用一系列使机器人倒退、旋转朝不同方向行走的动作子函数以避开障碍物。

下面的程序使机器人向前走直到碰到障碍物。在这种情况下，机器人用它的一根或者两根胡须探测障碍物。一旦胡须探测到障碍物，调用第 3 章中的导航程序和子程序使小车倒退或者旋转，然后再重新向前行走，直到遇到另一个障碍物。

赋值运算符“=”与关系运算符“==”

注意赋值运算符“=”与关系运算符“==”的区别：赋值运算符“=”用来给变量赋值；关系运算符“==”用于判断两个值是否相等。

逻辑与“&&”运算符的运算规则：A&&B 表示若 A、B 都为真，则 A&&B 为真。

注意区分位操作符“&”和逻辑运算符“&&”。

例程：RoamingWithWhiskers.c

- 打开主板和伺服电机的电源；
- 输入、保存并运行程序 RoamingWithWhiskers.c；
- 尝试让机器人运动，当在其路线上遇到障碍物时，它将后退、旋转并向另一个方向运动。

```
#include "stm32f10x_heads.h"
#include "HelloRobot.h"

int PE2state(void)//获取 PE2 的状态
{
    return GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_2);
}

int PE3state(void)//获取 PE3 的状态
{
    return GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_3);
}

void Forward(void)
{
    GPIO_SetBits(GPIOD, GPIO_Pin_10);
    delay_nus(1700);
    GPIO_ResetBits(GPIOD,GPIO_Pin_10);

    GPIO_SetBits(GPIOD, GPIO_Pin_9);
    delay_nus(1300);
    GPIO_ResetBits(GPIOD,GPIO_Pin_9);

    delay_nms(20);
}
void Left_Turn(void)
{
    int i;
    for(i=1;i<=26;i++)
     {
        GPIO_SetBits(GPIOD, GPIO_Pin_10);
        delay_nus(1300);
        GPIO_ResetBits(GPIOD,GPIO_Pin_10);

        GPIO_SetBits(GPIOD, GPIO_Pin_9);
        delay_nus(1300);
        GPIO_ResetBits(GPIOD,GPIO_Pin_9);

        delay_nms(20);
    }
}
void Right_Turn(void)
{
    int i;
    for(i=1;i<=26;i++)
     {
```

```
            GPIO_SetBits(GPIOD, GPIO_Pin_10);
            delay_nus(1700);
            GPIO_ResetBits(GPIOD,GPIO_Pin_10);

            GPIO_SetBits(GPIOD, GPIO_Pin_9);
            delay_nus(1700);
            GPIO_ResetBits(GPIOD,GPIO_Pin_9);

            delay_nms(20);
        }
}
void Backward(void)
{
    int i;
    for(i=1;i<=65;i++)
     {
            GPIO_SetBits(GPIOD, GPIO_Pin_10);
            delay_nus(1300);
            GPIO_ResetBits(GPIOD,GPIO_Pin_10);

            GPIO_SetBits(GPIOD, GPIO_Pin_9);
            delay_nus(1700);
            GPIO_ResetBits(GPIOD,GPIO_Pin_9);

            delay_nms(20);
        }
}
int main(void)
{
     BSP_Init();
     USART_Configuration();
     printf("Program Running!\n");

    while(1)
     {
            if((PE2state()==0)&&( PE3state()==0))        //两胡须同时碰到
            {
                Backward(); //向后
                Left_Turn();//向左
                Left_Turn();//向左
            }
            else if(PE2state()==0)                      //右胡须碰到
            {
                Backward();//向后
                Left_Turn();//向左
            }
```

```
            else if(PE3state()==0)                    //左胡须碰到
             {
                  Backward();                         //向后
                  Right_Turn();                       //向右
             }
             else                                     //胡须没有碰到
                  Forward();                          //向前
        }
}
```

注意：函数 Forward()有一个变动。它只发送一个脉冲，然后返回。这点相当重要，因为机器人可以在向前运动中的每两个脉冲之间检测胡须的状态。这意味着，机器人在向前行走的过程中，每秒检查触须状态大概 43 次（1000ms/23ms≈43）。

因为每个全速前进的脉冲都使得机器人前进大约半厘米，只发送一个脉冲，然后回去检查胡须的状态是一个好主意。每次程序从 Forward()返回后，程序再次从 while 循环的开始处执行，此时 if…else 语句会再次检测胡须的状态。

任务四　机器人进入死区后的人工智能决策

你或许已经注意到机器人卡在墙角里的情况。当机器人进入墙角时，左胡须触墙，于是它右转，向前行走，右胡须触墙，于是左转前进，又碰到左墙，再次碰到右墙……如果不是你把它从墙角拿出来，它会一直困在墙角里而出不来。

编程逃离墙角死区

可以修改 RoamingWithWhiskers.c 使机器人碰到上述问题时能够逃离死区。技巧是记下胡须交替触动的总次数。技巧的关键是程序必须记住每个胡须的前一次触动状态，并和当前触动状态对比。如果状态相反，就在交替总数上加 1。如果这个交替总数超过了程序中预先给定的阈值，表示这是个墙角（死区），那么就该做一个“U”形转弯，并且把胡须交替计数器复位。

这个技巧的编程实现依赖于 if…else 嵌套语句。换句话说，程序检查一种条件，如果该条件成立（条件为真），则再检查包含于这个条件之内的另一个条件。下面是用伪代码说明嵌套语句的用法。

```
IF (condition1)
{
    commands for condition1
    IF(condition2)
    {
        commands for both condition2 and condition1
    }
    ELSE
    {
        commands for condition1 but not condition2
```

```
    }
}
ELSE
{
    commands for not condition1
}
```

伪代码

通常用来描述不依赖于计算机语言的算法。实际上在前面几章的任务和小结中，已经多次提醒和暗示你，无论是哪种计算机语言，都必须能够描述人类知识的逻辑结构。而人类知识的逻辑结构是统一的，如条件判断就是人类知识最核心的逻辑之一。因此，各种计算机语言都有语法和关键词来实现条件判别。因此，在写条件判断算法时，经常用一种用于描述人类知识结构逻辑的伪代码来描述在计算机中如何实现这些逻辑算法，以使算法具有通用性。有了伪代码，用具体的语言来实现算法就很简单了。

下面的例程用于探测连续的、交替出现的胡须触动过程。这个程序使机器人在第 4 次或第 5 次交替探测到墙角后，完成一个“U”形的拐弯，次数依赖于哪一个胡须先被触动。

- 输入、保存并运行程序 EscapingCorners.c；
- 在机器人行走时，轮流触动它的胡须，测试该程序。

例程：EscapingCorners.c

```
#include "stm32f10x_heads.h"
#include "HelloRobot.h"

int PE2state(void)//获取 PE2 的状态
{
      return GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_2);
}

int PE3state(void)//获取 PE3 的状态
{
      return GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_3);
}

void Forward(void)
{
…  //略，同前
}
void Left_Turn(void)
{
…  //略，同前
}
void Right_Turn(void)
{
…  //略，同前
```

```
}
void Backward(void)
{
…  //略，同前
}

int main(void)
{
    int counter=1;    //胡须碰撞总次数
    int old2=1;       //右胡须旧状态
    int old3=0;       //左胡须旧状态

    BSP_Init();
    USART_Configuration();
    printf("Program Running!\n");

   while(1)
    {
        if(PE3state()!=PE2state())
        {
            if((old2!=PE2state())&&(old3!=PE3state()))
            {
                counter=counter+1;
                old2=PE2state();
                old3=PE3state();
                if(counter>4)
                {
                    counter=1;
                    Backward();//向后
                    Left_Turn();//向左
                    Left_Turn();//向左
                }
            }
            else
                counter=1;
        }
        if((PE3state()==0)&&(PE2state()==0))
        {
            Backward();//向后
            Left_Turn();//向左
            Left_Turn();//向左
        }
        else if(PE2state()==0)
        {
            Backward();//向后
            Left_Turn();//向左
```

```
            }
            else if(PE3state()==0)
            {
                Backward();//向后
                Right_Turn();//向右
            }
            else
                Forward();//向前
        }
    }
```

EscapingCorners.c 是如何工作的?

由于该程序是由 RoamingWithWhiskers.c 修改而来，故这里只讨论与探测和逃离墙角相关的新特征。

```
int counter=1；
int old2=1；
int old3=0；
```

这三个变量用于探测墙角。int 型变量 counter 用来存储交替探测的次数。例程中，设定的交替探测的最大值为 4。int 型变量 old2、old3 存储胡须旧的状态值。

程序赋 counter 初值为 1，当机器人卡在墙角此值累计到 4 时，counter 复位为 1。old2 和 old3 必须赋值以至于看起来好像两根胡须中的其中一根在程序开始之前被触动了。这些工作之所以必须做，是因为探测墙角的程序总是对比交替触动的部分，或者 PE2state()==0，或者 PE3state()==0。与之对应，old2 和 old3 的值也相互不同。

现在看探测连续而交替触动墙角的部分。

首先要检查的是，是否有且只有一个胡须被触动。简单的方法就是询问“是否 PE2state() 不等于 PE3state()”。其具体判断语句如下：

```
if(PE3state()!=PE3state())
```

假如真有胡须被触动，接下来要做的事情就是检查当前状态是否确实与上次不同。换句话说，是 old2 不等于 PE2state()和 old3 不等于 PE3state()吗？如果是，就在胡须触动计数器上加 1，同时记下当前的状态，设置 old2 等于当前的 PE2state()，old3 等于当前的 PE3state()。

```
            if((PE2state()==0)&&(PE3state()==0))
            {
                Backward();//向后
                Left_Turn();//向左
                Left_Turn();//向左
            }
```

如果发现胡须连续 4 次被触动，那么计数值置 1，并且进行“U”形拐弯。

```
if(counter>4)
```

```
    {
        counter=1;
        Backward();
        Left_Turn();
        Left_Turn();
    }
```

紧接的 else 语句是机器人没有陷入墙角情况，故需要将计数器值置 1。之后的程序和 RoamingWithWhiskers.c 中的一样。

- 尝试增加变量 counter 的数值为 5 和 6，注意结果；
- 尝试减小变量 counter 的数值，观察小车在正常行走过程中是否有所不同。

4.3　STM32 单片机中断编程

中断是计算机和嵌入式系统中的一个十分重要的概念，在现代计算机和嵌入式系统中毫无例外地都要采用中断技术。什么是中断呢？可以举一个日常生活中的例子来说明，假如你正在看书，电话铃响了，这时你放下书，去接电话，通话完毕，再继续看书。这个例子就表达了中断及其处理过程：电话铃声使你暂时中止当前的工作，而去处理更为急需处理的事情（接电话），把急需处理的事情处理完毕之后，再回头来继续原来的事情。在这个例子中，电话铃声称为“中断请求”，你暂停看书去接电话叫做“中断响应”，接电话的过程就是“中断处理”。

在计算机执行程序的过程中，由于出现某个特殊情况（或称为“事件”），使得 CPU 中止现行程序，而转去执行处理该事件的处理程序（俗称中断处理或中断服务程序），待中断服务程序执行完毕，再返回断点处继续执行原来的程序，这个过程称为“中断”。

计算机为什么要采用中断

为了说明问题，再举一个例子。假设你有一个朋友来拜访你，但是由于不知道何时到达，你只能在大门等待，于是什么事情也干不了。如果在门口装一个门铃，你就不必在门口等待而去做其他的工作，朋友来了按门铃通知你，你这时才中断你的工作去开门，这样就可避免等待和浪费时间。计算机也是一样，如打印输出，CPU 传送数据的速率高，而打印机打印的速率低，如果不采用中断技术，CPU 将经常处于等待状态，效率极低。而采用了中断方式，CPU 可以进行其他的工作，只在打印机缓冲区中的当前内容打印完毕发出中断请求之后，才予以响应，暂时中断当前的工作转去执行向缓冲区传送数据，传送完成后又返回执行原来的程序，这样就大大地提高了计算机系统的效率。

中断是单片机实时处理内部或外部事件的一种内部机制。当某种内部或外部事件发生时，单片机的中断系统将迫使 CPU 暂停正在执行的程序，转而去进行中断事件的处理，中断处理完毕后，又返回被中断的程序处，继续执行下去。也就是说，中断是一种发生了一个事件时

调用相应的处理程序的过程。在一定条件下，CPU 响应中断后，暂停源程序的执行，转至为这个事件服务的中断处理程序。

中断是由于软件的或硬件的信号，使得 CPU 放弃当前的任务，转而去执行另一段子程序。可见，中断是可以人为参与（软件）或者硬件自动完成的、使 CPU 发生的一种程序跳转。通常外部中断是由外部设备通过请求引脚向 CPU 提出的。中断信号也可以是 CPU 内部产生的，如定时器、实时时钟等。

在 STM32 单片机复位期间和刚复位后，复用功能未开启，I/O 端口被配置成浮空输入模式，所有端口都有外部中断能力。为了使用外部中断线，端口必须配置成输入模式。

STM32 单片机的中断系统

相对于 ARM7 使用的外部中断控制器，Cortex-M3 内核中集成了中断控制器和中断优先级控制寄存器，支持 256 个中断（16 个内核+240 个外部）和可编程 256 级中断优先级的设置。NVIC 使用的是基于堆栈的异常模型。在处理中断时，将程序计数器、程序状态寄存器、链接寄存器和通用寄存器压入堆栈，中断处理完成后，再恢复这些寄存器。堆栈处理是由硬件完成的，无须在中断服务程序中进行堆栈操作。使用尾链（Tail-chaining）连续中断技术只需消耗 3 个时钟周期，相比于 32 个时钟周期的连续压、出堆栈，大大降低了延迟，提供了确定的、低延迟的中断处理，提高了性能。

STM32 单片机并没有使用 ARM Cortex-M3 内核全部的东西（如内存保护单元 MPU、8 位中断优先级等），因此它的 NVIC 是 ARM Cortex-M3 内核的 NVIC 的子集。STM32F10×系列单片机的嵌套中断向量控制器（NVIC）支持 68 个可屏蔽中断通道（不包含 16 个 Cortex-M3 内核的中断线），具有 16 级可编程中断优先级的设置（仅使用中断优先级设置 8bit 中的高 4 位），见表 4.1 和表 4.2。

表 4.1　ARM Cortex-M3 内核的 16 个中断通道对应的中断向量

位　置	优 先 级	优先级类型	名　称	说　明	地　址
—	—	—	—	保留	0x0000_0000
—	-3（最高）	固定	Reset	复位	0x0000_0004
—	-2	固定	NMI	不可屏蔽中断，RCC 时钟安全系统(CSS)连接到 NMI 向量	0x0000_0008
—	-1	固定	硬件失效	所有类型的失效	0x0000_000C
—	0	可设置	存储管理	存储器管理	0x0000_0010
—	1	可设置	总线错误	预取指失败，存储器访问失败	0x0000_0014
—	2	可设置	错误应用	未定义的指令或非法状态	0x0000_0018
—	—	—	—	保留	0x0000_001C
—	—	—	—	保留	0x0000_0020
—	—	—	—	保留	0x0000_0024
—	—	—	—	保留	0x0000_0028
—	3	可设置	SVCall	通过 SWI 指令的系统服务调用	0x0000_002C

续表

位　置	优 先 级	优先级类型	名　称	说　明	地　址
—	4	可设置	调试监控	调试监控器	0x0000_0030
—	—	—	—	保留	0x0000_0034
—	5	可设置	PendSV	可挂起的系统服务	0x0000_0038
—	6	可设置	SysTick	系统嘀嗒定时器	0x0000_003C

表 4.2　STM32F10×系列单片机的可屏蔽中断通道对应的中断向量

位　置	优 先 级	优先级类型	名　称	说　明	地　址
0	7	可设置	WWDG	窗口看门狗定时器中断	0x0000_0040
1	8	可设置	PVD	连到 EXTI 的电源电压检测(PVD)中断	0x0000_0044
2	9	可设置	TAMPER	侵入检测中断	0x0000_0048
3	10	可设置	RTC	实时时钟(RTC)全局中断	0x0000_004C
4	11	可设置	FLASH	闪存全局中断	0x0000_0050
5	12	可设置	RCC	复位和时钟控制(RCC)中断	0x0000_0054
6	13	可设置	EXTI0	EXTI 线 0 中断	0x0000_0058
7	14	可设置	EXTI1	EXTI 线 1 中断	0x0000_005C
8	15	可设置	EXTI2	EXTI 线 2 中断	0x0000_0060
9	16	可设置	EXTI3	EXTI 线 3 中断	0x0000_0064
10	17	可设置	EXTI4	EXTI 线 4 中断	0x0000_0068
11	18	可设置	DMA 通道 1	DMA 通道 1 全局中断	0x0000_006C
12	19	可设置	DMA 通道 2	DMA 通道 2 全局中断	0x0000_0070
13	20	可设置	DMA 通道 3	DMA 通道 3 全局中断	0x0000_0074
14	21	可设置	DMA 通道 4	DMA 通道 4 全局中断	0x0000_0078
15	22	可设置	DMA 通道 5	DMA 通道 5 全局中断	0x0000_007C
16	23	可设置	DMA 通道 6	DMA 通道 6 全局中断	0x0000_0080
17	24	可设置	DMA 通道 7	DMA 通道 7 全局中断	0x0000_0084
18	25	可设置	ADC	ADC 全局中断	0x0000_0088
19	26	可设置	USB_HP_CAN_TX	USB 高优先级或 CAN 发送中断	0x0000_008C
20	27	可设置	USB_LP_CAN_RX0	USB 低优先级或 CAN 接收 0 中断	0x0000_0090
21	28	可设置	CAN_RX1	CAN 接收 1 中断	0x0000_0094
22	29	可设置	CAN_SCE	CAN SCE 中断	0x0000_0098
23	30	可设置	EXTI9_5	EXTI 线[9:5]中断	0x0000_009C

续表

位　置	优 先 级	优先级类型	名　称	说　明	地　址
24	31	可设置	TIM1_BRK	TIM1 刹车中断	0x0000_00A0
25	32	可设置	TIM1_UP	TIM1 更新中断	0x0000_00A4
26	33	可设置	TIM1_TRG_COM	TIM1 触发和通信中断	0x0000_00A8
27	34	可设置	TIM1_CC	TIM1 捕获比较中断	0x0000_00AC
28	35	可设置	TIM2	TIM2 全局中断	0x0000_00B0
29	36	可设置	TIM3	TIM3 全局中断	0x0000_00B4
30	37	可设置	TIM4	TIM4 全局中断	0x0000_00B8
31	38	可设置	I^2C1_EV	I^2C1 事件中断	0x0000_00BC
32	39	可设置	I^2C1_ER	I^2C1 错误中断	0x0000_00C0
33	40	可设置	I^2C2_EV	I^2C2 事件中断	0x0000_00C4
34	41	可设置	I^2C2_ER	I^2C2 错误中断	0x0000_00C8
35	42	可设置	SPI1	SPI1 全局中断	0x0000_00CC
36	43	可设置	SPI2	SPI2 全局中断	0x0000_00D0
37	44	可设置	USART1	USART1 全局中断	0x0000_00D4
38	45	可设置	USART2	USART2 全局中断	0x0000_00D8
39	46	可设置	USART3	USART3 全局中断	0x0000_00DC
40	47	可设置	EXTI15_10	EXTI 线[15:10]中断	0x0000_00E0
41	48	可设置	RTCAlarm	连接到 EXTI 的 RTC 闹钟中断	0x0000_00E4
42	49	可设置	USB 唤醒	连接到 EXTI 的从 USB 待机唤醒中断	0x0000_00E8
43	50	可设置	TIM8_BRK	TIM8 刹车中断	0x0000_00EC
44	51	可设置	TIM8_UP	TIM8 更新中断	0x0000_00F0
45	52	可设置	TIM8_TRG_COM	TIM8 触发和通信中断	0x0000_00F4
46	53	可设置	TIM8_CC	TIM8 捕获比较中断	0x0000_00F8
47	54	可设置	ADC3	ADC3 全局中断	0x0000_00FC
48	55	可设置	FSMC	FSMC 全局中断	0x0000_0100
49	56	可设置	SDIO	SDIO 全局中断	0x0000_0104
50	57	可设置	TIM5	TIM5 全局中断	0x0000_0108
51	58	可设置	SPI3	SPI3 全局中断	0x0000_010C
52	59	可设置	UART4	UART4 全局中断	0x0000_0110
53	60	可设置	UART5	UART5 全局中断	0x0000_0114