

**DOCK开发指南**

最新版本：1.0

2017-8-19

目录

[版本历史 3](#_Toc491364363)

[引 言 4](#_Toc491364364)

[1.编写目的 4](#_Toc491364365)

[2.背景 4](#_Toc491364366)

[3.术语和缩写词 4](#_Toc491364367)

[1、DOCK的概念 5](#_Toc491364368)

[1.1硬件介绍 5](#_Toc491364369)

[1.1.1 充电座 5](#_Toc491364370)

[1.1.2 扫地机 6](#_Toc491364371)

[1.1.3 扫地机如何上座 8](#_Toc491364372)

[1.2软件逻辑 9](#_Toc491364373)

[1.2.1 传统逻辑 9](#_Toc491364374)

[1.2.2 引入基于APP-SDK的架构 10](#_Toc491364375)

[2、DOCK的重要基础知识 13](#_Toc491364376)

[2.1 behavior 13](#_Toc491364377)

[2.1.1 Dock\_Data 13](#_Toc491364378)

[2.1.1.1 边沿触发和电平触发的区别 13](#_Toc491364379)

[2.1.2 behavior详解 14](#_Toc491364380)

[2.2 行为优先级 16](#_Toc491364381)

[2.2.1 行为的切换 16](#_Toc491364382)

[2.3 红外信号封装 16](#_Toc491364383)

[2.4 Debouncer定义 18](#_Toc491364384)

[2.5 机器运动控制 19](#_Toc491364385)

[2.6 行为编写举例 20](#_Toc491364386)

[3、DOCK设计思想 22](#_Toc491364387)

[3.1 区域一 23](#_Toc491364388)

[3.2 区域二 23](#_Toc491364389)

[3.3 区域三 24](#_Toc491364390)

[3.4 区域四 25](#_Toc491364391)

[4、DOCK开发步骤 25](#_Toc491364392)

[4.1配置接收头对应I/O 25](#_Toc491364393)

[4.2 充电座信号 26](#_Toc491364394)

[4.2.1 确认信号类型 26](#_Toc491364395)

[4.2.2 确认接收到的信号是否正常 26](#_Toc491364396)

[4.2.3 确认充电座的信号分布 27](#_Toc491364397)

[4.3上座调试 27](#_Toc491364398)

[4.3.1 确定上座思想 27](#_Toc491364399)

[4.3.2 行为编写 27](#_Toc491364400)

[4.3.3 实际调试 27](#_Toc491364401)

# 版本历史

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本号** | **注释** | **作者** |
| 2017-8-19 | 1.0 | 建立 | 黎杰焕 |
|  |  |  |  |
|  |  |  |  |

# 引 言

1.编写目的

目的：为了新的开发人员能够迅速掌握DOCK的开发

阅读对象： 第一次接触扫地机的软件开发人员

2.背景

当前，每一个新接触dock 的开发人员都需要花费大量的时间来熟悉dock的架构才能进行后续开发，所以希望通过本文档帮助开发人员尽快熟悉dock架构及其开发流程。

3.术语和缩写词

|  |  |
| --- | --- |
| **缩写和术语** | **解释** |
|  |  |
| dock | 回座充电 |
| lt | 即light touch ，墙壁检测 |
| bump | 碰撞检测 |
| cliff | 悬崖检测 |
| 圆泡 | 安装于充电座上可以360°发射信号或安装于扫地机上可以360°接收信号的圆形部件 |

# 1、DOCK的概念

Dock简称回座充电，即通过充电座发出的红外信号指引扫地机器人回到充电座进行充电的过程。下面从硬件和软件两个方面讲解什么是dock。

1.1硬件介绍

dock的硬件由充电座与扫地机两部分组成。充电座上安装有几个红外发射头，每个发射头只能发出一种码值的红外信号；扫地机上安装有几个红外接收头，每个接收头都能对所有码值的红外信号进行接收。

1.1.1 充电座

充电座主要由充电触片和红外发射头组成，其实物如图1.1所示。



左信号发射头

右信号发射头

充电触片

圆泡信号发射头

图1.1 某款充电座实物图

由图可看出，该充电座有两片充电触片，用于和扫地机底盘的充电触片对接，从而进行充电。它还有三个红外发射头，每个发射头发射一种码值不同的红外信号，分别命名为左信号（B4）、右信号（B8）和圆泡信号（B1），每个信号都能覆盖一小片不同的区域，以便扫地机进行定位。该充电座发出的红外信号分布如图1.2所示。



图1.2 充电座的信号分布平面图

从图中可以看到，左信号和右信号覆盖的距离比较长，大概有3米左右，但是角度比较小，大概30度左右，所以看起来就像一个扇形一样。（不同型号的充电座发出的信号覆盖的距离和角度都是不一样的）

而圆泡信号刚好相反，它的覆盖半径只有50cm左右，但是其角度有360度，覆盖到的区域就像个圆一样，而且它的外观也是圆的，所以把它命名为圆泡信号。该信号可以防止机器在清扫和上座的过程中撞到充电座，就像一个护栏保护着充电座一样，所以有时候也叫它为护栏信号。

不同的区域有不同的信号，所以扫地机就可以根据接收头接收到的不同信号判断其所处的位置，进而根据信号的指引慢慢调整姿态使机器回到充电座上，最后使机器底盘的金属片和充电座的触片相接触，就可以成功的对机器进行充电了。

另外，如果发射头同时发射信号，那么信号之间就会相互叠加，形成一个新的码值，如：左信号和右信号叠加形成了BC信号。但是无论信号怎么叠加，都可以在程序中把信号分离出来，比如某接收头接收到BC信号，相当于同时接收到了左信号和右信号。

1.1.2 扫地机

扫地机上安装有红外接收头，用于接收红外信号。红外接收头主要分两种：

一种是普通红外接收头，其水平方向接收角大概是30°，可以安装在机器四周各个位置上；如某款机型安装了六个普通红外接收头，分别位于样机的前左、前右、左、右、后左、后右方向（除了前左和前右两个接收头面正对正前方，其余四个接收头都是与所在点的切线垂直的）。接收头的平面分布大体如图1.3所示。（图中的英文命名对应例程代码中的英文命名）



图1.3六个普通接收头分布平面图

实物及各红外接收头大体所在位置如图1.4所示：



前左

前右

左

右

后左

后右

图1.4 有六个普通接收头的样机实物图

另一种是圆泡接收头，其接收角是360°，像个圆一样，而且它本身也是圆的，所以把它命名为圆泡；它可以全方位接收到红外信号，一般安装在机顶前部。圆泡有定位的功能，可以帮助机器判断其所处的位置和机头的朝向，比如当圆泡接收到信号而机器前方的接收头接收不到信号，就可以判断机器位于信号区中，但是机头的正前方不是充电座的方向，所以需要旋转机头使其对准充电座。

如某款安装了圆泡的机型的接收头平面分布如图1.5所示。



图1.5 有圆泡的机型接收头分布平面图

实物及各红外接收头大体所在位置如图1.6所示：



前右

前左

圆泡

图1.6 有圆泡的样机实物图

扫地机的底盘前方还安装有两块金属片，只要它们和充电座的接触片成功对接，就能使机器进入充电模式进行充电。机器底盘上的两块金属片所在位置如图1.7所示。



机尾

机头

用于充电的金属片

图1.7 底盘结构图

1.1.3 扫地机如何上座

从前面的介绍，我们认识到充电座能发出几种码值不同的红外信号，每个信号都覆盖了一小片区域；扫地机上安装了几个用来接收红外信号的接收头。

当扫地机进入信号区域时，接收头就会接收到红外信号，然后软件判断是哪个接收头接收到了信号，是何种信号，从而就能判断出机器所处的位置，进而控制机器做前进、后退、左转、右转等动作来调整其姿态，使机器慢慢回到充电座上。

比如当机器处于图1.8的位置时怎样才能让机器成功回到充电座呢？

首先，当前左和前右两个接收头都能接收到BC信号时，我们就可以判断机器基本是正对着充电座的，这时我们可以让机器直行去靠近充电座；

直行过程中如果这两个接收头接收不到BC信号而是接收到了B8信号（如图中a点），则可以让机器左轮速度慢点，右轮速度快点，使机器微微往左偏一点，矫正一下机器，然后继续直行；

直行过程中如果这两个接收头接收不到BC信号而是接收到了B4信号（如图中b点），则可以让机器右轮速度慢点，左轮速度快点，使机器微微往右偏一点，矫正一下机器，然后继续直行；

如此往复，不断调整其姿态，就能让机器成功回到充电座上了。



图1.8 扫地机回座充电示例图

1.2软件逻辑

前面说过，扫地机先通过接收头接收到的红外信号的类型来判断其所处的位置，然后控制轮子不断地调整机器的姿态使机器回到充电座上进行充电。所以软件的逻辑无非就是判断信号类型，然后控制机器做出相应动作调整其姿态。

1.2.1 传统逻辑

使用if…else if…else…结构把所有的判断都放在同一个函数中，使用while循环不断地轮询每一个判断条件，然后执行相应的动作，这样就会使各个判断之间相互有了关联，使整个函数结构非常的大，而且逻辑也会很乱。当有需要更改某个条件的时候，往往牵一发而动全身，这非常不利于用户根据自己的想法开发出自己的回座充电功能。

传统逻辑如下：

if(判断条件1)

{

控制机器做动作1;

}

else if(判断条件2)

{

控制机器做动作2;

}

else if(判断条件3)

{

控制机器做动作3;

}

︰

︰

︰

1.2.2 引入基于APP-SDK的架构

而我们使用的基于APP-SDK的架构就很巧妙了，它把各个判断条件及其对应的机器动作都拆分成了一个个行为，行为之间相互独立，互不干扰，这样修改和操作起来就很方便了。

这里，引入了一个概念叫“behavior(行为)”,即如果其判断条件成立，则控制机器完成某些动作叫一个行为。其判断条件单独放在一个触发条件函数中；机器所需要做的动作放在另一个实现函数中。

比如根据上面的传统逻辑改写得到的行为如下：

行为1：

函数名1(void) // 行为1的触发条件函数

{

if(判断条件1)

{

return TRUE;

}

retuen FALSE;

}

函数名2(viod) //行为1的行为实现函数

{

控制机器做动作1;

}

行为2：

函数名3(void) // 行为2的触发条件函数

{

if(判断条件2)

{

return TRUE

}

return FALAE

}

函数名4(void) // 行为2的行为实现函数

{

控制机器做动作2;

}

︰

︰

︰

这样，就可以把全部行为的触发条件函数集中放到一个中断服务函数中进行轮询，然后再对条件成立的行为进行调用，如果同时有多个行为的条件都成立，则可以引入优先级，对优先级最高的行为进行调用。

我们把这种架构称为Dock架构，其逻辑大致如下：

1. 提前为每个行为都分配一个优先级。（关于行为的优先级，这里只须明白其作用，如何为行为分配优先级，将在2.1和2.2小节进行说明）
2. 申请一个每10ms中断一次的定时器，然后把全部行为的触发条件函数都放到该定时器的中断服务函数中，让其调用。（可以通过行为注册的方法把触发条件函数放到该定时器的中断服务函数中，行为注册在2.1小节进行说明)
3. 中断服务函数每隔10ms运行一次，对其中的触发条件函数进行扫描，看有哪些函数返回TRUE（触发条件函数返回TRUE表示该条件被触发）。
4. 如果同时有多个行为的触发条件被触发，则优先级最高的行为被执行，其余行为被忽略。所以各个行为之间的切换主要由优先级决定。
5. 高优先级行为可以直接打断低优先级行为。比如当前正在执行一个优先级为5的行为，这过程中有一个优先级为3的行为的触发条件被触发了，则系统会直接打断行为5而切换到行为3中。（行为的切换由系统自动完成，我们暂时不用理会）

该架构的逻辑如图1.9所示：



图1.9 dock架构逻辑图

为了方便理解，前面从整体上介绍了dock架构的逻辑，但实际上我们把这个架构的代码放在了三个不同的层之中，如图1.10所示



图1.10 dock架构层次图

Core层：该层主要是实现行为的切换，它被封装在了SDK中，普通用户无法看到。回座行为当前最多支持32个，最高优先级为0，最低优先级为31。

Dock层：该层实现回座的行为，普通用户可以看见，并且该层代码由用户根据实际充电座和机型的差异进行修改。

硬件层：该层封装在了SDK中，普通用户不可见，但是提供了相应API。用户通过dock层调用相应API就可以控制轮子做出相应动作，从而实现上座。

这三个层之间的配合流程如图1.11所示。

图1.11 cock三个层的配合流程图

# 2、DOCK的重要基础知识

前面提到，整个dock概括起来所需要做的无非就是判断信号类型（判断条件），然后控制机器做出相应动作调整其姿态（行为实现）这两件事。而为了更方便的判断信号类型，就需要对信号进行相应的封装；为了管理判断条件，引入了debouncer结构体；为了管理behavior（行为，该概念在1.2.2节有提到），引入了Dock\_Data结构体；最后用一个个的behavior切换来实现dock（其中最重要的是behavior和debouncer）。所以只要把这几个重要内容掌握了，就可以进行实际的dock开发了。其涉及到的代码都在下面几个文件中：

……\app-project\src \dock\

├──docking-core.h

├── docking-new.c

├──docking-new.h

├── docking-sensors.c

docking-new.c 中实现了各个行为， docking-sensors.c 对接收的红外信号进行了封装。

## 2.1 behavior

2.1.1 Dock\_Data

在介绍behavior之前，先认识一下Dock\_Data这个结构体，因为dock中用到的每个行为，都必须使用该结构体进行注册才能生效，该结构体定义如下：

typedef struct

{

dock\_function current\_function;

judge\_condition start\_when;

judge\_condition run\_when;

judge\_condition abort\_when;

abort\_code abort\_code;

AM\_BOOLEAN last\_start\_state;

am\_int8 priorty;

}Dock\_Data;

current\_function指针，指向对应的行为实现函数。

start\_when指针，指向边沿触发条件函数，此函数的返回值由FALSE变为TRUE时，此行为触发。

run\_when指针，指向电平触发条件函数，此函数返回TRUE时，行为启动，返回FASLE，行为马上结束。

abort\_when指针，指向的函数返回TRUE时，行为退出，与start\_when配合使用。

abort\_code指针，在行为退出时，执行其指向的函数。

last\_start\_state，记录上一次start\_when的状态。

priorty，代表此behavior的优先级，0为最高优先级，31为最低优先级。

注意：start\_when和run\_when都指向触发条件函数，所以不能同时有效，只能用其一。

2.1.1.1 边沿触发和电平触发的区别

start\_when：边沿触发，当其上一次状态为FALSE，当前状态为TRUE,且没有更高优先级的行为正在运行时，该行为启动。并且在该行为运行的过程中，即使该触发条件变为了FALSE，该行为没退出之前，比它低的行为不能中途打断该行为。只有比它高的行为优先级才能中途打断该行为。

run\_when： 电平触发，只要其返回TRUE，且没有更高优先级的行为正在运行，则该行为启动。其返回FALSE，则该行为立即退出。所以如果一个优先级比较高的行为使用这种触发方式，当其信号发生很短暂的丢失时，行为马上退出切换到更低优先级行为，看起来就像是低优先级行为打断了高优先级行为一样，但其实是因为信号的丢失造成的。

注意：因为机器接收红外信号的不稳定性，电平触发方式很容易被中途打断，所以一般都是使用边沿触发方式。在实际应用中如果遇到高优先级行为看起来像被低优先级行为打断了，多半是没有注意这两个触发条件的使用。

2.1.2 behavior详解

在1.2.2小节中，简单提到过behavior这个概念，当时只提到过其中最重要的两个函数：触发条件函数和行为实现函数，现在完整介绍如下：

一般的行为，主要包含以下几个函数：（前面三个函数必须有，其它的视情况而定）

①行为触发条件：系统每10ms扫描一次该函数，就像按键扫描一样。（该函数会被注册到一个中断服务函数中）

②行为实现：当触发条件为真且它的行为优先级高于当前正在运行的行为的优先级时，系统就会切换到该行为，调用该函数，进而控制机器做相应运动。

③行为注册：把该行为优先级和涉及到的函数注册到SDK中，系统才能根据优先级对该行为进行调度。

注册方法：在dock\_new\_init()函数中调用一次该行为注册函数即可。

④行为退出条件：系统以一定的频率扫描该函数，函数为真，则该行为退出。

⑤行为退出时系统调用的函数：一般用于清零该行为涉及到的一些变量。

自己编写行为的时候需要考虑三个因素：一是行为触发条件是什么；二是行为触发之后机器做什么动作；三是把行为的优先级定为多少级合适。

考虑好之后只要按照这个框架编写就可以写出自己想要的行为了。

例如，按照以上几个要素写出的矫正左右摆行为的框架如下：

BOOLEAN docking\_correct\_start\_when(void) //行为触发条件

{

if (判断条件)

{

return TRUE;

}

else

return FALSE;

}

DOCK\_FN\_DECL(docking\_correct) //行为实现

{

控制机器完成相应的动作;

return ;

}

void set\_docking\_correct\_abort(void) //用户在行为实现中调用该函数可使本行为退出

{

docking\_correct\_abort = TRUE;

return;

}

BOOLEAN docking\_correct\_abort\_when(void) //行为退出条件

{

if (docking\_correct\_abort != FALSE)

return TRUE;

else

return FALSE;

}

void docking\_correct\_abort\_code(void) //行为退出时由系统自动调用该函数

{

docking\_correct\_abort = FALSE;

return;

}

void dock\_correct\_register(void) //行为注册

{

Dock\_Data dock\_funtion; //注册行为所使用的结构体

dock\_funtion.priorty = DOCKING\_CORRECT; //优先级，0为最高优先级

dock\_funtion.start\_when = &docking\_correct\_start\_when; //行为触发条件（电平触发）

dock\_funtion.run\_when = NULL; //行为触发条件（边沿触发）

dock\_funtion.abort\_when = &docking\_correct\_abort\_when; //行为退出条件

dock\_funtion.abort\_code = &docking\_correct\_abort\_code; //行为退出时系统调用的函数

dock\_funtion.last\_start\_state = FALSE; //记录上一次start\_when的状态

dock\_funtion.current\_function = docking\_correct; //行为实现

register\_dock\_function(&dock\_funtion); //把该行为注册到系统中

return;

}

（注意：dock\_funtion.start\_when和dock\_funtion.run\_when都是触发条件函数，所以不能同时有效，只能用其一）

然后在dock\_new\_init()函数中调用一次该行为注册函数dock\_correct\_register完成注册，最后把代表该行为优先级的DOCKING\_CORRECT这个宏放到Bhavior\_Id这个枚举中相应的位置，为该行为定一个优先级，这样一个完整的行为就完成了，接下来就可以在Bhavior\_Id这个枚举中随时调整行为的优先级进行行为的调试了。

## 2.2 行为优先级

前面很多地方都提到了行为的优先级，其重要性不言而喻，它是行为切换的依据。

通过调整下面的枚举元素的位置，就可以把相应的行为设置成0至31之中的任何一个优先级。（其中的名称对应行为注册时使用的名称）

行为优先级列举如下：

typedef enum

{

DOCKING\_BOUNCE, //0 优先级最高

DOCKING\_SUCCESS, //1

DOCKING\_FORCE\_FIELD, //2

DOCKING\_AVOID\_OBSTACLE, //3

DOCKING\_DOCK\_AHEAD, //4

DOCKING\_LEFT\_RIGHT, //5

DOCKING\_FIND\_BUOY, //6

DOCKING\_LINE\_BOUNCE, //7

DOCKING\_GO\_FORWARD, //8

DOCKING\_LEFT, //9

DOCKING\_RIGHT, //10

DOCKING\_CORRECT, //11

DOCKING\_ONLY\_FORCE\_FIELD, //12

︰

︰

︰

DOCKING\_LINE //31 优先级最低

}Bhavior\_Id;

2.2.1 行为的切换

各个行为之间如何进行行为的切换,在1.2.2小节也有提到，因其重要性，这里再说一下：

首先，在各个行为的注册函数中使用register\_dock\_function()函数注册行为的时候，每个行为都附带有一个优先级，优先级高的行为如果被触发则可以直接打断优先级低的行为，就像中断的优先级一样。

其次，core层中有一个定时器中断服务函数，中断到来时对各个行为的触发条件函数进行扫描，看该行为是否达到触发条件。如果有行为达到了触发条件，则开始判断它的优先级是否高于当前行为的优先级，进而判断是否切换行为。

这种dock架构的各个行为之间是相互独立的，它们的调度由优先级决定，然后系统内部自动完成，这样就可以实现代码的低耦合性，层次分明，利于代码的编写与重用。

## 2.3 红外信号封装

在behavior中对行为触发条件进行判断时，其主要是对红外信号的判断，所以这一节介绍如何对红外信号进行封装，给debouncer调用（2.4节提到）。

红外信号宏定义如下所示：

#define YUANPAO\_BEACON\_BYTE 0xB1 //圆泡信号

#define LEFT\_BEACON\_BYTE 0XB4 //左信号

#define RIGHT\_BEACON\_BYTE 0XB8 //右信号

#define BOTH\_BEACON\_BYTE 0xBC //左右混合信号

首先，下面介绍三个最基础的红外信号封装函数，dock用到的所有用于判读信号的函数都是基于这三个函数进行封装的，所以这三个函数是所有信号函数的基石：

AM\_BOOLEAN force\_field(IR\_local\_Index chan)

{

Return ((dock\_signals[chan]&YUANPAO\_BEACON\_BYTE) == YUANPAO\_BEACON\_BYTE);

}

AM\_BOOLEAN buoy\_left(IR\_local\_Index chan)

{

Return ((dock\_signals[chan]&LEFT\_BEACON\_BYTE) == LEFT\_BEACON\_BYTE);

}

AM\_BOOLEAN buoy\_right(IR\_local\_Index chan)

{

Return ((dock\_signals[chan]&RIGHT\_BEACON\_BYTE) == RIGHT\_BEACON\_BYTE);

}

force\_field函数表示某个红外接收头是否收到圆泡信号。

buoy\_left函数表示某个红外接收头是否收到左侧信号。

buoy\_right函数表示某个红外接收头是否收到右侧信号。

然后，使用这三个的函数进行进一步的组合封装，就可以得到自己想要的任何一种效果。所以在dock开发中想要何种接收头接收到何种信号自己进行封装就行了。

例如：下面这个函数表示左边接收头是否接收到右边信号。

BOOLEAN check\_recently\_left\_right(void)

{

return (buoy\_right(IR\_LOCAL\_LEFT));

}

再如：下面这个函数表示前右接收头是否同时接收到了左信号和右信号。

BOOLEAN check\_docking\_go\_forward\_right(void)

{

if (buoy\_right(IR\_LOCAL\_MID\_RIGHT) && buoy\_left(IR\_LOCAL\_MID\_RIGHT))

return TRUE;

else

return FALSE;

}

## 2.4 Debouncer定义

这里将介绍一种在dock中最重要的结构：Debouncer，它主要是用来管理信号触发条件的，比如用来管理上节封装好的函数。

而为什么不直接使用上节封装好的函数进行判断而是把它进一步封装成debouncer结构呢？因为充电座发射红外信号时，码值之间都有一个时间间隔（如某机型充电座的码值间隔在90ms左右），而机器解码的速度为10ms，比信号发射码值快的多，在这个时间间隔里机器收不到信号就会把上一次接收到的信号清零，从而使机器进入信号真空期。因此为了使相应的触发条件在这个时间间隔内不至于马上失效，需要对相应的触发条件进行相应的延时（相当于对接收到的信号进行保存）。（封装好的Debouncer可以用于所有需要信号判断的地方）

其结构体如下：

typedef struct

{

judge\_condition predicate;

am\_uint16 trigger\_on;

am\_uint16 trigger\_off;

am\_uint16 on\_count;

am\_uint16 off\_count;

AM\_BOOLEAN current\_state;

set\_context set\_dock\_context;

}Debouncer\_Data;

predicate指针，指向对应的信号触发函数，如上面2.3节末尾提到的二类封装函数。

tigger\_on，记录on门限值，当连续N次触发函数返回TURE，且N大于等于tigger\_on时，current\_state为TRUE。

tigger\_off，记录off门限值，当连续N次触发函数返回FALSE，且N大于等于tigger\_off时，current\_state为FALSE。

on\_count，off\_count，为core层内部使用。

current\_state，此debouncer当前的状态（TRUE或FALSE）。

set\_dock\_context，设置上下文环境。用于在特定的环境设置样机对应的参数。如靠近充电座时，对应的debouncer为TRUE时，提高cliff阈值，关闭light touch等。远离充电座时，再恢复其对应的参数。

例如：下面把函数check\_near\_dock()封装成debouncer

Debouncer\_Data recently\_near\_dock = {

.predicate = &check\_near\_dock,

.trigger\_on = 1,

.trigger\_off = 200,

.on\_count = 0,

.off\_count = 0,

.current\_state = FALSE,

.set\_dock\_context = &set\_near\_dock\_context

};

check\_near\_dock：该信号函数表示机器是否靠近充电座。

trigger\_on =1：只要有一次检测周期检测到check\_near\_dock函数为TRUE，其current\_state就会马上变为TRUE。（检测周期是10ms）；

trigger\_off =200：只有当连续200个检测周期都检测到check\_near\_dock函数为FALSE时，其current\_state才会从TRUE变为FALSE。（检测周期是10ms）；

current\_state = FALSE：current\_state默认为FALSE；

set\_near\_dock\_context：当靠近充电座时，提高slip，黑白轮，light\_touch，cliff的门限参数，当远离座子时恢复其参数。

所以，通过Debouncer这个结构体，可以对相应的信号触发条件的状态进行一个延时保存，如上面这个例子，当其current\_state状态为TRUE时，需要check\_near\_dock函数连续2000ms都为FALSE，current\_state状态才会变回FALSE，所以current\_state的状态至少保存了2000ms。

在实际应用中可以根据实际情况修改这个trigger\_off的值，从而达到自己想要的效果。

注意：这些Debouncer必须使用register\_debouncer()函数注册到core层中才能生效。如：上面举例的recently\_near\_dock，只要在docking\_sensors.h中添加一句

extern Debouncer\_Data recently\_signal;声明一下，然后再在dock\_new\_init()函数中添加一句register\_debouncer(&recently\_signal);就可以完成注册了。

## 2.5 机器运动控制

机器运动的控制，主要使用下面两个宏和一个函数来完成：

①AM\_GO\_TO\_PLACE：该宏控制机器旋转一个角度。

AM\_GO\_TO\_PLACE(rotation\_angle,l\_speed,r\_speed,condition,care\_bump,result)

rotation\_angle：需要旋转的角度，正数表示逆时针旋转，负数表示顺时针旋转；

l\_speed：左轮速度；

r\_speed：右轮速度；

condition：旋转条件，该条件的值为TRUE时，机器旋转；旋转过程中如果该值突然变为FALSE，则马上停止旋转；

care\_bump：是否考虑碰撞或悬崖，使用CARE\_BUMP和CARE\_CLIFF这两个宏进行设置；

result：是否成功转到目标角度；返回0：成功；返回1:发生cliff；返回2：发生bump

例如：

S8 result = 0;

AM\_GO\_TO\_PLACE(180,120,120,(!docking\_ahead\_start\_when()), CARE\_CLIFF,result);

它能控制机器以左右轮都是120的速度进行原地逆时针旋转180°，旋转过程中如果(!docking\_ahead\_start\_when())这个条件从TRUE变成了FALSE，则机器马上停止旋转；如果机器检测到了cliff也会停止旋转；result用于存储返回的结果。

②DRIVE\_GO：该宏控制机器直行一段距离。

DRIVE\_GO(distance,speed,condition,care\_bump,result)

distance：需要直行的距离，单位毫米；

speed：直行的速度；

condition：直行条件，该条件的值为TRUE时，机器直行；直行过程中如果该值突然变为FALSE，则马上停止运动；

care\_bump：是否考虑碰撞或悬崖，使用CARE\_BUMP和CARE\_CLIFF这两个宏进行设置；

result：是否成功到达目的地；返回0：成功；返回1:发生cliff；返回2：发生bump

例如：

S8 result = 0;

DRIVE\_GO(300,240,(!recently\_left\_left.current\_state),(CARE\_CLIFF|CARE\_BUMP),result);

它能控制机器以240的速度直线前进300mm，直行过程中如果(!recently\_left\_left.current\_state)这个条件从TRUE变成了FALSE，则机器马上停止；如果机器检测到了cliff或bump也会停止旋转；result用于存储返回的结果。

③set\_motor\_vels()：该函数驱动轮子行走

EXPORT void set\_motor\_vels( int16\_t left,int16\_t right, uint16\_t acc);

left：左轮的速度 ， =0：停止； <0 后退 ； >0，前进 单位是mm/s

right：右轮的速度 ， =0， 停止； <0 后退； >0，前进 单位是mm/s

acc：加速度 ， 单位mm/s

例如：

set\_motor\_vels(100, 200, 600);

它能控制机器以600的加速度，左轮100、右轮200的速度行走，即机器向左前方行驶。

## 2.6 行为编写举例

如何按照自己的思路写出自己的行为呢？每个行为最主要的是确定其触发条件，只有确定了触发条件，才能在行为实现中操作机器完成自己想要的动作。

比如：在图2.1中，正当扫地机正对着充电座上座时机器往左偏了一下（如图中a点），此时机器的中右接收头刚好接收到了左信号，那你是不是需要写一个行为来应付这种情况呢？



图2.1行为编写举例图

首先，确定其触发条件，即中右接收头刚好接收到了左信号，用代码表示如下：

BOOLEAN check\_docking\_right(void)

{

if (buoy\_left(IR\_LOCAL\_MID\_RIGHT))

return TRUE;

else

return FALSE;

}

然后把它封装成debouncer形式，代码如下：

Debouncer\_Data recently\_docking\_right = {

.predicate = &check\_docking\_right,

.trigger\_on = 1,

.trigger\_off = 40,

.on\_count = 0,

.off\_count = 0,

.current\_state = FALSE,

.set\_dock\_context = NULL

};

对该debouncer进行注册一下就可以使用了，即在docking-sendors.h中添加如下代码：

extern Debouncer\_Data recently\_docking\_right;

然后再在docking\_new.c文件的dock\_new\_init()函数中添加如下代码：

register\_debouncer(&recently\_docking\_right);

至此，准备工作做完之后就可以开始写行为了。

首先编写行为触发条件函数，代码如下：

BOOLEAN docking\_right\_run\_when(void)

{

if (recently\_docking\_right.current\_state)

return TRUE;

else

return FALSE;

}

然后编写行为实现函数，因为触发条件是因为机器往左偏了一点，所以我们需要控制机器往右偏一下，使机器重新对着充电座上座。代码如下：

DOCK\_FN\_DECL(docking\_right)

{

TransVel left\_vel;

TransVel right\_vel;

left\_vel = 150；//左轮速度

right\_vel =110;//右轮速度

set\_motor\_vels(left\_vel, right\_vel, 600);

return ;

}

我们现在暂时不需要行为退出条件函数，如果你觉得需要，可以自己添加。

最后编写行为注册函数，代码如下：

void dock\_right\_register(void)

{

Dock\_Data dock\_funtion;

dock\_funtion.priorty = DOCKING\_RIGHT;

dock\_funtion.start\_when = NULL;

dock\_funtion.run\_when = &docking\_right\_run\_when;

dock\_funtion.abort\_when = NULL;

dock\_funtion.abort\_code = NULL;

dock\_funtion.last\_start\_state = FALSE;

dock\_funtion.current\_function = docking\_right;

register\_dock\_function(&dock\_funtion);

return;

}

函数都写完之后，别忘了进行行为的注册，即在docking\_new.c文件的dock\_new\_init()函数中调用一下该行为注册函数，也就是添加如下代码：

dock\_right\_register();

到这里，一个全新的行为就完成了，最后需要在实际调节过程中确认其优先级到底定为哪一级比较好，所以只好待君慢慢尝试了。优先级在如下枚举类型中：

typedef enum

{

DOCKING\_SUCCESS,

DOCKING\_GO\_FORWARD,

DOCKING\_RIGHT,

}Bhavior\_Id;

**3、DOCK设计思想**

通过前面的基础知识学习，想必你已经能按照自己的思路写出自己的行为了，那么这里将介绍一种例程代码中用到的dock思路，帮助你进一步理解dock的思想。

将充电座信号区分成四种区域，如下图所示。



区域1，扫地机可以同时收到B4和B8，表明充电座就在扫地机的前方。

区域2，扫地机只收到B4或B8信号，说明充电座的正前方就在附近。

区域3，扫地机只收到B1信号，说明当前扫地机离充电座很近，有可能撞到充电座。

区域4，扫地机什么信号都收不到，因此跑随机，搜索信号。

因为区域一是最方便机器成功上座的，所以开发时，先从区域一开始调试，只有把这个区域的上座成功率调到百分之九十以上了，才可以开始其他区域的调试。当机器位于其他区域时，遵循的一个中心思想就是如何让机器回到区域一，只要机器回到了区域一，它就可以切换到区域一的行为顺利上座了。所以最好是先调区域一，再到区域二，其余两个区域放最后。

3.1 区域一



扫地机在区域一时，扫地机根据正前方的两个红外接收头(即前左和前右接收头)接收到B4，B8两个信号的情况，实时调整样机姿态，使得扫地机能够精准回到充电座，进行充电。

在这个区域，例程代码中使用了docking\_go\_forward（直行）、docking\_left（左摆：向左走弧形）、docking\_right（右摆：向右走弧形）、docking\_correct（矫正左右摆）这四个行为使机器精准上座。

3.2 区域二



扫地机在区域二时，可以收到F4或F8信号，实现如下行为，去寻找中间线（即区域一），并将扫地机姿态调整为正对着充电座。所以在该区域中需要思考的就是如何让机器回到区域一中。

在这个区域，例程代码中只使用了docking\_left\_right行为让机器寻找中线，回到区域一，并将扫地机的正前方对着充电座，然后此行为退出，将控制权交给区域一的行为。

docking\_left\_right：每前进一段距离（如20cm），转向充电座，看是否扫地机已经位于区域一中。如果没有则转回直行方向，继续前进一小段距离，继续转向充电座，………，直到样机位于区域一中，将扫地机正对着充电座，行为退出。

3.3 区域三



如图所示，扫地机收到B1时，表明扫地机离充电座很近了。

如果这时，扫地机前面红外接收头没有同时收到B4和B8（即不是位于区域一），此时启动绕充电座行为，围绕圆泡信号的边沿进行运动。

当收到B4或B8信号（如图a点），掉头，直走一段距离，然后转回原来的方向（如图b点），最后此行为退出，将控制权交给区域一的行为。所以在该区域中需要思考的就是如何让机器回到区域一中。

在这个区域，例程代码中使用了docking\_force\_field、docking\_find\_buoy两个行为让机器回到区域二，接着通过区域二行为使机器回到区域一。

docking\_force\_field：实现绕座功能。当扫地机圆泡接收头，收到F1时，远离座子走弧线；当扫地机圆泡接收头，收不到F1时，靠近座子走弧线；直到扫地机有接收头收到座子发出的左右信号，启动docking\_find\_buoy行为，docking\_force\_field行为退出。

docking\_find\_buoy：在docking\_force\_field退出后启动，寻找左右(F4或F8)信号。扫地机每前进一段距离（如20cm），转向充电座，看是否扫地机可以收到左右（F4或F8）信号，样机直行方向及充电座方向，docking\_force\_field行为在启动docking\_find\_buoy行为前都已经设置好了。

3.4 区域四



扫地机在区域四时接收不到任何红外信号，扫地机跑随机，寻找座子发出的红外信号。

只要找到了信号，就可以根据区域二或区域三的行为回到区域一了。

在这个区域，例程代码中使用了docking\_line和docking\_line\_bounce两个行为。

docking\_line：实现走直线，它的运行条件永远有效。但其优先级为31，所有行为中优先级最低。因此当其它行为都无效时，它才会运行。

docking\_line\_bounce：实现遇到障碍物时，随机转一个角度。

**4、DOCK开发步骤**

下面通过三个步骤教你如何实际调试机器的回座充电功能。

4.1配置接收头对应I/O

在hal\_amicro\_gpio.h文件中找到以下宏定义，进行相应I/O配置，对应关系请看图1.2。

#define AM\_IO\_IR\_REC\_MIDDLE\_LEFT GPIOF(1)

#define AM\_IO\_IR\_REC\_MIDDLE\_RIGHT GPIOE(12)

#define AM\_IO\_IR\_REC\_RIGHT GPIOF(2)

#define AM\_IO\_IR\_REC\_LEFT GPIOF(3)

#define AM\_IO\_IR\_REC\_BACK\_LEFT GPIOB(6)

#define AM\_IO\_IR\_REC\_BACK\_RIGHT GPIOB(7)

因为例程代码所使用的机型没有圆泡，所以如果需要添加圆泡，则自己在该文件中添加一个宏，名字自取，配上相应I/O，然后在ir\_gpio[IR\_DECODER\_MAX]这个数组中加入刚才自己定义的宏，同时在IR\_local\_Index和IR\_REMOT\_POSITION\_E这两个枚举中都添加一个元素。然后把#define REMOTE\_DECODER\_MAX 6 这个宏的值该为7；同时把#define IR\_DECODER\_MAX 6 这个宏的值也改为7就可以了。

配好I/O之后，打开robot\_get\_dock\_signals()函数中的

dprintf(DEBUG\_DOCK\_SIGNAL, "IR%d : %x \r\n", index, dock\_signals[index]);这个打印，就可以用串口助手看到接收到的信号了，同时它会打印出来是哪个接收头接收到的信号，然后一个一个接收头地检验是否都能正常接收到数据了。

4.2 充电座信号

4.2.1 确认信号类型

把充电座拆开，看其有多少个红外发射头，一般来说都有左、右、圆泡三个信号发射头，有一些充电座还有中间发射头，或者同时存在两个圆泡信号发射头。然后遮住其它发射头，只留一个，用机器接收该信号然后打印出来就能得到该发射头的码值。全部码值测出来之后，修改以下几个宏就可以了（docking-new.h文件中）：

#define DOCK\_CLOSE\_BEACON 0xB1 ----------->圆泡信号

#define LEFT\_BEACON\_BYTE 0xB4 ----------->左信号

#define RIGHT\_BEACON\_BYTE 0xB8 ----------->右信号

#define BOTH\_BEACONS\_BYTE 0xBC ----------->左和右的叠加信号

4.2.2 确认接收到的信号是否正常

先用串口打印看其每秒钟接收到的信号是否很快（如每秒钟打印二三十个信号），如果  
是，则表明信号接收正常，如果每秒钟只打印十个、几个信号，则信号接收肯定不正常。但  
最保险的办法就是用示波器看信号波形。因为充电座的信号中加入了载波，所以不能直接用  
示波器测量充电座电路中的信号，而是把充电座正对着扫地机，然后用示波器测量扫地机接  
收头电路中的信号。正常情况下，从示波器中可看到的波形如图4.1所示。

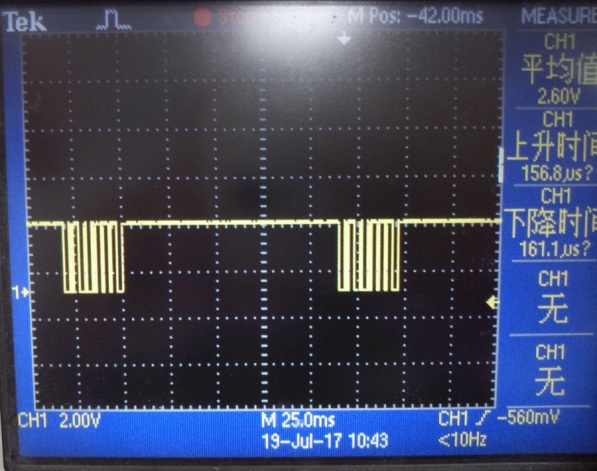


图4.1 正常信号波形图

对波形进行放大，看其高低电平持续的时间，其中3ms低电平连着1ms高电平表示逻辑“1”，1ms低电平连着3ms高电平表示逻辑“0”，而一个码值由8个逻辑电平组成，码值之间间隔90ms左右。然后对照ir\_decode.c中的remote\_decode\_ir()红外解码程序，看其能否以很低的误码率把码值解出来（注意：程序中的逻辑和示波器的逻辑刚好相反，即程序中3ms高电平1ms低电平表示逻辑“1”；1ms高电平3ms低电平表示逻辑“0”）。

4.2.3 确认充电座的信号分布

因为每种型号的充电座的信号分布都不一样，所以编写行为之前必须先确认其信号分布，才有可能写出自己想要的行为。把充电座放在一个比较宽敞的地方，然后用串口打印的方式确认充电座周围的信号分布，然后划分信号区域，比如某机型的信号分布如图1.1所示。所以可以把它划分为4个区域，如图4.1所示



图4.1 充电座信号区域划分图

4.3上座调试

4.3.1 确定上座思想

第三章已经对dock的设计思想做了详细说明，这里就不重复了，如果忘记了可以回头去再看一下。而这个设计思想也不是固定不变的，如果自己有想法，完全可以按照自己的想法编写自己的行为，这里只是提供一个参考。

4.3.2 行为编写

把上座思想编写成一个个行为，定好每个行为的优先级，然后系统根据优先级不断切换相应的行为实现上座。至于如何编写行为，已经在第二章详细说过了，这里就不再赘述了。

4.3.3 实际调试

因为对信号分了区域，所以调试时也是一个一个区域的去调。第三章也说过，因为区域一是最容易实现上座的，所以最好是先调区域一，再到区域二，其余两个区域放最后。