

***Package na***

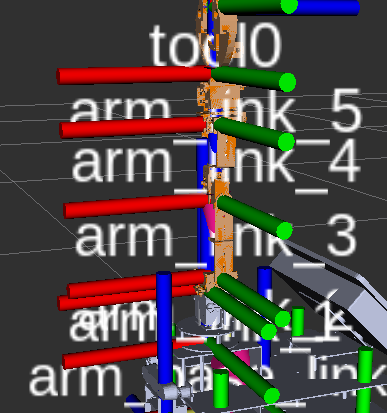
|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

# 关于机械臂关节舵机驱动

|  |
| --- |
| def deocdePos(self, name, msg):  # 这里的/xcar/arm消息为6轴,其中位于列表第一个参数可以为0进行填充,后面5个列表参数如下  jn = ["arm\_joint\_5", "arm\_joint\_4", "arm\_joint\_3", "arm\_joint\_2", "arm\_joint\_1"]  # 由于机械臂模型中夹具旋转轴表示为joint5，但是发布的消息中为关节1，这里数据要颠倒并且转换为角度  direction = [1, -1, -1, 1, 1]  offset = [ 0, 0, 0, 0, 0]  sv = []  i = 0  for i in range(len(jn)):  try:  pos = msg.positions[name.index(jn[i])]  pos = int(pos \* (180 / 3.1415926)) \* direction[i] + offset[i]  except:  pos = 0  sv.append(pos)  return sv |



由于上层应用中已经将模型中关节的方向确定了，驱动层已经帮我们把这个模型中的方向调整了，所以这里我们基本上不需要管deocdePos函数中方向的确定性。

|  |
| --- |
| <!--0.085 -0.01 Important parameters-->  <joint name="arm\_joint\_1" type="revolute">  <origin rpy="0 0 0" xyz="0.085 -0.01 0.18"/>  <parent link="base\_link" />  <child link="arm\_link\_1" />  <axis xyz="0 0 1" />  <limit effort="2.5" velocity="3.49" lower="-1.57" upper="1.57" />  </joint>  <!---0.01 0.019 Important parameters-->  <joint name="arm\_joint\_2" type="revolute">  <origin rpy="0 0 0" xyz="-0.01 0.015 0.019"/>  <parent link="arm\_link\_1" />  <child link="arm\_link\_2" />  <axis xyz="0 1 0" />  <limit effort="2.5" velocity="3.49" lower="-0.785" upper="2.35619" />  </joint>  <!--0.103 Important parameters-->  <joint name="arm\_joint\_3" type="revolute">  <origin rpy="0 0 0" xyz="0 0 0.103"/>  <parent link="arm\_link\_2" />  <child link="arm\_link\_3" />  <axis xyz="0 1 0" />  <limit effort="2.5" velocity="3.49" lower="-2.35619" upper="2.35619" />  </joint>  <!--0.098 Important parameters-->  <joint name="arm\_joint\_4" type="revolute">  <origin rpy="0 0 0" xyz="0 0 0.098"/>  <parent link="arm\_link\_3" />  <child link="arm\_link\_4" />  <axis xyz="0 1 0" />  <limit effort="2.5" velocity="3.49" lower="-2.35619" upper="2.35619" />  </joint>  <!--0.063 Important parameters-->  <joint name="arm\_joint\_5" type="revolute">  <origin rpy="0 0 0" xyz="0 0 0.063"/>  <parent link="arm\_link\_4" />  <child link="arm\_link\_5" />  <axis xyz="0 0 1" />  <limit effort="2.5" velocity="3.49" lower="-1.57" upper="1.57" />  </joint> |

如上就是我们协议中V1参数的关节角度范围和方向。

|  |
| --- |
|  |

# linux系统中的信号量、线程、互斥量

<https://www.bilibili.com/read/cv17513349/>

## LINUX系统编程之线程

### 二、操作

void \*fun(void \*arg)

注意线程函数参数和返回值类型

pthread\_t pth;

创建线程pthread\_create(&pth, NULL, fun, (void \*)arg);（可用结构体或数组传递多个参数）

等待线程结束回收其资源pthread\_join(pth, NULL);

分离线程pthread\_detach(pth);

退出线程pthread\_exit();

取消线程pthread\_cancle();

取消状态pthread\_setcancelstate();

取消类型pthread\_setcanceltype();

设置取消点pthread\_testcancel();

清理pthread\_cleanup\_push();pthread\_cleanup\_pop();两个函数必须成对存在

编译gcc a.c 加-lpthread

gtk编程中多个线程可能使用同一资源照成界面冻结，所以要线程互斥

可使用gtk\_threads\_enter();和gtk\_threads\_leave();实现 作者：云V小编 https://www.bilibili.com/read/cv17513349/ 出处：bilibili

### 三、线程的同步和互斥

互斥：多个任务访问同一公共资源，同一时刻只有一个任务可以访问

互斥锁和信号量

1.互斥锁：mutex，上锁解锁两种状态，解锁必须由上锁者完成

申请mutex,如果lock则阻塞申请者

pthread\_mutex\_t mutex;

pthread\_mutex\_lock(&mutex);

pthread\_mutex\_trylock(&mutex);

pthread\_mutex\_unlock(&mutex);

pthread\_mutex\_destroy(&mutex); 作者：云V小编 https://www.bilibili.com/read/cv17513349/ 出处：bilibili

2.信号量

非负的整数计数器

对信号量进行减操作，如果为0则阻塞

PV原语，P减，V加

sem\_t sem;

sem\_init(&sem, 0, 1);

sem\_wait(&sem);

sem\_trywait(&sem);

sem\_post(&sem);

int val;

sem\_getvalue(&sem, &val);

sem\_destroy(&sem);

通过信号量同步操作实现多任务之间按照顺序运行

线程：无名信号量，进程：有名信号量

一个任务一个信号量

有名信号量

sem\_t \*sem\_open("sem", O\_RDWR);

sem\_close(sem);

sem\_unlink("sem");

有名信号量的名字在程序中和文件系统中不一样

有名信号量会保存之前的值所以使用前应该先删除再创建 作者：云V小编 https://www.bilibili.com/read/cv17513349/ 出处：bilibili

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

## 消息队列

<https://blog.csdn.net/m0_60970579/article/details/120463151>

<https://blog.csdn.net/fukangwei_lite/article/details/120617007> Linux消息队列

### 消息格式

在消息队列中，每一个消息的格式为struct msgbuf：

struct msgbuf {

long mtype; // 消息类型

char mtext[1]; // 消息数据的首地址

};

### 创建或打开消息队列（msgget）

创建一个新的或打开一个已经存在的消息队列。不同的进程调用此函数，只要用相同的key值就能得到同一个消息队列的标识符

|  |
| --- |
| #include <sys/msg.h>  #include <sys/ipc.h>  #include <sys/types.h>  int msgget(key\_t key, int msgflg); // 用于得到一个已存在的消息队列标识符或创建一个消息队列对象  // key表示消息队列的键值，这个键值用于标识一个消息队列，可以自定义一个键值，或者使用ftok函数生成  // key如果取宏IPC\_PRIVATE(0),表示创建一个私有队列，这个理论上只可以被当前进程所访问  // msgflg表示创建或者访问消息队列的具体方式，取值如下：  // IPC\_CREAT(01000): 如果消息队列对象不存在，则创建之  // IPC\_EXCL(02000): 如果消息队列对象存在，则报错  // msgflg往往使用多个值进行“或”操作来取值，至于原因，还需要专门探究，我目前创建消息队列时使用的msgflg值为：IPC\_CREAT|06000  // 返回值：执行成功，则返回一个正数作为消息队列的标识符，其他操作消息队列的函数将会使用这个标识符；执行失败，返回-1。 |

int msgget(key\_t key, int msgflg);

参数：

key：IPC键值

msgflg：标识函数的行为及消息队列的权限

msgflg的取值：

IPC\_CREAT：创建新的消息队列。

IPC\_EXCL：与IPC\_CREAT一同使用，表示如果要创建的消息队列已经存在，则返回错误。

IPC\_NOWAIT：当消息队列无法满足读写需求时，msgsnd与msgrcv不被阻塞。

位或权限位：消息队列位或权限位后可以设置消息队列的访问权限，格式和 open函数的mode\_t一样，但可执行权限未使用。

返回值：

成功：消息队列的标识符

失败：-1

系统调用：msgget();

原型：intmsgget(key\_t key,int msgflg);

返回值：如果成功，返回消息队列标识符

如果失败，则返回-1：

errno=EACCESS(权限不允许)

EEXIST(队列已经存在，无法创建)

EIDRM(队列标志为删除)

ENOENT(队列不存在)

ENOMEM(创建队列时内存不够)

ENOSPC(超出最大队列限制)

### 获取和设置消息队列的属性（msgctl)

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  int msgctl(int msqid, int cmd, struct msqid\_ds \*buf); // 目前我只是知道这个函数可以用来释放消息队列，当实现这一功能时，buf可以为NULL  // msqid表示消息队列标识符，为msgget函数的有效返回值  // cmd表示要对消息队列进行的操作，取值可以是：  // IPC\_STAT: 读取消息队列的msqid\_ds数据，并将其存储在buf指定的地址中。  // IPC\_SET: 设置消息队列的属性，要设置的属性需要先存储在buf中  // IPC\_EMID: 将队列从系统内核中删除  // buf指向消息队列管理结构体msqid\_ds，该结构定义我就先不写了。  // 返回值：0,执行成功 |

### 将消息送入消息队列（msgsnd）

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg);  // msqid为消息队列标识符  // msgp指向消息缓存区，通常可用一个通用结构表示消息  struct msgbuf  {  long mtype; // 消息类型，这个值要大于0,数值自定义，接收时会根据消息类型接收  char mtext[SIZE]; // 消息数据，大小自定义 // 我没有尝试过将char类型改为其他类型  };  // msgsz是要发送信息的长度，大小为sizeof(mtext)  // msgflg是控制函数行为的标志, 0表示阻塞方式，IPC\_NOWAIT表示非阻塞方式  // 执行成功返回0,执行失败返回-1 |

|  |
| --- |
| joint\_target\_cmd.msg\_st.msg\_type=1; //定义消息的类型,与接收类型一致  strcpy ( joint\_target\_cmd.msg\_st.text, pval );  if(msgsnd(joint\_target\_cmd.msg\_id,(void \*)&joint\_target\_cmd.msg\_st,strlen(pval),IPC\_NOWAIT)== -1){  fprintf ( stderr, "msgsnd failed\r\n" );  }; |

### 从消息队列中读取一条新消息（msgrcv）

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  int msgrcv(int msqid, const void \*msgp, size\_t msgsz, long msgtyp, int msgflg);  // msqid为消息队列标识符  // msgp指向消息缓存区，通常可用一个通用结构表示消息  // msgsz大小为sizeof(mtext)  // msgtyp表示从消息队列内读取的消息形态，0表示消息队列中所有消息都会被读取。  // msgflg是控制函数行为的标志, 0表示阻塞方式，IPC\_NOWAIT表示非阻塞方式  // 执行成功返回0,执行失败返回-1 |

### 生成键值函数（ftok）

|  |
| --- |
| // 如2.1中所说，生成一个消息队列需要一个键值，这个键值可以自定义，也可以调用ftok函数来生成  key\_t ftok(char \*fname, int id);  // fname是指定的文件名，这个文件必须是存在且可以访问，并且一旦创建后就不能在删除（即使删除了后重新建一个同名文件也不行)  // id是子序号，范围0-255,大小自定义  // 执行成功返回key\_t值，否则返回-1。这个函数我没有写示例代码。 |

|  |
| --- |
|  |

|  |
| --- |
|  |

### 自己遇到的问题

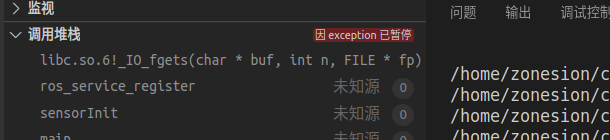
#### 1、

|  |
| --- |
| "/home/zonesion/catkin\_ws/src/aiarm/tmp",1 //这里文件夹必须存在，有t属性并且上级目录也要有t属性  printf("msg->msg\_id:%d\r\n",msg->msg\_id);  printf("msg->key1:%d\r\n",msg->key);  msg->key=ftok( msg->pathname, msg->proj);  printf("msg->key2:%d\r\n",msg->key); //16921086  if (msg->key == -1) {  fprintf ( stderr, "ftok failed width erro: %d\r\n", errno );  return -1;  } |
|  |
|  |
|  |

打印信息，有时候会创建错误

|  |
| --- |
| msg->msg\_id:0  msg->key1:0  msg->key2:16912258  msg->msg\_id:4  <<< {A0=,A1=} |

#### 2、fgets使用这个函数时出现错误



|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |