Lab1

0. 封面

- **实验题目**：实验1 小车弓字形移动和旋转
- **院系**：软件学院
- **设计者**：

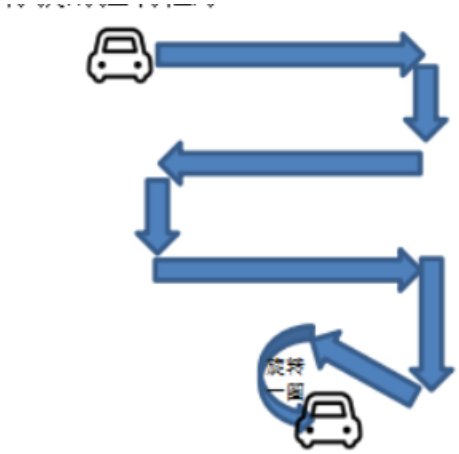
学号	姓名
202100300063	李彦浩
202100300340	黄幸兒
202100300078	李世会
202100161049	徐芃恺

- **指导教师**：李新
- **实验时间**：2023-10-24(18:00-20:00)

1. 实验内容

1.1. 实验目标

阅读小车前后左右移动、旋转等功能相关开发文档，开发一个能够沿着弓字形路线行驶的控制程序。



1.2. 实验要求

1. 完成小车移动、旋转控制函数的学习;
2. 每组同学开发一个沿着弓字形路线行驶和最后旋转的程序，让小车在1分钟内完成所有移动和旋转操作;
3. 程序开始后，不允许人为移动小车，验证程序的正确性和是否满足时间要求。

2. 需求分析

功能性需求分析

1. 需要实现小车根据在路上按照记号笔标记的路径进行移动，需要小车进行准确的移动，并在路线的拐角处进行九十度旋转。
2. 需要实现在路径的最后进行225度的转角然后走出一条斜的线路，最后走到终点最后在进行360度旋转一周最后停下。

非功能性需求分析

1. 代码的复用性高，仅仅需要修改代码中的参数，无需其他改动就可以使小车走出任意的路径。
2. 代码可读性高，代码的功能都用函数表示，模块分明，易于修改。

3. 实验设计

3.1. 程序与硬件的交互：小车是怎么动起来的？

首先先用WinSCP把树莓派板上的源码抓下来。

根据官方提供的教程，进入src→armpi_pro_demo，里面有关于小车前后左右、斜向、原地旋转的样例代码。

随便进一个看一下，这里以car_forward_demo.py为例。

先不管跟配置相关的代码，只看控制逻辑。

```
while start:
        # 发布底盘控制消息,线速度60，方向角90，偏航角速度0(小于0，为顺时针方向)
        set_velocity.publish(60,90,0) # 向前移动
        rospy.sleep(1)
```

这里有3个关注点：

3.1.1. 底盘控制参数

注意到:

```
set_velocity.publish(60,90,0) # 向前移动
```

这里面60，90，0分别代表三个物理量：

实参	物理量	物理意义
60	线速度	车在当前前进方向上的前进速度
90	方向角	可以理解为初始方向
0	偏航角速度	车的方向的改变速度

在当前平台上，偏航角速度<0，顺时针转；否则，逆时针转。

3.1.2. 发布-订阅模式(publisher-subscriber)/观察者模式(observer)

注意到:

```
set_velocity.publish(60,90,0) # 向前移动
```

其中publish的源码:

```
def publish(self, event, *args, **kwargs):
        """
        Publish a event and return a list of values returned by its
        subscribers.

        :param event: The event to publish.
        :param args: The positional arguments to pass to the event's
                     subscribers.
        :param kwargs: The keyword arguments to pass to the event's
                       subscribers.
        """
        result = []
        for subscriber in self.get_subscribers(event):
            try:
                value = subscriber(event, *args, **kwargs)
            except Exception:
                logger.exception('Exception during event publication')
                value = None
            result.append(value)
        logger.debug('publish %s: args = %s, kwargs = %s, result = %s',
```

```
                    event, args, kwargs, result)
        return result
```

看不懂代码没事，有注释：发布一个事件，返回订阅该发布者返回的值。

比如这里的set_velocity.publish，可以理解为地盘控制车的活动的硬件发布了一个事件/命令：命令车以线速度60，初始方向90，偏航角速度0（也就是向前直走）的参数运动。硬件在接收到这一命令后，就将接收到的参数设置上，这样车将来一段时间关于速度的参数就是这些了。

3.1.3. 主控程序睡眠

注意到:

```
rospy.sleep(1)
```

其中rospy意思就是rospberry的python程序，也就是主控程序，做小车行为控制的。

那么这个线程睡眠一秒是想干什么？

我们不妨脑补一下：假设现在桌上有杯水，我们想喝点水。那么大脑很快反应出三个步骤：

1. 伸出空闲的手
2. 拿起桌上的水
3. 喝

那么大脑是怎么和我们的肢体交互的呢？

1. 大脑发出命令，“伸出手”
2. 大脑停止发出命令
3. 伸出手
4. 大脑发出命令，“拿起水”
5. 大脑停止发出命令
6. 拿起水
7. 大脑发出命令，“喝”
8. 大脑停止发出命令
9. 喝

这里尤其要强调的就是，“大脑停止发出命令”这一步。不妨设想一下没有这一步会怎么样。

1. 大脑发出命令，”伸出手“
2. 大脑发出命令，“拿起水”
3. 错误，还没有伸出手，怎么执行“拿起水”？

换句话说，大脑停止发出命令是为了让大脑和肢体“同步”的。如果你一直不断地发出命令，但上一条命令又没有被执行（完），那么当前发出的命令就会因为缺少前序操作而失败！

这里，rospy代表大脑，车上的各种硬件代表肢体。所以，rospy.spleep的作用，就是为了让硬件有一段时间，执行2.1.2中rospy发布的指令的。

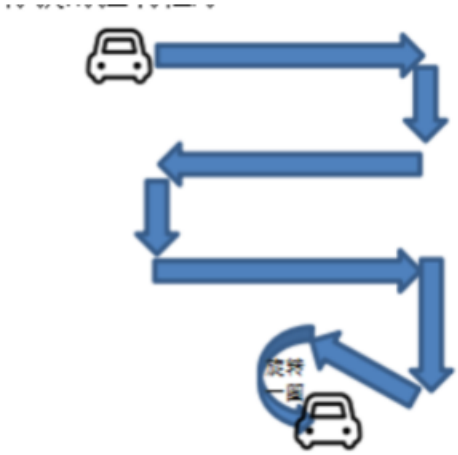
回到这段代码：

```
set_velocity.publish(60,90,0) # 向前移动
rospy.sleep(1)
```

含义为，设置这三个参数到硬件中，然后给硬件（车）1s的时间来执行。

3.2. 逻辑控制：怎么让小车以预想的方式运动？

先确定“预想的方式”是什么方式。



车从起点出发：

1. 向前走
2. 右转
3. 向前走
4. 右转
5. 向前走
6. 左转
7. 向前走
8. 左转
9. 向前走
10. 右转
11. 向前走
12. 转一圈

这个行为模式是定死的。所以我们只要写个驱动表，每一步查表获取当前行为的参数（因为每个不同行为的参数其实就那三个，线速度，方向角，偏航角速度），然后执行以下就ok了。

3.2.1. 主程睡眠与系统时钟

问题：怎么确定当前执行到上面那12步中的哪一步了？

回答：加入系统时钟

我们可以把时间想成一个偏移量，从第一步（或者说从起点）开始，就是0个单位时间。之后每进行一次原子操作，系统时钟就加1个单位时间。每一步可能包含多个原子操作。比如我们可以把向前走x秒拆解成向前走1秒，走x次。（显然x是个自然数）

根据上图中每条线段路径的长度（目测），我决定将不同动作的参数设置为：

动作	时间
直走	6s
转向	2s

PS：转向时间固定，每次转的角度不同，则偏航角速度不同。

此外还有一个小trick，我们不用维护一个前缀和作为系统时钟，而可以维护一个差分偏移量作为时钟偏移量。简单来说，到第3步时，系统时钟为10个单位时间，但在程序中不需要维护这个10，而是从0开始计数，直到增长到第三步结束的时间（第18个单位时间），本质上是维护了差分数组。

根据上表设定的时间，这个差分数组以及其对应的参数数组应该是：

```
    # the corresponding offset ticks for the chassis control
    # forward->turn right->forward->turn right->forward->turn left->forward->turn left->forward->turn right->forward
    ticks = [8.0, 2.0, 8.0, 2.0, 8.0, 2.0, 8.0, 2.0, 8.0, 2.0, 8.0, 2.0, 5.656, 2.0]
    args = [
        [60.0, 90.0, 0.0],
        [0.0, 90.0, -0.55],
        [60.0, 90.0, 0.0],
        [0.0, 90.0, -0.55],
        [60.0, 90.0, 0.0],
        [0.0, 90.0, 0.55],
        [60.0, 90.0, 0.0],
        [0.0, 90.0, 0.55],
        [60.0, 90.0, 0.0],
        [0.0, 90.0, -0.55],
        [60.0, 90.0, 0.0],
        [0.0, 90.0, -0.83],
        [60.0, 90.0, 0.0],
        [0.0, 90.0, -1.375]
    ]
```

随后维护一个时钟偏移量，代表当前一步已经执行的单位时间，并维护一个索引指针，指向当前步骤对应的时间片。

```
offset_ticks = 0
cur_i = 0
```

3.2.2. 控制逻辑

有了以上准备，就可以开始编写控制逻辑了。

流程如下：

1. **当索引指针仍处于数组边界内时，代表预想的行为模式并没有结束，应该继续执行；否则，结束行为**
2. **获取当前时间片截止时间**
3. **获取当前一步行为参数**
4. **根据这些参数做出行为（一个原子操作）**
5. **当前时钟偏移量自增一个单位时间**
6. **判断当前时间片是否结束**
    a. **结束，索引指针指向下一步操作，时钟偏移量清零**
    b. **没结束，索引指针保持不动**
7. **“大脑停止发出命令”，主程休眠一个单位时间，让硬件执行刚才发布的指令**

代码如下：

```
# without interrupt from the keyboard,do the routes
    route_steps_len = len(ticks)
    while start and cur_i < route_steps_len:
        # get tick
        fin_tick = ticks[cur_i]
        # get args
        arg = args[cur_i]
        # act
        activity(args[0], args[1], args[2])
        # inc offset_ticks
        offset_ticks += 1
        # next route?
        if offset_ticks >= fin_tick:
            cur_i += 1
            offset_ticks = 0
        # actual tick inc
        rospy.sleep(1)
```

3.3. 杂项

3.3.1. 中断与回调

用户应该拥有随时终止程序的能力。所以我们要搞一个程序是否结束的标识。

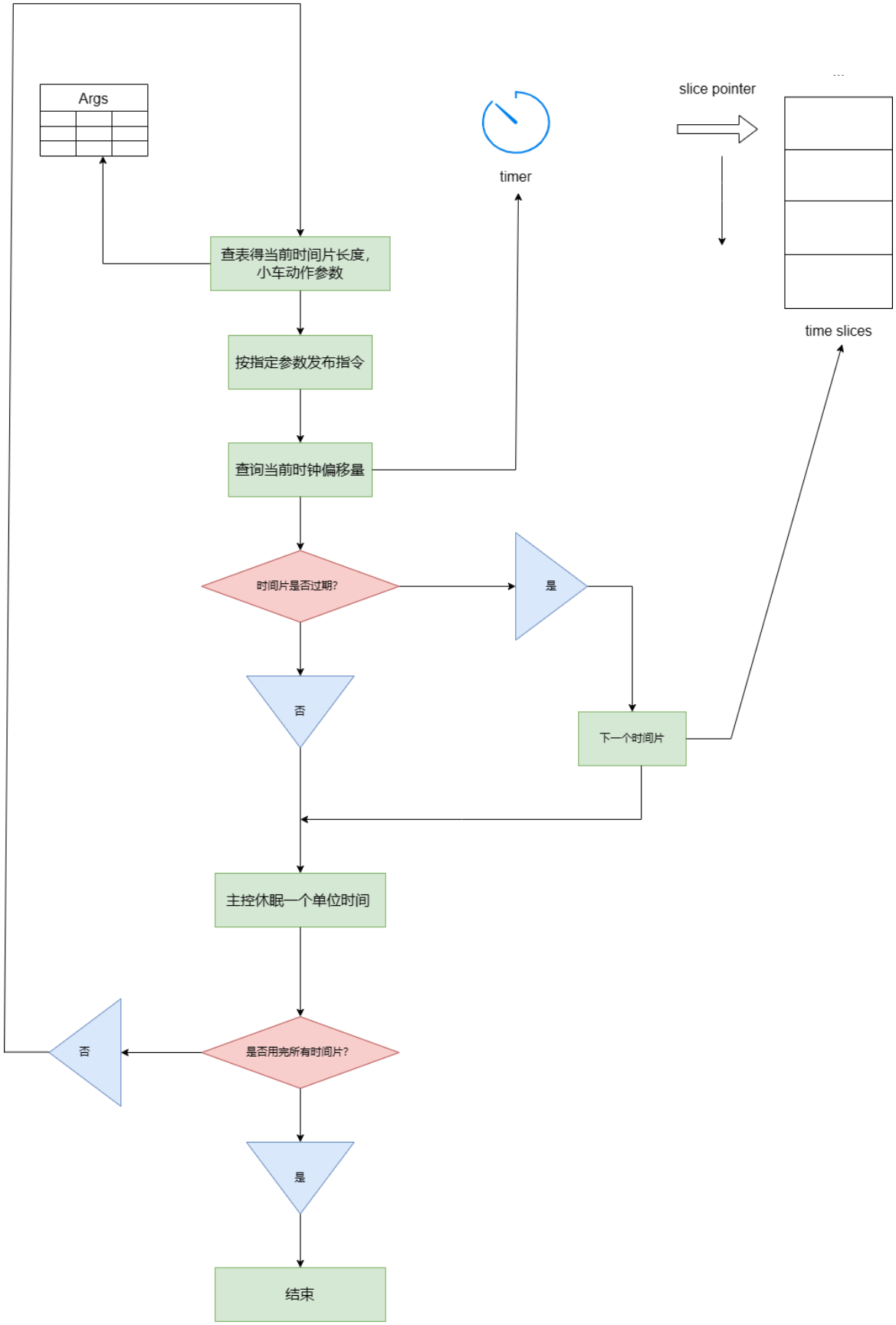
```
start = True

def stop():
    # similar to external in C
    global start

    start = False
    print('Shutting down...')
    set_velocity.publish(0, 0, 0)  # stop
```

当用户想要暂停时，将这个标识置为false，并停止所有电机转动。

随后我们要为这个终止函数注册回调。直到键盘中断产生，调用该回调函数。

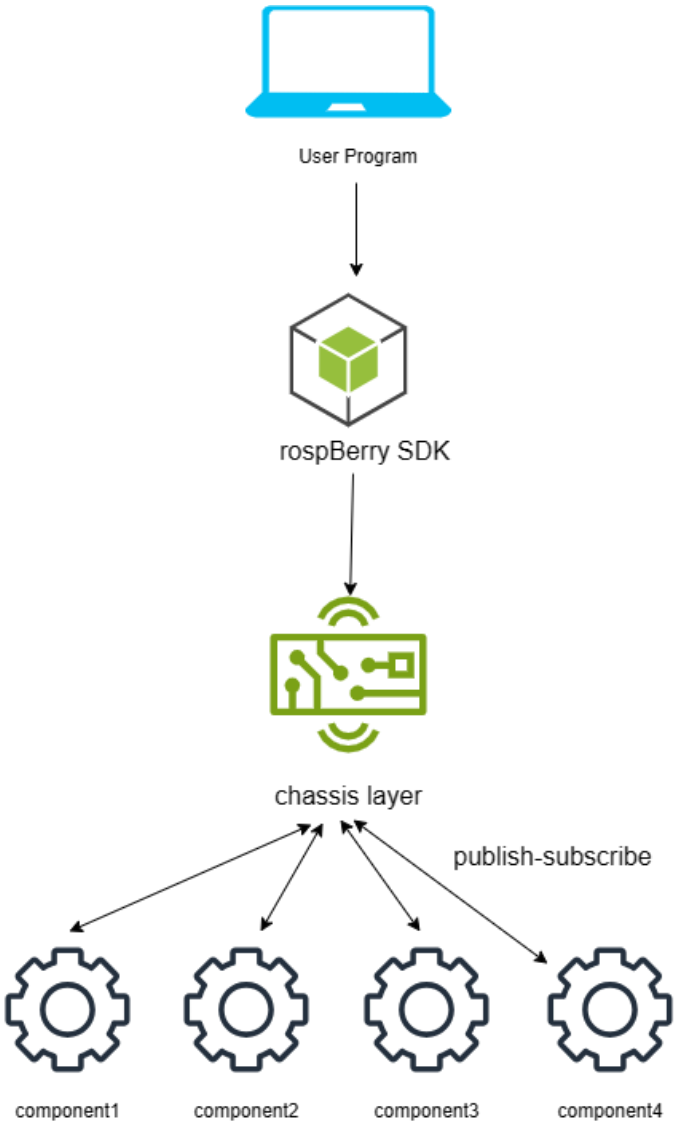
```
rospy.on_shutdown(stop)  # stop callback function
```

3.4 流程图


Args
timer
slice pointer
...
time slices
查表得当前时间片长度，
小车动作参数
按指定参数发布指令
查询当前时钟偏移量
时间片是否过期?
是
否
下一个时间片
主控休眠一个单位时间
是否用完所有时间片?
否
是
结束

这个流程上面已经详述过了，不再赘述。

3.5 架构图



rospBerry的工具包与小车底盘进行交互，通过观察者模式，将上层User Program的命令发布给底层的硬件组件。

4. 完整工程

就一个单文件，代码如下；

```
#!/usr/bin/python3
# coding=utf8
import sys
import rospy
```

```
from chassis_control.msg import *

if sys.version_info.major == 2:
    print('Please run this program with python3!')
    sys.exit(0)

print('Bow Shaped Move Lab')

start = True
offset_ticks = 0
cur_i = 0


# before shut down
def stop():
    # similar to external in C
    global start

    start = False
    print('Shutting down...')
    set_velocity.publish(0, 0, 0)  # stop


# def activity(linV, directA, yawR):
#     # publish a chassis control msg, with linear velocity 60, direction angle 90, yaw rate 0(<0, clockwise)
#     set_velocity.publish(linV, directA, yawR)


if __name__ == '__main__':
    # init node
    rospy.init_node('bow_shaped_move', log_level=rospy.DEBUG)
    rospy.on_shutdown(stop)  # stop callback function
    # Mc Wheel chassis control
    set_velocity = rospy.Publisher('/chassis_control/set_velocity', SetVelocity, queue_size=1)

    # the corresponding offset ticks for the chassis control
    # forward->turn right->forward->turn right->forward->turn left->forward->turn left->forward->turn right->forward
    ticks = [8.0, 1.3, 8.0, 1.3, 8.0, 1.3, 8.0, 1.3, 8.0, 1.3, 8.0, 1.3, 5.656, 1.3]
    args = [
        [60.0, 90.0, 0.0],
        [0.0, 90.0, -0.55],
        [60.0, 90.0, 0.0],
        [0.0, 90.0, -0.55],
        [60.0, 90.0, 0.0],
        [0.0, 90.0, 0.55],
        [60.0, 90.0, 0.0],
        [0.0, 90.0, 0.55],
        [60.0, 90.0, 0.0],
        [0.0, 90.0, -0.55],
        [60.0, 90.0, 0.0],
        [0.0, 90.0, -0.83],
        [60.0, 90.0, 0.0],
        [0.0, 90.0, -1.375]
    ]

    # without interrupt from the keyboard,do the routes
    route_steps_len = len(ticks)
    while start and cur_i < route_steps_len:
        # get tick
        fin_tick = ticks[cur_i]
        # get args
        arg = args[cur_i]
        # act
        set_velocity.publish(arg[0], arg[1], arg[2])
        # inc offset_ticks
```

```
        offset_ticks += 1
        # next route?
        if offset_ticks >= fin_tick:
            cur_i += 1
            offset_ticks = 0
        # actual tick inc
        rospy.sleep(1)

    set_velocity.publish(0, 0, 0)  # stop
    print('Shut down')
```