《 智能系统 》实验报告

实验题目	十字路口红绿灯智能控制完整系统实现与测试

一、实验目的

为实现十字路口红绿灯智能控制，本次实验的目的是：

（1） 了解上位机与下位机实时通讯原理。

（2） 设计与实现上位机与下位机的实时通讯。

（3） 实现十字路口红绿灯智能控制的完整功能。

二、实验项目内容

1、定义上位机与下位机的通讯协议。

2、设计实现上位机的串口通讯程序。

3、设计实现下位机的串口通讯程序。

4、包含上下位机的完整系统的实现与测试。

三、实验过程或算法（代码）

1、定义上位机与下位机的通讯协议

本次实验共使用1个红外避障模块与2个交通灯模块，它们的引脚连接表如下：

交通灯模块（南北交通灯）：

交通灯模块引脚	Arduino UNO R3引脚
GND	GND
R	12号数字引脚
Y	不接入
G	13号数字引脚

交通灯模块（东西交通灯）：

交通灯模块引脚	Arduino UNO R3引脚
GND	GND
R	7号数字引脚
Y	不接入

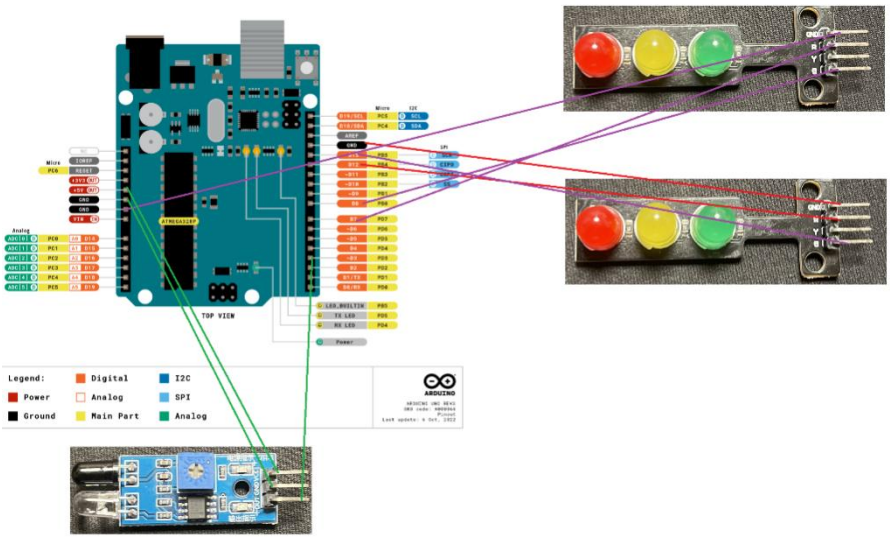
报告创建时间：

G	8号数字引脚

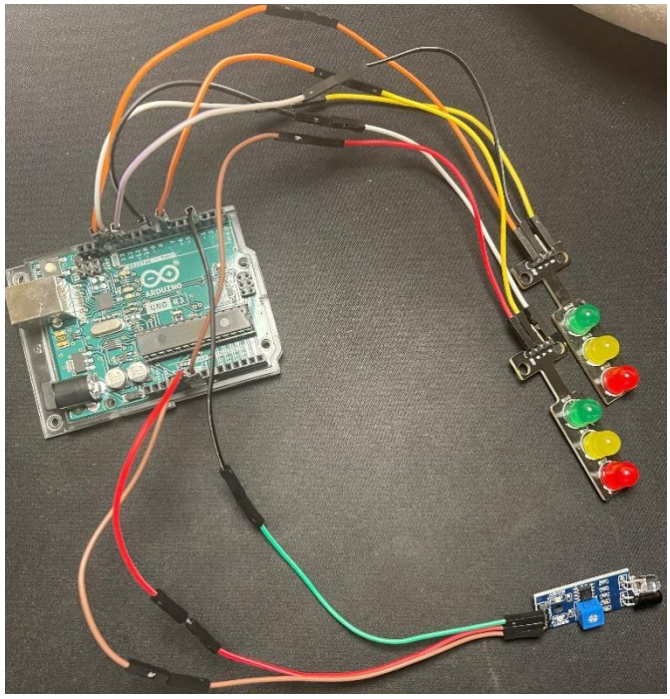
红外避障模块：

红外避障模块引脚	Arduino UNO R3引脚
VCC	5V供电引脚
GND	GND
OUT	3号数字引脚

结合官方引脚定义说明：



实物连接图：



需求分析：

在本实验中，需要使用下位机采集传感器信息，传输给上位机，上位机需要利用采集得到的信息进行推理，并将推理结果传给下位机，由下位机再传给交通灯模块，实现十字路口红绿灯智能控制。

下位机传给上位机的信息：只有红外避障模块的检测信息，通过0、1（ASCII编码）串的形式传输给上位机。

上位机传给下位机的信息：十字路口的红绿灯组包含南北方向的红绿灯和东西方向的红绿灯，这两个方向的红绿灯开关是互斥的，故红绿灯的开关情况只有2种，1种是南北红绿灯是绿灯，东西红绿灯是红灯，还有1种是南北红绿灯是红灯，东西红绿灯是绿灯。故也可只用0、1（ASCII编码）串的形式通知下位机当前时刻红绿灯组的状态——0代表南北红绿灯是红灯、东西红绿灯是绿灯，1代表南北红绿灯是绿灯、东西红绿灯是红灯。

2、设计实现上位机的串口通讯程序

上位机读取下位机的红外避障模块检测结果：

```python
class CarReader:
    def __init__(self, serial) -> None:
        self.curr_cnt = 0
        self.serial = serial

    def read_serial_data(self):
        while True:
            data = self.serial.read().decode()
            if data == '1':
                self.curr_cnt += 1

    def start(self):
        read_thread = threading.Thread(target=self.read_serial_data)
        read_thread.start()

    def harvest(self) -> int:
        cnt = self.curr_cnt
        self.curr_cnt = 0
        return cnt
```

定义了一个 CarReader 类，用于读取 0、1 串。

其拥有两个成员变量，其中 curr_cnt 代表了当前累计检测到障碍的次数——用于推理引擎中南北、东西方向通过的车辆计数。

serial 则代表了一个串口连接，CarReader 使用该串口连接读取下位机传来的 0、1 串，具体的逻辑在 read_serial_data 函数中体现。

start 函数用于启动读取串口数据的后台线程，因为 pyknow 知识引擎在推理的过程中是单线程的，如果将串口读取放置于推理线程中则会阻塞推理，故需将其放于单独的后台线程。

harvest 函数是提供给 pyknow 知识引擎的结果获取接口，因为知识引擎会在一段时间等待推理结束后才空闲能来获取结果，故需设计一个函数将累计的结果返回给知识引擎，并清空累计结果。

上位机控制下位机的红绿灯组信号：

```python
class LEDController:
    def __init__(self, serial) -> None:
        self.state = b'0' # 0—南北为红灯，东西为绿灯；1—南北为绿灯，东西为红灯
        self.serial = serial

    def write_serial_data(self):
        while True:
            self.serial.write(self.state)

    def start(self):
        write_thread = threading.Thread(target=self.write_serial_data)
        write_thread.start()
```

定义了一个 LEDController 类，用于向串口写入 0、1 串，指导下位机控制红绿灯组。

其拥有两个成员变量，其中 state 存储当前上位机的推理结果，因为红绿灯的变换需要时间，红绿灯组当前的状态会持续数秒，而在这期间仍需要指导下位机控制红绿灯组，故需要存储红绿灯组的状态。

serial 为一个串口连接，LEDController 向该串口连接写入 0、1 串指导下位机控制红绿灯组。

write_serial_data 函数描述了 LEDController 向串口写入的逻辑：循环写入当前的红绿灯组状态。

start 函数用于启动写入串口数据的后台线程，原因与 CarReader 中相同，写入串口数据会阻塞推理线程，需要在单独的后台线程中进行。

3、设计实现下位机的串口通讯程序

接口定义：

```
int ns_led_green_pin = 13;
int ns_led_red_pin = 12;
int we_led_green_pin = 8;
int we_led_red_pin = 7;
int detect_pin = 3;

void setup() {
  // put your setup code here, to run once:
  pinMode(ns_led_green_pin, OUTPUT);
  pinMode(ns_led_red_pin, OUTPUT);
  pinMode(we_led_green_pin, OUTPUT);
  pinMode(we_led_red_pin, OUTPUT);
  pinMode(detect_pin, INPUT);
  Serial.begin(9600);
}
```

在下位机代码的开头定义了两个交通灯模块和红外避障模块的引脚常量，并在 setup 函数中定义了它们的输入输出属性——交通灯模块的红绿灯引脚为输出，红外避障模块的检测结果为输入。

串口通讯与解析：

```
void loop() {
  // put your main code here, to run repeatedly:
  int detect_result = digitalRead(detect_pin);
  if (detect_result == HIGH)
    Serial.print("1");
  else
    Serial.print("0");

  char read_result = Serial.read();
  if (read_result == '1') // 1表示南北为绿灯，东西为红灯
  {
    digitalWrite(ns_led_green_pin, HIGH);
    digitalWrite(ns_led_red_pin, LOW);
    digitalWrite(we_led_green_pin, LOW);
    digitalWrite(we_led_red_pin, HIGH);
  }
  else
  {
    digitalWrite(ns_led_green_pin, LOW);
    digitalWrite(ns_led_red_pin, HIGH);
    digitalWrite(we_led_green_pin, HIGH);
    digitalWrite(we_led_red_pin, LOW);
  }
}
```

在每个循环中先读取红外避障模块的检测结果 detect_result，并将其发送给上位机。

随后读取上位机发送来的红绿灯组控制指令，若为 1，则代表南北红绿灯为绿灯，东西红绿灯为红灯。若为 0，则代表南北红绿灯为红灯，东西红绿灯为绿灯。按照指令将对应的输出引脚置为对应的高低电平。

4、包含上下位机的完整系统的实现与测试

本章节主要描述为实现完整系统，而对实验2中知识引擎所进行的修改。

知识引擎初始化的修改：

```
class TrafficLights(KnowledgeEngine):
    @DefFacts()
    def _initial_action(self):
        # 初始化串口连接
        self.serial = serial.Serial('COM3', 9600)  # 串口名称和波特率
        self.car_reader = CarReader(self.serial)
        self.led_controller = LEDController(self.serial)
        self.car_reader.start()
        self.led_controller.start()
        yield Fact(Ticks = 0)
        yield Fact(Second = 0)
        yield Fact(ASecond = False)
        yield Fact(SwitchTime = 5)
        yield Fact(Period = 10)
        yield Fact(NSLight = 'GREEN')
        yield Fact(WELight = 'RED')
        yield Fact(NSCars = 0)
        yield Fact(WECars = 0)
```

在知识引擎的初始化函数中，添加了串口连接的建立，后台串口读取和写入线程的启动。

计数函数的修改：

```
# 设置计数规则
@Rule(AS.oldFact << Fact(Ticks = MATCH.times))
def ticks(self,times,oldFact):
    self.retract(oldFact) # 收回事实
    self.declare(Fact(Ticks = times + 1))
    time.sleep(0.1)
    # if times == exeTimes * 10:
    #     print('bye!')
    #     self.halt()
    # else:
    if times%10 == 0:
        self.declare(Fact(ASecond = True))
    # self.comm()
    curr_cnt = self.car_reader.harvest()
    self.declare(Fact(ReadCarCount = curr_cnt))
```

移除了计数函数中的计数上限，让程序一直执行，便于调试与测试，并修改其使用随机数增加南北、东西车辆的逻辑，替换为读取串口数据来增加南北、东西通过车辆数。

红绿灯组状态展示函数的修改：

```
# 展示灯当前情况的规则
@Rule(
    Fact(NSLight = MATCH.NScolor),
    Fact(WELight = MATCH.WEcolor),
    salience = 2
  )
def show(self,NScolor,WEcolor):
    #print('\n NS:WE={}:{} \n'.format(NScolor,WEcolor))
    if NScolor == 'RED':
        print('-X-  -V-')
    else:
        print('-V-  -X-')
    if NScolor == 'RED':
        self.led_controller.state = b'0'
    else:
        self.led_controller.state = b'1'
```

在原来展示红绿灯组状态的函数中插入对串口写入数据的更新逻辑，每当红绿灯组状态发生改变，就会同步改变LEDController中的写入数据。

南北车辆计数逻辑的修改：

```
# 南北车辆计数规则
@Rule(
    AS.fact1 << Fact(ReadCarCount = MATCH.count),
    AS.fact2 << Fact(NSCars = MATCH.cars),
    AS.fact3 << Fact(NSLight = "GREEN")
)
def countNS(self,fact1,fact2,cars,count):
    self.retract(fact1)
    self.retract(fact2)
    self.declare(Fact(NSCars = cars + count ))
```

修改南北车辆计数增加逻辑，不再是检测到有车辆通过就触发（因为只有一个红外避障模块，我们只能假设当前通过路口的车辆全为绿灯路口的车辆），而是需要判断当前路口为绿灯才触发。且每次触发不再只+1，而是增加累计通过数（知识引擎每隔一段时间去取回读取结果，故有累计）。

东西车辆计数逻辑的修改：

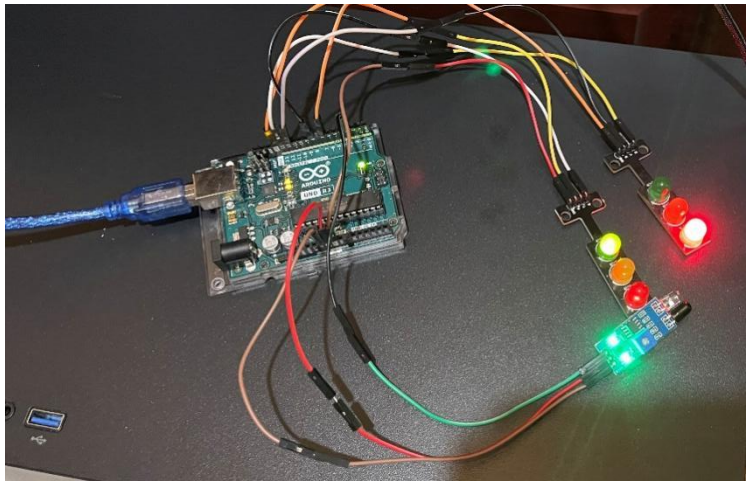
```
# 东西车辆计数规则
@Rule(
    AS.fact1 << Fact(ReadCarCount = MATCH.count),
    AS.fact2 << Fact(WECars = MATCH.cars),
    AS.fact3 << Fact(WELight = "GREEN")
)
def countWE(self,fact1,fact2,cars,count):
    self.retract(fact1)
    self.retract(fact2)
    self.declare(Fact(WECars = cars + count ))
```

与南北车辆计数逻辑的修改部分相似。

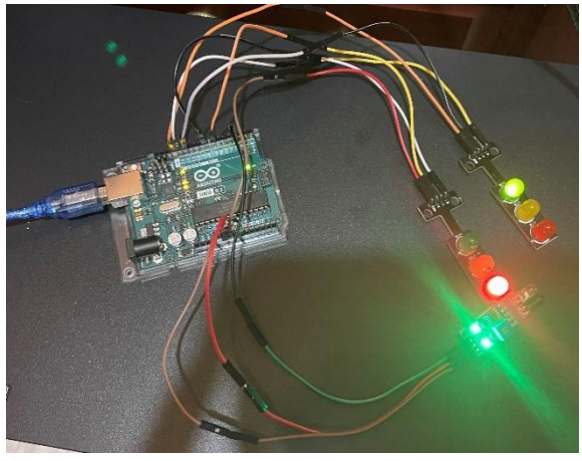
四、实验结果及分析

红绿灯组的两种状态：

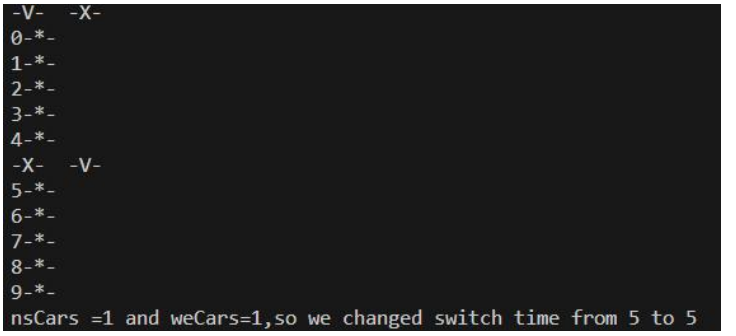
南北为红灯，东西为绿灯：



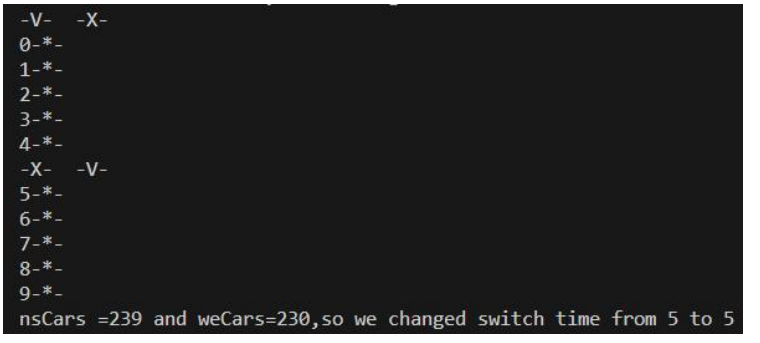
南北为绿灯，东西为红灯：



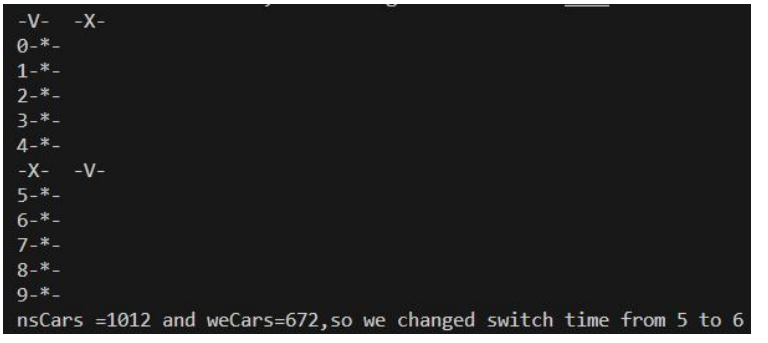
南北路口与东西路口都没有车辆通过：

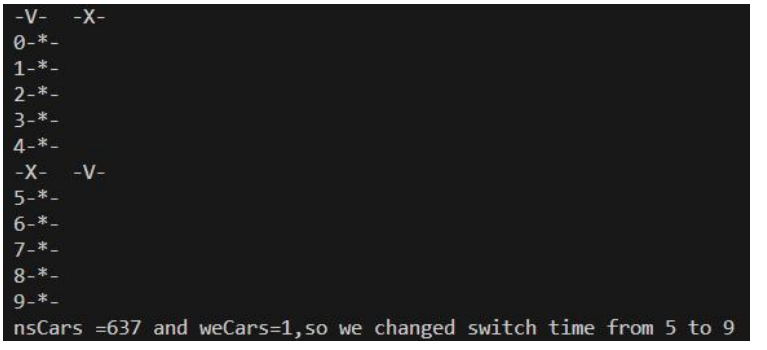


南北路口与东西路口通过车辆数相似：

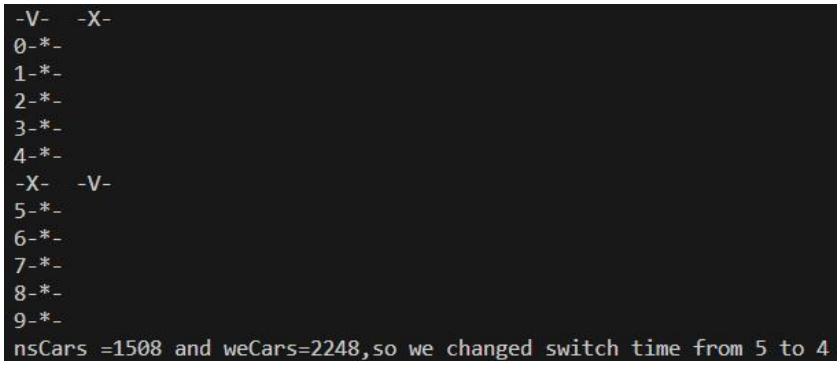


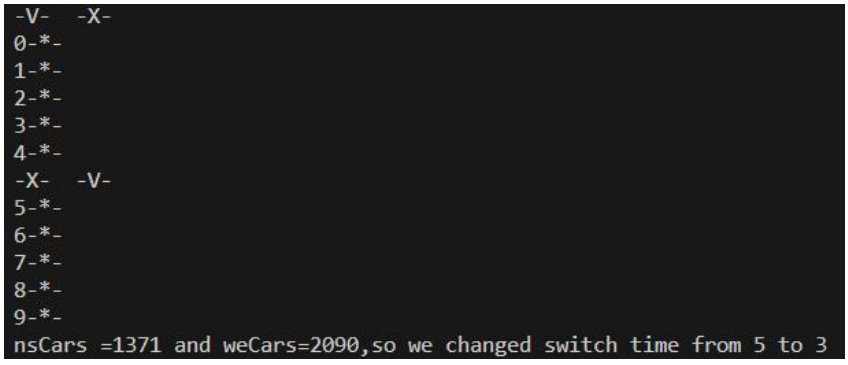
南北路口大于东西路口通过车辆数：





南北路口小于东西路口通过车辆数：





评价总结：

该十字路口红绿灯智能控制系统在实验环境中表现优秀，能够根据南北和东西路口的车流量动态调整红绿灯时长。它为解决实际交通场景中因静态红绿灯时间导致的交通拥堵问题提供了有效的解决方案。

系统的核心在于，它可以通过分析一轮红绿灯周期中的车辆数来确定下一轮中两个方向红绿灯的时长。通过车辆数多的方向将享有更长的绿灯时间，而车辆数少的方向则会有更长的红灯时间。这种动态调整机制能够更好地优化交通流量，避免交通堵塞。

然而，系统在实验和实际场景之间存在一定的差距。例如，实验中设置的每轮红绿灯时间为 10 秒，而在实际情况中这种情况较少。这使得系统的红绿灯时间划分粒度较粗，可能无法细致地反映出两个路口车辆数的差异。

此外，该系统在检测车辆是否通过路口时未实现“防抖”机制，因此在使用红外避障检测模块遮挡时，会在短时间内发送大量的信号。虽然引入“防抖”机制可以解决这个问题，但这也会增加系统的实现复杂度，并给模拟操作带来额外的负担。

总的来说，这是一个具有潜力的系统，能够提供有效的交通管理解决方案。然而，为了使其更好地适应实际情况，还需要进一步

改进和优化，例如改进红绿灯时间划分的粒度，以及引入“防抖”机制来更准确地检测车辆流量。
五、完成时间 （1）实验时间：2023.5.28，2023.6.4 （2）检查时间：2023.6.4 （3）2023 年 6 月 18 日 23:59 之前提交实验报告