IEECS-Car 软件设计总结与指南

1、概述

本篇主要讲述有关IEECS-Car硬件平台的编程设计与上位机与下位机的通讯问题，旨在提出一种切实可行的实现方式，为日后更新换代奠基。

依据本篇总结的技术实现的软件可以满足IEECS实验中所有的要求。

本次软件设计主要涉及到硬件车的嵌入式C语言开发与上位机Soar的Python开发，其中主要工作是让硬件车能稳定可靠的受Soar控制，两者通讯应该具备较强的鲁棒性。

本次软件设计相较于Pioneer-3，有所改动，并不完全相同。

2、功能要求

1. 硬件车能够采集8路测距传感器
2. 硬件车能够接受4路外部ADC并输出1路DAC
3. 硬件车具备PID调速控制，车轮可以正反转
4. 硬件车具备与上位机Soar进行无线通讯的能力
5. 上位机Soar与硬件车具备解析通讯格式的能力
6. 硬件车具备定位的能力
7. 硬件车具备上位机Soar与之连接或断开时的提示音

3、涉及的技术

1. PID控制
2. 无线通讯，通讯协议的制定，多车同时通讯时互相干扰的问题
3. ADC采样
4. PWM代替DAC输出
5. 定位算法设计
6. 蜂鸣器的控制

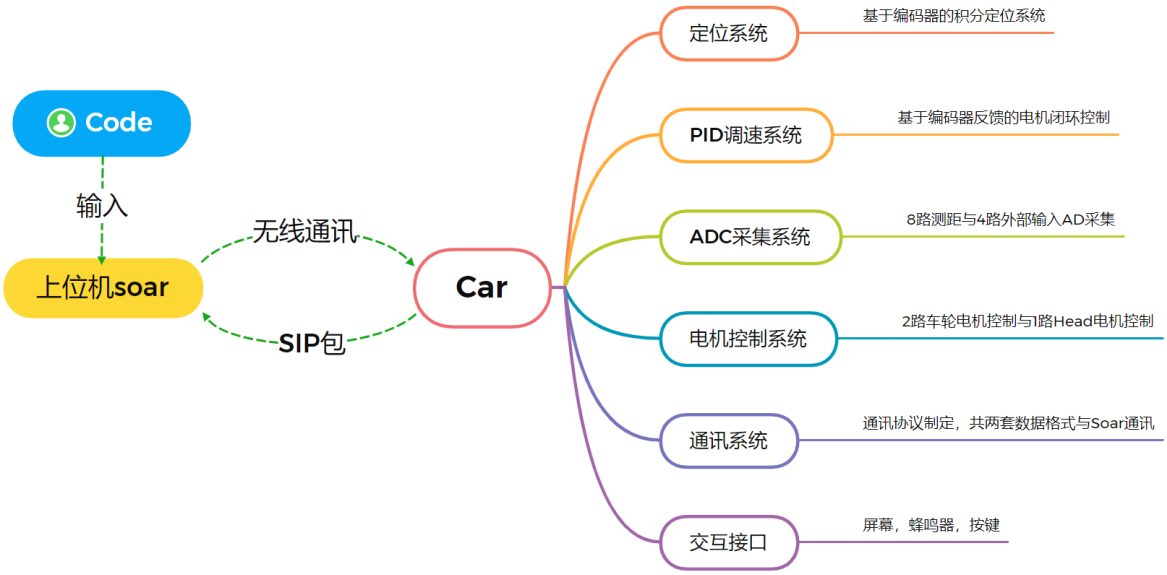
4、可能遇到的问题

1. 定位算法的精度
2. 无线通讯的鲁棒性
3. PID控制调参
4. 多车通讯的干扰问题

5、系统框架

图一展示了整套软件系统的框架图与涉及的模块划分。

依照MIT6.01课程配套设计的Soar，使用者编写代码后从“Brain”内导入，按照状态机设计的理念，连续的系统可以被划分为连续的Step，每一个Step内使用者需要在指定函数中返回其状态(State)。在大部分的Design Lab中，最终需要落实到控制车轮的转速，在电路实验中额外需要控制输出电压。省略其中计算步骤，使用者最终对soar的输入为对Car角速度，线速度，输出电压三个规划值。Soar向Car传输的内容即为这三个规划值，Car向Soar传输的内容主要为采集到的8个距离信息，车当前的位置等信息。采用有线连接或无线连接对传输的内容无本质的区别，传输内容的格式根据Intel，Pioneer-3的数据手册中的SIP包明确制定。SIP包并非一种特定的数据格式，此处仅为一种代号，本质只是一种特殊的通讯协议。在车内部，首先要完成PID调速系统与对两个车轮电机的控制，其反馈通过编码器的值反映实际速度完成，最终要使控制速度与使用者设定速度相互匹配。此外，在部分实验中使用到了定位，为尽可能减少传感器数量，此处仅使用编码器实现定位系统。8路测距与4路外部输入ADC采集由单片机直接完成，通讯系统的实现依赖于无线串口与通讯协议的指定。还需要输出1路对Head电机的控制，此处与原始设计有差别，原方案由主控输出DA信号，考虑到实际应用只是驱动Head电机正反转，相比原方案对电机一侧接5V，一侧接DA信号的方案，直接如同车轮电机一样进行控制显然更加合理。蜂鸣器用于提示与soar连接成功与断开。屏幕，按键预留便于调试，方案成熟后可以去除。



图一

6、实现细则

本章开始详细讲解图一中涉及到的所有细节问题。下文将首先对上位机Soar的代码进行梳理，再讲解通讯协议的格式，最终讲解Car内部的程序设计。

Soar系统整体

首先从最基础的SM(State Machine)状态机类切入，对于此类的使用几乎贯穿了所有的实验。

图二展示了最基础的状态机类创建，在实验中，我们往往需要自己继承SM类后，改写其中getNextValues函数，实现各种各样的功能，其中state为我们需要当前时刻存储的某种状态，inp为来自外部的输入，这个输入可能为8组距离信息或定位信息，角度等，在每一个Brain文件中会在Step函数中定义inp，例如图三所示。

```
class MySMClass(sm.SM):
    def getNextValues(self, state, inp):
        print("{:.2f}".format(inp.sonars[0]),
              "{:.2f}".format(inp.sonars[1]),
              "{:.2f}".format(inp.sonars[2]),
              "{:.2f}".format(inp.sonars[3]),
              "{:.2f}".format(inp.sonars[4]),
              "{:.2f}".format(inp.sonars[5]),
              "{:.2f}".format(inp.sonars[6]),
              "{:.2f}".format(inp.sonars[7]))
        return (state, io.Action(fvel = 0.0, rvel = 0))

mySM = MySMClass()
```

图二

```
# this function is called 10 times per second
def step():
    inp = io.SensorInput()
    # print inp.sonars[3]
    robot.behavior.step(inp).execute()
    io.done(robot.behavior.isDone())
```

图三

有关io.SensorInput()，在MIT6.01给出的官方软件文档中已经详细介绍，如图四。完整的软件文档在此处：https://ocw.mit.edu/ans7870/6/6.01sc/documentation/index.html，后续不在展示软件文档内的内容。

Module io :: Class SensorInput

Class SensorInput

Represents one set of sensor readings from the robot, incluing sonars, odometry, and readings from the analogInputs

Instance Methods	
	`__init__(self, cheat=False)`
	`__str__(self)`

Instance Variables	
	sonars List of 8 sonar readings, in meters.
	odometry Instance of util.Pose, representing robot's pose in the global frame if `cheat = True` and the odometry frame if `cheat = False`.
	analogInputs List of 4 analog input values.

图四

有关state，在此处再次明确，所谓状态，是一种使用者希望传递到下一时刻的值，其本质利用下面这段代码将非常好理解：

```
state = 1

while(1):
    if(state % 3 == 0):
        state = state + 1
    elif(state % 3 == 1):
        state = state * 2
    else:
        state = state + 2
```

以上代码并无什么实际意义，在while循环外部定义state相当于赋予其初始状态，当状态机运行开始运行后可看做在一个while循环中一直执行，在每一次循环中，对符合其当前各种判断条件的，根据自己的需求更改state内的值，当下一次循环开始时，就相当于当前时刻的state被传递到下一时刻了，如此反复。由此可见，state内部只是起到了一个全局变量的作用。

回到SM中，最后就是要关注io.Action()中，详细介绍可以查看软件文档，此处的作用实际上将设定的线速度，角速度作为SM的输出，这个输出值会被Soar的仿真系统或实车系统捕获作为输入，如果在仿真，则直接转换为仿真车的行为，如果是实车，还需要进行数据的转换。

Soar的运行逻辑是，每时每刻不停的执行Step()函数(见图三)，在每一个Step中，接受外部传感器新的输入，根据重写的getNextValues()方法得到下一时刻的状态和输出，状态会被直接传递，输出会被仿真或实车系统捕获进一步进行处理。

我们主要需要理解的为Soar是如何控制车运行的，相关的代码文件在Soar文件夹内部的Pioneer.py，在跑实车时往往需要在这个文件夹修改串口号和波特率。

我们直接查看Pioneer类下的update方法，如图五：

```
# update sonars, odometry and stalled
def update(self, dummyparameter=0):
  if self.serialready:
    self.sendMotorCmd()
    self.sendDigitalOutCmd()
    # if we receive no packets for a few cycles, we've lost touch with the robot
    if (self.sipRecv() > 0):
      self.zero_recv_cnt = 0
    else:
      self.zero_recv_cnt += 1
      if (self.zero_recv_cnt > 20):
        self.serialready = False
        app.soar.closePioneer()
        sys.stderr.write("Robot turned off or no longer connected.\nDead reckoning is no longer valid.\n")
```

图五

update方法即为Pioneer的核心方法，我们进行代码解读，方法进来后首先判断标志位是否为真，有关这个标志位的开启，需要在Soar和实车最开始连接的三次同步信号验证通过后置1，这个标志位在open方法中，可自行阅读理解。至此，已经可以确定Pioneer与硬件车的连接依靠的为串口通讯。如果串口号打开了，进行两个看似是发送命令的函数，这两个函数分别通过串口向硬件车发送了线速度/角速度和需要设置的DA输出值。先不考虑其具体实现，接着往下，此处有一个sipRecv()的方法，这个方法内部实现了监听硬件车向Soar发送的串口数据，假如数据符合制定的通讯协议且通过校验，则数据有效，对一标志位置0，否则则判断没有接受到来自硬件车的信息，当连续二十次判断没有监听到有效的来自硬件车的数据，则判断串口已经断开连接，中止连接。

顺着update这个核心方法，我们查看如何实现串口发送速度与DA值，以及监听串口接收的数据。

下面主要以对发送线速度代码进行解读，其余同理，更详细的解读放到下一章通信格式的讲解中。

如图六，此处已对原版代码进行修改，其中sleep函数需要根据实际表现进行调整，由于电脑端运行过快，其发送速度太快会导致硬件车的串口接收产生错误，这个值需要根据实际表现调整，太慢也不行，会影响时效性。同时原始代码每次都同时发送线速度和角速度，此处为了缓解接收端的压力，通过简单的标志位实现线速度和角速度轮流发送，其中两个get函数分别获得了前文提到的状态机输出的速度，起到了捕获线速度和角速度的作用。

```
def sendMotorCmd(self):
    self.send_v_cnt = (self.send_v_cnt + 1) % 2
    if self.send_v_cnt:
        self.cmdVel(int(self.trans.get() * 1000.0))
    else:
        self.cmdRvel(int(self.rot.get() * 180.0/pi))
    sleep(0.002)
```

图六

再看图七发送线速度，我们可以看到此处待发送的数据即为data，其最后两位分别表示了线速度的高八位和第八位，一般通信协议对数据长度的单位都是8比特，超出8比特的数据需要拆分成若干个8比特的数。由于我们再程序中设置的为其实际速度(即对线速度设置为0.1即表示要求车按照0.1m/s进行运行)，其本身是浮点数且数字很小，我们如图六中乘上比例系数进行放大，在进行数据的拆分放入data中，放大数据也起到了一定的抗干扰作用，只需要在嵌入式端恢复即可。此外，通讯协议中一般约定传输的数都为无符号数，所以其正负需要有单独的标志位，即下面data中的ARGINT和ARGNINT，分别标注后面跟着的速度的正负，最前面的ArcosCmd.VEL为标志后面传输的数据位线速度，相应的，其余不同命令有不同的命令符号，在下一章详细讲解。

```
def cmdVel(self,v):
    absv = int(abs(v))
    if v >= 0:
        data = [ArcosCmd.VEL,ArgType.ARGINT,absv&0xFF,absv>>8]
    else:
        data = [ArcosCmd.VEL,ArgType.ARGNINT,absv&0xFF,absv>>8]
    self.sipSendReceive(data)
    self.sipSend(data)
```

图七

紧接着就是最终将data发送出去的过程，其余传输命令的过程相似，都是使用命令符号+数据类型+数据的格式拼接出需要传输的data，在最终有sendPacket()完成传输。如图八，需要对data添加帧头，数据长度，末尾需要添加校验和得到最终完整的符合通讯协议的格式pkt。紧接着利用Python的Serial库自带的串口发送函数发送即可，有关串口发送部分，已改正为Pyhon3的发送方式，这部分与源代码不同，源代码为在Python2环境下编写， Python2和3版本的串口发送并不兼容。

```
# Send a packet to robot
def sendPacket(self,data):
  pkt = [0xFA,0xFB,len(data)+2]
  for d in data:
    pkt.append(d)
  pkt.append(0)
  pkt.append(0)
  chk = self.calcChecksum(pkt)
  pkt[len(pkt)-2] = (chk >> 8)
  pkt[len(pkt)-1] = (chk & 0xFF)
  # s = reduce(lambda x,y: x+chr(y),pkt,"")
  s = self.Get_Bytes_Str(pkt)
  try:
    self.port.write(s)
  except:
    sys.stderr.write("Could not write to serials port.  Is robot turned on and connected?\n")
  sleep(0.008)
```

图八

如图九，类似的，串口的收首先判断等待队列中是否有数据，有的话利用图十recvPacket()判断当前数据是否是完整的，符合通讯协议格式的，可以通过校验的，如果符合，则存储到recv中。总共有两种不同的数据包，源代码中分别进行了两种不同对数据包的解析，解析过程较为简单，可自行查看，主要分别存储定位信息，测距信息和外部4路ADC输入信息。

```
# Receive all waiting packets.
# returns the number of packets read
def sipRecv(self):
  iters = 0
  while(self.port.inWaiting() > 0):
    try:
      recv = self.recvPacket()
    except:
      break
    iters += 1
    # 0x3s means sip packet
    if (recv[3]&0xF0)==0x30:
      self.parseSip(recv)
    # 0xF0 means io_cx packet
    elif recv[3]==0xF0:
      self.parseIO(recv)
    else:
      return 0
  return iters
```

图九

```
# Block until packet recieved
def recvPacket(self):
  timeout = 1.0
  data = [0,0,0]
  while True:
    tstart = time()
    while (time()-tstart) < timeout:
      if self.port.inWaiting() > 0:
        data[2] = ord(self.port.read())
        break
    if (time()-tstart) > timeout:
      raise Exception("Read timeout")
    if data[0] == 0xFA and data[1] == 0xFB:
      break

    data[0] = data[1]
    data[1] = data[2]

  for d in range(data[2]):
    data.append(ord(self.port.read()))

  crc = self.calcChecksum(data)
  if data[len(data)-1] != (crc & 0xFF) or data[len(data)-2] != (crc >> 8):
    self.port.flushInput()
    raise Exception("Checksum failure")
  return data
```

图十

简单进行一下总结，使用者在改写getNextValues后，其输出的线速度，角速度等会被Soar的仿真或实车控制系统捕获，以实车控制系统为例，会将捕获的数据通过串口发送给实车，同时监听来自实车的反馈，对收到的数据进行解析，保存，这之中对数据的格式有着明确的规定，此处未展开讲解，将在下一节详细讲解。到此处，只需要对整个系统大致运行流程理解即可。

通讯协议讲解

通讯协议的制定严格按照“\Software\reference”路径下的数据手册制定。

相关规定可以从第23页开始阅读。

通俗的将，从Soar向实车发送的数据，其通信协议格式满足："帧头+数据长度+数据+校验和”的格式，其中在数据中需要包含数据的命令标识，数据的正负。

从实车向Soar发送的数据，其通信协议格式满足“帧头+数据长度+数据+校验和”，其中的数据共有两种数据格式，不含命令标识符和正负，上位机Soar直接按照数据的先后顺序读取对应的值。

下面以Soar与实车建立通讯时的握手协议为例，在原文档中如图十一写到：

THE CLIENT-SERVER CONNECTION

Before exerting any control, a client application must first establish a connection with the robot server via a serial link through the robot microcontroller's HOST serial port either via the internal HOST or the User Control Panel SERIAL connector. After establishing the communication link, the client then sends commands to and receives operating information from the server.

When first started or reset, ARCOS is in a special wait state listening for communication packets to establish a client-server connection.[7] To establish a connection, the client application must send a series of three synchronization packets containing the SYNC0, SYNC1 and SYNC2 commands in succession, and retrieve the server responses.

Specifically, and as examples of the client command protocol described below, the sequence of synchronization bytes is:

```
SYNC0:  250, 251, 3, 0, 0, 0
SYNC1:  250, 251, 3, 1, 0, 1
SYNC2:  250, 251, 3, 2, 0, 2
```

When in wait mode, ARCOS echoes the packets verbatim back to the client. The client should listen for the returned packets and only issue the next synchronization packet after it has received the appropriate echo.

图十一

总共需要发出三次同步信号SYNC0,1,2，同时需要收到来此实车对应正确的回复信号，才可形成连接，在上位机的相关实现在Pioneer类中的open方法中非常清楚的写明了三次同步信号的发送与接受，以及最终判断前面提到的serialready标志位置1，让update方法开始反复执行。

以SYNC0信号为例，其信息为：250,251,3,0,0,0。250,251对应帧头的0xFA和0xFB，3对应后续的数据长度为3,第一个0指示命令标志位，第二、三个0代表校验和的高八位和第八位。此处由于不需要传输例如角速度，线速度等带正负的数据，所以省去了数据正负的标志位。

而从实车向Soar发送的数据，一共有两种形式，为了表示方便此处直接贴出实际实现的代码及注释，在数据手册中可相应找到依据：

```
//标注*表示soar缺省值 后续观察若无必要可在soar中对应删除
uint8 standard_sip_info[SIP_LEN] = {
      FIRST_HEAD,                                  //Head_first                0
      SECOND_HEAD,                                 //Head_second               1
      46,                                          //Len                       2
      STOP_TYPE,                                   //motors status             3
      0,                                           //xPos 低八位               4
      0,                                           //xPos 高八位               5
      0,                                           //yPos 低八位               6
      0,                                           //yPos 高八位               7
      0,                                           //thPos ---> angle 低八位   8
      0,                                           //thPos ---> angle 高八位   9
      0,                                           //*l_vel 低八位             10
      0,                                           //*l_vel 高八位             11
      0,                                           //*r_vel 低八位             12
      0,                                           //*r_vel 高八位             13
      80,                                          //*Battery 80 ---> 8.0V     14
      0x00,                                        //stall and bumpers 低八位      15
      0x00,                                        //stall and bumpers 高八位      16
      0,                                           //*setPointdegree diff 低八位   17
      0,                                           //*setPointdegree diff 高八位   18
      0x00,                                        //*flags 低八位                 19
      0x00,                                        //*flags 高八位                 20
      0,                                           //*Electronic compass       21
      8,                                           //new sonar reading         22

      0,                                           //sonar_0 标志号            23
      1,                                           //sonar_0 低八位            24
      0,                                           //sonar_0 高八位            25
```

```
      1,                                        //sonar_1 标志号              26
      1,                                        //sonar_1 低八位              27
      0,                                        //sonar_1 高八位              28

      2,                                        //sonar_2 标志号              29
      1,                                        //sonar_2 低八位              30
      0,                                        //sonar_2 高八位              31

      3,                                        //sonar_3 标志号              32
      1,                                        //sonar_3 低八位              33
      0,                                        //sonar_3 高八位              34

      4,                                        //sonar_4 标志号              35
      1,                                        //sonar_4 低八位              36
      0,                                        //sonar_4 高八位              37

      5,                                        //sonar_5 标志号              38
      1,                                        //sonar_5 低八位              39
      0,                                        //sonar_5 高八位              40

      6,                                        //sonar_6 标志号              41
      1,                                        //sonar_6 低八位              42
      0,                                        //sonar_6 高八位              43

      7,                                        //sonar_7 标志号              44
      1,                                        //sonar_7 低八位              45
      0,                                        //sonar_7 高八位              46

      //后续标准sip协议中全部缺省 故不再列出

      0,                                        //crc校验                     47
      0,                                        //crc校验                     48
};

//soar中只使用了analogInputs 5 6 7 8
uint8 standard_io_info[IO_LEN] = {
      FIRST_HEAD,                               //Head_first                  0
      SECOND_HEAD,                              //Head_second                 1
      28,                                       //Len                         2
      IO_TYPE,                                  //IO_flag                     3

      0,                                        //意义不明的缺省占位           4
      0,                                        //若后续要删除在上位机要同步更改
      0,
      0,
      0,
      0,
      0,
      0,
      0,                                        //以上均为缺省占位            12

      0,                                        //analogInputs_0 低八位       13
      0,                                        //analogInputs_0 高八位       14
```

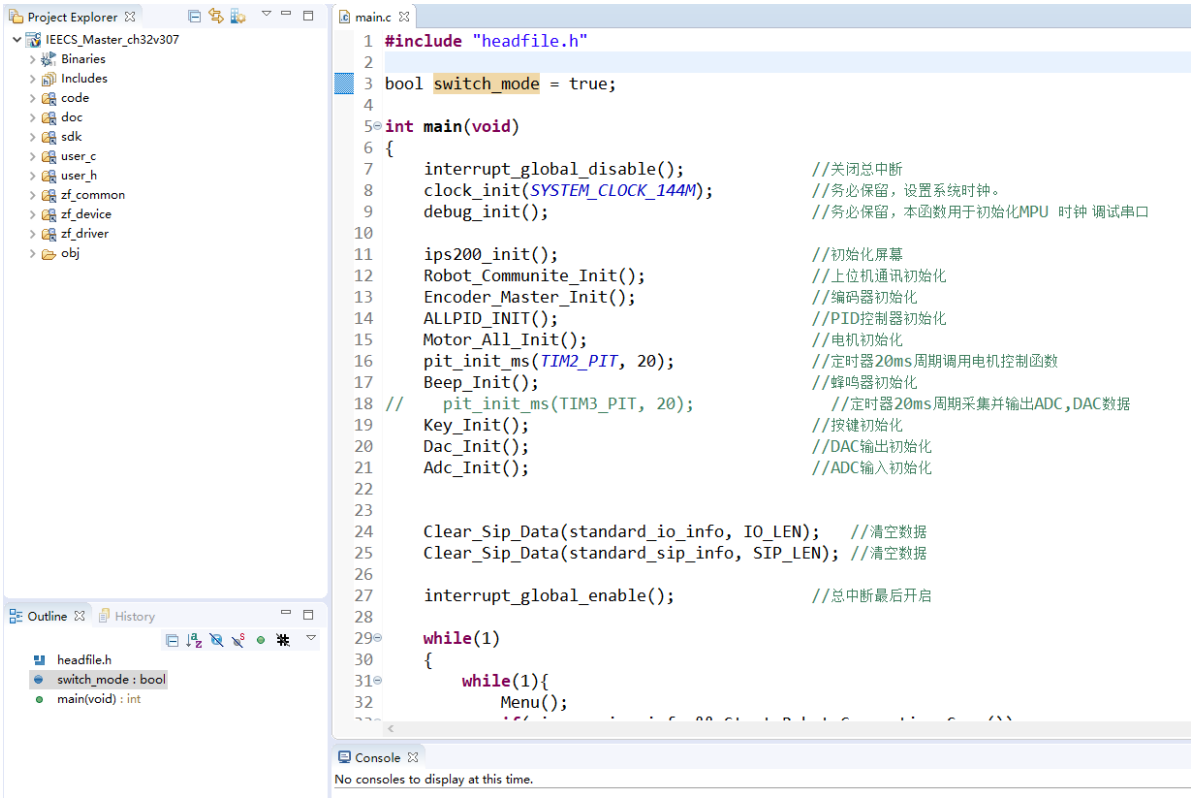
```
    0,                                  //analogInputs_1 低八位      15
    0,                                  //analogInputs_1 高八位      16

    0,                                  //analogInputs_2 低八位      17
    0,                                  //analogInputs_2 高八位      18

    0,                                  //analogInputs_3 低八位      19
    0,                                  //analogInputs_3 高八位      20

    0,                                  //analogInputs_4 低八位      21
    0,                                  //analogInputs_4 高八位      22

    0,                                  //analogInputs_5 低八位      23
    0,                                  //analogInputs_5 高八位      24

    0,                                  //analogInputs_6 低八位      25
    0,                                  //analogInputs_6 高八位      26

    0,                                  //analogInputs_7 低八位      27
    0,                                  //analogInputs_7 高八位      28

    0,                                  //crc校验                    29
    0,                                  //crc校验                    30
};
```

如上代码，一种数据格式为解析sip时使用，一种数据格式为解析io时使用。根据注释很好理解在每一位上需要放置什么数据，只需要将这个数组通过串口从实车向Soar发送即可。更具体信息请参考数据手册。

Car内部系统设计

代码存放于“\Software\IEECS_Master_ch32v307”，此工程需要配合南京沁恒的MounRiver Studio编译器使用，相关使用手册也已存放。

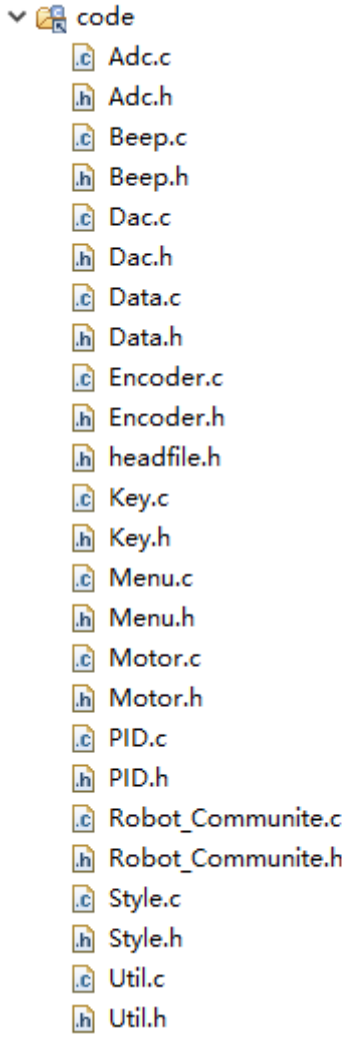
在成功导入工程后，如图十二所示：



图十二

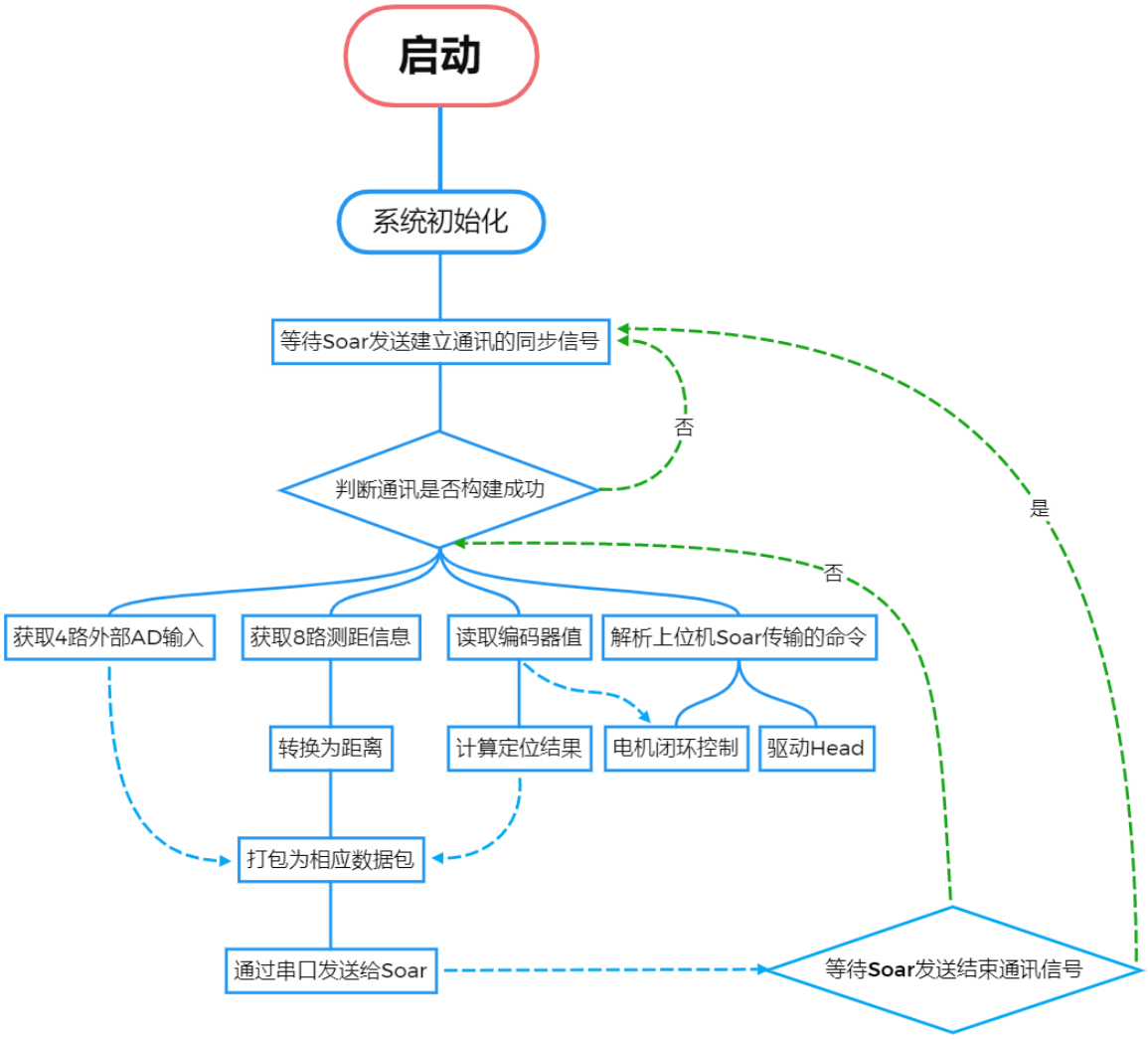
其中，main.c函数和中断相关的函数isr.c存放于user_c文件中，code目录下存放本次编写的全部代码，包括所有的头文件与源文件。其余目录下主要存放sdk与常见的接口函数。

展开code目录后内涵如图十三所示的代码，其功能与名称相对应。其中Robot_Communite(写到这里才发现拼错了，将错就错了)中存放了与soar建立通讯与保持通讯的函数，在代码中也有详细注释。Style中存放了根据编码器值解算位置信息的代码。Menu中存放辅助调试的菜单系统，其他文件名对应功能，比较好理解。



图十三

图十四展示了实车的运行框架结构图。



图十四

首先在系统初始化的阶段，需要对传感器，外设，内部控制系统进行初始化，传感器包括编码器，电机等，外设包括无线串口，屏幕，蜂鸣器等，内部控制系统包括PID控制器等。初始化完成后，首先需要等待上位机发布构建通讯的同步信号命令，此处利用串口中断在中断函数内不断判断是否接收到了有效的命令来判断，总共需要三次握手信号进行同步，当三次同步完成后，则双方构建了正常的通讯。

通讯开始后，Soar便会按照Update方法，不断的发送线速度，角速度和输出电压设定的命令(未加载代码时为0)，而Car也一直向Soar发送包含测距，定位等信息的数据包，以使得两者相互确认彼此处在联系之中。实际实现如图十五。

```
while(1)
{
    while(1){
        Menu();
        if(sip_receive_info && Start_Robot_Connection_Sync())
            break;
        Sonar_Adc_Get();
    }
    Beep_Set_ms(50);
    while(1){
        Menu();
        if(!gpio_get(UART_ROBOT_RTS)){
            Add_SIP_Info(standard_sip_info);
            uart_write_buffer(UART_ROBOT, Add_Checksum(standard_sip_info), SIP_LEN);
            Add_IO_Info(standard_io_info);
            uart_write_buffer(UART_ROBOT, Add_Checksum(standard_io_info), IO_LEN);
        }
        if(sip_receive_info && Keep_Robot_Communite())
            break;
        Sonar_Adc_Get();
    }
    Beep_Set_ms(100);
    system_delay_ms(200);
    Beep_Set_ms(100);
}
```

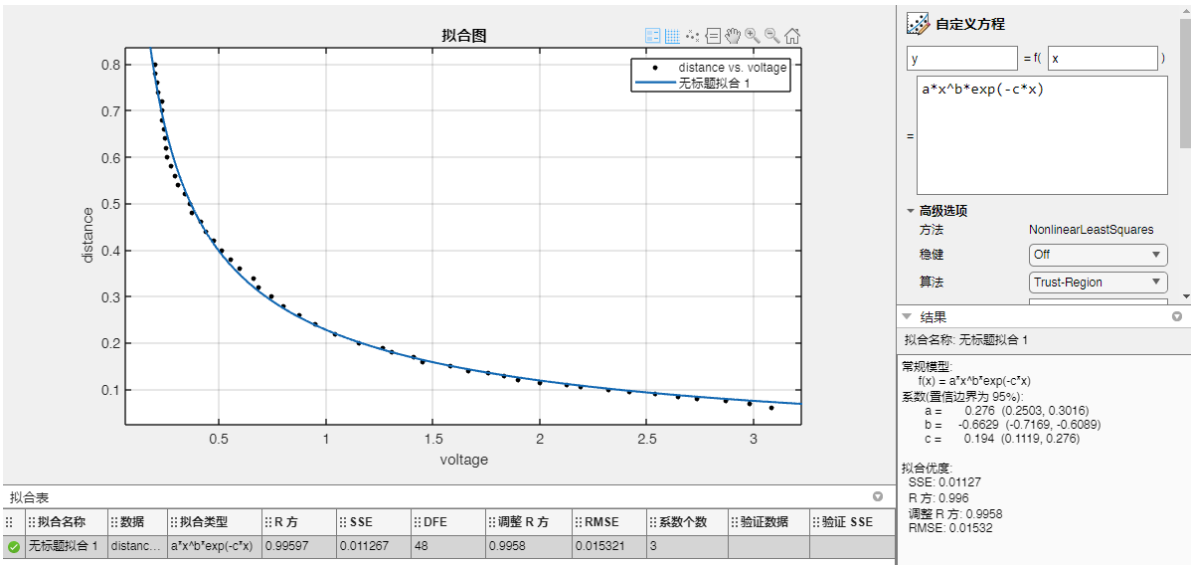
图十五

如图十五，最外层while循环确保小车一直处在运行状态，第一个while循环中小车循环判断是否接受到了同步信号，其中sip_receive_info存放了接受Soar命令数据的指针，当不为空时说明可能接收到了有用的数据，就需要进行判断，对同步结果的判断再Start_Robot_Connection_Sync中完成，如果三次同步完成，则退出第一个while循环，进入第二个，Car开始不停的发送两种格式的数据包，同时解析来自Soar的命令，解析的部分在Keep_Robot_Communite中完成。Sonar_Adc_Get完成了对8路测距信息的采集与转换。

相关的函数功能在代码中已经做了详细的备注，只需要捋清所有的数据是如何获取的即可，此处不在赘述，有关PID控制器在网上有很多详细的资料，可自行学习后再阅读源代码进行理解，其余内容可根据代码中有引导性的注释来阅读理解。下面主要讲解定位算法的实现和ADC的采集。

ADC采集

在“\Software”目录下储存了对于选用的红外测距模块输出电压与实际距离的关系，并利用matlab拟合出有效数据段的近似函数，如图十六所示：



图十六

由于红外测距模块具备一定的盲区，根据实际测试，当距离低于越6厘米时，其输出电压值反而将降低(理论上距离越近电压越大)，这样的区域称为盲区，为避免对盲区的处理，应当尽量避免车运行到盲区范围内，可以对其顶层盖板进行加宽处理，使其可以覆盖所有盲区范围。

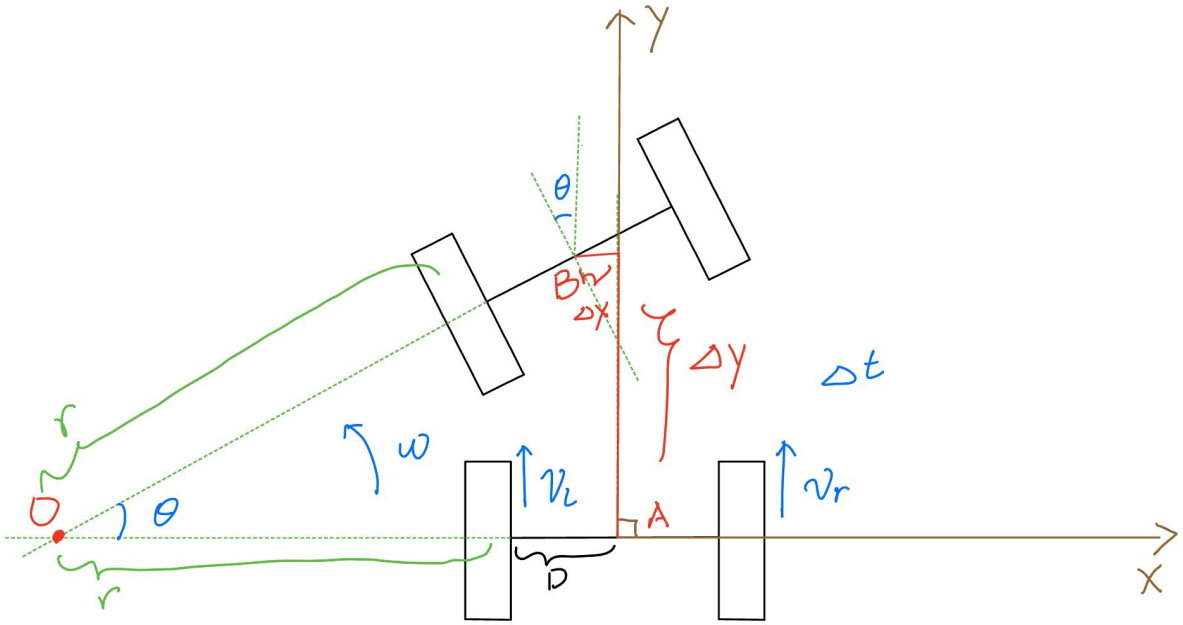
代码中对8路AD测距输入使用了12比特采样，则需要对采样值乘上$\frac{3.3}{4096}$转换为实际单片机引脚输入的电压值，再利用在硬件指南中提到的线性映射关系，转化为实际测距模块输出的电压，再利用以上近似函数求取实际距离即可。

定位算法实现

依靠编码器实现的定位算法实际上是利用了积分的方法。

从编码器在定时器间隔内读取的脉冲数，可以转化为实车的运行速度，例如，本次选用的500线GMP带编码器电机，减速比1:30，则车轮旋转完整的一圈，带动编码器输出15000个脉冲，在20ms的定时器内，若读出编码器产生1000个脉冲，则说明车轮在20ms内旋转了十五分之一圈，则很容易求出车轮的线速度。

下面构建如下运动模型，如图十七：



图十七

设想对于第t时刻车的位置，考虑从t时刻运动到$t+1$时刻的过程，假设间隔为$\triangle t$，则在这段时间间隔内，可以认为车按照第t时刻的速度进行匀速运动。假设第t时刻左右车轮速度为V_l，V_r。左车轮到车中心轴长为D。

从t时刻运动到$t+1$时刻的过程，可以近似认为是匀速圆周运动，假设在$\triangle t$时间内转过角度为θ，则我们以第t时刻车的坐标A为原点建立坐标系，求出θ角，即可求出$t+1$时刻车的坐标B。

这个求解过程并不复杂，首先假设左轮的旋转半径为r（以向左旋转为例）,由于旋转过程中左右轮的角速度是相同的，则有：

$$\omega = \frac{V_l}{r} = \frac{V_r}{r+2D}$$

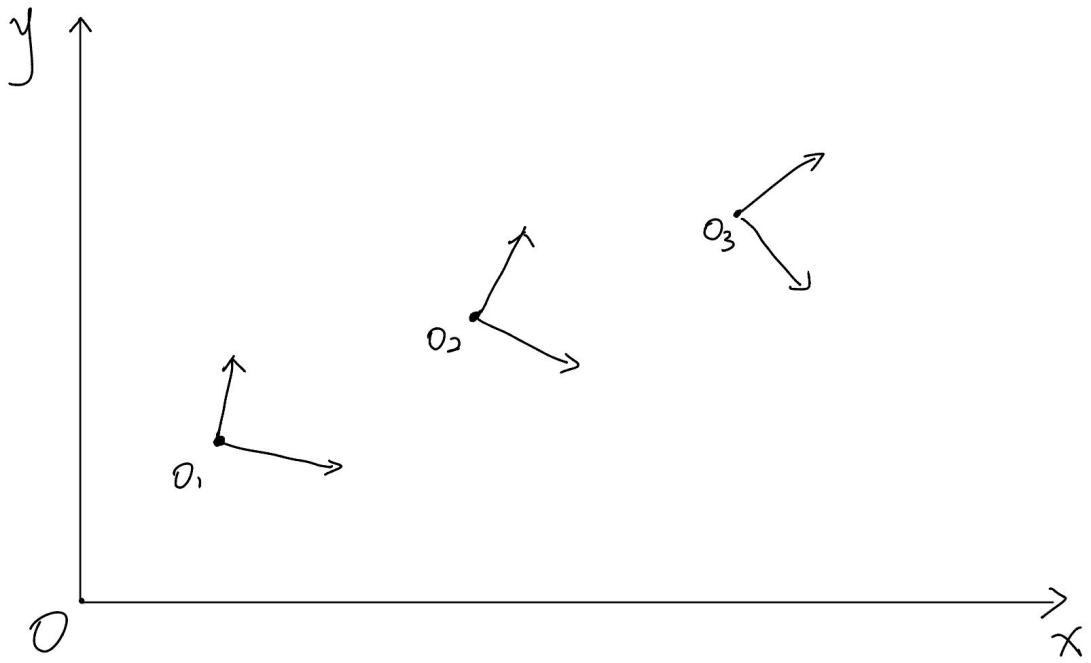
而θ可以用角速度乘时间计算，则：

$$\theta = \omega \cdot \triangle t = \frac{V_l}{r} \cdot \triangle t$$

有了角度后可求出A,B两点坐标差的绝对值：

$$\triangle x = (r + D) \cdot (1 - \cos\theta)$$
$$\triangle y = (r + D) \cdot \sin\theta$$

接下来我们考虑连续变化的过程，如图十八：



图十八

以启动时为全局的坐标原点，而后每间隔$\triangle t$时间记录一次坐标点，即图中$O_1, O_2 \ldots$，而O_1相对于O，O_2相对于O_1,...,O_n相对于O_{n-1}都符合图十七中的情况，也就是说，以O_{n-1}为参考系(此时x, y轴正方向为第$t-1$时刻车轮的水平方向和垂直方向)，O_n坐标在O_{n-1}为参考系下的计算公式就是图十七中的计算公式。但现在需要O_n在初始O为参考系下的坐标。

首先，从初始的起点O运动到O_1,则O_1的坐标很容易通过前面讲解的公式求解(注意正负号即可)，假设为x_1, y_1，而再经过$\triangle t$,运动到O_2点，此时如何求解O_2的坐标成为问题。现在已知的，可以根据前文讲解的公式求解O_2点在以O_1为参考系下的坐标，不妨假设为x_2', y_2',这个坐标显然不是我们需要的，因为他只是以O_1为参考系的坐标，而我们需要的是以O为参考的坐标，如此得到往后所有点的信息，即实现了定位。现在所要完成的就是将x_2', y_2'映射到x_2，y_2上。此时我们可以思考，这里面应该存在某种映射，这种映射实际上就是坐标系的变换法，可以利用下面的式子求解：

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos\triangle\theta_1, -\sin\triangle\theta_1 \\ \sin\triangle\theta_1, \cos\triangle\theta_1 \end{bmatrix} \cdot \begin{bmatrix} x_2' \\ y_2' \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

其中，$\triangle\theta_1$代表了O_1参考系与O参考系角度的差。

类似的，有：

$$\begin{bmatrix} x_3 \\ y_3 \end{bmatrix} = \begin{bmatrix} \cos\triangle\theta_2, -\sin\triangle\theta_2 \\ \sin\triangle\theta_2, \cos\triangle\theta_2 \end{bmatrix} \cdot \begin{bmatrix} x_3' \\ y_3' \end{bmatrix} + \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$$

其中，$\triangle\theta_2$代表了O_2参考系与O参考系角度的差。

依照以上思路，便可以进行迭代求解出任意时刻车以O为参考系的坐标。

以上过程可能并不好理解，建议反复阅读，深入理解坐标系变化的概念以及一个坐标系中的点如何转化到另一个坐标系中。