TUTORIAL TO SET UP THE FT205EV WIND SENSOR AND PLOT WIND SPEED DATA

1. INTRODUCTION

The FT205EV is the first in a new generation of lightweight ultrasonic wind sensors. It is low weight(100g) and a wind speed range up to 75m/s. It has a graphite and nylon composite body. The light weight of the FT205EV together with the proven FT Acu-Res technology make it ideal for use on aerial drones and other weight critical applications.

The FT205EV allows configurable RS422 (full-duplex), RS485 (half-duplex) or buffered UART (full-duplex) communication outputs.

2.FUNCTIONAL DESCRIPTION

2.1 TECHNICAL PERFORMANCE

Measurement Principle:

- Acoustic Resonance (compensated against variations in temperature, pressure and humidity)

Wind Speed Measurement:

Range	0-75 m/s (0-270 km/h, 0 – 145.8 knots)
Resolution	0.1 m/s
Accuracy	±0.3 m/s (0-16 m/s) ±2% (16 m/s-40 m/s) ±4% (40 m/s-75 m/s)

Wind Direction Measurement

Range	0 to 360°
Sensor Accuracy	4° RMS (Root Mean Square)
Compass Accuracy	5° RMS (Root Mean Square)
Resolution	1°

DATA I/O:

Interface options:

- RS422(full-duplex)
- RS485(half-duplex)
- Buffered UART full-duplex (3V & 5V TTL compatible)

Format:

- ASCII ‘Polled’ or ‘Continuous Update’
- Polar and NMEA 0183

Data Update Rate:

- Up to 10 measurements per second (10Hz)

Power Requirements

- Supply Voltage: 6-30 VDC range (24 VDC nominal)
- Sensor Current: 30mA typical

3. INSTALLATION

3.1 Pin Connections:

Fig 3.1: (Above) Molex CLIK-Mate 8-way single-row connector (Below) Pin Numbering of the connector

Pin	RS422 (full-duplex)	RS485 (half-duplex)	Buffered UART (full-duplex)	Molex Wire Colour
1	Ground			Black
2	6-30 VDC			
3	RS422_A RX (-)	RS485_A (-)		
4	RS422_B RX (+)	RS485_B (+)		
5	RS422_A TX (-)			
6	RS422_B TX (+)			
7			UART_RX	
8			UART_TX	

Fig 3.2: Pin connections for FT205EV Wind Sensor

The sensor has reverse polarity protection and is protected against miswiring of the intended cable and power supply rating.

All connections from the wind sensor to any data acquisition equipment and power supply should run through suitable Surge Protection Devices (SPD). This will suppress unwanted overvoltage transient present on the signal or power lines.

The Figure 3.3 gives the connection diagram for the sensor with the Arduino using UART interface. We make pins 10 and 11 on the Arduino as Software Serial connections with the sensor. To give power to the sensor we use 12V Power Supply Adapter.

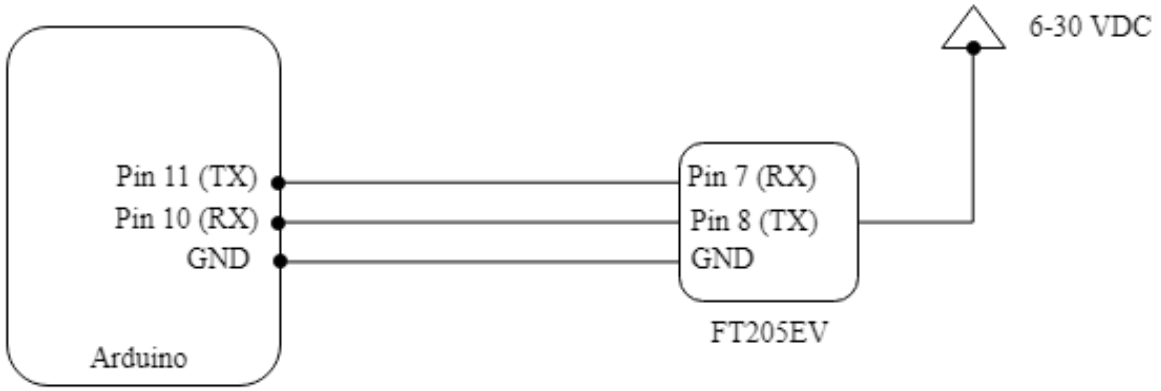

Fig 3.3: Connection with Arduino for UART interface with sensor

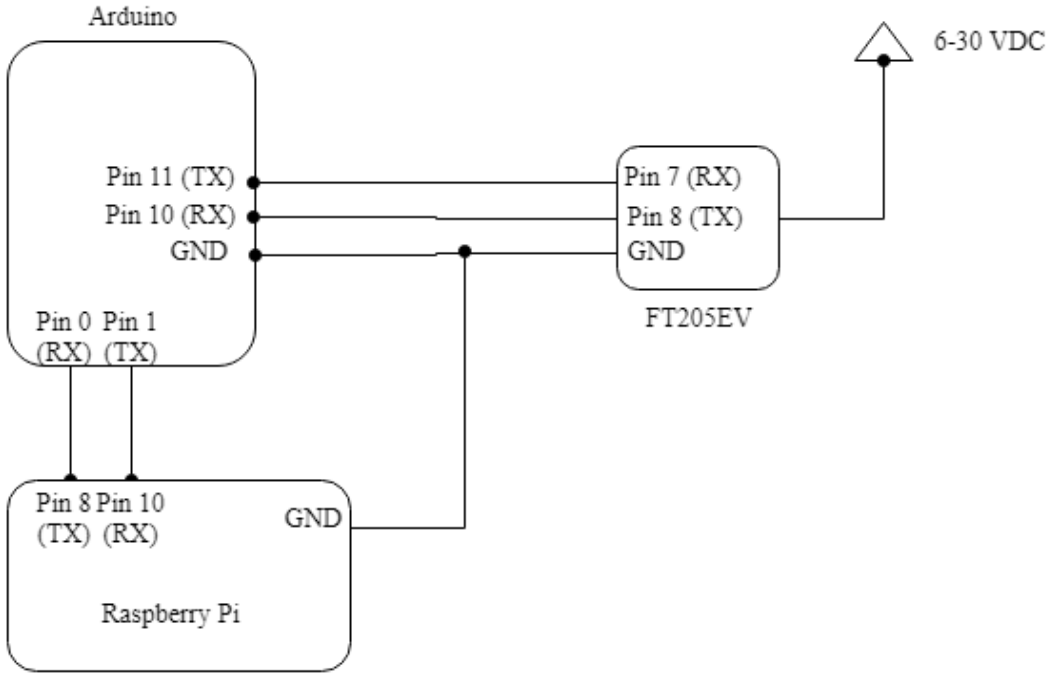

Fig 3.4: Connection between Arduino, Raspberry Pi and FT205EV with UART interface

Figure 3.4 gives the connection diagram between the Arduino, raspberry pi and the Wind Sensor. The raspberry pi is used to collect the data from the Arduino and plot wind speed and direction collected from the FT205EV wind sensor.

4. Sensor Communication

4.1 UART Protocol

UART is typically used over short transmission distance (typically 0-600mm) and can only support one master (controller) and 1 slave (wind sensor) device. Note: it is 3/5V TTL compatible

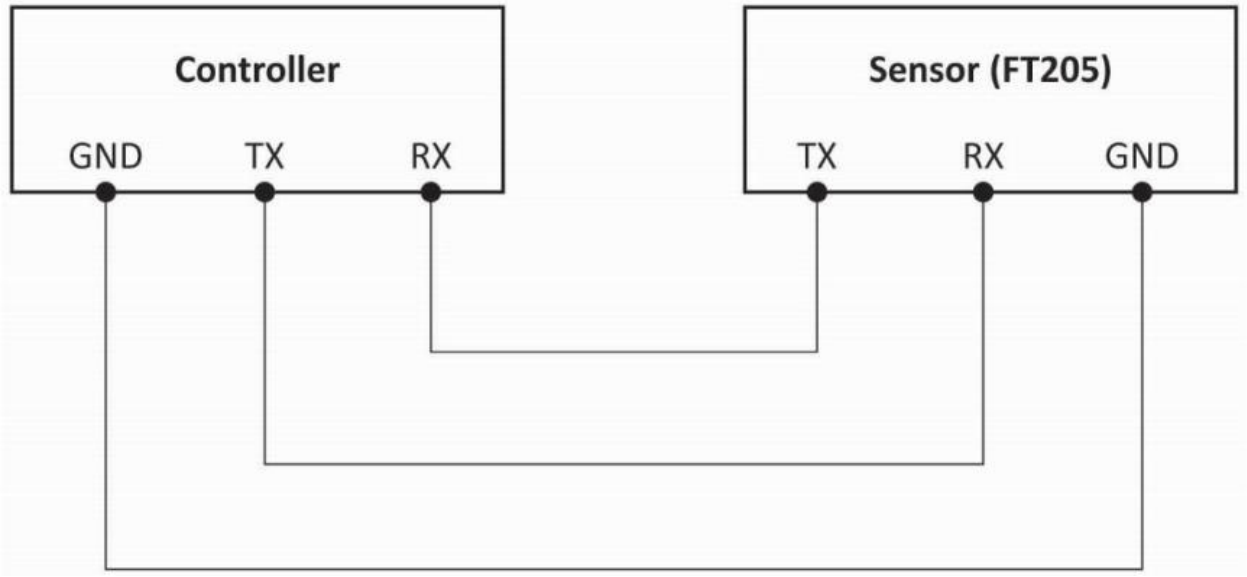

Fig 4.1: UART Connection Diagram

4.2 Configuring the Sensor

All user parameter settings are stored in non-volatile memory and are retained when the sensor is switched off. When the sensor is next switched on (or a user reset command is sent) the sensor will revert to these settings. The sensor can therefore be configured prior to final installation if required.

Data is transmitted and received via an asynchronous serial communication interface using ASCII characters. To set the sensor baud rate use the BR command.

4.2.1 Message Format

Data communication between the sensor and the host computer is performed by the transmission of ASCII messages. The same message format is used for both received and transmitted messages.

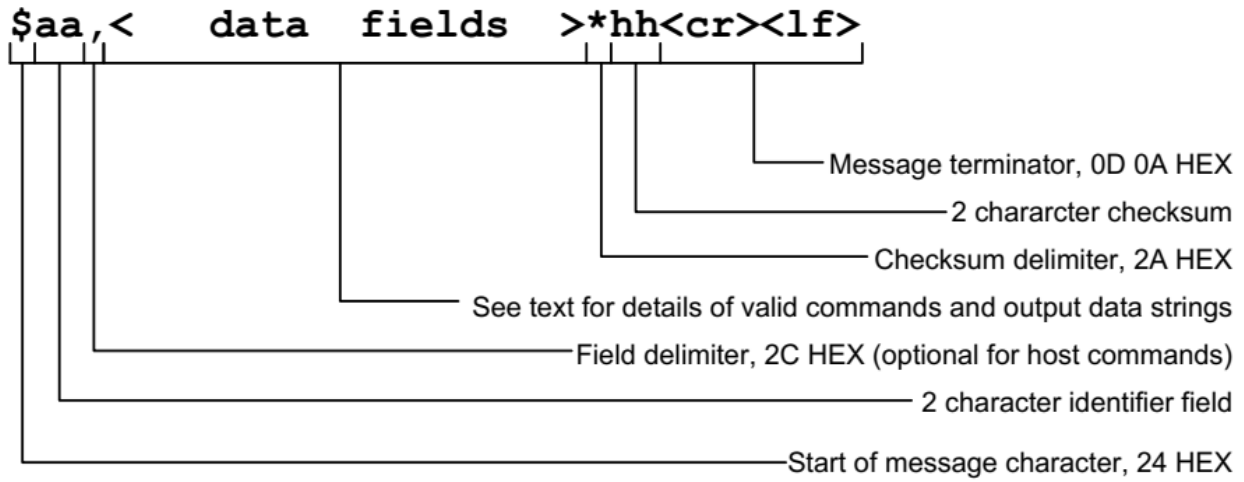

Fig 4.2: Message Format used to communicate with the sensor

All messages start with the '$' start of message character, followed by the 2-character talker/listener identifier field.

Following the first delimiter is the main body of the message which comprises a variable number of data fields (dependent on the message being transmitted), each separated by the field delimiter character (','). Data fields may contain alpha, numeric, or alphanumeric data depending on the information content of the field.

Messages sent to the sensor will contain a command in <data fields> and messages from the sensor will contain output data in <data fields>.

The data field section of the message is terminated by the checksum delimiter character '*'. Following the checksum delimiter is the two-character checksum field.

All messages are terminated with a carriage return <cr> and line feed <lf>.

4.2.2 Listener and Talker Identifiers

The sensor is assigned with both a Listener and a Talker identifier address that allows an individual sensor to be uniquely identified in a system comprising more than one sensor.

Whenever a message is sent to the sensor, the identifier field of the message (the 2 characters immediately following the '$' start of message character) must correspond to the sensor Listener

identifier address, otherwise the sensor will ignore the message. If you do not wish to use the Listener ID in the messages sent from the host computer, you can replace the Listener ID with '//'. Sending '//' in place of the Listener ID will allow any sensor, irrespective of its Listener ID setting, to respond to the message.

Whenever a message is transmitted from the sensor, the identifier field of the message (the 2 characters immediately following the '$' start of message character) will contain the Talker ID. The Talker ID is used as a message tag to identify which sensor has transmitted the message.

The factory default value for the Listener ID is 01 and for the Talker ID it is WI (Weather Instrument). To change the Listener and/or Talker ID use the ID Command.

4.2.3 Calculating the Message Checksum

All messages sent to, or received from, the sensor include a checksum field. Messages that are transmitted from the sensor always include a checksum value in the checksum field. Messages sent to the sensor by the host computer can either contain a checksum value or an 'ignore checksum identifier' in the checksum field.

The checksum value is calculated by Exclusive OR'ing (XOR'ing) all the bytes between (but not including) the '$' and the '*' characters of the message. The resulting single byte value is then represented by 2 HEX characters in the message string. The most significant character is transmitted first.

Since a message only contains ASCII characters (which have values in the range 0-7F) the checksum value will always be between 0 and 7F.

4.2.4 Disabling the Checksum

It is recommended that a checksum value be computed for all messages which are sent to the sensor, in some cases this may not be convenient (i.e. when communicating with the sensor with a terminal). To prevent the sensor from performing checksum validation of incoming messages, send the ASCII characters '//' in place of the checksum value.

4.2.5 Timing Constraints

When a valid command is received by the sensor input buffer, there will be a time delay whist the command is being processed. The actual command latency depends on exactly when the last character of the command is received within the sensor internal processing cycle. The sensor can process only one SET or QUERY command at a time.

Once a SET command has been received by the sensor, it can take approximately 400ms for the command to be processed and any setting change implemented. If other commands are sent during this period, they may be ignored by the wind sensor. Therefore, all SET commands must be separated by a period of at least 500ms before further commands are sent.

Once a QUERY command has been received by the sensor, it takes up to 50ms for the command to be processed. The sensor will then wait for a predefined delay before sending a response. This delay time is programmable in increments of 50ms.

4.3 Some Basic Commands to The Sensor

4.3.1 COMMAND PARAMETER - BR

Use the BR command to change the sensor serial interface baud rate. The new baud rate setting will only come into effect when the sensor is next powered-up or after a Reset command (RSU) has been received.

If the baud rate is changed, you will only be able to communicate with the sensor if the host computer's baud rate is set to the same baud rate. If you do not know what the current setting of the sensor baud rate is, you will need to try each baud rate in turn until you establish communication.

Command Syntax		
	SET Sensor:	*$<listenerID>,BR<baudrate>*<checksum><cr><lf>* *$aa,BRx*hh<cr><lf>*
	QUERY Sensor:	*$<listenerID>,BR?*<checksum><cr><lf>* *$aa,BR?*hh<cr><lf>*
	Sensor Output:	*$<talkerID>,BR=<baudrate>*<checksum><cr><lf>* *$aa,BR=x*hh<cr><lf>*

Parameters	<baudrate>	
	0	Set the baud rate to 38400 baud
	1	Set the baud rate to 19200 baud
	2	Set the baud rate to 9600 baud (Factory Default Setting)
	3	Set the baud rate to 4800 baud
	4	Set the baud rate to 2400 baud
	5	Set the baud rate to 1200 baud

Examples	Example 1 Set the baud rate to 19200 baud, verify the new setting and send a user reset command to activate the new baud rate	
	Message	Comment
	$01,BR1//<cr><lf>*	Set baud rate to 19200
	$01,BR?//<cr><lf>*	Query baud rate setting
	*$WI,BR=1*2E<cr><lf>*	Sensor Output
	$01,RSU//<cr><lf>*	Send user reset

4.3.2 COMMAND PARAMETER - CI

Use the CI command to switch between communications interface modes. Only one communications interface can be used at a time, however the sensor is able to receive host commands through any of the three supported interface modes. When selecting the RS422 interface, the sensor's RS422 transmitter may be set to remain permanently enabled (or only enabled during message transmission).

Command Syntax	SET Sensor:	*$<listenerID>,CI<interface>*<checksum><cr><lf>* *$aa,FLc*hh<cr><lf>* or *$aa,FLcc*hh<cr><lf>*
	QUERY Sensor:	*$<listenerID>,CI?*<checksum><cr><lf>* *$aa,CI?*hh<cr><lf>*
	Sensor output:	*$<talkerID>,CI=<interface string>*<checksum><cr><lf>* *$aa,CI=ccccc*hh<cr><lf>*
	QUERY RS422 Transmitter Mode:	*$<listenerID>,CI?2*<checksum><cr><lf>* *$aa,CI?2*hh<cr><lf>*
	Sensor output:	*$<talkerID>,CI=<RS422 tx mode>*<checksum><cr><lf>* *$aa,CI=c*hh<cr><lf>*

Parameters	<interface>	
	8	Set the RS485 interface (Factory Default Setting)
	2A	Set the RS422 interface with the transmitter always enabled.
	2I	Set the RS422 interface with the transmitter disabled when not transmitting.
	U	Set the UART communication interface
	<interface string>	
	RS485	The RS485 Interface is enabled.
	RS422	The RS422 Interface is enabled.
	UART	The UART Interface is enabled.
	<RS422 tx mode>	
	A	The RS422 transmitter is always enabled.
	I	The RS422 transmitter is disabled when not transmitting.

Examples	Example 1 Enable the RS485 interface. Verify that the command has been accepted.	
	Message	Comment
	$01,CI8//<cr><lf>*	Enable RS485
	$01,CI?//<cr><lf>*	Query communications interface
	*$WI,CI=RS485*3D<cr><lf>*	Sensor response
	Example 2 Enable the RS422 interface with the transmitter always enabled. Verify that the interface and transmitter settings.	
	Message	Comment
	$01,CI2A//<cr><lf>*	Enable RS422 with transmitter on
	$01,CI?//<cr><lf>*	Query communications interface
	*$WI,CI=RS422*30<cr><lf>*	Sensor response
	$01,CI?2//<cr><lf>*	Query RS422 transmitter mode
	*$WI,CI=A*44<cr><lf>*	Sensor response

4.3.3 COMMAND PARAMETER - CU

Use the CU command to enable or disable the continuous update mode of operation. When continuous update is enabled, the sensor will output wind velocity readings at a rate determined by the <interval> setting.

Each time the continuous update mode is enabled, the required <interval> setting must be sent (even if this has been sent to the sensor previously). When the continuous update mode is enabled, if the sensor is switched-off, when power is reapplied the sensor will automatically resume outputting readings.

Once the sensor has been put into continuous update mode then it becomes a talker only and will not respond to any further commands. To be able to send commands again the continuous mode must be disabled. To achieve this, the CUD (disable continuous update mode) command must be sent within the first four seconds of the power being applied to the sensor.

Command Syntax	SET Sensor:	*$<listenerID>,CU<cont.update>,<interval>*<checksum><cr> <lf>* *$aa,CUcxxxxx*hh<cr><lf>*
	QUERY Sensor:	*$<listenerID>,CU?*<checksum><cr><lf>* *$aa,CU?*hh<cr><lf>*
	Sensor Output:	*$<talkerID>,CU=<cont.update>,<interval>*<checksum><cr> <lf>* *$aa,CU=c,xxxxx*hh<cr><lf>*

Parameters	<continuous update>	
	E	Enabled
	D	Disabled (Factory Default Setting)
	<interval>	
	1 to 59999	interval, in 0.1s increments, between outputs in continuous mode

Examples	Example 1 Set the sensor to output readings automatically every 10 seconds. Verify that the command has been accepted.	
	Message	Comment
	$01,CUE00100//<cr><lf>*	Enable CU mode, rate = 0.1Hz
	Example 2 Disable the continuous updating. Verify that the command has been accepted.	
	Message	Comment
	$01,CUD//<cr><lf>*	Disable CU mode
	$01,CU?//<cr><lf>*	Query CU mode setting
	*$WI,CU=D,00100*40<cr><lf>*	Sensor response

4.3.4 COMMAND PARAMETER - DF

Use the DF command to set the required format of the wind velocity readings. When a DF Set command is sent to the sensor, a reset of the minimum and maximum readings to their default values is automatically performed.

Polar Format: The sensor returns the magnitude of the wind speed (m/s only) and the wind direction (0-359 degrees).

NMEA 0183 Format: The sensor returns the wind angle (0-359 degrees, Relative) and wind speed (m/s, knots or km/h). The sensor TalkerID is always set to WI when NMEA format is selected irrespective of any value that may have been set with the ID command.

Command Syntax	SET Sensor:	*$<listenerID>,DF<format>*<checksum><cr><lf>* *$aa,DFc*hh<cr><lf>* or *$aa,DFcc*hh<cr><lf>*
	QUERY Sensor:	*$<listenerID>,DF?*<checksum><cr><lf>* *$aa,DF?*hh<cr><lf>*
	Sensor output:	*$<talkerID>,DF=<format>*<checksum><cr><lf>* *$aa,DF=c*hh<cr><lf>*

Parameters	<format>	
	P	Set the data format to Polar (wind speed and direction) (Factory Default Setting)
	N	Set the data format to NMEA 0183 with wind speed in m/s
	NN	Set the data format to NMEA 0183 with wind speed in knots
	NK	Set the data format to NMEA 0183 with wind speed in km/h

Examples

Example 1

Set the wind velocity output data format to NMEA with wind speed in m/s and verify the new setting.

Message	Comment
$01,DFN//<cr><lf>*	Set format to NMEA (m/s)
$01,DF?//<cr><lf>*	Query format setting
*$WI,DF=N*43<cr><lf>*	Sensor response

Example 2

Set the wind velocity output data format to NMEA with wind speed in knots and verify the new setting.

Message	Comment
$01,DFNN//<cr><lf>*	Set format to NMEA (knots)
$01,DF?//<cr><lf>*	Query format setting
*$WI,DF=NN*0D<cr><lf>*	Sensor response

4.3.5 COMMAND PARAMETER - ID

Use the ID command to set the listener and talker address identifiers.

Command Syntax	SET Sensor:	*$<listenerID>,ID<RxID><TxID>*<checksum><cr><lf>* *$aa,ID=cccc*hh<cr><lf>*
	QUERY Sensor:	*$<listenerID>,ID?*<checksum><cr><lf>* *$aa,ID?*hh<cr><lf>*
	Sensor output:	*$<talkerID>,ID=<RxID><TxID>*<checksum><cr><lf>* *$aa,ID=cccc*hh<cr><lf>*

Parameters	<RxID>	
	00 to ZZ	The sensor 2 digit listener address identifier (Factory Default RxID = 01)
	<TxID>	
	00 to ZZ	The sensor 2 digit talker address identifier (Factory Default TxID = WI)

Examples	Example 1 Set the sensor listener address identifier to A1 and the talker address identifier to B1. Verify that the command has been accepted.	
	Message	Comment
	$01,IDA1B1//<cr><lf>*	Set address ID's
	$A1,ID?//<cr><lf>*	Query ID settings
	*$B1,ID=A1B1*6C<cr><lf>*	Sensor response
	Note: the ID? command must use the new listener ID otherwise the command will not be recognised.	

4.3.6 COMMAND PARAMETER - RS

Use the RS command to reset the sensor software. The sensor will be ready to receive new commands or take readings from a maximum of 2 seconds after any reset command is sent. To restart the software but continue to use the previous user parameter settings use the RSU command to restart the software but load the saved parameter settings use the RSS command. To restart the software but load the factory default parameter settings use the RSF command.

Command Syntax	SET Sensor:	*$<listenerID>,RS<mode>*<checksum><cr><lf>* *$aa,RSc*hh<cr><lf>*
	QUERY Sensor:	*NA*
	Sensor output:	*None*

Parameters	<mode>	
	F	Reset the sensor, loading the factory default settings
	S	Reset the sensor, loading saved parameters settings
	U	Reset the sensor, reloading the user parameter settings

Examples	Example 1 Reset the sensor, reloading the last parameter settings	
	Message	Comment
	$01,RSU//<cr><lf>*	Reset sensor, reloading last settings

4.3.7 COMMAND PARAMETER - WV

The WV command returns the wind velocity value in the currently selected format. Polar or NMEA formats are available. Use the DF command, Section 7.4.9, to select the required output format.

Polar Format: The sensor returns the magnitude of the wind speed (m/s) and the wind direction (0-359°).

NMEA 0183 Format: The sensor returns the NMEA 0183 Wind Speed and Angle sentence MWV

Command Syntax	SET Sensor:	N/A
	QUERY Sensor:	*$<listenerID>,WV?*<checksum><cr><lf>* *$aa,WV?*hh<cr><lf>*
	Sensor output:	*$<talkerID>,WVP=<speed>,<angle>,<status>*<checksum><cr><lf>* *$aa,WVP=xxx.x,xxx,x*hh<cr><lf>*

Parameters	<speed> *000.0 to 075.0*	Measured wind speed in metres per second
	<angle> *000 to 359*	Measured wind direction in degrees relative to sensor datum
	< status > *0 to Z*	Indicates whether an error condition was detected by the sensor operating system. A status value of 0 indicates no issues have been detected (ASCII 30(HEX)). If the sensor detects an error condition, the status character will be set to 1. If the Overspeed Warning Scheme (see Section 2.8) is enabled and if the sensor detects wind speed above the maximum range, the status flag will be set to 2.

Examples	Example 1 The following example illustrates the polar wind velocity data format. The example shows the sensor output with a wind speed of 20m/s and a wind angle of 45°.	
	Message	Comment
	$01,WV?//<cr><lf>*	Query the wind velocity
	*$WI,WVP=020.0,045,0*73<cr><lf>*	Sensor polar response

5.1 Program to set up sensor and print sensor data

First set up the circuit according to the diagram given in Fig 3.3. The Arduino is used to send commands to the sensor using UART interface. The data sent by the sensor is displayed on the serial monitor on the computer.

We make pins 10 and 11 software serial pins to communicate with the sensor since the Arduino serial pins 0 and 1 (hardware serial) are used to display data on the serial monitor on the computer.

We give enough delay between the commands since the sensor takes minimum 400ms to process a command.

The Arduino sends a command to the sensor to set the communication interface to UART and then we send a Query to check if the communication interface has been set.

We then send a query to the sensor for the wind speed and direction every 1 second. The sensor responds with ASCII messages in the format as discussed earlier.

```
#include <SoftwareSerial.h>
// Declare software serial
SoftwareSerial mySerial(10, 11);
void setup() {
  Serial.begin(9600);
  mySerial.begin(9600);
  Serial.print("Wind sensor initialize");
  delay(3000);
  // SET communication interface of the sensor to UART
  mySerial.write("$01CIU*//\r\n");
  delay(2000);
    // QUERY to check if the communication protocol is set or not
  mySerial.write("$01CI?*//\r\n");
  delay(1000);
}
void loop() {
  // send query for WIND speed and direction
  mySerial.write("$//WV?*//\r\n");
  while  ( mySerial.available())
  {
    Serial.write(mySerial.read());
  }
  Serial.println();
  delay(1000);

}
```

Fig 5.1: Arduino Code to get Wind Sensor Data

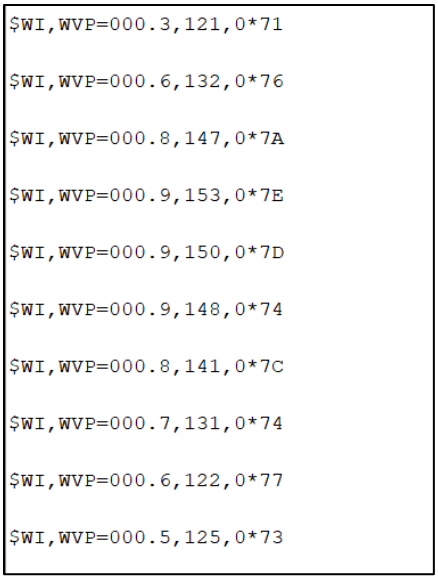

Fig 5.2: Wind Sensor Data Output to Serial Monitor

5.2 Program to Send Data Received By The Arduino To The Raspberrypi And Plot the Data

We now need to extract the wind speed and direction data from the message packets and send to a RaspberryPi for plotting. We will use a custom binary protocol to send data between the Arduino and the RaspberryPi.

A binary protocol is where all data is expressed in numerical values (byte-by-byte). All packets will have the following format:

[header, payload_length, ….payload….]

The header signifies if the packet is a request or response, the payload length gives the length of the payload contained in the packet. The RaspberryPi will send a request to the Arduino asking for the wind speed and direction. Since it is a request there is no payload and the payload length of this packet will be 0.

Once the Arduino receives the packet it will first check the packet if it is valid. It will then send a query to the wind sensor for the wind speed and direction. The sensor responds with fixed size packets in a fixed format. We parse the packet and extract the speed and direction data and create a response packet to be sent to the RaspberryPi.

We convert the sensor data into Float data-type which is of size 4-bytes. Hence the packet that we send to the RaspberryPi will have payload length of 8.

```
#define MCU_RX 10
#define MCU_TX 11

#define MIN_PACKET_SIZE 2

#include <SoftwareSerial.h>

SoftwareSerial mySerial(10,11);

//Packet sent from the raspi
typedef enum {
  HEADER_CMD_REQ_WIND_SPEED, // packet to request wind speed
} header_command_t;

// packet sent from arduino
typedef enum {
  HEADER_RSP_WIND_SPEED,   // response to wind speed request
} header_response_t;

// buffer for packet payload
// populated before calling handle_packet()
static uint8_t _payload_buffer[UINT8_MAX];

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600); // Start UART connection with baud rate 9600
  mySerial.begin(9600);
  mySerial.write("$01CIU*//\r\n");
  // delay(2000);
  // Query to check if Communiction Interface is set to UART
  // mySerial.write("$01CI?*//\r\n");
  // delay(1000);
}

void loop() {
  // all packets have the format: [header, payload_length, ...payload...]
  // once there are two bytes available, read the header and length
  // request packet contain only header and payload length and no payload the packet is [0, 0]
  if (Serial.available() >= 2) {
    header_command_t header = (header_command_t)Serial.read();
    uint8_t len = Serial.read();
    // handle the received packet
    handle_packet(header, len);
  }
}
```

```
void handle_packet(header_command_t header, uint8_t len) {
  char data[26];
  char ch;
  int i = 0, index = 0;
  float get_speed = 0.0;
  float direction_datum = 0.0;
  switch (header) {
    case HEADER_CMD_REQ_WIND_SPEED: {
        if (len == 0) {
          mySerial.write("$//WV?*//\r\n");
          delay(500);
          while  ( mySerial.available())
          {
            ch = mySerial.read();
                data[i++] = ch - 48; //convert ascii to numeric value
              }
         if ( data[18] == 0)   // Check error bit if 0 then no error
          {
            // extract speed from sensor data packet
           get_speed = data[9] * 10 + data[10] + data[12] * 0.1;
           direction_datum = data[14] * 100 + data[15] * 10 + data[16];
           uint8_t packet[2 + sizeof(float) + sizeof(float)];
            // uint8_t packet[2 + sizeof(long)];
           int k = 0;
            packet[0] = HEADER_RSP_WIND_SPEED;
            packet[1] = sizeof(float) + sizeof(float);
            memcpy(&packet[2], &get_speed, sizeof(float));
            memcpy(&packet[6], &direction_datum, sizeof(float));
            for(int  k = 0 ; k < 10 ; k ++){
              Serial.write(packet[k]);
            }
            }
        }
        break;
 }}}
```

On the RaspberryPi side the code is written in Python3. The RaspberryPi needs to send a packet to the Arduino requesting the sensor information. Once the Arduino sends the packet back we need to extract the payload containing the speed and direction information.

Once we have extracted the data then we can continue to plot. For convenience to plot real-time data we create 2 threads that can perform the tasks simultaneously. The first task is to send and receive the data packets from the Arduino and the second task is to plot the data received. We thus divide them into 2 functions and make the function that handles the data transfer between the Arduino and the RaspberryPi as the first thread that is executed first. Once we set the target to that thread then we call the animation function which performs the real time plotting of the data.

```
# enums are used to assign labels to a set of values
#allowing name based referencing of underlying value
from matplotlib import pyplot as plt
from enum import Enum
from array import array
from time import time
import matplotlib.animation as animation
import numpy as np
import struct
import serial
import time
import threading
  # used to plot data
class HeaderCommand(Enum):
      REQUEST_SPEED = 0     #request speed header value is 0

class HeaderResponse(Enum):
      SPEED = 0             #Response header Speed = 0

def get_speed_value(serial):
    # construct the packet no payload
    packet = [
        HeaderCommand.REQUEST_SPEED.value, # header
        0, #payload length
    ]
   # packet is [0,0]
    serial.write(packet)

    while serial.in_waiting < MIN_PACKET_SIZE:
       pass

    header = ord( serial.read())
    packet_recv = []  # empty packet_recv before filling it
    packet_recv.append( HeaderResponse(header)) #convert header byte into enum
    packet_recv.append( ord(serial.read()))
    length = packet_recv[1]
    assert(length == 8) # this packet should have 8 byte payload since speed and direction each is 4 bytes
              # wait for payload
    while serial.in_waiting < length:
        pass
     # append speed and direction as bytes in packet_recv list speed then direction
    for i in range( 0,8):
        packet_recv.append(ord(serial.read()))
    get_speed = struct.unpack_from('f', array('B', packet_recv[2:6]))[0]
    direction = struct.unpack_from('f' , array('B', packet_recv[6:10]))[0]
    return round(get_speed,1), round(direction,1) # we create a tuple using a comma so we return a tuple containing speed and dir
```

```
def plot_speed(i):
    global speed, dir,ax
    print(speed[1],dir[1],'\n')
    point, = ax.plot(dir,speed, 'b-')
    fig.canvas.draw()
    plt.cla()
    ax = fig.add_subplot(111, projection='polar')
    plt.title('speed in meters/sec')
    time.sleep(2)
    return point,


speed = [0,0]
dir = [0,0]
def get_spd():
    while True:
        print("Get Speed and Direction")
        s, d = get_speed_value(hardware);
        d = np.radians(d)
        print(s,np.degrees(d))
        speed[1] = s
        dir[1] = d
        time.sleep(4)


MIN_PACKET_SIZE = 2
i = 0
packet_recv = []
direction = 0.0
get_speed = 0.0
hardware = serial.Serial('/dev/ttyS0',115200)
fig = plt.figure(1)
ax = plt.subplot(111, projection = 'polar')
plt.title('speed in meters/sec')
    #Animation function that will make graph seem real-time and updating
    #will do the data handling -- calls function foo
thread1 = threading.Thread(target=get_spd)
thread1.start()
ani = animation.FuncAnimation(fig, plot_speed)
plt.show()
```

get_speed_value() is used to send a request to the Arduino and wait for the packet sent by the Arduino. Once we receive the payload we unpack it from the list and return the value to function call.

plot_speed() is used to plot the incoming data. We define a polar plot in our main function hence the incoming values are automatically converted to their polar coordinates and plotted. Since we need to get a real time feel of the data, we do not need the past data. Hence, we erase each plot as soon we plot it and create a subplot ready for the next set of data points.

The plot is designed to show the speed and direction of the wind in one figure. The length of the radii gives the magnitude of the wind speed and its direction about the axis gives the wind direction. The unit of wind speed is meters/second and the direction is in degrees (0° to 359°). However, for plotting purposes we need to convert the direction from degrees to radians which we do using the NumPy function radians().

Foo () is used to call the get_speed_value() function and prepare the points to be plotted. This is the target function and is the first thread that is called.

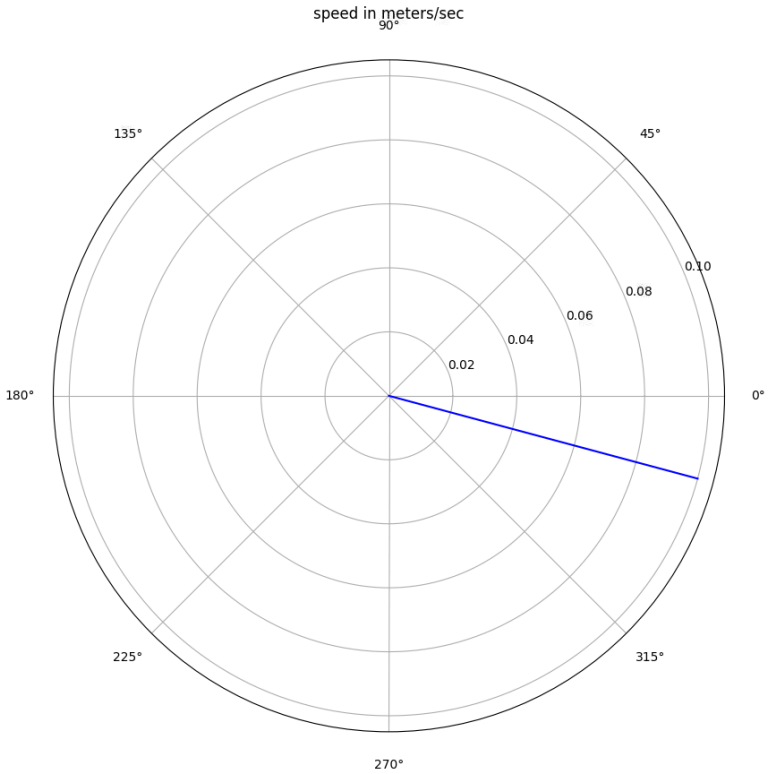

Fig: Plot showing Wind Speed and Direction.