Sistemas Digitales

Trabajo Final:

Desarrollo de Firmware para estación meteorológica

Martín Noblía

Profesores:

José Juarez

Eric Pernía

Universidad Nacional de Quilmes

2016

1 Introducción

Las estaciones Meteorológicas son una serie de sensores que actuan en conjunto para estimar variables referidas al clima, como ser:

- Temperaturas
- Dirección del viento
- velocidad del viento
- Humedad
- Si esta lloviendo (sensor de lluvia)

Estas variables son muy importantes para los procesos productivos donde su productividad depende de las condiciones climáticas.

- Viñedos
- Granjas
- Campos de agricultura

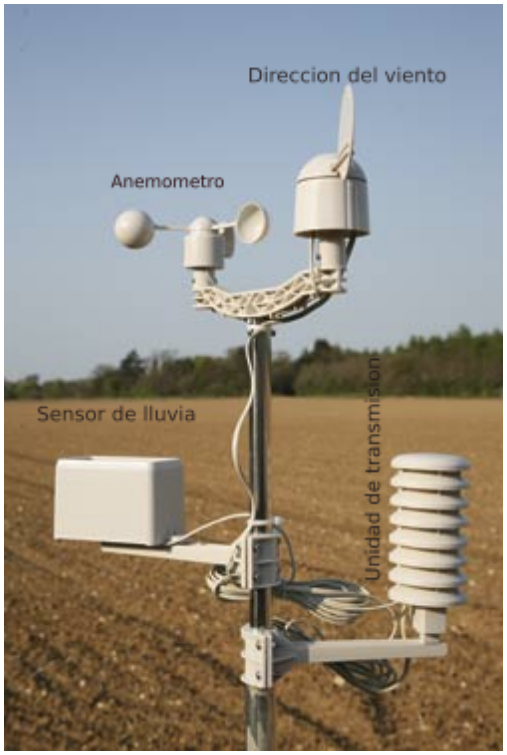

Figura 1: Estación meteorológica en Campo

En este trabajo nos centraremos en el desarrollo del firmware[1]de dos de ellos: Un sensor de velocidad del viento y un sensor de direccion de viento. Este ultimo ademas posee una intefaz visibles con leds.

[1] `https://es.wikipedia.org/wiki/Firmware`

2 Descripción del sistema

En el siguiente diagrama podemos ver como se relaciona cada componente del sistema:

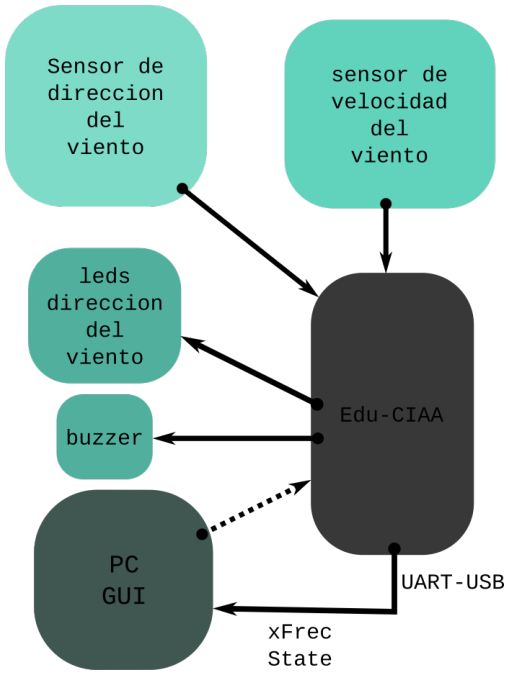

Figura 2: Diagrama esquemático del sistema

- Sensor de dirección del viento: Consta de una veleta que gira al soplar el viento, en su interior posse 8 sensores magnéticos[2], un imán que gira solidario a la veleta y un circuito (divisor resistivo) que entrega una sálida analógica.

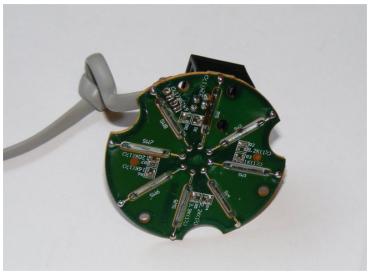

Figura 3: Sensor de dirección del viento por dentro (circuito interno)

[2] https://en.wikipedia.org/wiki/Reed_switch

Figura 4: Sensor de dirección del viento por dentro despiezado

- Sensor de velocidad del viento: Consta de un simple sensor magnético y un imán como el de el anterior sensor que entrega valores de alto o bajo que pueden ser leidos como entrada digital en la placa.

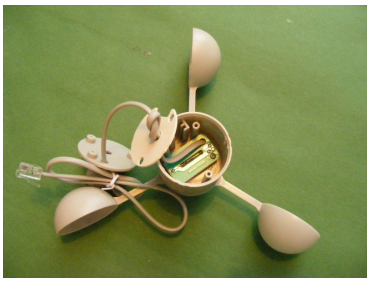

Figura 5: Sensor de velocidad del viento por dentro

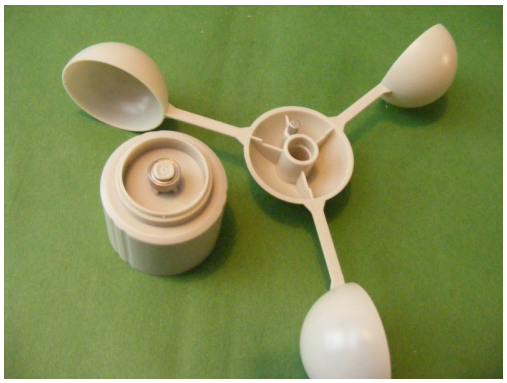

Figura 6: Sensor de velocidad del viento despiezado

- Leds de direccion del viento: Con esta interfaz podemos visualizar en que estado esta el sensor de dirección del viento

Figura 7: Interfaz grafica del sensor de direccion del viento a base de leds

- Buzzer: Se encarga de emitir una alarma cuando un evento ha ocurrido (ver la seccion 3 para mas detalles).
- Interfaz Grafica de Usuario: Se encarga de mostrar los valores de velocidad y direccion del viento que le fueron entregados por puerto serie (UART)
- Edu-CIAA[3]: Es una placa de desarrollo de Hardware y software abierto, la cual cuenta con las siguientes caracteristicas técnicas:

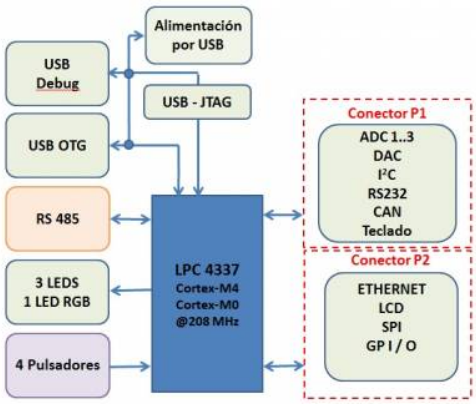

Figura 8: Caracteristicas técnicas de la Edu-CIAA

[3] http://proyecto-ciaa.com.ar/

Figura 9: Edu-CIAA-NXP

3 Diseño del Firmware

Las especificaciones que debe realizar nuestro sistema son:

- Calcular la velocidad del viento mediante la lectura del sensor correspondiente
- Calcular la direccion del viento mediante la lectura del sensor correspondiente
- Encender los leds correspondientes de acuerdo a la dirección calculada
- Activar una alarma en caso de que la velocidad del viento sea mas elevada que un umbral elegido
- Mostrar los resultados mediante una GUI(*Graphical User Interface*) en una PC conectada a la placa

Para el desarrollo del firmware se utlizaron dos Componentes principales:

- FreeRTOS™[4]: Es el Sistema operativo de tiempo real mas utilizado en el mundo de los sistemas embebidos
  - En FreeRTOS™cada hilo de ejecucion se llama task(tarea)
  - El FreeRTOS™para LPC17xx incluye todas las funcionalidades del standard
  - Podemos operar en modo *pre-emptive* y en modo *co-operative*
  - Asignacion de prioridad de tareas flexible
  - Colas(*Queues*)
  - Semaforos binarios
  - Semaforos contadores
  - Semaforos recursivos
  - *mutexes*
  - Funciones *Tick hook*
  - Funciones *Idle hook*
  - Chequeo de *Stackoverflow*
  - macros *Trace hook*
- Libreria de abtracción *HAL* sAPI[5] Nos permite interactuar con la placa de desarrollo a un nivel de abstraccion mas alto, esto hace que el desarrollo de prototipos como es este proyecto sea mas rapido

[4] `http://www.freertos.org/`
[5] `https://github.com/epernia/sAPI`

Ya que utilizamos FreeRTOS™nuestro disenio se basa en programar las tareas(*tasks*), como interactúan entre ellas. Para poder visualizar mejor estas relaciones tenemos el siguiente gráfico:

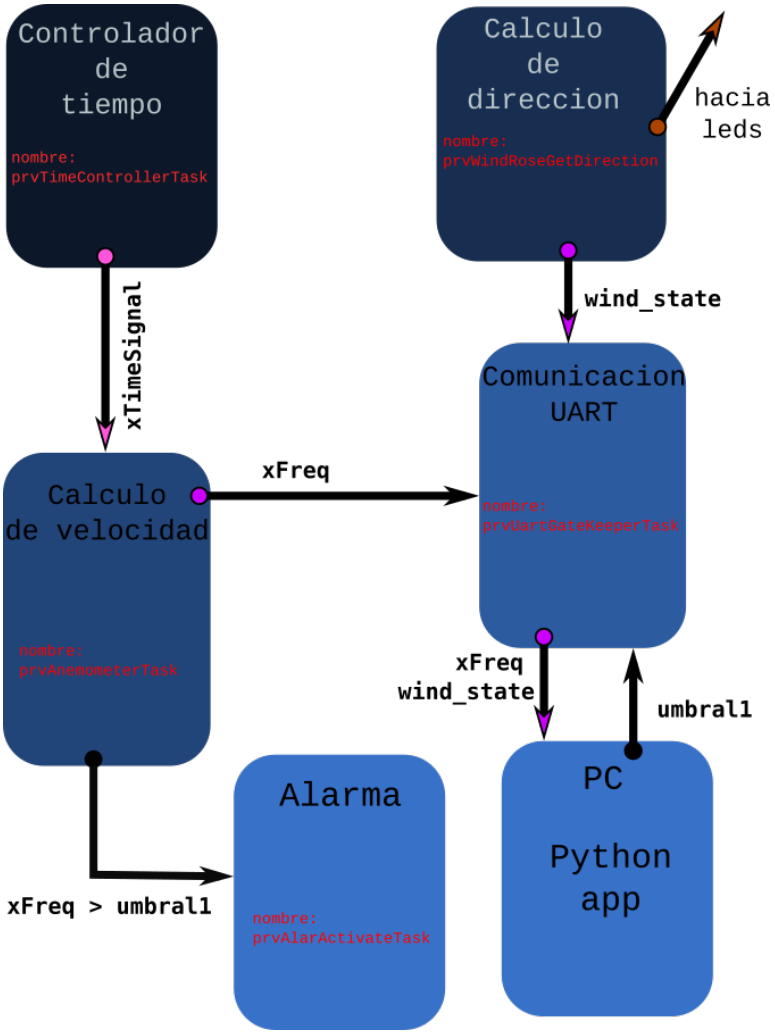

Figura 10: Diagrama de relacion de tareas

Pasamos a detallar cada una de las tareas:

- `prvTimeControllerTask`: Esta tarea se encarga de llevar una cuenta precisa del tiempo(ya que utiliza un delay exacto provisto por FreeRTOS™`vTaskDelayUntil`[6]) una vez que ha llegado el tiempo(1 segundo) envia una señal a la tarea que esta calculando la frecuencia con que ha girado el sensor de velocidad del viento. Código:

```
void prvTimeControllerTask(void *pvParameters)
{
   portTickType xLastWakeTime;
   xLastWakeTime = xTaskGetTickCount();
   while(1)
   {
      vTaskDelayUntil( &xLastWakeTime, (SIGNAL_MESSAGE_PERIOD / portTICK_RATE_MS));
      xSemaphoreGive(xTimeSignal);
```

[6] `http://www.freertos.org/vtaskdelayuntil.html`

```
    }

}
```

- `prvAnemometerTask`: Esta tarea es la encargada de calcular la frecuencia de giro del sensor del viento. Cada segundo le llega un mensage de tiempo ahi es cuando envia lo calculado a la tarea que se encarga de administrar el periférico UART, antes de volver a empezar *resetea* los valores calcualdos, ademas verifica si el valor calculado en mayor que un umbral, en caso de serlo envía una señal a una tarea de alarma. Código:

```
void prvAnemometerTaks(void *pvParameters)
{
   /* Anemometer pin states */
   typedef enum{PIN_UP, PIN_FALLING, PIN_DOWN, PIN_RISING} pin_state_t;
   /* Auxiliar variables */
   portBASE_TYPE xFreq = 0;
   portBASE_TYPE xCounter = 0;
   portBASE_TYPE xTemp = 0;
   /* initial condition */
   pin_state_t pin_state = PIN_UP;
   /* message data */
   xMetaData xAnemometerMessage;
   /* message flag for the Gatekeeper */
   xAnemometerMessage.xSource = SENDER_ANEMOMETER;
   /* Task processig */
   while(1)
   {

      /* MEF for counting the states changes */
      switch(pin_state)
      {
         case PIN_UP:
            {
               if(!digitalRead(DIO32))
               {
                  pin_state = PIN_FALLING;
               }
               break;
            }
         case PIN_FALLING:
            {
               xCounter += 3;
               if(!digitalRead(DIO32))
               {
                  pin_state = PIN_DOWN;
                  xFreq++;
                  digitalWrite(LEDR, ON);
               }
               else
                  pin_state = PIN_UP;
               break;
            }
         case PIN_DOWN:
               {
               if(digitalRead(DIO32))
               {
```

```
                        pin_state = PIN_RISING;
                    }
                    break;
                }
            case PIN_RISING:
                {
                    xCounter += 3;
                    if(digitalRead(DIO32))
                    {
                        pin_state = PIN_UP;
                        digitalWrite(LEDR, OFF);
                    }
                    else
                    {
                        pin_state = PIN_DOWN;
                    }
                    break;
                 }
            }

        if(xSemaphoreTake(xTimeSignal, ( TickType_t )0))
        {
            if(xFreq > FREQUENCY_ALARM_THRESHOLD_1)
            {
                xTemp = ALARM_MESSAGE_1;
                xQueueSendToBack(xALARMQueue, (void *)&xTemp, portMAX_DELAY);
            }
            if(xFreq > FREQUENCY_ALARM_THRESHOLD_2)
            {
                xTemp = ALARM_MESSAGE_2;
                xQueueSendToBack(xALARMQueue, (void *)&xTemp, portMAX_DELAY);
            }
            /* The time message arrive --> prepare the message package */
            xAnemometerMessage.xMessage = xFreq;
            /* send the package via the Gatekeeper */
            xQueueSendToBack(xUARTQueue, (void *)&xAnemometerMessage, ( TickType_t )0);
            /* reset the values */
            xFreq = 0;
            xCounter = 0;
        }
    }

}
```

- `vUartGateKeeperTask`: Esta tarea se encarga de administrar el periférico UART mediante una cola que solo tiene el poder de enviar datos al periférico por ella, además por como fueron diseñados los datos que maneja, estos pueden provenir de distintas fuentes. Asi *Gatekeeper* puede disernir de donde provienen los datos y actuar en consecuencia. Ademas envía los datos a la PC de tal manera que sean facilmente *parseables* por la aplicacion que los recibe. Código:

```
void vUartGatekeeperTask( void *pvParameters )
{
    xMetaData xReceived;
    portBASE_TYPE xStatus;
```

```
        while(1)
        {
           /* Wait for a message to arrive. */
           xStatus = xQueueReceive(xUARTQueue, &xReceived, portMAX_DELAY);

           if(xStatus == pdPASS)
           {
              switch(xReceived.xSource)
              {
                 case SENDER_ANEMOMETER:
                    {
                       vPrintStringAndNumber("Freq:", xReceived.xMessage);
                       break;
                    }
                 case SENDER_WIND_ROSE:
                    {
                       vPrintStringAndNumber("State:", xReceived.xMessage);
                       break;
                    }
                 /* case SENDER_PC: */
                 /*    { */
                 /*       break; */
                 /*    } */
              }
           }
           else
           {
              vPrintString("Error!!!");
           }

        }
}
```

- `prvWindRoseGetDirection`: Esta tarea se encarga de leer los datos analógicos del sensor de dirección de viento y asigna de acuerdo a una tabla dada por el fabricante el estado en que se encuentra el sensor. Además una vez que ha detectado el estado en cual se encuentra envía ese dato a una función auxiliar que mapea estados con pines, de acuerdo al fabricante de el visor de leds. Código:

```
void prvWindRoseGetDirection(void *pvParameters)
{
   /* initial state */
   wind_states_t wind_states = N;
   /* task metadata message for the Gatekeeper */
   xMetaData xWindRoseMessage;
   xWindRoseMessage.xSource = SENDER_WIND_ROSE;
   portBASE_TYPE uartBuff[10];
   portBASE_TYPE sample = 0;

   while(1)
   {
      vTaskDelay(WIND_ROSE_POOLING_PERIOD / portTICK_RATE_MS);
      /* read the analog sensor value */
      sample = analogRead(AI0);
      /* convert the sample to decimal number */
      itoa(sample, uartBuff, 10);
```

```
/* N */
if(CHECK(sample, N_MIN, N_MAX))
{
   wind_states = N;
   do_state(wind_states);
}
/* NNO */
if(CHECK(sample, NNO_MIN, NNO_MAX))
{
   wind_states = NNO;
   do_state(wind_states);
}
/* NO */
if(CHECK(sample, NO_MIN, NO_MAX))
{
   wind_states = NO;
   do_state(wind_states);
}
/* NOO */
if(CHECK(sample, NOO_MIN, NOO_MAX))
{
   wind_states = NOO;
   do_state(wind_states);
}
/* O */
if(CHECK(sample, O_MIN, O_MAX))
{
   wind_states = O;
   do_state(wind_states);
}
/* SOO */
if(CHECK(sample, SOO_MIN, SOO_MAX))
{
   wind_states = SOO;
   do_state(wind_states);
}
/* SO */
if(CHECK(sample, SO_MIN, SO_MAX))
{
   wind_states = SO;
   do_state(wind_states);
}
/* SSO */
if(CHECK(sample, SSO_MIN, SSO_MAX))
{
   wind_states = SSO;
   do_state(wind_states);
}
/* S */
if(CHECK(sample, S_MIN, S_MAX))
{
   wind_states = S;
   do_state(wind_states);
}
```

```
            /* SSE */
            if(CHECK(sample, SSE_MIN, SSE_MAX))
            {
               wind_states = SSE;
               do_state(wind_states);
            }
            /* SE */
            if(CHECK(sample, SE_MIN, SE_MAX))
            {
               wind_states = SE;
               do_state(wind_states);
            }
            /* SEE */
            if(CHECK(sample, SEE_MIN, SEE_MAX))
            {
               wind_states = SEE;
               do_state(wind_states);
            }
            /* E */
            if(CHECK(sample, E_MIN, E_MAX))
            {
               wind_states = E;
               do_state(wind_states);
            }
            /* NEE */
            if(CHECK(sample, NEE_MIN, NEE_MAX))
            {
               wind_states = NEE;
               do_state(wind_states);
            }
            /* NE */
            if(CHECK(sample, NE_MIN, NE_MAX))
            {
               wind_states = NE;
               do_state(wind_states);
            }
            /* NNE */
            if(CHECK(sample, NNE_MIN, NNE_MAX))
            {
               wind_states = NNE;
               do_state(wind_states);
            }

            xWindRoseMessage.xMessage = wind_states;
            xQueueSendToBack(xUARTQueue, (void *)&xWindRoseMessage, ( TickType_t )0);

         }
      }
```

- `prvAlarmActivateTask`: Esta tarea es una alarma que espera a que sea puesto en la cola un mensage de alarma(por la tarea `prvAnemometerTask`) por el momento pueden *setearse* dos niveles de umbral(y por consiguiente dos mensages que entiende cuando son recibidos). Al recibir alguno de los mensages activa un *buzzer* con distintas alarmas para cada uno. Código:

```
prvAlarmActivateTask(void *pvParameters)
```

```
{
   portBASE_TYPE xTemp;

   while(1)
   {
      vTaskDelay(ALARM_POOLING_PERIOD / portTICK_RATE_MS);
      xQueueReceive(xALARMQueue, &xTemp,  portMAX_DELAY);
      if(xTemp == ALARM_MESSAGE_1)
      {
         digitalWrite(DIO11, ON);
         vTaskDelay(ALARM_BEEP_WARNIG_PERIOD / portTICK_RATE_MS);
         digitalWrite(DIO11, OFF);
      }
      if(xTemp == ALARM_MESSAGE_2)
      {
         digitalWrite(DIO11, ON);
         vTaskDelay(ALARM_BEEP_DESASTER / portTICK_RATE_MS);
         digitalWrite(DIO11, OFF);
         digitalWrite(DIO11, ON);
         vTaskDelay(ALARM_BEEP_DESASTER / portTICK_RATE_MS);
         digitalWrite(DIO11, OFF);
      }
   }


}
```

- GUI: Se programo una aplicación en Python con la libreria Kivy[7], la cual recibe los datos de la UART y de acuerdo a que estado y frecuencia que recibio realiza:

```
from kivy.app import App
from kivy.uix.widget import Widget
from kivy.graphics import Color, Line
from kivy.uix.floatlayout import FloatLayout
#from math import cos, sin, pi
from numpy import cos, sin, pi
from kivy.clock import Clock
from kivy.lang import Builder
from kivy.properties import NumericProperty
import serial
import time
from threading import Thread
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.label import Label
from kivy.properties import ObjectProperty, StringProperty
import datetime
import re

kv = '''
#:import math math

[AnemometerNumber@Label]:
```

[7] https://kivy.org/

```
        text: ctx.d
        pos_hint: {"center_x": 0.5+0.42*math.sin(math.pi/8*(ctx.i-16)), "center_y": 0.5+0.42*mat
        font_size: self.height/16

    <Box_a>:
        Label:
            id: mylabel
            text: "hello, world"

    <MyAnemometerWidget>:
        face: face
        ticks: ticks
        FloatLayout:
            id: face
            size_hint: None, None
            pos_hint: {"center_x":0.5, "center_y":0.5}
            size: 0.9*min(root.size), 0.9*min(root.size)
            canvas:
                Color:
                    rgba: 1, 1, 1, .8
                Ellipse:
                    size: self.size
                    pos: self.pos
                    source:'/home/elsuizo/images/Reinel_wind_rose_round.png'
            AnemometerNumber:
                i: 1
                d: "N"
            AnemometerNumber:
                i: 2
                d: "NNO"
            AnemometerNumber:
                i: 3
                d: "NO"
            AnemometerNumber:
                i: 4
                d: "NOO"
            AnemometerNumber:
                i: 5
                d: "O"
            AnemometerNumber:
                i: 6
                d: "SOO"
            AnemometerNumber:
                i: 7
                d: "SO"
            AnemometerNumber:
                i: 8
                d: "SSO"
            AnemometerNumber:
                i: 9
                d: "S"
            AnemometerNumber:
                i: 10
                d: "SSE"
```

```
            AnemometerNumber:
                i: 11
                d: "SE"
            AnemometerNumber:
                i: 12
                d: "SEE"
            AnemometerNumber:
                i: 13
                d: "E"
            AnemometerNumber:
                i: 14
                d: "NEE"
            AnemometerNumber:
                i: 15
                d: "NE"
            AnemometerNumber:
                i: 16
                d: "NNE"
        Ticks:
            id: ticks
            r: min(root.size)*0.9/2
    '''
    Builder.load_string(kv)

    class MyAnemometerWidget(FloatLayout):
        pass

    class Ticks(Widget):
        def __init__(self, **kwargs):
            super(Ticks, self).__init__(**kwargs)
            self.bind(pos=self.update_anemometer)
            self.bind(size=self.update_anemometer)

        def read_serial(self, *args):

            '''Serial communications: get a response'''

            # open serial port
            try:
                serial_port = serial.Serial('/dev/ttyUSB1',115200)
                #serial_port = serial.Serial(com_port, baudrate=115200, timeout=1)
            except serial.SerialException as e:
                print("could not open serial port '{}': {}".format(com_port, e))

            # read response from serial port
            lines = []
            while True:
                line = serial_port.readline()
                lines.append(line.decode('utf-8').rstrip())

                # wait for new data after each line
                timeout = time.time() + 0.05
                while not serial_port.inWaiting() and timeout > time.time():
                    pass
```

```
                if not serial_port.inWaiting():
                    break

            #close the serial port
            serial_port.close()
            return lines
        def update_anemometer(self, *args):
            self.canvas.clear()
            with self.canvas:
                Color(1, 1, 0)
                #th = time.hour*60 + time.minute
                #lines = self.read_serial()
                state = 0
                for num in lines:
                    if num[0] == 'S':
                        state = re.findall(r'\d+', num)
                        state = int(state[-1])
                        print 'The state is ', state
                    if num[0] == 'F':
                        freq = re.findall(r'\d+', num)
                        freq = int(freq[-1])
                        print 'The freq is ', freq

                Line(points=[self.center_x, self.center_y, self.center_x+1*self.r*cos(2*pi/state

class MyAnemometerApp(App):
    def build(self):
        clock = MyAnemometerWidget()
        Clock.schedule_interval(clock.ticks.update_anemometer, 1)
        return clock

if __name__ == '__main__':
    MyAnemometerApp().run()
```

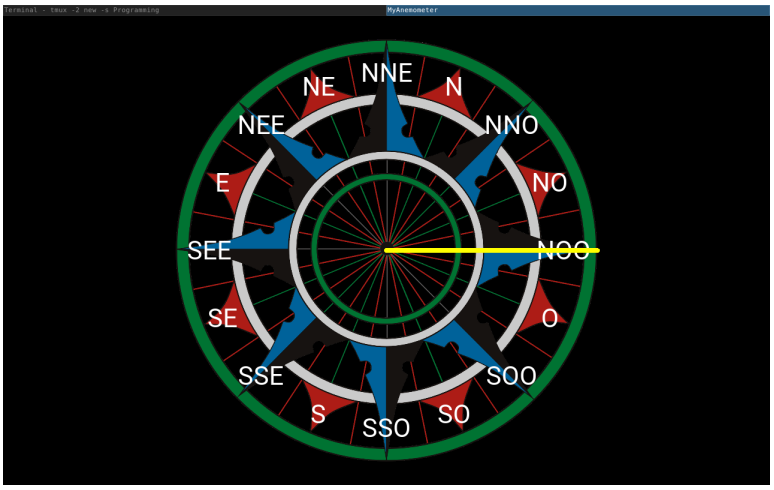

Figura 11: GUI para visualizar el estado del viento

- Calcula la velocidad del viento en Km/h. Ya que recibe la frecuencia por segundo en que esta girando el sensor de velocidad puede hacer una simple multiplicacion y convertir los datos, la constante de conversion

es: $\frac{1tick}{seg} \approx 2{,}4\frac{Km}{h}$

- Actualiza la pantalla con los datos del estado en que se encuentra el sensor de direccion del viento.

4 Resultados

Se logro el objetivo principal que era tener un firmware que adquiriera la información de los sensores en tiempo real. El sistema es flexible en al agregado de nuevos sensores ya que se ha modularizado el codigo de tal manera que sea facil la inclusion de nuevos componentes.

5 Trabajos a futuro

Como trabajos a futuro o cosas para mejorar de la implementación:

- Implementar la cuenta de *Ticks* con una interrupción, la cual solo tendría que actualizar un contador (`xFrec++`) y cuando le sea enviada la senial de tiempo activar un mensage a la tarea de *Gatekeeper*
- Permitir al usuario a traves de la GUI ingresar el/los valores umbrales de alarma.
- Hacer que la conexión entre los sensores y la placa sea remota
- Hacer un *dataloger* con los datos obtenidos.

Referencias

[1] Using The FreeRTOS™Real Time Kernel. Richard Barry

[2] Real Time Systems and Design. Phillip Laplante