
Sistemas Digitales
Trabajo Final:
Desarrollo de Firmware para estacion meteorologica

MARTÍN NOBLÍA

PROFESORES:

JOSE JUAREZ
ERIC PERNIA



Universidad Nacional de Quilmes

1

Introduccion

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

- Temperaturas
- Direccion 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 climaticas.

- Viniedos
- Granjas
- Campos de agricultura

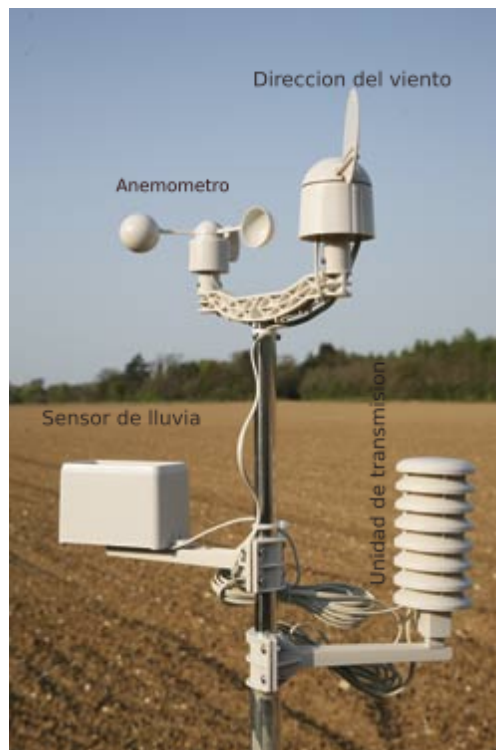


Figura 1: Estacion meteorologica en Campo

En este trabajo nos centraremos en el desarrollo del firmware¹ de dos de ellos: Un sensor de velocidad del viento y un sensor de direccion de viento. Este ultimo ademas posee una interfaz visible 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:

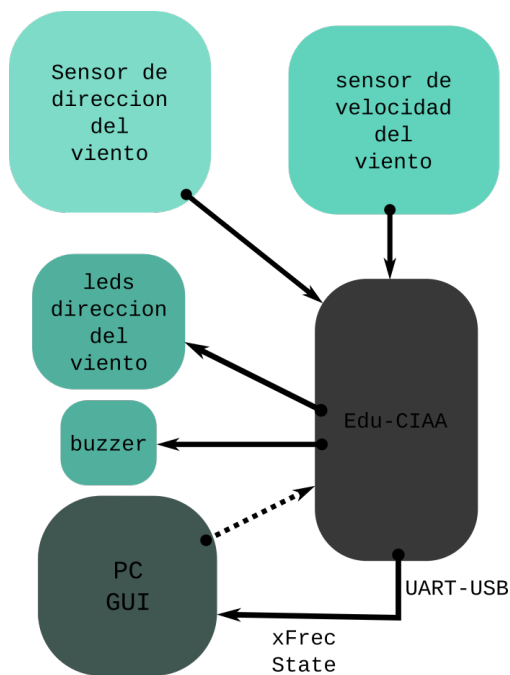


Figura 2: Diagrama esquemático del sistema

- Sensor de dirección del viento: Consta de una veleta que gira al soplar el viento, en su interior posee 8 sensores magnéticos², un imán que gira solidario a la veleta y un circuito (divisor resistivo) que entrega una salida analógica.

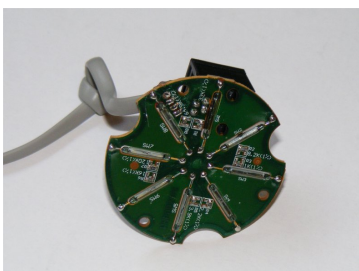


Figura 3: Sensor de dirección del viento por dentro (circuito interno)

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



Figura 4: Sensor de direccion del viento por dentro despiezado

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

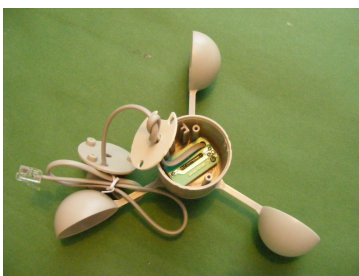


Figura 5: Sensor de velocidad del viento por dentro



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 direccion 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-CIAA³: Es una placa de desarrollo de Hardware y software abierto, la cual cuenta con las siguientes características técnicas:

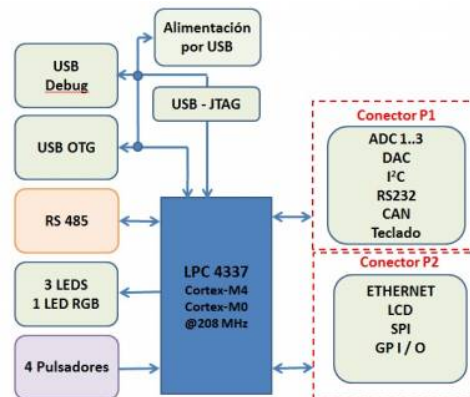


Figura 8: Características técnicas de la Edu-CIAA

³<http://proyecto-ciaa.com.ar/>

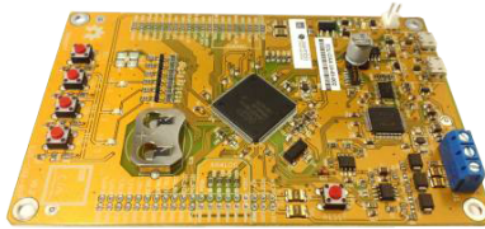


Figura 9: Edu-CIAA-NXP

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 dirección 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 más 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 utilizaron dos Componentes principales:

- FreeRTOS^{TM4}: Es el Sistema operativo de tiempo real más utilizado en el mundo de los sistemas embebidos
 - En FreeRTOSTM cada hilo de ejecución se llama task (tarea)
 - El FreeRTOSTM para LPC17xx incluye todas las funcionalidades del standard
 - Podemos operar en modo *pre-emptive* y en modo **co-operative**
 - Asignación 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*
- Librería de abstracción HAL sAPI⁵ Nos permite interactuar con la placa de desarrollo a un nivel de abstracción más alto, esto hace que el desarrollo de prototipos como es este proyecto sea más rápido

⁴<http://www.freertos.org/>

⁵<https://github.com/epernia/sAPI>

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

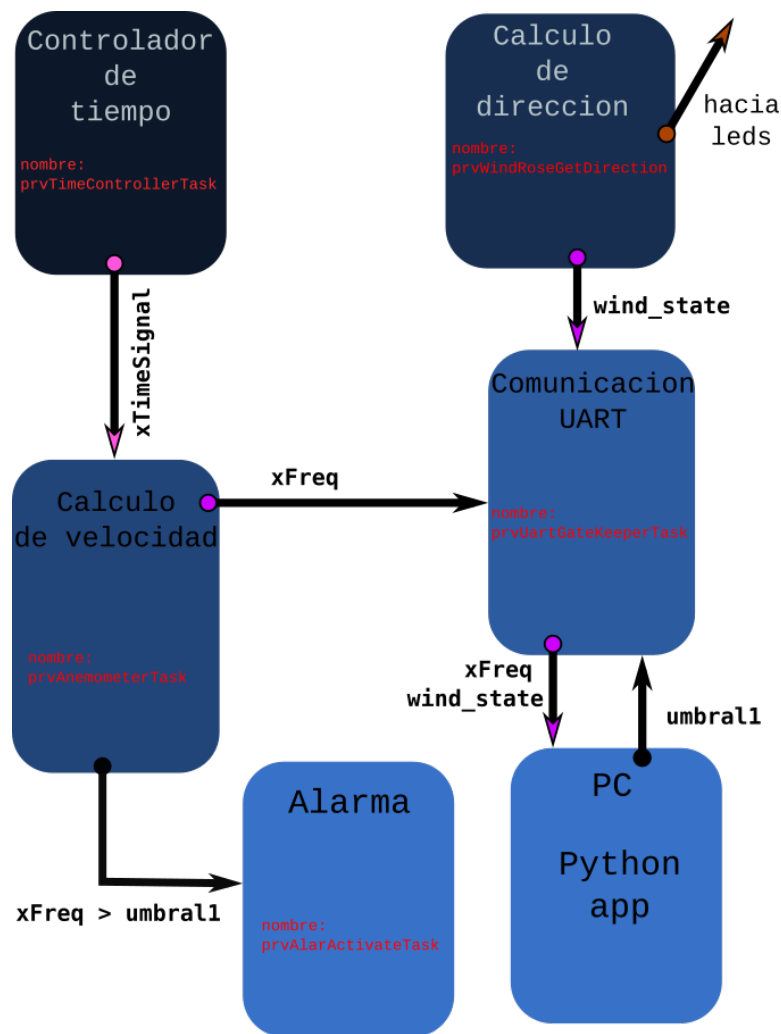


Figura 10: Diagrama de relacion de tareas

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) envía una señal a la tarea que está 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 mensaje de tiempo ahí es cuando envía lo calculado a la tarea que se encarga de administrar el periférico UART, antes de volver a empezar *resetea* los valores calculados, además verifica si el valor calculado es 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 processing */
    while(1)
    {

        /* MEF for counting the states changes */
        switch(pin_state)
        {
            case PIN_UP:
            {
                if(!digitalRead(DI032))
                {
                    pin_state = PIN_FALLING;
                }
                break;
            }
            case PIN_FALLING:
            {
                xCounter += 3;
                if(!digitalRead(DI032))
                {
                    pin_state = PIN_DOWN;
                    xFreq++;
                    digitalWrite(LED_R, ON);
                }
                else
                {
                    pin_state = PIN_UP;
                }
                break;
            }
            case PIN_DOWN:
            {
                if(digitalRead(DI032))
                {

```



```

        pin_state = PIN_RISING;
    }
    break;
}
case PIN_RISING:
{
    xCounter += 3;
    if(digitalRead(DI032))
    {
        pin_state = PIN_UP;
        digitalWrite(LED1, 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. Así *Gatekeeper* puede discernir de donde provienen los datos y actuar en consecuencia. Además envía los datos a la PC de tal manera que sean fácilmente *parseables* por la aplicación 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 analogicos del sensor de direccion de viento y asigna de acuerdo a una tabla dada por el fabricante el estado en que se encuentra el sensor. Ademas una vez que ha detectado el estado en cual se encuentra envia ese dato a una funcion auxiliar que mapea estados con pines, de acuerdo al fabricante de el visor de leds.Codigo:

```

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 mensaje de alarma(por la tarea **prvAnemometerTask**) por el momento pueden *setearse* dos niveles de umbral(y por consiguiente dos mensajes que entiende cuando son recibidos). Al recibir alguno de los mensajes 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 aplicacion en Python con la libreria Kivy⁷, la cual recibe los datos de la UART y de acuerdo a que estado y frecuencia que recibio realiza:
 - 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 informacion 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 implementacion:

- Implementar la cuenta de *Ticks* con una interrupcion, la cual solo tendria que actualizar un contador (`xFrec++`) y cuando le sea enviada la senial de tiempo activar un mensaje a la tarea de *Gatekeeper*
- Permitir al usuario a traves de la GUI ingresar el/los valores umbrales de alarma.

⁷<https://kivy.org/>

- 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