Trabajo Práctico N°03

Registrador de datos de temperatura y humedad ambiente

Circuitos Digitales y Microcontroladores – UNLP

NOTA 10

CALDERÓN Sergio Leandro

4 de julio de 2022

Índice

1 – Conexionado

1.1 – Enunciado

Implementar un registrador de datos de temperatura y humedad ambiente a partir del sensor digital DHT11 y el periférico de comunicación serie.

1.2 – Resolución

El microcontrolador ATMega328p incluye recursos para comunicarse de manera serial con dispositivos externos, empleando interfaces estándares, uno de ellos es el Transmisor-Receptor Serial Universal Síncrono y Asíncrono (USART en inglés).

La USART incluye recursos independientes para la transmisión y la recepción de datos, lo que posibilita operar full-duplex, permitiendo enviar y recibir datos en forma simultánea. La terminal destinada para la transmisión de datos es TXD (corresponde al pin PD1 en el kit de desarrollo) y para la recepción es RXD (corresponde al pin PD0 en el kit de desarrollo).

Respecto al periférico DHT11, la comunicación entre el MCU y el sensor se realiza mediante una línea única de datos bidireccional “DATA”, como se verá en el apartado 5.

A continuación, se muestra el esquema de conexionado del ATMega328p con la terminal serie y el sensor DHT11, utilizado para el presente proyecto.

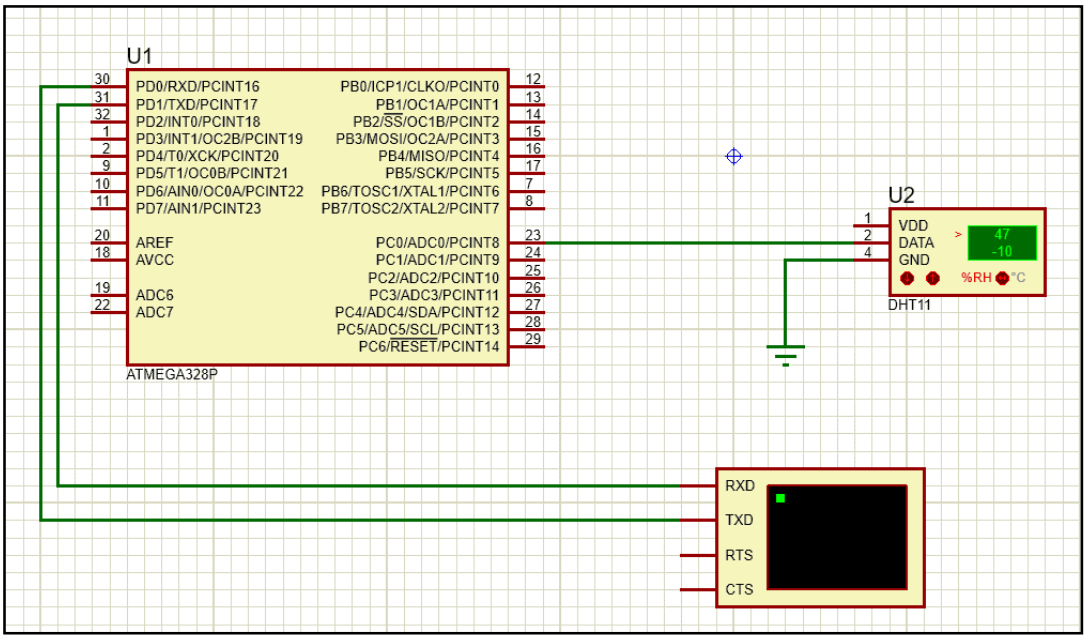


Fig. 1.1. Conexionado final en el simulador Proteus.

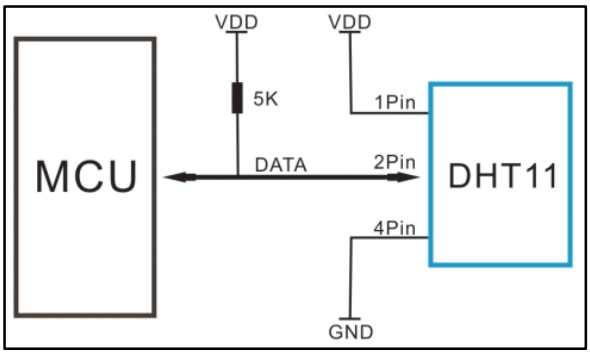


Fig. 1.2. Aplicación típica del sensor DHT11, de acuerdo a su hoja de datos.

2 – Arquitectura

2.1 – Enunciado

La arquitectura deberá ser del tipo Background/Foreground y además se deberá aplicar modularización, abstracción y el modelo productor-consumidor para el periférico UART, es decir, las funciones para el manejo del periférico se deberán implementar, en una biblioteca, usando las interrupciones de Transmisión y Recepción con sus respectivos buffers.

2.2 – Interpretación

Para el programa principal se deberá utilizar la arquitectura *Background/Foreground*, es decir, en el *Loop* se deberá preguntar por *Flags* de eventos y realizar la tarea asociada si corresponde (Background); por otra parte, en las rutinas de servicio de interrupción se deberá activar el Flag correspondiente al evento ocurrido (Foreground).

Respecto al periférico UART, se deberá crear un archivo o biblioteca para modularizar el mismo, creando funciones públicas que permitan ejecutar una serie de instrucciones sin la necesidad de indicar explícitamente cuáles son los registros y bits a escribir o leer.

Además, se deberá aplicar el modelo de productor-consumidor para la recepción y la transmisión de datos, mediante 2 buffers (uno para cada operación) con sus índices de lectura y escritura para que un productor agregue datos y un consumidor los retire del mismo.

2.3 – Resolución

2.3.1 – Biblioteca SerialPort

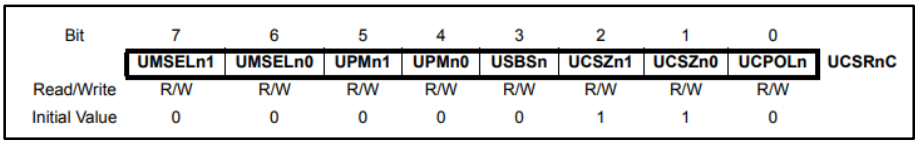
El archivo implementado para el periférico UART se denomina “SerialPort”, e incluye las funciones descritas a continuación, todas con prefijo SerialPort en su nombre.

Tabla 2.1. Funciones públicas de la biblioteca SerialPort

Nombre	Parámetros	Acción	Retorna
Init	f_cpu	Inicializa el periférico a 9600 baudios, para la frecuencia de CPU especificada	-
TX_Enable	-	Habilita el transmisor	-
TX_Interrupt_Enable	-	Habilita las interrupciones de transmisión (libre)	-
TX_Interrupt_Disable	-	Deshabilita las interrupciones de transmisión (libre)	-
RX_Enable	-	Habilita el receptor	-
RX_Interrupt_Enable	-	Habilita las interrupciones de recepción (hay dato)	-
Send_Data	dato	Escribe el dato pasado por parámetro en UDR0	-
Receive_Data	-	Retorna el dato actual en UDR0	dato

2.3.1.1 – Inicialización

En la función de inicialización, primero se limpia el registro UCSR0B para deshabilitar el receptor, transmisor y las interrupciones. Luego, se define el tamaño de dato para cada *frame* recibido y transmitido, mediante los bits el registro UCSR0C.



Bit	7	6	5	4	3	2	1	0	
	UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn	UCSRnC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	1	1	0	

Fig. 2.1. Bits del registro UCSR0C del periférico UART.

Para un tamaño de dato de 8 bits se escribe un “1” en los bits UCSZ00 y UCSZ01, como se puede observar en la tabla a continuación (cap. 19, sección 10):

Tabla 2.2. Determinación del tamaño de dato según bits UCSZ00 y UCSZ01.

UCSZn1	UCSZn0	Character Size
0	0	5-bit
0	1	6-bit
1	0	7-bit
1	1	8-bit

Por último, se configura el periférico con una tasa de baudios de 9600 bps, escribiendo los bits correspondientes en el registro UBRR0L según la frecuencia de CPU especificada. En este caso, para 16 MHz se debe escribir el valor 103, según la tabla a continuación:

Tabla 2.3. Valor del registro UBRR0 para 16 MHz, según la tasa de baudios.

	f_{osc} = 16.0000MHz			
	U2Xn = 0		U2Xn = 1	
Baud Rate (bps)	UBRRn	Error	UBRRn	Error
2400	416	–0.1%	832	0.0%
4800	207	0.2%	416	–0.1%
9600	103	0.2%	207	0.2%
14.4k	68	0.6%	138	–0.1%
19.2k	51	0.2%	103	0.2%
28.8k	34	–0.8%	68	0.6%
38.4k	25	0.2%	51	0.2%
57.6k	16	2.1%	34	–0.8%
76.8k	12	0.2%	25	0.2%
115.2k	8	–3.5%	16	2.1%
230.4k	3	8.5%	8	–3.5%

El pseudocódigo de la función descrita es el siguiente:

Limpiar registro UCSR0B

Activar bits UCSZ00 y UCSZ01 de UCSR0C

Si la frecuencia de CPU es 16 MHz

    Escribir 103 en UBRR0L

Sino si la frecuencia de CPU es 8 MHz

    Escribir 51 en UBRR0L

Sino si la frecuencia de CPU es 4 MHz

    Escribir 25 en UBRR0L

Pseudocódigo 2.1. Función pública SerialPort_Init.

2.3.1.2 – Transmisor y receptor

Como se mencionó en la sección anterior, la habilitación del transmisor y del receptor se realiza mediante la activación de los bits correspondientes en el registro UCSR0B:

Bit	7	6	5	4	3	2	1	0	
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n	UCSRnB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 2.2. Bits del registro UCSR0B del periférico UART.

Los bits TXEN0 y RXEN0 habilitan el transmisor y receptor, respectivamente.

2.3.1.3 – Interrupciones

Las interrupciones de recepción y transmisión también se controlan mediante el registro UCSR0B. Cuando se completa la recepción de un dato se genera una interrupción “USART RX” si el bit RXCIE0 está activo. Por otra parte, cuando el registro de datos del periférico se vacía (UDR0) se genera una interrupción “USART UDRE” si el bit UDRIE0 está activo.

Para evitar que la interrupción de “libre para transmitir” suceda incluso cuando no se tienen datos disponibles para transmitir, se proporcionan funciones tanto para habilitar como deshabilitar la interrupción de UDR vacío. Respecto a RX, solo se requiere la habilitación.

2.3.1.4 – Lectura y escritura

Como se mostró en la tabla 2.1, la lectura del registro de datos UDR se realiza mediante la función *SerialPort_Receive_Data*, y la escritura inmediata con *SerialPort_Send_Data*

2.3.2 – Biblioteca Buffer

Para que el productor y el consumidor se abstraigan del uso y actualización de índices, se decidió crear una biblioteca dedicada al Buffer de recepción (RX) y transmisión (TX). Como la estructura de ambos es similar, se definió una estructura TipoBuffer.

```
// Estructura tipo Buffer
typedef struct {
    uint8_t* data;
    uint8_t readIndex;
    uint8_t writeIndex;
} TipoBuffer;
```

Fig. 2.3. Estructura TipoBuffer definida en la biblioteca Buffer

Como se observa en la Figura 2.3, un buffer está compuesto por un vector de un tamaño determinado, un índice de lectura y un índice de escritura. Cuando un productor debe agregar un dato a transmitir o recibido, se invoca a la función *Buffer_TX_Push* o *Buffer_RX_Push* respectivamente. El pseudocódigo genérico de las mismas se muestra a continuación.

Asignar dato en la posición de escritura

Incrementar índice de escritura

Pseudocódigo 2.2. Funciones públicas tipo Buffer_Push.

Por otra parte, cuando un consumidor debe retirar un dato recibido o para transmitir se invoca a la función *Buffer_RX_Pop* o *Buffer_TX_Pop* respectivamente.

```
Recuperar dato en la posición de lectura

Incrementar índice de lectura

Devolver dato
```

Pseudocódigo 2.3. Funciones públicas tipo Buffer_Pop.

Además, se agregan otras funciones que reutilizan las descritas anteriormente. Se hará referencia a las mismas al procesar comandos y al enviar mensajes a la terminal.

Tabla 2.4. Funciones no elementales de la biblioteca Buffer.

Nombre	Parámetros	Acción	Retorna
RX_PopString	texto maxLength	Recupera los datos del Buffer RX y los asigna en el texto pasado por referencia. Si no cabe en el mismo, se descartan los demás datos	-
TX_PushString	texto	Agrega los caracteres del texto pasado por parámetro al Buffer TX	-
TX_PushLine	texto	Ídem anterior, pero además agrega un salto de línea al final	-

Por último, también existe una función *Buffer_TX_Empty* que devuelve verdadero si los índices de lectura y escritura de TX coinciden, es decir, no hay más datos para transmitir.

2.3.3 – Productores

En la recepción de datos, el productor es la rutina de servicio de interrupción *ISR USART_RX_vect,* que se encarga de agregar cada dato (carácter) recibido al Buffer RX hasta que el usuario presione Enter, representado como el carácter '\r' (ASCII 13).

En la transmisión de datos, el productor es quien genera los mensajes de salida, que pueden ser el menú de opciones, la presentación de datos o un error. En este caso, los módulos de interfaz de usuario (GUI) y registrador de datos invocarán a las funciones provistas para agregar cadenas de caracteres al Buffer TX, como se verá en sus respectivos apartados.

A continuación, se muestra el pseudocódigo de la ISR de recepción.

```
Leer dato de SerialPort
Si el dato es '\r'
        Agregar un fin de cadena al Buffer RX
        Activar Flag de "Hay Comando"
Sino
        Agregar dato al Buffer RX
```

Pseudocódigo 2.4. Rutina de servicio de interrupción USART_RX.

2.3.4 – Consumidores

En la recepción de datos, el consumidor es la tarea procesadora de comandos, que retira los caracteres del Buffer RX para compararlos con un comando válido (ver apartado 4). En la transmisión, el consumidor es la rutina de servicio de interrupción *ISR USART_UDRE_vect*, que se encarga de retirar y enviar un dato del Buffer TX, mientras haya datos disponibles en el mismo. Si se acaban los datos, se deshabilitan las interrupciones de transmisión hasta que un productor genere un nuevo mensaje. El pseudocódigo de la ISR de transmisión es el siguiente:

```
Retirar dato del Buffer TX
Enviar dato a SerialPort
Si el Buffer TX está vacío
        Deshabilitar interrupciones de transmisión
```

Pseudocódigo 2.5. Rutina de servicio de interrupción USART_UDRE.

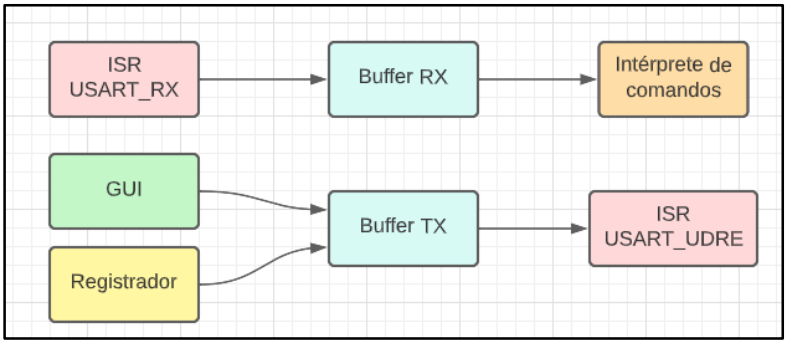


Fig. 2.4. Participantes del modelo Productor-Consumidor en el presente proyecto.

3 – Inicio del sistema

3.1 – Enunciado

Al iniciar, el sistema deberá presentar en la pantalla de una terminal serie el menú de opciones para que el usuario comande el registrador. Por ejemplo, el menú puede ser:

- *"Registrador de temperatura y humedad"*
- *Ingrese ON: para encender, OFF para apagar, RST para reiniciar*
- *>:*

3.2 – Interpretación

Una vez inicializados todos los componentes, el primer mensaje que deberá mostrarse por la pantalla de la terminal serie será un menú principal de opciones. Los caracteres que lo forman deberán agregarse al Buffer TX para que se envíen uno a uno al periférico UART.

3.3 – Resolución

3.3.1 – GUI

En este caso donde el sistema se controla desde una terminal se denomina "CLI"

Para el envío de todos los mensajes por pantalla, se decidió crear una biblioteca de interfaz de usuario "GUI", que provee 3 funciones públicas que agregan Strings al Buffer TX:

Tabla 3.1. Funciones de escritura de la biblioteca GUI.

Nombre	Parámetros	Acción	Retorna
Escribir_Menu	-	Escribe el menú de opciones por pantalla de forma completa	-
Escribir_Error	texto	Escribe el mensaje de texto pasado por parámetro, con un salto de línea	-
Escribir_Datos	h_ent, t_ent, t_dec	Escribe una línea de texto con datos de humedad y temperatura	-

De esta forma, el único componente que actúa con el Buffer TX y, por tanto, es el único productor directo de mensajes, es "GUI". Además, se encarga de habilitar las interrupciones de transmisión antes de finalizar cada función, quitando responsabilidad al registrador.

El menú mostrado por la pantalla de la terminal consta de 4 líneas, con un texto similar al del enunciado. Se agrega una línea para el nombre del programa, y luego tres más para indicar cada uno de los comandos disponibles, que se tratarán en su apartado correspondiente.

3.4 – Simulación

A continuación, se muestra la representación del menú en el simulador Proteus.

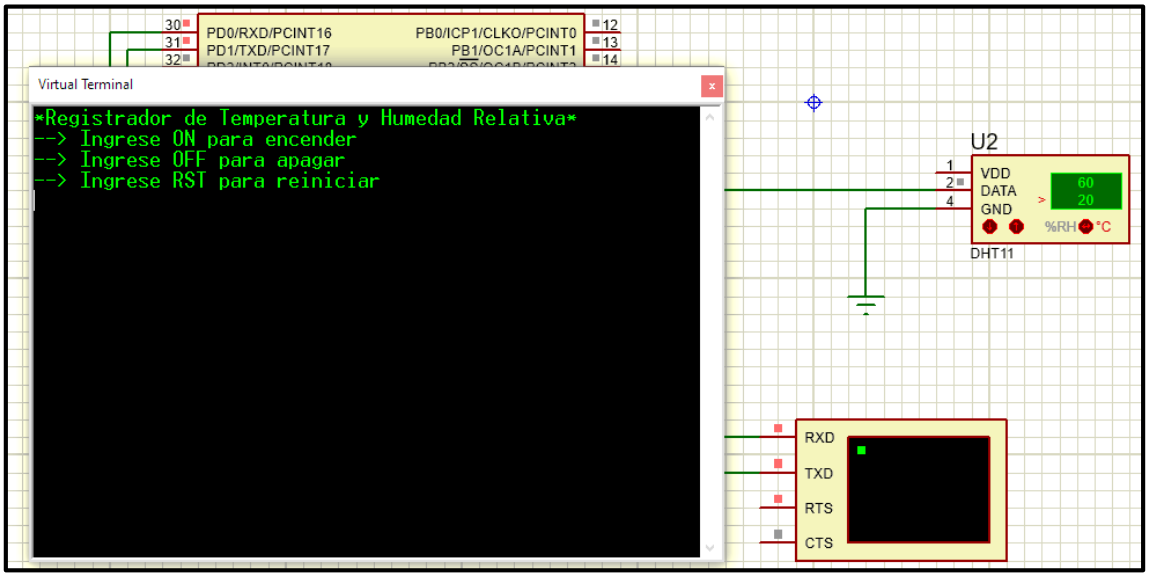


Fig. 3.1. Visualización del inicio del programa en Proteus.

También se midió el tiempo que tarda en imprimirse los 140 caracteres del menú:

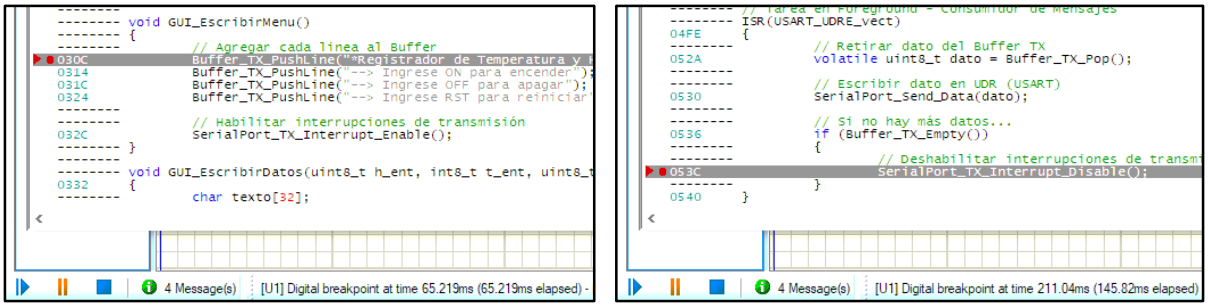


Fig. 3.2. Medición del tiempo de impresión del menú. Son 145 ms, contando los Push.

4 – Comandos

4.1 – Enunciado

Se deberá realizar la verificación de los comandos recibidos y en el caso que no corresponda a lo especificado deberá enviar el mensaje "Comando no válido" y seguirá realizando la misma tarea. Los comandos son:

Tabla 4.1. Comandos disponibles para el usuario.

Comando	Descripción
ON<ENTER>	Encender Registrador
OFF<ENTER>	Apagar Registrador
RST<ENTER>	Detener, volver al estado inicial y mostrar menú

4.2 – Interpretación

Se deberán recibir los caracteres ingresados en la terminal serie y almacenarlos para su posterior procesamiento hasta que el usuario presione la tecla Enter. Si el comando ingresado es “ON”, se deberá encender el registrador de datos; si el comando es “OFF”, se deberá apagar el registrador; y si el comando es “RST”, se deberá detener el registrador en caso de que se encuentre encendido y volver al estado inicial del programa, volviendo a mostrar el menú de opciones. Si no se corresponde con alguna de las opciones válidas, se debe mostrar el mensaje *“Comando no válido”* por la pantalla de la terminal, sin efectuar cambios en el registrador.

4.3 – Resolución

4.3.1 – Intérprete de comandos

Una vez activado el Flag de “Hay Comando” en la rutina *ISR USART_RX_vect*, se detecta el mismo en el *Loop* del programa principal, realizando el siguiente algoritmo:

```
Si el Flag “Hay Comando” está activo
        Retirar texto de Buffer RX
        Procesar comando de texto
        Reiniciar Flag “Hay Comando” con exclusión mutua
```

Pseudocódigo 4.1. Fragmento del Loop de Background en el Main.

Para retirar el texto del Buffer RX, se cuenta con una variable “comando” que almacena como máximo 4 caracteres, incluyendo el fin de cadena. Se invoca *Buffer_RX_PopString* pasando esta variable y el valor 4 como longitud máxima, de modo que si el texto ingresado es de mayor longitud se descarten los demás caracteres y se adelante el índice de lectura al de escritura en el Buffer RX. Luego, se pasa el comando a una función *GUI_LeerComando* que actúa de intérprete, y se reinicia el Flag deshabilitando temporalmente las interrupciones, mediante las funciones *cli()* y *sei()*, para evitar que se pierda un posible dato recibido.

En caso de que el comando ingresado no se corresponda con alguna de las opciones disponibles, se produce el mensaje *“Comando no válido”*, que se almacena en el Buffer TX.

Si se trata de un comando válido, se notificará a un registrador para que realice las acciones correspondientes a dicho comando, según el estado en que se encuentre.

Inicializar índice de String en cero

Realizar…

Retirar dato del Buffer RX
Colocar dato en la posición índice del String
Incrementar índice de String

… mientras dato no sea '\0' y el índice sea menor a longitud máxima

Asignar valor del índice de escritura RX al índice de lectura RX

Pseudocódigo 4.2. Función pública Buffer_RX_PopString.

Si el comando es igual a "ON"
Actualizar registrador con valor ON

Sino si el comando es igual a "OFF"
Actualizar registrador con valor OFF

Sino si el comando es igual a "RST"
Actualizar registrador con valor RST

Sino
Agregar "Comando no válido" al Buffer TX

Pseudocódigo 4.3. Función pública GUI_LeerComando.

4.3.2 – Registrador de datos

La implementación del registrador se realizó mediante una máquina de estados finita de Mealy, con únicamente 2 estados que definen si el sensado de datos se encuentra *encendido* o *apagado*. Posee una entrada de 3 valores posibles: ON, OFF y RST, es decir, los comandos, resultando entonces una MEF *event-driven*. Respecto a la salida, se tienen 5 valores posibles:

- NONE: no se realiza ninguna acción, se continúa sin cambios.
- SHOW_MENU: se muestra el menú principal en la terminal serie.
- START_CLOCK: se inicia o reanuda el sensado periódico de datos.
- STOP_CLOCK: se detiene el reloj y, por ende, el sensado de datos.
- RESET: acciones combinadas de STOP_CLOCK y SHOW_MENU.

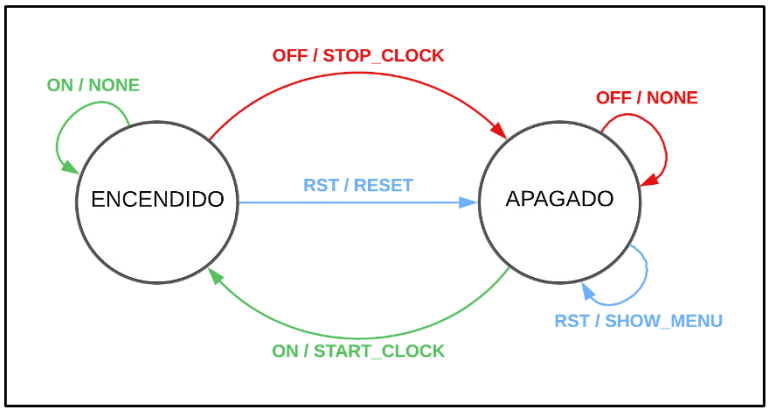


Fig. 4.1. Estados y transiciones definidas para la MEF del registrador.

Por defecto, en la inicialización, mediante la función *Registrador_Init*, se asigna el estado APAGADO y se fuerza que la entrada sea RST para que se muestre el menú. Luego, la actualización de la MEF se produce con la invocación de la función *Registrador_Update*, que recibe el valor de la entrada como parámetro. También existe la función *Registrador_Tick* que incrementa un contador auxiliar de *ticks* para ejecutar instrucciones de forma periódica.

Los estados, los valores de salida y la tabla de transiciones son privadas al módulo registrador; únicamente los valores de entrada son públicos ya que se deben conocer para que el programa principal lo pase como parámetro al momento de procesar un comando.

4.4 – Simulación

En el simulador Proteus, se comprobó que al ingresar los comandos se ejecutaran las acciones correspondientes. A continuación, se adjuntan imágenes de los mensajes mostrados y el contenido del Buffer RX y TX luego ingresar comandos válidos y no válidos.

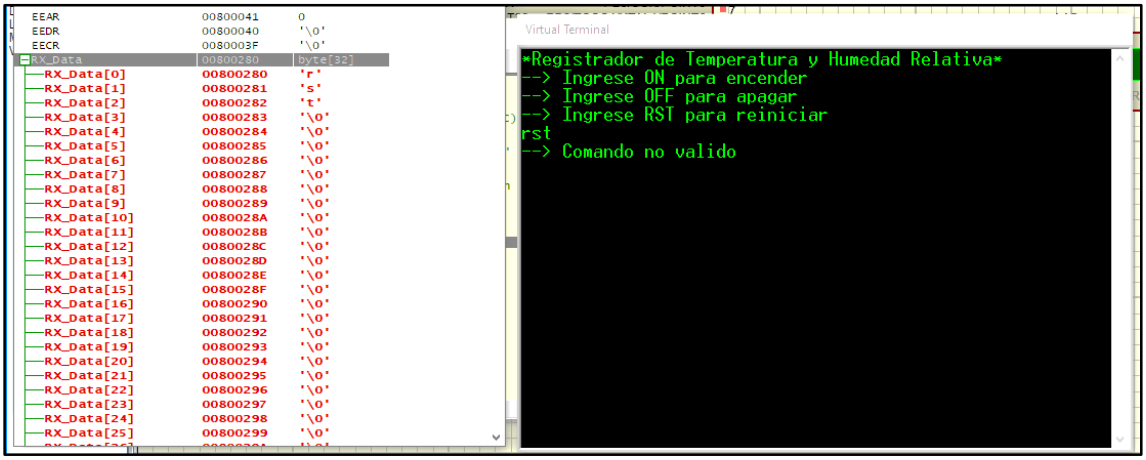


Fig. 4.2. Pantalla de la terminal y contenido de Buffer RX luego de un comando no válido.

Se puede observar que el carácter '\r' no se guarda en el Buffer.

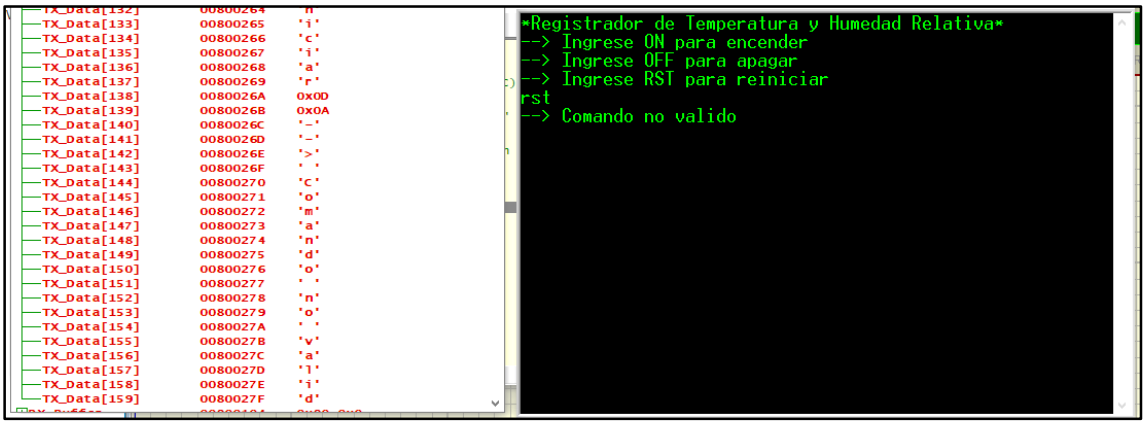


Fig. 4.3. Contenido del Buffer TX en el mismo instante que la Figura 4.2.

Respecto a los mensajes, se puede observar el funcionamiento de índices “circulares” en la Figura 4.3, que tiene como objetivo reducir el riesgo de que, al seguir encolando mensajes, se sobrescriban caracteres aún no mostrados, aprovechando el espacio libre del Buffer.

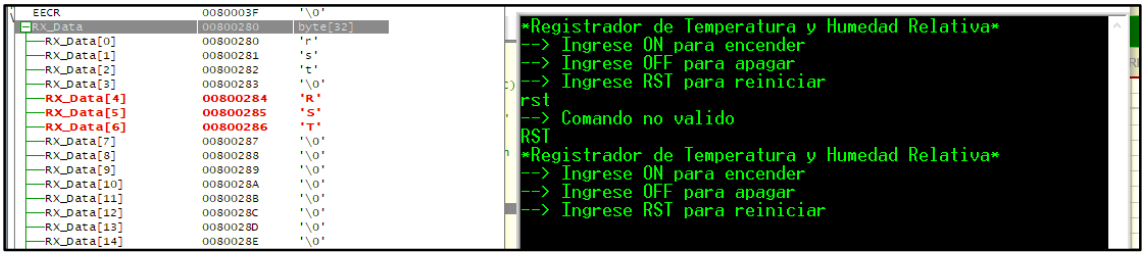


Fig. 4.4. Pantalla de la terminal y contenido del Buffer RX luego de ingresar el comando RST.

5 – Sensado de datos

5.1 – Enunciado

Cuando el usuario encienda el registrador, el MCU obtendrá los datos del sensor y los presentará en la terminal serie cada 1 segundo hasta que el usuario envíe otro comando.

5.2 – Interpretación

Si el registrador se encuentra en estado apagado o detenido, una vez que el usuario ingrese el comando “ON” se deberán realizar solicitudes al sensor de temperatura y humedad ambiente DHT11, provisto en el kit de desarrollo y en el simulador Proteus. Las solicitudes y la posterior recepción y procesamiento de los datos deberán realizarse de manera periódica cada 1 segundo, pudiéndose utilizar interrupciones de Timer o demoras de tiempo.

La presentación de los datos deberá mostrarse en una única línea por cada iteración, y se deberá investigar la cantidad de decimales de cada dato y si pueden ser negativos.

Luego, si el usuario ingresa el comando “OFF” o “RST” mientras está activo, se deberá detener el registrador. La comunicación entre el microcontrolador (MCU) y el sensor deberá realizarse de acuerdo al protocolo indicado en la hoja de datos de este último.

5.3 – Resolución

La implementación de las instrucciones que operan directamente con el pin conectado al sensor (en este caso, PC0), se encuentran modularizadas en una biblioteca “DHT11”, con tres funciones públicas: *DHT_StartRequest*, *DHT_ReceiveData* y *DHT_ReceiveSignedData*.

5.3.1 – Inicio de solicitud

De acuerdo a la hoja de datos del sensor, primero el microcontrolador debe enviar una “señal de comienzo” (Start Signal) al DHT. Dicha señal consiste en mantener su valor en bajo o cero durante un mínimo de 18 ms para asegurarse que el sensor lo detecte, y luego mantener el valor en alto o uno hasta que el DHT responda. El pin está configurado como salida.

Luego, para la lectura de la respuesta del DHT, se configura el pin como entrada y con resistencia de Pull-Up (escribiendo un 0 en el pin correspondiente). La respuesta del sensor tiene una demora entre 20 y 40 μs, y consiste en leer un valor bajo en el pin durante 80 μs, seguido de un valor alto durante también 80 μs. La secuencia se representa a continuación:

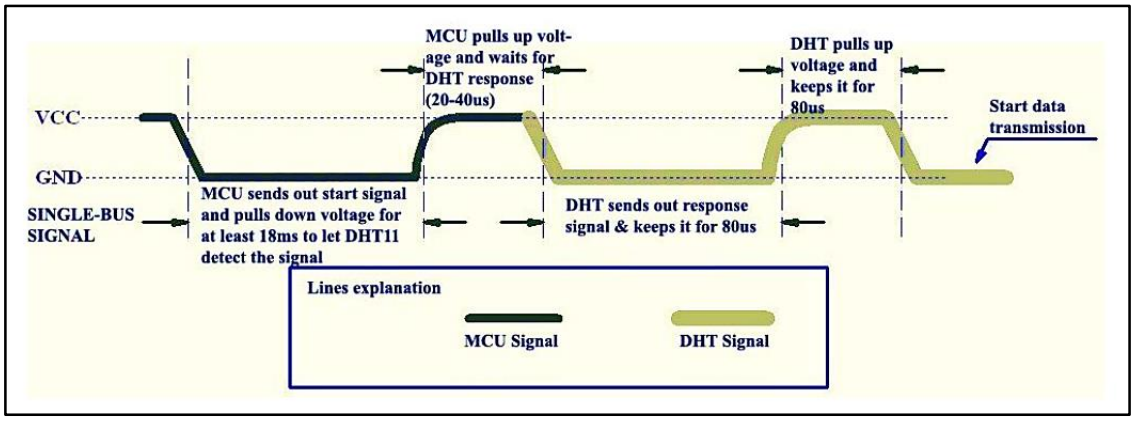


Fig. 5.1. Secuencia de la señal de inicio en el pin conectado al DHT11.

La secuencia descrita se encuentra implementada en la función *DHT_StartRequest*, con tolerancias de hasta 10 μs para las detecciones de valores de señal esperadas del DHT, que implicaría un margen de error del 25% para 40 μs y del 12,5% para 80 μs. Si en algún caso se supera el tiempo máximo de respuesta, implementado de manera genérica en una función privada *esperarCambio*, se cancela el pedido actual para evitar un bloqueo.

```
Configurar el pin como salida

Escribir un 0 en PC0

Demorar un tiempo de 18 ms

Escribir un 1 en PC0

Configurar el pin como entrada + Pull Up

Esperar un cambio de 1 a 0 por 40 us

Si se excedió el tiempo de espera
        Retornar que el sensor no está listo

Esperar un cambio de 0 a 1 por 80 us

Si se excedió el tiempo de espera
        Retornar que el sensor no está listo

Esperar un cambio de 1 a 0 por 80 us

Si se excedió el tiempo de espera
        Retornar que el sensor no está listo

Retornar que el sensor está listo
```

Pseudocódigo 5.1. Función pública DHT_StartRequest.

```
Inicializar contador en cero

Sumar tolerancia de 10 us al máximo de espera

Mientras no cambie el valor del pin y no se alcance el máximo
        Incrementar contador
        Demorar un tiempo de 1 us

Si el contador alcanzó el máximo de espera
        Retornar valor de TIMEOUT
Sino
        Retornar valor del contador
```

Pseudocódigo 5.2. Función privada esperarCambio de la biblioteca DHT11.

5.3.2 – Recepción de un dato

Una vez realizado el proceso anterior, el sensor continúa con el envío de un dato de tamaño 8 bits. Cada bit se debe detectar uno a continuación del otro, mediante la siguiente codificación: primero se recibe un valor "bajo" en el pin durante 50 μs, y luego se recibe un valor "alto" durante una cantidad de tiempo que indica si el bit es un "0" (entre 26 y 28 μs) o un "1" (idealmente 70 μs). Para dar un margen de seguridad, se decidió que se interprete el bit como "0" si el valor en alto tiene una duración mayor o igual a 30 μs, sino es un "1".

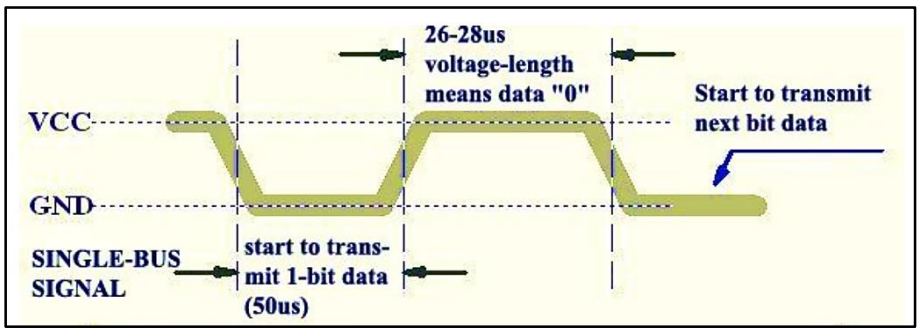


Fig. 5.2. Codificación del bit "0" en la señal del pin conectado al DHT11.

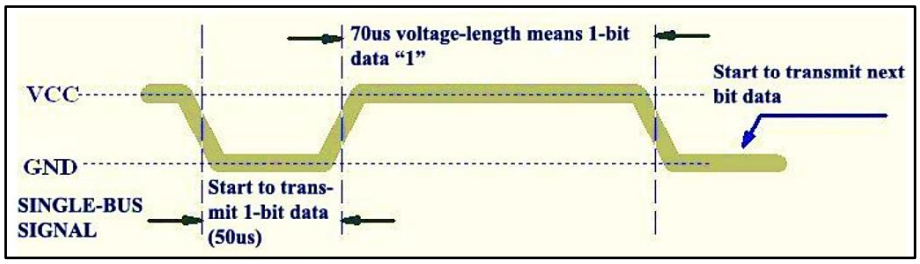


Fig. 5.3. Codificación del bit "1" en la señal del pin conectado al DHT11.

Esperar un cambio de 0 a 1 en el pin hasta 50 us

Esperar un cambio de 1 a 0 en el pin hasta 70 us

Si el tiempo transcurrido es mayor a 30 us

    Retornar valor 1

Sino

    Retornar valor 0

Pseudocódigo 5.3. Función privada recibitBit de la biblioteca DHT11.

El proceso de recepción de un bit se repite unas 8 veces, y se ordenan los mismos para formar el dato, teniendo en cuenta que el primer bit corresponde al más significativo.

Inicializar byte de resultado en cero

Por cada "i" en el rango 0 a 7…

Recibir bit del sensor

Asignar bit en la posición 7-i de resultado

Retornar byte de resultado

Pseudocódigo 5.4. Función pública DHT_ReceiveData.

5.3.3 – Interpretación de datos

El orden de los datos recibidos del sensor DHT11 es el siguiente:

1) Parte entera de la humedad relativa de ambiente.
2) Parte decimal de la humedad relativa. En este modelo siempre es cero.
3) Parte entera de la temperatura. En este modelo puede ser positiva o negativa.
4) Parte decimal de la temperatura. La resolución en este modelo es 0.1 °C
5) Valor de checksum. Permite comprobar si la lectura de los datos fue correcta.

El pseudocódigo del pedido completo de datos se muestra a continuación.

Iniciar una nueva solicitud

Si el sensor está listo

Recibir parte entera de humedad

Recibir parte decimal de humedad

Recibir parte entera de temperatura con signo

Recibir parte decimal de temperatura

Recibir valor de checksum

Si el checksum es igual a la suma de las 4 partes leídas

Escribir línea de datos por pantalla

Sino

Escribir "Error de lectura" por pantalla

Pseudocódigo 5.5. Función privada hacerPedido de la biblioteca Registrador.

5.3.4 – Periodicidad de solicitud

En el estado ENCENDIDO del registrador, se realizan los pedidos de datos cada 1 segundo. Además de ser una condición del enunciado, el sensor no puede devolver pedidos a mayor velocidad, es decir, en períodos de tiempo menores. Para ello, se implementó una biblioteca Reloj, que utiliza al Timer 1 para producir interrupciones cada 200 ms. En cada una, se activa un FLAG_TIMER que indica que debe contarse un *tick*. Un segundo son 5 ticks.

Se decidió también que el reloj únicamente funcione cuando el registrador se encuentre encendido, evitando así la consulta del estado actual por cada interrupción producida. Entonces, en la inicialización del reloj se asigna el modo de funcionamiento CTC, el valor de comparación de 3124 (ya que 3125 x 1024 / 16 MHz es exactamente 200 ms), pero el prescaler se asigna en N = 0, en lugar de N = 1024, de modo que el reloj se mantenga detenido.

```
Deshabilitar las interrupciones del Timer

Asignar Modo CTC y prescaler de 0

Asignar valor de comparación de 3124

Reiniciar contador TCNT1 a cero

Habilitar interrupción de comparación
```

Pseudocódigo 5.6. Función pública RELOJ_Init.

Luego, se proveen 2 funciones de inicio y parada del reloj, llamadas *RELOJ_Start* y *RELOJ_Stop* respectivamente. En la primera únicamente se asigna el prescaler de 1024 para que se reanude su funcionamiento, y en la segunda se asigna prescaler 0 y se reinicia TCNT1.

En el registrador, el inicio de reloj es invocado para la salida START_CLOCK, y su parada para las salidas STOP_CLOCK y RESET. Es importante aclarar estos 2 detalles:

- Cuando se enciende el registrador (transición de apagado a encendido), se realiza un pedido inmediato, de modo que no se espera 1 segundo para el primer dato.
- Al detener el reloj, también se reinician los ticks a 0 para que efectivamente se realice el próximo pedido luego de 1 segundo, sino podría ocurrir a los 200 ms.

Los algoritmos para contar ticks y actualizar los estados se muestran a continuación.

```
Incrementar contador de ticks
Si el contador es igual a 5
        Reiniciar contador de ticks a cero
        Realizar un pedido de datos
```

Pseudocódigo 5.7. Función pública Registrador_Tick

```
Obtener salida a partir de estado actual y entrada
Obtener próximo estado a partir de estado actual y entrada
Asignar próximo estado al estado actual

Si la salida es START_CLOCK…
        Iniciar reloj
        Realizar un pedido
Sino si es STOP_CLOCK…
        Detener reloj
        Reiniciar contador de ticks a cero
Sino si es RESET…
        Detener reloj
        Reiniciar contador de ticks a cero
        Mostrar menú de opciones por pantalla
Sino si es SHOW_MENU
        Mostrar menú de opciones por pantalla
```

Pseudocódigo 5.8. Función pública Registrador_Update

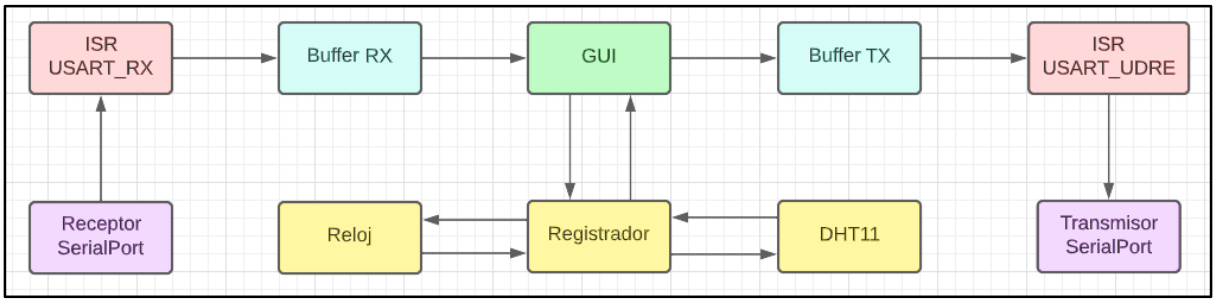


Fig. 5.4. Resumen de la transferencia de datos entre los componentes del proyecto.

5.4 – Simulación

Para verificar la correcta presentación de los datos, primero se decidió realizar la medición del tiempo entre cada impresión de línea del registrador, que tiene que ser cada 1 segundo según lo calculado para las interrupciones de Timer 1 y el contador de ticks.

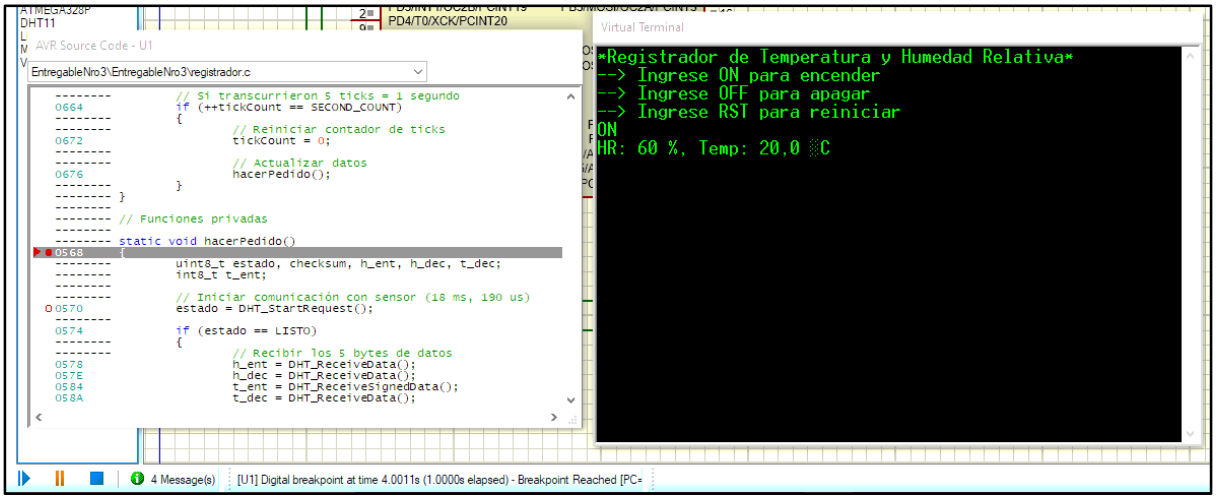


Fig. 5.5. Medición del tiempo entre dos invocaciones a la función hacerPedido. Es 1 segundo.

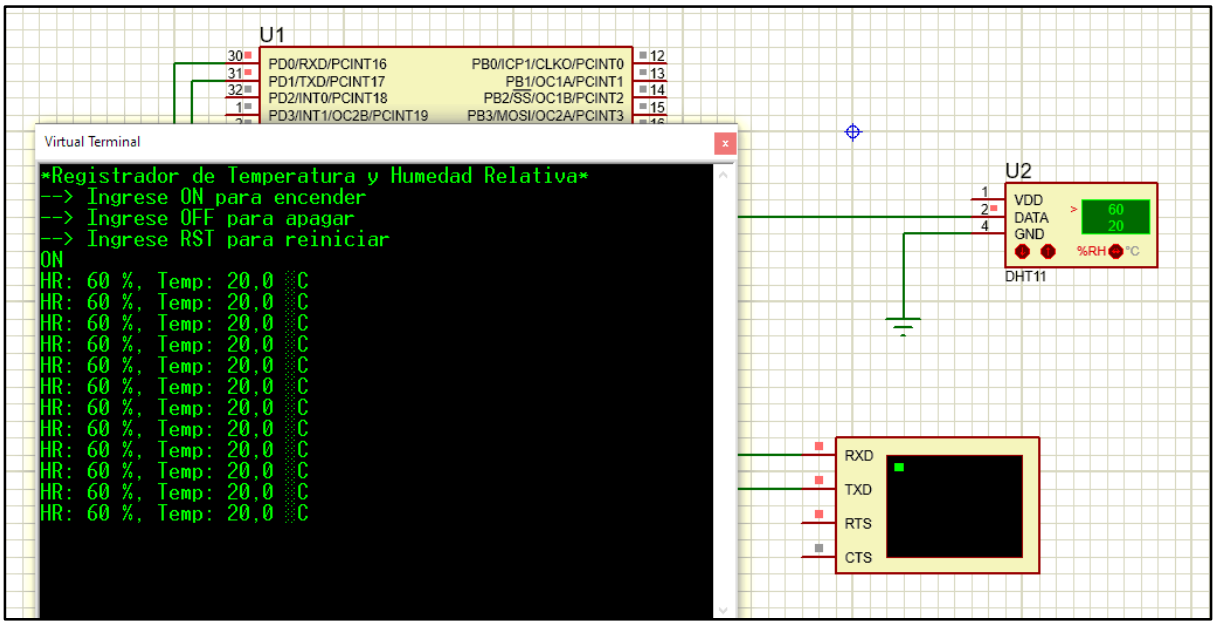


Fig. 5.6. Visualización de los datos en la terminal serie de Proteus.

Luego, cuando el usuario ingresa el comando “OFF” debe detenerse el reloj y, por ende, el registrador. Esto puede verificarse mediante un breakpoint en *Registrador_Tick*, que es invocada en el Loop de Background si FLAG_TIMER está activo.

Se adjunta un enlace al vídeo de dicha demostración: https://bit.ly/cdym-tp3-01

También se verificó que al reanudar el registrador se produzca la primera solicitud de manera inmediata, y el que la segunda solicitud se realiza 1 segundo después de la anterior.

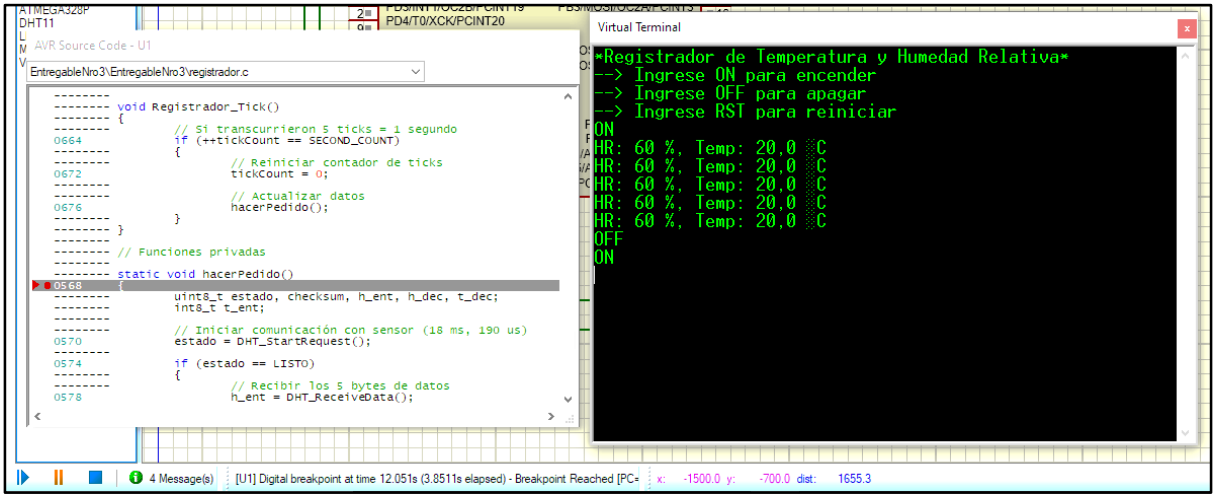


Fig. 5.7. Inicio del primer pedido luego de reanudarse el registrador.

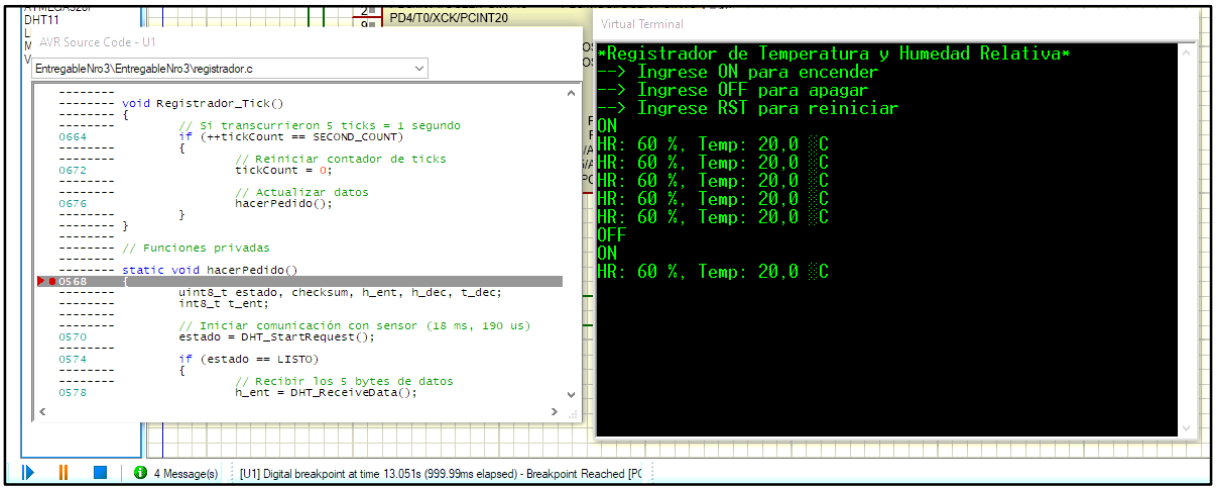


Fig. 5.8. Medición del tiempo hasta el inicio del siguiente pedido. Es 1 segundo.

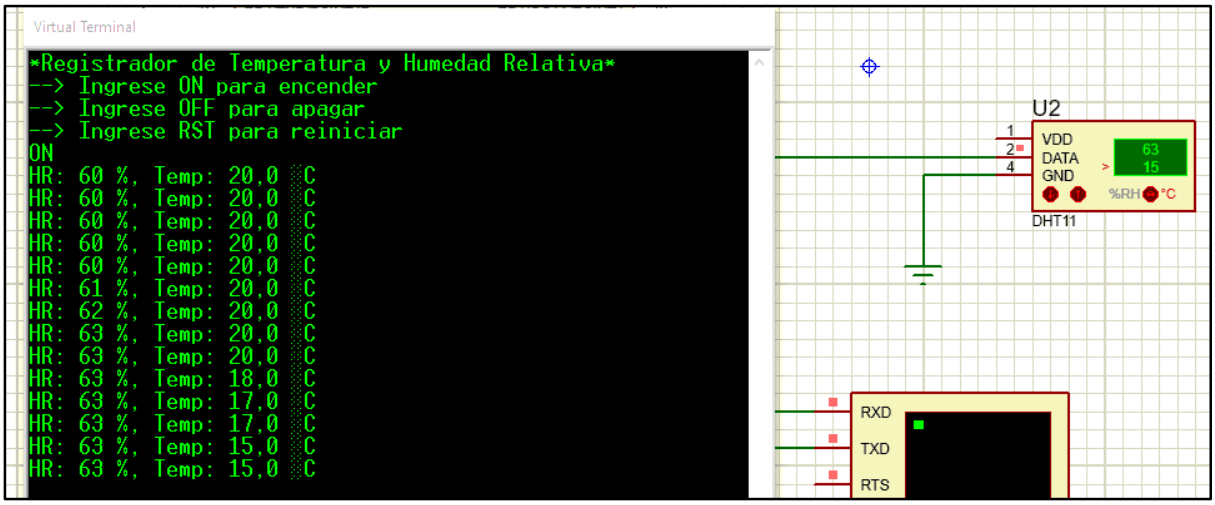


Fig. 5.9. Visualización de datos cambiantes en la terminal serie. El sensor de Proteus no posee decimales.

6 – Validación en el kit

El funcionamiento del programa se verificó con el kit de desarrollo, conectado a un puerto USB de una computadora con una terminal serie, en este caso, la terminal Bray.

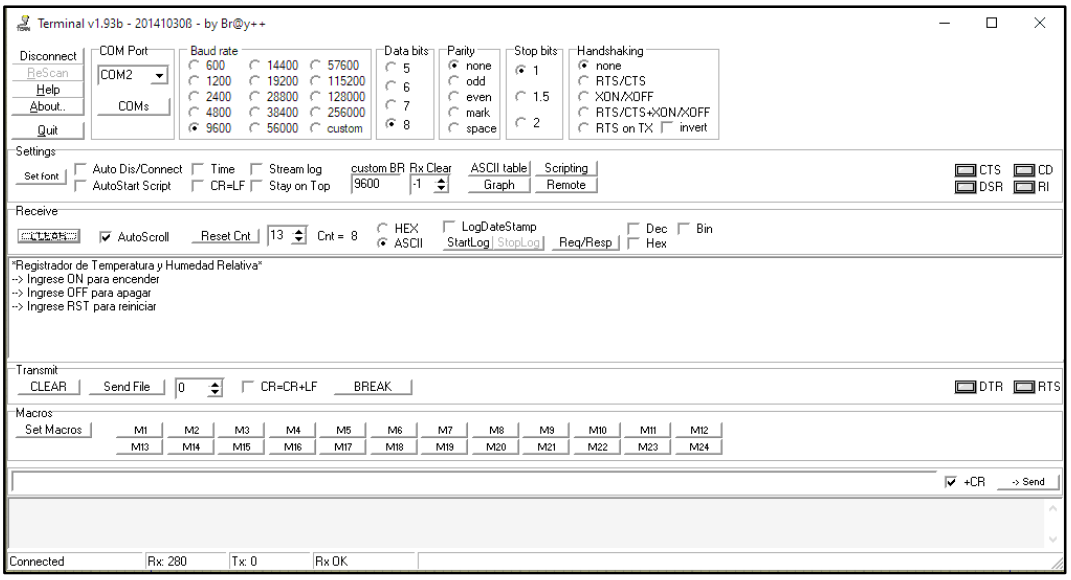


Fig. 6.1. Visualización del menú principal en la terminal Bray.

Para el ingreso de comandos, se escribe el mismo en el campo de texto inferior, y debe activarse la casilla “+CR” para que al presionar Enter o “Send” se envíe el carácter ‘\r’.

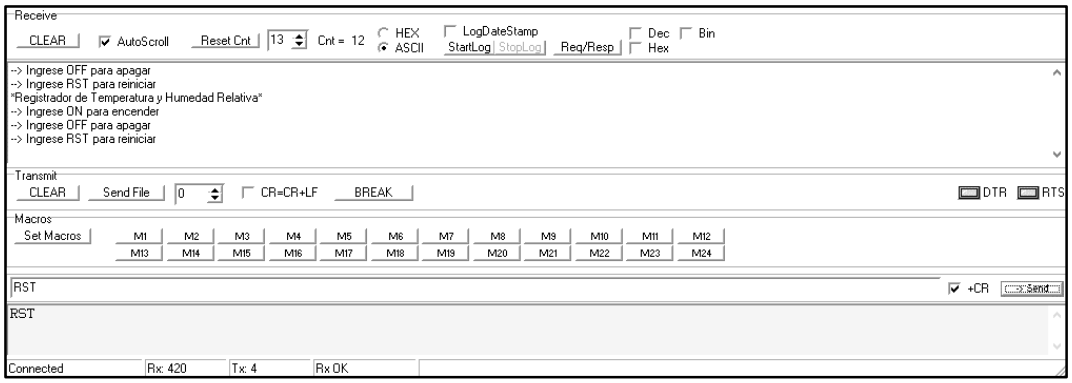


Fig. 6.2. Resultado posterior al ingreso del comando “RST” en la terminal Bray.

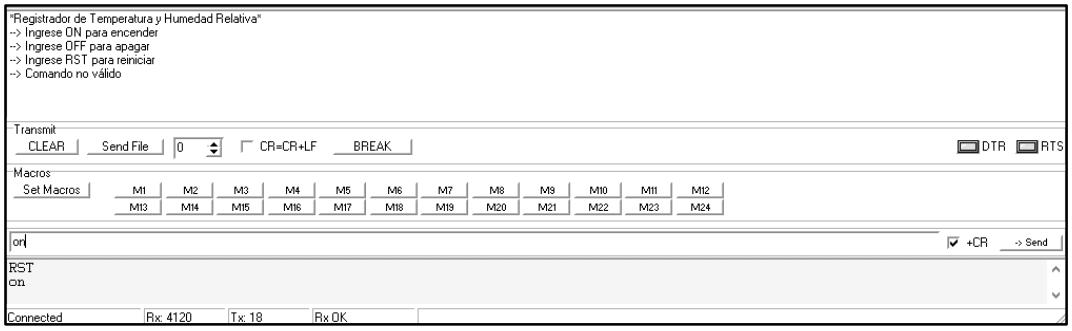


Fig. 6.3. Resultado posterior al ingreso de un comando no válido en la terminal Bray.

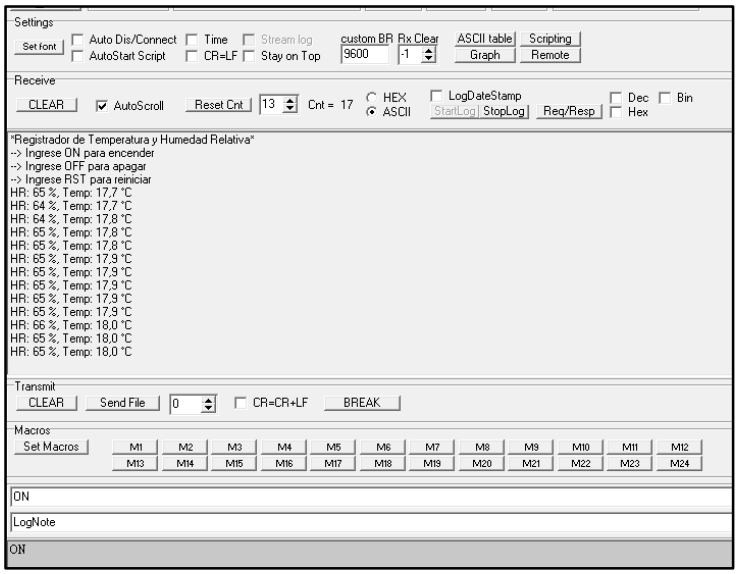


Fig. 6.4. Muestreo de datos luego de ingresar el comando ON en la terminal Bray.

Para finalizar, se activó el Log en la terminal para poder graficar los datos:

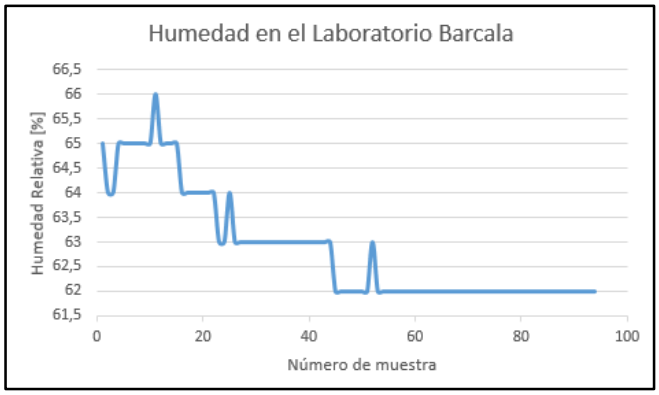


Fig. 6.5. Evolución de la humedad relativa según datos del DHT11 en el Laboratorio Barcala.

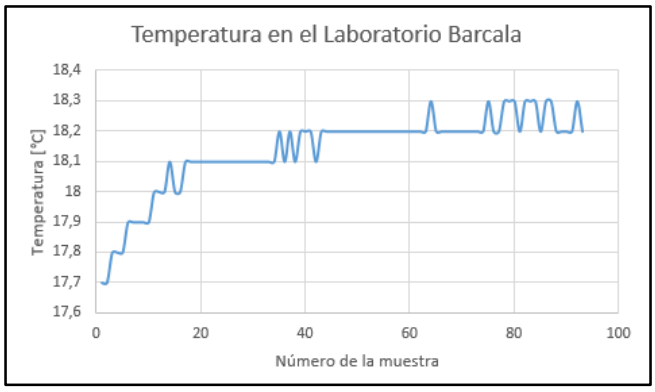


Fig. 6.6. Evolución de la temperatura según datos del DHT11 en el Laboratorio Barcala

7 – Conclusiones

En la implementación de este proyecto, se tuvieron en cuenta diferentes problemáticas y diferencias encontradas entre la simulación en condiciones ideales de Proteus y los resultados reales obtenidos al momento de testear el programa en el kit de desarrollo:

- Los rangos y la resolución de los datos de humedad y temperatura son diferentes en el modelo DHT11 de Proteus y el kit. En el caso de la temperatura, se agregó un decimal y se utiliza una variable tipo int8_t para la parte entera ya que puede ser negativa.
- El sensor real puede dejar de responder si las solicitudes se realizan en períodos cortos de tiempo o por otros eventos externos. Se cambiaron las lecturas bloqueantes por controles de tiempo máximo de espera de respuesta (timeout), ya que no se garantiza que todos los pedidos sean correctos, evitando así un posible bloqueo del programa.
- La respuesta del sensor y sus cambios de señal no tardan exactamente los indicados en la hoja de datos, por lo que se agregó una tolerancia de hasta 10 μs para el timeout.

Por otra parte, se tuvo como objetivo priorizar la modularización de los componentes y de ciertos “objetos” o estructuras. En el caso del Buffer, al ser totalmente independiente del resto del programa, tiene una alta portabilidad. El registrador, que es una máquina de estados, facilita el agregado de nuevas características. Por último, el módulo del DHT11 se limita a realizar la comunicación con el sensor: el registrador lo invoca y luego procesa los datos.

8 – Bibliografía

- Atmel Corporation (2015). *ATmega 328P Datasheet.* Capítulos 11, 14, 15 y 19. https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf
- Mouser Electronics (s.f.). *DHT11 Humidity & Temperature Sensor (Datasheet)*. https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf
- Naylamp Mechatronics (s.f.). *Sensor de Temperatura y Humedad Relativa DHT11*. https://naylampmechatronics.com/sensores-temperatura-y-humedad/57-sensor-de-temperatura-y-humedad-relativa-dht11.html

Apéndice A – Archivos de cabecera

A continuación, se muestran los archivos de cabecera del proyecto realizado.

buffer.h

```
/*
 * Created: 24/6/2022 22:40:20
 * Author: Calderón Sergio
 */

#ifndef BUFFER_H_
	#define BUFFER_H_

	// ------------------- Includes ---------------------------

	// Archivo de cabecera del Microcontrolador
	#include <avr/io.h>


	// ------------------- Constantes ---------------------------

	// Tamaño del buffer RX en bytes
	#define RX_BUFFER_SIZE 32

	// Tamaño del buffer TX en bytes
	#define TX_BUFFER_SIZE 160


	// -------- Prototipos de funciones públicas --------------

	// --------- Operaciones sobre Buffer RX ------------------

	// Agrega un dato al Buffer RX
	void Buffer_RX_Push(uint8_t);

	// Extrae los datos del Buffer RX en el texto de longitud maxLength
	void Buffer_RX_PopString(char* texto, uint8_t maxLength);

	// Retira un dato del Buffer RX
	uint8_t Buffer_RX_Pop();

	// --------- Operaciones sobre Buffer TX -----------------

	// Agrega un dato al Buffer TX
	void Buffer_TX_Push(uint8_t);

	// Agrega una cadena de texto al Buffer TX
	void Buffer_TX_PushString(char*);

	// Agrega una cadena de texto + un salto de línea al Buffer TX
	void Buffer_TX_PushLine(char*);

	// Retira un dato del Buffer TX
	uint8_t Buffer_TX_Pop();

	// Retorna 1 si no hay datos sin retirar en Buffer TX, 0 en caso contrario
	uint8_t Buffer_TX_Empty();

#endif /* BUFFER_H_ */
```

dht11.h

```
/*
 * Created: 7/6/2022 00:28:42
 * Author: Calderón Sergio
 */

#ifndef DHT11_H_
	#define DHT11_H_

	// ------------------- Includes ----------------------------

	// Archivo de cabecera del programa principal
	#include "main.h"

	// Demoras de tiempo
	#include <util/delay.h>

	// ------------------- Constantes ----------------------------

	// Pin de Port C por el cual se realiza la comunicación
	#define PIN_DATA PORTC0

	// Margen de error para el tiempo máximo de espera de cambio de señal
	#define TOLERANCIA 10

	// Código de error de tiempo excedido de respuesta
	#define TIMEOUT 255

	// Código de validación para comenzar la recepción de datos
	#define LISTO 1

	// -------- Prototipos de funciones públicas ---------------

	// Solicitud para iniciar comunicación con sensor
	// Retorna la constante LISTO si el sensor respondió correctamente
	uint8_t DHT_StartRequest();

	// Recepción de un byte sin signo
	uint8_t DHT_ReceiveData();

	// Recepción de un byte con signo
	int8_t DHT_ReceiveSignedData();

#endif /* DHT11_H_ */
```

GUI.h

```
/*
 * Created: 25/6/2022 20:10:40
 * Author: Calderón Sergio
 */

#ifndef GUI_H_
	#define GUI_H_

	// ------------------- Includes ----------------------------

	// Archivo de cabecera del Microcontrolador
	#include <avr/io.h>
```

```
	// Archivo de cabecera del registrador
	#include "registrador.h"

	// Archivo de cabecera del Buffer
	#include "buffer.h"

	// Archivo de cabecera de la terminal serie
	#include "serialPort.h"

	// Utilidades con Strings
	#include <string.h>
	#include <stdio.h>

	// -------- Prototipos de funciones públicas ---------------

	void GUI_EscribirMenu();
	void GUI_LeerComando(char*);
	void GUI_EscribirError(char*);
	void GUI_EscribirDatos(uint8_t h_ent, int8_t t_ent, uint8_t t_dec);

#endif /* GUI_H_ */
```

main.h

```
/*
 * Created: 6/6/2022 15:02:49
 * Author: Calderón Sergio
 */

#ifndef MAIN_H_
	#define MAIN_H_

	// ------------------- Constantes ----------------------------

	// Frecuencia de reloj del Microcontrolador
	#define F_CPU 16000000UL

	// Cantidad de caracteres máximo (contando '\0') de un comando válido
	#define MAX_CMD_SIZE 4

	// ------------------- Includes ----------------------------

	// Archivo de cabecera del Microcontrolador
	#include <avr/io.h>

	// Interrupciones del Microcontrolador
	#include <avr/interrupt.h>

	// Modos de ahorro de energía
	#include <avr/sleep.h>

	// Archivos de cabecera de los módulos del proyecto
	#include "serialPort.h"
	#include "reloj.h"
	#include "buffer.h"
	#include "registrador.h"
	#include "GUI.h"

#endif /* MAIN_H_ */
```

registrador.h

```
/*
 * Created: 25/6/2022 18:33:07
 *  Author: Calderón Sergio
 */


#ifndef REGISTRADOR_H_
	#define REGISTRADOR_H_

	// ------------------- Includes ----------------------------

	// Archivo de cabecera del Microcontrolador
	#include <avr/io.h>

	// Archivo de cabecera de interfaz de usuario
	#include "GUI.h"

	// Archivo de cabecera del reloj
	#include "reloj.h"

	// Archivo de cabecera del sensor
	#include "dht11.h"

	// ------------------- Constantes ----------------------------

	// Entradas definidas
	typedef enum { ON, OFF, RST } MEF_Entrada;

	// -------- Prototipos de funciones públicas ---------------

	// Prepara el registrador, define el estado inicial y muestra el menú
	void Registrador_Init();

	// Incrementa el contador para las solicitudes temporizadas
	void Registrador_Tick();

	// Actualiza el registrador a partir de un comando ingresado
	void Registrador_Update(uint8_t);

#endif /* REGISTRADOR_H_ */
```

reloj.h

```
/*
 * Created: 25/6/2022 18:33:07
 *  Author: Calderón Sergio
 */


#ifndef RELOJ_H_
	#define RELOJ_H_

	// ------------------- Includes ----------------------------

	// Archivo de cabecera del Microcontrolador
	#include <avr/io.h>
```

```
	// ------------------- Constantes ---------------------------

	// Periodo de interrupción, en milisegundos
	#define TIMER_OFFSET 200

	// Interrupciones requeridas para completar 1 segundo
	#define SECOND_COUNT (1000 / TIMER_OFFSET)

	// -------- Prototipos de funciones públicas --------------

	// Inicializa el reloj usando Timer 1
	void RELOJ_Init();

	// Reasigna el preescaler para reanudar el reloj
	void RELOJ_Start();

	// Detiene el reloj y resetea el contador a cero
	void RELOJ_Stop();

#endif /* RELOJ_H_ */
```

serialPort.h

```
/*
 * Created: 07/10/2020 03:02:42
 * Author: vfperri, Calderón Sergio
 */

#ifndef SERIALPORT_H_
	#define SERIALPORT_H_

	// ------------------- Includes ---------------------------

	// Archivo de cabecera del Microcontrolador
	#include <avr/io.h>

	// -------- Prototipos de funciones públicas --------------

	// Inicializa el puerto serie para la frecuencia de CPU especificada
	void SerialPort_Init(uint32_t);

	// Inicializa y habilita el receptor
	void SerialPort_RX_Enable(void);

	// Inicializa y habilita el transmisor
	void SerialPort_TX_Enable(void);

	// Habilitación y deshabilitación de interrupciones
	void SerialPort_RX_Interrupt_Enable(void);
	void SerialPort_TX_Interrupt_Enable(void);
	void SerialPort_TX_Interrupt_Disable(void);

	// Transmite el dato pasado por parámetro
	void SerialPort_Send_Data(uint8_t);

	// Recibe y retorna el dato actual
	uint8_t SerialPort_Receive_Data(void);

#endif /* SERIALPORT_H_ */
```

Apéndice B – Archivos .C

A continuación, se muestran los archivos .C que conforman el programa completo, los cuales se compilan y utilizan para la simulación y la validación en el kit de desarrollo.

buffer.c

```
/*
 * Created: 24/6/2022 22:42:16
 *  Author: Calderon Sergio
 */

// Archivo de cabecera
#include "buffer.h"

// Estructura tipo Buffer
typedef struct {
	uint8_t* data;
	uint8_t readIndex;
	uint8_t writeIndex;
} TipoBuffer;


// ---------------- Variables privadas ----------------------

// Búfferes de tamaño asignado según su función
static uint8_t RX_Data[RX_BUFFER_SIZE];
static uint8_t TX_Data[TX_BUFFER_SIZE];

static TipoBuffer RX_Buffer = {RX_Data, 0, 0};
static TipoBuffer TX_Buffer = {TX_Data, 0, 0};

// ---------------- Funciones públicas ----------------------

void Buffer_RX_Push(uint8_t dato)
{
	// Asignar dato en la posición de escritura
	RX_Buffer.data[RX_Buffer.writeIndex] = dato;

	// Incrementar indice de escritura
	RX_Buffer.writeIndex = (RX_Buffer.writeIndex + 1) % RX_BUFFER_SIZE;
}

uint8_t Buffer_RX_Pop()
{
	// Recuperar dato en la posición de lectura
	uint8_t dato = RX_Buffer.data[RX_Buffer.readIndex];

	// Incrementar indice de lectura
	RX_Buffer.readIndex = (RX_Buffer.readIndex + 1) % RX_BUFFER_SIZE;

	// Devolver dato
	return dato;
}

void Buffer_RX_PopString(char* texto, uint8_t maxLength)
{
	// Inicializar indice de String en cero
	uint8_t dato, i = 0;
```

```
	// Hasta alcanzar un fin de cadena o la longitud máxima...
	do {
		// Retirar dato del Buffer RX
		dato = Buffer_RX_Pop();

		// Colocar dato en la posición "i" del String
		texto[i] = dato;
		i++;
	} while (dato != '\0' && i < maxLength);

	// Adelantar índice de lectura hasta el de escritura
	RX_Buffer.readIndex = RX_Buffer.writeIndex;
}

void Buffer_TX_Push(uint8_t dato)
{
	// Asignar dato en la posición de escritura
	TX_Buffer.data[TX_Buffer.writeIndex] = dato;

	// Incrementar indice de escritura
	TX_Buffer.writeIndex = (TX_Buffer.writeIndex + 1) % TX_BUFFER_SIZE;
}

void Buffer_TX_PushString(char* texto)
{
	// Por cada caracter del String
	for(uint8_t i=0; texto[i] != '\0'; i++){
		// Agregar caracter al Buffer TX
		Buffer_TX_Push(texto[i]);
	}
}

void Buffer_TX_PushLine(char* texto)
{
	// Por cada caracter del String
	for(uint8_t i=0; texto[i] != '\0'; i++) {
		// Agregar caracter al Buffer TX
		Buffer_TX_Push(texto[i]);
	}

	// Agregar salto de linea
	Buffer_TX_Push('\r');
	Buffer_TX_Push('\n');
}

uint8_t Buffer_TX_Pop()
{
	// Obtener dato en la posición de lectura
	uint8_t dato = TX_Buffer.data[TX_Buffer.readIndex];

	// Incrementar indice de lectura
	TX_Buffer.readIndex = (TX_Buffer.readIndex + 1) % TX_BUFFER_SIZE;

	// Devolver dato
	return dato;
}

uint8_t Buffer_TX_Empty()
{
	// Devolver si los indices de lectura y escritura coinciden
	return TX_Buffer.readIndex == TX_Buffer.writeIndex;
}
```

dht11.c

```
/*
 * Created: 7/6/2022 00:29:55
 * Author: Calderón Sergio
 */

// Archivo de cabecera
#include "dht11.h"


// -------- Prototipos de funciones privadas ---------------

static void setSalida();
static void setPullUp();
static void bajar();
static void subir();
static uint8_t estadoPIN();
static uint8_t recibirBit();
static uint8_t esperarCambio(uint8_t max_us, uint8_t valor_actual);


// -------- Implementación de funciones públicas ---------------

uint8_t DHT_StartRequest()
{
	uint8_t t;

	// Configurar el pin como salida
	setSalida();

	// Señal de Start del MCU a DHT
	bajar();                    // Escribir un 0
	_delay_ms(18);              // Demorar 18 ms
	subir();                    // Escribir un 1

	// Configurar el pin como entrada + Pull Up
	setPullUp();

	// Esperar respuesta del sensor, entre 20 y 40 us
	t = esperarCambio(40, 1);
	if (t == TIMEOUT) return !LISTO;

	// Recibir un 0 por 80 us
	t = esperarCambio(80, 0);
	if (t == TIMEOUT) return !LISTO;

	// Recibir un 1 por 80 us
	t = esperarCambio(80, 1);
	if (t == TIMEOUT) return !LISTO;

	// Retornar que el sensor está listo
	return LISTO;
}

uint8_t DHT_ReceiveData()
{
	// Inicializar byte resultado en cero
	uint8_t bit, resultado = 0;

	// Se recibe cada bit, del más significativo al menos significativo
	// Por ejemplo: 0101 0000 es 80
```

```
	for (uint8_t i=0; i<8; i++){
		bit = recibirBit();
		if (bit) resultado |= (1 << (7-i));
	}

	// Retornar resultado formado
	return resultado;
}

int8_t DHT_ReceiveSignedData()
{
	uint8_t bit;
	int8_t resultado = 0;

	// Se recibe cada bit, del más significativo al menos significativo
	// Por ejemplo: 0101 0000 es 80
	for (uint8_t i=0; i<8; i++){
		bit = recibirBit();
		if (bit) resultado |= (1 << (7-i));
	}

	return resultado;
}


// -------- Implementación de funciones privadas ---------------

static void setSalida()
{
	DDRC |= (1<<PIN_DATA);		// Establece el pin como salida
}

static void setPullUp()
{
	DDRC &= ~(1<<PIN_DATA);		// Configuración como entrada
	PORTC |= (1<<PIN_DATA);		// Pull-Up
}

static void subir()
{
	PORTC |= (1<<PIN_DATA);		// Escribe un 1 en el pin
}

static void bajar()
{
	PORTC &= ~(1<<PIN_DATA);	// Escribe un 0 en el pin
}

static uint8_t recibirBit()
{
	uint8_t t_transcurrido;

	// Primero el sensor mantiene en bajo durante 50 us
	t_transcurrido = esperarCambio(50, 0);

	// Se mide el tiempo en alto, máximo 70 us en condiciones ideales
	t_transcurrido = esperarCambio(70, 1);

	// Si la medida es mayor a 30 us, se interpreta como 1, sino es un 0
	return t_transcurrido > 30;
}
```

```
static uint8_t estadoPIN()
{
	return PINC & (1<<PIN_DATA);		// Retorna la lectura del pin
}

static uint8_t esperarCambio(uint8_t max_us, uint8_t valor_actual)
{
	// Inicializar contador en cero
	uint8_t aux = 0;

	// Sumar tolerancia al máximo de espera
	max_us += TOLERANCIA;

	// Mientras no cambie el valor del pin y reste tiempo
	while(estadoPIN() == valor_actual && aux < max_us)
	{
		// Incrementar contador
		aux++;

		// Demorar un tiempo de 1 us
		_delay_us(1);
	}

	// Si se alcanzó el tiempo máximo, devolver timeout
	// Sino devolver el tiempo transcurrido (contador)
	return aux == max_us ? TIMEOUT : aux;
}
```

GUI.c

```
/*
 * Created: 25/6/2022 20:12:25
 *  Author: Calderón Sergio
 */

// Archivo de cabecera
#include "GUI.h"

// ---------------- Funciones públicas -----------------------

// Tareas en Background - Productores de Mensajes

void GUI_EscribirMenu()
{
	// Agregar cada línea del menú al Buffer
	Buffer_TX_PushLine("*Registrador de Temperatura y Humedad Relativa*");
	Buffer_TX_PushLine("--> Ingrese ON para encender");
	Buffer_TX_PushLine("--> Ingrese OFF para apagar");
	Buffer_TX_PushLine("--> Ingrese RST para reiniciar");

	// Habilitar interrupciones de transmisión
	SerialPort_TX_Interrupt_Enable();
}

void GUI_EscribirDatos(uint8_t h_ent, int8_t t_ent, uint8_t t_dec)
{
	char texto[32];

	// Construir mensaje con los datos
	sprintf(texto, "HR: %u %%, Temp: %d,%u °C", h_ent, t_ent, t_dec);
```

```
	// Agregar mensaje al Buffer
	Buffer_TX_PushLine(texto);

	// Habilitar interrupciones de transmisión
	SerialPort_TX_Interrupt_Enable();
}

void GUI_EscribirError(char* mensaje)
{
	// Agregar mensaje al Buffer
	Buffer_TX_PushLine(mensaje);

	// Habilitar interrupciones de transmisión
	SerialPort_TX_Interrupt_Enable();
}

// Background - Procesador de Mensajes
void GUI_LeerComando(char* comando)
{
	// Se envia el comando al registrador unicamente si es válido
	if (strcmp(comando, "ON") == 0) Registrador_Update(ON);
	else if (strcmp(comando, "OFF") == 0) Registrador_Update(OFF);
	else if (strcmp(comando, "RST") == 0) Registrador_Update(RST);
	else GUI_EscribirError("--> Comando no válido");
}
```

main.c

```
// Archivo de cabecera
#include "main.h"

// -------- Variables compartidas entre ISR y programa principal ---------------
static volatile uint8_t FLAG_TIMER = 0;		// Flag de Timer
static volatile uint8_t FLAG_CMD = 0;		// Flag "Hay Comando"


int main(void)
{
	// Variables locales
	char comando[MAX_CMD_SIZE];

	/* Setup */

	// Inicialización de USART
	SerialPort_Init(F_CPU);				// Inicializar periférico
	SerialPort_TX_Enable();				// Habilitar transmisor
	SerialPort_RX_Enable();				// Habilitar receptor
	SerialPort_RX_Interrupt_Enable();		// Habilitar int. de recepción

	// Inicialización del reloj
	RELOJ_Init();

	// Inicialización del registrador
	Registrador_Init();

	// Se habilitan las interrupciones globales
	sei();
```

```
    /* Loop de Background */
    while (1)
    {
            // Si se activó Flag de Timer
            if (FLAG_TIMER)
            {
                    // Actualizar Registrador y desactivar flag
                    Registrador_Tick();
                    FLAG_TIMER = 0;
            }

            // Tarea en Background - Consumidor de Comandos
            // Si "Hay Comando"...
            if (FLAG_CMD)
            {
                    // Obtener comando del Buffer
                    Buffer_RX_PopString(comando, MAX_CMD_SIZE);

                    // Procesar comando
                    GUI_LeerComando(comando);

                    // Desactivar Flag con exclusión mutua
                    cli();
                    FLAG_CMD = 0;
                    sei();
            }

            // Entrar en modo ahorro de energía
            sleep_mode();
        }

        return 0;
}

// -------- Rutinas de interrupción ---------------

// Timer 1, se activa el flag cada 200 ms
ISR(TIMER1_COMPA_vect)
{
        FLAG_TIMER = 1;
}

// Rutina de Servicio de Interrupción de Byte Recibido
// Tarea en Foreground - Productor de Comandos
ISR(USART_RX_vect)
{
        // Leer dato de UDR (USART)
        volatile uint8_t aux = SerialPort_Receive_Data();

        // Si se presionó Enter
        if (aux == '\r')
        {
                // Agregar un fin de cadena al Buffer RX
                Buffer_RX_Push('\0');

                // Activar Flag "Hay Comando"
                FLAG_CMD = 1;
        } else {
                // Sino, agregar dato al Buffer RX
                Buffer_RX_Push(aux);
        }
}
```

```
// Rutina de Servicio de Interrupción "Libre para Transmitir"
// Tarea en Foreground - Consumidor de Mensajes
ISR(USART_UDRE_vect)
{
	// Retirar dato del Buffer TX
	volatile uint8_t dato = Buffer_TX_Pop();

	// Escribir dato en UDR (USART)
	SerialPort_Send_Data(dato);

	// Si no hay más datos...
	if (Buffer_TX_Empty())
	{
		// Deshabilitar interrupciones de transmisión
		SerialPort_TX_Interrupt_Disable();
	}
}
```

registrador.c

```
#include "registrador.h"

// Estados y salidas definidas
typedef enum { ENCENDIDO, APAGADO } MEF_Estado;
typedef enum { NONE, SHOW_MENU, START_CLOCK, STOP_CLOCK, RESET } MEF_Salida;

// Variables privadas
static MEF_Estado tablaEstados[2][3];
static MEF_Salida tablaSalidas[2][3];
static MEF_Estado estadoActual;
static uint8_t tickCount = 0;

// Prototipos de funciones privadas
static void hacerPedido();

// -------------- Funciones públicas --------------------

void Registrador_Init()
{
	// Construir la tabla de transición de estados
	tablaEstados[ENCENDIDO][ON] = ENCENDIDO;
	tablaEstados[ENCENDIDO][OFF] = APAGADO;
	tablaEstados[ENCENDIDO][RST] = APAGADO;
	tablaEstados[APAGADO][ON] = ENCENDIDO;
	tablaEstados[APAGADO][OFF] = APAGADO;
	tablaEstados[APAGADO][RST] = APAGADO;

	// Asignar las salidas para cada transición
	tablaSalidas[APAGADO][ON] = START_CLOCK;
	tablaSalidas[APAGADO][OFF] = NONE;
	tablaSalidas[APAGADO][RST] = SHOW_MENU;
	tablaSalidas[ENCENDIDO][ON] = NONE;
	tablaSalidas[ENCENDIDO][OFF] = STOP_CLOCK;
	tablaSalidas[ENCENDIDO][RST] = RESET;

	// Asignar estado inicial y mostrar menú
	estadoActual = APAGADO;
	Registrador_Update(RST);
}
```

```
void Registrador_Update(MEF_Entrada entrada)
{
	MEF_Salida salida = tablaSalidas[estadoActual][entrada];
	estadoActual = tablaEstados[estadoActual][entrada];

	switch(salida)
	{
		case START_CLOCK:
			RELOJ_Start();
			hacerPedido();
			break;
		case STOP_CLOCK:
			RELOJ_Stop();
			tickCount = 0;
			break;
		case RESET:
			RELOJ_Stop();
			tickCount = 0;
		case SHOW_MENU:
			GUI_EscribirMenu();
		case NONE:
			break;
	}
}

void Registrador_Tick()
{
	// Si transcurrieron 5 ticks = 1 segundo
	if (++tickCount == SECOND_COUNT)
	{
		// Reiniciar contador de ticks
		tickCount = 0;

		// Actualizar datos
		hacerPedido();
	}
}

// ----------------- Funciones privadas -----------------------

static void hacerPedido()
{
	uint8_t estado, checksum, h_ent, h_dec, t_dec;
	int8_t t_ent;

	// Iniciar comunicación con sensor (18 ms, 190 us)
	estado = DHT_StartRequest();

	if (estado == LISTO) {
		// Recibir los 5 bytes de datos
		h_ent = DHT_ReceiveData();
		h_dec = DHT_ReceiveData();
		t_ent = DHT_ReceiveSignedData();
		t_dec = DHT_ReceiveData();
		checksum = DHT_ReceiveData();

		// Comprobar integridad de datos y mostrar valores
		if (checksum == h_ent + h_dec + (uint8_t) t_ent + t_dec) {
			GUI_EscribirDatos(h_ent, t_ent, t_dec);
		} else GUI_EscribirError("--> Error de lectura");
	}
}
```

reloj.c

```
/*
 * Created: 7/6/2022 00:29:55
 * Author: Calderón Sergio
 */

// Archivo de cabecera
#include "reloj.h"


// -------- Implementación de funciones públicas ---------------

void RELOJ_Init()
{
	// Deshabilitar todas las interrupciones de Timer
	TIMSK1 &= ~((1<<TOIE1)|(1<<OCIE1A)|(1<<OCIE1B));

	// Asignar Modo CTC, preescaler de 0 (detenido)
	TCCR1B = (1<<WGM12);

	// Asignar valor para la comparación
	// 3125 x 1024 / 16 MHz = 200 ms
	OCR1A = 3124;

	// Reiniciar contador a cero
	TCNT1 = 0;

	// Habilitar interrupción de comparación
	TIMSK1 |= (1<<OCIE1A);
}

void RELOJ_Start()
{
	// Asignar preescaler de 1024
	TCCR1B |= (1<<CS10)|(1<<CS12);
}

void RELOJ_Stop()
{
	// Asignar preescaler de cero
	// Se limpian los bits asignados para el prescaler de 1024
	TCCR1B &= ~((1<<CS10)|(1<<CS12));

	// Reiniciar contador a cero
	TCNT1 = 0;
}
```

serialPort.c

```
/*
 * Created: 07/10/2020 03:02:18 p. m.
 * Author: Calderón Sergio
 */

// Archivo de cabecera
#include "SerialPort.h"
```

```
// --------------- Funciones públicas ---------------------

// Inicialización de Puerto Serie
void SerialPort_Init(uint32_t f_cpu){

	// Deshabilitar receptor, transmisor e interrupciones
	UCSR0B = 0;

	// Definir tamaño de dato de 8 bits (8 bit data)
	UCSR0C = (1<<UCSZ01) | (1<<UCSZ00);

	// Para la frecuencia de CPU actual, configurar a 9600 bps, 1 stop
	if (f_cpu == 16000000UL) UBRR0L = 103;
	else if (f_cpu == 8000000UL) UBRR0L = 51;
	else if (f_cpu == 4000000UL) UBRR0L = 25;
}


// Inicialización de Transmisor

void SerialPort_TX_Enable(void){
	UCSR0B |= (1<<TXEN0);
}

void SerialPort_TX_Interrupt_Enable(void){
	UCSR0B |= (1<<UDRIE0);
}

void SerialPort_TX_Interrupt_Disable(void)
{
	UCSR0B &=~(1<<UDRIE0);
}


// Inicialización de Receptor

void SerialPort_RX_Enable(void){
	UCSR0B |= (1<<RXEN0);
}

void SerialPort_RX_Interrupt_Enable(void){
	UCSR0B |= (1<<RXCIE0);
}


// Transmisión

void SerialPort_Send_Data(uint8_t data){
	UDR0 = data;
}


// Recepción

uint8_t SerialPort_Receive_Data(void){
	return UDR0;
}
```