Taller de proyecto I (E0305)
Facultad de Ingeniería
Universidad Nacional de La Plata

FACULTAD DE INGENIERÍA

Trabajo práctico 4

Despertador Inteligente

Adra Federico (02090/4)
Alfonsín Jerónimo (02003/7)
Reinoso Francisco (01981/8)

14 de febrero de 2022

Índice

Despertador Inteligente

Enunciado del problema

Requerimientos mínimos para el proyecto grupal:

a. El proyecto Grupal será de hasta 3 alumnos y tendrá calificación.
b. El tiempo de ejecución de los proyectos se estima en 3 semanas, teniendo en cuenta que luego del receso de verano tendremos 2 semanas de febrero de 2022.
c. El alcance del proyecto estimado en horas se considera de 4hs semanales por cada integrante durante el periodo de ejecución.
d. Dispositivos principales a utilizar: placa blue-pill (MCU STM32F103) y 1 o más dispositivos (analógicos o digitales) disponibles en Proteus.
e. Presentación de los objetivos del proyecto para su aval a través del aula virtual: 1 carilla, formato pdf, (ver ejemplo más abajo)
f. Diseño del firmware, simulación y depuración: desarrollar el firmware para el MCU que permita implementar la solución al problema planteado. Realizar la simulación con Proteus utilizando un esquema simplificado (no tiene que utilizar el circuito esquemático completo con el cual diseñará el PCB).
g. Diseño de hardware: como los trabajos prácticos anteriores, diseñar el esquema eléctrico completo con todos los componentes y sus conexiones, luego diseñar el circuito PCB del sistema implementado.
h. Realizar en el informe una descripción detallada de la arquitectura del firmware, de la modularización, y de la forma en que aborda la solución. Mostrar en el informe como validó el diseño. Por otro lado, explicar cómo realizó el PCB y adjuntar imágenes de la capa superior (capa de componentes), capa inferior (capa de soldadura) y vistas 3D del circuito completo.
i. Adjuntar en el aula virtual un .zip con los proyectos.

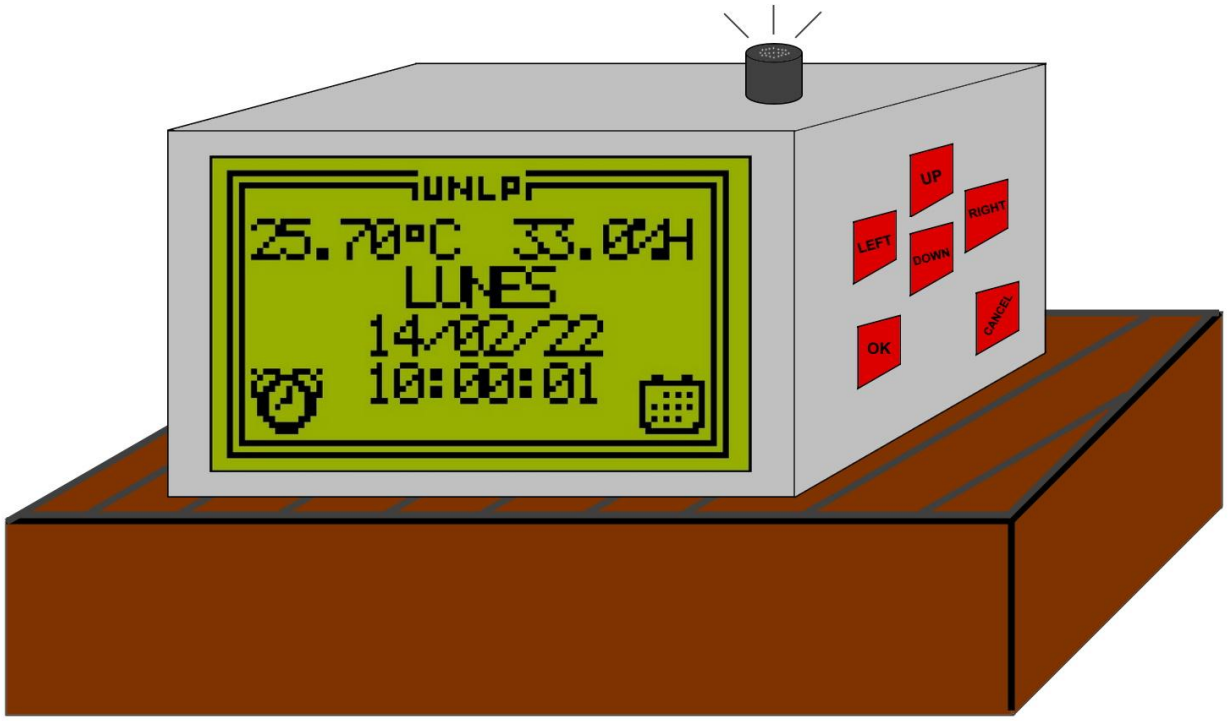
Objetivos:

Se desea realizar un despertador inteligente, que muestre la fecha y hora, junto con la temperatura y humedad en un display gráfico. El despertador sonará a la hora que sea configurado en un buzzer activo genérico (en simulación usaremos el del Proteus, pero tenemos pensado en otra iteración reemplazarlo por un speaker para poder reproducir audios). Para medir la temperatura y humedad se usará el sensor DHT22cuya precisión nos parece aceptable teniendo en cuenta que la aplicación principal es un despertador inteligente (±2°C para medir temperatura y ±5%RH para medir humedad). Para el display gráfico se va a usar en Proteus el Ampire128x64 que tiene un controlador KS0108, para el cual encontramos una librería de drivers implementada que dice funcionar en la familia cortex M3.

Por último, el RTC que guardará información de en qué segundo, minuto, hora, día, mes y año estamos, será un DS1307.

El despertador permitirá configurar la fecha y hora, agregar una alarma (tal vez más de una si llegamos) y, cuando llegue la hora indicada, hacer sonar el buzzer hasta que se presione algún botón con el que la alarma se detendrá.

Respecto a cómo el usuario interactúa con el sistema hemos pensado agregar 5 o 6 botones (arriba, abajo, izquierda, derecha, ok, cancelar) y hacer un menú amigable con el mismo, tomando inspiración de los smartwatches de hoy en día.


UNLP
25.70ºC 33.0%H
LUNES
14/02/22
10:00:01
UP
RIGHT
LEFT
DOWN
CANCEL
OK

1.Interpretación

En este trabajo práctico desarrollaremos un sistema que todos hemos utilizado en algún momento, se trata de un reloj despertador. Claramente, no se trata simplemente de un reloj despertador (ya que el sistema sería demasiado sencillo), sino que a su vez, cuenta con un display gráfico donde se muestra la hora, la fecha, la temperatura y la humedad actual. Además de todo esto, cuenta con la posibilidad de establecer una alarma a la hora deseada por el usuario.
Para el desarrollo de dicho proyecto, utilizaremos los siguientes componentes:

- Un microcontrolador STM32F103C8 (encapsulado bluepill para el PCB modelo C6 para el Proteus)
- Display gráfico Nokia 5110 LCD (modelo: PCD8544)
- Sensor de temperatura y humedad DHT22
- Dispositivo RTC, DS1307
- Seis botones genéricos de 4.5x6mm
- Un buzzer zumbador piezoeléctrico de 3.3V
- Varios componentes electrónicos tales como resistencias, que servirán para conformar el RTC
- Finalmente: cables para el conexionado de los componentes.

Programaremos el microcontrolador utilizando lenguaje C y el compilador de Keil. Para hacer la simulación del sistema en sí, utilizaremos Proteus.
En lo que respecta al desarrollo, realizaremos una solución modular, encapsulando así, la funcionalidad de cada componente, como la comunicación entre los dispositivos.
Para esto, implementaremos librerías, las cuales se comunicarán por medio de funciones públicas definidas en cada una de ellas. Esto, nos exigirá llevar una documentación en cada librería (qué hace cada función, sus constantes) mientras que nos permitirá llegar a una solución fácil de entender, depurar y modificar.
Además de tener un sistema funcional, uno de los objetivos de este trabajo es realizar el circuito impreso (PCB), que tendrá una sección dedicada en el informe explicando todos los procedimientos seguidos, y mostrando el resultado obtenido.

1.1 Comentarios antes de arrancar

En este trabajo se nos presentaron muchísimas dificultades y todas fueron directa o indirectamente culpa de las limitaciones del simulador Proteus. Si bien es una herramienta muy potente y ayudó muchísimo en nuestro aprendizaje de los conceptos básicos de microcontroladores. Con este proyecto quedaron más que demostradas las numerosas limitaciones que nos presenta trabajar con simuladores. Tal es así que tuvimos no sólo que cambiar muchos de los periféricos que en un principio pensábamos usar (otro display gráfico, un LED en lugar del buzzer) si no que llegamos a tener que implementar a mano un periférico del microcontrolador (el I2C) debido a que no funcionaba directamente.
Es claro que al no disponer de la experiencia previa, no teníamos manera de saber si los problemas que fuimos encontrándonos venían de Proteus o de nuestra incapacidad de programar y configurar los periféricos. Pero aún así, con perseverancia logramos llegar a un resultado del que estamos satisfechos y que nos encantaría poder probar físicamente por fuera del simulador en algún momento.

1.2 Organización

Desde un principio, tuvimos claro que el Proteus no es perfecto, y antes de hacer nada, armamos un plan de trabajo por fases.
En la primera fase, buscamos implementar librerías para hacer andar individualmente cada periférico en el microcontrolador en un proyecto de Proteus. Para el RTC, poder establecer fecha y hora y leerla, para el sensor de temperatura y humedad leer estas variables, para el display gráfico ver cómo imprimir caracteres, imágenes, y así. Esta fase fue crucial para poder elegir correctamente qué periféricos usar (aquellos que funcionen en el simulador) y verificar que los módulos del microcontrolador funcionaran correctamente.
Una vez conseguido eso, la segunda fase consta en llevar las librerías de cada periférico a un proyecto principal en donde implementamos el firmware y hacemos los ajustes necesarios para que todo funcione, el modelado del sistema y el agregado de botones y buzzer. Aquí ya como sabemos con cuales periféricos vamos a trabajar podemos en paralelo ir armando esquemático y PCB y dividir mucho mejor las tareas.

2.Resolución

2.1 Modelado del sistema

Como mencionamos en el apartado de la interpretación, el sistema a desarrollar básicamente es un reloj despertador. Para el diseño de la arquitectura del sistema, pensamos en utilizar el concepto de máquina de estados finitos. Más específicamente, intentamos pensar al sistema como una máquina de Moore. De esta manera, identificamos que tendríamos cuatro estados correspondientes a las cuatro funcionalidades principales del despertador. El estado por defecto, el IDLE, en donde se mostraría la fecha, hora, temperatura y humedad; el estado en donde el usuario configura la fecha y hora, SET_FECHA_Y_HORA, el estado en donde el usuario configura la hora de la alarma SET_ALARMA y, por último, el estado en donde la alarma está sonando, IDLE_SONANDO.
Esta manera muy general de ver al sistema, nos permite aislar cada parte del mismo y poder enfocarnos en cada funcionalidad por separado.
Como toda MEF, además de estados hay entradas y salidas. Particularmente, si estamos pensando en un modelo de Moore, vamos a decir que la salida, junto con el estado actual van a depender las entradas. Con esto en mente, podemos pensar en lo que mostramos en el display gráfico junto con el buzzer (modelado como LED en Proteus) como la salida de nuestro sistema y a los botones, el resultado del sensor de temperatura y humedad, la hora actual y la hora de la alarma como entradas de nuestro sistema.
Así, cada uno de los cuatro estados principales tiene una función bien definida. Sin embargo, cabe aclarar que no estamos siguiendo al pie de la letra lo que es una MEF. Por ejemplo: dentro del estado IDLE, al cambiar la temperatura (que hemos dicho que era entrada) podría debatirse que va a cambiar la salida (el número asociado a la temperatura va a cambiar en el display) o podría decirse que la salida sigue siendo la misma ya que estamos "mostrando la temperatura". Esta segunda interpretación es la que decidimos tomar con el fin de hacer el diagramado del sistema más sencillo. Un análisis en profundidad requeriría ponerse a declarar subestados para cada estado con cada posible valor en la salida del display gráfico. Al hacerlo de nuestra manera, evitamos que el informe se llene de diagramas y máquinas de subestados (pero no podíamos dejar pasar la aclaración). Veremos entonces, cómo funcionan cada uno de los cuatro estados.
El estado IDLE (estado por defecto), va a estar mostrando la fecha y hora en todo momento, junto con la temperatura y humedad. Puede verse una captura de este estado en la figura 2.1.1.

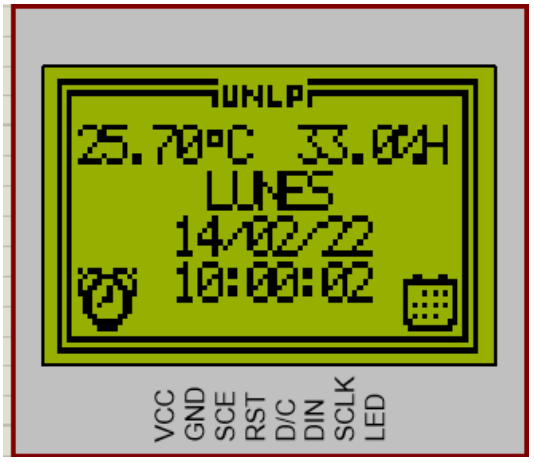


Figura 2.1.1 - Estado IDLE

Si desde el estado IDLE presionamos el botón de derecha transicionamos al estado SET_FECHA_Y_HORA que, como su nombre lo indica, es el estado en donde podemos configurar la fecha y la hora. En la figura 2.1.2 puede verse la pantalla de configuración de fecha y hora (que

más adelante explicaremos junto con la librería de la MEF). Desde este estado, podemos volver al estado IDLE presionando los botones OK o CANCELAR. Presionar OK va a guardar nuestros cambios y presionar CANCELAR va a descartarlos.

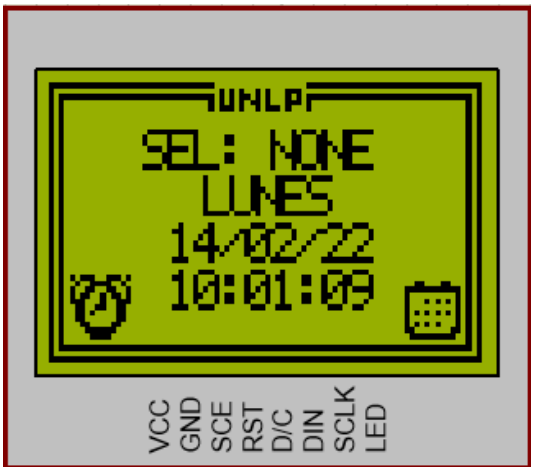


Figura 2.1.2 - Estado SET_FECHA_HORA

Desde el estado IDLE también podemos presionar el botón izquierdo para pasar al estado SET_ALARMA. En este estado, podemos configurar la hora de la alarma y, al igual que el estado de establecer fecha y hora, si se presiona OK se guardará la hora de la alarma. Al presionar CANCELAR se descartan los cambios. Ambos botones nos devuelven al estado IDLE. En la figura 2.1.3 puede verse el display al estar en este estado

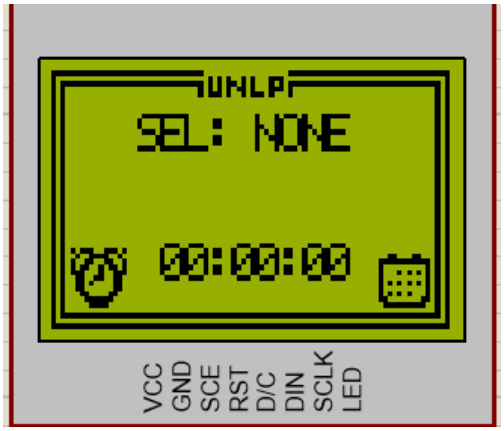


Figura 2.1.3 - Estado SET_ALARMA

Por último, al ocurrir que la hora actual es igual a la hora de la alarma, y estar en el estado IDLE, transicionamos al estado IDLE_SONANDO, en donde se muestra una pantalla con el logo de la UNLP y empieza a sonar el buzzer (que en esta simulación establecimos como un LED ya que no logramos hacerlo sonar). Desde este estado, presionando cualquier botón, se pasará al estado IDLE y se desactivará el sonido del buzzer (el led). En la figura 2.1.4 podemos ver el display en este estado.

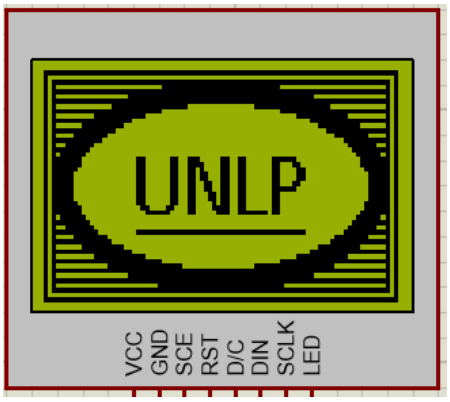


Figura 2.1.4 - Estado IDLE_SONANDO

En la siguiente figura, la 2.1.5 se puede ver el diagrama de estados de la MEF antes descrita

Descripción ESTADO: Salida

- IDLE: Muestra FFHH T° y H% y alarma OFF
- IDLE_SONANDO: Muestra logo UNLP y alarma ON
- SET F&H: Para setear F&H y alarma OFF
- SET ALARMA: Setea hora (de la alarma) y alarma OFF

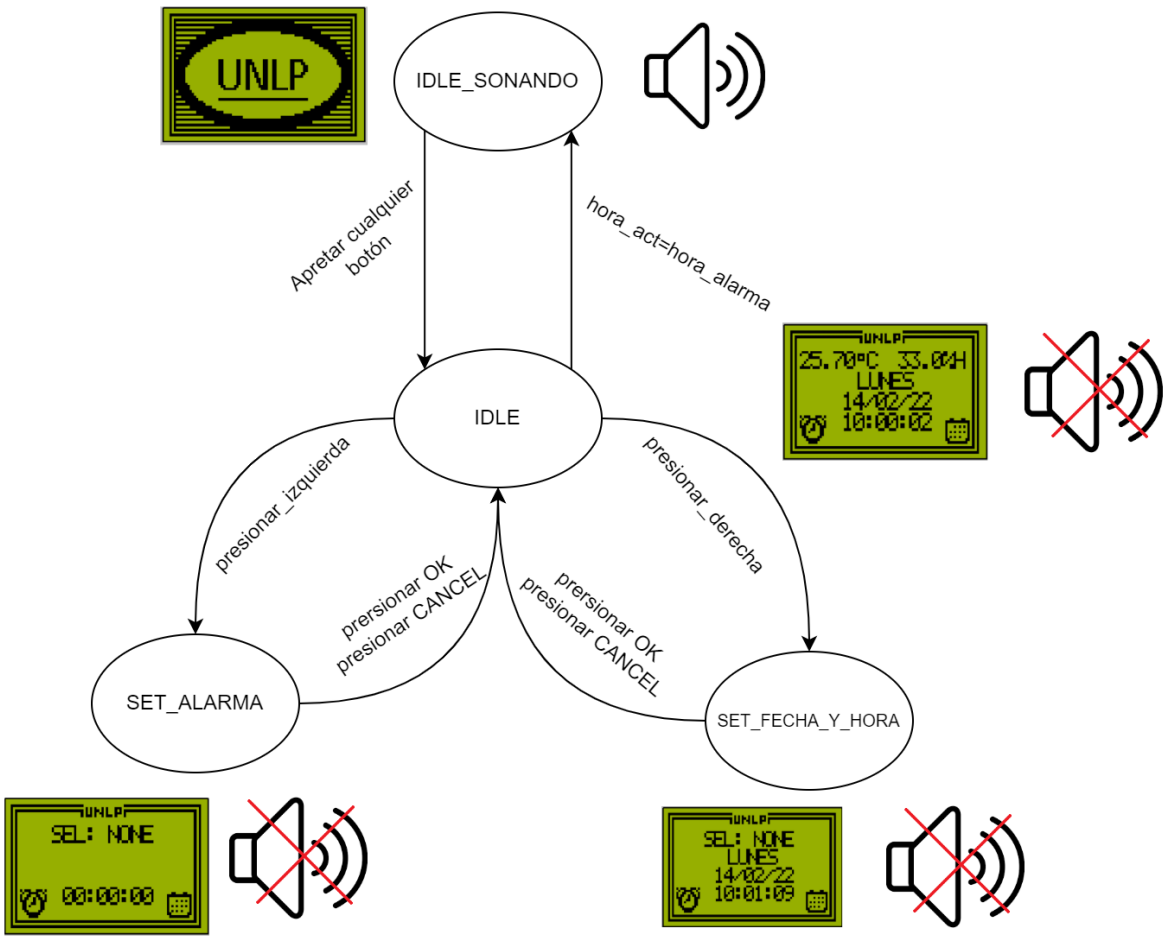


Figura 2.1.5 - Diagrama de estados MEF

2.2 Periférico I2C

En esta sección vamos a explicar cómo implementamos la librería del I2C, necesaria para el funcionamiento del Real Time Clock que decidimos usar.
Nos gustaría decir que la implementación de la librería fue sencilla, pero la realidad es que nos demoró mucho más de lo esperado. El problema es que Proteus no simula el I2C del STM32F103C6 y por mucho que intentamos, no pudimos pasar del start del protocolo I2C. Luego de mencionarlo a un estudiante amigo que cursó el año anterior la materia, nos informó que se enfrentó al mismo problema con el Proteus y que optó por implementar vía software el protocolo.
Con esto dicho, decidimos implementar a mano, por software, el protocolo I2C, usando los pines del microcontrolador y estableciendo los niveles a mano.
El protocolo I2C es un estándar de comunicación muy potente que hemos visto en la materia de Circuitos Digitales y Microcontroladores. Repasando su funcionamiento, consta de dos cables, uno de datos (SDA, Serial Data) y otro de reloj (SCL, Serial Clock) que se extienden por toda la red en donde se quiera realizar la comunicación. Cada uno de estos cables está conectado a su vez en serie con una resistencia de pull-up (de 3.3kΩ en nuestro caso) a Vcc (que será 3.3V, la alimentación de nuestro sistema).
Este protocolo define dos roles en la comunicación:

- Maestro: quien comienza la comunicación, elige el esclavo, se encarga de controlar la línea del reloj (SCL) y envía o recibe la data.
- Esclavo: quien responde al maestro y se identifica con una dirección de 7 bits.

Para nuestra implementación, basta con saber que el microcontrolador tomará el rol de maestro y el RTC el de esclavo (como indica en su datasheet, ver figura 2.3.1). Entonces, en nuestra librería vamos a tener control total sobre el pin asociado al SCL y vamos a tener que ir leyendo y escribiendo en el pin SDA.
Antes de empezar una comunicación, debemos configurar los pines del microcontrolador, y antes de ello, elegir qué pines vamos a usar. En la librería “i2c.h” tenemos varias macros definidas para configurar esto, al igual que varias funciones. Vamos a verlo en código:

```
#ifndef i2c_h
#define i2c_h
#include <stm32f103x6.h>
#include "utils.h"

#define I2C_SCL_PIN 6 //Pin de SCL
#define I2C_SDA_PIN 7 //Pin de SDA
#define I2C_DR GPIOB->ODR //DirecciÃ³n del registro
#define I2C_DR_IN GPIOB->IDR //DirecciÃ³n del registro de entrada

void I2C_Init(void);
void I2C_SendStart(void);
void I2C_SendStop(void);
unsigned char I2C_SendAddrForWrite(unsigned char);
unsigned char I2C_SendAddrForRead(unsigned char);
unsigned char I2C_SendData(unsigned char);
unsigned char I2C_ReadData(unsigned char);

#endif
```

Como puede verse, las macros I2C_SCL_PIN e I2C_SDA_PIN nos indican el número de pin en el que tenemos los cables de clock y data. Luego, el I2C_DR nos apunta al registro de salida de datos (GPIO**B**->ODR porque usamos el puerto B) y el I2C_DR_IN al registro de entrada de datos (GPIOB->IDR). Además de esto, tenemos los prototipos de las siguientes funciones:

- I2C_Init(): se encarga de inicializar la librería
- I2C_SendStart(): se encarga de enviar el start para iniciar la comunicación
- I2C_SendStop(): se encarga de enviar el stop para terminar la comunicación
- I2C_SendAddrForWrite(): se encarga de enviar el slave address por SDA y avisarle que se va a escribir en el periférico del slave
- I2C_SendAddrForRead(): se encarga de enviar el slave address por SDA y avisarle que se va a leer un dato del periférico del slave
- I2C_SendData(): se encarga de enviar 8 bits de datos por SDA
- I2C_ReadData(): se encarga de recibir 8 bits de datos por SDA

En el archivo de "i2c.c" se encuentran las implementaciones de cada una de estas funciones y un par de macros que nos facilitarán el desarrollo y el entendimiento del código.

```
#include "i2c.h"

#define I2C_set_scl() { I2C_DR |= (1 << I2C_SCL_PIN); } //Poner SCL en alto
#define I2C_clear_scl() { I2C_DR &= ~(1 << I2C_SCL_PIN); } //Poner SCL en bajo
#define I2C_set_sda() { I2C_DR |= (1 << I2C_SDA_PIN); } //Poner SDA en alto
#define I2C_clear_sda() { I2C_DR &= ~(1 << I2C_SDA_PIN); } //Poner SDA en bajo
#define I2C_bus_init() { I2C_DR |= ((1 << I2C_SDA_PIN) | (1 <<I2C_SCL_PIN)); } //Poner ambos
pines en alto
#define I2C_get_sda I2C_DR_IN &  (1 << I2C_SDA_PIN) //Obtener valor de SDA


//Half a bit delay
#define HALF_BIT_DELAY delay_us(5);

//I2C doesn't work on STM32. We implement it by software. It's super ADHOC.
void I2C_Init(){
   //Config GPIO pins
   RCC->APB2ENR |= (0xFC); /* enable clocks for GPIOs */
   //RCC->APB1ENR |= (1<<21); /* enable clock for I2C1 */
   GPIOB->CRL |= 0x77000000; /* configure PA6 and PA7 as GPO. open drain */
   I2C_set_scl();
   I2C_set_sda();
}
```

Como puede verse, acá también tenemos macros, que hacen lo que su nombre indica.

- I2C_set_scl(): establece en alto (1) el valor del pin de SCL
- I2C_clear_scl(): establece en bajo (0) el valor del pin de SCL
- I2C_set_sda(): establece en alto (1) el valor del pin de SDA
- I2C_clear_sda(): establece en bajo (0) el valor del pin de SDA
- I2C_bus_init(): establece ambos pines SCL y SDA en alto.
- I2C_get_sda(): nos permite obtener el valor de SDA (alto o bajo)

- HALF_BIT_DELAY: genera un delay de 5us.

En lo que respecta al Init, tenemos la primer línea que activa los clocks para GPIO, luego la línea en donde seteamos el registro de configuración del puerto para establecer a los pines PA6 y PA7 como General Purpose Output, Open Drain y finalmente las dos líneas en las que ponemos en alto los pines de SCL y SDA.
De aquí en adelante puede comenzarse la comunicación. Antes de explicar esto, vamos a tomarnos un segundo para pensar por qué tienen que estar en open drain los pines. En este modo, como dice en la hoja de datos, "Un cero en el registro de salida activa la N-MOS (conexión a GND) mientras que un uno deja el puerto en Hi-Z (alta impedancia). La P-MOS (conexión directa a Vcc) nunca se activa". Esto significa que podemos o bien establecer manualmente en bajo el estado del pin, o bien dejarlo en alto, y que quede en alta impedancia. En este último caso, recordemos que si no hay ningún otro dispositivo conectado, la resistencia de pull-up en paralelo con Vcc por fuera del micro se encargará de establecer el nivel alto de voltaje en la línea. Sin embargo, también abre la posibilidad a otro dispositivo de poner en bajo el nivel de voltaje. Y así es como funciona el protocolo. Si queremos recibir un dato o leer lo que nos transmiten, debemos fijarnos en qué nivel lógico está el pin (habiéndolo establecido previamente en alto). Si queremos enviar un dato, tomaremos el control de la línea y podremos bajar el nivel de voltaje o dejarlo como está. Por esto el protocolo debe establecer un maestro, un esclavo y tener dos líneas en donde se transmite por un lado la información y por otro el "reloj" (SCL) que indica en qué momento lo que hay por el cable de la información (SDA) es válido.
En el protocolo I2C, los *valores de SDA cambian cuando SCL está en bajo*, y el receptor va a leer SDA en el *flanco descendente* de SCL.
También, se establecen dos condiciones: la condición de inicio (START) y la condición de parada (STOP). La condición de inicio se da cuando SCL está en alto y SDA pasa de valor alto a bajo. La condición de parada se da cuando SCL está en alto y SDA pasa de valor bajo a alto. Nótese que en ambos casos, SCL arranca en alto y que, previo a la condición de inicio (así como después de la condición de parada), SDA también queda en alto (en el Init, dejamos todo listo para una condición de inicio).
Yendo a la transmisión, el paquete que se transmite consta de 9 bits de largo. Los primeros 8 los transmite el transmisor por SDA y el 9no bit se transmite por el receptor, actuando de acknowledge. En la figura 2.2.1 puede verse un diagrama ilustrativo de esto.

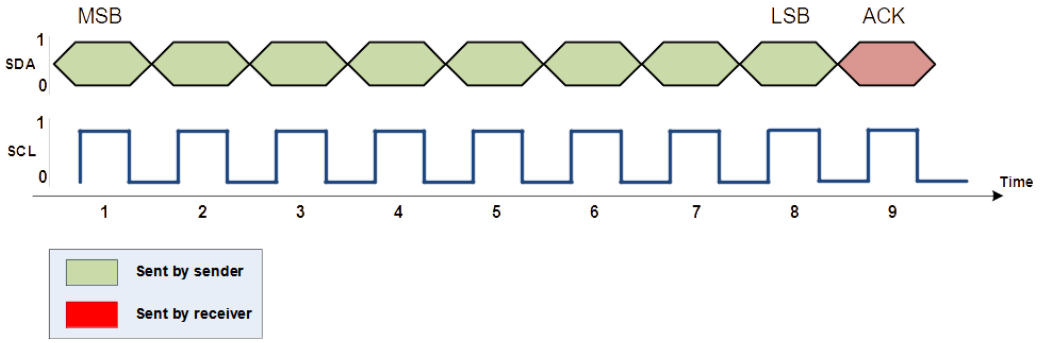


Figura 2.1.1 - Diagrama de comunicación de un paquete

Habiendo explicado todo esto, una comunicación típica con I2C consta de los siguientes pasos (mirándolo desde el master):

- Enviar el Start: esto se hace desde el máster, inicia la comunicación.
- Enviar el Address: el máster envía la dirección del slave con el que se quiere comunicar (7 bits)
- Enviar el bit de R/W: a los 7 bits de address se le agrega el 8vo bit que indica si el master va a enviar o recibir datos del slave (R/W=1 se recibe R/W=0 se envia)

- Esperar el acknowledge del slave, que nos indica que recibió el address+W/R
- Enviar o recibir el byte de datos (según el bit R/W el comportamiento)
- Esperar el acknowledge del receptor.
- Enviar el Stop: esto se hace también desde el master, se genera la condición de parada.

En la figura 2.1.2 puede verse un ejemplo de recibir un byte con el valor 27 en BSS al dispositivo 9.

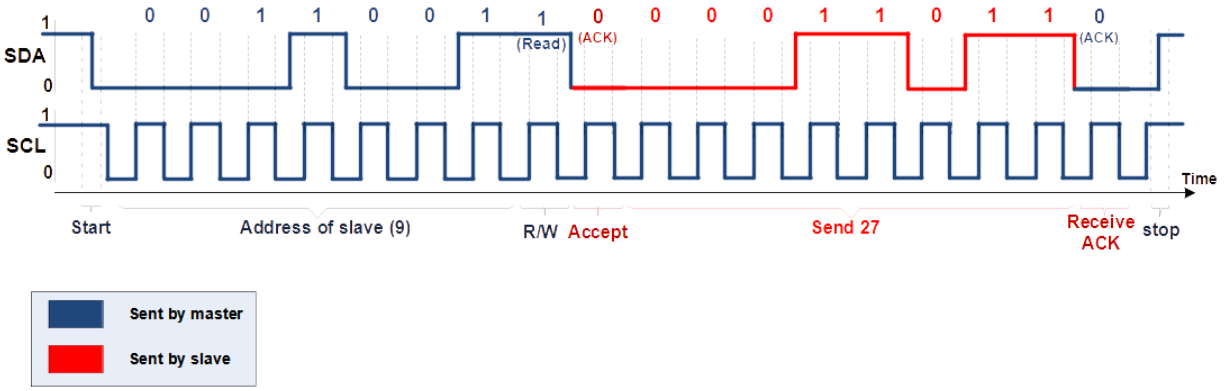


Figura 2.1.2 - Envío de número 27 a dispositivo 9

En todo el tramo rojo, dado que el microcontrolador va a actuar de master, deberemos asegurarnos de tener el pin de SDA en alto ya que en caso de tenerlo en bajo, quedaría forzado y el slave no tendría manera de enviarnos "unos". Además, como hemos dicho, se leerá el SDA en los flancos ascendentes del clock.

Habiendo explicado esto, podemos ir al código y entenderlo mejor. Veamos la función I2C_SendStart():

```
//Send start signal
void I2C_SendStart(){
   I2C_bus_init(); //put both pins on high

   //Wait half bit, put SDA low
         HALF_BIT_DELAY
   I2C_clear_sda(); //SDA goes low

   //Wait half bit, put SCL LOW
   HALF_BIT_DELAY
   I2C_clear_scl(); //SCL goes low
   HALF_BIT_DELAY
}
```

Como puede verse, primero se establecen en alto ambos pines antes de generar la condición de inicio, y se espera por HALF_BIT_DELAY. Ahora tenemos algo de contexto para poder entender que este delay de 5us nos va a ayudar a temporizar correctamente el periférico y corresponde al valor de ½ de lo que va a ser el período de la señal de clock SCL del I2C. Con un período entonces de 10us, estamos hablando de una tasa de transmisión de 100kHZ (lo que se conoce como I2C en modo estándar) que es justamente lo que soporta como máximo el RTC como puede verse en la hoja de datos del mismo (figura 2.3.2).

Luego de establecer ambos pines en alto, se baja primero SDA para generar la condición de inicio y luego SCL, estando listo para meter información en el pin de SDA.

En lo que respecta a I2C_SendStop tenemos el siguiente código:

```
//Send stop signal
void I2C_SendStop(){
   I2C_clear_sda(); //SDA goes low
   I2C_clear_scl(); //SCL goes low
   HALF_BIT_DELAY

   //Wait half bit, put SCL HIGH
   I2C_set_scl(); //SCL goes high
   HALF_BIT_DELAY

   //Wait half bit, put SDA HIGH
   I2C_set_sda(); //SDA goes high
   HALF_BIT_DELAY

}
```

Que como su nombre lo indica genera la condición de parada. Primero pone ambos pines en bajo, luego pone en alto la señal del SCL y finalmente pone en alto SDA.

Una vez generada la condición de inicio, hemos visto que el siguiente paso es enviar el address del slave junto con el bit de read write para indicar con quién nos queremos comunicar y si vamos a leer o escribir en el periférico slave.
Entonces se definen dos funciones, que reciben el address del slave y se encargan de enviar los 8 bits de datos por la línea de SDA. El código puede verse a continuación.

```
//This function will send Address for read
uint8_t I2C_SendAddrForRead(uint8_t addr){
   return I2C_SendData((addr<<1) | 1); //addr + Read(1)
}

//This function will send Address for write
uint8_t I2C_SendAddrForWrite(uint8_t addr){
   return I2C_SendData(addr<<1); //addr+Write(0)
}
```

Se observa que en ambas funciones se hace un llamado a la función SendData y que se retorna lo mismo que retorna el SendData.

En lo que respecta a esta función, se encarga de hacer el envío de un paquete de datos. Como entrada recibe los 8 bits a transmitir y retorna 1 si recibió el ACK y 0 si recibió un NACK del receptor.
El código es:

```
//This function will send data to the slave
//Return 1 if ACK received, 0 if NACK received
```

```
uint8_t I2C_SendData(uint8_t data){
   unsigned char msk = 0x80;     //Mask, starts in 1000 0000
   unsigned char ack;            //Acknowledge received from slave

   //We transmit the 7+1 bits of data
   do
   {
      //We use the mask to check if the bit is 1 or 0
      if (data & msk)  {I2C_set_sda();}
      else {I2C_clear_sda();}

      //Set SCL, then wait half bit and clear SCL, in order for slave to read SDA.
      I2C_set_scl();
      HALF_BIT_DELAY
      I2C_clear_scl();
   }
   while ((msk>>=1) != 0); //Shift mask to the right 1 bit and check if it's 0

   //After sending 8 bits, we set both SDA and SCL, and wait half a bit
   I2C_set_sda();
   I2C_set_scl();
   HALF_BIT_DELAY

   //Here, if ACK is sent by slave, SDA will be drained to low, otherwise it will be high
   //So we read it.
   ack =(I2C_get_sda);

   //Finally, we clear SCL before leaving function
   I2C_clear_scl();
   HALF_BIT_DELAY

   //We return true if sending data was successful. False otherwise.
   return (!ack);
}
```

Puede verse que se usa una máscara inicializada en 0b10000000 que se va desplazando bit a bit a la derecha para tomar cada bit de datos a transmitir. Se envía el paquete desde el MSB al LSB. Si el bit a enviar es un uno (1) se pone en alto SDA y si es un cero (0) se pone en bajo SDA. Luego se genera el flanco ascendente del clock poniendo en alto el pin SCL (aquí el receptor va a tomar el valor de SDA), se espera 5us y se pone en bajo nuevamente. Finalmente se desplaza la máscara y comienza la transmisión del siguiente bit.

Antes de salir de la función, se ponen en alto tanto SDA como SCL, hace el delay y se lee el valor de SDA que nos indicará si el slave receptor recibió correctamente los 8 bits. Luego, se deja el clock en bajo preparándonos para el siguiente bit a transmitir y se retorna el ACK leído negado de tal manera que si recibimos un 0 (si el receptor nos puso en bajo SDA indicando la correcta recepción) retornamos un 1 (que nos representa “verdadero/true”).

Para recibir data el código es muy similar:

```
//This function will read data from the slave
//It receives ack which is used in burst reading to know when to stop
//Data read is returned
uint8_t I2C_ReadData(uint8_t send_nack){
    unsigned char msk = 0x80;     //Mask, starts in 1000 0000
    unsigned char b = 0;          //Buffer for the received data

    //Receive bit by bit
    do
    {
        //Set SCL and wait half bit
        I2C_set_scl();
        HALF_BIT_DELAY

        //We read SDA, if it is 1 we set the bit in the buffer (using mask to know which bit)
        if(I2C_get_sda)  b|=msk;

        //Clear SCL and wait half bit for next bit
        I2C_clear_scl();
        HALF_BIT_DELAY
    }
    while ((msk>>=1) != 0); //We shift mask to the right 1 bit and check if it's 0 in order to read every bit.

    //We check ack parameter to know if we need to send ACK or NACK
    if(send_nack == 0) //We send ACK (still not finished reading)
    {
        I2C_clear_sda();
    }
    else //send_nack = 1, We send NACK (notify slave we end read )
    {
        I2C_set_sda();
    }

    //Set SCL to send ACK/NACK (then wait half bit)
    I2C_set_scl();
    HALF_BIT_DELAY
    //Clear SCL to finish sending ACK/NACK
    I2C_clear_scl();
    //Free SDA line. Then wait half bit we are ready for next operation
    I2C_set_sda();
    HALF_BIT_DELAY

    //Return read data
    return (b);
}
```

De nuevo, tenemos la máscara y agregamos una variable para guardar el paquete recibido.
Acá vamos a manejar el reloj e ir leyendo de SDA el valor del bit para ir armando el paquete. Como puede verse, primero se pone en alto el reloj, se espera medio bit, y se lee del pin SDA el bit que nos marca la máscara (que arranca en 0b10000000 y se va desplazando a la derecha en cada lectura de bit). Luego de leer el bit, se pone en bajo el SCL y se espera 5us para que el emisor nos establezca el valor de SDA y podamos leerlo en la siguiente iteración.
Al final de toda la recepción nos toca enviar el ACK o NACK dependiendo de si el parámetro recibido en la función era un cero (0) o un uno (1) respectivamente (en la siguiente sección veremos cuándo se desea enviar cada cosa). Esto lo hacemos poniendo en alto o bajo el pin de SDA y luego, generando el flanco ascendente en el reloj poniendo SCL en alto. Para terminar, luego de 5us, se pone en bajo SCL y en alto SDA, se esperan otros 5us (dejando todo listo para continuar con la comunicación) y se retorna el dato leído.

Una última cosa a tener en cuenta de esta librería, es que todas las funciones son bloqueantes ya que usan el delay_us bloqueante de la librería utils que implementamos (explicada en la sección 2.10). Además, está implementada y optimizada para su uso con el RTC.

2.3 Periférico RTC con I2C

Una vez definida la librería I2C podemos usarla para implementar la librería que se comunica con el RTC desde el micro. Recordemos que hemos optado por usar el DS1307 y lo primero que hemos hecho ha sido leer la hoja de datos del mismo para entender cómo se configura. En la figura 2.3.1 puede verse que este periférico soporta el protocolo I2C y opera como esclavo.

I²C DATA BUS

The DS1307 supports the I²C protocol. A device that sends data onto the bus is defined as a transmitter and a device receiving data as a receiver. The device that controls the message is called a master. The devices that are controlled by the master are referred to as slaves. The bus must be controlled by a master device that generates the serial clock (SCL), controls the bus access, and generates the START and STOP conditions. The DS1307 operates as a slave on the I²C bus.

Figures 3, 4, and 5 detail how data is transferred on the I²C bus.

- Data transfer can be initiated only when the bus is not busy.
- During data transfer, the data line must remain stable whenever the clock line is HIGH. Changes in the data line while the clock line is high will be interpreted as control signals.

Figura 2.3.1 - Información del I2C con el DS1307

Además podemos obtener información sobre la frecuencia del reloj máxima que admite (como puede verse en la figura 2.3.2), sus voltajes de operación, corriente máxima soportada, tiempos, etcétera. Con estas especificaciones técnicas podemos darnos una idea de si es posible utilizarlo con el microcontrolador y qué cuidados hay que tener en el apartado eléctrico de la conexión.

AC ELECTRICAL CHARACTERISTICS

(V_{CC} = 4.5V to 5.5V; T_A = 0°C to +70°C, T_A = -40°C to +85°C.)

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
SCL Clock Frequency	f_{SCL}		0		100	kHz

Figura 2.3.2 - Frecuencia máxima soportada

Una vez verificado que podemos usar el periférico con este microcontrolador operando a 3.3V, vamos a ver cómo opera.
El DS1307 tiene su memoria mapeada en registros de 8 bits como puede verse en la Tabla 2 de la hoja de datos. Representa los números en BCD (Binario codificado en decimal) y a continuación se encuentran las direcciones de memoria junto con el dato que contiene el registro:

- 0x00: Segundos (rango de 0 a 59)
- 0x01: Minutos (rango de 0 a 59)
- 0x02: Horas (soporta formato 12/24 y se switchea entre uno y otro con el bit 6. Vamos a forzarlo a formato 24hs)
- 0x03: Día de semana (rango de 1 a 7)
- 0x04: Día (rango de 1 a 31)
- 0x05: Mes (rango de 1 a 12)
- 0x06: Año (rango de 0 a 99)
- 0x07: Se usa para control, no nos interesa en esta implementación
- 0x08-0x3F: RAM

El slave address del DS1307 es el 0b1101000 o 104 en decimal. Y en la hoja de datos se encuentran explicados los dos modos de operación del mismo periférico

- Slave Receiver Mode (Write Mode): este modo es el de recepción, lo usaremos para configurar la fecha y hora del RTC. Luego de que el master genera la condición de inicio, se realiza por hardware un checkeo del slave address recibido. Se recibe el bit de R/W (que será 0 en este modo de escritura) y posterior a eso se genera el ACK desde el DS1307 en SDA. Entonces, el master transmite los 8 primeros bits de datos, que configurarán el registro puntero en el DS1307 (acá indicaremos la dirección de memoria que queremos configurar, 0x00 para segundos por ejemplo). A esto el DS1307 responde con un ACK y, finalmente, se envían cero o más bytes de datos (todos seguidos por el respectivo ACK del DS1307) para ir estableciendo el valor de cada registro del RTC. El registro puntero se incrementará en cada valor recibido. Para terminar, se genera la condición de parada. En la figura 2.3.3 se encuentra un dibujo que explica cómo es la comunicación.
- Slave Transmitter Mode (Read Mode): en este modo, los primeros pasos son iguales que el anterior; solo que luego del slave address en lugar de enviar el bit R/W en 0 se lo envía en 1, preparando al DS1307 para enviar datos. Luego de hacer esto, el periférico va a empezar a enviar paquetes de 8 bits con el dato almacenado en la dirección que está escrita en el registro puntero, esperando recibir el ACK del microcontrolador (master) luego de cada paquete. El registro puntero se va autoincrementando después de cada transmisión y el DS1307 va a continuar transmitiendo paquetes hasta recibir un NACK del microcontrolador. En la figura 2.3.4 se encuentra un dibujo que explica la comunicación.

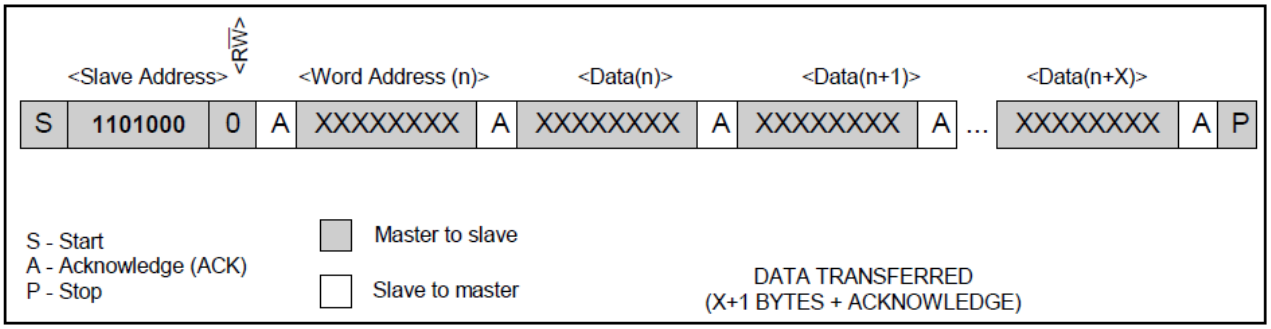


Figura 2.3.3 - Slave Receiver Mode

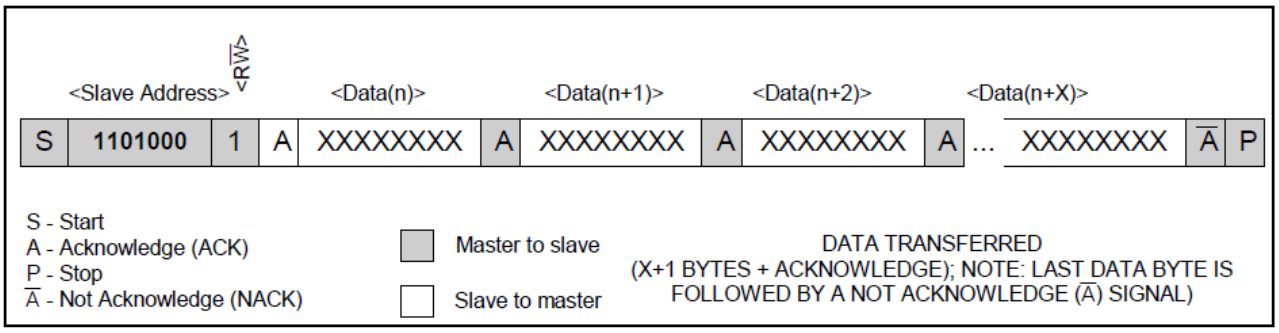


Figura 2.3.4 - Slave Transmitter Mode

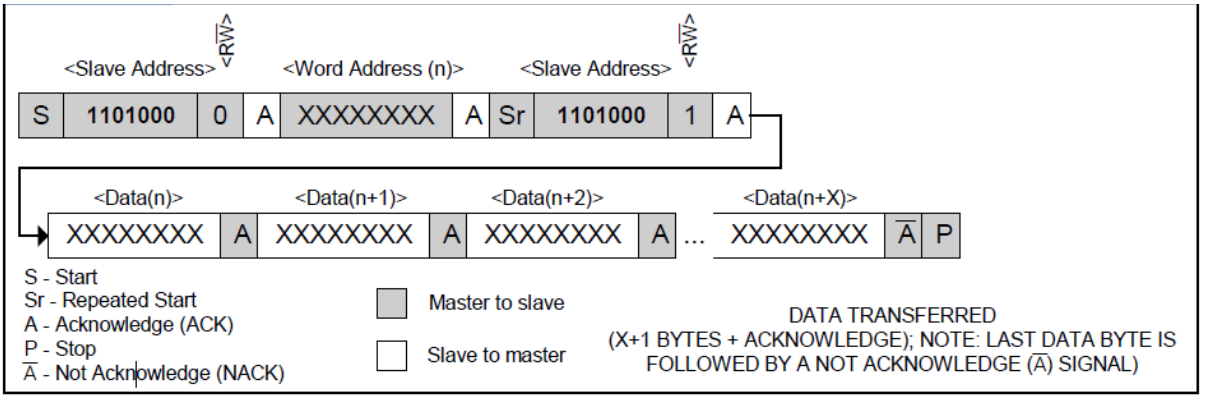


Figura 2.3.6 - Data read (slave receive and then transmit)

Con todo este análisis realizado, estamos listos para ver el código de la librería que controla el periférico. En el archivo cabecera “ds1307.h” se encuentra lo siguiente:

```
#ifndef ds1307_h
#define ds1307_h
#include <stm32f103x6.h>

#include "i2c.h"

#define DS1307_ADDRESS 0x68 // 7 bit address (add )
#define DS1307_SECONDS_REGISTER 0x00
#define DS1307_MINUTES_REGISTER 0x01
#define DS1307_HOURS_REGISTER 0x02
#define DS1307_DAY_REGISTER 0x03
#define DS1307_DATE_REGISTER 0x04
#define DS1307_MONTH_REGISTER 0x05
#define DS1307_YEAR_REGISTER 0x06
#define DS1307_CONTROL_REGISTER 0x07

#define MONDAY 1
#define TUESDAY 2
#define WEDNESDAY 3
#define THURSDAY 4
#define FRIDAY 5
#define SATURDAY 6
```

```
#define SUNDAY 7


typedef struct{
    uint8_t seconds;
    uint8_t minutes;
    uint8_t hours;
}RTC_TIME_t;

typedef struct{
    uint8_t day;
    uint8_t date;
    uint8_t month;
    uint8_t year;
}RTC_DATE_t;


void DS1307_Init(void);
void DS1307_set_time(RTC_TIME_t *time);
void DS1307_get_time(RTC_TIME_t *time);
void DS1307_set_date(RTC_DATE_t *date);
void DS1307_get_date(RTC_DATE_t *date);
void DS1307_set_full_time(RTC_TIME_t *time, RTC_DATE_t *date);
uint8_t * DS1307_get_time_string(RTC_TIME_t time);
uint8_t * DS1307_get_date_string(RTC_DATE_t date);
uint8_t * DS1307_get_date_string_with_day(RTC_DATE_t date);

//Utilitary to compare dates and times
uint8_t DS1307_dateEquals(RTC_DATE_t *date1, RTC_DATE_t *date2);
uint8_t DS1307_timeEquals(RTC_TIME_t *time1, RTC_TIME_t *time2);
void DS1307_copyTime(RTC_TIME_t *dest, RTC_TIME_t *src);
void DS1307_copyDate(RTC_DATE_t *dest, RTC_DATE_t *src);
#endif
```

Como pueden verse, hay varias macros definidas. Entendemos que los nombres son autoexplicativos, guardamos el slave address del periférico, las direcciones de los registros que tienen los segundos, minutos, horas, día de semana, día, mes, año, y el de control.
Luego, tenemos los días de las semanas definidos como números del 1 al 7 para respetar la manera en la que se guardan en el DS1307.
Abajo de esto, se encuentran definidas dos estructuras RTC_TIME_t para guardar tiempo (horas minutos y segundos) y RTC_DATE_t para guardar fechas (día de semana, día, mes, año).
Finalmente se encuentran las funciones principales de la librería que explicaremos a continuación.

DS1307_Init() inicializa la librería. Hace el llamado a dos funciones. Primero a I2C_Init() de la librería del I2C que hemos importado en la cabecera y analizado en el anterior apartado, luego al DS1307_config() que es una función privada de la librería. Este es el código:

```
void DS1307_Init(void)
{
   I2C_Init();
   DS1307_config();
}
```

La función DS1307_Config() se encarga de inicializar el periférico utilizando las funciones de la librería I2C para la comunicación. Primero envía una condición de inicio, y el slave address junto con el bit de read en 0, para poder escribir en los registros del RTC. Puede verse que se van a generar tantas condiciones de inicio y envíos de slave address como sean necesarias hasta que el periférico reaccione y nos envíe un ACK. La función es bloqueante.
Luego, de recibir el ACK envíamos el valor que queremos setear en el registro de dirección (0x00 para arrancar con los segundos). Y finalmente, enviamos de corrido todos los datos (acá no hacemos chequeos de si se está enviando correctamente la información porque nunca tuvimos problemas pero podría ser una futura mejora de la librería) recibiendo los ACKs correspondientes. El resultado final es tener al RTC configurado para marcar las 10hs (formato configurado de 24hs) del lunes 14 de febrero del 2022. Por último, enviamos la condición de parada (STOP).
El código es el siguiente:

```
//private functions
static void DS1307_config(){
   do
         {
               I2C_SendStart();                                          /* generate a start condition */
         } while (I2C_SendAddrForWrite(0x68) == 0); /* repeat if returned false */

         I2C_SendData(0x0);      /* set addr. pointer to 0 */
         I2C_SendData(0x00); /* second */
         I2C_SendData(0x00); /* min */
         I2C_SendData(0x0A); /* hour in 24-h format*/
         I2C_SendData(MONDAY); /* day of week */
         I2C_SendData(0x14); /* day of month */
         I2C_SendData(0x02); /* month */
         I2C_SendData(0x22); /* year */
         I2C_SendStop(); /* generate a stop condition */
}

void DS1307_get_full_time(RTC_TIME_t *time, RTC_DATE_t *date)
{

         if (date == NULL || time == NULL)
               return;

   DS1307_get_date(date);
   DS1307_get_time(time);
}
```

La función DS1307_get_time() recibe un puntero a la estructura que almacena tiempo RTC_TIME_t.
Su implementación es la siguiente:

```
void DS1307_get_time(RTC_TIME_t *time)
{
   if (time == 0)
      return;

   do
   {
      I2C_SendStart();                    /* generate a start condition */
   } while (I2C_SendAddrForWrite(0x68) == 0); /* repeat if returned false */

   I2C_SendData(0x00); /* set addr. pointer to 0 */

   do
   {
      I2C_SendStart();                    /* generate a REPEATED START condition */
   } while (I2C_SendAddrForRead(0x68) == 0); /* repeat if returned false */

   time->seconds = bcd2int(I2C_ReadData(0));
   time->minutes = bcd2int(I2C_ReadData(0));
   //Clean hours 6th bit by masking it
   time->hours = bcd2int(I2C_ReadData(1) & 0xBF); //send nack

   I2C_SendStop(); /* generate a stop condition */
}
```

Como puede verse, primero se descarta el caso en el que el puntero a la estructura no haya venido inicializado, después se genera la condición de inicio (START) y se espera a que el RTC responda con un ACK al enviarle el slave address y el bit R/W indicando modo escritura.
Una vez que nos responde, vamos a enviarle la dirección para que el registro puntero apunte a la dirección del registro de segundos. Luego, generamos una condición de repeated start (como indica la figura 2.3.6) y enviamos el slave address con el bit de R/W seteado para leer (nos quedamos esperando el ACK). Una vez hecho esto, estamos listos para leer los datos que tenga para mandarnos el RTC que serán los segundos, luego los minutos y finalmente la hora (a la cual le limpiamos el 6to bit que es el del formato de hora). No debemos olvidarnos que recibiremos los datos en BCD por lo que hay que transformarlos a int antes de guardarlos. Para esto, se usa la función privada bcd2int() que recibe un BCD de 8 bits y retorna un BSS de 8 bits. Su código puede verse a continuación, junto con la función inversa int2bcd() que recibe un BSS de 8 bits y retorna un BCD de 8 bits.

```
/* The function gets a BCD number and converts it to binary */
static uint8_t bcd2int(uint8_t n)
{
        return ((n & 0xF0) >> 4) * 10 + (n & 0x0F);
}

static uint8_t int2bcd(uint8_t n)
{
   return ((n / 10) << 4) + (n % 10);
}
```

Tampoco debemos olvidarnos que en la última lectura debemos avisarle al DS1307, enviando un NACK (este es el último read_data, el de la hora, que se le pasa un 1 de parámetro). Por último, antes de retornar, generamos la condición de parada: enviamos el STOP.

Una lógica muy similar está implementada en el DS1307_get_date() con las diferencias de que se recibe un puntero a la estructura que almacena fechas y que el primer write que hacemos, en donde indicamos el registro de dirección a leer, es el 0x03 (el del día de semana). El código es:

```
void DS1307_get_date(RTC_DATE_t *date)
{
   if (date == 0)
      return;

   do
   {
      I2C_SendStart();                  /* generate a start condition */
   } while (I2C_SendAddrForWrite(0x68) == 0); /* repeat if returned false */

   I2C_SendData(0x03); /* set addr. pointer to 0 */

   do
   {
      I2C_SendStart();                  /* generate a REPEATED START condition */
   } while (I2C_SendAddrForRead(0x68) == 0); /* repeat if returned false */

   date->day = bcd2int(I2C_ReadData(0));
   date->date = bcd2int(I2C_ReadData(0));
   date->month = bcd2int(I2C_ReadData(0));
   date->year = bcd2int(I2C_ReadData(1));

   I2C_SendStop(); /* generate a stop condition */
}
```

Para la función DS1307_set_time() se espera recibir también un puntero a una estructura de tiempo RTC_TIME_t. En esta función, se va a escribir el tiempo (horas minutos y segundos) en el

periférico. Para esto, primero generamos la condición de start, después enviamos el slave address con el bit de R/W en cero y repetimos en caso de que no hayamos recibido el ACK del periférico. Luego, estamos listos para enviarle los segundos, minutos y horas al DS1307 pero no sin antes convertirlos a BCD con la función privada int2bcd que mostramos. Finalmente, generamos la condición de parada (STOP) para terminar la transmisión.

El código que hace esto es:

```
void DS1307_set_time(RTC_TIME_t *time)
{
   do
         {
            I2C_SendStart();                                        /* generate a start condition */
         } while (I2C_SendAddrForWrite(0x68) == 0); /* repeat if returned false */

         I2C_SendData(0x0);      /* set addr. pointer to 0 */
         I2C_SendData(int2bcd(time->seconds) & 0x7F); /* seconds without 7th bit */
         I2C_SendData(int2bcd(time->minutes)); /* min */
         I2C_SendData(int2bcd(time->hours)); /* hour in 24-h format*/
   I2C_SendStop(); //generate a stop condition
}
```

Para la función DS1307_set_date() el código es casi igual, solo que en lugar de recibir un puntero a estructura de tiempo se recibe un puntero a una estructura de fecha RTC_DATE_t y se envían los datos correspondientes a el día de semana, día, mes y año.
El código es:

```
void DS1307_set_date(RTC_DATE_t *date)
{
   do
         {
            I2C_SendStart();                                        /* generate a start condition */
         } while (I2C_SendAddrForWrite(0x68) == 0); /* repeat if returned false */

         I2C_SendData(0x03);    /* set addr. pointer to 0 */
         I2C_SendData(int2bcd(date->day)); /* day of week */
         I2C_SendData(int2bcd(date->date)); /* day of month */
         I2C_SendData(int2bcd(date->month)); /* month */
         I2C_SendData(int2bcd(date->year)); /* year */
         I2C_SendStop(); /* generate a stop condition */
}
```

La función DS1307_set_full_time() recibe dos punteros, uno a una estructura de tiempo y otro a una estructura de fecha RTC_TIME_t y RTC_DATE_t y básicamente llama a las funciones previamente mencionadas que establecen fecha y hora.

```
void DS1307_set_full_time(RTC_TIME_t *time, RTC_DATE_t *date)
{
   DS1307_set_date(date);
   DS1307_set_time(time);
}
```

Las funciones DS1307_get_time_string(), DS1307_get_date_string() y DS1307_get_date_string_with_day() no se usaron en este trabajo, pero básicamente nos devuelven un string que tiene la hora, fecha o fecha con día de semana formateados y reciben de parámetro punteros a estructuras de tiempo o fecha. Hemos optado por hacer el "formatting" de la fecha y hora en el código de la MEF, ya que nos daba mayor flexibilidad.

Por último quedan explicar las últimas cuatro funciones que son utilitarias y usadas en la MEF

- DS1307_dateEquals(RTC_DATE_t *date1, RTC_DATE_t *date2) compara campo a campo date1 con date2 y determina si las fechas son iguales. Retorna 1 en ese caso y 0 en caso contrario
- DS1307_timeEquals(RTC_TIME_t *time1, RTC_TIME_t *time2) compara campo a campo time1 con time2 y determina si las horas son iguales. Retorna 1 en ese caso y 0 en caso contrario
- DS1307_copyTime(RTC_TIME_t *dest, RTC_TIME_t *src): copia campo a campo la estructura src a la estructura dest.
- DS1307_copyDate(RTC_DATE_t *dest, RTC_DATE_t *src): copia campo a campo la estructura src a la estructura dest.

Su código es el siguiente:

```
uint8_t DS1307_dateEquals(RTC_DATE_t *date1, RTC_DATE_t *date2){
   return (date1->day == date2->day && date1->date == date2->date && date1->month == date2->month
&& date1->year == date2->year);
}
uint8_t DS1307_timeEquals(RTC_TIME_t *time1, RTC_TIME_t *time2){
   return (time1->hours == time2->hours && time1->minutes == time2->minutes && time1->seconds ==
time2->seconds);
}

void DS1307_copyTime(RTC_TIME_t *dest, RTC_TIME_t *src){
   dest->hours = src->hours;
   dest->minutes = src->minutes;
   dest->seconds = src->seconds;
}
void DS1307_copyDate(RTC_DATE_t *dest, RTC_DATE_t *src){
   dest->day = src->day;
   dest->date = src->date;
   dest->month = src->month;
   dest->year = src->year;
}
```

2.4 Periférico SPI

La librería SPI fue implementada para utilizarse en conjunto con el display Nokia5110 que se explicará en la siguiente sección. Esta librería tiene solo 3 funciones, spi_init(), void spi_tx(uint8_t tx_char) y void spi_tx_msg(const uint8_t* str,uint8_t). Notar que el SPI 1 será utilizado exclusivamente para enviar datos, y por ende la librería solo implementa esas funciones (ya que es completamente innecesario para nuestro proyecto la lectura por parte del display gráfico).
El archivo de cabecera de la librería entonces es el siguiente

```
/*
Actual setup
SPI - 1
-->
PA4 --> SS
PA5 --> SCLK
PA6 --> MISO
PA7 --> MOSI

SPI2 - 2
PB12 --> SS
PB13 --> SCLK
PB14 --> MISO
PB15 --> MOSI
*/
#ifndef SPI_DRIVE_H
#define SPI_DRIVE_H

#include <stm32f103x6.h>

#define SPI1_SS 4
#define SPI1_SCLK 5
#define SPI1_MISO 6
#define SPI1_MOSI 7

void spi_init(void);
void spi_tx(uint8_t tx_char);
void spi_tx_msg(const uint8_t* str,uint8_t);

#endif
```

Como puede verse, define cuatro macros para identificar los pines que actuarán de SlaveSelect (SS, también conocido como ChipSelect), SCLK (Clock), MISO (Master Input Slave Output) y MOSI (Master Output Slave Input). Luego, tenemos las cabeceras de las funciones mencionadas.

La función spi_init() sirve para la inicialización del SPI del MCU, activa el clock del SPI1, y el de AFIO (Alternate function input output), configura los pines que se usarán para el SPI (el pin de SS en modo output, el de MOSI y CLK como alternate function push-pull y el MISO como input) y los

registros de configuración del periférico SPI para usarlo en Modo Master, BR = 2.25 MHz (36MHz/16). Tener en cuenta que el display gráfico que utilizamos soporta hasta 4MHz MAX.
El código que hace esto es el siguiente:

```
void spi_init()
{
	RCC->APB2ENR |= 1; //Enable AFIO function
	/* SPI1 PINS SETUP CODE*/
	RCC->APB2ENR |= 0x1000; // Enabling the SPI1 periph

	/* PINS SETUP */
	GPIOA->CRL = 0xB2B34444; //PA7 - DIN | PA6 - DOUT | PA5 - CLK | PA4 - NSS

	/* SPI1 PERIPHERAL SETUP CODE*/
	//SPI1 enabled, Master mode, BR = 36MHz/16 = 2.25MHz (GLCD supports up to 4MHz)
	SPI1->CR1 = (1<<2) | (0x18) | (1<<6);
	SPI1->CR2 |= 0x4; //SSOE = 1 (SS output enabled)
	/* SS HIGH */
	spi_set_ss();
}
```

La función spi_tx() nos sirve para transmitir un único carácter con SPI. Siempre antes de una transmisión debemos poner en bajo el SS y después de la misma ponerlo en alto.
Acá aprovechando que el periférico SPI sí funciona en el microcontrolador, se pueden usar los registros de configuración. Como extra, debemos observar que la función es bloqueante. Se queda esperando a que el bit de BSY se limpie.

La función spi_tx_msg nos permite transmitir una cadena de bytes. A diferencia de la anterior función, acá vamos a tener el SS en bajo durante más tiempo, hasta que se termine de transmitir toda la cadena recibida como parámetro. El segundo parámetro que recibe la función indica el tamaño de la cadena a transmitir.

El código de ambas funciones es:

```
/* Transmission of a Single Character by SPI
	spi -> 1 or 2
	tx_char -> Char Data
*/
void spi_tx(uint8_t tx_char)
{
	/* SPI1 SEND CHAR */
	spi_clear_ss(); //CS to low
	SPI1->DR = tx_char; //Data register set
	while ((SPI1->SR & (1<<7))!=0); //Wait until BSY is clear
	spi_set_ss();
}

/* Transmission of a String of databy SPI
	str -> String Data */
```

```
void spi_tx_msg(const uint8_t* str,uint8_t size)
{
	uint8_t i=0;
	/* SPI1 SEND STRING */
	spi_clear_ss(); //CS to low
	while (i++<size)
	{
		SPI1->DR = *str;
		while ((SPI1->SR & (1<<7))!=0); //Wait until BSY is clear
		str++;
	}
	spi_set_ss();
}
```

2.6 Display gráfico con SPI

El display gráfico fue una de las cosas más desafiantes de este proyecto. Por suerte, la comunicación con el mismo se realizó con el SPI que sabíamos que funcionaba correctamente. Además teníamos en internet numerosos videos de gente haciendo andar este display gráfico en Proteus, algunos incluso con el micro elegido, usando el STM32 Development kit.
El Nokia5110 (cuyo controller es el PCD8544) es la pantallita del modelo de Nokia con el mismo nombre. Es de 48x84 píxeles.
En la figura 4 de la hoja de datos se puede ver como están mapeados los registros de RAM y nos da una idea de cómo escribir en cada píxel. En la figura 6, nos muestra cómo se escriben los bytes con el modo horizontal addressing (el que vamos a usar, V=0). En este display entonces, configuramos una coordenada x indicando el número de píxel horizontal a modificar y una coordenada Y indicando cuál de los 6 bancos de bytes vamos a modificar. Luego, la escritura se hace de a bytes, y debe tenerse en cuenta que la coordenada x se autoincrementa posterior a una escritura.
En el apartado 8.1 podemos ver como se debe realizar la inicialización del periférico. Nos dicen que inmediatamente ni bien se prende el dispositivo, debemos enviar un pulso /RES por al menos 100ms para asegurarnos que los registros tengan un valor consistente.
El LCD tiene unos registros de configuración cuyo estado después del reset pueden verse en el apartado 8.2. Para terminar la inicialización, debemos configurar el valor de V_{OP} que nos permite decirle al LCD el voltaje de operación que usaremos. El valor que ponemos en V_{OP} lo podemos determinar con la fórmula de la sección 8.9 y depende del voltaje de operación (que será 3.3V) y la temperatura del ambiente en donde esté el display. En nuestro caso lo hemos calculado como 0x84.
El display gráfico tiene dos modos de instrucciones. Por defecto, vamos a estar en el modo básico, pero para hacer estos cambios de configuración de V_{OP} debemos cambiar al modo extendido.
Cuando enviamos bytes al display, podemos estar queriendo dibujar algo en el display o configurar un registro. Para discriminar entre ambas opciones, tenemos un pin que le indica al display como interpretar lo que le estamos mandando. El pin D/C (llamado D de data y C de command), que pondremos en alto cuando deseemos enviar datos (dibujar) y en bajo cuando estemos enviando un comando. Podemos ver la tabla 1 que nos muestra el conjunto de instrucciones. Principalmente usaremos la de escribir data (write data, D=1) y la que nos permite configurar la dirección X e Y del display (Set X or Y address of RAM, D=0). En el init, se usará también function set y Set V_{OP} para configurar el voltaje de operación.

Vamos a ir explicando a partir del código la configuración de este display gráfico, como se puede ver, las funciones públicas que exponemos al usuario son muy similares a las que teníamos en el LCD. Este es el archivo de cabecera:

```
#ifndef GLCD_H
#define GLCD_H

#include <stm32f103x6.h>
#include "spi_drive.h"
#include "utils.h"

/*MACROS (change for port)*/
#define DISPLAY_PORT GPIOA
#define DISPLAY_PORT_DR GPIOA->ODR
#define DISPLAY_PORT_CONFIG GPIOA->CRL

#define GLCD_RST 0 //Reset pin
#define GLCD_DC 1 //Data/Command pin
#define GLCD_CS 4 //Chip select pin -> also slave select of SPI

#define GLCD_SCK 5   //Clock pin
#define GLCD_DIN 7 //Data in pin

#define GLCD_WIDTH 84
#define GLCD_HEIGHT 6

/* Function Headers */

void GLCD_Init(void);
void GLCD_sendChar(char);
void GLCD_sendString(char *);
void GLCD_setXY(uint8_t x, uint8_t y);
void GLCD_clean(void);
void GLCD_drawImage(uint8_t,uint8_t,uint8_t*,uint8_t,uint8_t);
void GLCD_drawImageNonDestructive(uint8_t,uint8_t,uint8_t*,uint8_t,uint8_t);
```

Como es costumbre, tenemos varias macros que nos permiten modularizar mejor el código. Entendemos que su nombre es autoexplicativo a esta altura del informe.

Las funciones públicas que define se explican a continuación

2.6.1 GLCD_Init()

GLCD_Init() se encarga de inicializar el display gráfico, siguiendo lo que hemos comentado más arriba. Primero llama al spi_init() para inicializar el periférico del micro que usará en la comunicación con el display, configura el puerto de manera que se establezcan los pines conectados al GLCD como salida y no se pise la configuración del SPI. Luego, viene el pulso de /RST y posterior a eso, ponemos el pin de D/C en bajo, para enviar comandos (glcd_sendCommand) que configurarán V_{OP}.
Por último, llamamos a la función con la que pintamos la pantalla de bienvenida en el GLCD (ya la veremos más adelante).
El código para este init es el siguiente:

```
/* Code for init of the Graphic LCD N5110.
It depends on the SPI_drive.h files.
This is not tested for SPI2 but should work with some minor changes!*/
void GLCD_Init(){

	/* DISPLAY CODE */
	RCC->APB2ENR |= 0x4; //Activate GPIOA clock in case it is not activated
	spi_init();
	//MOSI PA7 and SCK as Alternate Function push-pull
	//CE, RESET, D/C as GPIO push-pull 50MHz

	//DISPLAY_PORT_CONFIG = (0x0000000B<<(4*GLCD_DIN) | 0x0000000B <<
(4*GLCD_SCK) | 0x00000003 << (4*GLCD_CS) | 0x00000003 << (4*GLCD_RST) | 0x00000003 <<
(4*GLCD_DC));
	DISPLAY_PORT_CONFIG = 0xB2B34433;

	//Set rst pin in case it was not already set
	GLCD_set_RST();
	delay_ms(10);
	//Reset display (maximum 100ms after power or else it may get damaged)
	GLCD_clear_RST();
	delay_ms(100); //100ns is enough acording to the datasheet
	GLCD_set_RST();

	GLCD_clear_DC();

	//Init the LCD ITS A HOLE SEQUENCE
	glcd_sendCommand(0x21); //Switch to extended mode
	glcd_sendCommand(0x84); //Set LCD Vop to 3.3V

	glcd_sendCommand(0x20); //Switch to basic instruction set
	glcd_sendCommand(0x0C); //Set normal display mode
	GLCD_drawImage(0, 0, (uint8_t *)welcome_screen, GLCD_WIDTH, GLCD_HEIGHT);
}
```

Antes de explicar las siguientes funciones, vamos a mostrar y explicar las declaraciones de macros, constantes y otras cosas que hay en el archivo de “glcd.c”

```
#include "glcd.h"

#define GLCD_clear_CS() {GPIOA->BRR = (1 << GLCD_CS);}
#define GLCD_set_CS() {GPIOA->BSRR = (1 << GLCD_CS);}
#define GLCD_clear_DC() {GPIOA->BRR = (1 << GLCD_DC);}
#define GLCD_set_DC() {GPIOA->BSRR = (1 << GLCD_DC);}
#define GLCD_clear_RST() {GPIOA->BRR = (1 << GLCD_RST);}
#define GLCD_set_RST() {GPIOA->BSRR = (1 << GLCD_RST);}

/*Private functions prototyps*/
void glcd_sendCommand(uint8_t);
void glcd_sendData(uint8_t);

/*Private variables*/
static uint8_t glcd_x = 0;
static uint8_t glcd_y = 0;
//this matrix is used to store the data of the screen
static uint8_t glcd_current_state[GLCD_HEIGHT][GLCD_WIDTH];
```

Como puede verse, y como es costumbre, tenemos macros para establecer los niveles lógicos de los pines del GLCD. Podemos controlar el pin de CS (Chip Select, SS en SPI), el de D/C que hemos mencionado, y el de /RST.
Además, tenemos las tres constantes que nos permiten mapear por software el estado del GLCD. Esto lo implementamos para tener dentro del microcontrolador información del estado actual del display gráfico, dónde está el cursor parado y qué imágen está pintada. *glcd_x* guarda la coordenada x del cursor, *glcd_y* guarda la coordenada y *glcd_current_state* es una matriz de 48x64 que tiene la información de lo que está pintado en el display.
A lo largo de todo el código, antes de llamar a un sendData, vamos a actualizar este *glcd_current_state* y antes de mover el cursor, vamos a actualizar los valores de *glcd_x* y *glcd_y*.

Por último, tenemos dos funciones privadas que nos permiten modularizar el proceso de enviar un comando o data al display. Veamoslas rápidamente

```
/*
Send command function
*/
void glcd_sendCommand(uint8_t command){
	GLCD_clear_DC();
	spi_tx(command);
}

//Sends data (writes 8 pixels)
void glcd_sendData(uint8_t data){
	GLCD_set_DC();
	spi_tx(data);
}
```

Sencillamente, en el caso de sendCommand se configura en bajo el pin D/C y luego se transmite por SPI el byte de configuración. Análogamente, para sendData, se configura en alto el pin de D/C y se hace el llamado a la misma función de SPI que transfiere la data.

2.6.2 GLCD_SendChar(char)

Esta es la función que nos permite enviar un caracter al display gráfico. Antes de explicarla vamos a mencionar un par de constantes declaradas en el archivo de cabecera:

- static const uint8_t ASCII[][5]: es una matriz, que para cada fila contiene un caracter ASCII. Cada elemento es un byte, y tiene 5 columnas. Por lo que el ancho de los caracteres ASCII en nuestro display va a ser de 5 píxeles y el alto de 8. Esta constante la sacamos de un repositorio, y le agregamos la letra ñ y el símbolo ° para nuestro proyecto. Ambos caracteres reemplazan al *" ' "* y al *" * "* respectivamente.
- static unsigned short welcome_screen_rows = 6;
  static unsigned short welcome_screen_cols = 84;
  static unsigned char welcome_screen[] = ….

  Estas tres constantes definen la matriz welcome_screen que armamos para el proyecto. Guardamos el tamaño de esta matriz en las primeras dos constantes. La matriz es de 84x6 Bytes y se usa para dibujar la pantalla que aparece cuando la alarma suena, y cuando se inicia el sistema. Puede verse en la figura 2.1.4

- static unsigned short display_rows = 6;
  static unsigned short display_cols = 84;
  static unsigned char display[] = …

Estas otras tres constantes definen otro dibujo, que es el que ponemos de fondo en todos los otros estados de la MEF que no son cuando suena la alarma. En la figura 2.1.1, 2.1.2 y 2.1.3 pueden verse.

Con esto explicado, volvemos a sendChar. Esta función va a primero verificar que el caracter entre en la fila en donde lo queremos dibujar (usando la variable de glcd_x) y luego establecer el pin de D/C en alto, preparandonos para la transmisión de 5 bytes (que componen el caracter) que se hace con el llamado a *spi_tx_msg()*. Antes de salir, actualizamos el estado del glcd en nuestras variables. El código es:

```
//Sends Character (writes 8*5 pixels)
void GLCD_sendChar(char character){
        uint8_t i;
        if(glcd_x>GLCD_WIDTH-5) //will overflow, don't write
                return;
        GLCD_set_DC();
        spi_tx_msg(ASCII[character-32],5);
        for(i=0;i<5;i++){ //update the current state
                glcd_current_state[glcd_y][glcd_x+i] = ASCII[character-32][i];
        }
        glcd_x += 5; //update the x position
}
```

2.6.3 GLCD_SendString(char)*

Esta función simplemente hace un llamado para cada uno de los caracteres del string a la función de GLCD_sendChar(). Su código sencillo puede verse a continuación:

```
//Prints string finished with '\0' on the screen
void GLCD_sendString(char* string){
	int i = 0;
	while(string[i] != '\0'){
		GLCD_sendChar(string[i]);
		i++;
	}
}
```

2.6.4 GLCD_SetXY(uint8_t x, uint8_t y)

Esta función nos permite configurar el cursor en donde escribimos los datos en el display. Recordemos de las tablas de registros, que el eje x toma valores entre 0 y la cantidad de píxeles de ancho del display (0-84) y que el eje y, toma valores entre 0 y la cantidad de píxeles de alto del display dividido 8 (0-6, ya que cada byte representa una linea vertical de 8 píxeles en el display). Por lo tanto, en caso de que los parámetros recibidos nos pidan configurar el XY en una posición inválida, no hacemos nada. Luego, si esta validación pasa, enviamos los comandos respectivos para configurar la dirección del eje Y y el eje X (a partir de la tabla 1 de set de instrucciones pueden verse). Finalmente, actualizamos nuestras variables internas para que reflejen al GLCD.

2.6.5 GLCD_Clean()

Simil al LCD, tenemos una función que nos permite limpiar el contenido del display. Para hacer esto, primero posicionamos al cursor en 0,0 y después, aprovechando el autoincremento de las coordenadas post escritura. Vamos a mandar el 0x00 (o sea, todos los píxeles en blanco) como data para limpiar toda la pantalla.
Antes de salir, reposicionamos el cursor en 0,0. El código que hace esto es:

```
//Cleans glcd then sets X and Y to 0
void GLCD_clean(void){
	int i,j;
	GLCD_setXY(0,0);
	for(i=0;i<GLCD_HEIGHT;i++){
		for(j=0;j<GLCD_WIDTH;j++){
			glcd_current_state[i][j] = 0;
			glcd_sendData(0x00);
			//delay_us(1);
		}
	}
	GLCD_setXY(0,0);
}
```

2.6.6 GLCD_drawImage(uint8_t x,uint8_t y,uint8_t* image,uint8_t width,uint8_t height)

Esta función como su nombre lo indica, nos permite dibujar una imágen. Ya la hemos visto en acción en el GLCD_Init() y lo cierto es que no tiene mucha ciencia. Recibe como parámetros x e y, el lugar desde donde se comienza a dibujar la imágen, un puntero a un arreglo de bytes que va a ser nuestra imágen a dibujar, y el ancho y largo de la misma.
Con toda esta información, estamos listos para hacer un doble bucle for, por cada fila y columna, y mandar los bytes de cada posición del arreglo de bytes.
El código entonces es:

```
/*
Draws an image at given coordinates
Image is an array of uint8_t of size width*height
You must specify the width and height of the image
*/
void GLCD_drawImage(uint8_t x, uint8_t y, uint8_t* image, uint8_t width, uint8_t height){
	int i,j;
	GLCD_setXY(x,y);
	for(i=0;i<height;i++){
		for(j=0;j<width;j++){
			glcd_current_state[y+i][x+j] = image[i*width+j];
			glcd_sendData(image[i*width+j]);
		}
	}
	glcd_x=x+width;
	glcd_y=y+height;
}
```

Por último, tenemos la función GLCD_drawImageNonDestructive() que nos permite hacer lo mismo que el drawImage, pero sin sobreescribir los píxeles que ya estaban pintados. Para esto, se hace uso de la variable *glcd_current_state* en la que guardabamos el estado del display, y en lugar de enviar el byte de datos, se le hace un "OR" con lo que ya estaba dibujado. Así, podemos "pintar encima" sin destruir lo que ya estaba. Su código es:

```
/*
Draws an image at given coordinates
Image is an array of uint8_t of size width*height
You must specify the width and height of the image
*/
void GLCD_drawImageNonDestructive(uint8_t x, uint8_t y, uint8_t* image, uint8_t width, uint8_t
height){
	int i,j;
	GLCD_setXY(x,y);
	for(i=0;i<height;i++){
		for(j=0;j<width;j++){
			glcd_current_state[y+i][x+j] |= image[i*width+j];
			glcd_sendData(image[i*width+j]|glcd_current_state[y+i][x+j]);
		}
	}
```

```
        glcd_x=x+width;
        glcd_y=y+height;
}
```

En un principio teníamos pensado usar esta función para algunos dibujos, pero luego optamos por resolver el problema de otra manera.

2.6.7 Script de python

No podemos terminar esta sección sin hacer mención honorífica a un script de python que utilizamos para poder armar los dibujos y las imágenes que pintamos en el display gráfico. Este script, toma una imágen con extensión ".bmp" y a partir de sus datos nos devuelve un ".txt" con las tres constantes que vimos arriba, en la sección 2.6.2, para poder definir una imágen que luego se le envía de parámetro al drawImage. Fue muy útil para hacer pruebas y pueden verse varias de ellas en la carpeta Auxiliares del repositorio de Github.

2.7 Periférico DHT22

Después de pasar un tiempo mirando sensores de Temperatura y Humedad, decidimos que lo mejor era usar un DHT11 o un DHT22. Si bien en un principio se iba a utilizar el DHT11, luego de jugar con los 2 sensores (y básicamente implementar los 2) decidimos utilizar el DHT22 para el proyecto porque tiene mejores características que el DHT11.

Las características del DHT11 son:
±2°C de precisión para medir temperatura y ±5% de precisión RH para medir humedad
=0.5°C de resolución para medir temperatura y =0.1 % RH para medir humedad

Las características del DHT22, en cambio, son:
±0.5°C de precisión para medir temperatura y ±2% de precisión RH para medir humedad
=0.1°C de resolución para medir temperatura y =0.1 % RH para medir humedad

Si bien el DHT22 es más caro, en este momento estamos utilizando Proteus para la simulación, por lo cual el precio no supone un problema.

Para iniciar la comunicación el MCU debe enviar una señal de 'Start' al DHT22. En este caso estaremos utilizando el PIN15 del Puerto A en modo open drain (al igual que el I2C) con una resistencia de Pull-Up externa. Cuando el MCU envía el 'start' cambia de HIGH a LOW durante al menos 1 ms para asegurar que el sensor detectó la señal. Luego, el MCU debe liberar el pin (ponerlo en alto) para esperar entre 20-40us la respuesta del DHT22.
Cuando el DHT22 detecta la señal, setea el pin en LOW por 80us, y luego lo libera dejandolo en HIGH por 80us como preparación para enviar DATA. Todo este proceso puede verse en la figura 2.7.1

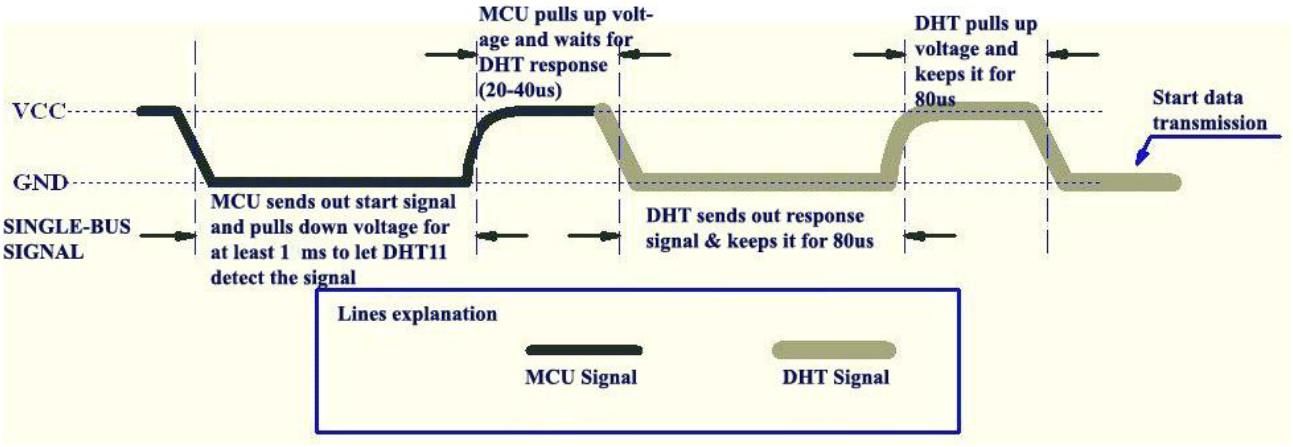


Figura 2.7.1 - Inicio de comunicación con DHT

El DHT envía cada bit de transmisión poniendo primero el pin en LOW por 50us. Luego, codificará el valor del bit con la longitud del tiempo en el que mantiene liberado el pin en HIGH. Si lo deja por entre 26 y 28 us en HIGH, se considera que es un 0. Si lo deja en high por 70us se considera que es un 1. Note que si el DHT envía constantemente HIGH es porque no está funcionando correctamente. En la figura 2.7.2 pueden verse primero cómo se codifica el 1 y luego como se codifica el 0.

La cadena de bits que nos envía el DHT22 luego del inicio de la comunicación, es de 40 bits, o 5 bytes. Los primeros 2 bytes van a ser la parte entera y decimal de la humedad relativa sensada, los segundos 2 bytes serán la parte entera y decimal de la temperatura sensada y el último byte enviado es un checksum, que contiene los 8 LSB de la suma de estos 4 primeros bytes.

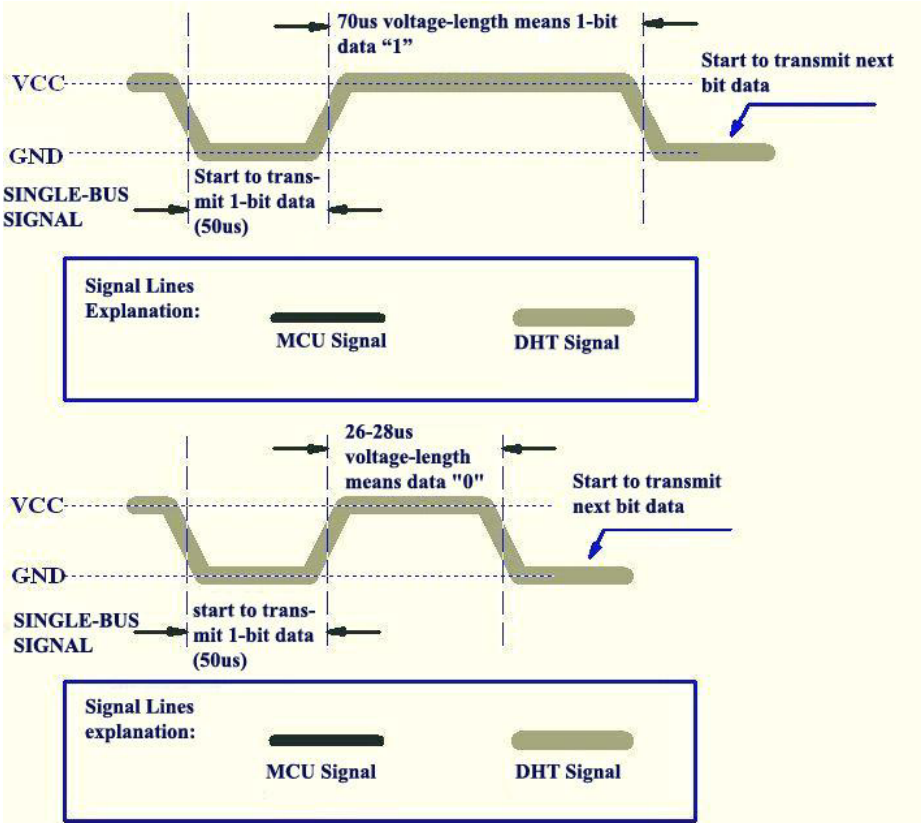


Figura 2.7.2 - Codificación 1 y 0 en DHT22

Yendo al código, en la cabecera podemos ver el siguiente código:
Que define las macros de tolerancia de respuesta (por si no nos responde el DHT22, para que no nos quedemos bloqueados), el pin de DATA, el PUERTO y los registros de datos de entrada y salida.
Luego podemos ver dos funciones: DHT22_Init() para inicializar la librería y DHT22_GetTemp_Humidity() que recibe dos flotantes (temperatura y humedad respectivamente) y devuelve 1 en caso de haber obtenido correctamente los valores y 0 en caso contrario.
En la implementación de la librería (DHT22.c) podemos ver tres macros para establecer el pin de DATA en HIGH en LOW y leerlo. Similar a lo que teníamos en el I2C para los pines SDA y SCL.

```
#ifndef DHT22_H
#define DHT22_H

#include <stm32f103x6.h>
#include "utils.h"
#ifndef DHT22_RESPONSE_TOLERANCE
#define DHT22_RESPONSE_TOLERANCE 1000000UL //at 72MHZ clock this is about 2ms
#endif

#define DHT22_DATA_PIN 15
#define DHT22_DATA_PORT GPIOA
#define DHT22_DR DHT22_DATA_PORT->ODR
#define DHT22_DR_IN DHT22_DATA_PORT->IDR
//*** Functions prototypes ***//
//OneWire Initialise
void DHT22_Init(void);

//Get Temperature and Humidity data
uint8_t DHT22_GetTemp_Humidity(float *, float *);

#endif
```

Yendo al código del DHT22_Init podemos ver que se inicializa el Pin 15 del puerto A como GPO open drain.

```
void DHT22_Init(void){
   //We use PA15 as the DHT22 data pin, output open drain
   DHT22_DATA_PORT->CRH |= 0x70000000;
   //PA15 is GPO open drain -> setting it means pin gets pulled up
   DHT22_data_set(); //PA15 set to high
   delay_ms(1000);
}
```

A partir de ese momento, se puede llamar a la función DHT22_GetTemp_Humidity(&temp, &humidity) para obtener los valores de temperatura y humedad.

Esta función llamará a la función privada DHT22_SendStart(), la cual llevará a cabo la comunicación entre el MCU y DHT22 explicada anteriormente. Esta comunicación se manejó con

los delays de la librería utils, y con un factor de tolerancia de aproximadamente 2 ms para evitar dejar el código en un bucle infinito al utilizar polling.

```
//This function will send start signal to DHT22
//Return 1 if success, 0 if fail
//Returns when DHT22 is about to start data transmission
uint8_t DHT22_SendStart(){
    aux=1;
    //Set data pin to low
    DHT22_data_clear(); //PA15 set to low
    //Wait for 1ms
    delay_ms(18);
    //Set data pin to high and wait DHT22 response
    DHT22_data_set(); //PA15 set -> high impedance
    VAR  = DHT22_data_get();
    //Wait for DHT22 response
    while(DHT22_data_get()!=0 && ++aux!=DHT22_RESPONSE_TOLERANCE); //Wait sensor
response
    VAR = DHT22_data_get();
    if(aux==DHT22_RESPONSE_TOLERANCE)
        return 0;
    //Wait for DHT22 to release data pin
    aux=1;
    VAR = DHT22_data_get();
    delay_us(40);
    VAR = DHT22_data_get()==0;
    while(DHT22_data_get()==0 && ++aux!=DHT22_RESPONSE_TOLERANCE); //Wait sensor to set
voltage in high (~80us)
    VAR = DHT22_data_get()==0;
    delay_us(40);
    VAR = DHT22_data_get();
    while(DHT22_data_get()!=0 && ++aux!=DHT22_RESPONSE_TOLERANCE); //Wait sensor to set
voltage in low (~80us)
    VAR = DHT22_data_get();
    //DHT is ready to send data
    return aux!=DHT22_RESPONSE_TOLERANCE;
}

//This function is supposed to be called AFTER a data start transmission has been sent
//It will read one byte from DHT22 assuming single-bus signal is being sent
//Return the byte read
uint8_t ReadByte(){
    uint8_t i=0;
    datar=0;
    while(i<8){
        //Wait for DHT22 response
        while(DHT22_data_get()==0); //Wait until PA15 gets high (~50us)
        //PA15 is high, if it stills high after more than 28us it means DHT22 is sending a 1
```

```
        //We wait for 29us just in case
        delay_us(29);
        if(DHT22_data_get() !=0)
            datar |= (1<<(7-i)); //We get PA15 state and set data bit accordingly
        //Wait for DHT22 to set pin to low before reading next bit
        while(DHT22_data_get()!=0); //PA15 is low
        i++;
    }
    return datar;
}
```

La función nos devolverá 1 si la comunicación fue exitosa y 0 en caso contrario. Si nos devolvió 1, entonces entramos en la lógica de leer los 5 bytes del DHT22. Para esto, se llama 5 veces a la función ReadByte que nos devuelve el byte leído. El código de esta función que lee bytes es:

```
//This function is supposed to be called AFTER a data start transmission has been sent
//It will read one byte from DHT22 assuming single-bus signal is being sent. //Return the byte
read
uint8_t ReadByte(){
    uint8_t i=0;
    datar=0;
    while(i<8){
        //Wait for DHT22 response
        while(DHT22_data_get()==0); //Wait until PA15 gets high (~50us)
        //PA15 is high, if it stills high after more than 28us
            //it means DHT22 is sending a 1

        delay_us(29);
        if(DHT22_data_get() !=0)
            datar |= (1<<(7-i));
        //We get PA15 state and set data bit accordingly

        //Wait for DHT22 to set pin to low before reading next bit
        while(DHT22_data_get()!=0); //PA15 is low
        i++;
    }
    return datar;
}
```

Y el código hasta ahora de nuestro DHT22_GetTemp_Humidity nos viene quedando:

```
uint8_t DHT22_GetTemp_Humidity(float *temp, float *humidity){
   uint8_t i;
   if(temp==0 || humidity==0)
      return 0;
   if(DHT22_SendStart()){
      for(i=0; i<5; i++){
         data[i] = ReadByte();
      }
```

Luego de la lectura, de los 5 bytes, se verificará con el checksum la integridad de datos obtenidos. Si esta validación pasa exitosamente, se calculan los valores de temperatura y humedad con los bytes leídos, se castean a flotantes y se asignan a las variables &temp, &humidity recibidas en la llamada de la función.

```
   //Checksum check
      if(data[4] == ((data[0] + data[1] + data[2] + data[3]) & 0xFF)){
         *temp = (float)((data[2] & 0x7F) << 8 | data[3]) / 10; //TODO check this
         *humidity = (float)(data[0] << 8 | data[1]) / 10; //TODO check this
         return 1;
      }
   }
   return 0;
}
```

Una vez obtenidos los valores, podrán utilizarse en el programa principal de la manera que se quiera. Observar que retornaremos 0 si hubo algún error en el transcurso de la obtención de temperatura y humedad.

2.8 Librería GPIO

Esta librería se utiliza para inicializar los pines de los botones y el buzzer. Y encapsular el comportamiento relacionado a eso.
En el archivo de cabecera tenemos definidos lo siguiente:

```
#ifndef GPIO_H
#define GPIO_H

#include <stm32f103x6.h>



//This on port A
#define GPIOA_BUTTON_OK_PIN 8
#define GPIOA_BUTTON_CANCEL_PIN 9

//This on port B
```

```
#define GPIOB_BUTTON_BUZZER_PIN 9
#define GPIOB_BUTTON_UP_PIN 12
#define GPIOB_BUTTON_DOWN_PIN 13
#define GPIOB_BUTTON_LEFT_PIN 14
#define GPIOB_BUTTON_RIGHT_PIN 15


#define getOk() (GPIOA->IDR & (1 << GPIOA_BUTTON_OK_PIN))
#define getCancel() (GPIOA->IDR & (1 << GPIOA_BUTTON_CANCEL_PIN))
#define getUp() (GPIOB->IDR & (1 << GPIOB_BUTTON_UP_PIN))
#define getDown() (GPIOB->IDR & (1 << GPIOB_BUTTON_DOWN_PIN))
#define getLeft() (GPIOB->IDR & (1 << GPIOB_BUTTON_LEFT_PIN))
#define getRight() (GPIOB->IDR & (1 << GPIOB_BUTTON_RIGHT_PIN))

#define setBuzzer() {GPIOB->ODR |= 1 << GPIOB_BUTTON_BUZZER_PIN;}
#define resetBuzzer() {GPIOB->ODR &= ~(1 << GPIOB_BUTTON_BUZZER_PIN);}


#define BUTTON_NONE 0
#define BUTTON_OK 1
#define BUTTON_CANCEL 2
#define BUTTON_UP 3
#define BUTTON_DOWN 4
#define BUTTON_LEFT 5
#define BUTTON_RIGHT 6
```

Como pueden verse hay definidas varias macros que hacen referencia a los pines de cada botón y del buzzer. Además, varias funciones macros, que nos permiten conocer el valor lógico de los pines asociados a los botones (y saber si están siendo presionados) y establecer y limpiar el pin del buzzer (prenderlo y apagarlo).
Los botones y el buzzer están mapeados en el proyecto de la siguiente manera

- Botón OK -> PA8
- Botón CANCELAR -> PA9
- Botón ARRIBA -> PB12
- Botón ABAJO -> PB13
- Botón IZQUIERDA -> PB14
- Botón DERECHA -> PB15
- Buzzer -> PB9

Además hay 7 constantes definidas para los 6 botones más una extra que indica que no se está presionando ningún botón. Esto se usa en la función de GPIOS_getButtonPressed, para setear el botón presionado.

Las dos funciones que nos da la librería son, GPIO_Init y GPIOS_getButtonPressed(). Esta última recibe un puntero a un entero de 8 bits sin signo.
La implementación del Init es sencilla

```
void GPIO_Init(){
   //---
   GPIOA->CRH = 0x44444488; //PA8 and PA9 as input with pull-up
   GPIOB->CRH = 0x88884434; //PB12, 13,14,15 as input with pull-up
   //---                    PB9 as output with push-pull (0011)
}
```

Como puede verse simplemente establecemos los pines asociados a los botones como entradas con pull-up y el pin del buzzer como output push-pull. Esto está bastante hardcodeado y debería cambiarse si se desean cambiar los pines.
En lo que respecta a la función getButtonPressed() tenemos el siguiente sencillo código:

```
//RETURNS BUTTON_MENU or BUTTON_SELECT
void GPIOS_getButtonPressed(uint8_t* buttonPressed)
{
   //TODO: ver de mover esto a una librerÃa
   //Also, order of this ifs matters. CANCEL has priority
   if (getCancel() != 0)
   {
      *buttonPressed = BUTTON_CANCEL;
      return;
      //return BUTTON_CANCEL;
   }
   if (getOk() != 0)
   {
      *buttonPressed = BUTTON_OK;
      return;
      //return BUTTON_OK;
   }
   if (getUp() != 0)
   {
      *buttonPressed = BUTTON_UP;
      return;
      //return BUTTON_UP;
   }
   if (getDown() != 0)
   {
      *buttonPressed = BUTTON_DOWN;
      return;
      //return BUTTON_DOWN;
   }
   if (getLeft() != 0)
   {
      *buttonPressed = BUTTON_LEFT;
      return;
      //return BUTTON_LEFT;
   }
```

```
    if (getRight() != 0)
    {
        *buttonPressed = BUTTON_RIGHT;
        return;
        //return BUTTON_RIGHT;
    }
    *buttonPressed = BUTTON_NONE;
    return;
    //return BUTTON_NONE;
}
```

Como puede verse son varios if que revisan si el botón presionado es cancel, ok, arriba, abajo, izquierda o derecha, en ese orden de prioridad. Esto significa, que si tenemos más de un botón presionado, se tomará el primero por el que se entre. Acá, decidimos darle prioridad al CANCEL por sobre todos los demás.
Cabe destacar que en caso de no detectarse ningún botón presionado, la función va a devolver el buttonPressed seteado a BUTTON_NONE, el septimo botón que hemos mencionado que simboliza "no se está presionando ningún botón".

2.9 Mef

Como mencionamos en los incisos anteriores, en este proyecto utilizaremos una máquina de estados, y en esta sección pasaremos a explicar el desarrollo lógico de la misma.
Comenzaremos en primer lugar por la cabecera, como se puede ver contiene el prototipo de las 2 funciones públicas que implementa nuestra MEF, MEF_Init(), y MEF_Update, sumado a varias macros utilizadas dentro el archivo .c y un tipo enumerador que tiene 4 valores correspondientes a los estados.

```
#ifndef MEF_H
#define MEF_H

#include <stm32f103x6.h>
#include <stdio.h>
#include <stdlib.h>
#include "gpio.h"
#include "dht22.h"
#include "ds1307.h"
#include "glcd.h"

#define SELECTION_NONE 0
#define SELECTION_DAY 1
#define SELECTION_DATE 2
#define SELECTION_MONTH 3
#define SELECTION_YEAR 4
#define SELECTION_HOUR 5
#define SELECTION_MINUTE 6
#define SELECTION_SECOND 7
#define SELECTION_START 1
#define SELECTION_END 7
```

```
#define SELECTION_START_ALARM 5
#define SELECTION_END_ALARM 7

typedef enum {IDLE,IDLE_ACTIVE, SETTING_DATE,SETTING_ALARM} MEF_state;


void MEF_Init(void);
void MEF_Update(void);


#endif
```

Con la cabecera ya definida, "metámonos" en el archivo .c.
Nuestro subsistema a su vez, cuenta con varias funciones privadas y el prototipo de las mismas se encuentran dentro del MEF.c, ya que como su nombre lo indica solo pueden ser usadas dentro de este archivo. A la hora de definir las funciones privadas, nos encontramos con un problema. Por definición, dichas funciones deben ser declaradas como static, pero Proteus nos dió varios problemas que se solucionaron al remover esta palabra clave. Suponemos que está relacionado con cómo se compila el proyecto.
A su vez, contamos con variables privadas, que son conocidas por todos los métodos dentro del archivo por ejemplo: el estado actual, la hora actual, a la hora que debe sonar la alarma, la hora renderizada por última vez (time_old), la fecha actual y renderizada por última vez, la temperatura, la humedad, un string de 100 caracteres que usaremos en las funciones para escribir en el display gráfico, y la variable current_selection, que se usa en los estados de modificar fecha y configurar alarma.
La definición de los prototipos de función y variables privadas son los siguientes

```
static uint8_t buttonPressed = 0;
static uint8_t firstTime = 1; //first time bit to avoid rendering stuff more than once
static MEF_state system_state;
static RTC_TIME_t time, time_old,time_alarm,time_alarm_old;
static RTC_DATE_t date, date_old;
static float temperature, humidity;
static char str[100];
static uint8_t current_selection;

//private functions (not static because proteus breaks)
void switchToSetAlarm(void);
void switchToSetDate(void);
void switchToIDLE(void);
void switchToIDLEActive(void);

void updateDate(void);
void updateTime(void);
void renderIDLE(void);
void renderIDLEActive(void);
void renderSetDate(void);
void renderSetAlarm(void);
```

```
uint8_t checkAlarm(void);
void disableBuzzer(void);
void enableBuzzer(void);

void renderDay(uint8_t day);
void renderDate(uint8_t);
void renderTime(uint8_t);

void updateTemperatureAndHumidity(void);
void renderTemperatureAndHumidity(uint8_t);

void animateCurrentSelection(void);
void animateCurrentSelection_alarm(void);
//Functions for setDate and setAlarm states
void writeTime(RTC_TIME_t);
void writeDate(RTC_DATE_t);
void modify_date(uint8_t selection, int8_t increment);
void modify_alarm(uint8_t selection, int8_t increment);
void renderTime_alarm(uint8_t force);
```

Antes de utilizar nuestra MEF, debemos inicializarla, justamente esto es lo que se hace en uno de las 2 funciones públicas del mismo, MEF_Init().
La primera acción que realiza esta función, es llamar a la función GPIO_Init() para inicializar los pines del MCU utilizados en los botones y el buzzer. Luego se define el estado inicial en el que comenzará a operar la MEF (IDLE), y por último se setean en 0 las variables referidas a la hora en la que debe sonar la alarma (se establece la alarma a las 00:00:00).
Luego de lo descrito el código del MEF_Init() es el que se puede observar a continuación

```
void MEF_Init(void)
{
   GPIO_Init();
   system_state = IDLE;
   time_alarm.hours = 0;
   time_alarm.minutes = 0;
   time_alarm.seconds = 0;
   time_alarm_old=time_alarm;
}
```

La otra función pública que posee nuestro sistema, es el MEF_Update(). Como siempre que hemos trabajado con máquinas de estado, esta función es un switch-case que va tomando ciertas acciones en base al estado actual del en el que el sistema se encuentra y el botón que haya presionado el usuario.
Un pseudocódigo general del MEF_Update, sería el que se observa en la figura 2.9.1

```
void MEF_Update ()
begin
        GPIOS_getButtonPressed(&buttonPressed);
        switch (estadoAct)
        begin
                case (estado1):
                                acciones del estado 1
                break;
                case (estado2):
                                acciones del estado 2
                break;
        …
                case (estadoN):
                                acciones del estado N
                break;
        end
end
```

Figura 2.9.1-Pseudocódigo MEF_Update

Vamos a analizar cada segmento de acciones para cada estado por separado. Recordemos que a este switch case, le antecede el llamado a la función de la librería de GPIO para obtener el botón presionado.

2.9.1 Estado IDLE

En este estado, vamos a primero que nada, verificar que la hora de la alarma no sea igual a la hora actual. En ese caso, debemos cambiar al estado de IDLE_ACTIVE (en el que suena la alarma). A este checkeo, le agregamos una vuelta de tuerca ya que verifica que no sea la primera vez entrando al estado IDLE. Esto es, para que cuando volvamos de IDLE_ACTIVE al IDLE, se ejecute al menos una vez el IDLE, actualizandose la hora y evitandose que se redispare la alarma.
Luego de este if, vienen los llamados a las funciones privadas de updateTime, updateDate y updateTemperatureAndHumidity.
Las tres funciones, se encargan de establecer en las variables time, date, temperature y humidity los valores leidos del RTC y los sensores, haciendo llamados a las funciones de las librerías:

```
void updateTime()
{
  DS1307_get_time(&time);
}

void updateDate()
{
  DS1307_get_date(&date);
}

void updateTemperatureAndHumidity(void)
{
  DHT22_GetTemp_Humidity(&temperature, &humidity);
}
```

Una vez tengamos los valores de temperatura humedad fecha y hora actualizados, se llama a la función renderIDLE() que se encarga de renderizar en el display gráfico estos valores.
Adentro de esta función privada, preguntamos si es la primera vez que se ejecuta el renderIDLE() desde que se cambió al estado (para eso sirve el firstTime) y de ser así, renderizamos la imágen de fondo (y actualizamos a 0 el valor de firstTime).
Luego, vamos a renderizar la fecha, la hora, la temperatura y la humedad, para lo que tenemos funciones privadas definidas que reciben de parámetro el valor de firstTime. Dentro de estas funciones, hay una lógica implementada para no repintar el valor correspondiente, en caso de que sea el mismo que ya estaba pintado.
En el caso de renderDate entonces, tenemos que si la fecha actual es igual a la fecha vieja (acá usamos la función utilitaria de la librería del RTC para comparar fechas) y no estamos forzados a renderizar (el force recibido de parámetro), retornamos sin pintar nada. En caso contrario, hay que actualizar el valor de la fecha vieja y pintar la fecha actual. Para esto, primero se hace un llamado a la función privada renderDay() que se encarga de escribir el día de la semana, luego con la función sprintf se formatea el string con la fecha y se usan las funciones GLCD_setXY y GLCD_sendString para dibujarla en donde corresponde.
El código que hace todo esto es:

```
void renderDate(uint8_t force)
{
   if (DS1307_dateEquals(&date, &date_old) && !force)
   {
      return; //no need to render
   }
   date_old = date;
   renderDay(date.day);
   sprintf(str, "%02d/%02d/%02d", date.date, date.month, date.year);
   GLCD_setXY(23, 3);
   GLCD_sendString(str);
}

void renderDay(uint8_t day)
{
   GLCD_setXY(20, 2);
   switch (day)
   {
   case MONDAY:
      GLCD_sendString("  LUNES  ");
      break;
   case TUESDAY:
      GLCD_sendString(" MARTES  ");
      break;
   case WEDNESDAY:
      GLCD_sendString("MIERCOLES");
      break;
   case THURSDAY:
      GLCD_sendString(" JUEVES  ");
      break;
   case FRIDAY:
```

```
      GLCD_sendString(" VIERNES ");
      break;
    case SATURDAY:
      GLCD_sendString(" SABADO  ");
      break;
    case SUNDAY:
      GLCD_sendString(" DOMINGO ");
      break;
    }
}
```

En lo que respecta al renderTime la lógica es la misma, pero en lugar de pintar la fecha, pintamos la hora. El código es:

```
void renderTime(uint8_t force)
{
    if (DS1307_timeEquals(&time, &time_old) && !force)
    {
        return; //no need to render
    }
    time_old = time;
    sprintf(str, "%02d:%02d:%02d", time.hours, time.minutes, time.seconds);
    GLCD_setXY(23, 4);
    GLCD_sendString(str);
}
```

Para renderizar la temperatura y humedad, no hacemos validaciones de sí cambiaron por lo que van a pintarse en todas las iteraciones. Simplemente nos paramos en Y=1 (la segunda fila) y pintamos la temperatura a la izquierda y la humedad a la derecha ayudándonos del sprintf. El código de esta función es:

```
void renderTemperatureAndHumidity(uint8_t force)
{
    sprintf(str, "%.2f*C", temperature);
    GLCD_setXY(4, 1);
    GLCD_sendString(str);
    sprintf(str, "%.1f%%H", humidity);
    GLCD_setXY(48, 1);
    GLCD_sendString(str);
}
```

Volviendo al estado IDLE, luego de este llamado al renderIDLE(), tenemos la lógica que verifica cuál es el botón presionado. En caso de no ser ninguno, retornamos. Si presionamos derecha, vamos a cambiar al estado SETTING_DATE (el de cambiar fecha y hora) con el llamado a la función privada switchToSetDate(). Si presionamos izquierda, en cambio, vamos a hacer el llamado a switchToSetAlarm(), que nos cambia al estado SETTING_ALARM (para cambiar la hora de la alarma). Con todo esto dicho, el código del IDLE es:

```
case IDLE:
      //check if alarm must be activated AFTER first time render
      if(checkAlarm() && !firstTime){
         switchToIDLEActive();
         return;
      }
      //get data from sensors and RTC
      updateTime();
      updateDate();
      updateTemperatureAndHumidity();
      //render it
      renderIDLE();
      if (buttonPressed == BUTTON_NONE)
         return;
      if (buttonPressed == BUTTON_LEFT)
      {
         switchToSetAlarm();
         return;
      }
      if (buttonPressed == BUTTON_RIGHT)
      {
         switchToSetDate();
         return;
      }
      break;
```

Las funciones privadas switchToSetAlarm(), switchToSetDate(), switchToIDLE() y switchToIdleActive() todas cambian la variable system_state al estado correspondiente y setean la variable firstTime a 1, indicando que en el próximo MEF update es la primera vez que se ejecutan. Particularmente, las funciones switchToSetAlarm() y switchToSetDate() modifican otra variable: current_selection, que se usa para indicar la selección actual del usuario en los estados de modificación de fecha y alarma. De este modo, arrancan seleccionando “ninguna opción”. Explicaremos esto más adelante.

```
//This function switchs to IDLEActive state, which is a state where the buzzer is sounding
void switchToIDLEActive(void)
{
   system_state = IDLE_ACTIVE;
   firstTime = 1;
   return;
}
//This function switchs state from IDLE to SET_ALARM
void switchToSetAlarm(void)
{
   system_state = SETTING_ALARM;
   current_selection=SELECTION_NONE;
   firstTime = 1;
   return;
```

```
}
//This function switchs state from IDLE to SET_DATE
void switchToSetDate(void)
{
   system_state = SETTING_DATE;
   current_selection = SELECTION_NONE;
   firstTime = 1;
   return;
}

//This function returns to IDLE state
void switchToIDLE(void)
{
   system_state = IDLE;
   firstTime = 1;
   return;
}
```

2.9.2 Estado IDLE_ACTIVE

En este estado, lo único que se hace es, primero el llamado a la función privada renderIDLEActive() que en caso de que sea la primera vez, renderiza el dibujo de welcome_screen y activa el buzzer (llamando a la función enableBuzzer).
Luego, se pregunta si la tecla presionada es “BUTTON_NONE” o sea, pregunta si no hay ninguna tecla presionada. En ese caso, retorna de la función.
En caso contrario, va a desactivar el buzzer (llamando a la función disableBuzzer) y llamara a la función switchToIDLE() que hará la transición al estado IDLE.
El código de este segmento del switch-case y de las funciones privadas que se llaman es:

```
case IDLE_ACTIVE:
      renderIDLEActive();
      if (buttonPressed == BUTTON_NONE)
         return;

      disableBuzzer(); //first disable buzzer
      switchToIDLE();  //then switch to idle
      break;
   case SETTING_DATE:
      renderSetDate();
      if (buttonPressed == BUTTON_NONE)
         return;
      //CANCEL -> IDLE, don't save
      if (buttonPressed == BUTTON_CANCEL)
      {
         switchToIDLE();
         return;
      }
      //OK -> IDLE, save
```

```
        if (buttonPressed == BUTTON_OK)
        {
            writeDate(date);
            writeTime(time);
            switchToIDLE();
            return;
        }
        //UP -> increment
        if (buttonPressed == BUTTON_UP)
        {
            modify_date(current_selection, 1);
        }
        //DOWN -> decrement
        if (buttonPressed == BUTTON_DOWN)
        {
            modify_date(current_selection, -1);
        }
        //LEFT -> select next value
        if (buttonPressed == BUTTON_RIGHT)
        {
            current_selection++;
            if (current_selection > SELECTION_END)
                current_selection = SELECTION_START;
        }
        //RIGHT -> select previous value
        if (buttonPressed == BUTTON_LEFT)
        {
            current_selection--;
            if (current_selection < SELECTION_START)
                current_selection = SELECTION_END;
        }
        animateCurrentSelection();
        break;

void renderIDLEActive(void)
{
    if (firstTime == 1)
    {
        GLCD_drawImage(0, 0, (uint8_t *)welcome_screen, GLCD_WIDTH, GLCD_HEIGHT);
        enableBuzzer();
        firstTime = 0;
    }
}

//This function disables the buzzer
void disableBuzzer(void)
{
```

```
    resetBuzzer();
}

//This function enables the buzzer
void enableBuzzer(void)
{
    setBuzzer();
}
```

2.9.3 Estado SETTING_DATE

Este es el estado que en la MEF de la figura 2.1.5 llamamos SET_FECHA_Y_HORA. En este, el usuario podrá configurar la fecha y la hora del sistema.
Como todos los estados, comienza con un render. En este caso, renderSetDate() es la función privada que va a, en caso de ser la primera vez, renderizar la imágen del display en la pantalla y luego, el campo seleccionado para modificar la primera vez (que será SEL: NONE ya que aún no ha elegido ningún campo para modificar).
Posterior a esto, se renderiza la fecha y hora, usando la misma lógica que en el estado IDLE.
Este es el código asociado a esa función:

```
//This function renders the SET_DATE
void renderSetDate(void)
{
    uint8_t force = firstTime;
    if (firstTime)
    {
        GLCD_drawImage(0, 0, (uint8_t *)display, GLCD_WIDTH, GLCD_HEIGHT);
        GLCD_setXY(13,1);
        sprintf(str," SEL: NONE  ");
        GLCD_sendString(str);
        firstTime = 0;
    }
    renderDate(force);
    renderTime(force);
}
```

Como se puede observar, en este estado, usamos las variables date y time que antes tenían el valor de la fecha y hora actualizada con el RTC. Como no hacemos ningun llamado a updateTime o updateDate en este estado, van a quedar 100% bajo el control del usuario.

Después de la llamada a renderSetDate, viene toda la lógica relacionada al botón que hayamos presionado.

- Si no se presionó nada, retornamos de la función
- Si se presionó CANCEL, llamamos al switchToIDLE() y no guardamos nada
- Si se presionó OK, llamamos a las funciones writeDate() y writeTime() enviando el date y el time que hemos configurado y luego, llamamos al switchToIDLE() que nos cambia al estado IDLE.

- Los botones de izquierda y derecha, nos permitiran ir cambiando la selección actual. La selección actual, se guarda en current_selection y recordemos nos referencia el campo (hora, minuto, segundo, día, mes, etc) que estamos modificando actualmente.
- Los botones de arriba y abajo, nos permiten incrementar y decrementar el valor numérico de lo que estamos selecionando.

Por último, se hace un update del texto “SEL: xxx” con el llamado a animateCurrentSelection() que nos permite mostrar al usuario cuál es el campo que está modificando.

```
void animateCurrentSelection(void){
   GLCD_setXY(13,1);
   switch(current_selection){
      case SELECTION_YEAR:
         sprintf(str,"SEL A'O: %2d",date.year);
      break;
      case SELECTION_MONTH:
         sprintf(str,"SEL MES: %2d",date.month);
         //TODO: animate month
      break;
      case SELECTION_DATE:
         sprintf(str,"SEL DIA: %2d",date.date);
         //TODO: animate date
      break;
      case SELECTION_DAY:
         sprintf(str,"SEL: DIASEM");
         //TODO: animate day
      break;
      case SELECTION_HOUR:
         sprintf(str,"SEL HS: %2d ",time.hours);
         //TODO: animate hour
      break;
      case SELECTION_MINUTE:
         sprintf(str,"SEL MIN: %2d",time.minutes);
         //TODO: animate minute
      break;
      case SELECTION_SECOND:
         sprintf(str,"SEL SEG: %2d",time.seconds);
         //TODO: animate second
      break;
      case SELECTION_NONE:
         sprintf(str," SEL: NONE  ");
      default:
         sprintf(str," SEL: NONE  ");
      break;
   }
   GLCD_sendString(str);
}
```

El código de esto entonces es:

```
case SETTING_DATE:
        renderSetDate();
        if (buttonPressed == BUTTON_NONE)
            return;
        //CANCEL -> IDLE, don't save
        if (buttonPressed == BUTTON_CANCEL)
        {
            switchToIDLE();
            return;
        }
        //OK -> IDLE, save
        if (buttonPressed == BUTTON_OK)
        {
            writeDate(date);
            writeTime(time);
            switchToIDLE();
            return;
        }
        //UP -> increment
        if (buttonPressed == BUTTON_UP)
        {
            modify_date(current_selection, 1);
        }
        //DOWN -> decrement
        if (buttonPressed == BUTTON_DOWN)
        {
            modify_date(current_selection, -1);
        }
        //LEFT -> select next value
        if (buttonPressed == BUTTON_RIGHT)
        {
            current_selection++;
            if (current_selection > SELECTION_END)
                current_selection = SELECTION_START;
        }
        //RIGHT -> select previous value
        if (buttonPressed == BUTTON_LEFT)
        {
            current_selection--;
            if (current_selection < SELECTION_START)
                current_selection = SELECTION_END;
        }
        animateCurrentSelection();
        break;
```

Puede verse que se aprovecha que current_selection está modelado con un número para incrementarlo o decrementarlo. Y que se mantiene siempre dentro del rango de las posibles 7 cosas que podemos seleccionar

Las funciones writeDate y writeTime son sencillas, llaman a las funciones de la librería del RTC para establecer la fecha y hora:

```
//Functions for setting date and alarm

void writeDate(RTC_DATE_t date){
   DS1307_set_date(&date);
}


void writeTime(RTC_TIME_t timeR){
   DS1307_set_time(&timeR);
}
```

La función privada modify_date() recibe como parámetro primero el campo a modificar (el seleccionado por el usuario) y segundo el incremento (que será +1 o -1 en caso de querer decrementar el valor)
Adentro, tiene lógica para que en caso de pasarnos de los rangos para cada campo, “pegue la vuelta”. Imitando el comportamiento de los relojes despertadores.

```
void modify_date(uint8_t selection, int8_t increment){
   switch (selection)
   {
   case SELECTION_YEAR:
      date.year += increment;
      if (date.year > 99)
         date.year = 0;
      break;
   case SELECTION_MONTH:
      date.month += increment;
      if (date.month > 12)
         date.month = 1;
      if (date.month < 1)
         date.month = 12;
      break;
   case SELECTION_DATE:
      date.date += increment;
      if (date.date > 31)
         date.year = 1;
      if (date.date < 1)
         date.date = 31;
      break;
   case SELECTION_DAY:
      date.day += increment;
      if (date.day > SUNDAY)
         date.day = MONDAY;
      if(date.day<MONDAY)
         date.day = SUNDAY;
```

```
        break;
    case SELECTION_HOUR:
        time.hours += increment;
        if (time.hours > 24)
            time.hours = 0;
        break;
    case SELECTION_MINUTE:
        time.minutes += increment;
        if (time.minutes > 60)
            time.minutes = 0;
        break;
    case SELECTION_SECOND:
        time.seconds += increment;
        if (time.seconds > 60)
            time.seconds = 0;
        break;
    default: break;
    }
    return;
}
```

Por últmo, la función animateCurrentSelection(), toma la variable current_selection e imprime en el GLCD el texto en la fila Y=1 indicando al usuario el valor que está modificando actualmente.

2.9.4 Estado SETTING_ALARM

En este cuarto estado que es el SET_ALARM de la MEF de la figura 2.1.5, tenemos una implementación muy análoga a la vista en el estado de modificar fecha.
Primero se hace un llamado al render. En este caso, renderSetAlarm(). Que hace lo mismo que el renderSetDate() pero no renderiza la fecha ni el día de semana. Además, el valor del *time*, lo toma de la variable *time_alarm* que es la que se usará en todo este estado para la hora de la alarma.

```
//This function renders the SET_ALARM
void renderSetAlarm(void)
{
   if (firstTime)
   {
      GLCD_drawImage(0, 0, (uint8_t *)display, GLCD_WIDTH, GLCD_HEIGHT);
      GLCD_setXY(13,1);
      sprintf(str," SEL: NONE  ");
      GLCD_sendString(str);
      renderTime_alarm(1); //force render
      firstTime = 0;
      return;
   }
   renderTime_alarm(firstTime);
}

void renderTime_alarm(uint8_t force)
{
   if (DS1307_timeEquals(&time_alarm, &time_old) && !force)
   {
      return; //no need to render
   }
   time_old = time_alarm;
   sprintf(str, "%02d:%02d:%02d", time_alarm.hours, time_alarm.minutes, time_alarm.seconds);
   GLCD_setXY(23, 4);
   GLCD_sendString(str);
}
```

Puede verse que en lugar de llamarse a renderTime se llama a renderTime_alarm() que usa la variable *time_alarm*.

Luego, se verifica la entrada del usuario

- Si no se presionó nada, retornamos de la función
- Si se presionó CANCEL, llamamos al switchToIDLE() y ponemos en la variable *time_alarm* el valor anterior: *time_alarm_old*
- Si se presionó OK, se guarda en la variable *time_alarm_old* el valor del *time_alarm* y luego, llamamos al switchToIDLE() que nos cambia al estado IDLE.

- Los botones de izquierda y derecha, nos permitiran ir cambiando la selección actual. La selección actual, se guarda en current_selection y nos referencia el campo (hora, minuto o segundo) que estamos modificando actualmente.
- Los botones de arriba y abajo, nos permiten incrementar y decrementar el valor numérico de lo que estamos selecionando.

Del mismo modo que el estado anterior, tenemos la función de animación animateCurrentSelection_alarm() que muestra el texto con lo que estamos seleccionando (“SEL:xxx”).

```
void animateCurrentSelection_alarm(void){
   GLCD_setXY(13,1);
   switch(current_selection){
      case SELECTION_HOUR:
         sprintf(str,"SEL HS: %2d ",time_alarm.hours);
         //TODO: animate hour
      break;
      case SELECTION_MINUTE:
         sprintf(str,"SEL MIN: %2d",time_alarm.minutes);
         //TODO: animate minute
      break;
      case SELECTION_SECOND:
         sprintf(str,"SEL SEG: %2d",time_alarm.seconds);
         //TODO: animate second
      break;
      case SELECTION_NONE:
         sprintf(str," SEL: NONE  ");
      default: break;
   }
   GLCD_sendString(str);
}
```

El código entonces de todo este bloque del case es

```
case SETTING_ALARM:
      renderSetAlarm();
      if (buttonPressed == BUTTON_NONE)
         return;
      //CANCEL -> IDLE, don't save
      if (buttonPressed == BUTTON_CANCEL)
      {
         time_alarm=time_alarm_old;
         switchToIDLE();
         return;
      }
      //OK -> IDLE, save
      if (buttonPressed == BUTTON_OK)
      {
         time_alarm_old=time_alarm;
```

```
            switchToIDLE();
            return;
        }
        //UP -> increment
        if (buttonPressed == BUTTON_UP)
        {
            modify_alarm(current_selection, 1);
        }
        //DOWN -> decrement
        if (buttonPressed == BUTTON_DOWN)
        {
            modify_alarm(current_selection, -1);
        }
        //LEFT -> select next value
        if (buttonPressed == BUTTON_RIGHT)
        {
            current_selection++;
        }
        //RIGHT -> select previous value
        if (buttonPressed == BUTTON_LEFT)
        {
            current_selection--;
        }
        if (current_selection > SELECTION_END_ALARM)
            current_selection = SELECTION_START_ALARM;
        if (current_selection < SELECTION_START_ALARM)
            current_selection = SELECTION_END_ALARM;
        animateCurrentSelection_alarm();
        /*TODO*/
        break;
    }
```

Como puede verse, en vez de modify_date, llamamos a modify_alarm. Esta función privada, hace el comportamiento análogo a la que se usa en la modificación de fecha y hora, solo que modificando la variable *time_alarm*.

```
void modify_alarm(uint8_t selection, int8_t increment){
    switch (selection)
    {
    case SELECTION_HOUR:
        time_alarm.hours += increment;
        if (time_alarm.hours > 24)
            time_alarm.hours = 0;
        break;
    case SELECTION_MINUTE:
        time_alarm.minutes += increment;
        if (time_alarm.minutes > 60)
```

```
            time_alarm.minutes = 0;
        break;
    case SELECTION_SECOND:
        time_alarm.seconds += increment;
        if (time_alarm.seconds > 60)
            time_alarm.seconds = 0;
        break;
    default: break;
    }
    return;
}
```

2.10 Librería utils para delays

Recordando el TP1, 2 y 3 y que las librerías del MCU no nos proveen una función de delay, vamos a reutilizar la librería de utils que creamos. Recordando, el código de cabecera de esta librería es:

```
#ifndef UTILS_H_
#define UTILS_H_
#include <stm32f103x6.h>

#define CONST_FOR_US_DELAY 3
#define CONST_FOR_MS_DELAY 3000

void delay_ms(unsigned long);
void delay_us(unsigned long);

#endif
```

Que podemos ver que define las dos macros para el delay (que luego se utilizan para determinar la cantidad de iteraciones del bucle for que nos genera el delay).
A continuación puede verse el código:

```
#include <utils.h>


static unsigned long ms_delay_const = CONST_FOR_MS_DELAY;
static unsigned long us_delay_const = CONST_FOR_US_DELAY;

void delay_us(unsigned long amount){
        unsigned long i,l;
        for(i=0;i<amount;i++)
                for(l=0;l<us_delay_const;l++);
}
```

```
void delay_ms(unsigned long amount){
	unsigned long i,l;
	for(i=0;i<amount;i++)
		for(l=0;l<ms_delay_const;l++);
}
```

Algo a mencionar es que el delay_us no es perfecto, con valores muy bajos de delay se demora más tiempo. Esto se verá en la sección de validación pero por suerte no fue un problema (al menos en simulación) la baja precisión del mismo a la hora de implementar los protocolos de comunicación con los periféricos.

2.11 Periférico Usart

Nuevamente, debido a que teníamos a mano una implementación de la usart para este MCU, decidimos usarla para imprimir información extra de debug. Esta implementación ya la utilizamos en el trabajo práctico anterior, y solo se menciona porque se utilizó constantemente durante la fase uno del proyecto, en la que probamos individualmente los periféricos para mostrar información tal como la Temperatura y Humedad, los datos que estamos enviando al Display, entre otras cosas. En el proyecto de Github pueden verse varias carpetas con cada proyecto individual.

2.12 Programa principal

En esta sección, le daremos una rápida pasada el programa principal de nuestro proyecto.
Comencemos por la cabecera, como se puede ver contiene el import de todas las librerías que componen el sistema. Y a su vez, como en los anteriores tps la librería del MCU.
Por lo que el código de cabecera del main es el siguiente:

```
#ifndef MAIN_H
#define MAIN_H

#include <stm32f103x6.h>
#include "utils.h"
#include "dht22.h"
#include "ds1307.h"
#include "glcd.h"
#include "mef.h"


#endif
```

Ahora si la lógica en sí, en este caso nuestro main inicializa todos los periféricos utilizados, y luego entra en un loop infinito, en el cual cada 100 ms llama al método MEF update.
Cabe aclarar que en este caso, el delay utilizado para “dormir” el programa es bloqueante, ya que debido la naturaleza del sistema, nos pareció innecesario el uso de un timer que nos interrumpa cada 100 ms ya que a efectos prácticos sería lo mismo o peor. El razonamiento seguido fue el siguiente, una vez que el usuario configura por primera vez el sistema (setear hora y fecha actual y la hora a la

que la alarma debería sonar), es altamente probable que no interaccione nunca más con el sistema, salvo para apagar la alarma, que en el peor de los casos responderemos a ese estimulo 100 ms más tarde de lo que el usuario presiono el boton, totalmente imperceptible desde el lado del usuario. Tener además una interrupción periódica es meter complejidad en donde no es necesario. El sistema funciona a la perfección con este while (1). Podría reemplazarse tal vez el delay de 100ms por alguna función que ponga al MCU en bajo consumo por 100ms. Queda como mejora a futuro.

El código del main.c es el siguiente:

```
#include <stm32f103x6.h>
#include "main.h"

float temp,humidity;
char str[30];
RTC_TIME_t time;
RTC_DATE_t date;

int main (void){
  RCC->APB2ENR= 0xFC;
  MEF_Init();
  GLCD_Init();
  DS1307_Init();
  DHT22_Init();
  delay_ms(100);
  while(1){
    delay_ms(100);
    MEF_Update();
  }
  return 0;
}
```

3. Validación

3.1 Videos de simulador

Como es costumbre, dejaremos el enlace a un drive en donde pueden verse los videos de simulación grabados junto con la explicación del funcionamiento del sistema.

https://drive.google.com/drive/folders/1a38RQErOdhnOL-fzD8IHSn6wZpAlGcvv?usp=sharing

Tiene un solo video, editado, con voz, en el que se hace una demo del sistema.

3.2 Librería de github del proyecto

En el transcurso de todo el proyecto, además de usar las herramientas de Google para la redacción del informe, la planificación de la ruta del trabajo, la ejecución del script de python, entre otras cosas, se usó Github, como estamos acostumbrados. En el proyecto de github, en la rama master pueden verse una carpeta por cada proyecto, de Proteus.

- La carpeta Auxiliares tiene los auxiliares del display gráfico, junto con el script de python. Acá hicimos varias pruebas y tuvimos varias ideas, al final no pudimos hacer animaciones ya que el micro se quedaba sin memoria para guardar tantas constantes.
- Luego está la carpeta de Datasheets, en donde hemos guardado todos los datasheets de los periféricos que ibamos usando para tenerlos a mano.
- En las carpetas DHT22, DHT22 with Display, DHT22_Fede, Display_SPI_FEDE, RTC, SPI+Display, UART Proteus, tenemos varias pruebas que fuimos haciendo antes de mezclar todo en el firmware
- En la carpeta de Firmware se encuentra el proyecto de Proteus con el sistema listo para simular, con código y todo.
- En la carpeta Schematic se encuentra una versión del esquemático del sistema, pero la versión final está en una de las branches
- Luego, en la raíz del proyecto tenemos el diagrama de la MEF en formatos drawio y png y el pdf con la propuesta del trabajo.

Cada rama tiene también desarrollos asociados, hay una rama donde se empezó y terminó el desarrollo del PCB y otra para el Schematic.

4. Diseño de Hardware

Antes de comenzar con lo que fue la construcción del PCB, arranquemos por lo primero el esquemático.

En la siguiente imagen se puede ver lo que es el esquemático del sistema

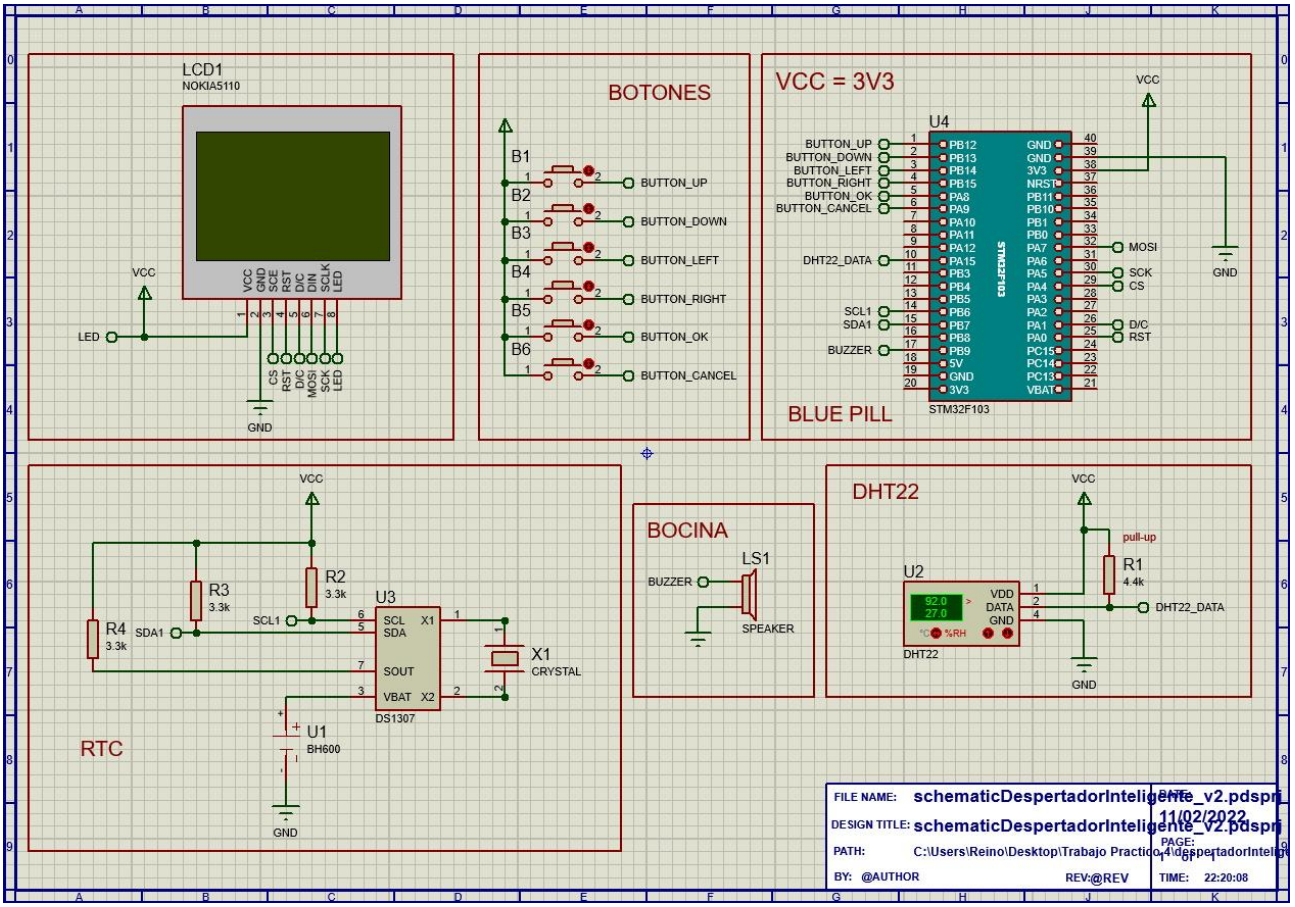


Figura 4.0 - Schematic

Al igual que en el trabajo anterior, no vemos lo que eran los módulos de boot, reset, alimentación y oscilador. Esto se debe, a que como en este trabajo desarrollamos el PCB del sistema, necesitábamos contar con una placa comercial y no con el MCU "pelado". En este caso se optó por la blue pill, que internamente ya trae todos los módulos antes descritos.

Uno se podría preguntar, porque no armamos nuestra propia placa si teníamos el esquemático para hacerla (TP 2), y la respuesta es sencilla, para fabricar una placa no se puede hacer un PCB casero, ya que la separación que marcan las distintas hojas de datos entre terminales por ejemplo son casi imposible de cumplir trazando las vías a "mano". Por eso se optó por utilizar la placa comercial.

Luego en el esquemático, se puede ver el display gráfico utilizado, el sensor de temperatura. Y por último, el Real Time Counter (RTC), que como se puede ver está compuesto por 3 resistores (de 3.3 KΩ), un dispositivo DS1307, un osilador a cristal y una bateria de 3V (representada con un portapilas)

Adjuntamos también la lista de materiales del proyecto

Bill Of Materials for schematicDespertadorInteligente_v2

Design Title	schematicDespertadorInteligente_v2
Author	
Document Number	
Revision	
Design Created	Thursday, 10 February 2022
Design Last Modified	Friday, 11 February 2022
Total Parts In Design	17

4 Resistors

Quantity	References	Value
1	R1	4.4k
3	R2,R3,R4	3.3k

Sub-totals:

4 Integrated Circuits

Quantity	References	Value
1	U1	BH600
1	U2	DHT22
1	U3	DS1307
1	U4	STM32F103

Sub-totals:

9 Miscellaneous

Quantity	References	Value
6	B1,B2,B3,B4,B5,B6	BUTTON
1	LCD1	NOKIA5110
1	LS1	SPEAKER
1	X1	CRYSTAL

Sub-totals:

Totals:

Friday, 11 February 2022 22:19:19

Para el PCB se volvió a utilizar la Placa Blue Pill Custom que armamos en el Trabajo Práctico anterior. Además, por recomendación del profesor, se cambió el DS1307 a una versión custom para usar los pads de 70 x 30. También se evitó el uso de curvas de pistas de 90° para evitar posibles fallos en la fabricación de la placa o la posibilidad de emisión de interferencias electromagnéticas. Por último, se utilizaron Push Buttons Custom que cumplen con las características indicadas en la figura 4.0.1 y se agregó un portapilas Custom tal y como nos indicaron en la Pre-Entrega.

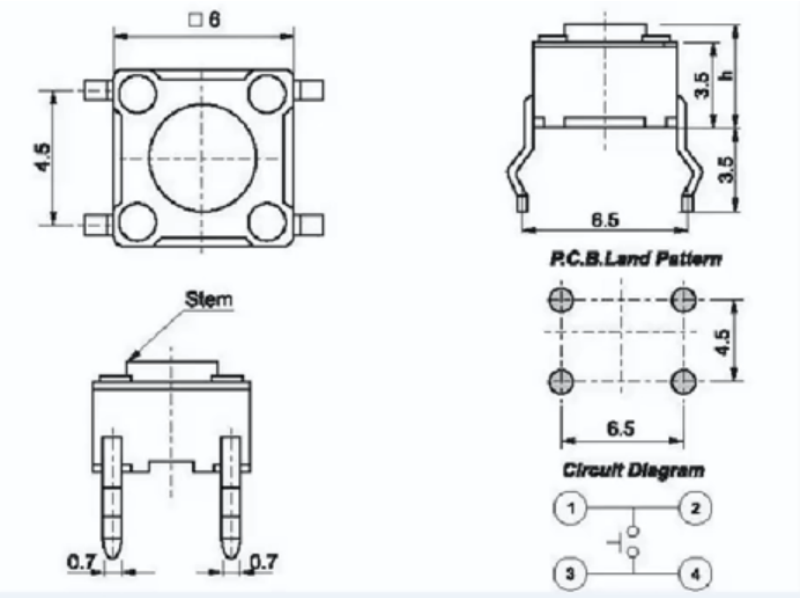


Figura 4.0.1 - Anexo del Botón

Se intentó modificar la huella del cristal a un AB26T. Si bien funciona correctamente el modelado 3D, al intentar utilizar el PCB la primer opción es muy grande y no coincide con el adjunto que nos envió el profesor:
https://datasheet.octopart.com/AB26T-32.768KHZ-Abracon-datasheet-37142359.pdf
Por otro lado, la segunda opción si coincide con el adjunto, pero solamente nos permite la conexión mediante el Top Copper.

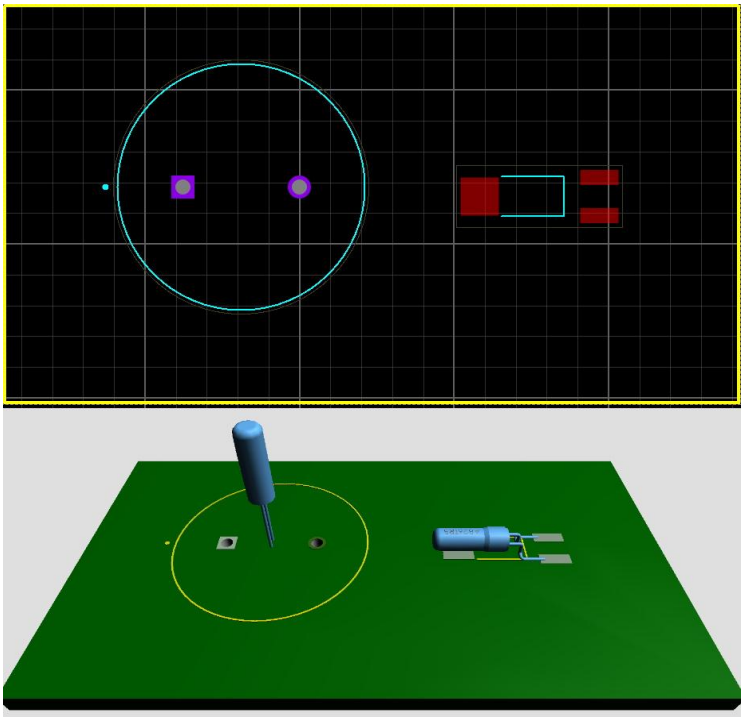


Figura 4.0.2 - Anexo de los Crystales

Por lo tanto, se decidió crear un componente custom con las medidas de la segunda opción, y el tipo de conexión de la primer opción, dejando como resultado la *Figura 4.0.3*

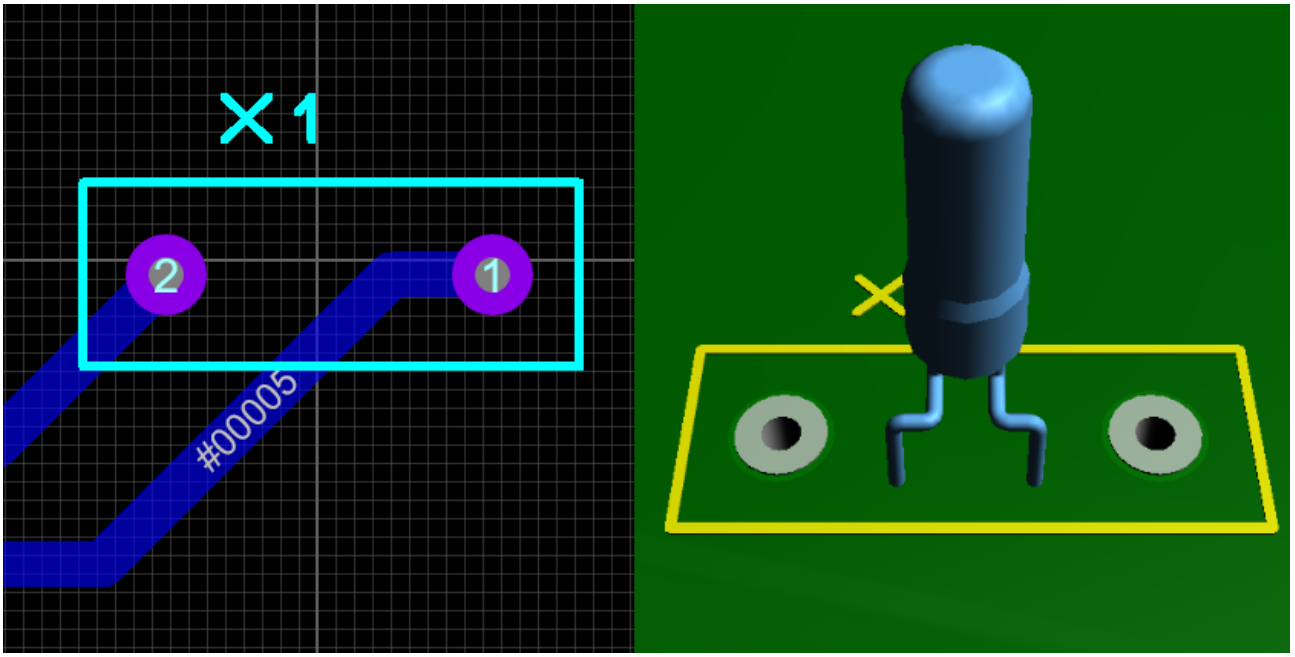


Figura 4.0.2 - Anexo del Crystal Custom

Se adjunta el PCB completo resultante y la visualización 3D.

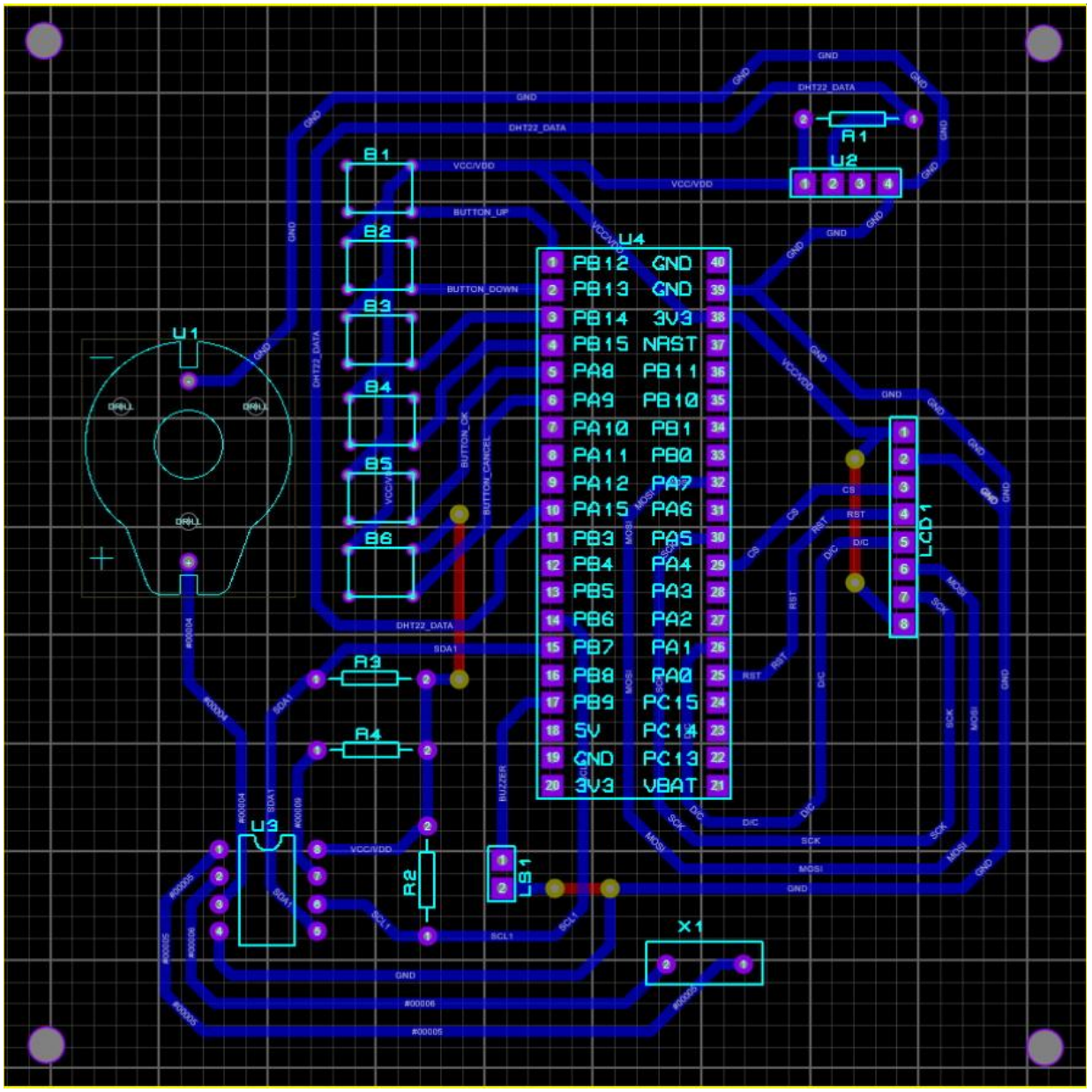


Figura 4.1 - PCB del sistema Completo

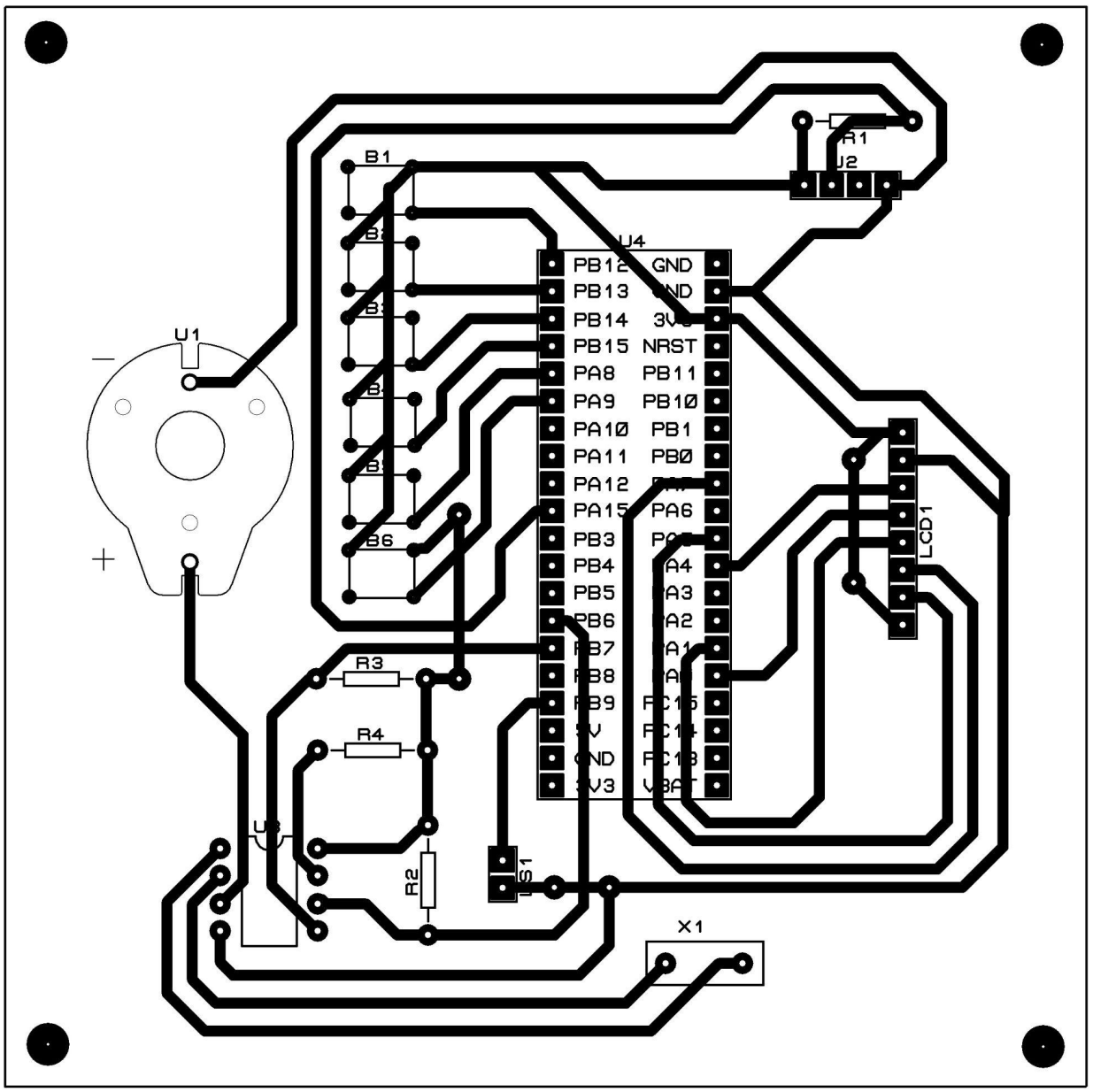


Figura 4.2 - Impresión del sistema con Capa de Serigrafía

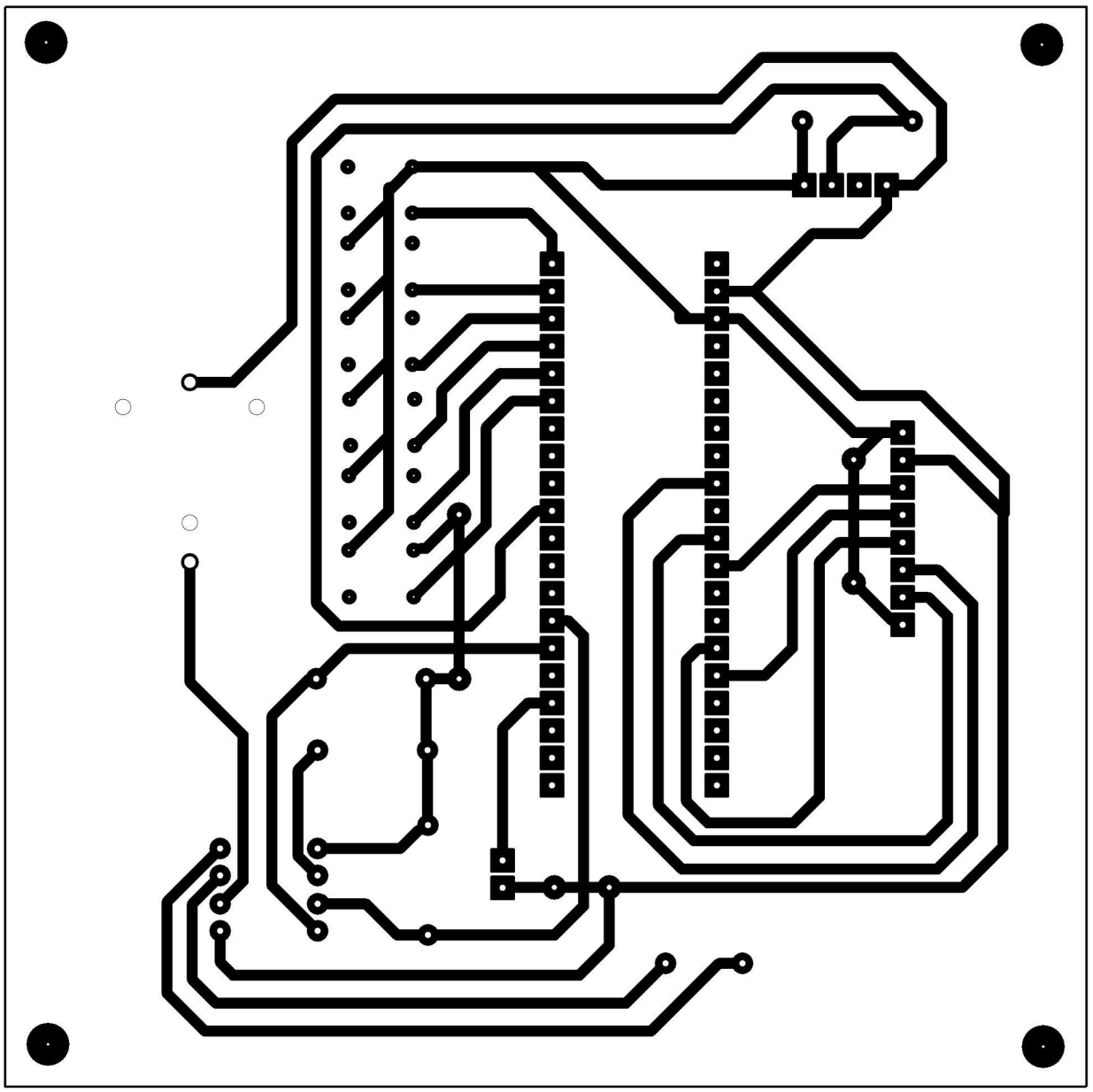


Figura 4.3 - Impresión del sistema sin Capa de Serigrafía

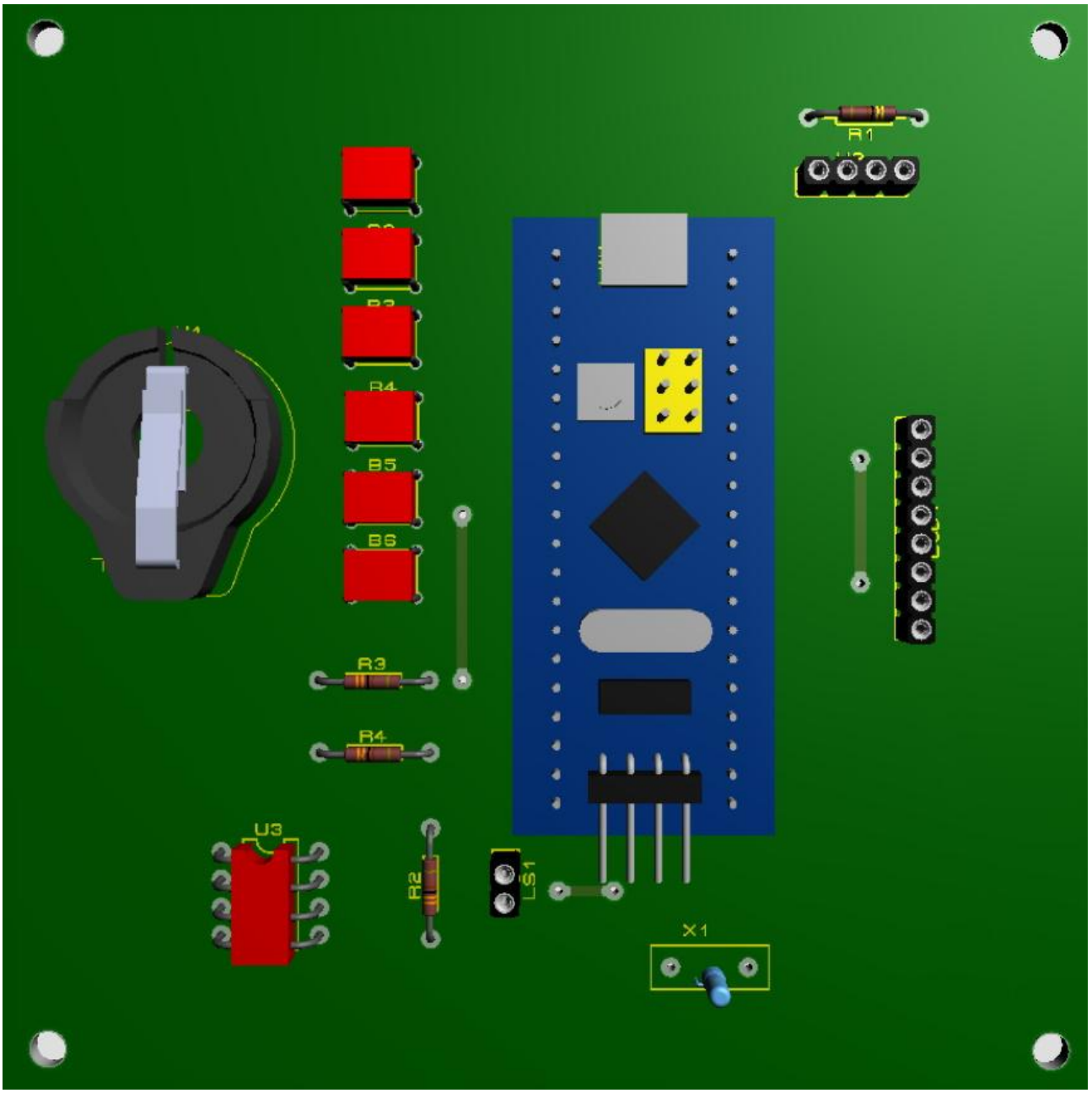


Figura 4.4 - Vista 3D con Componentes. Cara superior.

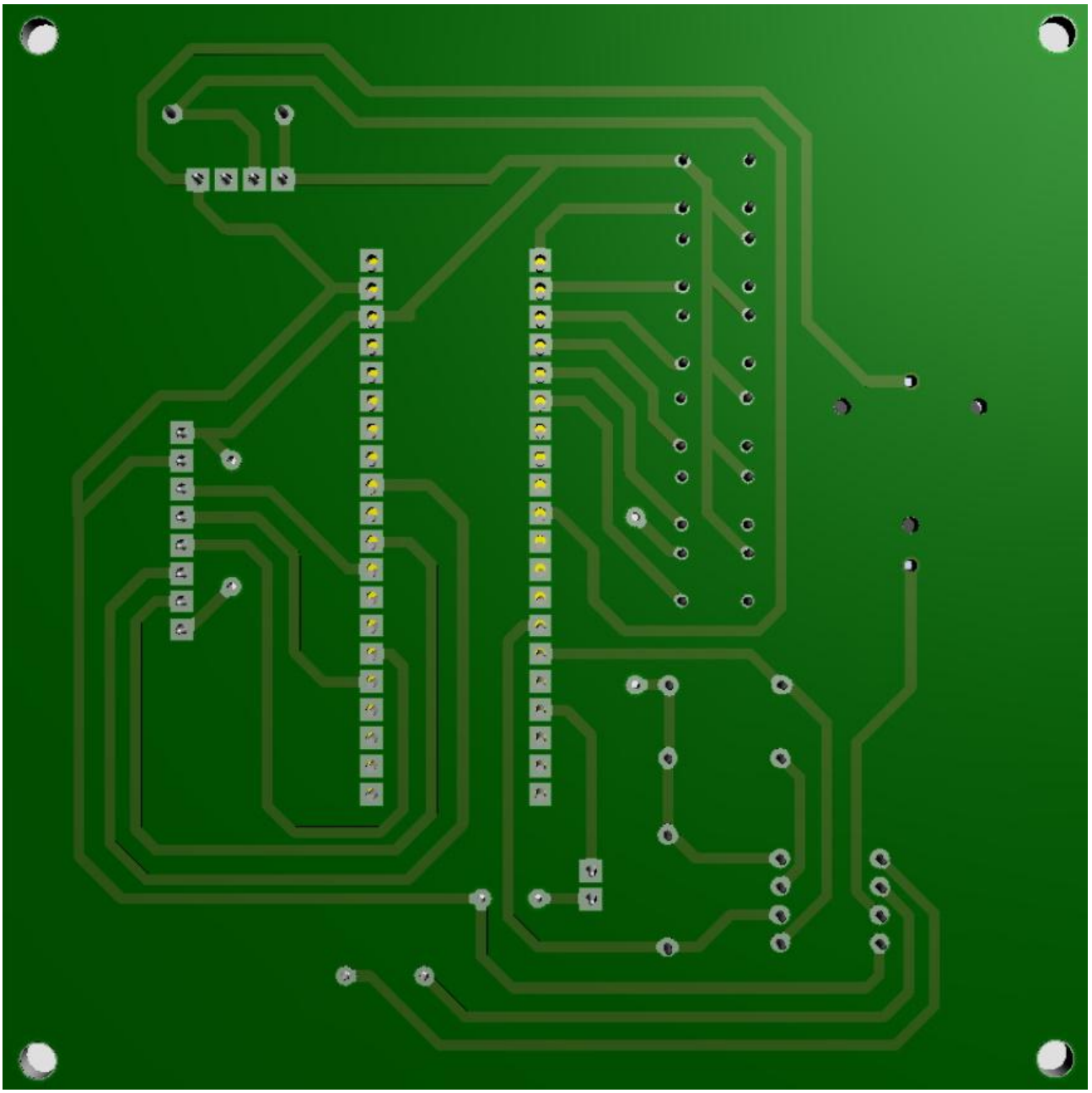


Figura 4.5 - Vista 3D con Componentes. Cara inferior.

5 Conclusiones

Como hemos mencionado, este proyecto se hizo cuesta arriba por las limitaciones del simulador. Varias ideas y funcionalidades que queríamos meter no llegamos a implementarlas a causa del tiempo que nos tomó resolver los problemas.
Sin embargo nos llevamos una experiencia muy linda de haber podido armar un sistema completo de algo que en su momento solíamos usar cotidianamente y no teníamos idea de cómo funcionaba por dentro. Además de haber podido poner en práctica todos los conocimientos que venimos arrastrando de las distintas materias de la carrera, destacamos que, a diferencia de hace un año, tenemos la capacidad de agarrar hojas de datos de encapsulados, interpretarlas y extraer de ellas toda la información importante sin marearnos en el camino.
También, destacamos la importancia de tener un plan de acción, una ruta de trabajo de la cuál agarrarnos, bien definida antes de hacer nada. Gracias a haber probado todos los periféricos individualmente pudimos encontrar los errores de cada uno de ellos en el simulador y asegurarnos que todo estuviera funcional antes de armar el sistema grande. Haber realizado tantos trabajos en conjunto nos permite además resolver los problemas con muchísima eficacia ya que sabemos en quién delegar cada responsabilidad del proyecto y nos potenciamos entre sí.
Por último, la espina que nos queda clavada es poder llevar esto a la vida real y ver si funciona. Tal vez en un futuro lo consigamos.

6. Código

6.1 Librería I2C

6.1.1 I2C.h

```
#ifndef i2c_h
#define i2c_h
#include <stm32f103x6.h>
#include "utils.h"

#define I2C_SCL_PIN 6 //Pin de SCL
#define I2C_SDA_PIN 7 //Pin de SDA
#define I2C_DR GPIOB->ODR //DirecciÃ³n del registro
#define I2C_DR_IN GPIOB->IDR //DirecciÃ³n del registro de entrada

void I2C_Init(void);
void I2C_SendStart(void);
void I2C_SendStop(void);
unsigned char I2C_SendAddrForWrite(unsigned char);
unsigned char I2C_SendAddrForRead(unsigned char);
unsigned char I2C_SendData(unsigned char);
unsigned char I2C_ReadData(unsigned char);

#endif
```

6.1.2 I2C.c

```
#include "i2c.h"

#define I2C_set_scl() { I2C_DR |= (1 << I2C_SCL_PIN); } //Poner SCL en alto
#define I2C_clear_scl() { I2C_DR &= ~(1 << I2C_SCL_PIN); } //Poner SCL en
bajo
#define I2C_set_sda() { I2C_DR |= (1 << I2C_SDA_PIN); } //Poner SDA en alto
#define I2C_clear_sda() { I2C_DR &= ~(1 << I2C_SDA_PIN); } //Poner SDA en
bajo
#define I2C_bus_init() { I2C_DR |= ((1 << I2C_SDA_PIN) | (1 <<I2C_SCL_PIN));
} //Poner ambos pines en alto
#define I2C_get_sda I2C_DR_IN &  (1 << I2C_SDA_PIN) //Obtener valor de SDA

//Half a bit delay
#define HALF_BIT_DELAY delay_us(5);
//I2C doesn't work on STM32. We implement it by software. It's super ADHOC.
void I2C_Init(){
    //Config GPIO pins
    RCC->APB2ENR |= (0xFC); /* enable clocks for GPIOs */
    //RCC->APB1ENR |= (1<<21); /* enable clock for I2C1 */
    GPIOB->CRL |= 0x77000000; /* configure PA6 and PA7 as GPO. open drain */
    I2C_set_scl();
```

```
    I2C_set_sda();
}

//Send start signal
void I2C_SendStart(){
    I2C_bus_init(); //put both pins on high

    //Wait half bit, put SDA low
      HALF_BIT_DELAY
    I2C_clear_sda(); //SDA goes low

    //Wait half bit, put SCL LOW
    HALF_BIT_DELAY
    I2C_clear_scl(); //SCL goes low
    HALF_BIT_DELAY
}

//Send stop signal
void I2C_SendStop(){
    I2C_clear_sda(); //SDA goes low
    I2C_clear_scl(); //SCL goes low
    HALF_BIT_DELAY

    //Wait half bit, put SCL HIGH
    I2C_set_scl(); //SCL goes high
    HALF_BIT_DELAY

    //Wait half bit, put SDA HIGH
    I2C_set_sda(); //SDA goes high
    HALF_BIT_DELAY

}

//This function will send Address for read
uint8_t I2C_SendAddrForRead(uint8_t addr){
    return I2C_SendData((addr<<1) | 1); //addr + Read(1)
}

//This function will send Address for write
uint8_t I2C_SendAddrForWrite(uint8_t addr){
    return I2C_SendData(addr<<1); //addr+Write(0)
}

//This function will send data to the slave
//Return 1 if ACK received, 0 if NACK received
uint8_t I2C_SendData(uint8_t data){
```

```
    unsigned char msk = 0x80; //Mask, starts in 1000 0000
    unsigned char ack;        //Acknowledge received from slave

    //We transmit the 7+1 bits of data
    do
    {
        //We use the mask to check if the bit is 1 or 0
        if (data & msk)  {I2C_set_sda();}
        else {I2C_clear_sda();}

        //Set SCL, then wait half bit and clear SCL, in order for slave to
read SDA.
        I2C_set_scl();
        HALF_BIT_DELAY
        I2C_clear_scl();
    }
    while ((msk>>=1) != 0); //Shift mask to the right 1 bit and check if it's
0

    //After sending 8 bits, we set both SDA and SCL, and wait half a bit
    I2C_set_sda();
    I2C_set_scl();
    HALF_BIT_DELAY

    //Here, if ACK is sent by slave, SDA will be drained to low, otherwise it
will be high
    //So we read it.
    ack =(I2C_get_sda);

    //Finally, we clear SCL before leaving function
    I2C_clear_scl();
    HALF_BIT_DELAY

    //We return true if sending data was successful. False otherwise.
    return (!ack);
}

//This function will read data from the slave
//It receives ack which is used in burst reading to know when to stop
//Data read is returned
uint8_t I2C_ReadData(uint8_t send_nack){
    unsigned char msk = 0x80; //Mask, starts in 1000 0000
    unsigned char b = 0;            //Buffer for the received data

    //Receive bit by bit
    do
```

```
    {
        //Set SCL and wait half bit
        I2C_set_scl();
        HALF_BIT_DELAY

        //We read SDA, if it is 1 we set the bit in the buffer (using mask to
know which bit)
        if(I2C_get_sda)  b|=msk;

        //Clear SCL and wait half bit for next bit
        I2C_clear_scl();
        HALF_BIT_DELAY
    }
    while ((msk>>=1) != 0); //We shift mask to the right 1 bit and check if
it's 0 in order to read every bit.

    //We check ack parameter to know if we need to send ACK or NACK
    if(send_nack == 0) //We send ACK (still not finished reading)
    {
        I2C_clear_sda();
    }
    else //send_nack = 1, We send NACK (notify slave we end read )
    {
        I2C_set_sda();
    }

    //Set SCL to send ACK/NACK (then wait half bit)
    I2C_set_scl();
    HALF_BIT_DELAY
    //Clear SCL to finish sending ACK/NACK
    I2C_clear_scl();
    //Free SDA line. Then wait half bit we are ready for next operation
    I2C_set_sda();
    HALF_BIT_DELAY

    //Return read data
    return (b);
}
```

6.2 Librería LCD

6.2.1 LCD.h

```
#ifndef LCD_H
#define LCD_H

#include <stm32f103x6.h>
```

```
#include <utils.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LCD_CLR 0 //DB0: clear display

//PARA CAMBIAR EL PUERTO CAMBIAR GPIOx DONDE x CORRESPONDE AL PUERTO
#define LCD_PORT        GPIOB->CRH
#define LCD_PIN_IN            GPIOB->IDR              //Input Data Register
#define LCD_PIN_OUT     GPIOB->ODR              //Output Data Register

//Since BRR and BSRR are opposite of each other,
//you can use both if you don't want to do the bit shift left operation
#define LCD_PORT_BSRR   GPIOB->BSRR
#define LCD_PORT_BRR    GPIOB->BRR

#define LCD_RS 10
#define LCD_RW 12
#define LCD_EN 11
// cursor position to DDRAM mapping
#define LCD_LINE0_DDRAMADDR 0x00
#define LCD_LINE1_DDRAMADDR 0x40
#define LCD_LINE2_DDRAMADDR 0x14
#define LCD_LINE3_DDRAMADDR 0x54
#define LCD_DDRAM 7     //DB7: set DD RAM address -> 1<<LCD_DDRAM => 0x80

/*CABECERAS*/
uint8_t string_len(char* data);
void LCDinit(void);                 //Initializes LCD
void LCDsendChar(uint8_t);          //forms data ready to send to 74HC164
void LCDsendCommand(uint8_t); //forms data ready to send to 74HC164
//void lcd_putValue (uint8_t value); la movÃ porque es privada
void LCDstring(uint8_t*, uint8_t);  //Outputs string to LCD
void LCDSendInt(unsigned int i); //Outputs int as string to LCD

//TODO - Son necesarias para la MEF
void LCDGotoXY(uint8_t, uint8_t);   //Cursor to X Y position
void LCDclr(void);                        //Clears LCD
void LCDcursorOFF(void);            //Cursor OFF
#endif
```

6.2.2 LCD.c

```
#include <stm32f103x6.h>
#include <lcd.h>
```

```
#include <utils.h>//Delay timer

 //Cabecera de funciÃ³n privada
 void LCDputValue(uint8_t value);

/*Outputs string to LCD.
Receives the String + the size of the String in Characters*/
void LCDstring(uint8_t* data, uint8_t nBytes)
{
uint8_t i;

      if (!data) return;                  //check to make sure we have a good
pointer

      for(i=0; i<nBytes; i++){            //Print data in LCD
            LCDsendChar(data[i]);
      }
}

//Inicializacion del LCD en modo 4 bits
//Â¡Â¡Setea los pines!!
void LCDinit(){

  LCD_PORT = 0x33333344;            /* PB10-PB15 as outputs */
  LCD_PIN_OUT &= ~(1<<LCD_EN);      //LCD_EN=0

  delay_ms(3);                              //Delay de 3ms
  LCDsendCommand(0x33);                     //Send $33 for init
  LCDsendCommand(0x32);                     //Send $32 for init
  LCDsendCommand(0x28);                     //Init LCD 2 line, 5x7 Matrix
  LCDsendCommand(0x0c);                     //Display On, Curson On
  LCDsendCommand(0x01);                     //Clear LCD

  delay_ms(2);                              //Delay de 2ms
  LCDsendCommand(0x06);//Shift Cursor Right
}

//Codigo para enviar un comando al LCD
void LCDsendCommand (uint8_t cmd)
{
   LCD_PORT_BRR = (1<<LCD_RS); /* RS = 0 for command */
   LCDputValue(cmd);
}

//Codigo para enviar un Char al LCD
void LCDsendChar (uint8_t data)
```

```
{
   LCD_PORT_BSRR = (1<<LCD_RS); /* RS = 1 for data */
   LCDputValue(data);
}

/* Codigo para enviar valores al LCD.
Funciona en 4 bits. Primero envia la parte superior,
y luego envia la parte inferior del dato.
*/
void LCDputValue(unsigned char value)
{
   LCD_PORT_BRR = 0xF000;                                    /* clear PA0-PA3
PB12-PB15*/
   LCD_PORT_BSRR = (value<<8)&0xF000;                /* put high nibble on
PA0-PA3 PB12-PB15 */
   LCD_PIN_OUT |= (1<<LCD_EN);                       /* EN = 1 for H-to-L
pulse */
   delay_us(1);                                                 /* make EN
pulse wider. You can use delay_us(2); too */
   LCD_PIN_OUT &= ~ (1<<LCD_EN);               /* EN = 0 for H-to-L pulse */
   delay_ms(2);                                              /* wait */

   LCD_PORT_BRR = 0xF000;                                    /* clear PA0-PA3
PB12-PB15*/
   LCD_PORT_BSRR = (value<<12)&0xF000;                   /* put low nibble
on PA0-PA3 PB12-PB15*/
   LCD_PIN_OUT |= (1<<LCD_EN);                       /* EN = 1 for H-to-L
pulse */
   delay_us(1);                                                 /* make EN
pulse wider */
   LCD_PIN_OUT &= ~(1<<LCD_EN);                /* EN = 0 for H-to-L pulse */
   delay_ms(2);                                              /* wait */
}

//Cursor to X Y position
void LCDGotoXY(uint8_t x, uint8_t y){
      register uint8_t DDRAMAddr;
      // remap lines into proper order
      switch(y)
      {
      case 0: DDRAMAddr = LCD_LINE0_DDRAMADDR+x; break;
      case 1: DDRAMAddr = LCD_LINE1_DDRAMADDR+x; break;
      case 2: DDRAMAddr = LCD_LINE2_DDRAMADDR+x; break;
      case 3: DDRAMAddr = LCD_LINE3_DDRAMADDR+x; break;
      default: DDRAMAddr = LCD_LINE0_DDRAMADDR+x;
      }
```

```
	// set data address
	LCDsendCommand(1<<LCD_DDRAM | DDRAMAddr);
}
//Clears LCD
void LCDclr(void){
	LCDsendCommand(1<<LCD_CLR);
}
//Cursor OFF
void LCDcursorOFF(void){
	LCDsendCommand(0x0C);
}

void LCDSendInt(unsigned int i){
	char str[10];
	sprintf(str,"%d",i);
	LCDstring((unsigned char*)str,strlen(str));
}
```

6.3 Librería DS1307

6.3.1 DS1307.h

```
#ifndef ds1307_h
#define ds1307_h
#include <stm32f103x6.h>

#include "i2c.h"

#define DS1307_ADDRESS 0x68 // 7 bit address (add )
#define DS1307_SECONDS_REGISTER 0x00
#define DS1307_MINUTES_REGISTER 0x01
#define DS1307_HOURS_REGISTER 0x02
#define DS1307_DAY_REGISTER 0x03
#define DS1307_DATE_REGISTER 0x04
#define DS1307_MONTH_REGISTER 0x05
#define DS1307_YEAR_REGISTER 0x06
#define DS1307_CONTROL_REGISTER 0x07

#define MONDAY 1
#define TUESDAY 2
#define WEDNESDAY 3
#define THURSDAY 4
#define FRIDAY 5
#define SATURDAY 6
#define SUNDAY 7

typedef struct{
    uint8_t seconds;
```

```
    uint8_t minutes;
    uint8_t hours;
}RTC_TIME_t;

typedef struct{
    uint8_t day;
    uint8_t date;
    uint8_t month;
    uint8_t year;
}RTC_DATE_t;


void DS1307_Init(void);
void DS1307_set_time(RTC_TIME_t *time);
void DS1307_get_time(RTC_TIME_t *time);
void DS1307_set_date(RTC_DATE_t *date);
void DS1307_get_date(RTC_DATE_t *date);
void DS1307_set_full_time(RTC_TIME_t *time, RTC_DATE_t *date);
uint8_t * DS1307_get_time_string(RTC_TIME_t time);
uint8_t * DS1307_get_date_string(RTC_DATE_t date);
uint8_t * DS1307_get_date_string_with_day(RTC_DATE_t date);

//Utilitary to compare dates and times
uint8_t DS1307_dateEquals(RTC_DATE_t *date1, RTC_DATE_t *date2);
uint8_t DS1307_timeEquals(RTC_TIME_t *time1, RTC_TIME_t *time2);
void DS1307_copyTime(RTC_TIME_t *dest, RTC_TIME_t *src);
void DS1307_copyDate(RTC_DATE_t *dest, RTC_DATE_t *src);
#endif
```

6.3.2 DS1307.c

```
#include "ds1307.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

static void DS1307_config(void);
static uint8_t bcd2int(uint8_t);
static uint8_t int2bcd(uint8_t);
static uint8_t* day2string(uint8_t);


void DS1307_Init(void)
{
    I2C_Init();
    DS1307_config();
}
```

```
void DS1307_set_full_time(RTC_TIME_t *time, RTC_DATE_t *date)
{
    DS1307_set_date(date);
    DS1307_set_time(time);
}

void DS1307_set_time(RTC_TIME_t *time)
{
    do
      {
          I2C_SendStart();                              /* generate a start
condition */
      } while (I2C_SendAddrForWrite(0x68) == 0); /* repeat if returned false
*/

      I2C_SendData(0x0);      /* set addr. pointer to 0 */
      I2C_SendData(int2bcd(time->seconds) & 0x7F); /* seconds without 7th bit
*/
      I2C_SendData(int2bcd(time->minutes)); /* min */
      I2C_SendData(int2bcd(time->hours)); /* hour in 24-h format*/
    I2C_SendStop(); //generate a stop condition
}

void DS1307_get_time(RTC_TIME_t *time)
{
    if (time == 0)
        return;

    do
    {
        I2C_SendStart();                       /* generate a start condition
*/
    } while (I2C_SendAddrForWrite(0x68) == 0); /* repeat if returned false */

    I2C_SendData(0x00); /* set addr. pointer to 0 */

    do
    {
        I2C_SendStart();                      /* generate a REPEATED START
condition */
    } while (I2C_SendAddrForRead(0x68) == 0); /* repeat if returned false */

    time->seconds = bcd2int(I2C_ReadData(0));
    time->minutes = bcd2int(I2C_ReadData(0));
    //Clean hours 6th bit by masking it
```

```
    time->hours = bcd2int(I2C_ReadData(1) & 0xBF); //send nack

    I2C_SendStop(); /* generate a stop condition */
}

void DS1307_set_date(RTC_DATE_t *date)
{
    do
      {
          I2C_SendStart();                              /* generate a start
condition */
      } while (I2C_SendAddrForWrite(0x68) == 0); /* repeat if returned false
*/

      I2C_SendData(0x03);     /* set addr. pointer to 0 */
      I2C_SendData(int2bcd(date->day)); /* day of week */
      I2C_SendData(int2bcd(date->date)); /* day of month */
      I2C_SendData(int2bcd(date->month)); /* month */
      I2C_SendData(int2bcd(date->year)); /* year */
      I2C_SendStop(); /* generate a stop condition */
}

void DS1307_get_date(RTC_DATE_t *date)
{
    if (date == 0)
        return;

    do
    {
        I2C_SendStart();                       /* generate a start condition
*/
    } while (I2C_SendAddrForWrite(0x68) == 0); /* repeat if returned false */

    I2C_SendData(0x03); /* set addr. pointer to 0 */

    do
    {
        I2C_SendStart();                      /* generate a REPEATED START
condition */
    } while (I2C_SendAddrForRead(0x68) == 0); /* repeat if returned false */

    date->day = bcd2int(I2C_ReadData(0));
    date->date = bcd2int(I2C_ReadData(0));
    date->month = bcd2int(I2C_ReadData(0));
    date->year = bcd2int(I2C_ReadData(1));
```

```
    I2C_SendStop(); /* generate a stop condition */
}

uint8_t * DS1307_get_time_string(RTC_TIME_t time){
    uint8_t *time_string = (uint8_t *) malloc(sizeof(uint8_t) * 6);
    if (time_string == 0)
        return 0;
    sprintf((char*)time_string, "%02d:%02d:%02d", time.hours, time.minutes,
time.seconds);
    return time_string;

}
uint8_t * DS1307_get_date_string(RTC_DATE_t date){
    uint8_t *date_string = (uint8_t *) malloc(sizeof(uint8_t) * 11);
    if (date_string == 0)
        return 0;

    sprintf((char*)date_string, "%02d/%02d/%02d", date.date, date.month,
date.year);
    return date_string;
}
uint8_t * DS1307_get_date_string_with_day(RTC_DATE_t date){
    uint8_t *date_string = (uint8_t *) malloc(sizeof(uint8_t) * 11);
    uint8_t *day = day2string(date.day);
    if (date_string == 0)
        return 0;
    sprintf((char *)date_string, "%s %s", day, DS1307_get_date_string(date));
    free(day);
    return date_string;
}


//private functions
static void DS1307_config(){
    do
      {
            I2C_SendStart();                          /* generate a start
condition */
      } while (I2C_SendAddrForWrite(0x68) == 0); /* repeat if returned false
*/

      I2C_SendData(0x0);      /* set addr. pointer to 0 */
      I2C_SendData(0x50); /* second */
      I2C_SendData(0x00); /* min */
      I2C_SendData(0x0A); /* hour in 24-h format*/
      I2C_SendData(MONDAY); /* day of week */
```

```
      I2C_SendData(0x14); /* day of month */
      I2C_SendData(0x02); /* month */
      I2C_SendData(0x22); /* year */
      I2C_SendStop(); /* generate a stop condition */
}

void DS1307_get_full_time(RTC_TIME_t *time, RTC_DATE_t *date)
{

      if (date == NULL || time == NULL)
            return;

    DS1307_get_date(date);
    DS1307_get_time(time);
}
/* The function gets a BCD number and converts it to binary */
static uint8_t bcd2int(uint8_t n)
{
      return ((n & 0xF0) >> 4) * 10 + (n & 0x0F);
}
static uint8_t int2bcd(uint8_t n)
{
    return ((n / 10) << 4) + (n % 10);
}

static uint8_t* day2string(uint8_t day_r){
    uint8_t * day = (uint8_t *) malloc(sizeof(uint8_t) * 3);
    switch(day_r){
        case MONDAY:
            strcpy((char*)day, "LUN");
            break;
        case TUESDAY:
            strcpy((char*)day, "MAR");
            break;
        case WEDNESDAY:
            strcpy((char*)day, "MIE");
            break;
        case THURSDAY:
            strcpy((char*)day, "JUE");
            break;
        case FRIDAY:
            strcpy((char*)day, "VIE");
            break;
        case SATURDAY:
            strcpy((char*)day, "SAB");
            break;
```

```
        case SUNDAY:
            strcpy((char*)day, "DOM");
            break;
        default:
            strcpy((char*)day, "ERR");
            break;
    }
    return day;
}
uint8_t DS1307_dateEquals(RTC_DATE_t *date1, RTC_DATE_t *date2){
    return (date1->day == date2->day && date1->date == date2->date &&
date1->month == date2->month && date1->year == date2->year);
}
uint8_t DS1307_timeEquals(RTC_TIME_t *time1, RTC_TIME_t *time2){
    return (time1->hours == time2->hours && time1->minutes == time2->minutes
&& time1->seconds == time2->seconds);
}

void DS1307_copyTime(RTC_TIME_t *dest, RTC_TIME_t *src){
    dest->hours = src->hours;
    dest->minutes = src->minutes;
    dest->seconds = src->seconds;
}
void DS1307_copyDate(RTC_DATE_t *dest, RTC_DATE_t *src){
    dest->day = src->day;
    dest->date = src->date;
    dest->month = src->month;
    dest->year = src->year;
}
```

6.4 Librería GLCD

6.4.1 GLCD.h

```
#ifndef GLCD_H
#define GLCD_H

#include <stm32f103x6.h>
#include "spi_drive.h"
#include "utils.h"

/*MACROS (change for port)*/
#define DISPLAY_PORT GPIOA
#define DISPLAY_PORT_DR GPIOA->ODR
#define DISPLAY_PORT_CONFIG GPIOA->CRL

#define GLCD_RST 0 //Reset pin
#define GLCD_DC 1 //Data/Command pin
```

```
#define GLCD_CS 4 //Chip select pin -> also slave select of SPI

#define GLCD_SCK 5      //Clock pin
#define GLCD_DIN 7 //Data in pin

#define GLCD_WIDTH 84
#define GLCD_HEIGHT 6

/* Function Headers */

void GLCD_Init(void);
void GLCD_sendChar(char);
void GLCD_sendString(char *);
void GLCD_setXY(uint8_t x, uint8_t y);
void GLCD_clean(void);
void GLCD_drawImage(uint8_t,uint8_t,uint8_t*,uint8_t,uint8_t);
void GLCD_drawImageNonDestructive(uint8_t,uint8_t,uint8_t*,uint8_t,uint8_t);

/* https://playground.arduino.cc/Code/PCD8544/
Changed byte to Char for C code.
*/
static const uint8_t ASCII[][5] =
{
 {0x00, 0x00, 0x00, 0x00, 0x00} // 20
,{0x00, 0x00, 0x5f, 0x00, 0x00} // 21 !
,{0x00, 0x07, 0x00, 0x07, 0x00} // 22 "
,{0x14, 0x7f, 0x14, 0x7f, 0x14} // 23 #
,{0x24, 0x2a, 0x7f, 0x2a, 0x12} // 24 $
,{0x23, 0x13, 0x08, 0x64, 0x62} // 25 %
,{0x36, 0x49, 0x55, 0x22, 0x50} // 26 &
,{0x7e, 0x04, 0x09, 0x10, 0x7e} // 27 ' -> se reemplaza por Ã'
,{0x00, 0x1c, 0x22, 0x41, 0x00} // 28 (
,{0x00, 0x41, 0x22, 0x1c, 0x00} // 29 )
,{0x00, 0x0e, 0x0A, 0x0e, 0x00} // 2a * -> Se reemplaza por Â°
,{0x08, 0x08, 0x3e, 0x08, 0x08} // 2b +
,{0x00, 0x50, 0x30, 0x00, 0x00} // 2c ,
,{0x08, 0x08, 0x08, 0x08, 0x08} // 2d -
,{0x00, 0x60, 0x60, 0x00, 0x00} // 2e .
,{0x20, 0x10, 0x08, 0x04, 0x02} // 2f /
,{0x3e, 0x51, 0x49, 0x45, 0x3e} // 30 0
,{0x00, 0x42, 0x7f, 0x40, 0x00} // 31 1
,{0x42, 0x61, 0x51, 0x49, 0x46} // 32 2
,{0x21, 0x41, 0x45, 0x4b, 0x31} // 33 3
,{0x18, 0x14, 0x12, 0x7f, 0x10} // 34 4
,{0x27, 0x45, 0x45, 0x45, 0x39} // 35 5
,{0x3c, 0x4a, 0x49, 0x49, 0x30} // 36 6
```

```
,{0x01, 0x71, 0x09, 0x05, 0x03} // 37 7
,{0x36, 0x49, 0x49, 0x49, 0x36} // 38 8
,{0x06, 0x49, 0x49, 0x29, 0x1e} // 39 9
,{0x00, 0x36, 0x36, 0x00, 0x00} // 3a :
,{0x00, 0x56, 0x36, 0x00, 0x00} // 3b ;
,{0x08, 0x14, 0x22, 0x41, 0x00} // 3c <
,{0x14, 0x14, 0x14, 0x14, 0x14} // 3d =
,{0x00, 0x41, 0x22, 0x14, 0x08} // 3e >
,{0x02, 0x01, 0x51, 0x09, 0x06} // 3f ?
,{0x32, 0x49, 0x79, 0x41, 0x3e} // 40 @
,{0x7e, 0x11, 0x11, 0x11, 0x7e} // 41 A
,{0x7f, 0x49, 0x49, 0x49, 0x36} // 42 B
,{0x3e, 0x41, 0x41, 0x41, 0x22} // 43 C
,{0x7f, 0x41, 0x41, 0x22, 0x1c} // 44 D
,{0x7f, 0x49, 0x49, 0x49, 0x41} // 45 E
,{0x7f, 0x09, 0x09, 0x09, 0x01} // 46 F
,{0x3e, 0x41, 0x49, 0x49, 0x7a} // 47 G
,{0x7f, 0x08, 0x08, 0x08, 0x7f} // 48 H
,{0x00, 0x41, 0x7f, 0x41, 0x00} // 49 I
,{0x20, 0x40, 0x41, 0x3f, 0x01} // 4a J
,{0x7f, 0x08, 0x14, 0x22, 0x41} // 4b K
,{0x7f, 0x40, 0x40, 0x40, 0x40} // 4c L
,{0x7f, 0x02, 0x0c, 0x02, 0x7f} // 4d M
,{0x7f, 0x04, 0x08, 0x10, 0x7f} // 4e N
,{0x3e, 0x41, 0x41, 0x41, 0x3e} // 4f O
,{0x7f, 0x09, 0x09, 0x09, 0x06} // 50 P
,{0x3e, 0x41, 0x51, 0x21, 0x5e} // 51 Q
,{0x7f, 0x09, 0x19, 0x29, 0x46} // 52 R
,{0x46, 0x49, 0x49, 0x49, 0x31} // 53 S
,{0x01, 0x01, 0x7f, 0x01, 0x01} // 54 T
,{0x3f, 0x40, 0x40, 0x40, 0x3f} // 55 U
,{0x1f, 0x20, 0x40, 0x20, 0x1f} // 56 V
,{0x3f, 0x40, 0x38, 0x40, 0x3f} // 57 W
,{0x63, 0x14, 0x08, 0x14, 0x63} // 58 X
,{0x07, 0x08, 0x70, 0x08, 0x07} // 59 Y
,{0x61, 0x51, 0x49, 0x45, 0x43} // 5a Z
,{0x00, 0x7f, 0x41, 0x41, 0x00} // 5b [
,{0x02, 0x04, 0x08, 0x10, 0x20} // 5c ï¿1/2
,{0x00, 0x41, 0x41, 0x7f, 0x00} // 5d ]
,{0x04, 0x02, 0x01, 0x02, 0x04} // 5e ^
,{0x40, 0x40, 0x40, 0x40, 0x40} // 5f _
,{0x00, 0x01, 0x02, 0x04, 0x00} // 60 `
,{0x20, 0x54, 0x54, 0x54, 0x78} // 61 a
,{0x7f, 0x48, 0x44, 0x44, 0x38} // 62 b
,{0x38, 0x44, 0x44, 0x44, 0x20} // 63 c
,{0x38, 0x44, 0x44, 0x48, 0x7f} // 64 d
```

```
,{0x38, 0x54, 0x54, 0x54, 0x18} // 65 e
,{0x08, 0x7e, 0x09, 0x01, 0x02} // 66 f
,{0x0c, 0x52, 0x52, 0x52, 0x3e} // 67 g
,{0x7f, 0x08, 0x04, 0x04, 0x78} // 68 h
,{0x00, 0x44, 0x7d, 0x40, 0x00} // 69 i
,{0x20, 0x40, 0x44, 0x3d, 0x00} // 6a j
,{0x7f, 0x10, 0x28, 0x44, 0x00} // 6b k
,{0x00, 0x41, 0x7f, 0x40, 0x00} // 6c l
,{0x7c, 0x04, 0x18, 0x04, 0x78} // 6d m
,{0x7c, 0x08, 0x04, 0x04, 0x78} // 6e n
,{0x38, 0x44, 0x44, 0x44, 0x38} // 6f o
,{0x7c, 0x14, 0x14, 0x14, 0x08} // 70 p
,{0x08, 0x14, 0x14, 0x18, 0x7c} // 71 q
,{0x7c, 0x08, 0x04, 0x04, 0x08} // 72 r
,{0x48, 0x54, 0x54, 0x54, 0x20} // 73 s
,{0x04, 0x3f, 0x44, 0x40, 0x20} // 74 t
,{0x3c, 0x40, 0x40, 0x20, 0x7c} // 75 u
,{0x1c, 0x20, 0x40, 0x20, 0x1c} // 76 v
,{0x3c, 0x40, 0x30, 0x40, 0x3c} // 77 w
,{0x44, 0x28, 0x10, 0x28, 0x44} // 78 x
,{0x0c, 0x50, 0x50, 0x50, 0x3c} // 79 y
,{0x44, 0x64, 0x54, 0x4c, 0x44} // 7a z
,{0x00, 0x08, 0x36, 0x41, 0x00} // 7b {
,{0x00, 0x00, 0x7f, 0x00, 0x00} // 7c |
,{0x00, 0x41, 0x36, 0x08, 0x00} // 7d }
,{0x10, 0x08, 0x08, 0x10, 0x08} // 7e ?
,{0x78, 0x46, 0x41, 0x46, 0x78} // 7f ?
};

/* IMAGE TYPE DECLARATIONS.
THIS DATA IS PROVIDED BY A PYTHON SCRYPT*/
/* IMAGE DATA provided by a python Script*/
/* WELCOME SCREEN UNLP */

static unsigned short welcome_screen_rows = 6;
static unsigned short welcome_screen_cols = 84;
static unsigned char welcome_screen[] =
{
0xff,0x1,0xfd,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x5
5,0xd5,0xd5,0xd5,0xd5,0xf5,0xf5,0xf5,0xf5,0xf5,0xf5,0xf5,0xf5,0xfd,0xfd,0xfd,
0xfd,0xfd,0xfd,0x7d,0x7d,0x7d,0x7d,0x7d,0x7d,0x7d,0x7d,0x7d,0x7d,0x7d,0x7d,0x
7d,0x7d,0x7d,0x7d,0xfd,0xfd,0xfd,0xfd,0xfd,0xfd,0xf5,0xf5,0xf5,0xf5,0xf5,0xf5
,0xf5,0xf5,0xd5,0xd5,0xd5,0xd5,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0x55,0
x55,0x55,0x55,0x55,0xfd,0x1,0xff,
0xff,0x0,0xff,0x55,0x55,0xd5,0xd5,0xf5,0xf5,0xfd,0xfd,0xfd,0x7f,0x7f,0x3f,0x1
f,0x1f,0xf,0xf,0x7,0x7,0x3,0x3,0x3,0x1,0x1,0x1,0x1,0x0,0x0,0x0,0x0,0x0,0x0,0x
```

```
0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0
,0x0,0x0,0x1,0x1,0x1,0x1,0x3,0x3,0x3,0x7,0x7,0xf,0xf,0x1f,0x1f,0x3f,0x3f,0x7f
,0xfd,0xfd,0xfd,0xf5,0xf5,0xd5,0xd5,0x55,0x55,0xff,0x0,0xff,
0xff,0x0,0xff,0xfd,0xff,0xff,0xff,0x1f,0xf,0x3,0x1,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x
0,0x0,0x0,0x0,0x0,0xfc,0xfc,0x0,0x0,0x0,0x0,0x0,0x0,0xfc,0xfc,0x0,0x0,0xfc,0x
fc,0xc,0x78,0xe0,0x80,0x0,0x0,0xfc,0xfc,0x0,0x0,0xfc,0xfc,0x0,0x0,0x0,0x0,0x0
,0x0,0xfc,0xfc,0x4,0x4,0x4,0x8c,0xf8,0x70,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0
,0x0,0x0,0x1,0x3,0xf,0x1f,0xff,0xff,0xff,0xfd,0xff,0x0,0xff,
0xff,0x0,0xff,0xbf,0xff,0xff,0xff,0xf8,0xf0,0xc0,0x80,0x0,0x0,0x0,0x0,0x0,0x0,0x0
,0x0,0x0,0x0,0x0,0x0,0x0,0xf,0x1f,0x30,0x20,0x20,0x20,0x20,0x30,0x1f,0xf,0x0,0x0,
0x3f,0x3f,0x0,0x0,0x0,0x1,0x7,0x1e,0x30,0x3f,0x3f,0x0,0x0,0x0,0x3f,0x3f,0x20,0x20,0x2
0,0x20,0x20,0x0,0x3f,0x3f,0x1,0x1,0x1,0x1,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0
,0x0,0x0,0x0,0x0,0x0,0x80,0xc0,0xe0,0xf8,0xff,0xff,0xff,0xbf,0xff,0x0,0xff,
0xff,0x0,0xff,0xaa,0xaa,0xab,0xab,0xaf,0xaf,0xbf,0xbf,0xbf,0xfe,0xfc,0xfc,0xf
8,0xf0,0xf0,0xf0,0xe0,0xe0,0xc0,0xc2,0xc2,0x82,0x82,0x82,0x82,0x2,0x2,0x2,0x2
,0x2,0x2,0x2,0x2,0x2,0x2,0x2,0x2,0x2,0x2,0x2,0x2,0x2,0x2,0x2,0x2,0x2,0x2,0x2,0x2,
0x2,0x2,0x2,0x2,0x2,0x2,0x82,0x82,0x82,0x82,0xc2,0xc2,0xc0,0xe0,0xe0,0xf0,0xf0,0x
f0,0xf8,0xfc,0xfc,0xfe,0xbf,0xbf,0xbf,0xaf,0xaf,0xab,0xab,0xaa,0xaa,0xff,0x0,
0xff,
0xff,0x80,0xbf,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0x
ab,0xab,0xab,0xab,0xab,0xaf,0xaf,0xaf,0xaf,0xaf,0xaf,0xaf,0xbf,0xbf,0xbf,0xbf
,0xbf,0xbf,0xbe,0xbe,0xbe,0xbe,0xbe,0xbe,0xbe,0xbe,0xbe,0xbe,0xbe,0xbe,0xbe,0
xbe,0xbe,0xbe,0xbe,0xbf,0xbf,0xbf,0xbf,0xbf,0xbf,0xbf,0xaf,0xaf,0xaf,0xaf,0xa
f,0xaf,0xaf,0xab,0xab,0xab,0xab,0xab,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,
0xaa,0xaa,0xaa,0xaa,0xbf,0x80,0xff
};


/* Default NO-DAY-SELECTED Display */


static unsigned short display_rows = 6;
static unsigned short display_cols = 84;
static unsigned char display[] =
{
0xff,0x1,0xfd,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5
,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x1d,0x0,0x1e,0x10,0x1e,0x0,
0x1e,0x4,0x8,0x1e,0x0,0x1e,0x10,0x10,0x0,0x1e,0xa,0xe,0x0,0x1d,0x5,0x5,0x5,0x
5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0x5
,0x5,0x5,0x5,0x5,0x5,0x5,0x5,0xfd,0x1,0xff,
0xff,0x0,0xff,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0
,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0
x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x
0,0x0,0x0,0x0,0x0,0x0,0xff,0x0,0xff,
0xff,0x0,0xff,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0
```

```
,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0
x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x
0,0x0,0x0,0x0,0x0,0xff,0x0,0xff,
0xff,0x0,0xff,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0
,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0
x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x
0,0x0,0x0,0x0,0x0,0xff,0x0,0xff,
0xff,0x0,0xff,0x0,0x3a,0xe4,0xf2,0x1a,0x8e,0x44,0x74,0xe,0x1a,0xf2,0xe4,0x3a,
0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0
x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x
0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0xf8,0x8,0x8c,0xc,0
xa8,0x8,0xa8,0xc,0xac,0x8,0xf8,0x0,0xff,0x0,0xff,
0xff,0x80,0xbf,0xa0,0xa0,0xa1,0xa3,0xa6,0xac,0xac,0xac,0xac,0xa6,0xa3,0xa1,0x
a0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0
,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0
xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa
0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa0,0xa7,0xac,0xaa,0xa8,0xaa,0xa8,0xaa,0xa8,
0xa8,0xac,0xa7,0xa0,0xbf,0x80,0xff
};

#endif
```

6.4.2 GLCD.c

```c
#include "glcd.h"

#define GLCD_clear_CS() {GPIOA->BRR = (1 << GLCD_CS);}
#define GLCD_set_CS() {GPIOA->BSRR = (1 << GLCD_CS);}
#define GLCD_clear_DC() {GPIOA->BRR = (1 << GLCD_DC);}
#define GLCD_set_DC() {GPIOA->BSRR = (1 << GLCD_DC);}
#define GLCD_clear_RST() {GPIOA->BRR = (1 << GLCD_RST);}
#define GLCD_set_RST() {GPIOA->BSRR = (1 << GLCD_RST);}

/*Private functions prototyps*/
void glcd_sendCommand(uint8_t);
void glcd_sendData(uint8_t);

/*Private variables*/
static uint8_t glcd_x = 0;
static uint8_t glcd_y = 0;
//this matrix is used to store the data of the screen
static uint8_t glcd_current_state[GLCD_HEIGHT][GLCD_WIDTH];

/* Code for init of the Graphic LCD N5110.
It depends on the SPI_drive.h files.
```

```
This is not tested for SPI2 but should work with some minor changes!*/
void GLCD_Init(){

	/* DISPLAY CODE */
	RCC->APB2ENR |= 0x4; //Activate GPIOA clock in case it is not activated
	spi_init();
	//MOSI PA7 and SCK as Alternate Function push-pull
	//CE, RESET, D/C as GPIO push-pull 50MHz

	//DISPLAY_PORT_CONFIG = (0x0000000B<<(4*GLCD_DIN) | 0x0000000B <<
(4*GLCD_SCK) | 0x00000003 << (4*GLCD_CS) | 0x00000003 << (4*GLCD_RST) |
0x00000003 << (4*GLCD_DC));
	DISPLAY_PORT_CONFIG = 0xB2B34433;

	//Set rst pin in case it was not already set
	GLCD_set_RST();
	delay_ms(10);
	//Reset display (maximum 100ms after power or else it may get damaged)
	GLCD_clear_RST();
	delay_ms(100); //100ns is enough acording to the datasheet
	GLCD_set_RST();

	GLCD_clear_DC();

	//Init the LCD ITS A HOLE SEQUENCE
	glcd_sendCommand(0x21); //Switch to extended mode
	glcd_sendCommand(0x84); //Set LCD Vop to 3.3V

	glcd_sendCommand(0x20); //Switch to basic instruction set
	glcd_sendCommand(0x0C); //Set normal display mode
	GLCD_drawImage(0, 0, (uint8_t *)welcome_screen, GLCD_WIDTH,
GLCD_HEIGHT);
}

/*
Send command function
*/
void glcd_sendCommand(uint8_t command){
	GLCD_clear_DC();
	spi_tx(command);
}

//Sends data (writes 8 pixels)
void glcd_sendData(uint8_t data){
	GLCD_set_DC();
	spi_tx(data);
```

```
}

//Sends Character (writes 8*5 pixels)
void GLCD_sendChar(char character){
	uint8_t i;
	if(glcd_x>GLCD_WIDTH-5) //will overflow, don't write
		return;
	GLCD_set_DC();
	spi_tx_msg(ASCII[character-32],5);
	for(i=0;i<5;i++){ //update the current state
		glcd_current_state[glcd_y][glcd_x+i] = ASCII[character-32][i];
	}
	glcd_x += 5; //update the x position
}

//Prints string finished with '\0' on the screen
void GLCD_sendString(char* string){
	int i = 0;
	while(string[i] != '\0'){
		GLCD_sendChar(string[i]);
		i++;
	}
}

/*
Sets X and Y coordinates
X is between 0 and 83
Y is between 0 and 5
*/
void GLCD_setXY(uint8_t x, uint8_t y){
	if(x>GLCD_WIDTH-1 || y>GLCD_HEIGHT-1){
		return;
	}
	glcd_sendCommand(0x40 | y); //Set Y address (1 in DB6)
	glcd_sendCommand(0x80 | x); //Set X address
	glcd_x = x;
	glcd_y = y;
}

//Cleans glcd then sets X and Y to 0
void GLCD_clean(void){
	int i,j;
	GLCD_setXY(0,0);
	for(i=0;i<GLCD_HEIGHT;i++){
		for(j=0;j<GLCD_WIDTH;j++){
			glcd_current_state[i][j] = 0;
```

```
                glcd_sendData(0x00);
                //delay_us(1);
            }
        }
        GLCD_setXY(0,0);
}

/*
Draws an image at given coordinates
Image is an array of uint8_t of size width*height
You must specify the width and height of the image
*/
void GLCD_drawImage(uint8_t x, uint8_t y, uint8_t* image, uint8_t width,
uint8_t height){
        int i,j;
        GLCD_setXY(x,y);
        for(i=0;i<height;i++){
            for(j=0;j<width;j++){
                glcd_current_state[y+i][x+j] = image[i*width+j];
                glcd_sendData(image[i*width+j]);
            }
        }
        glcd_x=x+width;
        glcd_y=y+height;
}

/*
Draws an image at given coordinates
Image is an array of uint8_t of size width*height
You must specify the width and height of the image
*/
void GLCD_drawImageNonDestructive(uint8_t x, uint8_t y, uint8_t* image,
uint8_t width, uint8_t height){
        int i,j;
        GLCD_setXY(x,y);
        for(i=0;i<height;i++){
            for(j=0;j<width;j++){
                glcd_current_state[y+i][x+j] |= image[i*width+j];

glcd_sendData(image[i*width+j]|glcd_current_state[y+i][x+j]);
            }
        }
        glcd_x=x+width;
        glcd_y=y+height;
}
```

6.5 Librería SPI_drive

6.5.1 SPI_drive.h

```
/*
Actual setup
SPI - 1
-->
PA4 --> SS
PA5 --> SCLK
PA6 --> MISO
PA7 --> MOSI

SPI2 - 2
PB12 --> SS
PB13 --> SCLK
PB14 --> MISO
PB15 --> MOSI
*/
#ifndef SPI_DRIVE_H
#define SPI_DRIVE_H

#include <stm32f103x6.h>

#define SPI1_SS 4
#define SPI1_SCLK 5
#define SPI1_MISO 6
#define SPI1_MOSI 7

void spi_init(void);
void spi_tx(uint8_t tx_char);
void spi_tx_msg(const uint8_t* str,uint8_t);

#endif
```

6.5.2 SPI_drive.c

```
#include "stm32f1xx.h"
#include "spi_drive.h"

#define spi_set_ss() {GPIOA->BSRR = (1 << SPI1_SS);}
#define spi_clear_ss() {GPIOA->BRR = (1 << SPI1_SS);}

void spi_init()
{
	RCC->APB2ENR |= 1; //Enable AFIO function
	/* SPI1 PINS SETUP CODE*/
	RCC->APB2ENR |= 0x1000; // Enabling the SPI1 periph
```

```
	/* PINS SETUP */
	GPIOA->CRL = 0xB2B34444; //PA7 - DIN | PA6 - DOUT | PA5 - CLK | PA4 -
NSS

	/* SPI1 PERIPHERAL SETUP CODE*/
	//SPI1 enabled, Master mode, BR = 36MHz/16 = 2.25MHz (GLCD supports up
to 4MHz)
	SPI1->CR1 = (1<<2) | (0x18) | (1<<6);
	SPI1->CR2 |= 0x4; //SSOE = 1 (SS output enabled)
	/* SS HIGH */
	spi_set_ss();
}

/* Transmission of a Single Character by SPI
	spi -> 1 or 2
	tx_char -> Char Data
*/
void spi_tx(uint8_t tx_char)
{
	/* SPI1 SEND CHAR */
	spi_clear_ss(); //CS to low
	SPI1->DR = tx_char; //Data register set
	while ((SPI1->SR & (1<<7))!=0); //Wait until BSY is clear
	spi_set_ss();
}

/* Transmission of a String of databy SPI
	str -> String Data
*/
void spi_tx_msg(const uint8_t* str,uint8_t size)
{
	uint8_t i=0;
	/* SPI1 SEND STRING */
	spi_clear_ss(); //CS to low
	while (i++<size)
	{
		SPI1->DR = *str;
		while ((SPI1->SR & (1<<7))!=0); //Wait until BSY is clear
		str++;
	}
	spi_set_ss();
}
```

6.6 Librería Utils

6.6.1 Utils.h

```
#ifndef UTILS_H_
```

```
#define UTILS_H_
#include <stm32f103x6.h>

#define CONST_FOR_US_DELAY 3
#define CONST_FOR_MS_DELAY 3000

void delay_ms(unsigned long);
void delay_us(unsigned long);

#endif
```

6.6.2 Utils.c

```
#include <utils.h>

static unsigned long ms_delay_const = CONST_FOR_MS_DELAY;
static unsigned long us_delay_const = CONST_FOR_US_DELAY;

void delay_us(unsigned long amount){
      unsigned long i,l;
      for(i=0;i<amount;i++)
            for(l=0;l<us_delay_const;l++);
}

void delay_ms(unsigned long amount){
      unsigned long i,l;
      for(i=0;i<amount;i++)
            for(l=0;l<ms_delay_const;l++);
}
```

6.7 Librería DHT22

6.7.1 DHT22.h

```
#ifndef DHT22_H
#define DHT22_H

#include <stm32f103x6.h>
#include "utils.h"
#ifndef DHT22_RESPONSE_TOLERANCE
#define DHT22_RESPONSE_TOLERANCE 1000000UL //at 72MHZ clock this is about 2ms
#endif

#define DHT22_DATA_PIN 15
#define DHT22_DATA_PORT GPIOA
#define DHT22_DR DHT22_DATA_PORT->ODR
#define DHT22_DR_IN DHT22_DATA_PORT->IDR
```

```
//*** Functions prototypes ***//
//OneWire Initialise
void DHT22_Init(void);

//Get Temperature and Humidity data
uint8_t DHT22_GetTemp_Humidity(float *, float *);

#endif
```

6.7.2 DHT22.c

```
#include "dht22.h"

//Macros for GPIO

#define DHT22_data_set() {DHT22_DR |= (1 << DHT22_DATA_PIN);}
#define DHT22_data_clear() {DHT22_DR &= ~(1 << DHT22_DATA_PIN);}
#define DHT22_data_get() (DHT22_DR_IN>>DHT22_DATA_PIN & 1)

//Private Functions Prototypes
uint8_t DHT22_SendStart(void);
uint8_t ReadByte(void);
uint8_t data[5]={0};
uint8_t datar=0;
uint32_t aux=0;

uint8_t VAR = 0;

void DHT22_Init(void){
    //We use PA15 as the DHT22 data pin, output open drain
    DHT22_DATA_PORT->CRH |= 0x70000000; //PA15 is GPO open drain -> setting
it means pin gets pulled up
    DHT22_data_set(); //PA15 set to high
    delay_ms(1000);
}


uint8_t DHT22_GetTemp_Humidity(float *temp, float *humidity){
    uint8_t i;
    if(temp==0 || humidity==0)
        return 0;
    if(DHT22_SendStart()){
        for(i=0; i<5; i++){
            data[i] = ReadByte();
        }
        //Checksum check
```

```
        if(data[4] == ((data[0] + data[1] + data[2] + data[3]) & 0xFF)){
            *temp = (float)((data[2] & 0x7F) << 8 | data[3]) / 10; //TODO
check this
            *humidity = (float)(data[0] << 8 | data[1]) / 10; //TODO check
this
            return 1;
        }
    }
    return 0;
}

//This function will send start signal to DHT22
//Return 1 if success, 0 if fail
//Returns when DHT22 is about to start data transmission
uint8_t DHT22_SendStart(){
    aux=1;
    //Set data pin to low
    DHT22_data_clear(); //PA15 set to low
    //Wait for 1ms
    delay_ms(18);
    //Set data pin to high and wait DHT22 response
    DHT22_data_set(); //PA15 set -> high impedance
    VAR  = DHT22_data_get();
    //Wait for DHT22 response
    while(DHT22_data_get()!=0 && ++aux!=DHT22_RESPONSE_TOLERANCE); //Wait
sensor response
    VAR = DHT22_data_get();
    if(aux==DHT22_RESPONSE_TOLERANCE)
        return 0;
    //Wait for DHT22 to release data pin
    aux=1;
    VAR = DHT22_data_get();
    delay_us(40);
    VAR = DHT22_data_get()==0;
    while(DHT22_data_get()==0 && ++aux!=DHT22_RESPONSE_TOLERANCE); //Wait
sensor to set voltage in high (~80us)
    VAR = DHT22_data_get()==0;
    delay_us(40);
    VAR = DHT22_data_get();
    while(DHT22_data_get()!=0 && ++aux!=DHT22_RESPONSE_TOLERANCE); //Wait
sensor to set voltage in low (~80us)
    VAR = DHT22_data_get();
    //DHT is ready to send data
    return aux!=DHT22_RESPONSE_TOLERANCE;
}
```

```
//This function is supposed to be called AFTER a data start transmission has
been sent
//It will read one byte from DHT22 assuming single-bus signal is being sent
//Return the byte read
uint8_t ReadByte(){
    uint8_t i=0;
    datar=0;
    while(i<8){
        //Wait for DHT22 response
        while(DHT22_data_get()==0); //Wait until PA15 gets high (~50us)
        //PA15 is high, if it stills high after more than 28us it means DHT22
is sending a 1
        //We wait for 29us just in case
        delay_us(29);
        if(DHT22_data_get() !=0)
            datar |= (1<<(7-i)); //We get PA15 state and set data bit
accordingly
        //Wait for DHT22 to set pin to low before reading next bit
        while(DHT22_data_get()!=0); //PA15 is low
        i++;
    }
    return datar;
}
```

6.8 Librería GPIO

6.8.1 GPIO.h

```
#ifndef GPIO_H
#define GPIO_H

#include <stm32f103x6.h>

//This on port A
#define GPIOA_BUTTON_OK_PIN 8
#define GPIOA_BUTTON_CANCEL_PIN 9

//This on port B
#define GPIOB_BUTTON_BUZZER_PIN 9
#define GPIOB_BUTTON_UP_PIN 12
#define GPIOB_BUTTON_DOWN_PIN 13
#define GPIOB_BUTTON_LEFT_PIN 14
#define GPIOB_BUTTON_RIGHT_PIN 15

#define getOk() (GPIOA->IDR & (1 << GPIOA_BUTTON_OK_PIN))
#define getCancel() (GPIOA->IDR & (1 << GPIOA_BUTTON_CANCEL_PIN))
#define getUp() (GPIOB->IDR & (1 << GPIOB_BUTTON_UP_PIN))
#define getDown() (GPIOB->IDR & (1 << GPIOB_BUTTON_DOWN_PIN))
```

```
#define getLeft() (GPIOB->IDR & (1 << GPIOB_BUTTON_LEFT_PIN))
#define getRight() (GPIOB->IDR & (1 << GPIOB_BUTTON_RIGHT_PIN))

#define setBuzzer() {GPIOB->ODR |= 1 << GPIOB_BUTTON_BUZZER_PIN;}
#define resetBuzzer() {GPIOB->ODR &= ~(1 << GPIOB_BUTTON_BUZZER_PIN);}

#define BUTTON_NONE 0
#define BUTTON_OK 1
#define BUTTON_CANCEL 2
#define BUTTON_UP 3
#define BUTTON_DOWN 4
#define BUTTON_LEFT 5
#define BUTTON_RIGHT 6
```

6.8.2 GPIO.c

```
#include "gpio.h"

void GPIO_Init(){
    //---
    GPIOA->CRH = 0x44444488; //PA8 and PA9 as input with pull-up
    GPIOB->CRH = 0x88884434; //PB12, 13,14,15 as input with pull-up
    //---                    PB9 as output with push-pull (0011)
}

//RETURNS BUTTON_MENU or BUTTON_SELECT
void GPIOS_getButtonPressed(uint8_t* buttonPressed)
{
    //TODO: ver de mover esto a una librerÃa
    //Also, order of this ifs matters. CANCEL has priority
    if (getCancel() != 0)
    {
        *buttonPressed = BUTTON_CANCEL;
        return;
        //return BUTTON_CANCEL;
    }
    if (getOk() != 0)
    {
        *buttonPressed = BUTTON_OK;
        return;
        //return BUTTON_OK;
    }
    if (getUp() != 0)
    {
        *buttonPressed = BUTTON_UP;
        return;
        //return BUTTON_UP;
```

```
    }
    if (getDown() != 0)
    {
        *buttonPressed = BUTTON_DOWN;
        return;
        //return BUTTON_DOWN;
    }
    if (getLeft() != 0)
    {
        *buttonPressed = BUTTON_LEFT;
        return;
        //return BUTTON_LEFT;
    }
    if (getRight() != 0)
    {
        *buttonPressed = BUTTON_RIGHT;
        return;
        //return BUTTON_RIGHT;
    }
    *buttonPressed = BUTTON_NONE;
    return;
    //return BUTTON_NONE;
}
```

6.9 Librería MEF

6.9.1 MEF.h

```
#ifndef MEF_H
#define MEF_H

#include <stm32f103x6.h>
#include <stdio.h>
#include <stdlib.h>
#include "gpio.h"
#include "dht22.h"
#include "ds1307.h"
#include "glcd.h"

#define SELECTION_NONE 0
#define SELECTION_DAY 1
#define SELECTION_DATE 2
#define SELECTION_MONTH 3
#define SELECTION_YEAR 4
#define SELECTION_HOUR 5
#define SELECTION_MINUTE 6
#define SELECTION_SECOND 7
#define SELECTION_START 1
```

```
#define SELECTION_END 7
#define SELECTION_START_ALARM 5
#define SELECTION_END_ALARM 7

typedef enum {IDLE,IDLE_ACTIVE, SETTING_DATE,SETTING_ALARM} MEF_state;

void MEF_Init(void);
void MEF_Update(void);

#endif
```

6.9.2 MEF.c

```
#include "mef.h"

static uint8_t buttonPressed = 0;
static uint8_t firstTime = 1; //first time bit to avoid rendering stuff more
than once
static MEF_state system_state;
static RTC_TIME_t time, time_old,time_alarm,time_alarm_old;
static RTC_DATE_t date, date_old;
static float temperature, humidity;
static char str[100];
static uint8_t current_selection;

//private functions (not static because proteus breaks)
void switchToSetAlarm(void);
void switchToSetDate(void);
void switchToIDLE(void);
void switchToIDLEActive(void);

void updateDate(void);
void updateTime(void);
void renderIDLE(void);
void renderIDLEActive(void);
void renderSetDate(void);
void renderSetAlarm(void);


uint8_t checkAlarm(void);
void disableBuzzer(void);
void enableBuzzer(void);

void renderDay(uint8_t day);
void renderDate(uint8_t);
void renderTime(uint8_t);
```

```
void updateTemperatureAndHumidity(void);
void renderTemperatureAndHumidity(uint8_t);

void animateCurrentSelection(void);
void animateCurrentSelection_alarm(void);
//Functions for setDate and setAlarm states
void writeTime(RTC_TIME_t);
void writeDate(RTC_DATE_t);
void modify_date(uint8_t selection, int8_t increment);
void modify_alarm(uint8_t selection, int8_t increment);
void renderTime_alarm(uint8_t force);

void MEF_Init(void)
{
    GPIO_Init();
    system_state = IDLE;
    time_alarm.hours = 0;
    time_alarm.minutes = 0;
    time_alarm.seconds = 0;
    time_alarm_old=time_alarm;
}

void MEF_Update(void)
{
    GPIOS_getButtonPressed(&buttonPressed);
    switch (system_state)
    {
    case IDLE:
        //check if alarm must be activated AFTER first time render
        if(checkAlarm() && !firstTime){
            switchToIDLEActive();
            return;
        }
        //get data from sensors and RTC
        updateTime();
        updateDate();
        updateTemperatureAndHumidity();
        //render it
        renderIDLE();
        if (buttonPressed == BUTTON_NONE)
            return;
        if (buttonPressed == BUTTON_LEFT)
        {
            switchToSetAlarm();
            return;
        }
```

```
        if (buttonPressed == BUTTON_RIGHT)
        {
            switchToSetDate();
            return;
        }
        break;
    case IDLE_ACTIVE:
        renderIDLEActive();
        if (buttonPressed == BUTTON_NONE)
            return;

        disableBuzzer(); //first disable buzzer
        switchToIDLE();  //then switch to idle
        break;
    case SETTING_DATE:
        renderSetDate();
        if (buttonPressed == BUTTON_NONE)
            return;
        //CANCEL -> IDLE, don't save
        if (buttonPressed == BUTTON_CANCEL)
        {
            switchToIDLE();
            return;
        }
        //OK -> IDLE, save
        if (buttonPressed == BUTTON_OK)
        {
            writeDate(date);
            writeTime(time);
            switchToIDLE();
            return;
        }
        //UP -> increment
        if (buttonPressed == BUTTON_UP)
        {
            modify_date(current_selection, 1);
        }
        //DOWN -> decrement
        if (buttonPressed == BUTTON_DOWN)
        {
            modify_date(current_selection, -1);
        }
        //LEFT -> select next value
        if (buttonPressed == BUTTON_RIGHT)
        {
            current_selection++;
```

```
            if (current_selection > SELECTION_END)
                current_selection = SELECTION_START;
        }
        //RIGHT -> select previous value
        if (buttonPressed == BUTTON_LEFT)
        {
            current_selection--;
            if (current_selection < SELECTION_START)
                current_selection = SELECTION_END;
        }
        animateCurrentSelection();
        break;
    case SETTING_ALARM:
        renderSetAlarm();
        if (buttonPressed == BUTTON_NONE)
            return;
        //CANCEL -> IDLE, don't save
        if (buttonPressed == BUTTON_CANCEL)
        {
            time_alarm=time_alarm_old;
            switchToIDLE();
            return;
        }
        //OK -> IDLE, save
        if (buttonPressed == BUTTON_OK)
        {
            time_alarm_old=time_alarm;
            switchToIDLE();
            return;
        }
        //UP -> increment
        if (buttonPressed == BUTTON_UP)
        {
            modify_alarm(current_selection, 1);
        }
        //DOWN -> decrement
        if (buttonPressed == BUTTON_DOWN)
        {
            modify_alarm(current_selection, -1);
        }
        //LEFT -> select next value
        if (buttonPressed == BUTTON_RIGHT)
        {
            current_selection++;
        }
        //RIGHT -> select previous value
```

```
        if (buttonPressed == BUTTON_LEFT)
        {
            current_selection--;
        }
        if (current_selection > SELECTION_END_ALARM)
            current_selection = SELECTION_START_ALARM;
        if (current_selection < SELECTION_START_ALARM)
            current_selection = SELECTION_END_ALARM;
        animateCurrentSelection_alarm();
        /*TODO*/
        break;
    }
}

//Returns 1 if alarm must be sounding
uint8_t checkAlarm(void){
    if(DS1307_timeEquals(&time, &time_alarm)){
        return 1;
    }
    return 0;
}

//This function switchs state from IDLE to SET_ALARM
void switchToSetAlarm(void)
{
    system_state = SETTING_ALARM;
    current_selection=SELECTION_NONE;
    firstTime = 1;
    return;
}
//This function switchs state from IDLE to SET_DATE
void switchToSetDate(void)
{
    system_state = SETTING_DATE;
    current_selection = SELECTION_NONE;
    firstTime = 1;
    return;
}

//This function returns to IDLE state
void switchToIDLE(void)
{
    system_state = IDLE;
    firstTime = 1;
    return;
}
```

```
//This function switchs to IDLEActive state, which is a state where the
buzzer is sounding
void switchToIDLEActive(void)
{
    system_state = IDLE_ACTIVE;
    firstTime = 1;
    return;
}

//This function renders the IDLE
void renderIDLE(void)
{
    uint8_t force = firstTime;
    if (firstTime)
    {
        GLCD_drawImage(0, 0, (uint8_t *)display, GLCD_WIDTH, GLCD_HEIGHT);
        firstTime = 0;
    }
    renderDate(force);
    renderTime(force);
    renderTemperatureAndHumidity(force);
}

void renderDate(uint8_t force)
{
    if (DS1307_dateEquals(&date, &date_old) && !force)
    {
        return; //no need to render
    }
    date_old = date;
    renderDay(date.day);
    sprintf(str, "%02d/%02d/%02d", date.date, date.month, date.year);
    GLCD_setXY(23, 3);
    GLCD_sendString(str);
}

void renderDay(uint8_t day)
{
    GLCD_setXY(20, 2);
    switch (day)
    {
    case MONDAY:
        GLCD_sendString("  LUNES  ");
        break;
    case TUESDAY:
```

```
        GLCD_sendString(" MARTES  ");
        break;
    case WEDNESDAY:
        GLCD_sendString("MIERCOLES");
        break;
    case THURSDAY:
        GLCD_sendString(" JUEVES  ");
        break;
    case FRIDAY:
        GLCD_sendString(" VIERNES ");
        break;
    case SATURDAY:
        GLCD_sendString(" SABADO  ");
        break;
    case SUNDAY:
        GLCD_sendString(" DOMINGO ");
        break;
    }
}
void updateTime()
{
    DS1307_get_time(&time);
}
void updateDate()
{
    DS1307_get_date(&date);
}

void renderTime(uint8_t force)
{
    if (DS1307_timeEquals(&time, &time_old) && !force)
    {
        return; //no need to render
    }
    time_old = time;
    sprintf(str, "%02d:%02d:%02d", time.hours, time.minutes, time.seconds);
    GLCD_setXY(23, 4);
    GLCD_sendString(str);
}

void updateTemperatureAndHumidity(void)
{
    DHT22_GetTemp_Humidity(&temperature, &humidity);
}

void renderTemperatureAndHumidity(uint8_t force)
```

```
{
    sprintf(str, "%.2f*C", temperature);
    GLCD_setXY(4, 1);
    GLCD_sendString(str);
    sprintf(str, "%.1f%%H", humidity);
    GLCD_setXY(48, 1);
    GLCD_sendString(str);
}

//This function renders the IDLEActive
void renderIDLEActive(void)
{
    if (firstTime == 1)
    {
        GLCD_drawImage(0, 0, (uint8_t *)welcome_screen, GLCD_WIDTH,
GLCD_HEIGHT);
        enableBuzzer();
        firstTime = 0;
    }
}

//This function renders the SET_DATE
void renderSetDate(void)
{
    uint8_t force = firstTime;
    if (firstTime)
    {
        GLCD_drawImage(0, 0, (uint8_t *)display, GLCD_WIDTH, GLCD_HEIGHT);
        GLCD_setXY(13,1);
        sprintf(str," SEL: NONE  ");
        GLCD_sendString(str);
        firstTime = 0;
    }
    renderDate(force);
    renderTime(force);
}

//This function renders the SET_ALARM
void renderSetAlarm(void)
{
    if (firstTime)
    {
        GLCD_drawImage(0, 0, (uint8_t *)display, GLCD_WIDTH, GLCD_HEIGHT);
        GLCD_setXY(13,1);
        sprintf(str," SEL: NONE  ");
        GLCD_sendString(str);
```

```
        renderTime_alarm(1); //force render
        firstTime = 0;
        return;
    }
    renderTime_alarm(firstTime);
}

//This function disables the buzzer
void disableBuzzer(void)
{
    resetBuzzer();
}

//This function enables the buzzer
void enableBuzzer(void)
{
    setBuzzer();
}

//Functions for setting date and alarm

void writeDate(RTC_DATE_t date){
    DS1307_set_date(&date);
}


void writeTime(RTC_TIME_t timeR){
    DS1307_set_time(&timeR);
}

void modify_date(uint8_t selection, int8_t increment){
    switch (selection)
    {
    case SELECTION_YEAR:
        date.year += increment;
        if (date.year > 99)
            date.year = 0;
        break;
    case SELECTION_MONTH:
        date.month += increment;
        if (date.month > 12)
            date.month = 1;
        if (date.month < 1)
            date.month = 12;
        break;
    case SELECTION_DATE:
```

```
        date.date += increment;
        if (date.date > 31)
            date.year = 1;
        if (date.date < 1)
            date.date = 31;
        break;
    case SELECTION_DAY:
        date.day += increment;
        if (date.day > SUNDAY)
            date.day = MONDAY;
        if(date.day<MONDAY)
            date.day = SUNDAY;
        break;
    case SELECTION_HOUR:
        time.hours += increment;
        if (time.hours > 24)
            time.hours = 0;
        break;
    case SELECTION_MINUTE:
        time.minutes += increment;
        if (time.minutes > 59)
            time.minutes = 0;
        break;
    case SELECTION_SECOND:
        time.seconds += increment;
        if (time.seconds > 59)
            time.seconds = 0;
        break;
    default: break;
    }
    return;
}
void modify_alarm(uint8_t selection, int8_t increment){
    switch (selection)
    {
    case SELECTION_HOUR:
        time_alarm.hours += increment;
        if (time_alarm.hours > 24)
            time_alarm.hours = 0;
        break;
    case SELECTION_MINUTE:
        time_alarm.minutes += increment;
        if (time_alarm.minutes > 59)
            time_alarm.minutes = 0;
        break;
    case SELECTION_SECOND:
```

```
        time_alarm.seconds += increment;
        if (time_alarm.seconds > 59)
            time_alarm.seconds = 0;
        break;
    default: break;
    }
    return;
}
void renderTime_alarm(uint8_t force)
{
    if (DS1307_timeEquals(&time_alarm, &time_old) && !force)
    {
        return; //no need to render
    }
    time_old = time_alarm;
    sprintf(str, "%02d:%02d:%02d", time_alarm.hours, time_alarm.minutes,
time_alarm.seconds);
    GLCD_setXY(23, 4);
    GLCD_sendString(str);
}

void animateCurrentSelection(void){
    GLCD_setXY(13,1);
    switch(current_selection){
        case SELECTION_YEAR:
            sprintf(str,"SEL A'O: %2d",date.year);
        break;
        case SELECTION_MONTH:
            sprintf(str,"SEL MES: %2d",date.month);
            //TODO: animate month
        break;
        case SELECTION_DATE:
            sprintf(str,"SEL DIA: %2d",date.date);
            //TODO: animate date
        break;
        case SELECTION_DAY:
            sprintf(str,"SEL: DIASEM");
            //TODO: animate day
        break;
        case SELECTION_HOUR:
            sprintf(str,"SEL HS: %2d ",time.hours);
            //TODO: animate hour
        break;
        case SELECTION_MINUTE:
            sprintf(str,"SEL MIN: %2d",time.minutes);
            //TODO: animate minute
```

```
        break;
        case SELECTION_SECOND:
            sprintf(str,"SEL SEG: %2d",time.seconds);
            //TODO: animate second
        break;
        case SELECTION_NONE:
            sprintf(str," SEL: NONE  ");
        default:
            sprintf(str," SEL: NONE  ");
        break;
    }
    GLCD_sendString(str);
}
void animateCurrentSelection_alarm(void){
    GLCD_setXY(13,1);
    switch(current_selection){
        case SELECTION_HOUR:
            sprintf(str,"SEL HS: %2d ",time_alarm.hours);
            //TODO: animate hour
        break;
        case SELECTION_MINUTE:
            sprintf(str,"SEL MIN: %2d",time_alarm.minutes);
            //TODO: animate minute
        break;
        case SELECTION_SECOND:
            sprintf(str,"SEL SEG: %2d",time_alarm.seconds);
            //TODO: animate second
        break;
        case SELECTION_NONE:
            sprintf(str," SEL: NONE  ");
        default: break;
    }
    GLCD_sendString(str);
}
```

6.10 Librería Main

6.10.1 Main.h

```
#ifndef MAIN_H
#define MAIN_H

#include <stm32f103x6.h>
#include "utils.h"
#include "dht22.h"
#include "ds1307.h"
```

```
#include "glcd.h"
#include "mef.h"

#endif
```

6.10.2 Main.c

```
#include <stm32f103x6.h>
#include "main.h"

float temp,humidity;
char str[30];
RTC_TIME_t time;
RTC_DATE_t date;

int main (void){
  RCC->APB2ENR= 0xFC;
  MEF_Init();
  GLCD_Init();
  DS1307_Init();
  DHT22_Init();
  delay_ms(100);
  while(1){
    delay_ms(100);
    MEF_Update();
    }
  return 0;
}
```